HIGH-PERFORMANCE TENSOR COMPUTATIONS IN SCIENTIFIC COMPUTING AND DATA SCIENCE

EDITED BY: Edoardo Angelo Di Napoli, Paolo Bientinesi, Jiajia Li and André Uschmajew
PUBLISHED IN: Frontiers in Applied Mathematics and Statistics

frontiers Research Topics

frontiers

ISSN 1664-8714
ISBN 978-2-83250-425-3
DOI 10.3389/978-2-83250-425-3

About Frontiers

Frontiers is more than just an open-access publisher of scholarly articles: it is a pioneering approach to the world of academia, radically improving the way scholarly research is managed. The grand vision of Frontiers is a world where all people have an equal opportunity to seek, share and generate knowledge. Frontiers provides immediate and permanent online open access to all its publications, but this alone is not enough to realize our grand goals.

Frontiers Journal Series

The Frontiers Journal Series is a multi-tier and interdisciplinary set of open-access, online journals, promising a paradigm shift from the current review, selection and dissemination processes in academic publishing. All Frontiers journals are driven by researchers for researchers; therefore, they constitute a service to the scholarly community. At the same time, the Frontiers Journal Series operates on a revolutionary invention, the tiered publishing system, initially addressing specific communities of scholars, and gradually climbing up to broader public understanding, thus serving the interests of the lay society, too.

Dedication to Quality

Each Frontiers article is a landmark of the highest quality, thanks to genuinely collaborative interactions between authors and review editors, who include some of the world's best academicians. Research must be certified by peers before entering a stream of knowledge that may eventually reach the public - and shape society; therefore, Frontiers only applies the most rigorous and unbiased reviews.
Frontiers revolutionizes research publishing by freely delivering the most outstanding research, evaluated with no bias from both the academic and social point of view. By applying the most advanced information technologies, Frontiers is catapulting scholarly publishing into a new generation.

What are Frontiers Research Topics?

Frontiers Research Topics are very popular trademarks of the Frontiers Journals Series: they are collections of at least ten articles, all centered on a particular subject. With their unique mix of varied contributions from Original Research to Review Articles, Frontiers Research Topics unify the most influential researchers, the latest key findings and historical advances in a hot research area! Find out more on how to host your own Frontiers Research Topic or contribute to one as an author by contacting the Frontiers Editorial Office: frontiersin.org/about/contact

HIGH-PERFORMANCE TENSOR COMPUTATIONS IN SCIENTIFIC COMPUTING AND DATA SCIENCE

Topic Editors:
Edoardo Angelo Di Napoli, Julich Research Center, Helmholtz Association of German Research Centres (HZ), Germany
Paolo Bientinesi, Umeå University, Sweden
Jiajia Li, College of William & Mary, United States
André Uschmajew, Max Planck Institute for Mathematics in the Sciences, Germany

Citation: Di Napoli, E. A., Bientinesi, P., Li, J., Uschmajew, A., eds. (2022). High-Performance Tensor Computations in Scientific Computing and Data Science. Lausanne: Frontiers Media SA. doi: 10.3389/978-2-83250-425-3

Table of Contents

frontiers | Frontiers in Applied Mathematics and Statistics

TYPE Editorial
PUBLISHED 23 September 2022
DOI 10.3389/fams.2022.1038885

OPEN ACCESS

EDITED AND REVIEWED BY
Daniel Potts,
Chemnitz University of Technology, Germany

*CORRESPONDENCE
Edoardo Di Napoli
e.di.napoli@fz-juelich.de

SPECIALTY SECTION
This article was submitted to Mathematics of Computation and Data Science, a section of the journal Frontiers in Applied Mathematics and Statistics

RECEIVED 07 September 2022
ACCEPTED 08 September 2022
PUBLISHED 23 September 2022

CITATION
Di Napoli E, Bientinesi P, Li J and Uschmajew A (2022) Editorial: High-performance tensor computations in scientific computing and data science.
Front. Appl. Math. Stat. 8:1038885.
doi: 10.3389/fams.2022.1038885

Editorial: High-performance tensor computations in scientific computing and data science

Edoardo Di Napoli[1]*, Paolo Bientinesi[2], Jiajia Li[3] and André Uschmajew[4]

[1]Simulation and Data Lab Quantum Materials, Jülich Supercomputing Centre, Forschungszentrum Jülich, Jülich, Germany, [2]Department of Computing Science, Umeå University, Umeå, Sweden, [3]Department of Computer Science, North Carolina State University, Raleigh, NC, United States, [4]Max Planck Institute for Mathematics in the Sciences, Leipzig, Germany

KEYWORDS

tensor operation, tensor decomposition, tensor network, multilinear algebra, high performance optimization, low-rank approximation, Deep Learning, tensor library

Editorial on the Research Topic
High-performance tensor computations in scientific computing and data science

Introduction

In the last two decade, tensor computations developed from a small and little known subject to a vast and heterogeneous field with many diverse topics ranging from high-order decomposition and low-rank approximation to optimization and multi-linear contractions. At the same time, several of these operations with tensors are progressively and diversely applied to many, rather distinct domains; from Quantum Chemistry to Deep Learning, and from Condensed Matter Physics to Remote Sensing. These domain-specific applications of tensor computations present a number of particular challenges originating from their high dimensionality, computational cost, and complexity. Usually, because these challenges could be quite diverse among application areas, there is not an homogeneous and uniform approach in the development of software programs tackling tensor operations. On the contrary, very often developers implement domain-specific libraries which compromise their use across disciplines. The end result is a fragmented community where efforts are often replicated and scattered [1].

This Research Topic represents an attempt in bringing together different communities, spearheading the latest cutting-edge results at the frontier of tensor computations, and sharing the lessons learned in domain-specific applications. The issue includes ten research articles written by experts in the field. For the sake of clarity, the articles can be somewhat artificially divided in four main areas: (i) decompositions, (ii) low-rank approximations, (iii) high-performance operations, and (iv) tensor networks. In practice, many of the works in this Research Topic spill over the boundaries of such

areas and are interdisciplinary in nature, thus demonstrating how cross-fertilizing the field of tensor computations is.

Decompositions

In multilinear algebra, the Canonical Polyadic decomposition (CP) is one of several generalization of the matrix Singular Value Decomposition (SVD) to tensors. The problem considered in Psarras et al. is the estimation of the uncertainty associated with the parameters of a Canonical Polyadic tensor decomposition. The authors demonstrate that it is possible to perform such an estimation (jackknife resampling) without altering the input tensor at the cost of a modest increase in floating point operations. This observation makes it possible to take advantage of a recent technique—Concurrent Alternating Least Squares (CALS, [2])—to accelerate the computation of jackknife resampling. The authors make the software generated publicly available.

Khoromskaia and Khoromskij present the reduced higher-order SVD (RHOSVD), which is an efficient version of the high-order SVD (HOSVD) applicable to tensors in CP format. The authors focus on the important step of rank truncation necessary in domain-specific computations with large scale tensors in scientific computing. Besides a survey, the article offers new error and stability results for the RHOSVD, as well as several applications to problems for computational physics, notably the rank-structured computations involving multi-particle interaction potentials by using range-separated tensor format.

While recovering the decomposition of a tensor can be seen as an a-posterior operation on a given tensor, a specific tensor decomposition can be a-priory imposed as initial condition to the solution of a given problem. The work by Hendrikx et al. studies problems that can be formulated as a block row Kronecker-structured (BRKS) linear system with a constrained tensor as the solution. The authors consider low-rank multilinear singular value decomposition (MLSVD), CP, and tensor train (TT) as the constrained tensors. Efficient algorithms to find these solutions are provided for large and high-order data tensors. This work also derives conditions under which the constrained tensors can be retrieved from a BRKS system. The experimental results demonstrate effectiveness of the proposed algorithms including an application to hyper-spectral image reconstruction.

Low-rank approximations

One important application of low-rank tensors is the representation of high-dimensional functions. In their respective papers, Ayvaz et al. and Götte et al. demonstrate how low-rank tensor decomposition can be used for representing and optimizing certain classes of multivariate polynomials, essentially by using a low-rank model for their coefficient tensors. This approach provides practical access to quite a rich set of nonlinear classes of multivariate polynomials in low-parametric format that can be used as models in several tasks of data science and machine learning. These applications are amply demonstrated in the papers and used as a confirmation of the efficacy and correctness of the methods. While in Ayvaz et al. the authors focus on the CP format and efficient optimization based on Gauss-Newton-type algorithms, the work presented in Götte et al. proposes a block sparse TT format in combination with alternating least squares optimization.

Cohen introduces a framework for structured low-rank approximations of matrices and tensors in which the columns of one of the factor matrices are known or required to be sparse with respect to a fixed dictionary. Such a model subsumes several special cases with important applications in signal processing and data science. The focus of the work is on efficient optimization algorithms, especially on the sparse-coding sub-problem that appears when applying an alternating optimization strategy, which is of interest in itself. Several approaches, both convex and non-convex, are considered for handling this important problem and their performance is compared. The paper therefore also serves as a valuable overview on the subject.

HPC operations

In their rather comprehensive paper Georganas et al. present a programming abstraction (the Tensor Processing Primitives or TPP for short) striving for efficient and portable implementation of tensor operations, with a special focus on Deep Learning (DL) workloads. The aim of these primitives is to provide a 'middle way' between the monolithic and inflexible operators offered by DL libraries and the high level of abstraction provided by Tensor Compilers. The TPP attempt to strike a balance between these two extremes by providing relatively low-level 2D tensor primitives that act as building blocks for more complex and high-level DL operators. In other words, the TPP specification are platform agnostic while their implementation is platform specific. The article provides numerous practical examples where TPP are used in the realm of DL workloads as well as HPC tasks not specific to data science.

On a completely different direction, Bassoy presents a technique to implement basic tensor operations in C++ avoiding pointer arithmetic and instead relying on iterators. The technique is incorporated into the uBlas extension of Boost, and is demonstrated on element-wise tensor operations (e.g., tensor addition), as well as tensor multiplications (tensor-times-vector, tensor-times-matrix, and tensor-times-tensor). The aim is a modular design to deal with tensors

and sub-tensors of arbitrary dimension, abstracting from storage formats.

Tensor Networks

Tensor Networks methods originated from Condensed Matter Physics but their application nowadays can span diverse fields such as Quantum Computing and Artificial Intelligence and has emerged as a mainstream field in tensor computations [3]. This Research Topic includes two publications which are at the crossroad between HPC and Tensor Networks. The paper by Lyakh et al. considers the processing of tensor networks. Specifically, it introduces a high-performance library to build, transform, and numerically evaluate tensor networks with arbitrary graph structures and complexity. The library is designed to run on laptops, workstations, as well as HPC platforms, including shared-memory, distributed-memory, and GPU-accelerated systems.

While Lyakh et al. focus on the specifics of tensor networks operations, the work by Evenbly maintains an high-level approach and is aimed at researchers already familiar with the theoretical setup of Tensor Networks that want to code their own software programs. It provides a practical description of how such programs need to be designed and implemented if they are going to ripe the benefits of High-Performance low-level numerical libraries and parallel architectures. The content is organized in sections, each covering a specific building block appearing in Tensor Network algorithms, such as contractions, decompositions, and gauge transformations. At the end of each section a useful summary is provided as a sort of recipe to realize in practice the specific Tensor Network operation in terms of the building blocks.

Author contributions

EDN wrote the introduction and finalized the manuscript. All authors contributed to the manuscript a short summary for the papers they edited, and approved the submitted version.

Acknowledgments

We would like to thank the effort and contribution of all review editors whose meticulous and time-consuming work made this Research Topic possible.

Conflict of interest

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's note

All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher.

References

1. Psarras C, Karlsson L, Li J, Bientinesi P. The landscape of software for tensor computations. *CoRR. abs/2103.13756* (2021) doi: 10.48550/arXiv.2103.13756

2. Psarras C, Karlsson L, Bro R, Bientinesi P. Algorithm XXX: concurrent alternating least squares for multiple simultaneous canonical polyadic decompositions. *ACM Trans Math Softw*. (2022) 48:1–20. doi: 10.1145/3519383

3. Silvi P, Tschirsich F, Gerster M, Jünemann J, Jaschke D, Rizzi M, et al. The Tensor Networks Anthology: Simulation techniques for many-body quantum lattice systems. *SciPost Phys Lect Notes*. (2019) 8:8. doi: 10.21468/SciPostPhysLectNotes.8

frontiers in Applied Mathematics and Statistics

ORIGINAL RESEARCH
published: 07 September 2021
doi: 10.3389/fams.2021.702486

A Block-Sparse Tensor Train Format for Sample-Efficient High-Dimensional Polynomial Regression

*Michael Götte, Reinhold Schneider and Philipp Trunschke**

Department of Mathematics, Technische Universität Berlin, Berlin, Germany

Low-rank tensors are an established framework for the parametrization of multivariate polynomials. We propose to extend this framework by including the concept of block-sparsity to efficiently parametrize homogeneous, multivariate polynomials with low-rank tensors. This provides a representation of general multivariate polynomials as a sum of homogeneous, multivariate polynomials, represented by block-sparse, low-rank tensors. We show that this sum can be concisely represented by a single block-sparse, low-rank tensor.

We further prove cases, where low-rank tensors are particularly well suited by showing that for banded symmetric tensors of homogeneous polynomials the block sizes in the block-sparse multivariate polynomial space can be bounded independent of the number of variables.

We showcase this format by applying it to high-dimensional least squares regression problems where it demonstrates improved computational resource utilization and sample efficiency.

Keywords: sample efficiency, homogeneous polynomials, sparse tensor networks, alternating least square, empirical L2 approximation

OPEN ACCESS

Edited by:
Edoardo Angelo Di Napoli, Helmholtz-Verband Deutscher Forschungszentren (HZ), Germany

Reviewed by:
Mazen Ali, École centrale de Nantes, France
Katharina Kormann, Uppsala University, Sweden
Antonio Falco, Universidad CEU Cardenal Herrera, Spain

****Correspondence:***
Philipp Trunschke
ptrunschke@mail.tu-berlin.de

Specialty section:
This article was submitted to Mathematics of Computation and Data Science, a section of the journal Frontiers in Applied Mathematics and Statistics

Received: *29 April 2021*
Accepted: *21 July 2021*
Published: *07 September 2021*

Citation:
Götte M, Schneider R and Trunschke P (2021) A Block-Sparse Tensor Train Format for Sample-Efficient High-Dimensional Polynomial Regression. Front. Appl. Math. Stat. 7:702486. doi: 10.3389/fams.2021.702486

1 INTRODUCTION

An important problem in many applications is the identification of a function from measurements or random samples. For this problem to be well-posed, some prior information about the function has to be assumed and a common requirement is that the function can be approximated in a finite dimensional ansatz space. For the purpose of extracting governing equations the most famous approach in recent years has been SINDy [1]. However, the applicability of SINDy to high-dimensional problems is limited since truly high-dimensional problems require a nonlinear parameterization of the ansatz space. One particular reparametrization that has proven itself in many applications are tensor networks. These allow for a straight-forward extension of SINDy [2] but can also encode additional structure as presented in [3]. The compressive capabilities of tensor networks originate from this ability to exploit additional structure like smoothness, locality or self-similarity and have hence been used in solving high-dimensional equations [4–7]. In the context of optimal control tensor train networks have been utilized for solving the Hamilton–Jacobi–Bellman equation in [8,9], for solving backward stochastic differential equations in [10] and for the

calculation of stock options prices in [11,12]. In the context of uncertainty quantification they are used in [13–15] and in the context of image classification they are used in [16,17].

A common thread in these publications is the parametrization of a high-dimensional ansatz space by a tensor train network which is then optimized. In most cases this means that the least-squares error of the parametrized function to the data is minimized. There exist many methods to perform this minimization. A well-known algorithm in the mathematics community is the *alternating least-squares (ALS)* [18,19], which is related to the famous DMRG method [20] for solving the Schrödinger equation in quantum physics. Although, not directly suitable for recovery tasks, it became apparent that DMRG and ALS can be adapted to work in this context. Two of these extensions to the ALS algorithm are the *stablilized ALS approximation (SALSA)* [21] and the *block alternating steepest descent for Recovery (bASD)* algorithm [13]. Both adapt the tensor network ranks and are better suited to the problem of data identification. Since the set of tensor trains of fixed rank forms a manifold [22] it is also possible to perform gradient based optimization schemes [48]. This however is not a path that we pursue in this work. Our contribution extends the ALS (and SALSA) algorithm and we believe that it can be applied to many of the fields stated above.

In this work we consider ansatz spaces of homogeneous polynomials of fixed degree and their extension to polynomials of bounded degree. We introduce the concept of block-sparsity as an efficient way to parametrize homogeneous polynomials with low rank tensors. Although, this is not the first instance in which sparsity is used in the context of low-rank tensors (see [24–26]), we believe, that this is the first time where block-sparsity is used to parametrize homogeneous polynomials. The sparsity used in the previous works is substantially different to the block-sparsity discussed in this work. Block-sparsity is preserved under most tensor network operations such as summation, orthogonalization and rounding and the parametrization of tangent spaces which is not the case for standard sparsity. This is important since orthogonalization is an essential part of numerically stable and efficient optimization schemes and means that most of the existing tensor methods, like HSVD (see [27]), ALS, SALSA or Riemannian optimization can be performed in this format. We also show that, if the symmetric tensor of a homogeneous polynomial is banded, it can be represented very efficiently in the tensor train format, since the sizes of the non-zero blocks can be bounded independently of the number of variables. In physics this property can be associated with the property of locality, which can be used to identify cases where tensor trains work exceptionally well.

Quantum physicists have used the concept of block-sparsity for at least a decade [28] but it was introduced to the mathematics community only recently in [29]. In the language of quantum mechanics one would say that there exists an operator for which the coefficient tensor of any homogeneous polynomial is an eigenvector. This encodes a symmetry, where the eigenvalue of this eigenvector is the degree of the homogeneous polynomial, which acts as a quantum number and corresponds to the particle number of bosons and fermions.

The presented approach is very versatile and can be combined with many polynomial approximation strategies like the use of Taylor's theorem in [30] and there exist many approximation theoretic results that ensure a good approximation with a low degree polynomial for many classes of functions (see e.g. [31]).

In addition to the approximation theoretic results, we can motivate these polynomial spaces by thinking about the sample complexity for successful recovery in the case of regression problems. In [32] it was shown that for tensor networks the sample complexity, meaning the number of data points needed, is related to the dimension of the high-dimensional ansatz space. But, these huge sample sizes are not needed in most practical examples [14]. This suggests that the regularity of the sought function must have a strong influence on the number of samples that are required. However, for most practical applications, suitable regularity guarantees cannot be made — neither for the best approximation nor for the initial guess, nor any iterate of the optimization process. By restricting ourselves to spaces of homogeneous polynomials, the gap between observed sample complexity and proven worst-case bound is reduced.

In the regression setting, this means that we kill two birds with one stone. By applying block-sparsity to the coefficient tensor we can restrict the ansatz space to well-behaved functions which can be identified with a reasonable sample size. At the same time we reduce the number of parameters and speed up the least-squares minimization task. Finally, note that this parametrization allows practitioners to devise algorithms that are adaptive in the degree of the polynomial, thereby increasing the computational resource utilization even further. This solves a real problem in practical applications where the additional and unnecessary degrees of freedom of conventional low-rank tensor formats cause many optimization algorithms to get stuck in local minima.

The remainder of this work is structured as follows. *Notation* introduces basic tensor notation, the different parametrizations of polynomials that are used in this work and then formulates the associated least-squares problems. In *Theoretical Foundation* we state the known results on sampling complexity and block sparsity. Furthermore, we set the two results in relation and argue why this leads to more favorable ansatz spaces. This includes a proof of rank-bounds for a class of homogeneous polynomials which can be represented particularly efficient as tensor trains. *Method Description* derives two parametrizations from the results of *Theoretical Foundation* and presents the algorithms that are used to solve the associated least-squares problems. Finally, *Numerical Results* gives some numerical results for different classes of problems focusing on the comparison of the sample complexity for the full- and sub-spaces. Most notably, the recovery of a quantity of interest for a parametric PDE, where our approach achieves successful recovery with relatively few parameters and samples. We observed that for suitable problems the number of parameters can be reduced by a factor of almost 10.

2 NOTATION

In our opinion, using a graphical notation for the involved contractions in a tensor network drastically simplifies the expressions making the whole setup more approachable. This section introduces this graphical notation for tensor networks, the

spaces that will be used in the remainder of this work and the regression framework.

2.1 Tensors and Indices

Definition 2.1. Let $d \in \mathbb{N}$. Then $\boldsymbol{n} = (n_1, \ldots, n_d) \in \mathbb{N}^d$ is called a *dimension tuple of order d* and $x \in \mathbb{R}^{n_1 \times \cdots \times n_d} =: \mathbb{R}^{\boldsymbol{n}}$ is called a *tensor of order d and dimension* $\boldsymbol{n}$. Let $\mathbb{N}_n = \{1, \ldots, n\}$ then a tuple $(l_1, \ldots, l_d) \in \mathbb{N}_{n_1} \times \cdots \times \mathbb{N}_{n_d} =: \mathbb{N}_{\boldsymbol{n}}$ is called a *multi-index* and the corresponding entry of x is denoted by $x(l_1, \ldots, l_d)$. The positions $1, \ldots, d$ of the indices $l_1, \ldots, l_d$ in the expression $x(l_1, \ldots, l_d)$ are called *modes of x*.

To define further operations on tensors it is often useful to associate each mode with a symbolic index.

Definition 2.2. A *symbolic index i* of dimension n is a placeholder for an arbitrary but fixed natural number between 1 and n. For a dimension tuple $\boldsymbol{n}$ of order d and a tensor $x \in \mathbb{R}^{\boldsymbol{n}}$ we may write $x(i_1, \ldots, i_d)$ and tacitly assume that i_k are indices of dimension n_k for each $k = 1, \ldots, d$. When standing for itself this notation means $x(i_1, \ldots, i_d) = x \in \mathbb{R}^{\boldsymbol{n}}$ and may be used to *slice* the tensor

$$x(i_1, l_2, \ldots, l_d) \in \mathbb{R}^{n_1}$$

where $l_k \in \mathbb{N}_{n_k}$ are fixed indices for all $k = 2, \ldots, d$. For any dimension tuple $\boldsymbol{n}$ of order d we define the symbolic multi-index $i^{\boldsymbol{n}} = (i_1, \ldots, i_d)$ of dimension $\boldsymbol{n}$ where i_k is a symbolic index of dimension n_k for all $k = 1, \ldots, d$.

Remark 2.3. We use the letters i and j (with appropriate subscripts) for symbolic indices while reserving the letters k, l and m for ordinary indices.

Example 2.4. Let x be an order 2 tensor with mode dimensions n_1 and n_2, i.e. an n_1-by-n_2 matrix. Then $x(\ell_1, j)$ denotes the ℓ_1-th row of x and $x(i, \ell_2)$ denotes the ℓ_2-th column of x.

Inspired by Einstein notation we use the concept of symbolic indices to define different operations on tensors.

Definition 2.5. Let i_1 and i_2 be (symbolic) indices of dimension n_1 and n_2, respectively and let φ be a bijection

$$\varphi\colon \mathbb{N}_{n_1} \times \mathbb{N}_{n_2} \to \mathbb{N}_{n_1 n_2}.$$

We then define *the product of indices* with respect to φ as $j = \varphi(i_1, i_2)$ where j is a (symbolic) index of dimension $n_1 n_2$. In most cases the choice of bijection is not important and we will write $i_1 \cdot i_2 := \varphi(i_1, i_2)$ for an arbitrary but fixed bijection φ. For a tensor x of dimension (n_1, n_2) the expression

$$y(i_1 \cdot i_2) = x(i_1, i_2)$$

defines the tensor y of dimension $(n_1 n_2)$ while the expression

$$x(i_1, i_2) = y(i_1 \cdot i_2)$$

defines $x \in \mathbb{R}^{n_1 \times n_2}$ from $y \in \mathbb{R}^{n_1 n_2}$.

Definition 2.6. Consider the tensors $x \in \mathbb{R}^{\boldsymbol{n}_1 \times a \times \boldsymbol{n}_2}$ and $y \in \mathbb{R}^{\boldsymbol{n}_3 \times b \times \boldsymbol{n}_4}$. Then the expression

$$z\left(i^{\boldsymbol{n}_1}, i^{\boldsymbol{n}_2}, j_1, j_2, i^{\boldsymbol{n}_3}, i^{\boldsymbol{n}_4}\right) = x\left(i^{\boldsymbol{n}_1}, j_1, i^{\boldsymbol{n}_2}\right) \cdot y\left(i^{\boldsymbol{n}_3}, j_2, i^{\boldsymbol{n}_4}\right) \tag{1}$$

defines the tensor $z \in \mathbb{R}^{\boldsymbol{n}_1 \times \boldsymbol{n}_2 \times a \times b \times \boldsymbol{n}_3 \times \boldsymbol{n}_4}$ in the obvious way. Similary, for $a = b$ the expression

$$z\left(i^{\boldsymbol{n}_1}, i^{\boldsymbol{n}_2}, j, i^{\boldsymbol{n}_3}, i^{\boldsymbol{n}_4}\right) = x\left(i^{\boldsymbol{n}_1}, j, i^{\boldsymbol{n}_2}\right) \cdot y\left(i^{\boldsymbol{n}_3}, j, i^{\boldsymbol{n}_4}\right) \tag{2}$$

defines the tensor $z \in \mathbb{R}^{\boldsymbol{n}_1 \times \boldsymbol{n}_2 \times a \times \boldsymbol{n}_3 \times \boldsymbol{n}_4}$. Finally, also for $a = b$ the expression

$$z\left(i^{\boldsymbol{n}_1}, i^{\boldsymbol{n}_2}, i^{\boldsymbol{n}_3}, i^{\boldsymbol{n}_4}\right) = x\left(i^{\boldsymbol{n}_1}, j, i^{\boldsymbol{n}_2}\right) \cdot y\left(i^{\boldsymbol{n}_3}, j, i^{\boldsymbol{n}_4}\right) \tag{3}$$

defines the tensor $z \in \mathbb{R}^{\boldsymbol{n}_1 \times \boldsymbol{n}_2 \times \boldsymbol{n}_3 \times \boldsymbol{n}_4}$ as

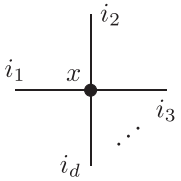
$$z\left(i^{\boldsymbol{n}_1}, i^{\boldsymbol{n}_2}, i^{\boldsymbol{n}_3}, i^{\boldsymbol{n}_4}\right) = \sum_{k=1}^{a} x\left(i^{\boldsymbol{n}_1}, k, i^{\boldsymbol{n}_2}\right) \cdot y\left(i^{\boldsymbol{n}_3}, k, i^{\boldsymbol{n}_4}\right).$$

We choose this description mainly because of its simplicity and how it relates to the implementation of these operations in the numeric libraries numpy [33] and xerus [34].

2.2 Graphical Notation and Tensor Networks

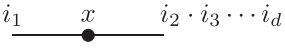
This section will introduce the concept of *tensor networks* [35] and a graphical notation for certain operations which will simplify working with these structures. To this end we reformulate the operations introduced in the last section in terms of nodes, edges and half-edges.

Definition 2.7. For a dimension tuple $\boldsymbol{n}$ of order d and a tensor $x \in \mathbb{R}^{\boldsymbol{n}}$ the *graphical representation* of x is given by.



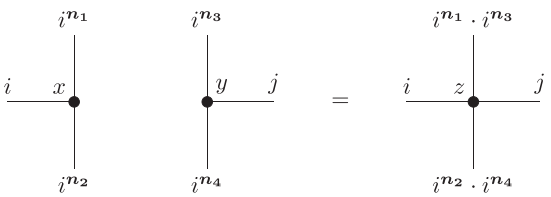
where the node represents the tensor and the half-edges represent the d different modes of the tensor illustrated by the symbolic indices $i_1, \ldots, i_d$.

With this definition we can write the reshapings of Defintion 2.5 simply as

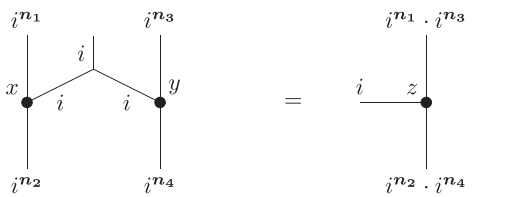
$x(i_1, i_2 \cdot i_3 \cdots i_d) \quad =$ 

and also simplify the binary operations of Definition 2.6.

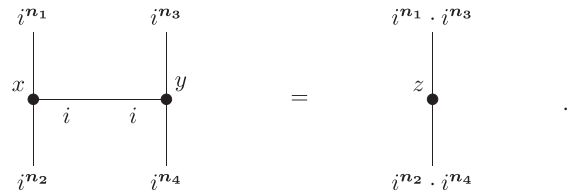
Definition 2.8. Let $x \in \mathbb{R}^{\boldsymbol{n}_1 \times a \times \boldsymbol{n}_2}$ and $y \in \mathbb{R}^{\boldsymbol{n}_3 \times b \times \boldsymbol{n}_4}$ be two tensors. Then Operation **Eq. 1** is represented by



and defines $z \in \mathbb{R}^{\cdots \times a \times b \times \cdots}$. For $a = b$ Operation **Eq. 2** is represented by



and defines $z \in \mathbb{R}^{\cdots \times a \times \cdots}$ and Operation **Eq. 3** defines $z \in \mathbb{R}^{\cdots \times \cdots}$ by.



With these definitions we can compose entire networks of multiple tensors which are called tensor networks.

2.3 The Tensor Train Format

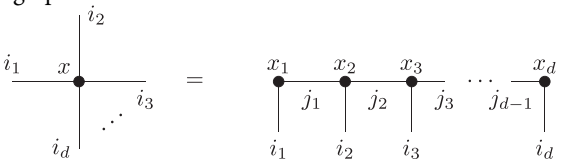
A prominent example of a tensor network is the *tensor train (TT)* [19,36], which is the main tensor network used throughout this work. This network is discussed in the following subsection.

Definition 2.9. Let $\boldsymbol{n}$ be an dimensional tuple of order-d. The TT format decomposes an order d tensor $x \in \mathbb{R}^{\boldsymbol{n}}$ into d *component tensors* $x_k \in \mathbb{R}^{r_{k-1} \times n_k \times r_k}$ for $k = 1, \ldots, d$ with $r_0 = r_d = 1$. This can be written in tensor network formula notation as

$$x(i_1, \cdots, i_d) = x_1(i_1, j_1) \cdot x_2(j_1, i_2, j_2) \cdots x_d(j_{d-1}, i_d).$$

The tuple $(r_1, \ldots, r_{d-1})$ is called the *representation rank* of this representation.

In graphical notation it looks like this.



Remark 2.10. Note that this representation is not unique. For any pair of matrices (A, B) that satisfies $AB = Id$ we can replace x_k by $x_k(i_1, i_2, j) \cdot A(j, i_3)$ and x_{k+1} by $B(i_1, j) \cdot x(j, i_2, i_3)$ without changing the tensor x.

The representation rank of x is therefore dependent on the specific representation of x as a TT, hence the name. Analogous to the concept of matrix rank we can define a minimal necessary rank that is required to represent a tensor x in the TT format.

Definition 2.11. The *tensor train rank* of a tensor $x \in \mathbb{R}^{\boldsymbol{n}}$ with tensor train components $x_1 \in \mathbb{R}^{n_1 \times r_1}$, $x_k \in \mathbb{R}^{r_{k-1} \times n_k \times r_k}$ for $k = 2, \ldots, d-1$ and $x_d \in \mathbb{R}^{r_{d-1} \times n_d}$ is the set

$$\mathrm{TT-rank}(x) = (r_1, \ldots, r_d)$$

of minimal r_k's such that the x_k compose x.

In [[22], Theorem 1a] it is shown that the TT-rank can be computed by simple matrix operations. Namely, r_k can be computed by joining the first k indices and the remaining $d-k$ indices and computing the rank of the resulting matrix. At last, we need to introduce the concept of left and right orthogonality for the tensor train format.

Definition 2.12. Let $x \in \mathbb{R}^{\boldsymbol{m} \times n}$ be a tensor of order $d+1$. We call x *left orthogonal* if

$$x(i^{\boldsymbol{m}}, j_1) \cdot x(i^{\boldsymbol{m}}, j_2) = Id(j_1, j_2).$$

Similarly, we call a tensor $x \in \mathbb{R}^{m \times \boldsymbol{n}}$ of order $d+1$ *right orthogonal* if

$$x(i_1, j^{\boldsymbol{n}}) \cdot x(i_2, j^{\boldsymbol{n}}) = Id(i_1, i_2).$$

A tensor train is *left orthogonal* if all component tensors $x_1, \ldots, x_{d-1}$ are left orthogonal. It is *right orthogonal* if all component tensors $x_2, \ldots, x_d$ are right orthogonal.

Lemma 2.1 [36]**.** *For every tensor $x \in \mathbb{R}^{\boldsymbol{n}}$ of order d we can find left and right orthogonal decompositions.*

For technical purposes it is also useful to define the so-called *interface tensors*, which are based on left and right orthogonal decompositions.

Definition 2.13. Let x be a tensor train of order d with rank tuple $\boldsymbol{r}$.

For every $k = 1, \ldots, d$ and $\ell = 1, \ldots, r_k$, the ℓ-th *left interface vector* is given by

$$\tau^{\leq}_{k,\ell}(x)(i_1, i_2, \cdots, i_k) = x_1(i_1, j_1) \cdots x_k(j_{k-1}, i_k, \ell)$$

where x is assumed to be left orthogonal. The ℓ-th *right interface vector* is given by

$$\tau^{\geq}_{k+1,\ell}(x)(i_{k+1}, \cdots, i_d) = x_{k+1}(\ell, i_{k+1}, j_{k+1}) \cdots x_d(j_{d-1}, i_d)$$

where x is assumed to be right orthogonal.

2.4 Sets of Polynomials

In this section we specify the setup for our method and define the majority of the different sets of polynomials that are used. We start by defining dictionaries of one dimensional functions which we then use to construct the different sets of high-dimensional functions.

Definition 2.14. Let $p \in \mathbb{N}$ be given. A function dictionary of size p is a vector valued function $\Psi = (\Psi_1, \ldots, \Psi_p)\colon \mathbb{R} \to \mathbb{R}^p$.

Example 2.15. Two simple examples of a function dictionary that we use in this work are given by the monomial basis of dimension p, i.e.

$$\Psi_{\mathrm{monomial}}(x) = \begin{pmatrix} 1 & x & x^2 & \ldots & x^{p-1} \end{pmatrix}^T \tag{4}$$

and by the basis of the first p Legendre polynomials, i.e.

$$\Psi_{\mathrm{Legendre}}(x) = \begin{pmatrix} 1 & x & \frac{1}{2}(3x^2 - 1) & \frac{1}{2}(5x^3 - 3x) & \ldots \end{pmatrix}^T. \tag{5}$$

Using function dictionaries we can define the following high-dimensional space of multivariate functions. Let Ψ be a function dictionary of size $p \in \mathbb{N}$. The d-th order product space that corresponds to the function dictionary Ψ is the linear span

$$V_p^d := \langle \bigotimes_{k=1}^{d} \Psi_{m_k} : \boldsymbol{m} \in \mathbb{N}_p^d \rangle. \tag{6}$$

This means that every function $u \in V_p^d$ can be written as

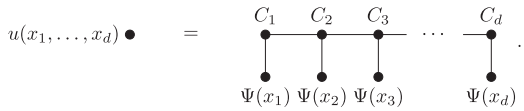
$$u(x_1, \ldots, x_d) = c(i_1, \ldots, i_d) \prod_{k=1}^{d} \Psi(x_k)(i_k) \tag{7}$$

with a coefficient tensor $c \in \mathbb{R}^{\boldsymbol{p}}$ where $\boldsymbol{p} = (p, \ldots, p)$ is a dimension tuple of order d. Note that equation **Eq. 7** uses the index notation

from Definition 2.6 with arbitrary but fixed x_k's. Since $\mathbb{R}^{\boldsymbol{p}}$ is an intractably large space, it makes sense for numerical purposes to consider the subset

$$T_r\left(V_p^d\right) \coloneqq \left\{u \in V_p^d : \ \mathrm{TT}-\mathrm{rank}\,(c) \le r\right\} \tag{8}$$

where the TT rank of the coefficient is bounded. Every $u \in T_r(V_p^d)$ can thus be represented graphically as



where the C_k's are the components of the tensor train representation of the coefficient tensor $c \in \mathbb{R}^{\boldsymbol{p}}$ of $u \in V_p^d$.

Remark 2.16. In this way every tensor $c \in \mathbb{R}^{\boldsymbol{p}}$ (in the tensor train format) corresponds one to one to a function $u \in V_p^d$.

An important subspace of V_p^d is the space of homogeneous polynomials. For the purpose of this paper we define the subspace of homogeneous polynomials of degree g as the space

$$W_g^d \coloneqq \langle \otimes_{k=1}^d \Psi_{m_k} : \ \boldsymbol{m} \in \mathbb{N}_p^d \quad \text{and} \quad \sum_{k=1}^d m_k = d + g \rangle, \tag{10}$$

where again $\langle \bullet \rangle$ is the linear span. From this definition it is easy to see that a homogeneous polynomial of degree g can be represented as an element of V_p^d where the coefficient tensor c satisfies

$$c(m_1, \ldots, m_d) = 0, \quad \text{if} \quad \sum_{k=1}^d m_k \neq d + g.$$

In *Theoretical Foundation* we will introduce an efficient representation of such coefficient tensors c in a block sparse tensor format.

Using W_g^d we can also define the space of polynomials of degree at most g by

$$S_g^d = \oplus_{\tilde{g}=0}^g W_{\tilde{g}}^d. \tag{11}$$

Based on this characterization we will define a block-sparse tensor train version of this space in *Theoretical Foundation.*

2.5 Parametrizing Homogeneous Polynomials by Symmetric Tensors

In algebraic geometry the space W_g^d is considered classically only for the dictionary Ψ_{monomial} of monomials and is typically parameterized by a symmetric tensor

$$u(x) = B\left(i_1, \cdots, i_g\right) \cdot x(i_1) \cdots x\left(i_g\right), \quad x \in \mathbb{R}^d \tag{12}$$

where $\boldsymbol{d} = (d, \ldots, d)$ is a dimension tuple of order g and $B \in \mathbb{R}^{\boldsymbol{d}}$ satisfies $B(m_1, \ldots, m_g) = B(\sigma(m_1, \ldots, m_g))$ for every permutation σ in the symmetric group S_g. We conclude this section by showing how the representation **Eq. 7** can be calculated from the symmetric tensor representation **Eq. 12**, and vice versa. By equating coefficients we find that for every $(m_1, \ldots, m_d) \in \mathbb{N}_p^d$ either $m_1 + \cdots + m_d \neq d + g$ and $c(m_1, \ldots, m_d) = 0$ or

$$\begin{aligned} c(m_1, \ldots, m_d) &= \sum_{\{\sigma(\boldsymbol{n}) : \ \sigma \in S_g\}} B\left(\sigma\left(n_1, \ldots, n_g\right)\right) \quad \text{where} \quad \left(n_1, \ldots, n_g\right) \\ &= \left(\underbrace{1, \ldots, 1}_{m_1 - 1 \text{ times}}, \underbrace{2, \ldots, 2}_{m_2 - 1 \text{ times}}, \ldots,\right) \in \mathbb{N}_d^g. \end{aligned}$$

Since B is symmetric the sum simplifies to

$$\sum_{\{\sigma(\boldsymbol{n}) : \ \sigma \in S_g\}} B\left(\sigma\left(n_1, \ldots, n_g\right)\right) = \binom{g}{m_1 - 1, \ldots, m_d - 1} B\left(n_1, \ldots, n_g\right).$$

From this follows that for $(n_1, \ldots, n_g) \in \mathbb{N}_d^g$

$$B\left(n_1, \ldots, n_g\right) = \frac{1}{\binom{g}{m_1 - 1, \ldots, m_d - 1}} c(m_1, \ldots, m_d) \quad \text{where}$$
$$m_k = 1 + \sum_{\ell=1}^g \delta_{k, n_\ell} \quad \text{for all} \quad k = 1, \ldots, d$$

and $\delta_{\mathrm{k},\ell}$ denotes the *Kronecker delta*. This demonstrates how our approach can alleviate the difficulties that arise when symmetric tensors are represented in the hierarchical tucker format [37] in a very simple fashion.

2.6 Least Squares

Let in the following V_p^d be the product space of a function dictionary Ψ such that $V_p^d \subseteq L_2(\Omega)$. Consider a high-dimensional function $f \in L_2(\Omega)$ on some domain $\Omega \subset \mathbb{R}^d$ and assume that the point-wise evaluation $f(x)$ is well-defined for $x \in \Omega$. In practice it is often possible to choose Ω as a product domain $\Omega = \Omega_1 \times \Omega_2 \times \cdots \Omega_d$ by extending f accordingly. To find the best approximation u_W of f in the space $W \subseteq V_p^d$ we then need to solve the problem

$$u_W = \mathrm{argmin}_{u \in W} \|f - u\|_{L_2(\Omega)}^2. \tag{13}$$

A practical problem that often arises when computing u_W is that computing the $L_2(\Omega)$-norm is intractable for large d. Instead of using classical quadrature rules one often resorts to a Monte Carlo estimation of the high-dimensional integral. This means one draws M random samples $\{x^{(m)}\}_{m=1,\ldots,M}$ from Ω and estimates

$$\|f - u\|_{L_2(\Omega)}^2 \approx \frac{1}{M} \sum_{m=1}^M \|f\left(x^{(m)}\right) - u\left(x^{(m)}\right)\|_{\mathrm{F}}^2,$$

where $\|\cdot\|_F$ is the Frobenius norm. With this approximation we can define an empirical version of u_W as

$$u_{W,M} = \operatorname*{argmin}_{u \in W} \frac{1}{M} \sum_{m=1}^M \|f\left(x^{(m)}\right) - u\left(x^{(m)}\right)\|_{\mathrm{F}}^2. \tag{14}$$

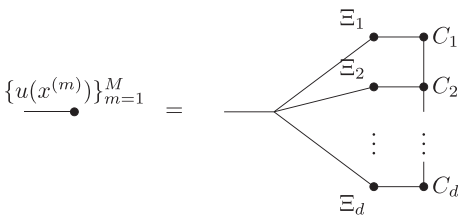
For a linear space W, computing $u_{W,M}$ amounts to solving a linear system and does not pose an algorithmic problem. We

use the remainder of this section to comment on the minimization problem **Eq. 14** when a set of tensor trains is used instead.

Given samples $(x^{(m)})_{m=1,\dots,M}$ we can evaluate $u \in V_p^d$ for each $x^{(m)} = (x_1^{(m)}, \dots, x_d^{(m)})$ using **Eq. 7** If the coefficient tensor c of u can be represented in the TT format then we can use **Eq. 9** to perform this evaluation efficiently for all samples $(x^{(m)})_{m=1,\dots,M}$ at once. For this we introduce for each $k = 1, \dots, d$ the matrix

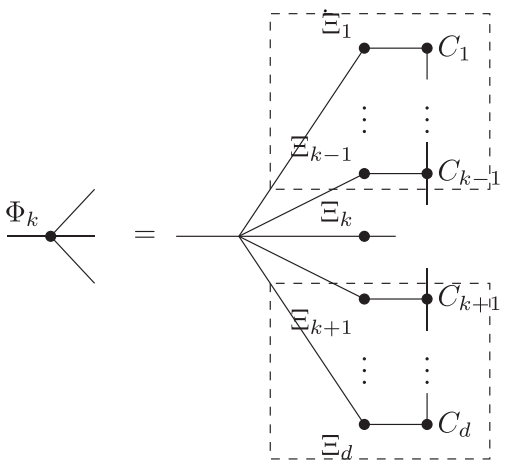
$$\Xi_k = \begin{pmatrix} \Psi(x_k^{(1)}) & \dots & \Psi(x_k^{(M)}) \end{pmatrix} \in \mathbb{R}^{p \times M}. \tag{15}$$

Then the M-dimensional vector of evaluations of u at all given sample points is given by.



where we use Operation **Eq. 2** to join the different M-dimensional indices.

The alternating least-squares algorithm cyclically updates each component tensor C_k by minimizing the residual corresponding to this contraction. To formalize this we define the operator $\Phi_k \in \mathbb{R}^{M \times r_{k-1} \times n_k \times r_k}$ as



(16)

Then the update for C_k is given by a minimal residual solution of the linear system

$$C_k = \underset{C \in \mathbb{R}^{r_{k-1} \times n_k \times r_k}}{\operatorname{argmin}} \|\Phi_k(j, i_1, i_2, i_3) \cdot C(i_1, i_2, i_3) - F(j)\|_F^2$$

where $F(m) := y^{(m)} := f(x^{(m)})$ and i_1, i_2, i_3, j are symbolic indices of dimensions r_{k-1}, n_k, r_k, M, respectively. The particular algorithm that is used for this minimization may be adapted to the problem at hand. These contractions are the basis for our algorithms in *Method Description*. We refer to [19] for more details on the ALS algorithm.

Note that it is possible to reuse parts of the contractions in Φ_k through so called *stacks*. In this way not the entire contraction has to be computed for every k. The dashed boxes mark the parts of the contraction that can be reused. Details on that can be found in [38].

3 THEORETICAL FOUNDATION

3.1 Sample Complexity for Polynomials

The accuracy of the solution $u_{W,M}$ of **Eq. 14** in relation to u_W is subject to tremendous interest on the part of the mathematics community. Two particular papers that consider this problem are [32,39]. While the former provides sharper error bounds for the case of linear ansatz spaces the latter generalizes the work and is applicable to tensor network spaces. We now recall the relevant result for convenience.

Proposition 3.1. *For any set* $W \subseteq L^2(\Omega) \cap L^\infty(\Omega)$*, define the* variation constant

$$K(W) := \sup_{v \in W \setminus \{0\}} \frac{\|v\|_{L^\infty(\Omega)}^2}{\|v\|_{L^2(\Omega)}^2}.$$

Let $\delta \in (0, 2^{-1/2})$*. If* W *is a subset of a finite dimensional linear space and* $k := \max\{K(\{f - u_W\}), K(\{u_W\} - W)\} < \infty$ *it holds that*

$$\mathbb{P}\left[\|f - u_{W,M}\|_{L^2(\Omega)} \le (3 + 4\delta)\|f - u_W\|_{L^2(\Omega)}\right] \ge 1 - q$$

where q *decreases exponentially with a rate of* $\ln(q) \in \mathcal{O}(-M\delta^2 k^{-2})$.

Proof. Since $k < \infty$, Theorems 2.7 and 2.12 in [32] ensure that

$$\|f - u_{W,M}\|_{L^2(\Omega)} \le \left(1 + 2\sqrt{\frac{1+\delta}{1-\delta}}\right)\|f - u_W\|_{L^2(\Omega)}.$$

holds with a probability of at least $1 - 2C\exp(-\frac{1}{2}M\delta^2 k^{-2})$. The constant C is independent of M and, since W is a subset of a finite dimensional linear space, depends only polynomially on δ and k^{-1}. For $\delta \in (0, 2^{-1/2})$ it holds that $\sqrt{\frac{1+\delta}{1-\delta}} \le 1 + 2\delta$. This concludes the proof.

Note that the value of k depends only on f and on the set W but not on the particular choice of representation of W. However, the variation constant of spaces like V_p^d still depends on the underlying dictionary Ψ. Although the proposition indicates that a low value of k is necessary to achieve a fast convergence, the tensor product spaces V_p^d considered thus far does not exhibit a small variation constant. The consequence of Proposition 3.1 is that elements of this space are hard to learn in general and may require an infeasible number of samples. To see this consider $\Omega = [-1,1]^d$ and the function dictionary Ψ_{Legendre} of Legendre polynomials **Eq. 5**. Let $L \subseteq \mathbb{N}_p^d$ and define $P_\ell(x) := \prod_{k=1}^d \sqrt{2\ell_k - 1}(\Psi_{\text{Legendre}}(x_k))_{\ell_k}$ for all $\ell \in L$. Then, $\{P_\ell\}_{\ell \in L}$ is an L^2-orthonormal basis for the linear subspace $V := \langle P_\ell : \ell \in L \rangle \subseteq V_p^d$ and one can show that

$$K(V) = \sup_{x \in \Omega} \sum_{\ell \in L} P_\ell(x)^2 = \sum_{\ell \in L} \prod_{k=1}^d (2\ell_k - 1), \tag{17}$$

by using techniques from [[32], *Sample Complexity for Polynomials*] and the fact that each P_ℓ attains its maximum at 1. If $L = \mathbb{N}_p^d$, we can

interchange the sum and product in **Eq. 17** and can conclude that $K(V_p^d) = p^{2d}$. This means that we have to restrict the space V_p^d to obtain an admissible variation constant. We propose to use the space W_g^d of homogeneous polynomials of degree g. Employing **Eq. 17** with $L = \{\boldsymbol{\ell} : |\boldsymbol{\ell}| = d + g\}$ we obtain the upper bound

$$K\left(W_g^d\right) \leq \binom{d-1+g}{d-1} \max_{|\boldsymbol{\ell}|=d+g} \prod_{k=1}^{d} (2\ell_k - 1)$$
$$\leq \binom{d-1+g}{d-1} \left(2\left\lfloor \frac{g}{d} \right\rfloor + 3\right)^{g \bmod d} \left(2\left\lfloor \frac{g}{d} \right\rfloor + 1\right)^{d-g \bmod d}$$

where the maximum is estimated by observing that $(2(\ell_1 + 1) - 1)(2\ell_2 - 1) \leq (2\ell_1 - 1)(2(\ell_2 + 1) - 1) \Leftrightarrow \ell_2 \leq \ell_1$. For $g \leq d$ this results in the simplified bound $K(W_g^d) \leq (3e\frac{d-1+g}{g})^g$, where e is the Euler number. This improves the variation constant substantially compared to the bound $K(V_p^d) \leq p^{2d}$, when $g \ll d$. A similar bound for the dictionary of monomials Ψ_{monomial} is more involved but can theoretically be computed in the same way.

In this work, we focus on the case where the samples are drawn according to a probability measure on Ω. This however is not a necessity and it is indeed beneficial to draw the samples from an adapted sampling measure. Doing so, the theory in [32] ensures that $K(V) = \dim(V)$ for all linear spaces V — independent of the underlying dictionary Ψ. This in turn leads to the bounds $K(V_p^d) = p^d$ and $K(W_g^d) = \binom{d-1+g}{d-1} \leq \left(e\frac{d-1+g}{g}\right)g$ for $g \leq d$. These optimally weighted least-squares methods however, are not the focus of this work and we refer the interested reader to the works [39,40].

3.2 Block Sparse Tensor Trains

Now that we have seen that it is advantageous to restrict ourselves to the space W_g^d we need to find a way to do so without loosing the advantages of the tensor train format. In [29] it was rediscovered from the physics community (see [28]) that if a tensor train is an eigenvector of certain Laplace-like operators it admits a block sparse structure. This means for a tensor train c the components C_k have zero blocks. Furthermore, this block sparse structure is preserved under key operations, like e.g. the TT-SVD. One possible operator which introduces such a structure is the Laplace-like operator

$$L = \sum_{k=1}^{d} \left(\bigotimes_{\ell=1}^{k-1} I_p\right) \otimes \operatorname{diag}(0, 1, \ldots, p-1) \otimes \left(\bigotimes_{\ell=k+1}^{d} I_p\right). \tag{18}$$

This is the operator mentioned in the introduction encoding a quantum symmetry. In the context of quantum mechanics this operator is known as the bosonic particle number operator but we simply call it the degree operator. The reason for this is that for the function dictionary of monomials Ψ_{monomial} the eigenspaces of L for eigenvalue g are associated with homogeneous polynomials of degreee g. Simply put, if the coefficient tensor c for the multivariate polynomial $u \in V_p^d$ is an eigenvector of L with eigenvalue g, then u is homogeneous and the degree of u is g. In general there are polynomials in V_p^d with degree up to $(p-1)d$. To state the results on the block-sparse representation of the coefficient tensor we need the partial operators

$$L_k^{\leq} = \sum_{m=1}^{k} \left(\bigotimes_{\ell=1}^{m-1} I_p\right) \otimes \operatorname{diag}(0, 1, \ldots, p-1) \otimes \left(\bigotimes_{\ell=m+1}^{k} I_p\right)$$
$$L_{k+1}^{\geq} = \sum_{m=k+1}^{d} \left(\bigotimes_{\ell=k+1}^{m-1} I_p\right) \otimes \operatorname{diag}(0, 1, \ldots, p-1) \otimes \left(\bigotimes_{\ell=m+1}^{d} I_p\right),$$

for which we have

$$L = L_k^{\leq} \otimes \bigotimes_{\ell=k+1}^{d} I_p + \bigotimes_{l=1}^{k} I_p \otimes L_{k+1}^{\geq}.$$

In the following we adopt the notation $x = Lc$ to abbreviate the equation

$$x(i_1, \ldots, i_d) = L(i_1, \ldots, i_d, j_1, \ldots, j_d)c(j_1, \ldots, j_d)$$

where L is a tensor operator acting on a tensor c with result x.

Recall that by Remark 2.16 every TT corresponds to a polynomial by multiplying function dictionaries onto the cores. This means that for every $\ell = 1, \ldots, r$ the TT $\tau_{k,\ell}^{\leq}(c)$ corresponds to a polynomial in the variables $x_1, \ldots, x_k$ and the TT $\tau_{k+1,\ell}^{\geq}(c)$ corresponds to a polynomial in the variables $x_{k+1}, \ldots, x_d$. In general these polynomials are not homogeneous, i.e. they are not eigenvectors of the degree operators $L_k^{\leq}$ and $L_{k+1}^{\geq}$. But since TTs are not uniquely defined (cf. Remark 2.10) it is possible to find transformations of the component tensors C_k and C_{k+1} that do not change the tensor c or the rank r but result in a representation where each $\tau_{k,\ell}^{\leq}(c)$ and each $\tau_{k+1,\ell}^{\geq}(c)$ correspond to a homogeneous polynomial. Thus, if c represents a homogeneous polynomial of degree g and $\tau_{k,\ell}^{\leq}(c)$ is homogeneous with $\deg(\tau_{k,\ell}^{\leq}(c)) = \tilde{g}$ then $\tau_{k+1,\ell}^{\geq}(c)$ must be homogeneous with $\deg(\tau_{k,\ell}^{\geq}(c)) = g - \tilde{g}$.

This is put rigorously in the first assertion in the subsequent Theorem 3.2. There $\mathcal{S}_{k,\tilde{g}}$ contains all the indices ℓ for which the reduced basis polynomials satisfy $\deg(\tau_{k,\ell}^{\leq}(c)) = \tilde{g}$. Equivalently, it groups the basis functions $\tau_{k+1,\ell}^{\geq}(c)$ into functions of order $g - \tilde{g}$. The second assertion in Theorem 3.2 states that we can only obtain a homogeneous polynomial of degree $\tilde{g} + m$ in the variables $x_1, \ldots, x_k$ by multiplying a homogeneous polynomial of degree $\tilde{g}$ in the variables $x_1, \ldots, x_{k-1}$ with a univariate polynomial of degree m in the variable x_k. This provides a constructive argument for the proof and can be used to ensure block-sparsity in the implementation. Note that this condition forces entire blocks in the component tensor C_k in equation **(20)** to be zero and thus decreases the degrees of freedom.

Theorem 3.2 [[29], *Theorem 3.2]. Let* $\boldsymbol{p} = (p, \ldots, p)$ *be a dimension tuple of size d and* $c \in \mathbb{R}^{\boldsymbol{p}}$ $\{0\}$, *be a tensor train of rank* $r = (r_1, \ldots, r_{d-1})$. *Then* $Lc = gc$ *if and only if c has a representation with component tensors* $C_k \in \mathbb{R}^{r_{k-1} \times p \times r_k}$ *that satisfies the following two properties.*

1. For all $\tilde{g} \in \{0, 1, \ldots, g\}$ *there exist* $\mathcal{S}_{k,\tilde{g}} \subseteq \{1, \ldots, r_k\}$ *such that the left and right unfoldings satsify*

$$\begin{aligned} L_k^{\leq} \tau_{k,\ell}^{\leq}(c) &= \tilde{g} \tau_{k,\ell}^{\leq}(c) \\ L_{k+1}^{\geq} \tau_{k+1,\ell}^{\geq}(c) &= (g - \tilde{g}) \tau_{k+1,\ell}^{\geq}(c) \end{aligned} \tag{19}$$

for $\ell \in \mathcal{S}_{k,\tilde{g}}$.

2. The component tensors satisfy a block structure in the sets $\mathcal{S}_{k,\tilde{g}}$ *for* $m = 1, \ldots, p$

$$C_k(\ell_1, m, \ell_2) \neq 0 \quad \Rightarrow \quad \exists\, 0 \leq \tilde{g} \leq g - (m-1): \\ \ell_1 \in \mathcal{S}_{k-1,\tilde{g}} \wedge \ell_2 \in \mathcal{S}_{k,\tilde{g}+(m-1)} \tag{20}$$

where we set $\mathcal{S}_{0,0} = \mathcal{S}_{d,g} = \{1\}$.

Note that this generalizes to other dictionaries and is not restricted to monomials.

Although, block sparsity also appears for $g + 1 \neq p$ we restrict ourselves to the case $g + 1 = p$ in this work. Note that then the eigenspace of L for the eigenvalue g has a dimension equal to the dimension of the space of homogeneous polynomials, namely $\binom{d-1+g}{d-1}$. Defining $\rho_{k,\tilde{g}} := |\mathcal{S}_{k,\tilde{g}}|$, we can derive the following rank bounds.

Lemma 3.3 [[29], *Lemma 3.6]. Let* $\boldsymbol{p} = (p, \ldots, p)$ *be a dimension tuple of size d and* $c \in \mathbb{R}^{\boldsymbol{p}}$ {0}, *with Lc = gc. Assume that* $g + 1 = p$ *then the block sizes* $\rho_{k,\tilde{g}}$ *from Theorem* 3.2 *are bounded by*

$$\rho_{k,\,\tilde{g}} \leq \min\left\{\binom{k+\tilde{g}-1}{k-1}, \binom{d-k+g-\tilde{g}-1}{d-1}\right\} \tag{21}$$

for all $k = 1, \ldots, d-1$ *and* $\tilde{g} = 0, \ldots, g$ *and* $\rho_{k,0} = \rho_{k,g} = 1$.

The proof of this lemma is based on a simple combinatorial argument. For every k consider the size of the groups $\rho_{k-1,\tilde{g}}$ for $\bar{g} \leq \tilde{g}$. Then $\rho_{k,\tilde{g}}$ can not exceed the sum of these sizes. Similarly, $\rho_{k,\bar{g}}$ can not exceed $\sum_{\tilde{g} \leq \bar{g}} \rho_{k+1,\tilde{g}}$. Solving these recurrence relations yields the bound.

Example 3.1 (*Block Sparsity*)**.** Let $p = 4$ and $g = 3$ be given and let c be a tensor train such that $Lc = gc$. Then for $k = 2, \ldots, d-1$ the component tensors C_k of c exhibit the following block sparsity (up to permutation). For indices i of order r_{k-1} and j of order r_k

$$C_k(i,1,j) = \begin{pmatrix} * & 0 & 0 & 0 \\ 0 & * & 0 & 0 \\ 0 & 0 & * & 0 \\ 0 & 0 & 0 & * \end{pmatrix} \quad C_k(i,2,j) = \begin{pmatrix} 0 & * & 0 & 0 \\ 0 & 0 & * & 0 \\ 0 & 0 & 0 & * \\ 0 & 0 & 0 & 0 \end{pmatrix}$$
$$C_k(i,3,j) = \begin{pmatrix} 0 & 0 & * & 0 \\ 0 & 0 & 0 & * \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad C_k(i,4,j) = \begin{pmatrix} 0 & 0 & 0 & * \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}.$$

This block structure results from sorting the indices i and j in such a way that $\max \mathcal{S}_{k,\tilde{g}} + 1 = \min \mathcal{S}_{k,\tilde{g}+1}$ for every $\tilde{g}$.

The maximal block sizes $\rho_{k,\tilde{g}}$ for $k = 1, \ldots, d-1$ are given by

$$\rho_{k,0} = 1, \quad \rho_{k,1} = \min\{k, d-k\}, \quad \rho_{k,2} = \min\{k, d-k\}, \quad \rho_{k,3} = 1.$$

As one can see by Lemma 3.3 the block sizes $\rho_{k,\tilde{g}}$ can still be quite high.

The expressive power of tensor train parametrizations can be understood by different concepts, such as locality or self similarity. We use the remainder of this section to provide d-independent rank bounds in the context of locality.

Definition 3.2. Let $u \in W_g^d$ be a homogeneous polynomial and B be the symmetric coefficient tensor introduced in *Parametrizing homogeneous polynomials by symmetric tensors* We say that u has a variable locality of K_{loc} if $B(\ell_1, \ldots, \ell_g) = 0$ for all $(\ell_1, \ldots, \ell_g) \in \mathbb{N}_d^g$ with

$$\max\{|\ell_{m_1} - \ell_{m_2}| : \; m_1, m_2 = 1, \ldots, g\} > K_{\text{loc}}.$$

Example 3.3. Let u be a homogeneous polynomial of degree 2 with variable locality K_{loc}. Then the symmetric matrix B (cf. **Eq. 12**) is K_{loc}-banded. For $K_{\text{loc}} = 0$ this means that B is diagonal and that u takes the form

$$u(x) = \sum_{\ell=1}^{d} B_{\ell\ell} x_\ell^2.$$

This shows that variable locality removes mixed terms.

Remark 3.4. The locality condition in the following Theorem 3.4 is a sufficient, but in no way necessary, condition for a low rank. But since locality is a prominent feature of many physical phenomena, this condition allows us to identify an entire class of highly relevant functions which can be approximated very efficiently.

Consider, for example, a many-body system in one dimension, where each body is described by position and velocity coordinates. If the influence of neighboring bodies is much higher than the influence of more distant ones, the coefficients of the polynomial parts that depend on multiple variables often can be neglected. The forces in this system then exhibit a locality structure. An example of this is given in equation **Eq. 6** in [3], where this structure is exhibited by the force that acts on the bodies. A similar structure also appears in the microscopic traffic models in *Notation* of [41].

Another example is given by the polynomial chaos expansion of the stochastic process

$$X_t(\xi_1, \ldots, \xi_t) := c(i_1, \ldots, i_t) \prod_{k=1}^{t} \Psi(\xi_k)(i_k)$$

for $t \in \mathbb{N}$, where Ψ is the function dictionary of Hermite polynomials. In many applications, it is justified to assume that the magnitude of the covariance $\text{Cov}(X_{t_1}, X_{t_2})$ decays with the distance of the indices $|t_1 - t_2|$ If the covariance decays fast enough, the coefficient tensor exhibits approximate locality, i.e. it can be well approximated by a coefficient tensor that satisfies the locality condition. Examples of this are Gaussian processes with a Matérn kernel [42,43] or Markov processes.

Theorem 3.4. *Let* $\boldsymbol{p} = (p, \ldots, p)$ *be a dimension tuple of size d and* $c \in \mathbb{R}^{\boldsymbol{p}}$ {0} *correspond to a homogeneous polynomial of degree* $g + 1 = p$ *(i.e. Lc = gc) with variable locality* K_{loc}. *Then the block sizes* $\rho_{k\tilde{g}}$ *are bounded by*

$$\rho_{k,\tilde{g}} \leq \sum_{\ell=1}^{K_{\text{loc}}} \min\left\{\binom{K_{\text{loc}} - \ell + 1 + \tilde{g} - 2}{K_{\text{loc}} - \ell}, \binom{\ell + g - \tilde{g} - 2}{\ell - 1}\right\} \tag{22}$$

for all $k = 1, \ldots, d-1$ *and* $\tilde{g} = 1, \ldots, g-1$ *as well as* $\rho_{k,0} = \rho_{k,g} = 1$.

Proof. For fixed $g > 0$ and a fixed component C_k recall that for each l the tensor $\tau_{k,l}^{\leq}(c)$ corresponds to a reduced basis function v_l in the variables $x_1, \ldots, x_k$ and that for each l the tensor $\tau_{k+1,l}^{\geq}(c)$ corresponds to a reduced basis function w_l in the variables $x_{k+1}, \ldots, x_d$. Further recall that the sets $\mathcal{S}_{k,\tilde{g}}$ group these v_l and w_l. For all $l \in \mathcal{S}_{k,\tilde{g}}$ it holds that $\deg(v_l) = \tilde{g}$ and $\deg(w_l) = g - \tilde{g}$. For $\tilde{g} = 0$ and $\tilde{g} = g$ we know from Lemma 3.3 that $\rho_{k,\tilde{g}} = 1$. Now fix any $0 < \tilde{g} < g$ and arrange all the polynomials v_l of degree $\tilde{g}$ in a vector v and all polynomials w_l of degree $g - \tilde{g}$ in a vector w. Then every polynomial of the form $v^\top Q w$ for some matrix Q satisfies the degree constraint and the maximal

possible rank of Q provides an upper bound for the block size $\rho_{k,\tilde{g}}$. However, due to the locality constraint we know that certain entries of Q have to be zero. We denote a variable of a polynomial as inactive if the polynomial is constant with respect to changes in this variable and active otherwise. Assume that the polynomials in v are ordered (ascendingly) according to the smallest index of their active variables and that the polynomials in w are ordered (ascendingly) according to the largest index of their active variables. With this ordering Q takes the form

$$Q = \begin{pmatrix} & & & & 0 \\ Q_1 & & & & \\ * & Q_2 & & & \\ * & * & Q_3 & & \\ \vdots & \vdots & \vdots & \ddots & \\ * & * & * & \cdots & Q_{K_{\mathrm{loc}}} \end{pmatrix}.$$

This means that for $\ell = 1, \ldots, K_{\mathrm{loc}}$ each block Q_ℓ matches a polynomial v_l of degree $\tilde{g}$ in the variables $x_{k-K_{\mathrm{loc}}+\ell}, \ldots, x_k$ with a polynomial w_l of degree $g - \tilde{g}$ in the variables $x_{k+1}, \ldots, x_{k+\ell}$.

Observe that the number of rows in Q_ℓ decreases while the number columns increases with ℓ. This means that we can subdivide Q as

$$Q = \begin{pmatrix} 0 & 0 & 0 \\ Q_{\mathrm{C}} & 0 & 0 \\ * & Q_{\mathrm{R}} & 0 \end{pmatrix},$$

where Q_{C} contains the blocks Q_ℓ with more rows than columns (i.e. full column rank) and Q_{R} contains the blocks Q_ℓ with more columns than rows (i.e. full row rank). So Q_{C} is a tall-and-skinny matrix while Q_{R} is a short-and-wide matrix and the rank for general Q is bounded by the sum over the column sizes of the Q_ℓ in Q_{C} plus the sum over the row sizes of the Q_ℓ in Q_{R} i.e.

$$rank(Q) = \sum_{\ell=1}^{K_{\mathrm{loc}}} rank(Q_\ell).$$

To conclude the proof it remains to compute the row and column sizes of Q_ℓ. Recall that the number of rows of Q_ℓ equals the number of polynomials u of degree $\tilde{g}$ in the variables $x_{k-K_{\mathrm{loc}}+\ell}, \ldots, x_k$ that can be represented as $u(x_{k-K_{\mathrm{loc}}+\ell}, \ldots, x_k) = x_{k-K_{\mathrm{loc}}+\ell}\tilde{u}(x_{k-K_{\mathrm{loc}}+\ell}, \ldots, x_k)$. This corresponds to all possible $\tilde{u}$ of degree $\tilde{g} - 1$ in the $K_{\mathrm{loc}} - \ell + 1$ variables $x_{k-K_{\mathrm{loc}}+\ell}, \ldots, x_k$. This means that

$$\#rows(Q_\ell) \leq \binom{K_{\mathrm{loc}} - \ell + 1 + \tilde{g} - 2}{K_{\mathrm{loc}} - \ell}$$

and a similar argument yields

$$\#columns(Q_\ell) \leq \binom{\ell + g - \tilde{g} - 2}{\ell - 1}.$$

This concludes the proof.

This lemma demonstrates how the combination of the model space W_g^d with a tensor network space can reduce the space complexity by incorporating locality.

Remark 3.5. The rank bound in Theorem 3.4 is only sharp for the highest possible rank. The precise bounds can be much smaller, especially for the first and last ranks, but are quite technical to write down. For this reason, we do not provide them.

One sees that the bound only depends on g and K_{loc} and is therefore d-independent.

Remark 3.6. The rank bounds presented in this section do not only hold for the monomial dictionary Ψ_{monomial} but for all polynomial dictionaries Ψ that satisfy $\deg(\Psi_k) = k - 1$ for all $k = 1, \ldots, p$. When we speak of polynomials of degree g, we mean the space $W_g^d = \{v \in V_p^d : \deg(v) = g\}$. For the dictionary of monomials Ψ_{monomial} the space W_g^d contains only homogeneous polynomials in the classical sense. However, when the basis of Legendre polynomials Ψ_{Legendre} is used one obtains a space in which the functions are not homogeneous in the classical sense. Note that we use polynomials since they have been applied successfully in practice, but other function dictionaries can be used as well. Also note that the theory is much more general as shown in [29] and is not restricted to the degree counting operator.

With Theorem 3.4 one sees that tensor trains are well suited to parametrize homogeneous polynomials of fixed degree where the symmetric coefficient tensor B (cf. **Eq. 12**) is approximately banded (see also Example 3.3). This means, that there exist an K_{loc} such that the error for a best approximation of B by a tensor B with variable locality K_{loc} is small. However, K_{loc} is not known precisely in practice but can only be assumed by physical understanding of the problem at hand. Therefore, we still rely on rank adaptive schemes to find appropriate rank and block sizes. Moreover, the locality property heavily depends on the ordering of the modes. This ordering can be optimized, for example, by using entropy measures for the correlation of different modes, as it is done in quantum chemistry (cf. [[44], Remark 4.2]) or by model selection methods (cf. [25,27,45,46]).

4 METHOD DESCRIPTION

In this section we utilize the insights of *Theoretical Foundation* to refine the approximation spaces W_g^d and S_g^d and adapt the *alternating least-squares (ALS)* method to solve the related least-squares problems. First, we define the subset

$$B_\rho\left(W_g^d\right) := \Big\{ u \in W_g^d : \ c \text{ is block-sparse with } \rho_{k,\tilde{g}} \leq \rho \text{ for } 0 \leq \tilde{g} \leq g, k = 1, \ldots, d \Big\} \tag{23}$$

and provide an algorithm for the related least-squares problem in Algorithm 1 which is a slightly modified version of the classical ALS [19].[1] With this definition a straight-forward application of the concept of block-sparsity to the space S_g^d is given by

$$S_{g,\rho}^d = \bigotimes_{\tilde{g}=0}^{g} B_\rho\left(W_{\tilde{g}}^d\right). \tag{24}$$

This means that every polynomial in $S_{g,\rho}^d$ can be represented by a sum of orthogonal coefficient tensors[2]

[1]It is possible to include rank adaptivity as in SALSA [21] or bASD [13] and we have noted this in the relevant places.

[2]The orthogonality comes from the symmetry of L which results in orthogonal eigenspaces.

$$\sum_{\tilde{g}=0}^{g} c^{(\tilde{g})} \quad \text{where} \quad Lc^{(\tilde{g})} = \tilde{g}c^{(\tilde{g})}. \tag{25}$$

There is however another, more compact, way to represent this function. Instead of storing $g+1$ different tensor trains $c^{(0)}, \ldots, c^{(g)}$ of order d, we can merge them into a single tensor c of order $d+1$ such that $c(i^d, \tilde{g}) = c^{(\tilde{g})}(i^d)$. The summation over $\tilde{g}$ can then be represented by a contraction of a vector of 1's to the $(d+1)$-th mode. To retain the block-sparse representation we can view the $(d+1)$-th component as an artificial component representing a shadow variable x_{d+1}.

Remark 4.1. The introduction of the shadow variable x_{d+1} contradicts the locality assumptions of Theorem 3.4 and implies that the worst case rank bounds must increase. This can be problematic since the block size contributes quadratically to the number of parameters. However, a proof similar to that of Theorem 3.4 can be made in this setting and one can show that the bounds remain independent of d

$$\rho_{k,\tilde{g}} \leq \underline{1+} \sum_{\ell=1}^{K_{\text{loc}}} \min\left\{ \binom{K_{\text{loc}} - \ell + 1 + \tilde{g} - 2}{K_{\text{loc}} - \ell}, \binom{\ell \,\underline{+1} \, + g - \tilde{g} - 2}{\ell \,\underline{+1} \, - 1} \right\} \tag{26}$$

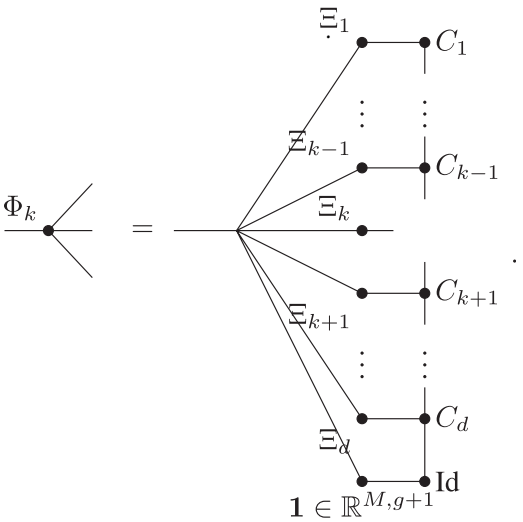
where the changes to **Eq. 22** are underlined. This is crucial, since in practice one can assume locality by physical understanding of the problem at hand. With this statement, we can guarantee that the ranks are only slightly changed by the auxiliary contraction and the locality property is not destroyed.

We denote the set of polynomials that results from this augmented block-sparse tensor train representation as

$$S^{d,\text{aug}}_{g,\rho} \tag{27}$$

where again ρ provides a bound for the block-size in the representation.

Since $S^{d,\text{aug}}_{g,\rho}$ is defined analogously to $B_\rho(W^d_g)$ we can use **Algorithm 1** to solve the related least-squares problem by changing the contraction **Eq. 16** to



(28)

To optimize the coefficient tensors $c^{(0)}, \ldots, c^{(g)}$ in the space $S^d_{g,\rho}$ we resort to an alternating scheme. Since the coefficient tensors are mutually orthogonal we propose to optimize each $c^{(\tilde{g})}$ individually while keeping the other summands $\{c^{(k)}\}_{k \neq \tilde{g}}$ fixed. This means that we solve the problem

$$u^{(\tilde{g})} = \operatorname*{argmin}_{u \in W^d_{\tilde{g}}} \frac{1}{M} \sum_{m=1}^{M} \left\| f\left(x^{(m)}\right) - \sum_{\substack{k=0 \\ k \neq \tilde{g}}}^{g} u^{(k)}\left(x^{(m)}\right) - u\left(x^{(m)}\right) \right\|_{\text{F}}^2 \tag{29}$$

which can be solved using **Algorithm 1**. The original problem **Eq. 14** is then solved by alternating over g until a suitable convergence criterion is met. The complete algorithm is summarized in **Algorithm 2**.

The proposed representation has several advantages. The optimization with the tensor train structure is computationally less demanding than solving directly in S^d_g. Let $D = \dim(S^d_g) = \binom{d}{d+g}$. Then a reconstruction on S^d_g requires to solve a linear system of size $M \times D$ while a microstep in an ALS sweep only requires the solution of systems of size less than Mpr^2 (depending on the block sizes $\rho_{k,\tilde{g}}$). Moreover, the stack contractions as shown in *Least Squares* also benefit from the block sparse structure. This also means that the number of parameters of a full rank r tensor train can be much higher than the number of parameters of several $c^{(m)}$'s which individually have ranks that are even larger than r.

Remark 4.2. Let us comment on the practical pondering behind choosing $S^{d,\text{aug}}_{g,\rho}$ or $S^d_{g,\rho}$ by stating some pros and cons of [1]these parametrizations. We expect that solving the least-squares problem for $S^{d,\text{aug}}_{g,\rho}$ will be faster than for $S^d_{g,\rho}$ since it is computational more efficient to optimize all polynomials simultaneously than every degree individually in an alternating fashion. On the other hand, the hierarchical scheme of the summation approach may allow one to utilize multi-level Monte Carlo approaches. Together with the fact that every degree g possesses a different optimal sampling density this may result in a drastically improved best case sample efficiency for the direct method. Additionally, with $S^d_{g,\rho}$ it is easy to extend the ansatz space simply by increasing g which is not so straight-forward for $S^{d,\text{aug}}_{g,\rho}$. Which approach is superior depends on the problem at hand.

Algorithm 1 | Extended ALS (SALSA) for the least-squares problem on $B_\rho(W^d_g)$

input: Data pairs $(x^{(m)}, y^{(m)}) \in \mathbb{R}^d \times \mathbb{R}$ for $m = 1, \ldots, M$, a function dictionary Ψ, a maximal degree g, and a maximal block size ρ.
output: Coefficent tensor c of a function $u \in B(W^d_g)$ that approximates the data.
For $k = 1, \ldots, d$ compute Ξ_k according to **Eq. 15**;
Initialize the coefficient tensor c for $u \in B(W^d_g)$;
Initialize SALSA parameters;
while *not converged* **do**
  Right orthogonalize c;
  for $k = 1, \ldots, d$ **do**
    Compute Φ_k according to **Eq. 16**;
    Compute the index set $\mathcal{I}$ of the non-zeros components in C_k according to **Eq. 20**;
    Update C_k by solving the SALSA-regularized version of $\Phi_k(j, i^3) \cdot C_k(i^3) = y(j)$ restricted to $i^3 \in \mathcal{I}$;
    Left orthogonalize C_k and adapt the k_{th} rank while respecting block size bounds ρ and **Eq. 21**;
  end
  Update SALSA parameters;
end
return c

Algorithm 2 | Alternating extended ALS (SALSA) for the least-squares problem on $S^d_{g,\rho}$

input: Data pairs $(x^{(m)}, y^{(m)}) \in \mathbb{R}^d \times \mathbb{R}$ for $m = 1, \ldots, M$, a function dictionary Ψ, a maximal degree g, and a maximal block size ρ.
output: Coefficent tensors $c^{(0)}, \ldots, c^{(g)}$ of a function $u \in S^d_{g,\rho}$ that approximates the data.
Initialize the coefficient tensors $c^{(\tilde{g})}$ of $u^{(\tilde{g})} \in B_\rho(W^d_{\tilde{g}})$ for $\tilde{g} = 0, \ldots, g$;
while *not converged* **do**
  for $\tilde{g} = 0, \ldots, g$ **do**
    Compute $z^{(m)} := y^{(m)} - \sum_{k \neq \tilde{g}} u^{(k)}(x^{(m)})$ for $m = 1, \ldots, M$;
    Update $c^{(\tilde{g})}$ by using Algorithm 1 on the data pairs $(x^{(m)}, z^{(m)})$ for $m = 1, \ldots, M$;
  end
end
return $c^{(\tilde{g})}$ *for* $\tilde{g} = 0, \ldots, g$

5 NUMERICAL RESULTS

In this section we illustrate the numerical viability of the proposed framework on some simple but common problems. We estimate the relative errors on test sets with respect to the sought function f and are interested in the required number of samples leading to recovery. Our implementation is meant only as a proof of concept and does not lay any emphasis on efficiency. The rank is chosen a priori, the stopping criteria are naïvely implemented and rank adaptivity, as would be provided by SALSA, is missing all together.[3] For this reason we only compare the methods in terms of degrees of freedom and accuracy and not in terms of computing time. These are relevant quantities nonetheless, since the degrees of freedom are often the limiting factor in high dimensions and the computing time is directly related to the number of degrees of freedom.

In the following we always assume $p = g + 1$. We also restrict the group sizes to be bounded by the parameter $\rho_{\max}$. In our experiments we choose $\rho_{\max}$ without any particular strategy but ideally, $\rho_{\max}$ would be determined adaptively by the use of SALSA, which we did not do in this work. For every sample size the error plots show the distribution of the errors between the 0.15 and 0.85 quantile. The code for all experiments has been made publicly available at https://github.com/ptrunschke/block_sparse_tt.

5.1 Riccati Equation

In this section we consider the closed-loop linear quadratic optimal control problem

$$\begin{aligned} \underset{u}{\text{minimize}} \quad & \|y\|^2_{L^2([0,\infty]\times[-1,1])} + \lambda \|u\|^2_{L^2([0,\infty])} \\ \text{subject to} \quad & \partial_t y = \partial_x^2 y + u(t)\chi_{[-0.4,0.4]}, \\ (t,x) \in [0,\infty] \times [-1,1] \quad & \\ & y(0,x) = y_0(x), \; x \in [-1,1] \\ & \partial_x y(t,-1) = \partial_x y(t,1) = 0 \end{aligned}$$

After a spatial discretization of the heat equation with finite differences we obtain a d-dimensional system of the form

$$\underset{u}{\text{minimize}} \int_0^\infty \boldsymbol{y}(t)^\top Q \boldsymbol{y}(t) + \lambda u(t)^2 \, \mathrm{d}t \quad \text{subject to} \quad \dot{\boldsymbol{y}} = A\boldsymbol{y} + Bu \quad \text{and} \quad \boldsymbol{y}(0) = \boldsymbol{y}_0.$$

It is well known [47] that the value function for this problem takes the form $v(\boldsymbol{y}_0) = \boldsymbol{y}_0^\top P \boldsymbol{y}_0$ where P can be computed by solving the *algebraic Riccati equation (ARE)*. It is therefore a homogeneous polynomial of degree 2. This function is a perfect example of a function that can be well-approximated in the space W^d_2. We approximate the value function on the domain $\Omega = [-1,1]^d$ for $d = 8$ with the parameters $g = 2$ and $\rho_{\max} = 4$.

In this experiment we use the dictionary of monomials $\Psi = \Psi_{\text{monomial}}$ (cf. **Eq. 4**) and compare the ansatz spaces W^8_2, $B_4(W^8_2)$, $T_6(V^8_3)$ and V^8_3. Since the function $v(x)$ is a general polynomial we use Lemma 3.3 to calculate the maximal block size 4. This guarantees perfect reconstruction since $B_4(W^8_2) = W^8_2$. The rank bound 6 is chosen s.t. $B_4(W^8_2) \subseteq T_6(V^8_3)$. The degrees of freedom of all used spaces are listed in **Table 1**. In **Figure 1** we compare the relative error of the respective ansatz spaces. It can be seen that the block sparse ansatz space recovers almost as well as the sparse approach. As expected, the dense TT format is less favorable with respect to the sample size.

A clever change of basis, given by the diagonalization of Q, can reduce the required block size from 4 to 1. This allows to extend the presented approach to higher dimensional problems. The advantage over the classical Riccati approach becomes clear when considering non-linear versions of the control problem that do not exhibit a Riccati solution. This is done in [8,9] using the dense TT-format $T_r(V^d_p)$.

TABLE 1 | Degrees of freedom for the full space W^d_g of homogeneous polynomials of degree g =2, the TT variant $B_{\rho_{\max}}(W^d_g)$ with maximal block size $\rho_{\max}$ =4, the space $T_r(V^d_p)$ with TT rank bounded by r =6, and the full space V^d_p for completeness.

W^8_2	$B_4(W^8_2)$	$T_6(V^8_3)$	V^8_3
36	94	390	6561

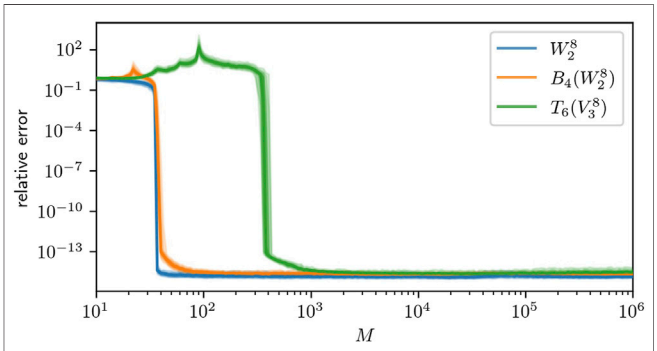


FIGURE 1 | 0.15–0.85 quantiles for the recovery error in W^8_2 (blue), $B_4(W^8_2)$ (orange), and $T_6(V^8_3)$ (green). The relative error is computed with respect to the L^2-norm using a Monte Carlo estimation with 10^6 samples.

5.2 Gaussian Density

As a second example we consider the reconstruction of an unnormalized Gaussian density

$$f(x) = \exp\left(-\|x\|_2^2\right).$$

TABLE 2 | Degrees of freedom for the full space S_g^d, the TT variant $S_{g,\rho_{\max}}^d$ with maximal block size $\rho_{\max}=1$, the space $T_r(V_p^d)$ with TT rank bounded by $r=1$, the space $T_r(V_p^d)$ with TT rank bounded by $r=8$, and the full space V_p^d for completeness.

S_7^6	$S_{7,1}^6$	$T_1(V_8^6)$	$T_8(V_8^6)$	V_8^6
1716	552	48	2,176	262,144

again on the domain $\Omega=[-1,1]^d$ with $d=6$. For the dictionary $\Psi=\Psi_{\text{Legendre}}$ [cf. **Eq. 5**] we chose $g=7$, $\rho_{\max}=1$ and $r=8$ and compare the reconstruction w.r.t. S_g^d, $S_{g,\rho_{\max}}^d$ and $T_r(V_p^d)$, defined in **(11)**, **(24)** and **(8)**. The degrees of freedom resulting from these different discretizations are compared in **Table 2**. This example is interesting because here the roles of the spaces are reversed. The function has product structure

$$f(x)=\exp\left(-x_1^2\right)\cdots\exp\left(-x_d^2\right)$$

and can therefore be well approximated as a rank 1 tensor train with each component C_k just being a best approximation for $\exp(-x_k^2)$ in the used function dictionary. Therefore, we expect the higher degree polynomials to be important. A comparison of the relative errors to the exact solution are depicted in **Figure 2**. This example demonstrates the limitations of the ansatz space S_7^6 which is not able to exploit the low-rank structure of the function f. Using $S_{7,1}^6$ can partially remedy this problem as can be seen by the improved sample efficiency. But since $S_{7,1}^6\subseteq S_7^6$ the final approximation error of $S_{7,1}^6$ can not deceed that of S_7^6. One can see that the dense format $T_1(V_8^6)$ produces the best results but is quite unstable compared to the other ansatz classes. This instability is a result of the non-convexity of the set $T_r(V_p^d)$ and we observe that the chance of getting stuck in a local minimum increases when the rank r is reduced from 8 to 1. Finally, we want to address the peaks that are observable at $M\approx 500$ samples for $T_8(V_8^6)$ and $M\approx 1716$ samples for S_7^6. For this recall that the approximation in S_7^6 amounts to solving a linear system which is underdetermined for $M<1716$ samples and overdetermined for $M>1716$ samples. In the underdetermined case we compute the minimum norm solution and in the overdetermined case we compute the least-squares solution. It is well-known that the solution to such a reconstruction problem is particularly unstable in the area of this transition [39]. Although the set $S_{7,1}^6$ is non-linear we take the peak at $M\approx 500$ as evidence for a similar effect which is produced by the similar linear systems that are solved in the micro steps in the ALS.

5.3 Quantities of Interest

The next considered problem often arises when computing quantities of interest from random partial differential equations. We consider the stationary diffusion equation

$$\begin{aligned}\nabla_x a(x,y)\nabla_x u(x,y) &= f(x) \quad x\in D\\ u(x,y)&=0 \quad x\in\partial D\end{aligned}$$

on $D=[-1,1]^2$. This equation is parametric in $y\in[-1,1]^d$. The randomness is introduced by the uniformly distributed random variable $y\sim\mathcal{U}([-1,1]^d)$ that enters the diffusion coefficient

$$a(x,y):=1+\frac{6}{\pi^2}\sum_{k=1}^{d}k^{-2}\sin\left(\hat{\omega}_k x_1\right)\sin\left(\check{\omega}_k x_2\right)y_k$$

with $\hat{\omega}_k=\pi\left\lfloor\frac{k}{2}\right\rfloor$ and $\check{\omega}_k=\pi\left\lceil\frac{k}{2}\right\rceil$. The solution u often measures the concentration of some substance in the domain Ω and one is interested in the total amount of this substance in the entire domain

$$M(y):=\int_\Omega u(x,y)\,\mathrm{d}x.$$

An important result proven in [31] ensures the ℓ^p summability, for some $0<p\le 1$, of the polynomial coefficients of the solution of this equation in the dictionary of Chebyshev polynomials. This means that the function is very regular and we presume that it can be well approximated in S_g^d for the dictionary of Legendre polynomials Ψ_{Legendre}. For our numerical experiments we chose $d=10$, $g=5$ and $\rho_{\max}=3$ and again compare the reconstruction w.r.t. S_g^d, the block-sparse TT representations of $S_{g,\rho_{\max}}^d$ and $S_{g,\rho_{\max}}^{d,\text{aug}}$ and a dense

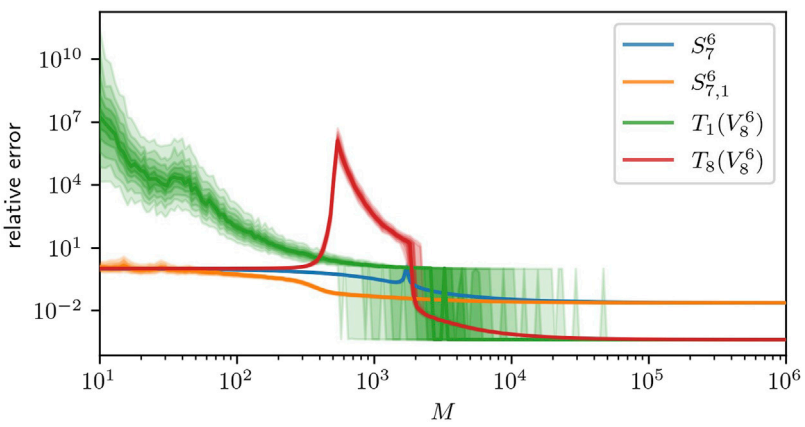


FIGURE 2 | 0.15–0.85 quantiles for the recovery error in S_7^6 (blue), $S_{7,1}^6$ (orange), $T_1(V_8^6)$ (green), and $T_8(V_8^6)$ (red). The relative error is computed with respect to the L^2-norm using a Monte Carlo estimation with 10^6 samples.

TABLE 3 | Degrees of freedom for the full space S_g^d, the TT variant $S_{g,\rho_{\max}}^d$ with maximal block size $\rho_{\max}$ =3, the space $T_r(V_p^d)$ with TT rank bounded by r =14, and the full space V_p^d for completeness.

S_5^{10}	$S_{5,3}^{10}$	$S_{5,3}^{10,\mathrm{aug}}$	$T_{14}(V_6^{10})$	V_6^{10}
3,003	1726	803	7,896	60,466,176

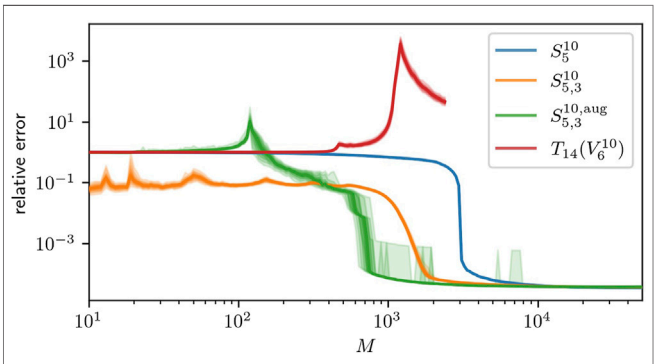


FIGURE 3 | 0.15–0.85 quantiles for the recovery error in S_5^{10} (blue), $S_{5,3}^{10}$ (orange), $S_{5,3}^{10,\mathrm{aug}}$ (green), and $T_{14}(V_6^{10})$ (red). The relative error is computed with respect to the L^2-norm using a Monte Carlo estimation with 10^6 samples. The experiment for $T_{14}(V_6^{10})$ was stopped early at M =1,200 due to its prohibitive computational demand and because the expected behaviour is already observable.

TT representation of $T_r(V_p^d)$ with rank $r \leq 14$. Admittedly, the choice $d = 10$ is relatively small for this problem but was necessary since the computation on S_g^d took prohibitively long for larger values. A comparison of the degrees of freedom for the different ansatz spaces is given in **Table 3** the relative errors to the exact solution are depicted in **Figure 3**. In this plot we can recognize the general pattern that a lower number of parameters can be associated with an improved sample efficiency. However, we also observe that for small M the relative error for $S_{g,\rho}^d$ is smaller than for $S_{g,\rho}^{d,\mathrm{aug}}$. We interpret this as a consequence of the regularity of u since the alternating scheme for the optimization in $S_{g,\rho}^d$ favors lower degree polynomials by construction. In spite of this success, we have to point out that optimizing over $S_{g,\rho}^d$ took about 10 times longer than optimizing over $S_{g,\rho}^{d,\mathrm{aug}}$. Finally, we observe that the recovery in $T_{14}(V_6^{10})$ produces unexpectedly large relative errors when compared to previous results in [13]. This suggests that the rank-adaptive algorithm from [13] has a strong regularizing effect that improves the sample efficiency.

6 CONCLUSION

We introduce block sparsity [28,29] as an efficient tool to parametrize, multivariate polynomials of bounded degree. We discuss how to extend this to general multivariate polynomials of bounded degree and prove bounds for the block sizes for certain polynomials. As an application we discuss the problem of function identification from data for tensor train based ansatz spaces and give some insights into when these ansatz spaces can be used efficiently. For this we motivate the usage of low degree multivariate polynomials by approximation results (e.g. [30,31]) and recent results on sample complexity [32]. This leads to a novel algorithm for the problem at hand. We then demonstrate the applicability of this algorithm to different problems. Up until now block sparse tensor trains are not used for these recovery tasks. The numerical examples, however, demonstrate that at least dense tensor trains can not compete with our novel block-sparse approach. We observe that the sample complexity can be much more favorable for successful system identification with block sparse tensor trains than with dense tensor trains or purely sparse representations. We expect that the inclusion of rank-adaptivity using techniques from SALSA or bASD is straight forward, which we therefore consider an interesting direction from an applied point of view for forthcoming papers. We expect, that this would improve the numerical results even further. The introduction of rank-adaptivity would moreover alleviate the problem of having to choose a block size a-priori. Finally, we want to reiterate that the spaces of polynomials with bounded degree are predestined for the application of least-squares recovery with an optimal sampling density (cf [39]) which holds opportunities for further improvement of the sample efficiency. This leads us to the strong believe that the proposed algorithm can be applied successfully to other high dimensional problems in which the sought function exhibits sufficient regularity.

DATA AVAILABILITY STATEMENT

The datasets presented in this study can be found in online repositories. The names of the repository/repositories and accession number(s) can be found below: https://github.com/ptrunschke/block_sparse_tt.

AUTHOR CONTRIBUTIONS

MG is the main contributor of the block-sparse tensor train section. RS is the main contributor of the method section. PT is the main contributor of the sample complexity section, the implementation of the experiments and contributed to the proof of the main Theorem. All authors contributed to every section and contributed equally to the introduction and to the discussion.

FUNDING

MG was funded by DFG (SCHN530/15-1). RS was supported by the Einstein Foundation Berlin. PT acknowledges support by the Berlin International Graduate School in Model and Simulation based Research (BIMoS).

REFERENCES

1. Brunton SL, Proctor JL, and Kutz JN. Discovering Governing Equations from Data by Sparse Identification of Nonlinear Dynamical Systems. *Proc Natl Acad Sci USA* (2016) 113(15):3932–7. doi:10.1073/pnas.1517384113
2. Gelß P, Klus S, Eisert J, and Schütte C. Multidimensional Approximation of Nonlinear Dynamical Systems. *J Comput Nonlinear Dyn* (2019) 14(6). doi:10.1115/1.4043148
3. Goeßmann A, Götte M, Roth I, Ryan S, Kutyniok G, and Eisert J. *Tensor Network Approaches for Data-Driven Identification of Non-linear Dynamical Laws* (2020). NeurIPS2020 - Tensorworkshop.
4. Kazeev V, and Khoromskij BN. Low-Rank Explicit QTT Representation of the Laplace Operator and its Inverse. *SIAM J Matrix Anal Appl* (2012) 33(3): 742–58. doi:10.1137/100820479
5. Kazeev V, and Schwab C. Quantized Tensor-Structured Finite Elements for Second-Order Elliptic PDEs in Two Dimensions. *Numer Math* (2018) 138(1): 133–90. doi:10.1007/s00211-017-0899-1
6. Bachmayr M, and Kazeev V. Stability of Low-Rank Tensor Representations and Structured Multilevel Preconditioning for Elliptic PDEs. *Found Comput Math* (2020) 20(5):1175–236. doi:10.1007/s10208-020-09446-z
7. Eigel M, Pfeffer M, and Schneider R. Adaptive Stochastic Galerkin FEM with Hierarchical Tensor Representations. *Numer Math* (2016) 136(3):765–803. doi:10.1007/s00211-016-0850-x
8. Dolgov S, Kalise D, and Kunisch K. *Tensor Decomposition Methods for High-Dimensional Hamilton-Jacobi-Bellman Equations*. arXiv (2021). 1908.01533 [cs, math].
9. Oster M, Sallandt L, and Schneider R. *Approximating the Stationary Hamilton-Jacobi-Bellman Equation by Hierarchical Tensor Products*. arXiv (2021). arXiv: 1911.00279 [math].
10. Richter L, Sallandt L, and Nüsken N. *Solving High-Dimensional Parabolic PDEs Using the Tensor Train Format*. arXiv (2021). arXiv:2102.11830 [cs, math, stat].
11. Christian B, Martin E, Leon S, and Philipp T. *Pricing High-Dimensional Bermudan Options with Hierarchical Tensor Formats*. arXiv (2021). arXiv: 2103.01934 [cs, math, q-fin].
12. Glau K, Kressner D, and Statti F. Low-Rank Tensor Approximation for Chebyshev Interpolation in Parametric Option Pricing. *SIAM J Finan Math* (2020) 11(3):897–927. Publisher: Society for Industrial and Applied Mathematics doi:10.1137/19m1244172
13. Eigel M, Neumann J, Schneider R, and Wolf S. Non-intrusive Tensor Reconstruction for High-Dimensional Random PDEs. *Comput Methods Appl Math* (2019) 19(1):39–53. doi:10.1515/cmam-2018-0028
14. Eigel M, Schneider R, Trunschke P, and Wolf S. Variational Monte Carlo–Bridging Concepts of Machine Learning and High-Dimensional Partial Differential Equations. *Adv Comput Math* (2019) 45(5):2503–32. doi:10.1007/s10444-019-09723-8
15. Zhang Z, Yang X, Oseledets IV, Karniadakis GE, and Daniel L. Enabling High-Dimensional Hierarchical Uncertainty Quantification by Anova and Tensor-Train Decomposition. *IEEE Trans Comput.-Aided Des Integr Circuits Syst* (2015) 34(1):63–76. doi:10.1109/tcad.2014.2369505
16. Klus S, and Gelß P. Tensor-Based Algorithms for Image Classification. *Algorithms* (2019) 12(11):240. doi:10.3390/a12110240
17. Stoudenmire E, and Schwab DJ "Advances in Neural Information Processing Systems," in *Supervised Learning with Tensor Networks*. Editors D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett. Curran Associates, Inc. (2016) 29. Available at: https://proceedings.neurips.cc/paper/2016/file/5314b9674c86e3f9d1ba25ef9bb32895-Paper.pdf/
18. Oseledets IV. DMRG Approach to Fast Linear Algebra in the TT-Format. *Comput Methods Appl Math* (2011) 11(3):382–393. doi:10.2478/cmam-2011-0021
19. Holtz S, Rohwedder T, and Schneider R. The Alternating Linear Scheme for Tensor Optimization in the Tensor Train Format. *SIAM J Sci Comput* (2012) 34(2):A683–A713. doi:10.1137/100818893
20. White SR. Density Matrix Formulation for Quantum Renormalization Groups. *Phys Rev Lett* (1992) 69(19):2863–6. doi:10.1103/physrevlett.69.2863
21. Grasedyck L, and Krämer S. Stable ALS Approximation in the TT-Format for Rank-Adaptive Tensor Completion. *Numer Math* (2019) 143(4):855–904. doi:10.1007/s00211-019-01072-4
22. Holtz S, Rohwedder T, and Schneider R. On Manifolds of Tensors of Fixed TT-Rank. *Numer Math* (2012) 120(4):701–31. doi:10.1007/s00211-011-0419-7
23. Lubich C, Oseledets IV, and Vandereycken B. Time Integration of Tensor Trains. *SIAM J Numer Anal* (2015) 53(2):917–41. doi:10.1137/140976546
24. Chevreuil M, Lebrun R, Nouy A, and Rai P. A Least-Squares Method for Sparse Low Rank Approximation of Multivariate Functions. *Siam/asa J Uncertainty Quantification* (2015) 3(1):897–921. doi:10.1137/13091899x
25. Grelier E, Anthony N, and Chevreuil M. *Learning with Tree-Based Tensor Formats*. arXiv (2019). arXiv:1811.04455 [cs, math, stat].
26. Grelier E, Anthony N, and Lebrun R. *Learning High-Dimensional Probability Distributions Using Tree Tensor Networks*. arXiv (2021). arXiv:1912.07913 [cs, math, stat].
27. Haberstich C. *Adaptive Approximation of High-Dimensional Functions with Tree Tensor Networks for Uncertainty Quantification* (2020) Theses, École centrale de Nantes.
28. Singh S, Pfeifer RNC, and Vidal. G. Tensor Network Decompositions in the Presence of a Global Symmetry. *Phys Rev A* (2010) 82(5):050301. doi:10.1103/physreva.82.050301
29. Markus B, Michael G, and Max P. *Particle Number Conservation and Block Structures in Matrix Product States*. arXiv (2021). arXiv:2104.13483 [math.NA, quant-ph].
30. Breiten T, Kunisch K, and Pfeiffer L. Taylor Expansions of the Value Function Associated with a Bilinear Optimal Control Problem. *Ann de l'Institut Henri Poincaré C, Analyse non linéaire* (2019) 36(5):1361–99. doi:10.1016/j.anihpc.2019.01.001
31. Hansen M, and Schwab C. Analytic Regularity and Nonlinear Approximation of a Class of Parametric Semilinear Elliptic PDEs. *Mathematische Nachrichten* (2012) 286(8-9):832–60. doi:10.1002/mana.201100131
32. Eigel M, Schneider R, and Trunschke P. *Convergence Bounds for Empirical Nonlinear Least-Squares*. arXiv. arXiv:2001.00639 [cs, math], 2020.
33. Oliphant T. *Guide to NumPy* (2006).
34. Huber B, and Wolf S. *Xerus - A General Purpose* (2014). Tensor Library.
35. Espig M, Hackbusch W, Handschuh S, and Schneider R. Optimization Problems in Contracted Tensor Networks. *Comput Vis Sci.* (2011) 14(6): 271–85. doi:10.1007/s00791-012-0183-y
36. Oseledets IVV. Tensor-Train Decomposition. *SIAM J Sci Comput* (2011) 33(5):2295–317. doi:10.1137/090752286
37. Hackbusch W. *On the Representation of Symmetric and Antisymmetric Tensors. Preprint*. Leipzig, Germany: Max Planck Institute for Mathematics in the Sciences (2016).
38. Wolf S. *Low Rank Tensor Decompositions for High Dimensional Data Approximation, Recovery and Prediction*. [PhD thesis]. TU Berlin (2019).
39. Cohen A, and Migliorati G. Optimal Weighted Least-Squares Methods. *SMAI J Comput Math* (2017) 3:181–203. doi:10.5802/smai-jcm.24
40. Haberstich C, Anthony N, and Perrin G. *Boosted Optimal Weighted Least-Squares*. arXiv (2020). arXiv:1912.07075 [math.NA].
41. Göttlich S, and Schillinger T. *Microscopic and Macroscopic Traffic Flow Models Including Random Accidents* (2021).
42. Rasmussen CE, and Williams CKI. *Gaussian Processes for Machine Learning*. Cambridge: MIT Press (2006).
43. Cornford D, Nabney IT, and Williams CKI. Modelling Frontal Discontinuities in Wind fields. *J Nonparametric Stat* (2002) 14(1-2):43–58. doi:10.1080/10485250211392
44. Szalay S, Pfeffer M, Murg V, Barcza G, Verstraete F, Schneider R, et al. Tensor Product Methods and Entanglement Optimization Forab Initioquantum Chemistry. *Int J Quan Chem.* (2015) 115(19):1342–91. doi:10.1002/qua.24898
45. Michel B, and Nouy A. *Learning with Tree Tensor Networks: Complexity Estimates and Model Selection*. arXiv (2021). arXiv:2007.01165 [math.ST].
46. Ballani J, and Grasedyck L. Tree Adaptive Approximation in the Hierarchical Tensor Format. *SIAM J Sci Comput* (2014) 36(4):A1415–A1431. doi:10.1137/130926328
47. Curtain RF, and Hans Z. *An Introduction to Infinite-Dimensional Linear Systems Theory*. New York: Springer (1995).

48. Steinlechner M. Riemannian Optimization for High-Dimensional Tensor Completion. *SIAM J Sci Comput* (2016) 38(5):S461–S484. doi:10.1137/15m1010506

Conflict of Interest: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's Note: All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher.

frontiers in Applied Mathematics and Statistics

ORIGINAL RESEARCH
published: 25 February 2022
doi: 10.3389/fams.2022.801650

Dictionary-Based Low-Rank Approximations and the Mixed Sparse Coding Problem

*Jeremy E. Cohen**

Univ Lyon, INSA-Lyon, UCBL, UJM-Saint Etienne, CNRS, Inserm, CREATIS UMR 5220, Villeurbanne, France

Constrained tensor and matrix factorization models allow to extract interpretable patterns from multiway data. Therefore crafting efficient algorithms for constrained low-rank approximations is nowadays an important research topic. This work deals with columns of factor matrices of a low-rank approximation being sparse in a known and possibly overcomplete basis, a model coined as Dictionary-based Low-Rank Approximation (DLRA). While earlier contributions focused on finding factor columns inside a dictionary of candidate columns, i.e., one-sparse approximations, this work is the first to tackle DLRA with sparsity larger than one. I propose to focus on the sparse-coding subproblem coined Mixed Sparse-Coding (MSC) that emerges when solving DLRA with an alternating optimization strategy. Several algorithms based on sparse-coding heuristics (greedy methods, convex relaxations) are provided to solve MSC. The performance of these heuristics is evaluated on simulated data. Then, I show how to adapt an efficient MSC solver based on the LASSO to compute Dictionary-based Matrix Factorization and Canonical Polyadic Decomposition in the context of hyperspectral image processing and chemometrics. These experiments suggest that DLRA extends the modeling capabilities of low-rank approximations, helps reducing estimation variance and enhances the identifiability and interpretability of estimated factors.

Keywords: dictionary, tensors, sparse, low-rank, non-convex optimization, LASSO

OPEN ACCESS

Edited by:
André Uschmajew,
Max Planck Institute for Mathematics in the Sciences, Germany

Reviewed by:
Yunlong Feng,
University at Albany, United States
Jiajia Li,
College of William & Mary, United States

***Correspondence:**
Jeremy E. Cohen
jeremy.cohen@cnrs.fr

Specialty section:
This article was submitted to Mathematics of Computation and Data Science, a section of the journal Frontiers in Applied Mathematics and Statistics

Received: *25 October 2021*
Accepted: *19 January 2022*
Published: *25 February 2022*

Citation:
Cohen JE (2022) Dictionary-Based Low-Rank Approximations and the Mixed Sparse Coding Problem. Front. Appl. Math. Stat. 8:801650. doi: 10.3389/fams.2022.801650

1. INTRODUCTION

Low-Rank Approximations (LRA) are well-known dimensionality reduction techniques that allow to represent tensors or matrices as sums of a few separable terms. One of the main reasons why these methods are used extensively for pattern recognition is their ability to provide part-based representations of the input data. This is particularly true for Nonnegative Matrix Factorization (NMF) or Canonical Polyadic Decomposition (CPD), see section 2 for a quick introduction and the following surveys and book for more details [1–3]. In order to fix notations while retaining generality, let us make use of the following informal mathematical formulation of LRA.

DEFINITION 1 (Low-Rank Approximation). *Given a data matrix $Y \in \mathbb{R}^{n\times m}$ and a small nonzero rank $r \in \mathbb{N}$, compute so-called factor matrices $A \in \mathbb{R}^{n\times r}$ and $B \in \mathbb{R}^{m\times r}$ in*

$$\underset{A\in\Omega_A,\ B\in\Omega_B}{\arg,\min} \|Y - AB^T\|_F^2 . \tag{1}$$

The set Ω_A is an additional constraint set for A, such as nonnegativity elementwise. The set Ω_B is the structure required on B to obtain a specific LRA model; for instance in the unconstrained CPD of an order three tensor, Ω_B is the set of matrices written as the Khatri-Rao product of two factor matrices. A precise qualification of "small" rank depends on the sets Ω and is omitted for simplicity.

Among LRA models, identifiability[1] properties vary significantly. While CPD is usually considered to have mild identifiability conditions [4], NMF on the other hand is often not essentially unique [3][2]. Generally speaking, additional regularizations are often imposed in both matrix and tensor LRA models to help with the interpretability of the part-based representation. These regularizations may take the form of constraints on the parameters or penalizations, like the sparsity-inducing ℓ_1 norm [5, 6]. They can also take the form of a parameterization of the factors [7, 8] if such a parameterization is known.

This work focuses on imposing an implicit parameterization on A. More precisely, each column of factor A is assumed to be well represented in a known dictionary $D \in \mathbb{R}^{n\times d}$ using only a small number $k < n$ of coefficients. In other words, in this work the set Ω_A is the union of subspaces of dimension k spanned by columns of a given dictionary D, and there exist a columnwise k-sparse code matrix $X \in \mathbb{R}^{d\times r}$ such that $A = DX$. A low-rank approximation model which such a constraint on at least one mode is called Dictionary-based LRA (DLRA) thereafter.

DEFINITION 2 (Dictionary-based Low-Rank Approximation). *Given a data matrix $Y \in \mathbb{R}^{n\times m}$, a small nonzero rank $r \in \mathbb{N}$, a sparsity level $k < n$ and a dictionary $D \in \mathbb{R}^{n\times d}$, compute columnwise k-sparse code matrix $X \in \mathbb{R}^{n\times r}$ and factor matrix $B \in \mathbb{R}^{m\times r}$ in*

$$\underset{X\in\Omega_X,\ \forall i\leq r,\ \|X_i\|_0\leq k,\ B\in\Omega_B}{\arg,\min} \|Y - DXB^T\|_F^2 , \tag{2}$$

where X_i is the i-th column of X. The set Ω_X is an additional constraint set for X, such as nonnegativity elementwise. The set Ω_B is the structure required on B to obtain a specific LRA model. Additionally, assume that B is full column-rank.

Abbreviations: LRA, low-rank approximation; CPD, canonical polyadic decomposition; DLRA, dictionary-based LRA; DCPD, dictionary-based CPD; DNMF, dictionary-based Nonnegative Matrix Factorization; NMF, nonnegative matrix factorization; DMF, dictionary-based matrix factorization; MSC, mixed sparse coding; AO, alternating optimization; HT, hard thresholding; IHT, iterative hard thresholding; OMP, orthognal matching pursuit; FISTA, fast iterative soft thresholding algorithm; HOMP, hierarchical OMP; SNR, signal to noise ratio; SAM, spectral angular mapper.

[1]Informally, the parameters of a model are identifiable if they can be uniquely recovered from the data.

[2]Essential uniqueness means uniqueness up to trivial scaling ambiguities and rank-one terms permutation.

1.1. Motivations

To better advocate for the usefulness of DLRA, below two particular DLRA models are introduced:

- Dictionary-based Nonnegative Matrix Factorization (DNMF) may be formulated as

$$\underset{X\in\mathbb{R}_+^{d\times r},\ \forall i\leq r,\ \|X_i\|_0\leq k,\ B\in\mathbb{R}_+^{m\times r}}{\arg,\min} \|Y - DXB^T\|_F^2 . \tag{3}$$

  To ensure that $A = DX$ is nonnegative, dictionary D is supposed to be entry-wise nonnegative so that the constraint $X \geq 0$ is sufficient. NMF is known to have strict identifiability conditions [3, 9], and in general one should expect that NMF has infinitely many solutions. The dictionary constraint can enhance identifiability by restraining the set of solutions. For instance setting $D = Y$ and $k = 1$ yields the Separable NMF model, which is solvable in polynomial time and which factors are generically identifiable even in the presence of noise [10]. Moreover, DNMF is also a more flexible model than NMF. In section 5.2, it is shown that DNMF can be used to solve a matrix completion problem with missing rows in the data that NMF cannot solve.
- Dictionary-based Canonical Polyadic Decomposition (DCPD), for an input tensor $Y \in \mathbb{R}^{n\times m_1\times m_2}$ may be formulated as

$$\underset{X\in\mathbb{R}^{d\times r},\ \forall i\leq r,\ \|X_i\|_0\leq k,\ B\in\mathbb{R}^{m_1\times r},\ C\in\mathbb{R}^{m_2\times r}}{\arg,\min} \|Y - DX\,(B\odot C)^T\|_F^2 . \tag{4}$$

  where $\odot$ is the Khatri-Rao product [1]. Unlike NMF, CPD factor are identifiable under mild conditions [4]. But in practice, identifiability may still be elusive, and the approximation problem is in general ill-posed [11] and algorithmically challenging [12]. Therefore, the dictionary constraint can be used to reduce estimation variance given an adequate choice of dictionary D. It was shown when $k = 1$ in Equation (4) that it enhances identifiability of the CPD and makes the optimization problem well-posed [13].

Note that these models have known interesting properties when $k = 1$, but have not been studied in the general case. A motivation for this work is therefore to expand these previous works to the general case $n > k \geq 1$, focusing on the algorithmic aspects. Section 2 provides more details on the one-sparse case.

1.2. Contributions

The first contribution of this work is to propose the new DLRA framework. The proposed DLRA allows to constrain low-rank approximation problems so that some of the factor matrices are sparse columnwise in a known basis. This includes sparse coding the patterns estimated by constrained factorization models such as NMF, or tensor decomposition models such as CPD. The impact of the dictionary constraint on the LRA parameter estimation error is studied experimentally in section 5.2 dedicated to experiments with DLRA, where the flexibility of DLRA is furthermore illustrated on real applications. I show that DLRA allows to complete entirely missing rows in incomplete low-rank matrices. I also show the advantage of finding the

best atoms algorithmically when imposing smoothness in DCPD using B-splines.

A second contribution is to design an efficient algorithm to solve Problem (2). To that end, I shall focus on (approximate) Alternating Optimization (AO), understood as Block Coordinate Descent (BCD) where each block update consists in minimizing almost exactly the cost with respect to the updated block while other parameters are fixed. There are two reasons for this choice. First, AO algorithms are known to perform very well in many LRA problems. They are state-of-the-art for NMF [3] and standard for Dictionary Learning [14, 15] and tensor factorization problems [16, 17]. Nevertheless inexact BCD methods and all-at-once methods are also competitive [18–21] and an inertial Proximal Alternating Linearization Method (iPALM) for DLRA is quickly discussed in section 5.1.2. The proposed algorithm is coined AO-DLRA.

As a third contribution, I study the subproblem in Problem (2) with respect to X. Developing a subroutine to solve it for known B allows not only for designing AO-DLRA, but also for post-processing a readily available estimation of A. In fact a significant part of this manuscript is devoted to studying the subproblem of minimizing the cost in Problem (2) with respect to X, which is labeled Mixed Sparse Coding (MSC),

$$\underset{\forall i \leq r,\ \|X_i\|_0 \leq k}{\arg\min} \ \|Y - DXB^T\|_F^2 . \tag{MSC}$$

In this formulation of MSC, no further constraints are imposed on X and therefore Ω_X is $\mathbb{R}^{d\times r}$. While similar to a matrix sparse coding, which is obtained by setting B to the identity matrix, it will become clear in this manuscript that MSC should not be handled directly using sparse coding solvers in general. For instance, it is shown in section 3.1.4 that while having an orthogonal dictionary D yields a polynomial time algorithm to solve MSC when $r = 1$ using Hard Thresholding, this does not extend when $r > 1$. In this work, the MSC problem, which is NP-hard as a generalization of sparse coding, is studied in order to build several reasonable heuristics that may be plugged into an AO algorithm for DLRA.

1.3. Structure

The article is divided in three remaining sections. The first section provides the necessary background for this manuscript. The second one is devoted to studying MSC and heuristics to solve it. The third section shows how to compute DLRA using the heuristics developed in the first part. In section 3.1, we study formally the MSC problem and its relationship with sparse coding. In section 3.2, we study a simple heuristic based solely on sparse coding each column of $YB^\dagger$. Two ℓ_0 heuristics similar to Iterative Hard Thresholding (IHT) and Orthogonal Matching Pursuit (OMP) are also introduced, while two types of convex relaxations are studied in Section 3.3. Section 3.5 is devoted to compare the practical performance of the various algorithms proposed to solve MSC and shows that a columnwise ℓ_1 regularization coined Block LASSO is a reasonable heuristic for MSC. Section 5.1 shows how Block LASSO can be used to compute various DLRA models, while section 5.2 illustrates DLRA on synthetic and real-life source-separation problems.

All the proofs are deferred to the **Supplementary Material** attached to this article, as well as the pseudo-codes of some proposed heuristics and additional experiments.

1.4. Notations

Vectors are denoted by small letters x, matrices by capital letters X. The indexed quantity X_i refers to the i-th column of matrix X, while X_{ij} is the (i,j)-th entry in X. A subset S of columns of X is denoted X_S, while a submatrix with columns in S_i and rows in S_j is denoted $X_{S_iS_j}$. The submatrix of X obtained by removing the i-th column is denoted by X_{-i}. The i-th row of matrix X is denoted by $X_{.i}$ and $X_{.I}$ is the submatrix of X with rows in set I. The $\ell_0(x) = \|x\|_0$ map counts the number of nonzero elements in x. The product $\otimes$ denotes the Kronecker product, a particular instance of tensor product [22]. The Khatri Rao product $\odot$ is the columnwise Kronecker product. The support of a vector or matrix x, i.e., the location of its nonzero elements, is denoted by $\mathrm{Supp}(x)$. If $S = \mathrm{Supp}(x)$, the location of the zero elements is denoted $\bar{S}$. The list $[M,N,P]$ denotes the concatenation of columns of matrices M, N and P. A set I contains $|I|$ elements.

2. BACKGROUND

Let us review the foundations of the proposed work, matrix and tensor decompositions and sparse coding, as well as existing models closely related to the proposed DLRA.

2.1. Matrix and Tensor Decompositions

Matrix and tensor decompositions can be understood as pattern mining techniques, which extract meaningful information out of collections of input vectors in an unsupervised manner. Arguably one of the earliest form of interpretable matrix factorization is Principal Component Analysis [23, 24], which extract a few orthogonal significant patterns out of a given matrix while performing dimensionality reduction.

Other matrix factorization models exploit other constraints than orthogonality to mine interpretable patterns. Blind source separation models such as Independent Component Analysis historically exploited statistical independence [25], while NMF, which assumes all parameters are elementwise nonnegative, has received significant scrutiny over the last two decades following the seminal paper of Lee and Seung [26]. Sparse models such as Sparse Component Analysis exploit sparsity on the coefficients [27]. It can be noted that while all those factorization techniques aim at providing interpretable representations, they are typically identifiable under strict conditions not necessarily met in practice [9, 28, 29]. It should be noted that the most important underlying hypothesis in matrix factorization is the linear dependency of the input data with respect to the templates/principal components stored in matrix A following the notations of Equation (1).

In practice, to compute for instance NMF, one solves an optimization problem of the form

$$\underset{A\geq 0,\ B\geq 0}{\arg\min} \ \|Y - AB^T\|_F^2 \tag{5}$$

which is non-convex with respect to A, B jointly but convex for each block. A common family of methods therefore uses alternating optimization in the spirit of Alternating Least Squares [30]. Other loss functions can easily be used [26]. However, computing matrix factorization models is often a difficult task; in fact NMF and sparse component analysis are both NP-hard problems in general and existing polynomial time approximation algorithms should be considered heuristics [31, 32].

Tensor decompositions follow the same ideas of unsupervised pattern mining and linearity with respect to the representation basis, but extract information out of tensors rather than matrices. Tensors in this manuscript are considered simply as multiway arrays [1] as is usually done in data sciences. Tensors have become an important data structure as they appear naturally in a variety of applications such as chemometrics [33], neurosciences [34], remote sensing [35], or deep learning [36].

At least two families of tensor decomposition models can be considered, with quite different identifiability properties and applications. A first family contains intepretable models such as the Canonical Polyadic Decomposition, often called PARAFAC [37], or closely related models such as PARAFAC2 [38]. Contrarily to constrained matrix decomposition models, the addition of at least a dimension compared to matrix factorization fixes the rotation ambiguity inherent to matrix factorization models, and therefore the CPD model is identifiable under mild conditions [4, 39, 40]. Nevertheless, additional constraints are commonly imposed on the parameters of these models to refine the interpretability of the parameters, reduce estimation errors or improve the properties of the underlying optimization problem [41, 42]. A second family is composed of tensor formats, in particular the Tucker decomposition [43] and a wide class of tensor networks such as tensor trains [44]. These models are not used in general for solving inverse problems but rather for compression or dimensionality reduction. Nevertheless, they turn into interesting pattern mining tools given adequate constraints such as nonnegativity [45, 46].

2.2. Sparse Coding

Spare Coding (SC) and other sparse approximation problems typically try to describe an input vector as a sparse linear combination of well-chosen basis vectors. A typical formulation for sparse coding an input vector $y \in \mathbb{R}^n$ in a code book or dictionary $D \in \mathbb{R}^{n\times d}$ is the following non-convex optimization problem

$$\underset{\|x\|_0\leq k}{\arg,\min} \|y - Dx\|_2^2, \tag{SC}$$

where $k \leq n$ is the largest number of nonzero entries allowed in vector x, i.e., the size of the support of x. As long as the dictionary D is not orthogonal, this problem is difficult to solve efficiently and is in fact NP-hard [31]. The body of literature of algorithms proposed to solve SC is very large, see for instance [47] for a comprehensive overview. Overall, there exist at least three kind of heuristics to provide candidate solutions to SC:

- Greedy methods, such as OMP [48, 49] which is described in more details in section 3.2.3. These methods select indices where x is nonzero greedily until the target sparsity level or a tolerance on the reconstruction error is reached. They benefit from optimality guarantees when the dictionary columns, called atoms, are "far" from each others. In that case the dictionary is said to be incoherent [50].
- Nonconvex first order methods, such as IHT [51], that are based on proximal gradient descent [52] knowing that the projection on the set of k-sparse vectors is obtained by simply clipping the $n - k$ smallest absolute values in x. Convergence guarantees for first order constrained optimization methods is a rapidly evolving topic out of the scope of this communication, but a discussion on optimality guarantees of IHT is available in section 3.2.2.
- Convex relaxation methods, such as Least Absolute Shrinkage and Selection Operation (LASSO). The non-convex ℓ_0 constraint in Equation (SC) is replaced by a surrogate convex constraint, typically using the ℓ_1 norm. This makes the problem convex and easier to solve, at the cost of potentially changing the support of the solution [53]. Solving the LASSO can be tackled with the Fast Iterative Soft Thresholding Algorithm (FISTA) [54, 55] which is essentially an accelerated proximal gradient method with convergence guarantees.

2.3. Models Closely Related to DLRA

While this work is interested in constraining the factor matrix A in Equation (1) so that its columns are sparse in a known dictionary (in other words, sparse coding the columns of A), a few previous works have been concerned with encoding each such column with only one atom. Most related to this work is the so-called Dictionary-based CPD [13], which I shall rename as one-sparse dictionary-based LRA. Within this framework, a model such as NMF becomes

$$\underset{\mathcal{K}\in\mathcal{P}_r([1,d]),\, B\geq 0}{\arg,\min} \|Y - D_{\mathcal{K}}B^T\|_F^2 \tag{6}$$

where $\mathcal{P}_r([1,d])$ is the set of all parts of $[1,d]$ with r elements, therefore $\mathcal{K}$ is a set of r indices from 1 to d. This problem is a particular case of the proposed DLRA framework because $D_{\mathcal{K}}$ can be written as DX where the columns of X are one-sparse. It was shown that one-sparse DLRA makes low-rank matrix factorization identifiable under mild coherence conditions on the dictionary, and makes the computation of CPD a well-posed problem. Intuitively, the dictionary constraint may be used to enforce a set of known templates to be used as patterns in the pattern mining procedure, and therefore one-sparse DLRA may be seen as a glorified pattern matching technique.

Other works in the sparse coding literature are related to DLRA, in particular the so-called multiple measurement vectors or collaborative sparse coding [56–58], which extends sparse coding when several inputs collected in a matrix Y are coded with the same dictionary using the same support. Effectively this means solving a problem of the form

$$\underset{\|\sum_{i=1}^r Z_i^2\|_0\leq kr}{\arg,\min} \|Y - DZ^T\|_F^2 \tag{7}$$

where the square is meant elementwise. DLRA can also be seen as a collaborative sparse coding by noticing that $Z := XB^T$ is at least kr row-sparse in Equation (2). However the low-rank hypothesis is lost, as well as the affectation of at most k atoms per column of matrix A.

3. MIXED SPARSE CODING HEURISTICS

3.1. Properties of Mixed Sparse Coding

In order to design efficient heuristics to solve MSC, let us first cover a few properties of MSC solutions, relate MSC with other sparse coding problems, and check whether special simpler cases exists such as when support of the solution is known or when the dictionary is orthogonal. The following section covers this material.

3.1.1. Equivalent Formulations and Relation to Sparse Coding

Let us start our study by linking MSC to other sparse coding problems. For reference sake, in this work the standard matrix sparse coding problem is formulated as

$$\underset{\forall i \leq r,\ \|X_i\|_0 \leq k}{\arg,\min} \|Y - DX\|_F^2 \,. \tag{8}$$

which is equivalent to solving r vector sparse coding problems as defined in Equation (SC).

Because the Frobenius norm is invariant to a reordering of the entries, it can be seen easily that Problem (MSC) is equivalent after vectorization[3] to a structured vector sparse coding Problem (SC) with block-sparsity constraints:

$$\underset{\forall i \leq r,\ \|\mathrm{vec}(X_i)\|_0 \leq k}{\arg,\min} \|\mathrm{vec}(Y) - (D \otimes B)\left[\mathrm{vec}(X_1), \ldots, \mathrm{vec}(X_r)\right]\|_2^2 \,. \tag{9}$$

It appears that MSC is therefore a structured sparse coding problem since the dictionary is a Kronecker product of two matrices. Moreover, the sparsity constraint applies on blocks of coordinates in the vectorized input vec(X), as studied in [60]. To the best of the author's knowledge, block sparse coding with Kronecker structured dictionary have not been specifically studied. In particular, the conjunction of Kronecker structured dictionary [61] and structured sparsity leads to specific heuristics detailed in the rest of this work. Nevertheless, in section 3.2, it is shown that MSC reduces to columnwise sparse coding under a small noise regime.

On a different note, consider the following problem

$$\min_{\|Y - DXB^T\|_F^2 \leq \epsilon_1} \max_i \|X_i\|_0 \,. \tag{10}$$

for some positive constant ϵ_1. Just like how Quadratically Constrained Basis Pursuit and LASSO are equivalent [47], one can show using similar arguments that Problems (MSC) and (10) have the same solution given a mild uniqueness assumption.

[3]Vectorization in this manuscript is row-first [59].

PROPOSITION 1. *Suppose that Problem (10) has a unique solution X^* with $\epsilon_1 \geq 0$. Then there exist a particular instance of Problem (MSC) such that X^* is the unique solution of (MSC). Conversely, a unique solution to Problem (MSC) is the unique solution to a particular instance of Problem (10).*

Formulation (10) of MSC could also be expressed with matrix induced norms. Indeed, defining $\ell_{0,0}(X) := \sup_{\|z\|_0=1} \|Xz\|_0 = \max_i \|X_i\|_0$ as an extension of matrix induced norms[4] $\ell_{p,q}(X) := \sup_{\|z\|_p=1} \|Xz\|_q$ [62], Problem (10) is equivalent to

$$\min_{\|Y - DXB^T\|_F^2 \leq \epsilon_1} \ell_{0,0}(X) \,, \tag{11}$$

a clear structured extension of quadratically constrained sparse coding [47]. Some authors preferably work with Problem (10) rather than (MSC) because in specific applications, choosing an error tolerance ϵ_1 is more natural than choosing the sparsity level k. While heuristics introduced further are geared toward solving Problem (MSC), they can be adapted to solve Problem (10).

The quadratically constrained formulation also sheds light on the fact that, if there exist some solution X to MSC such that the residual $\|Y - DXB^T\|_F^2$ is zero, then MSC is equivalent to matrix sparse coding Problem (8). Indeed, since it is assumed that B is full column rank, any such solution yields $YB(B^TB)^{-1} = DX$. Consequently, noiseless formulations of MSC will not be considered any further.

3.1.2. Generic Uniqueness of Solutions to MSC

In sparse coding, sparsity is introduced as a regularizer to enforce uniqueness of the regression solution. It is therefore natural to wonder if this property also holds for MSC. It is not difficult to observe that indeed generically the solution to MSC will be unique, under the usual spark condition on the dictionary D [63].

PROPOSITION 2. *Define spark(D) as the smallest number of columns of D that are linearly dependent. Suppose that spark(D) $> 2k$ and suppose B is a full column rank matrix. Then the set of Y such that Problem (MSC) has strictly more than one solution has zero Lebesgue measure.*

This result basically states that in practice, most MSC instances will have unique solutions as long as the dictionary is not too coherent.

3.1.3. Solving MSC Exactly When the Support Is Known

Solving the NP-hard MSC problem exactly is difficult because the naive, brute force algorithm implies testing all combinations of supports for all columns, which means computing $\binom{k}{d}^r$ least squares and is significantly slower than brute force for Problem (8). However, in the spirit of sparse coding which boils down to finding the optimal support for a sparse solution, MSC also reduces to a least squares problem when the locations of the zeros in a solution X are fixed and known. Below is detailed

[4]Mind the possible confusion with the $\mathcal{L}_{p,q}$ convention $\ell_{0,\infty}$ commonly encountered.

how to process this least squares problem to avoid forming the Kronecker matrix $D \otimes B$ explicitly.

From the observation in Equation (9) that the vectorized problem is really a structured sparse coding problem, one can check that for a given support $S = \text{Supp}(\text{vec}(X))$, solving MSC amounts to solving a linear system

$$\underset{z \in \mathbb{R}^{kr}}{\arg,\min} \|y - (D \otimes B)_S z\|_2^2 . \tag{12}$$

where $y := \text{vec}(Y)$.

It is not actually required to form the full Kronecker product $D \otimes B$ and then select a subset S of its columns, nor is it necessary to vectorize Y. More precisely, one needs to compute two quantities: $\left((D \otimes B)_S\right)^T (D \otimes B)_S$ and $\left((D \otimes B)_S\right)^T y$ to feed a linear system solver, and these products can be computed efficiently. Denote S_i the support of column X_i. Formally, one may notice that $(D \otimes B)_S$ is a block matrix $[D_{S_1} \otimes b_1, \ldots, D_{S_r} \otimes b_r]$. Therefore, using the identity $(D \otimes b)^T (D \otimes b) = b^T b D^T D$, one may compute the cross product by first precomputing $U = D^T D$[5] and $V = B^T B$, then computing each block $(D_{S_i} \otimes b_i)^T (D_{S_j} \otimes b_j)$ as $v_{ij} U_{S_i S_j}$.

For the data and mixing matrix product, notice that $(D_{S_i} \otimes b_i)^T y = D_{S_i} Y b_i^T$. Therefore, the model-data product can be obtained by first precomputing $N = YB^T$, then computing each block $\left((D \otimes B)_{S_i}\right)^T y$ as $D_{S_i} N_i$.

Remark on regularization: It may happen that for a fixed support, the linear system (12) is ill-posed. This may be caused by a highly coherent dictionary D or the choice of a large rank r. To avoid this issue, when the system is ill-conditioned in later experiments, a small ridge penalization may be added.

3.1.4. MSC With Orthogonal Dictionary Is Easy for Rank One LRA

An important question is whether the problem generally becomes easier if the dictionary is left orthogonal. This is the case for Problem (8), where orthogonal D allows to solve the problem using only Hard Thresholding (HT) on $D^T Y$ columnwise. Below, it is shown that a similar HT procedure can be used when $r = 1$, but not in general when $r > 1$.

Supposing D is left orthogonal, the MSC problem becomes

$$\underset{\|X_i\|_0 \leq k \ \forall i \in [1,r]}{\arg,\min} \|D^T Y - XB^T\|_F^2 . \tag{13}$$

When $r = 1$, matrix B^T is simply a row vector, which is a right orthogonal matrix after ℓ_2 normalization such that solving Problem (13) amounts to minimizing $\| \frac{1}{\|b\|_2^2} D^T Y b - x\|_2^2$. Then the solution is obtained using the thresholding operator $\text{HT}_k(\frac{1}{\|b\|_2^2} D^T Y b)$ which selects the k largest entries of its input.

Sadly when $r > 1$, matrix B^T is not right orthogonal in general and neither is $D \otimes B$. Therefore this thresholding strategy does not yield a MSC solution in general despite D being left orthogonal.

[5] If this is not possible because D is too large, one may instead compute the blocks $U_{S_i S_j}$ without pre-computing $D^T D$.

3.2. Non-convex Heuristics to Solve Mixed Sparse Coding

We now focus on heuristics to find candidate solutions to MSC. Similarly to sparse coding, MSC is an NP-hard problem for which obtaining the global solution typically requires costly algorithms [64, 65]. Therefore, in the following section, several heuristics are proposed that aim at finding good sparse approximations in reasonable time.

- A classical sparse coding algorithm [here OMP [49]] applied columnwise on $YB(B^T B)^{-1}$. Indeed in a small noise regime, columnwise sparse coding on $YB(B^T B)^{-1}$ is proven to be equivalent to solving MSC.
- A Block-coordinate descent algorithm that features OMP as a subroutine.
- A proximal gradient algorithm similar to Iterative Hard Thresholding.
- Two convex relaxations analog to LASSO [66], which are solved by accelerated proximal gradient in the spirit of FISTA [55]. Maximum regularization levels are computed, and a few properties concerning the sparsity and the uniqueness of the solutions are provided.

The performance of all these heuristics as MSC solvers is studied in section 3.5.

3.2.1. A Provable Reduction to Columnwise Sparse Coding for Small Noise Regimes

At first glance, one may think that projecting Y onto the row-space of B^T maps solutions of Problem (MSC) to solutions of Problem (8). Indeed, writing $Y = Y_B + Y_{-B}$ with $Y_B := YB(B^T B)^{-1} B^T$ the orthogonal projection of Y on the row-space of B^T, it holds that Problem (MSC) has the same minimizers than

$$\min_{\|X_i\|_0 \leq k \ \forall i \in [1,r]} \|Y_B - DXB^T\|_F^2 . \tag{14}$$

This problem is however not in general equivalent to matrix sparse coding because B^T distorts the error distribution.

Even though they are not equivalent, one could hope to solve MSC, in particular settings, by using a candidate solution to the matrix sparse coding Problem (8) with $YB(B^T B)^{-1}$ as input. The particular case of $k = 1$ is striking: matrix sparse coding is solved in closed form whereas MSC has no general solution as far as I know. This kind of heuristic replacement of Problem (MSC) by Problem (8) has been used heuristically in [13] when $k = 1$, and can be interpreted as "First find Z that minimizes $\|Y - ZB^T\|_F^2$, then find a k-sparse matrix X that minimizes $\|Z - DX\|_F^2$".

We show below that in a small perturbation regime, for a dictionary D and a mixing matrix B satisfying classical assumptions in compressive sensing, the matrix sparse coding solution has the same support than the MSC solution.

Lemma 1. *Let X, X' columnwise k-sparse matrices and $\epsilon > 0$, $\delta > 0$ such that $\|Y - DXB\|_F^2 \leq \epsilon$ and $\|YB(B^T B)^{-1} - DX'\|_F^2 \leq \delta$. Further suppose that spark$(D) > 2k$ and that B has*

full column-rank. Then

$$\|X - X'\|_F \leq \frac{1}{\sigma_{\min}^{(2k)}(D)} \sqrt{\delta + \frac{\epsilon}{\sigma_{\min}^2(B)}} \tag{15}$$

where $\sigma_{\min}(B)$ *is the smallest nonzero singular value of* B *and* $\sigma_{\min}^{(2k)}$ *is the smallest nonzero singular value of all submatrices of* D *constructed with* $2k$ *columns.*

Remarkably, if $k = 1$, then Problem (8) has a closed-form solution, and one may replace δ by the best residual to remove this unknown quantity in Lemma 1. On a different note, variants of this result can be derived under similar assumptions, for instance if D satisfies a $2k$ restricted isometry property.

We can now derive a support recovery equivalence between MSC and SC.

PROPOSITION 3. *Under the hypotheses of Lemma 1, supposing that* X *and* X' *are exactly columnwise* k*-sparse, if*

$$\min_{j \leq r} \sqrt{\min_{i \in Supp(X_j)} {X_{ij}}^2 + \min_{i' \in Supp(X'_j)} {X'_{i'j}}^2} > \frac{1}{\sigma_{\min}^{(2k)}(D)} \sqrt{\delta + \frac{\epsilon}{\sigma_{\min}^2(B)}}, \tag{16}$$

then $Supp(X) = Supp(X')$.

In practice, this bound can be used to grossly check whether support estimation in MSC may reduce to support estimation in matrix sparse coding or not. To do so, one may first tentatively solve Problem (8) with $YB(B^TB)^{-1}$ as input and obtain a candidate value X' as well as some residual δ. Furthermore, while this is costly, the values of $\sigma_{\min}(B)$ and $\sigma_{\min}^{(2k)}(D)$ may be computed. Then removing the term X_{ij} from Equation (16) yields a bound of the noise level ϵ:

$$\sigma_{\min}(B) \left(-\delta + \left(\sigma_{\min}^{(2k)}(D)\right)^2 \min_{j,i \in \text{Supp}(X')} {X'_{ij}}^2 \right) > \epsilon \tag{17}$$

under which the reduction is well-grounded.

These observations lead to the design of a first heuristic, which applies OMP to each column of $YB(B^TB)^{-1}$. The obtained support is then used to compute a solution X to MSC as described in section 3.1.3. This heuristic will be denoted TrickOMP in the following.

3.2.2. A First Order Strategy: Iterative Hard Thresholding

Maybe the most simple way to solve MSC is by proximal gradient descent. In sparse coding, this type of algorithm is often referred to as Iterative Hard Thresholding (IHT), and this is how I will denote this algorithm as well for MSC.

Computing the gradient of the differentiable convex term $f(X) = \|Y - DXB\|_F^2$ with respect to X is easy and yields

$$\frac{\partial f}{2\partial X} = -D^TYB + D^TDXBB^T\,. \tag{18}$$

Note that depending on the structure Ω_B, products D^TYB and B^TB may computed efficiently.

Then it is required to compute a projection on the set of columnwise k-sparse matrices. This can easily be done columnwise by hard thresholding. However, for the algorithm to be well-defined and deterministic, we need to suppose that projecting on the set of columnwise k-sparse matrices is a closed-form single-valued operation. This holds if, when several solution exist, i.e., when several entries are the k-th largest, one picks the right number of these entries in for instance the lexicographic order. The proposed IHT algorithm leverages the classic IHT algorithm and is summarized in the **Supplementary Material**. However, contrarily to the usual IHT, the proposed implementation makes use of the inertial acceleration proposed in FISTA [55]. Compared to using IHT for solving sparse coding, as hinted in Equation (9), here IHT is moreover applied to a structured sparse coding problem. This does not significantly modify the practical implementation of the algorithm.

IHT benefits from both convergence results as a (accelerated) proximal gradient algorithm with semi-algebraic regularization [51, 67] and support recovery guaranties for sparse coding [47, 68]. While the convergence results directly apply to MSC, extending support recovery guaranties to MSC is an interesting research avenue.

3.2.3. Hierarchical OMP

Before describing the proposed hierarchical algorithm, it might be helpful to understand how greedy heuristics for sparse coding, such as the OMP algorithm, are derived. Greedy heuristics select the best atom to reconstruct the data, then remove its contribution and repeats this process with the residuals. The key to understanding these techniques is that selecting only one atom is a problem with a closed-form solution. Indeed, for any $y \in \mathbb{R}^n$,

$$\underset{\|x\|_0 \leq 1}{\arg,\min} \|y - Dx\|_2^2 = \underset{z \in \mathbb{R}}{\arg,\min} \underset{j \in [1,d]}{\arg,\min} \|y - zD_j\|_2^2 \tag{19}$$

and after some algebra, for a dictionary normalized columnwise,

$$\|y - zD_j\|_2^2 = \|y\|_2^2 + z^2 - 2zD_j^Ty = cst(j) - 2zD_j^Ty \tag{20}$$

which minimum with respect to j does not depend on the value z of the nonzero coefficient z in the vector x (except for the sign of z). Therefore the support of x is $\text{argmax}_j |D_j^Ty|$.

Reasoning in the same manner for MSC does not yield a similar simple solution for finding the support. Indeed, even setting $k = 1$,

$$\underset{\|X_i\|_0 \leq 1}{\arg,\min} \|Y - DXB^T\|_F^2 = \underset{\forall i \leq r,\, z_i \in \mathbb{R}}{\arg,\min} \underset{\forall i \leq r,\, j_i \in [1,d]}{\arg,\min} \|Y - \sum_{i=1}^{r} z_i D_{j_i} B_i^T\|_F^2 \tag{21}$$

and the interior minimization problem of the right-hand side is still difficult, referred to as dictionary-based low-rank matrix factorization in [13]. Indeed, the cost can be rewritten as $\|Y - D(:,\mathcal{K})Diag(z)B^T\|_F^2$ with $\mathcal{K}$ the list of selected atoms in D. As discussed in section 3.2.1, the solutions to this problem in a noisy setting are in general not obtained by minimizing instead $\|YB(B^TB)^{-1} - D(:,\mathcal{K})Diag(z)\|_F^2$ with respect to $\mathcal{K}$ which would be solved in closed form.

Therefore, even with $k = 1$, a solution to MSC is not straightforward. While a greedy selection heuristic may not be straightforward to design, one may notice that if the rank had been set to one, i.e., if $r = 1$, then for any k one actually ends up with the usual sparse coding problem.

Indeed, given a matrix $V \in \mathbb{R}^{n\times m}$, the rank-one case means we try to solve the problem

$$\underset{x\in\mathbb{R}^d, \|x\|_0\leq k}{\arg,\min} \ \|V - Dxb^T\|_F^2 \tag{22}$$

which is equivalent to

$$\underset{x\in\mathbb{R}^d, \|x\|_0\leq k}{\arg,\min} \ \|\frac{1}{\|b\|_2^2}Vb - Dx\|_2^2 \,. \tag{23}$$

This is nothing more than Problem (SC). Consequently, we are now ready to design a hierarchical, *i.e.* block-coordinate, greedy algorithm. The algorithm updates one column of X at a time, fixing all the others. Then, finding the optimal solution for that single column is exactly a sparse-coding problem.

This leads to an adaptation of OMP for MSC that I call Hierarchical OMP (HOMP), where OMP is used to solve each sparse coding subproblem, see **Algorithm 1**. The routine OMP(x, D, k) applies Orthogonal Matching Pursuit to the input vector x with normalized dictionary D and sparsity level k and returns estimated codes and support. Note that after HOMP has stopped, it is useful to run a least square joint final update with fixed support as described in section 3.1.3 because the final HOMP estimates may not be optimal for the output support. HOMP does not easily inherit from OMP recovery conditions [50] because it employs OMP inside an alternating algorithm. On a practical side, any optimized implementation of OMP such as batch OMP [69] can be used to implement HOMP as a subroutine.

A note on restart: A restart condition is required, i.e., checking if the error increases after an inner iteration and rejecting the update in that case. Indeed OMP is simply a heuristic which, in general, is not guarantied to find the best solution to the sparse coding subproblem. When restart occurs, simply compute the best update with respect to the previously known support and move to the next column update. If restart occurs on all modes, then the algorithm stops with a warning. Due to this restart condition, the cost always decreases after each iteration, therefore it is guarantied that the HOMP algorithm either converges or stops with a warning.

Algorithm 1 Hierarchical OMP.

Input: data Y, dictionary D, sparsity level k, initial value X.
Output: estimated codes X, support S
Precompute D^TD and D^TY if memory allows.
while stopping criterion is not reached **do**
  for p from 1 to r **do**
    Set $Vb = \frac{1}{\|B_p\|_2^2}(Y - DX_{-p}B_{-p}^T)B_p$.
    Compute $X_p, S_p = \text{OMP}(Vb, D, k)$
    If error increased, reject this update, and perform a least squares update with the previous support. End if rejection happened for each column.
  end for
end while
Set X as the least squares solution following section 3.1.3 with support S.

3.3. Convex Heuristics to Solve MSC

The proposed greedy strategies TrickOMP and HOMP may not provide the best solutions to MSC or even converge to a critical point, and the practical performance of IHT is often not satisfactory (see section 3.5). Therefore, taking inspiration from existing works on sparse coding, one might as well tackle a convex problem which solutions are provably sparse. This means, first of all, finding convex relaxations to the $\ell_{0,0}$ regularizer.

In what follows, we study two convex relaxations and propose a FISTA-like algorithm for each. In both cases, the solution is provably sparse and there exist a regularization level such that the only solution is zero. Therefore, these regularizers force the presence of zeros in the columns of the solution.

3.3.1. A Columnwise Convex Relaxation: The Block LASSO Heuristic

A first convex relaxation of MSC is obtained by replacing each sparsity constraint $\|X_i\|_0 \leq k$ by an independent ℓ_1 regularization. This idea has already been theoretically explored in the literature, in particular in [60] where several support recovery results are established.

Practically, fixing a collection $\{\lambda_i\}_{i\leq r}$ of positive regularization parameters, the following convex relaxed problem, coined Block LASSO,

$$\underset{X\in\mathbb{R}^{d\times r}}{\arg,\min}\frac{1}{2}\|Y - DXB^T\|_F^2 + \sum_{i=1}^{r}\lambda_i\|X_i\|_1 \tag{CVX1}$$

provides candidates solutions to MSC. This is a convex problem since each term is convex. Moreover, the cost is coercive so that a solution always exists. However, it is not strictly convex in general, thus several solutions may co-exist.

Adapting the proof in [47], one can easily show that under uniqueness assumptions, solutions to Problem (CVX1) are indeed sparse.

PROPOSITION 4. *Let X^* a solution to Problem (CVX1), and suppose that X^* is unique. Then denoting S_i the support of each column X_i^*, it holds that D_{S_i} is full column-rank, and that $|S_i| \leq n$.*

This shows that the Block LASSO solutions have at most n non-zeros in each column. Therefore, the columnwise ℓ_1 regularization induces sparsity in all columns of X and solving Problem (CVX1) is relevant to perform MSC. But this does not show that the support recovered by solving Block LASSO is always the support of the solution of MSC, see [60] for such recovery results given assumptions on the regularizations parameters λ_i.

Conversely, it is of interest to know above which values of λ_i the solution X is null. Proposition 5 states that such a columnwise

maximum regularization can be defined. This can be used to set the λ_i regularization parameters individually given a percentage of $\lambda_{i,\max}$ and provide a better intuition for choosing each λ_i.

PROPOSITION 5. *A solution X^* of Problem (CVX1) satisfies $X^* = 0$ if and only if for all $i \leq r$, $\lambda_i \geq \lambda_{i,\max}$ where $\lambda_{i,\max} = \|D^T Y B_i\|_\infty$ (element-wise absolute value maximum). Moreover, if $X_i^* = 0$ is a column of a solution, then $\lambda_i \geq \lambda_{i,\max}$.*

Relation with other models: Vectorization easily transform the matrix Problem (CVX1) into a vector problem reminiscent of the LASSO. In fact, if the regularization parameters $\lambda_i = \lambda$ are set equal, then Problem (CVX1) is nothing else than the LASSO. A related but different model is SLOPE [70]. Indeed, in SLOPE, it is possible to have a particular λ_{ji} for each **sorted** entry $X_{\sigma_i(j)i}$, but in Problem (CVX1) the regularization parameters are fixed by blocks and the relative order of elements among the blocks changes in the admissible space. Thus the relaxed Problem (CVX1) cannot be expressed as a particular SLOPE problem.

Solving block LASSO with FISTA: A workhorse algorithm for solving the LASSO is FISTA [55]. Moreover, the regularization term is separable in X_i, and the proximal operator for each separable term is well-known to be the soft-thresholding operator

$$S_{\lambda_i}(x) = [|x| - \lambda]^+ \text{sign}(x),\ S_{[\lambda_1,\ldots,\lambda_r]}(X) = [S_{\lambda_1}(X_1), \ldots, S_{\lambda_r}(X_r)] \tag{24}$$

understood element-wise. Therefore, the FISTA algorithm can be directly leveraged to solve Problem (CVX1), see **Algorithm 2** coined Block-FISTA hereafter. Convergence of the cost iterates of this proposed extrapolated proximal gradient method is ensured as soon as the gradient step is smaller than the inverse of the Lipschitz constant of the quadratic term, which is given by $\sigma(D)^2\sigma(B)^2$ with $\sigma(M)$ the largest singular value of M. The resulting FISTA algorithm is denoted as Block FISTA. Note that after Block FISTA returns a candidate solution, this solution's support is extracted, truncated to be of size k columnwise using hard thresholding, and an unbiased MSC solution is computed with that fixed support.

3.3.2. Mixed ℓ_1 Norm for Tightest Convex Relaxation

The columnwise convex relaxation introduced in section 3.3.1 has the disadvantage of introducing a potentially large number of regularization parameters that must be controlled individually to obtain a target sparsity level columnwise. Moreover, this relaxation is not the tightest convex relaxation of the $\ell_{0,0}$ regularizer on $[-1, 1]$.

It turns out that using the tightest convex relaxation of the $\ell_{0,0}$ regularizer does solve the proliferation of regularization parameters problem, and in fact this yields a uniform regularization on the columns of X. This comes however at the cost of loosing some sparsity guaranties as detailed below.

PROPOSITION 6. *The tightest convex relaxation of $\ell_{0,0}$ on $[-1, 1]^{d\times r}$ is $\ell_{1,1} : X \mapsto \max_i \|X_i\|_1 =: \|X\|_{1,1}$.*

According to Proposition 6, Problem (MSC) may be relaxed into the following convex optimization problem coined Mixed LASSO hereafter:

$$\arg,\min_{X\in\mathbb{R}^{d\times r}} \frac{1}{2}\|Y - DXB^T\|_F^2 + \lambda\|X\|_{1,1}\,. \tag{25}$$

To again leverage FISTA to solve Problem (25) and produce a support for a solution of MSC requires to compute the proximal operator of the regularization term. For the $\ell_{1,1}$ norm, the proximal operator has been shown to be computable exactly with little cost using a bisection search [71–73]. In this work I used the low-level implementation of [72][6]. The resulting Mixed-FISTA algorithm is very similar to **Algorithm 2** but using the $\ell_{1,1}$ proximal operator instead of soft-thresholding, and its pseudo-code is therefore differed to the **Supplementary Material**.

Properties of the Mixed LASSO: Differently from the Block LASSO problem, the Mixed LASSO may not have sparse solutions if the regularization is not set high enough.

PROPOSITION 7. *Suppose there exist a unique solution X^* to the Mixed LASSO problem. Let $\mathcal{I}$ the set of indices such that for all i in $\mathcal{I}$, $\|X_i^*\|_1 = \|X^*\|_{1,1}$. Denote S the support of X^*. Then there exist at least one i in $\mathcal{I}$ such that D_{S_i} is full column rank, and $\|X_i^*\|_0 \leq n$. Moreover, if D is overcomplete, $\mathcal{I} = \{1, \ldots, r\}$.*

This result is actually quite unsatisfactory. The uniqueness condition, which is generally satisfied for the LASSO, seems a much stricter restriction in the Mixed LASSO problem. In particular all columns of the solution must have equal ℓ_1 norm in the overcomplete case. Moreover sparsity is only ensured for one column. However, from the simulations in the Experiment section, it seems that in general the Mixed LASSO problem does yield sparser solutions than the above theory predicts.

Maximum regularization: Intuitively, by setting the regularization parameter λ high enough, one expects the

Algorithm 2 FISTA for Block LASSO (Block-FISTA)

Input: data Y, dictionary D, mixing matrix B, regularization ratio $\alpha \in [0, 1]^r$, sparsity level k, initial value X.
Output: estimated codes X, support S.
Precompute D^TD, D^TYB and B^TB if memory allows.
Compute $\lambda_{i,\max}$ as in Proposition 5, and set $\lambda_i = \alpha_i\lambda_{i,\max},\ \lambda = [\lambda_1, \ldots, \lambda_r]$
Compute stepsize $\eta = \frac{1}{\sigma(D)^2\sigma(B)^2}$
Initialize $Z = X,\ \beta = 1$.
while stopping criterion is not reached **do**
  $X_{\text{old}} = X$
  $X = S_{\eta\lambda}\left(Z - \eta(D^TDZB^TB - D^TYB)\right)$
  $\beta_{\text{old}} = \beta$
  $\beta = \frac{1}{2}(1 + \sqrt{1 + 4\beta^2})$
  $Z = X + \frac{\beta_{\text{old}}-1}{\beta}(X - X_{\text{old}})$
end while
Estimate the support $S = S(X)$
Set X as the least squares solution following section 3.1.3 with support S.

[6] https://github.com/bbejar/prox-l1oo.

solution of the Mixed LASSO to be null. Below I show that this is indeed true.

PROPOSITION 8. *$X^* = 0$ is the unique solution to the Mixed LASSO problem if and only if $\lambda \geq \lambda_{max}$ where $\lambda_{max} = \sum_{i=1}^{r} \|D^T Y B_i\|_\infty$.*

Similarly to Block-FISTA, this result can be used to chose the regularization parameter as a percentage of $\lambda_{\max}$.

3.4. Nonnegative MSC

In this section I quickly describe how to adapt the Block LASSO strategy in the presence of nonnegativity constraints on X. This is a very useful special case of MSC since many LRA models use nonnegativity to enhance identifiability such as Nonnegative Matrix Factorization or Nonnegative Tucker Decomposition.

The nonnegative MSC problem can be written as follows:

$$\underset{X\geq 0,\ \|X_i\|_0 \leq k}{\arg,\min} \ \|Y - DXB^T\|_F^2 \,. \tag{26}$$

Block LASSO minimizes a regularized cost which becomes quadratic and smooth due to the nonnegativity constraints:

$$\underset{X\geq 0}{\arg,\min} \|Y - DXB^T\|_F^2 + \sum_{i=1}^{n} \lambda_i 1^T X_i \,. \tag{27}$$

The Nonnegative least squares Problem (26) is easily solved by a modified nonnegative Block-FISTA, where the proximal operator is a projection on the nonnegative orthant.

While in the general case a least-squares update with fixed support is performed at the end of Block-FISTA to remove the bias induced by the convex penalty, in the non-negative case a nonnegative least squares solver is used on the estimated support with a small ridge regularization. One drawback of this approach is that the final estimate for X might have strictly smaller sparsity level than the target k in a few columns. In the **Supplementary Material**, a quick comparison between Block-FISTA and its nonnegative variant shows the positive impact of accounting for nonnegativity for support recovery.

3.5. Comparison of the Proposed Heuristics

Numerical experiments discussed hereafter provide a first analysis of the proposed heuristics to solve MSC. A few natural questions arise upon studying these methods:

- Are some of the proposed heuristics performing well or poorly in terms of support recovery in a variety of settings?
- Are some heuristics much faster than others in practice?

We shall provide tentative answers after conducting synthetic experiments. However, because of the variety of proposed methods and the large number of experimental parameters (dimensions, noise level, conditioning of B and coherence of D, distribution of the true X, regularization levels), it is virtually impossible to test out all possible combinations and the conclusions of this section can hardly be extrapolated outside our study cases. All the codes used in the experiments below are freely available online[7]. In particular, all the proposed algorithms are implemented in Python, and all experiments and figures can be reproduced using the distributed code.

In the **Supplementary Material**, I cover additional questions of importance: the sensitivity of convex relaxation methods to the choice of the regularization parameter, the sensitivity of all methods to the conditioning of B and the sensitivity to random and zero initializations.

3.5.1. Synthetic Experiments

In general and unless specified otherwise we set $n = 50, m = 50, d = 100, k = 5, r = 6$. The variance of additive Gaussian white noise is tuned so that the empirical SNR is exactly 20dB. To generate D, its entries are drawn independently from the Uniform distribution on $[0, 1]$ and its columns are then normalized. The uniform distribution is meant to make atoms more correlated and therefore increase the dictionary coherence. The entries of B are drawn similarly. However, the singular value decomposition of B is then computed, its original singular values discarded and replaced with linearly spaced values from 1 to $\frac{1}{\text{cond}(B)}$ where $\text{cond}(B) = 200$. The values of the true X are generated by first selecting a support randomly (uniformly), then sampling nonzero entries from standard Gaussian i.i.d. distributions. The initial X is set to zero.

In most test settings, the metric used to assess performance is based on support recovery. To measure support recovery, the number of correctly found nonzero position is divided by the total number of non-zeros to be found, yielding a 100% recovery rate if the support is perfectly estimated and 0% if no element in the support of X is correctly estimated.

The input regularization in Mixed-FISTA and Block-FISTA is always scaled from 0 to 1 by computing the maximum regularization, see Propositions 5 and 8. To choose the regularization ratio α, before running each experiment, Mixed-FISTA and Block-FISTA are ran on three instances of separately generated problems with the same parameters as the current test using a grid $[10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}]$. Then the average best α for these three tests is used as the regularization level for the whole test. This procedure is meant to mimic how a user would tune regularization for a given problem, generating a few instances by simulation and picking a vaguely adequate amount of regularization.

The stopping criterion for all methods is the same: when the relative decrease in cost $\frac{|err^{it+1} - err^{it}|}{err^{it}}$ reaches 10^{-6}, the algorithm stops. The absolute value allows for increasing the cost. Note that the cost includes the penalty terms for convex methods. Additionally, the maximum number of iterations is set to 1,000.

3.5.2. Test 1: Support Recovery vs. Noise Level

For the first experiment, the noise level varies in power such that the SNR is exactly on a grid $[1,000, 100, 50, 40, 30, 20, 15, 10, 5, 2, 0]$. A total of 50 realizations of triplets (Y, D, B) are used in this experiment,

[7] https://github.com/cohenjer/mscode and https://github.com/cohenjer/dlra.

as well as in Test 2 and 3. Results are shown in **Figure 1**.

It can be observed that IHT performs poorly for all noise levels. As expected, TrickOMP performs well only at low noise levels. HOMP and the FISTA methods have degraded performance when the SNR decreases, but provide with satisfactory results overall. Block-FISTA seems to perform the best overall.

3.5.3. Test 2: Support Recovery vs. Dimensions (k,d)

This time the sparsity level k is on a grid $[1, 2, 5, 10, 20]$ while the number of atoms d is also on a grid $[20, 50, 100, 200, 400]$. **Figure 2** shows the heat map results.

The TrickOMP has strikingly worse performance than the other methods. Moreover, as the number of atoms d increases or when the number of admissible supports $\binom{d}{k}$, correct atom selection becomes more difficult for all methods. Again, Block-FISTA seems to perform better overall, in particular for large d.

3.5.4. Test 3: Runtime vs. Dimensions (n,m) and (k,d)

In this last test, all algorithms are run until convergence for various sizes $n = [10, 50, 1,000]$ and $m = [10, 50, 1,000]$, or for various sparsity parameters $k = [5, 10, 30]$ and $[50, 100, 1,000]$. **Table 1** provides runtime and number of iterations averaged for $N = 10$ runs.

From **Table 1**, it can be inferred that HOMP is often much slower than the other methods. TrickOMP is always very fast since it relies on OMP which runs in exactly k iterations. Block-FISTA generally runs faster than Mixed-FISTA. Moreover, it is not much slower than faster methods such as IHT and TrickOMP, in particular for larger sparsity values.

4. DISCUSSION

In all the experiments conducted above and in the **Supplementary Material**, the Block-FISTA algorithm provides the best trade-off between support recovery and computation

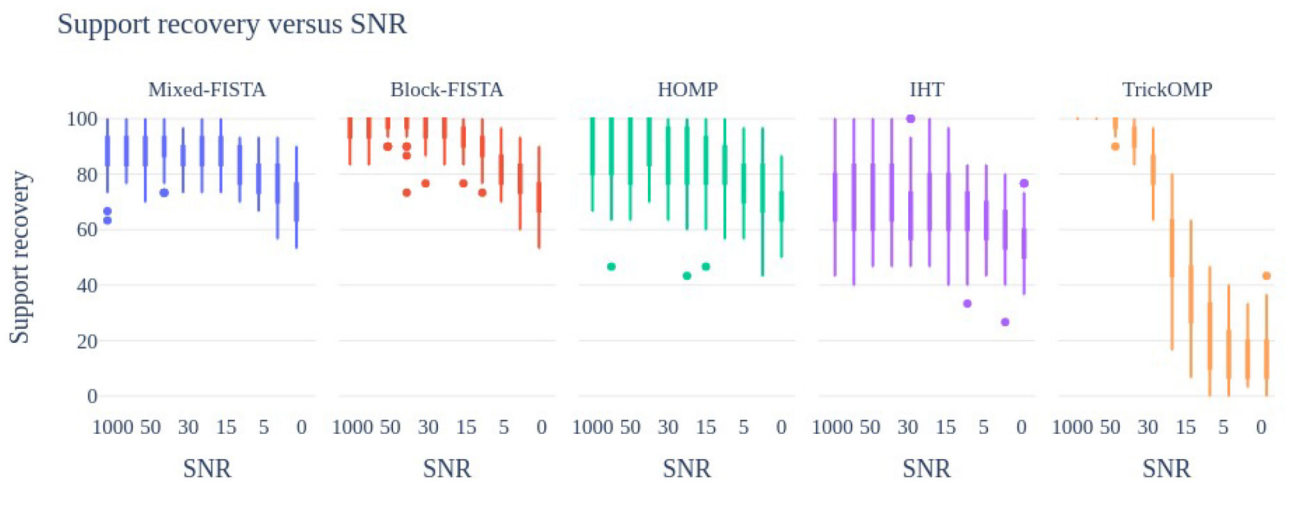


FIGURE 1 | Support recovery (in %) of the proposed heuristics to solve MSC at various Signal to Noise ratios. A total of 50 problems instances are used, with a single random but shared initialization for all methods.

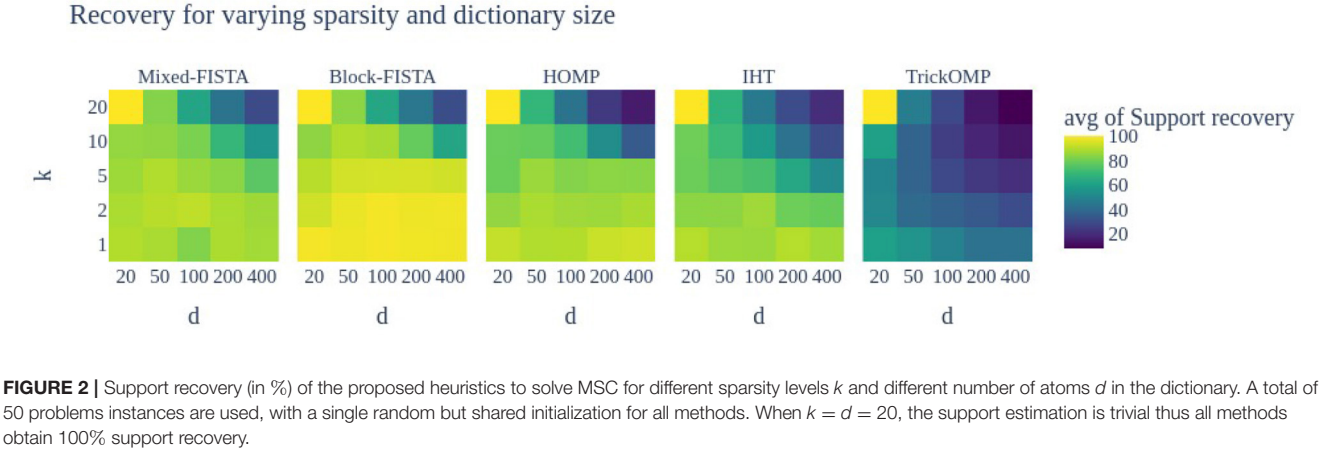


FIGURE 2 | Support recovery (in %) of the proposed heuristics to solve MSC for different sparsity levels k and different number of atoms d in the dictionary. A total of 50 problems instances are used, with a single random but shared initialization for all methods. When $k = d = 20$, the support estimation is trivial thus all methods obtain 100% support recovery.

TABLE 1 | Average computation time in seconds and number of iterations with respect to (n, m) and (k, d).

(n,m)	(10, 10)	(10, 50)	(10, 1,000)	(50, 10)	(50, 50)	(50, 1,000)	(1,000, 10)	(1,000, 50)	(1,000, 1,000)
HOMP	0.8, 148	0.9, 149	2, 88	2.6, 450	2, 330	10, 290	13, 280	13, 247	70, 376
M.-FISTA	0.4, 1000	0.4, 950	0.5, 875	0.4, 842	0.4, 799	1.6, 734	0.6, 393	1, 365	4.4, 388
B.-FISTA	0.3, 998	0.3, 898	0.3, 907	0.3, 843	0.3, 764	1, 724	0.5, 376	0.7, 369	4.2, 388
IHT	0.1, 356	0.2, 354	0.1, 254	0.07, 121	0.09, 105	0.2, 116	0.1, 92	0.2, 63	0.9, 80
TrickOMP	0.03, –	0.03, –	0.03, –	0.03, –	0.03, –	0.04, –	0.08, –	0.07, –	0.09, –
(d,k)	**(50, 5)**	**(50, 10)**	**(50, 30)**	**(100, 5)**	**(100, 10)**	**(100, 30)**	**(1,000, 5)**	**(1,000, 10)**	**(1,000, 30)**
HOMP	0.59, 95	2.2, 185	64, 886	0.39, 50	4.3, 345	14, 176	1.3, 40	3.9, 53	53, 161
M.-FISTA	0.1, 356	0.3, 467	0.5, 456	0.37, 691	0.56, 814	0.78, 609	8.9, 1000	9.6, 1000	9.5, 1000
B.-FISTA	0.12, 492	0.23, 319	0.57, 373	0.23, 668	0.38, 679	0.63, 619	5.2, 1000	4.7, 369	5.4, 1000
IHT	0.05, 103	0.15, 92	0.69, 1000	0.06, 111	0.15, 100	0.52, 324	2.5, 215	3.3, 353	4.6, 528
TrickOMP	0.02, –	0.08, –	0.57, –	0.43, –	0.14, –	0.55, –	0.04, –	0.13, –	0.67, –

The format is (time, it). Computations are single threaded, and use a Intel® Core™ i7-8650U CPU @ 1.90 GHz × 8 processor. TrickOMP exactly runs k steps of OMP equivalent to one HOMP inner iteration, therefore TrickOMP iterations are not reported in the table. Maximal number of iteration was set to 1,000.

time. Moreover, it is very easy to extend to nonnegative low-rank approximation models which are very common in practice. Therefore, to design an algorithm for DLRA, we shall make use of Block-FISTA (topped with a least-squares update with fixed support) as a solver for the MSC sub-problem. Note however than Block-FISTA required to select many regularization parameters, but the proposed heuristic using a fixed percentage of $\lambda_{i,\max}$ worked well in the above experiments.

5. DICTIONARY-BASED LOW RANK APPROXIMATIONS

5.1. A Generic AO Algorithm for DLRA

Now that a reasonably good heuristic for solving MSC has been found, let us introduce an AO method for DLRA based on Block-FISTA. The proposed algorithm is coined AO-DLRA and is summarized in **Algorithm 3**. It boils down to solving for X with Block-FISTA (**Algorithm 2**) and automatically computed regularization parameters, and then solving for the other blocks using any classical alternating method specific to the LRA at hand. Because solving exactly the MSC problem is difficult, even using Block-FISTA with well tuned regularization parameters, it is not guarantied that the X update will decrease the global cost. In fact in practice the cost may go up, and storing the best update along the iterations is good practice.

5.1.1. Selecting Regularization Parameters

It had already been noted in section 3.5 that choosing the multiple regularization parameters λ_i of Block-FISTA can be challenging. In the context of Alternating Optimization, this is even more true. Indeed, the (structured) matrix B is updated at each outer iteration, therefore there is a scaling ambiguity between X and B that makes any arbitrary choice of regularization level λ_i meaningless. Moreover the values $\lambda_{i,\max}$ change at each outer iteration. Consequently, obtaining a sparsity regularization percentage α_i in each column of X at each iteration is challenging without some *ad-hoc* tuning in each outer iteration. To that end, the regularization percentages $\alpha = [\alpha_1, \ldots, \alpha_r]$ are tuned inside each inner loop until the columnwise number of non-zeros reaches a target range $[k, k+\tau]$ where $\tau \geq 0$ is user-defined. This range is deliberately shifted to the right so that the each column does not have size strictly less than k non-zeros. Indeed, in that situation, a few atoms would have to be chosen arbitrarily during the unbiased estimation. More precisely, when a column has too

Algorithm 3 An AO algorithm for DLRA (AO-DLRA)

Input: Initial guesses $X^{(0)}, B^{(0)}$, data Y, dictionary D, sparsity level $k \leq n$, initial regularization $\alpha \in [0,1]^r$, iteration number $l_{\max}$, sparsity tolerance τ.
Output: Best estimated factors $X^{(l)}$ and $B^{(l)}$
Precompute D^TD, D^TY and $\sigma_D = \sigma(D^TD)$ if memory allows.
for $l = 1 \ldots l_{\max}$ **do**
  B update:
  Compute $B^{(l)} \in \Omega_B$ that decreases the cost in Problem (2) with respect to B.
  X update:
  Stepsize evaluation: $\eta^{(l)} = \frac{1}{\sigma_D \sigma({B^{(l)}}^T B^{(l)})}$.
  Compute efficiently data-factor product$(D^TY)B^{(l)}$ and inner products ${B^{(l)}}^T B^{(l)}$.
  $(*)$ Update $X^{(l)}$ using Block-FISTA (**Algorithm 2**) with regularization parameters α, stepsize $\eta^{(l)}$ and initial guess $X^{(l-1)}$
  while $\exists i \leq r,\ \|X_i^{(l)}\|_0 \notin [k, k+\tau]$ **do**
    for $i = 1 \ldots r$ **do**
      if $\|X_i^{(l)}\|_0 \leq k$ **then**
        Decrease regularization $\alpha := \alpha/1.3$
      else if $\|X_i^{(l)}\|_0 \geq k+\tau$ **then**
        Increase regularization $\alpha := \min(1.01\alpha, 1)$
      end if
    end for
    Go to $(*)$ with $X^{(l-1)} := X^{(l)}$
  end while
  Unbiaised estimation: update $X^{(l)}$ with fixed support $S_{X^{(l)}}$ as in section 3.1.3.
  Store $X^{(l)}$ and $B^{(l)}$ if residuals $\|Y - DX^{(l)}B^{(l)}\|_F^2$ have improved with respect to previous best.
end for

many zeros, α_i is divided by 1.3, while it is multiplied by 1.01 if it has few non-zeros. Note that an interesting research avenue would be to use an adaption of homotopy methods [74] for Block LASSO instead of FISTA, which would remove the need for this heuristic tuning.

5.1.2. A Provably Convergent Algorithm: Inertial Proximal Alternating Linear Minimization (iPALM)

The AO-DLRA algorithm proposed above is a heuristic with several arbitrary choices and no convergence guaranties. If designing an efficient AO algorithm with convergence guarantees proved difficult, designing a convergent algorithm is in fact straightforward using standard block-coordinate non-convex methods. I focus in the following on the iPALM algorithm, which alternates between a proximal gradient step on X similar the one discussed in section 3.2.2 and a proximal gradient step on B. iPALM is guarantied to converge to a stationary point of the DLRA cost, despite the irregularity of the semi-algebraic $\ell_{0,0}$ map. A pseudo-code for iPALM to initialize **Algorithm 3** is provided in the **Supplementary Material**.

5.1.3. Initialization Strategies

Because DLRA is a highly non-convex problem, one can only hope to reach some stationary point of the cost in Problem (2). Furthermore, the sparsity constraint on the columns of X implies that X_i must belong to a finite union of subspaces, making the problem combinatorial by nature. Using a local heuristic such as AO-DLRA or iPALM, it is expected to encounter many local minima—a fact also confirmed in practical experiments reported in section 5.2 and in previous works [13]. Therefore, providing an initial guess for X and B close to a good local minimum is important.

There are at least two reasonable strategies to initialize the DCPD model. First, for any low-rank approximation model which is mildly identifiable (such as NMF, CPD), the suggested method is to first compute the low-rank approximation with standard algorithms to estimate $A^{(0)}$ and $B^{(0)}$, and then perform sparse coding on the columns of $A^{(0)}$ to estimate $X^{(0)}$. The identifiability properties of these models should ensure that $A^{(0)}$ is well approximated by DX with sparse X. Second, several random initialization can be carried out, only to keep the best result. A third option for AO-DLRA would be to use a few iterations of the iPALM algorithm itself initialized randomly, since iPALM iterations are relatively cheap. However, it is shown in the experiments below that this method does not yield good results.

5.2. Experiments for DLRA

In the next section, two DLRA models are showcased on synthetic and real-life data. First, the Dictionary-based Matrix Factorization (DMF, see below) model is explored for the task of matrix completion in remote sensing. It is shown that DMF allows to complete entirely missing rows, something that low-rank completion cannot do. Second, nonnegative DCPD (nnDCPD) is used for denoising smooth images in the context of chemometrics, and better denoising performance are obtained with nnDCPD than with plain nonnegative CPD (nnCPD) or when post-processing the results of nnCPD. Nevertheless, the goal of these experiments is not to establish a new state-of-the-art in these particular, well-studied applications, but rather to demonstrate the versatile problems that may be cast as DLRA and the efficiency of DLRA when compared to other low-rank strategies. The performance of AO-DLRA and iPALM in terms of support recovery and relative reconstruction error for DMF and DCPD is then further assessed on synthetic data.

5.2.1. Dictionary-Based Matrix Factorization With Application to Matrix Completion

Let us study the following Dictionary-based Matrix Factorization model:

$$\underset{X\in\mathbb{R}^{d\times r},\ \forall i\leq r,\ \|X_i\|_0\leq k,\ B\in\mathbb{R}^{m\times r}}{\arg,\min} \|Y - DXB^T\|_F^2 . \tag{28}$$

Low-rank factorizations have been extensively used in machine learning for matrix completion [75], since the low-rank hypothesis serves as regularization for this otherwise ill-posed problem. A use case for DMF is the completion of a low-rank matrix which has missing rows. Using a simple low-rank factorization approach would fail since a missing row removes all information about the column-space on that row. Formally, if a data matrix $Y \approx AB$ has missing rows indexed by I, then the rows of matrix A in I cannot be estimated directly from Y. Dictionary-based low-rank matrix factorization circumvents this problem by expressing each column of A as a sparse combination of atoms in a dictionary D, such that $Y \approx DXB^T$ with X columnwise sparse. While fitting matrices X and B can be done on the known entries, the reconstruction DXB^T will provide an estimation of the whole data matrix Y, including the missing rows. Formally, first solve

$$\underset{X\in\mathbb{R}^{(n-|I|)\times r},\ \forall i\leq r,\ \|X_i\|_0\leq k,\ B\in\mathbb{R}^{m\times r}}{\arg,\min} \|Y_{:,\bar{I}} - D_{:,\bar{I}}XB^T\|_F^2 \tag{29}$$

using **Algorithm 3** and then build an estimation for the missing values in Y as $\widehat{Y}_{:,I} = D_{:,I}XB^T$. Initialization is carried out using random factors sampled element-wise from a normal distribution.

In remotely acquired hyperspectral images, missing rows in the data matrix are common as they correspond to corrupted pixels. Moreover, it is well-known that many hyperspectral images are approximately low-rank. Therefore, in this toy experiment, a small portion of the Urban hyperspectral image is used to showcase the proposed inpainting strategy. Urban is often considered to be between rank 4, 5, or 6 [76]. I will use $r = 4$ hereafter. Urban is a collection of 307×307 images collected on 162 clean bands. For the sake of simplicity, only a 20×20 patch is considered, with 50 randomly-chosen pixels removed from this patch in all bands, and therefore after pixel vectorization the data matrix $Y \in \mathbb{R}^{400\times 162}$ has 50 missing rows. Since columns of factor A in this factorization should stand for patches of abundance maps, they are reasonably sparse in a 2D-Discrete Cosine Transform dictionary D, here using $d = 400$ atoms. Note that similar strategies for HSI denoising have been studied in the literature albeit without columnwise sparsity imposed on X, see [77].

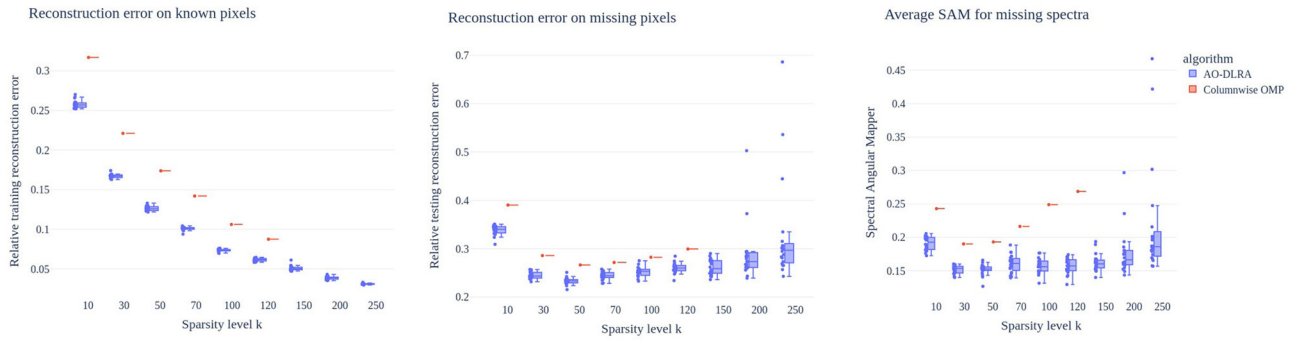


FIGURE 3 | Performance of DMF (blue) vs. Columnwise OMP (red, single valued box) for inpainting a patch of the Urban HSI with missing pixels at various sparsity levels. Twenty random initializations are used for DMF.

We measure the performance of two strategies: DMF computed on known pixels solving Problem (29), and OMP for each band individually. Both approaches use the same dictionary D. To evaluate performance, the test estimation error on missing pixels $\|Y_I - \widehat{Y}_I\|_F / \|Y_I\|_F$ is computed alongside with the average Spectral Angular Mapper (SAM) $\frac{1}{|I|}\sum_{i\in I} \arccos(\frac{Y_i^T \widehat{Y}_i}{\|Y_i\|_2 \|\widehat{Y}_i\|_2})$ on all missing pixels. The sparsity level k is defined on the grid $[10, 30, 50, 70, 100, 120, 150^*, 200^*, 250^*]$, where k^* is not used for the OMP inpainting because of memory issues. Results are averaged on $N = 20$ different initializations. For AO-DLRA, the initial regularization α is set to 5×10^{-3}, and $\tau = 20$.

Figure 3 shows the reconstruction error and SAM obtained after each initialization for various sparsity levels. First, for both metrics, there exist a clear advantage of the DMF approach when compared to sparse coding band per band. In particular, the band-wise OMP reconstruction does not yield good spectral reconstruction. Second, DMF apparently works similarly to sparse coding approaches for inpainting: if the sparsity level is too low the reconstruction is not precise, but if the sparsity level is too large the reconstruction is biased. Therefore, DMF effectively allows to perform inpainting with sparse coding on the factors of a low-rank matrix factorization.

5.2.2. Dictionary-Based Smooth Canonical Polyadic Decomposition With Application to Data Denoising

The second study examines the DCPD discussed around Equation (4) to perform denoising using smoothness. In this context, the data tensor Y is noisy, meaning formally that

$$\tilde{Y} = Y + \epsilon, \text{ and } Y = A(B \odot C)^T \tag{30}$$

where ϵ has a large power compared to $A(B \odot C)^T$ (*e.g.* Signal to Noise Ratio at -8.7dB in the following). Furthermore, let us suppose that A has smooth columns. The dictionary constraint can enforce smoothness on A by choosing D as a large collection of smooth atoms, in this case B-splines[8]. Because the first mode is constrained such that $A = DX$, each column of A is a sparse combination of smooth functions and will therefore itself be smooth. The sparsity constraint k prevents the use of too many splines and ensures that A is indeed smooth. Hereafter, the sparsity value is fixed to $k = 6$.

There has been significant previous works on smooth CPD, perhaps most related to the proposed approach is the work of [8]. Their method also consists in choosing a dictionary D of B-splines. However, this dictionary has very few atoms (the actual number is determined by cross-validation), and picking the knots for the splines requires either time-consuming hand crafting, or some cross validation set. The advantage of their approach is that no sparsity constraint is imposed since there are already so few atoms in D, and the problem becomes equivalent to CPD on the smoothed data.

The rationale however is that heavily crafting the splines is unnecessary. Using DCPD allows an automatic picking of good (if not best) B-splines. Furthermore, each component in the CPD may use different splines while the method of [8] uses the same splines for all the components. Hand-crafting is still required to build the dictionary, but one does not have to fear introducing an inadequate spline.

An advantage of B-splines is that they are nonnegative, therefore one can compute nonnegative DCPD by imposing nonnegativity on the sparse coefficients X as explained in section 3.4. Imposing nonnegativity in the method of [8] is not as straightforward albeit doable [35].

For this study the toy fluorescence spectroscopy dataset "amino-acids" available online[9] is used. Its rank is known to be $r = 3$, and dimensions are $201 \times 61 \times 5$. Fluorescence spectroscopy tensors are nonnegative, low-rank and feature smooth factors. In fact factors are smooth on two modes related to excitation and emission spectra, therefore a double-constrained DCPD is also of interest. It boils down to solving

$$\underset{\|X_i^{(A)}\|_0 \leq k^{(A)},\ \|X_i^{(B)}\|_0 \leq k^{(B)},\ C \in \mathbb{R}^{m_2 \times r}}{\text{minimize}} \|\tilde{Y} - D^{(A)}X^{(A)}(D^{(B)}X^{(B)} \odot C)^T\|_F^2 . \tag{31}$$

where $D^{(A,B)}$ and $k^{(A,B)}$ are the respective dictionaries of sizes 201×180 and 61×81 and sparsity targets for modes one and

[8]The exact implementation of D is detailed in the code.

[9]http://www.models.life.ku.dk/Amino_Acid_fluo.

two. The third mode factor contains relative concentrations [78]. Additional Gaussian noise is added to the data so that the effective SNR used in the experiments is $-8.7dB$. Because CPD is identifiable, in particular for the amino-acids dataset, nnCPD of the noisy data is used for initialization, computed using Hierarchical Alternating Least Squares [45]. We compare sparse coding one or two modes of the output of HALS with AO-DCPD (i.e., AO-DLRA used to compute DCPD) with smoothness on one mode, two modes, and on two modes with nonnegativity (AO-nnDCPD). We set $\alpha = 10^{-3}$ and $\tau = 5$, and $k^{(A)} = k^{(B)} = 6$.

Figure 4 shows the reconstructed factors and the relative error $\|\widehat{Y} - Y\|_F / \|Y\|_F$ with respect to the true data, and **Figure 5** shows one slice (here the fourth one) of the reconstructed tensor. It can be observed both graphically and numerically that DCPD techniques are overall superior to post-processing the output of HALS. In particular, the factors recovered using nnDCPD with dictionary constraints on two modes are very close to the true factors (obtained from the nnCPD of the clean data) using only $k = 6$ splines at most.

5.2.3. Performance of AO-DLRA for DMF and DCPD on Synthetic Data

To assess the performance of AO-DLRA in computing DMF and DCPD, the support recovery and reconstruction error are monitored on synthetic data. For both model, a single initialization is performed for $N = 100$ problem instances with the same hyperparameters. Matrices D, X, B, C involved in both problems and the noise tensors are generated as in section 3.5. The rank is $r = 6$ for a sparsity level of $k = 8$ and the conditioning of B is set to 2×10^2. In the DMF experiment, the sizes are $n = m = 50$, $d = 60$ and the SNR is 100dB. For DPCD, we set $n = 20, m_1 = 21, m_2 = 22, d = 30$ and the SNR is 30 dB. We set $\alpha = 10^{-2}, \tau = 20$ for AO-DMF and $\alpha = 10^{-4}, \tau = 20$ for AO-DCPD.

A few strategies are compared: AO-DLRA initialized randomly, iPALM initialized randomly, and AO-DLRA initialized with iPALM. The same random initialization is used for all methods in each problem instance. For DCPD, we also consider AO-DLRA and iPALM initialized with a CPD solver (here Alternating Least Squares), and sparse coding the output

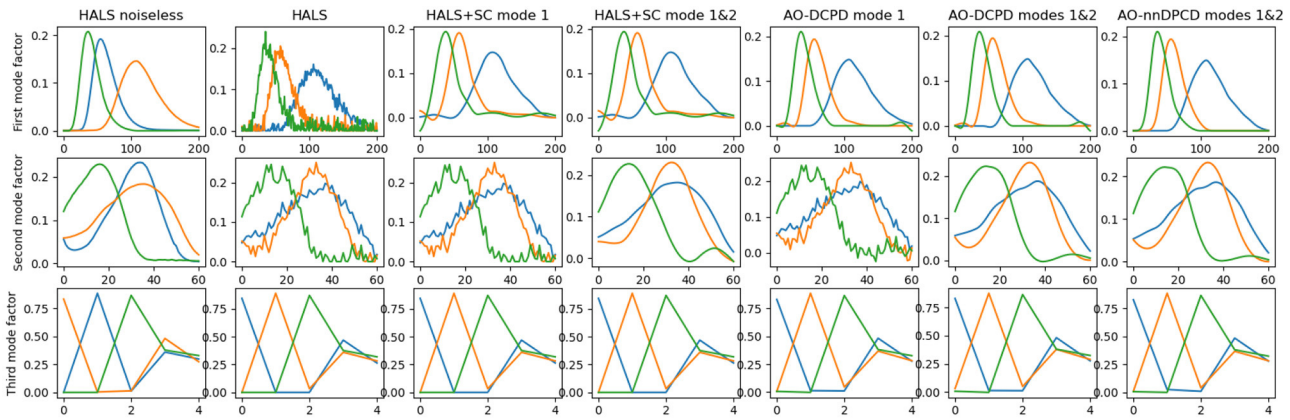


FIGURE 4 | Factors estimated from nonnegative CPD and dictionary-based (non-negative) CPD with smoothness imposed by B-splines on either one (emission, **top row**) or two (emission, excitation in **middle row**) modes. The **(bottom)** row shows the third mode factor that relates to relative concentration of the amino-acids in the mixture. The right-most plot shows AO-DLRA when imposing smoothness on two modes and non-negativity on all modes.

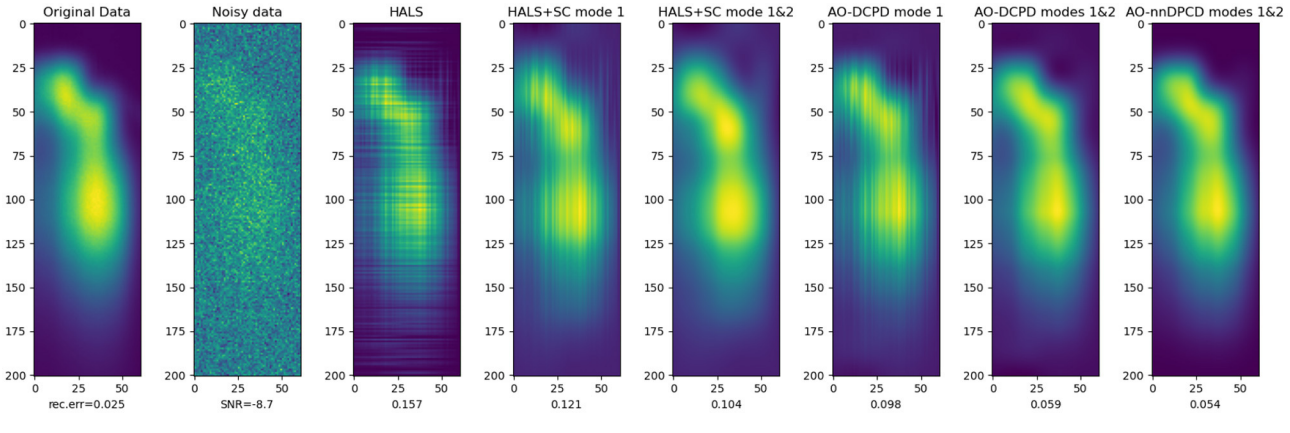


FIGURE 5 | The fourth slice (index 3 if counting from 0) of the reconstructed tensors, see **Figure 4** for details on each method. Row index relates to the emission wavelength while the column index relates to the excitation wavelength. The bottom values indicate the relative reconstruction error with respect to the clean tensor.

of the ALS. AO-DLRA always stops after at most 100 iterations, while iPALM runs for at most 1,000 iterations or when relative error decrease is below 10^{-8}. The safeguard for stepsize in iPALM is set to $\mu = 0.5$ for DMF and $\mu = 1$ for DCPD.

Figure 6 shows the obtained results. It can be observed that both in DMF and DCPD, iPALM initialization is not significantly better than random initialization, and iPALM in fact provides quite poor results on its own. For DCPD, support recovery and reconstruction error is always better with AO-DLRA than when sparse coding the output of the ALS, even with random initialization. Overall, these experiments show that while the DLRA problem is challenging (the optimal support is almost always never found), reasonably good results are obtained using the proposed AO-DLRA algorithm. Furthermore, the support recovery scores of DMF using AO-DLRA are much higher than what could be obtained by chance, randomly picking elements in the support. This observation hints toward the identifiability of DMF. Remember indeed that without the dictionary constraint, matrix factorization is never unique when $r > 1$, thus any posterior support identification for a matrix A given a factorization $Y = AB$ should fail.

6. CONCLUSIONS AND OPEN QUESTIONS

In this manuscript, a Dictionary-based Low-Rank Approximation framework has been proposed. It allows to constrain any factor in a LRA to live in the union of k-dimensional subspaces generated by subsets of columns of a given dictionary. DLRA is shown to be useful for various signal processing tasks such as image completion or image denoising. A contribution of this work is an Alternating Optimization algorithm (AO-DLRA) to compute candidate solutions to DLRA. Additionally, the subproblem of estimating the sparse codes of a factor in a LRA, coined Mixed Sparse Coding, is extensively discussed. A heuristic convex relaxation adapted from LASSO is shown to perform very well for solving MSC when compared to other modified sparse coding strategies, and along the way, several theoretical results regarding MSC are provided.

There are several research directions that stem from this work. First, the identifiability properties of DLRA have not been addressed here. It was shown in previous works [13] that dictionary-based matrix factorization is identifiable when

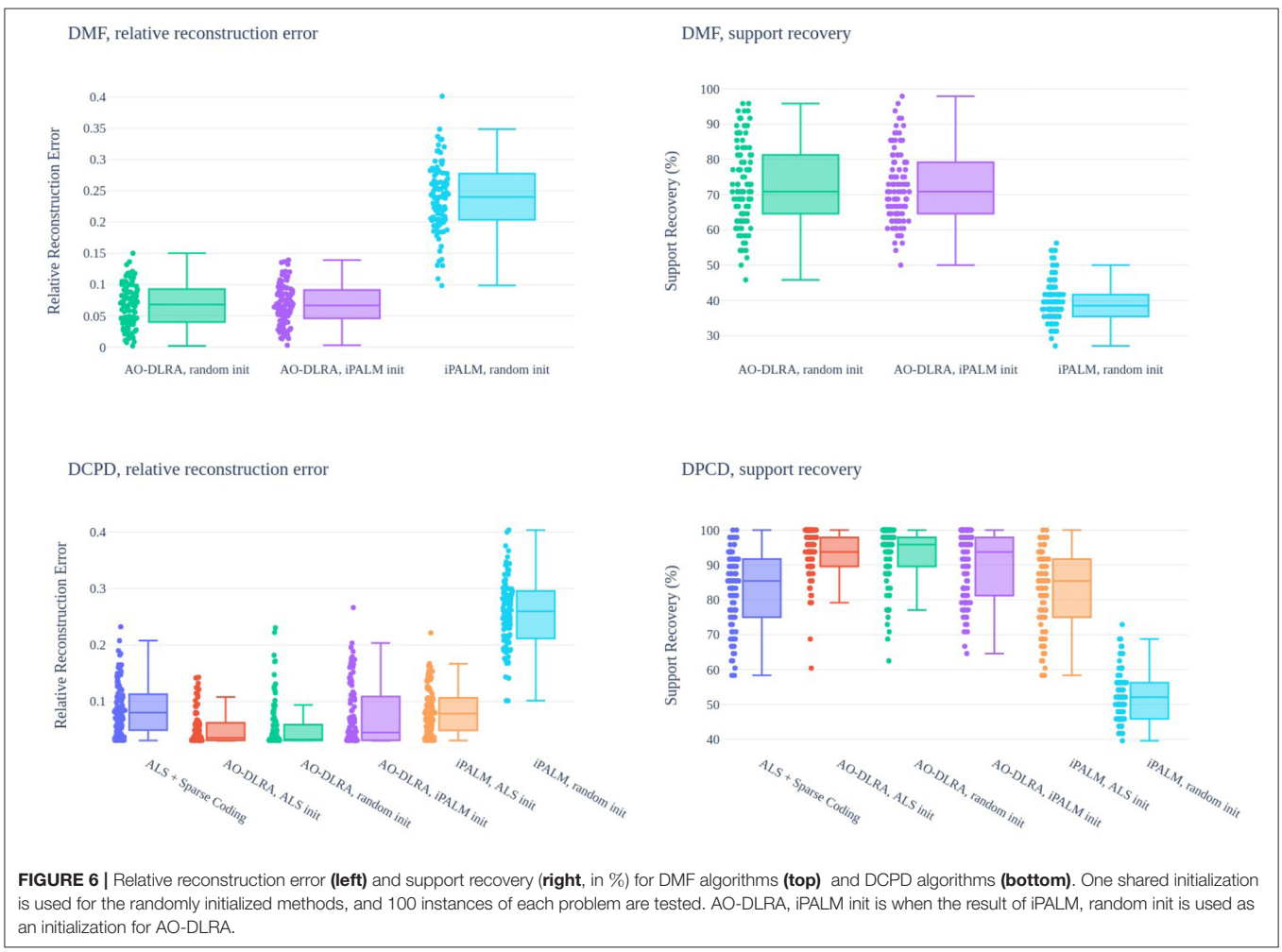


FIGURE 6 | Relative reconstruction error **(left)** and support recovery (**right**, in %) for DMF algorithms **(top)** and DCPD algorithms **(bottom)**. One shared initialization is used for the randomly initialized methods, and 100 instances of each problem are tested. AO-DLRA, iPALM init is when the result of iPALM, random init is used as an initialization for AO-DLRA.

sparsity is exactly one, but the general case is harder to analyse. The identifiability analysis is furthermore model dependent while this work aims at tackling the computation of any DLRA. Second, while the proposed AO-DLRA algorithm proved efficient in practice, its convergence properties are lacking. It is reasonably easy to design an AO algorithm with guaranties for DLRA, but I could not obtain an algorithm with convergence guarantees which performance matched the proposed AO-DLRA. Finally, the proposed DLRA model could be extended to a supervised setting, where D is trained in a similar fashion to Dictionary Learning [15]. This would mean computing a DLRA for several tensors with the same dictionary, a problem closely related to coupled matrix and tensor factorization with linearly coupled factors [79].

DATA AVAILABILITY STATEMENT

The scripts used to generate the synthetic dataset for this study can be found alongside the code in the online repositories https://github.com/cohenjer/mscode and https://github.com/cohenjer/dlra. Further inquiries can be directed to the corresponding author/s.

AUTHOR CONTRIBUTIONS

JC is the sole contributor to this work. He studied the theory around Mixed Sparse Coding and Dictionary-based low-rank approximations, implemented the methods, conducted the experiments, and wrote the manuscript.

FUNDING

This work was funded by ANR JCJC LoRAiA ANR-20-CE23-0010.

ACKNOWLEDGMENTS

The author thanks Rémi Gribonval for commenting an early version of this work.

SUPPLEMENTARY MATERIAL

The Supplementary Material for this article can be found online at: https://www.frontiersin.org/articles/10.3389/fams.2022.801650/full#supplementary-material

REFERENCES

1. Kolda TG, Bader BW. Tensor decompositions and applications. *SIAM Rev*. (2009) 51:455–500. doi: 10.1137/07070111X
2. Sidiropoulos ND, De Lathauwer L, Fu X, Huang K, Papalexakis EE, Faloutsos C. Tensor decomposition for signal processing and machine learning. *IEEE Trans Signal Process*. (2017) 65:3551–82. doi: 10.1109/TSP.2017.2690524
3. Gillis N. *Nonnegative Matrix Factorization*. Philadelphia: Society for Industrial and Applied Mathematics (2020). doi: 10.1137/1.9781611976410
4. Kruskal JB. Three-way arrays: rank and uniqueness of trilinear decompositions, with application to arithmetic complexity and statistics. *Linear Algeb Appl*. (1977) 18:95–138. doi: 10.1016/0024-3795(77)90069-6
5. Hoyer PO. Non-negative sparse coding. In: *IEEE Workshop on Neural Networks for Signal Processing*. Martigny (2002). p. 557–65.
6. Mørup M, Hansen LK, Arnfred SM. Algorithms for sparse nonnegative Tucker decompositions. *Neural Comput*. (2008) 20:2112–31. doi: 10.1162/neco.2008.11-06-407
7. Hennequin R, Badeau R, David B. Time-dependent parametric and harmonic templates in non-negative matrix factorization. In: *Proceedings of the 13th International Conference on Digital Audio Effects (DAFx)*. Graz (2010).
8. Timmerman ME, Kiers HA. Three-way component analysis with smoothness constraints. *Comput Stat Data Anal*. (2002) 40:447–70. doi: 10.1016/S0167-9473(02)00059-2
9. Fu X, Huang K, Sidiropoulos ND. On identifiability of nonnegative matrix factorization. *IEEE Signal Process Lett*. (2018) 25:328–32. doi: 10.1109/LSP.2018.2789405
10. Gillis N, Luce R. Robust near-separable nonnegative matrix factorization using linear optimization. *J Mach Learn Res*. (2014) 15:1249–80.
11. Silva VD, Lim LH. Tensor rank and the ill-posedness of the best low-rank approximation problem. *SIAM J Matrix Anal Appl*. (2008) 30:1084–127. doi: 10.1137/06066518X
12. Mohlenkamp MJ. The dynamics of swamps in the canonical tensor approximation problem. *SIAM J Appl Dyn Syst*. (2019) 18:1293–333. doi: 10.1137/18M1181389
13. Cohen JE, Gillis N. Dictionary-based tensor canonical polyadic decomposition. *IEEE Trans Signal Process*. (2018) 66:1876–89. doi: 10.1109/TSP.2017.2777393
14. Kjersti E, Sven OA, John HH. Method of optimal directions for frame design. *IEEE Int Conf Acoust Speech Signal Process Proc*. (1999) 5:2443–6.
15. Mairal J, Bach F, Ponce J, Sapiro G. Online learning for matrix factorization and sparse coding. *J Mach Learn Res*. (2010) 11:19–60.
16. Huang K, Sidiropoulos ND, Liavas AP. A flexible and efficient algorithmic framework for constrained matrix and tensor factorization. *IEEE Trans Signal Process*. (2016) 64:5052–65. doi: 10.1109/TSP.2016.2576427
17. Fu X, Vervliet N, De Lathauwer L, Huang K, Gillis N. Computing large-scale matrix and tensor decomposition with structured factors: a unified nonconvex optimization perspective. *IEEE Signal Process Mag*. (2020) 37:78–94. doi: 10.1109/MSP.2020.3003544
18. Aharon M, Elad M, Bruckstein AM. K-SVD and its non-negative variant for dictionary design. In: *Wavelets XI*. Vol. 5914. San Diego, CA: International Society for Optics and Photonics (2005). p. 591411. doi: 10.1117/12.613878
19. Xu Y. Alternating proximal gradient method for sparse nonnegative Tucker decomposition. *Math Programm Comput*. (2015) 7:39–70. doi: 10.1007/s12532-014-0074-y
20. Kolda TG, Hong D. Stochastic gradients for large-scale tensor decomposition. *SIAM J Math Data Sci*. (2020) 2:1066–95. doi: 10.1137/19M1266265
21. Marmin A, Goulart JHM, Févotte C. Joint majorization-minimization for nonnegative matrix factorization with the β-divergence. *arXiv [preprint]. arXiv:210615214* (2021).
22. Brewer J. Kronecker products and matrix calculus in system theory. *IEEE Trans Circuits Syst*. (1978) 25:772–81. doi: 10.1109/TCS.1978.1084534
23. Pearson K. On lines and planes of closest fit to systems of points in space. *Lond Edinburgh Dublin Philos Mag J Sci*. (1901) 2:559–72. doi: 10.1080/14786440109462720
24. Jolliffe I. *Principal Component Analysis*. Wiley Online Library (2002).
25. Comon P. Independent component analysis, a new concept? *Signal Process*. (1994) 36:287–314. doi: 10.1016/0165-1684(94)90029-9
26. Lee DD, Seung HS. Learning the parts of objects by non-negative matrix factorization. *Nature*. (1999) 401:788–91. doi: 10.1038/44565
27. Gribonval R, Lesage S. A survey of sparse component analysis for blind source separation: principles, perspectives, and new challenges. In: *ESANN'06 Proceedings-14th European Symposium on Artificial Neural Networks*. Bruges (2006). p. 323–30.

28. Donoho DL, Stodden V. When does non-negative matrix factorization give a correct decomposition into parts? In: *Advances in Neural Information Processing 16*. Vancouver (2003).
29. Cohen JE, Gillis N. Identifiability of complete dictionary learning. *SIAM J Math Data Sci*. (2019) 1:518–36. doi: 10.1137/18M1233339
30. Gillis N, Glineur F. Accelerated multiplicative updates and hierarchical ALS algorithms for nonnegative matrix factorization. *Neural Comput*. (2012) 24:1085–105. doi: 10.1162/NECO_a_00256
31. Natarajan B. Sparse approximate solutions to linear systems. *SIAM J Comput*. (1995) 24:227–34. doi: 10.1137/S0097539792240406
32. Vavasis SA. On the complexity of nonnegative matrix factorization. *SIAM J Optim*. (2010) 20:1364–77. doi: 10.1137/070709967
33. Bro R. PARAFAC, tutorial and applications. *Chemom Intel Lab Syst*. (1997) 38:149–71. doi: 10.1016/S0169-7439(97)00032-4
34. Becker H, Albera L, Comon P, Gribonval R, Wendling F, Merlet I. Brain source imaging: from sparse to tensor models. *IEEE Signal Proc Mag*. (2015) 32:100–12. doi: 10.1109/MSP.2015.2413711
35. Cohen JE, Cabral-Farias R, Comon P. Fast decomposition of large nonnegative tensors. *IEEE Signal Process Lett*. (2015) 22:862–6. doi: 10.1109/LSP.2014.2374838
36. Kossaifi J, Lipton ZC, Kolbeinsson A, Khanna A, Furlanello T, Anandkumar A. Tensor regression networks. *J Mach Learn Res*. (2020) 21:1–21.
37. Hitchcock FL. The expression of a tensor or a polyadic as a sum of products. *J Math Phys*. (1927) 6:164–89. doi: 10.1002/sapm192761164
38. Harshman RA. PARAFAC2: mathematical and technical notes. *UCLA Work Pap Phonet*. (1972) 22:122215.
39. Sidiropoulos ND, Bro R. On the uniqueness of multilinear decomposition of N-way arrays. *J Chemometr*. (2000) 14:229–39. doi: 10.1002/1099-128X(200005/06)14:3<229::AID-CEM587>3.0.CO;2-N
40. Domanov I, De Lathauwer L. On the uniqueness of the canonical polyadic decomposition of third-order tensors - part i: basic results and uniqueness of one factor matrix. *SIAM J Matrix Anal Appl*. (2013) 34:855–75. doi: 10.1137/120877234
41. Lim LH, Comon P. Nonnegative approximations of nonnegative tensors. *J Chemometr*. (2009) 23:432–41. doi: 10.1002/cem.1244
42. Roald M, Schenker C, Cohen J, Acar E. PARAFAC2 AO-ADMM: constraints in all modes. In: *EUSIPCO*. Dublin (2021). doi: 10.23919/EUSIPCO54536.2021.9615927
43. Tucker LR. Some mathematical notes on three-mode factor analysis. *Psychometrika*. (1966) 31:279–311. doi: 10.1007/BF02289464
44. Oseledets IV. Tensor-train decomposition. *SIAM J Sci Comput*. (2011) 33:2295–317. doi: 10.1137/090752286
45. Phan AH, Cichocki A. Extended HALS algorithm for nonnegative Tucker decomposition and its applications for multiway analysis and classification. *Neurocomputing*. (2011) 74:1956–69. doi: 10.1016/j.neucom.2010.06.031
46. Marmoret A, Cohen JE, Bertin N, Bimbot F. Uncovering audio patterns in music with nonnegative tucker decomposition for structural segmentation. In: *ISMIR 2020-21st International Society for Music Information Retrieval*. Montreal (2020).
47. Foucart S, Rauhut H. In: Birkhausen, editor. *A Mathematical Introduction to Compressive Sensing*. Applied and Numeric Harmonic Analysis. Birkhauser; Springer (2013). doi: 10.1007/978-0-8176-4948-7
48. Mallat SG, Zhang Z. Matching pursuits with time-frequency dictionaries. *IEEE Trans Signal Process*. (1993) 41:3397–415. doi: 10.1109/78.258082
49. Pati YC, Rezaiifar R, Krishnaprasad PS. Orthogonal matching pursuit: recursive function approximation with applications to wavelet decomposition. In: *1993 Conference Record of The 27th Asilomar Conference on Signals, Systems and Computers*. Pacific Grove, CA (1993). p. 40–4.
50. Tropp JA. Greed is good: algorithmic results for sparse approximation. *IEEE Trans Inform Theory*. (2004) 50:2231–42. doi: 10.1109/TIT.2004.834793
51. Blumensath T, Davies ME. Iterative thresholding for sparse approximations. *J Four Anal Appl*. (2008) 14:629–54. doi: 10.1007/s00041-008-9035-z
52. Parikh N, Boyd SP. Proximal algorithms. *Found Trends Optim*. (2014) 1:127–239. doi: 10.1561/2400000003
53. Tropp JA. Just relax: Convex programming methods for identifying sparse signals in noise. *IEEE Trans Inform Theory*. (2006) 52:1030–51. doi: 10.1109/TIT.2005.864420
54. Nesterov Y. A method of solving a convex programming problem with convergence rate O (1/k2). In: *Soviet Mathematics Doklady*. Vol. 27 (1983). p. 372–6.
55. Beck A, Teboulle M. A fast iterative shrinkage-thresholding algorithm for linear inverse problems. *SIAM J Imaging Sci*. (2009) 2:183–202. doi: 10.1137/080716542
56. Cotter SF, Rao BD, Engan K, Kreutz-Delgado K. Sparse solutions to linear inverse problems with multiple measurement vectors. *IEEE Trans Signal Process*. (2005) 53:2477–88. doi: 10.1109/TSP.2005.849172
57. Elhamifar E, Sapiro G, Vidal R. See all by looking at a few: sparse modeling for finding representative objects. In: *2012 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Providence (2012). p. 1600–7. doi: 10.1109/CVPR.2012.6247852
58. Iordache MD, Bioucas-Dias JM, Plaza A. Collaborative sparse regression for hyperspectral unmixing. *IEEE Trans Geosci Remote Sensing*. (2014) 52:341–54. doi: 10.1109/TGRS.2013.2240001
59. Cohen JE. About notations in multiway array processing. *arXiv [preprint]. arXiv:151101306*. (2015).
60. Traonmilin Y, Gribonval R. Stable recovery of low-dimensional cones in Hilbert spaces: one RIP to rule them all. *Appl Comput Harm Anal*. (2018) 45:170–205. doi: 10.1016/j.acha.2016.08.004
61. Tsiligkaridis T, Hero AO. Covariance estimation in high dimensions *via* Kronecker product expansions. *IEEE Trans Signal Process*. (2013) 61:5347–60. doi: 10.1109/TSP.2013.2279355
62. Golub GH, Loan CFV. *Matrix Computations*. Baltimore: The John Hopkins University Press (1989).
63. Donoho DL, Elad M. Optimally sparse representation in general (nonorthogonal) dictionaries via ℓ_1 minimization. *Proc Natl Acad Sci USA*. (2003) 100:2197–202. doi: 10.1073/pnas.0437847100
64. Bourguignon S, Ninin J, Carfantan H, Mongeau M. Exact sparse approximation problems via mixed-integer programming: formulations and computational performance. *IEEE Trans Signal Process*. (2015) 64:1405–19. doi: 10.1109/TSP.2015.2496367
65. Nadisic N, Vandaele A, Gillis N, Cohen JE. Exact sparse nonnegative least squares. In: *ICASSP*. Barcelona (2020). p. 5395–9. doi: 10.1109/ICASSP40776.2020.9053295
66. Tibshirani R. Regression shrinkage and selection via the lasso. *J R Stat Soc Ser B*. (1996) 58:267–88. doi: 10.1111/j.2517-6161.1996.tb02080.x
67. Attouch H, Bolte J, Svaiter BF. Convergence of descent methods for semi-algebraic and tame problems: proximal algorithms, forward-backward splitting, and regularized Gauss-Seidel methods. *Math Programm*. (2013) 137:91–129. doi: 10.1007/s10107-011-0484-9
68. Blumensath T, Davies ME. Iterative hard thresholding for compressed sensing. *Appl Comput Harm Anal*. (2009) 27:265–74. doi: 10.1016/j.acha.2009.04.002
69. Rubinstein R, Zibulevsky M, Elad M. Efficient implementation of the K-SVD algorithm using batch orthogonal matching pursuit. *CS Technion*. (2008) 40:1–15.
70. Bogdan M, van den Berg E, Sabatti C, Su W, Candès EJ. SLOPE-Adaptive variable selection via convex optimization. *Ann Appl Stat*. (2015) 9:1103–40. doi: 10.1214/15-AOAS842
71. Quattoni A, Carreras X, Collins M, Darrell T. An efficient projection for ℓ_1, ∞ regularization. In: *Proceedings of the 26th Annual International Conference on Machine Learning*. Montreal (2009). p. 857–64. doi: 10.1145/1553374.1553484
72. Bejar B, Dokmanic I, Vidal R. The fastest ℓ_1, ∞ prox in the west. *IEEE Trans Pattern Anal Mach Intell*. (2021). doi: 10.1109/TPAMI.2021.3059301
73. Cohen JE. Computing the proximal operator of the ℓ_1, 1 induced matrix norm. *arXiv [preprint]. arXiv:200506804*. (2020).
74. Osborne MR, Presnell B, Turlach BA. A new approach to variable selection in least squares problems. *IMA J Numer Anal*. (2000) 20:389–403. doi: 10.1093/imanum/20.3.389
75. Candés EJ, Recht B. Exact matrix completion via convex optimization. *Found Comput Math*. (2009) 9:717–72. doi: 10.1007/s10208-009-9045-5
76. Zhu F. Hyperspectral unmixing: ground truth labeling, datasets, benchmark performances and survey. *arXiv [preprint]. arXiv:170805125* (2017).
77. Zhuang L, Bioucas-Dias JM. Fast hyperspectral image denoising and inpainting based on low-rank and sparse representations. *IEEE J*

Select Top Appl Earth Observ Remote Sensing. (2018) 11:730–42. doi: 10.1109/JSTARS.2018.2796570
78. Bro R. *Multi-way analysis in the food industry: models, algorithms, and applications* (Ph.D.). University of Amsterdam, Amsterdam, Netherlands (1998).
79. Schenker C, Cohen JE, Acar E. A Flexible optimization framework for regularized matrix-tensor factorizations with linear couplings. *IEEE J Select Top Signal Process*. (2020) 15:506–21. doi: 10.1109/JSTSP.2020.3045848

Conflict of Interest: The author declares that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's Note: All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher.

frontiers in Applied Mathematics and Statistics

ORIGINAL RESEARCH
published: 17 March 2022
doi: 10.3389/fams.2022.832883

Block Row Kronecker-Structured Linear Systems With a Low-Rank Tensor Solution

Stijn Hendrikx[1,2]* *and Lieven De Lathauwer*[1,2]

[1] Dynamical Systems, Signal Processing and Data Analytics (STADIUS), Department of Electrical Engineering (ESAT), KU Leuven, Leuven, Belgium, [2] Group Science, Engineering and Technology, KU Leuven Kulak, Kortrijk, Belgium

Several problems in compressed sensing and randomized tensor decomposition can be formulated as a structured linear system with a constrained tensor as the solution. In particular, we consider block row Kronecker-structured linear systems with a low multilinear rank multilinear singular value decomposition, a low-rank canonical polyadic decomposition or a low tensor train rank tensor train constrained solution. In this paper, we provide algorithms that serve as tools for finding such solutions for a large, higher-order data tensor, given Kronecker-structured linear combinations of its entries. Consistent with the literature on compressed sensing, the number of linear combinations of entries needed to find a constrained solution is far smaller than the corresponding total number of entries in the original tensor. We derive conditions under which a multilinear singular value decomposition, canonical polyadic decomposition or tensor train solution can be retrieved from this type of structured linear systems and also derive the corresponding generic conditions. Finally, we validate our algorithms by comparing them to related randomized tensor decomposition algorithms and by reconstructing a hyperspectral image from compressed measurements.

Keywords: tensor, decomposition, compressed sensing (CS), randomized, Kronecker, linear system

OPEN ACCESS

Edited by:
Jiajia Li,
College of William & Mary,
United States

Reviewed by:
Mark Iwen,
Michigan State University,
United States
Martin Stoll,
Chemnitz University of Technology,
Germany

***Correspondence:**
Stijn Hendrikx
stijn.hendrikx@kuleuven.be

Specialty section:
This article was submitted to Mathematics of Computation and Data Science, a section of the journal Frontiers in Applied Mathematics and Statistics

Received: *10 December 2021*
Accepted: *14 February 2022*
Published: *17 March 2022*

Citation:
Hendrikx S and De Lathauwer L (2022) Block Row Kronecker-Structured Linear Systems With a Low-Rank Tensor Solution. Front. Appl. Math. Stat. 8:832883. doi: 10.3389/fams.2022.832883

1. INTRODUCTION

In a wide array of applications within signal processing, machine learning, and data analysis, sampling all entries of a dataset is infeasible. Datasets can be infeasibly large either because their dimensions are huge, like a matrix with millions of rows and columns, or because they are higher-order. In several cases, a relatively limited set of indirectly sampled datapoints, i.e., linear combinations $\mathbf{A}\,\mathbf{x} = \mathbf{b}$ of the datapoints $\mathbf{x}$, suffices for recovering an accurate approximation of the full dataset, making the problem tractable again. For example in compressed sensing [1, 2] and randomized tensor decomposition algorithms [3], random measurement matrices are used to compress the data $\mathbf{x}$. Directly sampling a subset of the datapoints can also be written in the format $\mathbf{A}\,\mathbf{x} = \mathbf{b}$, in which the measurement matrix a now consists of a subset of the rows of the identity matrix. Hence, problems such as incomplete tensor decomposition [4], non-uniform sampling [5] and cross-approximation [6] can also be formulated in this manner.

Retrieving the original data from such a linear system is generally only possible if it is overdetermined. This would mean that the number of indirectly sampled datapoints equals at least the total number of dataset entries, which is the opposite of what is needed in the compressed sensing (CS) setting. This requirement becomes especially restrictive for higher-order datasets.

"Higher-order" means that the dataset consists of more than two dimensions or modes, in which case the number of entries increases exponentially with the number of modes. This phenomenon is commonly known as the curse of dimensionality (CoD).

Real data often allows compact representations thanks to some intrinsic structure, such as the data being generated by an underlying lower-dimensional process [1, 7]. In this case, $\mathbf{x}$ can be well approximated by a sparsifying basis $\mathbf{\Phi}$ and a sparse coefficient vector $\boldsymbol{\theta}$, namely $\mathbf{x} \approx \mathbf{\Phi}\boldsymbol{\theta}$. The literature on compressed sensing (CS) shows that for a measurement matrix and sparsifying basis pair with low coherence, the linear system can be solved in the underdetermined case [1, 2]. This means that $\mathbf{x}$ can be recovered using far fewer compressed measurements $\mathbf{A}\mathbf{x}$ than the total number of entries in $\mathbf{x}$, breaking the CoD in the case of a higher-order dataset. An appropriate sparsifying basis is known a priori for some types of data, for example a wavelet basis for images, or can be obtained through dictionary learning [8]. If the sparsifying basis is known, then the measurement matrix can be chosen such that the coherence between the sparsifying basis and the measurement matrix is low. If no sparsifying basis for $\mathbf{x}$ is known a priori, one often chooses a random measurement matrix, as they are largely incoherent with any fixed basis [1].

In this paper, we exploit intrinsic structure that is common in real data by compactly approximating $\mathbf{x}$ using tensor decompositions, which in turn allows us to solve $\mathbf{A}\,\mathbf{x} = \mathbf{b}$ in the underdetermined case. This means that we will solve

$$\mathbf{A}\mathrm{vec}\,(\boldsymbol{\mathcal{X}}) = \mathbf{b} \quad \text{with} \quad \boldsymbol{\mathcal{X}} \text{ of low rank} \quad \text{and} \quad \mathrm{vec}\,(\boldsymbol{\mathcal{X}}) = \mathbf{x}. \quad (1)$$

Concretely, we will consider $\boldsymbol{\mathcal{X}}$ constrained to a multilinear singular value decomposition (MLSVD), a canonical polyadic decomposition (CPD) or a tensor train (TT) of low rank[1]. Note that at this point we have addressed the dimensionality issue only partially: on one hand, $\mathbf{x}$ is compactly modeled by a tensor decomposition; however, on the other hand, the number of columns of $\mathbf{A}$ remains equal to the total number of entries in $\mathbf{x}$ and thus still suffers from the CoD. Therefore, we will employ a block row Kronecker-structured (BRKS) measurement matrix $\mathbf{A}$. Efficient algebraic algorithms for solving this linear system will be obtained by combining this Kronecker structure with the low-rank constraints on $\mathbf{x}$ such that $\mathbf{A}$ and $\mathbf{x}$ do not need to be fully constructed. A standard approach for computing the CPD of a tensor is to first orthogonally compress this tensor, for example using the MLSVD, and then compute the CPD of the compressed tensor [9]. In this paper, we generalize this approach to the CS-setting.

Unstructured measurement matrices compress all modes of $\boldsymbol{\mathcal{X}}$ simultaneously. On the contrary, Kronecker-structured measurement matrices produce compressed versions of $\boldsymbol{\mathcal{X}}$ by compressing each mode individually, making them useful for higher-order datasets. Therefore, these Kronecker-structured measurement matrices are used in the CS-setting [10–12]. In Sidiropoulos and Kyrillidis [10] the CPD of a tensor is computed by first decomposing multiple compressed versions of $\boldsymbol{\mathcal{X}}$ and then retrieving the factor matrices of the full tensor under the assumption that their columns are sparse. There is some similarity between the Kronecker compressive sampling (KCS) approach [11] and ours, because it uses a Kronecker-structured measurement matrix and assumes that $\mathbf{x}$ is sparse in a Kronecker-structured basis. However, in KCS this basis is assumed to be known a priori, while in our approach it is estimated as well. In Kressner and Tobler [13], a low-rank approximation to the solution of a parametrized set of linear systems, which can be rewritten as a large linear system in which the coefficient matrix consists of a sum of Kronecker products, is computed. This approach is suited toward applications such as solving partial differential equations rather than CS, as the requirement that the smaller individual systems should be overdetermined makes it infeasible for the latter purpose. Additionally, this approach utilizes the hierarchical Tucker decomposition to constrain the solution, as opposed to the MLSVD, CPD and TT constraints in this paper.

Algorithms similar to the ones in this paper appear in the literature on randomized tensor decomposition (RTD) [3, 14–16]. The main difference is that the full tensor is available in such a randomized algorithm, while our algorithms can also be applied when only compressed measurements are available. In a randomized algorithm, the tensor is compressed in multiple modes to speed up further computations. In Zhou et al. [16] the factor matrices of an MLSVD are computed by randomly compressing the tensor. However, this compression is carried out simultaneously in multiple modes in a manner that is not Kronecker-structured. Also, the full tensor is needed to retrieve the core $\boldsymbol{\mathcal{S}}$, as opposed to only compressed measurements like in our approach. A similar randomized approach for computing the MLSVD is introduced in Che et al. [15], with the difference that the compression is carried out independently in different modes. An overview of RTD algorithms for computing an MLSVD is given in Ahmadi-Asl et al. [3, Section 5]. In Sidiropoulos et al. [12], a CPD is computed by decomposing multiple randomly compressed versions of the tensor in parallel and then combining the results. The algorithm in Yang et al. [17] improves upon this by replacing the dense random matrices with sketching matrices in the compression step to reduce the computational complexity. In Battaglino et al. [14], sketching matrices are used to speed up the least squares subproblems in the alternating least squares approach for computing the CPD. Multiplication with a sketching matrix is a Johnson-Lindenstrauss Transform (JLT), which transforms points in a high dimensional subspace to a lower dimensional subspace while preserving the distance between points up to a certain bound. In Jin et al. [18], it is proven that applying a JLT along each mode of a tensor is also a JLT. On the other hand, in cross approximation, CUR decompositions and pseudo-skeleton decompositions, a tensor is decomposed using a subset of directly sampled vectors along each mode [6, 19–21]. These subsets are determined on the basis of heuristics, for which algebraic results on the obtained quality of the approximation are available [22].

In the next part of this section, we introduce notations and definitions for further use in this paper. In Section 2, we propose

[1]In this context, rank pertains to the definition of rank that corresponds to the respective tensor decomposition.

an algorithm for computing an MLSVD from a BRKS linear system and derive conditions under which a solution can be found. In Section 3, we generalize the standard approach for computing a CPD, in which the tensor is first compressed using its MLSVD, to computing a CPD from a BRKS linear system. We also derive conditions under which the CPD can be retrieved. In Section 4, we compute a TT from a BRKS linear system and derive conditions under which the TT can be retrieved. Finally, in Section 5, we validate our algorithms by computing tensor decompositions in a randomized approach using synthetic data and by applying them to a hyperspectral imaging application.

1.1. Notations and Definitions

A scalar, vector, matrix and tensor are, respectively, denoted by $x, \mathbf{x}, \mathbf{X}$ and $\boldsymbol{\mathcal{X}}$. The dimensions of a tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ of order N are denoted by I_n for $n = 1, \ldots, N$. The rank of a matrix $\mathbf{X}$ is denoted by $\mathrm{r}(\mathbf{X})$. The identity matrix is denoted by $\mathbf{I}_I \in \mathbb{R}^{I \times I}$. A set and its complementary set are, respectively, denoted by $\mathcal{I}$ and $\mathcal{I}^c$. A matricization of a tensor is obtained by reshaping the tensor into a matrix and is denoted by $\mathbf{X}_{[\mathcal{I};\mathcal{I}^c]}$, with $\mathcal{I} \subset \{1, \ldots, N\}$. The sets $\mathcal{I}$ and $\mathcal{I}^c$, respectively, indicate which modes of the tensor are in the rows and columns of the matricization. See Kolda [23] for a more detailed, elementwise definition of a matricization. We use a shorthand notation for the matricization that contains a single mode in its rows, also known as the mode-n unfolding, namely

$$\mathbf{X}_{[n]} \coloneqq \mathbf{X}_{[n;1,\ldots,n-1,n+1,\ldots,N]}.$$

The mode-n, outer, Kronecker and Khatri–Rao product are denoted by $\cdot_n, \otimes, \bigotimes$ and $\odot$. The mixed product property of the Kronecker product is $(\mathbf{A} \otimes \mathbf{B})(\mathbf{C} \otimes \mathbf{D}) = (\mathbf{AC}) \otimes (\mathbf{BD})$, with $\mathbf{A} \in \mathbb{R}^{I \times J}$, $\mathbf{B} \in \mathbb{R}^{L \times M}$, $\mathbf{C} \in \mathbb{R}^{J \times K}$ and $\mathbf{D} \in \mathbb{R}^{M \times N}$. Similarly, the mixed product property of the Khatri–Rao product is $(\mathbf{A} \otimes \mathbf{B})(\mathbf{C} \odot \mathbf{D}) = (\mathbf{AC}) \odot (\mathbf{BD})$. The rank of the Kronecker product of matrices equals the product of the ranks of those matrices, i.e., $\mathrm{r}(\mathbf{A} \otimes \mathbf{B}) = \mathrm{r}(\mathbf{A})\,\mathrm{r}(\mathbf{B})$. A shorthand notation for a sequence of products is:

$$\bigotimes_{n=1}^{N} \mathbf{U}^{(n)} \coloneqq \mathbf{U}^{(1)} \otimes \cdots \otimes \mathbf{U}^{(N)}, \qquad \bigodot_{n=1}^{N} \mathbf{U}^{(n)} \coloneqq \mathbf{U}^{(1)} \odot \cdots \odot \mathbf{U}^{(N)}.$$

The multilinear singular value decomposition (MLSVD) decomposes a tensor as

$$\boldsymbol{\mathcal{X}} = \boldsymbol{\mathcal{S}} \cdot_1 \mathbf{U}^{(1)} \cdots \cdot_N \mathbf{U}^{(N)} =: \left[\!\!\left[\boldsymbol{\mathcal{S}}; \mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)} \right]\!\!\right],$$

with column-wise orthonormal factor matrices $\mathbf{U}^{(n)} \in \mathbb{R}^{I_n \times R_n}$ for $n = 1, \ldots, N$ and an all-orthogonal core tensor $\boldsymbol{\mathcal{S}} \in \mathbb{R}^{R_1 \times \cdots \times R_N}$ [24]. The tuple $(R_1, \ldots, R_N)$ is the multilinear rank of $\boldsymbol{\mathcal{X}}$, in which $R_n = \mathrm{r}\left(\mathbf{X}_{[n]}\right)$ for $n = 1, \ldots, N$. In vectorized form, this decomposition equals

$$\mathrm{vec}\left(\boldsymbol{\mathcal{X}}\right) = \left(\bigotimes_{n=1}^{N} \mathbf{U}^{(n)} \right) \mathrm{vec}\left(\boldsymbol{\mathcal{S}}\right). \tag{2}$$

The canonical polyadic decomposition (CPD) decomposes a tensor as a minimal sum of rank-1 tensors

$$\boldsymbol{\mathcal{X}} = \sum_{r=1}^{R} \mathbf{u}_r^{(1)} \otimes \cdots \otimes \mathbf{u}_r^{(N)} =: \left[\!\!\left[\mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)} \right]\!\!\right],$$

with factor matrices $\mathbf{U}^{(n)} \in \mathbb{R}^{I_n \times R}$ for $n = 1, \ldots, N$. The number of rank-1 tensors R equals the rank of $\boldsymbol{\mathcal{X}}$. In vectorized form, the CPD equals

$$\mathrm{vec}\left(\boldsymbol{\mathcal{X}}\right) = \left(\bigodot_{n=1}^{N} \mathbf{U}^{(n)} \right) \mathbf{1}_R$$

with $\mathbf{1}_R$ a vector of length R containing all ones. The tensor train (TT) factorizes each entry of $\boldsymbol{\mathcal{X}}$ as a sequence of matrix products

$$x_{i_1 \cdots i_N} = \mathbf{G}^{(1)}_{:\,i_1\,:} \cdots \mathbf{G}^{(N)}_{:\,i_N\,:},$$

with $\mathbf{G}^{(n)}_{:\,i_n\,:} \in \mathbb{R}^{R_{n-1} \times R_n}$ for $n = 1, \ldots, N$ and $R_0 = R_N = 1$ [25]. An index that has not been fixed is indicated by :, meaning that $\mathbf{G}^{(n)}_{:\,i\,:}$ is the ith mode-2 slice of a third-order tensor. The cores of the TT are obtained by stacking $\mathbf{G}^{(n)}_{:\,i_n\,:}$ for $n = 2, \ldots, N-1$ and for $n = 1, N$ into third-order tensors and matrices $\boldsymbol{\mathcal{G}}^{(n)} \in \mathbb{R}^{R_{n-1} \times I_n \times R_n}$, respectively. The tuple $(R_0, \ldots, R_N)$ is the TT-rank of $\boldsymbol{\mathcal{X}}$. We use

$$\boldsymbol{\mathcal{X}} = \langle\!\langle \boldsymbol{\mathcal{G}}^{(1)}, \ldots, \boldsymbol{\mathcal{G}}^{(N)} \rangle\!\rangle$$

as a shorthand notation for the TT.

2. COMPUTING AN MLSVD FROM A BRKS LINEAR SYSTEM

Using a BRKS linear system avoids the need for constructing and storing the full measurement matrix $\mathbf{A}$ and results in efficient algorithms for retrieving a low-rank constrained $\mathbf{x}$. With this structure, Equation (1) becomes

$$\begin{bmatrix} \bigotimes_{n=1}^{N} \mathbf{A}^{(1,n)} \\ \bigotimes_{n=1}^{N} \mathbf{A}^{(2,n)} \\ \vdots \\ \bigotimes_{n=1}^{N} \mathbf{A}^{(M,n)} \end{bmatrix} \mathrm{vec}\left(\boldsymbol{\mathcal{X}}\right) = \begin{bmatrix} \mathbf{b}^{(1)} \\ \mathbf{b}^{(2)} \\ \vdots \\ \mathbf{b}^{(M)} \end{bmatrix}, \tag{3}$$

with generating matrices $\mathbf{A}^{(m,n)} \in \mathbb{R}^{P_{mn} \times I_n}$ and compressed measurements $\mathbf{b}^{(m)} \in \mathbb{R}^{\prod_{n=1}^{N} P_{mn}}$ for $m = 1, \ldots, M$ and $n = 1, \ldots, N$. This type of linear system appears, for instance, in RTD and hyperspectral imaging, as illustrated in Section 5. Each block row of this linear system corresponds to a linear subsystem that produces a compressed version of $\boldsymbol{\mathcal{X}}$. This can be seen by tensorizing the mth block row as

$$\boldsymbol{\mathcal{B}}^{(m)} = \boldsymbol{\mathcal{X}} \cdot_1 \mathbf{A}^{(m,1)} \cdots \cdot_N \mathbf{A}^{(m,N)}, \tag{4}$$

in which $\boldsymbol{\mathcal{B}}^{(m)}$ is $\mathbf{b}^{(m)}$ reshaped into a tensor of dimensions $P_{m1} \times \cdots \times P_{mN}$. Each mode of $\boldsymbol{\mathcal{B}}^{(m)}$ has been compressed by mode-n multiplication with $\mathbf{A}^{(m,n)}$ for $n = 1, \ldots, N$.

If $\mathcal{X}$ is approximately of low multilinear rank $(R_1, \ldots, R_N)$, reflecting some inherent structure, then the vectorized MLSVD in Equation (2) can be substituted into Equation (3):

$$\mathbf{A}\left(\bigotimes_{n=1}^{N}\mathbf{U}^{(n)}\right)\mathrm{vec}\,(\mathcal{S}) = \mathbf{b}. \tag{5}$$

Note that the factor matrices would be in the reverse order when vectorizing conventionally, meaning $\otimes_{n=N}^{1}\mathbf{U}^{(n)}$. However, by vectorizing like in Equation (5), the index n of the generating matrices $\mathbf{A}^{(m,n)}$ for $m = 1, \ldots, M$ and $n = 1, \ldots, N$ corresponds nicely to the mode it operates on.

In order to solve the linear system in Equation (5) in the underdetermined case, it will not be directly solved for $\left(\otimes_{n=1}^{N}\mathbf{U}^{(n)}\right)\mathrm{vec}\,(\mathcal{S})$. Instead, the factor matrices $\mathbf{U}^{(n)}$ for $n = 1, \ldots, N$ will be retrieved individually using the linear subsystems

$$\left(\bigotimes_{n=1}^{N}\mathbf{A}^{(m,n)}\right)\left(\bigotimes_{n=1}^{N}\mathbf{U}^{(n)}\right)\mathrm{vec}\,(\mathcal{S}) = \mathbf{b}^{(m)} \quad \text{for } m = 1, \ldots, M \tag{6}$$

in Equation (5). If the BRKS linear system consists of N linear subsystems, then all factor matrices can be computed. Therefore, we assume from this point on that $M = N$. The case where $M < N$ is of use when not all factor matrices need to be retrieved. Next, the core tensor will be retrieved using the computed factor matrices. The linear system in Equation (6) is similar to the problem that is solved in KCS, namely $\otimes_{n=1}^{N}\mathbf{A}^{(m,n)}$, $\otimes_{n=1}^{N}\mathbf{U}^{(n)}$ and $\mathrm{vec}\,(\mathcal{S})$, respectively, correspond to the Kronecker-structured measurement matrix, the Kronecker-structured sparsifying basis and the sparse coefficients. In CS, choosing a good basis that sparsifies the data [8] and determining the sparse coefficients of the data in this basis [1, 2] are separate problems. In this paper, both problems are solved simultaneously using one BRKS linear system. Furthermore, unlike in the CS-setting, the coefficients $\mathrm{vec}\,(\mathcal{S})$ are not necessarily sparse. In the remainder of this section, we discuss the methods for retrieving the factor matrices and the core tensor.

2.1. Computing the Factor Matrices

The mth linear subsystem in Equation (6) can be simplified to

$$\left(\bigotimes_{n=1}^{N}\left(\mathbf{A}^{(m,n)}\mathbf{U}^{(n)}\right)\right)\mathrm{vec}\,(\mathcal{S}) = \mathbf{b}^{(m)} \quad \text{for } m = 1, \ldots, N \tag{7}$$

using the mixed product property of the Kronecker product. This corresponds to a vectorized MLSVD with factor matrices $\mathbf{A}^{(m,n)}\mathbf{U}^{(n)}$ for $n = 1, \ldots, N$ and core $\mathcal{S}$. Rearranging Equation (7) as the mode-m matrix unfolding of this MLSVD

$$\begin{aligned}\mathbf{A}^{(m,m)}\mathbf{U}^{(m)}\mathbf{S}^{(m)} &= \mathbf{B}_{[m]}^{(m)} \quad \text{with} \\ \mathbf{S}^{(m)} &= \left(\prod_{n\neq m}\mathcal{S}\cdot_n\left(\mathbf{A}^{(m,n)}\mathbf{U}^{(n)}\right)\right)_{[m]} \quad \text{for } m = 1, \ldots, N \\ &= \mathbf{S}_{[m]}\left(\bigotimes_{n\neq m}\left(\mathbf{A}^{(m,n)}\mathbf{U}^{(n)}\right)\right)^{\mathrm{T}},\end{aligned} \tag{8}$$

shows that solving the linear system

$$\mathbf{A}^{(m,m)}\mathbf{F}^{(m)} = \mathbf{B}_{[m]}^{(m)} \quad \text{with} \quad \mathbf{F}^{(m)} = \mathbf{U}^{(m)}\mathbf{S}^{(m)} \quad \text{for } m = 1, \ldots, N \tag{9}$$

for the unknown matrix $\mathbf{F}^{(m)}$ yields linear combinations of the columns of $\mathbf{U}^{(m)}$. Therefore, the dominant column space of $\mathbf{F}^{(m)}$ is the same as the subspace spanned by the columns of $\mathbf{U}^{(m)}$ if $\mathbf{S}^{(m)}$ is of full column rank. This means that we can find the column space of the mth factor matrix by computing an orthonormal basis that spans the dominant column space of $\mathbf{F}^{(m)}$. In applications that allow the generating matrices to be chosen, setting $\mathbf{A}^{(m,m)} = \mathbf{I}_{I_m}$ for $m = 1, \ldots, N$ avoids the need for solving this linear system altogether. In this case, $\mathcal{B}^{(m)}$ in Equation (4) equals $\mathcal{X}$ multiplied in every mode except the mth with a generating matrix. This situation is similar to RTD methods for computing an MLSVD, where the tensor is compressed in all modes except the mth in order to retrieve $\mathbf{U}^{(m)}$ [15, 16]. Generically, the dimensions P_{mn}, for $m, n = 1, \ldots, N$ and $n \neq m$, of the generating matrices can be chosen much smaller than the rank R_n of the mode-n unfolding of $\mathcal{X}$, as we will further explain in Section 2.4. As a first indication, note that for the factorization in the right-hand side of Equation (9), we expect $\prod_{n\neq m}P_{mn} = R_m$ to be sufficient, allowing $P_{mn} \ll R_m$ for $m, n = 1, \ldots, N$ and $n \neq m$.

2.2. Computing the Core Tensor

In some applications, not only the factor matrices $\mathbf{U}^{(n)}$ for $n = 1, \ldots, N$ but also the core $\mathcal{S}$ are needed. At first sight, it looks like a data efficient way to compute the core is reusing the linear combinations with which the column space of the factor matrices has been computed. The core tensor that corresponds to the retrieved column spaces of the factor matrices is then obtained by solving the linear system in Equation (5) for $\mathrm{vec}\,(\mathcal{S})$, with the column space of $\mathbf{U}^{(n)}$ already available for $n = 1, \ldots, N$, which is

$$\begin{bmatrix}\otimes_{n=1}^{N}\left(\mathbf{A}^{(1,n)}\mathbf{U}^{(n)}\right)\\ \otimes_{n=1}^{N}\left(\mathbf{A}^{(2,n)}\mathbf{U}^{(n)}\right)\\ \vdots\\ \otimes_{n=1}^{N}\left(\mathbf{A}^{(N,n)}\mathbf{U}^{(n)}\right)\end{bmatrix}\mathrm{vec}\,(\mathcal{S}) = \mathbf{b}. \tag{10}$$

The core can be retrieved if the coefficient matrix of this system is of full column rank. However, the requirement that this coefficient matrix must be of full column rank imposes much more severe constraints on the dimensions P_{mn} than the constraints for computing the factor matrices in Section 2.4. Under these harder constraints, the right hand sides $\mathbf{b}^{(n)}$ would have to contain far more compressed measurements than actually needed to retrieve the nth factor matrix $\mathbf{U}^{(n)}$ for $n = 1, \ldots, N$. The reason is that in the coefficient matrix in Equation (10) there are many linear dependencies between the rows, because the same factor matrices $\mathbf{U}^{(n)}$ appear in each block row. Therefore, it is not practical to compute the core by solving the system in Equation (10) for $\mathrm{vec}\,(\mathcal{S})$.

Instead, the core can be computed by solving a separate linear system, independent from the N block rows of the BRKS system

used for computing the factor matrices, for vec $(\mathcal{S})$, namely

$$\left(\bigotimes_{n=1}^{N}\left(\mathbf{C}^{(n)}\mathbf{U}^{(n)}\right)\right)\text{vec}\,(\mathcal{S}) = \mathbf{d} \tag{11}$$

with extra generating matrices $\mathbf{C}^{(n)} \in \mathbb{R}^{Q_n \times I_n}$ for $n = 1, \ldots, N$ and compressed measurements $\mathbf{d} \in \mathbb{R}^{\prod_{n=1}^{N} Q_n}$. This approach allows us to choose the dimensions Q_n for $n = 1, \ldots, N$ such that the core can be retrieved, without the need to increase the dimensions P_{mn} for $m, n = 1, \ldots, N$. Additionally, this additional system can be solved efficiently by subsequently solving smaller linear systems

$$\left(\mathbf{C}^{(n)}\mathbf{U}^{(n)}\right)\tilde{\mathbf{S}}_{\text{new}}^{(n)} = \tilde{\mathbf{S}}^{(n)} \quad \text{for } n = 1, \ldots, N \tag{12}$$

for $\tilde{\mathbf{S}}_{\text{new}}^{(n)}$. Here, $\tilde{\mathbf{S}}^{(1)} = \mathbf{D}_{[1]}$, in which $\mathbf{D}_{[1]}$ is the mode-1 matrix unfolding of the tensorization $\mathcal{D} \in \mathbb{R}^{Q_1 \times \cdots \times Q_N}$ of $\mathbf{d}$, and $\tilde{\mathbf{S}}^{(n)} = \text{reshape}\left(\tilde{\mathbf{S}}_{\text{new}}^{(n-1)}, [Q_n, \prod_{i=1}^{n-1} R_i \prod_{i=n+1}^{N} Q_i]\right)$ for $n = 2, \ldots, N$. After solving all N systems, $\tilde{\mathbf{S}}_{\text{new}}^{(N)}$ is the mode-N matrix unfolding of $\mathcal{S}$. These linear systems, with coefficient matrices of size $Q_n \times R_n$ for $n = 1, \ldots, N$, are much smaller than the linear system in Equation (10), consisting of all compressed measurements used to estimate the factor matrices, with a coefficient matrix of size $\sum_{m=1}^{N}\left(\prod_{n=1}^{N} P_{mn}\right) \times \prod_{n=1}^{N} R_n$. This coefficient matrix is so large because of the Kronecker products. In Section 2.4, we show that generically the core can be retrieved if the dimensions Q_n are greater than or equal to R_n for $n = 1, \ldots, N$. Therefore, solving the subsequent systems in Equation (12) is a computationally much cheaper approach for computing $\mathcal{S}$ than solving the system in Equation (10) is. This additional linear system can be added to the BRKS linear system in Equation (3) by renaming $\mathbf{C}^{(n)}$ for $n = 1, \ldots, N$ and $\mathbf{d}$ to $\mathbf{A}^{(N+1,n)}$ and $\mathbf{b}^{(N+1)}$, respectively. The complete BRKS linear system now consists of $N + 1$ block rows with generating matrices $\mathbf{A}^{(m,n)} \in \mathbb{R}^{P_{mn} \times I_n}$ and compressed measurements $\mathbf{b}^{(m)} \in \mathbb{R}^{\prod_{n=1}^{N} P_{mn}}$ for $m = 1, \ldots, N + 1$ and for $n = 1, \ldots, N$.

If we estimate the core tensor using a separate linear system, we do not use all available compressed measurements, since we disregard the first N block rows of the BRKS linear system. This can be resolved by solving the full BRKS linear system for vec $(\mathcal{S})$ with a numerical algorithm such as conjugate gradients, using the matrix-vector product

$$\begin{bmatrix} \otimes_{n=1}^{N}\left(\mathbf{A}^{(1,n)}\mathbf{U}^{(n)}\right) \\ \vdots \\ \otimes_{n=1}^{N}\left(\mathbf{A}^{(N+1,n)}\mathbf{U}^{(n)}\right) \end{bmatrix} \text{vec}\,(\mathcal{S}),$$

which can be computed efficiently by exploiting the block rowwise Kronecker structure. Alternatively, all block rows of the system can also be used if the full BRKS system is solved for a sparse core $\mathcal{S}$, which we will illustrate in an experiment in Section 5.

Algorithm 1: MLSVD from a BRKS linear system (`lsmlsvd_brks`).

Input: (CS-setting) $\mathbf{A}^{(m,n)}$, $\mathbf{b}^{(m)}$ for $m = 1, \ldots, N+1$; $n = 1, \ldots, N$
Input: (RTD-setting) $\mathbf{A}^{(m,n)}$ for $m = 1, \ldots, N+1$; $n = 1, \ldots, N$ and $\mathcal{X}$
Output: $\mathcal{S}$, $\mathbf{U}^{(m)}$ for $m = 1, \ldots, N$
begin
  if *RTD-setting* **then**
    Compute $\mathbf{b}^{(m)}$ for $m = 1, \ldots, N$ using Equation (3)
  `/* Estimate factor matrices */`
  for $m = 1$ **to** N **do**
    Solve $\mathbf{A}^{(m,m)}\mathbf{F}^{(m)} = \mathbf{B}_{[m]}^{(m)}$ for $\mathbf{F}^{(m)}$
    Find $\mathbf{U}^{(m)}$ by computing an orthonormal basis that spans the dominant column space of $\mathbf{F}^{(m)}$
  `/* Estimate core tensor */`
  $\tilde{\mathbf{S}} = \text{reshape}\left(\mathbf{b}^{(N+1)}, [P_{N+1,1}, \prod_{i=2}^{N} P_{N+1,i}]\right)$
  for $n = 1$ **to** N **do**
    Solve $\left(\mathbf{A}^{(N+1,n)}\mathbf{U}^{(n)}\right)\tilde{\mathbf{S}}_{\text{new}} = \tilde{\mathbf{S}}$ for $\tilde{\mathbf{S}}_{\text{new}}$
    $\tilde{\mathbf{S}} = \text{reshape}\left(\tilde{\mathbf{S}}_{\text{new}}, [P_{N+1,n}, \prod_{i=1}^{n-1} R_i \prod_{i=n+1}^{N} P_{N+1,i}]\right)$

2.3. Algorithm

By combining the steps in Sections 2.1 and 2.2, we obtain **Algorithm 1** for computing the MLSVD of $\mathcal{X}$ from a BRKS linear system. In the outlined version of the algorithm, the additional linear system in Equation (11) is used to estimate the core tensor. The computation of each factor matrix depends only on one compressed tensor $\mathcal{B}^{(m)}$ for $m = 1, \ldots, N$, allowing the different factor matrices to be computed in parallel. In the RTD-setting, the full tensor is available and is then randomly compressed in order to efficiently compute a tensor decomposition. On the other hand, compressed measurements can be obtained directly in the data acquisition stage in the CS-setting. To accommodate for both settings, **Algorithm 1** accepts either the generating matrices and the compressed measurements or the generating matrices and the full tensor as inputs.

2.4. Conditions for MLSVD Retrieval

In this section, we derive the conditions in Theorem 1, under which the retrieval of the MLSVD of $\mathcal{X}$ from a BRKS system is guaranteed. Since the factor matrices and the core are intrinsic to the data $\mathcal{X}$, the conditions in this section are to be seen as constraints on the generating matrices.

Theorem 1. *Consider a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ of multilinear rank $(R_1, \ldots, R_N)$, admitting an MLSVD $[\![\mathcal{S}; \mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}]\!]$ with factor matrices $\mathbf{U}^{(n)} \in \mathbb{R}^{I_n \times R_n}$ for $n = 1, \ldots, N$ and core tensor $\mathcal{S} \in \mathbb{R}^{R_1 \times \cdots \times R_N}$. Given linear combinations b of vec $(\mathcal{X})$ obtained from a BRKS linear system with generating matrices $\mathbf{A}^{(m,n)} \in \mathbb{R}^{P_{mn} \times I_n}$ for $m = 1, \ldots, N+1$ and $n = 1, \ldots, N$, the factor matrices $\mathbf{U}^{(n)}$ for $n = 1, \ldots, N$ can be retrieved if and*

only if

$$1) \quad \mathrm{r}\left(\mathbf{A}^{(m,m)}\right) = I_m \quad \text{for } m = 1, \ldots, N;$$

$$2) \quad \prod_{n \neq m}^{N} \mathrm{r}\left(\mathbf{A}^{(m,n)}\mathbf{U}^{(n)}\right) \geq R_m \quad \text{for } m = 1, \ldots, N.$$

The core $\boldsymbol{\mathcal{S}}$ *can be retrieved if and only if*

$$3) \prod_{n=1}^{N} \mathrm{r}\left(\mathbf{A}^{(N+1,n)}\mathbf{U}^{(n)}\right) = \prod_{n=1}^{N} R_n.$$

Proof: Condition 1): The linear system in Equation (9) can be uniquely solved for $\mathbf{F}^{(m)}$ if and only if $\mathbf{A}^{(m,m)}$ is of full column rank I_m for $m = 1, \ldots, N$.
Condition 2): An orthonormal basis of dimension R_m for the column space of $\mathbf{F}^{(m)}$ can be retrieved if and only if

$$\begin{aligned}\mathrm{r}\left(\mathbf{U}^{(m)}\mathbf{S}^{(m)}\right) &= \mathrm{r}\left(\mathbf{U}^{(m)}\mathbf{S}_{[m]}\left(\bigotimes_{n \neq m}\left(\mathbf{A}^{(m,n)}\mathbf{U}^{(n)}\right)\right)\right) \\ &= R_m \quad \text{for } m = 1, \ldots, N\end{aligned}$$

holds. Since $\mathrm{r}\left(\mathbf{U}^{(m)}\right) = \mathrm{r}\left(\boldsymbol{\mathcal{S}}_{[m]}\right) = R_m$, which follows from the definition of the MLSVD, this condition reduces to

$$\begin{aligned}\mathrm{r}\left(\bigotimes_{n \neq m}\left(\mathbf{A}^{(m,n)}\mathbf{U}^{(n)}\right)\right) &= \prod_{n \neq m}^{N} \mathrm{r}\left(\mathbf{A}^{(m,n)}\mathbf{U}^{(n)}\right) \\ &\geq R_m \quad \text{for } m = 1, \ldots, N.\end{aligned}$$

Condition 3): The linear system in Equation (11) can be uniquely solved for $\mathrm{vec}\left(\boldsymbol{\mathcal{S}}\right)$ if and only if the coefficient matrix $\otimes_{n=1}^{N}\left(\mathbf{A}^{(N+1,n)}\mathbf{U}^{(n)}\right)$ is of full column rank:

$$\mathrm{r}\left(\bigotimes_{n=1}^{N} \mathbf{A}^{(N+1,n)}\mathbf{U}^{(n)}\right) = \prod_{n=1}^{N} \mathrm{r}\left(\mathbf{A}^{(N+1,n)}\mathbf{U}^{(n)}\right) = \prod_{n=1}^{N} R_n.$$

The conditions in Theorem 1 are satisfied in the generic case, in which the generating matrices are sampled from a continuous probability distribution and thus are of full rank with probability 1, if and only if the conditions in Theorem 2 hold. The conditions in the latter theorem allow us to determine the dimensions P_{mn} for $m = 1, \ldots, N+1$ and $n = 1, \ldots, N$ of the generating matrices such that generically the MLSVD of $\boldsymbol{\mathcal{X}}$ can be retrieved.

Theorem 2. *With generic generating matrices* $\mathbf{A}^{(m,n)}$ for $m = 1, \ldots, N+1$ *and* $n = 1, \ldots, N$, *the conditions in Theorem 1 hold if and only if*

$$1) \quad P_{mm} \geq I_m \quad \text{for } m = 1, \ldots, N;$$

$$2) \quad \prod_{n \neq m} \min\left(P_{mn}, R_n\right) \geq R_m \quad \text{for } m = 1, \ldots, N;$$

$$3) \quad P_{N+1,n} \geq R_n \quad \text{for } n = 1, \ldots, N.$$

Proof: The proof of these conditions follows from the conditions in Theorem 1 and the properties of generic matrices.
Condition 1): Since a generic matrix is of full rank, $\mathrm{r}\left(\mathbf{A}^{(m,m)}\right) = \min(P_{mm}, I_m)$ for $m = 1, \ldots, N$.
Condition 2): The matrix product $\mathbf{A}^{(m,n)}\mathbf{U}^{(n)}$ is of full rank $\min\left(P_{mn}, R_n\right)$ with probability 1 for a generic matrix $\mathbf{A}^{(m,n)}$ and $\mathbf{U}^{(n)}$ of full column rank (the latter following from the definition of an MLSVD) with $P_{mn}, R_n \leq I_n$ for $m = 1, \ldots, N+1$ and $n = 1, \ldots, N$ [10, Lemma 1].
Condition 3): Similar to the proof of condition 2), generically $\mathrm{r}\left(\mathbf{A}^{(N+1,n)}\mathbf{U}^{(n)}\right) = \min\left(P_{N+1,n}, R_n\right)$ for $n = 1, \ldots, N$ according to Sidiropoulos and Kyrillidis [10, Lemma 1]. Condition 3) in Theorem 1 then reduces to $\prod_{n=1}^{N} \min(P_{N+1,n}, R_n) \geq \prod_{n=1}^{N} R_n$, which holds if and only if condition 3) in this theorem is satisfied.

In practice, the dimensions P_{mn} can easily be chosen such that the second generic condition is satisfied with $P_{mn} < R_n$ for $m, n = 1, \ldots, N$ and $n \neq m$. The second condition then reduces to $\prod_{n \neq m}^{N} P_{mn} \geq R_m$ for $m = 1, \ldots, N$. Assuming that P_{mn} for $m = 1, \ldots, N$ and $n \neq m$ are approximately equally large, we obtain the condition $P_{mn} \geq \sqrt[N-1]{R_m}$, meaning that the generating matrices $\mathbf{A}^{(m,n)} \in \mathbb{R}^{P_{mn} \times I_n}$ for $m, n = 1, \ldots, N$ and $n \neq m$ can be chosen as fat matrices. Therefore, the mth block row of Equation (3) compresses all modes of $\boldsymbol{\mathcal{X}}$ except the mth, as illustrated in **Figure 1**, similar to an RTD algorithm for the MLSVD. Additionally, this implies that $\mathbf{B}_{[m]}^{(m)} \in \mathbb{R}^{P_{mm} \times \prod_{n \neq m} P_{mn}}$ holds at least R_m columns, which is indeed the minimum required number for estimating the R_m-dimensional mode-m subspace of $\boldsymbol{\mathcal{X}}$. **Algorithm 1** uses an oversampling factor $q \geq 1$ such that this matrix holds more than the minimum required number of columns, namely $\prod_{n \neq m} P_{mn} = qR_m$ columns for $m = 1, \ldots, N$. These additional compressed measurements allow a better estimation of the factor matrices if the data is noisy.

2.5. Noisy Data

In applications, noise can be present on the compressed measurements and/or on the entries of the tensor. In the former

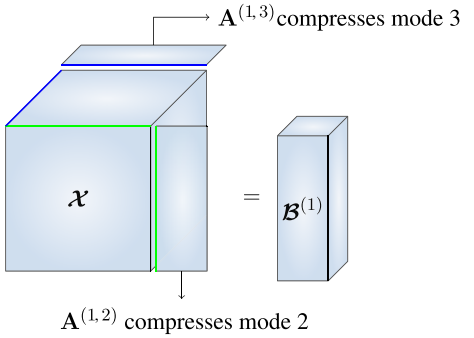


FIGURE 1 | Similar to the RTD algorithm in Che et al. [15], we recover the first MLSVD factor matrix from $\boldsymbol{\mathcal{B}}^{(1)}$, which is a version of $\boldsymbol{\mathcal{X}}$ that is compressed in every mode except the first.

case, the linear system becomes $\mathbf{A}\text{vec}(\mathcal{X}) = \mathbf{b} + \mathbf{n}$, in which the noise vector $\mathbf{n}$ is partitioned into subvectors $\mathbf{n}^{(m)}$ for $m = 1, \ldots, N$, consistent with the partitioning of $\mathbf{b}$ in Equation (3). The linear system in Equation (8), with a noise term $\mathbf{n}$, becomes

$$\mathbf{A}^{(m,m)}\mathbf{F}^{(m)} = \mathbf{B}^{(m)}_{[m]} + \mathbf{N}^{(m)}_{[m]} \quad \text{for } m = 1, \ldots, N,$$

in which $\mathcal{N}^{(m)} \in \mathbb{R}^{P_{m1} \times \cdots \times P_{mN}}$ is the tensor representation of $\mathbf{n}^{(m)}$ and $\mathbf{N}^{(m)}_{[m]}$ its mode-m matrix unfolding. The rank of the matrix, obtained by solving this subsystem for $\mathbf{F}^{(m)}$, generically equals $\min\left(I_m, \prod_{n \neq m}^{N} P_{mn}\right)$ due to the presence of the noise and thus exceeds R_m if $q > 1$ for $m = 1, \ldots, N$. If the signal-to-noise ratio $\frac{||\mathbf{b}||_2}{||\mathbf{n}||_2}$ is sufficiently high, then the dominant column space of $\mathbf{F}^{(m)}$ is nevertheless expected to be a good approximation for the column space of $\mathbf{U}^{(m)}$. A basis for this dominant column space can be computed with the singular value decomposition (SVD) and the approximation is optimal in least squares sense if $\mathbf{n}^{(m)}$ is white Gaussian noise.

In the case of noisy tensor entries, the linear system becomes $\mathbf{A}\text{vec}(\mathcal{X} + \mathcal{N}) = \mathbf{b}$, with $\mathcal{N} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$. In contrast with the former case, the coefficient matrix now also operates on the noise. Equation (8) then becomes

$$\mathbf{A}^{(m,m)}\left(\mathbf{F}^{(m)} + \mathbf{N}^{(m)}\right) = \mathbf{B}^{(m)}_{[m]} \quad \text{with}$$
$$\mathbf{N}^{(m)} = \left(\prod_{n \neq m} \left(\mathcal{N} \cdot_i \mathbf{A}^{(m,n)}\right)\right)_{[m]} \quad \text{for } m = 1, \ldots, N.$$

Because of the noise, the rank of $\mathbf{F}^{(m)} + \mathbf{N}^{(m)}$ generically exceeds R_m for $m = 1, \ldots, N$. The least squares optimal approximation of the column space of $\mathbf{U}^{(m)}$ can again be retrieved using the SVD if $\left(\otimes_{n=1}^{N} \mathbf{A}^{(m,n)}\right)\text{vec}(\mathcal{N})$ is white Gaussian noise. As derived in Sidiropoulos et al. [12], this holds true if $\text{vec}(\mathcal{N})$ is white Gaussian noise and $\mathbf{A}^{(m,n)}\mathbf{A}^{(m,n)^{\mathrm{T}}} = \mathbf{I}_{P_{mn}}$ for $m, n = 1, \ldots, N$. The latter condition is approximately satisfied for large tensor dimensions if the generating matrices are sampled from a zero-mean uncorrelated distribution.

3. COMPUTING AN ORTHOGONALLY COMPRESSED CPD FROM A BRKS LINEAR SYSTEM

Since the core of the MLSVD of an Nth order tensor is also an Nth order tensor, the MLSVD suffers from the CoD. On the contrary, the number of parameters of the CPD scales linearly with the order of the tensor. Additionally, the CPD is unique under mild conditions. For applications that are more suited to these properties, such as blind signal separation, we introduce an algorithm for computing a CPD from a BRKS linear system in this section. An efficient, three-step approach for computing the CPD of a tensor is: 1) compressing the tensor, 2) decomposing the compressed tensor and 3) expanding the factor matrices to the original dimensions. In Bro and Andersson [9], the tensor is orthogonally compressed using the factor matrices of its MLSVD, i.e., the orthogonally compressed tensor corresponds to the core tensor of the MLSVD. In this section, we generalize this popular approach to computing a CPD from the BRKS linear system in Equation (3). Following this approach, we can find the CPD of a large tensor of order N by computing the CPDs of N small tensors.

3.1. Computing the Factor Matrices

If $\mathcal{X}$ is approximately of rank R, it admits a CPD

$$\mathcal{X} = \left[\!\left[\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(N)}\right]\!\right] \tag{13}$$

with $\mathbf{W}^{(n)} \in \mathbb{R}^{I_n \times R}$ for $n = 1, \ldots, N$. Since the dimension of the subspace spanned by the mode-n vectors of $\mathcal{X}$ is at most of dimension R for $n = 1, \ldots, N$, as $\mathcal{X}$ is of rank R, $\mathcal{X}$ also admits an MLSVD

$$\mathcal{X} = \left[\!\left[\mathcal{S}; \mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}\right]\!\right] \tag{14}$$

with $\mathbf{U}^{(n)} \in \mathbb{R}^{I_n \times R}$ for $n = 1, \ldots, N$ and $\mathcal{S} \in \mathbb{R}^{R \times \cdots \times R}$. Note that the rank R_n of the mode-n vectors of $\mathcal{X}$ can be smaller than R for any $n = 1, \ldots, N$, namely when $\mathbf{W}^{(n)}$ is not of full column rank. In this case, an orthogonal compression matrix $\mathbf{U}^{(n)}$ of dimensions $I_n \times R_n$ can still be used. Equation (13) and (14) together imply that the core tensor $\mathcal{S}$ admits the following CPD:

$$\mathcal{S} = \left[\!\left[\mathbf{U}^{(1)^{\mathrm{T}}}\mathbf{W}^{(1)}, \ldots, \mathbf{U}^{(N)^{\mathrm{T}}}\mathbf{W}^{(N)}\right]\!\right] = \left[\!\left[\mathbf{V}^{(1)}, \ldots, \mathbf{V}^{(N)}\right]\!\right]. \tag{15}$$

Alternatively, the CPD of $\mathcal{X}$ can be written as

$$\mathcal{X} = \left[\!\left[\mathbf{U}^{(1)}\mathbf{V}^{(1)}, \ldots, \mathbf{U}^{(N)}\mathbf{V}^{(N)}\right]\!\right]$$

by substituting Equation (15) into Equation (14). This CPD can be substituted for $\mathcal{S}$ in the BRKS system in Equation (3). Using the mixed product property of the Khatri–Rao product, the mth block row of the BRKS system becomes

$$\left(\bigodot_{n=1}^{N}\left(\mathbf{A}^{(m,n)}\mathbf{U}^{(n)}\mathbf{V}^{(n)}\right)\right)\mathbf{1}_R = \mathbf{b}^{(m)} \quad \text{for } m = 1, \ldots, N.$$

Tensorizing this equation yields a polyadic decomposition (PD) expression for each block row of the BRKS system:

$$\left[\!\left[\mathbf{A}^{(m,1)}\mathbf{U}^{(1)}\mathbf{V}^{(1)}, \ldots, \mathbf{A}^{(m,N)}\mathbf{U}^{(N)}\mathbf{V}^{(N)}\right]\!\right] = \mathcal{B}^{(m)} \text{ for } m = 1, \ldots, N.$$

(Note that, since the number of rank-1 terms in this PD is not necessarily minimal, it is a priori not necessarily canonical. However, we will assume further on that the m-th factor matrix in the PD of $\tilde{\mathcal{B}}^{(m)}$ is unique for $m = 1, \ldots, N$. As that implies that a decomposition in fewer terms is impossible, the PDs are CPDs by our assumption). After solving Equation (9) for $\mathbf{F}^{(m)}$ and estimating $\mathbf{U}^{(m)}$ for $m = 1, \ldots, N$ as described in Section 2.1, the tensorization $\mathcal{F}^{(m)} \in \mathbb{R}^{P_{m1} \times \cdots \times I_m \times \cdots \times P_{mN}}$ of $\mathbf{F}^{(m)}$ is orthogonally compressed

$$\tilde{\mathcal{B}}^{(m)} = \mathcal{F}^{(m)} \cdot_m \mathbf{U}^{(m)^{\mathrm{T}}}$$

$$= \left[\!\left[\mathbf{A}^{(m,1)}\mathbf{U}^{(1)}\mathbf{V}^{(1)}, \ldots, \mathbf{V}^{(m)}, \ldots, \mathbf{A}^{(m,N)}\mathbf{U}^{(N)}\mathbf{V}^{(N)} \right]\!\right] \quad \text{for } m = 1, \ldots, N. \tag{16}$$

The CPD of the compressed tensor $\tilde{\boldsymbol{\mathcal{B}}}^{(m)}$ shares the factor matrix $\mathbf{V}^{(m)}$ with the CPD of the core tensor $\boldsymbol{\mathcal{S}}$. The factor matrices $\mathbf{V}^{(n)}$ for $n = 1, \ldots, N$ of the CPD of the core can thus be found by computing the CPD of each compressed tensor if the mth factor matrix of the CPD of $\tilde{\boldsymbol{\mathcal{B}}}^{(m)}$ is unique for $m = 1, \ldots, N$. If the CPD of the full tensor $\boldsymbol{\mathcal{X}}$ is also unique, its factor matrices can be retrieved by expanding the factor matrices $\mathbf{W}^{(n)} = \mathbf{U}^{(n)}\mathbf{V}^{(n)}$ for $n = 1, \ldots, N$. Since a CPD can only be unique up to the factor scaling and permutation indeterminacies and each factor matrix $\mathbf{V}^{(n)}$ for $n = 1, \ldots, N$ is retrieved from a different compressed tensor, the indeterminacies must be addressed. To this end, the fact that the N tensors $\tilde{\boldsymbol{\mathcal{B}}}^{(m)}$ all have the same CPD, up to the (known) compression matrices $\mathbf{A}^{(m,n)}$ for $m, n = 1, \ldots, N$, can be exploited.

3.2. Algorithm

The steps in Section 3.1 mimic the popular approach in Bro and Andersson [9] for computing the CPD of a fully given tensor, which is: 1) compute the MLSVD of $\boldsymbol{\mathcal{X}}$, 2) compute the CPD of the core $\boldsymbol{\mathcal{S}}$ and 3) expand the factor matrices $\mathbf{W}^{(n)} = \mathbf{U}^{(n)}\mathbf{V}^{(n)}$ for $n = 1, \ldots, N$. The corresponding steps in the approach in this paper are: 1) compute the MLSVD factor matrices $\mathbf{U}^{(n)}$ for $n = 1, \ldots, N$ and orthogonally compress the tensors $\boldsymbol{\mathcal{F}}^{(m)}$ for $m = 1, \ldots, N$, 2) compute the CPD of the compressed tensors and 3) expand the factor matrices $\mathbf{W}^{(n)} = \mathbf{U}^{(n)}\mathbf{V}^{(n)}$ for $n = 1, \ldots, N$. Instead of computing the CPD of a core tensor of dimensions $R \times \cdots \times R$, this approach computes the CPD of the N tensors in Equation (16) which are of dimensions $P_{m1} \times \cdots \times P_{m,m-1} \times R \times P_{m,m+1} \times \cdots \times P_{m,N}$ for $m = 1, \ldots, N$. As will be shown in Section 3.3, these tensors can be far smaller than the core tensor. **Algorithm 2** outlines all steps needed to compute a CPD with orthogonal compression from a BRKS linear system. All steps of this algorithm can be computed in parallel. Like **Algorithm 1**, **Algorithm 2** accomodates both the RTD- and CS-setting.

Instead of computing the CPDs of the tensors $\tilde{\boldsymbol{\mathcal{B}}}^{(m)}$ for $m = 1, \ldots, N$ separately, they can also be computed simultaneously as a set of coupled CPDs. These CPDs are coupled since their factor matrices all depend linearly, with coefficients $\mathbf{A}^{(m,n)}\mathbf{U}^{(n)}$, on the same factors $\mathbf{V}^{(n)}$ for $m, n = 1, \ldots, N$ with $n \neq m$. This set of coupled CPDs can be computed in Tensorlab [26] through structured data fusion [27].

3.3. Conditions for CPD Retrieval

In this section, we derive conditions for the identifiability of the CPD of $\boldsymbol{\mathcal{X}}$ from a BRKS system. The CPD can be identified if the conditions in Theorem 3 hold. In Domanov and De Lathauwer [28], conditions are provided to guarantee that one factor matrix of the CPD is unique.

Theorem 3. *Consider a tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ of rank R, admitting a CPD $\left[\!\left[\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(N)} \right]\!\right]$ with factor matrices $\mathbf{W}^{(n)} \in \mathbb{R}^{I_n \times R}$ for $n = 1, \ldots, N$. Given linear combinations $\boldsymbol{b}$ of vec $(\boldsymbol{\mathcal{X}})$, obtained from a BRKS linear system with generating matrices $\mathbf{A}^{(m,n)} \in \mathbb{R}^{P_{mn} \times I_n}$ for $m, n = 1, \ldots, N$, the factor matrices $\mathbf{W}^{(n)}$ for $n = 1, \ldots, N$ can be retrieved if*

1) $\mathrm{r}\left(\mathbf{A}^{(m,m)}\right) = I_m \quad \text{for } m = 1, \ldots, N;$
2) $\prod_{n \neq m}^{N} \mathrm{r}\left(\mathbf{A}^{(m,n)}\mathbf{U}^{(n)}\right) \geq R \quad \text{for } m = 1, \ldots, N;$
3) The mth factor matrix of $\tilde{\boldsymbol{\mathcal{B}}}^{(m)}$ is unique for $m = 1, \ldots, N$;
4) The CPD of $\boldsymbol{\mathcal{X}}$ is unique.

Proof: Conditions 1) and 2): For the compression matrices $\mathbf{U}^{(n)}$ for $n = 1, \ldots, N$, the factor matrices of the MLSVD of $\boldsymbol{\mathcal{X}}$ must be retrievable. These conditions are the same as the first two conditions in Theorem 1.
Condition 3): The CPD of the compressed tensor $\tilde{\boldsymbol{\mathcal{B}}}^{(m)}$ for $m = 1, \ldots, N$ in Equation (16) shares the mth factor matrix with the core of the MLSVD of $\boldsymbol{\mathcal{X}}$ if the mth factor matrix is unique for $m = 1, \ldots, N$.
Condition 4): While condition 3) ensures that the factor matrices $\mathbf{W}^{(n)}$ for $n = 1, \ldots, N$ are unique, condition 4) is needed to ensure that there is only one set of rank-1 tensors, consisting of the columns of $\mathbf{W}^{(n)}$ for $n = 1, \ldots, N$, that forms a CPD of $\boldsymbol{\mathcal{X}}$, i.e., to exclude different ways of pairing.

In the generic case, in which the generating matrices and the factor matrices of the CPD are sampled from a continuous probability distribution, the CPD of $\boldsymbol{\mathcal{X}}$ can be identified if the conditions in Theorem 4 hold. Here we used a generic condition to prove the uniqueness of the full CPD of $\tilde{\boldsymbol{\mathcal{B}}}^{(m)}$ for $m = 1, \ldots, N$, which a fortiori guarantees the uniqueness of its mth

Algorithm 2: Orthogonally compressed CPD from a BRKS linear system (`lscpd_brks`).

Input: (CS-setting) $\mathbf{A}^{(m,n)}$, $\mathbf{b}^{(m)}$ for $m, n = 1, \ldots, N$
Input: (RTD-setting) $\mathbf{A}^{(m,n)}$ for $m, n = 1, \ldots, N$ and $\boldsymbol{\mathcal{X}}$
Output: $\mathbf{W}^{(m)}$ for $m = 1, \ldots, N$
begin
  if *RTD-setting* **then**
    Compute $\mathbf{b}^{(m)}$ for $m = 1, \ldots, N$ using Equation (3)
  for $m = 1$ **to** N **do**
    Solve $\mathbf{A}^{(m,m)}\mathbf{F}^{(m)} = \mathbf{B}^{(m)}_{[m]}$ for $\mathbf{F}^{(m)}$
    Find $\mathbf{U}^{(m)}$ by computing an orthonormal basis that spans the dominant column space of $\mathbf{F}^{(m)}$
    Orthogonally compress $\tilde{\boldsymbol{\mathcal{B}}}^{(m)} = \boldsymbol{\mathcal{F}}^{(m)} \cdot_m \mathbf{U}^{(m)^{\mathrm{T}}}$
    Compute the CPD of $\tilde{\boldsymbol{\mathcal{B}}}^{(m)}$ and set $\mathbf{V}^{(m)}$ equal to the mth factor matrix
    Scale and permute $\mathbf{V}^{(m)} = \mathbf{D}^{(m)^{-1}}\mathbf{P}^{(m)^{\mathrm{T}}}\mathbf{V}^{(m)}$ to fix CPD indeterminacies
    Expand $\mathbf{W}^{(m)} = \mathbf{U}^{(m)}\mathbf{V}^{(m)}$

factor matrix. The conditions in Theorem 4 can be used to determine the dimensions of the generating matrices that are generically required for the identifiability of the CPD of $\boldsymbol{\mathcal{X}}$.

Theorem 4. *With generic generating matrices* $\mathbf{A}^{(m,n)}$ *for* $m, n = 1, \ldots, N$, *the conditions in Theorem 3 hold if*

1) $P_{mm} \geq I_m \quad \text{for } m = 1, \ldots, N;$
2) $\prod_{n \neq m}^{N} \min(P_{mn}, R) \geq R \quad \text{for } m = 1, \ldots, N;$
3) $I_m \geq R \quad \text{for } m = 1, \ldots, N;$
4) $\left\{ \exists \mathcal{N} \subset \{[1, \ldots, N] \setminus m\} : \min(J_2, J_3) \geq 3 \right.$
$\left. \text{and } (J_2 - 1)(J_3 - 1) \geq R \right\} \quad \text{for } m = 1, \ldots, N$
$\text{with } J_2 = \prod_{n \in \mathcal{N}} P_{mn} \text{ and } J_3 = \prod_{n \in \mathcal{N}^c} P_{mn}$

Proof: Conditions 1) and 2): These conditions are the same as the first two conditions in Theorem 2.
Condition 3) and 4): The mth factor matrix of $\tilde{\boldsymbol{\mathcal{B}}}^{(m)}$ is unique for $m = 1, \ldots, N$ if the CPD of these tensors is unique. The Nth order tensor $\tilde{\boldsymbol{\mathcal{B}}}^{(m)}$ can be reshaped to a third-order tensor $\boldsymbol{\mathcal{Y}} \in \mathbb{R}^{J_1 \times J_2 \times J_3}$, with $J_1 = I_m$, $J_2 = \prod_{n \in \mathcal{N}} P_{mn}$, $J_3 = \prod_{n \in \mathcal{N}^c} P_{mn}$ and $\mathcal{N} \subset \{[1, \ldots, N] \setminus m\}$, of which the first mode corresponds to the uncompressed mth mode of $\tilde{\boldsymbol{\mathcal{B}}}^{(m)}$. The second and third mode, respectively, correspond to a subset $\mathcal{N}$ of the remaining $N - 1$ modes and its complementary subset $\mathcal{N}^c$. The rank R CPD of $\tilde{\boldsymbol{\mathcal{B}}}^{(m)}$ is unique if there exists a subset $\mathcal{N}$ such that the rank R CPD of the reshaped third-order tensor $\boldsymbol{\mathcal{Y}}$ is unique. Generically, a third-order tensor $\boldsymbol{\mathcal{Y}} \in \mathbb{R}^{J_1 \times J_2 \times J_3}$ of rank R is unique if $J_1 \geq R$, $\min(J_2, J_3) \geq 3$ and $(J_2 - 1)(J_3 - 1) \geq R$ [29]. Condition 3) guarantees that the former condition is satisfied and condition 4) guarantees that the latter two conditions are satisfied for $\tilde{\boldsymbol{\mathcal{B}}}^{(m)}$ for $m = 1, \ldots, N$. Additionally, condition 3) guarantees generic uniqueness of the CPD of $\boldsymbol{\mathcal{X}}$ since for each reshaped, third-order version of $\boldsymbol{\mathcal{X}}$, all dimensions exceed R [30, Theorem 3].

As explained in Section 2.4, the second condition in Theorem 4 implies that $P_{mn} \geq \sqrt[N-1]{R}$ if the values P_{mn} are approximately equally large for $m, n = 1, \ldots, N$ and $n \neq m$. Similarly, the fourth condition in Theorem 4 implies that

$$P_{mn} \geq \sqrt[N-1]{\left(1 + \sqrt{R}\right)^2} \quad \text{for } m, n = 1, \ldots, N \text{ and } n \neq m.$$

(Note that this bound is derived for tensors of uneven order N. The bound for tensors of even order is similar, but does not have a simple expression). The latter constraint poses only slightly more restrictive bounds than the former, meaning that the dimensions P_{mn} for $m, n = 1, \ldots, N$ and $n \neq m$ can still be chosen such that they are much smaller than R. In real applications, tensor dimensions often exceed the tensor rank, satisfying the third condition in Theorem 4.

Remark: Note that the rank of a tensor can exceed some of its dimensions, in which case Theorem 4 cannot be satisfied. This can be resolved by using a different uniqueness condition to guarantee the uniqueness of the CPDs of $\tilde{\boldsymbol{\mathcal{B}}}^{(m)}$ for $m = 1, \ldots, N$, such as the generic version of Kruskal's condition [31]. However, using this condition also results in stricter bounds on P_{mn} for $m, n = 1, \ldots, N$ and $n \neq m$.

4. COMPUTING A TT FROM A BRKS LINEAR SYSTEM

Since the TT also does not suffer from the CoD, we derive an algorithm for computing a TT from a BRKS linear system in this section. The TT-SVD algorithm in Oseledets [25] computes the cores using sequential SVDs. Unlike in TT-SVD, for a BRKS linear system the SVDs can be computed in parallel by processing the compressed tensors $\boldsymbol{\mathcal{B}}^{(m)}$ for $m = 1, \ldots, N$ of $\boldsymbol{\mathcal{X}}$ separately.

4.1. Computing the TT Cores

If $\boldsymbol{\mathcal{X}}$ is approximately of TT-rank $(R_0, \ldots, R_N)$, it admits a TT $(\!|\boldsymbol{\mathcal{G}}^{(1)}, \ldots, \boldsymbol{\mathcal{G}}^{(N)}|\!)$ with cores $\boldsymbol{\mathcal{G}}^{(n)} \in \mathbb{R}^{R_{n-1} \times I_n \times R_n}$ for $n = 1, \ldots, N$. Substituting this TT for $\boldsymbol{\mathcal{X}}$ into the BRKS linear system in Equation (3) leads to

$$\mathbf{A}\text{vec}\left((\!|\boldsymbol{\mathcal{G}}^{(1)}, \ldots, \boldsymbol{\mathcal{G}}^{(N)}|\!)\right) = \mathbf{b}.$$

It follows that the tensorized mth block row of this BRKS system corresponds to the TT of $\boldsymbol{\mathcal{X}}$ transformed through mode-n multiplication with the generating matrices $\mathbf{A}^{(m,n)}$:

$$\boldsymbol{\mathcal{B}}^{(m)} = (\!|\boldsymbol{\mathcal{H}}^{(m,1)}, \ldots, \boldsymbol{\mathcal{H}}^{(m,N)}|\!)$$

with $\boldsymbol{\mathcal{H}}^{(m,n)} = \boldsymbol{\mathcal{G}}^{(n)} \bullet_2 \mathbf{A}^{(m,n)}$ for $n = 1, \ldots, N$. Rearranging this transformed TT into its mode-m unfolding leads to

$$\mathbf{A}^{(m,m)} \left((\!|\boldsymbol{\mathcal{H}}^{(m,1)}, \ldots, \boldsymbol{\mathcal{H}}^{(m,m-1)}, \boldsymbol{\mathcal{G}}^{(m)}, \boldsymbol{\mathcal{H}}^{(m,m+1)}, \ldots, \boldsymbol{\mathcal{H}}^{(m,N)}|\!)\right)_{[m]} = \mathbf{B}^{(m)}_{[m]}.$$

This unfolding corresponds to a linear system

$$\mathbf{A}^{(m,m)} \tilde{\mathbf{B}}^{(m)}_{[m]} = \mathbf{B}^{(m)}_{[m]} \tag{17}$$

that can be solved for $\tilde{\mathbf{B}}^{(m)}_{[m]}$. The tensor $\tilde{\boldsymbol{\mathcal{B}}}^{(m)}$ is a transformed TT of $\boldsymbol{\mathcal{X}}$ that shares the mth core $\boldsymbol{\mathcal{G}}^{(m)}$ with the TT of the full tensor $\boldsymbol{\mathcal{X}}$. Following the definition of a TT, this core can be retrieved from the column space of the following matrix unfolding

$$\tilde{\mathbf{B}}^{(m)}_{[1,\ldots,m;\,m+1,\ldots,N]} = \left((\!|\boldsymbol{\mathcal{H}}^{(m,1)}, \ldots, \boldsymbol{\mathcal{H}}^{(m,m-1)}, \boldsymbol{\mathcal{G}}^{(m)}|\!)\right)^{\mathrm{T}}_{[m+1]} \left((\!|\boldsymbol{\mathcal{H}}^{(m,m+1)}, \ldots, \boldsymbol{\mathcal{H}}^{(m,N)}|\!)\right)_{[1]}. \tag{18}$$

(W.r.t. the matricization, note that while $(\!|\boldsymbol{\mathcal{H}}^{(m,1)}, \ldots, \boldsymbol{\mathcal{H}}^{(m,m-1)}, \boldsymbol{\mathcal{G}}^{(m)}|\!)$ consists of m cores, it is a tensor of order $m + 1$ since R_m is not necessarily equal to one).

First, we retrieve $\mathcal{G}^{(1)}$ by computing an orthonormal basis of dimension R_1 for the dominant column space of the matrix unfolding in Equation (18) for $m = 1$. For $m = 2, \ldots, N$, the column space of this unfolding also involves the preceding cores $\mathcal{H}^{(m,1)}, \ldots, \mathcal{H}^{(m,m-1)}$. If the cores are computed in order, then these preceding cores are known and can be compensated for by subsequently solving smaller linear systems

$$\mathbf{H}^{(m,n)}_{[1,2;3]} \mathbf{B}^{(n)}_{\text{new}} = \mathbf{B}^{(n)} \quad \text{for } n = 1, \ldots, m-1, \tag{19}$$

for $\mathbf{B}^{(n)}_{\text{new}}$, with $\mathbf{B}^{(1)} = \tilde{\mathbf{B}}^{(m)}_{[1]}$ and $\mathbf{B}^{(n)} = \text{reshape}\left(\mathbf{B}^{(n-1)}_{\text{new}}, [R_{n-1}P_{mn}, I_m \prod_{k=n+1}^{N} P_{mk}]\right)$ for $n = 2, \ldots, m-1$. After solving these systems, $\mathcal{G}^{(m)}$ is retrieved by computing an orthonormal basis of dimension R_m for the dominant column space of reshape $\left(\mathbf{B}^{(n)}_{\text{new}}, [R_{m-1}I_m, \prod_{k=m+1}^{N} P_{mk}]\right)$ for $m = 1, \ldots, N$.

Alternatively, we can immediately compute an orthonormal basis of dimension R_m for the dominant column space of the matrix unfolding in Equation (18) for $m = 1, \ldots, N$, without compensating for preceding cores first. Computing these bases is more expensive, since they are obtained from a larger matrix than in the case where the preceding cores have already been compensated for. On the other hand, the orthonormal bases can be computed in parallel, as there is no dependency on any preceding core. If after the orthonormal bases have been found, also the cores of the TT are desired, compensation of preceding cores can be done in a similar manner as described above, namely by subsequently solving linear systems. These linear systems have the same coefficient matrix as the linear systems in Equation (19).

4.2. Algorithm

Algorithm 3 outlines all steps needed to compute a TT from a BRKS linear system for the approach in which the orthonormal bases are computed first and the preceding cores are compensated for second. Note that only $N - 1$ SVDs need to be computed, just like in the standard TT-SVD algorithm, in which the final SVD reveals both cores $N - 1$ and N. Like **Algorithm 1** and **2**, **Algorithm 3** also accommodates both the RTD- and CS-setting.

Remark: When computing the mth core in **Algorithm 3**, the $m-1$ preceding compressed cores $\mathcal{H}^{(m,1)}, \ldots, \mathcal{H}^{(m,m-1)}$ in $\tilde{\mathcal{B}}^{(m)}$ are compensated for from left to right, i.e., for $m = 1, \ldots, n-1$. If $\tilde{\mathcal{B}}^{(m)}$ is unfolded in reverse, i.e.,

$$\tilde{\mathbf{B}}^{(m)}_{[N,\ldots,m;m-1,\ldots,1]} = \left(\left\Vert\!\left\vert \mathcal{H}^{(m,N)}, \ldots, \mathcal{H}^{(m,m+1)}, \mathcal{G}^{(m)} \right\vert\!\right\Vert\right)^{\mathrm{T}}_{[N-m+2]} \left(\left\Vert\!\left\vert \mathcal{H}^{(m,m-1)}, \ldots, \mathcal{H}^{(m,1)} \right\vert\!\right\Vert\right)_{[1]}, \tag{20}$$

then the column space of the unfolding contains, besides $\mathcal{G}^{(m)}$, the $N - m$ next compressed cores $\mathcal{H}^{(m,N)}, \ldots, \mathcal{H}^{(m,m+1)}$. This way, it is possible to compute the mth core by compensating for these next compressed cores from right to left, i.e., $\mathcal{H}^{(m,N)}, \ldots, \mathcal{H}^{(m,m+1)}$. The efficiency of **Algorithm 3** can be improved by computing the first half of the cores using the unfolding in Equation (18) and compensating for the preceding cores from left to right, and the second half of the cores using the unfolding in Equation (20) and compensating for the next cores from right to left. This halves the number of cores that need to be compensated for compared to **Algorithm 3**, in which cores are only compensated for from left to right to simplify the pseudocode.

Algorithm 3: TT from a BRKS linear system (`lstt_brks`).

Input: (CS-setting) $\mathbf{A}^{(m,n)}, \mathbf{b}^{(m)}$ for $m, n = 1, \ldots, N$
Input: (RTD-setting) $\mathbf{A}^{(m,n)}$ for $m, n = 1, \ldots, N$ and $\boldsymbol{\mathcal{X}}$
Output: $\mathcal{G}^{(m)}$ for $m = 1, \ldots, N$
begin
  if *RTD-setting* **then**
    Compute $\mathbf{b}^{(m)}$ for $m = 1, \ldots, N$ using Equation (3)
  for $m = 1$ **to** N **do**
    Solve $\mathbf{A}^{(m,m)} \tilde{\mathbf{B}}^{(m)}_{[m]} = \mathbf{B}^{(m)}_{[m]}$ for $\tilde{\mathbf{B}}^{(m)}_{[m]}$
    Estimate $\tilde{\mathbf{G}}^{(m)}$ as an orthonormal basis for the dominant column space of $\tilde{\mathbf{B}}^{(m)}_{[1,\ldots,m;m+1,\ldots,N]}$
    for $n = 1$ **to** $m - 1$ **do**
      Reshape $\tilde{\mathbf{G}}^{(m)}$ to dimensions $R_{n-1}P_{mn} \times P_{m,n+1} \cdots P_{m,m-1} I_m R_m$
      Solve the system $\mathbf{H}^{(m,n)}_{[1,2;3]} \mathbf{G}_{\text{new}} = \tilde{\mathbf{G}}^{(m)}$ for $\mathbf{G}_{\text{new}}$
      Set $\tilde{\mathbf{G}}^{(m)} = \mathbf{G}_{\text{new}}$
    $\mathcal{G}^{(m)} = \text{reshape}\left(\tilde{\mathbf{G}}^{(m)}, [R_{m-1}, I_m, R_m]\right)$

4.3. Conditions for TT Retrieval

In this section, we derive the conditions in Theorem 5, under which retrieval of the TT of $\boldsymbol{\mathcal{X}}$ from a BRKS system is guaranteed. The compensation for preceding cores occurs in both approaches for computing the TT in Section 4.1. Therefore, the linear systems that are solved to compensate for them must have a unique solution. Since these linear systems have the same coefficient matrices in both approaches, the condition under which they have a unique solution is the same, regardless of the chosen approach.

Theorem 5. *Consider a tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ of TT-rank $(R_0, \ldots, R_N)$, admitting a TT $\left(\left\Vert\!\left\vert \mathcal{G}^{(1)}, \ldots, \mathcal{G}^{(N)} \right\vert\!\right\Vert\right)$ with cores $\mathcal{G}^{(n)} \in \mathbb{R}^{R_{n-1} \times I_n \times R_n}$ for $n = 1, \ldots, N$. Given linear combinations $\boldsymbol{b}$ of vec $(\boldsymbol{\mathcal{X}})$, obtained from a BRKS linear system with generating matrices $\mathbf{A}^{(m,n)} \in \mathbb{R}^{P_{mn} \times I_n}$ for $m, n = 1, \ldots, N$, the cores $\mathcal{G}^{(n)}$ for $n = 1, \ldots, N$ can be retrieved if and only if*

1) $\text{r}\left(\mathbf{A}^{(m,m)}\right) = I_m \quad \text{for } m = 1, \ldots, N;$
2) $\text{r}\left(\tilde{\mathbf{B}}^{(m)}_{[1,\ldots,m;m+1,\ldots,N]}\right) = R_m \quad \text{for } m = 1, \ldots, N;$
3) $\text{r}\left(\mathbf{H}^{(m,n)}_{[1,2;3]}\right) = R_n \quad \text{for } m = 1, \ldots, N \text{ and } n = 1, \ldots, m-1.$

Proof: Condition 1): The linear system in Equation (17) can be uniquely solved for $\tilde{\mathbf{B}}^{(m)}_{[m]}$ if and only if $\mathbf{A}^{(m,m)}$ is of full column

rank I_m for $m = 1, \ldots, N$.
Condition 2): An orthonormal basis of dimension R_m for the column space of the matricization $\tilde{\mathbf{B}}^{(m)}_{[1,\ldots,m;m+1,\ldots,N]}$ in Equation (18) can be retrieved if and only if the rank of this matricization equals R_m.

Condition 3): The linear systems that are solved to compensate for compressed cores have a unique solution if and only if

$$\mathrm{r}\left(\mathbf{H}^{(m,n)}_{[1,2;3]}\right) = R_n \quad \text{for } m = 1, \ldots, N \text{ and } n = 1, \ldots, m-1$$

holds.

In the generic case, in which the generating matrices are sampled from a continuous probability distribution, the TT of $\boldsymbol{\mathcal{X}}$ can be retrieved if the conditions in Theorem 6 hold. These conditions can be used to determine the dimensions of the generating matrices such that the TT of $\boldsymbol{\mathcal{X}}$ can generically be retrieved.

Theorem 6. *With generic generating matrices* $\mathbf{A}^{(m,n)}$ *for* $m, n = 1, \ldots, N$, *the conditions in Theorem 1 hold if and only if*

1) $P_{mm} \geq I_m \quad \text{for } m = 1, \ldots, N;$
2) $P_{mn} \geq \dfrac{R_m}{R_{n-1} I_m \prod_{k=n+1}^{m-1} P_{mk}}$
for $m = 1, \ldots, N$ and $n = 1, \ldots, n-1;$
3) $P_{mn} \geq \dfrac{R_m}{R_n \prod_{k=m+1}^{n-1} P_{mk}}$
for $m = 1, \ldots, N$ and $n = m+1, \ldots, N;$
4) $P_{mn} \geq \dfrac{R_n}{R_{n-1}} \quad \text{for } m = 1, \ldots, N \text{ and } n = 1, \ldots, m-1.$

Proof: Condition 1): This condition is the same as condition 1) in Theorem 2.
Condition 2): The matricization $\tilde{\mathbf{B}}^{(m)}_{[1,\ldots,m;m+1,\ldots,N]}$ in Equation (18) equals

$$\tilde{\mathbf{B}}^{(m)}_{[1,\ldots,m;m+1,\ldots,N]} = \left(\mathbf{G}^{(m)}_{[3]}\left(\mathbf{H}^{(m,m-1)}_{[3]}\left(\cdots \mathbf{H}^{(m,2)}_{[3]}\left(\mathbf{H}^{(m,1)}_{[3]} \otimes \mathbf{I}_{P_{m2}}\right)\cdots \otimes \mathbf{I}_{P_{m,m-1}}\right)\otimes \mathbf{I}_{I_m}\right)\right)^{\mathrm{T}} \left(\mathbf{H}^{(m,m+1)}_{[1]}\left(\mathbf{I}_{P_{m,m+1}} \otimes \cdots \mathbf{H}^{(m,N-1)}_{[1]}\left(\mathbf{I}_{P_{m,N-1}} \otimes \mathbf{H}^{(m,N)}_{[1]}\right)\right)\right) =: \mathbf{D}^{\mathrm{T}}\mathbf{E}. \tag{21}$$

Generically, the rank of $\tilde{\mathbf{B}}^{(m)}_{[1,\ldots,m;m+1,\ldots,N]}$ equals R_m if and only if the rank of both $\mathbf{D}$ and $\mathbf{E}$ equals R_m. Condition 2) relates to $\mathbf{D}$ and condition 3) to $\mathbf{E}$. Since $\mathbf{G}^{(m)}_{[3]}$ is of full rank, which follows from the definition of a TT, and $\mathbf{H}^{(m,n)}_{[3]}$ for $n \neq m$ is generically of full rank, each matrix product in Equation (21) is also of full rank [10, Lemma 1]. Therefore, the rank of $\mathbf{D}$ generically equals

$$\min\left(\mathrm{r}\left(\mathbf{G}^{(m)}_{[3]}\right), \min\left(\mathrm{r}\left(\mathbf{H}^{(m,m-1)}_{[3]}\right), \ldots, \min\left(\mathrm{r}\left(\mathbf{H}^{(m,2)}_{[3]}\right), \mathrm{r}\left(\mathbf{H}^{(m,1)}_{[3]}\right) P_{m2}\right) P_{m3} \cdots P_{m,m-1}\right) I_m\right).$$

Both arguments of the leftmost min(·) must at least equal R_m such that $\mathrm{r}(\mathbf{D}) \geq R_m$ holds. Following the definition of a TT, this always holds true for the first argument $\mathrm{r}\left(\mathbf{G}^{(m)}_{[3]}\right)$. The second argument is $\min(\cdot) I_m$, so both arguments of this second min(·) must at least equal $\frac{R_m}{I_m}$. Generically $\mathrm{r}\left(\mathbf{H}^{(m,m-1)}_{[3]}\right) = \min(P_{m,m-1} R_{m-2}, R_{m-1})$, leading to the conditions $P_{m,m-1} \geq \frac{R_m}{R_{m-2} I_m}$ and $R_{m-1} I_m \geq R_m$. The latter condition is satisfied by the definition of a TT. Repeating the same steps for each subsequent min(·) leads to the conditions

$$P_{mn} \geq \frac{R_m}{R_{n-1} I_m \prod_{k=n+1}^{m-1} P_{mk}} \quad \text{for } m = 1, \ldots, N \text{ and } n = 1, \ldots, m-1.$$

Condition 3): In a similar fashion, it can be proven that $\mathrm{r}(\mathbf{E}) \geq R_m$ holds if and only if

$$P_{mn} \geq \frac{R_m}{R_n \prod_{k=m+1}^{n-1} P_{mk}} \text{ for } m = 1, \ldots, N \text{ and } n = m+1, \ldots, N.$$

Condition 4): Since a generic matrix is of full rank, $\mathrm{r}\left(\mathbf{H}^{(m,n)}_{[1,2;3]}\right) = \min\left(R_{n-1} P_{mn}, R_n\right)$ for $m, n = 1, \ldots, N$.

5. EXPERIMENTS

In this section, we validate our algorithms using synthetic and real data. The algorithms are implemented in MATLAB and are available at https://www.tensorlabplus.net. In the practical implementation of the algorithms, column spaces are estimated using the SVD and linear systems are solved using the MATLAB backslash operator. All experiments are performed on a laptop with an AMD Ryzen 7 PRO 3700U processor and 32GB RAM. The algorithms are run sequentially even though each algorithm can (partly) be executed in parallel.

Since it is possible to choose the generating matrices in the experiments in this section, we set $\mathbf{A}_{mm} = \mathbf{I}_{I_m}$ for $m = 1, \ldots, N$. This means that the first step in each algorithm, namely solving a linear system with $\mathbf{A}_{mm}$ as the coefficient matrix, can be skipped. First, we use synthetic data to compare the accuracy and computation time of these algorithms to related algorithms. All synthetic problems are constructed by sampling the entries of the factor matrices and/or core(s) of a tensor decomposition from the standard normal distribution. Additive Gaussian noise $\boldsymbol{\mathcal{N}}$ is added to these randomly generated tensors $\boldsymbol{\mathcal{X}}$ and the noise level is quantified using the signal-to-noise ratio (SNR):

$$\mathrm{SNR} = 10 \log_{10}\left(\frac{||\boldsymbol{\mathcal{X}}||_{\mathrm{F}}^2}{||\boldsymbol{\mathcal{N}}||_{\mathrm{F}}^2}\right).$$

5.1. Randomized MLSVD

In the first experiment, a random third-order tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I \times I \times I}$ of low multilinear rank (R, R, R), with $I = 200$ and $R = 10$,

is generated with varying levels of noise. The factor matrices of the MLSVD of $\mathcal{X}$ are then retrieved from a BRKS linear system using **Algorithm 1**. The entries of the generating matrices are sampled from the standard normal distribution. This means that we are using **Algorithm 1** in the RTD-setting in this experiment. The dimensions of the generating matrices are determined using the oversampling factor $q = 5$ and the multilinear rank of $\mathcal{X}$. For this value of q, the sampling ratio equals 0.005. This ratio is defined as the number of compressed measurements in $\mathbf{b}$ divided by the number of entries in $\mathcal{X}$. We compare the accuracy of our algorithm with related RTD algorithms and a cross-approximation algorithm for low multilinear rank approximation. These algorithms are:

- `rand_tucker` Algorithm 2 in [16]: The factor matrices $\mathbf{U}^{(n)}$ for $n = 1, \ldots, N$ are retrieved as follows: 1) the mode-n matrix unfolding of $\mathcal{X}$ is compressed through multiplication with a random Gaussian matrix and 2) an orthonormal basis for this compressed matrix unfolding is computed using the QR decomposition. After computing each factor matrix, $\mathcal{X}$ is orthogonally compressed using this factor matrix like in the sequentially truncated higher-order SVD algorithm [32]. The oversampling factor is set to $p = 5$, which means that this algorithm actually estimates an MLSVD of multilinear rank $(R + p, R + p, R + p)$.
- `rand_tucker_kron` algorithm 4.2 in [15]: Similar to `rand_tucker`, but the algorithm uses a Kronecker-structured matrix for compression and the SVD for computing an orthonormal basis. The oversampling factor for this algorithm is chosen such that it is the same as our oversampling factor q in Section 2.4.
- `mlsvd_rsi` [33]: Computes an MLSVD using sequential truncation [32] and uses randomized compression and subspace iteration for estimating the SVD [34]. In the randomized compression step, the mode-n unfolding $\mathbf{X}_{[n]}$ is compressed through matrix multiplication with a random matrix of dimensions $\prod_{i \neq n} I_i \times R_n + p$, with an oversampling factor $p = 5$. Next, the nth factor matrix is estimated by computing an SVD of this randomly compressed matrix and further refined using two subspace iteration steps with the full mode-n unfolding $\mathbf{X}_{[n]}$.
- `lmlra_aca` [21]: Cross approximation approach for low multilinear rank approximation.

The last two algorithms are available in Tensorlab [26]. As the full tensor $\mathcal{X}$ is usually available in the RTD-setting, the core is computed as

$$\mathcal{S} = \mathcal{X} \cdot_1 \mathbf{U}^{(1)^{\mathrm{T}}} \cdots \cdot_N \mathbf{U}^{(N)^{\mathrm{T}}}.$$

Figure 2 compares the accuracy, quantified as a relative error

$$E_{\mathrm{rel}} = \frac{||\mathcal{X} - \hat{\mathcal{S}} \cdot_1 \hat{\mathbf{U}}^{(1)} \cdots \cdot_N \hat{\mathbf{U}}^{(N)}||_{\mathrm{F}}}{||\mathcal{X}||_{\mathrm{F}}}$$

with $\hat{\mathcal{S}}$ and $\hat{\mathbf{U}}^{(n)}$ for $n = 1, \ldots, N$ the estimated core and factor matrices, of these algorithms. The error shown in **Figure 2** is the average relative error over 10 trials. Algorithm `lsmlsvd_brks` is more accurate than `rand_tucker` and `lmlra_aca`. Algorithm `mlsvd_rsi` is far more accurate than all other algorithms because it uses subspace iteration with the full mode-n matrix unfolding of $\mathcal{X}$ for computing the nth factor matrix. For this reason it also has the longest computation time of all algorithms. Algorithm `rand_tucker_kron` is more accurate than `lsmlsvd_brks` due to the sequential truncation step in `rand_tucker_kron` after each factor matrix is computed. If this step is omitted, which corresponds to Che et al. algorithm 4.1 in [15], it achieves the same accuracy as `lsmlsvd_brks`. This sequential truncation step is not possible in `lsmlsvd_brks` since this algorithm only uses the compressed measurements $\mathbf{b}$ instead of the full tensor $\mathcal{X}$. Note that increasing the oversampling factors of the algorithms results in higher accuracy in exchange for a longer computation time and requiring more compressed datapoints.

5.2. Randomized CPD

In this experiment, we generate a CPD with random factor matrices of a third-order tensor $\mathcal{X} \in \mathbb{R}^{I \times I \times I}$, with $I = 100$, of rank $R = 10$. Varying levels of noise are added to this tensor. **Algorithm 2** is used in the RTD-setting to estimate the factor matrices of the CPD of $\mathcal{X}$. To compute the CPDs of the compressed tensors, we use the `cpd` function in Tensorlab [26]. This function initializes the factor matrices with a generalized eigenvalue decomposition if possible and further improves them using (second-order) optimization algorithms. The accuracy of the factor matrices estimated by this algorithm are compared to results obtained with related RTD algorithms:

- `cpd_rbs` [35]: In each iteration, a random subtensor of $\mathcal{X}$ is sampled and the corresponding rows of the factor matrices are updated. These updates are computed using a Gauss–Newton algorithm. In this

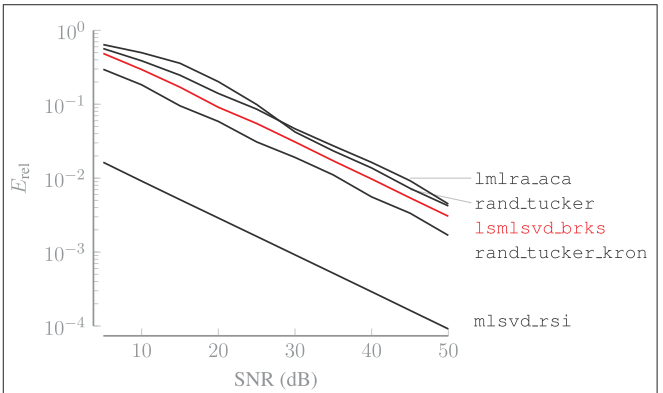


FIGURE 2 | Algorithm `mlsvd_rsi` is by far the most accurate because it used the full-sized matrix unfolding in the subspace iteration step. Algorithm `lsmlsvd_brks` is more accurate than `lmlra_aca` and `rand_tucker` and less accurate than `rand_tucker_kron`. If the sequential truncation step in `rand_tucker_kron` is omitted, it achieves the same accuracy as `lsmlsvd_brks`.

experiment, the algorithm starts with random initial factor matrices.
- `cp_arls` [14]: Alternating least squares with random sketching for solving the least squares subproblems.

The accuracy of the estimated factor matrices is quantified as a relative error

$$E_{\text{CPD}} = \max_n \frac{||\hat{\mathbf{U}}^{(n)} - \mathbf{U}^{(n)}||_{\text{F}}}{||\mathbf{U}^{(n)}||_{\text{F}}},$$

in which the scaling and permutation ambiguities between the true $\mathbf{U}^{(n)}$ and estimated $\hat{\mathbf{U}}^{(n)}$ factor matrix have been resolved for $n = 1, \ldots, N$. **Figure 3** shows the average relative error on the left and the average computation time for tensors with increasing dimensions on the right. Both averages are computed over 10 trials. For `lscpd_brks`, sampling the randomly compressed tensors, i.e., evaluating Equation (3), is included in the computation time. The oversampling factor q of `lscpd_brks` is set to 5, 10 or 50. For a larger value of q, $\mathbf{U}^{(n)}$ better captures the mode-n subspace of $\boldsymbol{\mathcal{X}}$ and less information is lost during the orthogonal compression step. **Figure 3** illustrates that the algorithm is much more accurate for $q = 50$, while the increase in computation time compared to $q = 5$ is negligible. If q is set such that the size of the compressed tensors in Equation (4) is of the same order as the full tensor $\boldsymbol{\mathcal{X}}$, then the computation time will of course increase significantly. For $q = 50$, the number of compressed datapoints in $\mathbf{b}$ equals just 15% of the total number of datapoints in $\boldsymbol{\mathcal{X}}$. This sampling ratio can be even lower for tensors with order greater than three. Algorithm `cp_arls` is slightly more accurate than `lscpd_brks` with $q = 50$ while being much slower in terms of computation time. Algorithm `cpd_rbs` is even more accurate and is situated in between `lscpd_brks` and `cp_arls` for smaller values of I. The computation time of `lscpd_brks`is dominated by sampling the randomly compressed tensors, since computing the CPD of these small tensors is very fast. Therefore, the computation time of `cpd_rbs` scales better for higher values of I since sampling random subtensors of $\boldsymbol{\mathcal{X}}$ is less time consuming than the random compression of $\boldsymbol{\mathcal{X}}$ in `lsmlsvd_brks`, which requires multiple large matrix products. In contrast to `lscpd_brks`, which uses a fixed amount of compressed datapoints determined by the size of the problem and the oversampling factor, `cpd_rbs` and `cp_arls` continue randomly sampling from $\boldsymbol{\mathcal{X}}$ every iteration until a stopping criterion is met.

5.3. Randomized TT

In this experiment, we generate a TT with random core tensors of a third-order tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I \times I \times I}$, with $I = 500$, of TT-rank $(1, 10, 10, 1)$. We add varying levels of noise to this tensor and estimate the cores using **Algorithm 3** in the RTD-setting. For this experiment, we use the version of this algorithm that first computes orthonormal bases and then compensates for preceding cores. Additionally, preceding cores are compensated for from left to right for the first half of the cores and from right to left for the second half, as explained in Section 4.2. The dimensions of the generating matrices are chosen such that the sampling ratio equals 0.005.

The results of `lstt_brks` are compared with the results of a related RTD algorithm and a TT cross-approximation approach:

- `rand_tt` Algorithm 5.1 in [36]: This algorithm is a randomized version of the standard TT-SVD algorithm. The TT-SVD algorithm consists of three steps that are performed for the first $N - 1$ cores: 1) tensor $\boldsymbol{\mathcal{X}}$ is matricized, 2) a core is estimated by computing a basis for the column space of this matricization using the SVD and 3) tensor $\boldsymbol{\mathcal{X}}$ is compressed using this basis. In `rand_tt`, the matricization in the first step is compressed by multiplying it with a random matrix from the right in order to speed up the computation of the SVD in the next step.
- `cross_tt` [37]: This algorithm reduces the size of the matricizations of $\boldsymbol{\mathcal{X}}$ using a maximal volume cross-approximation approach. The matricizations used in this algorithm allow the TT-ranks to be determined adaptively.

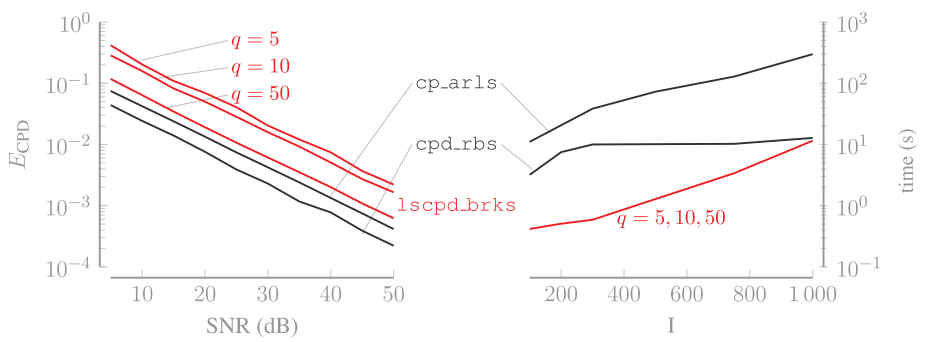


FIGURE 3 | Increasing the oversampling factor q of `lscpd_brks`, for which the value is shown in the figure, greatly improves its accuracy while having a negligible effect on the computation time. Algorithm `lscpd_brks` is faster than `cp_arls` and `cp_rbs`, for a range of tensor dimensions that are relevant for applications, with a limited loss of accuracy.

The accuracy of the results of these algorithms are quantified as a relative error

$$E_{\text{rel}} = \frac{||\boldsymbol{\mathcal{X}} - (\!|\hat{\boldsymbol{\mathcal{G}}}^{(1)}, \ldots, \hat{\boldsymbol{\mathcal{G}}}^{(N)}|\!)||_{\text{F}}}{||\boldsymbol{\mathcal{X}}||_{\text{F}}},$$

in which $\hat{\boldsymbol{\mathcal{G}}}^{(n)}$ for $n = 1, \ldots, N$ are the estimated cores. **Figure 4** shows the relative error, averaged over 50 trials, and the computation time, averaged over 10 trials, for all algorithms. Tensors of increasing size are used to estimate the computation time. The relative errors achieved by the algorithms with these tensors are in proportion with the errors shown in **Figure 4**. The relative error is approximately the same for both `rand_tt` and `lstt_brks`. The former is faster thanks to the compression step that is performed after computing each core, which causes the tensor to get progressively smaller as the algorithm progresses. However, this algorithm requires the full tensor to be available, whereas the latter works with a limited amount of Kronecker-structured linear combinations of the entries of $\boldsymbol{\mathcal{X}}$. Therefore, `lstt_brks` enables us to achieve accuracy comparable to `rand_tt` in CS applications. If a method for obtaining any entry of $\boldsymbol{\mathcal{X}}$ is available, then `cross_tt` can be used to recover a more accurate approximation of the tensor, in exchange for a longer computation time. However, this algorithm is not applicable in the CS-setting either.

5.4. Compressed Sensing Hyperspectral Imaging

In the RTD-setting, the tensor is fully acquired first and then randomly compressed. In the CS-setting, the data can in some applications be compressed during the data acquisition step. For example in hyperspectral imaging, compressed measurements can be obtained using the single-pixel camera [38] or the coded aperture snapshot spectral imager (CASSI) [39]. A hyperspectral image is a third-order tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times I_2 \times I_3}$, consisting of images of I_1 by I_2 pixels taken at I_3 different wavelengths. The single-pixel camera randomly compresses the spatial dimensions of a hyperspectral image. In CASSI, a mask is applied to each I_1 by I_2 image for different wavelengths and these masked images are then aggregated into a single one, i.e., a snapshot. In general, compressed measurements are often obtained in (hyperspectral) imaging by compressing along each dimension separately, meaning that the measurement matrix is Kronecker-structured [11, 40]. With a small number of compressed measurements, a large hyperspectral image can often be accurately reconstructed. Whereas, the random compression step dominated the computation time of our algorithms in the previous experiments, it is missing altogether in this experiment since it is inherent to the application. Therefore, our algorithms enable fast reconstruction of hyperspectral images.

We use `lsmlsvd_brks` to reconstruct a hyperspectral image using compressed measurements sampled at different sampling ratios. Similar to KCS, this approach expresses $\boldsymbol{\mathcal{X}}$ in a Kronecker-structured basis, namely the Kronecker product of $\mathbf{U}^{(n)}$ for $n = 1, 2, 3$, in which the core tensor $\boldsymbol{\mathcal{S}}$ contains the coefficients. In KCS, the bases are chosen a priori such that these coefficients are sparse, which is not necessarily true for the core of an MLSVD. Also, in `lsmlsvd_brks`, the bases do not have to be chosen a priori as this algorithm also estimates them using compressed measurements. Additionally, in KCS the measurement matrix is Kronecker-structured, whereas it consists of multiple Kronecker-structured block rows in our approach.

Instead of using just the final, $(N + 1)$th block row of the BRKS system to estimate the core, it makes more sense to use all available compressed measurements to improve the quality of the estimation, since the number of measurements is limited. Therefore, we solve the full BRKS system, with all $N + 1$ block rows, for a sparse vec $(\boldsymbol{\mathcal{S}})$ by optimizing:

$$\min_{\text{vec}(\boldsymbol{\mathcal{S}})} ||\text{vec}\,(\boldsymbol{\mathcal{S}})\,||_1 \quad \text{subject to } ||\mathbf{A}\text{vec}\,(\boldsymbol{\mathcal{S}}) - b||_2 \leq \sigma.$$

This is a basis pursuit denoising problem, which can be solved using the SPGL1 solver [41, 42]. The reason for imposing sparsity is as follows. If we were to estimate a dense core tensor using `lsmlsvd_brks` in this experiment, the number of compressed measurements $\mathbf{b}^{(N+1)}$ certainly cannot be less than the number of

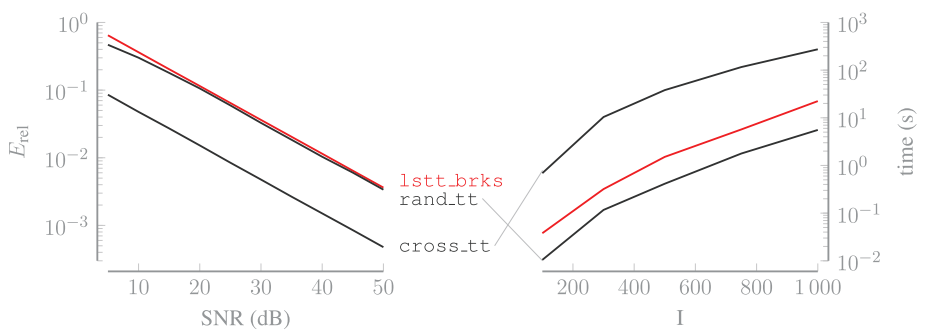


FIGURE 4 | Algorithm `lstt_brks` enables us to achieve accuracy similar to that of `rand_tt` in CS applications. If a method for obtaining any entry of $\boldsymbol{\mathcal{X}}$ is available, then `cross_tt` can be used to recover a more accurate approximation of the tensor, in exchange for a longer computation time.

entries in the core if we want to be able to retrieve it. However, it turns out that the mode-n vectors of the hyperspectral imaging data $\mathcal{X}$ cannot be well approximated as linear combinations of a small number of multilinear singular vectors, indicating that the multilinear rank is not small. Estimating a dense core tensor would then require a large number of measurements. On the other hand, it also turns out that in this data only a relatively small number of the entries in a relatively large core is important. We exploit this by computing a sparse approximation of the core, which results in reconstructions of better quality than with the dense core approach, while requiring far less compressed measurements.

The reconstruction results of our algorithm are compared with two other approaches for reconstructing a hyperspectral image from compressed measurements:

- `GAP_TV` [43]: A generalized alternating projection algorithm that solves the total variation minimization problem. Minimizing total variation leads to accurate image reconstruction because images are generally locally self-similar.
- `KCS` [11]: We used a two-dimensional Daubechies wavelet basis to sparsify the spatial dimensions and the Fourier basis for the spectral dimension. The sparse coefficients are computed using the SPGL1 solver.

The quality of the reconstructed images is quantified using the peak signal-to-noise ratio (PSNR)

$$\mathrm{PSNR} = \log_{10}\left(\frac{\max(\mathcal{X})}{\sqrt{\frac{1}{I_1 I_2 I_3}||\mathcal{X}-\hat{\mathcal{X}}||_{\mathrm{F}}}}\right),$$

in which $\hat{\mathcal{X}}$ is the reconstructed hyperspectral image. In this experiment, we used the corrected Indian Pines dataset, in which some very noisy wavelengths have been left out [44]. This results in a hyperspectral image of dimensions $145 \times 145 \times 200$. For `lsmlsvd_brks`, we compute an MLSVD of multilinear rank $(70, 70, 30)$ with a sparse core. This multilinear rank was obtained by trying a wide range of multilinear ranks and assessing the quality of their corresponding reconstructions.

TABLE 1 | This table shows the reconstruction quality, quantified in PSNR (dB), of a hyperspectral image for a range of sampling ratios, obtained with different algorithms. Algorithm `lsmlsvd_brks` performs approximately equally well as `GAP_TV`, which is an algorithm specifically suited for image reconstruction. The lower reconstruction quality for `KCS` indicates that the bases estimated by `lsmlsvd_brks` suit the data better than the a priori determined bases used in `KCS`.

Algorithm	Sampling ratio		
	0.02	0.05	0.1
`lsmlsvd_brks`	25.58	28.67	28.25
`GAP_TV`	25.68	28.18	30.28
`KCS`	10.94	16.40	22.83

Table 1 shows the quality of the reconstructed images for a range of sampling ratios. Whereas, `GAP_TV` is specifically designed for imaging applications, `lsmlsvd_brks` can be applied to a wide range of problems. Regardless, `lsmlsvd_brks` performs approximately equally well in terms of reconstruction quality. The reconstruction quality achieved by `lsmlsvd_brks` is higher than for `KCS`, indicating that the bases estimated by the former suit the data better than the a priori determined bases used in the latter.

6. CONCLUSION AND FURTHER WORK

In this work, we have considered a general framework of BRKS linear systems with a compact solution, which suits a wide variety of problems. We developed efficient algorithms for computing an MLSVD, CPD or TT constrained solution from a BRKS system, allowing the user to choose the decomposition that best matches their specific application. The efficiency of these algorithms is enabled on one hand by the BRKS linear system, since such a system produces multiple compressed versions of the tensor and thus splits the problem into a number of smaller ones, and on the other hand by the low (multilinear-/TT-)rank constrained solution. With these algorithms, real data can be accurately reconstructed using far fewer compressed measurements than the total number of entries in the dataset. We have derived conditions under which an MLSVD, CPD or TT can be retrieved from a BRKS system. The corresponding generic versions of these conditions allow us to choose the dimensions of the generating matrices such that a solution can generically be found. Through numerical experiments, we have shown that these algorithms can be used for computing tensor decompositions in a randomized approach. In the case of the CPD, our algorithm needs less computation time than the alternative algorithms. Additionally, we have illustrated the good performance of the algorithms for reconstructing compressed hyperspectral images, despite not being specifically developed for this application. In further work, we will look into parallel implementations for the algorithms in this paper.

DATA AVAILABILITY STATEMENT

Publicly available datasets were analyzed in this study. This data can be found at: https://rslab.ut.ac.ir/data.

AUTHOR CONTRIBUTIONS

SH derived the algorithms and theory and implemented the algorithms in Matlab. LDL conceived the main idea and supervised the research. Both authors contributed to the article and approved the submitted version.

FUNDING

This research received funding from the Flemish Government (AI Research Program). LDL and SH are affiliated to Leuven.AI -

KU Leuven institute for AI, B-3000, Leuven, Belgium. This work was supported by the Fonds de la Recherche Scientifique–FNRS and the Fonds Wetenschappelijk Onderzoek- Vlaanderen under EOS Project no G0F6718N (SeLMA). KU Leuven Internal Funds: C16/15/059, IDN/19/014.

ACKNOWLEDGMENTS

The authors would like to thank N. Vervliet and M. Ayvaz for proofreading the manuscript and E. Evert for his help with the mathematical proofs.

REFERENCES

1. Candès EJ, Wakin MB. An introduction to compressive sampling. *IEEE Signal Process Mag*. (2008). 25:21–30. doi: 10.1109/MSP.2007.914731
2. Donoho DL. Compressed sensing. *IEEE Trans Inf Theory*. (2006) 52:1289–306. doi: 10.1109/TIT.2006.871582
3. Ahmadi-Asl S, Abukhovich S, Asante-Mensah MG, Cichocki A, Phan AH, Tanaka T, et al. Randomized algorithms for computation of tucker decomposition and higher order SVD (HOSVD). *IEEE Access*. (2021) 9:28684–706. doi: 10.1109/ACCESS.2021.3058103
4. Acar E, Dunlavy DM, Kolda TG, Morup M. Scalable tensor factorizations for incomplete data. *Chemometr Intell Lab*. (2011) 3106:41–56. doi: 10.1016/j.chemolab.2010.08.004
5. Aldroubi A, Gröchenig K. Nonuniform sampling and reconstruction in shift-invariant spaces. *SIAM Rev*. (2001) 43:585–620. doi: 10.1137/S0036144501386986
6. Oseledets I, Tyrtyshnikov E. TT-cross approximation for multidimensional arrays. *Linear Algebra Appl*. (2010) 432:70–88. doi: 10.1016/j.laa.2009.07.024
7. Udell M, Townsend A. Why are big data matrices approximately low rank? *SIAM J Math Data Sci*. (2019) 1:144–60. doi: 10.1137/18M1183480
8. Rubinstein R, Bruckstein AM, Elad M. Dictionaries for sparse representation modeling. *Proc IEEE*. (2010) 98:1045–57. doi: 10.1109/JPROC.2010.2040551
9. Bro R, Andersson C. Improving the speed of multiway algorithms: part II: compression. *Chemometr Intell Lab Syst*. (1998) 42:105–13. doi: 10.1016/S0169-7439(98)00011-2
10. Sidiropoulos ND, Kyrillidis A. Multi-way compressed sensing for sparse low-rank tensors. *IEEE Signal Process. Lett*. (2012) 19:757–60. doi: 10.1109/LSP.2012.2210872
11. Duarte MF, Baraniuk RG. Kronecker compressive sensing. *IEEE Trans Image Process*. (2012) 21:494–504. doi: 10.1109/TIP.2011.2165289
12. Sidiropoulos N, Papalexakis EE, Faloutsos C. Parallel randomly compressed cubes: a scalable distributed architecture for big tensor decomposition. *IEEE Signal Process Mag*. (2014) 31:57–70. doi: 10.1109/MSP.2014.2329196
13. Kressner D, Tobler C. Low-rank tensor krylov subspace methods for parametrized linear systems. *SIAM J Matrix Anal Appl*. (2011). 32:1288–316. doi: 10.1137/100799010
14. Battaglino C, Ballard G, Kolda TG. A practical randomized CP tensor decomposition. *SIAM J Matrix Anal Appl*. (2018) 39:876–901. doi: 10.1137/17M1112303
15. Che M, Wei Y, Yan H. Randomized algorithms for the low multilinear rank approximations of tensors. *J Computat Appl Math*. (2021) 390:113380. doi: 10.1016/j.cam.2020.113380
16. Zhou G, Cichocki A, Xie S. Decomposition of big tensors with low multilinear rank. (2014) *CoRR. abs/1412.1885*.
17. Yang B, Zamzam A, Sidiropoulos ND. ParaSketch: parallel tensor factorization via sketching. In: *Proceedings of the 2018 SIAM International Conference on Data Mining (SDM)*. (2018). p. 396–404.
18. Jin R, Kolda TG, Ward R. Faster johnson–lindenstrauss transforms via kronecker products. *Inf Inference*. (2020) 10:1533–62. doi: 10.1093/imaiai/iaaa028
19. Mahoney MW, Maggioni M, Drineas P. Tensor-CUR decompositions for tensor-based data. *SIAM J Matrix Anal Appl*. (2008). 30:957–87. doi: 10.1137/060665336
20. Oseledets I, Savostianov DV, Tyrtyshnikov E. Tucker dimensionality reduction of three-dimensional arrays in linear time. *SIAM J Matrix Anal Appl*. (2008) 30:939–56. doi: 10.1137/060655894
21. Caiafa CF, Cichocki A. Generalizing the column-row matrix decomposition to multi-way arrays. *Linear Algebra Appl*. (2010) 433:557–73. doi: 10.1016/j.laa.2010.03.020
22. Goreinov SA, Tyrtyshnikov EE, Zamarashkin NL. A theory of pseudoskeleton approximations. *Linear Algebra Appl*. (1997) 261:1–21. doi: 10.1016/S0024-3795(96)00301-1
23. Kolda TG. *Multilinear Operators for Higher-Order Decompositions*. Albuquerque, NM; Livermore, CA: Sandia National Laboratories (2006).
24. De Lathauwer L, De Moor B, Vandewalle J. A multilinear singular value decomposition. *SIAM J Matrix Anal Appl*. (2000) 21:1253–78. doi: 10.1137/S0895479896305696
25. Oseledets I. Tensor-train decomposition. *SIAM J Sci Comput*. (2011) 33:2295–317. doi: 10.1137/090752286
26. Vervliet N, Debals O, Sorber L, Van Barel M, De Lathauwer L. *Tensorlab 3.0*. (2016). Available online at: https://www.tensorlab.net.
27. Sorber L, Van Barel M, De Lathauwer L. Structured data fusion. *IEEE J Select Top Signal Process*. (2015) 9:586–600. doi: 10.1109/JSTSP.2015.2400415
28. Domanov I, De Lathauwer L. On the uniqueness of the canonical polyadic decomposition of third-order tensors- Part I: Basic results and uniqueness of one factor matrix. *SIAM J Matrix Anal Appl*. (2013) 34:855–75. doi: 10.1137/120877234
29. Chiantini L, Ottaviani G. On generic identifiability of 3-tensors of small rank. *SIAM J Matrix Anal Appl*. (2012) 33:1018–37. doi: 10.1137/110829180
30. Sidiropoulos N, De Lathauwer L, Fu X, Huang K, Papalexakis EE, Faloutsos C. Tensor decomposition for signal processing and machine learning. *IEEE Trans Signal Process*. (2017) 65:3551–82. doi: 10.1109/TSP.2017.2690524
31. Kruskal JB. Three-way arrays: rank and uniqueness of trilinear decompositions, with application to arithmetic complexity and statistics. *Linear Algebra Appl*. (1977) 18:95–138. doi: 10.1016/0024-3795(77)90069-6
32. Vannieuwenhoven N, Vandebril R, Meerbergen K. A new truncation strategy for the higher-order singular value decomposition. *SIAM J Sci Comput*. (2012) 34:A1027–52. doi: 10.1137/110836067
33. Vervliet N, Debals O, De Lathauwer L. Tensorlab 3.0 – Numerical optimization strategies for large-scale constrained and coupled matrix/tensor factorization. In: *Proceedings of the 50th Asilomar Conference on Signals, Systems and Computers*. Pacific Grove, CA (2016). p. 1733–8.
34. Halko N, Martinsson PG, Tropp JA. Finding structure with randomness: probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Rev*. (2011) 53:217–88. doi: 10.1137/090771806
35. Vervliet N, De Lathauwer L. A randomized block sampling approach to canonical polyadic decomposition of large-scale tensors. *IEEE J Select Top Signal Process*. (2016) 10:284–95. doi: 10.1109/JSTSP.2015.2503260
36. Che M, Wei Y. Randomized algorithms for the approximations of Tucker and the tensor train decompositions. *Adv Comput Math*. (2019) 45:395–428. doi: 10.1007/s10444-018-9622-8
37. Savostyanov D, Oseledets I. Fast adaptive interpolation of multi-dimensional arrays in tensor train format. In: *The 2011 International Workshop on Multidimensional (nD) Systems*. (2011). p. 1–8.
38. Duarte MF, Davenport MA, Takhar D, Laska JN, Sun T, Kelly KF, et al. Single-pixel imaging via compressive sampling. *IEEE Signal Process Mag*. (2008) 25:83–91. doi: 10.1109/MSP.2007.914730
39. Wagadarikar AA, Pitsianis NP, Sun X, Brady DJ. Video rate spectral imaging using a coded aperture snapshot spectral imager. *Optics Express*. (2009) 17:6368–6388. doi: 10.1364/OE.17.006368
40. Rivenson Y, Stern A. Compressed imaging with a separable sensing operator. *IEEE Signal Process Lett*. (2009) 16:449–52. doi: 10.1109/LSP.2009.2017817

41. den berg EV, Friedlander MP. Probing the Pareto frontier for basis pursuit solutions. *SIAM J Sci Comput*. (2008) 31:890–912. doi: 10.1137/080714488
42. den berg EV, Friedlander MP. *SPGL1: A Solver for Large-Scale Sparse Reconstruction*. (2019). Available online at: https://friedlander.io/spgl1.
43. Yuan X. Generalized alternating projection based total variation minimization for compressive sensing. In: *2016 IEEE International Conference on Image Processing (ICIP)*. (2016). p. 2539–43.
44. Baumgardner MF, Biehl LL, Landgrebe DA. *220 Band AVIRIS Hyperspectral Image Data Set: June 12, 1992*. Indian Pine Test Site (2015).

Conflict of Interest: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's Note: All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher.

frontiers in Applied Mathematics and Statistics

ORIGINAL RESEARCH
published: 30 March 2022
doi: 10.3389/fams.2022.836433

CPD-Structured Multivariate Polynomial Optimization

Muzaffer Ayvaz[1,2]* *and Lieven De Lathauwer*[1,2]

[1] Department of Electrical Engineering (ESAT), KU Leuven, Leuven, Belgium, [2] Group Science, Engineering and Technology, KU Leuven Kulak, Kortrijk, Belgium

We introduce the Tensor-Based Multivariate Optimization (TeMPO) framework for use in nonlinear optimization problems commonly encountered in signal processing, machine learning, and artificial intelligence. Within our framework, we model nonlinear relations by a multivariate polynomial that can be represented by low-rank symmetric tensors (multi-indexed arrays), making a compromise between model generality and efficiency of computation. Put the other way around, our approach both breaks the curse of dimensionality in the system parameters and captures the nonlinear relations with a good accuracy. Moreover, by taking advantage of the symmetric CPD format, we develop an efficient second-order Gauss–Newton algorithm for multivariate polynomial optimization. The presented algorithm has a quadratic per-iteration complexity in the number of optimization variables in the worst case scenario, and a linear per-iteration complexity in practice. We demonstrate the efficiency of our algorithm with some illustrative examples, apply it to the blind deconvolution of constant modulus signals, and the classification problem in supervised learning. We show that TeMPO achieves similar or better accuracy than multilayer perceptrons (MLPs), tensor networks with tensor trains (TT) and projected entangled pair states (PEPS) architectures for the classification of the MNIST and Fashion MNIST datasets while at the same time optimizing for fewer parameters and using less memory. Last but not least, our framework can be interpreted as an advancement of higher-order factorization machines: we introduce an efficient second-order algorithm for higher-order factorization machines.

Keywords: multivariate polynomial, numerical optimization, tensor decomposition, Gauss-Newton algorithm, factorization machines, higher order factorization machines, tensor network, image classification

OPEN ACCESS

Edited by:
André Uschmajew,
Max Planck Institute for Mathematics in the Sciences, Germany

Reviewed by:
Edgar Solomonik,
University of Illinois at Urbana-Champaign, United States
Guillaume Rabusseau,
Université de Montréal, Canada

***Correspondence:**
Muzaffer Ayvaz
muzaffer.ayvaz@kuleuven.be

Specialty section:
This article was submitted to Mathematics of Computation and Data Science, a section of the journal Frontiers in Applied Mathematics and Statistics

Received: *15 December 2021*
Accepted: *28 February 2022*
Published: *30 March 2022*

Citation:
Ayvaz M and De Lathauwer L (2022) CPD-Structured Multivariate Polynomial Optimization. Front. Appl. Math. Stat. 8:836433. doi: 10.3389/fams.2022.836433

1. INTRODUCTION

Many problems in data science, signal processing, machine learning and artificial intelligence (AI) can be thought of determining the nonlinear relationship between input and output data. Several strategies have been developed to efficiently model these nonlinear interactions. However, due to the higher-order nature of input and output data, developing scalable algorithms to model these nonlinear interactions is a challenging research direction. Another major issue is the large number of system parameters needed to model the physical phenomena under consideration. For example, large numbers of layers and neurons are needed in deep neural networks (DNNs). Multivariate polynomials are also utilized to model nonlinear continuous functions. However, this approach suffers from an exponential increase in the number of coefficients with the degree of the polynomial. This is known as the curse of dimensionality and is a major drawback that inhibits the development of efficient algorithms.

Tensor decompositions such as canonical polyadic decomposition (CPD) and tensor trains (TT) are promising tools for breaking the curse of dimensionality. Tensors are multi-indexed arrays. They preserve the higher-order structure which is inherent in data, are able to model nonlinear interactions, and can be decomposed uniquely under mild conditions [1–3]. Efficient numerical optimization algorithms have been developed for tensor decompositions. In the context of CPD, the Gauss–Newton algorithm using both line search and trust-region frameworks have been effectively implemented by exploiting the CPD structure [4–6]. A low complexity damped Gauss-Newton algorithm has also been proposed [7]. Moreover, a randomized block sampling approach has been proposed which achieves linear time complexity for the CPD of large tensors by utilizing the Gauss–Newton algorithm [8]. Many data science problems such as latent factor analysis have been solved by reformulating them as tensor decomposition problems [9–12]. An inexact Gauss–Newton algorithm has been proposed for scaling the CPD of large tensors with non-least-squares cost functions [13]. Moreover, generalized Gauss–Newton algorithm with its efficient parallel implementation has been proposed for tensor completion with generalized loss functions [14]. Our aim in this work is to extend the efficient numerical approaches to a broader class of problems that includes not only tensor decompositions but also the optimization of multilinear/polynomial cost functions. Examples include, but are not limited to matrix and tensor eigenvalue problems, nonlinear dimensionality reduction, nonlinear blind source separation, multivariate polynomial regression, and classification problems.

In this study, we develop a framework called Tensor-Based Multivariate Polynomial Optimization (TeMPO) to deal with nonlinear optimization problems commonly encountered in signal processing, machine learning and artificial intelligence. A preliminary version, where only rank-1 CPD is considered with application in blind identification, appeared as the conference paper [15]. In the TeMPO framework, these nonlinear functions are approximated or modeled by multivariate polynomials. Then, low-rank tensors are used to represent the polynomial under consideration. This approach reduces the number of parameters that define the system, and hence enables us to develop efficient numerical optimization algorithms. To further elaborate on the proposed methodology, let us consider the optimization problem

$$\min_{p} \quad l(p(\mathbf{z}), \boldsymbol{\theta}), \tag{1}$$

where $l : \mathcal{R} \times \mathcal{R}^M \rightarrow \mathcal{R}^+$ denotes a loss function such as the mean squared error, $p : \mathcal{R}^I \rightarrow \mathcal{R}$ denotes an unknown multivariate polynomial, $\mathbf{z} \in \mathcal{R}^I$ denotes input data, and $\boldsymbol{\theta} \in \mathcal{R}^M$ denotes output data. We compactly represent the polynomial $p(\mathbf{z})$ through low-rank tensors. One possible way to do this is to write the polynomial as a sum of homogeneous polynomials as follows:

$$p(\mathbf{z}) := \sum_{j=0}^{d} \mathcal{T}_j \mathbf{z}^j, \tag{2}$$

where $\mathcal{T}_j$ denotes a low-rank tensor of order j, and $\mathcal{T}_j \mathbf{z}^j$ denotes the mode-n product (see Section 2.1) of a tensor $\mathcal{T}_j$ and the vector $\mathbf{z}$ for all modes. As by convention, $\mathcal{T}_0$ is assumed to be scalar and $\mathbf{z}^0$ is assumed to be scalar 1. From now on, we call (2) a type I model. We can represent a multivariate polynomial with a single tensor by utilizing a process called homogenization, and augmenting the independent variable $\mathbf{z}$ by a constant 1 as

$$p(\widetilde{\mathbf{z}}) := \mathcal{W} \widetilde{\mathbf{z}}^d, \tag{3}$$

where $\mathcal{W}$ is a tensor of order d, and $\widetilde{\mathbf{z}} = [1; \mathbf{z}]$. Hereafter, we call (3) a type II model.

An n-variate polynomial of degree d has $\mathcal{O}\left(n^d\right)$ coefficients. This exponential dependence on d is the so-called curse of dimensionality. In the TeMPO framework, we break the curse of dimensionality by assuming low-rank structure in the coefficient tensors. For example, when rank- R symmetric CPD structure is used, the number of parameters needed to represent the n-variate polynomial of degree d is ndR which is linear in the number of variables. Several low-rank structures for tensors have been introduced in the literature [1, 2, 16], e.g., canonical polyadic decomposition (CPD), Tucker decomposition, hierarchical Tucker decomposition (HT) [17], tensor train decomposition (TT) [18]. All of these structures can be incorporated into the TEMPO framework; however, in this paper we restrict ourselves to symmetric CPDs. Note that different types of low-rank structure allow us to represent different sub-classes of polynomials. Of course, different representations differ in storage space, and computational complexity. A more detailed exposition will be given in Section 3.2. Note also that the type I model allows us to constrain each term separately while the type II model does not. Therefore, the type I model is a more general representation of multivariate polynomials which may provide better results depending on the applications.

Besides breaking the curse of dimensionality, exploiting low-rank representations of tensors enables us to derive efficient expressions for objective function and gradient evaluations. These then lead us to develop scalable algorithms. We apply our framework for image classification by adapting the second-order Gauss–Newton algorithm and exploiting the symmetric CPD structure in two different tensor representations of multivariate polynomials. We show that the TeMPO framework with symmetric CPD structure achieves similar or better accuracy than various methods such as MLPs, and tensor networks with different structures for the classical MNIST and Fashion MNIST datasets while using fewer parameters and therefore less memory.

Related Work

Several tensor-based methods have been reported in the literature for regression and classification, two problems that are in the class of problems (1). In most of these approaches, a linear model

$$y = \langle \mathcal{W}, \mathcal{X} \rangle + b, \tag{4}$$

is used where $\mathcal{W}$ denotes a weight tensor and $\mathcal{X}$ represents nonlinear features of the input data. This model corresponds to the type II model when a symmetric CPD structure is imposed

on the weight tensor $\mathcal{W}$ and $\mathcal{X}$ is composed of polynomial features of input data. Clearly, imposing different structures to the weight tensor $\mathcal{W}$ and using different nonlinear features in the tensor $\mathcal{X}$ leads to a different representation of the nonlinear interaction between input data and output data. For example, *exponential machines* utilize the tensor train format in the weight tensor with a norm regularization term in the optimization [19]. In this approach, the Riemannian gradient descent algorithm is used for solving the optimization problem. In a similar approach, tensor trains is used with the feature map $\phi(x_j) = \left[\cos\left(\frac{\pi x_j}{2}\right), \sin\left(\frac{\pi x_j}{2}\right)\right]$, by using the density matrix renormalization group (DMRG) algorithm and the first-order ADAM algorithm for the optimization of different cost functions [20, 21]. The same feature map is also used for the linear model (4) by imposing projected entangled pair states (PEPS) structure on the weight tensor $\mathcal{W}$ [22]. The CPD format in model (4) has also been studied in the realm of tensor regression with the Frobenius norm and group sparsity norm regularization terms while using a coordinate-descent approach [23]. A similar model is also considered by utilizing the symmetric CPD format and the second-order Gauss–Newton algorithm with algebraic initialization for multivariate polynomial regression [24]. Several approaches have been proposed that utilize CPD or Tucker formats in tensor regression that use different regularization strategies to prevent the overfitting [25, 26]. Also, the hierarchical Tucker (HT) format has been used in the tensor regression context for the generalized linear model (GLM) $y = \boldsymbol{\alpha}^{\mathrm{T}}\mathbf{x} + \langle \mathcal{W}, \mathcal{X} \rangle$. This approach was successfully applied to brain imaging data sets and uses a block relaxation algorithm, which solves a sequence of lower dimensional optimization problems [27].

Similarly, several models related to the type I model are considered in various settings. For example, Kar and Karnick use random polynomial features and parameterize the coefficients of the polynomial under consideration [28]. The parameterization used in this approach has been shown to be equivalent to imposing the CPD format to the weight tensor $\mathcal{W}$ [29]. Another approach is *factorization machines* which use a multivariate polynomial kernel in the realm of support vector machines (SVM) [30]. For second-order factorization machines a first-order stochastic gradient descent algorithm has been proposed. This approach has a linear time complexity. Higher-order factorization machines use the ANOVA kernel to achieve a linear time complexity and have been successfully applied to link prediction models using stochastic gradient descent [31]. The ANOVA kernel does not use symmetric tensors in the representation and instead only considers combinations of distinct features [31]. Also, factorization machines in the symmetric CPD format have been considered using first-order and BFGS type algorithms [32]. *Tensor machines* generalize both the Kar-Karnick random features approach and factorization machines. It has been shown that these approaches correspond to specific types of tensor machines in the CPD format. Further, it has been shown that empirical risk minimization is an efficient method for finding locally optimal tensor machines if the optimization algorithm avoids saddle points [29].

As can be seen from the literature summary above, one of the differences between our approach and the above methods is the model used. The type I model (2) has not been examined with the symmetric CPD structure in the weight tensors, to the best of our knowledge. Another difference of our approach from the above methods is the algorithm used. While first-order algorithms are used in most of these approaches, we utilize the second-order batch Gauss–Newton (GN) algorithm. Although first-order methods have the advantage of lower per-iteration complexity, second-order GN algorithms generally require fewer iterations to converge and fewer hyperparameters to be optimized. Moreover, the GN algorithm using trust-region is more robust in the sense that it converges to a (local) minimum for any starting point under mild conditions and it is less prone to swamps (many iterations with little to no improvement) [5, 6, 33].

We summarize our contributions as follows:

- We develop a TeMPO framework that is able to solve many nonlinear problems with ubiquitous applications in signal processing, machine learning and artificial intelligence. Moreover, we develop an efficient second-order Gauss–Newton algorithm for optimizing multivariate polynomials in the CPD format.
- We determine the conditions where the tensorized linear model (4) with polynomial features and the multivariate polynomial model (2) coincide when the symmetric CPD format is used in their representations.
- We show that TeMPO achieves similar or better accuracy than various methods such as multilayer perceptrons (MLPs), tensor networks with different architectures including tensor trains (TT), tree tensor networks, and projected entangled pair states (PEPS). We also show that TeMPO requires the optimization for fewer parameters and less memory than these methods for the classification of the MNIST and Fashion MNIST datasets.
- Last but not least, our framework can be interpreted as an advancement of higher-order factorization machines; we introduce an efficient second-order Gauss–Newton algorithm for higher-order factorization machines.

The remaining part of this article is organized as follows. In Section 2, we describe notation and background information concerning tensors. In Section 3, we describe the TeMPO framework in a more detailed manner. Section 3 also covers the details of representation of polynomials by symmetric CPD structured tensors. In Section 3, we also show how to exploit the symmetric CPD structure to obtain efficient expressions for the gradient and Jacobian-vector products which are necessary for the Gauss–Newton algorithm. The formulation of the image classification problem in the context of TeMPO, numerical experiments and related discussions will be covered in Section 4. We conclude our paper with future remarks in the last section.

2. PRELIMINARIES

2.1. Notation

A tensor is a higher-order generalization of a vector (first-order) and a matrix (second-order). Following established conventions, we denote scalars, vectors, matrices, and tensors by a, $\mathbf{a}$, $\mathbf{A}$, and $\mathcal{A}$,

respectively. The transpose of a matrix $\mathbf{A}$ is denoted as $\mathbf{A}^{\mathrm{T}}$. The ith column vector of a matrix $\mathbf{A}$ is denoted as $\mathbf{a}_i$, i.e., $\mathbf{A} = (\mathbf{a}_1, \mathbf{a}_2 \ldots)$. The entry with row index i and column index j in a matrix $\mathbf{A}$, i.e., $(\mathbf{A})_{ij}$, is denoted by a_{ij}. Similarly, $(\boldsymbol{\mathcal{A}})_{i_1 i_2 \ldots i_N}$ is denoted by $a_{i_1 i_2 \ldots i_N}$. Diag($\mathbf{a}$) denotes the diagonal matrix whose entries are composed from the vector $\mathbf{a}$. On the other hand, diag($\mathbf{A}$) denotes a vector composed from the diagonal elements of $\mathbf{A}$. The vectorization operator vec($\mathbf{A}$) for $\mathbf{A} \in \mathbb{K}^{I\times J}$ stacks all the columns of $\mathbf{A}$ into a column vector $\mathbf{a} \in \mathbb{K}^{IJ}$. The reverse operation unvec($\mathbf{a}$) reshapes a vector $\mathbf{a}$ into a matrix $\mathbf{A} \in \mathbb{K}^{I\times J}$. The identity matrix of size $(K \times K)$ is denoted by $\mathbf{I}_K$. A vector of length K with all entries equal to 1 is denoted by $\mathbf{1}_K$. The l_2 norm of a vector $\mathbf{a}$ is denoted by $\|\mathbf{a}\|_2$. The row-wise and column-wise concatenation of two vectors $\mathbf{a}$ and $\mathbf{b}$ is denoted by $[\mathbf{a}, \mathbf{b}]$ and $[\mathbf{a}; \mathbf{b}]$, respectively. The outer product, Kronecker product, Khatri–Rao product, and Hadamard product are denoted by $\circledast$, $\otimes$, $\odot$, and $*$, respectively. The nth power of a vector $\mathbf{x}$ with respect to Kronecker product is defined as $\mathbf{x}^{\otimes n} = \mathbf{x} \otimes \mathbf{x}^{\otimes(n-1)}$, with $\mathbf{x}^{\otimes 0} = 1$. Similarly, $\mathbf{x}^{\odot n}$ and $\mathbf{x}^{*n}$ denotes the nth power of vector $\mathbf{x}$ with respect to Khatri–Rao product and Hadamard product, respectively. The mode-n product of a tensor $\boldsymbol{\mathcal{A}} \in \mathbb{K}^{I_1\times I_2\times \ldots \times I_N}$ (with $\mathbb{K}$ meaning either $\mathcal{R}$ or $\mathbb{C}$) and a vector $\mathbf{x} \in \mathbb{K}^{I_n}$, denoted by $\boldsymbol{\mathcal{A}} \cdot_n \mathbf{x}^{\mathrm{T}}$, is defined element-wise as $\left(\boldsymbol{\mathcal{A}} \cdot_n \mathbf{x}^{\mathrm{T}}\right)_{i_1 i_2 \cdots i_{n-1} i_{n+1} \cdots i_N} = \sum_{i_n=1}^{I_N} a_{i_1 i_2 \cdots i_n \cdots i_N} x_{i_n}$. The mode-$n$ product of a tensor $\boldsymbol{\mathcal{A}} \in \mathbb{K}^{I\times I\times \ldots \times I}$ of order k and a vector $\mathbf{x} \in \mathbb{K}^I$ for all modes is defined as

$$\boldsymbol{\mathcal{A}}\mathbf{x}^k \overset{\text{def}}{=} \boldsymbol{\mathcal{A}} \cdot_1 \mathbf{x}^{\mathrm{T}} \cdot_2 \mathbf{x}^{\mathrm{T}} \ldots \cdot_k \mathbf{x}^{\mathrm{T}}.$$

A mode-n vector or mode-n fiber of a tensor $\boldsymbol{\mathcal{A}} \in \mathbb{K}^{I_1\times I_2\times \ldots \times I_N}$ is a vector obtained by fixing every index except the nth. The mode-n matricization of $\boldsymbol{\mathcal{A}}$ is a matrix $\mathbf{A}_{[n;N,N-1,\ldots,n+1,n-1,\ldots,1]}$ collecting all the mode-n vectors as its columns. For example, an entry $a_{i_1 i_2 i_3}$ of a tensor $\boldsymbol{\mathcal{A}} \in \mathbb{K}^{I\times J\times K}$ is mapped to the (i_2, q) entry of the matrix $\mathbf{A}_{[2;3,1]}$ with $q = i_1 + (i_3 - 1)I$. The binomial coefficient is denoted by $C_n^k = \frac{n!}{(n-k)!k!}$. Some useful definitions are listed below.

Definition 1 (Symmetric Tensor). *A tensor $\boldsymbol{\mathcal{A}} \in \mathbb{K}^{I\times I\times \ldots \times I}$ of order k is called symmetric if its entries are invariant under the permutation of its indices.*

As a consequence of this definition, the matrix representations of symmetric tensors in different modes are all equal.

Definition 2 (Rank of a Tensor). *A rank-1 tensor of order N is the outer product of N nonzero vectors. The rank of a tensor is equal to the minimal number of rank-1 terms that yield the tensor as their sum.*

Definition 3 (Kronecker Product). *Given two matrices $\boldsymbol{A} \in \mathbb{K}^{I\times J}$ and $\boldsymbol{B} \in \mathbb{K}^{K\times L}$, their Kronecker product is*

$$\boldsymbol{A} \otimes \boldsymbol{B} = \begin{bmatrix} a_{1,1}\boldsymbol{B} & \cdots & a_{1,J}\boldsymbol{B} \\ \vdots & \ddots & \vdots \\ a_{I,1}\boldsymbol{B} & \cdots & a_{I,J}\boldsymbol{B} \end{bmatrix} \in \mathbb{K}^{IK\times JL}.$$

Definition 4 (Khatri–Rao Product). *Given two matrices $\boldsymbol{A} \in \mathbb{K}^{I\times K}$ and $\boldsymbol{B} \in \mathbb{K}^{J\times K}$ with the same number of columns, their Khatri–Rao product, also known as columnwise Kronecker product, is*

$$\boldsymbol{A} \odot \boldsymbol{B} = [\boldsymbol{a}_1 \otimes \boldsymbol{b}_1, \boldsymbol{a}_2 \otimes \boldsymbol{b}_2, \ldots, \boldsymbol{a}_K \otimes \boldsymbol{b}_K] \in \mathbb{K}^{IJ\times K},$$

where $\boldsymbol{a}_i$ and $\boldsymbol{b}_i$ denote the ith column of the matrices $\boldsymbol{A}$ and $\boldsymbol{B}$, respectively.

Definition 5 (Hadamard Product). *Given two matrices $\boldsymbol{A} \in \mathbb{K}^{I\times J}$ and $\boldsymbol{B} \in \mathbb{K}^{I\times J}$ with the same size, their Hadamard product is the elementwise product, i.e.,*

$$\boldsymbol{A} * \boldsymbol{B} = \begin{bmatrix} a_{1,1}b_{1,1} & \cdots & a_{1,J}b_{1,J} \\ \vdots & \ddots & \vdots \\ a_{I,1}b_{I,1} & \cdots & a_{I,J}b_{I,J} \end{bmatrix} \in \mathbb{K}^{I\times J}.$$

The following properties will be useful for our derivations.

Property 1. *Let $\boldsymbol{A} \in \mathbb{K}^{I\times J}, \boldsymbol{X} \in \mathbb{K}^{J\times K}, \boldsymbol{B} \in \mathbb{K}^{K\times L}$. Then*

$$vec(\boldsymbol{AXB}) = \left(\boldsymbol{B}^T \otimes \boldsymbol{A}\right) vec(\boldsymbol{X}) \in \mathbb{K}^{IL}.$$

Moreover, if $\boldsymbol{X} \in \mathbb{K}^{J\times J}$ is a diagonal matrix and $\boldsymbol{B} \in \mathbb{K}^{J\times L}$, then

$$vec(\boldsymbol{AXB}) = \left(\boldsymbol{B}^T \odot \boldsymbol{A}\right) diag(\boldsymbol{X}) \in \mathbb{K}^{IL}.$$

Property 2. *Let $\boldsymbol{A} \in \mathbb{K}^{I\times J}, \boldsymbol{B} \in \mathbb{K}^{K\times J}, \boldsymbol{C} \in \mathbb{K}^{I\times L}$, and $\boldsymbol{D} \in \mathbb{K}^{K\times L}$. Then*

$$(\boldsymbol{A} \odot \boldsymbol{B})^T (\boldsymbol{C} \odot \boldsymbol{D}) = \left(\boldsymbol{A}^T\boldsymbol{C}\right) * \left(\boldsymbol{B}^T\boldsymbol{D}\right) \in \mathbb{K}^{J\times L}.$$

Property 3. *For matrices $\boldsymbol{A} \in \mathcal{R}^{I\times J}$ and $\boldsymbol{B} \in \mathcal{R}^{J\times K}$, and for the function $f(\boldsymbol{A}, \boldsymbol{B}) = \boldsymbol{AB}$, the following equations hold:*

$$\frac{\partial vec(f(\boldsymbol{A},\boldsymbol{B}))}{\partial vec(\boldsymbol{A})} = \boldsymbol{B}^T \otimes \boldsymbol{I}_I, \quad \frac{\partial vec(f(\boldsymbol{A},\boldsymbol{B}))}{\partial vec(\boldsymbol{B})} = \boldsymbol{I}_K \otimes \boldsymbol{A}.$$

2.2. Canonical Polyadic Decomposition

Here, we will briefly describe the canonical polyadic decomposition. A more detailed description of CPD can be found in [1] and references therein. The CPD writes a tensor $\boldsymbol{\mathcal{T}} \in \mathbb{K}^{I_1\times I_2\times \ldots \times I_N}$ as a sum of R rank-1 tensors and is denoted by $\left[\!\left[\mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}\right]\!\right]$, with its factor matrices $\mathbf{U}^{(n)} \in \mathbb{K}^{I_n\times R}$, where R equals the rank of the tensor. This is a shortcut notation for

$$\boldsymbol{\mathcal{T}} = \sum_{r=1}^{R} \mathbf{u}_r^{(1)} \circledast \mathbf{u}_r^{(2)} \circledast \ldots \circledast \mathbf{u}_r^{(N)},$$

where $\mathbf{u}_r^{(n)}$ denotes the rth column of the factor matrix $\mathbf{U}^{(n)}$. CPD is essentially unique under mild conditions [34–37], and has found many applications in signal processing and machine learning [1].

For symmetric tensors, all the factor matrices are equal, i.e.,

$$\boldsymbol{\mathcal{T}} = \left[\!\left[\mathbf{U}, \mathbf{U} \ldots, \mathbf{U}; \mathbf{c}^{\mathrm{T}}\right]\!\right] = \sum_{r=1}^{R} c_r(\mathbf{u}_r \circledast \mathbf{u}_r \circledast \ldots \circledast \mathbf{u}_r) \in \mathbb{K}^{I\times I\times \ldots \times I},$$

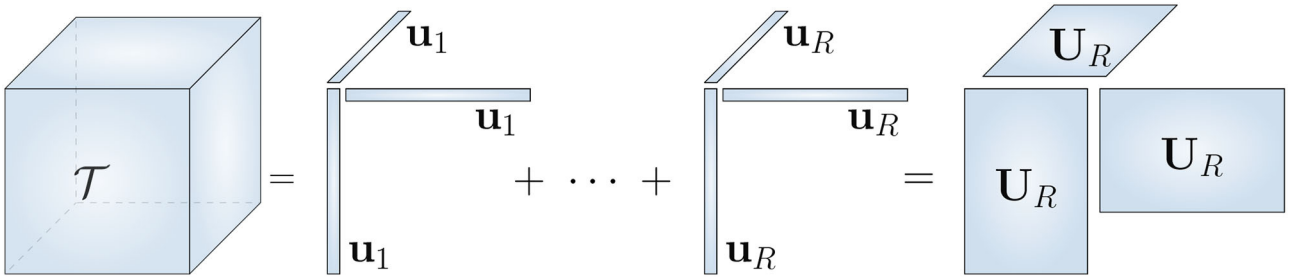


FIGURE 1 | Polyadic decomposition of a third order symmetric tensor $\mathcal{T}$. It is called canonical (CPD) if R is equal to the rank of $\mathcal{T}$, i.e., R is minimal. It allows compact representation of polynomials.

where $\mathbf{U} \in \mathbb{K}^{I \times R}$, and $\mathbf{c} \in \mathbb{K}^R$ is a vector of weights which allows us to give minus signs to the factors for even-degree symmetric tensors, see **Figure 1**. The matrix unfolding of a symmetric CPD is given by

$$\mathbf{T} = \mathbf{U}\mathrm{Diag}(\mathbf{c})(\mathbf{U} \odot \mathbf{U} \odot \cdots \odot \mathbf{U})^{\mathrm{T}}.$$

3. TENSOR-BASED MULTIVARIATE POLYNOMIAL OPTIMIZATION

The primary aim of the TeMPO framework is to develop efficient algorithms for modeling nonlinear phenomena commonly encountered in the areas of signal processing, machine learning, and artificial intelligence [15]. To achieve this, we assume structure in the nonlinear function $f : \mathcal{R}^I \to \mathcal{R}^N$ that maps the input data to output data. In our framework, we first assume smoothness in f and approximate it as multivariate polynomial $p : \mathcal{R}^I \to \mathcal{R}^N$. Then, we approximate p with low-rank tensors. This allows us to achieve efficiency both in storing the coefficients of the approximation and in performing computations with those coefficients. Although any continuous function on a compact domain can be approximated by polynomials arbitrarily well according to the Stone–Weierstrass theorem, polynomial approximations used in practice can pose several numerical issues such as the Runge phenomenon. Several strategies have been proposed to overcome these numerical issues, such as using different polynomial bases and Tikhonov regularization [38, 39]. In this work, we will focus more on computational issues of the TeMPO framework; however, it is possible to incorporate these strategies with TeMPO using slight modifications. In the remaining part of this section, we describe the scope of TeMPO. Then we will describe two types of tensor representations of multivariate polynomials where the symmetric CPD structure is imposed on the coefficient tensors. Next we will briefly describe the Gauss–Newton algorithm using the dogleg trust-region method and show how to exploit the symmetric CPD structure in the computation of Jacobian and Jacobian-vector products that are necessary for the Gauss–Newton algorithm.

3.1. Scope of the TeMPO Framework

The TeMPO framework concerns optimization problems with continuous cost functions on compact domains, namely multilinear/polynomial cost functions with or without additional constraints, which is a more general setting than tensor decomposition or retrieval of a tensor factorization. To better describe the scope, let us consider the following class of objective functions:

$$l(\boldsymbol{\theta}, p(\mathbf{z})), \tag{5}$$

where $l : \mathcal{R} \times \mathcal{R}^M \to \mathcal{R}^+$ denotes the performance measure of the model to be optimized, $p : \mathcal{R}^I \to \mathcal{R}$ denotes a multivariate polynomial represented by low-rank tensors, $\mathbf{z} \in \mathcal{R}^I$ denotes input data, and $\boldsymbol{\theta} \in \mathcal{R}^M$ denotes output data. A broad range of objective functions are in the class of (5). For example, the objective function for the estimation of the CPD of a third-order tensor $\mathcal{T}$ can be written as

$$\frac{1}{2}\left\|\boldsymbol{\theta} - p(\mathbf{A}, \mathbf{B}, \mathbf{C})\right\|_2^2, \quad \text{where} \quad p(\mathbf{A}, \mathbf{B}, \mathbf{C}) = \mathrm{vec}\left(\llbracket \mathbf{A}, \mathbf{B}, \mathbf{C} \rrbracket\right).$$

Other tensor decomposition problems, such as block term decomposition (BTD), also fit into TeMPO. The symmetric best rank-1 approximation problem [40], which can also be formulated as

$$\max_{\mathbf{z} \in \mathcal{R}^I} \quad \mathcal{T}\mathbf{z}^d, \quad \text{subject to} \quad \|\mathbf{z}\| = 1, \tag{6}$$

is another example problem that fits into the framework. Note that (6) is expressed as the maximization of an objective function, rather than as the decomposition of a tensor; indeed TeMPO allows one to address more general problems. For the symmetric best rank-1 approximation problem, several approaches such as higher-order power method [40], generalized Rayleigh–Newton iteration and the alternating least squares methods [41], SVD-based algorithms [42], semi-definite relaxations [43] have been proposed. Problems from unsupervised learning such as nonlinear dimensionality reduction, manifold learning, nonlinear blind source separation, and nonlinear independent component analysis also fit into TeMPO. Similarly, problems from supervised learning fit into TeMPO as well. In this work, we will focus on the regression and classification problem and derive expressions for Jacobian and Jacobian-vector products, which are necessary for the Gauss–Newton algorithm. However, the derivations here can be extended to the other problems without much effort.

Given data points $(y_k, \mathbf{z}_k)$, the regression problem can be formulated within the TeMPO framework for the type I model as

$$\begin{aligned} &\min_{\mathcal{T}_0,\ldots,\mathcal{T}_d} \; p(\mathcal{T}_0,\ldots,\mathcal{T}_d,\mathbf{Z}) \\ &\quad = \min_{\mathcal{T}_0,\ldots,\mathcal{T}_d} \; \frac{1}{2}\sum_{k=1}^{K}\left(y_k - \mathcal{T}_0 - \sum_{j=1}^{d}\mathcal{T}_j\mathbf{z}_k^j\right)^2, \\ &\quad \text{subject to} \quad \operatorname{rank}(\mathcal{T}_j) = R_j \end{aligned} \tag{7}$$

where $R_j \in \mathbb{N}^+$ is a small integer, $\mathcal{T}_j \in \mathcal{R}^{I\times I\times\ldots\times I}$ denotes the low-rank structured coefficient tensor of order j to be optimized, $\mathcal{T}_0 \in \mathcal{R}$ denotes a scalar, $\mathbf{Z} \in \mathcal{R}^{I\times K}$ denotes the data matrix, $\mathbf{z}_k$ denotes the kth column of $\mathbf{Z}$ and K is the number of available data points. For the type II model, the regression problem takes the form

$$\begin{aligned} &\min_{\mathcal{T}} \; p(\mathcal{T},\widetilde{\mathbf{Z}}) = \min_{\mathcal{T}} \; \frac{1}{2}\sum_{k=1}^{K}\left(y_k - \mathcal{T}\widetilde{\mathbf{z}}_i^d\right)^2, \\ &\quad \text{subject to} \quad \operatorname{rank}(\mathcal{T}) = R \end{aligned} \tag{8}$$

where $\mathcal{T}$ denotes the low-rank structured coefficient tensor of order d to be optimized, $\widetilde{\mathbf{Z}} \in \mathcal{R}^{(I+1)\times K}$ denotes the augmented input data matrix, and $\widetilde{\mathbf{z}}_k$ denotes the kth column of $\widetilde{\mathbf{Z}}$, i.e., $\widetilde{\mathbf{z}}_k = [1; \mathbf{z}_k]$.

3.2. Tensor Representation of Polynomials

In this subsection, we examine the type I and type II model in detail. A (symmetric) tensor $\mathcal{T}$ of order d and dimension n can be associated with a homogeneous n-variate polynomial $p(\mathbf{z})$ of degree d [44], as shown in Equation (3).

Type I: Since any polynomial can be written as a sum of homogeneous polynomials of increasing degrees, any polynomial of degree d can be written by using tensors of order up to d, as shown in Equation (2). Note that in the tensor representation of polynomials, any tensor can be assumed to be symmetric without loss of generality. Indeed, any homogeneous polynomial $p(\mathbf{z})$ of degree $d \in \mathbb{N}$ can be represented by a multilinear form $\mathcal{T}\mathbf{z}^d$, where $\mathcal{T} \in \mathbb{K}^{I\times I\times\ldots\times I}$ is a symmetric tensor of order d and $\mathbf{z} \in \mathbb{K}^I$.

To see this, suppose a homogeneous polynomial $p(\mathbf{z})$ is represented as

$$p(\mathbf{z}) = \widetilde{\mathcal{T}}\mathbf{z}^d = \sum_{i_1,i_2\ldots,i_d=1}^{I} \widetilde{t}_{i_1 i_2\ldots i_d} z_{i_1} z_{i_2}\ldots z_{i_d},$$

where $\widetilde{\mathcal{T}} \in \mathbb{K}^{I\times I\times\ldots\times I}$ is a tensor of order d. Since the terms $z_{i_1} z_{i_2}\ldots z_{i_d}$ are invariant under the permutation of indices, we may write

$$\begin{aligned} p(\mathbf{x}) &= \sum_{i_1,i_2\ldots,i_d=1}^{I} t_{i_1 i_2\ldots i_d} z_{i_1} z_{i_2}\ldots z_{i_d}, \\ \text{where} \quad t_{i_1 i_2\ldots i_d} &= \frac{1}{d!}\sum_{(i_1,i_2,\ldots,i_d)\in\Pi(i_1 i_2\ldots i_d)} \widetilde{t}_{i_1 i_2\ldots i_d}, \end{aligned}$$

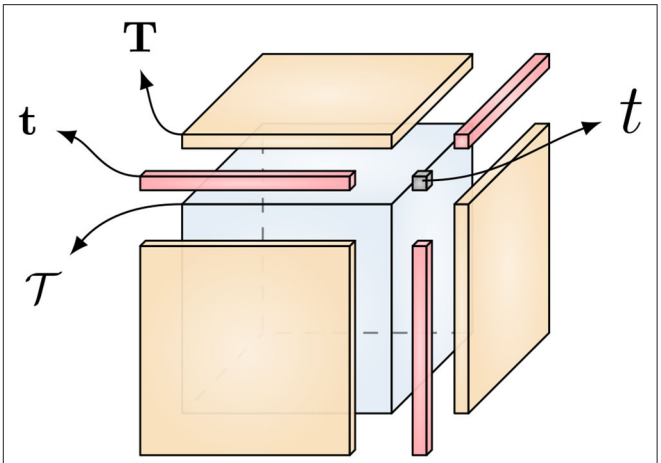


FIGURE 2 | By applying the homogenization process, symmetric tensors can represent the coefficients of non-homogeneous polynomials. For example, by stacking the coefficients t, $\mathbf{t}$, $\mathbf{T}$, and $\mathcal{T}$ of the third degree polynomial into a tensor as shown above, we can represent it with a symmetric third-order tensor. Image reproduced from Debals [46].

here $\Pi(i_1 i_2\ldots i_d)$ denotes the collection of all permutation of indices $(i_1, i_2, \ldots, i_d)$. Since the entries of $\mathcal{T}$ are invariant under the permutation of indices, we can conclude that $\mathcal{T}$ is symmetric.

The above discussion reveals the fact that there are infinitely many representations of a given polynomial. Indeed two representations with tensors $\mathcal{T}$ and $\mathcal{W}$ are equal so long as the summation of the corresponding entries over the permutation of indices remains the same, i.e.,

$$\sum_{(i_1,i_2,\ldots,i_d)\in\Pi(i_1 i_2\ldots i_d)} t_{i_1 i_2\ldots i_d} = \sum_{(i_1,i_2,\ldots,i_d)\in\Pi(i_1 i_2\ldots i_d)} w_{i_1 i_2\ldots i_d}$$

In the ANOVA kernel used in higher-order factorization machines, all $t_{\Pi(i_1 i_2\ldots i_d)}$ are set to zero except $t_{(i_1<i_2<\ldots<i_d)}$ [31], which leads to a sparse representation. In this paper, we use symmetric tensors for two reasons. The first reason is that the CPD of a symmetric tensor can be expressed by a single factor matrix. Therefore, the symmetric CPD representation of multivariate polynomial requires fewer number of parameters in comparison with a non-symmetric representation. The second reason is that there is a rich history of the representation of polynomials with symmetric tensors in the field of algebraic geometry under the name of the Waring problem [45].

Type II: Augmenting the independent variable vector $\mathbf{z}$, by a constant 1[1], i.e., $\widetilde{\mathbf{z}} = [1; \mathbf{z}^{\mathrm{T}}]$ leads to a different representation of non-homogeneous polynomials that uses a single dth order symmetric tensor for the inhomogeneous multivariate polynomial of degree d, as shown in Equation (3). This process is called homogenization [46] and is graphically illustrated in **Figure 2**. If we just use full tensors, the type I and II

[1]Since the weight vector $\mathbf{c}$ is used in the parametrization of tensors, different choices of constant in $\widetilde{\mathbf{z}}$ lead to mathematically equivalent cost functions in the optimization problems. On the other hand, the choice of the constant may imply numerical differences—in situations of this type, one should generally choose a constant that "makes sense for the application."

models are interchangeable. However, it is important to note that *when low-rank structure is imposed on the coefficient tensors,* both representations yield different classes of low-rank multivariate polynomial. Hence, these approaches may lead to different results depending on the application. The former approach requires more parameters since it uses more factor matrices. The difference in the number of parameters should be taken into account to prevent underfitting and overfitting. A more detailed description for storage complexity is given in Section 3.5. Moreover, the type I model allows us to constrain each term in the representation separately. In modeling multivariate polynomials, one might not wish the terms of different order to have some shared structure, in which case one should choose the type I model to work with. Similarly, the type II model should be chosen, if some shared structure is desired in the terms of different order. To further elaborate on the effects of homogenization on the rank of a tensor, let us consider the following proposition.

Proposition 1. *Let $p(\boldsymbol{z}) : \mathcal{R}^I \to \mathcal{R}$ be a multivariate polynomial of order d defined as in equation (2) by a scalar $\mathcal{T}_0$ and symmetric tensors $\mathcal{T}_j \in \mathcal{R}^{I\times I\times\ldots\times I}$ for $j = 1, 2, \ldots, d$. Moreover, let $\mathcal{W} \in \mathcal{R}^{(I+1)\times\ldots\times(I+1)}$ be the corresponding tensor obtained from the homogenization process. The tensors $\mathcal{W}$ and $\mathcal{T}_j$ have the same rank R if and only if the tensors $\mathcal{T}_j$ admit unique CPDs with shared factor matrices and a weight vector $\boldsymbol{c}$, i.e.,*

$$\mathcal{T}_j = \left[\!\left[\boldsymbol{U}, \ldots, \boldsymbol{U}; C_d^j (\boldsymbol{c}^T)^{\odot(d-j)} \right]\!\right], \quad \text{and} \quad \mathcal{T}_0 = \sum_{r=1}^{R} \left((\boldsymbol{c}^T)^{\odot d} \right)_r .$$

Proof 1. *Let the CPD of the tensor $\mathcal{W}$ be defined as $[\![\boldsymbol{V}, \ldots, \boldsymbol{V}]\!]$, where, for convenience but without loss of generality, the weights of the rank-1 terms are assumed to be 1. Since $\mathcal{W}$ is obtained by the homogenization process, partitioning $\boldsymbol{V}$ as $[\boldsymbol{v}^T; \boldsymbol{Q}]$ and using the definition of CPD, we obtain*

$$\mathcal{T}_j = \left[\!\left[\boldsymbol{Q}, \ldots, \boldsymbol{Q}; C_d^j (\boldsymbol{v}^T)^{\odot(d-j)} \right]\!\right], \quad \text{and} \quad \mathcal{T}_0 = \sum_{r=1}^{R} \left((\boldsymbol{v}^T)^{\odot d} \right)_r . \tag{9}$$

Since the CPDs of the tensors $\mathcal{T}_j$ are unique, the equality (9) holds if and only if the equalities $\boldsymbol{Q} = \boldsymbol{U}$ and $\boldsymbol{v} = \boldsymbol{c}$ also hold.

Remark 1. *In the above proof, we assumed that the vector $\boldsymbol{v}$ does not contain any zero elements. Note that if the vector $\boldsymbol{v}$ does contain zero elements, it cancels the corresponding rank-1 terms. Therefore, in that case $rank(\mathcal{W}) > rank(\mathcal{T}_j)$, for $j = 1, \ldots, d-1$. Moreover, the uniqueness of the CPDs of $\mathcal{T}_j$ implies that $rank(\mathcal{W}) \geq rank(\mathcal{T}_j)$. Since the equality $rank(\mathcal{W}) = rank(\mathcal{T}_j)$ holds only when the tensors $\mathcal{T}_j$ have shared factor matrices as described above, we can conclude that in all other cases $rank(\mathcal{W}) > rank(\mathcal{T}_j)$.*

Proposition 1 together with Remark 1 reveals the fact that if $\mathcal{W}$ admits a rank-R CPD, there exists tensors $\mathcal{T}_j$ that admit rank-R_j CPDs with shared factors and $R_j \leq R$. Hence, the expressive power of the type II model is weaker than the type I model, i.e., the type II model requires higher rank values than the type I model to be able to model functions of the same complexity. In other words, the set of polynomials represented by the type II model is a strict subset of the set of polynomials represented by the type I model for the same rank values.

Although we focus in this study on the type I and type II models in the symmetric CPD format, the TeMPO framework is not limited to these. TeMPO collects low-rank tensor representations of multivariate polynomials under a roof by utilizing various other tensor decompositions such as TT, HT, and non-symmetric and partially symmetric CPD formats[2]. In this way, TeMPO breaks the curse of dimensionality and makes it possible to develop second-order efficient algorithms for the optimization of a more general class of multivariate polynomials. Moreover, use of structured tensors and multilinear algebra makes it easy to incorporate other polynomial bases and, more generally, other nonlinear feature maps rather than the standard polynomial bases to the TeMPO framework. From this point of view, TeMPO can be interpreted as a generalization of higher-order factorization machines that use particular types of multivariate polynomials with the standard polynomial bases and utilize first-order and BFGS type algorithms [30–32, 47].

3.3. Gauss–Newton Algorithm

Most standard first-order and second-order numerical optimization algorithms can be used for solving problem (8). Since the objective function under consideration is a least-squares function, we will utilize the second-order batch Gauss–Newton (GN) algorithm using a trust-region to take advantage of its attractive properties such as quadratic convergence near a local optimum point, resistant to swamps, suitable to incorporate constraints easily and eligible to exploit multilinear structure. In the case the objective function is not least squares, the inexact GN algorithm can also be utilized. Below, we briefly describe the GN algorithm using a trust-region, and then derive the expressions for Jacobian and Jacobian-vector products for tensors in the symmetric CPD format. In nonlinear least-squares problems, the objective function is the squared error between a data vector $\mathbf{y}$ and a nonlinear model $m(\mathbf{z})$ [6, 33]:

$$f(\mathbf{z}) = \frac{1}{2} \left\| m(\mathbf{z}) - \mathbf{y} \right\|_2^2 = \frac{1}{2} \mathbf{r}^{\mathrm{T}} \mathbf{r}, \tag{10}$$

where $\mathbf{z} \in \mathcal{R}^I$. The algorithm updates the initial guess iteratively by taking a step length α_k in the direction $\mathbf{p}_k$ at the iteration k, i.e.,

$$\mathbf{z}_k = \mathbf{z}_{k-1} + \alpha_k \mathbf{p}_k,$$

until some stopping criteria are satisfied. Line search and trust-region are the two main approaches used to determine α_k and $\mathbf{p}_k$. Here, we focus on the dogleg trust-region approach. In this approach, one sets $\alpha_k = 1$. Then, given a trust-region of radius δ_k, the GN step $\mathbf{p}_k^{gn}$ and the steepest descent step $\mathbf{p}_k^{sd}$ for the current iteration, the step direction $\mathbf{p}_k$ is determined by the following procedure:

[2]Note that the non-symmetric and partially symmetric CPD formats are fairly straightforward variants of the symmetric CPD format, and derivations presented in Section 3.4 can be generalized to these formats with slight modifications.

Algorithm 1: GN algorithm using dogleg trust-region for the type II model.

Input : $\mathbf{Z}$ – Input data matrix
$\mathbf{y}$ – Vector of values (labels in the classification case) for each data point in $\mathbf{Z}$
$\mathbf{U}, \mathbf{c}$ – Initial factor matrix and weight vector
$\mathcal{T}_0$ – Initial scalar
Output: $\mathbf{U}, \mathbf{c}$ – Optimized factor matrix and weight vector
$\mathcal{T}_0$ – Optimized scalar

while *not converged* **do**
  $\mathbf{r}_k \leftarrow$ Compute residual vector using equations (21) and (22)
  $\mathbf{g}_k \leftarrow$ Compute gradient using equation (27)
  $\mathbf{p}_k \leftarrow$ Solve linear systems of equations (12) using CG method
  $\mathbf{U}, \mathbf{c}, \mathcal{T}_0 \leftarrow$ Update via dogleg trust-region explained in Subsection 3.3
end

- If $\left\|\mathbf{p}_k^{gn}\right\|_2 \leq \delta_k$, then $\mathbf{p}_k = \mathbf{p}_k^{gn}$.
- If $\left\|\mathbf{p}_k^{gn}\right\|_2 > \delta_k$ and $\left\|\mathbf{p}_k^{sd}\right\|_2 > \delta_k$, then $\mathbf{p}_k = \delta_k \mathbf{p}_k^{sd} / \left\|\mathbf{p}_k^{sd}\right\|_2$.
- If $\left\|\mathbf{p}_k^{gn}\right\|_2 > \delta_k$ and $\left\|\mathbf{p}_k^{sd}\right\|_2 \leq \delta_k$, then $\mathbf{p}_k = \tau_k \mathbf{p}_k^{sd} + \beta_k(\mathbf{p}_k^{gn} - \tau_k \mathbf{p}_k^{sd})$, where $\tau_k = -\left\|\mathbf{p}_k^{sd}\right\|_2^2 / \left\|\mathbf{J}_k \mathbf{p}_k^{sd}\right\|_2^2$, and β_k is selected such that $\left\|\mathbf{p}_k\right\|_2 = \delta_k$.

The steepest descent step $\mathbf{p}_k^{sd}$ is given by $-\mathbf{J}_k^{\mathrm{T}}\mathbf{r}_k$. To compute the GN step, a second order Taylor series approximation for the objective function is used. The optimal direction for the GN step $\mathbf{p}_k^{gn}$ can be obtained by solving the optimization problem,

$$\mathbf{p}_k^{gn} = \arg\min_{\mathbf{p}} \widetilde{f}(\mathbf{p}), \quad \text{with} \quad \widetilde{f}(\mathbf{p}) = f(\mathbf{z}_k) + \mathbf{p}^{\mathrm{T}}\mathbf{g}_k + \frac{1}{2}\mathbf{p}^{\mathrm{T}}\mathbf{H}_k\mathbf{p}, \tag{11}$$

where $\mathbf{g}_k$ denotes the gradient and $\mathbf{H}_k$ denotes the Hessian at the current iteration. Setting $\partial\widetilde{f}(\mathbf{p})/\partial\mathbf{p}$ to zero, the solution of (11) can be obtained by solving the linear system of equations

$$\mathbf{H}_k\mathbf{p}_k^{gn} = -\mathbf{g}_k, \quad \text{with} \quad \mathbf{g}_k = \mathbf{J}_k^{\mathrm{T}}\mathbf{r}_k, \tag{12}$$

where $\mathbf{J}_k$ denotes the Jacobian of $f(\mathbf{z}_k)$ at iteration k, and $\mathbf{r}_k = m(\mathbf{z}_k) - \mathbf{y}$. However, in real-life applications, explicit computation of the Hessian is often expensive. To overcome this, GN approximates the Hessian by the Grammian matrix as

$$\mathbf{H}_k \approx \mathbf{J}_k^{\mathrm{T}}\mathbf{J}_k.$$

In this study, we used the conjugate gradient (CG) algorithm for solving (12) together with the dogleg trust-region approach which is implemented in Tensorlab [11]. The overall algorithm is summarized in **Algorithm 1**.

3.4. Exploiting the Symmetric CPD Format

As described above, the GN algorithm minimizes a cost function in the form of Equation (10). The gradient of this objective function can be written as $\mathbf{J}^{\mathrm{T}}\mathbf{r}$, and the Hessian is approximated by $\mathbf{J}^{\mathrm{T}}\mathbf{J}$, where $\mathbf{J}$ is the Jacobian matrix composed of partial derivatives of the residual vector $\mathbf{r}$. Hence, it is sufficient to derive expressions for the Jacobian and Jacobian-vector products. We begin with the first-order derivatives of the multilinear form $\mathcal{T}\mathbf{z}^d$, where $\mathcal{T}$ is in the symmetric CPD format, with respect to its factors and then proceed to the derivation of Jacobian and Jacobian-vector products for problems (7) and (8). The derivations made here can be used in other TeMPO problems with slight modifications.

3.4.1. Derivatives of the Multilinear Form in the Symmetric CPD Format

By using the matrix unfolding of the tensor in the symmetric CPD format and Property 2 of Khatri–Rao product, the multilinear form $\mathcal{T}\mathbf{z}^d$ can be written as

$$\mathcal{T}\mathbf{z}^d = \mathbf{c}^{\mathrm{T}}\left(\mathbf{U}^{\mathrm{T}}\mathbf{z}\right)^{*d}, \tag{13}$$

which will be useful for our derivations below.

Lemma 1. *Let $\mathcal{T} \in \mathbb{K}^{I\times I\times\ldots\times I}$ be a symmetric tensor of order d and its CPD given as $\mathcal{T} = [\![\boldsymbol{U}, \ldots, \boldsymbol{U}; \boldsymbol{c}^T]\!]$. Then the derivative of the multilinear form $\mathcal{T}\boldsymbol{z}^d$ with respect to vec($\boldsymbol{U}$) can be obtained as*

$$\frac{\partial \mathcal{T}\boldsymbol{z}^d}{\partial vec(\boldsymbol{U})} = \left((\boldsymbol{c} * \boldsymbol{w}) \otimes \boldsymbol{z}\right)^T,$$

where $\boldsymbol{w} = d\left(\boldsymbol{U}^T\boldsymbol{z}\right)^{*(d-1)}$.

Proof 2. *The proof immediately follows from Equation (13) and successive application of Property 3.*

Lemma 2. *Let $\mathcal{T} \in \mathbb{K}^{I\times I\times\ldots\times I}$ be symmetric tensor of order d and its CPD is given as $\mathcal{T} = [\![\boldsymbol{U}, \ldots, \boldsymbol{U}; \boldsymbol{c}^T]\!]$. Then the derivative of multilinear form $\mathcal{T}\boldsymbol{z}^d$ with respect to vector $\boldsymbol{c}$ can be obtained as*

$$\frac{\partial \mathcal{T}\boldsymbol{z}^d}{\partial \boldsymbol{c}} = \left(\boldsymbol{z}^T\boldsymbol{U}\right)^{*d}.$$

Proof 3. *The proof immediately follows from Property 3 and Equation (13).*

3.4.2. Exploiting Structure in the Type I Model

Objective Function: The construction of the residual vector $\mathbf{r}$ and the computation of its l_2 norm is sufficient for computing the objective function in (7). By utilizing Property 2 and Equation (13), the residual vector can be expressed as $\mathbf{r} = \mathbf{y} - \boldsymbol{\mu}$, where each entry of the vector $\boldsymbol{\mu} \in \mathcal{R}^K$ is defined as

$$\boldsymbol{\mu}_k = \mathcal{T}_0 + \sum_{j=1}^{d} \mathbf{c}_j^{\mathrm{T}}\mathbf{w}_{j,k}^{*j},$$

in which $\mathbf{w}_{j,k} = \mathbf{U}_j^{\mathrm{T}}\mathbf{z}_k$ with $\mathbf{U}_j \in \mathcal{R}^{I\times R}$, and $\mathbf{w}_{j,k}^{*j}$ denotes the jth elementwise power of the vector $\mathbf{w}_{j,k}$. By defining $\mathbf{W}_j =$

$\left[\mathbf{w}_{j,1}, \mathbf{w}_{j,2}, \ldots, \mathbf{w}_{j,K}\right]$, we can write the residual vector $\mathbf{r}$ in a compact form as

$$\mathbf{r} = \mathbf{y} - \mathcal{T}_0 \cdot \mathbf{1}_K - \sum_{j=1}^{d} \left(\mathbf{c}_j^{\mathrm{T}} \left(\mathbf{W}_j^{*j}\right)\right)^{\mathrm{T}}, \tag{14}$$

Using the above Equation (14), the objective function can be computed as the l_2 norm of the residual vector $\mathbf{r}$.

Jacobian: The Jacobian matrix for problem (7), with the tensors $\mathcal{T}_j$ in their symmetric CPD format, can be written in a compact form as

$$\begin{aligned} \mathbf{J} &= [\mathbf{J}_1; \ldots; \mathbf{J}_K], \quad \text{where} \\ \mathbf{J}_k &= \left[1, \frac{\partial \mathbf{r}_k}{\partial \mathrm{vec}(\mathbf{U}_1)}, \ldots, \frac{\partial \mathbf{r}_k}{\partial \mathrm{vec}(\mathbf{U}_d)}, \frac{\partial \mathbf{r}_k}{\partial \mathbf{c}_1}, \ldots, \frac{\partial \mathbf{r}_k}{\partial \mathbf{c}_d}\right]. \end{aligned} \tag{15}$$

Note that we used the fact $\partial \mathbf{r}_k / \partial \mathcal{T}_0 = 1$ in the above equation. By utilizing Lemma 1 and Lemma 2, the derivative of each term of the residual vector with respect to $\mathbf{U}_j$ and $\mathbf{c}_j$ can be expressed as

$$\frac{\partial \mathbf{r}_k}{\partial \mathrm{vec}(\mathbf{U}_j)} = -j\left[\left(\mathbf{c}_j * \mathbf{w}_{j,k}^{*(j-1)}\right) \otimes \mathbf{z}_k\right]^{\mathrm{T}}, \quad \text{and} \quad \frac{\partial \mathbf{r}_k}{\partial \mathbf{c}_j} = \left(\mathbf{w}_{j,k}^{*j}\right)^{\mathrm{T}}. \tag{16}$$

By defining $\widetilde{\mathbf{W}}_j = -j\left(\mathbf{W}_j^{*(j-1)}\right)$ for $j = 1, \ldots, d$, and $\mathbf{Z} = [\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_K]$, the Jacobian matrix $\mathbf{J}$ in (15) can be obtained in the following compact block form:

$$\mathbf{J} = \left[\mathbf{1}_K, ((\mathbf{C}_1 \widetilde{\mathbf{W}}_1) \odot \mathbf{Z})^{\mathrm{T}}, \ldots, ((\mathbf{C}_d \widetilde{\mathbf{W}}_d) \odot \mathbf{Z})^{\mathrm{T}}, \mathbf{V}\right], \tag{17}$$

where $\mathbf{V}$ is a $K \times d$ block matrix in which each block is defined as $\mathbf{V}_{k,j} = \left(\mathbf{w}_{j,k}^{*j}\right)^{\mathrm{T}}$, $\mathbf{C}_j = \mathrm{Diag}(\mathbf{c}_j)$, and d is the degree of the polynomial under consideration. Since we only need the Jacobian-vector products for the GN algorithm, the explicit construction of the Jacobian matrix is not required. The Jacobian-vector products can be obtained in a more memory-efficient way as described below.

Jacobian-Vector Product: The product of Jacobian $\mathbf{J}$ by a vector $\mathbf{x}$ can be obtained using block matrix operations. The product of each block term by a vector $\mathrm{vec}(\mathbf{X}_j) = \mathbf{x}_j$ can be obtained by utilizing properties 1 and 2 as

$$((\mathbf{C}_j \widetilde{\mathbf{W}}_j) \odot \mathbf{Z})^{\mathrm{T}} \mathbf{x}_j = \left[\left(\mathbf{X}_j^{\mathrm{T}} \mathbf{Z}\right) * (\mathbf{C}_j \widetilde{\mathbf{W}}_j)\right]^{\mathrm{T}} \mathbf{1}_R. \tag{18}$$

Note that the multiplication of a matrix by $\mathbf{1}_R$ from the right is equivalent to summing the columns of the matrix under consideration. Therefore, neither the multiplication by $\mathbf{1}_R$ nor the transposition of the matrix $\left(\mathbf{X}_j^{\mathrm{T}} \mathbf{Z}\right) * (\mathbf{C}_j \widetilde{\mathbf{W}}_j)$ in Equation (18) is necessary to obtain the Jacobian-vector product. Note also that, since the matrices $\mathbf{C}_j$ are diagonal, the product $\mathbf{C}_j \widetilde{\mathbf{W}}_j$ can be obtained in a memory efficient way by multiplying the rows of $\widetilde{\mathbf{W}}_j$ by the corresponding diagonal elements of $\mathbf{C}_j$ without explicitly forming the matrices $\mathbf{C}_j$. Overall, the product of the Jacobian $\mathbf{J}$ and the vector $\mathbf{x}$ can be obtained by partitioning the vector $\mathbf{x}$, i.e., $\mathbf{x} = [x_1; \mathbf{x}_1; \mathbf{x}_2; \ldots; \mathbf{x}_d; \mathbf{x}_v]$, and by using the Equations (17) and (18) as

$$\mathbf{J}\mathbf{x} = x_1 \cdot \mathbf{1}_K + \sum_{j=1}^{d} \left[\left(\mathbf{X}_j^{\mathrm{T}} \mathbf{Z}\right) * (\mathbf{C}_j \widetilde{\mathbf{W}}_j)\right]^{\mathrm{T}} \mathbf{1}_R + \mathbf{V}\mathbf{x}_v,$$

where $\mathbf{X}_j = \mathrm{unvec}(\mathbf{x}_j)$.

Jacobian Transpose -Vector Product and Gradient: In a similar way, block-wise multiplication of the Jacobian transpose $\mathbf{J}^{\mathrm{T}}$ by a vector can be obtained from the expression

$$((\mathbf{C}_j \widetilde{\mathbf{W}}_j) \odot \mathbf{Z})\mathbf{x} = \mathrm{vec}\left(\mathbf{Z}\mathrm{Diag}(\mathbf{x})(\mathbf{C}_j \widetilde{\mathbf{W}}_j)^{\mathrm{T}}\right). \tag{19}$$

Note that right multiplication by a diagonal matrix can be done efficiently by only multiplying the columns of the matrix with the corresponding diagonal elements without explicitly forming the diagonal matrix. Overall, by defining $\boldsymbol{\xi}_j = \mathrm{vec}\left(\mathbf{Z}\mathrm{Diag}(\mathbf{x})(\mathbf{C}_j \widetilde{\mathbf{W}}_j)^{\mathrm{T}}\right)$, we can obtain the product of the Jacobian transpose $\mathbf{J}^{\mathrm{T}}$ and a vector $\mathbf{x}$ in the following form:

$$\mathbf{J}^{\mathrm{T}}\mathbf{x} = \left[\sum_{k=1}^{K} x_k; \boldsymbol{\xi}_1; \boldsymbol{\xi}_2; \ldots; \boldsymbol{\xi}_d; \mathbf{V}^{\mathrm{T}}\mathbf{x}\right]. \tag{20}$$

The gradient can be obtained by the product of the Jacobian transpose $\mathbf{J}^{\mathrm{T}}$ and the residual vector $\mathbf{r}$. Defining $\boldsymbol{\eta}_j = \mathrm{vec}\left(\mathbf{Z}\mathrm{Diag}(\mathbf{r})(\mathbf{C}_j \widetilde{\mathbf{W}}_j)^{\mathrm{T}}\right)$ and utilizing the Equations (19) and (20), we can obtain the gradient as

$$\mathbf{g} = \left[\sum_{k=1}^{K} r_k; \boldsymbol{\eta}_1; \boldsymbol{\eta}_2; \ldots; \boldsymbol{\eta}_d; \mathbf{V}^{\mathrm{T}}\mathbf{r}\right].$$

3.4.3. Exploiting Structure in the Type II Model

Objective Function: The computation of the objective function for the type II model is similar to that of the type I model. Utilizing Property 2 and Equation (13), the residual vector for problem (8) can be obtained as $\mathbf{r} = \mathbf{y} - \boldsymbol{\mu}$ with

$$\boldsymbol{\mu} = \left[\mathbf{c}^{\mathrm{T}}\mathbf{w}_1^{*d};\ \mathbf{c}^{\mathrm{T}}\mathbf{w}_2^{*d};\ \ldots;\ \mathbf{c}^{\mathrm{T}}\mathbf{w}_K^{*d}\right], \tag{21}$$

where $\mathbf{w}_k = \mathbf{U}^{\mathrm{T}}\widetilde{\mathbf{z}}_k$. By defining $\mathbf{W} = [\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_K]$, we can write the residual vector $\mathbf{r}$ in a compact form as

$$\mathbf{r} = \mathbf{y} - \left(\mathbf{c}^{\mathrm{T}}\left(\mathbf{W}^{*d}\right)\right)^{\mathrm{T}}, \tag{22}$$

Using the above Equation (22), the objective function can be computed as the l_2 norm of the residual vector $\mathbf{r}$.

Jacobian: The Jacobian matrix of the cost function in (8) can be defined in a compact form as

$$\mathbf{J} = [\mathbf{J}_1; \mathbf{J}_2; \ldots; \mathbf{J}_K], \quad \text{where} \quad \mathbf{J}_k = \left[\frac{\partial \mathbf{r}_k}{\partial \mathrm{vec}(\mathbf{U})}, \frac{\partial \mathbf{r}_k}{\partial \mathbf{c}}\right]. \tag{23}$$

Utilizing Lemma 1 and Lemma 2 and using the equations in (16), the parts of $\mathbf{J}_k$ in Equation (23) can be written as

$$\frac{\partial \mathbf{r}_k}{\partial \mathrm{vec}(\mathbf{U})} = -d\Big[\Big(\mathbf{c} * \mathbf{w}_k^{*(d-1)}\Big) \otimes \widetilde{\mathbf{z}}_k\Big]^{\mathrm{T}}, \quad \frac{\partial \mathbf{r}_k}{\partial \mathbf{c}} = \Big(\mathbf{w}_k^{*d}\Big)^{\mathrm{T}}.$$

By defining $\widetilde{\mathbf{W}} = -d\left(\mathbf{W}^{*(d-1)}\right)$, $\mathbf{V} = \left[\frac{\partial \mathbf{r}_1}{\partial \mathbf{c}}; \frac{\partial \mathbf{r}_2}{\partial \mathbf{c}}; \ldots; \frac{\partial \mathbf{r}_K}{\partial \mathbf{c}}\right]$, and $\mathbf{Z} = [\widetilde{\mathbf{z}}_1, \widetilde{\mathbf{z}}_2, \ldots, \widetilde{\mathbf{z}}_K]$, the Jacobian matrix can be obtained in the following compact form:

$$\mathbf{J} = \Big[\big((\mathbf{C}\widetilde{\mathbf{W}}) \odot \mathbf{Z}\big)^{\mathrm{T}}, \mathbf{V}\Big]. \tag{24}$$

As mentioned earlier, explicit construction of the Jacobian matrix $\mathbf{J}$ is not required. We only require the Jacobian-vector and Jacobian transpose-vector products and derive efficient expressions for these products below.

Jacobian-Vector Product: The product of the Jacobian matrix $\mathbf{J}$ and a vector $\mathbf{x}$ can be obtained in a similar way as for the type I model, by partitioning the vector $\mathbf{x}$, i.e., $\mathbf{x} = [\mathbf{x}_u; \mathbf{x}_c]$ and utilizing properties 1 and 2 and Equation (24), as

$$\mathbf{J}\mathbf{x} = \big[\big(\mathbf{X}_u^{\mathrm{T}}\mathbf{Z}\big) * (\mathbf{C}\widetilde{\mathbf{W}})\big]^{\mathrm{T}}\mathbf{1}_R + \mathbf{V}\mathbf{x}_c, \tag{25}$$

where $\mathbf{X}_u = \mathrm{unvec}(\mathbf{x}_u)$. As mentioned earlier for Equation (18), explicit construction of the diagonal matrix $\mathbf{C}$ is not required. The product $\mathbf{C}\widetilde{\mathbf{W}}$ can be obtained in a memory efficient way by multiplying the rows of $\widetilde{\mathbf{W}}$ by the corresponding diagonal elements of $\mathbf{C}$.

Jacobian Transpose -Vector Product and Gradient: In similar way, utilizing properties 1 and 2 and Equation (24), the product of Jacobian transpose $\mathbf{J}^{\mathrm{T}}$ and a vector $\mathbf{x}$ can be written as

$$\mathbf{J}^{\mathrm{T}}\mathbf{x} = \Big[\mathrm{vec}\Big(\mathbf{Z}\mathrm{Diag}(\mathbf{x})(\mathbf{C}\widetilde{\mathbf{W}})^{\mathrm{T}}\Big); \mathbf{V}^{\mathrm{T}}\mathbf{x}\Big]. \tag{26}$$

Since the gradient is the product of the Jacobian transpose $\mathbf{J}^{\mathrm{T}}$ and the residual vector $\mathbf{r}$, it directly follows from the above Equation (26) as

$$\mathbf{g} = \Big[\mathrm{vec}\Big(\mathbf{Z}\mathrm{Diag}(\mathbf{r})(\mathbf{C}\widetilde{\mathbf{W}})^{\mathrm{T}}\Big); \mathbf{V}^{\mathrm{T}}\mathbf{r}\Big]. \tag{27}$$

3.5. Complexity Analysis

We now analyze the storage and computational complexity of TeMPO where we are optimizing over symmetric rank-R CPD structured tensors $\boldsymbol{\mathcal{T}} \in \mathbb{K}^{I\times I\times\ldots\times I}$ of order d. The analysis is presented here for the type II model. However, since the number of optimization parameters of the type I and type II models (see Equations 2, 3) for an I-variate polynomial of degree d are proportional to each other, the analysis also applies to the type I model. Indeed, the computational complexity of the type I model is d times the computational complexity of the type II model. We also compare with the storage and computational complexity of TT and PEPS tensor networks.

Representing a multivariate polynomial with I independent variables and of degree d in dense format requires storing $C^d_{(I+d)}$ elements. Using Stirling's approximation, it can be shown that the storage complexity for a multivariate polynomial represented in dense format is $\mathcal{O}\left(I^d\right)$ for $d \ll I$. In the symmetric CPD format, we need to store only the factor matrix $\mathbf{U} \in \mathcal{R}^{I\times R}$ and the vector of weights $\mathbf{c} \in \mathcal{R}^R$, where R is the rank of the symmetric CPD. Therefore, the storage complexity for the type II model using the symmetric CPD format is $\mathcal{O}(IR)$. This shows that the symmetric CPD format breaks the curse of dimensionality, since the storage complexity in this format is linear in terms of rank R and dimension I.

As is clear from Equation (22), the construction of the matrix $\mathbf{W}$ and its dth Hadamard (elementwise) power dominates the computational complexity of the objective function. The construction of a single column of the matrix $\mathbf{W}$ requires the multiplication of $\mathbf{U}^{\mathrm{T}} \in \mathcal{R}^{R\times I}$ and $\widetilde{\mathbf{z}}_k \in \mathcal{R}^I$. Thus, the computational complexity of constructing the matrix $\mathbf{W}$ is $\mathcal{O}(IKR)$. The dth Hadamard power of the matrix $\mathbf{W}$ can be computed recursively by using the relation $\mathbf{W}^{*(2m)} = \left(\mathbf{W}^{*m}\right)^{*2}$. Thus, the computational complexity of the dth Hadamard power of the matrix $\mathbf{W} \in \mathcal{R}^{R\times K}$ is $\mathcal{O}\left(\log(d)RK\right)$. Therefore, the total computational complexity for computing the objective function for a batch of size K is $\mathcal{O}\left((I + \log(d))KR\right)$. Since $\log(d) \ll I$, the computational complexity for the objective function in Equation (8) is $\mathcal{O}(IKR)$.

The gradient of the objective function in Equation (8) can be obtained by multiplying the Jacobian transpose $\mathbf{J}^{\mathrm{T}}$ by the residual vector $\mathbf{r}$. As shown in Equation (27), this operation requires multiplication of a matrix $\mathbf{Z} \in \mathcal{R}^{I\times K}$ by a diagonal matrix $\mathrm{Diag}(\mathbf{r})$, and the multiplication of the matrices $\mathbf{Z}\mathrm{Diag}(\mathbf{r})$ and $(\mathbf{C}\widetilde{\mathbf{W}})^{\mathrm{T}}$ with sizes $(I \times K)$ and $(K \times R)$, respectively. Note that the entries of the product $\mathbf{C}\widetilde{\mathbf{W}}$ were already obtained in the computation of the objective function. Further, the computational complexity for the product $\mathbf{Z}\mathrm{Diag}(\mathbf{r})$ is $\mathcal{O}(IK)$. Consequently, the computational complexity for the multiplication of $\mathbf{Z}\mathrm{Diag}(\mathbf{r})$ and $(\mathbf{C}\widetilde{\mathbf{W}})^{\mathrm{T}}$ is $\mathcal{O}(IKR)$. In addition, the computation of $\mathbf{V}^{\mathrm{T}}\mathbf{r}$ in Equation (27) requires $\mathcal{O}(KR)$ operations. However $KR \ll IKR$. Therefore, the total computational complexity for computing the gradient is $\mathcal{O}(IKR)$ for $R \gg 1$.

In addition, TeMPO uses the GN algorithm for the optimization. However, this is not a requirement and first-order methods can also be utilized within TeMPO as well. GN requires solving the linear system of equations in (12). Tensorlab's implementation of GN uses the conjugate-gradient (CG) method which requires only the Grammian-vector product for solving (12). This operation requires multiplication of the Jacobian and its transpose by a vector. The computational complexity of multiplying the transpose of Jacobian by a vector is $\mathcal{O}(IKR)$ as described above. The computationally most expensive operations in the multiplication of Jacobian by a vector are the multiplication of matrices $\mathbf{X}_u^{\mathrm{T}}$ and $\mathbf{Z}$ with sizes $(R \times I)$ and $(I \times K)$, and the Hadamard product of two matrices of size $(R \times K)$ as shown in Equation (25). Hence, the computational complexity of computing $\mathbf{J}\mathbf{x}$ is $\mathcal{O}(IKR)$. Note that the entries of the product $\mathbf{C}\widetilde{\mathbf{W}}$ were already obtained in the computation of the objective function. Therefore, the total computational complexity for a single CG iteration is $\mathcal{O}(2IKR)$. Note that a large number of

TABLE 1 | The comparison of the computational complexity of TEMPO with TT and PEPS tensor networks for a batch size of K.

	Calls (per iter)	TEMPO (Type I model)	PEPS	TT-N
Storage		$\mathcal{O}(dIR)$		$\mathcal{O}(nIR_{TT}^2)$
Objective func.	1	$\mathcal{O}((dIKR)$	$\mathcal{O}(KR_{BT}^3R_{PS}^6)$	$\mathcal{O}(nIR_{TT}^2 + R_{TT}^3\log(I))$
Gradient	1	$\mathcal{O}(dIKR)$	$\mathcal{O}(\alpha KR_{BT}^3R_{PS}^6)$	$\mathcal{O}(\alpha(nIKR_{TT}^2 + KR_{TT}^3\log(I)))$
Gramian-vector	it_{CG}	$\mathcal{O}(2dIKR)$	–	–

CG iterations in the solution of linear equations for the GN algorithm might increase the computation time compared to first-order algorithms. In fact, the number of CG iterations scales with the number of optimization variables (IR), if the exact solution is desired in the solution of the normal equations. This may lead to an quadratic complexity of $\mathcal{O}(2(IR)^2K)$. However, we observed in our experiments that a small number of CG iterations were sufficient to obtain accurate results. For example, we set the maximum number CG iterations to 10 for the classification of the MNIST and Fashion MNIST datasets, where the number of unknowns is $784 \times R$ with R ranging from 10 to 150.

The storage complexity of a tensor network with TT architecture is bounded by $\mathcal{O}(nIR_{TT}^2)$ for a tensor of order I with dimensions $(n \times n \times \ldots \times n)$, where R_{TT} denotes the TT-rank [48]. n is equal to 2 and I is the size of a single image in the image classification applications presented in [20, 21]. Note that the storage complexity of TT increases with powers of the TT-rank R_{TT}. The total computational complexity of TT for computing the objective function has been reported as $\mathcal{O}(nIR_{TT}^2 + R_{TT}^3\log(I))$, when the contraction order defined in [21] is used. When the sweeping algorithm described in [20] is used, the computational complexity of the objective function for TT is $\mathcal{O}(n^3R_{TT}^3I)$ for a single data point of size I. Similar to the storage complexity, the computational complexity of the objective function for TT increases with powers of the TT-rank of the tensor under consideration. On the other hand, automatic differentiation (AD) is one of methods used to compute the gradient of TT. The computational complexity of automatic differentiation is linear in the complexity of the evaluation of the objective function [49]. Therefore, the computational complexity of the gradient for TT tensor network presented in [21] is $\mathcal{O}(\alpha(nIKR_{TT}^2 + KR_{TT}^3\log(I)))$, for a batch size of K with $\alpha > 1$. The total computational complexity of TT tensor network for a batch size of K has been reported as $\mathcal{O}(mR_{TT}^2(R_{TT} + K))$ for a single iteration of the stochastic Riemannian gradient descent algorithm [19]. As it is clear from the above discussion, both the storage and the computational complexity of TT increases with a power of the TT-rank regardless of the algorithm used, while for TeMPO it increases linearly with the symmetric CPD rank in the symmetric CPD case.

The computational complexity of a single forward pass of PEPS for a batch size of K is $\mathcal{O}(KR_{BT}^3R_{PS}^6)$, when the boundary matrix product state method is used. Here R_{BT} is the bond dimension (rank) of the boundary matrix product state of PEPS and R_{PS} is the bond dimension of PEPS. In addition, the backward pass for PEPS requires $\mathcal{O}(\alpha KR_{BT}^3R_{PS}^6)$ operations (with $\alpha > 1$), when automatic differentiation is used [22].

The above analysis shows that TeMPO is computationally less expensive than TT and PEPS, even though it uses a second-order algorithm. All these results are summarized in **Table 1**. The fundamental reason for this is the linear storage complexity of the symmetric CPD format. Both TT and PEPS involve third and higher-order tensors, which makes their computational complexity increase with powers of the bond dimension. On the other hand, the CPD format is known to be numerically less stable than the TT format, which relies on orthogonal matrices.

4. NUMERICAL EXPERIMENTS

We conducted an experiment on the regression problem using synthetic data to illustrate the TeMPO framework and compared TeMPO with different implementations of SVMs in Section 4.1. Next, we applied our framework to the blind deconvolution of constant modulus (CM) signals and compared with the analytical CM algorithm (ACMA) [50], the optimal step-size CM algorithm (OSCMA) [51], and the LS-CPD framework [52] in Section 4.2. In Section 4.3, we further illustrate TeMPO with the image classification problem. We performed experiments on the MNIST and Fashion MNIST datasets and compared the accuracy and number of optimization parameters with MLPs, and TT and PEPS tensor networks. We performed experiments on a computer with an Intel Core i7-8850H CPU at 2.60 GHz with 6 cores and 32 GB of RAM using MATLAB R2021b and Tensorlab 3.0 [11].

In our blind deconvolution experiments, we used the complex GN algorithm with the conjugate gradient Steihaug method. We used the second-order batch Gauss–Newton algorithm for the regression and classification, following the same intuition as in [53]. In each epoch of the algorithm, we randomly shuffle the data points in the training set and process all data points by dividing them into batches. In the regression and binary classification case, we optimize a single cost function. In the multi-label classification case, for each batch, we randomly select a cost function f_l defined for each label to optimize. Thus our algorithm does not guarantee that each f_l will be trained by all training images in each epoch in the multi-label classification case. To guarantee this, the algorithm can be modified such that for each batch all cost functions f_l are optimized at the cost of increasing CPU time by a factor of the number of classes L. However, in that case the algorithm might need fewer epochs to converge. The overall algorithm is summarized in **Algorithm 2**. **Algorithm 2**

Algorithm 2: Batched GN algorithm using dogleg trust-region for regression and classification for the type II model.

Input : $\mathbf{Z}$ – Input data matrix
$\mathbf{y}$ – Vector of values (labels in the classification case) for each data point in $\mathbf{Z}$
$\mathbf{U}_1, \ldots, \mathbf{U}_L$ – Initial factor matrices for each label (single in the regression case)
$\mathbf{c}_1, \ldots, \mathbf{c}_L$ – Initial weight vectors for each label (single in the regression case)
$\mathcal{T}_{0,l}$ – Initial scalar for each label (single in the regression case)
epoch – Number of epochs
batchsize – Batch size
Output: $\mathbf{U}_1, \ldots, \mathbf{U}_L$ – Optimized factor matrices for each label (single in the regression case)
$\mathbf{c}_1, \ldots, \mathbf{c}_L$ – Optimized weight vectors for each label (single in the regression case)

```
for each epoch do
    Shuffle input data
    for each batch do
        l ← 1
        if multi-label classification then
            l ←
            Randomly select label l to optimize f_l , 0 < l ≤ L
        end
        U_l, c_l, T_{0,l} ← Optimize f_l using Algorithm 1
    end
end
```

is given for the type II model for the ease of explanation. Slight modifications are sufficient to obtain an algorithm for the type I model.

We define the relative error as the relative difference in l_2 norm $||\mathbf{f} - \widehat{\mathbf{f}}||_2 / \|\mathbf{f}\|_2$ with $\widehat{\mathbf{f}}$ an estimate for a vector $\mathbf{f}$, and the signal-to-noise ratio (SNR) as $20\log_{10}\left(\|\mathbf{f}\|_2/\|\boldsymbol{\eta}\|_2\right)$, where $\boldsymbol{\eta} = \widehat{\mathbf{f}} - \mathbf{f}$.

4.1. Regression

In this experiment, we considered a low-rank smooth function $f(\mathbf{x}) : \mathcal{R}^N \to \mathcal{R}$, namely

$$f(\mathbf{x}) = \sum_{r=1}^{R_f} \alpha_r e^{(\mathbf{a}_r^{\mathrm{T}}\mathbf{x})}, \tag{28}$$

where $\mathbf{x} \in [-1, 1]^N$, R_f is the rank of the function $f(\mathbf{x})$, and the coefficients α_r are scalars randomly chosen from the standard normal distribution. We generated 5,000 test samples and 1,000 training samples for $N = 50$ and $R_f = 8$. Each vector $\mathbf{a}_r \in \mathcal{R}^N$ was a unit norm vector drawn from the standard normal distribution. Each of the samples of $\mathbf{x}$ was drawn from the uniform distribution. We initialized each factor matrix with a matrix whose elements were randomly drawn from the standard normal distribution, and scaled it to unit norm. We initialized each weight vector in the same way as the factor matrices. We approximated $f(\mathbf{x})$ by the type I and type II model of degree 5 whose coefficient tensors were represented in the rank-R symmetric CPD format. We set the batch size to 500 and the maximum number GN iterations to 5 for each batch. In **Figure 3**, we show the median relative test and training errors for $R = \{2, 4, 8, 16\}$ as a function of the number of epochs for 100 trials. Each epoch corresponds to optimization over the full training set. It is clear from **Figure 3** that TeMPO produces more accurate results and generalizes better when using higher rank values, for both the type I and type II model. Good performance is also observed for $R = 16 > R_f = 8$, meaning that TeMPO is robust to over-estimation of the number of parameters. For low rank values, i.e., $R < R_f$, the type I model produces better results than the type II model because it involves more parameters that can be tuned, cf. the discussion of Proposition 1.

In the second stage of the experiment, we trained the type I and type II model for a multivariate polynomial of degree 5 with noisy measurements. We added Gaussian noise to the function values for a given SNR, i.e.,

$$\widetilde{f}(\mathbf{x}) = \sum_{r=1}^{R} \alpha_r e^{(\mathbf{a}_r^{\mathrm{T}}\mathbf{x})} + \eta, \tag{29}$$

where η denotes the noise. We run our algorithm with the same settings as in the noiseless case for an SNR ranging from 10 dB to 50 dB. In **Figure 4**, we show the median errors for 100 trials as a function of SNR. We have similar observations as in the noiseless case. Apart from these observations; although the accuracy of our algorithm decreases for SNR $\leq$ 20 (dB), it still maintains good accuracy for SNR $>$ 20 (dB), as shown in **Figure 4**. Moreover, as can be observed from the **Figure 4** (left), the type I model overfits for $R = \{8, 16\}$ and SNR $\leq$ 20 (dB) in agreement with the result of Proposition 1.

In our next experiment, we trained the type I and type II model with larger-size samples, i.e., $N = 250$ and $R = \{8, 16, 32, 64\}$, to assess how the CPU time depends on the rank. In **Figure 5**, we show the median CPU time per epoch as a function of the rank. It is evident from the figure that the computational complexity of the type I model is d times the computational complexity of the type II model (cf. Section 3.5). Moreover, **Figure 5** confirms that the computational complexity of our algorithm is linear in the rank (cf. Section 3.5).

In our next experiment, we examined the generalization abilities of the Gauss–Newton and ADAM [54] algorithms in our framework. We trained the type I model for a multivariate polynomial of degree 5 with both of these algorithms for different number of training samples to fit the rank-8 function given as in Equation (29). We set $R = 8$, $N = 50$, and SNR $= 20$(dB). For the ADAM algorithm, we set the step size, the exponential decay rate for the first momentum (β_1), and the exponential decay rate for the second momentum (β_2) to 0.01, 0.9, and 0.99, respectively. In **Figure 6**, we show the median training and test accuracies of these algorithms for the number of training samples ranging from 500 to 5,000 as a function of the number of epochs for 100 trials. It is evident from **Figure 6** that the presented Gauss–Newton algorithm produces more accurate results than

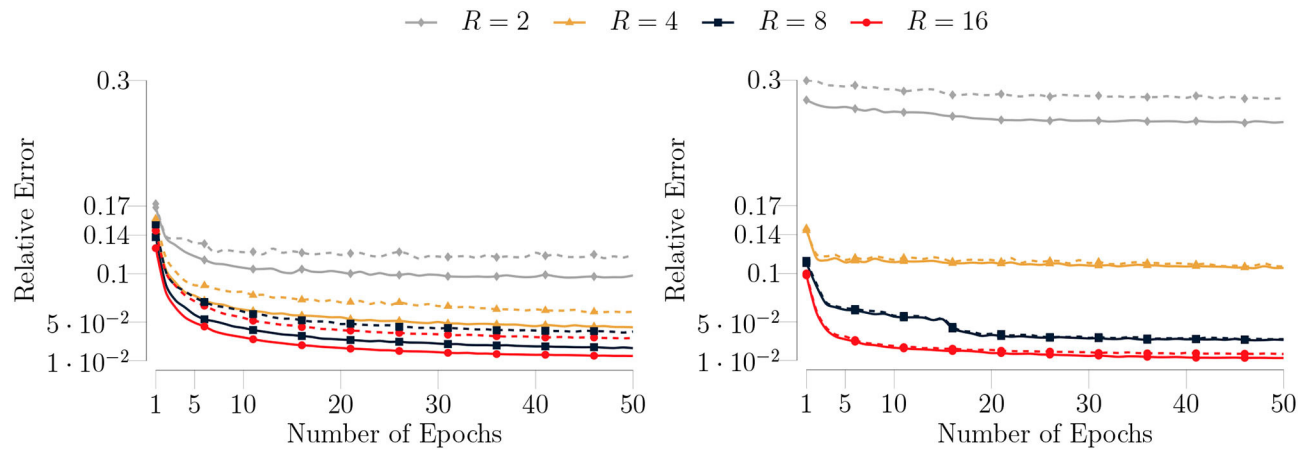


FIGURE 3 | (Left) The median test (dashed lines) and training (solid lines) errors of the type I model for 100 trials on the synthetic data for a rank-8 function given as in Equation (28). The number of samples for the training dataset is set to 5, 000 and for the test dataset it is set to 1, 000. The batch size is set to 500 and the maximum number of GN iterations is set to 5. **(Right)** The median test (dashed lines) and training (solid lines) errors of the type II model with the same algorithm settings. TeMPO produces more accurate results and generalizes better for higher rank values for both the type I and type II model. The performance is robust to overparameterization ($R > R_f$). The type I model produces better results for low rank values, i.e., $R < R_f$.

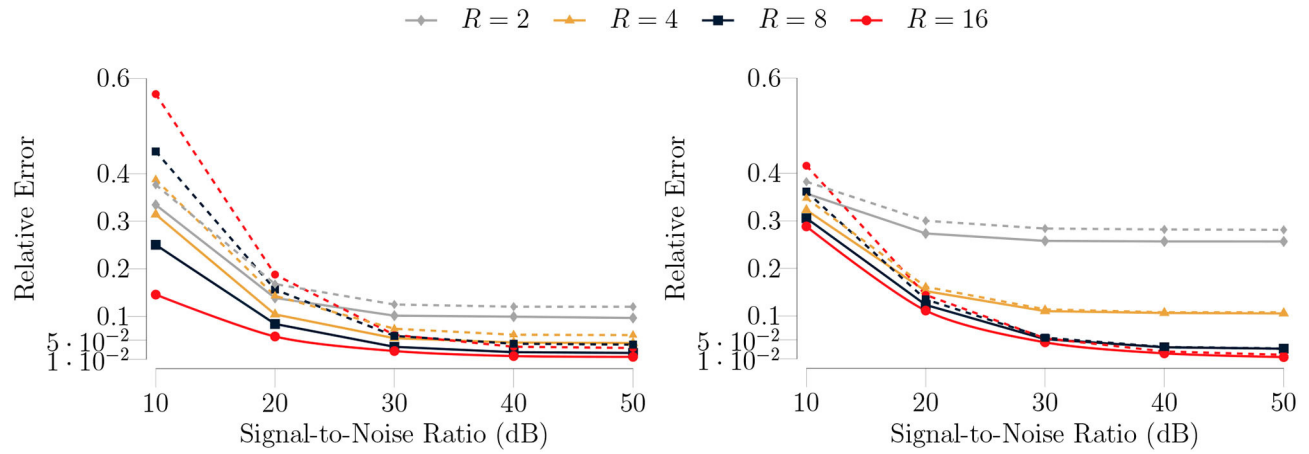


FIGURE 4 | (Left) The median test (dashed lines) and training (solid lines) errors of the type I model for 100 trials on the synthetic noisy data for a rank-8 function given as in Equation (29). The number of samples for the training dataset is set to 5, 000 and for test dataset is set to 1, 000. The batch size is set to 500 and the maximum number of GN iterations is set to 5. **(Right)** The median test (dashed lines) and training (solid lines) errors of the type II model with the same algorithm settings. TeMPO produces more accurate results and generalizes better for higher rank values for both the type I and type II model in the presence of noise as well. Again, the type I model produces better results for low rank values, i.e., $R < R_f$, because it involves more parameters than the type II model.

the ADAM algorithm and also requires fewer number of epochs to converge in these experimental settings.

We also compared TeMPO with SVMs using a polynomial kernel. We run the same experiment for a number of training samples ranging from 500 to 5, 000. We set the rank to 8, i.e., $R = R_f$ for TeMPO. We used the built-in Matlab routine `fitrsvm` and LS-SVMlab toolbox [55, 56]. We set the degree of polynomial kernel to 5, i.e., equal to the degree of the type I and type II model for `fitrsvm`. LS-SVMlab automatically tunes the degree to 3 to find the best fit. In **Figure 7** (left), we show the median test and training errors for SVM, the type I and type II model. It is clear from **Figure 7** (left) that the type I and type II model generalize better than `fitrsvm`. A possible reason is the dense parameterization of SVMs, while TeMPO uses low-rank parameterization. Moreover, as shown in **Figure 7** (right), our algorithm is faster than SVMs for numbers of training samples above 1, 000. This is due to the higher memory requirement of SVMs. Typically, kernel based methods such as LS-SVM have a storage and computational complexity of $\mathcal{O}\left(N^2\right)$ [55], with N the number of training samples. In contrast, **Figure 7** (right) confirms that the computational complexity of TeMPO is linear in the number of training samples (cf. Section 3.5).

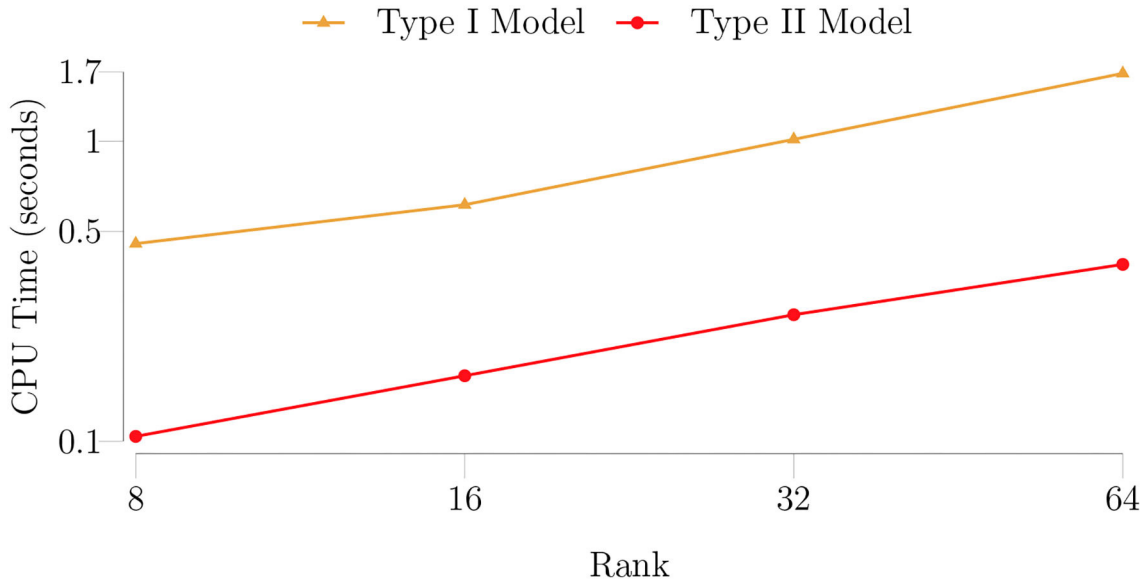


FIGURE 5 | The median CPU time (seconds) per epoch for the type I and type II model as a function of the rank for a rank-8 function given as in Equation (28) for 100 trials. The number of samples for the training dataset is set to 5,000 and for the test dataset it is set to 1,000. The batch size is set to 500 and the maximum number of GN iterations is set to 5. The figure confirms that the computational complexity of the type I model is d times the computational complexity of the type II model (cf. Section 3.5). Moreover, the computational complexity of the algorithm is linear in the rank (cf. Section 3.5). The figure is in a logarithmic scale on the horizontal axis.

4.2. Blind Deconvolution of Constant Modulus Signals

Blind deconvolution can be formulated as a multivariate polynomial optimization (MPO) problem and hence it fits into the TeMPO framework [15]. In this illustrative example, we limit ourselves to an autoregressive single-input single-output (SISO) system [57], given by

$$\sum_{l=0}^{L} w_l \cdot y[k-l] = s[k] + n[k], \text{ for } k = 1, \ldots, K, \tag{30}$$

where $y[k]$, $s[k]$, and $n[k]$ are the measured output signal, the input signal and the noise at the kth measurement, respectively, and w_l denotes the lth filter coefficient. Ignoring the noise for the ease of derivation, (30) can be written as:

$$\mathbf{Y}^{\mathrm{T}}\mathbf{w} = \mathbf{s}, \tag{31}$$

where $\mathbf{Y} \in \mathbb{C}^{L\times K}$ is a Toeplitz matrix and its rows are the subsequent observations under the assumption that we have $K + L - 1$ samples $y[-L+1], \ldots, y[K]$. The vector $\mathbf{w} \in \mathbb{K}^L$ contains the filter coefficients and the kth entry of the source vector $\mathbf{s} \in \mathbb{C}^K$ is the input signal at the kth time instance, i.e., $s_k = s[k]$. In blind deconvolution, one attempts to find the original input signal $\mathbf{s}$ and the filter coefficients $\mathbf{w}$ by only observing the output signal $\mathbf{Y}$. Thus, constraints on signals and/or channel have to be imposed to obtain interpretable results. The constant modulus (CM) criterion is a widely used input constraint [58]. The CM property, which holds for phase- or frequency-modulated signals [50, 59] can be written as:

$$|s_k|^2 = c, \text{ for } k = 1, 2, \ldots, K. \tag{32}$$

Here, c is a constant scalar. By substituting s_k defined in (31) into (32), we obtain

$$\left(\mathbf{Y} \odot \overline{\mathbf{Y}}\right)^{\mathrm{T}}(\mathbf{w} \otimes \overline{\mathbf{w}}) = c \cdot \mathbf{1}_K. \tag{33}$$

Following the same intuition as in [60], by multiplying (33) from the left with a Householder reflector $\mathbf{Q}$ [61], generated for $c \cdot \mathbf{1}_K$, and removing the first equation[3], we obtain

$$\mathbf{M}(\mathbf{w} \otimes \overline{\mathbf{w}}) = \mathbf{0}. \tag{34}$$

Here, $\mathbf{M} = \widetilde{\mathbf{Q}}\left(\mathbf{Y} \odot \overline{\mathbf{Y}}\right)^{\mathrm{T}}$, and $\widetilde{\mathbf{Q}}$ is obtained by removing the first row of the Householder reflector $\mathbf{Q}$. In applications, $\mathbf{M}(\mathbf{w} \otimes \overline{\mathbf{w}})$ will not vanish exactly due to the presence of noise. Hence, we look for the solution which minimizes its l_2 norm as

$$\min_{\mathbf{w},\overline{\mathbf{w}}} f(\mathbf{w}, \overline{\mathbf{w}}) = \min_{\mathbf{w},\overline{\mathbf{w}}} \frac{1}{2} \left\|\mathbf{M}(\mathbf{w} \otimes \overline{\mathbf{w}})\right\|_2^2, \quad \text{subject to } \|\mathbf{w}\| = 1. \tag{35}$$

The objective function in (35) is a homogeneous multivariate polynomial of degree 4 in which the coefficient tensor $\mathcal{W}$ is given as a rank-1 Hermitian symmetric CPD, i.e.,

$$\mathcal{W} = \mathbf{w} \otimes \overline{\mathbf{w}} \otimes \overline{\mathbf{w}} \otimes \mathbf{w} := \left[\!\left[\mathbf{w}, \overline{\mathbf{w}}, \overline{\mathbf{w}}, \mathbf{w}\right]\!\right]. \tag{36}$$

Exploiting the rank-1 Hermitian symmetric CPD structure in (36) and the structure of $\mathbf{M}$, which is a special case of Lemma 1 and Lemma 2, efficient expressions for the computation

[3]The first equation is only a normalization constraint.

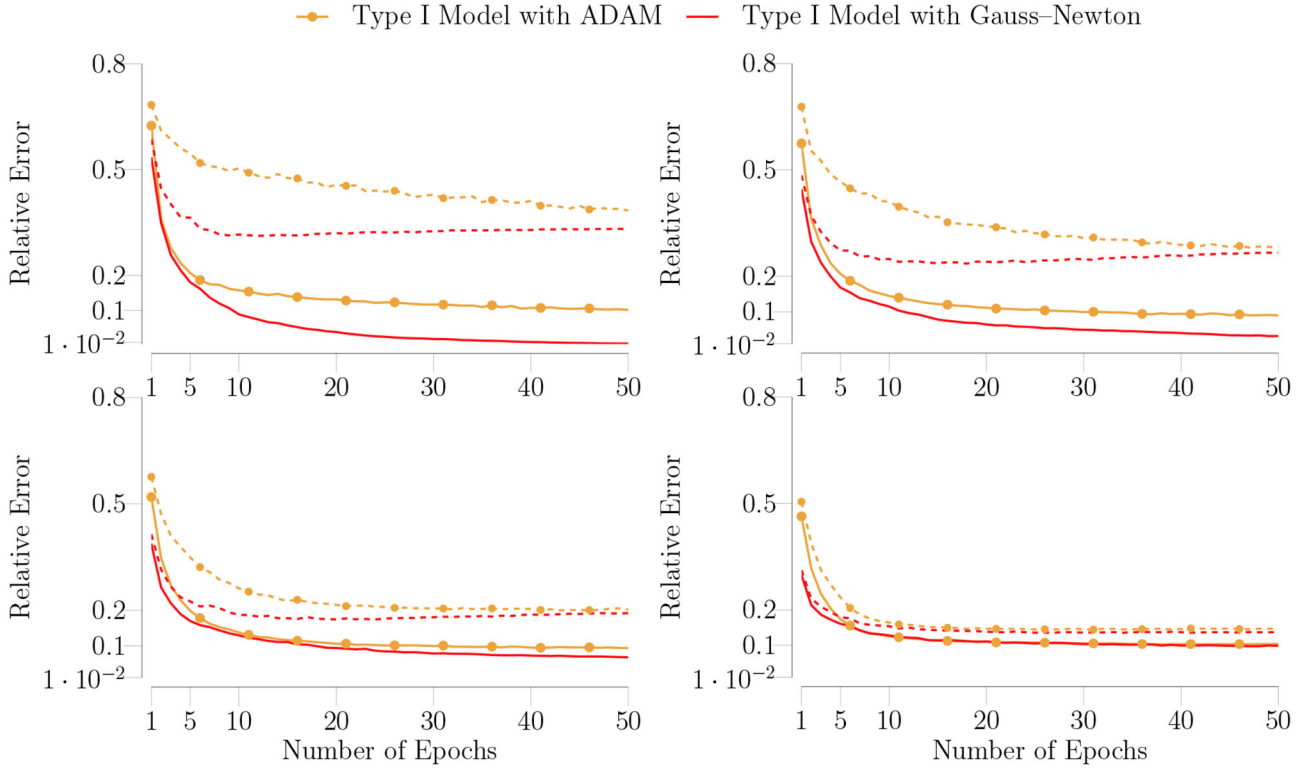


FIGURE 6 | Comparison of the median test (dashed lines) and training (solid lines) errors of the Gauss–Newton and the ADAM algorithms as a function of the number of epochs for 100 trials. The type I model for a rank-8 function given as in Equation (29) in the presence of SNR 20 dB Gaussian noise is used to generate the training and the test sets. The batch size is set to 10% of the training set size. For the Gauss–Newton algorithm, the maximum number of GN iterations and CG iterations is set to 1 and 5, respectively. For the ADAM algorithm, the step size, β_1 and β_2 are set to 0.01, 0.9, and 0.99, respectively. The number of training samples is set to 500 **(top-left)**, 1,000 **(top-right)**, 2,000 **(bottom-left)**, and 5,000 **(bottom-right)**. The presented Gauss–Newton algorithm produces more accurate results than the ADAM algorithm and also requires fewer number of epochs to converge in these experimental settings.

of Jacobian-vector products for the problem (35) have been presented in [15].

A number of algorithms have been developed to solve (33) and (34). The analytical CM algorithm (ACMA) [50] writes (34) as a generalized matrix eigenvalue problem in the absence of noise, and under the assumption that the null space of $\mathbf{M}$ is one dimensional, which makes ACMA more restrictive than TeMPO. In the presence of noise, ACMA writes (34) as the simultaneous diagonalization of a number of matrices and solves it by extended QZ iteration. Gradient descent and stochastic gradient descent algorithms have also been proposed for the minimization of the expected value of $\{(|\mathbf{y}_\mathbf{n}^\mathrm{T}\mathbf{w}| - c)^2\}$. The optimal step-size CMA (OSCMA) [51] algorithm uses a gradient descent algorithm, which computes the step size algebraically. The problem in (35) can also be interpreted as a linear system with a rank-1 constrained solution, which fits the LS-CPD framework in [52]. LS-CPD solves (33) by relaxing the complex conjugate $\overline{\mathbf{w}}$ to a possibly different vector $\mathbf{v} \in \mathbb{C}^L$ and utilizing the second-order GN algorithm using dogleg trust-region method. We solve (35) by utilizing the complex GN algorithm using the conjugate gradient Steihaug method implemented in TensorLab 3.0 [11]. We compare with these algorithms in terms of computation time and accuracy.

We consider an autoregressive model of degree $L = 10$ with coefficients uniformly distributed on $[0, 1]$, sample length $K = 600$, and $c = 1$. We add scaled Gaussian noise to the measurements to obtain a particular SNR. We run 50 experiments starting from the algebraic solution presented in [52] for LS-CPD, OSCMA, and TeMPO. In **Figure 8** (left), we show the median relative error on $\mathbf{w}$ as a function of SNR. It is clear from **Figure 8** (left) that TeMPO achieves similar accuracy as LS-CPD and OS-CMA, which are more accurate than ACMA. In **Figure 8** (right), we show the median CPU time in seconds as a function of SNR. Clearly, TeMPO is faster than ACMA, OS-CMA, and LS-CPD for SNR $\geq$ 10(dB) by exploiting the structure of the data.

4.3. Image Classification

Multi-class image classification amounts to the determination of a possibly nonlinear function f that maps input images $\mathbf{Z}_k$ to integer scalar labels y_k, which are known for a training set. In this study, we represent f by a multivariate polynomial p. Following the one-versus-all strategy, we define a cost function f_l for each label that maps the input image $\mathbf{Z}_k$ to a scalar value as

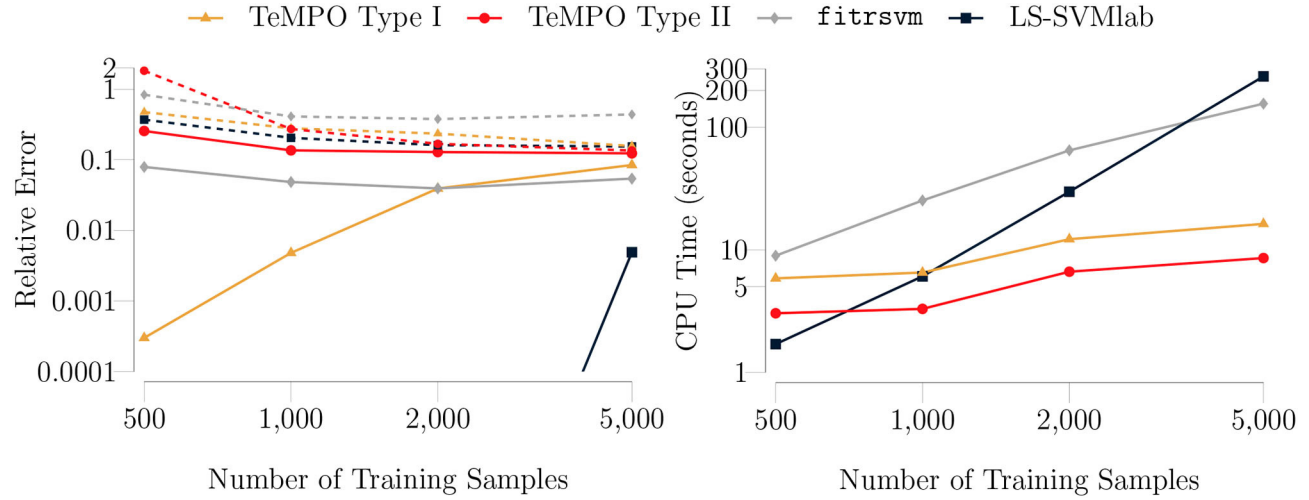


FIGURE 7 | (Left) The median test (dashed lines) and training (solid lines) errors of SVMs with polynomial kernel, the type I and type II model for a rank-8 function given as in Equation (29) in the presence of SNR 20 dB Gaussian noise as a function of the number of training samples for 100 trials. The batch size is equal to 10% of the training set size. The maximum number of GN iterations is set to 5 for the type I and type II model. Specifically, for the SVMs, the built-in Matlab routine `fitrsvm` and LS-SVMlab toolbox were used to obtain the results. The relative errors of LS-SVMLab for the sample sizes 500, 1,000, and 2,000 are 1.6e − 6, 2.2e − 6 and 3.3e − 6, respectively. The presented algorithm generalizes better than `fitrsvm` in these experimental settings. **(Right)** The median CPU times (seconds) with the same setting. The computational complexity of our algorithm is linear in the problem size as expected, and it is faster than SVMs for numbers of training samples above 1,000. The figures are in a logarithmic scale on both the horizontal and vertical axes.

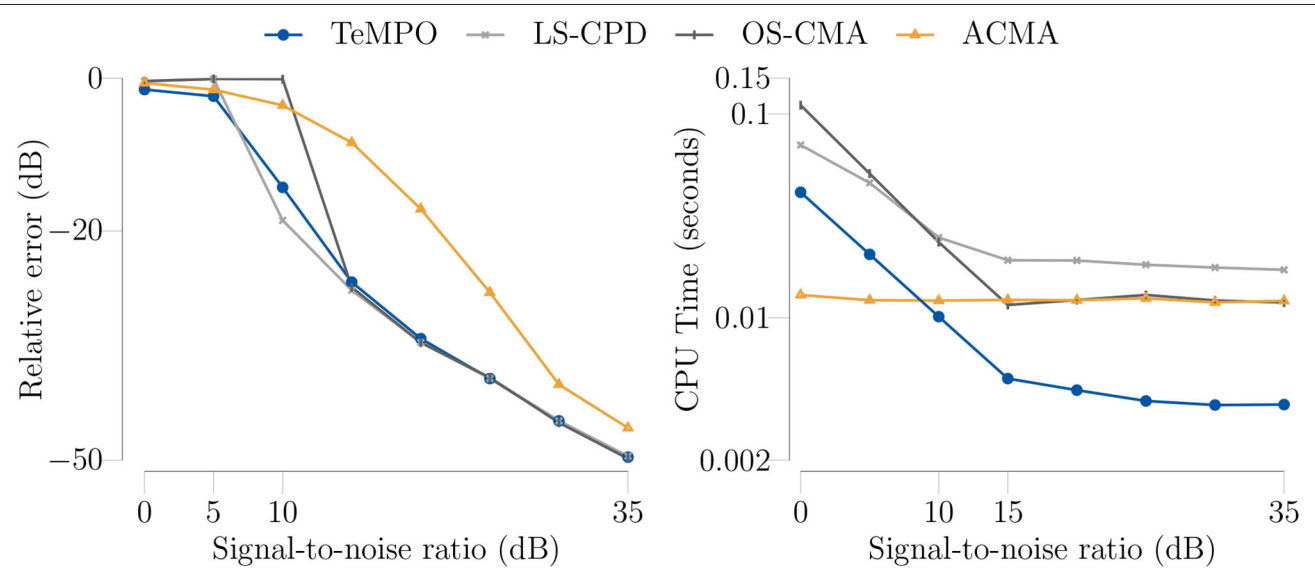


FIGURE 8 | (Left) The median relative errors (dB) of LS-CPD, OS-ACMA, ACMA, and TeMPO with respect to SNR (dB) for an autoregressive model of degree $L = 10$ with uniformly distributed coefficients between zero and one, sample length $K = 600$ for 50 trials. TeMPO obtains similar accuracy to LS-CPD, OS-CMA, while obtaining more accurate results than ACMA. **(Right)** The median CPU times (seconds) with the same settings. TeMPO is faster than other algorithms for SNR > 10 (dB).

$$f_l(p_l, \mathbf{z}_1, \ldots, \mathbf{z}_K) = \frac{1}{2}\sum_{k=1}^{K}\left(y_k - p_l(\mathbf{z}_k)\right)^2,$$

where $\mathbf{z}_k = \mathrm{vec}(\mathbf{Z}_k)$ and where $y_k = 1$ if $\mathbf{z}_k$ is labeled as l and $y_k = 0$ otherwise. The polynomial p_l can be chosen within the type I or the type II model class. For the type I model, the optimization problem can be written as

$$\min_{p_l} f_l(p_l, \mathbf{z}_1, \ldots, \mathbf{z}_K), \quad \text{subject to} \quad p_l(\mathbf{z}_k) = \mathcal{T}_{l,0} + \sum_{j=1}^{d}\mathcal{T}_{l,j}\mathbf{z}_k^j,$$
$$\text{and} \quad \mathcal{T}_{l,j} = \left[\!\left[\mathbf{U}_{l,j}, \ldots, \mathbf{U}_{l,j}; \mathbf{c}_{l,j}^{\mathrm{T}}\right]\!\right], \tag{37}$$

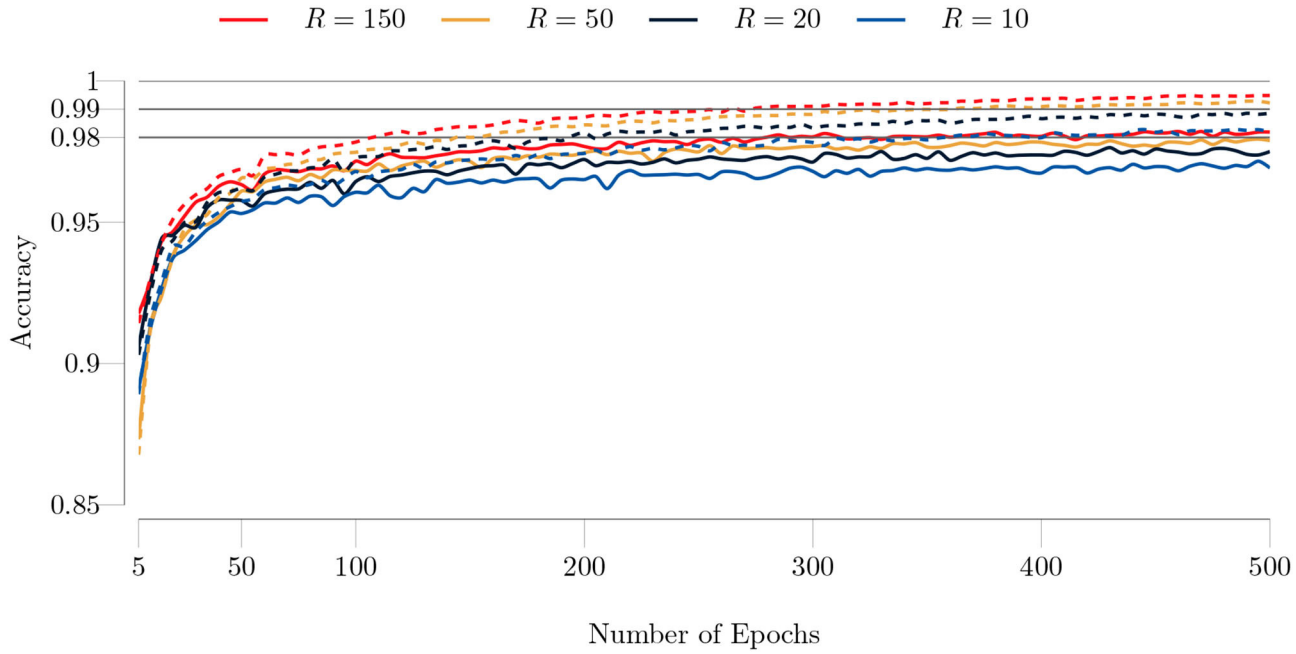


FIGURE 9 | Test (solid lines) and training (dashed lines) accuracies of the type I model for the MNIST dataset with respect to the number of epochs. The full training set (60, 000 images) and test set (10, 000 images) are used. The batch size is set to 100 and the maximum number of GN iterations is set to 1. TeMPO achieves high accuracy even for low rank values, i.e., $R = \{10, 20\}$. Both the test and training accuracy increase mildly as the rank increases.

where d is the degree of the polynomial under consideration. Note that we substitute the symmetric CPD structure given as a constraint into the objective function, and hence obtain and solve an unconstrained optimization problem. For the type II model, the optimization problem can be written as

$$\min_{p_l} f_l(p_l, \mathbf{z}_1, \ldots, \mathbf{z}_K), \quad \text{subject to} \quad p_l(\mathbf{z}_k) = \mathcal{T}_l \mathbf{z}_k^d,$$
$$\text{and} \quad \mathcal{T}_l = [\![\mathbf{U}_l, \ldots, \mathbf{U}_l; \mathbf{c}_l^{\mathrm{T}}]\!] .$$

After the optimization of f_l for each label l, the classification is done by computing each $p_l(\mathbf{s})$ for the data point $\mathbf{s}$ to be classified and selecting the value of l for which $|p_l(\mathbf{s})|$ is largest.

4.3.1. Experiments

We performed several experiments by varying the parameters rank and maximum number of GN iterations to illustrate the TeMPO framework for the classification of the MNIST and Fashion MNIST datasets. We kept the maximum number of CG iterations equal to 10, the degree of the multivariate polynomial to 3, the tolerance for the objective function and optimization variables equal to $1e-10$, the inner solver tolerance equal to $1e-10$, and the trust-region radius equal to 0.1, throughout the experiments.

We initialized each factor matrix with a matrix whose elements were randomly drawn from the standard normal distribution, and scaled it to unit norm. Similarly, we initialized each weight vector $\mathbf{c}_l$ with a vector whose elements were randomly drawn from the standard normal distribution and scaled it to unit norm.

Datasets

Modified National Institute of Standards and Technology (MNIST) handwritten digit database [62] and the Fashion MNIST database [63] are used for this study. Both datasets contain gray scale images of size (28×28). The training sets of both datasets are composed of 60, 000 images and test sets are composed of 10, 000 images. The images have been size-normalized and centered in a fixed-size image. We rescale images such that every pixel value is in the interval $[0, 1]$ and the mean of each image is zero. Then, we vectorize, i.e., stack each column vertically in a vector, each image to a vector of size 784. For the type II model, we augment the resulting vector by the scalar 1. Similar pre-processing steps are necessary for also tensor networks. Additionally, they may require the encoding input data which increases the storage and the computational resource requirement.

Results and Comparisons

Results of the Type I Model

We first trained the type I model on the total MNIST training set for various rank values ranging from 10 to 150 to illustrate the effect of rank on the accuracy. We set the batch size to 100 and the maximum number of GN iterations to 1. We show the training history in **Figure 9**. It is evident from **Figure 9** that TeMPO achieves high accuracy even for low rank values, i.e., $R = \{10, 20\}$. Increasing the rank mildly improves both the test and training accuracy, with the improvement getting smaller as the rank increases.

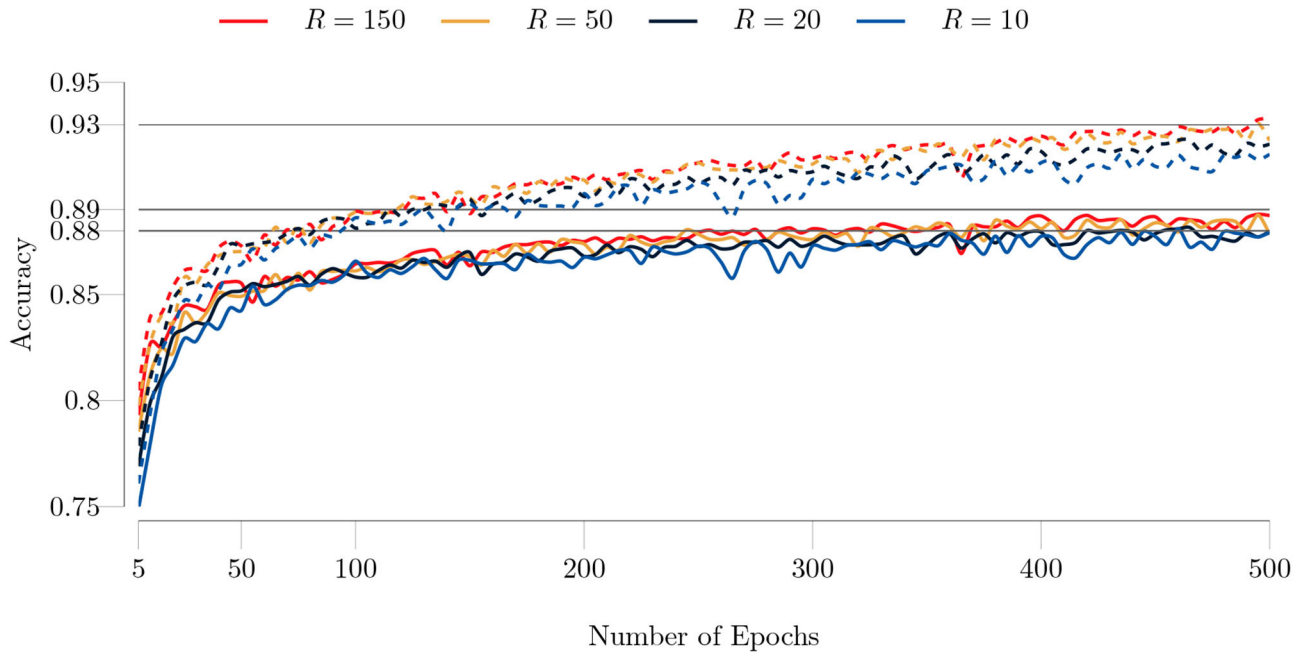


FIGURE 10 | Test (solid lines) and training (dashed lines) accuracies of the type I model for the Fashion MNIST dataset with respect to the number of epochs. The full training set (60, 000 images) and test set (10, 000 images) are used. The batch size is set to 100 and the maximum number of GN iterations is set to 1. Similar to the MNIST dataset, TeMPO achieves good accuracy even for low rank values and both the test and training accuracy mildly increase as the rank increases.

We repeated the same experiments for the Fashion MNIST dataset, which is harder to classify. We show the training history in **Figure 10**. The observations made for the MNIST dataset also apply to the Fashion MNIST dataset. However, the test and training accuracy are lower for the Fashion MNIST dataset in agreement with previous works. Also, our algorithm requires more epochs to converge for the Fashion MNIST dataset.

In our next experiment, we set the maximum number of GN iterations to 5. We observed that our algorithm needs fewer epochs to converge and produces more accurate results with this setting. The comparison for the MNIST and Fashion MNIST dataset is shown in **Figures 11**, **12**, respectively. The improvement in the test accuracy for the Fashion MNIST dataset is around 1% and more pronounced than the improvement in the test accuracy for the MNIST dataset. TeMPO achieves around 98.30% test accuracy for the MNIST dataset and around 90% test accuracy for the Fashion MNIST dataset with $R = 150$.

Results of the Type II Model

We repeated the same experiments for the type II model. We used the same settings as in the type I model. However, we set the batch size to 200 to obtain an accuracy similar to that of the type I model. We show the training history in **Figure 13**. Similar to previous experiments, our algorithm performs well even for low rank values, and produces more accurate results for higher rank values. TeMPO achieves around 98% test accuracy and 100% training accuracy after 200 epochs with $R = 150$ for the MNIST dataset.

In **Figure 14**, we show the training history for the Fashion MNIST dataset. Similar to the type I model, the test and training accuracy is lower than the MNIST dataset. The algorithm converges around 100 epochs and achieves around 89.30% test accuracy with $R = 150$. Moreover, our algorithm achieves around 99% training accuracy after 400 epochs.

We repeated the same experiments with the maximum number of GN iterations set to 5. The comparisons for the MNIST and Fashion MNIST datasets are shown in **Figure 15**. Contrary to our observation for the type I model, the test accuracy now decreases for both datasets. A possible reason is that when the residuals are big, doing more GN iterations may not lead a better direction for minimizing (37). A similar observation has been made in [53], for training DNNs. It is experimentally shown that higher number of CG iterations might not produce more accurate results if the Hessian obtained by mini-batch is not reliable due to non-representative batches and/or big residuals. On the other hand, if the residuals are small, higher number of CG iterations can produce more accurate results thanks to the curvature information [53].

Comparisons

We now compare TeMPO with different models, namely: TT tensor networks [21], TT structured tree tensor networks (TTN) [64], multi-layer perceptron (MLP) with $784-1000-10$ neurons, MLP with a convolution layer (CNN-MLP), PEPS, and PEPS with

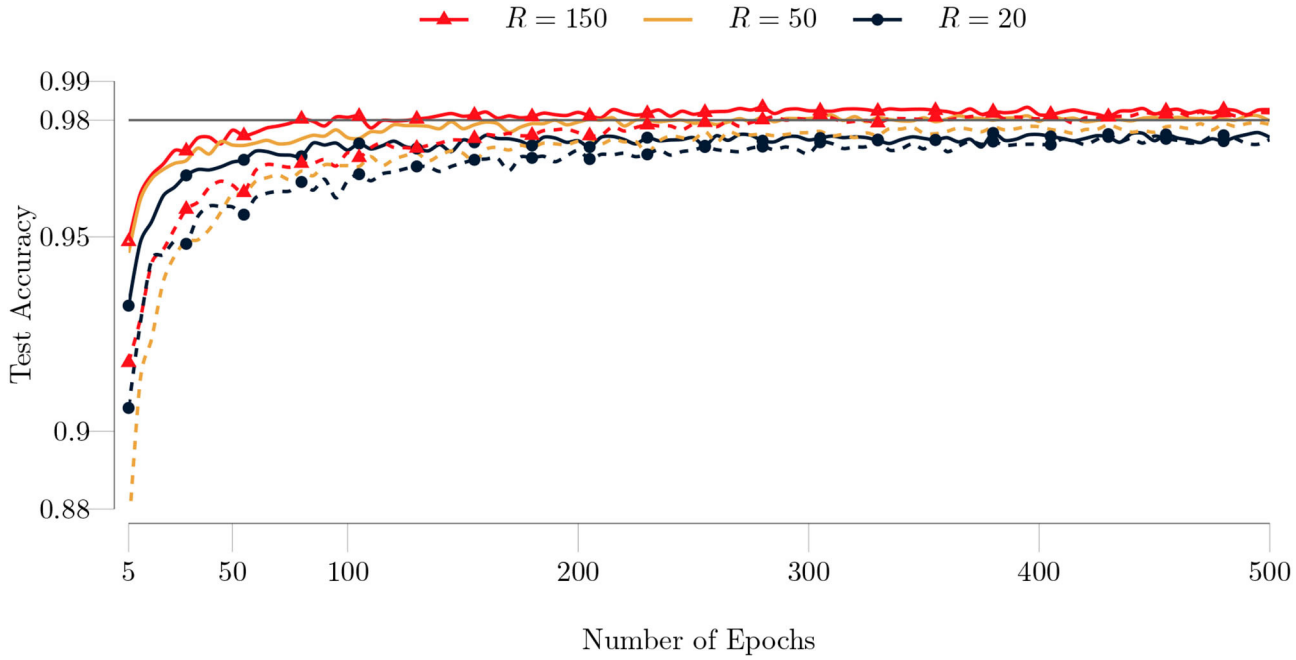


FIGURE 11 | Comparison of test accuracies of the type I model on the MNIST dataset for different maximum number of GN iterations as a function of the number of epochs. The full training set (60, 000 images) and test set (10, 000 images) are used. The batch size is set to 100 and the maximum number of GN iterations is set to 1 (dashed lines) and to 5 (solid lines).

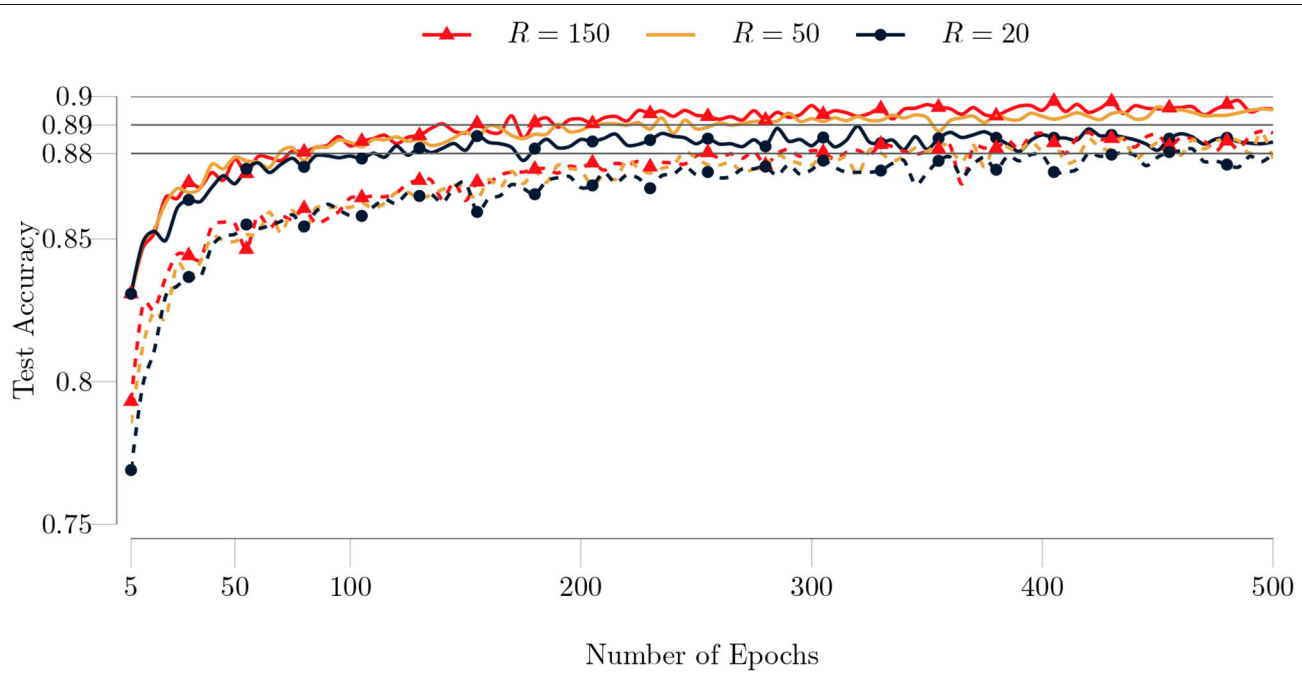


FIGURE 12 | Comparison of test accuracies of the type I model on the Fashion MNIST dataset for different maximum number of GN iterations as a function of the number of epochs. The full training set (60, 000 images) and test set (10, 000 images) are used. The batch size is set to 100 and the maximum number of GN iterations is set to 1 (dashed lines) and to 5 (solid lines).

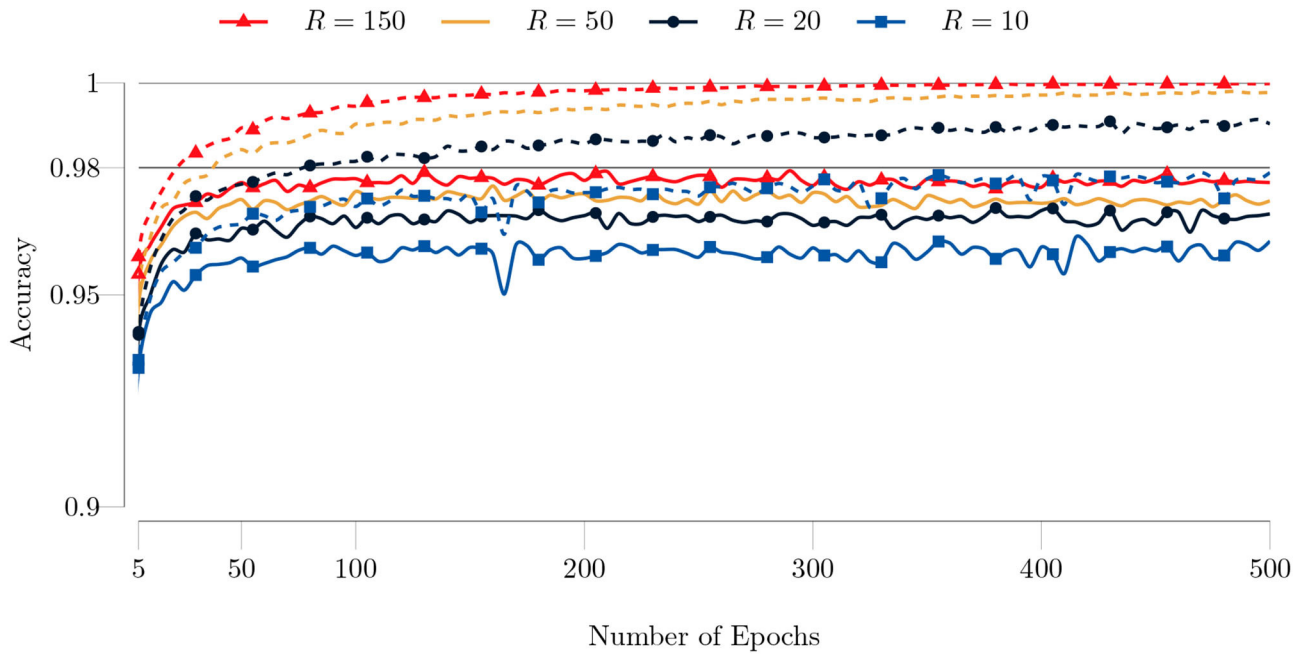


FIGURE 13 | Test (solid lines) and training (dashed lines) accuracies of the type II model for the MNIST dataset with respect to the number of epochs. The full training set (60, 000 images) and test set (10, 000 images) are used. The batch size is set to 200 and the maximum number of GN iterations is set to 1. Both the test and training accuracy increase as the rank increases. The improvement in the accuracy gets smaller as the rank increases. The algorithm achieves around 100% training accuracy after 200 epochs.

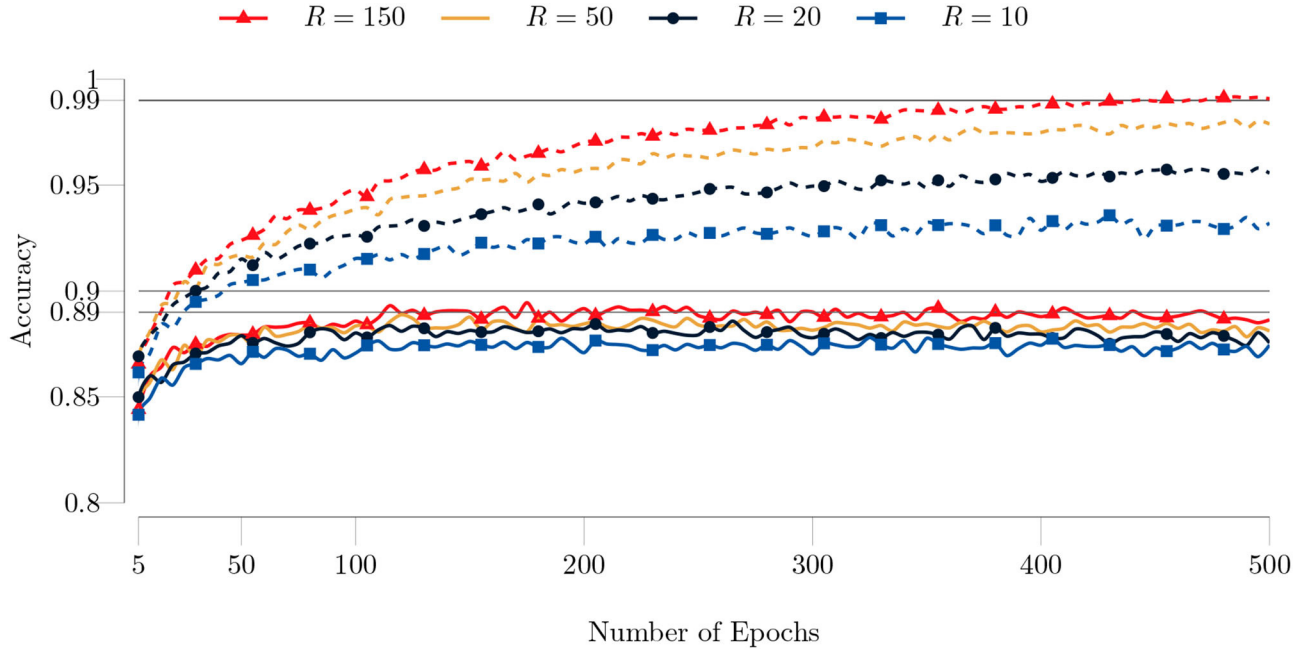


FIGURE 14 | Test (solid lines) and training (dashed lines) accuracies of the type I model for the MNIST dataset with respect to the number of epochs. The full training set (60, 000 images) and test set (10, 000 images) are used. The batch size is set to 200 and the maximum number of GN iterations is set to 1. Both the test and training accuracy increase as the rank increases. Also the improvement in the accuracy gets smaller as the rank increases. The algorithm achieves around 99% training accuracy after 400 epochs.

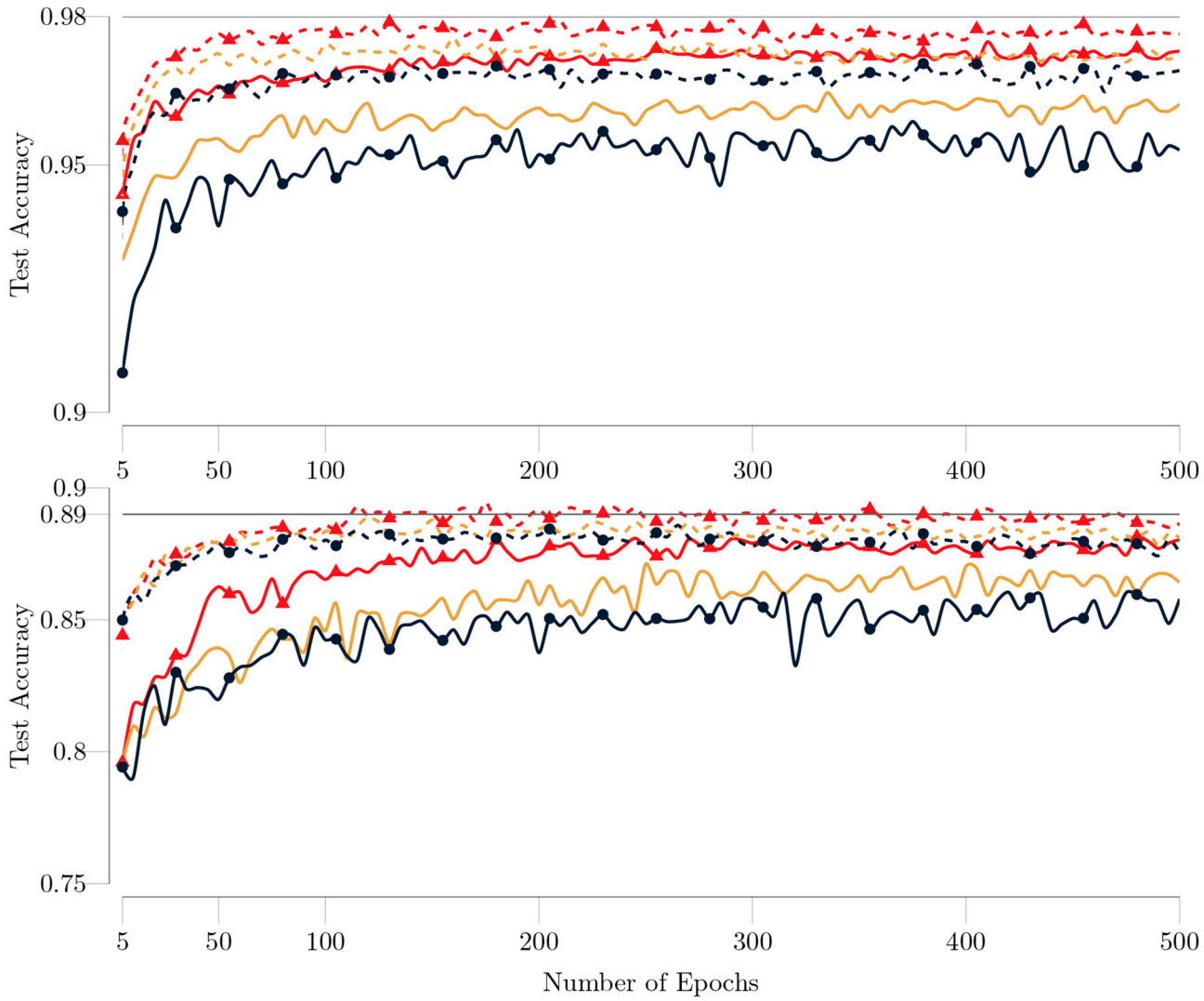


FIGURE 15 | Comparison of test accuracies of the type II model on the MNIST **(top)** and Fashion MNIST **(bottom)** datasets for different maximum number of GN iterations as a function of the number of epochs. The full training set (60, 000 images) and test set (10, 000 images) are used. The batch size is set to 200 and the maximum number of GN iterations is set to 1 (dashed lines) and to 5 (solid lines).

a convolution layer (CNN-PEPS) [22]. We compare in terms of the test accuracy for the Fashion MNIST dataset. We summarize the test accuracy of different models in **Table 2**. TeMPO achieves better accuracy than TT, PEPS and MLP, while optimizing for fewer parameters and using less memory (cf. **Table 1**). The accuracy of TeMPO is lower than CNN-MLP and CNN-PEPS as expected, since it does not use a convolution layer. Note that the accuracy of TeMPO can further be improved by tuning the parameters such as the rank, the number of CG iterations, the trust-region radius, the batch size and the degree of the multivariate polynomial.

5. CONCLUSION AND FUTURE WORK

We presented the TeMPO framework for use in nonlinear optimization problems arising in signal processing, machine learning, and artificial intelligence. We modeled the nonlinearities in these problems by multivariate polynomials represented by low rank tensors. In particular, we investigated the symmetric CPD format in this study. By taking the advantage of low rank symmetric CPD structure, we developed an efficient second-order batch Gauss–Newton algorithm. We demonstrated the efficiency of TeMPO with some illustrative examples, and

TABLE 2 | The test accuracy of different models for the Fashion MNIST dataset.

Model	Test accuracy (%)
TT	88.0
MLP	88.3
PEPS	88.3
TTN	89.0
TeMPO (Type II)	**89.3**
TeMPO (Type I)	**89.9**
CNN–MLP	91.0
CNN–PEPS	91.2

The bold values indicate the results from the proposed methods.

with the blind deconvolution of constant modulus signals. We showed that TeMPO achieves similar or better classification rates than MLPs, TT and PEPS tensor networks on the MNIST and Fashion MNIST datasets while optimizing for fewer parameters and using less memory space.

The non-symmetric and partially symmetric CPD formats are fairly straightforward variants of the symmetric CPD format in which the factor matrices can be mutually different. Efficient algorithms can be developed for multivariate polynomials in these formats by utilizing the derivations presented in this study. We are investigating other tensor formats such as HT and TT in our framework as well. HT and TT require more parameters than the CPD format. However, they break the curse of dimensionality in a numerically stable way. We are also exploring other polynomial bases, and more generally other nonlinear feature maps to further improve the accuracy and numerical stability of our framework.

DATA AVAILABILITY STATEMENT

Publicly available datasets were analyzed in this study. This data can be found at: http://yann.lecun.com/exdb/mnist/; https://github.com/zalandoresearch/fashion-mnist.

AUTHOR CONTRIBUTIONS

MA developed the theory and Matlab implementation. He is the main contributor to the numerical experiments and also wrote the first draft of the manuscript. LD conceived the idea and supervised the project. Both authors contributed to manuscript revision, read, and approved the submitted version.

FUNDING

Research supported by: (1) Flemish Government: This research received funding from the Flemish Government (AI Research Program). LD and MA are affiliated to Leuven. AI-KU Leuven institute for AI, B-3000, Leuven, Belgium. This work was supported by the Fonds de la Recherche Scientifique – FNRS and the Fonds Wetenschappelijk Onderzoek – Vlaanderen under EOS Project no G0F6718N (SeLMA). (2) KU Leuven Internal Funds: C16/15/059, IDN/19/014.

ACKNOWLEDGMENTS

The authors would like to thank E. Evert, N. Govindarajan, and S. Hendrikx for proofreading the manuscript and N. Vervliet for valuable discussions. The authors also thank the two referees whose comments/suggestions helped improve and clarify this manuscript.

REFERENCES

1. Sidiropoulos N, De Lathauwer L, Fu X, Huang K, Papalexakis EE, Faloutsos C. Tensor decomposition for signal processing and machine learning. *IEEE Trans Signal Process*. (2017) 65:3551–82. doi: 10.1109/TSP.2017.2690524
2. Cichocki A, Mandic DP, De Lathauwer L, Zhou G, Zhao Q, Caiafa CF, et al. Tensor decompositions for signal processing applications: from two-way to multiway component analysis. *IEEE Signal Process Mag*. (2015) 32:145–63. doi: 10.1109/MSP.2013.2297439
3. Kolda TG, Bader BW. Tensor decompositions and applications. *SIAM Rev*. (2009) 51:455–500. doi: 10.1137/07070111X
4. Sorber L, Van Barel M, De Lathauwer L. Optimization-based algorithms for tensor decompositions: Canonical polyadic decomposition, decomposition in rank-$(L_r, L_r, 1)$ terms, and a new generalization. *SIAM J Optim*. (2013) 23:695–720. doi: 10.1137/120868323
5. Sorber L, Van Barel M, De Lathauwer L. Unconstrained optimization of real functions in complex variables. *SIAM J Optim*. (2012) 22:879–98. doi: 10.1137/110832124
6. Vervliet N, De Lathauwer L. Numerical optimization based algorithms for data fusion. In: Cocchi M, editor. *Data Fusion Methodology and Applications*. Vol. 31. Amsterdam; Oxford; Cambridge: Elsevier (2019). p. 81–128. doi: 10.1016/B978-0-444-63984-4.00004-1
7. Phan AH, Tichavský P, Cichocki A. Low Complexity Damped Gauss-Newton Algorithms for CANDECOMP/PARAFAC. *arXiv:1205.2584*. (2013) 34:126–47. doi: 10.1137/100808034
8. Vervliet N, De Lathauwer L. A randomized block sampling approach to canonical polyadic decomposition of large-scale tensors. *IEEE J Selec Top Sign Process*. (2016) 10:284–95. doi: 10.1109/JSTSP.2015.2503260
9. Comon P, Jutten C. *Handbook of Blind Source Separation: Independent Component Analysis and Applications*. Oxford; Burlington: Academic Press; Elsevier (2009).
10. Vervliet N, Debals O, Sorber L, De Lathauwer L. Breaking the curse of dimensionality using decompositions of incomplete tensors: tensor-based scientific computing in big data analysis. *IEEE Signal Process Mag*. (2014) 31:71–9. doi: 10.1109/MSP.2014.2329429
11. Vervliet N, Debals O, Sorber L, Van Barel M, De Lathauwer L. *Tensorlab 3.0*. (2016). Available online at https://www.tensorlab.net (accessed December, 2021).
12. Vervliet N. *Compressed Sensing Approaches to Large-Scale Tensor Decompositions*. Leuven: KU Leuven (2018).
13. Vandecappelle M, Vervliet N, Lathauwer LD. Inexact generalized gauss-newton for scaling the canonical polyadic decomposition with non-least-squares cost functions. *IEEE J Selec Top Sign Process*. (2021) 15:491–505. doi: 10.1109/JSTSP.2020.3045911
14. Singh N, Zhang Z, Wu X, Zhang N, Zhang S, Solomonik E. Distributed-memory tensor completion for generalized loss functions in python using new sparse tensor kernels. *arXiv:191002371*. (2021). doi: 10.48550/arXiv.1910.02371
15. Ayvaz M, De Lathauwer L. Tensor-based multivariate polynomial optimization with application in blind identification. In: (*2021) 29th Europian Signal Processing Conference*, (*EUSIPCO)*. Dublin (2021). p. 1080–4. doi: 10.23919/EUSIPCO54536.2021.9616070
16. Grasedyck L, Kressner D, Tobler C. A literature survey of low-rank tensor approximation techniques. *GAMM-Mitteil*. (2013) 36:53–78. doi: 10.1002/gamm.201310004

17. Grasedyck L. Hierarchical singular value decomposition of tensors. *SIAM J Matrix Anal Appl*. (2010) 31:2029–54. doi: 10.1137/090764189
18. Oseledets IV, Tyrtyshnikov EE. Breaking the curse of dimensionality, or how to use SVD in many dimensions. *SIAM J Sci Comput*. (2009) 31:3744–59. doi: 10.1137/090748330
19. Novikov A, Trofimov M, Oseledets IV. Exponential machines. In: *5th International Conference on Learning Representations, ICLR 2017*. Toulon (2017). Available online at: https://openreview.net/forum?id=rkm1sE4tg
20. Stoudenmire EM, Schwab DJ. Supervised learning with tensor networks. In: Lee D, Sugiyama M, Luxburg U, Guyon I, Garnett R, editors. *Advances in Neural Information Processing Systems*. Vol. 29. Barcelona: Curran Associates, Inc. (2016). Available online at: https://proceedings.neurips.cc/paper/2016/file/5314b9674c86e3f9d1ba25ef9bb32895-Paper.pdf
21. Efthymiou S, Hidary J, Leichenauer S. TensorNetwork for machine learning. *arXiv: 190606329*. (2019). doi: 10.48550/arXiv.1906.06329
22. Cheng S, Wang L, Zhang P. Supervised learning with projected entangled pair states. *Phys Rev B*. (2021) 103:125117. doi: 10.1103/PhysRevB.103.125117
23. Guo W, Kotsia I, Patras I. Tensor learning for regression. *IEEE Trans Image Process*. (2012) 21:816–27. doi: 10.1109/TIP.2011.2165291
24. Hendrikx S, Boussé M, Vervliet N, De Lathauwer L. Algebraic and optimization based algorithms for multivariate regression using symmetric tensor decomposition. In: *Proceedings of the (2019) IEEE International Workshop on Computational Advances in Multi-Sensor Adaptive Processing (CAMSAP)*. Guadeloupe (2019). p. 475–9. doi: 10.1109/CAMSAP45676.2019.9022662
25. Rabusseau G, Kadri H. Low-rank regression with tensor responses. In: Lee D, Sugiyama M, Luxburg U, Guyon I, Garnett R, editors. *Advances in Neural Information Processing Systems*. Vol. 29. Barcelona: Curran Associates, Inc. (2016). Available online at: https://proceedings.neurips.cc/paper/2016/file/3806734b256c27e41ec2c6bffa26d9e7-Paper.pdf
26. Yu R, Liu Y. Learning from multiway data: simple and efficient tensor regression. In: Balcan MF, Weinberger KQ, editors. *Proceedings of the 33rd International Conference on Machine Learning, Vol. 48 of Proceedings of Machine Learning Research*. New York, NY (2016). p. 373–81. Available online at: https://proceedings.mlr.press/v48/yu16.html
27. Hou M, Chaib-Draa B. Hierarchical Tucker tensor regression: application to brain imaging data analysis. In: *Proceedings of the (2015) IEEE International Conference on Image Processing (ICIP 2015)*. Québec, QC (2015). p. 1344–8. doi: 10.1109/ICIP.2015.7351019
28. Kar P, Karnick H. Random feature maps for dot product kernels. In: Lawrence ND, Girolami M, editors. *Proceedings of the Fifteenth International Conference on Artificial Intelligence and Statistics, Vol. 22 of Proceedings of Machine Learning Research*. La Palma (2012). p. 583–91. Available online at: https://proceedings.mlr.press/v22/kar12.html
29. Yang J, Gittens A. Tensor machines for learning target-specific polynomial features. *arxiv: 150401697*. (2015). doi: 10.48550/arXiv.1504.01697
30. Rendle S. Factorization machines. In: (*2010) IEEE International Conference on Data Mining*. Sydney (2010). p. 995–1000. doi: 10.1109/ICDM.2010.127
31. Blondel M, Fujino A, Ueda N, Ishihata M. Higher-order factorization machines. In: *Proceedings of the 30th International Conference on Neural Information Processing Systems, NIPS'16*. Red Hook, NY: Curran Associates Inc. (2016). p. 3359–67.
32. Blondel M, Ishihata M, Fujino A, Ueda N. Polynomial networks and factorization machines: new insights and efficient training algorithms. In: *Proceedings of the 33rd International Conference on International Conference on Machine Learning*. Vol. 48. New York, NY (2016). p. 850–8.
33. Nocedal J, Wright S. *Numerical Optimization*. New York, NY: Springer (2006).
34. Kruskal JB. Three-way arrays: rank and uniqueness of trilinear decompositions, with application to arithmetic complexity and statistics. *Linear Algeb Appl*. (1977) 18:95–138. doi: 10.1016/0024-3795(77)90069-6
35. Sidiropoulos ND, Bro R. On the uniqueness of multilinear decomposition of N-way arrays. *J Chemometr*. (2000) 14:229–39. doi: 10.1002/1099-128X(200005/06)14:3<229::AID-CEM587>3.0.CO;2-N
36. Domanov I, De Lathauwer L. On the uniqueness of the canonical polyadic decomposition of third-order tensors – Part ii: uniqueness of the overall decomposition. *SIAM J Matrix Anal Appl*. (2013) 34:876–903. doi: 10.1137/120877258
37. Domanov I, De Lathauwer L. Canonical polyadic decomposition of third-order tensors: relaxed uniqueness conditions and algebraic algorithm. *arXiv:1501.07251*. (2017) 513:342–75. doi: 10.1016/j.laa.2016.10.019
38. Boyd JP, Ong JR. Exponentially-convergent strategies for defeating the Runge phenomenon for the approximation of non-periodic functions, part I: single-interval schemes. *Commun Comput Phys*. (2009) 5:484–97.
39. Trefethen LN. *Approximation Theory and Approximation Practice, Extended Edition*. Philadelphia, PA: SIAM (2019). doi: 10.1137/1.9781611975949
40. De Lathauwer L, De Moor B, Vandewalle J. On the best rank-1 and rank-$(R_1, R_2, \cdots, R_N)$ approximation of higher-order tensors. *SIAM J Matrix Anal Appl*. (2000) 21:1324–42. doi: 10.1137/S0895479898346995
41. Zhang T, Golub G. Rank-one approximation to high order tensors. *SIAM J Matrix Anal Appl*. (2001) 23:534–50. doi: 10.1137/S0895479899352045
42. Guan Y, Chu MT, Chu D. SVD-based algorithms for the best rank-1 approximation of a symmetric tensor. *SIAM J Matrix Anal Appl*. (2018) 39:1095–115. doi: 10.1137/17M1136699
43. Nie J, Wang L. Semidefinite relaxations for best rank-1 tensor approximations. *SIAM J Matrix Anal Appl*. (2013) 35:1155–79. doi: 10.1137/130935112
44. Brachat J, Comon P, Mourrain B, Tsigaridas E. Symmetric tensor decomposition. *Linear Algeb Appl*. (2010) 433:1851–72. doi: 10.1016/j.laa.2010.06.046
45. Alexander J, Hirschowitz A. Polynomial interpolation in several variables. *Adv Comput Math*. (1995) 4:201–22.
46. Debals O. *Tensorization and Applications in Blind Source Separation*. Leuven: KU Leuven (2017).
47. Blondel M, Niculae V, Otsuka T, Ueda N. Multi-output Polynomial Networks and Factorization Machines. In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017*. Long Beach, CA (2017). p. 3349–59.
48. Khoromskij BN. *Tensor Numerical Methods in Scientific Computing*. Berlin; Boston: De Gruyter (2018). doi: 10.1515/9783110365917
49. Margossian CC. A review of automatic differentiation and its efficient implementation. *WIREs Data Mining Knowl Discov*. (2019) 9:e1305. doi: 10.1002/widm.1305
50. van der Veen AJ, Paulraj A. An analytical constant modulus algorithm. *IEEE Trans Signal Process*. (1996) 44:1136–55. doi: 10.1109/78.502327
51. Zarzoso V, Comon P. Optimal step-size constant modulus algorithm. *IEEE Trans Commun*. (2008) 56:10–3. doi: 10.1109/TCOMM.2008.050484
52. Boussé M, Vervliet N, Domanov I, Debals O, De Lathauwer L. Linear systems with a canonical polyadic decomposition constrained solution: algorithms and applications. *Numer Linear Algeb Appl*. (2018) 25:e2190. doi: 10.1002/nla.2190
53. Gargiani M, Zanelli A, Diehl M, Hutter F. On the promise of the stochastic generalized Gauss-Newton method for training DNNs. *arXiv: 200602409*. (2020). doi: 10.48550/arXiv.2006.02409
54. Kingma DP, Ba J. Adam: a method for stochastic optimization. In: Bengio Y, LeCun Y, editors. *International Conference on Learning Representations, ICLR 2015*. 3rd Edn. San Diego, CA (2015). Available online at: http://arxiv.org/abs/1412.6980
55. De Brabanter K, Karsmakers P, Ojeda F, Alzate C, De Brabanter J, Pelckmans K, et al. *LS-SVMlab Toolbox User's Guide Version 1.8*. Leuven: ESAT-STADIUS (2010). p. 10–46.
56. Suykens JAK, Van Gestel T, De Brabanter J, De Moor B, Vandewalle J. *Least Squares Support Vector Machines*. Singapore: World Scientific (2002). doi: 10.1142/5089
57. Ljung L. *System Identification: Theory for the User*. 2nd ed. Upper Saddle River, NJ: Prentice Hall (1999). doi: 10.1002/047134608X.W1046
58. Johnson R, Schniter P, Endres TJ, Behm JD, Brown DR, Casas RA. Blind equalization using the constant modulus criterion: a review. *Proc IEEE*. (1998) 86:1927–50. doi: 10.1109/5.720246
59. van der Veen AJ. Algebraic methods for deterministic blind beamforming. *Proc IEEE*. (1998) 86:1987–2008. doi: 10.1109/5.720249
60. De Lathauwer L. Algebraic techniques for the blind deconvolution of Constant Modulus signals. In: *Proceedings of the 12th European Signal Processing Conference (EUSIPCO 2004)*. Vienna (2004). p. 225–8.
61. Householder AS. Unitary triangularization of a nonsymmetric matrix. *J ACM*. (1958) 5:339–42. doi: 10.1145/320941.320947

62. Deng L. The MNIST database of handwritten digit images for machine learning research. *IEEE Sign Process Mag*. (2012) 29:141–2. doi: 10.1109/MSP.2012.2211477
63. Xiao H, Rasul K, Vollgraf R. Fashion-MNIST: a novel image dataset for benchmarking machine learning algorithms *arXiv:1708.07747*. (2017). doi: 10.48550/arXiv.1708.07747
64. Stoudenmire EM. Learning relevant features of data with multi-scale tensor networks. *Quant Sci Technol*. (2018) 3:034003. doi: 10.1088/2058-9565/aaba1a

Conflict of Interest: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict ofinterest.

Publisher's Note: All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher.

frontiers | Frontiers in Applied Mathematics and Statistics

ORIGINAL RESEARCH
published: 07 April 2022
doi: 10.3389/fams.2022.806537

Iterator-Based Design of Generic C++ Algorithms for Basic Tensor Operations

*Cem Savas Bassoy**

Fraunhofer IOSB, Ettlingen, Germany

Numerical tensor calculus has recently gained increasing attention in many scientific fields including quantum computing and machine learning which contain basic tensor operations such as the pointwise tensor addition and multiplication of tensors. We present a C++ design of multi-dimensional iterators and iterator-based C++ functions for basic tensor operations using mode-specific iterators only, simplifying the implementation of algorithms with recursion and multiple loops. The proposed C++ functions are designed for dense tensor and subtensor types with any linear storage format, mode and dimensions. We demonstrate our findings with Boost's latest uBlas tensor extension and discuss how other C++ frameworks can utilize our proposal without modifying their code base. Our runtime measurements show that C++ functions with iterators can compute tensor operations at least as fast as their pointer-based counterpart.

Keywords: tensor *n*-rank, N-way array, multi-dimensional array, tensor computations, multi-dimensional iterator, software design and development

OPEN ACCESS

Edited by:
Paolo Bientinesi,
Umeå University, Sweden

Reviewed by:
Richard Veras,
University of Oklahoma, United States
Jiajia Li,
College of William & Mary, United States

***Correspondence:**
Cem Savas Bassoy
cem.bassoy@iosb.fraunhofer.de

Specialty section:
This article was submitted to Mathematics of Computation and Data Science, a section of the journal Frontiers in Applied Mathematics and Statistics

Received: *31 October 2021*
Accepted: *26 January 2022*
Published: *07 April 2022*

Citation:
Bassoy CS (2022) Iterator-Based Design of Generic C++ Algorithms for Basic Tensor Operations. Front. Appl. Math. Stat. 8:806537. doi: 10.3389/fams.2022.806537

1. INTRODUCTION

Numerical tensor calculus can be found in many application fields, such as signal processing [1], computer graphics [2, 3], and data mining [4, 5] in which tensors are attained by, e.g., discretizing multi-variate functions [6] or by sampling multi-modal data [7]. Tensors are interpreted as generalized matrices with more than two dimensions and are, therefore, also referred to as hypermatrices [8]. Similar to matrix computations, most numerical tensor methods are composed of basic tensor operations such as the tensor-tensor, tensor-matrix, tensor-vector multiplication, the inner and outer product of two tensors, the Kronecker, Hadamard and Khatri-Rao product [9–11]. Examples of such methods containing basic tensor operations are the higher-order orthogonal iteration, the higher-order singular value decomposition [12], the higher-order power method and variations thereof.

High-level libraries in Python or Matlab, such as NumPy, TensorLy, or TensorLab[1] offer a variety of tensor types and corresponding operations for numerical tensor computations. However, in case of tensor multiplication operations tensors are dynamically unfolded in order to make use of optimized matrix operations, consuming at least twice the memory than their in-place alternatives [13]. Depending on the program, Python or Matlab can also introduce runtime overhead due to just-in-time compilation or interpretation and automatic resource management.

To offer fast execution times with minimal memory consumption, many tensor libraries are implemented in C++ which provides a simple, direct mapping to hardware and zero-overhead

[1]https://numpy.org, http://tensorly.org, https://www.tensorlab.net.

abstraction mechanism [14, 15]. Their programming interface is close to the mathematical notation supporting elementwise and complex multiplication tensor operations [16–21]. All libraries offer a family of tensor classes that are parameterized by at least the element type. The library presented in [21] also parameterize the tensor template by the tensor order and dimensions. Tensor elements are linearly arranged in memory either according to the first-order or the last-order storage format. Most libraries use expression templates to aggregate and delay the execution of mathematical expressions for a data-parallel and even out-of-order execution [17, 19]. Some libraries can express the general form of the tensor-tensor multiplication with Einstein's summation convention using strings or user-defined objects. For instance, expressions like `C["ijk"]=A["ilj"]*B["kl"]` or `C(i,j,k)=A(i,l,k)*B(k,l)` specify a 2-mode multiplication of a 3-dimensional with a matrix. The interface can be very convenient utilized if the application or numerical method uses a fixed tensor order or contraction mode. However, many numerical methods such as the higher-order orthogonal iteration consist of variable tensor multiplications preventing the use of aforementioned expressions. In such cases, flexible interfaces and functions similar to the one presented in [22] are required allowing, e.g., the contraction mode to depend on other variables. A comprehensive and recent overview of the tensor software landscape is provided in [23] including all of the previously mentioned C++ libraries.

Most of the above mentioned libraries implement tensor operations using pointers, single and multi-indices. Accessing tensor elements with multi-indices, however, can slow down the execution of a recursively defined tensor function by a factor that is equal to the recursion depth and tensor order [24]. Using single indices or raw pointers on the other hand requires a combination of induction variables with mode-specific strides. This can be inconvenient and error-prone, especially when library users want to modify or extend C++ functions. The authors in [25] suggest to parameterize C++ functions in terms of tensor types and their proxies with which mode-specific iterators can be generated using the member functions `begin` and `end`. Index operations are hidden from the user by offering a simple iterator increment operation that is able to adjust its internal data pointer according to a predefined stride. However, their `begin` and `end` functions do not allow to specify a mode. The authors in [26] propose to use member functions `begin` and `end` of a tensor type that can generate mode-specific iterators. The mode is a non-type template parameter of the iterator requiring the recursion index and the contraction modes to be compile-time parameters. Similar to the aforementioned approaches, tensor functions are defined in terms of tensor types which makes the specification of iterator requirements difficult.

In this article, we present iterator-based C++ algorithms for basic tensor operations that have been discussed in [22] as part of a Matlab toolbox. Our software implementation follows the design pattern that has been used in the Standard Template Library (STL) and separates tensor functions from tensor types with the help of iterators only [27]. The separation helps to define iterator and function templates that are not bound to particular tensor and iterator types, respectively. We present C++ functions such as `for_each` and `transform` that perform unary and binary operations on tensor and subtensor elements. Our discussion also includes more complex multiplication operations such as tensor-vector (`ttv`), tensor-matrix (`ttm`), and the tensor-tensor multiplication (`ttt`). While we demonstrate their usability with Boost's uBlas tensor extension, the proposed C++ templates can be instantiated by tensor types that provide pointers to a contiguous memory region.

To our best knowledge, we are the first to propose a set of basic tensor functions that can process tensor types without relying on a specific linear data layout, eliminating the need to provide multiple algorithms for similar types. While a discussion of optimization techniques for data locality or parallel execution of tensor operations are beyond the scope of this article, we provide algorithmic changes to all proposed tensor functions to increase spatial locality. Moreover, we demonstrate that the introduced iterator abstraction does not penalize the performance of iterator-based C++ functions. On the contrary, our performance measurements with approximately 1,800 differently shaped tensors show that iterator- based functions compute elementwise tensor operations and the tensor-vector product at least as fast as pointer-based functions.

The remainder of the paper is organized as follows: Section 2 introduces mathematical notations used in this work and provides an overview of data organization for dense tensor and subtensor types. Section 3 describes Boost's uBlas tensor extension and class templates for tensors and subtensors. Section 4 introduces multi-dimensional iterators for a family of tensor types supporting any linear storage format. Section 5 discusses the design and implementation of tensor operations using multi-dimensional iterators. Section 6 presents runtime measurements of iterator- and pointer-based implementations of two elementwise tensor operations. Lastly, section 7 presents some conclusions of our work.

2. PRELIMINARIES

2.1. Mathematical Notation

A tensor is defined as an element of the tensor space that is given by the tensor product of vector spaces typically over the real or complex numbers [28]. For given finite basis of the vector spaces, tensors can be represented by multi-dimensional arrays [8]. We do not distinguish between tensors and multi-dimensional arrays and allow their elements to be bool or integer types. The number of dimensions is called the tensor order and is denoted by the letter p. Tensors are denoted by bold capital letters with an underscore, e.g., $\underline{\mathbf{A}}$ with $\underline{\mathbf{A}} = (a_{\mathbf{i}})_{\mathbf{i}\in\mathbf{I}}$ where $\mathbf{i}$ is a multi-index $\mathbf{i} = (i_1, i_2, \ldots, i_p)$ with $i_r \in I_r$ for all $1 \leq r \leq p$. The r-th index set I_r is defined as $I_r := \{1, 2, \ldots, n_r\}$ for all $1 \leq r \leq p$ with $n_r \in \mathbb{N}$. $\mathbf{n} = (n_1, \ldots, n_p)$ is called a dimension tuple of a p-dimensional tensor. The Cartesian product of all index sets of a p-order tensor $\underline{\mathbf{A}}$ is called the multi-index set $\mathbf{I}$ with $\mathbf{I} = I_1 \times I_2 \times \cdots \times I_p$. Elements of a p-dimensional tensor $\underline{\mathbf{A}}$ are given by $\underline{\mathbf{A}}(i_1, i_2, \ldots, i_p) = a_{i_1 i_2 \ldots i_p}$ or $\underline{\mathbf{A}}(\mathbf{i}) = a_{\mathbf{i}}$ with $\mathbf{i} \in \mathbf{I}$. Matrices have two dimensions and will be represented without an underscore $\mathbf{B}$. Vectors are given by small bold letters such as $\mathbf{b}$ where one of the first two dimensions are equal to or greater

than one. A tensor is a scalar if all dimensions are equal to one and denoted by small, non-bold letters.

A subtensor $\underline{\mathbf{A}}'$ of a tensor $\underline{\mathbf{A}}$ is a reference to a specified region or domain of $\underline{\mathbf{A}}$ and has the same order p and data layout $\boldsymbol{\pi}$ as the referenced tensor. It can be regarded as a lightweight handle with a dimension tuple $\mathbf{n}'$ where the subtensor dimensions satisfy $n'_r \leq n_r$ for $1 \leq r \leq p$. The r-th index set I'_r of a subtensor and its multi-index set $\mathbf{I}'$ are analogously defined to I_r with $I'_r \subseteq I_r$ and $\mathbf{I}$, respectively. Each dimension n'_r and the corresponding index subset I'_r are determined by f_r, t_r, and l_r where $f_r \in I_r$ and $l_r \in I_r$ are the lower and upper bound of the index range with $1 \leq f_r \leq l_r \leq n_r$. The integer t_r defines the step size for the r-th dimension satisfying $t_r \in \mathbb{N}$ for $1 \leq r \leq p$. The shape tuple $\mathbf{n}' = (n'_1, \ldots, n'_p)$ of a subtensor is given by $n'_r = \lfloor (l_r - f_r)/t_r \rfloor + 1$. Elements of a p-dimensional subtensor $\underline{\mathbf{A}}'$ are given by $\underline{\mathbf{A}}'(\mathbf{i}') = a'_{\mathbf{i}'}$ with $\mathbf{i}' \in \mathbf{I}'$.

Assuming a simple linear (flat) memory model, dense tensors shall be stored contiguously in memory. The (absolute) memory locations of tensor elements are given by $k = k_0 + j \cdot \delta$ with $k_0 \in \mathbb{N}_0$ being the memory location of the first tensor element and δ being the number of bytes required to store tensor elements. We call $J := \{0, 1, \ldots, \prod_{r=1}^{p} n_r - 1\}$ the single index set of $\underline{\mathbf{A}}$ where each $j \in J$ is the relative position of the j-th tensor element denoted by $\underline{\mathbf{A}}[j]$. A subtensor $\underline{\mathbf{A}}'$ of a tensor $\underline{\mathbf{A}}$ has its own single index set J' with $\prod_{r=1}^{p} n'_r$ elements. We write $\underline{\mathbf{A}}'[j']$ to denote the j'-th subtensor element.

2.2. Data Organization and Layout

The tensor layout or storage format of a dense tensor defines the ordering of its elements within a linearly addressable memory and, therefore, the transformation between multi-indices and single indices. A p-order tensor $\underline{\mathbf{A}}$ with a dimension tuple $\mathbf{n}$, has $(\prod_r n_r)!$ possible orderings where only a subset of those are considered in practice. In case of two dimensions, most programming languages arrange matrix elements either according to the row- or column-major storage format. More sophisticated non-linear layout or indexing functions have been investigated for instance in [29, 30] with the purpose to increase the data locality of dense matrix operations.

The most prominent element layouts are first- and last-order storage formats. The former format is defined in the Fortran, the latter in the C and C++ language specification, respectively. Any linear layout can be expressed in terms of a permutation tuple $\boldsymbol{\pi}$. The q-th element π_q corresponds to an index subscript r of a multi-index i_r with the precedence q where $i_r \in I_r$ and $1 \leq q, r \leq p$. In case of the first-order format, the layout tuple is defined as $\boldsymbol{\pi}_F := (1, 2, \ldots, p)$ where the precedence of the dimension ascends with increasing index subscript. The layout tuple of the last-order storage format is given by $\boldsymbol{\pi}_L := (p, p-1, \ldots, 1)$.

Given a layout tuple $\boldsymbol{\pi}$ and the shape tuple $\mathbf{n}$, elements of a stride tuple $\mathbf{w}$ are given by $w_{\pi_r} = 1$ for $r = 1$ and $w_{\pi_r} = \prod_{q=1}^{r-1} n_{\pi_q}$ otherwise, with $1 \leq w_{\pi_q} \leq w_{\pi_r}$ for $1 \leq q < r \leq p$, see also Equation (2) in [24]. The q-th stride w_q is a positive integer and defines the number of elements between two elements with an identical multi-index except that their q-th index differs by one. Fortran stores tensor elements according to the first-order storage format with $\mathbf{w}_F = (1,\ n_1,\ n_1 \cdot n_2,\ \ldots,\ \prod_{r=1}^{p-1} n_r)$. In case of the last-order storage format $\boldsymbol{\pi}_L = (p, p-1, \ldots, 1)$, the stride tuple is given by $\mathbf{w}_L = (\prod_{r=2}^{p} n_r, \prod_{r=3}^{p} n_r, \ldots, n_p, 1)$ which is used by the C and C++ language for the data layout of the built-in multi-dimensional arrays.

3. BOOST.UBLAS TENSOR EXTENSION

Initially equipped with basic matrix and vector operations, Boost's uBlas has been recently extended with tensor templates and corresponding tensor operations to support multi-linear algebra applications[2]. Tensor order, dimensions and contraction modes (if applicable) of the tensor and subtensor types are runtime variable. Common arithmetic operators are overloaded and evaluated using expression templates. In the following, we will only use the namespace `std` to denote the standard library namespace and skip `boost::numeric::ublas`. Boost's uBlas tensor extension offers a variety of basic dense tensor operations offering at least four important tensor functionality categories that have been discussed in [23].

3.1. Tensor and Subtensor Templates

The `tensor` template class represents a family of tensor types and adapts a contiguous container such as `std::vector`. It is designed to organize multi-dimensional data and to provide access with multi-indices and single indices.

```
template <class T,
          class F = first_order,
          class C = std::vector<value_type>>
class tensor;
```

The element type `T` of `tensor` needs to fullfill the requirements specified by the container type `C` and needs to support all basic arithmetic scalar operations such as addition, subtraction, multiplication, and division. The container `C` type must satisfy the requirements of a contiguous container. By default, if no container class is specified, `std::vector` is used. Public member types such as `value_type`, `size_type`, `difference_type`, `pointer`, `reference`, and `iterator` are derived from the `container` type which stores elements of type `value_type` and takes care of the memory management. The memory space for `tensor` is dynamically allocated by `std::vector::allocator_type`. Public member functions are provided in order to construct, copy and move tensors. Data elements can be assigned to the tensor using the assignment operator `=`. Elements can be accessed with a single index using the access operator `[]` and multi-indices with the function call operator `()`. The user can conveniently create subtensors with the function call operator `()`. Size and capacity member functions such as `size()`, `empty()`, `clear()`, and `data()` are provided as well. The user has multiple options to instantiate `tensor` types. The default constructor creates an empty tensor of order zero with an empty shape tuple. The following expression instantiats a three-dimensional tensor `A` with the extents 4, 2, and 3 with elements of type `double`.

[2]See GSoC18 link for the project description, Github link for the initial implementation and Github link for the most current development.

```
auto A = tensor<float>{4,2,3};
```

The user can also specify dimensions for each mode using the `extents` class from which the tensor order and size of the data vector is derived. The layout tuple is initialized according to the first-order storage format if not specified otherwise. Once the layout and dimensions are initialized, the constructor computes strides according to the computation in subsection 2.2 and Equation (2) in [24]. The following code snippet shows a possible instantiation of a three-dimensional tensor with a last-order storage format.

```
auto A = tensor<double,last_order>(extents{4,2,3});
```

The copy assignment operators `()` of the `tensor` class template are responsible for copy data and protecting against self-assignment. The user can expect the source and destination `tensor` class instances to be equal and independent after the copy operation. Two tensors are equal if they have the same shape tuple, tensor order and elements with the same multi-index independent of their layout tuple. Besides the type of the data elements, the user can change the content and the size of all member variables at runtime. The `subtensor` template class is a proxy of `tensor` for conveniently reference a subset of `tensor` elements.

```
template <class T>
class subtensor;
```

The `tensor` template specializes `subtensor` with `tensor<value_type,container>` such that `tensor::subtensor_t` equals `subtensor<tensor<value_type,container>>`. In general, `T` needs to provide an overloaded access operator and function call operator for accessing contiguously stored tensor elements. The `subtensor` template contains a reference of the viewed tensor instance, i.e., `subtensor::tensor_t`, a pointer to the first element of type `value_type*`, extent ranges of a single dimension using the class `span`, extents of type `size_type`, strides of type `size_type` and also provides the same public member types and methods as `tensor` allowing both types to be used in free functions interchangeably. A `subtensor` instance neither owns nor tracks the referenced `tensor` object. It might become invalid whenever the corresponding `tensor` instance does not exist any more.

The constructor of `subtensor` takes a reference of `subtensor::tensor_t` and might take range types such as `span` and `std::integral` types as additional arguments that specify the multi-index space of a `subtensor` instance. The r-th `span` instance defines an index set I'_r that is a subset of the index set I_r of a selected `tensor` instance. A `subtensor` instance without any `span` objects references all elements of a `subtensor::tensor_t` object. The `tensor` template provides an overloaded function call operator with a template parameter pack which simplifies the construction of a `subtensor` subject. For instance, if `A` is of type `tensor<float>` with a dimension tuple `(3, 4, 2)`, then `S` of the following expression is of type `subtensor<tensor<float>>` and has the dimensions 2, 2, 1.

```
auto S = A ( span(1,2), span(2,3), 1 );
```

The pointer to the first subtensor element is computed by adding an offset j^* to the pointer of the first tensor element. The offset j^* is computed by combining p lower bounds f of the `span` instances using the index function λ in Equation (1) in [24] such that $j^* = \lambda_{\mathbf{w}}(\mathbf{f})$ with $\mathbf{f} = (f_1, \ldots, f_p)$ where $\mathbf{w}$ is the stride tuple of a `tensor` and f_r is the lower bound of the r-th `span` instance.

3.2. Multi-Index Access

The `tensor` template provides multiple overloaded function call operators for conveniently accessing elements with multi-indices and scalar memory indices. The function call operator is a variadic template that computes the inner product of the stride and multi-index tuple in order to transform multi-indices onto single indices. Hence, the user can define the following statement which converts a three-dimensional tensor `A` into an identity tensor with ones in the superdiagonals.

```
for(auto i = 1u; i <= n; ++i)
  A(i,i,i) = 1.0;
```

Note that the statement is valid independent of `A`'s layout tuple. The template `tensor` additionally allows to dynamically specify multi-indices using `std::vector`. In that case the argument of the function call is given by `std::vector<std::size_t>(p,i)` where `p` is the tensor order. Using multi-indices abstracts from the underlying data layout and enables the user to write layout invariant programs as all elements have a unique multi-index independent of the data layout. Note that accessing elements of a p-dimensional tensor $\underline{\mathbf{A}}$ with multi-indices involves a multi-index to memory index transformation that is given by $\lambda_{\mathbf{w}}(\mathbf{i}) = \sum_{r=1}^{p} w_r(i_r - 1)$ where p is the tensor order with $p > 1$ and $\mathbf{w}$ is the stride tuple of $\underline{\mathbf{A}}$, see also Equation (1) in [24]. For fixed stride tuples $\mathbf{w}_F$ and $\mathbf{w}_L$, the index functions $\lambda_{\mathbf{w}_F}$ and $\lambda_{\mathbf{w}_L}$ coincide with definitions provided in [25, 29]. Tensor elements can also be accessed with a single index using the overloaded access operator of `tensor`. This is convenient whenever the complete memory index set needs to be accessed independent of the tensor layout or order of data access is not relevant for the implementation of the tensor operation. For instance, `A` with any dimensions and storage format can be initialized by the following statement.

```
for(auto j = 0u; j < A.size(); ++j)
  A[j] = 0;
```

In contrast to an access with multi-indices, accessing tensor elements with single indices does not involve index transformations. However, most of the more complex tensor operations such as the tensor transposition require some type of multi-index access.

Subtensor elements can be similarly accessed using multi-indices with the `subtensor`'s overloaded function call operator. Given the previously defined `subtensor` instance `S` with the dimensions `(2,2,1)`, all diagonal elements can be set to `1` using a single for-loop where `m` is equal to 2.

```
for(auto i = 1; i <= m; ++i)
  S(i,i,1) = 1;
```

Similar to the tensor case, the relative memory location needs to be computed as well, using index function λ transforming every index $\mathbf{i}' \in I'_r$ into an index of the set I_r with $j = j^* + \lambda_{\mathbf{w}''}(\mathbf{i}')$ where j^* is the relative memory location of the first subtensor element. Elements of the stride tuple $\mathbf{w}''$ is given by $w''_r = w'_r t_r$ for $1 \leq r \leq p$ in which $\mathbf{w}'$ is computed with $\mathbf{n}'$. The `subtensor` template also provides an overloaded access operator with a single index. The following statement sets all subtensor elements to zero.

```
for(auto j = 0u; j < S.size(); ++j)
  S[j] = 0;
```

In contrast to the tensor case, accessing a relative memory location of subtensor's element with a single index involves its transformation using the index function λ and its inverse λ^{-1}. Given a valid single index $j' \in J'$ and the relative memory location of the first subtensor element j^*, the relative memory location $j \in J$ of a subtensor element at index j' is given by $j = j^* + \lambda'_{\mathbf{w}'',\mathbf{w}'}(j')$ with $\lambda'_{\mathbf{w}'',\mathbf{w}'}$ being a composition of the index functions $\lambda_{\mathbf{w}''}$ and $\lambda^{-1}_{\mathbf{w}'}$. The latter is the inverse index function is given by $\lambda^{-1}_{\mathbf{w}}(j) = \mathbf{i}$ where $i_r = \lfloor x_r / w_r \rfloor + 1$ with $x_{\pi_r} = x_{\pi_{r+1}} - w_{\pi_{r+1}} \cdot (i_{\pi_{r+1}} - 1)$ for $r < p$ and $i_{\pi_p} = \lfloor j / w_{\pi_p} \rfloor + 1$, see also [24].

4. MULTI-DIMENSIONAL ITERATOR

C++ iterators are class templates that can traverse and access C++ container elements. They help to decouple the dependency between C++ container and C++ algorithms by parameterizing the latter in terms of iterators only. The following class template `multi_iterator` simplifies the iteration over a multi-index set of a tensor or subtensor independent of their storage formats and helps to decouple tensor types from tensor functions.

```
template<class iterator>
class multi_iterator;
```

The template parameter `iterator` should be a valid template parameter for `std::iterator_traits` with which iterator attributes can be queried. The `tensor` and `subtensor` templates can specialize `multi_iterator` with their corresponding `pointer` or `iterator` type. The constructor of `multi_iterator` initializes three private member variables, the current pointer of type `std::iterator_traits<iterator>::pointer`, a pointer to the strides of type `const std::size_t*` and a stride of type `std::size_t`. The following statement specializes the multi-dimensional iterator template and instantiates `it`.

```
auto it = multi_iterator<pointer>(k,w,1);
```

The argument `k` is a pointer to the first tensor element and `w` a pointer to the first stride tuple element. The last argument 1 selects the second stride from `w`. The copy-assignment operator of `iterator` copies the current position `k`, the pointer to the stride tuple `w`, and the stride `wc`. We consider two dimension-based iterators `i1` and `i2` equal if the current positions `i1.k`, `i2.k` and the strides `i1.wc`, `i2.wc` of the iterators are equal. Therefore, the statement `(i1=i2) == i2` is considered true as both iterators have equal position and stride after the assignment `(i1=i2)`.

The following example illustrates the difference of two ranges that are created by the random access iterator type `iterator` of `std::vector` and the `multi_iterator<pointer>` type. Let `A` be a three-dimensional dense tensor with elements of type `float` contiguously stored according to the first-order storage format. Let also `k` be a pointer to the first element of `A` initialized with `A.data()`. Given `4, 3, 2` be `A`'s extents and `w` the stride tuple with `(1,4,12)`, respectively, the two statements instantiate iterator pairs.

```
iterator                first(k    ),  last(k+w[2]    );
multi_iterator<pointer> mfirst(k,w,1), mlast(k+w[2],w,1);
```

The first half-open range `[first,last)` covers all tensor elements with memory indices from `0` and to `12`. The second range only covers elements with the multi-indices $(1, i, 1)$ for $1 \leq i \leq 2$ which corresponds to a mode-2 tensor fiber, i.e., the first row of the frontal tensor slice. Applying the index function λ, the relative memory positions of `A`'s elements are at position `0`, `4` and `8`. The iteration over the second mode of `A` can be performed with both iterator pairs.

```
for(;  first !=  last;  first+=w[1]) {  *first = 5.0; }
for(; mfirst != mlast; mfirst+=1   ) { *mfirst = 5.0; }
```

The statements initialize the first row of `A`. The first statement uses the C++ standard random-access iterator `first` which is explicitly incremented with the second stride `w[1]`. The same operation can be accomplished with the multi-dimensional iterator `mfirst` which is initialized and internally incremented with the second stride `w[1]`. Our implementation of multi-dimensional iterators can also be used with C++ algorithms of the standard library. For instance, `std::fill` can be used together with `mfirst` and `mlast` to initialize the first row of `A`.

```
std::fill(mfirst, mlast, 5.0);
```

The user can introduce member functions `begin` and `end` of `tensor` and `subtensor` or implement free functions, both simplifying the instantiation of multi-dimensional iterators. The user needs to specify a one-based mode that is greater than zero and equal to or smaller than the tensor order. Both functions could also allow to specify a multi-index with `std::vector<std::size_t>` and define the displacement within the multi-index space except for the dimension `dim`. In the following, `begin` and `end` shall be member functions of the `tensor` and `subtensor` types. The aforementioned initialization of `A`'s first row can be performed in one line which first generates mode-specific iterates using `begin` and `end` for the first mode.

```
std::fill(A.begin(1),A.end(1),5.0);
```

Note that the user can perform the initialization independent of `A`'s storage format. Moreover, fibers with different modes using C++ algorithms of the standard library can be combined. The following statement for instance computes the inner product of a mode-3 and mode-2 fiber.

```
std::inner_product(A.begin(3),A.end(3),
                   B.begin(2),0.0);
```

Listing 1 | Nested-loop with multi-dimensional iterators for tensor types of order 3 with any linear storage format.

```
for(auto it3=A.begin(3); it3!=A.end(3); ++it3)
  for(auto it2=it3.begin(2); it2!=it3.end(2); ++it2)
    for(auto it1=it2.begin(1); it1!=it2.end(1); ++it1)
      *it1 = v;
```

Again, `A` and `B` can be of different types (such as tensor or subtensor) with different storage formats. The user can invoke `begin` and `end` function with no mode or mode `0` with which the single-index space of a tensor or subtensor can be iterated through.

```
std::fill(A.begin(),A.end(), 0.0);
```

Note that range-based for-loops can also be used instead of `std::fill`. Similar to the tensor type, the `multi_iterator` template provides two methods `begin` and `end` with which multi-dimensional iterators can be instantiated. The new instantiated iterators have the same pointer position and stride tuple reference but a new stride depending on the argument which specifies the mode. For instance, a multi-dimensional iterator `it` can be used to define a multi-dimensional iterator pair that is able to iterate along the third mode.

```
auto first = it.begin(3), auto last = it.end(3);
```

Listing 1 illustrates the initialization of a three-dimensional tensor or subtensor `A` with multi-dimensional iterators. The code example consists of three nested for-loops. Within each loop a multi-dimensional iterator `it{r}` is initialized using the `begin` and `end` member function of either the tensor `A` or a multi-dimensional iterator of the previous loop. The iterator number corresponds with the position within the stride tuple so that `it{r}` will be internally incremented with the `w[r-1]` stride in case of tensors and with `w[r-1]*s[r-1]` in case of subtensors where `s[r-1]` is the step size. The inner loop assigns value `v` to the column elements of the (`it3`,`it2`)-th frontal slice. The innermost loop can be replaced with the following statement.

```
std::fill(it1.begin(1), it1.end(1), v);
```

In contrast to the iterator design in [25, 26], our iterator instances are able to clone themselves for different modes. Tensor `A` in the outer-most loop is replaceable by a multi-dimensional iterator `it3` that is generated in a previous statement with the expression `A.begin(3)`. In the next section we present tensor functions that iterate over the multi-index space of multi-dimensional tensors and subtensors with arbitrary storage format using multi-dimensional iterators only.

5. TENSOR FUNCTIONS

The following tensor functions implement basic tensor operations and iterate over the multi-index space of tensor types using multi-dimensional iterators combining multiple tensor elements. The user is not forced to use the aforementioned multi-dimensional iterator class templates. Yet the multi-dimensional iterator should be able to iterate over a specific mode and must provide `begin` and `end` member functions that can generate multi-dimensional iterators with the same capabilities. Most of the following tensor functions require input iterator attributes of the standard library.

Similar to the basic linear algebra subroutines (BLAS), we distinguish between first-level and higher-level tensor algorithms. The former generalize function templates of the C++ standard library for tensor types and have identical function names with almost the same function signature. They combine elements of one or more tensor or subtensor instances with the same multi-index and are often referred to as pointwise or elementwise tensor operations. Higher-level tensor operations have a more complex control-flow and tensor elements with different multi-indices such as the tensor-tensor multiplication.

All of the following C++ tensor functions implement tensor operations with multiple loops and contain two optimizations that have been suggested in [24] optimizing index computation (minimum-index) and inlining recursive function by compile-time optimization (inline). Comparing the tree-recursive and equivalent iteration-based implementations that have presented in [24], we favor the tree-recursion which has fewer lines of C++ code, is easier to understand and is only about 8% slower if the leading dimension of the tensors or subtensors is greater than or equal to 256.

5.1. First-Level Tensor Operations

The following proposed first-level tensor C++ function templates are akin to the ones provided by the algorithms library of the C++ standard library and combine elements with the same multi-index. With similar functions signatures, tensor functions pose different iterator requirements and has in most cases tensor order as an additional parameter. Almost all C++ tensor functions contain a function object (predicate) that is applied to every input element. The user can utilize existing function objects of the C++ standard library, define its own class or use lambda-expressions which is why first-level C++ tensor functions can be regarded as higher-order functions for tensors.

It should be noted that dense and contiguously stored tensors, C++ functions from the standard library such `std::transform` or `std::inner_product` can be used. However, the usage of loops utilizing a single-index or alike in case of subtensors slows down the performance by a factor which is proportional to the subtensor order [24]. If the leading dimension n_{π_1} of a tensor is large enough and greater than 512, the experiments in [24] show that the control- and data-flow overhead of a multi-loop approach only slows down the computation by at most 12%. In extreme cases where the leading dimension is smaller than 64, we observed a slow down of about 50%. This observation favors the usage of one implementation with nested recursion and multiple loops for dense tensors and their subtensors if the leading dimensions are in most cases greater than 256.

The implementation of basic tensor functions can be derived from the previous example in listing 1. In contrast to the C++ algorithms, first-level tensor function templates iterate

Listing 2 | Implementation of `for_each` with multi-dimensional iterators.

```
template <class InputIt, class UnaryFn>
void for_each(unsigned r,
              InputIt first, InputIt last, UnaryFn fn)
{
  const auto s=r-1;

  if(r > 1)
   for(; first != last; ++first)
     for_each(s, first.begin(s), first.end(s), fn);

  else /* base case: r = 1 */
   std::for_each(first,last,fn);
}
```

Listing 3 | Implementation of `transform` with multi-dimensional iterators.

```
template <unsigned r,
          class InputIt, class OutputIt, class UnaryOp>
void transform(InputIt  fin,  InputIt lin,
               OutputIt fout, UnaryUp op)
{
  constexpr auto s=r-1;

  if constexpr (r > 1)
    for(; fin!=lin; ++fin, ++fout)
      transform<s>(fin.begin (s), fin.end(s),
                   fout.begin(s), op);

  else /* base case: r = 1 */
    std::transform (fin, lin, fout, op);
}
```

over multiple ranges using multi-dimensional iterators. The function `for_each` in listing 2 applies the function object `fn` of type `UnaryFn` to every tensor element that is accessed by multi-dimensional iterator pairs `first` and `last`. Given a tensor or subtensor `A` of order `p` with `p>0`, `for_each` in listing 2 needs to be performed with an iterator pair `A.begin(p)` and `A.end(p)`. The parameter `r` corresponds to the inverse recursion depth which is initialized with the tensor order `p` and decremented until the base case of the recursion is reached where `r` is equal to `1`. `for_each` calls itself `std::distance(first,last)` times in line `6` with a new range defined by `first.begin(r-1)` and `first.end(r-1)` where `first` is an iterator instance of the previous function call. When the base case with `r=1` is reached, `std::for_each` is called in line `7` with the range specified by `first.begin(1)` , `first.end(1)`. If `for_each` is called with an `r` smaller than `p`, `for_each` skips `p-r` modes and only applies `fn` on the first `r` modes. If `r` is greater than `p`, any memory access is likely to cause a segmentation fault. If the user calls `for_each` with `r=0`, `std::for_each` is directly called and iterated along the single index space of the tensor or subtensor.

Note that `for_each` calls itself $n_2 \cdots n_p$ times if the tensor or subtensor is of order $p > 1$ and has the dimensions $n_1, n_2, \ldots, n_p$. Given a tensor or subtensor `A` of order p with any linear storage format and a unary function object `fn`, the arguments of `for_each` should be `p`, `A.begin(p)`, `A.end(p)`, and `fn`. For instance, adding a scalar `v` to all elements of `A` can be performed if `fn` is defined as `std::bind(std::plus<>{},_1,v)` or using a lambda function with the same computation.

```
//A:=A+v;
for_each(p, A.begin(p), A.end(p), [v](auto &a){a+=v;});
```

The user can implement elementwise subtraction, multiplication, division operations by defining a binary function object from the standard library such as `std::bind(std::multiplies<>{},_1,v)`. It is also possible to define bitwise tensor operations, e.g., `std::bind(std::bit_or<>{},_1,v)` if `v` satisfies the template parameter requirements of the binary operation. The user can conveniently create complex elementwise tensor operations that contain a sequence of scalar operations for each element. For instance, raising all tensor or subtensor elements to the power of 2, dividing the result by `v` and adding the value `w` is given by the following expression.

```
//A:=A.^2/v+w;
for_each(p, A.begin(p), A.end(p),
                        [v,w](auto &a){a*=a/v+w;});
```

In contrast to calling simple overloaded operators of tensor or subtensor types, this statement does not create temporary tensor objects and is as efficient as expression templates.

Function `transform`, presented in listing 3, has a signature which is similar to the one of `std::transform`. It operates on two multi-dimensional ranges which are defined by the iterators `fin`, `lin` of type `InputIt` and `fout` of type `OuputIt` defining the input and output ranges, respectively. Akin to the `for_each` implementation, the one-dimensional ranges are given by iterators that are instantiated either by the previous recursive call or when `transform` is initially called. For demonstration purposes, the inverse recursion depth and its initial value is specified using a non-type template parameter `r`. The `if` condition is modified with the `constexpr` specifier so that `r>1` is evaluated at compile time. A C++ compiler can decide to inline the recursive calls which leads faster runtimes in case of small dimensions [24]. Once the base case with `r=1` is reached `std::transform` performs the unary operation `op` on elements of tensor fibers that are given by the ranges `[fin,lin)` and `[fout, fout+std::distance(fin,lin))`.

Given `p+1`-dimensional tensors or subtensors `A` and `C` with the shape tuple `n` and any linear storage format. Let also `op` be a unary operation of type `UnaryOp`. The multiplication of a scalar `v` with the elements of `A` is accomplished by calling `transform` as follows.

```
// C:= A+v;
transform<p>(A.begin(p), A.end(p), C.begin(p),
                        [v](auto a){ return a*v;});
```

Given `p+1`-dimensional tensors or subtensors `A`, `B`, and `C` with the shape tuple `n` and any linear storage format. Let also `op` be a binary operation of type `BinaryOp` that can process elements of `A` and `B`. Elementwise addition of `A` and `B` can be performed by calling `transform` as follows.

```
// C:= A+B;
transform<p>(A.begin(p), A.end(p),
             B.begin(p), C.begin(p), std::plus<>{});
```

Users can implement their own multi-dimensional iterators supporting the input iterator type traits with the `begin` and `end` method for initializing iterators. The `copy` and `transform` functions have the same signature except the unary operator which can be left out in case of `copy`. Moreover, `copy` can be regarded as a specialization of `transform` where the unary function `op` returns a single element that is provided by the input iterator. With `r` specifying the inverse recursion depth, our implementation of `copy` is given by the following function call.

```
transform<r>(fin, lin, fout, [](auto a){return a;}));
```

Transposing a tensor can be accomplished using the `copy` function with minor modifications. Let `tau` of type, e.g., `std::array<unsigned,p>` be an additional standard container for the index permutation as a function parameter and let the function name `copy` be changed to `transpose`. An out-of-place tensor-transposition is performed with

```
// C := A^{tau};
transpose<p>(A.begin(tau[p-1]), A.end(tau[p-1]),
             C.begin(p), tau);
```

The recursive function call in `transpose` needs to be changed accordingly, replacing the argument `p` with `r-1`. Note this simple implementation of the tensor transposition does not conserve data locality only for both tensors unless the permutation tuple is trivial. A high-performance version of the transposition operation is given in [31].

An implementation of the inner product of two tensors or subtensors with any linear storage format is given in listing 4. The function signature and body corresponds to a modified `transform` function. The `std::inner_product` computes the inner product of tensor or subtensor fibers multiple times using results `init` of previous function calls. Computing the inner product of two tensors or subtensors `A` and `B` is given by the following function call.

```
// c := <A,B>;
auto inner = inner_product<p>(A.begin(p), A.end(p),
                              B.begin(p), Value{});
```

The initial value is given by the default constructor of `Value` which should be implicitly convertible to the elements type of `A` and `B`. The frobenius norm of a tensor `A` can be implemented using the `inner_product` as follows.

```
// c: = fnorm(A) = sqrt(inner(A,A));
auto c = std::sqrt(inner_product<p>(A.begin(p),A.end(p),
                                    A.begin(p),Value{}));
```

The computation of the frobenius norm is given by first executing the unary operation `[](auto const& a){return a*a;}` with `transform` and accumulate all elements of the output tensor `C` using the `accumulate` function.

5.2. Higher-Level Tensor Operations

Higher-level tensor operations perform one or more inner products over specified dimensions and, therefore, exhibit a higher arithmetic intensity ratio compared to first-level tensor operations. Prominent examples are the general tensor-times-tensor multiplication with variations.

Listing 4 | Implementation of `inner_product` with multi-dimensional iterators.

```
template <unsigned r, class InputIt,
          class OutputIt, class Value>
Value inner_product(InputIt  fin,  InputIt lin,
                    OutputIt fout, Value init)
{
  constexpr auto s=r-1;

  if constexpr (r > 1)
    for(; fin!=lin; ++fin, ++fout)
      init = inner_product<s>(fin.begin(s),fin.end(s),
                              fout.begin(s),init);

  else /* base case: r = 1 */
    init = std::inner_product(fin, lin, fout, init);

  return init;
}
```

5.2.1. Tensor-Vector Multiplication

One such variation is the q-mode tensor-vector multiplication where q equals the contraction dimension. Let $\underline{\mathbf{A}}$ be a tensor or subtensor of order $p > 1$ with dimensions $\mathbf{n}$ and any linear storage format. Let $\mathbf{b}$ be a vector with dimension n_q with $1 \leq q \leq p$. Let $\underline{\mathbf{C}}$ be a tensor or subtensor of order $p-1$ with dimensions $\mathbf{n}' = (n_1, \ldots, n_{q-1}, n_{q+1}, \ldots, n_p)$. The q-mode tensor-vector multiplication computes $1/n_q \prod_{r=1}^{p} n_r$ inner products, i.e., fiber-vector multiplications, according to

$$\underline{\mathbf{C}}(i_1, \ldots, i_{q-1}, i_{q+1}, \ldots, i_p) = \sum_{i_q=1}^{n_q} \underline{\mathbf{A}}(i_1, \ldots, i_q, \ldots, i_p) \cdot \mathbf{b}(i_q) \quad (1)$$

with $1 \leq i_r \leq n_r$. If $p = 2$, the tensor-vector multiplication computes a vector-matrix product of the form $\mathbf{c} = \mathbf{b}^T \cdot \mathbf{A}$ for $q = 1$ and a matrix-vector product of the form $\mathbf{c} = \mathbf{A} \cdot \mathbf{b}$ for $q = 2$. Vector $\mathbf{b}$ is multiplied with the frontal slices of $\underline{\mathbf{A}}$ if p is greater than 2 and $q = 1$ or $q = 2$.

Function `ttv` in listing 5 implements the general tensor-times-vector multiplication where the contracting dimension `q` is a one-based compile time parameter computing all contractions for $1 < q \leq p$. The second template parameter `ra` corresponds to the inverse recursion depth and ranges from $1 \leq r \leq p$. The third template parameter `rc` depends on `q` so that $r_c = r_a - 1$ for $q < r_a \leq p$ and $r_c = r_a$ for $1 \leq r_a \leq q$. The algorithm used in `ttv` is based on the algorithm 1 that has been proposed in [32]. The implementation can be regarded as an extension of the previously discussed functions with similar signature and body. The first `if`-statement is introduced to skip and place the iteration along the `q`-th dimension inside the base case. Therefore, iterators for the next recursion are generated for `A` based on the current position of `fa`. The second `if`-statement contains the recursive call that can be found in all previous listings. The `else`-statement contains the base case of the recursion which is executed if `ra=1`. The base case multiplies vector `b` with a selected slice of `A` and stores the

results in the corresponding fiber of C. Given a tensor A of order p, a vector b and a tensor C of order p-1, all with the same element type and storage format, then

```
// C = A *q b
ttv<q,p,p-1>(A.begin(p), b.begin(),C.begin(p-1));
```

computes the q-mode tensor-times-vector product for $1 < q \leq p$. Note that spatial data locality for A is maximized when stride w_q^a satisfies $w_q^a \leq w_{r_a}^a$ for all $r_a \neq q$ which is the case for a storage format with a layout tuple $(q, \pi_2, \ldots, \pi_p)$. For that purpose, C stride w_1^c needs to satisfy $w_1^c \leq w_{r_c}^c$ for all $r_c \neq q$. Assuming that only one storage format, the spatial data locality can be increased for any linear storage format by modifying the recursion order according to the storage format and reordering the loops in the base case as suggested in [32]. This is accomplished by using the layout vectors $\boldsymbol{\pi}$ of A and C that contain indices with $w_{\pi_r} \leq w_{\pi_{r+1}}$ for all $1 \leq r < p$. Replacing indices ra and rc with pia[ra-2] and pic[rc-2] allows to generate iterators with strides that are decreasing with the recursion depth. The base case needs to be changed as well with the following code snippet that computes a slice-vector product accessing A and C for any linear storage format.

```
auto ta = pia[0];
auto tc = pic[0];

for(auto faq=fa.begin(q); faq!=fa.end(q); ++faq,++fb){
  auto op = [b=*fb](auto const& a, auto const& c)
                                    {return c+a*b;});

  std::transform(faq.begin(ta), faq.end(ta),
                fc. begin(tc), fc. begin(tc), op);
}
```

Instead using std::inner_product, the base case scales A's fibers with b and writes the result in C's corresponding fibers. If A and B are contiguously stored, memory access can be performed in a coalesced manner. The algorithm can be further optimized for temporal data locality and parallel execution. Interested readers are referred to [32].

5.2.2. Tensor-Matrix Multiplication

A generalization of the q-mode tensor-vector multiplication and a specialization of the tensor-tensor multiplication is the q-mode tensor-matrix multiplication. Let $\underline{\mathbf{A}}$ be a tensor or subtensor of order $p > 1$ with dimensions $\mathbf{n}$ and any linear storage format. Let $\mathbf{B}$ be a matrix with dimensions (n_q, n_q') with $1 \leq q \leq p$. Let $\underline{\mathbf{C}}$ be a tensor or subtensor of order p with dimensions $\mathbf{n}' = (n_1, \ldots, m, \ldots, n_p)$. The q-mode tensor-matrix multiplication computes $(m/n_q) \prod_{r=1}^{p} n_r$ inner products, i.e., fiber-vector multiplications, according to

$$\underline{\mathbf{C}}(i_1, \ldots, j, \ldots, i_p) = \sum_{i_q=1}^{n_q} \underline{\mathbf{A}}(i_1, \ldots, i_q, \ldots, i_p) \cdot \mathbf{B}(j, i_q) \qquad (2)$$

with $1 \leq i_r \leq n_r$ and $1 \leq j \leq m$. If $p = 2$, a matrix-matrix product $\mathbf{C} = \mathbf{B} \cdot \mathbf{A}$ for $q = 1$ and $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}^T$ for $q = 2$, respectively. Matrix $\mathbf{B}$ is multiplied with the frontal slices of $\underline{\mathbf{A}}$ accordingly if p greater than 2 and $q = 1$ or $q = 2$.

Listing 5 | Implementation of the q-mode tensor-vector product with iterators for q>1.

```
template<unsigned q, unsigned ra, unsigned rc,
         class InputIt1,class InputIt2,class OutputIt>
void ttv(InputIt1 fa, InputIt1 la, InputIt2 fb,
         OutputIt fc)
{
  constexpr auto sa = ra-1;
  constexpr auto sc = rc-1;

  if constexpr (ra == q)
    ttv<q,sa,rc>(fa.begin(sa), fa.end(sa), fb, fc);

  else if constexpr (ra > 1)
    for(; fa != la; ++fa, ++fc)
      ttv<q,sa,sc>(fa.begin(sa),fa.end(sa),
                   fb,fc.begin(sc));

  else /* base case: ra = 1 and rc = 1 */
    for(; fa != la; ++fa, ++fc)
      *fc = std::inner_product(fa.begin(q),fa.end(q),
                               fb ,*fc);
}
```

The implementation of the q-mode tensor-matrix multiplication is almost identical to ttv except for the base case, minor modifications for the recursion cases and the function signature.

```
template<unsigned q, unsigned r,
         class InputIt1,class InputIt2,class OutputIt>
void ttm(InputIt1 fa, InputIt1 la, InputIt2 fb,
         OutputIt fc)
```

The contracting dimension q is a one-based compile time parameter of ttm which performs a valid computation for $1 < q \leq p$. As both tensors or subtensors have the same order, ttm requires only one template parameter r which equates to ra in ttv. The implementation of ttm's base-case computes a matrix-slice product of the form $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}^T$ by multiplying a two-dimensional slice of A with a transposed B and storing the results in corresponding fibers of C. The base case is presented in the following code section and executed when r=1.

```
for(auto fb1=fb; fa!=la; ++fa, ++fc,fb0 = fb)
  for(auto fcq=fc.begin(q);fcq!=fc.end(q);++fcq,++fb1)
    *fcq = std::inner_product(fa.begin(q),fa.end(q),
                              fb1.begin(2),*fcq);
```

When r=1, iterators fa, la, and fc have been instantiated by previously generated iterators with their begin and end methods for r=1. We postulate that fb is initialized with begin for the first dimension with r=1. The first for-loop iterates over the first mode of A and C using fa and fc. The second for-loop iterates over mode q of C with the starting address of the previous iterator and first mode of B and calling std::inner_product with A's fiber and one column of B. Given a tensor or subtensor A of order p, a matrix B and a tensor or subtensor C of order p, with similar element types and any linear data layout, then

```
// C = A *q B;
ttm<q,p> (A.begin(p), B.begin(1), C.begin(p));
```

```
auto fb2 = fb.begin(2);
auto pi0 = pi[0];

for(auto fcq=fc.begin(q);fcq!=fc.end(q);++fcq,++fb){
  auto fb1 = fb.begin(1);
  for(auto faq=fa.begin(q);faq!=fa.end(q);++faq,++fb1){
    auto op=[b=*fb1](auto const& a, auto const& c)
                                        {return c+a*b;});
    std::transform(faq.begin(pi0),faq.end(pi0),
                   fcq.begin(pi0),fcq.begin(pi0), op);
  }
}
```

computes the q-mode tensor-times-matrix product. Note that spatial data locality for A and C is high when their strides w_q satisfy $w_q \leq w_r$ for all $r \neq q$. Assuming that only one storage format, the spatial data locality can be increased for any linear storage format similar to ttv. This is done by utilizing the layout vectors $\boldsymbol{\pi}$ of both tensors and by replacing the index r with pi[r-2] that allows to generate iterators with decreasing strides and recursion depth. The loop ordering inside the base case of ttm is changed from (n_1,n_q,m) to (m,n_q,n_{π_1}). In that case A and C are accessed in a coalesced manner for any linear storage format if the tensors are contiguously stored in memory where one fiber of C is accessed n_q times. The algorithm can be further optimized for temporal data locality and parallel execution.

5.2.3. Tensor-Tensor Multiplication

The tensor-tensor product is the general form of the tensor-matrix and tensor-vector multiplication. Let $\underline{\mathbf{A}}$ and $\underline{\mathbf{B}}$ be tensors or subtensors of order p_a and p_b with dimensions $\mathbf{n}_a$ and $\mathbf{n}_b$, respectively. Given two permutation tuples $\boldsymbol{\varphi}$ and $\boldsymbol{\psi}$ of length p_a and p_b and the number of contractions q with $q_a = p_a - q$ and $q_b = p_b - q$, the q-fold tensor-tensor multiplication computes elements of tensor or subtensor $\underline{\mathbf{C}}$ of order $p_c = q_a + q_b$ with dimensions $\mathbf{n}_c$ and using permutation tuples $\boldsymbol{\varphi}$ and $\boldsymbol{\psi}$ according to

$$\underline{\mathbf{C}}(\mathbf{i}_c) = \sum_{j_1=1}^{m_1} \cdots \sum_{j_q=1}^{m_q} \underline{\mathbf{A}}(\mathbf{i}_a) \cdot \underline{\mathbf{B}}(\mathbf{i}_b), \tag{3}$$

where the shape tuples satisfy $n^c_{r_c} = n^a_{r_a}$ for $1 \leq r_c \leq q_a$ with $r_a = \varphi_r$, $n^c_{r_c} = n^b_{r_b}$ for $1 \leq r \leq q_b$ with $r_c = q_b + r$ and $r_b = \psi_r$, $m_r = n^a_{r_a} = n^b_{r_b}$ for $1 \leq r \leq q$ with $r_a = \varphi_{r+q_a}$ and $r_b = \psi_{r+q_b}$. The first q elements of $\boldsymbol{\varphi}$ and $\boldsymbol{\psi}$ specify the contraction modes, while the remaining q_a and q_b elements specify the free (non-contraction) modes. The k-mode tensor-matrix and k-mode tensor-vector multiplication are specializations of the q-fold tensor-tensor multiplication which corresponds to the k-mode tensor-vector multiplication, if $q = 1$, $p_a > 1$, $p_b = 1$ and $\boldsymbol{\varphi} = (1, \ldots, k-1, k+1, \ldots, p_a, k)$, $\boldsymbol{\psi} = (1)$. The k-mode tensor-matrix multiplication is given if $q = 1$, $p_a > 1$, $p_b = 2$ and $\boldsymbol{\varphi} = (1, \ldots, k-1, k+1, \ldots, p_a, k)$, $\boldsymbol{\psi} = (1, 2)$.

Function ttt in listing 6 implements the tensor-times-tensor multiplication as defined in Equation (3) for any number of contractions q>1. The contraction is performed with tensors or subtensors A and B of order pa and pb with any linear storage format and without unfolding A or B. The free and contraction modes reside within the permutation tuple phi and psi that must be a container with random access capabilities. Function ttt is defined with four non-type template parameter. The first three ra, rb, and rc are the current modes of each corresponding tensor or subtensor and should be initially instantiated with pa and pb and pc, respectively. The last non-type parameter q of ttt and equals to the number of contraction modes.

The control flow of ttt contains four main branches of which three contain a for-loop with a recursive function call. The first for-loop is exectued q_b times and iterates over free index spaces of B and C with $s = \psi_{r_b}$ for $q < r_b \leq p_b$ and $q_a < r_c \leq p_c$ without adjusting iterators of A. The second for-loop is executed q_a times and iterates over free index spaces of A and C where $s = \varphi_{r_a}$ for $q < r_a \leq p_a$ and $1 \leq r_c \leq q_a$ without adjusting iterators of B. The third for-loop is executed q times and iterates over the contraction index spaces of A and B where $s = \varphi_{r_a}$ and $r = \psi_{r_b}$ for $1 < r_{a,b} \leq q$ without adjusting iterators of C. If $r_a = 1$ and $r_b = 1$ the base case is reached and ttt performs an inner product with iterators that have been previously instantiated.

The q-mode tensor-tensor multiplication can be interpreted as a mix of the inner and outer tensor product with permutation tuples. The latter is partly accomplished by the $q_a + q_b$-fold execution with the first and second for-loop. However, input tensor elements of A and B are not multiplied to complete the outer product operation. Instead an inner product over q modes is computed for the recursion levels $r > q_a + q_b$. The last two branches could be replaced by the inner_product in listing 4 using the permutation tuples phi and psi. The minimum recursion depth is 1 when $q = 1$ and $q_{a,b} = 0$, while the maximum recursion depth equals $q + q_a + q_b$ with $q > 0$ and $q_{a,b} > 0$.

Given tensors or subtensors A of order 3, B of order 4 and C of order 3 with similar element types, any linear data layout. Let the dimension tuples of A and B be $\mathbf{n}_a = (4, 3, 2)$ and $\mathbf{n}_b = (5, 4, 6, 3)$, respectively. Let also $q = 2$ be the number of contractions and $\boldsymbol{\varphi} = (1, 2, 3)$ and $\boldsymbol{\psi} = (2, 4, 1, 2)$ be the elements of the permutation tuples phi and psi, respectively. Given the dimensions (n^a_3, n^b_1, n^b_2), i.e., (2,5,6), then

```
// C = A(_i,_j,_)*B(_,_i,_,_j)
ttt<pa,pb,pc,q>(phi,psi,
                A.begin(pa),B.begin(pb),C.begin(pc));
```

performs a 2-mode tensor-tensor multiplication of A and B according to phi, psi, and q. Spatial data locality for A and B is high when for $q > 0$ their strides $w^a_{\varphi_1}$ and $w^b_{\psi_1}$ satisfy $w^a_{\varphi_1} \leq w^a_r$ for all $r \neq \varphi_1$ and $w^b_{\psi_1} \leq w^b_r$ for all $r \neq \psi_1$, respectively. Performance analysis and optimization techniques for the general tensor-tensor multiplication are discussed in [33, 34].

6. RUNTIME ANALYSIS

This section presents runtime results of the transform function (listing 3) and the function inner_product (listing 4). The runtime measurements also include pointer-based implementations that have been presented in [24]. We

Listing 6 | Template Function `ttt` using multi-dimensional iterators implementing Equation (3).

```
template<unsigned ra, unsigned rb,
         unsigned rc, unsigned q,
         class InputIt1, class InputIt2,
         class OutputIt, class Permutation>

void ttt(Permutation const& phi,
         Permutation const& psi,
         InputIt1 fa, InputIt2 la,
         InputIt2 fb, InputIt2 lb,
         OutputIt fc)
{
  constexpr auto sa = ra-1;
  constexpr auto sb = rb-1,
  constexpr auto sc = rc-1;

  if constexpr (rb > q)
    for(; fb!=lb; ++fb,++fc)
      ttt<ra,sb,sc,q>(phi,psi,
                      fa,la,
                      fb.begin(sb),fb.end(sb),
                      fc.begin(sc));

  else if constexpr (ra > q)
    for(auto s=phi[sa]; fa!=la; ++fa,++fc)
      ttt<sa,rb,sc,q>(phi,psi,
                      fa.begin(s),fa.end(s),
                      fb,lb,fc.begin(sc));

  else if constexpr (ra > 1)
    for(auto s=phi[sa], r=psi[sb]; fa!=la; ++fa,++fb)
      ttt<sa,sb,rc,q>(phi,psi,
                      fa.begin(s),fa.end(s),
                      fb.begin(r),lb,
                      fc);

  else // base case: ra=1 and rb=1
    *fc = std::inner_product(fa,la,fb,*fc);
}
```

have also included runtime results of the `ttv` function (listing 5) that has been discussed in [32] as a sequential implementation for the tensor-times-vector multiplication. All pointer and iterator-based functions have identical with respect to their control-flow in which the recursion index is a template parameter.

6.1. Setup

The following runtime measurements have been performed with 1792 differently shaped tensors ranging from 32 to 1024 MiB for single- and 64 to 2048 MiB for double-precision floating-point numbers. The order of the tensors ranges from 2 to 14 while dimensions range from 256 to 32768. Dimension tuples are arranged within multiple two-dimensional arrays so that runtime data could be visualized as three-dimensional surfaces or contour plots in terms of the tensor order and tensor size. The contour plots consist of 100 height levels that correspond to averaged throughputs. We will refer to the contour plots as throughput maps. Spatial data locality is always preserved meaning that relative memory indices are generated according to storage format. Tensor elements are stored according to the first-order storage format. This setup is identical to the tensor test set that has been presented in [24]. For the tensor-times-vector multiplication, we have used a setup that is akin to the one described in [32]. All tensors are asymmetrically shaped ranging from 64 to 2048 MiB for single- and 128 to 4096 MiB for double-precision floating-point numbers. The tensor order ranges from 2 to 10 and the contraction mode has been set to 1 in order to preserve spatial data locality for all tensor objects.

The experiments have been carried out on a Core i9-7900X Intel Xeon processor with 10 cores and 20 hardware threads running at 3.3 GHz. It has a theoretical peak memory bandwidth of 85.312 GB/s resulting from four 64-bit wide channels with a data rate of 2666MT/s with a peak memory bandwidth of 21.328 GB/s. The sizes of the L3 cache and each L2 cache are 14MB and 1024KB. The source code has been compiled with GCC v9.3 using the highest optimization level `-Ofast` and `-march=native`.The benchmark results of each function are the average of 10 runs on a single core.

6.2. Results

Figure 1 contains two throughput maps of a pointer- and iterator-based `transform` function. Both implement an elementwise tensor addition of the form `C:=A+v;` using unary function object `[v](auto a){return a+v;}`. The throughput of `transform` with pointers and iterators are most effected when the tensor size smaller than 128. We assume that this is caused by the caching mechanism which is still able to hold some data inside the last level cache and to speed up the computation. This effect diminishes when the tensor size is greater than 256 MiB. The throughput also contains a slight variation for different tensor order. For tensor sizes greater than 256 MiB, pointer-based implementation of `transform` computes the tensor addition with approximately 12.2 GB/s varying with at most 10% from the mean value. The iterator-based implementation is more consistent and only slows down to approximately 12.2 GB/s if the tensor order is 4 and 5. The `std::transform` function of the C++ standard library, the pointer-based and iterator-based `transform` function reach a median throughput of 13.71, 12.01, and 12.60 GB/s for 95% of test cases and a maximum throughput of 15.57, 13.50, and 13.71 GB/s.

The runtime behavior of the `inner_product` implementations is similar, see **Figure 2**. The `std::inner_product` function of the C++ standard library, the pointer-based and iterator-based `inner_product` function reach a median throughput of 14.8, 12.03, and 12.47 GB/s for 95% of test cases. They exhibit maximum throughput of 15.36, 12.59, and 12.82 GB/s mostly when the tensor size is equal to 32 MiB. We have made similar runtime observations for other elementwise tensor operations such as `for_each` where the iterator-based implementation is in many cases 1 to 5% faster than their corresponding pointer-based counterparts.

Similar results are obtained for iterator-based and pointer-based implementations of the tensor-times-vector operations where both C++ functions compute the tensor-vector-product with 2.09 (single-precision) GFLOPS for about 95% test-cases. This can be observed in **Figure 3** which contains throughput

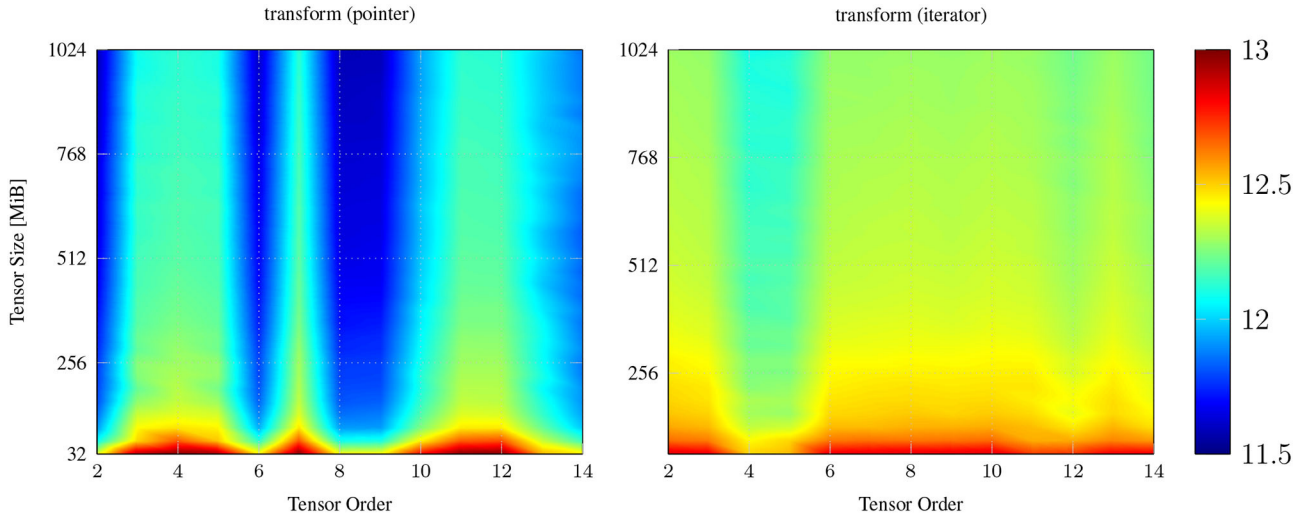


FIGURE 1 | Single core throughput in GB/s of the `transform` function with pointers (left) that has been discussed in [24] and iterators (right) that is presented in listing 3. Iterator-based `transform` function reaches a median throughput of 12.60 GB/s and is about 2% to 5% faster than its pointer-based counterpart.

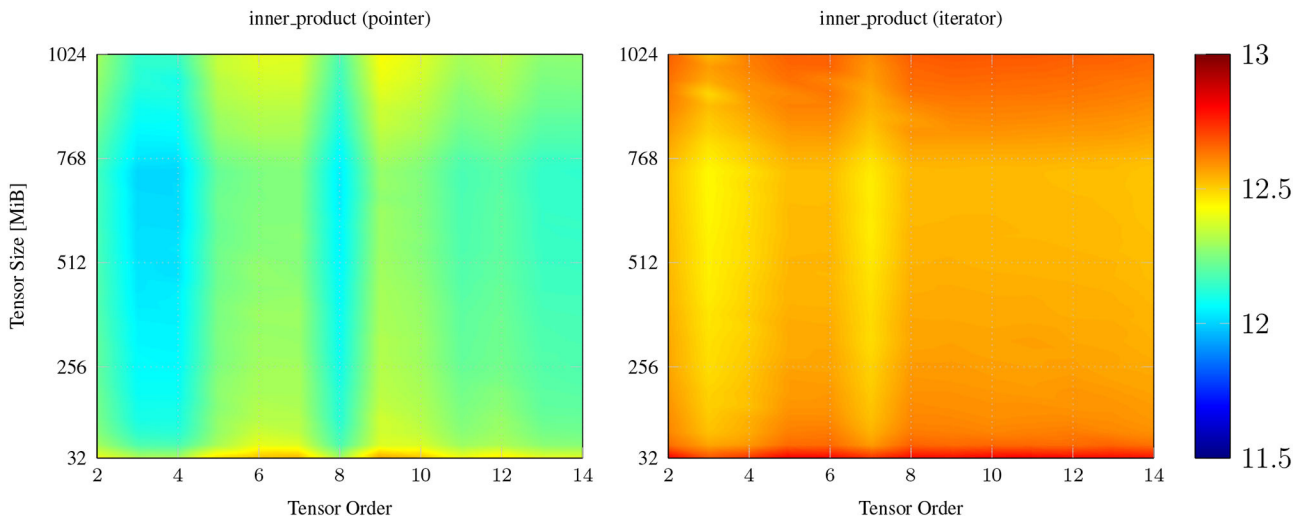


FIGURE 2 | Single core throughput in GB/s of the `inner_product` function with pointers (left) that has been discussed in [24] and with iterators (right) that is presented in listing 4. Iterator-based `inner_product` function reaches a median throughput of 12.47 GB/s and is about 2 to 5% faster than its pointer-based counterpart.

maps for the iterator-based and pointer-based implementation of the tensor-times-vector operation. The iterator-based function `ttv` in listing 5 reaches a peak throughput of 2.92 GFLOPS when tensor size and order are around 64 MiB and 10, respectively. The pointer-based counterpart exhibits a maximum throughput of 2.74 GFLOPS with the same tensor dimensions and is about 6.5% slower than the iterator-based function. Those performance peaks happen for larger tensor order when the first (contraction) dimension of the input tensor is relatively small. This results in a higher reuse of cache lines that belong to the input vector and output tensor fiber.

7. CONCLUSIONS

We have presented generic C++ functions for basic tensor operations that have been discussed in [22] as part of a Matlab toolbox for numeric tensor computations. Following design pattern of the Standard Template Library, all proposed C++ functions are defined in terms of only multi-dimensional iterators and avoid complex pointer arithmetic. The set of the C++ functions includes elementwise tensor operations and more complex tensor operations such as tensor-tensor multiplication. All C++ functions perform the corresponding

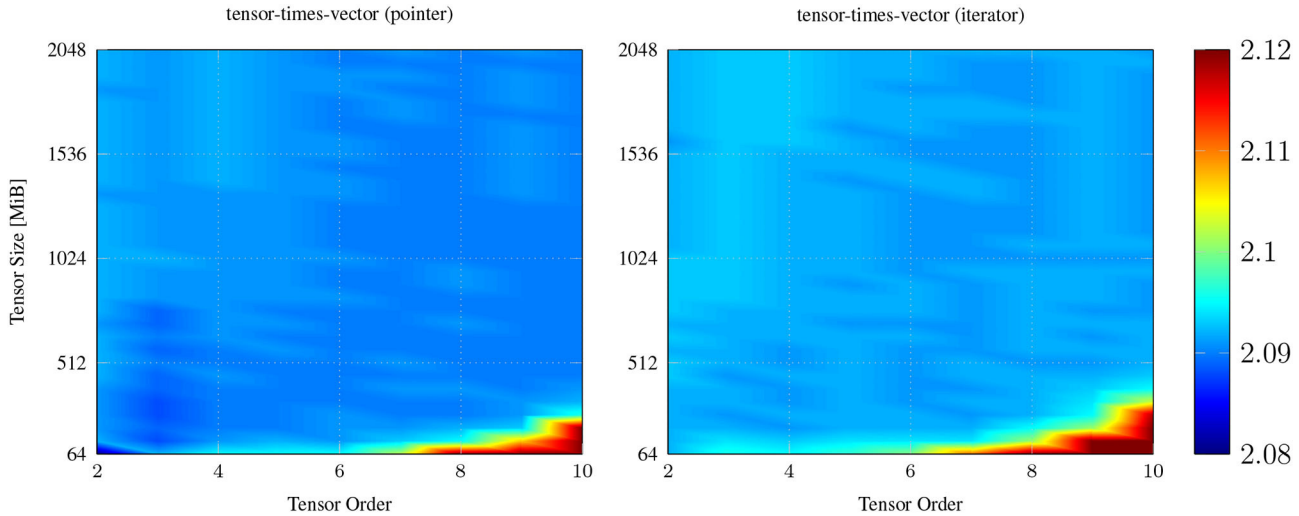


FIGURE 3 | Single core throughput in GFLOPS of the `ttv` function with pointers (left) and with iterators (right) that is presented in listing 5. Iterator-based `ttv` function computes the tensor-vector product with median throughput of 2.09 GFLOPS and performs as fast as the pointer-based implementation.

computation in-place and in a recursive fashion using two optimizations that have been discussed in [24]. We have introduced a multi-dimensional iterator that can be instantiated by Boost's uBlas tensor and subtensor types. Other C++ frameworks can utilize the proposed C++ functions for any linear storage format by implementing the proposed or their own multi-dimensional iterator fulfilling a minimal set of iterator requirements. Our performance measurements show that the iterator-based functions compute elementwise tensor operations and the tensor-times-vector product at least as fast as their corresponding pointer-based counterparts. Our iterator-based design method is applicable to other tensor operations such as the metricized-tensor times Khatri-Rao product (MTTKRP) which is used to decompose tensors according to the PARAFAC model [35, 36]. This implies that multi-dimensional iterators can be used for efficiently implementing tensor operations. In future, we intend to design C++ concepts for multi-dimensional iterator or ranges. We also would like to integrate optimization techniques that have been discussed in [32, 33] and to enable parallel execution of different type of tensor operations.

DATA AVAILABILITY STATEMENT

The raw data supporting the conclusions of this article will be made available by the authors, without undue reservation.

AUTHOR CONTRIBUTIONS

The author confirms being the sole contributor of this work and has approved it for publication.

REFERENCES

1. Savas B, Eldén L. Handwritten digit classification using higher order singular value decomposition. *Pattern Recognit.* (2007) 40:993–1003. doi: 10.1016/j.patcog.2006.08.004
2. Vasilescu MAO, Terzopoulos D. Multilinear image analysis for facial recognition. In: *Proceedings of the 16th International Conference on Pattern Recognition.* Vol. 2 Quebec City, QC (2002). p. 511–514.
3. Suter SK, Makhynia M, Pajarola R. TAMRESH - tensor approximation multiresolution hierarchy for interactive volume visualization. In: *Proceedings of the 15th Eurographics Conference on Visualization. EuroVis '13.* Chichester (2013). p. 151–60.
4. Kolda TG, Sun J. Scalable tensor decompositions for multi-aspect data mining. In: *Proceedings of the 8th IEEE International Conference on Data Mining.* (Pisa) 2008. p. 363–72.
5. Rendle S, Balby Marinho L, Nanopoulos A, Schmidt-Thieme L. Learning optimal ranking with tensor factorization for tag recommendation. In: *Proceedings of the International Conference on Knowledge Discovery and Data Mining.* Paris (2009). p. 727–36.
6. Khoromskij B. Tensors-structured numerical methods in scientific computing: survey on recent advances. *Chemometr. Intell. Lab. Syst.* (2012) 110:1–19. doi: 10.1016/J.CHEMOLAB.2011.09.001
7. Kolda TG, Bader BW. Tensor decompositions and applications. *SIAM Rev.* (2009) 51, 455–500. doi: 10.1137/07070111X
8. Lim LH. Tensors and hypermatrices. In: Hogben L, editor. *Handbook of Linear Algebra, 2nd Edn.* Chapman and Hall (2017).
9. Cichocki A, Zdunek R, H PA, Amari S. *Nonnegative Matrix and Tensor Factorizations, 1st Edn.* John Wiley & Sons, (2009).
10. da Silva JD, Machado A. Multilinear algebra. In: L. Hogben, editor. *Handbook of Linear Algebra, 2nd Edn.* Chapman and Hall, (2017).
11. Lee N, Cichocki A. Fundamental tensor operations for large-scale data analysis using tensor network formats. *Multidimensional Syst Signal Process.* (2018) 29:921–60. doi: 10.1007/s11045-017-0481-0
12. Lathauwer LD, Moor BD, Vandewalle J. A multilinear singular value decomposition. *SIAM J Matrix Anal Appl.* (2000) 21:1253–78. doi: 10.1137/S0895479896305696

13. Li J, Battaglino C, Perros I, Sun J, Vuduc R. An input-adaptive and in-place approach to dense tensor-times-matrix multiply. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC '15.* Austin, TX (2015). p. 1–12.
14. Stroustrup B. Foundations of C++. In: *Programming Languages and Systems - 21st European Symposium on Programming. Vol. 7211 of Lecture Notes in Computer Science.* Tallinn (2012). p. 1–25.
15. Stroustrup B. Software development for infrastructure. *Computer.* (2012) 45:47–58.
16. Veldhuizen TL. Arrays in Blitz++. In: Caromel D, Oldehoeft RR, Tholburn M, editors. *Lecture Notes in Computer Science. ISCOPE.* Vol. 1505. Berlin: Springer (1998). p. 223–30.
17. Reynders III, JV, Cummings JC. The POOMA framework. *Comput Phys.* (1998) 12:453–59.
18. Landry W. Implementing a high performance tensor library. *Sci Program.* (2003) 11:273–90.
19. Solomonik E, Matthews D, Hammond J, Demmel J. Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions. In: *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing. IPDPS '13.* Cambridge, MA (2013). p. 813–24.
20. Harrison AP, Joseph D. Numeric tensor framework: exploiting and extending Einstein notation. *J Comput Sci.* (2016) 16:128–39. doi: 10.1016/j.jocs.2016.05.004
21. Poya R, Gil AJ, Ortigosa R. A high performance data parallel tensor contraction framework: Application to coupled electro-mechanics. *Comput Phys Commun.* (2017) 216:35–52. doi: 10.1016/j.cpc.2017.02.016
22. Bader BW, Kolda TG. Algorithm 862: MATLAB tensor classes for fast algorithm prototyping. *ACM Trans Math Softw.* (2006) 32:635–53. doi: 10.1145/1186785.1186794
23. Psarras C, Karlsson L, Bientinesi P. The landscape of software for tensor computations. *CoRR.* 2021;abs/2103.13756.
24. Bassoy C, Schatz V. Fast higher-order functions for tensor calculus with tensors and subtensors. In: Shi Y, Fu H, Tian Y, Krzhizhanovskaya VV, Lees MH, Dongarra J, et al., editors. *Computational Science—ICCS 2018.* Springer International Publishing (2018). p. 639–52.
25. Garcia R, Lumsdaine A. MultiArray: a C++ library for generic programming with arrays. *Softw Pract Exp.* (2005) 35:159–88. doi: 10.1002/spe.630
26. Aragón AM. A C++ 11 implementation of arbitrary-rank tensors for high-performance computing. *Comput Phys Commun.* (2014) 185:1681–96. doi: 10.1016/j.cpc.2014.01.005
27. Stepanov A. The standard template library. *Byte.* (1995) 20:177–8.
28. Hackbusch W. Numerical tensor calculus. *Acta Numerica.* (2014) 23:651–742. doi: 10.1017/S0962492914000087
29. Chatterjee S, Lebeck AR, Patnala PK, Thottethodi M. Recursive array layouts and fast parallel matrix multiplication. In: *Proceedings of the Eleventh Annual ACM symposium on Parallel algorithms and architectures. SPAA '99.* New York, NY (1999). p. 222–31.
30. Elmroth E, Gustavson F, Jonsson I, Kågström B. Recursive blocked algorithms and hybrid data structures for dense matrix library software. *SIAM Rev.* (2004) 46:3–45. doi: 10.1137/S0036144503428693
31. Springer P, Su T, Bientinesi P. HPTT: a high-performance tensor transposition C++ library. In: *Proceedings of the 4th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming.* Barcelona (2017). p. 56–62.
32. Bassoy C. Design of a high-performance tensor-vector multiplication with BLAS. In: Rodrigues JMF, Cardoso PJS, Monteiro JM, Lam R, Krzhizhanovskaya VV, Lees MH, et al., editors. *Computational Science – ICCS 2019 Lecture Notes in Computer Science.* Vol. 11536. Cham: Springer. (2019). p. 32–45.
33. Springer P, Bientinesi P. Design of a high-performance GEMM-like tensor-tensor multiplication. *ACM Trans Math Softw.* (2018) 44:1–29. doi: 10.1145/3157733
34. Matthews DA. High-performance tensor contraction without transposition. *SIAM J Sci Comput.* (2018) 40:C1–C24. doi: 10.1137/16M108968X
35. Ballard G, Knight N, Rouse K. Communication lower bounds for matricized tensor times Khatri-Rao product. In: *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS).* Vancouver, BC: IEEE (2018). p. 557–67.
36. Bader BW, Kolda TG. Efficient MATLAB computations with sparse and factored tensors. *SIAM J Sci Comput.* (2008) 30:205–31. doi: 10.1137/060676489

Conflict of Interest: The author declares that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

frontiers | Frontiers in Applied Mathematics and Statistics

ORIGINAL RESEARCH
published: 12 April 2022
doi: 10.3389/fams.2022.830270

Accelerating Jackknife Resampling for the Canonical Polyadic Decomposition

Christos Psarras[1], *Lars Karlsson*[2], *Rasmus Bro*[3] *and Paolo Bientinesi*[2]*

[1] International Research Training Group, Aachen Institute for Advanced Study in Computational Engineering Science (AICES), Department of Computer Science, RWTH Aachen University, Aachen, Germany, [2] High-Performance and Automatic Computing Group, Department of Computing Science, Umeå University, Umeå, Sweden, [3] Department of Food Science, Institute for Fødevarevidenskab, University of Copenhagen, Copenhagen, Denmark

The Canonical Polyadic (CP) tensor decomposition is frequently used as a model in applications in a variety of different fields. Using jackknife resampling to estimate parameter uncertainties is often desirable but results in an increase of the already high computational cost. Upon observation that the resampled tensors, though different, are nearly identical, we show that it is possible to extend the recently proposed Concurrent ALS (CALS) technique to a jackknife resampling scenario. This extension gives access to the computational efficiency advantage of CALS for the price of a modest increase (typically a few percent) in the number of floating point operations. Numerical experiments on both synthetic and real-world datasets demonstrate that the new workflow based on a CALS extension can be several times faster than a straightforward workflow where the jackknife submodels are processed individually.

Keywords: jackknife, Tensors, decomposition, CP, ALS, Canonical Polyadic Decomposition, Alternating Least Squares

OPEN ACCESS

Edited by:
Stefan Kunis,
Osnabrück University, Germany

Reviewed by:
Markus Wageringel,
Osnabrück University, Germany
Paul Breiding,
Max-Planck-Institute for Mathematics in the Sciences, Germany

***Correspondence:**
Paolo Bientinesi
pauldj@cs.umu.se

Specialty section:
This article was submitted to Mathematics of Computation and Data Science, a section of the journal Frontiers in Applied Mathematics and Statistics

Received: *06 December 2021*
Accepted: *22 February 2022*
Published: *12 April 2022*

Citation:
Psarras C, Karlsson L, Bro R and Bientinesi P (2022) Accelerating Jackknife Resampling for the Canonical Polyadic Decomposition. Front. Appl. Math. Stat. 8:830270. doi: 10.3389/fams.2022.830270

1. INTRODUCTION

The CP model is used increasingly across a large diversity of fields. One of the fields in which CP is commonly applied is chemistry [1, 2], where there is often a need for estimating not only the parameters of the model, but also the associated uncertainty of those parameters [3]. In fact, in some areas it is a dogma that an estimate without an uncertainty is not a result. A common approach for estimating uncertainties of the parameters of CP models is through resampling, such as bootstrapping or jackknifing [4, 5]. The latter has added benefits, e.g., for variable selection [6] and outlier detection [4]. Here we consider a new technique, JK-CALS, that increases the performance of jackknife resampling applied to CP by more efficiently utilizing the computer's memory hierarchy.

The basic concept of jackknife is somewhat related to cross-validation. Let $\mathcal{T} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ be a tensor, and $\mathbf{U}_1, \ldots, \mathbf{U}_N$ the factor matrices of a CP model. Let us also make the assumption (typical in many applications) that the first mode corresponds to independent samples, and all the other modes correspond to variables. For the most basic type of jackknifing, namely

Leave-One-Out (LOO)[1], one sample (out of I_1) is left out at a time (resulting in a tensor with only $I_1 - 1$ samples) and a model is fitted to the remaining data; we refer to that model as a *submodel*. All samples are left out exactly once, resulting in I_1 distinct submodels. Each submodel provides an estimate of the parameters of the overall model. For example, each submodel provides an estimate of the factor (or loading) matrix $\mathbf{U}_2$. From these I_1 estimates it is possible to calculate the variance (or bias) of the overall loading matrix (the one obtained from all samples). One complication comes from some indeterminacies with CP that need to be taken into account. For example, when one (or more) samples are removed from the initial tensor, the order of components in the submodel may change; this phenomenon is explained and a solution is proposed in Riu and Bro [4].

Recently, the authors proposed a technique, Concurrent ALS (CALS) [7], that can fit *multiple* CP models to the *same* underlying tensor more rapidly than regular ALS. CALS achieves better performance not by altering the numerics but by utilizing the computer's memory hierarchy more efficiently than regular ALS. However, the CALS technique cannot be directly applied to jackknife resampling, since the I_1 submodels are fitted to *different* tensors. In this paper, we extend the idea that underpins CALS to jackknife resampling. The new technique takes advantage of the fact that the I_1 resampled tensors are *nearly* identical. At the price of a modest increase in arithmetic operations, the technique allows for more efficient fitting of the CP submodels and thus improved overall performance of a jackknife workflow. In applications in which the number of components in the CP model is relatively low, the technique can significantly reduce the overall time to solution.

Contributions

- An efficient technique, JK-CALS, for performing jackknife resampling of CP models. The technique is based on an extension of CALS to *nearly* identical tensors. To the best of our knowledge, this is the first attempt at accelerating jackknife resampling of CP models.
- Numerical experiments demonstrate that JK-CALS can lead to performance gains in a jackknife resampling workflow.
- Theoretical analysis shows that the technique generalizes from leave-one-out to grouped jackknife with a modest (less than a factor of two) increase in arithmetic.
- A C++ library with support for GPU acceleration and a Matlab interface.

Organization

The rest of the paper is organized as follows. In Section 2, we provide an overview of related research. In Section 3, we review the standard CP-ALS and CALS algorithms, as well as jackknife applied to CP. We describe the technique which enables us to use CALS to compute jackknife more efficiently in Section 4. In Section 5 we demonstrate the efficiency of our proposed technique, by applying it to perform jackknife resampling to CP models that have been fitted to artificial and real tensors. Finally, in Section 6, we conclude the paper and provide insights for further research.

[1]Henceforth, when we mention jackknifing we imply LOO jackknifing, unless otherwise stated.

2. RELATED WORK

Two popular techniques for uncertainty estimation for CP models are bootstrap and jackknife [4, 5, 8]. The main difference is that jackknife resamples *without* replacement whereas bootstrap resamples *with* replacement. Bootstrap frequently involves more submodels than jackknife and is therefore more expensive. The term jackknife typically refers to leave-one-out jackknife, where only one observation is removed when resampling. More than one observation can be removed at a time, leading to the variations called delete-*d* jackknife [9] and grouped jackknife [10, p. 7] (also known as Delete-A-Group jackknife [11] or DAGJK). Of the two, grouped jackknife is most often used for CP model uncertainty estimation, primarily due to the significantly smaller number of samples generated. When applied to CP, jackknife has certain benefits over bootstrap, e.g., for variable selection [6] and outlier detection [4].

Jackknife requires fitting multiple submodels. A straightforward way of accelerating jackknife is to separately accelerate the fitting of each submodel, e.g., using a faster implementation. The simplest and most extensively used numerical method for fitting CP models is the Alternating Least Squares (CP-ALS) method. Alternative methods for fitting CP models include eigendecomposition-based methods [12] and gradient-based (all-at-once) optimization methods [13].

Several techniques have been proposed to accelerate CP-ALS. Line search [14] and extrapolation [15] aim to reduce the number of iterations until convergence. Randomization-based techniques have also been proposed. These target very large tensors, and either randomly sample the tensor [16] or the Khatri-Rao product [17], to reduce their size and, by extension, the overall amount of computation. Similarly, compression-based techniques replace the target tensor with a compressed version, thus also reducing the amount of computation during fitting [18]. The CP model of the reduced tensor is inflated to correspond to a model of the original tensor.

Several projects offer high-performance implementations of CP-ALS, for example, Cyclops [19], PLANC [20], Partensor [21], SPLATT [22], and Genten [23]. For a more comprehensive list of software implementing some variant of CP-ALS, refer to Psarras et al. [24].

Similar to the present work, there have been attempts at accelerating jackknife although (to the best of our knowledge) not in the context of CP. In Buzas [25], the high computational cost of jackknife is tackled by using a numerical approximation that requires fewer operations at the price of lower accuracy. In Belotti and Peracchi [26], a general-purpose routine for fast jackknife estimation is presented. Some estimators (often linear ones) have leave-one-out formulas that allow for fast computation of the estimator after leaving one sample out. Jackknife is thus accelerated by computing the estimator on the full set and then

systematically applying the leave-one-out formula. In Hinkle and Stromberg [27], a similar technique is studied. Jackknife computes an estimator on s distinct subsets of the s samples. Any two of these subsets differ by only one sample, i.e., any one subset can be obtained from any other by replacing one and only one element. Some estimators have a fast updating formula, which can rapidly transform an estimator for one subset to an estimator for another subset. Jackknife is thus accelerated by computing the estimator from scratch on the first subset and then repeatedly updating the estimator using this fast updating formula.

3. CP-ALS, CALS AND JACKKNIFE

In this section, we first specify the notation to be used throughout the paper, we then review the CP-ALS and CALS techniques, and finally we describe jackknife resampling applied to CP.

3.1. Notation

For vectors and matrices, we use bold lowercase and uppercase roman letters, respectively, e.g., $\mathbf{v}$ and $\mathbf{U}$. For tensors, we follow the notation in Kolda and Bader [28]; specifically, we use bold calligraphic fonts, e.g., $\boldsymbol{\mathcal{T}}$. The order (number of indices or modes) of a tensor is denoted by uppercase roman letters, e.g., N. For each mode $n \in \{1, 2, \ldots, N\}$, a tensor $\boldsymbol{\mathcal{T}}$ can be unfolded (matricized) into a matrix, denoted by $\mathbf{T}_{(n)}$, where the columns are the mode-n fibers of $\boldsymbol{\mathcal{T}}$, i.e., the vectors obtained by fixing all indices except for mode n. Sets are denoted by calligraphic fonts, e.g., $\mathcal{S}$. Given two matrices $\mathbf{A}$ and $\mathbf{B}$ with the same number of columns, the Khatri-Rao product, denoted by $\mathbf{A} \odot \mathbf{B}$, is the column-wise Kronecker product of $\mathbf{A}$ and $\mathbf{B}$. Finally, the unary operator $\oplus$, when applied to a matrix, denotes the scalar which is the sum of all matrix elements.

3.2. CP-ALS

The standard alternating least-squares method for CP is shown in Algorithm 1 (CP-ALS). The input consists of a target tensor $\boldsymbol{\mathcal{T}}$. The output consists of a CP model represented by a sequence of factor matrices $\mathbf{U}_1, \ldots, \mathbf{U}_N$. The algorithm repeatedly updates the factor matrices one by one in sequence until either of the following criteria are met: a) the fit of the model to the target tensor falls below a certain tolerance threshold, or b) a maximum number of iterations has been reached. To update a specific factor matrix $\mathbf{U}_n$, the gradient of the least-squares objective function with respect to that factor matrix is set to zero and the resulting linear least-squares problem is solved directly from the normal equations. This entails computing the Matricized Tensor Times Khatri-Rao Product (MTTKRP) (line 4), which is the product between the mode-n unfolding $\mathbf{T}_{(n)}$ and the Khatri-Rao Product (KRP) of all factor matrices except $\mathbf{U}_n$. The MTTKRP is followed by the Hadamard product of the Gramians ($\mathbf{U}_i{}^T\mathbf{U}_i$) of each factor matrix in line 5. Factor matrix $\mathbf{U}_n$ is updated by solving the linear system in line 6. At the completion of an iteration, i.e., a full pass over all N modes, the error between the model and the target tensor is computed (line 8) using the efficient formula derived in Phan et al. [29].

Algorithm 1: CP-ALS: Alternating least squares method for CP decomposition.

Input: $\boldsymbol{\mathcal{T}} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ The target tensor
Output: $\mathbf{U}_1, \ldots, \mathbf{U}_N$ The fitted factor matrices

1 Initialize the factor matrices $\mathbf{U}_1, \ldots, \mathbf{U}_N$
2 **repeat**
3 **for** $n = 1, 2, \ldots, N$ **do**
4 $\mathbf{M}_n \leftarrow \mathbf{T}_{(n)}(\odot_{i \neq n}\mathbf{U}_i)$ MTTKRP
5 $\mathbf{H}_n \leftarrow *_{i \neq n}(\mathbf{U}_i{}^T\mathbf{U}_i)$ Hadamard product of Gramians
6 $\mathbf{U}_n \leftarrow \mathbf{M}_n\mathbf{H}_n{}^\dagger$ $\mathbf{H}_n{}^\dagger$: pseudoinverse of $\mathbf{H}_n$
7 **end**
8 $e \leftarrow ||\boldsymbol{\mathcal{T}}||^2 - (\oplus(\mathbf{H}_N * (\mathbf{U}_N{}^T\mathbf{U}_N))) - 2(\oplus(\mathbf{U}_N * \mathbf{M}_N))$ Error calculation
9 **until** *convergence detected or maximum number of iterations reached*

Assuming a small number of components (R), the most expensive step is the MTTKRP (line 4). This step involves $2R\prod_i I_i$ FLOPs (ignoring, for the sake of simplicity, the lower order of FLOPs required for the computation of the KRP). The operation touches slightly more than $\prod_i I_i$ memory locations, resulting in an arithmetic intensity less than $2R$ FLOPs per memory reference. Thus, unless R is sufficiently large, the speed of the computation will be limited by the memory bandwidth rather than the speed of the processor. The CP-ALS algorithm is inherently memory-bound for small R, regardless of how it is implemented.

The impact on performance of the memory-bound nature of MTTKRP is demonstrated in **Figure 1**, which shows the computational efficiency of a particular implementation of MTTKRP as a function of the number of components (for a tensor of size $50 \times 200 \times 200$). Efficiency is defined as the ratio of the performance achieved by MTTKRP (in FLOPs/sec), relative to the Theoretical Peak Performance (TPP, see below) of the machine, i.e.,

$$\text{EFFICIENCY} = \frac{\text{PERFORMANCE}}{\text{TPP}} = \frac{\#\text{FLOPS/TIME}}{\text{TPP}}.$$

The TPP of a machine is defined as the maximum number of (double precision) floating point operations the machine can perform in 1 s. **Table 1** shows the TPP for our particular machine (see Section 5 for details). In **Figure 1**, we see that the efficiency of MTTKRP tends to increase with the number of components, R, until eventually reaching a plateau. On this machine, the plateau is $R \geq 60$ at $\approx$ 70% efficiency for one thread and $R \geq 300$ at $\approx$ 35% efficiency for 24 threads. For $R \leq 20$, which is common in applications, the efficiency is well below the TPP.

3.3. Concurrent ALS (CALS)

When fitting *multiple* CP models to the *same* underlying tensor, the Concurrent ALS (CALS) technique can improve the efficiency if the number of components is not large enough for CP-ALS to

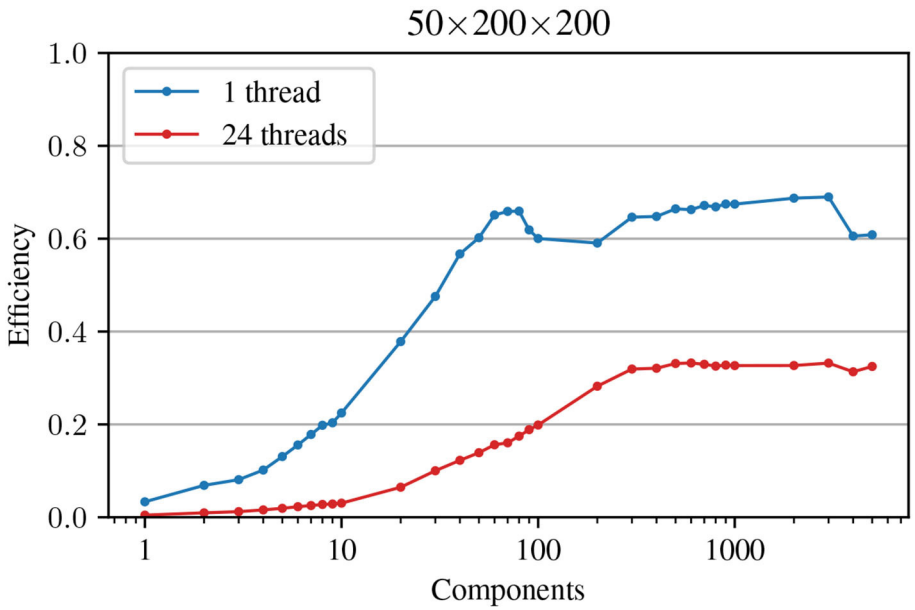


FIGURE 1 | Efficiency of MTTKRP on a 50 × 200 × 200 tensor for an increasing number of components. Note that in the multi-threaded execution, the theoretical peak performance increases while the total number of operations to be performed stays the same (as in the single-threaded case); this explains the drop in efficiency per thread.

TABLE 1 | Theoretical peak performance (TPP) for a particular machine.

System	TPP (GFlops/sec)	Threads	Frequency per core (Ghz)
CPU	112	1	3.5
	1,536	24	2

Due to the decrease in the peak frequency per core when all 24 cores are used, the TPP for 24 cores is less than 24× the TPP for 1 core.

reach its performance plateau [7]. A need to fit multiple models to the same tensor arises, for example, when trying different initial guesses or when trying different numbers of components.

The gist of CALS can be summarized as follows (see Psarras et al. [7] for details). Suppose K independent instances of CP-ALS have to be executed on the same underlying tensor. Rather than running them sequentially or in parallel, run them in lock-step fashion as follows. Advance every CP-ALS process one iteration before proceeding to the next iteration. One CALS iteration entails K CP-ALS iterations (one iteration per model). Each CP-ALS iteration in turn contains one MTTKRP operation, so one CALS iteration also entails K MTTKRP operations. But these MTTKRPs all involve the same tensor and can therefore be fused into one bigger MTTKRP operation (see Equation 3 of Psarras et al. [7]). The performance of the fused MTTKRP depends on the sum total of components, i.e., $\sum_{i=1}^{K} R_i$, where R_i is the number of components in model i. Due to the performance profile of MTTKRP (see **Figure 1**), the fused MTTKRP is expected to be more efficient than each of the K smaller operations it replaces.

The following example illustrates the impact on efficiency of MTTKRP fusion. Given a target tensor of size 50 × 200 × 200, $K = 50$ models to fit, and $R_i = 5$ components in each model, the efficiency for each of the K MTTKRPs in CP-ALS is about 15% (3%) for 1 (24) threads (see **Figure 1**). The efficiency of the fused MTTKRP in CALS will be as observed for $R = \sum_{i=1}^{K} R_i = 250$, i.e., 60% (30%) for 1 (24) threads. Since the MTTKRP operation dominates the cost, CALS is expected to be $\approx 4\times$ ($\approx 10\times$) faster than CP-ALS for 1 (24) threads.

3.4. Jackknife

Algorithm 2 shows a baseline (inefficient) application of leave-one-out jackknife resampling to a CP model. For details, see Riu and Bro [4]. The inputs are a target tensor $\mathcal{T}$, an overall CP model P fitted to all of $\mathcal{T}$, and a sampled mode $\hat{n} \in \{1, 2, \ldots, N\}$. For each sample $p \in \{1, 2, \ldots, I_{\hat{n}}\}$, the algorihm removes the slice corresponding to the sample from tensor $\mathcal{T}$ (line 3) and model P (line 4) and fits a reduced model P_{-p} (lines 4–6) to the reduced tensor $\hat{\mathcal{T}}$ using regular CP-ALS. After fitting all submodels, the standard deviation of every factor matrix except $\mathbf{U}_{\hat{n}}$ is computed from the $I_{\hat{n}}$ submodels in $\mathcal{Q}$ (line 10). The only costly part of Algorithm 2 is the repeated calls to CP-ALS in line 5.

4. ACCELERATING JACKKNIFE BY USING CALS

The straightforward application of jackknife to CP in Algorithm 2 involves $I_{\hat{n}}$ independent calls to CP-ALS on *nearly* the same tensor. Since the tensors are not exactly the same, CALS [7] cannot be directly applied. In this section, we show how one can rewrite Algorithm 2 in such a way that CALS *can* be applied. There is an associated overhead due to extra computation, but we

Algorithm 2: JK-ALS: An algorithm that performs (LOO) jackknife resampling on a CP model.

Input: $\mathcal{T} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ The target tensor
$P = \mathbf{U}_1, \ldots, \mathbf{U}_N$ A CP model fitted to $\mathcal{T}$
$\hat{n}$ The sampled mode
Output: $\mathbf{S}_1, \ldots, \mathbf{S}_N$ Uncertainty of each element of each factor matrix of P

$\mathcal{Q} \leftarrow \emptyset$ Set containing fitted jackknife models
for $p \in \{1, 2, \ldots, I_{\hat{n}}\}$ **do** For every index p in mode $\hat{n}$
  $\mathcal{T}_{-p} \leftarrow$ remove the slice with index p in mode $\hat{n}$ from tensor $\mathcal{T}$
  $P_{-p} \leftarrow$ remove row p from factor matrix $\mathbf{U}_{\hat{n}}$ of P
  $\hat{P}_{-p} \leftarrow$ cp_als$(\mathcal{T}_{-p}, P_{-p})$
  $\hat{P}_{-p} \leftarrow$ permutation and scale adjustment of $\hat{P}_{-p}$
  $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{\hat{P}_{-p}\}$
end
for $n \in \{1, 2, \ldots, N\} \setminus \{\hat{n}\}$ **do** For every mode n except $\hat{n}$
  $\mathbf{S}_n \leftarrow$ standard deviation of factor matrix $\mathbf{U}_n$ in $\mathcal{Q}$
end

will show that the overhead is modest (less than a 100% increase and typically only a few percent increase).

4.1. JK-CALS: Jackknife Extension of CALS

Let $\mathcal{T}$ be an N-mode tensor with a corresponding CP model $\mathbf{A}_1, \ldots, \mathbf{A}_N$. Let $\hat{\mathcal{T}}$ be identical to $\mathcal{T}$ except for one sample (with index p) removed from the sampled mode $\hat{n} \in \{1, 2, \ldots, N\}$. Let $\hat{\mathbf{A}}_1, \ldots, \hat{\mathbf{A}}_N$ be the CP submodel corresponding to the resampled tensor $\mathcal{T}$.

When fitting a CP model to $\mathcal{T}$ using CP-ALS, the MTTKRP for mode n is given by

$$\mathbf{M}_n \leftarrow \mathbf{T}_{(n)}(\mathbf{A}_N \odot \cdots \odot \mathbf{A}_{n+1} \odot \mathbf{A}_{n-1} \odot \cdots \odot \mathbf{A}_1). \quad (1)$$

Similarly, when fitting a model to $\hat{\mathcal{T}}$, the MTTKRP for mode n is given by

$$\hat{\mathbf{M}}_n \leftarrow \hat{\mathbf{T}}_{(n)}(\hat{\mathbf{A}}_N \odot \cdots \odot \hat{\mathbf{A}}_{n+1} \odot \hat{\mathbf{A}}_{n-1} \odot \cdots \odot \hat{\mathbf{A}}_1). \quad (2)$$

Can $\hat{\mathbf{M}}_n$ be computed from $\mathbf{T}_{(n)}$ instead of $\hat{\mathbf{T}}_{(n)}$? As we will see, the answer is yes. We separate two cases: $n = \hat{n}$ and $n \neq \hat{n}$.

Case I: $n = \hat{n}$**.** The slice of $\mathcal{T}$ removed when resampling corresponds to a *row* of the unfolding $\mathbf{T}_{(n)} = \mathbf{T}_{(\hat{n})}$. To see this, note that element $\mathcal{T}(i_1, i_2, \ldots, i_N)$ corresponds to element $\mathbf{T}_{(n)}(i_n, j)$ of its mode-n unfolding [28], where

$$j = 1 + \sum_{\substack{k=1 \\ k \neq n}}^{N} (i_k - 1) \prod_{\substack{m=1 \\ m \neq n}}^{k-1} I_m. \quad (3)$$

Algorithm 3: JK-CALS: Concurrent alternating least squares method for jackknife estimation.

Input: $\mathcal{T} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ The target tensor
$P = \mathbf{U}_1, \ldots, \mathbf{U}_N$ A CP model fitted to $\mathcal{T}$
$\hat{n}$ The sampled mode
Output: $\mathbf{U}_1^{(p)}, \ldots, \mathbf{U}_N^{(p)}$ for $p = 1, 2, \ldots, I_{\hat{n}}$ The fitted submodels

Initialize the submodels $\mathbf{U}_1^{(p)}, \ldots, \mathbf{U}_N^{(p)}$ for $p = 1, 2, \ldots, I_{\hat{n}}$
for $n = 1, 2, \ldots, N$ **do** Initialize one factor multi-matrix for each mode
  for $p = 1, 2, \ldots, I_{\hat{n}}$ **do**
    if $n = \hat{n}$ **then**
      $\overline{\mathbf{U}}_n^{(|p)} \leftarrow \mathbf{U}_n^{(p)}$ with a row of zeros inserted at index p
    else
      $\overline{\mathbf{U}}_n^{(|p)} \leftarrow \mathbf{U}_n^{(p)}$
    end
  end
end
repeat Concurrently run $I_{\hat{n}}$ instances of CP-ALS
  for $n = 1, 2, \ldots, N$ **do**
    $\overline{\mathbf{M}}_n \leftarrow \mathbf{T}_{(n)}(\odot_{i \neq n} \overline{\mathbf{U}}_i)$
    for $p = 1, 2, \ldots, I_{\hat{n}}$ **do**
      $\mathbf{H}_n^{(p)} \leftarrow *_{i \neq n}(\overline{\mathbf{U}}_i^{(|p)^T} \overline{\mathbf{U}}_i^{(|p)})$
      $\overline{\mathbf{U}}_n^{(|p)} \leftarrow \overline{\mathbf{M}}_n^{(|p)} \mathbf{H}_n^{(p)^\dagger}$
      if $n = \hat{n}$ **then**
        $\overline{\mathbf{U}}_n^{(|p)} \leftarrow$ zero out row p of $\overline{\mathbf{U}}_n^{(|p)}$
      end
    end
  end
  for $p = 1, 2, \ldots, I_{\hat{n}}$ **do**
    $e \leftarrow ||\mathcal{T}_{-p}||^2 - (\oplus(\mathbf{H}_N^{(p)} * (\overline{\mathbf{U}}_N^{(|p)^T} \overline{\mathbf{U}}_N^{(|p)}))) - 2(\oplus(\overline{\mathbf{U}}_N^{(|p)} * \overline{\mathbf{M}}_N^{(|p)}))$ Error calculation
  end
until *convergence detected for all instances or maximum number of iterations reached*

When we remove sample p, then $\hat{\mathbf{T}}_{(n)}$ will be identical to $\mathbf{T}_{(n)}$ except that row p from the latter is missing in the former. In other words, $\hat{\mathbf{T}}_{(n)} = \mathbf{E}_p \mathbf{T}_{(n)}$, where $\mathbf{E}_p$ is the matrix that removes row p. We can therefore compute $\hat{\mathbf{M}}_n$ by replacing $\hat{\mathbf{T}}_{(n)}$ with $\mathbf{T}_{(n)}$ in Equation (2) and then discarding row p from the result:

$$\hat{\mathbf{M}}_n \leftarrow \mathbf{E}_p(\mathbf{T}_{(n)}(\hat{\mathbf{A}}_N \odot \cdots \odot \hat{\mathbf{A}}_{n+1} \odot \hat{\mathbf{A}}_{n-1} \odot \cdots \odot \hat{\mathbf{A}}_1)).$$

Case II: $n \neq \hat{n}$**.** The slice of $\mathcal{T}$ removed when resampling corresponds to a *set of columns* in the unfolding $\mathbf{T}_{(n)}$. One could in principle remove these columns to obtain $\hat{\mathbf{T}}_{(n)}$. But instead of explicitly removing sample p from $\mathcal{T}$, we can simply zero out the corresponding slice of $\mathcal{T}$. To give the CP model matching

dimensions, we need only insert a row of zeros at index p in factor matrix $\hat{n}$. Crucially, the zeroing out of slice p is superfluous. In the MTTKRP, the elements that should have been zeroed out will be multiplied with zeros in the Khatri-Rao product generated by the row of zeros insert in factor matrix $\hat{n}$. Thus, to compute $\hat{\mathbf{M}}_n$ in Equation (2) we (a) replace $\hat{\mathbf{T}}_{(n)}$ with $\mathbf{T}_{(n)}$ and (b) insert a row of zeros at index p in factor matrix $\hat{\mathbf{A}}_{\hat{n}}$.

In summary, we have shown that it is possible to compute $\hat{\mathbf{M}}_n$ in Equation (2) without referencing the reduced tensor. There is an overhead associate with extra arithmetic. For the case $n = \hat{n}$, we compute numbers that are later discarded. For the case $n \neq \hat{n}$, we do some arithmetic with zeros.

4.1.1. The JK-CALS Algorithm

Based on the observations above, the CALS algorithm [7] can be modified to facilitate the concurrent fitting of all jackknife submodels. Algorithm 3 incorporates the necessary changes (colored red). The inputs are a target tensor $\boldsymbol{\mathcal{T}}$, and the sampled mode $\hat{n}$. The algorithm starts by initializing $I_{\hat{n}}$ submodels P_p for $p = 1, 2, \ldots, I_{\hat{n}}$ in line 1; each submodel P_p is created by removing row p from factor matrix $\mathbf{U}_{\hat{n}}$ of model P. As described in Psarras et al. [7], CALS creates a *multi-matrix n* for each mode n by horizontally concatenating the factor matrices of each submodel P_p in lines 2–10; the superscript $|p$ denotes the position in n where the factor matrix $\mathbf{U}_n^{(p)}$ is copied. In the case of JK-CALS, instead of just copying each factor matrix into its corresponding multi-matrix, the algorithm first checks whether zero padding is required (lines 5–7). The loop in line 11 performs ALS for all submodels concurrently. Specifically, in line 13 the MTTKRP (n) is computed for all models at the same time by using the multi-matrices *i*. Then, lines 15 and 16 are the same as lines 5 and 6 of Algorithm 1; each submodel is treated separately by reading its corresponding values within n and n (indicated by the superscript $|p$). In JK-CALS, when $n = \hat{n}$, the padded row is reset to 0 after $\overline{\mathbf{U}}_n^{(|p)}$ is updated (line 18). Finally, after a full ALS cycle has completed, the error of each model is calculated in line 23. In JK-CALS, the error formula is adjusted for each submodel by considering the L2 norm of its corresponding subtensor $\boldsymbol{\mathcal{T}}_{-p}$.

We remark that JK-CALS can be straightforwardly extended to grouped jackknife [10, p. 7], in which the samples are split into groups of d elements ($I_{\hat{n}}/d$ groups) and jackknife submodels are created by removing an entire group at a time. Instead of padding and periodically zeroing out one row, we pad and periodically zero out d rows.

4.2. Performance Considerations

While Algorithm 3 benefits from improved MTTKRP efficiency, the padding results in extra arithmetic operations. Let d denote the number of removed samples ($d = 1$ corresponds to leave-one-out). For the sake of simplicity, assume that the integer d divides $I_{\hat{n}}$. In grouped jackknife there are $I_{\hat{n}}/d$ submodels, each with R components. The only costly part is the MTTKRP.

The MTTKRPs in JK-ALS (for all submodels combined) requires

$$\left(\frac{I_{\hat{n}}}{d}\right)\left(2R(I_{\hat{n}} - d)\prod_{\substack{i=1\\ i\neq\hat{n}}}^{N} I_i\right)$$

FLOPs. Meanwhile, the fused MTTKRP in JK-CALS requires

$$2\left(\frac{I_{\hat{n}}}{d}R\right)\prod_{i=1}^{N} I_i$$

FLOPs. The ratio of the latter to the former comes down to

$$\frac{I_{\hat{n}}}{I_{\hat{n}} - d} \leq 2,$$

since $d \leq I_{\hat{n}}/2$ in grouped jackknife. Thus, in the worst case, JK-CALS requires less than twice the FLOPs of JK-ALS. More typically, the overhead is negligible.

5. EXPERIMENTS

We investigate the performance benefits of the JK-CALS algorithm to perform jackknife resampling on a CP model through two sets of experiments. In the first set of experiments, we focus on the scalability of the algorithm, with respect to both problem size and number of processor cores. For this purpose, we use synthetic datasets of increasing volume, mimicking the shape of real datasets. In the second set of experiments, we illustrate JK-CALS's practical impact by using it to perform jackknife resampling on two tensors arising in actual applications.

All experiments were conducted using a Linux-based system with an Intel® Xeon® Platinum 8160 Processor (Turbo Boost enabled, Hyper-Threading disabled), which contains 24 physical cores split in 2 NUMA regions of 12 cores each. The system also contains an Nvidia Tesla V100 GPU[2]. The experiments were conducted with double precision arithmetic and we report results for 1 thread, 24 threads (two NUMA regions), and the GPU (with 24 CPU threads). The source code (available online[3]) was compiled using GCC[4] and linked to the Intel® Math Kernel Library[5].

5.1. Scalability Analysis

In this first experiment, we use three synthetic tensors of size $50 \times m \times m$ with $m \in \{100, 200, 400\}$, referred to as "small", "medium" and "large" tensors, respectively. The samples are in the first mode. The other modes contain variables. The number of samples is kept low, since leave-one-out jackknife is usually performed on a small number of samples (usually < 100), while there can be arbitrarily many variables.

[2]Driver version: 470.57.02, CUDA Version: 11.2.
[3]https://github.com/HPAC/CP-CALS/tree/jackknife
[4]GCC version 9.
[5]MKL version 19.0.

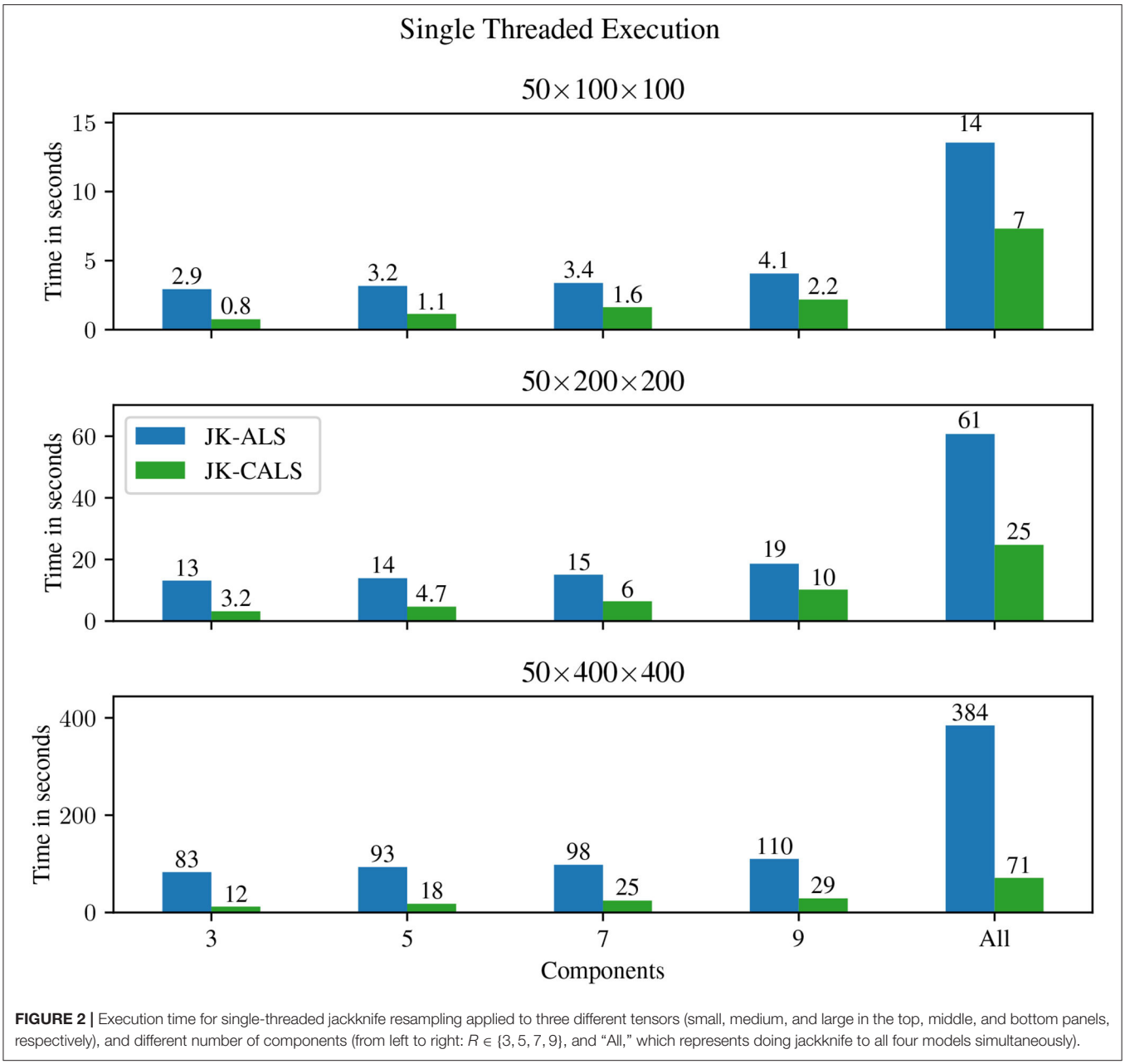


FIGURE 2 | Execution time for single-threaded jackknife resampling applied to three different tensors (small, medium, and large in the top, middle, and bottom panels, respectively), and different number of components (from left to right: $R \in \{3, 5, 7, 9\}$, and "All," which represents doing jackknife to all four models simultaneously).

For each tensor, we perform jackknife on four models with varying number of components ($R \in \{3, 5, 7, 9\}$). This range of component counts is typical in applications. In practice, it is often the case that multiple models are fitted to the target tensor, and many of those models are then further analyzed using jackknife. For this reason, we perform jackknife on each model individually, as well as to all models simultaneously (denoted by "All" in the figures), to better simulate multiple real-world application scenarios. In this experiment, the termination criteria based on maximum number of iterations and tolerance are ignored; instead, all models are forced to go through exactly 100 iterations, typically a small number of iterations for small values of tolerance (i.e., most models require more than 100 iterations). The reason for this choice is that we aim to isolate the performance difference of the methods tested; therefore, we maintain a consistent amount of workload throughout the experiment. (Tolerance and maximum number of iterations are instead used later on in the application experiments.)

For comparison, we perform jackknife using three methods: JK-ALS, JK-OALS and JK-CALS. JK-OALS uses OpenMP to take advantage of the inherent parallelism when fitting multiple submodels by parallelizing the loop in line 2 of Algorithm 2. Each thread maintains its own subsample $\boldsymbol{\mathcal{T}}_{-p}$ and P_{-p} of tensor $\boldsymbol{\mathcal{T}}$ and model P, respectively. This method is only used for multi-threaded and GPU experiments, and we are only going to focus on its performance, ignoring the memory overhead associated with it.

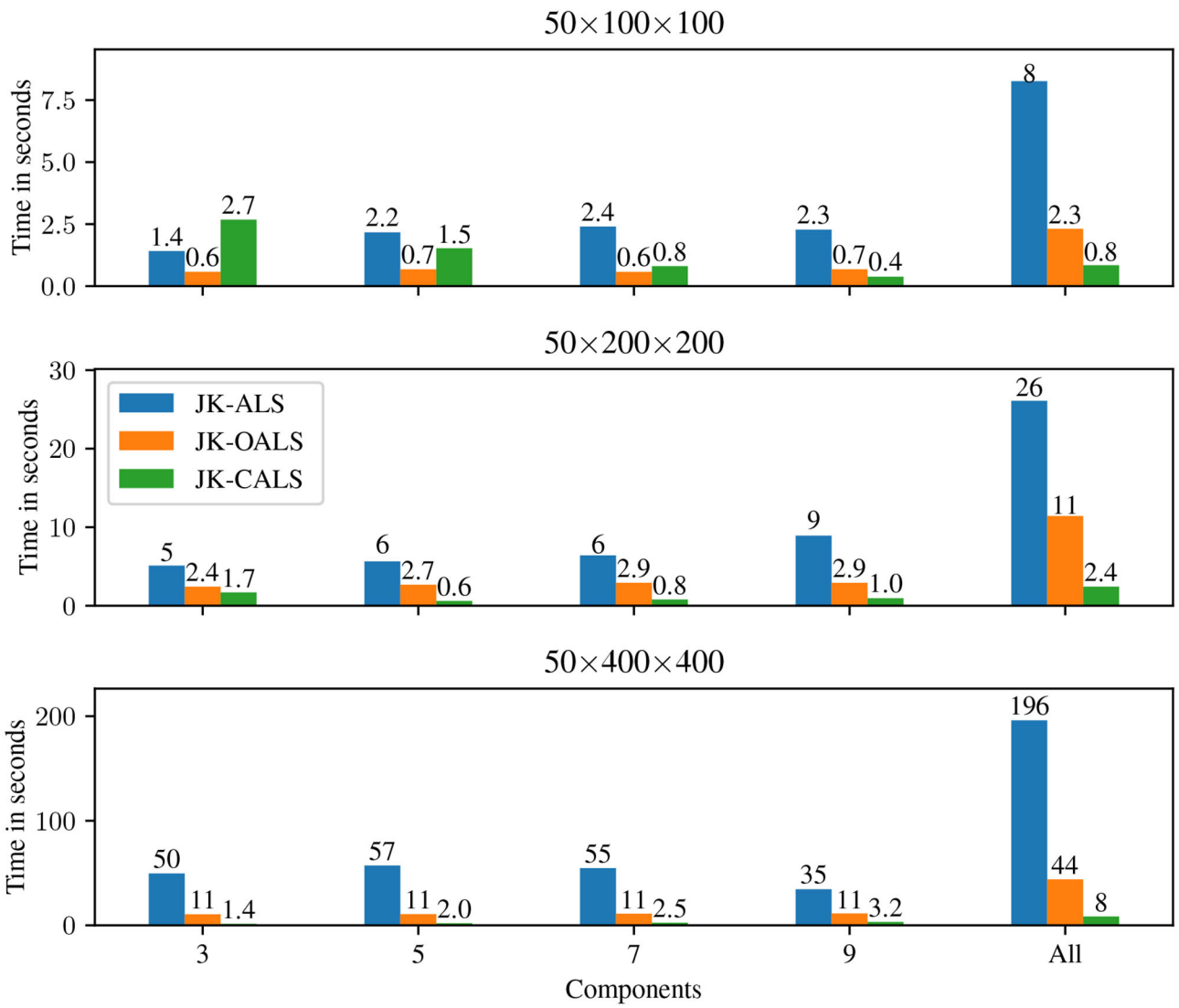


FIGURE 3 | Execution time for multi-threaded (24 threads) jackknife resampling applied to three different tensors (small, medium, and large in the top, middle, and bottom panels, respectively), and different number of components (from left to right: $R \in \{3, 5, 7, 9\}$, and "All", which represents doing jackknife to all four models simultaneously).

Figure 2 shows results for single threaded execution; in this case, JK-OALS is absent. JK-CALS consistently outperforms JK-ALS for all tensor sizes and workloads. Specifically, for any fixed amount of workload—i.e., a model of a specific number of components—JK-CALS exhibits increasing speedups compared to JK-ALS, as the tensor size increases. For example, for a model with 5 components, JK-CALS is 2.9, 3, 5.2 times faster than JK-ALS, for the small, medium and large tensor sizes, respectively.

Figure 3 shows results for multi-threaded execution, using 24 threads. In this case, JK-CALS outperforms the other two implementations (JK-ALS and JK-OALS) for the medium and large tensors, for all workloads (number of components), exhibiting speedups up to 35× and 8× compared to JK-ALS and JK-OALS, respectively. For the small tensor ($50 \times 100 \times 100$) and small workloads ($R \leq 7$), JK-CALS is outperformed by JK-OALS; for $R = 3$, it is also outperformed by JK-ALS. Investigating this further, for the small tensor and $R = 3$ and 5, the parallel speedup (the ratio between single threaded and multi-threaded execution time) of JK-CALS is 0.3× and 0.7× for 24 threads. However, for 12 threads, the corresponding timings are 0.28 and 0.27 s, resulting in speedups of 2.7× and 3.7×, respectively. This points to two main reasons for the observed performance of JK-CALS in these cases: a) the amount of available computational resources (24 threads) is disproportionately high compared to the volume

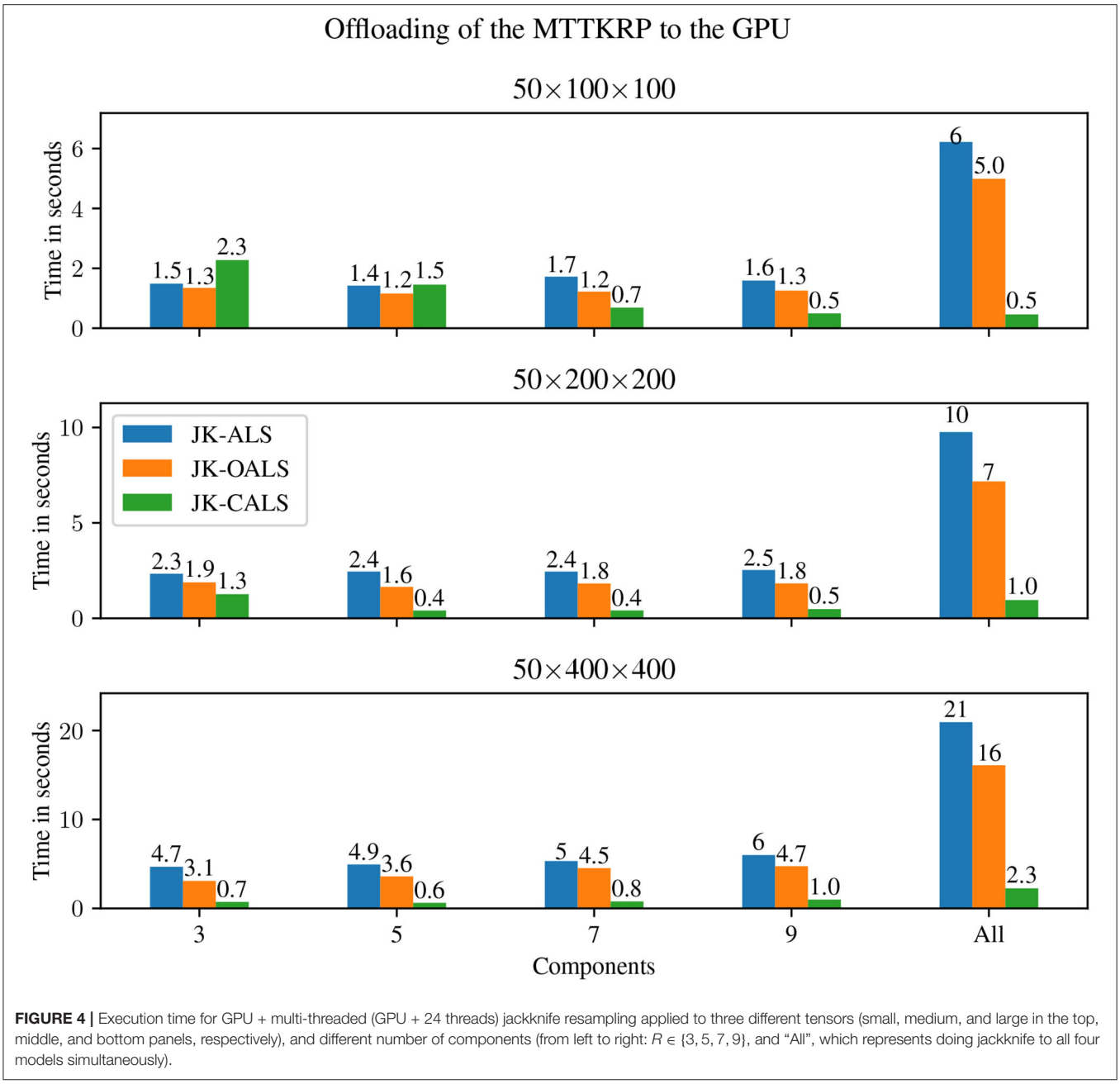


FIGURE 4 | Execution time for GPU + multi-threaded (GPU + 24 threads) jackknife resampling applied to three different tensors (small, medium, and large in the top, middle, and bottom panels, respectively), and different number of components (from left to right: $R \in \{3, 5, 7, 9\}$, and "All", which represents doing jackknife to all four models simultaneously).

of computation to be performed and b) because of the small amount of overall computation, the small overhead associated with the CALS methodology becomes more significant.

That being said, even for the small tensor, as the amount of workload increases—already for a single model with 9 components—JK-CALS again becomes the fastest method. Finally, similarly to the single threaded case, as the size of the tensor increases, so do the speedups achieved by JK-CALS over the other two methods.

Figure 4 shows results when the GPU is used to perform MTTKRP for all three methods; in this case, all 24 threads are used on the CPU. For the small tensor and small workloads ($R \leq 5$), there is not enough computation to warrant the shipping of data to and from the GPU, resulting in higher execution times compared to multi-threaded execution; for all other cases, all methods have reduced execution time when using the GPU compared to the execution on 24 threads. Furthermore, in those cases, JK-CALS is consistently faster than its counterparts, exhibiting the largest speedups when the workload is at its highest ("All"), with values of 10×, 7×, 7× compared to JK-OALS, and 12×, 10×, 9× compared to JK-ALS, for the small, medium and large tensors, respectively.

5.2. Real-World Applications

In this second experiment, we selected two tensors of size $89 \times 97 \times 549$ and $44 \times 2700 \times 200$ from the field of Chemometrics [30, 31]. In this field it is common to fit multiple, randomly initialized models in a range of low components (e.g., $R \in \{1, 2, \ldots, 20\}$,

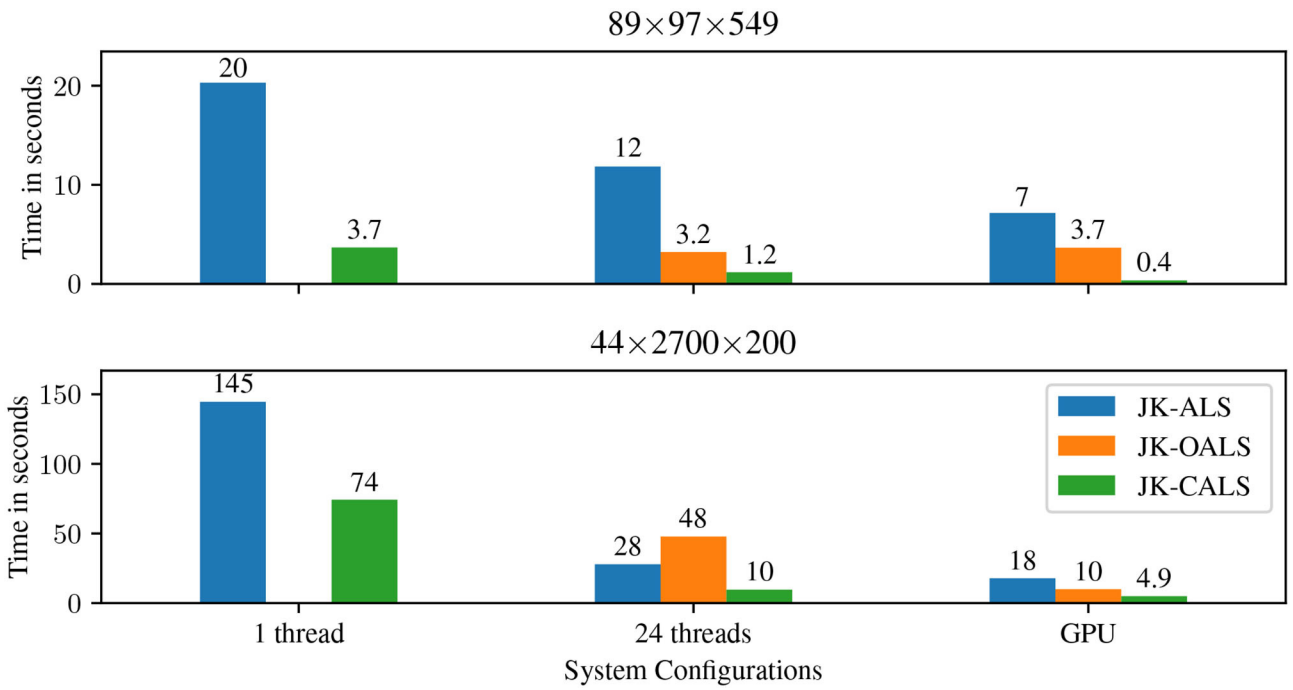


FIGURE 5 | Execution time for jackknife resampling applied to two applications tensors. For tensor 89 × 97 × 549, whose expected rank is 5, three models with $R \in \{4, 5, 6\}$ were fitted and then jackknife was applied to them (i.e., the "All" group from the previous section). Similarly, for tensor 44 × 2, 700 × 200, whose expected rank is 20, three models with $R \in \{19, 20, 21\}$ were fitted and then jackknifed. In both cases, tolerance and maximum number of iterations were set to 10^{-6} and 1, 000, respectively.

10–20 models for each R, and then analyze (e.g., using jackknife) those models that might be of particular interest (often those with components close to the expected rank of the target tensor); in the tensors we consider, the expected rank is 5 and 20, respectively. To mimic the typical workflow of practitioners, we fitted three models to each tensor, of components $R \in \{4, 5, 6\}$ and $R \in \{19, 20, 21\}$, respectively, and used the three methods (JK-ALS, JK-OALS and JK-CALS) to apply jackknife resampling to the fitted models[6]. The values for tolerance and maximum number of iterations were set according to typical values for the particular field, namely 10^{-6} and 1, 000, respectively.

In **Figure 5** we report the execution time for 1 thread, 24 threads, and GPU + 24 threads. For both datasets and for all configurations, JK-CALS is faster than the other two methods. Specifically, when compared to JK-ALS over the two tensors, JK-CALS achieves speedups of 5.4× and 2× for single threaded execution, 10× and 2.8× for 24-threaded execution. Similarly, when compared to JK-OALS, JK-CALS achieves speedups of 2.7× and 4.8× for 24-threaded execution. Finally, JK-CALS takes advantage of the GPU the most, exhibiting speedups of 17.5× and 3.7× over JK-ALS, and 9× and 2× over JK-OALS, for GPU execution.

[6]The same models were given as input to the three methods, and thus require the same number of iterations to converge.

6. CONCLUSION

Jackknife resampling of CP models is useful for estimating uncertainties, but the computation requires fitting multiple submodels and is therefore computationally expensive. We presented a new technique for implementing jackknife that better utilizes the computer's memory hierarchy. The technique is based on a novel extension of the Concurrent ALS (CALS) algorithm for fitting multiple CP models to the same underlying tensor, first introduced in Psarras et al. [7]. The new technique has a modest arithmetic overhead that is bounded above by factor of two in the worst case. Numerical experiments on both synthetic and real-world datasets using a multicore processor paired with a GPU demonstrated that the proposed algorithm can be several times faster than a straightforward implementation of jackknife resampling based on multiple calls to a regular CP-ALS implementation.

Future work includes extending the software to support grouped jackknife.

DATA AVAILABILITY STATEMENT

Publicly available datasets were analyzed in this study. This data can be found here: https://github.com/HPAC/CP-CALS/tree/jackknife.

AUTHOR CONTRIBUTIONS

CP drafted the main manuscript text, developed the source code, performed the experiments, and prepared all figures. LK and PB revised the main manuscript text. CP, LK, and PB discussed and formulated the jackknife extension of CALS. CP, LK, RB, and PB discussed and formulated the experiments. LK, RB, and CP discussed the related work section. PB oversaw the entire process. All authors reviewed and approved the final version of the manuscript.

FUNDING

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation)–333849990/GRK2379 (IRTG Modern Inverse Problems).

REFERENCES

1. Murphy KR, Stedmon CA, Graeber D, Bro R. Fluorescence spectroscopy and multi-way techniques. PARAFAC. *Anal Methods*. (2013) 5:6557–66. doi: 10.1039/c3ay41160e
2. Wiberg K, Jacobsson SP. Parallel factor analysis of HPLC-DAD data for binary mixtures of lidocaine and prilocaine with different levels of chromatographic separation. *Anal Chim Acta*. (2004) 514:203–9. doi: 10.1016/j.aca.2004.03.062
3. Farrance I, Frenkel R. Uncertainty of measurement: a review of the rules for calculating uncertainty components through functional relationships. *Clin Biochem Rev*. (2012) 33:49–75.
4. Riu J, Bro R. Jack-knife technique for outlier detection and estimation of standard errors in PARAFAC models. *Chemom Intell Lab Syst*. (2003) 65:35–49. doi: 10.1016/S0169-7439(02)00090-4
5. Kiers HAL. Bootstrap confidence intervals for three-way methods. *J Chemom*. (2004) 18:22–36. doi: 10.1002/cem.841
6. Martens H, Martens M. Modified Jack-knife estimation of parameter uncertainty in bilinear modelling by partial least squares regression (PLSR). *Food Qual Prefer*. (2000) 11:5–16. doi: 10.1016/S0950-3293(99)00039-7
7. Psarras C, Karlsson L, Bientinesi P. Concurrent alternating least squares for multiple simultaneous canonical polyadic decompositions. *arXiv preprint arXiv:2010.04678*. (2020).
8. Westad F, Marini F. Validation of chemometric models – a tutorial. *Anal Chim Acta*. (2015) 893:14–24. doi: 10.1016/j.aca.2015.06.056
9. Peddada SD. 21 Jackknife variance estimation and bias reduction. In: *Computational Statistics. Vol. 9 of Handbook of Statistics*. Elsevier (1993). p. 723-44. Available online at: https://www.sciencedirect.com/science/article/pii/S0169716105801452.
10. Efron B. The jackknife, the bootstrap and other resampling plans. In: *Society for Industrial and Applied Mathematics*. (1982). Available online at: https://epubs.siam.org/doi/book/10.1137/1.9781611970319
11. Kott PS. The delete-a-group jackknife. *J Off Stat*. (2001) 17:521. Available online at: https://www.scb.se/dokumentation/statistiska-metoder/JOS-archive/
12. Sanchez E, Kowalski BR. Generalized rank annihilation factor analysis. *Anal Chem*. (1986) 58:496–9. doi: 10.1021/ac00293a054
13. Acar E, Dunlavy DM, Kolda TG. A scalable optimization approach for fitting canonical tensor decompositions. *J Chemom*. (2011) 25:67–86. doi: 10.1002/cem.1335
14. Rajih M, Comon P, Harshman RA. Enhanced line search: a novel method to accelerate PARAFAC. *SIAM J Matrix Anal Appl*. (2008) 30:1128–47. doi: 10.1137/06065577
15. Shun Ang AM, Cohen JE, Khanh Hien LT, Gillis N. Extrapolated alternating algorithms for approximate canonical polyadic decomposition. In: *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. Barcelona: IEEE (2020). p. 3147–51.
16. Vervliet N, De Lathauwer L. A randomized block sampling approach to canonical polyadic decomposition of large-scale tensors. *IEEE J Sel Top Signal Process*. (2016) 10:284–95. doi: 10.1109/JSTSP.2015.2503260
17. Battaglino C, Ballard G, Kolda TG. A practical randomized CP tensor decomposition. *SIAM J Matrix Anal Appl*. (2018) 39:876–901. doi: 10.1137/17M1112303
18. Bro R, Andersson CA. Improving the speed of multiway algorithms: part II: compression. *Chemom Intell Lab Syst*. (1998) 42:105–13. doi: 10.1016/S0169-7439(98)00011-2
19. Solomonik E, Matthews D, Hammond J, Demmel J. Cyclops tensor framework: reducing communication and eliminating load imbalance in massively parallel contractions. In: *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. Cambridge, MA: IEEE (2013). p. 813–24.
20. Kannan R, Ballard G, Park H. A high-performance parallel algorithm for nonnegative matrix factorization. *SIGPLAN Not*. (2016) 51:1–11. doi: 10.1145/3016078.2851152
21. Lourakis G, Liavas AP. Nesterov-based alternating optimization for nonnegative tensor completion: algorithm and parallel implementation. In: *2018 IEEE 19th International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*. Kalamata: IEEE (2018). p. 1–5.
22. Smith S, Ravindran N, Sidiropoulos ND, Karypis G. SPLATT: efficient and parallel sparse tensor-matrix multiplication. In: *2015 IEEE International Parallel and Distributed Processing Symposium*. Hyderabad: IEEE (2015). p. 61–70.
23. Phipps ET, Kolda TG. Software for sparse tensor decomposition on emerging computing architectures. *SIAM J Sci Comput*. (2019) 41:C269–90. doi: 10.1137/18M1210691
24. Psarras C, Karlsson L, Li J, Bientinesi P. The landscape of software for tensor computations. *arXiv preprint arXiv:2103.13756*. (2021).
25. Buzas JS. Fast estimators of the jackknife. *Am Stat*. (1997) 51:235–40. doi: 10.1080/00031305.1997.10473969
26. Belotti F, Peracchi F. Fast leave-one-out methods for inference, model selection, and diagnostic checking. *Stata J*. (2020) 20:785–804. doi: 10.1177/1536867X20976312
27. Hinkle JE, Stromberg AJ. Efficient computation of statistical procedures based on all subsets of a specified size. *Commun Stat Theory Methods*. (1996) 25:489–500. doi: 10.1080/03610929608831709
28. Kolda TG, Bader BW. Tensor decompositions and applications. *SIAM Rev*. (2009) 51:455–500. doi: 10.1137/07070111X
29. Phan AH, Tichavský P, Cichocki A. Fast alternating LS algorithms for high order CANDECOMP/PARAFAC tensor factorizations. *IEEE Trans Signal Process*. (2013) 61:4834–4846. doi: 10.1109/TSP.2013.2269903
30. Acar E, Bro R, Schmidt B. New exploratory clustering tool. *J Chemom*. (2008) 22:91–100. doi: 10.1002/cem.1106
31. Skov T, Ballabio D, Bro R. Multiblock variance partitioning: a new approach for comparing variation in multiple data blocks. *Anal Chim Acta*. (2008) 615:18–29. doi: 10.1016/j.aca.2008.03.045

Conflict of Interest: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's Note: All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher.

frontiers | Frontiers in Applied Mathematics and Statistics

ORIGINAL RESEARCH
published: 18 April 2022
doi: 10.3389/fams.2022.826269

Tensor Processing Primitives: A Programming Abstraction for Efficiency and Portability in Deep Learning and HPC Workloads

Evangelos Georganas[1], Dhiraj Kalamkar[1], Sasikanth Avancha[1], Menachem Adelman[1], Deepti Aggarwal[1], Cristina Anderson[1], Alexander Breuer[2], Jeremy Bruestle[1], Narendra Chaudhary[1], Abhisek Kundu[1], Denise Kutnick[1], Frank Laub[1], Vasimuddin Md[1], Sanchit Misra[1], Ramanarayan Mohanty[1], Hans Pabst[1], Brian Retford[1], Barukh Ziv[1] and Alexander Heinecke[1]*

[1] Intel Corporation, Santa Clara, CA, United States, [2] Faculty of Mathematics and Computer Science, Friedrich-Schiller-Universität Jena, Jena, Germany

OPEN ACCESS

Edited by:
Edoardo Angelo Di Napoli, Helmholtz Association of German Research Centres (HZ), Germany

Reviewed by:
Markus Götz, Karlsruhe Institute of Technology (KIT), Germany
Jenia Jitsev, Helmholtz Association of German Research Centres (HZ), Germany

***Correspondence:**
Evangelos Georganas
evangelos.georganas@intel.com

Specialty section:
This article was submitted to Mathematics of Computation and Data Science, a section of the journal Frontiers in Applied Mathematics and Statistics

Received: 30 November 2021
Accepted: 21 March 2022
Published: 18 April 2022

Citation:
Georganas E, Kalamkar D, Avancha S, Adelman M, Aggarwal D, Anderson C, Breuer A, Bruestle J, Chaudhary N, Kundu A, Kutnick D, Laub F, Md V, Misra S, Mohanty R, Pabst H, Retford B, Ziv B and Heinecke A (2022) Tensor Processing Primitives: A Programming Abstraction for Efficiency and Portability in Deep Learning and HPC Workloads. Front. Appl. Math. Stat. 8:826269. doi: 10.3389/fams.2022.826269

During the past decade, novel Deep Learning (DL) algorithms, workloads and hardware have been developed to tackle a wide range of problems. Despite the advances in workload and hardware ecosystems, the programming methodology of DL systems is stagnant. DL workloads leverage either highly-optimized, yet platform-specific and inflexible kernels from DL libraries, or in the case of novel operators, reference implementations are built *via* DL framework primitives with underwhelming performance. This work introduces the Tensor Processing Primitives (TPP), a programming abstraction striving for efficient, portable implementation of DL workloads with high-productivity. TPPs define a compact, yet versatile set of 2D-tensor operators [or a virtual Tensor Instruction Set Architecture (ISA)], which subsequently can be utilized as building-blocks to construct complex operators on high-dimensional tensors. The TPP specification is platform-agnostic, thus, code expressed *via* TPPs is portable, whereas the TPP implementation is highly-optimized and platform-specific. We demonstrate the efficacy and viability of our approach using standalone kernels and end-to-end DL & High Performance Computing (HPC) workloads expressed entirely *via* TPPs that outperform state-of-the-art implementations on multiple platforms.

Keywords: deep learning, performance portability, programming abstraction, tensor processing, high productivity, high performance computing

1. INTRODUCTION

Since the advent of Deep Learning (DL) as one of the most promising machine learning paradigms almost 10 years ago, deep neural networks have advanced the fields of computer vision, natural language processing, recommender systems, and gradually pervade an increasing number of scientific domains [1–10]. Due to the diverse nature of the problems under consideration, these DL workloads exhibit a wide range of computational characteristics and demands. Furthermore, due to the immense computational cost of such workloads, industry and academia have developed specialized hardware features on commodity processors, and even specialized accelerators in order to harness these computational needs [11].

In contrary to the fast-evolving ecosystems of DL workloads and DL-oriented hardware/accelerators, the programming paradigm of DL systems has reached a plateau [12]. More specifically, the development of novel DL workloads involves two types of components: (i) Well-established operators within DL libraries (e.g., 2D convolutions, inner-product, batch-norm layers in oneDNN [13] and cuDNN [14]), and (ii) Unprecedented, custom primitives which typically instantiate new algorithmic concepts/computational motifs. Unfortunately both of these components come with their shortcomings.

On one hand, the operators within DL libraries are heavily optimized and tuned (usually by vendors) in a platform-specific fashion, leading to monolithic, non-portable, and inflexible kernels. Additionally, such opaque and high-level operators prohibit modular design choices since the user/frameworks have to adhere to particular interfaces that may not be adapted to fit the operation under consideration. On the other hand, the custom/unprecedented primitives are typically implemented by the user *via* the available generic/reference primitives of a Machine Learning (ML) framework which are not optimized and as such yield underwhelming performance. It is up to the user to create optimized implementations for the custom primitives, leading again to code which is non-portable and potentially requires hardware expertise in order to achieve peak performance. Unfortunately, most of the times such expertise is not available to the data/ML scientist who is developing the custom DL primitive. Therefore, the deployment (or even the evaluation) of a new operator typically requires yet another stage in the development cycle where low-level optimization experts are working on the re-write/fine-tuning of the operator. Later on, in case an operator proves to be important for the community, systems researchers and vendors standardize it, and potentially create yet another monolithic kernel within a DL library for further re-use within DL frameworks. This entire development cycle potentially takes a considerable amount of time (up to years in some cases) and inadvertently impedes the efficient exploration of innovative machine learning ideas [12]. An alternative approach to optimize both types of operators is to leverage contemporary Tensor Compilers (TC) (e.g., [15–18]), however, the state-of-the-art tools are only suitable for compiling small code-blocks whereas large-scale operators require prohibitive compilation times, and often the resulting code performs far from the achievable peak [12].

We identify that the common source of the problems mentioned in the previous paragraph is the extreme levels of abstraction offered by the DL libraries and the Tensor Compilers. The DL libraries offer coarse-grain, monolithic and inflexible operators whereas the Tensor Compilers usually go to the other extreme, allowing the user to express arbitrary low-level operators without any minimal restrictions that would readily enable efficient lifting and code-generation in their back-ends (e.g., they offer no minimal/compact set of allowed operations on tensors). To exacerbate the challenge of optimal code generation, Tensor Compilers usually undertake the cumbersome tasks of efficient parallelization, loop re-ordering, automatic tiling and layout transformations, which, to date, remain unsolved in the general setup. Also, there is not a well-established way to share state-of-the-art optimizations among the plethora of Tensor Compilers and as a result each one has its own advantages and disadvantages, which translates eventually to sub-optimal performance on real-world scenarios [19]. We note, here, the recent, promising effort of Multi-Level Intermediate Representation (MLIR) [20] toward unifying the optimization efforts in the Tensor Compiler Intermediate Representation (IR) infrastructure.

In this work, we introduce the Tensor Processing Primitives (TPP), a programming abstraction striving for efficient and portable implementation of Tensor operations, with a special focus on DL workloads. TPPs define a set of relatively low-level primitive operators on 2D Tensors, which, in turn, can be used as basic building blocks to construct more complex operators on high-dimensional tensors. TPPs comprise a minimal and compact, yet expressive set of precision-aware, 2D tensor level operators to which high-level DL operators can be reduced. TPPs's specification is agnostic to targeted platform, DL framework, and compiler back-end. As such the code which is expressed in terms of TPPs is portable. Since the level of abstraction that TPPs adopt is at the sub-tensor granularity, TPPs can be directly employed by DL workload developers within the frameworks, or could be alternatively used to back up an IR within a Tensor Compiler stack, i.e., TPPs could form the basis of an MLIR dialect.

While the TPP specification is agnostic of the targeted framework/platform/compiler stack, its implementation is platform specific, and is optimized for the target architectures. This subtle detail offers a clear separation of concerns: the user-entity of TPPs, either a developer or a compiler framework, can focus on expressing the desired algorithm and its execution schedule (e.g., parallelization, loop orders) using the TPP tensor abstraction, whereas the efficient, platform-specific code generation pertaining to the TPP operations belongs to the TPP back-end. To this extent, TPPs could be also viewed as a "virtual Tensor ISA" that abstracts the actual physical ISA of the target (e.g., SSE, AVX2, AVX512, AMX for x86, AArch64 and ARMv8 SVE, xPU).

Figure 1 shows various use-cases of TPPs within multiple software stacks. TPPs can be viewed as a layer abstraction of the actual physical target ISA, and the user-entities can rely on the TPP layer for the code generation pertaining to the tensor operations. Also, **Figure 1** illustrates the various user-entities that might leverage TPPs. First, the vendor-optimized DL libraries (e.g., oneDNN or oneDNN Graph) can use TPPs for optimized code generation in their back-end. Second, the user/developer of the DL operators can directly leverage TPPs within a DL framework extension to express the underlying tensor computations (e.g., the user may develop a framework extension for a novel DL operator by employing the TPPs as building blocks). Third, Tensor Compilers can leverage TPPs (e.g., as part of an MLIR dialect) to generate high-quality code for the corresponding tensor operators. As such, the TPP layer abstraction offers a clear separation of concerns where the Tensor Compiler may focus on higher-level optimizations (loop tiling and re-ordering, parallelization, etc.) whereas the platform-specific code generation of the tensor operations is undertaken

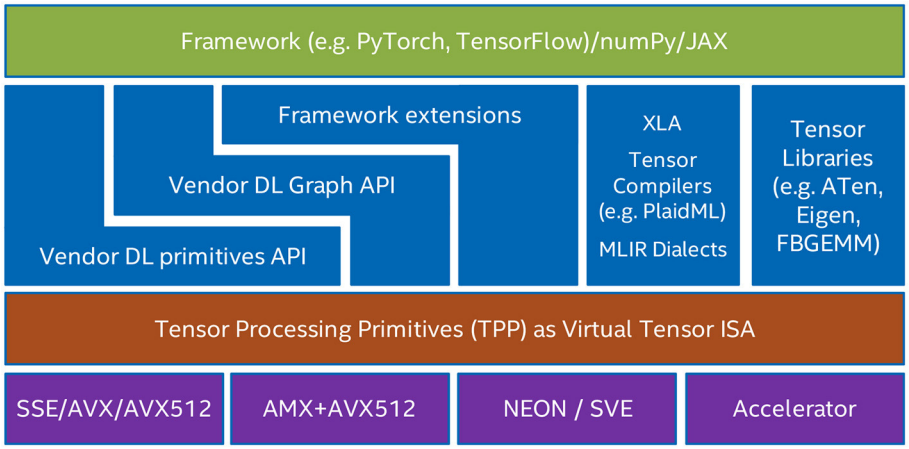


FIGURE 1 | Use-cases of TPPs in various software stacks.

by the TPP layer. Such a synergistic Tensor Compiler - TPP paradigm is illustrated in section 7. Last but not least, TPPs could be leveraged by more general Tensor Libraries (e.g., ATen, Eigen) where tensor computations constitute the primary focus and they can be naturally mapped to TPPs.

In our Proof-Of-Concept (POC) implementation of TPPs we leverage Just-In-Time (JIT) technology to emit performant and platform-specific code during runtime. Furthermore, in our POC we define a mini embedded Domain Specific Language (mini-eDSL) where the TPPs can be combined *via* matrix equations in order to build high-level operators without sacrificing performance.

We demonstrate the efficacy of our approach on multiple platforms using standalone kernels written entirely with TPPs and compare the performance to vectorized-by-expert code and compiler generated code. Finally, we showcase the expressiveness and viability of our methodology by implementing contemporary end-to-end DL workloads using solely the TPP abstractions and show how we can outperform the state-of-the-art implementations on multiple platforms. The main contributions of this work are:

- A TPP specification/foundation for primitive tensor operations.
- A Proof-Of-Concept implementation of the TPP specification along with a mini-eDSL (called TPP Matrix Equations), enabling efficient fusion of TPPs that lead to portable, high-level tensor operations. We describe in detail various standalone TPP implementations, and also we provide a detailed analysis of our TPP Matrix Equation mini-eDSL framework.
- A demonstration of how contemporary and novel DL algorithmic motifs/workloads can be expressed in their entirety *via* TPPs.
- An experimental evaluation of the TPP-based DL workloads from all relevant fields (image processing, recommendation systems, natural language processing, graph processing, and applications in science) on multiple platforms (different instruction set architectures (ISAs) x86_64 and aarch64, and micro-architectures for each ISA), including distributed-memory scaling. We show performance that matches/exceeds the state-of-the-art implementations, while maintaining flexibility, portability, and obviating the need for low-level platform-specific optimizations.
- We show how TPPs can be leveraged as a virtual Tensor ISA within a Tensor compiler software stack, yielding high-performance DL primitives.
- We illustrate examples of how TPPs are used outside of Deep Learning, in High Performance Computing (HPC) applications in order to accelerate tensor computations.

Section 2 details the specification of the TPPs. Then, section 3 illustrates a POC implementation of the TPP specification. Section 4 presents an infrastructure that enables efficient TPP fusion. In section 5, we exhibit how contemporary DL motifs/algorithmic paradigms can be expressed *via TPPs*. Section 6 presents an experimental evaluation of TPP-based DL kernels and workloads on multiple platforms. Section 7 outlines our POC implementation of a TPP backend within a tensor compiler (PlaidML), and also presents some results highlighting the viability of the TPP abstraction as a virtual Tensor ISA within tensor compiler stacks. Section 8 presents exemplary usage of TPPs within HPC applications in order to efficiently implement tensor computations. Sections 9 and 10 summarize the related work and conclude this article.

2. THE TPP SPECIFICATION

2.1. TPP Design Principles

The TPP specification is driven by a few design principles:

1) *Each TPP corresponds to a mathematical operator that takes a number of input(s) and produces an output*. We opt to specify TPPs that correspond to basic, well-defined mathematical tensor operations. In this way, we keep the set of TPPs *minimal* albeit *expressive*; basic TPPs can be combined to formulate more complex operators.

2) *The inputs/outputs of the TPPs are abstract 2D tensors that can be fully specified by their shape/size, leading dimensions, and precision*. Additionally, the 2D tensors hold the following complementary *runtime* information: (i) a *primary* field which corresponds to the memory address where the 2D (sub)tensor data resides, (ii) a *secondary* field holding optional data for the tensor (e.g., a mask for the tensor), and (iii) a *tertiary* field holding optional, auxiliary information of the tensor (e.g., scaling factors for a quantized tensor).

3) *TPPs are specified as "memory-to-memory" operations, or equivalently the input/output tensors are residing in memory locations specified by the user*. This design decision is critical in order to abstract the TPPs from all physical ISAs, and enables true platform-agnostic specification. For example, if the TPPs were accepting vector registers as inputs/outputs, then the number of physical registers, the vector length and dimensionality would be exposed in the Application Programming Interface (API) of TPPs, making the specification platform-specific.

4) *TPPs have declarative semantics*. As such, the TPP specification does not preclude potential parallelism [e.g., Single Instruction Multiple Data (SIMD), Single Instruction Multiple Threads (SIMT)] in the back-end implementation which is target-specific.

5) *TPPs are composable in a producer-consumer fashion*. Since the output of a TPP is a well-defined tensor O, it can be fed as input to a subsequent TPP. In such a scenario, this "intermediate" tensor O is not necessarily exposed to the user, unless the user explicitly requires it (e.g., by combining the TPPs in a manual fashion *via* an explicit temporary O buffer/tensor which lives in the user space/application). This flexibility allows the TPP implementation (which is platform-specific) to combine TPPs in the most efficient way for the target architecture (e.g., the O tensor can live at the physical register file in the composite TPP in order to avoid redundant memory movement).

6) *The TPP input/output tensors as well as the computation itself are precision aware*. This feature makes mixed precision computations (that are prominent in DL workloads) easy to express from the user point of view, and provides information to the TPP back-end that may enable efficient implementation.

2.2. TPP Arguments

As mentioned in the previous subsection, the input to TPPs are 2D tensors. Each 2D tensor can be specified by the number of rows M, columns N, its leading dimension ld and its datatype *dtype*. Additionally, during runtime each tensor gets fully characterized by specifying its location/address as *primary* info, optional companion tensor info as *secondary* (e.g., sparsity bitmask), and optionally *tertiary* info (e.g., in case the tensor shape is dynamically determined at runtime, this info may contain variables specifying M/N). Each TPP also specifies the shape/precision of the produced/output 2D tensor.

Each TPP also supports input tensors with *broadcast* semantics. More specifically, TPPs accept optional flags dictating that the input 2D tensor should be formed by broadcasting a column/row/scalar $N/M/M \times N$ times, respectively. Finally, the TPPs accept optional flags which further specify the TPP operation. For example, in case a TPP is computing a transcendental function, the flags may be specifying various approximation algorithms used for the computation. In the next subsection, we present the TPPs in three groups: *unary*, *binary*, and *ternary* TPPs given the number of input tensors they accept.

2.3. The TPP Collection

First, we highlight the ternary *Batch-Reduce GEneral Matrix to Matrix Multiplication* (BRGEMM) TPP which is the main building block for general tensor contractions in DL kernels [21]. BRGEMM materializes the operation $C = \beta \cdot C + \sum_{i=0}^{n-1} A_i \times B_i$. In essence, this kernel multiplies the specified blocks $A_i^{M \times K}$ and $B_i^{K \times N}$ and reduces the partial results to a block $C^{M \times N}$. It is noteworthy that tensors A and B can alias and also the blocks A_i and B_i can reside in any position in the input (potentially high-dimensional) tensors A and B. Previous work [21] has shown that this single building block is sufficient to express efficiently tensor contractions in the most prevalent DL computational motifs, namely: Convolution Neural Networks (CNN), Fully-Connected networks (FC), Multi-Layer Perceptrons (MLP), Recurrent Neural Networks (RNN)/Long Short-Term Memory (LSTM) Networks. In Section 5 we exhibit how BRGEMM can be further used to build efficient Attention Cells that comprise the cornerstone of modern Natural Language Processing (NLP) workloads. BRGEMM can be specialized to one of the following three variants that may enable more efficient implementations on various platforms: (i) *address-based BRGEMM*, where the addresses of the blocks A_i and B_i are explicitly provided by the user, (ii) *offset-based BRGEMM*, where the addresses of A_i and B_i can be computed as $address\_A_i = address\_A + offset_{A_i}$ and $address\_B_i = address\_B + offset_{B_i}$, and (iii) *stride-based BRGEMM*, where the addresses of A_i and B_i are: $address\_A_i = address\_A_{i-1} + stride\_A$ and $address\_B_i = address\_B_{i-1} + stride\_B$. In section 3.2, we present the implementation of the BRGEMM TPP in more depth for various ISAs and platforms.

Table 1 presents the unary TPPs that accept one 2D tensor as input. Since most of these TPPs map directly to the equivalent math function, we further elaborate only on the ones which are more complex. The *Identity* TPP essentially copies the input to the output. Since the input and output are fully specified in terms of their precision, this TPP can be also used to perform datatype conversions between tensors.

The *Quantize & Dequantize* TPPs are used to quantize/dequantize the input tensor whereas the exact algorithm employed is specified by a TPP flag.

The *Transform* TPP uses a flag to determine the exact transformation applied on the input 2D tensor. The *Transpose* transformation is the usual mathematical matrix transpose. The rest two types of transformation, namely *Vector Neural Network Instructions (VNNI) formatting*, and *VNNI to VNNI-transpose* are DL specific. More specifically, modern hardware (e.g., Intel's Cooper Lake) requires tensors to be in specific format called *VNNI* in order to employ hardware acceleration for specific operations, e.g., dot-products (see section 3.2.2 for more details). This format represents a logical 2D tensor $[D_1][D_0]$ as a 3D tensor $[D_1/\alpha][D_0][\alpha]$ where essentially the dimension D_1 is blocked in chunks of size α, which in turn are set as the inner-most tensor dimension. The *VNNI*

TABLE 1 | Unary TPPs.

Unary TPP	Description/Comments
Identity	Copies input to output. Given input/output datatype, it performs datatype conversions
Zero	Fills output with zeros
Square	Squares input and stores to output
Increment / decrement	Increments / Decrements input by 1 and stores to output
Square root	Computes the square root of input and stores to output
Reciprocal	Computes the reciprocal of input and stores to output
Rcp. Sqrt.	Computes the rcp. sqrt. of input and stores to output
Exp	Computes the exponential value of the input tensor entries and stores them to output
PRNG	Generates an output tensor with pseudo-random entries
(De)Quantize	Quantizes / Dequantizes the input
Reduce	Reduces the rows/columns of the input and stores to output. The reduction function can be SUM/MUL/MIN/MAX; (optionally) reduces the *squared* input
Transform	Transforms input and stores to output. Transformations are: Transpose, VNNI formatting, and VNNI to VNNI-transpose
Unpack	Takes each entry $x_{i,j}$ of the input tensor, splits it in two parts $x^{lo}_{i,j}$ and $x^{hi}_{i,j}$ with same bit-width, and stores them in two tensors X^{lo}, X^{hi}
Replicate columns	Takes an input column/vector, replicates it a variable number of times and forms the output
Gather / Scatter	Gathers/Scatters rows/columns from input and forms the tensor
2D Gather / 2D Scatter	Gathers/scatters elements from input using 2D offsets
2D-strided loads/stores	Loads/stores elements from/to a tensor using primary and secondary strides
Tanh &Tanh_inv	Computes the hyperbolic tangent function (or its inv used for back-propagation) on input
RELU & RELU_inv	Apply a Rectified Linear Unit function (or its inv used for back-propagation) on input
Sigmoid & Sigmoid_inv	Computes the logistic sigmoid (or its inv used for back-propagation) on input
GELU & GELU_inv	Apply a Gaussian Error Linear Unit function (or its inv used for back-propagation) on input
Dropout & Dropout_inv	Drops out values from the input tensor with probability p. For the inv/back-propagation pass, the same dropped units are zeroed out

formatting TPP performs this exact transformation: $[D_1][D_0] \rightarrow [D_1/\alpha][D_0][\alpha]$ and the *VNNI to VNNI-transpose* transposes a tensor which is already laid out in VNNI format, i.e., performs $[D_1/\alpha_1][D_0][\alpha_1] \rightarrow [D_0/\alpha_0][D_1][\alpha_0]$. In section 3.3.1, we outline how the Transform TPPs are implemented *via* Shuffle Networks.

The last four entries of **Table 1** correspond to DL-specific operations. They correspond to activation functions typically encountered in DL workloads. All these activation functions have a counterpart which is required during the back-propagation pass of training DL networks. These DL specific TPPs could be built on top of other TPPs, however, since they are prevalent in DL workloads we opt to define them as self-contained TPPs for ease of usage. In section 3.3.2, we describe the TPP implementation of non-linear approximations for several activation functions on various ISAs.

TABLE 2 | Binary TPPs.

Binary TPP	Description/Comments
Add	Add two inputs
Sub	Subtracts two inputs
Mul	Multiples (elementwise) two inputs
Div	Divides two inputs
Max/Min	Finds element-wise max/min of two inputs
MatMul	Performs matrix multiplication of two input
Pack	Concatenates pairs of entries $x^{lo}_{i,j}$ and $x^{hi}_{i,j}$ from the inputs X^{lo}, X^{hi} into $x_{i,j}$ and stores it to the output X
Compare	Compares element-wise two inputs and stores a bitmask of the comparison

TABLE 3 | Ternary TPPs.

Ternary TPP	Description/Comments
GEMM	Performs on 2D inputs A, B, C, scalar β: $C = \beta C + A \times B$
Batch-Reduce GEMM	Performs on 2D inputs A_i, B_i (with $i = 0, 1, \ldots, n-1$), C, scalar β: $C = \beta C + \sum_{i=0}^{i=n-1} A_i \times B_i$
(N)MulAdd	Performs on 2D inputs A, B, C: $C = C + A \odot B$ (or $C = C - A \odot B$); $\odot$ denotes element-wise multiplication
Blend	Blends 2D input tensors A, B according to bitmask C

Tables 2, **3** present the binary/ternary TPPs that accept two/three 2D tensor as inputs, respectively.

3. TPP IMPLEMENTATION

In this section, we briefly describe our Proof-Of-Concept (POC) implementation of the TPP specification. Our implementation targets multiple CPU architectures from various vendors that support different ISAs, but could be readily extended to support even GPU ISAs. We build upon and extend the open source LIBXSMM [22] library which leverages JIT techniques. Such JIT techniques have been successfully used for optimal code generation on CPUs by taking advantage of the known (at runtime) tensor shapes/dimensions in HPC and DL applications [21–23]. Nevertheless, the TPP specification is platform-agnostic and does not preclude any TPP back-end implementation. In our POC implementation, the usage of TPPs is governed by two APIs: (i) A dispatch API with which the user can request the code generation of a specific TPP, and such a dispatch call JITs a function implementing the requested operation, (ii) an API to call the JITed TPP kernel. First, in section 3.1, we provide a generic blueprint of our TPP implementation. Then, in section 3.2, we describe in more detail the BRGEMM TPP implementation which comprises the main tensor contraction tool in the TPP abstractions. Section 3.3.1 details the implementation of the unary transform TPPs *via* shuffle networks since their efficient implementation diverts from the generic TPP blueprint. Finally, section 3.3.2

Algorithm 1 | The generic unary/binary/ternary TPP algorithm.

Inputs: $\mathbf{X}^{M\times N}$, ($\mathbf{Y}^{M\times N}$, $\mathbf{Z}^{M\times N}$ if binary/ternary)
Output: $\mathbf{O}^{M\times N}$

1: **for** $i_n = 0 \ldots N-1$ **with step** $\mathbf{n_b}$ **do**
2: **for** $i_m = 0 \ldots M-1$ **with step** $\mathbf{m_b}$ **do**
3: ▷ *Generic loads, may have broadcast/gather semantics,*
4: ▷ *and may perform datatype conversions*
5: $X_b \leftarrow$ load_generic $m_b \times n_b$ X-subblock$_{i_m,i_n}$
6: **if** (unary TPP) **then**
7: $X_b \leftarrow$ **Unary_op**(X_b)
8: **if** (binary TPP) **then**
9: $Y_b \leftarrow$ load_generic $m_b \times n_b$ Y-subblock$_{i_m,i_n}$
10: $X_b \leftarrow$ **Binary_op**(X_b, Y_b)
11: **if** (ternary TPP) **then**
12: $Y_b \leftarrow$ load_generic $m_b \times n_b$ Y-subblock$_{i_m,i_n}$
13: $Z_b \leftarrow$ load_generic $m_b \times n_b$ Z-subblock$_{i_m,i_n}$
14: $X_b \leftarrow$ **Ternary_op**(X_b, Y_b, Z_b)
15: ▷ *Generic store, may have scatter semantics, and may*
16: ▷ *perform datatype conversion*
17: O-subblock$_{i_m,i_n}$ $\overleftarrow{\text{store\_generic}}$ X_b

Algorithm 2 | The batch-reduce GEMM TPP.

Inputs: $A_i^{M\times K}$, $B_i^{K\times N}$ for $i = 0, \ldots, n$-1, $C^{M\times N}$, $\beta \in \mathbb{R}$
Output: $C = \beta \cdot C + \sum_{i=0}^{n-1} A_i \times B_i$

1: **for** $i_n = 0 \ldots N-1$ **with step** $\mathbf{n_b}$ **do**
2: **for** $i_m = 0 \ldots M-1$ **with step** $\mathbf{m_b}$ **do**
3: acc_regs $\leftarrow$ load_generic $m_b \times n_b$ C-subblock$_{i_m,i_n}$
4: **for** $i = 0 \ldots n-1$ **with step 1 do**
5: **for** $i_k = 0 \ldots K-1$ **with step** $\mathbf{k_b}$ **do**
6: ▷ *Outer product GEMM microkernel*
7: acc_regs += A_i sub-panel$_{i_m,i_k}$ $\times$ B_i sub-panel$_{i_k,i_n}$
8: C-subblock$_{i_m,i_n}$ $\overleftarrow{\text{store\_generic}}$ acc_regs

outlines the approximation techniques we leverage in our TPP implementation of non-linear activation functions; such approximations are essential in achieving high-performance, while at the same time their accuracy is sufficient for the purposes of training DL workloads.

3.1. Generic TPP Implementation Blueprint

Algorithm 1 exhibits at a high-level the pseudocode that is used to implement the Unary/Binary/Ternary TPPs in a unified fashion. The inputs of the TPPs are tensors X, Y (in case of binary/ternary TPPs) and Z (in case of ternary TPP), and an output tensor O. For the purposes of this simplified presentation we assume all tensors are of size $M \times N$, however, depending on the operation these might have different sizes. For example, if the unary OP is a reduction-by-columns and the input is $M \times N$, then the output is an $M \times 1$ vector. First, we show that the M/N loops are blocked with factors m_b/n_b such that the working sets of each microkernels fits on the available register file. The latter is architecture specific, e.g., AVX2-enabled ISAs expose 16 256-bit vector registers, AVX512-enabled ISAs expose 32 512-bit vector registers, and Aarch64 features 32 128-bit (NEON)/512-bit (SVE) vector registers. The "load_generic" function used in **Algorithm 1** denotes the loading of a sub-tensor to a register block; this load may imply row/column/scalar broadcast semantics if the user specified the TPP in that way, or even may have strided-load/gather semantics if the TPP involves a strided-load/gather operation. Also, for simplicity we do not show here the handling of "secondary" fields of the tensors that may be required (e.g., indices array for the gather operation, bitmasks arrays). Additionally, the generic load also handles datatype conversion, for instance provided the input is in bfloat16 (BF16) [24] whereas the compute is going to happen in FP32 precision. Once all the required sub-tensors are loaded, then the corresponding Unary/Binary/Ternary operator is applied. This operator may be directly mapped to an available instruction (e.g., a vector add in case of binary addition), or to a sequence of instructions for more complicated operators (e.g., reductions, random number generation *via* xorshift algorithm [25], approximation algorithms for transcendental functions [26]). Last but not least, the optimal sequence generation depends on the available instructions and this is handled by the TPP back-end/JITer. For example, some ISAs may have masking/predicate support (e.g., AVX512 & SVE) that enable efficient handling of loop remainders, the selected unrolling degree heavily depends on the instructions in use, their latency and the number of available architectural registers. Once the result is computed, the resulting register block is stored back to the corresponding output sub-tensor position. Similarly to the generic load, the "generic" store may induce strided accesses or may be even a scatter operation. Additionally, the generic store also handles potential datatype conversions.

3.2. The BRGEMM TPP Implementation

3.2.1. The BRGEMM Kernel Structure

We present in more detail the BRGEMM TPP because it comprises the tensor contraction tool in the TPP abstraction, and is ubiquitous in the DL kernels and workloads described in section 5. **Algorithm 2** exhibits the high-level algorithm implementing: $C = \beta \cdot C + \sum_{i=0}^{n-1} A_i \times B_i$. Lines 1-2 block the computation of the result C in $m_b \times n_b$ tensor sub-blocks. Once such a subblock is loaded into the accumulation registers (line 3), we loop over all pairs A_i, B_i (line 4) and we accumulate into the loaded registers the products of the corresponding $m_b \times K$ subblocks of A_i with the relevant $K \times n_b$ subblocks of B_i (lines 5–7). In order to calculate a partial product of an $m_b \times k_b$ sub-panel of A_i with a $k_b \times n_b$ sub-panel of B_i, we follow an outer product formulation. The loading of A_i and B_i sub-panels, and the outer-product formulation is heavily dependent on the target platform. We provide BRGEMM implementations for multiple x86 ISAs: SSE, AVX, AVX2, AVX512, including the recently introduced Intel AMX (Advanced Matrix Extensions) ISA [27]. Additionally, we have implemented the BRGEMM TPP for AArch64 and ARMv8 SVE ISAs. Depending on the targeted platform, the "register" can be either a typical vector register with varying width (e.g., 128–512 bit vector length), or

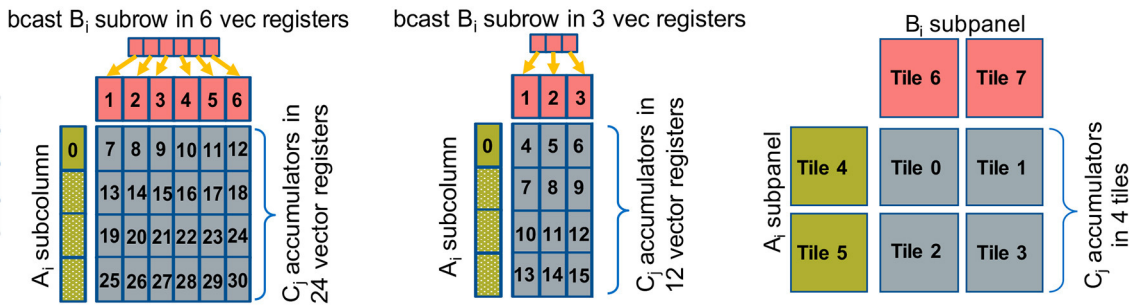


FIGURE 2 | Outer product GEMM microkernels, **Left**: On a platform with 32 vector registers, **Middle**: On a platform with 16 vector registers, **Right**: On a platform with 8 2D registers (tiles).

in the case of AMX-enabled target the "register" is a 2D tile-register. Similarly, the outer-product formulation may employ the available Fused-Multiply-Add (FMA) instructions, or even 2D tile-multiplication instructions. In all these cases, the TPP implementation emits the appropriate load/store/prefetch/FMA instructions, and takes into account the available architectural registers/unrolling factors/instruction mix in order to achieve close to peak performance. Last but not least, the BRGEMM supports multiple datatypes (FP64, FP32, BF16, INT8), and whenever possible employs hardware acceleration, e.g., *via* specialized FMA instructions for INT8/BF16 datatypes. In order to highlight the differences of the outer product GEMM microkernels that are heavily dependent on the target platform, we show in **Figure 2** three different implementations.

Figure 2-Left shows an exemplary outer product microkernel on a platform with 32 available vector registers, for example an x86 with AVX512 or on ARM AArch64/SVE. In this case vector register v7-v30 constitute the accumulators, vector registers v1-v6 hold a broadcasted subrow of B, and vector register v0 is used to load a partial subcolumn of A. First, we load on v1-v6 a subrow of B *via* broadcasts, then we load on v0 the first chunk of the A subcolumn and with six fused multiply-add (FMA) instructions (v0 with v1-v6) we multiply-and-add the corresponding partial results on the accumulators v7-v12 (first logical row of accumulators). Then, we load on v0 the second chunk of the A subcolumn, and subsequently with yet another six FMA instructions (v0 with v1-v6) we multiply-and-add the computed partial results on the accumulators v13-v18 (second logical row of accumulators), etc. The registers v1-v6 are reused four times throughout the outer product computation, and v0 is reused six times for each loaded A chunk. In other words, the corresponding A subcolumn and B subrow are loaded from memory/cache into the vector registers exactly once and we get to reuse them from the register file. Also, in such a formulation, we expose 24 independent accumulation chains which is critical in order to hide the latency of the FMA instruction. Last but not least, the platform (i.e., vector register width) and the datatype of the microkernel determine the exact values of the blocking parameters m_b, n_b, and k_b. For example for single precision datatype FP32 and an x86 AVX512 platform, each vector register can hold 16 FP32 values (the vector registers are 512-bit wide). Therefore, this microkernel operates with blocking values $m_b = 64$, $n_b = 6$, and $k_b = 1$ and it calculates a small matrix multiplication $C_{64\times6} \mathrel{+}= A_{64\times1} \times B_{1\times6}$.

Figure 2-Middle shows an exemplary outer product microkernel on a platform with 16 vector registers, for example an x86 with up to AVX2 ISA. The microkernel is similar with the previous case; since we have only 16 vector registers available, we dedicate 12 of those as C accumulators, 3 vector register are utilized for holding a partial B subrow, and 1 vector register is used to load a chunk of an A subcolumn. In this case 12 independent accumulation chains are also sufficient to hide the FMA latency. Analogously to the previous case, for single precision datatype FP32 and an x86 AVX2 platform, each vector register can hold now 8 FP32 values (the vector registers are now 256-bit wide). Thus, this microkernel operates with blocking values $m_b = 32$, $n_b = 3$, and $k_b = 1$ and it calculates a small matrix multiplication $C_{32\times3} \mathrel{+}= A_{32\times1} \times B_{1\times3}$.

Figure 2-Right shows a small GEMM microkernel on a platform with 8 2D registers (tiles), for example what is available in the recently introduced Intel AMX (Advanced Matrix Extensions) ISA. In this case each 2D tile register has size (up to) 1KB, logically holds (up to) 16 rows of a submatrix, and can be loaded with a proper tile-load instruction. In this particular example, tiles 0-3 comprise the C accumulators, tiles 4-5 are used to hold a subpanel of A and tiles 6-7 are used to hold a subpanel of B. Once we load the subpanels of A and B onto the respective tiles, we can perform 4 tile multiply-and-add instructions: $tile0 \mathrel{+}= tile4 \times tile6$, $tile1 \mathrel{+}= tile4 \times tile7$, $tile2 \mathrel{+}= tile5 \times tile6$ and $tile3 \mathrel{+}= tile5 \times tile7$, and we update the C accumulators. In such a microkernel, each A/B tile is reused 2 times. Given each tile may have size up to 1KB and may hold up to 16 rows of a submatrix, by considering BF16 datatype for A/B matrices and FP32 accumulator tiles, such a microkernel operates with blocking values $m_b = 32$, $n_b = 32$, $k_b = 32$, and can compute (up to) a small matrix multiplication $C_{32\times32} \mathrel{+}= A_{32\times32} \times B_{32\times32}$. Each A/B tile represents a logical 16×32 BF16 A/B submatrix, and each C tile represents a 16×16 FP32 accumulator. The AMX instructions will be available within the upcoming Intel Xeon processors code-named Sapphire Rapids, and the corresponding BF16-input/FP32-output tile multiplication instructions can deliver

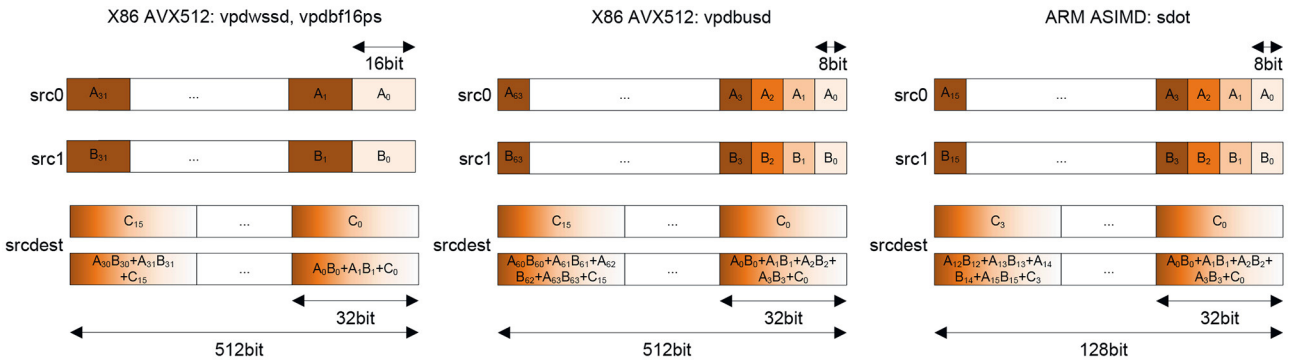


FIGURE 3 | Mixed-precision dot-product instructions, **Left**: 16 bit integer and bfloat16 on Intel AVX512, **Middle**: 8bit integer using Intel AVX512, **Right**: 8 bit integer using ARM ASIMD.

up to 16× more FLOPs/cycle compared to FP32 AVX512 FMA instructions on current Xeon platforms.

These considerably different GEMM microkernel variants highlight yet another aspect of the TPPs: The TPPs specify *what* needs to be done rather than how it is done/implemented. In this case, the user may just specify/employ a BRGEMM TPP in order to perform a tensor contraction, whereas the TPP backend/implementation is responsible for generating the optimal code for each platform at hand. In this methodology, all the architectural nuances are hidden completely by the user, and the same exact user code written in terms of TPPs may be reused across platforms with different characteristic/ISAs without sacrificing performance or portability.

3.2.2. Mixed Precision BRGEMM and Its Emulation

While the previous section presents the general structure of mapping matrix multiplication to various physical ISAs, this paragraph is used to demonstrate how the idea of a virtual ISA allows to implement operations efficiently which are not natively supported by a specific physical ISA. The example we are choosing here is our GEMM kernel and its support for bfloat16 and int8 on architectures which do not support these novel ISA SIMD-extension.

Before going into the details of the emulation, we first need to introduce special memory layouts which are used by x86 and aarch64 mixed-precision dot-product instructions as shown in **Figure 3**. As we can see in all cases (x86/aarch64 and bf16/int8), the overall concept is identical: Although doing mixed-precision and mixed-datatype-length computations, these instructions are functioning from a matrix multiplication point-of-view similar to 32 bit instructions. This is achieved by having an implicit 2-wide (BF16/int16) and 4-wide (int8) dot-product of A_i and B_i values leading to a horizontal summation per single 32 bit C_i, e.g., $C_0 = A_0 \cdot B_0 + A_1 \cdot B_1 + A_2 \cdot B_2 + A_3 \cdot B_3 + C_0$ as shown for the int8 variant. If we apply blockings with these instructions as discussed in **Figure 2-Left, Middle**, then we realize that matrix B is still read *via* 32-bit broadcast (containing 2 16-bit or 4 8-bit values along the inner-product or common dimension). However, matrix A is in need of reformatting. This is due to the fact that the GEMM kernel in **Figure 2-Left, Middle** requires full SIMD-width contiguous loads for optimal performance (which is along M and not K). Therefore, we need to reformat A into $[K^o][M][K^i]$ with $K^o \cdot K^i = K$ and $K^i = 2$ for 16-bit and $K^o = 4$ for 8-bit inputs. We refer to such a format as *VNNI-format* throughout this article. After such reformatting of A, we can perform full SIMD loads on A; combined with the 32-bit broadcast loads on B we have a 32-bit GEMM kernel which has a shorter K dimension, 2× for 16-bit datatypes and 4× for 8-bit datatypes.

In case these novel instructions are not available, especially for bfloat16 as this is a relatively new format, one might think, that an efficient mapping to a classic FP32 SIMD ISA is not possible. This is correct as long as the machine does not offer int16 support. However, with int16 support and SIMD masking we can implement the aforementioned non-trivial mixed-precision dot-product efficiently and even bit-accurately as shown in **Figure 4**. This is done by processing K^i in two rounds in the case of bfloat16 datatype. As shown in **Figure 4**, we first process the odd (or upper) bfloat16 number. This is done by exploiting the fact that a bfoat16 number perfectly aliases with an FP32 number in its 16 MSBs. Therefore, on AVX512 we can just execute a full SIMD load as a 16-bit-typed load with masking. As a mask we chose `0xaaaaaaaa` and as masking-mode we use zero masking. With this trick we automatically turn on-load the upper bfloat16 numbers in A into 16 valid FP32 numbers, and for B we broadcast and then perform an overriding register move. A little bit more work is needed for the lower/even bfloat16 number: In this case, we perform an unmasked load and then we use a 32-bit integer shift by 16 to create valid FP32 numbers. A simple inspection of the instruction sequence in **Figure 4** shows that we are mainly executing fused-multiply-add instructions with little overhead compared to a classic FP32 GEMM as illustrated in **Figure 2-Left, Middle**. Therefore, we can execute a bfloat16 GEMM with a reformatted matrix A with close to FP32-peak and still benefit from the smaller memory footprint (and, therefore, a small performance gain, as we will show later in section 6). Replacement sequences for int16 and int8 matrix inputs can be carried out in a similar way and their detailed discussion is skipped here.

```
__m512 c_0_0, c_0_1, c_0_2, c_0_3;
...
__m512 c_5_0, c_5_1, c_5_2, c_5_3;
__m512 a_0, a_1, a_2, a_3;
__m512 b_0, b_1, b_2, b_3, b_4, b_5;
__mmask32 m_0 = 0xaaaaaaaa;
/* load C */
...
/* load upper part of A & B and FP32 FMAs*/
a_0 = _mm512_maskz_loadu_epi16( m_0, ptrA );
a_1 = _mm512_maskz_loadu_epi16( m_0, ptrA+32 );
a_2 = _mm512_maskz_loadu_epi16( m_0, ptrA+64 );
a_3 = _mm512_maskz_loadu_epi16( m_0, ptrA+96 );
b_0 = _mm512_set1_epi32( ptrB );
b_0 = _mm512_maskz_mov_epi16( m_0, b_0 );
c_0_0 = _mm512_fmadd_ps( a_0, b_0, c_0_0 );
c_0_1 = _mm512_fmadd_ps( a_1, b_0, c_0_1 );
c_0_2 = _mm512_fmadd_ps( a_2, b_0, c_0_2 );
c_0_3 = _mm512_fmadd_ps( a_3, b_0, c_0_3 );
b_1 = _mm512_set1_epi32( ptrB+ldb );
b_1 = _mm512_maskz_mov_epi16( m_0, b_1 );
c_1_0 = _mm512_fmadd_ps( a_0, b_1, c_1_0 );
c_1_1 = _mm512_fmadd_ps( a_1, b_1, c_1_1 );
c_1_2 = _mm512_fmadd_ps( a_2, b_1, c_1_2 );
c_1_3 = _mm512_fmadd_ps( a_3, b_1, c_1_3 );
b_2 = _mm512_set1_epi32( ptrB+2*ldb );
b_2 = _mm512_maskz_mov_epi16( m_0, b_2 );
c_2_0 = _mm512_fmadd_ps( a_0, b_2, c_2_0 );
c_2_1 = _mm512_fmadd_ps( a_1, b_2, c_2_1 );
c_2_2 = _mm512_fmadd_ps( a_2, b_2, c_2_2 );
c_2_3 = _mm512_fmadd_ps( a_3, b_2, c_2_3 );
b_3 = _mm512_set1_epi32( ptrB+3*ldb );
b_3 = _mm512_maskz_mov_epi16( m_0, b_3 );
c_3_0 = _mm512_fmadd_ps( a_0, b_3, c_3_0 );
c_3_1 = _mm512_fmadd_ps( a_1, b_3, c_3_1 );
c_3_2 = _mm512_fmadd_ps( a_2, b_3, c_3_2 );
c_3_3 = _mm512_fmadd_ps( a_3, b_3, c_3_3 );
```

```
b_4 = _mm512_set1_epi32( ptrB+4*ldb );
b_4 = _mm512_maskz_mov_epi16( m_0, b_4 );
c_4_0 = _mm512_fmadd_ps( a_0, b_4, c_4_0 );
c_4_1 = _mm512_fmadd_ps( a_1, b_4, c_4_1 );
c_4_2 = _mm512_fmadd_ps( a_2, b_4, c_4_2 );
c_4_3 = _mm512_fmadd_ps( a_3, b_4, c_4_3 );
b_5 = _mm512_set1_epi32( ptrB+5*ldb );
b_5 = _mm512_maskz_mov_epi16( m_0, b_5 );
c_5_0 = _mm512_fmadd_ps( a_0, b_5, c_5_0 );
c_5_1 = _mm512_fmadd_ps( a_1, b_5, c_5_1 );
c_5_2 = _mm512_fmadd_ps( a_2, b_5, c_5_2 );
c_5_3 = _mm512_fmadd_ps( a_3, b_5, c_5_3 );
/* load lower part of A & B and FP32 FMAs*/
a_0 = _mm512_loadu_epi16( ptrA );
a_1 = _mm512_loadu_epi16( ptrA+32 );
a_2 = _mm512_loadu_epi16( ptrA+64 );
a_3 = _mm512_loadu_epi16( ptrA+96 );
a_0 = _mm512_slli_epi32( a_0, 16 );
a_1 = _mm512_slli_epi32( a_1, 16 );
a_2 = _mm512_slli_epi32( a_2, 16 );
a_3 = _mm512_slli_epi32( a_3, 16 );
b_0 = _mm512_set1_epi32( ptrB );
b_0 = _mm512_slli_epi32( b_0, 16 );
c_0_0 = _mm512_fmadd_ps( a_0, b_0, c_0_0 );
c_0_1 = _mm512_fmadd_ps( a_1, b_0, c_0_1 );
c_0_2 = _mm512_fmadd_ps( a_2, b_0, c_0_2 );
c_0_3 = _mm512_fmadd_ps( a_3, b_0, c_0_3 );
b_1 = _mm512_set1_epi32( ptrB+ldb );
b_1 = _mm512_slli_epi32( b_1, 16 );
c_1_0 = _mm512_fmadd_ps( a_0, b_1, c_1_0 );
c_1_1 = _mm512_fmadd_ps( a_1, b_1, c_1_1 );
c_1_2 = _mm512_fmadd_ps( a_2, b_1, c_1_2 );
c_1_3 = _mm512_fmadd_ps( a_3, b_1, c_1_3 );
```

```
b_2 = _mm512_set1_epi32( ptrB+2*ldb );
b_2 = _mm512_slli_epi32( b_2, 16 );
c_2_0 = _mm512_fmadd_ps( a_0, b_2, c_2_0 );
c_2_1 = _mm512_fmadd_ps( a_1, b_2, c_2_1 );
c_2_2 = _mm512_fmadd_ps( a_2, b_2, c_2_2 );
c_2_3 = _mm512_fmadd_ps( a_3, b_2, c_2_3 );
b_3 = _mm512_set1_epi32( ptrB+3*ldb );
b_3 = _mm512_slli_epi32( b_3, 16 );
c_3_0 = _mm512_fmadd_ps( a_0, b_3, c_3_0 );
c_3_1 = _mm512_fmadd_ps( a_1, b_3, c_3_1 );
c_3_2 = _mm512_fmadd_ps( a_2, b_3, c_3_2 );
c_3_3 = _mm512_fmadd_ps( a_3, b_3, c_3_3 );
b_4 = _mm512_set1_epi32( ptrB+4*ldb );
b_4 = _mm512_slli_epi32( b_4, 16 );
c_4_0 = _mm512_fmadd_ps( a_0, b_4, c_4_0 );
c_4_1 = _mm512_fmadd_ps( a_1, b_4, c_4_1 );
c_4_2 = _mm512_fmadd_ps( a_2, b_4, c_4_2 );
c_4_3 = _mm512_fmadd_ps( a_3, b_4, c_4_3 );
b_5 = _mm512_set1_epi32( ptrB+5*ldb );
b_5 = _mm512_slli_epi32( b_5, 16 );
c_5_0 = _mm512_fmadd_ps( a_0, b_5, c_5_0 );
c_5_1 = _mm512_fmadd_ps( a_1, b_5, c_5_1 );
c_5_2 = _mm512_fmadd_ps( a_2, b_5, c_5_2 );
c_5_3 = _mm512_fmadd_ps( a_3, b_5, c_5_3 );
/* store C*/
...
```

FIGURE 4 | Emulation of a bit accurate GEMM kernel using AVX512F instructions matching a GEMM kernel as depicted in **Figure 2** using vdpbf16ps AVX512 instructions. The glossary contains detailed descriptions of the used intrinsic functions.

In addition to the presented emulation of mixed-precision GEMM kernels using SIMD instructions, we have also added support for emulation of Intel AMX instructions bit-accurately on AVX512. This addition enables running numerical accuracy experiments, such as convergence studies, before the release of a chip that supports Intel AMX instructions. A similar path is possible for ARM's SME instruction set and subject to future work. These emulation capabilities further highlight the aspect of TPP as a virtual tensor ISA.

3.3. Examples of Non-trivial Non-GEMM TPPs

The previous sections covered most of the TPP implementations: straightforward element-wise unary/binary/ternary operations and various forms of mixed precision GEMMs including their emulation on older hardware. However, there are cases in which we are not operating on the data in an element-wise fashion, e.g., transpose, or the Unary_op, Binary_op, or Ternary_op is not an elementary operation. The goal of this section is to shed some light on these cases by presenting the transpose TPP in detail, and sketching fast non-linear approximations on SIMD machines that match the accuracy requirements of deep learning applications.

3.3.1. Transform-Transpose TPP *via* Shuffle Networks

When working with matrices, the transpose kernel is ubiquitous. It is needed to access the matrix's elements in various contractions along the mathematically correct dimension. However, a transpose operation is scalar at first sight. In this subsection we exhibit how transpose can be implemented using shuffle networks in a fully vectorized fashion, e.g., **Figure 5** demonstrates how a 16×16 matrix with 256 32-bit elements can be transposed in 64 cycles using AVX512 instructions.

The shuffle-network presented in **Figure 5** is a blueprint for all datatype-lengths and ISAs: in $\log_2$ *SIMD-Length* stages we can transpose a matrix held in a set of SIMD registers. In this particular example, we need $\log_2 16 = 4$ stages and in each stage we increase the shuffling/interleaving width of logical elements, and also increase the distance at which we access the 32 registers grouped into two sets of 16 registers each. More specifically, we start with registers i_0 to i_{15} and interleave elements at the same position in a pair of registers close to each other. This constructs now pairs of 32 bit values in o_0 and o_1 which are already containing the transpose's result for 2 out of 16 elements and we repeat this for all other 7 input register pairs. The analogous transformation is now repeated in the second stage with 64-bit values and accessing o_0 and o_2 as input pair pattern. This constructs a new set output registers i_0 and i_1 which are holding the transpose's result at 128-bit granularity. After that, stage 3 is shuffling at 128-bit granularity on register pairs which have a distance of "4" and creates output registers that hold 256-bit of transposed data. Finally, in stage 4, these 256-bit transposed input registers are shuffled once again creating the final set of 16 register holding the transposed 16×16 matrix. For non-square

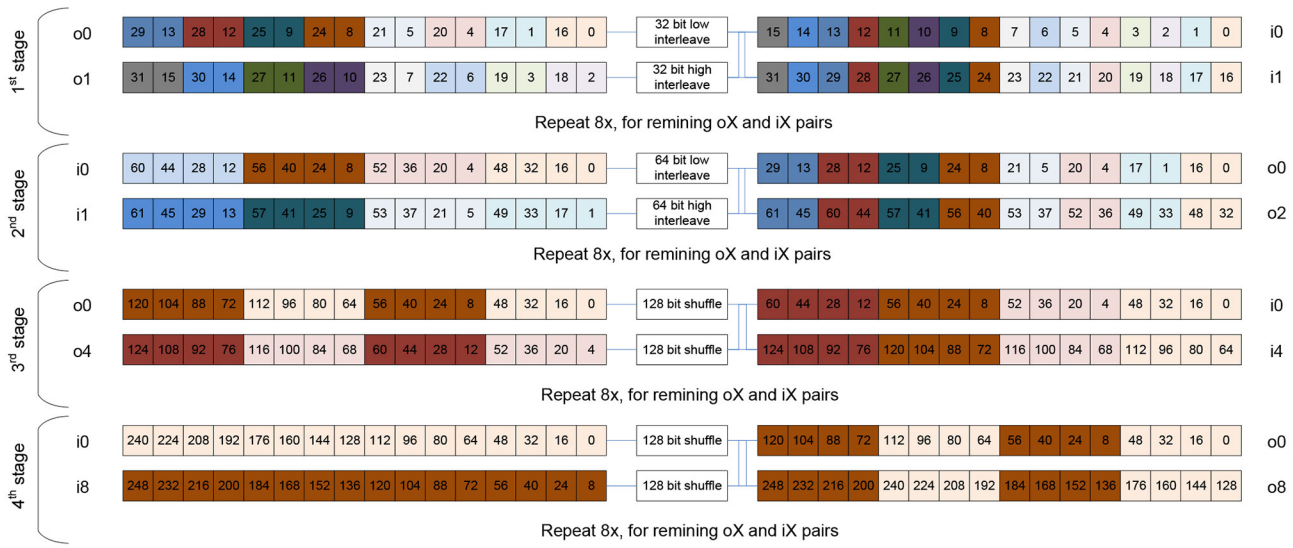


FIGURE 5 | Sketch of a shuffle network for a 32-bit transpose of a 16×16 matrix using Intel AVX512 instructions. *Via* four stages (each one having 16 independent shuffles that double in width per stage), the 16×16 matrix (256 elements) can be transposed with only 64 instructions and fully leverages the 32 architectural registers.

X86 Code

```
__m128 r0, r1, r2, r3, t0, t1, t2, t3;
r0 = _mm_loadu_ps(in + 0*ldi); // 0 1 2 3
r1 = _mm_loadu_ps(in + 1*ldi); // 4 5 6 7
r2 = _mm_loadu_ps(in + 2*ldi); // 8 9 10 11
r3 = _mm_loadu_ps(in + 3*ldi); // 12 13 14 15
t0 = _mm_unpacklo_ps(r0,r1); // 0 4 1 5
t1 = _mm_unpackhi_ps(r0,r1); // 2 6 3 7
t2 = _mm_unpacklo_ps(r2,r3); // 8 12 9 13
t3 = _mm_unpackhi_ps(r2,r3); // 10 14 11 15
r0 = _mm_unpacklo_pd(t0,t2); // 0 4 8 12
r1 = _mm_unpackhi_pd(t0,t2); // 1 5 9 13
r2 = _mm_unpacklo_pd(t1,t3); // 2 6 10 14
r3 = _mm_unpackhi_pd(t1,t3); // 3 7 11 15
_mm_storeu_ps(out + 0*ldo, r0);
_mm_storeu_ps(out + 1*ldo, r1);
_mm_storeu_ps(out + 2*ldo, r2);
_mm_storeu_ps(out + 3*ldo, r3);
```

ARM Code

```
float32x4_t r0, r1, r2, r3, t0, t1, t2, t3;
r0 = vld1q_f32(in + 0*ldi); // 0 1 2 3
r1 = vld1q_f32(in + 1*ldi); // 4 5 6 7
r2 = vld1q_f32(in + 2*ldi); // 8 9 10 11
r3 = vld1q_f32(in + 3*ldi); // 12 13 14 15
t0 = vtrn1q_f32(r0,r1); // 0 4 1 5
t1 = vtrn2q_f32(r0,r1); // 2 6 3 7
t2 = vtrn1q_f32(r2,r3); // 8 12 9 13
t3 = vtrn2q_f32(r2,r3); // 10 14 11 15
r0 = vtrn1q_f64(t0,t2); // 0 4 8 12
r1 = vtrn2q_f64(t0,t2); // 1 5 9 13
r2 = vtrn1q_f64(t1,t3); // 2 6 10 14
r3 = vtrn2q_f64(t1,t3); // 3 7 11 15
vst1q_f32(out + 0*ldo, r0);
vst1q_f32(out + 1*ldo, r1);
vst1q_f32(out + 2*ldo, r2);
vst1q_f32(out + 3*ldo, r3);
```

FIGURE 6 | Comparison of X86 and ARM code for a simple 4×4 single precision transpose using unpack instructions. The glossary contains detailed descriptions of the used intrinsic functions.

matrices we (a) just use masked loads or set registers to zero, (b) transpose the zeros as well, and then (c) do not store all registers or employ masked stores. This basic kernel is used as a basic building block to create large transpose operators by simply adding outer loops.

This algorithm can be implemented by any SIMD ISA which offers support for picking arbitrary values from a pair of SIMD registers to construct a result register containing values from the two sources, i.e., a general shuffler. However, "structured" shuffle instructions are adequate as shown in **Figure 6**. Both x86 and aarch64 offer instructions exactly implementing the needs for 32-bit and 64-bit interleaves as needed in the first two stages covered in the previous description. In the case of 128-bit-wide SIMD registers this is enough to carry out the entire transpose of 4 × 4 matrices as shown in **Figure 6**.

Finally, we want to note that broadcast loads, as supported by various ISAs, can be used to implement the first stage of the shuffle network. This has the advantage that one stage of the shuffle network can be executed faster and in parallel to the shuffler. The shuffle operations needed in all of these networks are relatively expensive in hardware, therefore modern CPUs often only provide one execution unit for such operations (such "shuffle-viruses" like transposes are pretty rare in general code). However, broadcasts on the load path are cheap and can run in parallel to the shuffle unit, hence the overall performance of the transpose operation improves. This microkernel variation leads

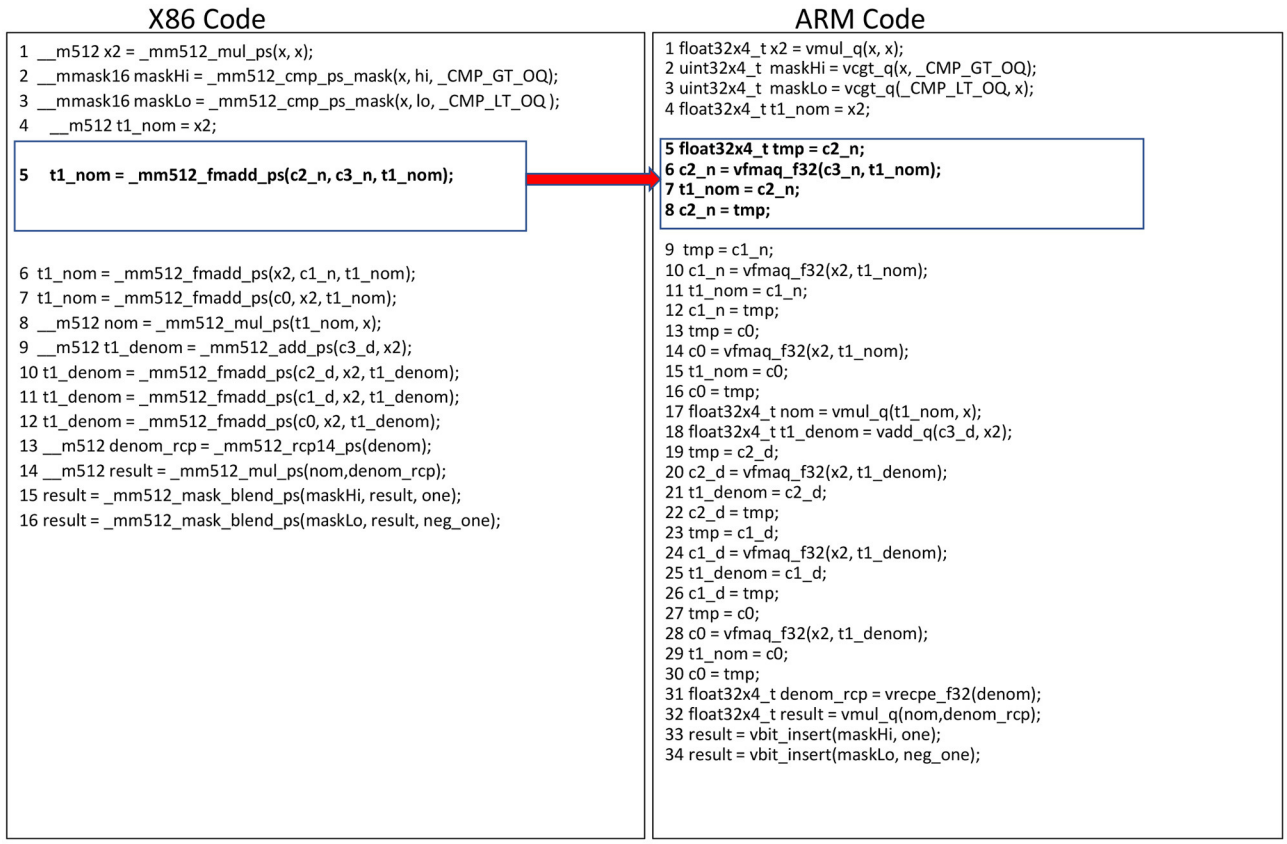


FIGURE 7 | Rational Padé 7/8 tanh approximation pseudocode with equivalent intrinsics on x86 and Arm/AArch64. We highlight here how the FMADD instruction on x86 ISAs has an equivalent instruction sequence on AArch64.

to relatively complex code, and as such we skip its presentation. However our TPP implementation back-end employs all these microkernel variations.

3.3.2. Approximations for Non-linear TPP Activation Functions

Activation functions are used to represent non-linear behavior of neural networks. Popular known activation functions are sigmoid, tanh and Gaussian Error Linear Unit (GELU). These activation functions can be approximated to increase the efficiency of deep learning networks without effecting its non-linear characteristics. In this section, we will discuss different approximation techniques based on Padé rational polynomials, piecewise minimax polynomials and Taylor expansions, along with their TPP implementation on different ISAs. For simplicity we present the relevant algorithms in terms of x86 and arm intrinsics (see glossary for the semantics of these intrinsics), however the actual TPP implementation relies on JIT code generation.

3.3.2.1. Rational Padé Polynomials

The Padé approximation of a function f is the ratio of two polynomials with degrees p and q:

$$Padé_{[p/q]f}(x) = \frac{\sum_{i=0}^{p} a_i x^i}{\sum_{i=0}^{q} b_i x^i}$$

The coefficients a_i and b_i can be calculated by considering the first $p + q$ derivatives of f at zero and solving the corresponding system of equations:

$$\begin{aligned} f(0) &= Padé_{[p/q]f}(0) \\ f'(0) &= Padé'_{[p/q]f}(0) \\ &\vdots \\ f^{(p+q)}(0) &= Padé^{(p+q)}_{[p/q]f}(0) \end{aligned}$$

As an example we consider the approximation of the tanh function which has two asymptotes, hence approximating it with a Taylor expansion of lower degree polynomials may not yield good results. The implementation of the $Padé_{[7/8]}(x)$ tanh approximation is shown in **Figure 7**. FMA operations are used to compute the numerators and denominators *via* Horner's rule. The reciprocal of the denominator is multiplied by the numerator to get the final result. The accuracy of reciprocal instruction is different among different CPU's. This difference in accuracy does

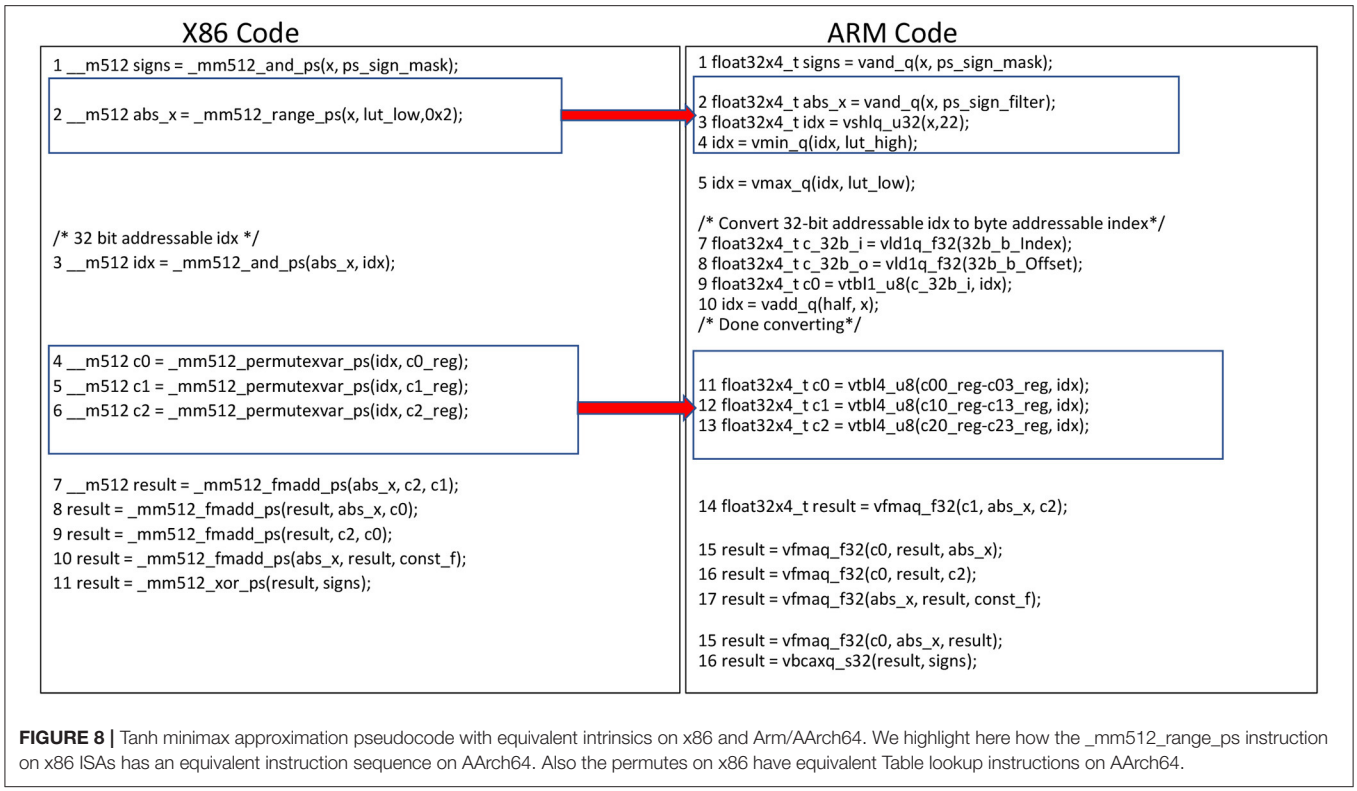


FIGURE 8 | Tanh minimax approximation pseudocode with equivalent intrinsics on x86 and Arm/AArch64. We highlight here how the _mm512_range_ps instruction on x86 ISAs has an equivalent instruction sequence on AArch64. Also the permutes on x86 have equivalent Table lookup instructions on AArch64.

not affect the non-linear region of the tanh function, keeping the TPP behavior same across different CPU's. The sigmoid activation function can be approximated *via* tanh by leveraging the following identity:

$$sigmoid(x) = (\tanh(x/2) + 1)/2$$

3.3.2.2. Piecewise Minimax Polynomial Approximations

In this section, we discuss the minimax polynomials approach [28] with the truncated Chebyshev series [29] for approximations of activation functions. In this approach, the input range of a function $f(x)$ is divided into intervals and for each interval $[a, b]$ we find a polynomial p of degree max n to minimize:

$$\max_{a \le x \le b} |f(x) - p(x)|$$

We approximate tanh and GELU activation functions using this approach in our TPP implementation. The input range is divided into 16 intervals and for each interval we investigate a polynomial p of 3^{rd} degree (i.e., we find appropriate p's coefficients c0, c1, c2 based on the minimized absolute maximum difference of f and p). **Figure 8** shows the x86 and arm implementation of evaluating such minimax polynomials. The register index (idx) is calculated using the exponent and Most Significant Bit (MSB) of the respective input values, and represents the 16 intervals where the input values are located. The range intrinsic _mm512_range_ps(A,B) is used to generate the register index (idx) on AVX512 platforms (**Figure 8-Left**, line 2). In ARM, the range functionality is emulated with equivalent and, shlq, min and max instructions as shown in **Figure 8-Right**, lines 2–4. To evaluate the 3^{rd} degree polynomial we need to locate 3 coefficients (c0,c1,c2) based on the values at the register index (idx), which holds 16 entries. We use 3 look up operations to find the three coefficients, each involving 16 FP32 entries. The 512-bit register length in AVX512 is sufficient to hold 16 coefficients required for each look up, resulting in using 3 registers for 3 look up operations (see **Figure 8-Left**, lines 4–6). Each ARM 128-bit wide vector register can only hold 4 FP32 entries, subsequently we are using 12 vector registers to hold the 16 entries for all 3 coefficients of the polynomial. The in-register look-up table is performed using _mm512_permutexvar_ps(A,B) instructions in x86 AVX512 as shown in **Figure 9**. In ARM we have byte addressable table look up instructions which are analogous to 32-bit addressable permutes instructions in x86. Hence, we need to convert the 32-bit addressable (0–16) register indexes to byte addressable (0-64 bytes) indexes. In order to do that, we use a constant register A with a table look up instruction to duplicate the register index (idx) to each byte in the 32-bit entry. A constant offset (0,1,2,3) is added to the duplicated byte index to get the byte addressable index for each FP32 entry in 16 FP32 entries (**Figure 8-Right**, lines 7–9). The table look up instruction in ARM provides the 64 byte look up capability, which is sufficient enough to search into 4 registers holding the 16 entries of each coefficient; we are using the generated byte indexes as shown in **Figure 10**. Finally, 4 FMA operations are used to evaluate the polynomial using Horner's rule. The FMA instruction in x86 provides the user the flexibility to decide among the sources to

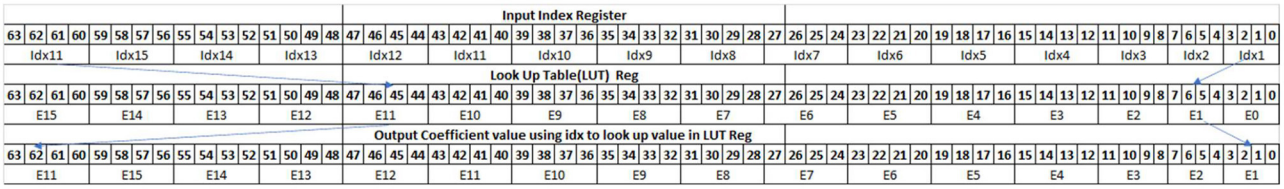


FIGURE 9 | 32Bit addressable Table look up setup on x86 AVX512 platforms.

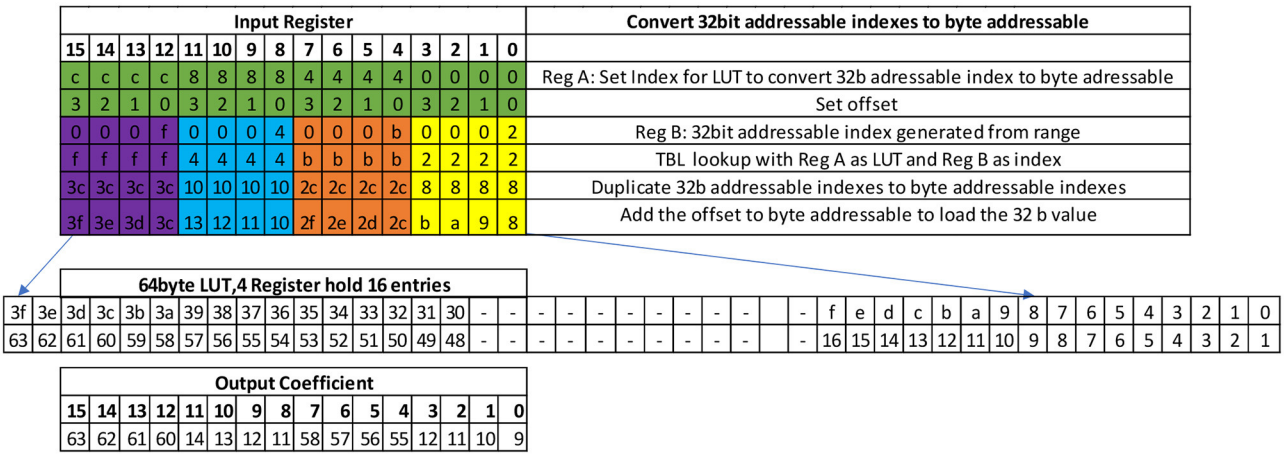


FIGURE 10 | Byte addressable table look up setup in ARM/AArch64. We highlight the conversion of 32bit indexes to byte indexes and the use of byte indexes to get the coefficients in 16 FP32 intervals.

destroy and the ones to preserve. ARM requires mov instructions to save intermediate results in order to avoid the data overwriting during FMA operations.

3.3.2.3. Approximation With Taylor Series

As an example of approximation with Taylor series we illustrate here the exp() activation function. The e^x is approximated using the identity $e^x = 2^{x \log_2 e} = 2^{n+y} = 2^n \cdot 2^y$ with $n = round(x \log_2 e)$ and $y = x \log_2 e - n$. We need to calculate 2^n with n being an integer and the term 2^y with $|y| \in [0, 1)$. A Taylor polynomial of third degree is used to calculate the term 2^y with 3 FMA instructions (see **Figure 11-Left**, lines 4–6). Once 2^y is calculated, we leverage the instruction `_mm512_scalef_ps(A,B)` which returns a vector register holding $a_i \cdot 2^{floor(b_i)}$ for each $a_i \in A$ and $b_i \in B$. This scale instruction concludes the exp() approximation on x86 with AVX512. On ARM we calculate 2^n and 2^y with equivalent replacement instructions as shown in **Figure 11**.

4. TPP MATRIX EQUATIONS

One of the main design principles of TPPs (as described in section 2.1) is that they can be composed in a producer-consumer fashion to form complex operations. For example consider the scenario where a user wants to implement the composite operation $C = Tanh(A + B)$. One way to express this *via* TPPs would be to allocate an intermediate tensor *tmp* with same shape as A and B, and perform first $tmp = Add(A, B)$ *via* the binary Add TPP. Then the user can compute the final result by leveraging the Tanh Unary TPP: $C = Tanh(tmp)$. Even though this approach is functionally correct, it requires the explicit management of intermediate tensors/buffers by the user and also may result in low performance since there are redundant loads/stores to the *tmp* tensor.

In order to increase the productivity, efficiency and expressiveness pertaining to composite operators, we implemented an embedded Domain Specific Language (eDSL) in LIBXSMM [22]. Our Proof-Of-Concept implementations allows the user to express the desired composite operator as a Matrix Equation. More specifically, the user can express the composite operator as an equation tree, where the head and internal nodes are the available TPPs, whereas the leaves of the tree are the input 2D tensors of the composite operation. In the next subsections, we describe in detail the methodology we employ for JITing matrix equations of TPPs.

4.1. Definitions and Notations for TPP Matrix Equations

A TPP matrix equation is represented as a tree with unary/binary/ternary TPP operations as internal nodes and

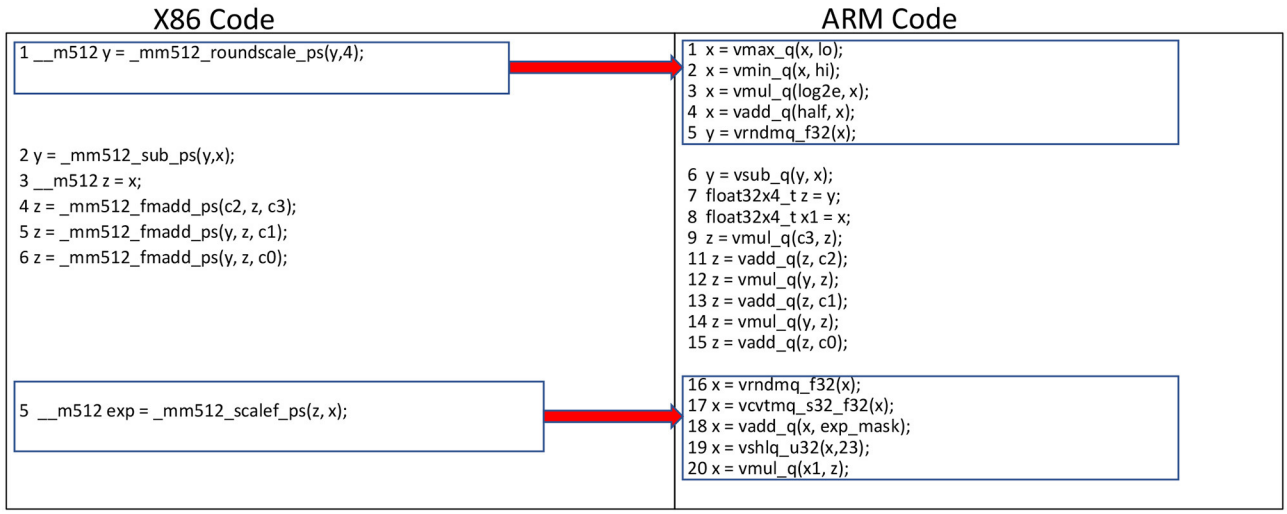


FIGURE 11 | Pseudocode for e^x approximation with Taylor series on AVX512 x86 and ARM.

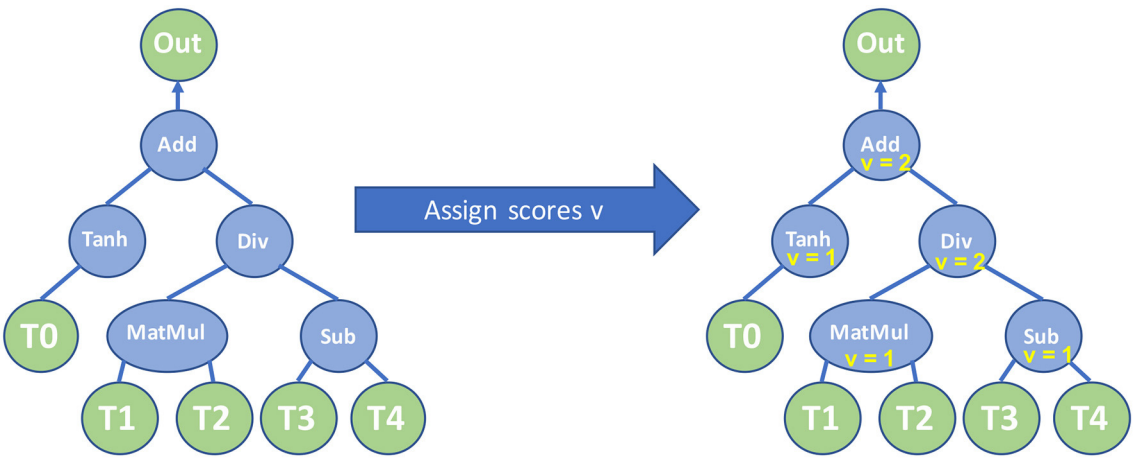


FIGURE 12 | Left: TPP Equation tree for $Out = Tanh(T_0) + (T_1 \times T_2)/(T_3 - T_4)$. **Right:** Assigned register scores v on the equation TPP nodes after running **Algorithm 3**.

the equation's input tensors are the leaves of the tree. The inputs of a TPP tree node are essentially its children in the equation tree. The output of an internal TPP node can be represented as a *temporary* intermediate tensor which in turn can be fed as input to the parent TPP node in the tree. Depending on the TPP node type (unary/binary/ternary), each internal node requires a number of inputs (one/two/three) to be computed/ready before performing the corresponding TPP operation. Let's consider for example the TPP equation tree in **Figure 12-Left** that is used to express the following operator:

$$Out = Tanh(T_0) + (T_1 \times T_2)/(T_3 - T_4) \quad (1)$$

We will illustrate with this example how our eDSL for TPP Matrix Equations works.

4.2. Optimized Execution Plan for TPP Matrix Equations

The equation tree in **Figure 12-Left** can be naively evaluated by assigning to each intermediate node a temporary tensor to hold the corresponding TPP output, and performing, e.g., (1) the Tanh operation, (2) the Matrix Multiplication, (3) the Subtract operation, (4) the Div operation, and finally (5) the Add TPP. In such an evaluation schedule, we would need 4 intermediate tensors to hold the corresponding intermediate results. In this subsection, we illustrate how we can construct optimized execution plans for TPP Matrix Equations that minimize the number of intermediate tensors.

For each TPP node r we can assign a *register score* value v_r that essentially dictates how many temporary/intermediate tensors are required to calculate the subtree in the equation where

Algorithm 3 | Assign_Register_Score(r).

Input: TPP equation tree with root node r
Output: TPP equation tree with assigned register score values on its nodes

if is_Leaf(r) **then**
  $v_r \leftarrow 0$
if r is **unary TPP then**
  Assign_Register_Score(Left_Child(r))
  ▷ *If child is leaf, then we assign current register score of 1, else we assign the child's register score*
  if is_Leaf(Left_Child(r)) **then**
    $v_r \leftarrow 1$
  else
    $v_r \leftarrow$ Register_Score(Left_Child(r))
if r is **binary TPP then**
  Assign_Register_Score(Left_Child(r))
  Assign_Register_Score(Right_Child(r))
  ▷ *If the register scores of children are equal, then we get the children's register score increased by one, otherwise we get the max value of the children's register score*
  if Register_Score(Left_Child(r)) **equals** Register_Score(Right_Child(r)) **then**
    $v_r \leftarrow$ Register_Score(Left_Child(r)) + 1
  else
    $v_L \leftarrow$ Register_Score(Left_Child(r))
    $v_R \leftarrow$ Register_Score(Right_Child(r))
    $v_r \leftarrow$ MAX(v_L, v_R)
if r is **ternary TPP then**
  Assign_Register_Score(Left_Child(r))
  Assign_Register_Score(Middle_Child(r))
  Assign_Register_Score(Right_Child(r))
  ▷ *If all children are leaves, then we assign current register score of 1. Otherwise, in a pairwise fashion we consider the register scores of the children in order of increasing value.*
  if is_Leaf(Left_Child(r)) **AND** is_Leaf(Middle_Child(r)) **AND** is_Leaf(Right_Child(r)) **then**
    $v_r \leftarrow 1$
  else
    $v_L \leftarrow$ Register_Score(Left_Child(r))
    $v_M \leftarrow$ Register_Score(Middle_Child(r))
    $v_R \leftarrow$ Register_Score(Right_Child(r))
    $v_0, v_1, v_2 \leftarrow$ Sort_Increasing_Order(v_L, v_M, v_R)
    if v_2 **equals** v_1 **then**
      $v_{tmp} \leftarrow v_2 + 1$
    else
      $v_{tmp} \leftarrow v_2$
    if v_{tmp} **greater than** $v_0 + 1$ **then**
      $v_r \leftarrow v_{tmp}$
    else
      $v_r \leftarrow v_{tmp} + 1$

node r is root. We extend the methodology of Flajolet et al. [30] and we generate the register score values using the recursive **Algorithm 3**. This algorithm calculates recursively the register

Algorithm 4 | Create_Execution_Plan(r).

Input: TPP equation tree with root node r and assigned register score values on its nodes
Output: TPP equation tree with assigned traversal timestamps t and temporary tensor ids tmp

if is_Leaf(r) **then**
  return
if r is **unary TPP then**
  Create_Execution_Plan(Left_Child(r))
  $t_r \leftarrow$ global_timesteamp++
  ▷ *If child is leaf, reserve a new tmp, else re-use tmp from child*
  if is_Leaf(Left_Child(r)) **then**
    $tmp_r \leftarrow$ Reserve_Tmp()
  else
    $tmp_r \leftarrow$ tmp_Left_Child(r)
if r is **binary TPP then**
  ▷ *Recursively visit children in order of decreasing register score*
  Create_Execution_Plan(Child_Max_Register_Score(r))
  Create_Execution_Plan(Child_Min_Register_Score(r))
  $t_r \leftarrow$ global_timesteamp++
  ▷ *If all children are leaves, reserve a new tmp, else re-use the tmp from a non-leaf child and recycle the tmp of the other non-leaf child*
  if is_Leaf(Left_Child(r) **AND** is_Leaf(Right_Child(r))) **then**
    $tmp_r \leftarrow$ Reserve_Tmp()
  else
    if not_Leaf(Left_Child(r) **then**
      $tmp_r \leftarrow$ tmp_Left_Child(r)
      Recycle_Tmp(tmp_Right_Child(r))
    else
      $tmp_r \leftarrow$ tmp_Right_Child(r)
      Recycle_Tmp(tmp_Left_Child(r))
if r is **ternary TPP then**
  ▷ *Recursively visit children in order of decreasing register score*
  Create_Execution_Plan(Child_Max_Register_Score(r))
  Create_Execution_Plan(Child_Mid_Register_Score(r))
  Create_Execution_Plan(Child_Min_Register_Score(r))
  $t_r \leftarrow$ global_timesteamp++
  ▷ *If all children are leaves, reserve a new tmp, else re-use the tmp from a non-leaf child and recycle the tmps of the other non-leaf children*
  if is_Leaf(Left_Child(r)) **AND** is_Leaf(Middle_Child(r)) **AND** is_Leaf(Right_Child(r)) **then**
    $tmp_r \leftarrow$ Reserve_Tmp()
  else
    if not_Leaf(Left_Child(r) **then**
      $tmp_r \leftarrow$ tmp_Left_Child(r)
      Recycle_Tmp(tmp_Middle_Child(r)), Recycle_Tmp(tmp_Right_Child(r))
    else
      if not_Leaf(Right_Child(r) **then**
        $tmp_r \leftarrow$ tmp_Right_Child(r)

```
42:            Recycle_Tmp(tmp_Middle_Child(r)),
    Recycle_Tmp(tmp_Left_Child(r))
43:        else
44:            tmp_r ← tmp_Middle_Child(r)
45:            Recycle_Tmp(tmp_Left_Child(r)),
    Recycle_Tmp(tmp_Right_Child(r))
```

scores of the children for a given node r, and in this way we know how many temporary tensors are required for the evaluation for each child. Now, if all of its children have the same register score, the node r get an increased register score value, otherwise the node gets as register score the maximum of its children's register score values. Intuitively this means that we can first evaluate a child c and its subtree with whatever intermediate tensor requirements it has, e.g., v_c temporary tensors, and eventually we need only one temporary tensor to hold c's output. We can do the same afterwards for all other siblings of c, however, we can reuse/recycle the rest $v_c - 1$ temporary tensors that were required by c since c and its subtree have been already computed.

This algorithm optimizes the number of temporary tensors/storage that are required for the equation evaluation, and it reuses the temporary storage as much as possible. For instance, for the equation in **Figure 12-Left**, after executing **Algorithm 3** on the TPP equation tree, we see that the root's register score value is 2 (see **Figure 12-Right**), meaning that only 2 intermediate tensors are required to evaluate the entire TPP tree rather than naively assigning one temporary tensor to each internal TPP node which would result in 4 intermediate tensors.

Now that we have assigned the register scores for each node we can devise an execution plan for the TPP equation tree that minimizes the number of required intermediate tensors. **Algorithm 4** recursively creates such an optimal execution plan and essentially it calculates: (1) the order/traversal timestamps t with which the TPP equation nodes have to be evaluated, and also (2) assigns to each intermediate node r a temporary tensor id tmp_r that holds the intermediate resulting tensor of that TPP node. **Figure 13-Right** shows the optimized execution plan by applying **Algorithm 4** on our example equation. This algorithm recursively visits/evaluates the children of a node r in order of decreasing register score value. This means that the child/subtree with the maximum register score value is evaluated first, one of the temporary tensors is dedicated to hold that child's intermediate output, whereas the remaining temporary tensors can be reused for the evaluation of the siblings/subtrees, which per definition/order of traversal, require less or equal number of intermediate tensors. Such a strategy guarantees that the temporary tensors are optimally reused/recycled, and as a result we can leverage the minimum required temporary tensors for the evaluation of the entire equation TPP tree. For simplicity in our description, we assumed that all intermediate temporary tensors have the same size, however, our implementation considers the actual sizes of the intermediate output tensors and takes the maximum one as representative size for all temporary tensors.

4.3. Implementation of Optimized Execution Plan for TPP Matrix Equations

By employing **Algorithm 4**, we can devise an optimal execution plan for the TPP Matrix equation, and, here, we describe the implementation of such a plan. We consider three implementation strategies:

- *Strategy 1*: Using stack-allocated buffers as intermediate temporary tensors.
- *Strategy 2*: Using vector-register blocks as intermediate temporary tensors.
- *Strategy 3*: Hybrid implementation where some intermediate temporary tensors are stack-allocated buffers and some are vector-register blocks.

So far in our description, we have used the abstract notation "temporary tensor" without specifying how such a temporary tensor is instantiated in the implementation. The exact instantiation of a temporary/intermediate tensor is the differentiation factor among the 3 implementation strategies for the TPP matrix equations.

Strategy 1 considers each intermediate tensor as a physical buffer, and our TPP equation implementation allocates on the stack some space/buffer for each temporary tensor. Then, by following the timestamp order of the optimal execution plan (e.g., see **Figure 13-Right**), we emit/JIT the corresponding TPP code (e.g., see **Algorithms 1** and **2**) where the input tensors might be either the equation's input buffers provided by the user, or one of the stack allocated buffers representing an intermediate result. The fact that we have minimized the number of intermediate temporary buffers/tensors is critical for performance since these stack-allocated buffers may remain in some level of cache. Such a strategy is generic and can be leveraged to implement arbitrary equations. However, Strategy 1 may suffer from store-to-load forwarding inefficiencies on modern processors. Additionally, some of the intermediate tensors may spill from cache (e.g., when the intermediate outputs exceed the corresponding cache capacity) which would make the communication of temporary tensors among TPP nodes *via* loads/stores from/to stack allocated buffers quite expensive.

Strategy 2 considers each intermediate tensor as an $r_m \times r_n$ vector-register block. For example, on an AVX512 platform with 32 512-bit wide registers we have available 2 KBytes of register file that may be used for intermediate tensors. Each one of such 512-bit wide vector registers can hold 16 single-precision values and by stacking, e.g., 4 of these we can form a logical 16×4 intermediate tensor and in total we have available $32/4 = 8$ of such intermediate tensors that could be used by the equation. In Strategy 2, we block the computation of the equation's output in blocks with size $r_m \times r_n$, and we can calculate the corresponding $r_m \times r_n$ output by following the timestamp order of the optimal execution plan. We emit/JIT the corresponding TPP code for sub-tensors with size $r_m \times r_n$ where each intermediate output tensor is the assigned temporary vector-register block. Essentially this strategy performs vertical register fusion within the equation TPP nodes and incurs *no* communication *via* loads/stores from/to stack allocated buffers.

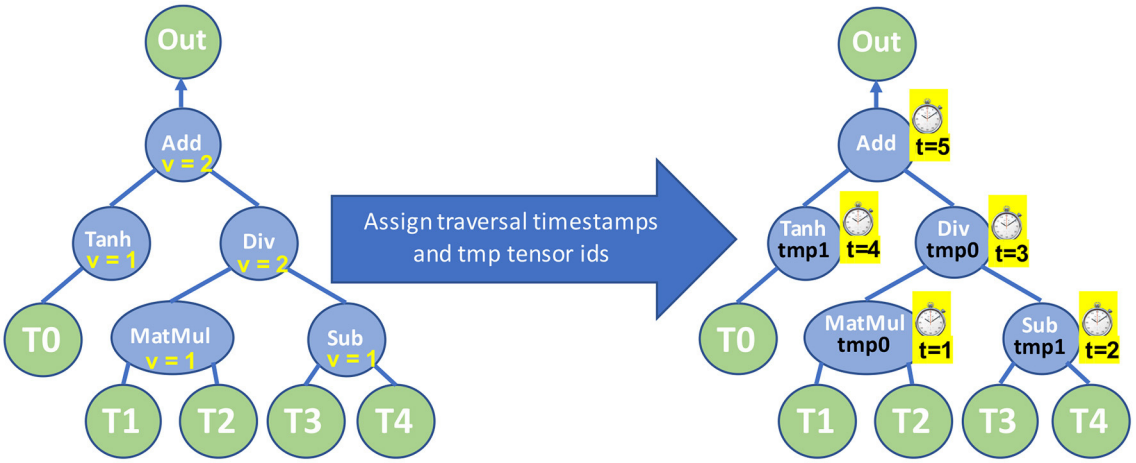


FIGURE 13 | Left: TPP equation tree with assigned register scores *v* on the nodes. Right: TPP equation tree with assigned traversal timestamps *t* and temporary tensor ids *tmp* after executing **Algorithm 4**.

However, such a methodology is limited by the number of available vector registers on each platform.

Strategy 3 combines the strengths of Strategies 1 and 2 by considering some intermediate tensors as stack-allocated buffers and some intermediate tensors as vector-register blocks. As such, in Strategy 3 the TPP operations/subtrees which exhibit *both* high register pressure and reuse (e.g., transposes, GEMM/BRGEMM, transcendental approximations), propagate the intermediate results toward the rest of the TPPs in the tree *via* stack-allocated temporal tensors. On the other hand, TPP subtrees without large register pressure are implemented using Strategy 2 that employs vertical register fusion and avoids loads/stores from/to stack-allocated buffers.

In addition to the aforementioned 3 strategies, in the TPP equation back-end we identify idioms/motifs of combined TPPs (e.g., a gather TPP followed by a reduce TPP) and we JIT an instruction sequence which is optimal for the composite access pattern. In section 5.1.5, we show an example of such a combined TPP motif that is optimized by the TPP backend.

Even though we developed a rudimentary method/POC of combining the TPPs *via* Matrix Equation Trees, we have found that it is sufficient to express all the complex operators we encountered in a wide-range of workloads discussed further in section 5. Nevertheless, we envision that when/if TPPs are widely adopted within Tensor Compiler frameworks (e.g., as an MLIR dialect) then more complicated Graphs (instead of simple trees) and more sophisticated analyses/optimization passes can be leveraged during the composition of TPPs. The key-ingredient that makes the composition of TPPs amenable to optimization opportunities is the TPP specification itself: TPPs comprise a small, well-defined compact set of tensor operators with declarative semantics as shown in section 2.

We would like also to highlight one use-case of Matrix Equations that can be beneficial for specialized DL accelerators. The BRGEMM TPP described in section 3.2 corresponds to an output-stationary flow that is suitable for CPUs and GPUs. Given an accelerator that favors, e.g., *A*-stationary GEMM formulations, one could express the following Matrix Equation: internal nodes G_i would be GEMM ternary TPPs, for each GEMM node G_i we would have the same input leaf A and a varying input B_i, and the output of each node would be a result C_i. Essentially this formulation dictates an *A*-stationary flow, and the back-end could optimize accordingly for the specific accelerator.

5. TPP-BASED KERNELS AND WORKLOADS

This section covers how DL kernels and workloads (image processing, recommendation systems, natural language processing, graph processing, and applications in science) can leverage TPPs to achieve high performance. Although this article's work is targeting CPUs, we cover the entire training pipeline and not only inference. The main purpose of this is to demonstrate the versatility of TPPs which is valuable in the more complicated backward pass kernels, and to handle training's implications to the forward pass.

5.1. TPP-Based Kernels

5.1.1. Softmax Kernel

Figure 14 illustrates two Matrix Equation trees that are used to express the softmax operator [31]:

$$Y = \text{softmax}(X) \text{ with } y_{ij} = \frac{e^{\left(x_{ij} - \max_{x_{ij} \in X} x_{ij}\right)}}{\sum_{x_{ij} \in X} e^{\left(x_{ij} - \max_{x_{ij} \in X} x_{ij}\right)}} \quad (2)$$

Equation 2 shows the formula for the softmax operator [31], which is often used as the last activation function of a neural network, aiming to normalize its output to a probability distribution. We can represent this operator *via* two TPP equation trees illustrated in **Figure 14**. The left tree computes the

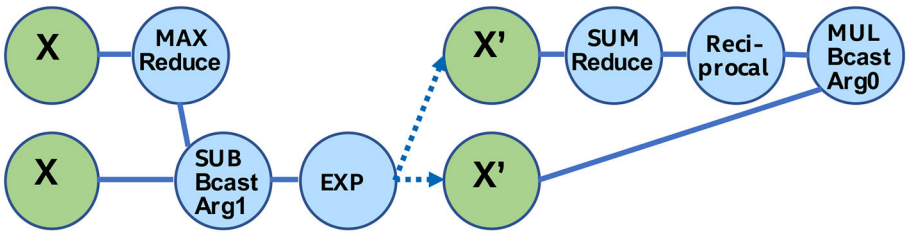


FIGURE 14 | Softmax operator by combining TPPs.

nominator of Equation 2: first the maximum value of the input tensor X is found (*via* the max-reduce TPP), then we subtract this max value from each entry of X (note the broadcast semantics in the second argument of the subtraction TPP), and a new tensor X' is computed by calculating the element-wise exponent on the earlier subtraction's outcome. Finally, in the right TPP tree each entry of the tensor X' is normalized by the sum of all values in X' to obtain the softmax output, a tensor Y. This example illustrates the expressiveness of the TPP abstractions, since the components of the mathematical formula map to TPPs in a straightforward way. At the same time, this example highlights the separation of concerns: the user does not need to worry about the efficient implementation of this equation on each different platform, since the TPP back-end is responsible for optimized code generation which is target-specific (contrary to the TPP expression itself which is platform-agnostic).

5.1.2. Normalization Kernels

Batch normalization (batchnorm) is a technique [32] that normalizes neuron layer input tensors to improve the overall training process. Batchnorm removes the need for careful parameter initialization and reduces the required training steps [32] in the neural networks. The batchnorm computations can be divided in two stages: (i) First the mean and variance of the input tensor are computed across the "batch" dimension: $\mu_j = \sum_{i=0}^{n-1} x_{ij}$, $\sigma_j^2 = \frac{1}{n}\sum_{i=0}^{n-1}(x_{ij} - \mu_i)^2$ where i is the "batch" dimension and j is the "feature" dimension, (ii) then the tensor entries x_{ij} are normalized based on μ and σ: $x'_{ij} = (x_{ij} - \mu_j)/(\sqrt{\sigma_j^2 + \epsilon})$.

Depending upon the workload, different TPPs and TPP equations can be employed to implement the batchnorm. Here, we take an example of batchnorm on a ResNet50 [33] convolution layer tensor X. The input tensor X has a four-dimensional shape of {N, C, H, W} with dimensions of batch (N), feature (C), height (H), and width (W). We first use sum-reduce TPPs on H and W dimensions to compute the sum ($m[N, C]$) and the sum of squared elements ($v[N, C]$) matrices. Subsequently, we use binary add TPPs across the batch dimension of $m[N, C]$ and $v[N, C]$ matrices for eventual computation of mean ($\mu[C]$) and variance ($\sigma^2[C]$) vectors. In the next step, we use a scaling equation to normalize each element of the input tensor. The scaling equation $Y = (m' * X + v') * G + B$ converts the input tensor X into a normalized tensor Y. Here, $G[C]$ and $B[C]$ are scaling vector inputs to batchnorm, and $m'[C]$ and $v'[C]$ are intermediate vectors that are computed from mean and variance vectors. We implement the scaling equation by a single TPP equation containing two FMADD ternary TPPs. The second equation tree of **Figure 15** shows an analogous scaling equation implementation. However, for this particular implementation, we broadcast m', v', G, B vectors into H, W, and N dimensions inside the TPP equation tree. An efficient implementation of batchnorm uses blocking on the C, H, and W dimensions along with multi-threading on the N and feature block dimension. We do not show the details of this implementation for sake of simplicity.

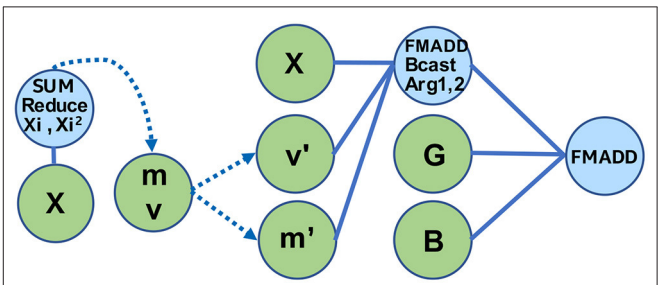


FIGURE 15 | Layernorm *via* TPPs.

Layer normalization (layernorm) [34] is a technique that normalizes the neurons *within* a layer, and was motivated by the limitations of Batch Normalization [32] in Recurrent Neural Networks. The layernorm computations can be divided in two stages: (i) First the mean and variance of the input tensor are computed across the "feature" dimension: $\mu_i = \sum_{j=0}^{m-1} x_{ij}$, $\sigma_i^2 = \frac{1}{m}\sum_{j=0}^{m-1}(x_{ij} - \mu_i)^2$ where i is the batch dimension and j is the "feature" dimension, (ii) then the tensor entries x_{ij} are normalized based on μ and σ: $x'_{ij} = (x_{ij} - \mu_i)/(\sqrt{\sigma_i^2 + \epsilon})$. Depending on the workload (e.g., attention cell in BERT), the scaled tensor may be further scaled with two other tensors γ and β. **Figure 15** illustrates two TPP equation trees that implement this composite layernorm operator. The left equation is using the sum-reduce TPP to compute the sum and sum of squared elements of the input tensor, namely m and v. These two scalars are combined (not shown in the equation for simplicity), and are fed as inputs to the right TPP tree, where the FMADD ternary TPP is used to scale the input tensor X. Finally, a cascading FMADD ternary TPP computes the final result *via* the scaling tensors G and B. We illustrate this layernorm *via* means of TPPs since all DL

norming layers essentially exhibit similar computational motif, and this specific norm is used in the BERT workload described in section 5.2.3.

Group normalization (groupnorm) [35] is a technique that normalizes the neurons within a group of features. Groupnorm was proposed as an alternative to batchnorm [32] to reduce normalization error for smaller batch sizes. In groupnorm, features are divided into groups, and mean and variance are computed within each group for normalization. Groupnorm is also a generalization of the layer normalization [34] and instance normalization [36] approach. Layernorm is groupnorm with a single group, and instance norm is groupnorm with group size equal to one. Groupnorm can be implemented with the same set of TPPs and TPP equations that were used in the batchnorm kernel. We again take the example of ResNet50 [33] convolution layer tensor X and apply groupnorm on it with g number of groups. We can ignore the batch dimension (N) for this discussion as groupnorm works independently upon each batch. Therefore, the input tensor X now has a three-dimensional shape of {C, H, W} with dimensions of feature (C), height (H), and width (W). We first use sum-reduce TPPs on H and W dimensions to compute the sum ($m[C]$) and the sum of squared elements ($v[C]$) vectors. Subsequently, we add $m[C]$ and $v[C]$ values within a feature group for eventual computation of group mean ($\mu[g]$) and group variance ($\sigma^2[g]$) vectors. Similar to batchnorm, we use a scaling equation to normalize each element of the input tensor. The scaling equation $Y = (m' * X + v') * G + B$ converts input tensor X into a normalized tensor Y. Here, $G[C]$ and $B[C]$ are scaling vector inputs to groupnorm, and $m'[C]$ and $v'[C]$ are intermediate vectors that are computed from group mean and group variance vectors. The second equation tree of **Figure 15** shows an analogous scaling equation implementation. However, for this particular implementation, we broadcast m', v', G, B vectors into H and W dimensions inside the TPP equation tree. We can also apply the same scaling equation to a single group or set of groups with few parameter changes. An efficient implementation of groupnorm uses blocking on the C, H, and W dimensions. We do not show the details of this implementation for sake of simplicity.

5.1.3. BF16 Split-Stochastic Gradient Descent Kernel

Unlike the previous kernels which are well-established in DL workloads, and as such potentially optimized in DL libraries, we present here an example of a novel operator, which per definition is not existent in DL libraries. BF16 split-SGD was recently introduced in the context of DLRM training with BF16 datatype [37]. The Split-SGD-BF16 solver aims at efficiently exploiting the aliasing of BF16 and FP32 (i.e., the 16 Most Significant Bits (MSB) on both are identical) in order to save bandwidth during the SGD-solver in training. The employed trick is that the weights are not stored as FP32 values in a single tensor. Instead, the FP32 tensors are split into their high and low 16 bit-wide parts: the 16 MSBs of the FP32 values, and the 16 LSBs of the same values are stored as two separate tensors X^{hi} and X^{lo}, respectively. The 16 MSBs represent a valid BF16 number and constitute the model/weight tensors during training. These BF16 weights are used exclusively in the forward and backward passes, whereas the lower 16 bits are only required in optimizer. More specifically, the X^{hi} and X^{lo} tensors are packed together to form an FP32 tensor, resulting in a fully FP32-accurate optimizer. **Figure 16** illustrates the BF16 Split-SGD operator written entirely *via* TPPs. First the X^{hi} and X^{lo} are packed, and the formed FP32 tensor is used in a cascading FMADD TPP that performs the SGD scaling with the corresponding Gradient Weight tensor and learning rate. Finally, the resulting FP32 tensor is unpacked to the X^{hi} and X^{lo} tensors for further use in the training process.

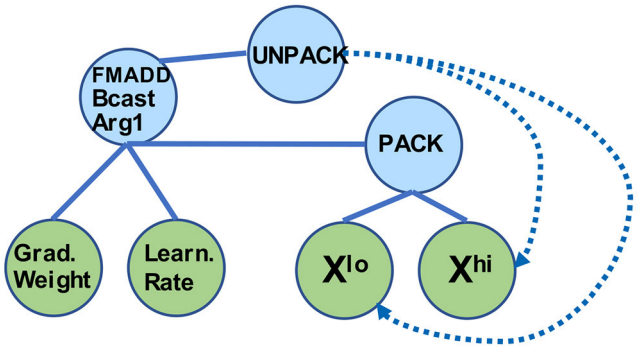


FIGURE 16 | BF16 Split-SGD operator by combining TPPs.

5.1.4. Convolutional Neural Network Kernel

Convolutional Neural Networks (CNN) consist of layers with multiple neurons connected by weights, and they have been applied with success in image recognition, semantic segmentation, autonomous driving, medical imaging and in an increasing number of scientific applications. Previous work [21, 23] has shown that CNNs, despite their seemingly complicated loop structure due to the involved high-dimensional tensors, can be mapped efficiently onto small 2D GEMMs and BRGEMMs. In this work, we adopt the same strategy to implement CNNs *via* the BRGEMM TPP. Unlike the previous work which presents only the address-based BRGEMM formulation, here, we leverage the CNN kernels with stride-based BRGEMM for 1×1 convolutions and offset-based BRGEMM for 3×3 convolutions to get even more performant implementations (see section 2.3 for a brief description of the BRGEMM variants).

5.1.5. Sparse Embedding Kernel

The sparse embedding kernel is comprised of multi-hot encoded lookups into an embedding table $W^{M\times E}$ with M being the number of rows and E the length of each row, whereas the multi-hot weight-vector is denoted as $\alpha^T = [0, \ldots, a_{p_1}, \ldots, a_{p_k}, \ldots, 0]$ with entries $a_p = 1$ for $p \in \{p_1, \ldots, p_k\}$ and 0 elsewhere (p being the index for the corresponding lookup items). Mathematically, the embedding lookup output vector o^T can be obtained *via* $o^T = a^T \times W$. This operation (assuming row-major storage for W) is equivalent to gathering the rows of W based on the non-zero indices a_p, and then adding them up to get the output vector o^T. **Figure 17** illustrates the TPP tree that is used to express the Sparse Embedding lookup kernel.

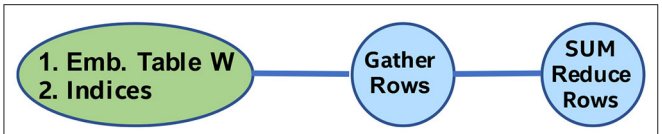


FIGURE 17 | Sparse Embedding Lookups *via* TPPs.

Algorithm 5 | Sparse Gather-Reduce operation.

Inputs: $\alpha^T = [0, \ldots, a_{p_1}, \ldots, a_{p_k}, \ldots, 0]$ with entries $a_p = 1$ for $p \in \{p_1, \ldots, p_k\}$ and 0 elsewhere, $W^{M \times E}$
Output: $o^T = a^T \times W$

1: **for** $j = 0 \ldots E$ **with step** $vlen \cdot U$ **do**
2: ▷ *Initializing accumulator registers to 0*
3: **for** $u = 0 \ldots U - 1$ **do**
4: $vec\_out_u \leftarrow 0$
5: ▷ *Iterating over non-zero entries/indices in* α^T
6: **for** i **in** $1, 2, \ldots, k$ **do**
7: $idx = p_i$
8: $next\_idx = p_{i+1}$
9: ▷ *Unroll innermost kernel U times: load indexed vector, prefetch next indexed vector, accumulate loaded vector to accumulator register*
10: **for** $u = 0 \ldots U - 1$ **do**
11: $vec\_W \leftarrow$ load_vector($W[idx][j + u \cdot vlen : j + (u + 1) \cdot vlen]$)
12: prefetch($W[next\_idx][j + u \cdot vlen : j + (u + 1) \cdot vlen]$)
13: vec_out_u += vec_W
14: ▷ *Store accumulator registers to* o^T
15: **for** $u = 0 \ldots U - 1$ **do**
16: $o^T[j + u \cdot vlen : j + (u + 1) \cdot vlen] \leftarrow vec\_out_u$

We note that the TPP backend optimizes this sequence of TPPs, and performs register fusion across the gather and the reduce TPP components. More specifically, given a non-zero index a_p, the corresponding row of W is loaded in vector registers, and is added to a set of running accumulators/vector registers that hold the output o^T. **Algorithm 5** illustrates the optimized JITed implementation in our TPP backend. The E dimension is vectorized in an SIMD-fashion with vector length *vlen*. Note that in line 13 we expose multiple independent accumulation chains in order to hide the latency of the vector-add SIMD instructions. Since we JIT this sub-procedure, we know the exact value of E at runtime. As such, we can pick appropriate unrolling factor U as well as the remainder handling can be performed optimally *via* masking in case E is not perfectly divisible by the vector length *vlen*. Last but not least, the JITed aggregation procedure employs prefetching of the subsequent indexed vectors in W (line 12) in order to hide the latency of these irregular accesses.

5.1.6. Multi-Layer Perceptron Kernel

Multilayer perceptrons (MLP) form a class of feed-forward artificial neural networks. An MLP consists of (at least three) *fully connected* layers of neurons. Each neuron in the topology

Algorithm 6 | Fully-Connected Layer with Unary Activation Function.

Inputs: $A^{M_b \times K_b \times b_k \times b_m}$, $B^{N_b \times K_b \times b_n \times b_k}$
Output: $C^{N_b \times M_b \times b_n \times b_m}$

1: Based on *thread_id* calculate M_b_start, M_b_end, N_b_start and N_b_end to assign output work items
2: **for** $ib_n = N_b\_start \ldots N_b\_end$ **do**
3: **for** $ib_m = M_b\_start \ldots M_b\_end$ **do**
4: $Out = \&C[ib_n][ib_m][0][0]$
5: ▷ *Stride-based BRGEMM, stride_A=*$b_k \cdot b_m$*, stride_B=*$b_n \cdot b_k$
6: **BRGEMM**($\&A[ib_m][0][0][0], \&B[ib_n][0][0][0], Out, K_b$)
7: $C[ib_n][ib_m][0][0] \leftarrow$ **UNARY**($C[ib_n][ib_m][0][0]$)

may be using a non-linear activation function. In this section, we present the implementation of the *Fully Connected* layers since they constitute the cornerstone of MLP. Even though, we illustrate the forward pass of Fully Connected layers, we also implement *via* TPPs the kernels of the back-propagation training in an analogous fashion. **Algorithm 6** shows the fully connected layer implementation which is mapped to TPPs. First we note that the input tensors are conceptually 2D matrices $A^{M \times K}$ and $B^{K \times N}$ that need to be multiplied. We follow the approach of previous work [21] and we block the dimensions M, K, and N by factors b_m, b_k, and b_n, respectively. Such a blocked layout is exposing better locality and avoids large, strided sub-tensor accesses which are known to cause Translation Lookaside Buffer (TLB) misses and cache conflict misses in case the leading dimensions are large powers of 2 [21]. We leverage the BRGEMM TPP in order to perform the tensor contraction with A and B across their dimensions K_b and b_k (which constitute the K/inner-product dimension of the original 2D matrices). We employ the stride-based BRGEMM because the sub-blocks "A_i" and "B_i" that have to be multiplied and reduced are apart by constant strides $stride\_A = b_k \cdot b_m$ and $stride\_B = b_n \cdot b_k$ respectively. Finally, we apply (optionally) a unary TPP corresponding to the requested activation function (e.g., RELU) onto the just-computed output block of C.

5.2. TPP-Based Workloads

5.2.1. 1D Dilated Convolutions and Computational Biology

In this subsection, we show the implementation of a special type of convolution *via* TPPs in their entirety, namely one-dimensional (1D) dilated convolution layer of a 1D CNN named ATACworks [38]. ATACworks is used for de-noising and peak calling from ATAC-Seq genomic sequencing data [38]. The 1D dilated convolution layer in ATACworks takes more than 90% of the training time, and it has input tensor width W, output tensor width Q, C input channels, K output channels, filter size of S, and dilation d. We employ the transpose TPPs, copy TPPs, and BRGEMM TPPs to optimize the forward pass and the backward pass of the PyTorch-based 1D convolution layer. **Algorithm 7** shows an example of the forward pass procedure with an input tensor I, a weight tensor W, and an output tensor O.

Algorithm 7 | 1D Dilated convolution forward pass using TPPs.

Inputs: $I^{C\times W}$, $W^{K\times C\times S}$, $d \in \mathbb{R}$
Output: $O^{K\times Q}$

1: $W^T \leftarrow$ **TRANSPOSE**(W)
2: **for** $pos = 0 \ldots Q-1$ **with step** $\mathbf{b_q}$ **do**
3: ▷ *Address-based BRGEMM, prepare arguments* A_{ptrs}, B_{ptrs}
4: **for** $s = 0 \ldots S-1$ **with step 1 do**
5: $A_{ptrs}[s] = \&W^T[s, 0, 0]$
6: $B_{ptrs}[s] = \&I[0, (pos + s \cdot d)]$
7: **BRGEMM**$(A_{ptrs}, B_{ptrs}, \&O[0, pos], S)$

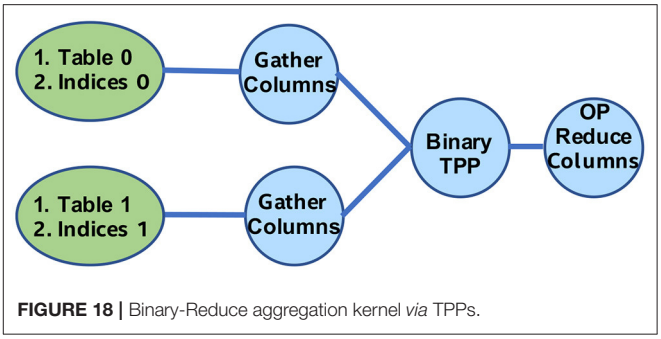


FIGURE 18 | Binary-Reduce aggregation kernel *via* TPPs.

5.2.2. Deep Learning Recommendation Model

Facebook recently proposed a deep learning recommendation model (DLRM) [39]. Its purpose is to assist the systematic hardware–software co-design for deep learning systems. DLRM is comprised of the following major components: (a) a sparse embedding (see section 5.1.5) involving tables (databases) of varying sizes, (b) a small dense Multi-Layer Perceptron (see section 5.1.6), and (c) a larger and deeper MLP which is fed by the interaction among (a) and (b). All three parts can be configured (number of features, mini-batch sizes, and table sizes) to stress different aspects of the system. We also note that in the case of training with BF16 datatype, we leverage the BF16 split-SGD optimizer (see section 5.1.3). For more details on the workload and CPU-oriented optimizations we refer to prior work [37].

5.2.3. Natural Language Processing - Bidirectional Encoder Representations From Transformers

The BERT model is a bidirectional transformer pre-trained *via* a combination of masked language modeling objective, and next-sentence prediction [40]. The heart of the BERT model is comprised by sequence of BERT layers which are built using smaller building blocks. For ease of use and implementation, we followed modular building blocks from Hugging Face transformers library [41] and implemented four fused layers using TPP building blocks, namely *Bert-Embeddings*, *Bert-SelfAttention*, *Bert-Output*/*Bert-SelfOutput*, and *Bert-Intermediate* layers.

The *SelfAttention* layer, in turn, can be formulated as a bunch of Matrix / batch Matrix-Multiplications mixed with element-wise scale, add, dropout and softmax operators. We formulate these Matrix-Multiplications as tensor contractions on blocked-tensors *via* the stride-based BRGEMM TPP (similarly to **Algorithm 6**). We opt to use blocked tensor layouts for the same reasons briefly described in section 5.1.6. Furthermore, by working on one small sub-tensor at a time we naturally follow a "dataflow" computation, which has been shown to maximize the out-of-cache-reuse of tensors among cascading operators [26, 42]. The softmax operator is also formulated entirely by TPPs as described in section 5.1.1. We note that the sequence of Matrix-Multiplications in the attention layer requires sub-tensors to be transposed (and VNNI transformed in case of BF16 implementation), and for this task we leverage the transpose/transform TPPs. *Bert-Output* and *Bert-SelfOutput* layers perform GEMM over blocked layout, and fuse bias addition, dropout, residual addition, and layernorm using TPPs. The *Bert-Embeddings* layer also performs layernorm and dropout after embedding lookups that are also implemented using TPPs. Finally, *Bert-Intermediate* layer performs blocked GEMM followed by bias addition and GELU activation function which we implement using the GELU TPP.

5.2.4. Emerging AI—Graph Neural Networks

Graph Neural Networks (GNN) [43] form an emerging class of Neural Networks for learning the structure of large, population-scale graphs. Depending on the specific algorithm and task that a GNN is designed for (e.g., node classification, link prediction), feature-vector aggregation precedes or succeeds a shallow neural network. Such a shallow neural network typically materializes one or more linear transformations, followed by a classification or regression mechanism [44], and the relevant TPP-based implementation is essentially the one we present in **Algorithm 6**.

We focus here on the TPP-based implementation of the feature-vector aggregation. This aggregation motif can be seen as a sequence of linear algebraic expressions involving node/edge features, along with the relevant operators. Prior work [44] has focused on the following two algebraic sequences: Copy-Reduce and Binary-Reduce. We elaborate here on the latter sequence Binary-Reduce (as the first is even simpler). The feature-vectors (either pertaining to vertices or edges) are represented *via* dense 2D matrices/tables. At the same time, the adjacency information in the graphs can be eventually found *via* arrays of indices. Therefore, by providing a set of indices and the appropriate Tables of feature-vectors (assuming column-major storage), one can extract selectively the desired feature-vectors *via* Gather-columns operations. Then, the extracted feature-vectors are fed into a binary operator, and the outcome of the binary operations are finally reduced (the reduce operation could be sum/max/min etc).

Figure 18 illustrates a TPP tree that is used to express the Binary-Reduce aggregation kernel. The TPP back-end optimizes this sequence of TPPs and performs horizontal register fusion across them. More precisely, two feature-vectors namely v_0 and v_1 are extracted at a time from Table 0 and Table 1 respectively by using the relevant indices arrays, and they are combined *via* the proper binary op to get an intermediate vector v_i. Subsequently, v_i is reduced with a running reduce-vector v_o

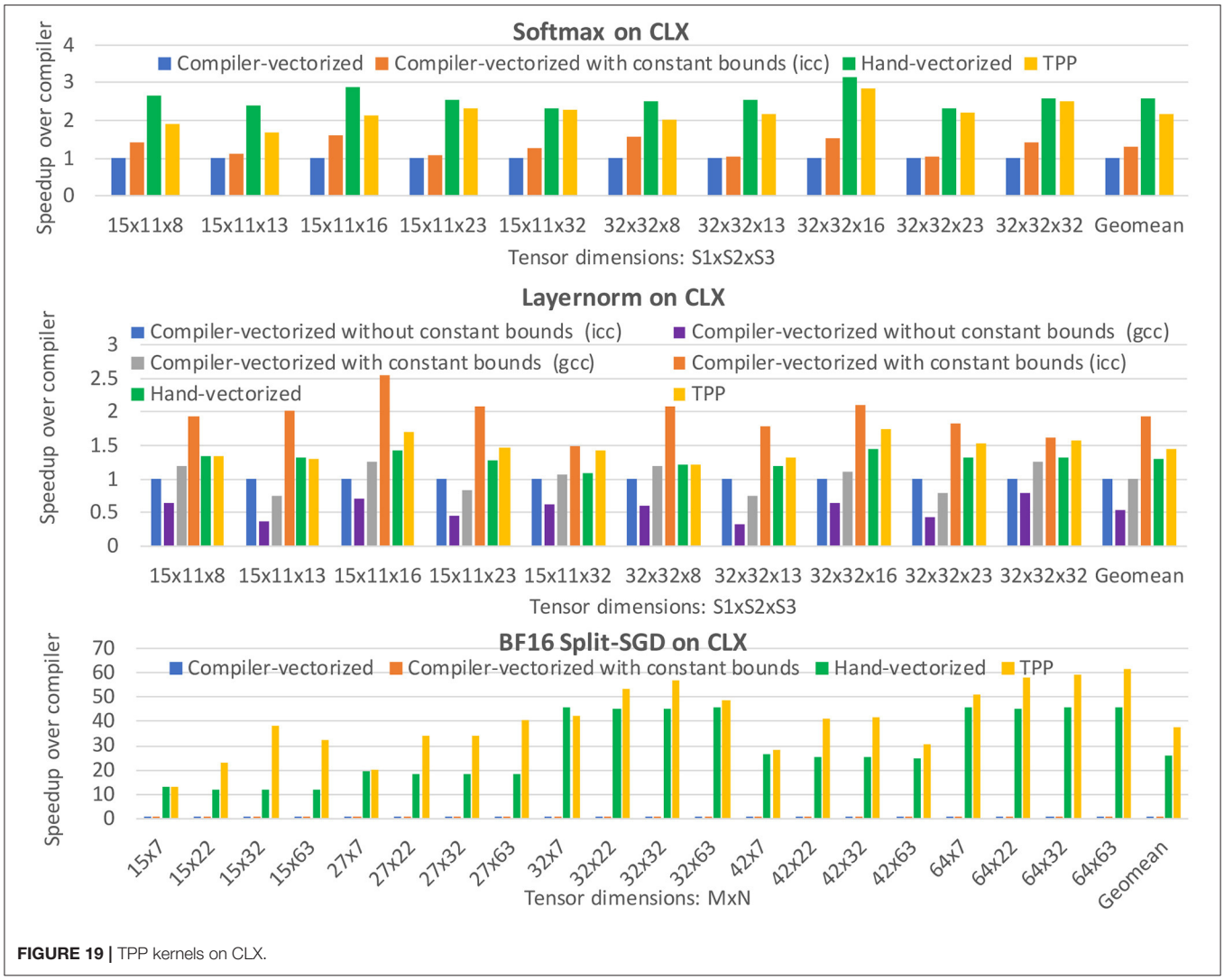


FIGURE 19 | TPP kernels on CLX.

that holds the output of this composite operator. Once the running reduction has been completed (i.e., all indexed columns from Table 0 and Table 1 have been accessed, processed and reduced), the output vector v_o is stored in the corresponding output subtensor.

6. EXPERIMENTAL RESULTS OF DL KERNELS AND WORKLOADS

We use a variety of platforms that span different ISAs, different vendors and micro-architectures. More specifically, our tested platforms include: (i) a 22-core Intel Xeon E5-2699 v4 (BDX) supporting up to AVX2 ISA, (ii) a 28-core Intel Xeon 8280 (CLX) supporting up to AVX512 ISA, (iii) a recently announced 40-core Intel Xeon 8380 (ICX) supporting also up to AVX512 ISA, (iv) a 28-core Intel Xeon 8380H (CPX) supporting up to AVX512 ISA, which also offers BF16 FMA acceleration, (v) a 64-core AMD EPYC 7742 (ROME) with AVX2 ISA, (vi) an AWS Graviton2 instance with 64 cores at fixed 2.5 GHz and AArch64 ISA, (vii) a 48-core Fujitsu A64FX at fixed 1.8 GHz with ARMv8 SVE ISA, and (viii) a 4-core client Intel i7-6700 CPU (i7) supporting up to AVX2 ISA. All Intel and AMD chips are operating in Turbo mode. For the cluster experiments, we used a 32 node CLX installation with a dual-rail Intel Omnipath 100 pruned 2:1 fat-tree topology.

6.1. Performance of Standalone DL Kernels

We start the performance evaluation with standalone TPP kernels presented in section 5.1. First, we want to highlight the productivity/efficiency provided by TPPs: the high-level code expressed *via* TPPs/trees of TPPs can match or outperform code by compilers, and hand-vectorized (thus non-portable code) written by performance experts. Second, we want to show the portability aspect of TPPs, since exactly the same high-level code yields high-performance across different ISAs and micro-architectures.

Figure 19-Top shows the performance of the Softmax operator of blocked 3D tensors with size $S1 \times S2 \times S3$, on the

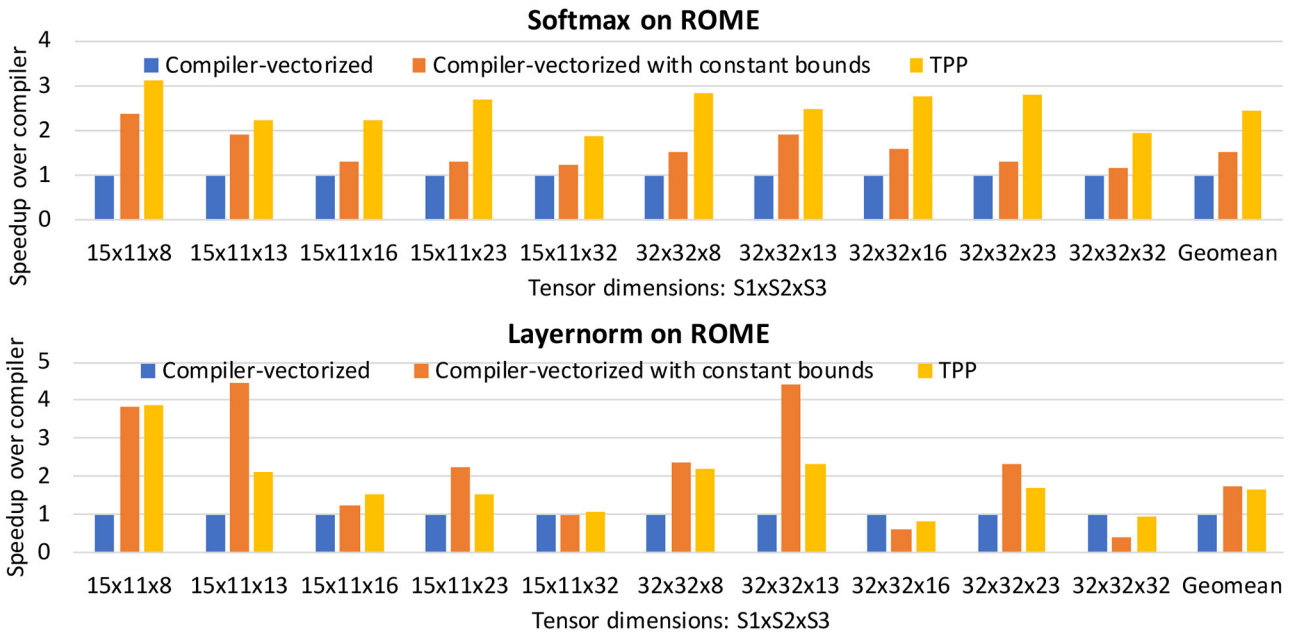


FIGURE 20 | TPP kernels on ROME.

CLX platform (i.e., targeting AVX512 ISA). Here, we perform $S2$ softmax operations over blocked $S1 \times S3$ dimensions. The sizes are chosen such that some of the dimensions do not match perfectly with the vector length. The baseline is the icc generated code with `-O3` optimization level and `high-zmm usage` flags. The second variant is also icc-generated code, but we propagate the tensor sizes/loop bounds *via* compile-time constants in order to assist the auto-vectorization/optimize remainder handling *via* masking. The third code variant is the AVX512 hand-vectorized by an expert, where the *exp* function uses fast Taylor approximation. Last, we evaluated the TPP-based softmax implementation. As we can see, by propagating the tensor sizes we achieve (geo-mean) speedup of 1.3× over the baseline. The hand-vectorized code is faster by 2.6× whereas the TPP-based variant shows similar speedups by being 2.2× faster. The main shortcoming of the hand-vectorized code is that it is platform-dependent and as such non-portable. More specifically, we didn't have to our avail AVX2 hand-optimized code in order to experiment with it on ROME. On the contrary, **Figure 20-Top** shows the softmax performance on AVX2 enabled platform for the compiler-generated code and the TPP based code. The TPP-based softmax exhibits geo-mean speedup of 2.45× over the baseline on ROME.

Figure 19-Middle shows the performance of the layernorm operator on the CLX platform. Since the layernorm code is more straightforward (i.e., no expensive *exp* function is involved), we see that icc with compile-constant bounds outperforms by 1.9× the baseline. We inspected the compiler-generated code and identified that the reduction-loops were recognized and were heavily optimized with multiple accumulation chains etc. Similarly, the hand-vectorized code and the TPP based code outperform the baseline by 1.3× and 1.5×. We also experimented with gcc and the `fast-math` flag, and it just matched baseline performance. We want to emphasize that propagating the tensor sizes as compile-time constants throughout the operators is not practical for real use-cases within DL frameworks. **Figure 20-Bottom** shows similar performance speedups on ROME, where the TPP-based code is 1.6× faster than the auto-vectorized baseline.

Figure 19-Bottom shows the performance of the BF16 split-SGD operator on CLX. This use-case represents a novel, mixed-precision operator where the compiler (icc with compile-time constant tensor sizes) struggles to yield good performance; the TPP-based code has geometric mean (geomean) speedup of 38× over the compiler generated code.

Figure 21 illustrates the TPP-based implementation of various ResNet50 [33] Convolution layers across all available platforms. The minibatch size used on each platform equals to the number of the corresponding cores. It is noteworthy that the TPP-user code is identical for all targets, hence, truly portable; it is merely that the TPP backend optimizes the code generation (BRGEMM in this case) in a platform/ISA-aware fashion. The geomean efficiencies of these convolutions are: 69% for BDX, 72% for CLX, 72% for CPX, 77% for CPX with BF16 datatype, 70% for ICX, 78% for ROME, 81% for Graviton2 and 52% for A64FX. Previous work [21] also showed on an x86 TPP-predecessor that BRGEMM-based convolutions matched or outperformed Intel's oneDNN library [13]. Fujitsu recently contributed an A64FX back-end to oneDNN [45] and our TPP implementation outperforms this by 22% on the geomean. We observe that our TPP convolutions not only run on all of these different platforms

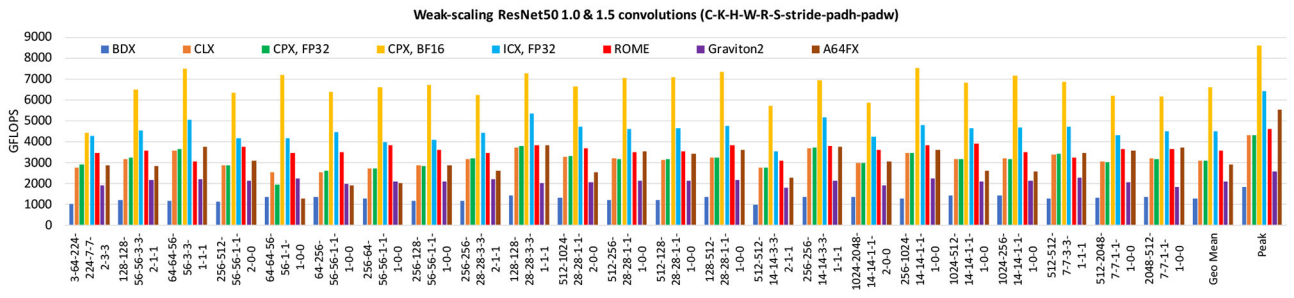


FIGURE 21 | Convolutions *via* BRGEMM TPP.

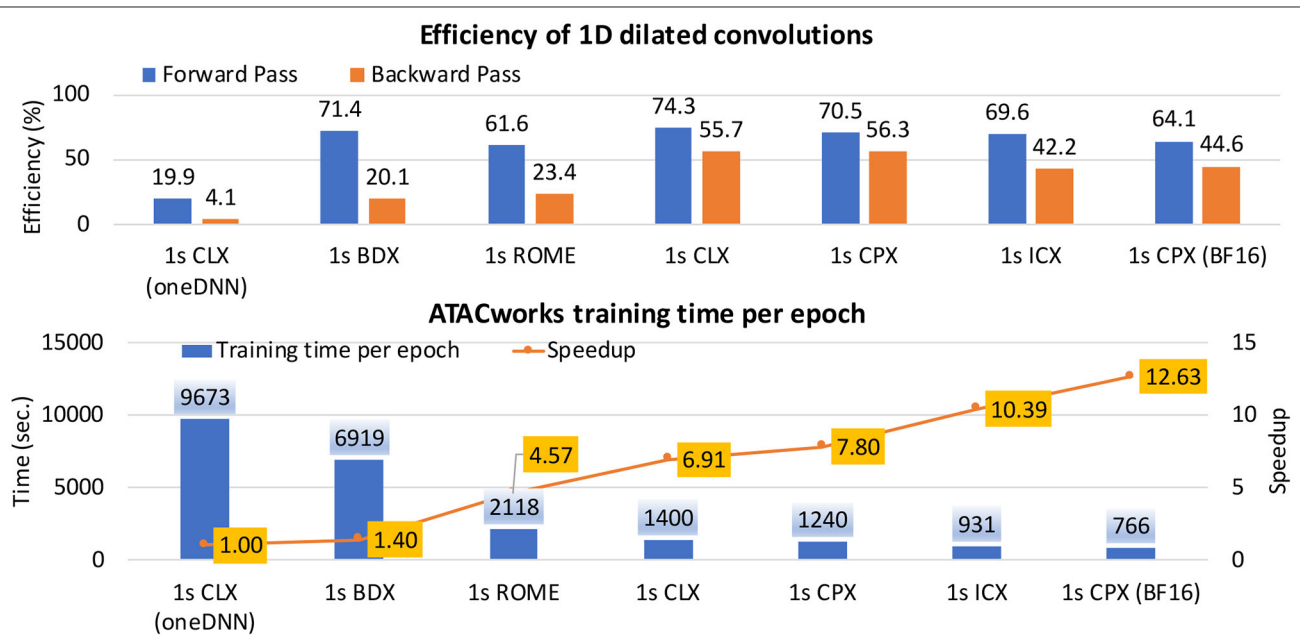


FIGURE 22 | 1D dilated convolutions.

without a single line of code change, but they run at very similar hardware utilization.

6.2. Performance of End-To-End DL Workloads

6.2.1. 1D Dilated Convolutions and Their Application to Computational Biology

Here, we evaluate the oneDNN [13] and TPP-based 1D dilated convolution layer of ATACworks [38] which takes more than 90% of the training time, and it has input tensor width (W) of 60,400, output tensor width (Q) of 60,000, 15 input channels (C), 15 filters (K), filter size (S) of 51, and dilation (d) of 8. **Figure 22-Top** shows the computational efficiency results of the 1D convolution layer. oneDNN is not reaching peak performance for these specialized convolutions, exhibiting 19.9% efficiency for the forward pass and only 4.1% for the backward pass on CLX. Our TPP-based implementation shows 74.3 and 55.7% efficiency for the corresponding training passes. We also highlight the performance portability of our TPP-based approach across all tested platforms. Finally, we show training time per epoch results for ATACworks in **Figure 22-Bottom**. The TPP-based kernels provide training time speedup of 6.91× on CLX when comparing to the oneDNN based implementation. We also show that by leveraging the BF16 FMA acceleration of the CPX platform we can further obtain 1.62× speedup compared to the FP32 implementation on the same platform. In total BF16 yields 12.6× speedup over the oneDNN baseline.

6.2.2. Deep Learning Recommendation—DLRM

Figure 23-Top shows the FP32 DLRM performance on CLX using two different configurations, namely small DLRM (blue bars) and MLPerf DLRM (orange basrs). We refer to previous work for the detailed specification of these configurations [37]. We evaluated 4 different implementations of DLRM: (i) the PyTorch reference implementation, (ii) PyTorch reference + custom Embedding extension auto-vectorized by the compiler,

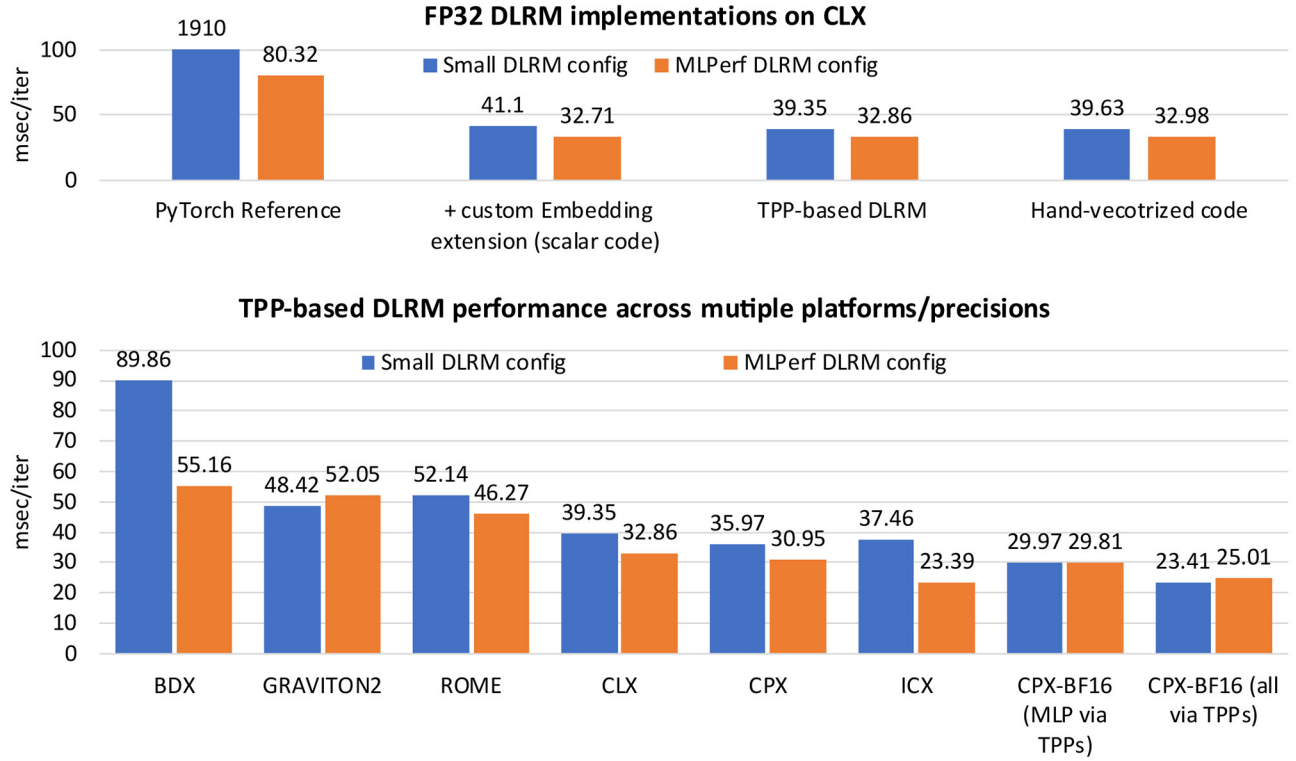


FIGURE 23 | DLRM performance on a small config (blue bars) and on the MLPerf config (orange bars).

(iii) DLRM expressed entirely *via* TPPs, and (iv) hand-vectorized Embedding extension + BRGEMM-TPP based MLPs [37]. We conclude that the TPP-based implementation matches the performance of the State-Of-The-Art implementation which is hand-vectorized specifically for AVX512 targets; both of these optimized versions substantially outperform the PyTorch CPU reference implementation by up to 48×. Compared to the version with the custom, auto-vectorized variant the TPP-version is up to 4.4% faster.

Figure 23-Bottom shows the DLRM performance of our TPP-based implementation across multiple platforms and compute precisions. We want to highlight two aspects: First, we are able to run the same TPP-code without any change across all platforms, something that is not doable with the hand-vectorized SOTA variant (iv) (since it is not able to run on the AVX2-only BDX and ROME platforms, or on the Graviton2 platform with AArch64 ISA). Second, the TPP-based BF16 shows speedup up to 28% over the variant with auto-vectorized Embedding extension. The culprit here is the mixed precision operations like split-SGD where the compiler struggles to yield efficient code as shown in section 6.1.

Figure 24 illustrates the performance breakdown of the small config on multiple platforms. The blue portions of the bars correspond to the time spent on the Embedding component, the orange parts reflect the MLP portion, and finally the yellow portions correspond to the remaining components of the DLRM workload. We observe that depending on the platform, the time spent on Embedding varies from 29 to 37% of the total execution time, the time spent on MLP is in the range of 33–56% of the total time, and the rest components account for 15–23% of the time. We can also observe the correlation of the MLP performance with the compute capabilities of each platform. For example, on CPX which has native BF16 FMA support, the BF16 MLPs are sped up by ∼2× compared to the FP32 MLPs on the same platform. In regard to the time spent on the Embedding kernel which tends to be bandwidth bound, we observe correlation with the corresponding bandwidth capabilities of the machines.

6.2.3. Natural Language Processing—BERT Large

Figure 25-Top shows end-to-end performance (in examples/second) on CLX for the BERT large SQuAD fine-tuning task in FP32, using a max sequence length of 384 and minibatch of 24. We observe that the TPP-based implementation (blue bar) matches the performance of the AVX512-hand-vectorized code/orange bar. At the same time, our implementation is 1.69× faster than the Reference Hugging Faces CPU reference code [46] (green bar).

Figure 25-Bottom shows the performance of the reference Hugging Faces code (green bars) versus the TPP-based code (blue bars) across multiple platforms (x86 and AArch64/Graviton2) and compute precisions (FP32 for all platforms, and BF16 for the CPX platform). The TPP-based BERT shows speedups ranging from 1.5× to 8.5× over the Hugging Faces code. This result highlights the performance portability through the TPP

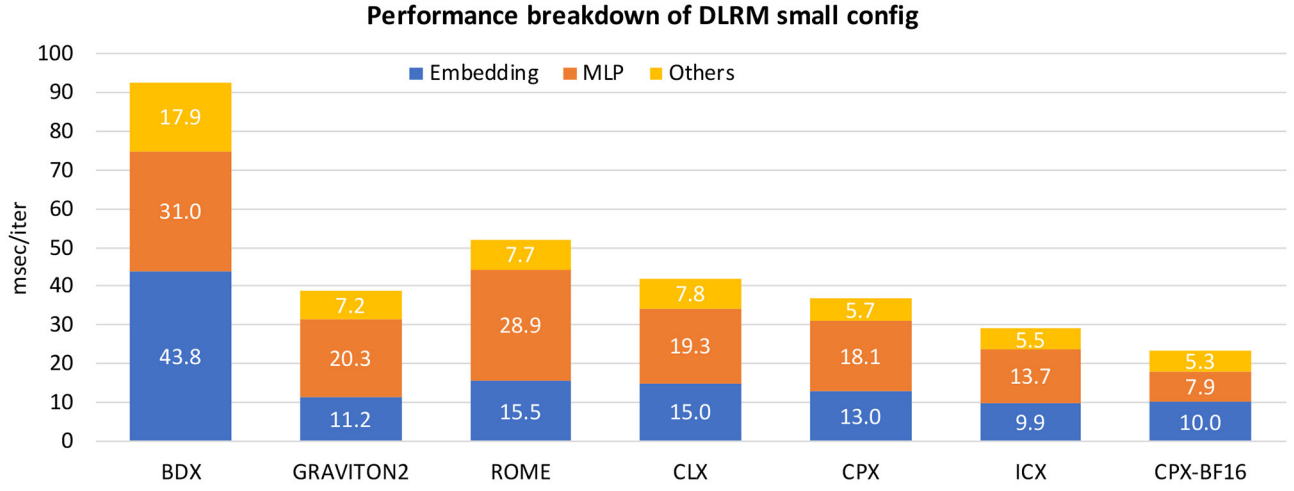


FIGURE 24 | DLRM performance breakdown of small config on multiple platforms.

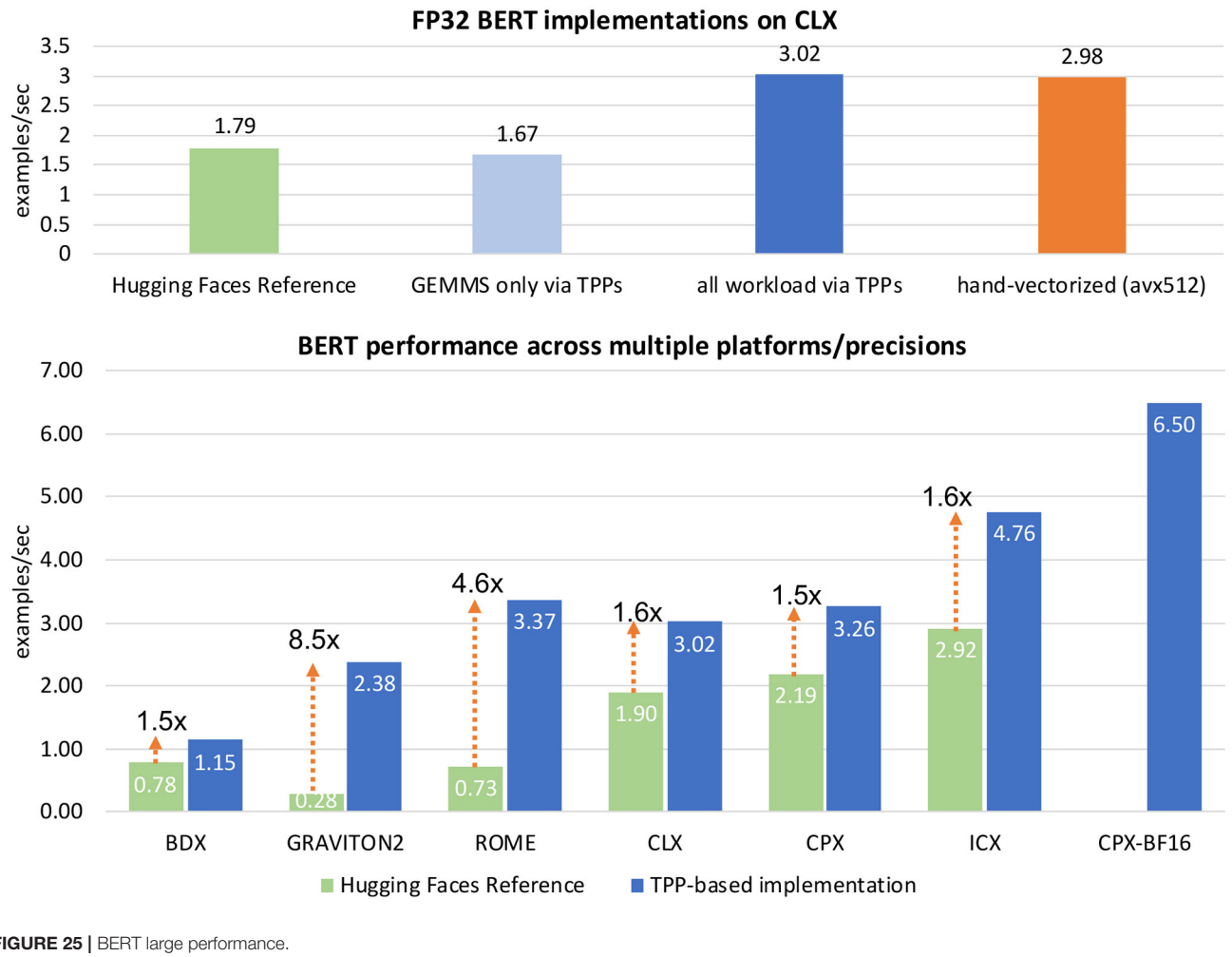


FIGURE 25 | BERT large performance.

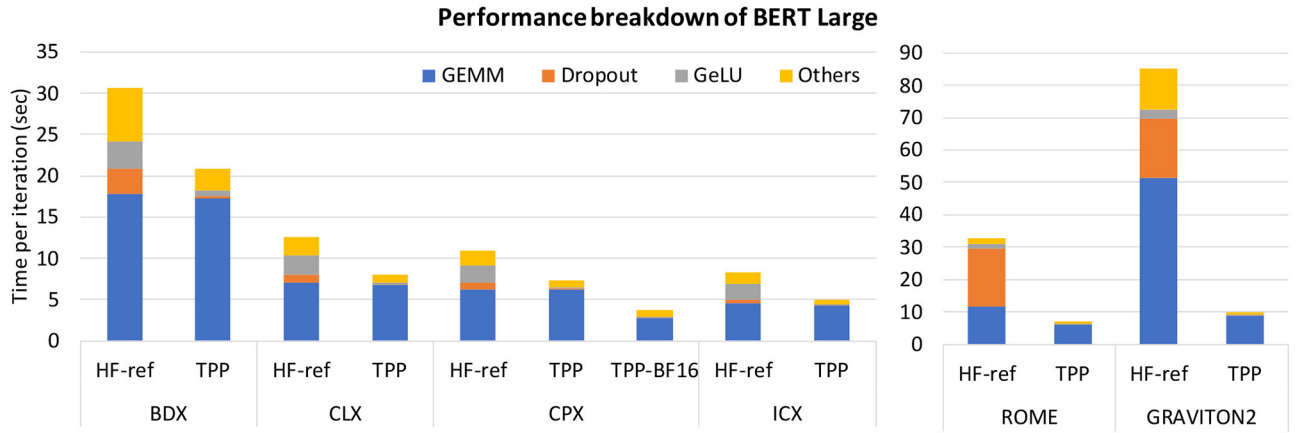


FIGURE 26 | BERT large performance breakdown on multiple platforms.

abstractions. In regard to various compute precisions, we note that with minimal changes inside the fused operators to handle the VNNI tensor layout (required for BF16 GEMM/BRGEMM), and a couple of lines changes in the application code to enable BF16 training, we were able to realize 2× speed up using BF16 training on CPX (compared to FP32 training on CPX) with 28 cores, surpassing 40-core FP32-ICX performance by 37%.

In order shed light on where the benefits are coming from, we present in **Figure 26** the performance breakdown of the Hugging Faces reference code and the TPP-based implementation. In particular we focus on four components:

1. *GEMM* which corresponds to the tensor contractions implemented *via* either the BRGEMM-TPP in the TPP implementation, or it leverages optimized GEMM routines within BLAS libraries in the Hugging Faces implementation (MKL for x86 platforms and OpenBLAS for AArch64/Graviton2).
2. *Dropout* corresponding to the dropout layer in BERT, where the TPP-based implementation employs fast random number generation *via* xorshift algorithm.
3. *GeLU* corresponding to the Gaussian Error Linear Unit activation function in BERT, where the TPP-based implementation leverages fast approximations as discussed in section 3.3.2.
4. *Others* capturing the remaining operators: Transpose, Layer-norm, softmax, bias addition, vnni-reformatting (in case of BF16 training), copy, add, scale, zero-kernel, reduce, optimizer. Note that all these operators map to either unary/binary/ternary TPPs (see section 2) or the can be expressed *via* Matrix Equation TPPs (see section 5).

First, we note that for the Intel x86 platforms (left part of the breakdown plot) the tensor contractions show speedups over the highly-optimized MKL GEMM implementation in Hugging Faces in the range of 2–6%. On the right side of the breakdown plot we observe that the BRGEMM-TPP benefits are even larger on the non-Intel platforms. More specifically, on AMD Rome (AVX2 x86 platform) the tensor contractions are sped up by 1.9× *via* the BRGEMM-TPP, and on Graviton2 (Arm AArch64 platform) the tensors contractions are 5.7× faster *via* the BRGEMM-TPP compared to the implementation relying on OpenBLAS GEMM calls. To further highlight the performance portability of the tensor contractions *via* the BRGEMM-TPP across multiple platforms and precisions, **Figure 27** shows the achieved GEMM performance (Left axis) on each platform for the entire training process (blue bars), whereas the orange line (Right axis) dictates the % of machine peak. The conclusion here is that the BRGEMM-TPP delivers high-efficiency for the corresponding tensor contractions in the range of 66–84% for all tested ISAs and micro-architectures.

The second conclusion we can draw from the performance breakdown in **Figure 26** is that our fused/dataflow TPP implementation outlined in section 5.2.3 makes the dropout and GeLU times shrink substantially, offering speedups in the range of 10–360×. The BERT implementation *via* the dropout/GeLU TPPs in tandem to the BRGEMM TPPs take advantage of temporal locality, and virtually make the corresponding times disappear from the overall execution time. Last but not least, the remaining components are sped-up in the TPP-based implementation by 2.5-14× depending on the platform. As a result of these optimizations, the TPP-based BERT implementation spends the majority of the time (75.5–88.8%) in tensor contractions which are executed at high-efficiency as **Figure 27** shows.

6.2.4. Emerging AI—Graph Neural Networks

Figure 28-Top shows end-to-end performance (in seconds/epoch, so lower is better) on CLX for the full-batch training of the GraphSAGE workload on OGB-Products with FP32 and BF16 precision. For the CLX BF16 experiments, since CLX doesn't have native support for BF16 FMAs, we use bit-wise accurate emulated-BF16 BRGEMM TPPs (see section 3.2.2), and we still expect savings due to the bandwidth reduction in the non-GEMM parts, e.g., graph traversal and edge/node aggregation. We observe that the TPP-based implementation outperforms the DGL with Xbyak JIT backend baseline

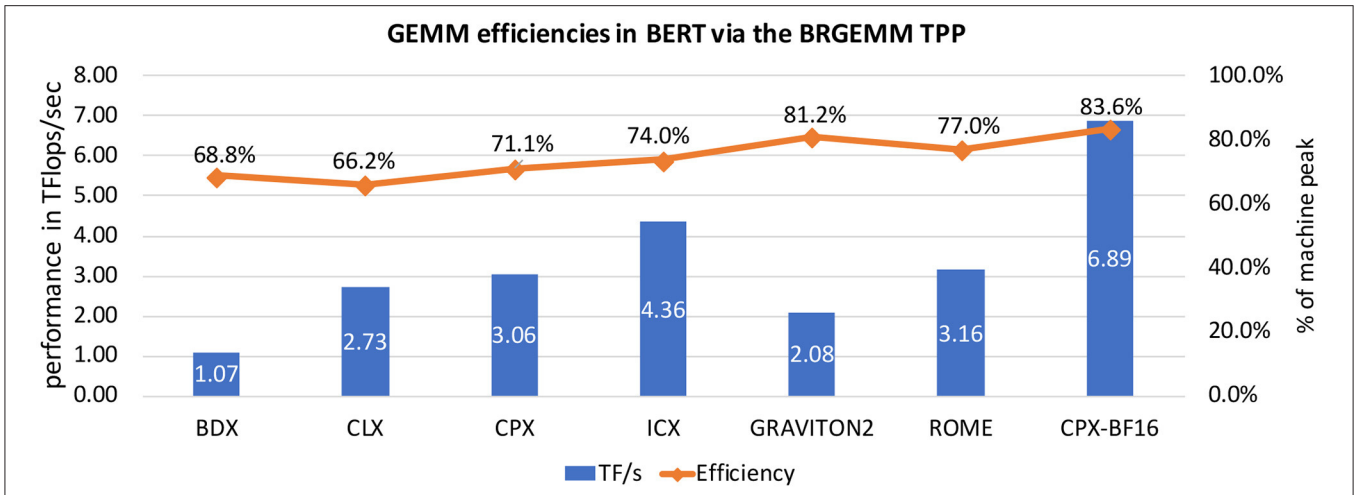


FIGURE 27 | BERT GEMM/tensor contraction efficiencies *via* the BRGEMM-TPP on multiple platforms.

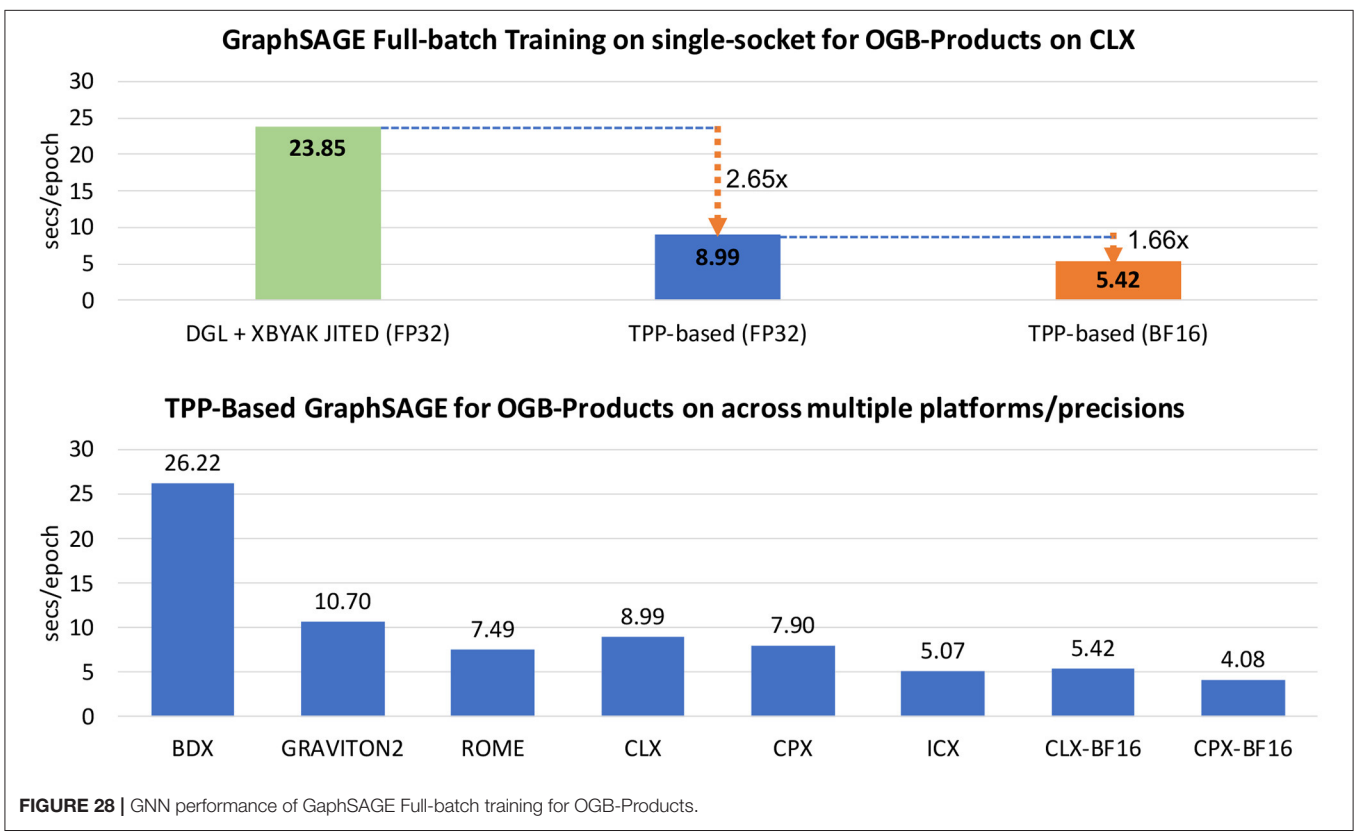


FIGURE 28 | GNN performance of GaphSAGE Full-batch training for OGB-Products.

version by 2.65×. The TPP-BF16 version yields another 1.66× speedup over the TPP-FP32 variant mainly due to reduced bandwidth requirements.

Figure 28-Bottom shows the performance of the TPP-based code across multiple platforms (x86 and Arm AArch64) and compute precisions (FP32 and BF16). The relative differences in the performance can be justified by the different compute/bandwidth specs of the benchmarked platforms. We highlight that with minimal changes in the MLP portion to handle VNNI layout required for BF16 BRGEMM, and a couple of lines changes in the application code to enable BF16 training, we were able to realize 1.94× speed up using BF16 training on CPX with 28 cores compared to the FP32 training on the same platform.

In order to further analyze the behavior of the various implementations on multiple platforms, we present on **Figure 29**

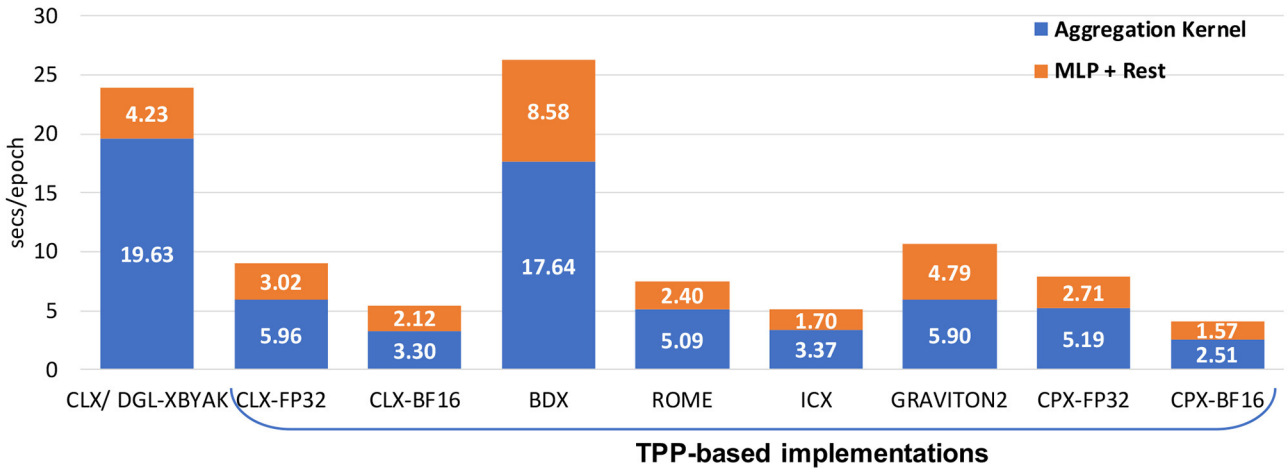


FIGURE 29 | GNN performance breakdown of GaphSAGE Full-batch training for OGB-Products.

the relevant performance breakdown. The very left bar shows the performance breakdown of the FP32 optimized DGL implementation that leverages JITed kernels through Xbyak on the CLX platform. The blue part corresponds to the Aggregation kernel described in section 5.2.4 whereas the orange portion represents the time required by the remaining kernels, namely Multilayer-Peceptrons with Activation functions. In the DGL implementation the activation functions are not fused within the MLP's tensor contractions. We observe that in this optimized DGL implementation, 82.3% is spent on the Aggregation kernel and only 17.7% is spent on the MLPs. On the second from the left bar (annotated as CLX-FP32) we show the performance of the FP32 TPP-based implementation on the same CLX platform. We conclude that the TPP-based Aggregation kernel exhibits a speedup of 3.29× compared to the DGL-Xbyak implementation, and the TPP-based MLP kernels (BRGEMM-TPP tensor contractions with *fused* TPP activation functions) exhibit a speedup of 1.4× compared to the respective DGL-Xbyak implementation. The FP32 TPP-based implementation spends 66.4% on the aggregation kernel and 33.6% on the fused MLP kernels.

The last 8 bars on **Figure 29** illustrate the performance breakdown of the TPP-based implementation on various platforms (CLX/BDX/ROME/ICX/GRAVITON2/CPX) and various precisions (FP32 and CPX-BF16). We want to emphasize that all these performance numbers are obtained by employing a the same exact TPP-based code (which is *platform-agnostic*); the only modification is pertaining to the BF16 TPP code where we changed the tensor layouts in the MLP portion in order to deal with the required VNNI format. When comparing the CPX-F32 and the CPX-BF16 performance breakdowns we observe a 2× speedup on the Aggregation kernel. This kernel is typically bandwidth bound due to its irregular/indexed accesses, and the BF16 TPP code moves half of the data compared to the FP32 TPP code since all the tensors are halved in size (BF16 vs FP32 datatype). The MLP portion of the TPP-based implementation is sped up by 1.73× by using the BF16 BRGEMM-TPP. The CPX platform supports the BF16 FMA instruction which has effectively 2× the compute throughput compared to the FP32 FMA on the same platform. The BF16 BRGEMM-TPP internally leverages this BF16 FMA instruction within the GEMM microkernel on CPX (see section 3.2) to speed up the tensor contraction. Finally, we highlight here the speedup of the Aggregation kernel when, e.g., comparing the CPX and the ICX FP32 TPP-based performance numbers. The ICX platform has STREAM bandwidth of 175 GB/s whereas CPX has 97.7 GB/s, and this trend is reflected also in the performance of the Aggregation kernel (1.54× faster on ICX than CPX).

6.3. Distributed-Memory Scaling of DL Workloads

Even though we focused on the evaluation of the TPP-based workloads on a single node, our approach is seamlessly incorporated into the DL frameworks, hence we can scale to multiple nodes in a cluster to accelerate the training process employing the oneCCL library [47]. **Figure 30** shows the distributed-memory scaling of the TPP-based workloads. DLRM and BERT show almost perfect weak-scaling from 1 to 64 sockets of CLX (32 nodes) with speedups 51.7 and 57.9×, Respectively. Regarding the scaling of the GNN workload, the efficiency is directly affected by the quality of the partitions produced by the graph partitioning tools. Using 64 sockets we achieve 10× speedup compared to single socket, and further scaling improvements constitute future work. We can conclude that TPPs for single node optimizations combined with small-size cluster level execution can accelerate deep learning training on CPUs by up to two orders of magnitude.

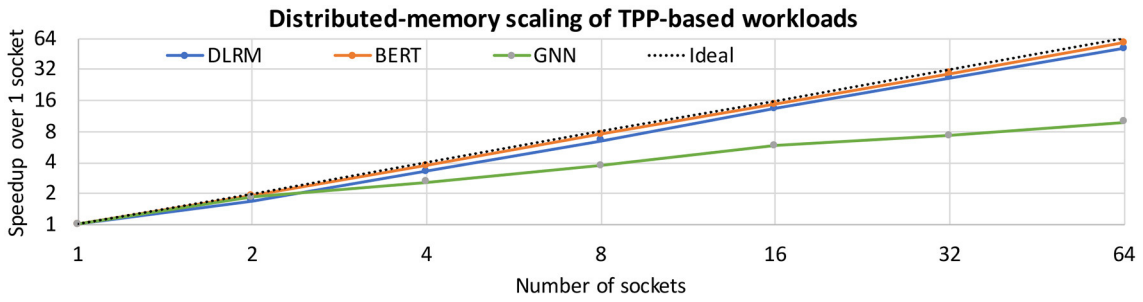


FIGURE 30 | Distributed-memory scaling of workloads.

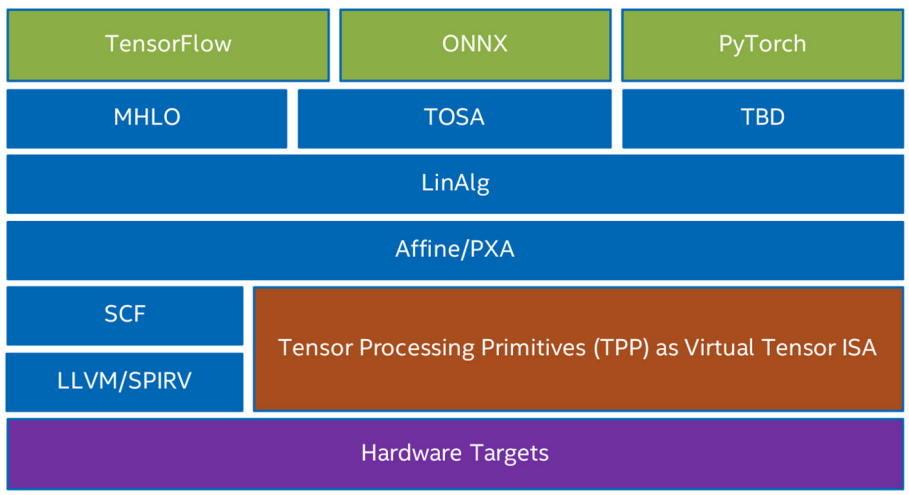


FIGURE 31 | Example lowering paths within the PlaidML Tensor compiler in order to achieve full network optimization from popular frameworks. The green boxes represent the DL frameworks, the blue boxes correspond to MLIR dialects, the brown box shows the TPP-MLIR dialect within the stack, and the purple box represents the targeted platforms.

7. TPP WITHIN MLIR AND A TENSOR COMPILER

In order to illustrate the viability of TPPs as a virtual Tensor ISA within MLIR and Tensor Compilers, we implemented a rudimentary MLIR dialect corresponding to the TPPs. We also implemented lowering passes within the PlaidML [15] Tensor Compiler that transform intermediate MLIR representations to the TPP-MLIR dialect. The TPP-MLIR dialect is subsequently lowered to the corresponding LIBXSMM TPP calls, therefore such a flow is not relying on LLVM for the code generation of the corresponding tensor operations.

The current lowering path through MLIR supports a variety of front-end interfaces with LinAlg or Tile as the lowest level common entry points, i.e., the lowest level of abstraction that inbound programs can be specified in such that they will be subject to the full range of optimizations necessary to achieve full performance. **Figure 31** details the lowering paths currently implemented in PlaidML and where key transforms map tensor operations into the TPP dialect. The key transformation is located in the stencil pass of the PXA dialect (Parallel eXtensions for Affine—a staging ground for PlaidML/TPP work that will be proposed upstream to the affine dialect). Operations that cannot be matched to TPP primitives are lowered through standard affine optimization pipelines.

We experimented with the use-case of FP32 inference on a client CPU (Intel i7-6700) on three different workloads: ResNet-152 [33], ResNext-50 [48], and I3D-Kinetics-400 [49]. **Figure 32** shows the results of three implementations: (i) The green bars show the performance of the code generated by PlaidML with MLIR for intermediate representations, and LLVM for the code generation, (ii) The orange bars show the performance of the code generated by PlaidML with MLIR for intermediate representations, and the TPP-MLIR dialect as virtual Tensor ISA for the code generation of the corresponding tensor contractions, and (iii) TensorFlow FP32 inference backed-up by the vendor-optimized oneDNN library. We observe that the Tensor Compiler variant which relies on the TPP-MLIR dialect for the tensor contractions outperforms the variant which relies exclusively on LLVM (for loop-tiling and vectorization) up to

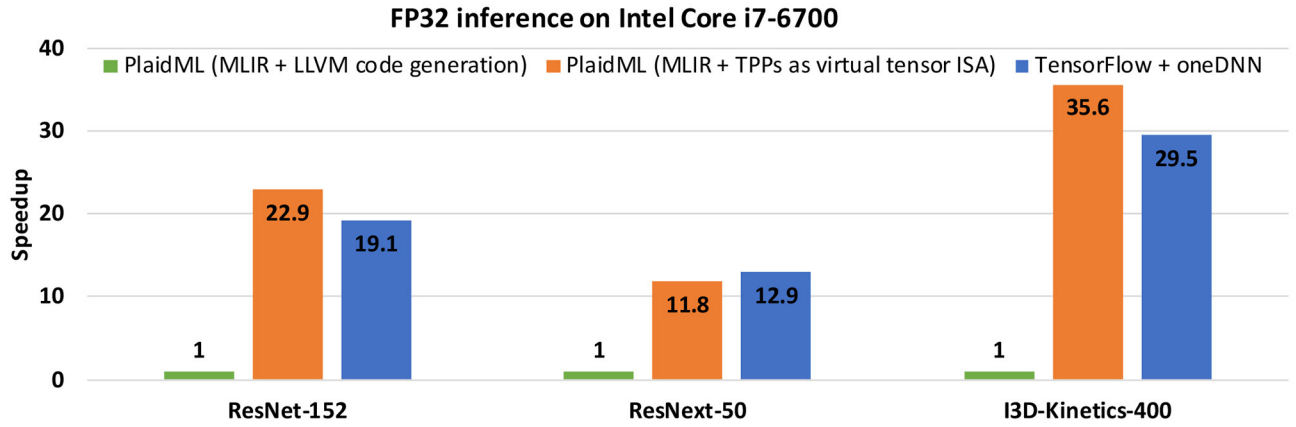


FIGURE 32 | FP32 inference with PlaidML on various workloads: ResNet-152, ResNext-50, and I3D-Kinetics-400.

35.6×. At the same time, PlaidML assisted by the TPP-MLIR dialect matches/outperforms the performance of TensorFlow which uses internally oneDNN, a highly-tuned vendor library for this CPU target. These preliminary results highlight the viability of the synergistic Tensor Compiler—TPP paradigm as discussed in section 1.

8. TPP AND HPC APPLICATIONS

So far, in this article, the focus was on how the TPP abstraction can be leveraged within the Deep Learning Domain. Tensor computations are ubiquitous, and in particular they constitute the cornerstone of many HPC applications. As such, the TPP abstraction can be readily employed by HPC applications to accelerate tensor computations without sacrificing portability. In the rest of this section, we examine how TPPs are used within two HPC applications, namely CP2K and EDGE.

8.1. CP2K

The tensor based formulation originated and became common in physics, and it is well adopted in the field of engineering or applied sciences, and in electronic structure (ES) theory in particular. CP2K is an open source ES- and MD-package (molecular dynamics) for atomistic simulations of solid-state, liquid, molecular, and biological systems [50]. CP2K is striving for good performance on HPC and massively parallel systems. Even though the use of novel algorithms in CP2K is the norm for scientific reasons, implementations have not widely tapped tensors in an explicit fashion. In contrast, Machine Learning emerged with similar, yet not coherent APIs and frameworks around the notions of tensors, layers, and image processing.

While ES calculations can be formulated with tensors of ranks two to four, CP2K (and similar packages) largely remain with matrix based formulation. Various libraries for tensor contractions gained some attraction for scientific applications but the level of generality is key, e.g., as sparse representations are desired. CP2K explored an API for sparse tensor contractions and published a proof of concept implementation built into the DBCSR library [51]. Efforts targeting accelerators in CP2K, namely GPUs, are not fully booked hence hardware specifically for Deep Learning (with focus on low and mixed precision arithmetic) is not yet a motivation of tensors as an implementation vehicle (and source of acceleration). Therefore, a collection of primitives, such as TPP is well-suited for an emerging discussion of a more general API.

CP2K 3.0 introduced LIBXSMM for Small Matrix Multiplications (SMMs). CP2K and DBCSR (previously part of CP2K's code base) since then additionally introduced element-wise operations (copy and transpose) with "elements" being small matrices based on LIBXSMM. Reformulating existing code to build on (batched) GEMM TPP and element-wise TPP operations is an established pattern for increased performance in CP2K.

To practically improve performance in CP2K one has to consider:

- Fusing kernels and increasing arithmetic intensity independent of the target being a CPU or an accelerator (performance bound by memory bandwidth).
- Specializing code at runtime based on workload/input of the application, e.g., generating code Just-In-Time (JIT) a.k.a. meta-programming.

These objectives can be delivered by TPPs as a domain-specific language (DSL), enabling the scientist to write more abstract code, e.g., by the means of meta-programming, and by relying on a specification which delivers versatile primitives deferring low-level optimizations to the TPP backend.

For CP2K's performance evaluation, we refer to BDX, CLX, ICX, and ROME as introduced earlier (section 6). To show the portability of our approach, we augmented our results by using the Oracle Cloud Infrastructure, namely the result for *Altra* processor (BM.Standard.A1.160 OCI shape). **Table 4** shows the performance benefit of LIBXSMM's GEMM-TPP in CP2K when compared to Intel's MKL GEMM routines.

TABLE 4 | CP2K performance (Cases/Day) of three workloads fitting into single systems with two processors.

System	Workload[a]	BLAS-GEMM[b]	TPP-GEMM[c]	TPP-Speedup
BDX	H2O-256	91	**101**	11%
	H2O-512	23	**27**	17%
CLX	H2O-256	154	**162**	5%
	H2O-512	39	**41**	5%
	H2O-DFT-LS4	45	**47**	4%
ICX	H2O-256	235	**249**	6%
	H2O-512	60	**65**	8%
	H2O-DFT-LS4	67	**70**	4%
ROME	H2O-256	225	**244**	8%
	H2O-512	55	**57**	4%
	H2O-DFT-LS4	**65**	**65**	0%
Altra	H2O-256	228	**236**	4%
	H2O-512	60	**62**	3%
	H2O-DFT-LS4	60	**66**	10%

Single-socket performance is reported here for consistency within this article. Intel MKL or OpenBLAS are always used for general BLAS operations including large GEMMs. Either (BLAS-)GEMM or TPP-GEMM was used for batched multiplication of small matrices (SMMs). Workloads utilizing CP2K's DBCSR library for distributed block-sparse matrix multiply benefit from (runtime-)specialized GEMM-TPP kernels where the set of matrix shapes is not known at compile-time of the application or depends on the workload in general.

[a] H2O-256 (CP2K bench.), H2O-512 (UEABS Case A) and H2O-DFT-LS NREP=4 (UEABS Case C) from PRACE UEABS 2.1.

[b] Intel MKL (x86-64) or OpenBLAS (otherwise).

[c] LIBXSMM.

The bold values indicate the best-performing implementation for each system and workload.

8.2. EDGE

The Extreme-Scale Discontinuous Galerkin Environment (EDGE) uses the Arbitrary high-order DERivatives (ADER) Discontinuous Galerkin (DG) finite element method to simulate seismic wave propagation [52]. The software uses unstructured tetrahedral meshes which are typically adapted to the used seismic velocity models. Additionally, modelers may introduce mountain topography. A sophisticated local time stepping scheme allows the solver to operate efficiently in very large and complex settings. The software is able to fuse multiple ensemble simulations into one execution of the software. EDGE uses an orthogonal polynomial expansion basis to discretize each of the considered variables in a tetrahedron of the mesh. In a typical setting, we use three relaxation mechanisms for the viscoelastic part, resulting in a total of 27 seismic variables. Additionally using a fifth order method gives us 35 basis functions, resulting in a total of $27 \cdot 35 = 945$ degrees of freedom per tetrahedral element. The solver advances the degrees of freedom in time by repeatedly computing a triplet of quadrature-free integrators. While the actual integrators are part of EDGE, their implementation relies heavily on TPPs. The GEMM-TPP with small and uncommon matrix sizes is the most crucial operation required by EDGE. For example, the surface integrator requires the multiplication of a 9×35 matrix with a 35×15 matrix. The solver's extension with additional, performance-portable TPPs in all parts of the integrators is work-in-progress. Especially, EDGE's support for viscoelastic attenuation or local time stepping requires "simpler" kernels, e.g., the unary TPPs Identity and Zero, or the binary TPPs Mul, Sub and Add.

We evaluate EDGE's performance-portability through the use of TPPs by studying the performance of a full setup of the Layer Over Halfspace 3 (LOH3) benchmark with 743,066 tetrahedral elements. The same setting was also used in Breuer and Heinecke [53] to study the performance of the solver on a single processor of the Frontera supercomputer located at the Texas Advanced Computing Center (position ten in the 06/21 TOP500-list). Following this study, a sophisticated simulation of the 2014 M_w 5.1 La Habra earthquake using a mesh with 237,861,634 tetrahedral elements and EDGE's advanced features yielded a performance of 2.20 FP32-PFLOPS on 1,536 nodes.

For the EDGE application, we study the software's raw floating point performance and time-to-solution by extending our LOH3-Frontera-only study [53] with diverse processors:

- *Cascade Lake* (similar to *CLX* as introduced in section 6): 2.7 GHz 28-core Intel Xeon Platinum 8,280 processor of the Frontera system at the Texas Advanced Computing Center. We only used a single 28-core processor of Frontera's dual-socketed compute nodes in our tests.
- *Ice Lake:* 2.3 GHz 40-core Intel Xeon Platinum 8,380 processor on Intel's on-premises cluster. We only used a single 40-core processor of the dual-socket compute nodes in our tests.
- *Rome* (similar to *ROME* as introduced in section 6): 2.25 GHz AMD EPYC 7,742 (BM.Standard.E3.128 OCI shape). We only used a single 64-core processor of the bare metal instance in our tests.
- *Milan:* 2.55 GHz AMD EPYC 7J13 (BM.Standard.E4.128 OCI shape). We only used single 64-core processor of the bare metal instance in our tests.
- *Altra:* 3.0 GHz Ampere Altra Q80-30 processor (BM.Standard.A1.160 OCI shape). We only used a single 80-Armv8.2-core processor of the bare metal instance in our tests.

TABLE 5 | Sustained 32-bit floating point performance on the studied systems.

System	GTS		LTS	
	Single	Fused[a]	Single	Fused[a]
Cascade lake	1.08	0.78	1.02	0.74
Ice lake	1.29	1.01	1.23	0.96
Rome	1.20	1.08	1.12	1.01
Milan	**1.39**	**1.16**	1.29	**1.07**
Altra	1.27	0.73	**1.51**	0.76

The performance is given in TFLOPS. Results are presented for Global Time Stepping (GTS) and Local Time Stepping (LTS) when using single and fused forward simulations.

[a] EDGE's fused simulations use sparse matrix kernels.

The bold values indicate the best performing system for each experimental setup.

Table 5 shows the sustained floating point performance of the conducted runs. All numbers are given in FP32-TFLOPS. Columns two and three present the performance of Global Time

TABLE 6 | Time-to-solution speedups of the studied systems when using different configurations of the solver EDGE.

System	GTS		LTS	
	Single	Fused	Single	Fused
Cascade lake	1.00	1.80	2.50	4.52
Ice lake	1.19	2.33	3.02	5.87
Rome	1.11	2.48	2.76	6.17
Milan	**1.28**	**2.67**	3.18	**6.55**
Altra	1.18	1.69	**3.71**	4.64

*The performance of the Cascade Lake system, running EDGE with Global Time Stepping (GTS) and a single forward simulation, is used as baseline. In contrast to **Table 5**, the speedups include the higher algorithmic efficiencies of EDGE's support for Local Time Stepping (LTS) and fused forward simulations.*
The bold values indicate the best performing system for each experimental setup.

Stepping (GTS), whereas columns four and five show that of Local Time Stepping (LTS). In general, the LTS configurations have a slightly lower peak utilization when compared to their GTS counterparts. Note, however, that **Table 5** only shows raw floating point performance and does not account for time-to-solution speedups through LTS (theoretically up to 2.67× in this case). The performance of GTS and LTS is further split into running a single forward simulation and fusing multiple simulations. In fused mode, the solver parallelizes over the right-hand-side by concurrently simulating seismic wave propagation for a collection of seismic sources. One of the fused mode's unique advantages is the opportunity for perfect vectorization of all small matrix multiplications, even when considering sparsity [52]. In this work, we matched the microarchitectures' SIMD-length by fusing 16 simulations on Cascade Lake and Ice Lake, eight simulations on Rome and Milan, and four simulations on Altra. Once again, note that **Table 5** does not include the respective sparsity-driven 2.49× increase of the floating point operations' value when running fused simulations. Comparing the performance of the different systems, we observe very high overall performance with architectural efficiency gains originating from decreasing SIMD-lengths. This is especially noticeable when running single forward simulations. In this case, the vectorized dimension of the small dense matrix kernels coincides with the number of basis functions, i.e., $M = 35$, which is challenging when optimizing for AVX512 (Cascade Lake and Ice Lake) and AVX2 (Rome and Milan). The short 128-bit ASIMD vector instruction (Altra) reach a very high peak utilization of 33.2% for GTS and 39.2% in LTS. For the fused simulations, the differences in relative peak utilization narrow further.

Table 6 describes the obtained performance numbers in terms of time-to-solution. Here, we use the runtime of the studied LOH3 setting on Cascade Lake for GTS and a single forward simulation as baseline. All other settings are given relative to this. Further, for the fused settings, we consider the per-simulation time. We observe that EDGE's overall performance is driven by the high floating point performance through the use of TPPs and the solver's advanced algorithmic features. Here, Altra performs best for single forward simulations using LTS, accelerating the baseline by 3.71×. Milan has the best time-to-solution in all other settings and is able to outperform the baseline by 6.55× when using LTS and fusing simulations. This performance lead originates from Milan's high theoretical peak combined with a high peak utilization (see **Table 5**).

9. RELATED WORK

The related work in terms of the development methodology of DL workloads has been referenced in the introduction, so here we mention community efforts that share the same design philosophy with TPPs. Tensor Operator Set Architecture (TOSA) is a recent work, concurrently developed with TPPs, that provides a set of whole-tensor operations commonly employed in DL [54]. TOSA allows users to express directly operators on up to 4D/5D tensors which are not naturally mapped even on contemporary 2D systolic hardware. We believe that staying at the 2D primitive level is expressive and sufficient, as we can build higher-order ops with loops around 2D operators, e.g., see **Algorithm 6**. Despite the similarities of TPP and TOSA specifications, the TOSA back-end is reference C code and is not showcased in full DL-workloads. CUTLASS [55] and Triton [56] strive for high-performance on GPUs, while also offer flexible composition that can be easily applied to solve new problems related in DL and linear algebra, and share many design principles with TPPs.

XLA [57] is a domain-specific compiler for linear algebra and DL that targets TensorFlow models with potentially no source code changes. JAX [58] provides automatic differentiation of Python and NumPy functions, and the compilation of the desired operators happens in a user-transparent way with JIT calls, yielding optimized XLA kernels. XLA and JAX share the same philosophy with TPPs: the user is focusing on the DL kernel/workload development using high-level, platform-agnostic, declarative-style programming, whereas the tensor-aware back-end infrastructure undertakes the efficient and portable code generation. Julia [59] is a high-level, dynamic programming language, designed to give users the speed of C/C++ while remaining easy to use. Since its incarnation, Julia has evolved with a strong Deep Learning/Machine Learning ecosystem, providing optimized libraries for such workloads. We envision that TPPs and tensor compilation frameworks (like JAX and Julia) will coexist in a synergistic fashion. For example, a program written in JAX could be lowered *via* an MLIR pass to the Linalg dialect, and from there the compilation stack could follow the path illustrated in **Figure 31** (JAX→Linalg→Affine/PXA→TPP) in order to leverage TPPs for efficient code generation. To this extend, Tensor Processing Primitives serve as a virtual tensor ISA within tensor compilation frameworks rather than trying to replace them.

Tensor Compilers (TC) [15–18] attempt to optimize DL operators in a platform-agnostics way, however their applicability is restricted to relatively small code-blocks whereas full workload integration is cumbersome. Also, TC undertake the tasks of efficient parallelization, loop re-ordering, automatic tiling and

layout transformations, nevertheless the obtained performance is typically underwhelming [12]. We envision that TPPs can be used as a tool by TC in order to attain efficient platform-specific code generation, therefore, TC could focus on optimizing the higher level aspects of the tensor programs (e.g., layout transformations). Along these lines, TPPs fit in the MLIR [20] ecosystem/stack as a lowering dialect (see section 7), and in this way the TPP back-end could be leveraged by multiple TC frameworks.

Tensor computations are also ubiquitous in HPC (e.g., physics, quantum chemistry, numerical simulations) and consequently a plethora of tensor computation frameworks have emerged to facilitate the development of such applications [60–64]. Typically these frameworks are comprised of a front-end that enables the expression of the underlying tensor computations (and can be domain-specific), and a back-end that optimizes the expressed computations using both high-level and low-level techniques. Since TPPs are agnostic of the user-entity, we envision that such tensor computation frameworks can leverage TPPs as a virtual tensor ISA instead of relying on generic compilers or low-level customized generators for efficient code generation across multiple platforms.

10. CONCLUSIONS AND FUTURE WORK

In this work, we presented the Tensor Processing Primitives (TPP), a compact, yet versatile set of 2D-tensor operators, which subsequently can be utilized as building-blocks to construct efficient, portable complex DL operators on high-dimensional tensors. We also show how TPPs can be used within HPC applications in order to accelerate tensor computations. We demonstrate the efficacy of our approach using standalone kernels and end-to-end training DL-workloads (CNNs, dilated convolutions, DLRM, BERT, GNNs) expressed entirely *via* TPPs that outperform state-of-the-art implementations on multiple platforms. As future work, we plan to create a full-fledged TPP-based MLIR dialect such that Tensor Compilers can leverage the strengths of TPPs. Also, we plan to further enrich the TPP back-end implementation by supporting more ISAs, including GPUs and POWER architectures.

11. OPTIMIZATION NOTICE

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations, and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information go to http://www.intel.com/performance.

Intel, Xeon, and Intel Xeon Phi are trademarks of Intel Corporation in the U.S. and/or other.

AUTHOR'S NOTE

This is an extended version of a technical paper presented at SC21 [65].

DATA AVAILABILITY STATEMENT

Publicly available datasets were analyzed in this study. This data can be found at: https://github.com/hfp/libxsmm/tree/sc21.

AUTHOR CONTRIBUTIONS

EG, AH, DKa, and SA worked on the design of the TPP framework. EG, AH, DKa, SA, MA, DA, CA, AB, AK, and BZ worked on the implementation of the TPP backend. EG, DKa, SA, NC, VM, SM, RM, and AH worked on the Deep Learning applications. AB and HP worked on the HPC applications. JB, DKu, FL, and BR worked on the Tensor Compiler integration. All authors read and approved the manuscript.

SUPPLEMENTARY MATERIAL

The Supplementary Material for this article can be found online at: https://www.frontiersin.org/articles/10.3389/fams.2022.826269/full#supplementary-material

REFERENCES

1. Krizhevsky A, Sutskever I, Hinton GE. ImageNet classification with deep convolutional neural networks. In: Pereira F, Burges CJC, Burges L, Weinberger KQ, editors. *Advances in Neural Information Processing Systems*. Curran Associates, Inc (2012). Available online at: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf
2. Szegedy C, Liu W, Jia Y, Sermanet P, Reed S, Anguelov D, et al. Going deeper with convolutions. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. Boston, MA (2015). p. 1–9.
3. Simonyan K, Zisserman A. Very deep convolutional networks for large-scale image recognition. *arXiv preprint* arXiv:14091556. (2014).
4. Yu D, Seltzer ML, Li J, Huang JT, Seide F. Feature learning in deep neural networks-studies on speech recognition tasks. *arXiv preprint* arXiv:13013605. (2013).
5. Wu Y, Schuster M, Chen Z, Le QV, Norouzi M, Macherey W, et al. Google's neural machine translation system: bridging the gap between human and machine translation. *arXiv preprint* arXiv:160908144. (2016).
6. Cheng HT, Koc L, Harmsen J, Shaked T, Chandra T, Aradhye H, et al. Wide & deep learning for recommender systems. In: *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems*. New York, NY: ACM (2016). p. 7–10.
7. Wolf T, Chaumond J, Debut L, Sanh V, Delangue C, Moi A, et al. Transformers: state-of-the-art natural language processing. In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. (2020). p. 38–45.
8. Gawehn E, Hiss JA, Schneider G. Deep learning in drug discovery. *Mol. Inf.* (2016) 35:3–14. doi: 10.1002/minf.201501008

9. Goh GB, Hodas NO, Vishnu A. Deep learning for computational chemistry. *J Comput Chem.* (2017) 38:1291–307. doi: 10.1002/jcc.24764
10. Raghu M, Schmidt E. A survey of deep learning for scientific discovery. *arXiv preprint* arXiv:200311755. (2020).
11. Alom MZ, Taha TM, Yakopcic C, Westberg S, Sidike P, Nasrin MS, et al. A state-of-the-art survey on deep learning theory and architectures. *Electronics.* (2019) 8:292. doi: 10.3390/electronics8030292
12. Barham P, Isard M. Machine learning systems are stuck in a rut. In: *Proceedings of the Workshop on Hot Topics in Operating Systems.* New York, NY: (2019). p. 177–83.
13. oneDNN. *Intel oneDNN GitHub.* Available online at: https://github.com/oneapi-src/oneDNN (accessed online at March 30, 2021).
14. Chetlur S, Woolley C, Vandermersch P, Cohen J, Tran J, Catanzaro B, et al. cudnn: efficient primitives for deep learning. *arXiv preprint* arXiv:14100759. (2014).
15. Zerrell T, Bruestle J. Stripe: tensor compilation via the nested polyhedral model. *arXiv preprint* arXiv:190306498. (2019).
16. Chen T, Moreau T, Jiang Z, Zheng L, Yan E, Shen H, et al. {TVM}: an Automated End-to-End Optimizing Compiler for Deep Learning. In: *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18).* Berkeley, CA (2018). p. 578–94.
17. Vasilache N, Zinenko O, Theodoridis T, Goyal P, DeVito Z, Moses WS, et al. Tensor comprehensions: framework-agnostic high-performance machine learning abstractions. *arXiv preprint* arXiv:180204730. (2018).
18. Zheng L, Jia C, Sun M, Wu Z, Yu CH, Haj-Ali A, et al. Ansor: generating high-performance tensor programs for deep learning. In: *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20).* Berkeley, CA (2020). p. 863–79.
19. Li M, Liu Y, Liu X, Sun Q, You X, Yang H, et al. The deep learning compiler: a comprehensive survey. *IEEE Trans Parallel Distrib Syst.* (2020) 32:708–27. doi: 10.1109/TPDS.2020.3030548
20. MLIR. *Multi-Level Intermediate Representation GitHub.* Available online at: https://github.com/tensorflow/mlir (accessed online at March 30, 2021).
21. Georganas E, Banerjee K, Kalamkar D, Avancha S, Venkat A, Anderson M, et al. Harnessing deep learning via a single building block. In: *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS).* New Orleans, LA: IEEE (2020). p. 222–33.
22. Heinecke A, Henry G, Hutchinson M, Pabst H. LIBXSMM: accelerating small matrix multiplications by runtime code generation. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC '16.* Piscataway, NJ: IEEE Press (2016). p. 84:1–84:11.
23. Georganas E, Avancha S, Banerjee K, Kalamkar D, Henry G, Pabst H, et al. Anatomy of high-performance deep learning convolutions on simd architectures. In: *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis.* Dallas, TX: IEEE (2018). p. 830–41.
24. Bfloat16. *Using bfloat16 With TensorFlow Models.* Available online at: https://cloud.google.com/tpu/docs/bfloat16 (accessed online at April 3, 2019).
25. Marsaglia G, et al. Xorshift rngs. *J Stat Softw.* (2003) 8:1–6. doi: 10.18637/jss.v008.i14
26. Banerjee K, Georganas E, Kalamkar DD, Ziv B, Segal E, Anderson C, et al. Optimizing deep learning rnn topologies on intel architecture. *Supercomput Front Innov.* (2019) 6:64–85. doi: 10.14529/jsfi190304
27. Intel-ISA. *Intel Architecture Instruction Set Extensions and Future Features Programming Reference.* Available online at: https://software.intel.com/content/dam/develop/public/us/en/documents/architecture-instruction-set-extensions-programming-reference.pdf (accessed online at March 30, 2021)
28. *city Powell MJD. *Approximation Theory and Methods.* Cambridge University Press. (1981).
29. Chebyshev-Polynomials. *Chebyshev Polynomials.* Available online at: https://en.wikipedia.org/wiki/Chebyshev_polynomials (accessed online at September 26, 2021)
30. Flajolet P, Raoult JC, Vuillemin J. The number of registers required for evaluating arithmetic expressions. *Theor Comput Sci.* (1979) 9:99–125.
31. Gibbs JW. *Elementary Principles in Statistical Mechanics: Developed with Especial Reference to the Rational Foundation of Thermodynamics.* Cambridge: Cambridge University Press (2010). doi: 10.1017/CBO9780511686948
32. Ioffe S, Szegedy C. Batch normalization: accelerating deep network training by reducing internal covariate shift. In: *International Conference on Machine Learning.* Lille: PMLR. (2015). p. 448–56.
33. He K, Zhang X, Ren S, Sun J. Deep residual learning for image recognition. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* (2016). p. 770–78.
34. Ba JL, Kiros JR, Hinton GE. Layer normalization. *arXiv preprint* arXiv:160706450. (2016).
35. Wu Y, He K. Group normalization. In: *Proceedings of the European Conference on Computer Vision (ECCV).* Munich (2018). p. 3–19.
36. Ulyanov D, Vedaldi A, Lempitsky V. Instance normalization: The missing ingredient for fast stylization. *arXiv preprint* arXiv:160708022. (2016).
37. Kalamkar D, Georganas E, Srinivasan S, Chen J, Shiryaev M, Heinecke A. Optimizing deep learning recommender systems training on CPU cluster architectures. In: *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis.* Atlanta, GA: IEEE (2020). p. 1–15.
38. Lal A, Chiang ZD, Yakovenko N, Duarte FM, Israeli J, Buenrostro JD. AtacWorks: a deep convolutional neural network toolkit for epigenomics. *bioRxiv.* (2019).
39. Naumov M, Mudigere D, Shi HJM, Huang J, Sundaraman N, Park J, et al. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint* arXiv:190600091. (2019).
40. Devlin J, Chang MW, Lee K, Toutanova K. Bert: pre-training of deep bidirectional transformers for language understanding. *arXiv preprint* arXiv:181004805. (2018).
41. Wolf T, Debut L, Sanh V, Chaumond J, Delangue C, Moi A, et al. Transformers: state-of-the-art natural language processing. In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations.* Association for Computational Linguistics (2020). p. 38–45.
42. Zhang M, Rajbhandari S, Wang W, He Y. Deepcpu: serving rnn-based deep learning models 10x faster. In: *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18).* Boston, MA (2018). p. 951–65.
43. Hamilton WL, Ying R, Leskovec J. Inductive representation learning on large graphs. *arXiv preprint* arXiv:170602216. (2017).
44. Avancha S, Md V, Misra S, Mohanty R. Deep Graph Library Optimizations for Intel (R) x86 Architecture. *arXiv preprint* arXiv:200706354. (2020).
45. oneDNN Fugaku. *A Deep Dive Into a Deep Learning Library for the A64FX Fugaku CPU - The Development Story in the Developer's Own Words Fujitsu.* Available online at: https://blog.fltech.dev/entry/2020/11/19/fugaku-onednn-deep-dive-en (accessed online at April 9, 2021).
46. Hugging-Faces. *Hugging Faces GitHub.* Available online at: https://github.com/huggingface/transformers (accessed online at April 9, 2021).
47. oneCCL. *Intel oneCCL GitHub.* Available online at: https://github.com/oneapi-src/oneCCL (accessed online at March 30, 2021).
48. Xie S, Girshick R, Dollár P, Tu Z, He K. Aggregated residual transformations for deep neural networks. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* Honolulu, HI (2017). p. 1492–500.
49. Carreira J, Zisserman A. Quo vadis, action recognition? a new model and the kinetics dataset. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* Honolulu, HI (2017). p. 6299–308.
50. Kühne TD, Iannuzzi M, Del Ben M, Rybkin VV, Seewald P, Stein F, et al. CP2K: an electronic structure and molecular dynamics software package - Quickstep: efficient and accurate electronic structure calculations. *J Chem Phys.* (2020) 152:194103. doi: 10.1063/5.0007045
51. Sivkov I, Seewald P, Lazzaro A, Hutter J. DBCSR: a blocked sparse tensor algebra library. *CoRR.* (2019) abs/1910.13555.
52. Breuer A, Heinecke A, Cui Y. EDGE: extreme scale fused seismic simulations with the discontinuous galerkin method. In: Kunkel JM, Yokota R, Balaji P, Keyes D, editors. *High Performance Computing.* Cham: Springer International Publishing. (2017). p. 41–60.
53. Breuer A, Heinecke A. *Next-Generation Local Time Stepping for the ADER-DG Finite Element Method. arXiv[Preprint].* (2022). arXiv: 2202.10313. Available online at: https://arxiv.org/abs/2202.10313
54. TOSA. *TOSA.* Available online at: https://developer.mlplatform.org/w/tosa/ (accessed online at March 30, 2021).

55. CUTLASS. *NVIDIA CUTLASS GitHub.* Available online at: https://github.com/NVIDIA/cutlass (accessed online at March 30, 2021).
56. Tillet P, Kung H, Cox D. Triton: an intermediate language and compiler for tiled neural network computations. In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages.* New York, NY (2019). p. 10–9.
57. XLA. *XLA: Optimizing Compiler for Machine Learning.* Available online at: https://www.tensorflow.org/xla (accessed online at March 30, 2021).
58. JAX. *JAX: Autograd and XLA.* Available online at: https://github.com/google/jax (accessed online at March 30, 2021).
59. Bezanson J, Edelman A, Karpinski S, Shah VB. Julia: a fresh approach to numerical computing. *SIAM Rev.* (2017) 59, 65–98. doi. 10.1137/141000671
60. Solomonik E, Matthews D, Hammond JR, Stanton JF, Demmel J. A massively parallel tensor contraction framework for coupled-cluster computations. *J Parallel Distrib Comput.* (2014) 74:3176–90. doi: 10.1016/j.jpdc.2014.06.002
61. Solomonik E, Hoefler T. Sparse tensor algebra as a parallel programming model. *arXiv preprint* arXiv:151200066. (2015).
62. Springer P, Bientinesi P, Wellein G. High-performance tensor operations: tensor transpositions, spin summations, and tensor contractions. *Fachgruppe Informatik.* (2019). Available online at: http://publications.rwth-aachen.de/record/755345/files/755345.pdf
63. Hirata S. Tensor contraction engine: abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories. *J Phys Chem A.* (2003) 107:9887–97. doi: 10.1021/jp034596z
64. Epifanovsky E, Wormit M, Kuś T, Landau A, Zuev D, Khistyaev K, et al. *New Implementation of High-Level Correlated Methods Using a General Block Tensor Library for High-Performance Electronic Structure Calculations.* Wiley Online Library. (2013). Available online at: https://onlinelibrary.wiley.com/doi/10.1002/jcc.23377
65. Georganas E, Kalamkar D, Avancha S, Adelman M, Anderson C, Breuer A, et al. Tensor processing primitives: a programming abstraction for efficiency and portability in deep learning workloads. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* New York, NY (2021). p. 1–14.

Conflict of Interest: EG, DKa, SA, MA, DA, CA, JB, NC, AK, DKu, FL, VM, SM, RM, HP, BR, BZ, and AH were employed by Intel Corporation.

The remaining author declares that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's Note: All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher.

GLOSSARY

Intel Pseudo Intrinsics

1. ***_mm128*** Represents a vector of width 128 bits.
2. ***_mm128_loadu_ps(addr)*** Loads 16byte of 32 bit elements.
3. ***_mm128_storeu_ps(addr)*** Stores 16byte of 32 bit elements.
4. ***_mm128_unpacklo_ps(A, B)*** Unpacks and interleaves 32 bit elements from the low half of A and B.
5. ***_mm128_unpackhi_ps(A, B)*** Unpacks and interleaves 32 bit elements from the high half of A and B.
6. ***_mm128_unpacklo_pd(A. B)*** Unpacks and interleaves 64 bit elements from the low half of A and B.
7. ***_mm128_unpackhi_pd(A, B)*** Unpacks and interleaves 64 bit elements from the high half of A and B.
8. ***_mm512*** Represents a vector of width 512 bits.
9. ***_mm512_permutexvar_ps(A,B)*** Shuffle single precision floating point elements in 512 wide vector length using indexes specified in B.
10. ***_mm512_roundscale_ps(A,B)*** Round single precision floating point elements to the rounding mode specified by argument B.
11. ***_mm512_sub_ps(A,B)*** Subtract single precision floating point elements in A from B.
12. ***_mm512_scalef_ps(A,B)*** Scales single precision floating point elements in A using values specified in B.
13. ***_mm512_range_ps(A,B, int imm8)*** Calculates the min, max or absolute max for each single precision- floating point elements in A and B. Lower 2 bits of imm8[1:0] specifies the operation(min/max/absolute max) to be performed.
14. ***_mm512_xor_ps(A,B)*** Performs XOR operation between each single precision floating point elements in A and B vector.
15. ***_mm512_and_ps(A,B)*** Performs AND operation between each single precision floating point elements in A and B vector.
16. ***_mm512_rcp14_ps(A,B)*** Calculates approximate reciprocal of each single precision floating point element in range less then 2^-14.
17. ***_mm512_cmp_ps_mask(A,B,int C)*** Compare the single precision elements in A and B specified by the comparison mode in C.
18. ***_mm512_mask_blend_ps(mask A,B,C)*** Copies single precision floating point element from vector A in vector C if the corresponding mask bit is set.
19. ***_mm512_fmadd_ps(mask A,B,C)*** Fused-Multiply-Add: Multiplies elements from vector A and B and adds them to elements of vector C.
20. ***_mm512_maskz_loadu_epi16(mask, addr)*** Loads 64byte of 16bit elements under zero masking from address addr.
21. ***_mm512_set1_epi32(value)*** sets a 32 bit value into all 16 entries of the vector, e.g. broadcast.
22. ***_mm512_maskz_mov_epi16(mask, A)*** Moves 16 bit-type register A under zero-masking to a different register.
23. ***_mm512_slli_epi32(A, imm)*** Shifts all entries in the vector registers (typed as 32 bit elements) by value imm to the left by shifting 0 in.

Arm Pseudo Intrinsics

1. ***vld1q_f32(addr)*** Loads 16byte of 32 bit elements.
2. ***vst1q_f32(addr)*** Loads 16byte of 32 bit elements.
3. ***vtrn1q_f32(A, B)*** Unpacks and interleaves 32 bit elements from the low half of A and B.
4. ***vtrn2q_f32(A, B)*** Unpacks and interleaves 32 bit elements from the high half of A and B.
5. ***vtrn1q_f64(A. B)*** Unpacks and interleaves 64 bit elements from the low half of A and B.
6. ***vtrn2q_f64(A, B)*** Unpacks and interleaves 64 bit elements from the high half of A and B.
7. ***vmax_q(A,B)*** Calculates the maximum between each single precision floating point elements in A and B vector.
8. ***vmin_q(A,B)*** Calculates the minimum between each single precision floating point elements in A and B vector.
9. ***vmul_q(A,B)*** Multiply single precision elements in A and B vector.
10. ***vsub_q(A,B)*** Subtract corresponding single precision elements in B from A.
11. ***vadd_q(A,B)*** Add single precision elements in B and A.
12. ***vshlq_u32(A,B)*** Shift left each single precision elements in A by the value specified in B.
13. ***vrndmq_f32(A)*** Round single precision floating point elements in A using minus infinity rounding mode.
14. ***vcvtmq_s32_f32(A)*** Converts single precision floating point elements in A to signed integers using minus infinity rounding mode.
15. ***float32x4_t*** Represents 4 single precision floating point elements in vector width of 128.
16. ***vand_q(A,B)*** Performs bit-wise AND operation between A and B vector.
17. ***vfmaq_f32(A,B,C)*** Multiply single precision elements in A and B.Add the intermediate result to C.
18. ***vld1q_f32(A)*** Load a single precision element from scalar to all single precision element in a vector.
19. ***vtbl1_u8(A,B)*** Performs a byte look up operation in vector A using byte addressable indexes specified in vector B.
20. ***vtbl4_u8(A,B)*** Performs a 64 byte look up operation in vector A, A+1, A+2, A+3 using byte addressable indexes specified in vector B.
21. ***vbcaxq_s32(A,B)*** Performs XOR operation between each single precision floating point elements in A and B vector.
22. ***vcgt_q(A,B)*** Compare corresponding single precision elements in A and B. If B is greater then A the corresponding bits are set in the destination vector.
23. ***vrecpe_f32(A)*** Calculates approximate reciprocal of each single precision floating point element in vector A.
24. ***vbit_insert(A,B)*** Copies single precision floating point element from vector A in destination vector if the corresponding bits are set in vector B.

frontiers | Frontiers in Applied Mathematics and Statistics

ORIGINAL RESEARCH
published: 19 April 2022
doi: 10.3389/fams.2022.826988

Ubiquitous Nature of the Reduced Higher Order SVD in Tensor-Based Scientific Computing

Venera Khoromskaia and Boris N. Khoromskij*

Max-Planck-Institute for Mathematics in the Sciences, Leipzig, Germany

Tensor numerical methods, based on the rank-structured tensor representation of d-variate functions and operators discretized on large $n^{\otimes d}$ grids, are designed to provide $O(dn)$ complexity of numerical calculations contrary to $O(n^d)$ scaling by conventional grid-based methods. However, multiple tensor operations may lead to enormous increase in the tensor ranks (curse of ranks) of the target data, making calculation intractable. Therefore, one of the most important steps in tensor calculations is the robust and efficient rank reduction procedure which should be performed many times in the course of various tensor transforms in multi-dimensional operator and function calculus. The rank reduction scheme based on the Reduced Higher Order SVD (RHOSVD) introduced by the authors, played a significant role in the development of tensor numerical methods. Here, we briefly survey the essentials of RHOSVD method and then focus on some new theoretical and computational aspects of the RHOSVD and demonstrate that this rank reduction technique constitutes the basic ingredient in tensor computations for real-life problems. In particular, the stability analysis of RHOSVD is presented. We introduce the multi-linear algebra of tensors represented in the range-separated (RS) tensor format. This allows to apply the RHOSVD rank-reduction techniques to non-regular functional data with many singularities, for example, to the rank-structured computation of the collective multi-particle interaction potentials in bio-molecular modeling, as well as to complicated composite radial functions. The new theoretical and numerical results on application of the RHOSVD in scattered data modeling are presented. We underline that RHOSVD proved to be the efficient rank reduction technique in numerous applications ranging from numerical treatment of multi-particle systems in material sciences up to a numerical solution of PDE constrained control problems in $\mathbb{R}^d$.

Keywords: low-rank tensor product approximation, multi-variate functions, tensor calculus, rank reduction, tucker format, canonical tensors, interaction potentials, scattered data modeling

OPEN ACCESS

Edited by:
Edoardo Angelo Di Napoli, Helmholtz Association of German Research Centres (HZ), Germany

Reviewed by:
Maxim Rakhuba, National Research University Higher School of Economics, Russia
Katharina Kormann, Uppsala University, Sweden
Akwum Onwunta, Lehigh University, United States

****Correspondence:***
Venera Khoromskaia
vekh@mis.mpg.de

Specialty section:
This article was submitted to Mathematics of Computation and Data Science, a section of the journal Frontiers in Applied Mathematics and Statistics

Received: *01 December 2021*
Accepted: *03 March 2022*
Published: *19 April 2022*

Citation:
Khoromskaia V and Khoromskij BN (2022) Ubiquitous Nature of the Reduced Higher Order SVD in Tensor-Based Scientific Computing. Front. Appl. Math. Stat. 8:826988. doi: 10.3389/fams.2022.826988

1. INTRODUCTION

The mathematical models in large-scale scientific computing are often described by steady state or dynamical PDEs. The underlying physical, chemical or biological systems usually live in 3D physical space $\mathbb{R}^3$ and may depend on many structural parameters. The solution of arising discrete systems of equations and optimization of the model parameters lead to the challenging numerical

problems. Indeed, the accurate grid-based approximation of operators and functions involved requires large spatial grids in $\mathbb{R}^d$, resulting in considerable storage space and implementation of various algebraic operations on huge vectors and matrices. For further discussion we shall assume that all functional entities are discretized on $n^{\otimes d}$ spatial grids where the univariate grid size n may vary in the range of several thousands. The linear algebra on N-vectors and $N \times N$ matrices with $N = n^d$ quickly becomes non-tractable as n and d increase.

Tensor numerical methods [1, 2] provide means to overcome the problem of the exponential increase of numerical complexity in the dimension of the problem d, due to their intrinsic feature of reducing the computational costs of multi-linear algebra on rank-structured data to merely linear scaling in both the grid-size n and dimension d. They appeared as bridging of the algebraic tensor decompositions initiated in chemometrics [3–10] and of the nonlinear approximation theory on separable low-rank representation of multi-variate functions and operators [11–13]. The canonical [14, 15], Tucker [16], tensor train (TT) [17, 18], and hierarchical Tucker (HT) [19] formats are the most commonly used rank-structured parametrizations in applications of modern tensor numerical methods. Further data-compression to the logarithmic scale can be achieved by using the quantized-TT (QTT) [20, 21] tensor approximation. At present there is an active research toward further progress of tensor numerical methods in scientific computing [1, 2, 22–26]. In particular, there are considerable achievements of tensor-based approaches in computational chemistry [27–31], in bio-molecular modeling [32–35], in optimal control problems (including the case of fractional control) [36–39], and in many other fields [6, 40–44].

Here, we notice that tensor numerical methods proved to be efficient when all input data and all intermediate quantities within the chosen computational scheme are presented in a certain low-rank tensor format with controllable rank parameters, i.e., on low-rank tensor manifolds. In turn, tensor decomposition of the full format data arrays is considered as an N-P hard problem. For example, the truncated HOSVD [7] of an $n^{\otimes d}$-tensor in the Tucker format amounts to $O(n^{d+1})$ arithmetic operations while the respective cost of the TT and HT higher-order SVD [18, 45] is estimated by $O(n^{\frac{3}{2}d})$, indicating that rank decomposition of full format tensors still suffers from the "curse of dimensionality" and practically could not be applied in large scale computations.

On the other hand, often, the initial data for complicated numerical algorithms may be chosen in the canonical/Tucker tensor formats, say as a result of discretization of a short sum of Gaussians or multi-variate polynomials, or as a result of the analytical approximation by using Laplace transform representation and sinc-quadratures [1]. However, the ranks of tensors are multiplied in the course of various tensor operations, leading to dramatic increase in the rank parameter ("curse of ranks") of a resulting tensor, thus making tensor-structured calculation intractable. Therefore, fast and stable rank reduction schemes are the main prerequisite for the success of rank-structured tensor techniques.

Invention of the Reduced Higher Order SVD (RHOSVD) in [46] and the corresponding rank reduction procedure based on the canonical-to-Tucker transform and subsequent canonical approximation of the small Tucker core (Tucker-to-canonical transform) was a decisive step in development of the tensor numerical methods in scientific computing. In contrast to the conventional HOSVD, the RHOSVD does not need a construction of the full size tensor for finding the orthogonal subspaces of the Tucker tensor representation. Instead, RHOSVD applies to numerical data in the canonical tensor format (with possibly large initial rank R) and exhibits the $O(dnR\min\{n, R\})$ complexity, uniformly in the dimensionality of the problem, d, and it was an essential step ahead in evolution of the tensor-structured numerical techniques.

In particular, this rank reduction scheme was applied to calculation of 3D and 6D convolution integrals in tensor-based solution of the Hartree-Fock equation [27, 46]. Combined with the Tucker-to-canonical transform, this algorithm provides a stable procedure for the rank reduction of possibly huge ranks in tensor-structured calculations of the Hartree potential. The RHOSVD based rank reduction scheme for the canonical tensors is specifically useful for 3D problems, which are most often in real-life applications. However, the RHOSVD-type procedure can be also efficiently applied in the construction of the TT tensor format from the canonical tensor input, which often appears in tensor calculations[1].

The RHOSVD is the basic tool for the construction of the range-separated (RS) tensor format introduced in [32] for the low-rank tensor representation of the bio-molecular long-range electrostatic potentials. Recent example on the RS representation of the multi-centered Dirac delta function [34] paves the way for efficient solution decomposition scheme introduced for the Poisson-Boltzmann equation [33, 35].

In some applications the data could be presented as a sum of highly localized and rank-structured components so that their further numerical treatment again requires the rank reduction procedure (see Section 4.5 concerning the long-range potential calculation for many-particle system). Here, we present the constructive description of multi-linear operations on tensors in RS format which allow to compute the short- and long-range parts of resulting combined tensors. In particular, this applies to commonly used addition of tensors, Hadamard and contracted products as well as to composite functions of RS tensors. We then introduce tensor-based modeling of the scattered data by a sum of Slater kernels and show the existence of the low-rank representation for such data in the RS tensor format. The numerical examples demonstrate the practical efficiency of such kind of tensor interpolation. This approach may be efficiently used in many applications in data science and in stochastic data modeling.

Rank reduction procedure by using the RHOSVD is a mandatory part in solving the three-dimensional elliptic and pseudo-differential equations in the rank-structured tensor format. In the course of preconditioned iterations, the tensor

[1]Otherwise one can not avoid the "curse of dimensionality", see the cost of the HT/TT SVD above.

ranks of the governing operator, the precoditioner and of the current iterand are multiplied at each iterative step, and, therefore, a fast and robust rank reduction techniques is the prerequisite for such methodology applied in the framework of iterative elliptic problem solvers. In particular, this approach was applied to the PDE constrained (including the case of fractional operators) optimal control problems [36, 39]. As result, the computational complexity can be reduced to almost linear scale, $O(nR)$, contrary to conventional $O(n^3)$ complexity, as demonstrated by numerics in [36, 39].

Tensor-based algorithms and methods are now being widely used and developed further in the communities of scientific computing and data science. Tensor techniques evolve in traditional tensor decompositions in data processing [5, 42, 47], and they are actively promoted for tensor-based solution of the multi-dimensional problems in numerical analysis and quantum chemistry [1, 24, 29, 38, 39, 48, 49]. Notice that in the case of higher dimensions the rank reduction in the canonical format can be performed directly (i.e., without intermediate use of the Tucker approximation) by using the cascading ALS iteration in the CP format (see [50] concerning the tensor-structured solution of the stochastic/parametric PDEs).

The rest of this article is organized as follows. In Section 2, we sketch some results on the construction of the RHOSVD and present some old and new results on the stability of error bounds. In Section 2.2, we recollect the mixed canonical-Tucker tensor format and the Tucker-to-canonical transform. Section 3 recalls the results from Khoromskaia [27] on calculation of the multi-dimensional convolution integrals with the Newton kernel arising in computational quantum chemistry. Section 4 addresses the application of RHOSVD to RS parametrized tensors. In Section 4.2, we discuss the application of RHOSVD in multilinear operations of data in the RS tensor format. The scattered data modeling is considered in section 4.5 from both theoretical and computational aspects. Application of RHOSVD for tensor-based representation of Greens kernels is discussed in Section 5. Section 6 gives a short sketch of RHOSVD in application to tensor-structured elliptic problem solvers.

2. REDUCED HOSVD AND CP-TO-TUCKER TRANSFORM

2.1. Reduced HOSVD: Error Bounds

In computational schemes including bilinear tensor-tensor or matrix-tensor operations the increase of tensor ranks leads to the critical loss of efficiency. Moreover, in many applications, for example in electronic structure calculations, the canonical tensors with large rank parameters arise as the result of polynomial type or convolution transforms of some function related tensors (say, electron density, the Hartree potential, etc.) In what follows, we present the new look on the direct method of rank reduction for the canonical tensors with large initial rank, the reduced HOSVD, first introduced and analyzed in [46].

In what follows, we consider the vector space of d-fold real-valued data arrays $\mathbb{R}^{n_1\times\cdots\times n_d}$ endorsed by the Euclidean scalar product $\langle\cdot,\cdot\rangle$ with the related norm $\|u\| = \langle u,u\rangle^{1/2}$. We denote by $\mathcal{T}_{\mathbf{r},\mathbf{n}}$ the class of tensors $\mathbf{A}\in\mathbb{R}^{n\otimes d}$ parametrized in the rank-$\mathbf{r}$, $\mathbf{r}=(r_1,\ldots,r_d)$ orthogonal Tucker format,

$$\mathbf{A} = \boldsymbol{\beta}\times_1 V^{(1)}\times_2\cdots\times_d V^{(d)}\in\mathcal{T}_{\mathbf{r},\mathbf{n}},$$

with the orthogonal side-matrices $V^{(\ell)} = [\mathbf{v}_1^{(\ell)}\ldots\mathbf{v}_{r_\ell}^{(\ell)}]\in\mathbb{R}^{n\times r_\ell}$ and with the core coefficient tensor $\boldsymbol{\beta}\in\mathbb{R}^{r_1\times\ldots\times r_d}$. Here and thereafter $\times_\ell$ denotes the contracted tensor-matrix product in the dimension ℓ, and $\mathbb{R}^{n\otimes d}$ denotes the Euclidean vector space of $n_1\times\cdots\times n_d$-tensors with equal mode size $n_\ell = n$, $\ell = 1,\ldots,d$.

Likewise, $\mathcal{C}_{R,\mathbf{n}}$ denotes the class of rank-R canonical tensors. For given $\mathbf{A}\in\mathcal{C}_{R,\mathbf{n}}$ in the rank-R canonical format,

$$\mathbf{A} = \sum\nolimits_{\nu=1}^{R}\xi_\nu\mathbf{u}_\nu^{(1)}\otimes\ldots\otimes\mathbf{u}_\nu^{(d)},\quad \xi_\nu\in\mathbb{R}, \tag{1}$$

with normalized canonical vectors, i.e., $\|\mathbf{u}_\nu^{(\ell)}\| = 1$ for $\ell = 1,\ldots,d$, $\nu = 1,\ldots,R$.

The standard algorithm for the Tucker tensor decomposition [7] is based on HOSVD applied to full tensors of size n^d which exhibits $O(n^{d+1})$ computational complexity. The question is how to simplify the HOSVD Tucker approximation in the case of canonical input tensor in the form Equation (1) without use of the full format representation of $\mathbf{A}$, and in the situation when the CP rank parameter R and the mode sizes n of the input can be sufficiently large.

First, let us use the equivalent (nonorthogonal) rank-$\mathbf{r} = (R,\ldots,R)$ Tucker representation of the tensor Equation (1),

$$\mathbf{A} = \boldsymbol{\xi}\times_1 U^{(1)}\times_2 U^{(2)}\cdots\times_d U^{(d)},\quad \boldsymbol{\xi} = \mathrm{diag}\{\xi_1,\ldots,\xi_R\}, \tag{2}$$

via contraction of the diagonal tensor $\boldsymbol{\xi} = \mathrm{diag}\{\xi_1,\ldots,\xi_R\}\in\mathbb{R}^{R\otimes d}$ with ℓ-mode side matrices $U^{(\ell)} = [\mathbf{u}_1^{(\ell)},\ldots,\mathbf{u}_R^{(\ell)}]\in\mathbb{R}^{n\times R}$ (see **Figure 1**). By definition the tensor $\mathrm{diag}\{\xi_1,\ldots,\xi_R\}\in\mathbb{R}^{R\otimes d}$ is called diagonal if it has all zero entries except the diagonal elements given by $\boldsymbol{\xi}(i_\ell,\ldots,i_\ell) = \xi_{i_\ell}$, $\ell = 1,\ldots,R$. Then the problem of canonical to Tucker approximation can be solved by the method of reduced HOSVD (RHOSVD) introduced in [46]. The basic idea of the reduced HOSVD is that for large (function related) tensors given in the canonical format their HOSVD does not require the construction of a tensor in the full format and SVD based computation of its matrix unfolding. Instead, it is sufficient to compute the SVD of the directional matrices $U^{(\ell)}$ in Equation (2) composed by only the vectors of the canonical tensor in every dimension separately, as shown in **Figure 1**. This will provide the initial guess for the Tucker orthogonal basis in the given dimension. For the practical applicability, the results of the approximation theory on the low-rank approximation to the multi-variate functions, exhibiting exponential error decay in the Tucker rank, are of the principal significance [51].

In the following, we suppose that $n\le R$ and denote the SVD of the side-matrix $U^{(\ell)}$ by

$$U^{(\ell)} = Z^{(\ell)}D_\ell V^{(\ell)^T} = \sum_{k=1}^{n}\sigma_{\ell,k}\mathbf{z}_k^{(\ell)}\,\mathbf{v}_k^{(\ell)^T},\quad \mathbf{z}_k^{(\ell)}\in\mathbb{R}^n,\ \mathbf{v}_k^{(\ell)}\in\mathbb{R}^R, \tag{3}$$

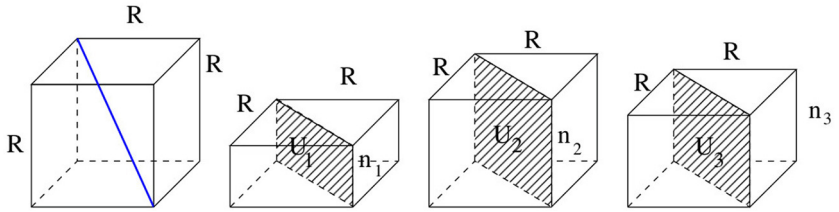


FIGURE 1 | Illustration to the contracted product representation Equation (2) of the rank-R canonical tensor. The first factor corresponds to the diagonal coefficient tensor $\boldsymbol{\xi}$.

with the orthogonal matrices $Z^{(\ell)} = [\mathbf{z}_1^{(\ell)}, \ldots, \mathbf{z}_n^{(\ell)}] \in \mathbb{R}^{n\times n}$, and $V^{(\ell)} = [\mathbf{v}_1^{(\ell)}, \ldots, \mathbf{v}_n^{(\ell)}] \in \mathbb{R}^{R\times n}$, $\ell = 1, \ldots, d$. We use the following notations for the vector entries, $v_k^{(\ell)}(\nu) = v_{k,\nu}^{(\ell)}$ $(\nu = 1, \ldots, R)$.

To fix the idea, we introduce the vector of rank parameters, $\mathbf{r} = (r_1, \ldots, r_d)$, and let

$$U^{(\ell)} \approx W^{(\ell)} := Z_0^{(\ell)} D_{\ell,0} V_0^{(\ell)\,T}, \tag{4}$$

be the rank-r_ℓ truncated SVD of the side-matrix $U^{(\ell)}$ $(\ell = 1, \ldots, d)$. Here, the matrix $D_{\ell,0} = \mathrm{diag}\{\sigma_{\ell,1}, \sigma_{\ell,2}, \ldots, \sigma_{\ell,r_\ell}\}$ is the submatrix of D_ℓ in Equation (3) and

$$Z_0^{(\ell)} = [\mathbf{z}_1^{(\ell)}, \ldots, \mathbf{z}_{r_\ell}^{(\ell)}] \in \mathbb{R}^{n\times r_\ell},\ V_0{}^{(\ell)} \in \mathbb{R}^{R\times r_\ell},$$

represent the respective dominating $(n \times r_\ell)$-submatrices of the left and right factors in the complete SVD decomposition in Equation (3).

Definition 2.1. *(Reduced HOSVD, [46]). Given the canonical tensor* $\mathbf{A} \in \mathcal{C}_{R,\mathbf{n}}$*, the truncation rank parameter* $\mathbf{r}$*,* $(r_\ell \le R)$*, and rank-*r_ℓ *truncated SVD of* $U^{(\ell)}$*, see Equation (4), then the RHOSVD approximation of* $\mathbf{A}$ *is defined by the rank-*$\mathbf{r}$ *orthogonal Tucker tensor*

$$\begin{aligned}\mathbf{A}_{(\mathbf{r})}^0 &:= \boldsymbol{\xi} \times_1 W^{(1)} \times_2 \cdots \times_d W^{(d)} = \boldsymbol{\xi} \times_1 \left[Z_0^{(1)} D_{1,0} V_0^{(1)\,T}\right] \\ &\times_2 \cdots \times_d \left[Z_0^{(d)} D_{d,0} V_0^{(d)\,T}\right] \\ &= \left(\boldsymbol{\xi} \times_1 [D_{1,0} V_0^{(1)\,T}] \times_2 \cdots \times_d [D_{d,0} V_0^{(d)\,T}]\right) \times_1 Z_0^{(1)} \\ &\times_2 \cdots \times_d Z_0^{(d)} \in \mathcal{T}_{\mathbf{r}}, \end{aligned} \tag{5}$$

obtained by the projection of canonical side matrices $U^{(\ell)}$ *onto the left orthogonal singular matrices* $Z_0^{(\ell)}$*, defined in Equation (4).*

Notice that the general error bound for the RHOSVD approximation will be presented by Theorem 2.3, see also the discussion afterwards. Corollary 2.4 provides the conditions which guarantee the stability of RHOSVD.

The sub-optimal Tucker approximand Equation (5) is simple to compute and it provides accurate approximation to the initial canonical tensor even with rather small Tucker rank. Moreover,

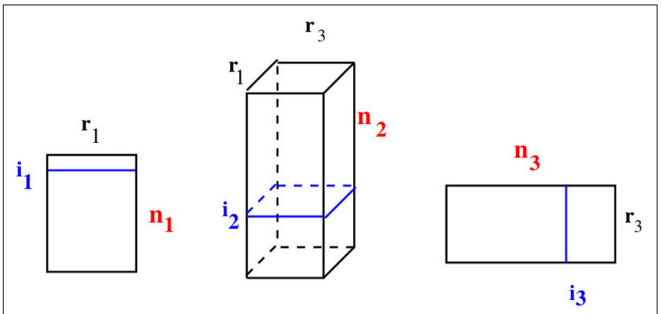


FIGURE 2 | A first step in canonical-to-Tucker decomposition.

this provides the good initial guess to calculate the best rank-$\mathbf{r}$ Tucker approximation by using the ALS iteration. In our numerical practice, usually, only one or two ALS iterations are required for convergence. For example, in case $d = 3$, algorithmically, the one step of the canonical-to-Tucker ALS algorithm reduces to the following operations. Substituting the orthogonal matrices $Z_0^{(1)}$ and $Z_0^{(3)}$ from Equation (5) into Equation (2), we perform the initial step of the first ALS iteration

$$\mathbf{A} \mapsto \mathbf{A}_{(1)} = Z_0^{(1)} \times_1 \mathbf{A}_2 \times_3 Z_0^{(3)}, \tag{6}$$

where $\mathbf{A}_2$ is given by the contraction

$$\mathbf{A}_2 = \boldsymbol{\xi} \times_1 D_{1,0} V_0^{(1)\,T} \times_2 U^{(2)} \times_3 D_{3,0} V_0^{(3)\,T} \in \mathbb{R}^{r_1\times n_2\times r_3},$$

as illustrated in **Figure 2**. Then we optimize the orthogonal subspace in the second variable by calculating the best rank-r_2 approximation to the $r_1 r_3 \times n_2$ matrix unfolding of the tensor $\mathbf{A}_2$. The similar contracted product representation can be used when $d > 3$, as well as for the construction of the TT representation for the canonical input.

Here, we notice that the core tensor in the RHOSVD decomposition can be represented in the CP data-sparse format.

Proposition 2.2. *The core tensor*

$$\boldsymbol{\beta}_0 = \boldsymbol{\xi} \times_1 [D_{1,0} V_0^{(1)\,T}] \times_2 \cdots \times_d [D_{d,0} V_0^{(d)\,T}] \in \mathbb{R}^{r_1\times\cdots\times r_d},$$

in the orthogonal Tucker representation Equation (5), $\mathbf{A}_{(\mathbf{r})}^0 = \boldsymbol{\beta}_0 \times_1 Z_0^{(1)} \times_2 \cdots \times_d Z_0^{(d)} \in \mathcal{T}_{\mathbf{r},\mathbf{n}}$*, can be recognized as*

the rank-R canonical tensor of size $r_1 \times \cdots \times r_d$ with the storage request $R(\sum_{\ell=1}^{d} r_\ell)$, which can be calculated entry-wise in $O(Rr_1 \cdots r_d)$ operations.

Indeed, introducing the matrices ${U_0^{(\ell)}}^T = D_{\ell,0} {V_0^{(\ell)}}^T \in \mathbb{R}^{r_\ell \times R}$, for $\ell = 1, \ldots d$, we conclude that the canonical core tensor $\boldsymbol{\beta}_0$ is determined by the ℓ-mode side matrices ${U_0^{(\ell)}}^T$. In the other words, the tensor $\mathbf{A}^0_{(\mathbf{r})}$ is represented in the mixed Tucker-canonical format getting rid of the "curse of dimensionality" (see also Section 2.2 below).

The accuracy of the RHOSVD approximation can be controlled by the given ε-threshold in truncated SVD of side matrices $U^{(\ell)}$. The following theorem proves the absolute error bound for the RHOSVD approximation.

Theorem 2.3. *(RHOSVD error bound, [46]). For given $\mathbf{A} \in \mathcal{C}_{R,\mathbf{n}}$ in Equation (1), let $\sigma_{\ell,1} \geq \sigma_{\ell,2} \ldots \geq \sigma_{\ell,\min(n,R)}$ be the singular values of ℓ-mode side matrices $U^{(\ell)} \in \mathbb{R}^{n \times R}$ $(\ell = 1, \ldots, d)$ with normalized skeleton vectors. Then the error of RHOSVD approximation, $\mathbf{A}^0_{(\mathbf{r})}$, is bounded by*

$$\|\mathbf{A} - \mathbf{A}^0_{(\mathbf{r})}\| \leq \|\boldsymbol{\xi}\| \sum_{\ell=1}^{d} \Big(\sum_{k=r_\ell+1}^{\min(n,R)} \sigma^2_{\ell,k} \Big)^{1/2}, \quad \|\boldsymbol{\xi}\| = \sqrt{\sum_{\nu=1}^{R} \xi^2_\nu}. \quad (7)$$

The complete proof can be found in **Section 8** (see **Appendix**).

The accuracy of the RHOSVD can be controlled in terms of the ε-criteria. To that end, given $\varepsilon > 0$, chose the Tucker ranks such that $\sum_{\ell=1}^{d} \big(\sum_{k=r_\ell+1}^{\min(n,R)} \sigma^2_{\ell,k} \big)^{1/2} \leq \varepsilon$ is satisfied, then Theorem 2.3 provided the error bound adapted to the ε-threshold.

The error estimate in Theorem 2.3 differs from the case of complete HOSVD by the extra factor $\|\boldsymbol{\xi}\|$, which is the payoff for the lack of orthogonality in the canonical input tensor. Hence, Theorem 2.3 does not provide, in general, the stable control of relative error since for the general canonical tensors there is no uniform upper bound on the constant C in the estimate

$$\|\boldsymbol{\xi}\| \leq C \|\mathbf{A}\|. \quad (8)$$

The problem is that Equation (8) applies to the general non-orthogonal canonical decomposition.

The stable RHOSVD approximation can be proven in the case of the so-called partially orthogonal or monotone decompositions. With partially orthogonal decomposition we mean that for each pair of indexes ν, μ in Equation (1) there holds $\mathrm{prod}_{\ell=1}^{d} \langle \mathbf{u}_\nu^{(\ell)}, \mathbf{u}_\mu^{(\ell)} \rangle = 0$. For monotone decompositions we assume that all coefficients and skeleton vectors in Equation (1) have non-negative values.

Corollary 2.4. *(Stability of RHOSVD) Assume the conditions of Theorem 2.3 are satisfied. (A) Suppose that at least one of the side matrices $U^{(\ell)}$, $\ell = 1, \cdots, d$, in Equation (2), is orthogonal or the decomposition Equation (1) is partially orthogonal. Then the RHOSVD error can be bounded by*

$$\|\mathbf{A} - \mathbf{A}^0_{(\mathbf{r})}\| \leq C \|\mathbf{A}\| \sum_{\ell=1}^{d} \Big(\sum_{k=r_\ell+1}^{\min(n,R)} \sigma^2_{\ell,k} \Big)^{1/2}. \quad (9)$$

(B) Let decomposition Equation (1) be monotone. Then (9) holds.

Proof: (A) The partial orthogonality assumption combined with normalization constraints for the canonical skeleton vectors imply

$$\begin{aligned} \|\mathbf{A}\|^2 &= \langle \textstyle\sum_{\nu=1}^{R} \xi_\nu \mathbf{u}_\nu^{(1)} \otimes \ldots \otimes \mathbf{u}_\nu^{(d)}, \sum_{\nu=1}^{R} \xi_\nu \mathbf{u}_\nu^{(1)} \otimes \ldots \otimes \mathbf{u}_\nu^{(d)} \rangle \\ &= \sum_{\nu=1}^{R} \xi^2_\nu \prod_{\ell=1}^{d} \langle \mathbf{u}_\nu^{(\ell)}, \mathbf{u}_\nu^{(\ell)} \rangle \\ &+ \sum_{\nu,\mu=1,\nu\neq\mu}^{R} \xi_\nu \xi_\mu \prod_{\ell=1}^{d} \langle \mathbf{u}_\nu^{(\ell)}, \mathbf{u}_\mu^{(\ell)} \rangle \\ &= \sum_{\nu=1}^{R} \xi^2_\nu = \|\boldsymbol{\xi}\|^2. \end{aligned}$$

The above relation also holds in the case of orthogonality of the side matrix $U^{(\ell)}$ for some fixed ℓ. Then the result follows by (7).

(B) In case of *monotone* decomposition we conclude that the pairwise scalar product of all summands in Equation (1) is non-negative, while the norm of each ν-term is equal to ξ_ν. Then the upper bound

$$\langle \mathbf{u}_1, \mathbf{u}_1 \rangle + \cdots + \langle \mathbf{u}_R, \mathbf{u}_R \rangle \leq \langle \sum_{\nu=1}^{R} \mathbf{u}_\nu, \sum_{\nu=1}^{R} \mathbf{u}_\nu \rangle,$$

holds for vectors $\mathbf{u}_\nu = \xi_\nu \mathbf{u}_\nu^{(1)} \otimes \ldots \otimes \mathbf{u}_\nu^{(d)}$, $\nu = 1, \cdots, R$, with non-negative entries applied to the case of R summands, thus implying $\|\boldsymbol{\xi}\|^2 \leq \|\mathbf{A}\|^2$. Now, the result follows.

Clearly, the orthogonality assumption may lead to slightly higher separation rank, however, this constructive decomposition stabilizes the RHOSVD approximation method applied to the canonical format tensor (i.e., it allows the stable control of relative error). The case of monotone canonical sums typically arises in the sinc-based canonical approximation to radially symmetric Green's kernels by a sum of Gaussians. On the other hand, in long term computational practice the numerical instability of RHOSVD approximation was not observed in case of physically relevant data.

2.2. Mixed Tucker Tensor Format and Tucker-to-CP Transform

In the procedure for the canonical tensor rank reduction the goal is to have a result in a canonical tensor format with a smaller rank. By converting the core tensor to CP format, one can use the mixed two-level Tucker data format [12, 27], or canonical CP format. **Figure 3** illustrates the computational scheme of the two-level Tucker approximation.

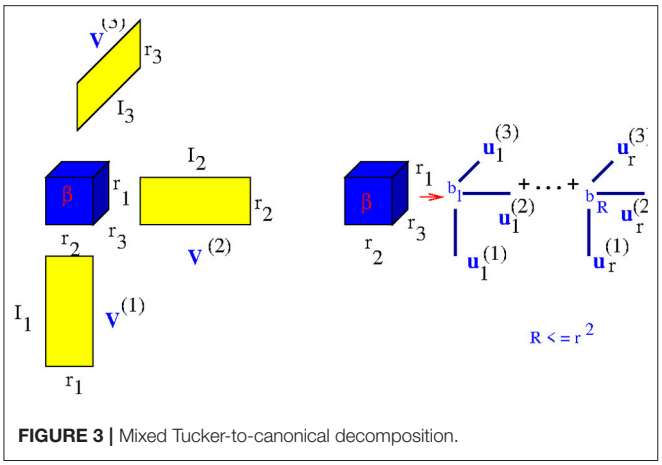


FIGURE 3 | Mixed Tucker-to-canonical decomposition.

We define by $\overline{n}_\ell$ the single-hole product of dimension-modes,

$$\overline{n}_\ell = n_1 \cdots n_{\ell-1} n_{\ell+1} \cdots n_d. \tag{10}$$

The same definition applies to the quantity $\overline{r}_\ell$.

Next lemma describes the approximation of the Tucker tensor by using canonical representation [12, 27].

Lemma 2.5. *(Mixed Tucker-to-canonical approximation, [27]).*

(A) Let the target tensor $\mathbf{A}$ *have the form* $\mathbf{A} = \boldsymbol{\beta} \times_1 V^{(1)} \times_2 \ldots \times_d V^{(d)} \in \mathcal{T}_{\mathbf{r},\mathbf{n}}$, *with the orthogonal side-matrices* $V^{(\ell)} = [\mathbf{v}_1^{(\ell)} \ldots \mathbf{v}_{r_\ell}^{(\ell)}] \in \mathbb{R}^{n\times r_\ell}$ *and* $\boldsymbol{\beta} \in \mathbb{R}^{r_1\times\ldots\times r_d}$. *Then, for a given* $R \le \min_{1\le\ell\le d} \overline{r}_\ell$,

$$\min_{\mathbf{Z}\in\mathcal{C}_{R,\mathbf{n}}} \|\mathbf{A}-\mathbf{Z}\| = \min_{\boldsymbol{\mu}\in\mathcal{C}_{R,\mathbf{r}}} \|\boldsymbol{\beta}-\boldsymbol{\mu}\|. \tag{11}$$

(B) Assume that there exists the best rank-R approximation $\mathbf{A}_{(R)} \in \mathcal{C}_{R,\mathbf{n}}$ *of* $\mathbf{A}$, *then there is the best rank-R approximation* $\boldsymbol{\beta}_{(R)} \in \mathcal{C}_{R,\mathbf{r}}$ *of* $\boldsymbol{\beta}$, *such that*

$$\mathbf{A}_{(R)} = \boldsymbol{\beta}_{(R)} \times_1 V^{(1)} \times_2 \ldots \times_d V^{(d)}. \tag{12}$$

The complete proof can be found in **Section 8** (see **Appendix**). Notice that condition $R \le \min_{1\le\ell\le d} \overline{r}_\ell$ simply means that the canonical rank does not exceed the maximal CP rank of the Tucker core tensor.

Combination of Theorem 2.3 and Lemma 2.5 paves the way to the rank optimization of canonical tensors with the large mode-size arising, for example, in the grid-based numerical methods for multi-dimensional PDEs with non-regular (singular) solutions. In such applications the univariate grid-size (i.e., the mode-size) may be about $n = 10^4$ and even larger.

Notice that the Tucker (for moderate d) and canonical formats allow to perform basic multi-linear algebra using *one-dimensional operations*, thus reducing the exponential scaling in d. Rank-truncated transforms between different formats can be applied in multi-linear algebra on mixed tensor representations as well, see Lemma 2.5. The particular application to tensor convolution in many dimensions was discussed, for example, in [1, 2].

We summarize that the direct methods of tensor approximation can be classified by:

(1) Analytic Tucker approximation to some classes of function-related dth order tensors ($d \ge 2$), say, by multi-variate polynomial interpolation [1].
(2) Sinc quadrature based approximation methods in the canonical format applied to a class of analytic function related tensors [11].
(3) Truncated HOSVD and RHOSVD, for quasi-optimal Tucker approximation of the full-format, respectively, canonical tensors [46].

Direct analytic approximation methods by sinc quadrature/interpolation are of principal importance. Basic examples are given by the tensor representation of Green's kernels, the elliptic operator inverse and analytic matrix-valued functions. In all cases, the algebraic methods for rank reduction by the ALS-type iterative Tucker/canonical approximation can be applied.

Further improvement and enhancement of algebraic tensor approximation methods can be based on the combination of advanced nonlinear iteration, multigrid tensor methods, greedy algorithms, hybrid tensor representations, and the use of new problem adapted tensor formats.

2.3. Tucker-to-Canonical Transform

In the rank reduction scheme for the canonical rank-R tensors, we use successively the canonical-to-Tucker (C2T) transform and then the Tucker-to-canonical (T2C) tensor approximation.

First, we notice that the canonical rank of a tensor $\mathbf{A} \in \mathbb{V}_\mathbf{n}$ has the upper bound (see [27, 46]),

$$R \le \min_{1\le\ell\le d} \overline{n}_\ell, \tag{13}$$

where $\overline{n_\ell}$ is given by Equation (10). Rank bound (13) applied to the Tucker core tensor of the size $r \times r \times r$, indicates that the ultimate canonical rank of a large-size tensor in $\mathbb{V}_\mathbf{n}$ has the upper bound r^2. Notice that for function related tensors the Tucker rank scales logarithmically in both approximation accuracy and the discretization grid size (see the proof for some classes of function in [51]).

The following remark shows that the maximal canonical rank of the Tucker core of 3rd order tensor can be easily reduced to the value less than r^2 by the SVD-based procedure applied to the matrix slices of the Tucker core tensor $\boldsymbol{\beta}$. Though, being not practically attractive for arbitrary high order tensors, the simple algorithm described in Remark 2.6 below is proved to be useful for the treatment of small size 3rd order Tucker core tensors within the rank reduction algorithms described in the previous sections.

Remark 2.6. *Let* $d = 3$ *for the sake of clarity [27, 46]. There is a simple procedure based on SVD to reduce the canonical rank of the core tensor* $\boldsymbol{\beta}$, *within the accuracy* $\varepsilon > 0$. *Denote by* $B_m \in \mathbb{R}^{r\times r}$,

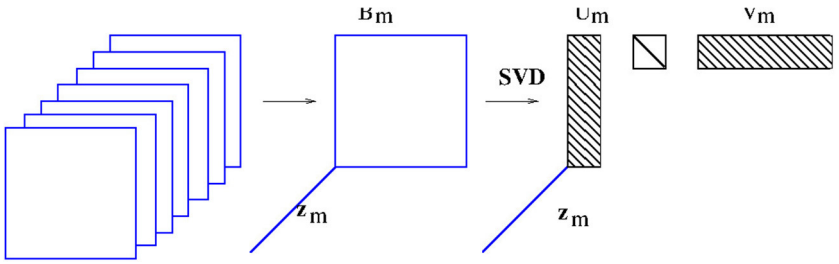


FIGURE 4 | Tucker-to-canonical decomposition for a small core tensor, see Remark 2.6.

$m = 1, \ldots, r$ the two-dimensional slices of $\boldsymbol{\beta}$ in each fixed mode and represent

$$\boldsymbol{\beta} = \sum_{m=1}^{r} B_m \otimes \mathbf{z}_m, \quad \mathbf{z}_m \in \mathbb{R}^r, \tag{14}$$

where $\mathbf{z}_m(m) = 1$, $\mathbf{z}_m(j) = 0$ for $j = 1, \ldots, r$, $j \neq m$ (there are exactly d possible decompositions). Let p_m be the minimal integer, such that the singular values of B_m satisfy $\sigma_k^{(m)} \leq \frac{\varepsilon}{r^{3/2}}$ for $k = p_m+1, \ldots, r$ (if $\sigma_r^{(m)} > \frac{\varepsilon}{r^{3/2}}$, then set $p_m = r$). Then, denoting by

$$B_{p_m} = \sum_{k_m=1}^{p_m} \sigma_{k_m}^{(m)} \mathbf{u}_{k_m} \otimes \mathbf{v}_{k_m},$$

the corresponding rank-p_m approximation to B_m (by truncation of $\sigma_{p_m+1}^{(m)}, \ldots, \sigma_r^{(m)}$), we arrive at the rank-R canonical approximation to $\boldsymbol{\beta}$,

$$\boldsymbol{\beta}_{(R)} := \sum_{m=1}^{r} B_{p_m} \otimes \mathbf{z}_m, \quad \mathbf{z}_m \in \mathbb{R}^r, \tag{15}$$

providing the error estimate

$$\|\boldsymbol{\beta} - \boldsymbol{\beta}_{(R)}\| \leq \sum_{m=1}^{r} \|B_m - B_{p_m}\| = \sum_{m=1}^{r} \sqrt{\sum_{k_m=p_m+1}^{r} (\sigma_{k_m}^{(m)})^2}$$
$$\leq \sum_{m=1}^{r} \sqrt{r \frac{\varepsilon^2}{r^3}} = \varepsilon$$

Representation (15) is a sum of rank-p_m terms so that the total rank is bounded by $R \leq p_1 + \ldots + p_r \leq r^2$. The approach can be extended to arbitrary $d \geq 3$ with the bound $R \leq r^{d-1}$.

Figure 4 illustrates the canonical decomposition of the core tensor by using the SVD of slices B_m of the core tensor $\boldsymbol{\beta}$, yielding matrices $U_m = \{\mathbf{u}_{k_m}\}_{k=1}^{p_m}$, $V_m = \{\mathbf{v}_{k_m}\}_{k=1}^{p_m}$ and a diagonal matrix of small size $p_m \times p_m$ containing the truncated singular values. It also shows the vector $\mathbf{z}_m = [0, \ldots, 0, 1, 0, \ldots, 0]$, containing all entries equal to 0 except 1 at the mth position.

It is worse to note that the rank reduction for the rank-R core tensor of small size $r_1 \times \cdots \times r_d$, can be also performed by using the cascading ALS algorithms in CP format applied to the canonical input tensor, as it was applied in [50]. Moreover, a number of numerical examples presented in the present paper and in the included literature (applied to function generated tensors) demonstrate the substantial reduction of the initial canonical rank R.

3. CALCULATION OF 3D INTEGRALS WITH THE NEWTON KERNEL

The first application of the RHOSVD was calculation of the 3D grid-based Hartree potential operator in the Hartree-Fock equation,

$$V_H(x) := \int_{\mathbb{R}^3} \frac{\rho(y)}{\|x - y\|} dy, \tag{16}$$

where the electron density,

$$\rho(x) = 2 \sum_{a=1}^{N_{orb}} (\varphi_a)^2, \tag{17}$$

is represented in terms of molecular orbitals, presented in the Gaussian-type basis (GTO), $\varphi_a(x) = \sum_{k=1}^{N_b} c_{a,k} g_k(x)$. The Hartree potential describes the repulsion energy of the electrons in a molecule. The intermediate goal here is the calculation of the so-called Coulomb matrix,

$$J_{km} := \int_{\mathbb{R}^3} g_k(x) g_m(x) V_H(x) dx, \quad k, m = 1, \ldots N_b \quad x \in \mathbb{R}^3,$$

which represents the Hartree potential in the given GTO basis.

In fact, calculation of this 3D convolution operator with the Newton kernel, requires high accuracy and it should be repeated multiply in the course of the iterative solution of the Hartree-Fock nonlinear eigenvalue problem. The presence of nuclear cusps in the electron density makes additional challenge to computation of the Hartree potential operator. Traditionally, these calculations are based on involved analytical evaluation of the corresponding integral in a separable Gaussian basis set by using erf function. Tensor-structured calculation of the multi-dimensional convolution integral operators with the Newton kernel have been introduced in [27, 29, 46].

The molecule is embedded in a computational box $\Omega = [-b, b]^3 \in \mathbb{R}^3$. The equidistant $n \times n \times n$ tensor grid $\omega_{3,n} = \{x_{\mathbf{i}}\}$, $\mathbf{i} \in \mathcal{I} := \{1, \ldots, n\}^3$, with the mesh-size $h = 2b/(n+1)$ is used. In calculations of integral terms, the Gaussian basis functions $g_k(x)$, $x \in \mathbb{R}^3$, are approximated by sampling their values at the centers of discretization intervals using one-dimensional piecewise constant basis functions $g_k(x) \approx \overline{g}_k(x) = \prod_{\ell=1}^{3} \overline{g}_k^{(\ell)}(x_\ell)$, $\ell = 1, 2, 3$, yielding their rank-1 tensor representation,

$$\mathbf{G}_k = \mathbf{g}_k^{(1)} \otimes \mathbf{g}_k^{(2)} \otimes \mathbf{g}_k^{(3)} \in \mathbb{R}^{n \times n \times n}, \quad k = 1, \ldots, N_b. \tag{18}$$

Given the discrete tensor representation of basis functions (18), the electron density is approximated using 1D Hadamard products of rank-1 tensors as

$$\rho \approx \Theta = 2 \sum_{a=1}^{N_{orb}} \sum_{k=1}^{N_b} \sum_{m=1}^{N_b} c_{a,m} c_{a,k} (\mathbf{g}_k^{(1)} \odot \mathbf{g}_m^{(1)}) \otimes \cdots \otimes (\mathbf{g}_k^{(3)} \odot \mathbf{g}_m^{(3)}) \in \mathbb{R}^{n \times n \times n}. \tag{19}$$

For convolution operator, the representation of the Newton kernel $\frac{1}{\|x-y\|}$ by a canonical rank-R_N tensor [1] is used (see Section 4.1 for details),

$$\mathbf{P}_R = \sum_{q=1}^{R_N} \mathbf{p}_q^{(1)} \otimes \mathbf{p}_q^{(2)} \otimes \mathbf{p}_q^{(3)} \in \mathbb{R}^{n \times n \times n}. \tag{20}$$

The initial rank of the electron density in the canonical tensor format Θ in Equation (17) is large even for small molecules. Rank reduction by using RHOSVD C2T plus T2C reduces the rank $\Theta \mapsto \Theta'$ by several orders of magnitude, from $N_b^2/2$ to $R_\rho \ll N_b^2/2$, from $\sim 10^4$ to $\sim 10^2$. Then the 3D tensor representation of the Hartree potential is calculated by using the 3D tensor product convolution, which is a sum of tensor products of 1D convolutions,

$$V_H \approx \mathbf{V}_H = \Theta' * \mathbf{P}_R = \sum_{m=1}^{R_\rho} \sum_{q=1}^{R_N} c_m \left(\mathbf{u}_m^{(1)} * \mathbf{p}_q^{(1)}\right) \otimes \left(\mathbf{u}_m^{(2)} * \mathbf{p}_q^{(2)}\right) \otimes \left(\mathbf{u}_m^{(3)} * \mathbf{p}_q^{(3)}\right).$$

The Coulomb matrix entries J_{km} are obtained by 1D scalar products of $\mathbf{V}_H$ with the Galerkin basis consisting of rank-1 tensors,

$$J_{km} s \approx \langle \mathbf{G}_k \odot \mathbf{G}_m, \mathbf{V}_H \rangle, \quad k, m = 1, \ldots N_b.$$

The cost of 3D tensor product convolution is $O(n \log n)$ instead of $O(n^3 \log n)$ for the standard benchmark 3D convolution using the 3D FFT. **Table 1** shows CPU times (sec) for the Matlab computation of V_H for H_2O molecule [46] on a SUN station using 8 Opteron Dual-Core/2600 processors (times for 3D FFT for $n \geq 1024$ are obtained by extrapolation). C2T shows the time for the canonical-to-Tucker rank reduction.

TABLE 1 | Times (sec) for the C2T transform and the 3D tensor product convolution vs. 3D FFT convolution.

n^3	1024^3	2048^3	4096^3	8192^3	16384^3
FFT_3	~ 6000	–	–	–	~ 2 years
C^*C	8.8	20.0	61.0	157.5	299.2
C2T	6.9	10.9	20.0	37.9	86.0

The grid-based tensor calculation of the multi-dimensional integrals in quantum chemistry provides the required high accuracy by using large grids and the ranks are controlled by the required ε in the rank truncation algorithms. The results of the tensor-based calculations have been compared with the results of the benchmark standard computations by the MOLPRO package. It was shown that the accuracy is of the order of 10^{-7} hartree in the resulting ground state energy (see [2, 27]).

4. RHOSVD IN THE RANGE-SEPARATED TENSOR FORMATS

The range-separated (RS) tensor formats have been introduced in [32] as the constructive tool for low-rank tensor representation (approximation) of function related data discretized on Cartesian grids in $\mathbb{R}^d$, which may have multiple singularities or cusps. Such highly non-regular data typically arise in computational quantum chemistry, in many-particle dynamics simulations and many-particle electrostatics calculations, in protein modeling and in data science. The key idea of the RS representation is the splitting of the short- and long-range parts in the functional data and further low-rank approximation of the rather regular long-range part in the classical tensor formats.

In this concern RHOSVD method becomes an essential ingredient of the rank reduction algorithms for the "long-range" input tensor, which usually inherits the large initial rank.

4.1. Low-Rank Approximation of Radial Functions

First, we recall the grid-based method for the low-rank canonical representation of a spherically symmetric kernel functions $p(\|x\|)$, $x \in \mathbb{R}^d$ for $d = 2, 3, \ldots$, by its projection onto the finite set of basis functions defined on tensor grid. The approximation theory by a sum of Gaussians for the class of analytic potentials $p(\|x\|)$ was presented in [1, 11, 51, 52]. The particular numerical schemes for rank-structured representation of the Newton and Slater kernels

$$p(\|x\|) = \frac{1}{4\pi \|x\|}, \quad \text{and} \quad p(\|x\|) = e^{-\lambda \|x\|}, \quad x \in \mathbb{R}^3, \tag{21}$$

discretized on a fine 3D Cartesian grid in the form of low-rank canonical tensor was described in [11, 51].

In what follows, for the ease of exposition, we confine ourselves to the case $d = 3$. In the computational domain $\Omega = [-b, b]^3$, let us introduce the uniform $n \times n \times n$ rectangular Cartesian grid Ω_n with mesh size $h = 2b/n$ (n even). Let $\{\psi_{\mathbf{i}} =$

$\prod_{\ell=1}^{3} \psi_{i_\ell}^{(\ell)}(x_\ell)\}$ be a set of tensor-product piecewise constant basis functions, labeled by the 3-tuple index $\mathbf{i} = (i_1, i_2, i_3)$, $i_\ell \in I_\ell = \{1, \ldots, n\}$, $\ell = 1, 2, 3$. The generating kernel $p(\|x\|)$ is discretized by its projection onto the basis set $\{\psi_{\mathbf{i}}\}$ in the form of a third order tensor of size $n \times n \times n$, defined entry-wise as

$$\mathbf{P} := [p_{\mathbf{i}}] \in \mathbb{R}^{n\times n\times n}, \quad p_{\mathbf{i}} = \int_{\mathbb{R}^3} \psi_{\mathbf{i}}(x) p(\|x\|)\, \mathrm{d}x. \tag{22}$$

The low-rank canonical decomposition of the 3rd order tensor $\mathbf{P}$ is based on using exponentially fast convergent sinc-quadratures for approximating the Laplace-Gauss transform to the analytic function $p(z)$, $z \in \mathbb{C}$, specified by a certain weight $\widehat{p}(t) > 0$,

$$p(z) = \int_{\mathbb{R}_+} \widehat{p}(t) e^{-t^2 z^2}\, \mathrm{d}t \approx \sum_{k=-M}^{M} p_k e^{-t_k^2 z^2} \quad \text{for} \quad |z| > 0, z \in \mathbb{R}, \tag{23}$$

with the proper choice of the quadrature points t_k and weights p_k. The *sinc*-quadrature based approximation to generating function by using the short-term Gaussian sums in Equation (23) are applicable to the class of analytic functions in certain strip $|z| \le D$ in the complex plane, such that on the real axis these functions decay polynomially or exponentially. We refer to basic results in [11, 52, 53], where the exponential convergence of the *sinc*-approximation in the number of terms (i.e., the canonical rank) was analyzed for certain classes of analytic integrands.

Now, for any fixed $x = (x_1, x_2, x_3) \in \mathbb{R}^3$, such that $\|x\| > a > 0$, we apply the sinc-quadrature approximation Equation (23) to obtain the separable expansion

$$\begin{aligned} p(\|x\|) &= \int_{\mathbb{R}_+} \widehat{p}(t) e^{-t^2\|x\|^2}\, \mathrm{d}t \approx \sum_{k=-M}^{M} p_k e^{-t_k^2\|x\|^2} \\ &= \sum_{k=-M}^{M} p_k \prod_{\ell=1}^{3} e^{-t_k^2 x_\ell^2}, \end{aligned} \tag{24}$$

providing an exponential convergence rate in M,

$$\left| p(\|x\|) - \sum_{k=-M}^{M} p_k e^{-t_k^2\|x\|^2} \right| \le \frac{C}{a} e^{-\beta\sqrt{M}}, \quad \text{with some } C, \beta > 0. \tag{25}$$

In the case of Newton kernel, we have $p(z) = 1/z$, $\widehat{p}(t) = \frac{2}{\sqrt{\pi}}$, so that the Laplace-Gauss transform representation reads

$$\frac{1}{z} = \frac{2}{\sqrt{\pi}} \int_{\mathbb{R}_+} e^{-z^2 t^2} dt, \quad \text{where} \quad z = \|x\|, \quad x \in \mathbb{R}^3, \tag{26}$$

which can be approximated by the sinc quadrature Equation (24) with the particular choice of quadrature points t_k, providing the exponential convergence rate as in Equation (25) [11, 51].

In the case of Yukawa potential the Laplace Gauss transform reads

$$\frac{e^{-\kappa z}}{z} = \frac{2}{\sqrt{\pi}} \int_{\mathbb{R}_+} e^{-\kappa^2/t^2} e^{-z^2 t^2} dt, \quad \text{where} \quad z = \|x\|, \quad x \in \mathbb{R}^3. \tag{27}$$

The analysis of the sinc quadrature approximation error for this case can be found, in particular, in [1, 51], section 2.4.7.

Combining (22) and (24), and taking into account the separability of the Gaussian basis functions, we arrive at the low-rank approximation to each entry of the tensor $\mathbf{P} = [p_{\mathbf{i}}]$,

$$\begin{aligned} p_{\mathbf{i}} &\approx \sum_{k=-M}^{M} p_k \int_{\mathbb{R}^3} \psi_{\mathbf{i}}(x) e^{-t_k^2\|x\|^2} \mathrm{d}x \\ &= \sum_{k=-M}^{M} p_k \prod_{\ell=1}^{3} \int_{\mathbb{R}} \psi_{i_\ell}^{(\ell)}(x_\ell) e^{-t_k^2 x_\ell^2} \mathrm{d}x_\ell. \end{aligned}$$

Define the vector (recall that $p_k > 0$)

$$\begin{aligned} \mathbf{p}_k^{(\ell)} &= p_k^{1/3} \left[b_{i_\ell}^{(\ell)}(t_k) \right]_{i_\ell=1}^{n_\ell} \in \mathbb{R}^{n_\ell} \quad \text{with} \\ b_{i_\ell}^{(\ell)}(t_k) &= \int_{\mathbb{R}} \psi_{i_\ell}^{(\ell)}(x_\ell) e^{-t_k^2 x_\ell^2} \mathrm{d}x_\ell, \end{aligned} \tag{28}$$

then the 3rd order tensor $\mathbf{P}$ can be approximated by the R-term ($R = 2M + 1$) canonical representation

$$\begin{aligned} \mathbf{P} \approx \mathbf{P}_R &= \sum_{k=-M}^{M} p_k \bigotimes_{\ell=1}^{3} \mathbf{b}^{(\ell)}(t_k) \\ &= \sum_{k=-M}^{M} \mathbf{p}_k^{(1)} \otimes \mathbf{p}_k^{(2)} \otimes \mathbf{p}_k^{(3)} \in \mathbb{R}^{n\times n\times n}, \mathbf{p}_k^{(\ell)} \in \mathbb{R}^n. \end{aligned} \tag{29}$$

Given a threshold $\varepsilon > 0$, in view of Equation (25), we can choose $M = O(\log^2 \varepsilon)$ such that in the max-norm

$$\|\mathbf{P} - \mathbf{P}_R\| \le \varepsilon \|\mathbf{P}\|.$$

In the case of continuous radial function $p(\|x\|)$, say the Slater potential, we use the collocation type discretization at the grid points including the origin, $x = 0$, so that the univariate mode size becomes $n \to n_1 = n + 1$. In what follows, we use the same notation $\mathbf{P}_R$ in the case of collocation type tensors (for example, the Slater potential) so that the particular meaning becomes clear from the context.

4.2. The RS Tensor Format Revisited

The range separated (RS) tensor format was introduced in [32] for efficient representation of the collective free-space electrostatic potential of large biomolecules. This rank-structured tensor representation of the collective electrostatic potential of many-particle systems of general type allows to reduce essentially computation of their interaction energy, and it provides convenient form for performing other algebraic transforms. The RS format proved to be useful for range-separated tensor representation of the Dirac delta [34] in $\mathbb{R}^d$ and based on that, for regularization of the Poisson-Boltzmann equation (PBE) by decomposition of the solution into short- and long-range parts, where the short-range part of the solution is evaluated by simple tensor operations without solving the PDE. The smooth long-range part is calculated by solving the PBE with the modified

right-hand side by using the RS decomposition of the Dirac delta, so that now it does not contain singularities. We refer to papers [33, 35] describing the approach in details.

First, we recall the definition of the range separated (RS) tensor format, see [32], for representation of d-tensors $\mathbf{A} \in \mathbb{R}^{n_1 \times \cdots \times n_d}$. The RS format is served for the hybrid tensor approximation of discretized functions with multiple cusps or singularities. This allows the splitting of the target tensor onto the highly localized components approximating the singularity and the component with global support that allows the low-rank tensor approximation. Such functions typically arise in computational quantum chemistry, in many-particle modeling and in the interpolation of multi-dimensional data measured at certain set of spatial points in $\mathbb{R}^{n\times n\times n}$.

In the following definition of RS-canonical tensor format, we use the notion of localized canonical tensor $\mathbf{U}_0$, which is characterized by the small support whose diameter has a size of a few grid points. This tensor will be used as the reference one for presentation of the short-range part in the RS tensor. To that end we use the operation $\text{Replica}_{x_\nu}(\mathbf{U}_0)$ which replicates $\mathbf{U}_0$ into some given grid point x_ν. In this construction, we assume that the chosen grid points x_ν are well separated, i.e., the distance between each pair of points is not less then some given threshold $n_\delta > 0$.

Definition 4.1. *(RS-canonical tensors, [32]). Given the rank-R_s reference localized CP tensor $\mathbf{U}_0$. The RS-canonical tensor format defines the class of d-tensors $\mathbf{A} \in \mathbb{R}^{n_1\times\cdots\times n_d}$, represented as a sum of a rank-R_l CP tensor $\mathbf{U}_{long} = \sum_{k=1}^{R_l} \xi_k \mathbf{u}_k^{(1)} \otimes \cdots \otimes \mathbf{u}_k^{(d)}$, and a cumulated CP tensor $\mathbf{U}_{short} = \sum_{\nu=1}^{N_0} c_\nu \mathbf{U}_\nu$, such that*

$$\mathbf{A} = \mathbf{U}_{long} + \mathbf{U}_{short} = \sum\nolimits_{k=1}^{R_l} \xi_k \mathbf{u}_k^{(1)} \otimes \cdots \otimes \mathbf{u}_k^{(d)} + \sum\nolimits_{\nu=1}^{N_0} c_\nu \mathbf{U}_\nu, \tag{30}$$

where $\mathbf{U}_{short}$ is generated by the localized reference CP tensor $\mathbf{U}_0$, i.e., $\mathbf{U}_\nu = Replica_{x_\nu}(\mathbf{U}_0)$, with $rank(\mathbf{U}_\nu) = rank(\mathbf{U}_0) \le R_s$, where, given the threshold $n_\delta > 0$, the effective support of $\mathbf{U}_\nu$ is bounded by $diam(supp\mathbf{U}_\nu) \le 2n_\delta$ in the index size.

Each RS-canonical tensor is, therefore, uniquely defined by the following parametrization: rank-R_l canonical tensor $\mathbf{U}_{long}$, the rank-R_s reference canonical tensor $\mathbf{U}_0$ with the small mode size bounded by $2n_\delta$, list $\mathcal{J}$ of the coordinates and weights of N_0 particles in $\mathbb{R}^d$. The storage size is linear in both the dimension and the univariate grid size,

$$\text{stor}(\mathbf{A}) \le dR_l n + (d+1)N_0 + dR_s n_\delta.$$

The main benefit of the RS-canonical tensor decomposition is the almost uniform bound on the CP/Tucker rank of the long-range part $\mathbf{U}_{long} = \sum_{k=1}^{R_l} \xi_k \mathbf{u}_k^{(1)} \otimes \cdots \otimes \mathbf{u}_k^{(d)}$, in the multi-particle potential discretized on fine $n \times n \times n$ spatial grid. It was proven in [32] that the canonical rank R scales logarithmically in both the number of particles N_0 and the approximation precision, see also Lemma 4.5.

Given the rank-R CP decomposition Equation (29) based on the sinc-quadrature approximation Equation (24) of the discretized radial function $p(\|x\|)$, we define the two subsets of indices, $\mathcal{K}_l := \{k : t_k \le 1\}$ and $\mathcal{K}_s := \{k : t_k > 1\}$, and then introduce the RS-representation of this tensor as follows,

$$\mathbf{P}_R = \mathbf{P}_{R_l} + \mathbf{P}_{R_s}, \quad R = R_l + R_s, \quad R_l = \#\mathcal{K}_l, \quad R_s = \#\mathcal{K}_s, \tag{31}$$

where

$$\mathbf{P}_{R_l} := \sum_{k\in\mathcal{K}_l} \mathbf{p}_k^{(1)} \otimes \mathbf{p}_k^{(2)} \otimes \mathbf{p}_k^{(3)}, \quad \mathbf{P}_{R_s} := \sum_{k\in\mathcal{K}_s} \mathbf{p}_k^{(1)} \otimes \mathbf{p}_k^{(2)} \otimes \mathbf{p}_k^{(3)}.$$

This representation allows to reduce the calculation of the multi-particle interaction energy of the many-particle system. Recall that the electrostatic interaction energy of N charged particles is represented in the form

$$E_N = E_N(x_1, \ldots, x_N) = \sum_{i=1}^{N} \sum_{j<i}^{N} \frac{z_i z_j}{\|x_i - x_j\|}, \tag{32}$$

and it can be computed by direct summation in $O(N^2)$ operations. The following statement is the modification of Lemma 4.2 in [32] (see [54] for more details).

Lemma 4.2. *[54] Let the effective support of the short-range components in the reference potential $\mathbf{P}_R$ for the Newton kernel does not exceed the minimal distance between particles, $\sigma > 0$. Then the interaction energy E_N of the N-particle system can be calculated by using only the long range part in the tensor $\mathbf{P}$ representing on the grid the total potential sum,*

$$E_N = \frac{1}{2}\sum_{j=1}^{N} z_j(\mathbf{P}_l(x_j) - z_j\mathbf{P}_{R_l}(0)) = \frac{1}{2}\langle \mathbf{z}, \mathbf{p}_l\rangle - \frac{\mathbf{P}_{R_l}(0)}{2}\sum_{j=1}^{N} z_j^2, \tag{33}$$

in $O(R_l N)$ operations, where R_l is the canonical rank of the long-range component in $\mathbf{P}$, $\mathbf{P}_l$.

Here, $\mathbf{z} \in \mathbb{R}^N$ is a vector composed of all charges of the multi-particle systems, and $\mathbf{p}_l \in \mathbb{R}^N$ is the vector of samples of the collective electrostatic long-range potential $\mathbf{P}_l$ in the nodes corresponding to particle locations. Thus, the term $\frac{1}{2}\langle \mathbf{z}, \mathbf{p}_l\rangle$ denotes the "non–calibrated" interaction energy associated with the long-range tensor component $\mathbf{P}_l$, while $\mathbf{P}_{R_l}$ denotes the long-range part in the tensor representing the *single reference Newton kernel*, and $\mathbf{P}_{R_l}(0)$ is its value at the origin.

Lemma 4.2 indicates that the interaction energy does not depend on the short-range part in the collective potential, and this is the key point for the construction of energy preserving regularized numerical schemes for solving the basic equations in bio-molecular modeling by using low-rank tensor decompositions.

4.3. Multi-Linear Operations in RS Tensor Formats

In what follows, we address the important question on how *the basic multi-linear operations* can be implemented in the RS tensor format by using the RHOSVD rank compression. The point is that various tensor operations arise in the

course of commonly used numerical schemes and iterative algorithms which usually include many sums and products of functions as well as the actions of differential/integral operators, always making the tensor structure of input data much more complicated requiring the robust rank reduction schemes.

The other important aspect is related to the use of large (fine resolution) discretization grids which is limited by the restriction on the size of the full input tensors, $O(n^d)$ (curse of dimensionality), representing the discrete functions and operators to be approximated in low rank tensor format. Remarkably, that tensor decomposition for special class of functions, which allow the sinc-quadrature approximation, can be performed on practically indefinitely large grids because the storage and numerical costs of such numerical schemes scale linearly in the univariate grid size, $O(dn)$. Hence, having constructed such low rank approximations for certain set of "reproducing" radial functions, makes it possible to construct the low rank RS representation at linear complexity, $O(dn)$, for the wide class of functions and operators by using the rank truncated multi-linear operations. The examples of such "reproducing" radial functions are commonly used in our computational practice.

First, consider the Hadamard product of two tensors $\mathbf{P}_R$ and $\mathbf{Q}_{R_1}$ corresponding to the pointwise product of two generating multi-variate functions centered at the same point. The RS representation of the product tensor is based on the observation that the long-range part of the Hadamard product of two tensors in RS-format is basically determined by the product of their long-range parts.

Lemma 4.3. *Suppose that the RS representation Equation (31) of tensors $\mathbf{P}_R$ and $\mathbf{Q}_{R_1}$ is constructed based on the sinc-quadrature CP approximation Equation (29). Then the long-range part of the Hadamard product of these RS-tensors,*

$$\mathbf{Z} = (\mathbf{P}_s + \mathbf{P}_l) \odot (\mathbf{Q}_s + \mathbf{Q}_l),$$

can be represented by the product of their long-range parts, $\mathbf{Z}_l = \mathbf{P}_l \odot \mathbf{Q}_l$, with the subsequent rank reduction. Moreover, we have $rank(\mathbf{Z}_l) \le R_l Q_l$.

Proof: We consider the case of collocation tensors and suppose that each skeleton vector in CP tensors $\mathbf{P}_R$ and $\mathbf{Q}_{R_1}$ is given by the restriction of certain Gaussians to the set of grid points. Chose the arbitrary short-range components in $\mathbf{P}_R$ and some component in $\mathbf{Q}_{R_1}$, generated by Gaussians $e^{-t_k x_\ell^2}$ and $e^{-t_m x_\ell^2}$, respectively. Then the effective support of the product of these two terms becomes smaller than that for each of the factors in view of the identity $e^{-t_k x_\ell^2} e^{-t_m x_\ell^2} = e^{-(t_k+t_m)x_\ell^2}$ considered for arbitrary $t_k, t_m > 0$. This means that each term that includes the short-range multiple remains to be in the short range. Then the long range part in $\mathbf{Z}$ takes a form $\mathbf{Z}_l = \mathbf{P}_l \odot \mathbf{Q}_l$ with the subsequent rank reduction.

The sums of several tensors in RS format can be easily split into short- and long-range parts by grouping the respective components in the summands. The other important operation is the operator-function product in RS tensor format (see the example in [34] related to the action of Laplacian with the singular Newton kernel resulting in the RS decomposition of the Dirac delta). This topic will be considered in detail elsewhere.

4.4. Representing the Slater Potential in RS Tensor Format

In what follows, we consider the RS-canonical tensor format for the rank-structured representation of the Slater function

$$G(x) = e^{-\lambda\|x\|}, \quad \lambda \in \mathbb{R}_+,$$

which has the principal significance in electronic structure calculations (say, based on the Hartree-Fock equation) since it represents the cusp behavior of electron density in the local vicinity of nuclei. This function (or its approximation) is considered as the best candidate to be used as the localized basis function for atomic orbitals basis sets. Another direction is related to the construction of the accurate low-rank global interpolant for big scattered data to be considered in the next section. In this way, we calculate the data adaptive basis set living on the fine Cartesian grid in the region of target data. The main challenge, however, is due to the presence of point singularities which are hard to approximate in the problem independent polynomial or trigonometric basis sets.

The construction of low-rank RS approximation to the Slater function is based on the generalized Laplace transform representation for the Slater function written in the form $G(\rho) = e^{-2\sqrt{\alpha\rho}}$, $\rho(x) = x_1^2 + ... + x_d^2$, reads

$$G(\rho) = e^{-2\sqrt{\alpha\rho}} = \frac{\sqrt{\alpha}}{\sqrt{\pi}} \int_{\mathbb{R}_+} \tau^{-3/2} \exp(-\alpha/\tau - \rho\tau) d\tau,$$

which corresponds to the choice $\widehat{G}(\tau) = \frac{\sqrt{\alpha}}{\sqrt{\pi}} \tau^{-3/2} e^{-\alpha/\tau}$ in the canonical form of the Laplace transform representation for $G(\rho)$,

$$G(\rho) = \int_{\mathbb{R}_+} \widehat{G}(\tau) e^{-\rho\tau} d\tau. \tag{34}$$

Denote by $\mathbf{G}_R$ the rank-R canonical approximation to the function $G(\rho)$ discretized on the $n \times n \times n$ Cartesian grid.

Lemma 4.4. *([51]) For given threshold $\varepsilon > 0$ let $\rho \in [1, A]$. Then the $(2M+1)$-term sinc-quadrature approximation of the integral in (34) with*

$$M = O(|\log\varepsilon|(|\log\varepsilon| + \log A)),$$

ensures the max-error of the order of $O(\varepsilon)$ for the corresponding rank-$(2M+1)$ CP approximation $\mathbf{G}_R$ to the tensor $\mathbf{G}$.

Figure 5 illustrates the RS splitting for the tensor $\mathbf{G}_R = \mathbf{G}_{R_l} + \mathbf{G}_{R_s}$ representing the Slater potential $G(x) = e^{-\lambda\|x\|}$, $\lambda = 1$, discretized on the $n \times n \times n$ grid with $n = 1024$. The rank parameters are chosen by $R = 24, R_l = 6$ and $R_s = 18$. Notice that for this radial function the long-range part (**Figure 5**, left) includes much less canonical vectors comparing with the case of Newton kernel. This anticipates the smaller total canonical rank

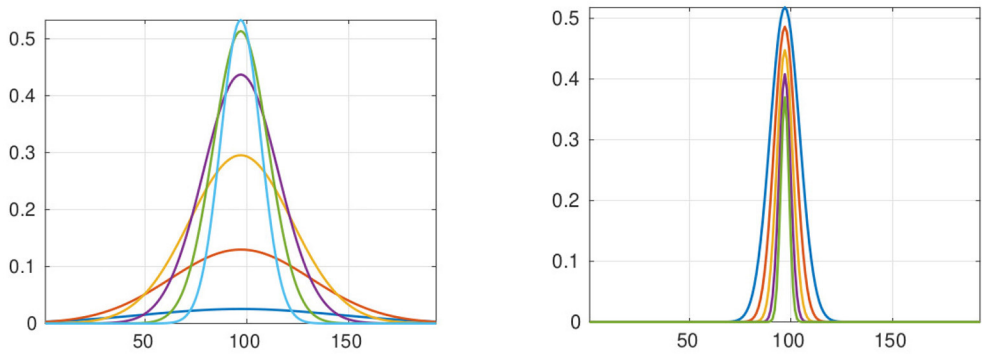


FIGURE 5 | Long-range (**left**) and short-range (**right**, a base 10 logarithmic scale) canonical vectors for the Slater function with the grid size $n = 1024$, $R = 24, R_l = 6, \lambda = 1$.

for the long-range part in the large sum of Slater-like potentials arising, for example, in the representation of molecular orbitals and the electron density in electronic structure calculations. For instance, the wave function for the Hydrogen atom is given by the Slater function $e^{-\mu\|x\|}$. In the following section, we consider the application of RS tensor format to interpolation of scattered data in $\mathbb{R}^d$.

4.5. Application of RHOSVD to Scattered Data Modeling

In scattered data modeling the problem is in a low parametric approximation of multi-variate functions $f : \mathbb{R}^d \to \mathbb{R}$ by sampling at a finite set $\mathcal{X} = \{x_1, \ldots, x_N\} \subset \mathbb{R}^d$ of piecewise distinct points. Here, the function f might be the surface of a solid body, the solution of a PDE, many-body potential field, multi-parametric characteristics of physical systems, or some other multi-dimensional data, etc.

Traditional ways of recovering f from a sampling vector $f_{|\mathcal{X}} = (f(x_1), \ldots, f(x_N))$ is the constructing a functional interpolant $P_N : \mathbb{R}^d \to \mathbb{R}$ such that $P_{N|\mathcal{X}} = f_{|\mathcal{X}} =: \mathbf{f} \in \mathbb{R}^N$, i.e.,

$$P_N(x_j) = f(x_j), \quad \forall\, 1 \le j \le N. \tag{35}$$

Using radial basis (RB) functions one can find interpolants P_N in the form

$$P_N(x) = \sum_{j=1}^{N} c_j p(\|x - x_j\|) + Q(x),$$
$$Q \text{ is some smooth function, say, polynomial,} \tag{36}$$

where $p = p(r) : [0, \infty) \to \mathbb{R}$ is a fixed RB function, and $r = \|\cdot\|$ is the Euclidean norm on $\mathbb{R}^d$. In further discussion, we set $Q(x) = 0$. For example, the following RB functions are commonly used

$$p = r^\nu, \quad (1+r^2)^\nu, \ (\nu \in \mathbb{R}), \quad \exp(-r^2), \quad \exp(-\lambda r), \quad r^2 \log(r).$$

The other examples of RB functions are defined by Green's kernels or by the class of Matérn functions [23].

We discuss the following computational tasks (A) and (B).

(A) For a fixed coefficient vector $\mathbf{c} = (c_1, \ldots, c_N)^T \in \mathbb{R}^N$, efficiently representing the interpolant $P_N(x)$ on the fine tensor grid in $\mathbb{R}^d$ providing
   (a) $O(1)$-fast point evaluation of P_N in the computational volume Ω,
   (b) computation of various integral-differential operations on that interpolant (say, gradients, scalar products, convolution integrals, etc.)

(B) Finding the coefficient vector $\mathbf{c}$ that solves the interpolation problem Equation (35) in the case of large number N.

Problem (A) exactly fits the RS tensor framework so that the RS tensor approximation solves the problem with low computational costs provided that the sum of long-range parts of the interpolating functions can be easily approximated in the low rank CP tensor format. We consider the case of interpolation by Slater functions $\exp(-\lambda r)$ in the more detail.

Problem (B): Suppose that we use some favorable preconditioned iteration for solving coefficient vector $\mathbf{c} = (c_1, \ldots, c_N)^T$,

$$A_{p,\mathcal{X}}\mathbf{c} = \mathbf{f}, \quad \text{with} \quad A_{p,\mathcal{X}} = A_{p,\mathcal{X}}^T = [p(\|x_i - x_j\|)]_{1 \le i,j \le N} \in \mathbb{R}^{N \times N}, \tag{37}$$

with the distance dependent symmetric system matrix $A_{p,\mathcal{X}}$. We assume $\mathcal{X} = \Omega_h$ be the $n^{\otimes d}$-set of grid-points located on tensor grid, i.e., $N = n^d$. Introduce the d-tuple multi-index $i \mapsto \mathbf{i} = (i_1, \ldots, i_d)$, and $j \mapsto \mathbf{j} = (j_1, \ldots, j_d)$ and reshape $A_{p,\mathcal{X}}$ into the tensor form

$$A_{p,\mathcal{X}} \mapsto \mathbf{A} = [a(i_1, j_1, \ldots, i_d, j_d)] \in \bigotimes_{\ell=1}^{d} \mathbb{R}^{n \times n},$$

which can be decomposed by using the RS based splitting

$$\mathbf{A} = \mathbf{A}_{R_s} + \mathbf{A}_{R_l},$$

generated by the RS representation of the weighted potential sum in Equation (36). Here, $\mathbf{A}_{R_s}$ is a banded diagonal matrix with dominating diagonal part, while $\mathbf{A}_{R_l} = \sum_{k=1}^{R_l} A_k^{(1)} \otimes$

$\cdots \otimes A_k^{(d)}$ is the low Kronecker rank matrix. This implies a bound on the storage, $O(N + dR_l n)$, and ensures a fast matrix-vector multiplication. Introducing the additional rank-structured representation in **c**, the solution of Equation (37) can be further simplified.

The above approach can be applied to the data sparse representation for the class of large covariance matrices in the spatial statistics, see for example [23, 55].

In application of tensor methods to data modeling (see Section 4.5) we consider the interpolation of 3D scattered data by a large sum of Slater functions

$$G_N(x) = \sum_{j=1}^{N} c_j e^{-\lambda\|x-x_j\|}, \quad \lambda > 0. \tag{38}$$

Given the coefficients c_j, we address the question how to efficiently represent the interpolant $G_N(x)$ on fine Cartesian grid in $\mathbb{R}^3$ by using the low-rank (i.e., low-parametric) CP tensor format, such that each value on the grid can be calculated in $O(1)$ operations. The main problem is that the generating Slater function $e^{-\lambda\|x\|}$ has the cusp at the origin so that the considered interpolant has very low regularity. As result, the tensor rank of the function $G_N(x)$ in Equation (38) discretized on a large $n \times n \times n$ grid increases almost proportionally to the number N of sampling points x_j, which in general may be very large. This increase in the canonical rank has been observed in a number of numerical tests. Hence, the straightforward tensor approximation of $G_N(x)$ does not work in this case.

Tables 2, **3** illustrate the stability of the canonical rank in the number N of sampling points in the case of random and function related distribution of the waiting coefficients c_j in the long-range part of the Slater interpolant Equation (38).

The generating Slater radial function can be proven to have the low-rank RS canonical tensor decomposition by using the sinc-approximation method (see section 4.1).

To complete this section, we present the numerical example demonstrating the application of RS tensor representation to scattered data modeling in $\mathbb{R}^3$. We denote by $\mathbf{G}_R \in \mathbb{R}^{n\otimes 3}$ the rank-R CP tensor approximation of the reference Slater potential $e^{-\lambda\|x\|}$ discretized on $n \times n \times n$ grid Ω_n, and introduce its RS splitting $\mathbf{G}_R = \mathbf{G}_{R_l} + \mathbf{G}_{R_s}$, with $R_l + R_s = R$. Here, $R_l \approx R/2$ is the rank parameter of the long-range part in $\mathbf{G}_R$. Assume that all measurement points x_j in Equation (38) are located on the discretization grid Ω_n, then the tensor representation of the long-range part of the total interpolant P_N can be obtained as the sum of the properly replicated reference potential $\mathbf{G}_l$, *via* the shift-and-windowing transform $\mathcal{W}_j, j = 1, \ldots, N$,

$$\mathbf{G}_{N,l} = \sum_{j=1}^{N} c_j \mathbf{G}_{l,j}, \quad \mathbf{G}_{l,j} = \mathcal{W}_j \mathbf{G}_l, \tag{39}$$

that includes about $N\,R_l$ terms. For large number of measurement points, N, the rank reduction is ubiquitous.

It can be proven (by slight modification of arguments in [32]) that both the CP and Tucker ranks of the N-term sum in Equation (39) depend only logarithmically (but not linearly) on N.

TABLE 2 | Reduced ranks for the case of random amplitudes.

$L_1 \times L_2 \times L_3$	N	Tucker ranks	R_{ini}	R_{comp}
$4 \times 4 \times 4$	64	$13 \times 13 \times 13$	192	56
$6 \times 6 \times 6$	216	$15 \times 15 \times 15$	649	95
$8 \times 8 \times 8$	512	$19 \times 19 \times 19$	1536	131
$16 \times 16 \times 8$	2048	$32 \times 32 \times 19$	6141	253
$16 \times 16 \times 16$	4096	$32 \times 32 \times 32$	12288	380

$\varepsilon_{Tuck} = 10^{-3}$, $\varepsilon_{T2C} = 10^{-5}$, $R_L = 3$, $R = 5$.

TABLE 3 | Reduced scanonical ranks for the case of functional amplitudes.

$L_1 \times L_2 \times L_3$	N	Tucker ranks	R_{ini}	R_{comp}
$4 \times 4 \times 4$	64	$7 \times 7 \times 7$	256	25
$6 \times 6 \times 6$	216	$7 \times 7 \times 7$	648	23
$8 \times 8 \times 8$	512	$7 \times 7 \times 7$	1536	30
$16 \times 16 \times 8$	2048	$10 \times 9 \times 6$	6144	47
$16 \times 16 \times 16$	4096	$10 \times 9 \times 8$	12288	55

$R_L = 3$, $R_L = 5$, $\varepsilon_N = 10^{-5}$, $\varepsilon_{Tuck} = 10^{-3}$, $\varepsilon_{T2C} = 10^{-5}$.

Proposition 4.5. *(Uniform rank bounds for the long-range part in the Slater interpolant). Let the long-range part* $\mathbf{G}_{N,l}$ *in the total Slater interpolant in Equation (39) be composed of those terms in Equation (24) which satisfy the relation* $t_k \leq 1$, *where* $M = O(\log^2 \varepsilon)$. *Then the total* ε*-rank* $\mathbf{r}_0$ *of the Tucker approximation to the canonical tensor sum* $\mathbf{G}_{N,l}$ *is bounded by*

$$|\mathbf{r}_0| := rank_{Tuck}(\mathbf{G}_{N,l}) = C\,b\,\log^{3/2}(\log(N/\varepsilon)), \tag{40}$$

where the constant C does not depend on the number of particles N, as well as on the size of the computational box, $[-b, b]^3$.

Proof: (Sketch) The main argument of the proof is based on the fact that the grid function $\mathbf{G}_{N,l}$ has the band-limited Fourier image, such that the frequency interval depends weakly (logarithmically) on N. Then we represent all Gaussians in the truncated Fourier basis and make the summation in the fixed set of orthogonal trigonometric basis functions, which defines the orthogonal Tucker representation with controllable rank parameter. □

The numerical illustrations below demonstrate the CP rank by RHOSVD decomposition of the long-range part $\mathbf{G}_{N,l}$ in the multi-point tensor interpolant *via* Slater functions.

Now, we generate a tensor composed of a sum of Slater functions, discretized by collocation over $n^{\otimes 3}$ representation grid with $n = 384$, and placed in the nodes of a sampling $L_1 \times L_2 \times L_3$ lattice with randomly chosen weights c_j in the interval $c_j \in [-5, 5]$ for every node. Every single Slater function is generated as a canonical tensor by using sinc-quadratures for the approximation of the related Laplace transform. **Table 2** shows ranks of the long-range part of this tensor composed of Slater potentials located in the nodes of the lattices of increasing size. N indicates the number of nodes, while R_{ini} and R_{comp}

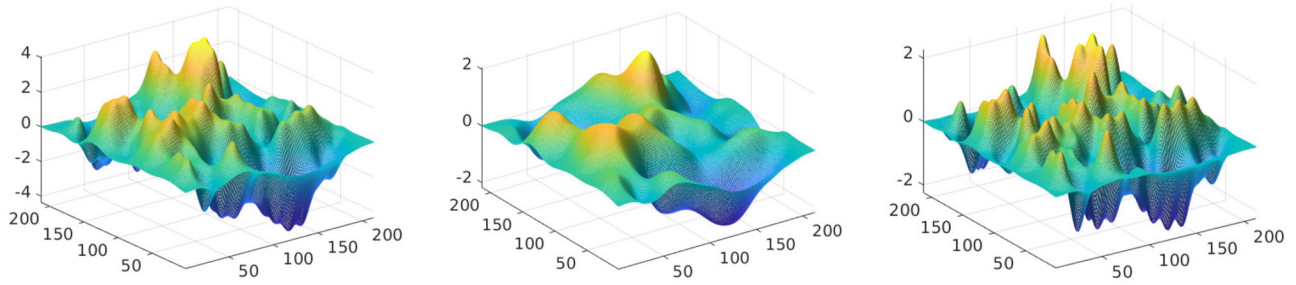


FIGURE 6 | Full-, long- and short-range components of the multi-Slater tensor. Slater kernels with $\lambda = 0.5$ and with random amplitudes in the range of $[-5, 5]$ are placed in the nodes of a $12 \times 12 \times 4$ lattice using 3D grid of size $n^{\otimes 3}$ with $n = 384$, $R = 8, R_l = 3$ and the number of nodes $N = 576$.

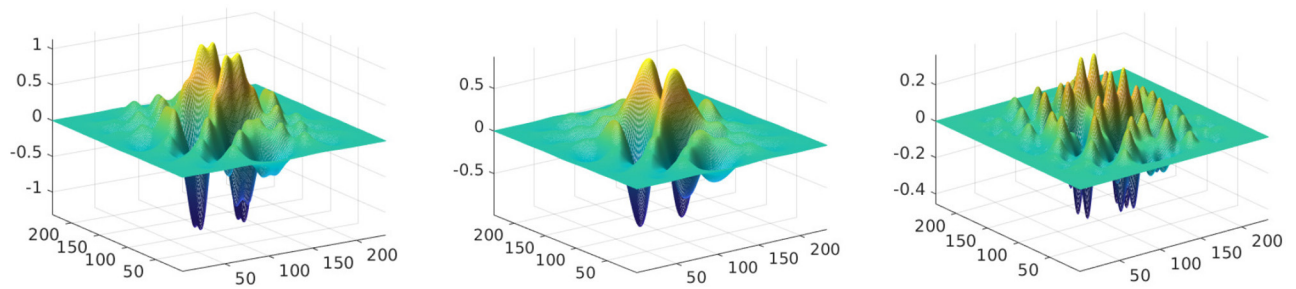


FIGURE 7 | Full-, long- and short-range components of the multi-Slater tensor. Slater kernels with $\lambda = 0.5$ and with amplitudes modulated by the function Equation (41) using the nodes of a $12 \times 12 \times 4$ lattice on 3D grid of size $n^{\otimes 3}$ with $n = 384$, $R = 8, R_l = 3$ and the number of nodes $N = 576$.

are the initial and compressed canonical ranks of the resulting long-range part tensor, respectively. Tucker ranks correspond to the ranks in the canonical-to-Tucker decomposition step. Threshold values for the Slater potential generator is $\varepsilon_N = 10^{-5}$, while the tolerance thresholds for the rank reduction procedure are given by $\varepsilon_{Tuck} = 10^{-3}$ and $\varepsilon_{T2C} = 10^{-5}$. We observe that the ranks of the long-range part of the potential increase only slightly in the size of the 3D sampling lattice, N.

Figure 6 demonstrates the full-, short-, and long-range components of the multi-Slater tensor constructed by the weighted sum of Slater functions with *randomly* chosen weights c_j in the interval $c_j \in [-5, 5]$. The positions of the generating nodes are located on the $12 \times 12 \times 4$ 3D lattice. The parameters of the tensor interpolant are set up as follows: $\lambda = 0.5$, the representation grid is of size $n^{\otimes 3}$ with $n = 384$, $R = 8, R_l = 3$ and the number of samples $N = 576$ (Figures zoom a part of the grid.). The initial CP rank of the sum of N_0 interpolating Slater potentials is about 4,468. Middle and right pictures show the long- and short-range parts of the composite tensor, respectively. The initial rank of the canonical tensor representing the long-range part is equal to $R_L = 2304$, which is reduced by the C2C procedure *via* RHOSVD to $R_{cc} = 71$. The rank truncation threshold is $\varepsilon = 10^{-3}$.

Figure 7 and **Table 3** demonstrate the decomposition of the multi-Slater tensor with the amplitudes c_j in the nodes (x_j, y_j, z_j) modulated by the function of the (x,y,z)-coordinates

$$F(x, y, z) = a_1 \cos(x + 2y + 4z) \exp(-a_2 \sqrt{x^2 + 2y^2 + 4z^2}), \quad (41)$$

with $a_1 = 6$ and $a_2 = 0.1$, , i.e., $c_j = F(x_j, y_j, z_j)$.

Next, we generate a tensor composed of a sum of discretized Slater functions on a sampling lattice $L_1 \times L_2 \times L_3$, living on 3D representation grid of size $n^{\otimes 3}$ with $n = 232$. The amplitudes of the individual Slater functions are modulated by a function of x, y, z-coordinates Equation (41) in every node of the lattice. **Table 3** shows rank of the long-range part of this multi-Slater tensor with respect to the increasing size of the lattice. $N = L_1 L_2 L_3$ is the number of nodes, and R_{ini} and R_{comp} are the initial and compressed canonical ranks, respectively. Tucker ranks are shown at the canonical-to-Tucker decomposition step. Threshold values for the Slater potential generation is $\varepsilon_N = 10^{-5}$, the thresholds for the canonical-to-canonical rank reduction procedure are given by $\varepsilon_{Tuck} = 10^{-3}$ and $\varepsilon_{T2C} = 10^{-5}$. **Table 3** demonstrates the very moderate icrease of the reduced rank in the long-range part of the Slater potential sum on the size of the 3D sampling lattice.

Figure 7 demonstrates the full-, long-, and short-range components of the multi-Slater tensor. Slater kernels with $\lambda = 0.5$ and with the amplitudes modulated by the function Equation (41) of the (x, y, z)-coordinates are places on the nodes of a $12 \times 12 \times 4$ sampling lattice, living on 3D grid of size $n^{\otimes 3}$ with

$n = 384$, $R = 8, R_l = 3$, and with the number of sampling nodes $N = 576$.

5. REPRESENTING GREEN'S KERNELS IN TENSOR FORMAT

In this section, we demonstrate how the RHOSVD can be applied for the efficient tensor decomposition of various singular radial functions composed by polynomial expansions of a few reference potentials already precomputed in the low-rank tensor format. Given the low-rank CP tensor $\mathbf{A}$ further considered as a reference tensor, the low rank representation of the tensor-valued polynomial function

$$P(\mathbf{A}) = a_0\mathbf{I} + a_1\mathbf{A} + a_2\mathbf{A}^2 + \cdots + a_n\mathbf{A}^n,$$

where the multiplication of tensors is understood in the sense of pointwise Hadamard product, can be calculated *via* n-times application of the RHOSVD by using the Horner scheme in the form

$$P(x) = a_0 + x(a_1 + x(a_2 + x(a_3 + \ldots x(a_{n-1} + a_n x)\ldots))).$$

Similar scheme can be also applied in the case of multivariate polynomials.

For examples considered, in this section, we make use of the discretized Slater $e^{-\|x\|}$ and Newton $\frac{1}{\|x\|}$, $x \in \mathbb{R}^d$, kernels as the reference tensors. The following statement was proven in [11, 51] (see also Lemma 4.4).

Proposition 5.1. *The discretized over $n^{\otimes d}$-grid radial functions $e^{-\|x\|}$ and $\frac{1}{\|x\|}$, $x \in \mathbb{R}^d$, included in representation of various Green kernels and fundamental solutions for elliptic operators with constant coefficients, both allow the low-rank CP tensor approximation. The corresponding rank-R representations can be calculated in $O(dRn)$ operations without precomputing and storage of the target tensor in the full (entry-wise) format.*

Tensor decomposition for discretized singular kernels such as $\|x\|$, $\frac{1}{\|x\|^m}$, $m \geq 2$, and $e^{-\kappa\|x\|}/\|x\|$, can be now calculated by applying the RHOSVD to polynomial combinations of the reference potentials as in Proposition 5.1. The most important benefit of the presented techniques is the opportunity to compute the rank-R tensor approximations without pre-computing and storage of the target tensor in the full format tensor.

In what follows, we present the particular examples of singular kernels in $\mathbb{R}^d$ which can be treated by the above presented techniques. Consider the fundamental solution of the advection-diffusion operator $\mathcal{L}_d$ with constant coefficients in $\mathbb{R}^d$

$$\mathcal{L}_d = -\Delta + 2\bar{b}\cdot\nabla + \kappa^2, \quad \bar{b} \in \mathbb{C}^d, \quad \kappa \in \mathbb{C}.$$

If $\kappa^2 + |\bar{b}|^2 = 0$, then for $d \geq 3$ it holds

$$\eta_0(x) = \frac{1}{(d-2)\omega_d}\frac{e^{\langle\bar{b},x\rangle}}{\|x\|^{d-2}},$$

where ω_d is the surface area of the unit sphere in $\mathbb{R}^d$, [56–58]. Notice that the radial function $\frac{1}{\|x\|^{d-2}}$ for $d \geq 3$ allows the RS decomposition of the corresponding discrete tensor representation based on the sinc quadrature approximation, which implies the RS representation of the kernel function $\eta_0(x)$, since the function $e^{\langle\bar{b},x\rangle}$ is already separable. From computational point of view, both the CP and RS canonical decompositions of discretized kernels $\frac{1}{\|x\|^{d-2}}$ can be computed by successive application of RHOSVD approximation to the products of canonical tensors for the discretized Newton potential $\frac{1}{\|x\|}$.

In the particular case $\bar{b} = 0$, we obtain the fundamental solution of the operator $\mathcal{L}_3 = -\Delta + \kappa^2$ for $d = 3$, also known as the Yukawa (for $\kappa \in \mathbb{R}_+$) or Helmholtz (for $\kappa \in \mathbb{C}$) Green kernels

$$\eta_\lambda(x) = \frac{1}{4\pi}e^{-\kappa\|x\|}/\|x\|, \quad x \in \mathbb{R}^3.$$

In the case of Yukawa kernel the tensor representations by using Gaussian sums are considered in [1, 2], see also references therein.

The Helmholtz equation with $\mathrm{Im}\,\kappa > 0$ (corresponds to the diffraction potentials) arises in problems of acoustics, electro-magnetics and optics. We refer to [59] for the detailed discussion of this class of fundamental solutions. Fast algorithms for the oscillating Helmholtz kernel have been considered in [1]. However, in this case the construction of the RS tensor decomposition remains an open question.

In the case of 3D biharmonic operator $\mathcal{L} = \Delta^2$ the fundamental solution reads as

$$p(\|x\|) = -\frac{1}{8\pi}\|x\|, \quad x \in \mathbb{R}^3.$$

The hydrodynamic potentials correspond to the classical Stokes operator

$$\nu\Delta u - \mathrm{grad}\, p = f, \quad \mathrm{div}\, u = 0,$$

where u is the velocity field, p denotes the pressure, and ν is the constant viscosity coefficient. The solution of the Stokes problem in $\mathbb{R}^3$ can be expressed by the hydrodynamic potentials

$$u_k(x) = \int_{\mathbb{R}^3}\sum_{\ell=1}^{3}\Psi_{k\ell}(x-y)f_\ell(y)\mathrm{d}y, \quad p(x) = \int_{\mathbb{R}^3}\langle\Theta(x-y),f)(y)\rangle\mathrm{d}y \tag{42}$$

with the fundamental solution

$$\Psi_{k\ell}(x) = \frac{1}{8\pi\nu}\left(\frac{\delta_{k\ell}}{\|x\|} + \frac{x_k x_\ell}{\|x\|^3}\right), \quad \Theta(x) = \frac{x}{4\pi\|x\|^3}, \quad x \in \mathbb{R}^3. \tag{43}$$

The existence of the low-rank RS tensor representation for the hydrodynamic potential is based on the same argument as in Remark 5.1. In turn, in the case of biharmonic fundamental solution we use the identity

$$\|x\| = \frac{\|x\|^2}{\|x\|},$$

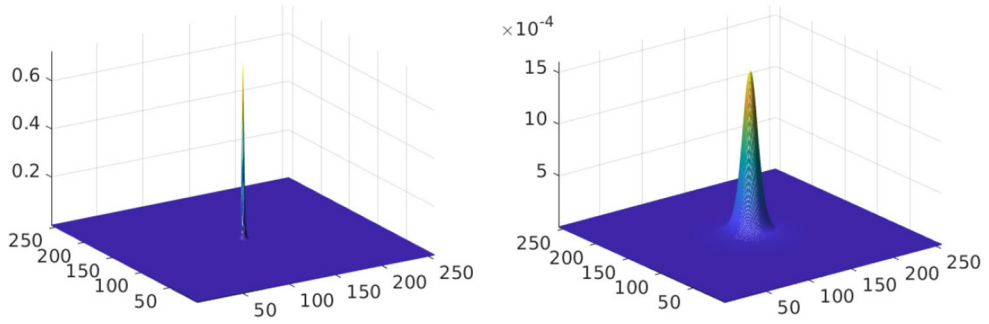


FIGURE 8 | RHOSVD approximation of the discretized cubic potential $\frac{1}{\|x\|^3}$ and its long-range part.

where the nominator has the separation rank equals to d. The latter representation can be also applied for calculation of the respective tensor approximations.

Here, we demonstrate how the application of RHOSVD allows to easily compute the low rank Tucker/CP approximation of the discretized singular potential $\frac{1}{\|x\|^3}$, $x \in \mathbb{R}^3$, as well as the respective RS-representation, having at hand the RS representation of the tensor $\mathbf{P} \in \mathbb{R}^{n\otimes 3}$ discretizing the Newton kernel. In this example, we use the discretization of $\frac{1}{\|x\|^3}$ in the form

$$\mathbf{P}^{(3)} = \mathbf{P} \odot \mathbf{P} \odot \mathbf{P},$$

where by $\mathbf{P}^{(3)}$ we denotes the collocation projection discretization of $\frac{1}{\|x\|^3}$. The low rank Tucker/CP tensor approximation to $\mathbf{P}^{(3)}$ can be computed by the direct application of the RHOSVD to the above product type representation. The RS representation of $\mathbf{P}^{(3)}$ is calculated based on Lemma 4.3. Given the RS-representation Equation (31) of the discretized Newton kernel, $\mathbf{P}_R$, we define the low rank CP approximation to the discretized singular part in the hydrodynamic potential $\mathbf{P}^{(3)}$ by

$$\mathbf{P}^{(3)}_{R'} = \mathbf{P}_R \odot \mathbf{P}_R \odot \mathbf{P}_R.$$

In view of Lemma 4.3, the long range part of RS decomposition of $\mathbf{P}^{(3)}_{R'}$, can be computed by RHOSVD approximation to the following Hadamard product of tensors,

$$\mathbf{P}^{(3)}_{R'_l} = \mathbf{P}_{R_l} \odot \mathbf{P}_{R_l} \odot \mathbf{P}_{R_l}.$$

Figure 8 visualizes the tensor $\mathbf{P}^{(3)}_{R'}$ as well as its long range part $\mathbf{P}^{(3)}_{R'_l}$.

The potentials are discretized on $n \times n \times n$ Cartesian grid with $n = 257$, the rank truncation threshold is chosen for $\varepsilon = 10^{-5}$. The CP rank of the Newton kernel is equal to $R = 19$, while we set $R_l = 10$, thus resulting in the initial ranks 6859 and 10^3 for RHOSVD decomposition of $\mathbf{P}^{(3)}_{R'}$ and $\mathbf{P}^{(3)}_{R'_l}$, respectively. The RHOSVD decomposition reduces the large rank parameters to $R' = 122$ (the Tucker rank is $r = 13$) and $R'_l = 58$ (the Tucker rank is $r = 8$), correspondingly.

6. RHOSVD FOR RANK REDUCTION IN 3D ELLIPTIC PROBLEM SOLVERS

Efficient rank reduction procedure based on the RHOSVD is a prerequisite for the development of the tensor-structured solvers for the three-dimensional elliptic problem, which reduce the computational complexity to almost linear scale, $O(nR)$, contrary to usual $O(n^3)$ complexity.

Assume that all input data in the governing PDE are given in the low-rank tensor form. The convenient tensor format for these problems is a canonical tensor representation of both the governing operator, and of the initial guess as well as of the right hand side. The commonly used numerical techniques are based on certain iterative schemes that include at each iterative step multiple matrix-vector and vector-vector algebraic operations each of them enlarges the tensor rank of the output in the additive or multiplicative way. It turns out that in common practice the most computationally intensive step in the rank-structured algorithms is the adaptive rank truncation, which makes the rank truncation procedure ubiquitous.

We notice that in PDE based mathematical models the total numerical complexity of the particular computational scheme, i.e., the overall cost of the rank truncation procedure is determined by the multiple of the number of calls to the rank truncation algorithm (merely the number of iterations) and the cost of a single RHOSVD transform (mainly determined by the rank parameter of the input tensor). In turn, both complexity characteristics depend on the quality of the rank-structured preconditioner so that optimization of the whole solution process is can be achieved by the trade-off between Kronecker rank of the preconditioner and the complexity of its implementation.

In the course of preconditioned iterations, the tensor ranks of the governing operator, the preconditioner and the iterand are multiplied, and, therefore, a robust rank reduction is mandatory procedure for such techniques applied to iterative solution of elliptic and pseudo-differential equations in the rank-structured tensor format.

In particular, the RHOSVD was applied to the numerical solution of PDE constrained (including the case of fractional operators) optimal control problems [36, 39], where the complexity of the order $O(nR \log n)$ was demonstrated.

In the case of higher dimensions the rank reduction in the canonical format can be performed directly (i.e., without intermediate use of the Tucker approximation) by using the cascading ALS iteration in the CP format, see [50] concerning the tensor-structured solution of the stochastic/parametric PDEs.

7. CONCLUSIONS

We discuss theoretical and computational aspects of the RHOSVD served for approximation of tensors in low-rank Tucker/canonical formats, and show that this rank reduction technique is the principal ingredient in tensor-based computations for real-life problems in scientific computing and data modeling. We recall rank reduction scheme for the canonical input tensors based on RHOSVD and subsequent Tucker-to-canonical transform. We present the detailed error analysis of low rank RHOSVD approximation to the canonical tensors (possibly with large input rank), and provide the proof on the uniform bound for the relative approximation error.

We recall that the first example on application of the RHOSVD was the rank-structured computation of the 3D convolution transform with the nonlocal Newton kernel in $\mathbb{R}^3$, which is the basic operation in the Hartree-Fock calculations.

The RHOSVD is the basic tools for utilizing the multilinear algebra in RS tensor format, which employs the sinc-analytic tensor approximation methods applied to the important class of radial functions in $\mathbb{R}^d$. This enables efficient rank decompositions of tensors generated by functions with multiple local cusps or singularities by separating their short- and long-range parts. As an example, we construct the RS tensor representation of the discretized Slater function $e^{-\lambda\|x\|}$, $x \in \mathbb{R}^d$. We then describe the RS tensor approximation to various Green's kernels obtained by combination of this function with other potentials, in particular, with the Newton kernel providing the Yukawa potential. In this way, we introduce the concept of reproducing radial functions which pave the way for efficient RS tensor decomposition applied to a wide range of function-related multidimensional data by combining the multilinear algebra in RS tensor format with the RHOSDV rank reduction techniques.

Our next example is related to application of RHOSVD to low-rank tensor interpolation of scattered data. Our numerical tests demonstrate the efficiency of this approach on the example of multi-Slater interpolant in the case of many measurement points. We apply the RHOSVD to the data generated *via* random or function modulated amplitudes of samples and demonstrate numerically that for both cases the rank of the long-range part remains small and depends weakly on the number of samples.

Finally, we notice that the described RHOSVD algorithms have proven their efficiency in a number of recent applications, in particular, in rank reduction for the tensor-structured iterative solvers for PDE constraint optimal control problems (including fractional control), in construction of the range-separated tensor representations for calculation of the electrostatic potentials of many-particle systems (arising in protein modeling), and for numerical analysis of large scattered data in $\mathbb{R}^d$.

DATA AVAILABILITY STATEMENT

The original contributions presented in the study are included in the article/supplementary material, further inquiries can be directed to the corresponding author/s.

AUTHOR CONTRIBUTIONS

BK: mathematical theory, basic concepts, and manuscript preparation. VK: basic concepts, numerical simulations, and manuscript preparation. All authors contributed to the article and approved the submitted version.

REFERENCES

1. Khoromskij BN. *Tensor Numerical Methods in Scientific Computing.* Berlin: De Gruyter Verlag (2018).
2. Khoromskaia V, and Khoromskij BN. *Tensor Numerical Methods in Quantum Chemistry.* Berlin: De Gruyter Verlag (2018).
3. P. Comon. "Tensor decompositions," in *Mathematics in Signal Processing V* (2002) 1–24.
4. Comon P, Luciani X, and De Almeida ALF. Tensor decompositions, alternating least squares and other tales. *J Chemometr J Chemometr Soc.* (2009) 23:393–405. doi: 10.1002/CEM.1236
5. Smilde A, Bro R, and Geladi P. *Multi-Way Analysis With Applications in the Chemical Sciences.* Wiley (2004).
6. Cichocki A, Lee N, Oseledets I, Pan AH, Zhao Q, and Mandic DP. Tensor networks for dimensionality reduction and large-scale optimization: part 1 low-rank tensor decompositions. *Found Trends Mach Learn* (2016) 9:249–429. doi: 10.1561/2200000059
7. De Lathauwer L, De Moor B, and Vandewalle J. A multilinear singular value decomposition. *SIAM J. Matrix Anal. Appl.* (2000) 21:1253–78. doi: 10.1137/S0895479896305696
8. Ten Berge JMF, and Sidiropoulos ND. On uniqueness in CANDECOMP/PARAFAC. *Psychometrika* (2002) 67:399–409.
9. Sidiropoulos ND, De Lathauwer L, Fu X, Huang K, Papalexakis EE, Faloutsos C. Tensor decomposition for signal processing and machine learning. *IEEE Trans Signal Process.* (2017) 65:3551–82. doi: 10.1109/TSP.2017.2690524
10. Golub GH, and Van Loan F. *Matrix Computations.* Baltimore, MD: Johns Hopkins University Press (1996).
11. Hackbusch W, and Khoromskij BN. Low-rank Kronecker product approximation to multi-dimensional nonlocal operators. Part I. Separable approximation of multi-variate functions. *Computing.* (2006) 76:177–202. doi: 10.1007/s00607-005-0144-0
12. Khoromskij BN, and Khoromskaia V. Low rank tucker-type tensor approximation to classical potentials. *Central Eur J Math.* (2007) 5:523–50. doi: 10.2478/s11533-007-0018-0
13. Marcati C, Rakhuba M,and Schwab C. Tensor rank bounds for point singularities in $\mathbb{R}^3$. *E-preprint* arXiv:1912.07996, (2019).
14. Hitchcock FL. The expression of a tensor or a polyadic as a sum of products. *J Math Phys.* (1927) 6:164–89.
15. Harshman RA. "Foundations of the PARAFAC procedure: models and conditions for an "explanatory" multimodal factor analysis," In: *UCLA Working Papers Phonetics.* vol. 16 (1970). 1–84.
16. Tucker LR. Some mathematical notes on three-mode factor analysis. *Psychometrika* (1966) 31:279–311.
17. Oseledets IV, and Tyrtyshnikov EE, Breaking the curse of dimensionality, or how to use svd in many dimensions. *SIAM J Sci Comput.* (2009) 31:3744–59. doi: 10.1137/090748330
18. Oseledets IV. Tensor-train decomposition. *SIAM J Sci Comput.* (2011) 33:2295–317. doi: 10.1137/090752286

19. Hackbusch W, and Kühn S. A new scheme for the tensor representation. *J. Fourier Anal. Appl.* (2009) 15:706–22. doi: 10.1007/s00041-009-9094-9
20. Khoromskij BN. $O(d \log N)$-quantics approximation of N-d tensors in high-dimensional numerical modeling. *Construct Approx.* (2011) 34:257–89. doi: 10.1007/S00365-011-9131-1
21. Oseledets I. Constructive representation of functions in low-rank tensor formats. *Constr. Approx.* (2013) 37:1–18. doi: 10.1007/s00365-012-9175-x
22. Kressner D, Steinlechner M, and Uschmajew A. Low-rank tensor methods with subspace correction for symmetric eigenvalue problems. *SIAM J Sci Comput.* (2014) 36:A2346–68. doi: 10.1137/130949919
23. Litvinenko A, Keyes D, Khoromskaia V, Khoromskij BN, and Matthies HG. Tucker tensor analysis of Matern functions in spatial statistics. *Comput. Meth. Appl. Math.* (2019) 19:101–22. doi: 10.1515/cmam-2018-0022
24. Rakhuba M, and Oseledets I. Fast multidimensional convolution in low-rank tensor formats via cross approximation. *SIAM J Sci Comput* (2015) 37:A565–82. doi: 10.1137/140958529
25. Uschmajew A. Local convergence of the alternating least squares algorithm for canonical tensor approximation. *SIAM J Mat Anal Appl* (2012) 33:639–52. doi: 10.1137/110843587
26. Hackbusch W and Uschmajew A. Modified iterations for data-sparse solution of linear systems. *Vietnam J Math.* (2021) 49:493–512. doi: 10.1007/s10013-021-00504-9
27. V. Khoromskaia. *Numerical Solution of the Hartree-Fock Equation by Multilevel Tensor-structured methods.* Berlin: TU Berlin (2010).
28. Khoromskaia V, and Khoromskij BN. Grid-based lattice summation of electrostatic potentials by assembled rank-structured tensor approximation. *Comp. Phys. Commun.* (2014) 185:3162–74. doi: 10.1016/j.cpc.2014.08.015
29. Khoromskij BN, Khoromskaia V, and Flad H-J. Numerical solution of the hartree-fock equation in multilevel tensor-structured format. *SIAM J. Sci. Comput.* (2011) 33:45–65. doi: 10.1137/090777372
30. Dolgov SV, Khoromskij BN, and Oseledets I. Fast solution of multi-dimensional parabolic problems in the TT/QTT formats with initial application to the Fokker-Planck equation. *SIAM J. Sci. Comput.* (2012) 34:A3016–38. doi: 10.1137/120864210
31. Kazeev, M. Khammash, M. Nip, and Ch. Schwab. Direct solution of the chemical master equation using quantized tensor trains. *PLoS Comput Biol.* (2014) 10:e1003359. doi: 10.1371/journal.pcbi.1003359
32. Benner P, Khoromskaia V, and Khoromskij BN. Range-separated tensor format for many-particle modeling. *SIAM J Sci Comput.* (2018) 40:A1034-062. doi: 10.1137/16M1098930
33. Benner P, Khoromskaia V, Khoromskij BN, Kweyu C, and Stein M. Regularization of Poisson–Boltzmann type equations with singular source terms using the range-separated tensor format. *SIAM J Sci Comput.* (2021) 43:A415-45. doi: 10.1137/19M1281435
34. Khoromskij BN. Range-separated tensor decomposition of the discretized Dirac delta and elliptic operator inverse. *J Comput Phys.* (2020) 401:108998. doi: 10.1016/j.jcp.2019.108998
35. Kweyu C, Khoromskaia V, Khoromskij B, Stein M, and Benner P. Solution decomposition for the nonlinear Poisson-Boltzmann equation using the range-separated tensor format. *arXiv preprint* arXiv:2109.14073. (2021).
36. Heidel G, Khoromskaia V, Khoromskij BN, and Schulz V. Tensor product method for fast solution of optimal control problems with fractional multidimensional Laplacian in constraints. *J Comput Phys.* (2021) 424:109865. doi: 10.1016/j.jcp.2020.109865
37. Dolgov S, Kalise D, and Kunisch KK. Tensor decomposition methods for high-dimensional Hamilton–Jacobi–Bellman equations. *SIAM J. Sci. Comput.* (2021) 43:A1625–50. doi: 10.1137/19M1305136
38. Dolgov S, and Pearson JW. Preconditioners and tensor product solvers for optimal control problems from chemotaxis. *SIAM J. Sci. Comput.* (2019) 41:B1228–53. doi: 10.1137/18M1198041
39. Schmitt B, Khoromskij BN, Khoromskaia V, and Schulz V. Tensor method for optimal control problems constrained by fractional three-dimensional elliptic operator with variable coefficients. *Numer. Lin Algeb Appl.* (2021) 1–24:e2404. doi: 10.1002/nla.2404
40. Bachmayr M, Schneider R, and Uschmajew A. Tensor networks and hierarchical tensors for the solution of high-dimensional partial differential equations. *Found Comput Math.* (2016) 16:1423–72. doi: 10.1007/s10208-016-9317-9
41. Boiveau T, Ehrlacher V, Ern A, and Nouy A. Low-rank approximation of linear parabolic equations by space-time tensor Galerkin methods. *ESAIM Math Model Numer Anal* (2019) 53:635-58. doi: 10.1051/m2an/2018073
42. Espig M, Hackbusch W, Litvinenko A, Matthies HG, Zander E. Post-processing of high-dimensional data. (2019) *E-Preprint* arXiv:1906.05669.
43. Ch. Lubich, T. Rohwedder, R. Schneider and B. Vandereycken. Dynamical approximation of hierarchical Tucker and tensor-train tensors. *SIAM J Matrix Anal. Appl.* (2013) 34:470–94. doi: 10.1137/120885723
44. A. Litvinenko, Y. Marzouk, H. G. Matthies, M. Scavino, A. Spantini. Computing f-divergences and distances of high-dimensional probability density functions–low-rank tensor approximations. *E-preprint* arXiv:2111.07164. (2021).
45. Grasedyck L. Hierarchical singular value decomposition of tensors. *SIAM. J. Matrix Anal. Appl.* (2010) 31:2029. doi: 10.1137/090764189
46. Khoromskij BN, and Khoromskaia V. Multigrid tensor approximation of function related arrays. *SIAM J. Sci. Comput.* (2009) 31:3002–26. doi: 10.1137/080730408
47. Ehrlacher V, Grigori L, Lombardi D, and Song H. Adaptive hierarchical subtensor partitioning for tensor compression. *SIAM J. Sci. Comput.* (2021) 43:A139–63. doi: 10.1137/19M128689X
48. Kressner D, and Uschmajew A. On low-rank approximability of solutions to high-dimensional operator equations and eigenvalue problems. *Lin Algeb Appl.* (2016) 493:556–72. doi: 10.1016/J.LAA.2015.12.016
49. Oseledets IV, Rakhuba MV, and Uschmajew A. Alternating least squares as moving subspace correction. *SIAM J Numer Anal.* (2018) 56:3459–79. doi: 10.1137/17M1148712
50. Khoromskij BN, and Schwab C. Tensor-structured Galerkin approximation of parametric and stochastic elliptic PDEs. *SIAM J. Sci. Comput.* (2011) 33:364–85. doi: 10.1137/100785715
51. Khoromskij BN. Structured rank-$(r_1, \ldots, r_d)$ decomposition of function-related tensors in $\mathbb{R}^d$. *Comput. Meth. Appl. Math.* (2006) 6:194–220. doi: 10.2478/cmam-2006-0010
52. Stenger F. *Numerical Methods Based on Sinc and Analytic Functions.* Springer-Verlag (1993).
53. Braess D. *Nonlinear Approximation Theory.* Berlin: Springer-Verlag (1986).
54. Khoromskaia V, and Khoromskij BN. Prospects of Tensor-Based Numerical Modeling of the Collective Electrostatics in Many-Particle Systems. *Comput Math Math Phys.* (2021) 61:864–86. doi: 10.1134/S0965542521050110
55. Matérn B. *Spatial Variation, Vol. 36 of Lecture Notes in Statistics.* 2nd Edn. Berlin: Springer-Verlag (1986).
56. Abramowitz M, and Stegun IA. *Handbook of Mathematical Functions.* New York, NY: Dover Publ., (1968).
57. Sauter SA, and Schwab C. *Boundary Element Methods.* Springer (2011).
58. Hsiao GC, and Wendland WL. *Boundary Integral Equations.* Berlin: Springer (2008).
59. Maz'ya V, and Schmidt G. Approximate approximations. *Math Surv Monograph* (2007) 141:349.

Conflict of Interest: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

8. APPENDIX: PROOFS OF THEOREM 2.3 AND LEMMA 2.5

Proof of Theorem 2.3.

Proof: Using the contracted product representations of $\mathbf{A} \in \mathcal{C}_{R,\mathbf{n}}$ and $\mathbf{A}^0_{(\mathbf{r})} \in \mathcal{T}_{\mathbf{r}}$, and introducing the ℓ-mode residual

$$\Delta^{(\ell)} = U^{(\ell)} - Z_0^{(\ell)} D_{\ell,0} V_0^{(\ell)^T},$$
$$\{\Delta^{(\ell)}\}_\nu = \sum_{k=r_\ell+1}^{n} \sigma_{\ell,k} \mathbf{z}_k^{(\ell)} v_{k,\nu}^{\ell}, \quad \nu = 1,\dots,R,$$

with notations $V_0^{(\ell)} = [\mathbf{v}_1^{(\ell)},...,\mathbf{v}_{r_\ell}^{(\ell)}]^T$, $\mathbf{v}_k^{(\ell)} = \{v_{k,\nu}^{\ell}\}_{\nu=1}^{R} \in \mathbb{R}^R$, we arrive at the following expansion for the approximation error in the form

$$\mathbf{A} - \mathbf{A}^0_{(\mathbf{r})} = \boldsymbol{\xi} \times_1 U^{(1)} \times_2 \cdots \times_d U^{(d)} - \boldsymbol{\xi} \times_1 W^{(1)} \times_2 \cdots \times_d W^{(d)} := \sum_{\ell=1}^{d} \mathbf{B}_\ell,$$

where

$$\begin{aligned}\mathbf{B}_\ell &= \boldsymbol{\xi} \times_1 U^{(1)} \cdots \times_{\ell-1} U^{(\ell-1)} \times_\ell \Delta^{(\ell)} \times_{\ell+1} W^{(\ell+1)} \cdots \times_d W^{(d)} \\ &= \sum_{\nu=1}^{R} \xi_\nu \Bigg[\mathbf{u}_\nu^{(\ell)} \cdots \times_{\ell-1} \mathbf{u}_\nu^{(\ell-1)} \times_\ell \{\Delta^{(\ell)}\}_\nu \times_{\ell+1} \sum_{k=1}^{r_{\ell+1}} \sigma_{\ell+1,k} \mathbf{z}_k^{(\ell+1)} v_{k,\nu}^{\ell+1} \cdots \times_d \sum_{k=1}^{r_d} \sigma_{d,k} \mathbf{z}_k^{(d)} v_{k,\nu}^{d} \Bigg].\end{aligned}$$

This leads to the error bound (by the triangle inequality)

$$\|\mathbf{A} - \mathbf{A}^0_{(\mathbf{r})}\| \le \sum_{\ell=1}^{d} \|\mathbf{B}_\ell\|,$$

providing the estimate (in view of $\|\mathbf{u}_\nu^{(\ell)}\| = 1$, $\ell = 1,\dots,d$, $\nu = 1,\dots,R$)

$$\|\mathbf{B}_\ell\| \le \sum_{\nu=1}^{R} |\xi_\nu| \left(\sum_{k=r_\ell+1}^{n} \sigma_{\ell,k}^2 (v_{k,\nu}^{\ell})^2 \right)^{1/2} \times \left(\sum_{k=1}^{r_{\ell+1}} \sigma_{\ell+1,k}^2 (v_{k,\nu}^{\ell+1})^2 \right)^{1/2} \cdots \left(\sum_{k=1}^{r_d} \sigma_{d,k}^2 (v_{k,\nu}^{d})^2 \right)^{1/2}.$$

Furthermore, since $U^{(\ell)}$ has normalized columns, i.e., $\|\mathbf{u}_\nu^{(\ell)}\| = \|\sum_{k=1}^{n} \sigma_{\ell,k} \mathbf{z}_k^{(\ell)} v_{k,\nu}^{\ell}\| = 1$, $\ell = 1,\dots,d$, we obtain $\sum_{k=1}^{n} \sigma_{\ell,k}^2 (v_{k,\nu}^{\ell})^2 = 1$ for $\ell = 1,\dots,d$ $\nu = 1,\dots,R$. Now the error estimate follows

$$\begin{aligned}\|\mathbf{A} - \mathbf{A}^0_{(\mathbf{r})}\| &\le \sum_{\ell=1}^{d} \sum_{\nu=1}^{R} |\xi_\nu| \left(\sum_{k=r_\ell+1}^{n} \sigma_{\ell,k}^2 (v_{k,\nu}^{\ell})^2 \right)^{1/2} \\ &\le \sum_{\ell=1}^{d} \left(\sum_{\nu=1}^{R} \xi_\nu^2 \right)^{1/2} \left(\sum_{\nu=1}^{R} \sum_{k=r_\ell+1}^{n} \sigma_{\ell,k}^2 (v_{k,\nu}^{\ell})^2 \right)^{1/2} \\ &= \sum_{\ell=1}^{d} \|\boldsymbol{\xi}\| \left(\sum_{k=r_\ell+1}^{n} \sigma_{\ell,k}^2 \sum_{\nu=1}^{R} (v_{k,\nu}^{\ell})^2 \right)^{1/2} \\ &= \|\boldsymbol{\xi}\| \sum_{\ell=1}^{d} \left(\sum_{k=r_\ell+1}^{n} \sigma_{\ell,k}^2 \right)^{1/2}.\end{aligned}$$

The case $R < n$ can be analyzed along the same line. □

Proof of Lemma 2.5.

Proof: (A) The canonical vectors $\mathbf{y}_k^{(\ell)}$ of any test element on the left-hand side of (11),

$$\mathbf{Z} = \sum_{k=1}^{R} \lambda_k \, \mathbf{y}_k^{(1)} \otimes \ldots \otimes \mathbf{y}_k^{(d)} \in \mathcal{C}_{R,\mathbf{n}}, \tag{A1}$$

can be chosen in $span\{\mathbf{v}_1^{(\ell)},\dots,\mathbf{v}_{r_\ell}^{(\ell)}\}$, that means

$$\mathbf{y}_k^{(\ell)} = \sum_{m=1}^{r_\ell} \mu_{k,m}^{(\ell)} \mathbf{v}_m^{(\ell)}, \quad k = 1,\dots,R, \ \ell = 1,\dots,d. \tag{A2}$$

Indeed, assuming

$$\mathbf{y}_k^{(\ell)} = \sum_{m=1}^{r_\ell} \mu_{k,m}^{(\ell)} \mathbf{v}_m^{(\ell)} + \mathbf{e}_k^{(\ell)} \quad \text{with} \quad \mathbf{e}_k^{(\ell)} \perp span\{\mathbf{v}_1^{(\ell)},\dots,\mathbf{v}_{r_\ell}^{(\ell)}\},$$

we conclude that $\mathbf{e}_k^{(\ell)}$ does not effect the cost function in (11) because of the orthogonality of $V^{(\ell)}$. Hence, setting $\mathbf{e}_k^{(\ell)} = 0$, and plugging (A2) in (A1), we arrive at the desired Tucker decomposition of $\mathbf{Z}$, $\mathbf{Z} = \boldsymbol{\beta}_z \times_1 V^{(1)} \times_2 \ldots \times_d V^{(d)}$, $\boldsymbol{\beta}_z \in \mathcal{C}_{R,\mathbf{r}}$. This implies

$$\begin{aligned}\|\mathbf{A} - \mathbf{Z}\|^2 &= \|(\boldsymbol{\beta}_z - \boldsymbol{\beta}) \times_1 V^{(1)} \times_2 \ldots \times_d V^{(d)}\|^2 \\ &= \|\boldsymbol{\beta} - \boldsymbol{\beta}_z\|^2 \ge \min_{\boldsymbol{\mu} \in \mathcal{C}_{R,\mathbf{r}}} \|\boldsymbol{\beta} - \boldsymbol{\mu}\|^2.\end{aligned}$$

On the other hand, we have

$$\min_{\mathbf{Z}\in\mathcal{C}_{R,\mathbf{n}}} \|\mathbf{A}-\mathbf{Z}\|^2 \le \min_{\boldsymbol{\beta}_z\in\mathcal{C}_{R,\mathbf{r}}} \|(\boldsymbol{\beta}-\boldsymbol{\beta}_z)\times_1 V^{(1)}\times_2 \ldots \times_d V^{(d)}\|^2 = \min_{\boldsymbol{\mu}\in\mathcal{C}_{R,\mathbf{r}}} \|\boldsymbol{\beta}-\boldsymbol{\mu}\|^2.$$

This proves (11).

(B) Likewise, for any minimizer $\mathbf{A}_{(R)} \in \mathcal{C}_{R,\mathbf{n}}$ in the right-hand side in (11), one obtains

$$\mathbf{A}_{(R)} = \boldsymbol{\beta}_{(R)} \times_1 V^{(1)} \times_2 V^{(2)} \ldots \times_d V^{(d)}$$

with the respective rank-R core tensor $\boldsymbol{\beta}_{(R)} = \sum_{k=1}^{R} \lambda_k \mathbf{u}_k^{(1)} \otimes \ldots \otimes \mathbf{u}_k^{(d)} \in \mathcal{C}_{R,\mathbf{r}}$. Here $\mathbf{u}_k^{(\ell)} = \{\mu_{k,m_\ell}^{(\ell)}\}_{m_\ell=1}^{r_\ell} \in \mathbb{R}^{r_\ell}$, are calculated by plugging representation (A2) in (A1), and then by changing the order of summation,

$$\begin{aligned}\mathbf{A}_{(R)} &= \sum_{k=1}^{R} \lambda_k \mathbf{y}_k^{(1)} \otimes \ldots \otimes \mathbf{y}_k^{(d)} \\ &= \sum_{k=1}^{R} \lambda_k \left(\sum_{m_1=1}^{r_1} \mu_{k,m_1}^{(1)} \mathbf{v}_{m_1}^{(1)}\right) \otimes \ldots \otimes \left(\sum_{m_d=1}^{r_d} \mu_{k,m_d}^{(d)} \mathbf{v}_{m_d}^{(d)}\right) \\ &= \sum_{m_1=1}^{r_1} \cdots \sum_{m_d=1}^{r_d} \left\{\sum_{k=1}^{R} \lambda_k \prod_{\ell=1}^{d} \mu_{k,m_\ell}^{(\ell)}\right\} \mathbf{v}_{m_1}^{(1)} \otimes \ldots \otimes \mathbf{v}_{m_d}^{(d)}.\end{aligned}$$

Now (12) implies that

$$\|\mathbf{A}-\mathbf{A}_R\| = \|\boldsymbol{\beta}-\boldsymbol{\beta}_{(R)}\|,$$

since the ℓ-mode multiplication with the orthogonal side matrices $V^{(\ell)}$ does not change the cost functional. Inspection of the left-hand side in (11) indicates that the latter equation ensures that $\boldsymbol{\beta}_{(R)}$ is, in fact, the minimizer of the right-hand side in (11). □

frontiers | Frontiers in Applied Mathematics and Statistics

METHODS
published: 01 June 2022
doi: 10.3389/fams.2022.806549

A Practical Guide to the Numerical Implementation of Tensor Networks I: Contractions, Decompositions, and Gauge Freedom

*Glen Evenbly**

School of Physics, Georgia Institute of Technology, Atlanta, GA, United States

We present an overview of the key ideas and skills necessary to begin implementing tensor network methods numerically, which is intended to facilitate the practical application of tensor network methods for researchers that are already versed with their theoretical foundations. These skills include an introduction to the contraction of tensor networks, to optimal tensor decompositions, and to the manipulation of gauge degrees of freedom in tensor networks. The topics presented are of key importance to many common tensor network algorithms such as DMRG, TEBD, TRG, PEPS, and MERA.

Keywords: tensor network algorithm, MPS, tensor contraction, DMRG, quantum many body theory

OPEN ACCESS

Edited by:
Edoardo Angelo Di Napoli, Julich Research Center, Helmholtz Association of German Research Centres (HZ), Germany

Reviewed by:
Jutho Haegeman, Ghent University, Belgium
Jiajia Li, College of William & Mary, United States

***Correspondence:**
Glen Evenbly
glen.evenbly@gmail.com

Specialty section:
This article was submitted to Mathematics of Computation and Data Science, a section of the journal Frontiers in Applied Mathematics and Statistics

Received: *31 October 2021*
Accepted: *02 May 2022*
Published: *01 June 2022*

Citation:
Evenbly G (2022) A Practical Guide to the Numerical Implementation of Tensor Networks I: Contractions, Decompositions, and Gauge Freedom. Front. Appl. Math. Stat. 8:806549. doi: 10.3389/fams.2022.806549

1. INTRODUCTION

Tensor networks have been developed as a useful formalism for the theoretical understanding of quantum many-body wavefunctions [1–10], especially in regards to entanglement [11–13], and are also applied as powerful numeric tools and simulation algorithms. Although developed primarily for the description of quantum many-body systems, they have since found use in a plethora of other applications such as quantum chemistry [14–18], holography [19–24], machine learning [25–29] and the simulation of quantum circuits [30–35].

There currently exist many useful references designed to introduce newcomers to the underlying theory of tensor networks [1–10]. Similarly, for established tensor network methods, there often exist instructional or review articles that address the particular method in great detail [36–40]. Nowadays, many research groups have also made available tensor network code libraries [41–48]. These libraries typically allow other researchers to make use of highly optimized tensor network routines for practical purposes (such as for the numerical simulation of quantum many-body systems).

Comparatively few are resources intended to help researchers that already possess a firm theoretical grounding to begin writing their own numerical implementations of tensor network codes. Yet such numerical skills are essential in many areas of tensor network research: new algorithmic proposals typically require experimentation, testing and bench-marking using numerics. Furthermore, even researchers solely interested in the application of tensor network methods to a problem of interest may be required to program their own version of a method, as a pre-built package may not contain the necessary features as to be suitable for the unique problem under consideration. The purpose of our present work is to help fill this aforementioned gap: to aid students and researchers, who are assumed to possess some prior theoretical understanding of tensor networks, to learn the practical skills required to program their own tensor networks codes and libraries. Indeed, our intent is to arm the interested reader with the key knowledge

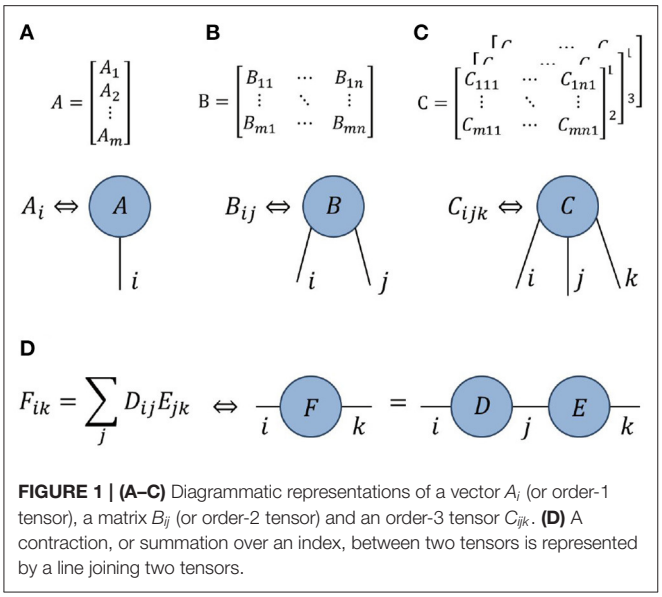


FIGURE 1 | (A–C) Diagrammatic representations of a vector A_i (or order-1 tensor), a matrix B_{ij} (or order-2 tensor) and an order-3 tensor C_{ijk}. **(D)** A contraction, or summation over an index, between two tensors is represented by a line joining two tensors.

that would allow them to implement their own versions of algorithms such as the density matrix renormalization group (DMRG) [49–51], time-evolving block decimation (TEBD) [52, 53], projected entangled pair states (PEPS) [54–56], multi-scale entanglement renormalization ansatz (MERA) [57], tensor renormalization group (TRG) [58, 59], or tensor network renormalization (TNR) [60]. Furthermore, this manuscript is designed to compliment online tensor network tutorials [61], which have a focus on code implementation, with more detailed explanations on tensor network theory.

2. PRELIMINARIES

2.1. Prior Knowledge

As stated above, the goal of this manuscript is to help readers that already possess some understanding of tensor network theory to apply this knowledge toward numeric calculations. Thus we assume that the reader has some basic knowledge of tensor networks, specifically that they understand what a tensor network is and have some familiarity with the standard diagrammatic notation used to represent them. An overview of these concepts is presented in **Figure 1**, otherwise more comprehensive introductions to tensor network theory can be found in [1–10].

Note that we shall not assume prior knowledge of quantum many-body physics, which is the most common application of tensor network algorithms. The skills and ideas that we introduce in this manuscript are intended to be general for the tensor network formalism, rather than for their use in a specific application, thus can also carry over to other area in which tensor networks have proven useful such as quantum chemistry [14–18], holography [19–24], machine learning [25–29], and the simulation of quantum circuits [30–35].

2.2. Software Libraries

Currently there exists a wide variety of tensor network code libraries, which include [41–48]. Many of these libraries differ greatly in not only their functioning but also their intended applications, and may have their own specific strengths and weaknesses (which we will not attempt to survey in the present manuscript). Almost all of these libraries contain tools to assist in the tasks described in this manuscript, such as contracting, decomposing and re-gauging tensor networks. Additionally many of these libraries also contain full featured versions of complete tensor network algorithms, such as DMRG or TEBD. For a serious numerical calculation involving tensor networks, one where high performance is required, most researchers would be well-advised to utilize an existing library.

However, even if ultimate intent is to use existing library, it is still desirable that one should understand the fundamental tensor network manipulations used in numerical calculations. Indeed, this understanding is necessary to properly discern the limitations of various tensor network tools, to ensure that they are applied in an appropriate way, and to customize the existing tools if necessary. Moreover, exploratory research into new tensor network ansatz, algorithms and applications often requires non-standard operations and tensor manipulations which may not be present in any existing library, thus may require the development of extensive new tensor code. In this setting it can be advantageous to minimize or to forgo the usage of an existing library (unless one was already intimately familiar with its inner workings), given the inherent challenge of extending a library beyond its intended function and the possibility of unintended behavior that this entails.

In the remaining manuscript we aim to describe key tensor network operations (namely contracting, decomposing and re-gauging tensor networks) with sufficient detail that would allow the interested reader to implement tasks numerically without the need to rely on an existing code library.

2.3. Programming Language

Before attempting to implement tensor methods numerically one must, of course, decide on which programming language to use. High-level languages with a focus on scientific computation, such as MATLAB, Julia, and Python (with Numpy) are well-suited for implementing tensor network methods as they have native support for multi-dimensional arrays (i.e., tensors), providing simple and convenient syntax for common operations on these arrays (indexing, slicing, scalar operations) as well as providing a plethora of useful functions for manipulating these tensor objects. Alternatively, some tensor network practitioners may prefer to use lower-level languages such as Fortran or C++ when implementing tensor network algorithms usually for the reason of maximizing performance. However, in many tensor network codes the bulk of the computation time is spent performing large matrix-matrix multiplications, for which even interpreted languages (like MATLAB) still have competitive performance with compiled languages as they call the same underlying BLAS routines. Nonetheless, there are some particular scenarios, such as in dealing with tensor networks in the presence of global symmetries [62–68], where a complied language may offer a

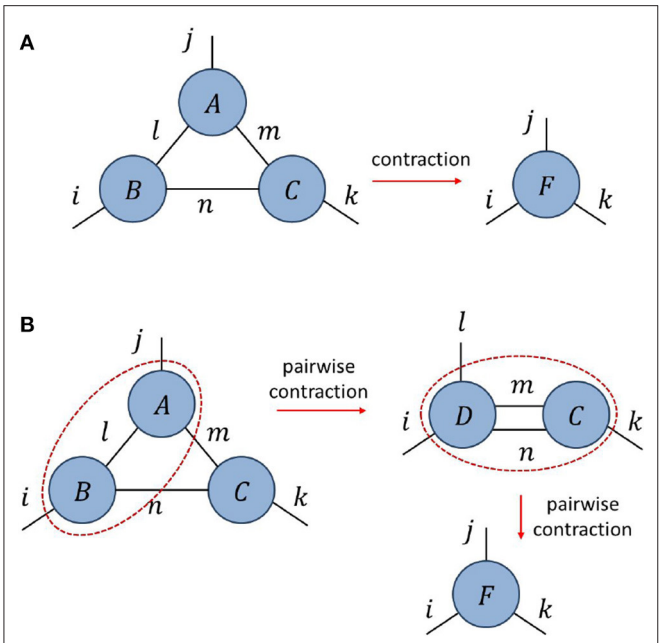


FIGURE 2 | (A) The internal indices (l, m, n) of the network $\{A, B, C\}$ are contracted to give tensor F. **(B)** The network is contracted *via* a sequence of two pairwise tensor contractions, the first of which results in the intermediate tensor D.

significant performance advantage. In this circumstance Julia, which is a compiled language, or Python, in conjunction with various frameworks which allow it to achieve some degree of compilation, may be appealing options.

2.4. Terminology

Before proceeding, let us establish the terminology that we will use when discussing tensor networks. We define the *order* of a tensor as the number of indices it has, i.e., such that a vector is order-1 and a matrix is order-2. The term *rank* (or decomposition rank) of a tensor will refer to the number of non-zero singular values with respect to a some bi-partition of the tensor indices. Note that many researchers also use the term *rank* to describe the number of tensor indices; here we use the alternative term *order* specifically to avoid the confusion that arises from the double usage of *rank*. The number of values a tensor index can take will be referred to as the dimension of an index (or bond dimension), which is most often denoted by χ but can also be denoted by m, d, or D. In most cases, the use of d or D to denote a bond dimension is less preferred, as this can be confused which the spatial dimension of the problem under consideration (e.g., when considering a model on a $1D$ or $2D$ lattice geometry).

3. TENSOR CONTRACTIONS

The foundation of all tensor networks routines is the contraction of a network containing multiple tensors into a single tensor. An example of the type problem that we consider is depicted in **Figure 2A**, where we wish to contract the network of tensors $\{A, B, C\}$ to form an order-3 tensor F, which has components defined

$$F_{ijk} = \sum_{l,m,n} A_{ljm} B_{iln} C_{nmk}. \tag{1}$$

Note that a convention for tensor index ordering is required for the figure to be unambiguously translated to an equation; here we assumed that indices progress clock-wise on each tensor starting from 6 o'clock. Perhaps the most obvious way to evaluate Equation (1) numerically would be through a direct summation over the indices (l, m, n), which could be implemented using a set of nested "FOR" loops. While this approach of summing over all internal indices of a network will produce the correct answer, there are numerous reasons why this is not the preferred approach for evaluating tensor networks. The foremost reason is that it is not the most computationally efficient approach (excluding, perhaps, certain contractions involving sparse tensors, which we will not consider here). Let us analyse the contraction cost for the example given in Equation (1), assuming all tensor indices are χ-dimensional. A single element of tensor F, which has χ^3 elements in total, is given through a sum over indices (l, m, n), which requires $O(\chi^3)$ operations. Thus the total cost of evaluating tensor F through a direct summation over all internal indices is $O(\chi^6)$.

Now, let us instead consider the evaluation of tensor F broken up into two steps, where we first compute an intermediate tensor D as depicted in **Figure 2B**,

$$D_{ijmn} = \sum_{l} A_{ljm} B_{iln}, \tag{2}$$

before performing a second contraction to give the final tensor F,

$$F_{ijk} = \sum_{n,m} D_{ijmn} C_{nm}. \tag{3}$$

Through similar logic as before, one finds that the cost of evaluating intermediate tensor D scales as $O(\chi^5)$, whilst the subsequent evaluation of F in Equation (3) is also $O(\chi^5)$. Thus breaking the network contraction down into a sequence of smaller contractions each only involving a pair of tensors (which we refer to as a *pairwise tensor contraction*) is as computationally cheap or cheaper for any non-trivial bond dimension ($\chi > 1$). This is true in general: for any network of 3 or more (dense) tensors it is always at least as efficient (and usually vastly more efficient) to break network contraction into sequence of pairwise contractions, as opposed to directly summing over all the internal indices of the network.

Two natural questions arise at this point. (i) What is optimal way to implement a single pairwise tensor contraction? (ii) Does the chosen sequence of pairwise contractions affect the total computational cost and, if so, how does one decide what sequence to use? We begin by addressing the first question.

3.1. Pairwise Tensor Contractions

Let us consider the problem of evaluating a pairwise tensor contraction, denoted $(A \times B)$, between tensors A and B that are

connected by one or more common indices. A straight-forward method to evaluate such contractions, as in the examples of Equations (2) and (3), is by using nested "FOR" loops to sum over the shared indices. The computational cost of this evaluation, in terms of the number of scalar multiplications required, can be expressed concisely as

$$\text{cost}:(A \times B) = \frac{\left|\dim(A)\right| \cdot \left|\dim(B)\right|}{\left|\dim(A \cap B)\right|}, \tag{4}$$

with $\left|\dim(A)\right|$ as the total dimension of A (i.e., the product of its index dimensions) and $\left|\dim(A \cap B)\right|$ as the total dimension of the shared indices.

Alternatively, one can recast a pairwise contraction as a matrix multiplication as follows: first reorder the free indices and contracted indices on each of A and B such that they appear sequentially (which can be achieved in MATLAB using the "permute" function) and then group the free-indices and the contracted indices each into a single index (which can be achieved in MATLAB using the "reshape" function). After these steps the contraction is evaluated using a single matrix-matrix multiplication, although the final product may also need to be reshaped back into a tensor. Recasting as a matrix multiplication does not reduce the formal computational cost from Equation (4). However, modern computers, leveraging highly optimized BLAS routines, typically perform matrix multiplications significantly more efficiently than the equivalent "FOR" loop summations. Thus, especially in the limit of tensors with large total dimension, recasting as a matrix multiplication is most often the preferred approach to evaluate pairwise tensor contractions, even though this requires some additional computational overhead from the necessity of rearranging tensor elements in memory when using "permute". Note that the "tensordot" function in the Numpy module for Python conveniently evaluates a pairwise tensor contraction using this matrix multiplication approach.

3.2. Contraction Sequence

It is straight-forward to establish that, when breaking a network contraction into a sequence of binary contractions, the choice of sequence can affect the total computational cost. As an example, we consider the product of two matrices A, B with vector C,

$$F_i = \sum_{j,k} A_{ij} B_{jk} C_k, \tag{5}$$

where all indices are assumed to be dimension χ, see also **Figure 3**. If we evaluate this expression by first performing the matrix-matrix multiplication, i.e., as $F = (A \times B) \times C$, then the leading order computational cost scales as $O(\chi^3)$ by Equation (4). Alternatively, if we evaluate the expression by first performing the matrix-vector multiplication, i.e., as $F = A \times (B \times C)$, then the leading order computational cost scales as $O(\chi^2)$. Thus it is evident that the sequence of binary contractions needs to be properly considered in order to minimize the overall computational cost.

So how does one find the optimal contraction sequence for some tensor network of interest? For the networks that arise in common algorithms (such as DMRG, MERA, PEPS, TRG and TNR) it is relatively easy, with some practice, to find the optimal sequence through manual inspection or trial-and-error. This follows as most networks one needs to evaluate contain fewer than 10 tensors and the tensor index dimensions take a only single or a few distinct values within a network, which limits the number of viable contraction sequences that need be considered. More generally, determination of optimal contraction sequences is known to be an NP-hard problem [69], such that it is very unlikely that an algorithm which scales polynomially with the number of tensors in the network will ever be found to exist. However, numerical methods based on exhaustive searches and/or heuristics can typically find optimal sequences for networks with fewer than 20 tensors in a reasonable amount of time [69–72], and larger networks are seldom encountered in practice.

Note that many tensor network optimization algorithms based on an iterative sweep, where the same network diagrams are contracted each iteration (although perhaps containing different tensors and with different bond dimensions). The usual approach in this setting is to determine the optimal sequences once, before beginning the iterative sweeps, using the initial bond dimensions and then cache the sequences for re-use in later iterations. The contraction sequences are then only recomputed the if the bond dimensions stray too far from the initial values.

3.3. Network Contraction Routines

Although certainly feasible, manually writing the code for each tensor network contraction as a sequence of pairwise contractions is not recommended. Not only is substantial programming effort required, but this also results in code which is error-prone and difficult to check. There is also a more fundamental problem: contracting a network by manually writing a sequence of binary contractions requires specifying a particular contraction sequence at the time of coding. However, in many cases the index dimensions within networks are variable, and the optimal sequence can change depending on the relative sizes of dimensions. For instance, one may have a network which contains indices of dimensions χ_1 and χ_2, where the optimal contraction sequence changes dependant of whether χ_1 is larger or smaller than χ_2. In order to have a program which works efficiently in both regimes, one would have to write code separately for both contraction sequences.

Given the considerations above, the use of an automated contraction routine, such as the "ncon" (Network-CONtractor) function [61, 73] or something similar from an existing tensor network library [41–48], is highly recommended. Automated contraction routines can evaluate any network contraction in a single call by appropriately generating and evaluating a sequence of binary contractions, hence greatly reducing both the programming effort required and the risk of programming errors occurring. Most contraction routines, such as "ncon", also remove the need to fix a contraction sequence at the time of writing the code, as the sequence can be specified as an input variable to the routine and thus can be changed without the

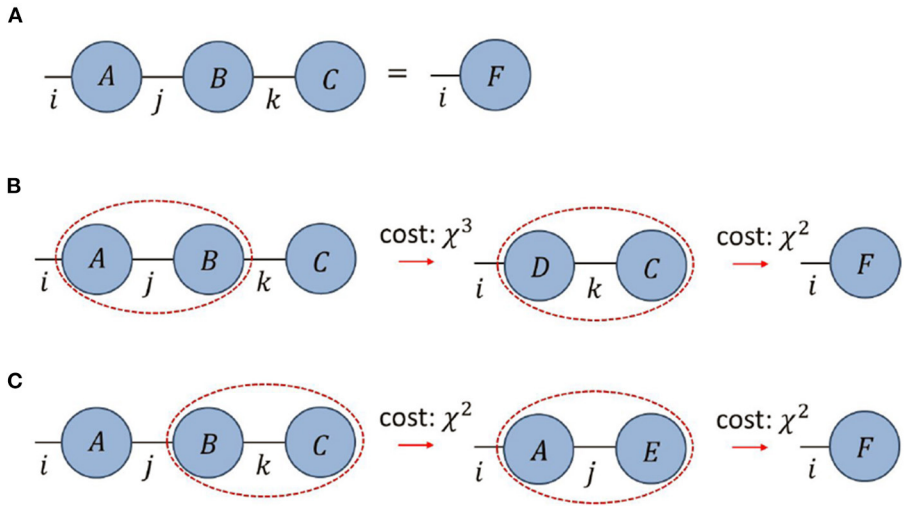


FIGURE 3 | (A) A product of three tensors $\{A, B, C\}$ is contracted to a tensor F, where all indices are assumed to be of dimension χ. **(B,C)** The total computational cost of contracting the network depends on the sequence of pairwise contractions; the cost from following the sequence in **(B)** scales as $(\chi^3 + \chi^2)$ as compared to the cost from **(C)** which scales as $(2\chi^2)$.

need to rewrite any code. This can also allow the contraction sequence to be adjusted dynamically at run-time to ensure that the sequence is optimal given the specific index dimensions in use.

3.4. Summary: Contractions

In evaluating a network of multiple tensors, it is always more efficient to break the contraction into a sequence of pairwise tensor contractions, each of which should (usually) be recast into a matrix-matrix multiplication in order to achieve optimal computational performance. The total cost of evaluating a network can depend on the particular sequence of pairwise contractions chosen. While there is no known method for determining an optimal contraction sequence that is efficiently scalable in the size of the network, manual inference or brute-force numeric searches are usually viable for the relatively small networks encountered in common tensor network algorithms. When coding a tensor network program it is useful to utilize an automated network contraction routine which can evaluate a tensor network in a single call by properly chaining together a sequence of pairwise contractions. This not only reduces the programming effort required, but also grants a program more flexibility in allowing a contraction sequence to be easily changed.

4. MATRIX FACTORIZATIONS

Another key operation common in tensor network algorithms, complimentary to the tensor contractions considered previously, are factorizations. In this section, we will discuss some of the various means by which a higher-order tensor can be split into a product of fewer-order tensors. In particular, the means that we consider involve applying standard matrix decompositions [74, 75], to tensor unfoldings, such that this section may serve as a review of the linear algebra necessary before consideration of more sophisticated network decompositions. Specifically we recount the spectral decomposition, QR decomposition and singular value decomposition and outline their usefulness in the context of tensor networks, particular in achieving optimal low-rank tensor approximations. Before discussing decompositions, we define some special types of tensor.

4.1. Special Tensor Types

A d-by-d matrix U is said to be unitary if it has orthonormal rows and columns, which implies that it annihilates to the identity when multiplied with its conjugate,

$$U^\dagger U = UU^\dagger = I, \tag{6}$$

where I is the d-by-d identity matrix. We define a tensor (whose order is greater than 2) as unitary with respect to a particular bi-partition of indices if the tensor can be *reshaped* into a unitary matrix according to this partition. Similarly an d_1-by-d_2 matrix W, with $d_1 > d_2$ is said to be an isometry if

$$W^\dagger W = I, \tag{7}$$

with I the d_2-by-d_2 identity matrix. Likewise we say that a tensor (order greater than 2) is isometric with respect to a particular bi-partition of indices if the tensor can be reshaped into a isometric matrix. Notice that, rather than equalling identity, the reverse order product does now evaluate to a projector P,

$$WW^\dagger = P, \tag{8}$$

where projectors are defined as Hermitian matrices that square to themselves,

$$P = P^\dagger, \quad P^2 = P. \tag{9}$$

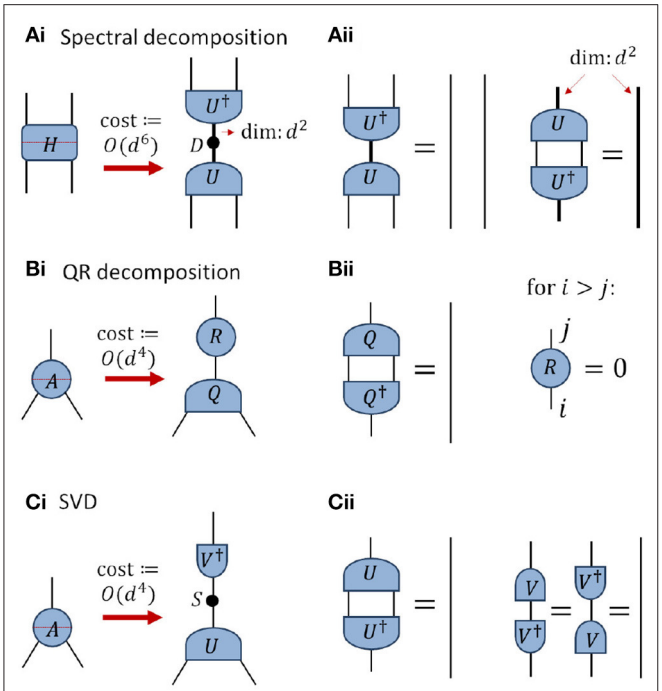


FIGURE 4 | Depiction of some common matrix decompositions. All indices are assumed to be of dimension d unless otherwise indicated. **(Ai)** The spectral decomposition is applied to the order-4 Hermitian tensor H across the partition indicated by the dashed line, yielding a diagonal matrix of eigenvalues D and a unitary U. **(Aii)** The unitary tensor U annihilates to identity with its conjugate, as per Equation (6). **(Bi)** The QR decomposition is applied to the order-3 tensor A across the partition indicated, yielding an isometry Q and an upper triangular matrix R. **(Bii)** The isometry Q annihilates to identity with its conjugate as per Equation (7), while the R matrix is upper triangular. **(Ci)** The singular value decomposition (SVD) is applied to the order-3 tensor A across the partition indicated, yielding an isometry U, a diagonal matrix of singular values S, and a unitary V. **(Cii)** Depiction of the constraints satisfied by the isometry U and unitary V.

4.2. Useful Matrix Decompositions

A commonly used matrix decomposition is the spectral decomposition (or eigen-decomposition). In the context of tensor network codes it is most often used for Hermitian, positive semi-definite matrices, such as for the density matrices used to describe quantum states. If H is a $d \times d$ Hermitian matrix, or tensor that can be reshaped into such, then the spectral decomposition yields

$$H = UDU^\dagger, \tag{10}$$

where U is $d \times d$ unitary matrix and D is diagonal matrix of eigenvalues, see also **Figure 4A**. The numerical cost of performing the decomposition scales as $O(d^3)$. In the context of tensor network algorithms the spectral decomposition is often applied to approximate a Hermitian tensor with one of smaller rank, as will be discussed in Section 4.4.

Another useful decomposition is the QR decomposition. If A be an arbitrary $d_1 \times d_2$ matrix with $d_1 > d_2$, or tensor that can be reshaped into such, then the QR decomposition gives

$$A = QR, \tag{11}$$

see also **Figure 4B**. Here Q is $d_1 \times d_2$ isometry, such that $Q^\dagger Q = I$, where I is the $d_2 \times d_2$ identity matrix, and R is $d_2 \times d_2$ upper triangular matrix. Note that we are considering the so-called economical decomposition (which is most often used in tensor network algorithms); otherwise the full decomposition gives Q as a $d_1 \times d_1$ unitary and R is dimension $d_1 \times d_2$. The numerical cost of the economical QR decomposition scales as the larger dimension times the square of the smaller dimension $O(d_1 {d_2}^2)$, as opposed to cost $O({d_1}^2 d_2)$ for the full decomposition. The QR decomposition is one of the most computationally efficient ways to obtain an orthonormal basis for the column space of a matrix, thus a common application is in orthogonalizing tensors within a network (i.e., transforming them into isometries), which will be discussed further in Section 5.3.1.

The final decomposition that we consider is the singular value decomposition (SVD), which is also widely used in many areas of mathematics, statistics, physics and engineering. The SVD allows an arbitrary $d_1 \times d_2$ matrix A, where we assume for simplicity that $d_1 \geq d_2$, to be decomposed as

$$A = USV^\dagger \tag{12}$$

where U is $d_1 \times d_2$ isometry (or unitary if $d_1 = d_2$), S is diagonal $d_2 \times d_2$ matrix of positive elements (called singular values), and V is $d_2 \times d_2$ unitary matrix, see also **Figure 4C**. Similar to the economical QR decomposition, we have also considered the economical form of the SVD; the full SVD would otherwise produce U as a $d_1 \times d_1$ unitary and S as a rectangular $d_1 \times d_2$ matrix padded with zeros. The numerical cost of the economical SVD scales as $O(d_1 {d_2}^2)$, identical to that of the economical QR decomposition. The rank of a tensor (across a specified bi-partition) is defined as the number of non-zero singular values that appear in the SVD. A common application of the SVD is in finding an approximation to a tensor by another of smaller rank, which will be discussed further in Section 4.4.

Notice that for any matrix A the spectral decompositions of $AA^\dagger$ and $A^\dagger A$ are related to the SVD of A; more precisely, the eigenvectors of $AA^\dagger$ and $A^\dagger A$ are equivalent to the singular vectors in U and V respectively of the SVD in Equation (12). Furthermore the (non-zero) eigenvalues in $AA^\dagger$ or $A^\dagger A$ are the squares of the singular values in S. It can also be seen that, for a Hermitian positive semi-definite matrix H, the spectral decomposition is equivalent to an SVD.

4.3. Tensor Norms

The primary use for matrix decompositions, such as the SVD, in the context of tensor networks is in accurately approximating a higher-order tensor as a product lower-order tensors. However, before discussing tensor approximations, it is necessary to define the tensor norm in use. A tensor norm that is particularly convenient is the Frobenius norm (or Hilbert-Schmidt norm). Given a tensor $A_{ijk\ldots}$ the Frobenius norm for A, denoted as $\|A\|$,

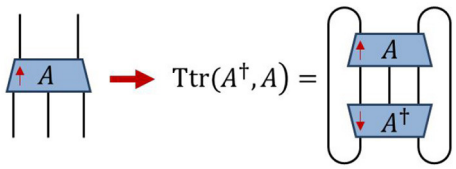


FIGURE 5 | For any tensor A the tensor trace Ttr of A with its conjugate $A^\dagger$ (drawn with opposite vertical orientation) is obtained by contracting over all matching indices. The Frobenius norm can be defined as the root of this tensor trace, see Equation (14).

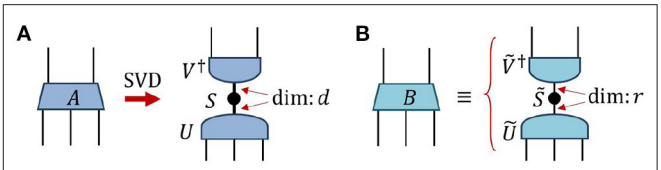


FIGURE 6 | (A) The singular value decomposition is taken on tensor A across a bi-partition between its top two and bottom three indices, and is assumed to produce d non-zero singular values. **(B)** Tensor B is now defined by truncating the matrix of singular values $S \to \tilde{S}$ to retain only $r < d$ of the largest singular values, while similarly truncating the matrices of singular vectors, $U \to \tilde{U}$ and $V \to \tilde{V}$, to retain only the corresponding singular vectors. By the Eckart–Young–Mirsky theorem [76] it is known that B is the optimal rank-r approximation to A (across the chosen bi-partition of tensor indices).

is defined as the square-root of the sum of the magnitude of each element squared,

$$\|A\| = \sqrt{\sum_{ijk\ldots} |A_{ijk\ldots}|^2}. \tag{13}$$

This can be equivalently expressed as the tensor trace of A multiplied by its conjugate,

$$\|A\| = \sqrt{\mathrm{Ttr}\left(A^\dagger, A\right)}, \tag{14}$$

where the tensor trace, $\mathrm{Ttr}(A^\dagger, A)$, represents the contraction of tensor A with its conjugate over all matching indices, see **Figure 5**. It also follows that Frobenius norm is related to the singular values s_k of A across any chosen bi-partition,

$$\|A\| = \sqrt{\sum_k (s_k)^2}. \tag{15}$$

Notice that Equation (14) implies that the difference $\varepsilon = \|A - B\|$ between two tensors A and B of equal dimension can equivalently be expressed as

$$\|A - B\|^2 = \mathrm{Ttr}(A^\dagger, A) - 2\left|\mathrm{Ttr}(A^\dagger, B)\right| + \mathrm{Ttr}(B^\dagger, B). \tag{16}$$

4.4. Optimal Low-Rank Approximations

Given some matrix A, or higher-order tensor that viewed as a matrix across a chosen bi-partition of its indices, we now focus on the problem of finding the tensor B that best approximates A according to the Frobenius norm (i.e., that which minimizes the difference in Equation 16), assuming B has a fixed rank r. Let us assume, without loss of generality, that tensor A is equivalent to a $d_1 \times d_2$ matrix (with $d_1 \geq d_2$) under a specified bi-partition of its indices, and that A has singular value decomposition,

$$A_{ij} = \sum_{k=1}^{d_2} U_{ik} s_k V^*_{kj}, \tag{17}$$

where the singular values are assumed to be in descending order, $s_k \geq s_{k+1}$. Then the optimal rank r tensor B that approximates A is known from the Eckart–Young–Mirsky theorem [76], which states that B is given by truncating to retain only the r largest singular values and their corresponding singular vectors,

$$B_{ij} = \sum_{k=1}^{r} U_{ik} s_k V^*_{kj}. \tag{18}$$

see also **Figure 6**. It follows that the error of approximation $\varepsilon = \|A - B\|$, as measured in the Frobenius norm, is related to the discarded singular values as

$$\varepsilon = \sqrt{\sum_{k>r} (s_k)^2}. \tag{19}$$

If the spectrum of singular values is sharply decaying then the error is well-approximated by the largest of the discarded singular values, $\varepsilon \approx s_{(r+1)}$.

Notice that, in the case that the tensor A under consideration is Hermitian and positive definite across the chosen bi-partition, that the spectral decomposition could instead be used in Equation (17). The low-rank approximation obtained by truncating the smallest eigenvalues would still be guaranteed optimal, as the spectral decomposition is equivalent to the SVD in this case.

4.5. Summary: Decompositions

In this section we have described how special types of matrices, such as unitary matrices and projections, can be generalized to the case of tensors (which can always be viewed as a matrix across a chosen bi-partition of their indices). Similarly we have shown how several concepts from matrices, such as the trace and the norm, are also generalized to tensors. Finally, we have described how the optimal low-rank approximation to a tensor can be obtained *via* the SVD.

5. GAUGE FREEDOM

All tensor networks possess gauge degrees of freedom; exploiting the gauge freedom allows one to change the tensors within a network whilst leaving the final product of the network unchanged. In this section, we describe methods

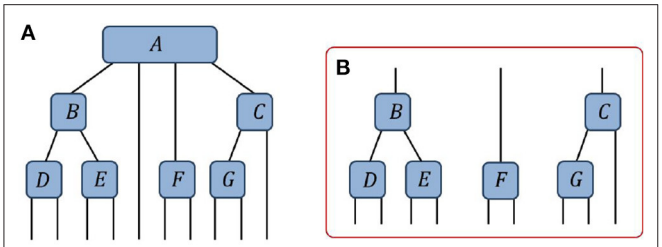


FIGURE 7 | (A) An example of a tree tensor network (TTN), here composed of 7 tensors. **(B)** The three (non-trivial) branches of the tree with respect to choosing A as the root tensor.

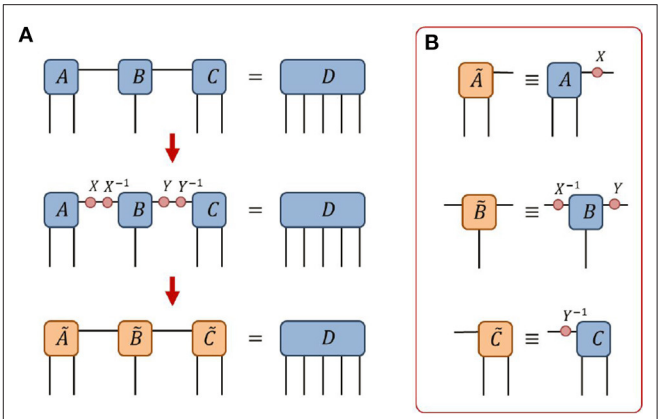


FIGURE 8 | (A) Given a network of three tensors $\{A, B, C\}$, one can introduce gauge change matrices X and Y (together with their inverses) on the internal indices of the network while leaving the final product D of the network unchanged. **(B)** Definitions of the new tensors $\{\tilde{A}, \tilde{B}, \tilde{C}\}$ after the change of gauge.

for manipulating the gauge degrees of freedom and discuss the utility of fixing the gauge in a specific manner.

5.1. Tree Tensor Networks

In this manuscript we shall restrict our considerations of gauge manipulations to focus only on tensors networks that do not possess closed loops (i.e., networks that correspond to acyclic graphs), which are commonly referred to as tree tensor networks (TTN) [77, 78]. **Figure 7A** presents an example of a tree tensor network. If we select a single tensor to act as the center (or root node) then we can understand the tree tensor network as being composed of a set of distinct branches extending from this chosen tensor. For instance, **Figure 7B** depicts the three branches (excluding the single trivial branch) extending from the 4th-order tensor A. It is important to note that connections between the different branches are not possible in networks without closed loops; this restriction is required for the re-gauging methods considered in this manuscript. However these methods can (mostly) be generalized to the case of networks containing closed loops by using a more sophisticated formalism as shown in [79].

5.2. Gauge Transformations

Consider a tensor network of multiple tensors that, under contraction of all internal indices, evaluates to some product tensor H. We now pose the following question: is there a different choice of tensors with the same network geometry that also evaluates to H? Clearly the answer to this question is yes! As shown below in **Figure 8A**, on any of the internal indices of the network one can introduce a resolution of the identity (i.e., a pair of matrices X and X^{-1}) which, by construction, does not change the final product that the network evaluates to. However, absorbing one of these matrices into each adjoining tensor changes the individual tensors, see **Figure 8B**, while leaving the product of the network unchanged. It follows that there are infinitely many choices of tensors such that the network product evaluates to some fixed output tensor, since the gauge change matrix X can be any invertible matrix. This ability to introduce an arbitrary resolution of the identity on an internal index, while leaving the product of the network unchanged, is referred to as the gauge freedom of the network.

While in some respects the gauge freedom could be considered bothersome, as it implies tensor decompositions are never unique, it can also be exploited to simplify many types of operations on tensor networks. Indeed, most tensor network algorithms require fixing the gauge in a prescribed manner in order to function correctly. In the following sections we discuss ways to fix the gauge degree of freedom as to create an *orthogonality center* and the benefits of doing so.

5.3. Orthogonality Centers

A given tensor A within a network is said to be an *orthogonality center* if every branch connected to tensor A annihilates to the identity when contracted with its conjugate as shown in **Figure 9**. Equivalently, each branch must (collectively) form an isometry across the bi-partition between its open indices and the index connected to tensor A. By properly manipulating the gauge degrees of freedom, it is possible to turn any tensor with a tree tensor network into an orthogonality center [80]. We now discuss two different methods for achieving this: a "pulling through" approach, which was a key ingredient in the original formulation of DMRG [49–51], and a "direct orthogonalization" approach, which was an important part of the TEBD algorithm [52, 53].

5.3.1. Creating an Orthogonality Center *via* "Pulling Through"

Here we describe a method for setting a tensor A within a network as an orthogonality center through iterative use of the QR decomposition. The idea behind his method is very simple: if each individual tensor within a branch is transformed into a (properly oriented) isometry, then the entire branch collectively becomes an isometry and thus the tensor at center of the branches becomes an orthogonality center. Let us begin by orienting each index of the network by drawing an arrow pointing toward the desired center A. Then, starting at the tip of each branch, we should perform a QR decomposition on each tensor based on a bi-partition between its incoming and outgoing indices. The R part of the decomposition should then be absorbed into the next tensor in the branch (i.e., closer to the root tensor A), and the

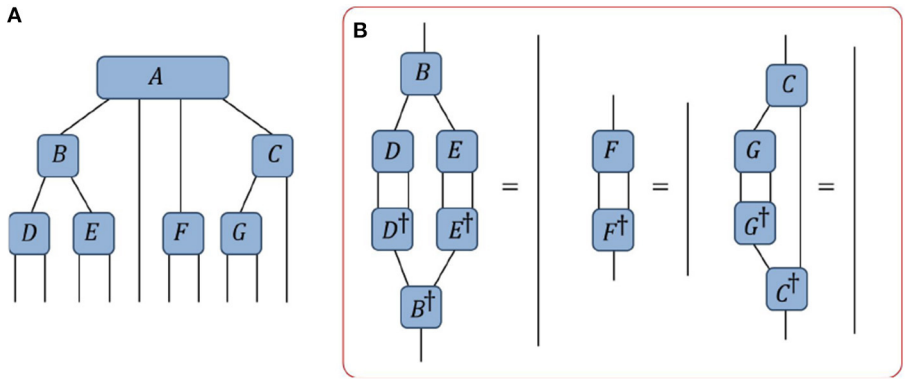


FIGURE 9 | (A) An example of a tree tensor network. **(B)** A depiction of the constraints required for the tensor A to be an *orthogonality center*: each of the branches must annihilate to the identity when contracted with its conjugate.

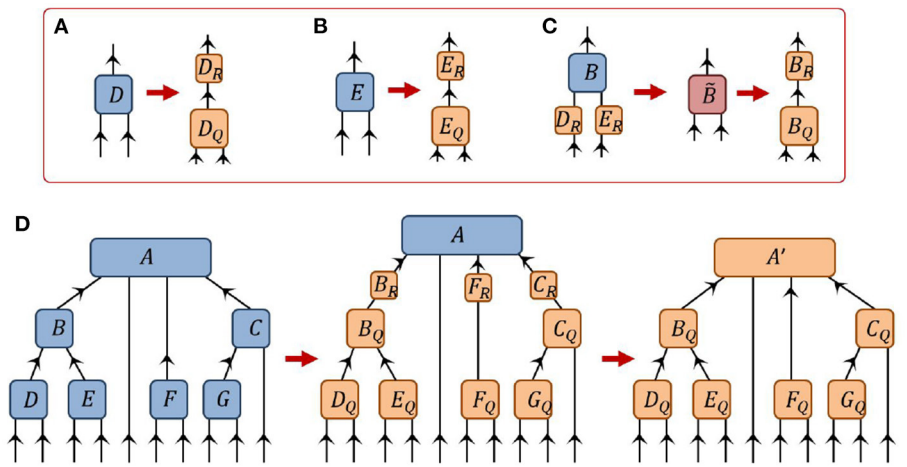


FIGURE 10 | A depiction of how the tensor A can be made into an orthogonality center of the network from **Figure 7** *via* the "pulling-though" approach. **(A,B)** Tensors D and E, which reside at the tips of a branch, are decomposed *via* the QR decomposition. **(C)** The R components of the previous QR decompositions are absorbed into the B tensor higher on the branch, which is then itself decomposed *via* the QR decomposition. **(D)** Following this procedure, all tensors in the network are orthogonalized (with respect to their incoming vs. outgoing indices) such that A' becomes an orthogonality center of the network.

process repeated as depicted in **Figures 10A–C**. At the final step an R part of the QR decomposition from each branch is absorbed into the central tensor A and the process is complete, see also **Figure 10D**.

Note that the SVD could be used as an alternative to the QR decomposition: instead of absorbing the R part of the QR decomposition into the next tensor in the branch one could absorb the product of the S and V part of the SVD from Equation (12). However, in practice, the QR decomposition is most often preferable as it computationally cheaper than the SVD.

5.3.2. Creating an Orthogonality Center *via* "Direct Orthogonalization"

Here we describe a method for setting a tensor A within a network as an orthogonality center based on use of a single decomposition for each branch, as depicted in **Figure 11**. (i) We begin by computing the positive-definite matrix ρ for each branch (with respect to the center tensor A) by contracting the branch with its Hermitian conjugates. (ii) The principle square root X of each of the matrices ρ is then computed, i.e., such that $\rho = XX^\dagger$, which can be computed using the spectral decomposition if necessary. (iii) Finally, a change of gauge is made on each of the indices of tensor A using the appropriate X matrix and its corresponding inverse X^{-1}, with the X part absorbed in tensor A and the X^{-1} absorbed in the leading tensor of the branch such that the branch matrix transforms as $\rho \to \tilde{\rho} = X^{-1}\rho(X^{-1})^\dagger$. It follows that the tensor A is now an orthogonality center as each branch matrix $\tilde{\rho}$ of the transformed network evaluates as the identity,

$$\tilde{\rho} = X^{-1}\rho(X^{-1})^\dagger = X^{-1}XX^\dagger(X^{-1})^\dagger = I. \tag{20}$$

Note that, for simplicity, we have assumed that the branch matrices ρ do not have zero eigenvalues, such that their inverses

exist. Otherwise, if zero eigenvalues are present, the current method is not valid unless the index dimensions are first reduced by truncating any zero eigenvalues.

5.3.3. Comparison of Methods for Creating Orthogonality Centers

Each of the two methods discussed to create an orthogonality center have their own advantages and disadvantages, such that the preferred method may depend on the specific application under consideration. In practice, the "direct orthogonalization" is typically computationally cheaper and easier to execute, since this method only requires changing the gauge on the indices connected to the center tensor. In contrast the "pulling through" method involves changing the gauge on all indices of the network. Additionally, the "direct orthogonalization" approach can easily be employed in networks of infinite extent, such as infinite MPS [52, 53], if the matrix ρ associated to a branch of infinite extent can be computed using by solving for a dominant eigenvector. While "pulling through" can also potentially be employed for networks of infinite extent, i.e., through successive decompositions until sufficient convergence is achieved, this is likely to be more computationally expensive. However the "pulling through" approach can be advantageous if the branch matrices ρ are ill-conditioned as the errors due to floating-point arithmetic are lesser. This follows since the "direct orthogonalization" requires one to square the tensors in each branch. The "pulling-through" approach also results in transforming every tensor in the network (with the exception of the center tensor) into an isometry, which may be desirable in certain applications.

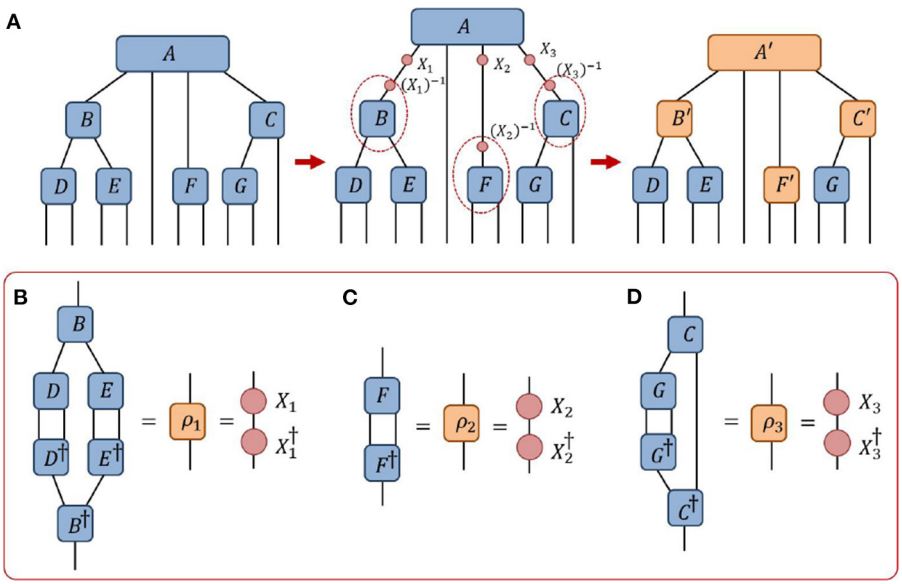


FIGURE 11 | A depiction of how the tensor A from the network of **Figure 7** can be made into an orthogonality center *via* the "direct orthogonalization" approach. **(A)** A change of gauge is made on every (non-trivial) branch connected to A, such that A becomes an orthogonality center. **(B–D)** The gauge change matrices $\{X_1, X_2, X_3\}$ are obtained by contracting each branch with its Hermitian conjugate and then taking the principle root.

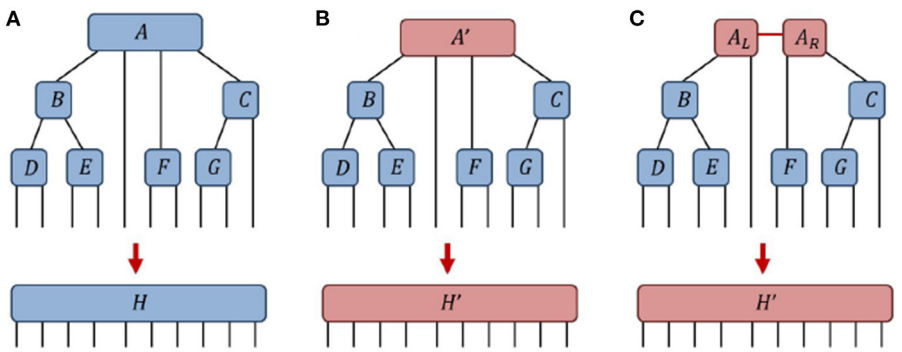


FIGURE 12 | (A) A network of 7 tensors $\{A, B, C, D, E, F, G\}$ contracts to give a tensor H. **(B)** After replacing a single tensor $A \rightarrow A'$ the network contracts to a different tensor H'. **(C)** The tensor A' is decomposed into a pair of tensors A_L and A_R, leaving the final tensor H' unchanged.

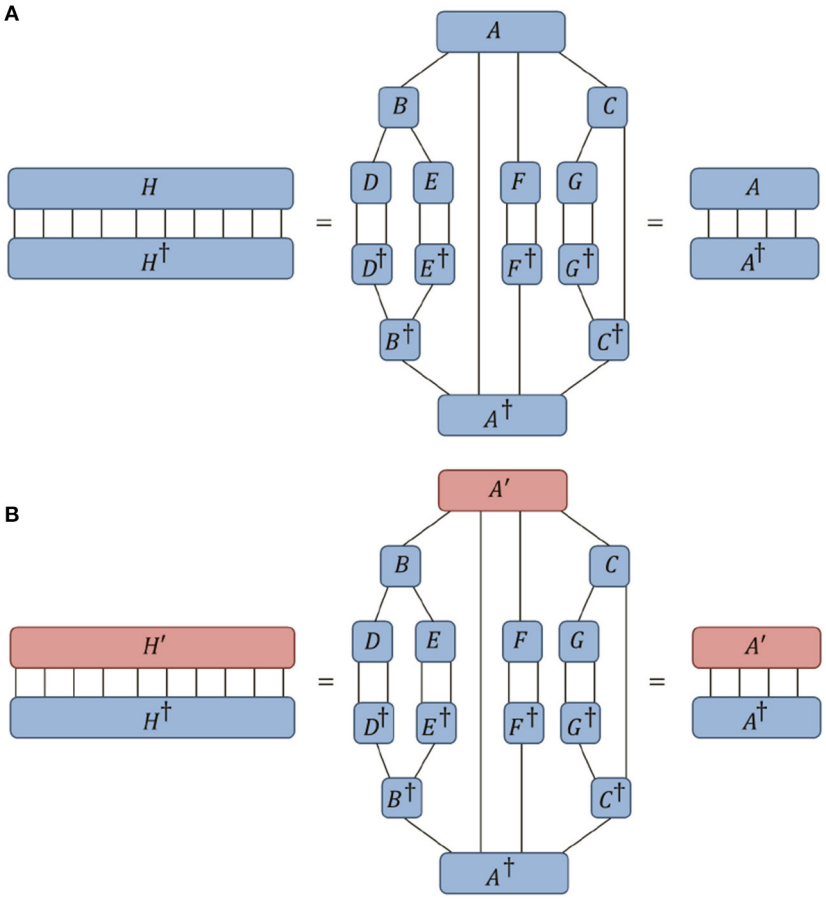


FIGURE 13 | (A) In the network from **Figure 12A**, if the tensor A is an orthogonality center then it follows that the norm of the network $\|H\|$ is equal to the norm of the center tensor $\|A\|$. **(B)** Similarly it follows that, in changing only the center tensor $A \to A'$, the global overlap between the networks H and H' is equal to the local overlap between the center tensors A and A'.

5.4. Decompositions of Tensors Within Networks

In Section 4, it was described how the SVD could be applied to find the optimal low-rank approximation to a tensor in terms of minimizing the Frobenius norm between the original tensor and the approximation. In the present section we extend this concept further and detail how, by first creating an orthogonality center, a tensor within a network can be optimally decomposed as to minimize the *global* error coming from consideration of the entire network.

Let us consider a tree tensor network of tensors $\{A, B, C, D, E, F, G\}$ that evaluates to a tensor H, as depicted in **Figure 12A**. We now replace a single tensor A from this network by a new tensor A' such that the network now evaluates to a tensor H' as depicted in **Figure 12B**. Our goal is to address the following question: how can we find the optimal low-rank approximation A' to tensor A such that the error from the full network, $\|H - H'\|$, is minimized? Notice that if we follow the method from Section 4 and simple truncate the smallest singular values of A, see **Figure 12C**, then this will only ensure that the local error, $\|A - A'\|$, is minimized. The key to resolving this issue is through creation of an orthogonality center, which can reduce the global norm of a network to the norm of a single tensor. Specifically if tensor A is an orthogonality center of a network that evaluates to a final tensor H then it follow from the definition of an orthogonality center that $\|H\| = \|A\|$, as depicted in **Figure 13A**. Thus it also can be seen that under replacement of the center tensor A with a new tensor A', such that the network now evaluates to a new tensor H', that the difference between the tensors $\|A - A'\|$ is precisely equal to the global difference between the networks $\|H - H'\|$. This follows as the overlap of H and H' equals the overlap of A and A', as depicted in **Figure 13B**. In other words, by appropriately manipulating the gauge degrees of freedom in a network, the *global* difference resulting from changing a single tensor in a network can become equivalent to the *local* difference between the single tensors. The solution to the problem of finding the optimal low-rank approximation A' to a tensor A within a network thus becomes clear; we should first adjust the gauge such that A becomes an orthogonality center, after which we can

follow the method from Section 4 and create the optimal global approximation (i.e., that which minimizes the global error) by truncating the smallest singular values of A. The importance of this result in the context tensor network algorithms cannot be overstated; this understanding for how to optimally truncate a single tensor within a tensor network, see also [81], is a key aspect of the DMRG algorithm [49–51], the TEBD algorithm [52, 53] and many other tensor network algorithms.

5.5. Summary: Gauge Freedom

In the preceding section, we discussed manipulations of the gauge degrees of freedom in a tensor network and described two methods that can be used to create an orthogonality center. The proper use of an orthogonality center was then demonstrated to allow one to decompose a tensor within a network in such a way as to minimize the global error. Note that while the results in this section were described only for tree tensor networks (i.e., networks based on acyclic graphs), they can be generalized to arbitrary networks by using more sophisticated methodology [79].

6. CONCLUSIONS

Network contractions and decompositions are the twin pillars of all tensor network algorithms. In this manuscript we have recounted the key theoretical considerations required for performing these operations efficiently and also discussed aspects of their implementation in numeric codes. We expect that a proper understanding of these results could facilitate an individuals effort to implement many common tensor network algorithms, such as DMRG, TEBD, TRG, PEPS and MERA, and also further aid researchers in the design and development of new tensor network algorithms.

However, there are still a wide variety of additional general ideas and methods, not covered in this manuscript, that are necessary for the implementation of more advanced tensor network algorithms. These include (i) strategies for performing variational optimization, (ii) methods for dealing with decompositions in networks containing closed loops, (iii) the use of approximations in tensor network contractions. We shall address several of these topics in a follow-up work.

DATA AVAILABILITY STATEMENT

The original contributions presented in the study are included in the article/supplementary material, further inquiries can be directed to the corresponding author/s.

AUTHOR CONTRIBUTIONS

GE was solely responsible for the preparation of this manuscript.

FUNDING

This work was funded by Institutional Startup Funds.

REFERENCES

1. Cirac JI, Verstraete F. Renormalization and tensor product states in spin chains and lattices. *J Phys A Math Theor*. (2009) 42:504004. doi: 10.1088/1751-8113/42/50/504004
2. Evenbly G, Vidal G. Tensor network states and geometry. *J Stat Phys*. (2011) 145:891–918. doi: 10.1007/s10955-011-0237-4
3. Orus R. A practical introduction to tensor networks: matrix product states and projected entangled pair states. *Ann Phys*. (2014) 349:117. doi: 10.1016/j.aop.2014.06.013
4. Bridgeman JC, Chubb CT. Hand-waving and interpretive dance: an introductory course on tensor networks. *J Phys A Math Theor*. (2017) 50:223001. doi: 10.1088/1751-8121/aa6dc3
5. Montangero S. Introduction to tensor network methods. In: *Numerical Simulations of Low-Dimensional Many-body Quantum Systems*. Berlin: Springer (2018). doi: 10.1007/978-3-030-01409-4
6. Orus R. Tensor networks for complex quantum systems. *Nat Rev Phys*. (2019) 1:538–50. doi: 10.1038/s42254-019-0086-7
7. Silvi P, Tschirsich F, Gerster M, Junemann J, Jaschke D, Rizzi M, et al. The tensor networks anthology: simulation techniques for many-body quantum lattice systems. *SciPost Phys*. (2019) 8. doi: 10.21468/SciPostPhysLectNotes.8
8. Ran S-J, Tirrito E, Peng C, Chen X, Tagliacozzo L, Su G, et al. *Tensor Network Contractions Methods and Applications to Quantum Many-Body Systems*. Springer (2020). doi: 10.1007/978-3-030-34489-4
9. Cirac JI, Perez-Garcia D, Schuch N, Verstraete F. Matrix product states and projected entangled pair states: concepts, symmetries, theorems. *Rev Mod Phys*. (2021) 93:045003. doi: 10.1103/RevModPhys.93.045003
10. Kolda TG, Bader BW. Tensor decompositions and applications. *SIAM Rev*. (2009) 51:455–500. doi: 10.1137/07070111X
11. Vidal G, Latorre JI, Rico E, Kitaev A. Entanglement in quantum critical phenomena. *Phys Rev Lett*. (2003) 90:227902. doi: 10.1103/PhysRevLett.90.227902
12. Hastings MB. An area law for one-dimensional quantum systems. *J Stat Mech*. (2007) 2007:P08024. doi: 10.1088/1742-5468/2007/08/P08024
13. Eisert J, Cramer M, Plenio M. Area laws for the entanglement entropy - a review. *Rev Mod Phys*. (2010) 82:277–306. doi: 10.1103/RevModPhys.82.277
14. Chan GKL, Sharma S. The density matrix renormalization group in quantum chemistry. *Annu Rev Phys Chem*. (2011) 62:465. doi: 10.1146/annurev-physchem-032210-103338
15. Keller S, Dolfi M, Troyer M, Reiher M. An efficient matrix product operator representation of the quantum chemical Hamiltonian. *J Chem Phys*. (2015) 143:244118. doi: 10.1063/1.4939000
16. Szalay S, Pfeffer M, Murg V, Barcza G, Verstraete F, Schneider R, et al. Tensor product methods entanglement optimization for *ab initio*. quantum chemistry. *Int J Quant Chem*. (2015) 115:1342. doi: 10.1002/qua.24898
17. Chan G K-L, Keselman A, Nakatani N, Li Z, White SR. Matrix product operators, matrix product states, and *ab initio*. density matrix renormalization group algorithms. *J Chem Phys*. (2016) 145:014102. doi: 10.1063/1.4955108
18. Zhai H, Chan GK-L. Low communication high performance *ab initio*. density matrix renormalization group algorithms. *J Chem Phys*. (2021) 154:224116. doi: 10.1063/5.0050902
19. Swingle B. Entanglement renormalization and holography. *Phys Rev D*. (2012) 86:065007. doi: 10.1103/PhysRevD.86.065007
20. Miyaji M, Numasawa T, Shiba N, Takayanagi T, Watanabe K. Continuous multiscale entanglement renormalization ansatz as holographic surface-state correspondence. *Phys Rev Lett*. (2015) 115:171602. doi: 10.1103/PhysRevLett.115.171602
21. Pastawski F, Yoshida B, Harlow D, Preskill J. Holographic quantum error-correcting codes: toy models for the bulk/boundary correspondence. *J High Energy Phys*. (2015) 6:149. doi: 10.1007/JHEP06(2015)149

22. Hayden P, Nezami S, Qi X-L, Thomas N, Walter M, Yang Z. Holographic duality from random tensor networks. *J High Energy Phys*. (2016) 11:009. doi: 10.1007/JHEP11(2016)009
23. Czech B, Lamprou L, McCandlish S, Sully J. Tensor networks from kinematic space. *J High Energy Phys*. (2016) 07:100. doi: 10.1007/JHEP07(2016)100
24. Evenbly G. Hyperinvariant tensor networks and holography. *Phys Rev Lett*. (2017) 119:141602. doi: 10.1103/PhysRevLett.119.141602
25. Stoudenmire EM, Schwab DJ. Supervised learning with tensor networks. *Adv Neural Inf Process Syst*. (2016) 29:4799–807.
26. Martyn J, Vidal G, Roberts C, Leichenauer S. Entanglement and tensor networks for supervised image classification. *arXiv preprint arXiv:2007.06082*. (2020). doi: 10.48550/arXiv.2007.06082
27. Cheng S, Wang L, Zhang P. Supervised learning with projected entangled pair states. *Phys Rev B*. (2021) 103:125117. doi: 10.1103/PhysRevB.103.125117
28. Liu J, Li S, Zhang J, Zhang P. Tensor networks for unsupervised machine learning. *arXiv preprint arXiv:2106.12974*. (2021). doi: 10.48550/arXiv.2106.12974
29. Liu Y, Li W-J, Zhang X, Lewenstein M, Su G, Ran SJ. Entanglement-based feature extraction by tensor network machine learning. *Front Appl Math Stat*. (2021) 7:716044. doi: 10.3389/fams.2021.716044
30. Fried ES, Sawaya NPD, Cao Y, Kivlichan ID, Romero J, Aspuru-Guzik A. qTorch: the quantum tensor contraction handler. *PLoS ONE*. (2018) 13:e0208510. doi: 10.1371/journal.pone.0208510
31. Villalonga B, Boixo S, Nelson B, Henze C, Rieffel E, Biswas R, et al. A flexible high-performance simulator for verifying and benchmarking quantum circuits implemented on real hardware. *NPJ Quantum Inf*. (2019) 5:86. doi: 10.1038/s41534-019-0196-1
32. Schutski R, Lykov D, Oseledets I. Adaptive algorithm for quantum circuit simulation. *Phys Rev A*. (2020) 101:042335. doi: 10.1103/PhysRevA.101.042335
33. Pan F, Zhang P. Simulation of quantum circuits using the big-batch tensor network method. *Phys Rev Lett*. (2022) 128:030501. doi: 10.1103/PhysRevLett.128.030501
34. Levental M. Tensor networks for simulating quantum circuits on FPGAs. *arXiv preprint arXiv preprint arXiv:2108.06831*. (2021). doi: 10.48550/arXiv.2108.06831
35. Vincent T, O'Riordan LJ, Andrenkov M, Brown J, Killoran N, Qi H, et al. Jet: fast quantum circuit simulations with parallel task-based tensor-network contraction. *Quantum*. (2022) 6:709. doi: 10.22331/q-2022-05-09-709
36. Evenbly G, Vidal G. Algorithms for entanglement renormalization. *Phys Rev B*. (2009) 79:144108. doi: 10.1103/PhysRevB.79.144108
37. Zhao H-H, Xie Z-Y, Chen Q-N, Wei Z-C, Cai JW, Xiang T. Renormalization of tensor-network states. *Phys Rev B*. (2010) 81:174411. doi: 10.1103/PhysRevB.81.174411
38. Schollwoeck U. The density-matrix renormalization group in the age of matrix product states. *Ann Phys*. (2011) 326:96. doi: 10.1016/j.aop.2010.09.012
39. Phien HN, Bengua JA, Tuan HD, Corboz P, Orus R. The iPEPS algorithm, improved: fast full update and gauge fixing. *Phys Rev B*. (2015) 92, 035142. doi: 10.1103/PhysRevB.92.035142
40. Evenbly G. Algorithms for tensor network renormalization. *Phys Rev B*. (2017) 95:045117. doi: 10.1103/PhysRevB.95.045117
41. Fishman M, White SR, Stoudenmire EM. The ITensor software library for tensor network calculations. *arXiv:2007.14822*. (2020). doi: 10.48550/arXiv.2007.14822
42. Kao Y-J, Hsieh Y-D, Chen P. Uni10: an open-source library for tensor network algorithms. *J Phys Conf Ser*. (2015) 640:012040. doi: 10.1088/1742-6596/640/1/012040
43. Haegeman J. *TensorOperations*. (2022). Available online at: https://github.com/Jutho/TensorOperations.jl (accessed April 14, 2022).
44. Hauschild J, Pollmann F. Efficient numerical simulations with Tensor Networks: Tensor Network Python (TeNPy). *SciPost Phys*. (2018). doi: 10.21468/SciPostPhysLectNotes.5
45. Al-Assam S, Clark SR, Jaksch D. The tensor network theory library. *J Stat Mech*. (2017) 2017:093102. doi: 10.1088/1742-5468/aa7df3
46. Olivares-Amaya R, Hu W, Nakatani N, Sharma S, Yang J, Chan G K-L. The *ab-initio*. density matrix renormalization group in practice. *J Chem Phys*. (2015) 142:034102. doi: 10.1063/1.4905329
47. Roberts C, Milsted A, Ganahl M, Zalcman A, Fontaine B, Zou Y, et al. TensorNetwork: a library for physics and machine learning. *arXiv preprint arXiv:1905.01330*. (2019). doi: 10.48550/arXiv.1905.01330
48. Oseledets V. *TT Toolbox*. (2014). Available online at: https://github.com/oseledets/TT-Toolbox (accessed July 7, 2021).
49. White SR. Density matrix formulation for quantum renormalization groups. *Phys Rev Lett*. (1992) 69:2863. doi: 10.1103/PhysRevLett.69.2863
50. White SR. Density-matrix algorithms for quantum renormalization groups. *Phys Rev B*. (1993) 48:10345. doi: 10.1103/PhysRevB.48.10345
51. Schollwoeck U. The density-matrix renormalization group. *Rev Mod Phys*. (2005) 77:259. doi: 10.1103/RevModPhys.77.259
52. Vidal G. Efficient classical simulation of slightly entangled quantum computations. *Phys Rev Lett*. (2003) 91:147902. doi: 10.1103/PhysRevLett.91.147902
53. Vidal G. Efficient simulation of one-dimensional quantum many-body systems. *Phys Rev Lett*. (2004) 93:040502. doi: 10.1103/PhysRevLett.93.040502
54. Verstraete F, Cirac JI. Renormalization algorithms for quantum-many-body systems in two and higher dimensions. *arXiv preprint arXiv:cond-mat/0407066*. doi: 10.48550/arXiv.cond-mat/0407066
55. Verstraete F, Cirac JI, Murg V. Matrix product states, projected entangled pair states, and variational renormalization group methods for quantum spin systems. *Adv Phys*. (2008) 57:143. doi: 10.1080/14789940801912366
56. Jordan J, Orus R, Vidal G, Verstraete F, Cirac JI. Classical simulation of infinite-size quantum lattice systems in two spatial dimensions. *Phys Rev Lett*. (2008) 101:250602. doi: 10.1103/PhysRevLett.101.250602
57. Vidal G. A class of quantum many-body states that can be efficiently simulated. *Phys Rev Lett*. (2008) 101:110501. doi: 10.1103/PhysRevLett.101.110501
58. Levin M, Nave CP. Tensor renormalization group approach to two-dimensional classical lattice models. *Phys Rev Lett*. (2007) 99:120601. doi: 10.1103/PhysRevLett.99.120601
59. Xie Z-Y, Chen J, Qin MP, Zhu JW, Yang LP, Xiang T. Coarse-graining renormalization by higher-order singular value decomposition. *Phys Rev B*. (2012) 86:045139. doi: 10.1103/PhysRevB.86.045139
60. Evenbly G, Vidal G. Tensor network renormalization. *Phys Rev Lett*. (2015) 115:180405. doi: 10.1103/PhysRevLett.115.180405
61. Evenbly G. *Tensors.net Website*. (2019). Available online at: https://www.tensors.net (accessed May 10, 2022).
62. Singh S, Pfeifer RNC, Vidal G. Tensor network decompositions in the presence of a global symmetry. *Phys Rev A*. (2010) 82:050301. doi: 10.1103/PhysRevA.82.050301
63. Singh S, Pfeifer RNC, Vidal G. Tensor network states and algorithms in the presence of a global U(1) symmetry. *Phys Rev B*. (2011) 83:115125. doi: 10.1103/PhysRevB.83.115125
64. Weichselbaum A. Non-abelian symmetries in tensor networks: a quantum symmetry space approach. *Ann Phys*. (2012) 327:2972–3047. doi: 10.1016/j.aop.2012.07.009
65. Sharma S. A general non-Abelian density matrix renormalization group algorithm with application to the C_2 dimer. *J Chem Phys*. (2015) 142:024107. doi: 10.1063/1.4905237
66. Keller S, Reiher M. Spin-adapted matrix product states and operators. *J Chem Phys*. (2016) 144:134101. doi: 10.1063/1.4944921
67. Nataf P, Mila F. Density matrix renormalization group simulations of SU(N) Heisenberg chains using standard Young tableaus: fundamental representation and comparison with a finite-size Bethe ansatz. *Phys Rev B*. (2018) 97:134420. doi: 10.1103/PhysRevB.97.134420
68. Schmoll P, Singh S, Rizzi M, Oras R. A programming guide for tensor networks with global SU(2) symmetry. *Ann Phys*. (2020) 419:168232. doi: 10.1016/j.aop.2020.168232
69. Pfeifer RNC, Haegeman J, Verstraete F. Faster identification of optimal contraction sequences for tensor networks. *Phys Rev E*. (2014) 90:033315. doi: 10.1103/PhysRevE.90.033315
70. Pfeifer RNC, Evenbly G. Improving the efficiency of variational tensor network algorithms. *Phys Rev B*. (2014) 89, 245118. doi: 10.1103/PhysRevB.89.245118

71. Dudek JM, Duenas-Osorio L, Vardi MY. Efficient contraction of large tensor networks for weighted model counting through graph decompositions. *arXiv preprint arXiv:1908.04381v2*. (2019). doi: 10.48550/arXiv.1908.04381
72. Gray J, Kourtis S. Hyper-optimized tensor network contraction. *Quantum*. (2021) 5:410. doi: 10.22331/q-2021-03-15-410
73. Pfeifer RNC, Evenbly G, Singh S, Vidal G. NCON: a tensor network contractor for MATLAB. *arXiv preprint arXiv:1402.0939*. (2014). doi: 10.48550/arXiv.1402.0939
74. Horn RA, Johnson CR. *Matrix Analysis*. Cambridge: Cambridge University Press (1985). doi: 10.1017/CBO9780511810817
75. Horn RA, Johnson CR. *Topics in Matrix Analysis*. Cambridge: Cambridge University Press (1991). doi: 10.1017/CBO9780511840371
76. Eckart C, Young G. The approximation of one matrix by another of lower rank. *Psychometrika*. (1936) 1:211–8. doi: 10.1007/BF02288367
77. Shi Y, Duan L, Vidal G. Classical simulation of quantum many-body systems with a tree tensor network. *Phys Rev A*. (2006) 74:022320. doi: 10.1103/PhysRevA.74.022320
78. Tagliacozzo L, Evenbly G, Vidal G. Simulation of two-dimensional quantum systems using a tree tensor network that exploits the entropic area law. *Phys Rev B*. (2009) 80:235127. doi: 10.1103/PhysRevB.80.235127
79. Evenbly G, Gauge fixing, canonical forms and optimal truncations in tensor networks with closed loops. *Phys Rev B*. (2018) 98:085155. doi: 10.1103/PhysRevB.98.085155
80. Holtz S, Rohwedder T, Schneider R. The alternating linear scheme for tensor optimization in the tensor train format. *SIAM J Sci Comput*. (2012) 34:A683–713. doi: 10.1137/100818893
81. Zhang Y, Solomonik E. On stability of tensor networks and canonical forms. *arXiv preprint arXiv:2001.01191*. (2020). doi: 10.48550/arXiv.2001.01191

Conflict of Interest: The author declares that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

frontiers | Frontiers in Applied Mathematics and Statistics

TECHNOLOGY AND CODE
published: 06 July 2022
doi: 10.3389/fams.2022.838601

ExaTN: Scalable GPU-Accelerated High-Performance Processing of General Tensor Networks at Exascale

Dmitry I. Lyakh[1]*, *Thien Nguyen*[2], *Daniel Claudino*[2], *Eugene Dumitrescu*[3] and *Alexander J. McCaskey*[4]

[1] Oak Ridge National Laboratory, National Center for Computational Sciences, Oak Ridge, TN, United States, [2] Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN, United States, [3] Computational Science and Engineering Division, Oak Ridge National Laboratory, Oak Ridge, TN, United States, [4] NVIDIA Corporation, Santa Clara, CA, United States

We present ExaTN (Exascale Tensor Networks), a scalable GPU-accelerated C++ library which can express and process tensor networks on shared- as well as distributed-memory high-performance computing platforms, including those equipped with GPU accelerators. Specifically, ExaTN provides the ability to build, transform, and numerically evaluate tensor networks with arbitrary graph structures and complexity. It also provides algorithmic primitives for the optimization of tensor factors inside a given tensor network in order to find an extremum of a chosen tensor network functional, which is one of the key numerical procedures in quantum many-body theory and quantum-inspired machine learning. Numerical primitives exposed by ExaTN provide the foundation for composing rather complex tensor network algorithms. We enumerate multiple application domains which can benefit from the capabilities of our library, including condensed matter physics, quantum chemistry, quantum circuit simulations, as well as quantum and classical machine learning, for some of which we provide preliminary demonstrations and performance benchmarks just to emphasize a broad utility of our library.

Keywords: tensor network, quantum many-body theory, quantum computing, quantum circuit, high performance computing, GPU

OPEN ACCESS

Edited by:
Paolo Bientinesi,
Umeå University, Sweden

Reviewed by:
Matthew Fishman,
Simons Foundation, United States
Devin Matthews,
Southern Methodist University, United States

***Correspondence:**
Dmitry I. Lyakh
quant4me@gmail.com

Specialty section:
This article was submitted to Mathematics of Computation and Data Science, a section of the journal Frontiers in Applied Mathematics and Statistics

Received: *18 December 2021*
Accepted: *15 June 2022*
Published: *06 July 2022*

Citation:
Lyakh DI, Nguyen T, Claudino D, Dumitrescu E and McCaskey AJ (2022) ExaTN: Scalable GPU-Accelerated High-Performance Processing of General Tensor Networks at Exascale. Front. Appl. Math. Stat. 8:838601. doi: 10.3389/fams.2022.838601

1. INTRODUCTION

Tensor networks have recently grown into a powerful and versatile tool for capturing and exploiting low-rank structure of rather diverse high-dimensional computational problems. A properly constructed tensor network, that is, a specific contraction of low-order/low-rank tensors forming a higher-order/higher-rank tensor, is capable of exposing the essential correlations between the components of the tensorized Hilbert space in which the solution to a given problem lives. The traditional application is quantum many-body theory where the exact quantum many-body wave-function is a vector in a high-dimensional Hilbert space constructed as a direct (tensor) product of elementary Hilbert spaces associated with individual quantum degrees of freedom. Having its roots in condensed matter physics, the structure of a tensor network is normally induced by the geometry of the problem (e.g., geometry of a spin lattice) and a suitably chosen renormalization procedure, reflecting the structure of the many-body entanglement (correlation) between quantum degrees of freedom

(e.g., spins, bosons, fermions). The well-known tensor network architectures from condensed matter physics include the matrix-product state (MPS) [1, 2] or tensor train (TT) [3], the projected entangled pair state (PEPS) [4], the tree tensor network (TTN) [5], and the multiscale entanglement renormalization ansatz (MERA) [6]. Not surprisingly, similar tensor network architectures have also been successfully utilized in quantum chemistry for describing electron correlations in molecules [7], [8], where individual molecular orbitals form quantum degrees of freedom (similar to spin sites in quantum lattice problems). Furthermore, tensor networks have found a prominent use in quantum circuit simulations, where they can be used for both the direct quantum circuit contraction [9–11] as well as approximate representations of the multi-qubit wave-functions and density matrices during their evolution [12–15], which reduces the computational cost of the simulation. Tensor networks have also found a prominent use in loading data into quantum circuits [16].

The ability of tensor networks to provide an efficient low-rank representation of high-dimensional tensors has recently spurred a number of applications in data analytics and machine learning. For example, tensor networks can be used for the tensor completion problem [17] or for the compression of the fully-connected deep neural network layers [18]. It was also shown that tensor networks can be employed in classification tasks (e.g., image classification) instead of deep neural networks [19–22]. Additionally, generative quantum machine learning can also benefit from tensor network representations [23].

Such a broad class of successful applications has resulted in a need for efficient software libraries [24] providing necessary primitives for composing tensor network algorithms. Apart from a plethora of basic tensor processing libraries, which are not the focus here, a number of specialized software packages have been developed recently, directly addressing the tensor network algorithms (in these latter software packages a tensor network is the first-class citizen). The ITensor library has been widely adopted in the quantum physics community [25], in particular because of its advanced support of abelian symmetries in tensor spaces. ITensor provides a rather rich set of features mostly targeting the density matrix renormalization group (DMRG) based algorithms executed on a single computer core/node (a recently introduced Julia version of ITensor brought in the GPU support). A more recent TensorTrace library focuses on more complex tensor network architectures, like MERA, and provides a nice graphical interface for building tensor networks [26] [the primary backend of TensorTrace is NCON [27]]. Another library gaining some popularity in condensed matter physics is TeNPy [28]. The CTF library [29] has been used to implement a number of advanced tensor network algorithms capable of running on distributed HPC systems [30, 31], also providing support for higher-order automated differentiation [31]. Perhaps the most advanced Python library for tensor network construction and processing is Quimb [32], which has been used in a number of diverse applications. Importantly, Quimb also supports distributed execution, either directly *via* MPI or *via* the DASK framework [33]. It also supports GPU execution *via* JAX [34]. Another Python library for performing tensor decompositions is TensorLy [35] which is mostly used in machine learning tasks. A more recent tensor network library is TensorNetwork [36], which is built on top of the TensorFlow framework aimed at quantum machine learning tasks.

Our C++ library ExaTN [37] has been independently developed in the recent years, with a main focus on high performance computing on current and future leadership computing platforms, in particular those equipped with GPU accelerators. The ExaTN library is not biased to any particular application domain and is rather general in the type of tensor networks that can be constructed, manipulated, and processed. It also provides several higher-level data structures and algorithms that can be used for remapping standard linear algebra problems to arbitrary tensor network manifolds. In this paper, we report the core functionality of ExaTN and show some initial demonstrations and performance benchmarks. To our knowledge, ExaTN provides one of the richest set of features for tensor network computations in C++, combined with native asynchronous parallel processing capabilities with support of distributed computing and GPU acceleration.

2. EXATN LIBRARY

2.1. Tensor Network Structures

The C++ API of ExaTN consists of two main groups of functions: declarative API and executive API. The declarative API functions (provided by multiple headers in `src/numerics` within the `exatn::numerics` namespace) are used for constructing and transforming tensor-based data structures, whereas the executive API functions (collected in the `src/exatn/exatn_numerics.hpp` header) are used for numerical processing (evaluation) of the constructed tensor-based data structures. Such separation of concerns enables a low-overhead manipulation with complex tensor networks consisting of tensors of arbitrary shape and size. The tensor storage allocation and the actual numerical computation is only performed when explicitly requested. Importantly, the specifics of the tensor storage and processing is completely transparent to the user, keeping the focus on the expression of the domain-specific numerical tensor algorithms without unnecessary exposure to the execution details.

The main basic object of the ExaTN library is `exatn::Tensor` (defined in `tensor.hpp`), which is an abstraction of the mathematical *tensor*. Loosely, we define a tensor $T^{abc\ldots}_{ijk\ldots}$ as a multi-indexed vector living in a linear space constructed as a direct product of basic (single-index) vector spaces. From the numerical point of view, a tensor (e.g., T^{abc}_{ijk}) can simply be viewed as a multi-dimensional array of real or complex numbers, `T[a,b,c,i,j,k]`. `exatn::Tensor` is defined by the following attributes:

- Name: Alphanumeric with optional underscores;
- Shape: Total number of tensor dimensions and their extents;
- Signature (optional): Identifies the tensor as a specific slice of a larger tensor, if needed;

Since an `exatn::Tensor` is subject to asynchronous processing, it must always be created as `std::shared_ptr<exatn::Tensor>` (a helper function `exatn::makeSharedTensor` is provided for convenience), for example:

```
#include ''exatn.hpp''
 auto my_tensor = exatn::makeSharedTensor(''MyTensor
     '',TensorShape{16,8,42});
```

In addition to the array-like tensor shape constructors, the ExaTN library also defines explicitly the concept of a vector space and subspace (`spaces.hpp`), enabling an optional definition of tensor dimensions over specific (named) vector spaces/subspaces which are expected to be defined and registered by the user beforehand (custom tensor signature). Otherwise, the tensor signature is simply specified by a tuple of base offsets defining the location of a tensor slice inside a larger tensor (defaults to a tuple of zeros). For example,

```
auto tensor_slice = makeSharedTensor("MyTensorSlice"
    ,TensorShape{12,8,20},TensorSignature{4,0,10});
```

defines a tensor slice `[4:12,0:8,10:20]` where each pair is *Start_Offset*:*Extent*.

Necessitated by many applications, ExaTN also enables the specification of the isometric groups of tensor dimensions. An *isometric group* is formed by one or more tensor dimensions such that a contraction over these dimensions with the complex-conjugate tensor results in the identity tensor over the remaining dimensions coming from both tensors, for example:

$$T^{\dagger}_{ijmn} T_{klmn} = \delta_{ij,kl} \tag{1}$$

where *mn* is an isometric group of indices (a summation over *mn* is implied). The identity tensor is just the identity map between the two groups of indices left after contraction over the isometric group of indices. A tensor can have either a single isometric group of dimensions or at most two such groups which together comprise all tensor dimensions, in which case the tensor is *unitary*, that is, in addition to Equation (1) we will also have:

$$T^{\dagger}_{mnij} T_{mnkl} = \delta_{ij,kl} \tag{2}$$

In order to register an isometric index group, one will need to invoke the `registerIsometry` method specifying the corresponding tuple of tensor dimensions (for example, first two dimensions of `MyTensor`):

```
my_tensor->registerIsometry({0,1});
```

ExaTN is capable of automatically identifying tensor contractions containing tensors with isometric index groups and subsequently simplifying them without computation by using rules analogous to (1) and (2). Apart from accounting for isometries, in a more general case, the current processing backend does not yet provide a special treatment for diagonal tensors of other kinds or other types of tensor sparsity (future work).

Of all basic tensor operations, *tensor contraction* is the most important operation in the tensor network calculus. A general contraction of two tensors can be expressed as

$$D_{i_1 i_2 \ldots i_N} = L_{k_1 k_2 \ldots k_M i_{j_1} i_{j_2} \ldots i_{j_L}} R_{k_1 k_2 \ldots k_M i_{j_{L+1}} i_{j_{L+2}} \ldots i_{j_N}} \tag{3}$$

up to an arbitrary permutation of indices inside each tensor, where a summation over all r.h.s-only indices is implied. The opposite operation, i.e., *tensor decomposition*, which decomposes a tensor into a contracted product of two tensors, is also supported by ExaTN. A *tensor network*, that is, a specific contraction of two or more tensors [2], is represented by the `exatn::TensorNetwork` class (`tensor_network.hpp`). Following the standard graphical notation illustrated in **Figure 1**, a tensor is graphically represented as a vertex with a number of directed or undirected edges, where each edge is uniquely associated with a specific tensor dimension (index), also called *mode*. A contraction over a pair of dimensions (modes) coming from two different tensors is then represented by a shared edge between two vertices associated with those tensors. In this case, a tensor network is generally represented as a directed multi-graph (note that **Figure 1** shows only undirected edges). In some cases, one may also need to consider tensor networks containing *hyper-contractions*, that is, simultaneous contractions of three or more dimensions (modes) coming from the same or multiple tensors that are labeled by the same index (hyper-edge). In such a case, the tensor network is generally represented as a directed multi-hypergraph in which some (hyper)-edges may connect more than two vertices. Currently, ExaTN does not support construction of general tensor hypergrahs, although it does support execution of pairwise pieces of tensor hyper-contractions, for example

$$D_{i_1 i_2 j_1} = L_{i_1 k_1 j_1} R_{j_1 i_2 k_1}, \tag{4}$$

where index j_1 is not summed over as it is present in the l.h.s. tensor as well (only the r.h.s.-only indices are implicitly summed over in our notation). We should note that tensor hypergraphs can always be converted to regular tensor graphs (tensor networks) by inserting order-3 Kronecker tensors which will convert all hyper-edges into regular edges connected to the Kronecker tensors.

An `exatn::TensorNetwork` object is constructed from one or more tensors called *input* tensors. Additionally, the ExaTN library automatically appends the so-called *output* tensor to each tensor network, which simply collects all uncontracted tensor dimensions from the input tensors. The total number of input tensors in a tensor network defines its *size*. The order of the output tensor defines the *order* of the tensor network. Additionally, one can also specify whether a tensor network describes a manifold in the primary (ket) or dual (bra) tensor space. ExaTN provides multiple ways for building a tensor network (see the `placeTensor`, `appendTensor`, and `appendTensorGate` methods in "`tensor_network.hpp`" for details). The most general way is to append tensors one-by-one by explicitly specifying their connectivity, i.e., connections between dimensions of distinct tensors *via* graph edges (`placeTensor`). In this way, one

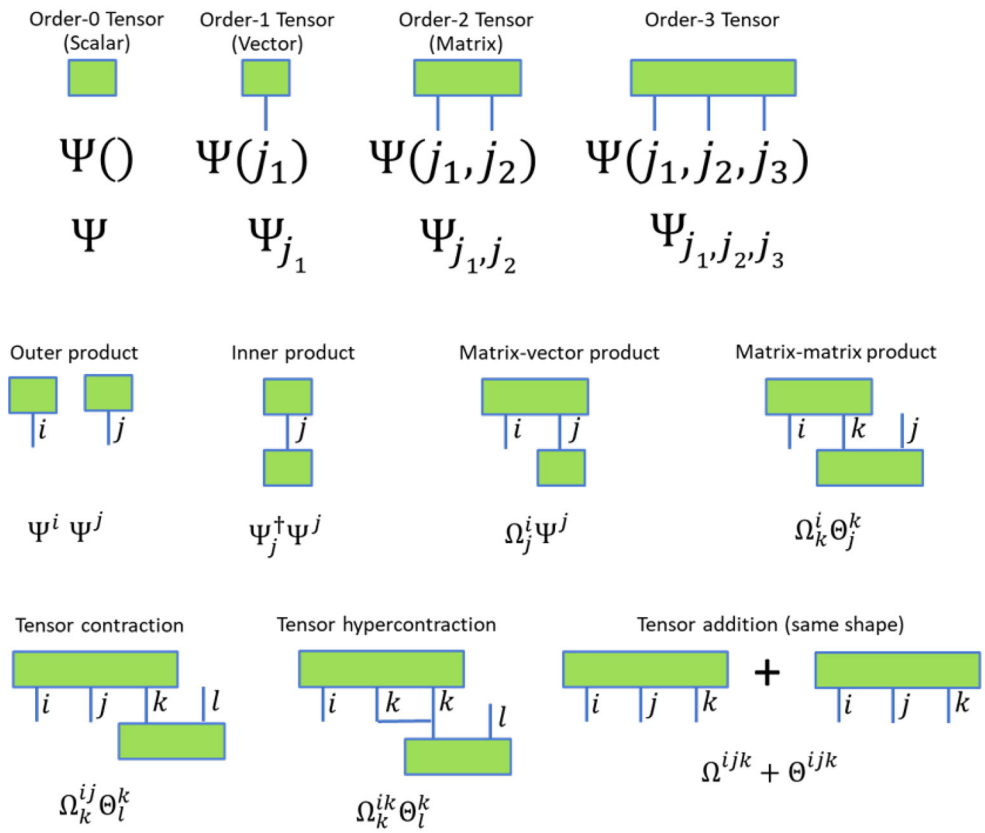


FIGURE 1 | Graphical diagrams representing tensors and tensor operations.

can construct an arbitrarily complex tensor network but this gradual construction mechanism has to be fully completed before a tensor network can be used. As an alternative, ExaTN also allows gradual construction of tensor networks where each intermediate tensor network is also a valid tensor network that can be used immediately. This is achieved by appending new input tensors by pairing their dimensions with those of the current output tensor, thus indirectly linking the input tensors to a desired network connectivity graph (`appendTensor` and `appendTensorGate`). Finally, `exatn::TensorNetwork` class also accepts user-defined custom *builders* (OOP builder pattern), that is, concrete implementations of an abstract OOP builder interface (`exatn::NetworkBuilder`) that are specialized for the construction of a desired tensor network topology (like MPS, TTN, PEPS, MERA, etc.) in one shot.

There are a number of transformation methods provided by the `exatn::TensorNetwork` class. These include inserting new tensors in the tensor network, deleting tensors from the tensor network, merging two tensors in the tensor network, splitting a tensor inside the tensor network into two tensors, combining two tensor networks into a larger tensor network, identifying and removing identities caused by the isometric tensor pairs, etc. All these are manipulations on abstract tensors that are not concerned with an immediate numerical evaluation (and storage). However, numerical evaluation of the tensor network, that is, evaluation of the output tensor of that tensor network, or any other necessary numerical operation can be performed at any stage *via* the executive API. Importantly, numerical evaluation of a tensor network requires determination of a cost-optimal tensor contraction path which prescribes the order in which the input tensors of the tensor network are contracted. The cost function is typically the total Flop count, but it can be more elaborate (Flop count balanced with memory requirements and/or arithmetic intensity). There is no efficient algorithm capable of determining the true optimum for a general case, but some efficient heuristics exist [38, 39]. For the sake of generality, ExaTN provides an abstract interface for the tensor contraction path finder that can bind to any concrete user-provided implementation of a desired contraction path optimization algorithm. The default optimization algorithm used by ExaTN is a simplified variant of the recursive multi-level graph partitioning algorithm from [38] implemented *via* the graph partitioning library Metis [40] (without Bayesian hyper-parameter optimization). Users who use NVIDIA CUDA can also leverage the cuQuantum::cuTensorNet library [1] which is fully integrated with ExaTN as an optional dependency. It delivers the state-of-the-art quality as well as performance in contraction path searches (in addition to highly-efficient tensor

[1]https://docs.nvidia.com/cuda/cuquantum/cutensornet/index.html

contraction execution). There is also an experimental binding to CoTenGra [38] (in a separate branch of ExaTN).

Importantly, apart from constructing and processing individual tensor networks, ExaTN also provides API for constructing and processing linear combinations of tensor networks, implemented by the `exatn::TensorExpansion` class (`tensor_expansion.hpp`). Specifically, a *tensor network expansion* is a linear combination of tensor networks of the same order and output shape (an example is illustrated in **Figure 2A**). A tensor network expansion can be constructed by gradually appending individual tensor networks with their respective complex coefficients. Numerical evaluation of a tensor network expansion results in computing the output tensor of each individual tensor network component, followed by the accumulation of all computed output tensors which have the same shape. ExaTN also provides API for constructing the inner and outer products of two tensor network expansions. By design, a given tensor network expansion either belongs to the primary (ket) or to the dual (bra) tensor space where it defines a tensor network manifold (a manifold of tensors which can be represented by the given tensor network or tensor network expansion exactly). In order to introduce the operator algebra on such tensor network manifolds, ExaTN provides the `exatn::TensorOperator` class (`tensor_operator.hpp`). A *tensor network operator* is a linear combination of tensor networks in which additionally the dimensions of the output tensor in each tensor network are individually assigned to either the ket or the bra tensor spaces (an example is illustrated in **Figure 2B**). Thus, such a tensor network operator defines an operator manifold, establishing a map between the ket and bra tensor spaces populated by the tensor network manifolds defined by the tensor network expansions. Naturally, ExaTN provides API for applying arbitrary tensor network operators to arbitrary tensor network expansions and for defining matrix elements of tensor network operators with respect to arbitrary ket and bra tensor network expansions, that is, in Dirac notation:

$$MatrixElement(i,j) = \langle TensorExpansion(i)|TensorOperator|TensorExpansion(j)\rangle, \quad (5)$$

In this construction, a tensor network expansion replaces the notion of a vector, and a tensor network operator replaces the notion of a linear operator: A tensor network operator maps tensor network expansions (tensor network manifolds) from one tensor space to tensor network expansions (tensor network manifolds) in another (or same) tensor space.

In many applications of tensor networks the computational problem lies in the optimization of a suitably chosen tensor network functional to find its extreme values. In ExaTN, a tensor network functional is defined as a tensor network expansion of order 0 (scalar), thus having no uncontracted edges. By optimizing the individual tensor factors inside the given tensor network functional, one can find its extrema using gradient-based optimization techniques. This requires computing the gradient of the tensor network functional with respect to each optimized tensor. ExaTN provides API for computing the gradient of an arbitrary tensor network functional with respect to any given tensor. Furthermore, ExaTN implements numerical procedures that can efficiently project a tensor network expansion living on one tensor network manifold to a tensor network expansion living on another tensor network manifold, as well as solve linear and eigen systems defined on arbitrary tensor network manifolds. This higher-level functionality, however, is not the focus of the current paper and will be described elsewhere.

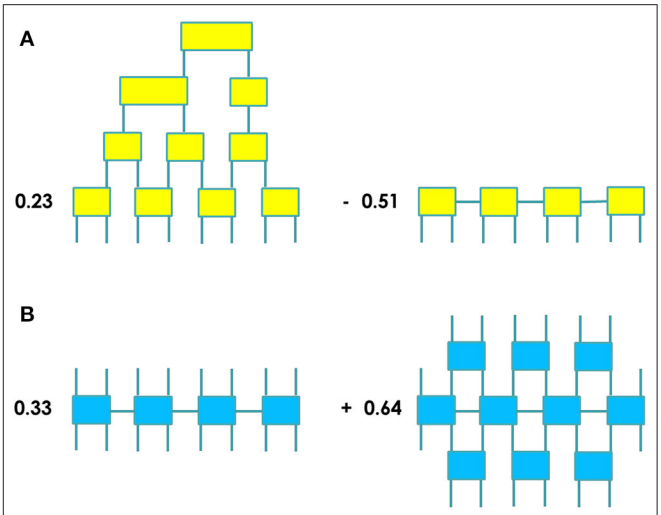


FIGURE 2 | (A) An example of a tensor network expansion as a linear combination of two tensor networks of the same order; **(B)** An example of a tensor network operator as a linear combination of two tensor networks with open legs establishing a map between the ket and bra spaces.

2.2. Tensor Network Processing

Processing of tensors, tensor networks and tensor expansions is done *via* the executive API (`exatn_numerics.hpp`) by the ExaTN parallel runtime (ExaTN-RT). The ExaTN parallel runtime provides a fully asynchronous execution of basic numerical tensor operations extending the abstract `exatn::TensorOperation` class (`tensor_operation.hpp`), in particular tensor creation, tensor destruction, tensor initialization, tensor transformation, tensor norm evaluation, tensor copy/slicing/insertion, tensor addition, tensor contraction, tensor decomposition (*via* the singular value decomposition of the tensor matricization), and some tensor communication/reduction operations. Additionally, new user-defined numerical tensor operations can be implemented either *via* extending the `exatn::TensorTransformation` class (for unary tensor transformations) or *via* extending the abstract `exatn::TensorOperation` class (`tensor_operation.hpp`) for more general operations (non-unary).

The executive API can be used for submitting individual basic tensor operations as well as entire tensor networks and tensor network expansions for their numerical evaluation. The latter

are first decomposed into basic tensor operations which are then submitted to the ExaTN runtime for asynchronous processing. The synchronization is done by either synchronizing on a desired tensor (to make sure all update operations have completed on that particular tensor) or synchronizing all outstanding tensor operations previously submitted to the ExaTN runtime (barrier semantics). Few examples:

```
include "exatn.hpp"

//Declare tensors:
auto tensor_A = exatn::makeSharedTensor("A",
     TensorShape{12,8,20});
auto tensor_B = exatn::makeSharedTensor("B",
     TensorShape{8,64,20});
auto tensor_C = exatn::makeSharedTensor("C",
     TensorShape{64,12});

//Allocate tensor storage:
bool success = true;
success = exatn::createTensor(tensor_A,
     TensorElementType::REAL64);
success = exatn::createTensor(tensor_B,
     TensorElementType::REAL64);
success = exatn::createTensor(tensor_C,
     TensorElementType::REAL64);

//Initialize tensors:
success = exatn::initTensorRnd("A");
success = exatn::initTensorRnd("B");
success = exatn::initTensorRnd("C",0.0);

//Perform tensor contraction (1.0 is a scalar
     multiplier):
success = exatn::contractTensors("C(a,b)+=A(b,i,j)*B
     (i,a,j)",1.0);

//Declare, allocate, and initialize a new tensor:
auto tensor_D = exatn::makeSharedTensor("D",
     TensorShape{12,12});
success = exatn::createTensor(tensor_D,
     TensorElementType::REAL64);
success = exatn::initTensorRnd("D",0.0);

//Evaluate a tensor network:
success = exatn::evaluateTensorNetwork("MyNetwork","
     D(a,b)+=A(b,i,j)*B(i,k,j)*C(k,a)");

//Sync all submitted tensor operations to this point
      (barrier):
success = exatn::sync();
```

In the above code snippet, all executive API calls are *non-blocking* (except `exatn::sync`). All submitted tensor operations will be complete after return from the `exatn::sync` call. When submitted for processing, tensor operations are appended to the dynamic directed acyclic graph (DAG) stored inside the ExaTN runtime. The dynamic DAG is tracking data (tensor) dependencies automatically, thus avoiding race conditions. Inside the ExaTN runtime, the DAG is being constantly traversed by the ExaTN *graph executor* which identifies dependency-free tensor operations and submits them for execution by the ExaTN *node executor*. The ExaTN graph executor implements the OOP *visitor pattern* where the visitor (ExaTN node executor) visits/executes DAG nodes (tensor operations) by implementing overloads of the `execute` method for each supported tensor operation. The default implementation of the polymorphic ExaTN node executor interface is backed by the tensor processing library TAL-SH [41]. However, other tensor processing backends can also be easily plugged-in as long as they provide the implementation of all required basic tensor operations. The default TAL-SH tensor processing backend supports concurrent execution of basic tensor operations on multicore CPU as well as single/multiple NVIDIA or AMD GPU (AMD support is largely experimental at the ExaTN level). TAL-SH provides an automatic tensor storage and residence management within the combined Host+GPU memory pool, supporting a fully asynchronous execution on GPUs. In particular, a tensor contraction involving large tensors can be executed on multiple GPUs using the entire Host memory pool. The selection of the execution device is performed by the TAL-SH library automatically during run time, based on tensor sizes, flop count (and possibly arithmetic intensity), and current data residence (data locality). The default GPU tensor contraction algorithm is based on the matrix-matrix multiplication (e.g., *via* cuBLAS) accompanied by an optimized tensor transpose algorithm [42, 43]. Optionally, the default tensor contraction implementation can be swapped with the NVIDIA cuTENSOR backend[2] integrated with the TAL-SH library as an external dependency specifically for NVIDIA GPU. Finally, we have recently integrated ExaTN with the cuQuantum::cuTensorNet library[1] that allows ExaTN to process a whole tensor network in one shot, with superior performance in both the contraction path search and actual numerical computation on NVIDIA GPUs.

During the execution of tensor workloads, the storage and execution details are completely hidden from the user (client). The only data exchange between the client and the runtime occurs when the client is initializing a tensor with some data or retrieving tensor data back to the user space. The tensor initialization accepts real or complex scalars or arrays of single or double precision. The tensor retrieval requires tensor synchronization and returns a C++ `talsh::Tensor` object defined in the `talshxx.hpp` header of the TAL-SH library [41]. A tensor can be retrieved either in whole or in part (by a slice), but in both cases it is just a *copy* of the tensor (or its slice). Tensors can also be stored on disk.

The ExaTN library also supports distributed execution across many (potentially GPU-accelerated) compute nodes *via* the MPI interface. Currently, there are multiple levels of distributed parallelism. At the most coarse level, a tensor network expansion submitted for numerical evaluation across multiple MPI processes can distribute evaluation of its individual components (tensor networks) among subgroups of those MPI processes. Then, each tensor network can be evaluated by multiple MPI processes within a subgroup in parallel. Specifically, the intermediate tensors of the tensor network, that is, temporary tensors which are neither inputs nor outputs of the tensor network, can be decomposed into smaller slices which can be computed independently (slices are obtained *via* segmentation of tensor dimensions). The complete tensor network evaluation

[2]https://docs.nvidia.com/cuda/cutensor/index.html

requires computation of all slices of intermediate tensors that can be distributed among multiple/many MPI processes, with a minimal communication at the end (`MPI_Allreduce` reduction of the output tensor). The ExaTN library provides an explicit API for creating and splitting groups of MPI processes into subgroups, thus providing a multi-level composable resource isolation mechanism. Additionally, another level of parallelization is possible by utilizing a distributed tensor processing backend for basic numerical tensor operations executed by the ExaTN runtime, which will allow (distributed) storage of larger tensors but will result in a dense communication pattern within an executing group of MPI processes.

3. RESULTS AND DISCUSSION

3.1. Condensed Matter Physics Simulations

Quantum-mechanical condensed matter problems are typically too complex to be addressed by brute-force numerical methods because the dimension of the matrix representation of the Hamiltonian grows exponentially with the number of spin lattice sites. Aside from a small set of exactly solvable models, which eliminate complexity by exploiting underlying symmetries and constants of motion, approximate techniques are needed to address this important class of problems. Mean-field approximations and low-order perturbation theory are only appropriate for problems containing relatively limited inter-particle correlations. Quantum Monte-Carlo is a state-of-the-art technique but is rendered inefficient in many settings by the ubiquitous sign problem [44]. Tensor network factorizations, with complexity varying with dimensionality of the problem and the system correlation length, constitute an alternative formalism to describe quantum states in condensed matter systems. A numerical solution to Wilson's renormalization group, specifically for the Kondo impurity problem, was the original motivation for the matrix-product state (MPS) tensor network [45], although the explicit MPS structure was not realized until later [1]. Following the famous density matrix renormalization group algorithm [45], the numerical optimization consists of a series of linear algebra operations, including tensor contractions and singular value decompositions (SVD), which are swept across the spatial extent of the MPS spin chain [46]. Building on early MPS developments, a suite of more flexible and advanced tensor networks have been developed to deal with situations which are not naturally amenable to the MPS description. For example, the extension of tensor networks to problems arising in two spatial dimensions may be addressed by the projected entangled-pair states (PEPS) [47, 48]. Further modifications of the MPS formalism have resulted in the tree tensor network (TTN) [47] and the multiscale entanglement renormalization ansatz (MERA) [6, 49]. The latter tensor network ansatz can efficiently represent critical long-range ordered states. Aside from the variational MPS optimization, real and imaginary time-evolving block decimation (TEBD) algorithms [50–53] are the other two algorithms worth mentioning as they provide ways to deal with dynamical correlations and provide alternative means for determining quantum eigenstates and sample partition functions [54], respectively.

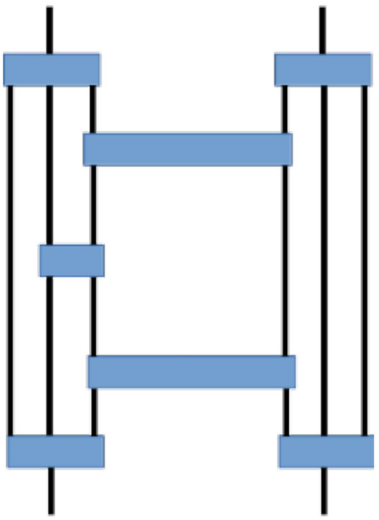


FIGURE 3 | Graphical diagram depicting a fragment of the 3:1 MERA tensor network.

The ExaTN library, combined with standard BLAS/LAPACK libraries, provides all necessary utilities for implementing the aforementioned numerical algorithms for arbitrary tensor network ansaetze, regardless of particular details such as network topology (as long as it is a graph-based topology). This also includes numerical algorithms for dealing with formally infinite (periodic) tensor networks [55]. Typically, all these algorithms are based on tensor contraction and tensor decomposition operations, where the latter is traditionally implemented *via* tensor matricization and SVD. **Figure 3** shows a typical example of a tensor network fragment (expressed graphically as a many-body diagram) for the 1D MERA 3:1 ansatz taken from Pfeifer et al. [56]. Such tensor network fragments are common in tensor network optimization procedures, representing gradients of optimization functionals, density matrices, etc. To illustrate the performance of the ExaTN library, we numerically evaluated this representative tensor network fragment on 4, 8, 16, 32, 64, and 128 nodes of the Summit supercomputer (each Summit node consists of 2 IBM Power 9 CPU with 256 GB RAM each and 6 NVIDIA V100 GPU with 16 GB RAM each). All tensor dimensions in this tensor network fragment were set to have the same extent of 64 (bond as well as lattice site dimension of 64). **Table 1** shows execution times and absolute performance. We observe both excellent parallel efficiency and high absolute efficiency when executed in a hybrid CPU+GPU setting (NVIDIA V100 GPU has a theoretical single-precision peak at ∼15 TFlop/s).

3.2. Quantum Chemistry Simulations

Tensor network methods used in condensed matter physics have also found many applications in quantum chemistry [7, 8] by simply remapping molecular (or spin) orbitals to spin sites while employing *ab initio* Hamiltonians instead of model Hamiltonians. However, these *ab initio* Hamiltonians,

TABLE 1 | Performance of numerical evaluation of the 3:1 MERA fragment on Summit supercomputer.

Number of nodes	Time, s	Performance, TFlop/s/GPU
4	77.11	10.743
8	38.88	10.716
16	19.96	10.435
32	10.54	10.117
64	5.53	9.637
128	3.96	7.333

Each Summit node has 6 NVIDIA V100 16 GB GPUs. The peak (single-precision) performance per GPU is around 15 TFlop/s.

TABLE 2 | Convergence of the ground state correlation energy with respect to the maximal bond dimension for the AIEM Hamiltonian describing a combined system of 48 2-level chemical fragments.

Max bond dimension	Correlation energy, Hartree
1	−1.967
2	−1.983
4	−1.992
8	−1.992

Total Hilbert space dimension is 2^{48}.

although quite accurate, could be numerically costly, limiting the scope of applicability of such tensor network methods. Fortunately, chemical properties that are largely governed by certain physical features can greatly benefit from reduced (effective) Hamiltonians, where the Hamiltonian is designed to specifically target the sought chemical property. For example, certain organic polymers and protein aggregates exhibit pronounced photochemical activity mediated by weakly-interacting chromophores [57]. The *ab initio* treatments in such cases are often intractable due to an enormous dimension of the corresponding Hilbert space, and this is aggravated by the requirement of inclusion of multiple low-lying excited states. Fortuitously, these problems lend themselves naturally to the so-called *ab initio* exciton model (AIEM) [58]. In this model, each (weakly-interacting) subunit/monomer is initially described by its own local *ab initio* Hamiltonian. The fact that the constituent monomers are spatially separated provides the justification for the approximations used by the model, namely (1) cross-fragment fermionic antisymmetry is relaxed, which means 2-body interactions can be reduced to dipole interactions between monomers, (2) only nearest-neighbor interactions are of numerical significance, and (3) the energy eigenspectrum can be approximated by configuration interaction of tensor products of ground and several subsequent excited monomer states. Consequently, the AEIM Hamiltonian can simply be expressed as a sum of monomer and dimer terms:

$$\hat{H} = \sum_{\mathrm{A}} h_{\mathrm{A}} \hat{H}_{\mathrm{A}} + \sum_{\mathrm{A,B}} h_{\mathrm{AB}} \hat{H}_{\mathrm{A}} \otimes \hat{H}_{\mathrm{B}}, \tag{6}$$

where A and B are the subunit (monomer) labels and the compound index AB sums over nearest-neighbor pairs of subunits (dimers), with the scalars h_{A} and h_{AB} quantifying the local and interaction energies, respectively. These matrix elements are normally computed by a relatively cheap self-consistent-field method, for example, the density functional theory.

The workflow involved in the AIEM Hamiltonian can be briefly summarized as follows: (1) local Hamiltonian is obtained from monomer quantum chemistry simulations; (2) dipole interactions between adjacent monomers using the outputs from (1) are computed; (3) AEIM Hamiltonian is constructed from computations in (2); (4) AIEM Hamiltonian in (3) is diagonalized in the space of configurations of tensor products of individual monomer states. In the simplest case, where only the first excited state in each monomer is considered, the eigenspace of Equation (6) is a 2^N-dimensional Hilbert space, with N being the number of monomers, which quickly becomes intractable with a growing N. However, the weakly entangled nature of many eigenstates of the AIEM Hamiltonian makes it an ideal target for approximations based on tensor networks. Alternatively, when a stronger entanglement is present, the AIEM Hamiltonian is a prospect application for quantum computing by exploring the isomorphism between the AIEM Hamiltonian in k-fold monomer excitations with a spin lattice Hamiltonian that is immediately expressible in the tensor product space of k-dimensional qudits [59].

To demonstrate the utility of the ExaTN library in this case, we implemented a brute-force version of the direct ground-state search procedure based on a chosen (arbitrary) tensor network ansatz. Specifically, given the AIEM Hamiltonian and a fully specified tensor network ansatz, the ExaTN library was used to minimize the Hamiltonian expectation value by optimizing the constituting tensors (inside the chosen tensor network ansatz) using the steepest descent algorithm. For demonstration, we chose the AIEM model representing a chemical system with 48 2-level fragments (monomers) that can be mapped to 48 qubits, with the total Hilbert space dimension of 2^{48}. We used the binary planar tensor tree topology for the tensor network ansatz and limited the maximal bond dimension in the tree to 1, 2, 4, and 8. **Table 2** shows the convergence of the obtained ground state correlation energy with respect to the maximal bond dimension. As one can see, the mHartree accuracy for the ground electronic state is already reached at the maximal bond dimension of 4, showing low entanglement in this weakly-interacting system. This electronic ground state search in a 2^{48}-dimensional Hilbert space was executed on 16 nodes of Summit supercomputer, with each iteration of the steepest descent algorithm taking around 20 s. We should note that in this illustrative example we did not enforce isometry on the tensors constituting the tree tensor network used for representing the ground state of the AIEM Hamiltonian. Further enforcing and exploiting tensor isometry will significantly reduce the computational cost, making it possible to treat much larger systems. We should also note that the convergence of the steepest descent algorithm used here was rather slow. Alternative algorithms, like conjugate gradient,

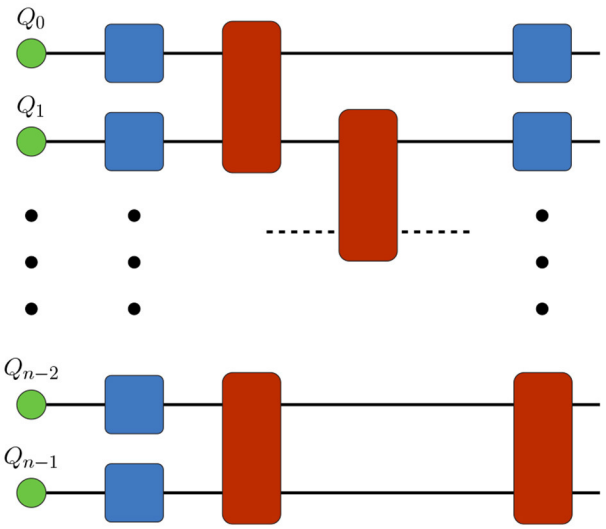


FIGURE 4 | Quantum circuit simulation by direct contraction of the circuit tensor network with ExaTN. Qubits are represented as order-1 tensors. Tensors describing quantum gates are appended to the qubit register according to the quantum circuit specification. Full contraction of the tensor network produces the complete wave-function.

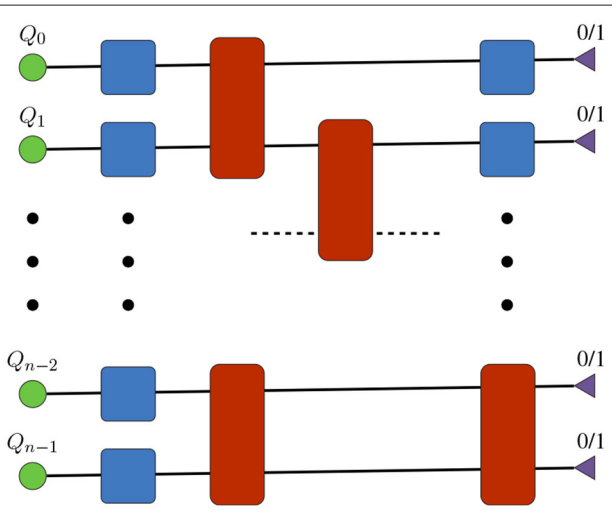


FIGURE 5 | ExaTN tensor network for bit-string amplitude computation. The tensor network representing the input quantum circuit is constructed as described in **Figure 4**. Triangular-shaped order-1 tensors are 0/1 tensors, representing the projected state.

or density matrix renormalization group, or imaginary-time evolution could potentially result in a faster convergence.

3.3. Simulations of Quantum Circuits

The ExaTN library has also been extensively employed as a parallel processing backend in the HPC quantum circuit simulator called TN-QVM [12, 60], one of the virtual quantum processing unit (QPU) backends available in the hybrid quantum/classical programming framework XACC [61]. TN-QVM implements a number of advanced quantum circuit simulation methods, where each method creates, transforms, and processes all necessary tensor network objects *via* the ExaTN library. Below we briefly discuss the utility of ExaTN in the implementation of these different simulation methods.

TABLE 3 | Average GPU performance in evaluation of a single amplitude of the 53-qubit Sycamore 2D random quantum circuit of depth 14.

Computing system	Precision	Average TFlop/s/GPU
DGX-A100, 8 A100 GPU	TF32	34.73
DGX-A100, 8 A100 GPU	FP32	15.06
Summit, 64 nodes, 384 V100 GPU	FP32	7.99
Dual 64-core AMD Rome CPU	FP32	2.98

3.3.1. Direct Contraction of Quantum Circuits

In this mode of simulation [9], TN-QVM represents the initial state of an n-qubit register as a rank-1 product of n order-1 tensors. Then it appends order-2 and order-4 tensors to this qubit register to simulate single- and two-qubit gates, respectively (**Figure 4**). Finally, for each qubit line one can either choose to keep it open or project it to any 1-qubit state, thus specifying an output wave-function slice to be computed in a chosen basis, as shown in **Figure 5**. Effectively, TN-QVM constructs a tensor network for

$$\langle \Psi_f | U_{circuit} | \Psi_0 \rangle, \tag{7}$$

where Ψ_0 is the initial rank-1 state of the n-qubit register while Ψ_f defines the output wave-function slice.

Once the obtained tensor network is submitted to ExaTN for parallel processing, the library analyzes the tensor network graph to heuristically determine the tensor contraction sequence (contraction path) which is pseudo-optimal in terms of the Flop count or time to solution (given some performance model). Any intermediate tensors that require more memory than available per MPI process are automatically split into smaller slices by splitting selected tensor modes. The computation of these slices is distributed across all MPI processes. Intermediate slicing in principle enables simulation of output amplitudes of arbitrarily large quantum circuits, that is, the memory constraints are lifted by the increased execution time. The resulting overhead in execution time is highly sensitive to the selection of tensor modes to be sliced, but there exists a rather efficient simple heuristics [62].

Table 3 illustrates performance of the TN-QVM/ExaTN software in simulating a single bit-string amplitude of a 2D random quantum circuit of depth 14 from Google's quantum supremacy experiments [63] on different classical HPC hardware [the performance data is taken from [60]].

3.3.2. Computation of Operator Expectation Values

A ubiquitous use case in quantum circuit simulations is calculation of the expectation values of measurement operators,

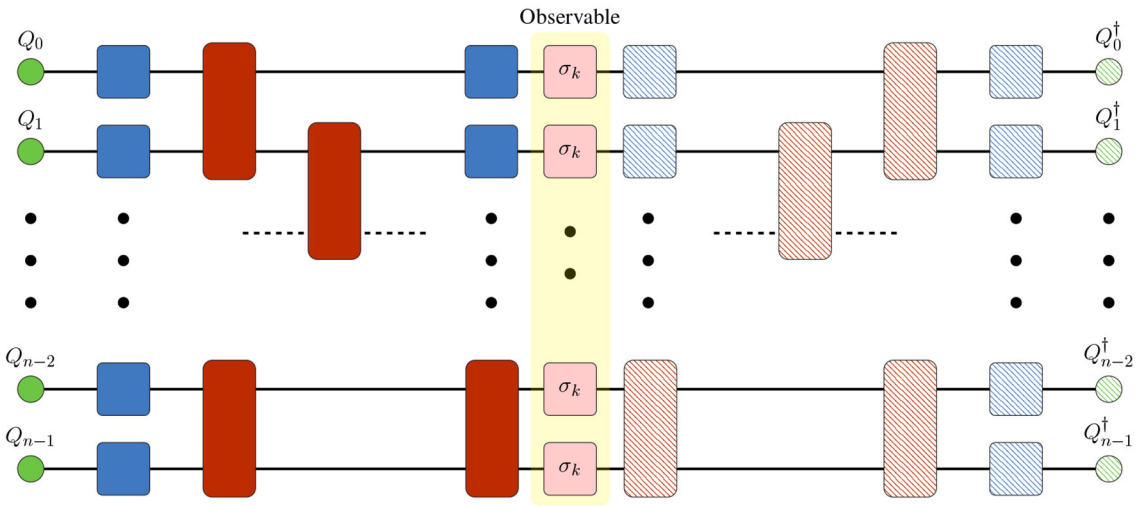


FIGURE 6 | ExaTN tensor network for expectation value evaluation using the circuit conjugation technique. Hashed-filled tensors represent the complex conjugates of the solid-filled counterparts.

which can be done with tensor networks very conveniently. TN-QVM provides two different methods for this purpose. First is based on appending the string of measurement operators to the output legs of the quantum circuit tensor network, followed by a closure with the conjugate tensor network, as illustrated in **Figure 6**. Numerical evaluation of this combined tensor network delivers the scalar expectation value. All necessary operations for combining tensor networks and subsequent numerical evaluation are provided by ExaTN API. Additionally, ExaTN can intelligently collapse a unitary tensor and its conjugate upon their direct contact in a tensor network, thus simplifying the tensor network if the measurement operators are sparse.

The second method is based on wavefunction slicing, where TN-QVM slices the output wave-function tensor as dictated by the memory constraints, computes the expectation value for each slice, and recombines them to form the final result, all done *via* the ExaTN API. As compared to the circuit conjugation method, this approach has an advantage in simulations of deeper quantum circuits with non-local observables and a moderate number of qubits. The partial expectation value calculation tasks can be distributed in a massively parallel manner.

3.3.3. Approximate Evaluation of Quantum Circuits

In addition to exact simulation methods, TN-QVM also provides the ability to evaluate the quantum circuit wave-function approximately as a projection on a user-defined tensor network manifold. Specifically, a user can choose a tensor network ansatz with arbitrary topology and bond dimensions. Once the ansatz is chosen, TN-QVM will cut the quantum circuit into chunks of equal depth and evaluate the action of each chunk on the chosen tensor network ansatz while remapping the result back to the same tensor network form (in general, one should allow tensor network bond dimensions to grow along the quantum circuit). In this simulation method, the

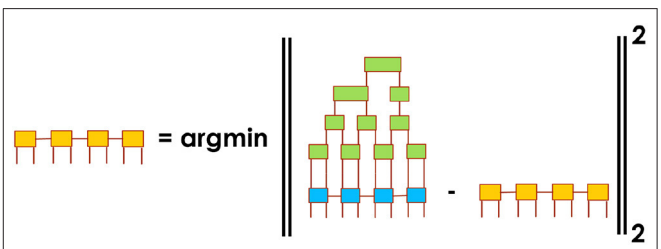


FIGURE 7 | An example of reconstruction of one (more complex) tensor network as another (simpler) tensor network by minimizing the Euclidean norm of the difference.

key procedure is a projection of a given tensor network to a tensor network manifold of a different form (different topology and/or bond dimensions), as illustrated in **Figure 7** where a more complex tensor network is approximately reconstructed by a simpler tensor network. ExaTN provides a simple API to perform such a reconstruction procedure, implemented by the `exatn::TensorNetworkReconstructor` class. Importantly, the reconstruction procedure also returns the reconstruction fidelity which can then be used for making decisions on dynamically increasing the bond dimensions in the reconstructing tensor network (adaptive tensor networks). The execution of the tensor network reconstruction automatically leverages multiple levels of parallelization provided by the ExaTN parallel runtime as described above.

Another approximate quantum circuit simulation method implemented in TN-QVM is based on a matrix product state (MPS) representation of the multi-qubit wave-function [60] which is evaluated *via* the classical contract/decompose algorithm [12]. This algorithm adapts

the simulation accuracy to the available computational resources. ExaTN provides a convenient MPS builder utility *via* the `exatn::numerics::NetworkBuilder` interface as well as API for tensor contraction and decomposition.

3.4. Machine Learning

The utility of tensor networks in classical machine learning was realized relatively recently. Here we can distinguish two categories of applications: (1) Building machine learning models with tensor networks, and (2) using tensor networks in conventional deep neural network models for compressing the neural network layers. In the first approach, a tensor network model can be trained to fulfill classification tasks [19–22]. The input data, for example, an image, is typically encoded as a direct-product state of many quantum degrees of freedom, where each quantum degree of freedom corresponds to a single pixel (in case of images). By optimizing the tensors constituting the tensor network, one minimizes the error of the classification. Image classification is particularly amenable to the tensor network analysis because of the locally correlated structure of typical images. In the second approach, tensor networks, i.e., MPS, are used for compressing the layers of a deep neural network, thus reducing the memory requirements and introducing regularization in the training phase [18, 64]. The ExaTN library provides necessary primitives for both use cases, in particular construction and contraction of an arbitrary tensor network as well as evaluation of the gradient of a tensor network functional with respect to a given tensor. Additionally, the first use case may also benefit from the availability of the `exatn::TensorExpansion` class suitable for representing a linear combination of tensor networks projected on different instances from the training data batch.

4. CONCLUSIONS

As demonstrated above, the ExaTN library provides state-of-the-art capabilities for construction, transformation, and parallel processing of tensor networks on laptops, workstations, and HPC platforms, including GPU-accelerated ones, in multiple domains. Furthermore, building upon regular tensor networks, ExaTN also introduces higher-level objects, specifically linear combinations of tensor networks or tensor network operators which serve as more flexible analogs of tensors and tensor operators living on differential manifolds instead of regular linear spaces. ExaTN also provides a general tensor network reconstruction procedure which can efficiently project any tensor network to another tensor network of different topology/configuration. Importantly, these mathematical primitives enable a systematic derivation of approximate tensor network renormalization schemes as well as reformulation of linear algebra solvers on low-rank tensor network manifolds, which is currently an active field of research in applied math. We are actively working on implementing such solvers in the ExaTN library, leveraging all benefits of multi-level parallelization and GPU acceleration provided by the ExaTN parallel runtime. Another direction of our development work is further adoption of vendor-provided highly-optimized math libraries that will enhance the performance of ExaTN on respective HPC platforms.

DATA AVAILABILITY STATEMENT

Publicly available datasets were analyzed in this study. This data can be found at: https://github.com/ORNL-QCI/exatn.git.

AUTHOR CONTRIBUTIONS

DL was the technical lead for the research and development efforts described in this paper, including conceptualization, algorithm/software design and implementation, simulations, and manuscript writing. TN was responsible for integrating the ExaTN library into the TN-QVM simulator as well as performing actual quantum computing simulations and describing them in the text. DC was responsible for performing quantum chemistry simulations and describing them in the text. ED was responsible for condensed matter physics applications and relevant text. AM coordinated the ExaTN development efforts and contributed to software design and implementation. All authors contributed to the article and approved the submitted version.

FUNDING

We would like to acknowledge the Laboratory Directed Research and Development (LDRD) funding provided by the Oak Ridge National Laboratory (LDRD award 9463) for the core ExaTN library development efforts. DL, DC, and AM would like to acknowledge funding by the US Department of Energy Office of Basic Energy Sciences Quantum Information Science award ERKCG13/ERKCG23.

ACKNOWLEDGMENTS

This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.

LICENSES AND PERMISSIONS

REFERENCES

1. Schollwöck U. The density-matrix renormalization group in the age of matrix product states. *Ann Phys*. (2011) 326:96–192. doi: 10.1016/j.aop.2010.09.012
2. Orús R. A practical introduction to tensor networks: Matrix product states and projected entangled pair states. *Ann Phys*. (2014) 349:117–58. doi: 10.1016/j.aop.2014.06.013
3. Oseledets IV. Tensor-train decomposition. *SIAM J Sci Comput*. (2011) 33:2295–317. doi: 10.1137/090752286
4. Verstraete F, Cirac JI. Valence-bond states for quantum computation. *Phys Rev A*. (2004) 70:060302. doi: 10.1103/PhysRevA.70.060302
5. Shi YY, Duan LM, Vidal G. Classical simulation of quantum many-body systems with a tree tensor network. *Phys Rev A*. (2006) 74:022320. doi: 10.1103/PhysRevA.74.022320
6. Vidal G. Class of quantum many-body states that can be efficiently simulated. *Phys Rev Lett*. (2008) 101:110501. doi: 10.1103/PhysRevLett.101.110501
7. Chan GKL, Keselman A, Nakatani N, Li Z, White SR. Matrix product operators, matrix product states, and *ab initio* density matrix renormalization group algorithms. *J Chem Phys*. (2016) 145:014102. doi: 10.1063/1.4955108
8. Nakatani N, Chan GKL. Efficient tree tensor network states (TTNS) for quantum chemistry: generalization of the density matrix renormalization group algorithm. *J Chem Phys*. (2013) 138:134113. doi: 10.1063/1.4798639
9. Markov IL, Shi Y. Simulating quantum computation by contracting tensor networks. *SIAM J Comput*. (2008) 38:963–81. doi: 10.1137/050644756
10. Villalonga B, Boixo S, Nelson B, Henze C, Rieffel E, Biswas R, et al. A flexible high-performance simulator for verifying and benchmarking quantum circuits implemented on real hardware. *NPJ Quant Inform*. (2019) 5:1–16. doi: 10.1038/s41534-019-0196-1
11. Villalonga B, Lyakh D, Boixo S, Neven H, Humble TS, Biswas R, et al. Establishing the quantum supremacy frontier with a 281 pflop/s simulation. *Quant Sci Technol*. (2020) 5:034003. doi: 10.1088/2058-9565/ab7eeb
12. McCaskey A, Dumitrescu E, Chen M, Lyakh D, Humble T. Validating quantum-classical programming models with tensor network simulations. *PLoS ONE*. (2018) 13:e0206704. doi: 10.1371/journal.pone.0206704
13. Zhou Y, Stoudenmire EM, Waintal X. What limits the simulation of quantum computers? *Phys Rev X*. (2020) 10:041038. doi: 10.1103/PhysRevX.10.041038
14. Pang Y, Hao T, Dugad A, Zhou Y, Solomonik E. Efficient 2D tensor network simulation of quantum systems. In: *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. (2020). p. 1–14. doi: 10.1109/SC41405.2020.00018
15. Noh K, Jiang L, Fefferman B. Efficient classical simulation of noisy random quantum circuits in one dimension. *Quantum*. (2020) 4:318. doi: 10.22331/q-2020-09-11-318
16. Holmes A, Matsuura AY. Efficient quantum circuits for accurate state preparation of smooth, differentiable functions. In: *2020 IEEE International Conference on Quantum Computing and Engineering (QCE)*. (2020). p. 169–79. doi: 10.1109/QCE49297.2020.00030
17. Song Q, Ge H, Caverlee J, Hu X. Tensor completion algorithms in big data analytics. *ACM Trans Knowl Discov Data*. (2019) 13:1–48. doi: 10.1145/3278607
18. Gao ZF, Cheng S, He RQ, Xie ZY, Zhao HH, Lu ZY, et al. Compressing deep neural networks by matrix product operators. *Phys Rev Res*. (2020) 2:023300. doi: 10.1103/PhysRevResearch.2.023300
19. Stoudenmire E, Schwab DJ. Supervised learning with tensor networks. In: *Proceedings of the 30th International Conference on Neural Information Processing Systems*. Barcelona (2016). p. 4806–14.
20. Reyes J, Stoudenmire EM. A multi-scale tensor network architecture for classification and regression. *arXiv[Preprint].arXiv:2001.08286*. (2020). doi: 10.48550/arXiv.2001.08286
21. Evenbly G. Number-state preserving tensor networks as classifiers for supervised learning. *arXiv[Preprint].arXiv:190506352*. (2019). doi: 10.48550/arXiv.1905.06352
22. Martyn J, Vidal G, Roberts C, Leichenauer S. Entanglement and tensor networks for supervised image classification. *arXiv[preprint].arXiv:200706082*. (2020). doi: 10.48550/arXiv.2007.06082
23. Wall ML, Abernathy MR, Quiroz G. Generative machine learning with tensor networks: benchmarks on near-term quantum computers. *Phys Rev Res*. (2021) 3:023010. doi: 10.1103/PhysRevResearch.3.023010
24. Psarras C, Karlsson L, Bientinesi P. The landscape of software for tensor computations. *arXiv[Preprint].arXiv:210313756*. (2021). doi: 10.48550/arXiv.2103.13756
25. Fishman M, White SR, Stoudenmire EM. The ITensor software library for tensor network calculations. *arXiv[Preprint].arXiv:200714822*. (2020). doi: 10.48550/arXiv.2007.14822
26. Evenbly G. TensorTrace: an application to contract tensor networks. *arXiv:191102558*. (2019). doi: 10.48550/arXiv.1911.02558
27. Pfeifer RNC, Evenbly G, Singh S, Vidal G. NCON: a tensor network contractor for MATLAB. *arXiv[Preprint].arXiv:14020939*. (2015). doi: 10.48550/arXiv.1402.0939
28. Hauschild J, Pollmann F. Efficient numerical simulations with Tensor Networks: tensor Network Python (TeNPy). *SciPost Phys Lect Notes*. (2018) 5. doi: 10.21468/SciPostPhysLectNotes.5. Available online at: https://scipost.org/SciPostPhysLectNotes.5/pdf
29. Solomonik E, Matthews D, Hammond J, Demmel J. Cyclops tensor framework: reducing communication and eliminating load imbalance in massively parallel contractions. In: *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, IPDPS '13*. Boston, MA: IEEE Computer Society (2013). p. 813–24. doi: 10.1109/IPDPS.2013.112
30. Levy R, Solomonik E, Clark BK. Distributed-memory DMRG via sparse and dense parallel tensor contractions. In: *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. (2020). p. 1–14. doi: 10.1109/SC41405.2020.00028
31. Ma L, Ye J, Solomonik E. AutoHOOT: Automatic high-order optimization for tensors. In: *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques, PACT '20*. New York, NY: Association for Computing Machinery (2020). p. 125–37. doi: 10.1145/3410463.3414647
32. Gray J. quimb: a python library for quantum information and many-body calculations. *J Open Source Softw*. (2018) 3:819. doi: 10.21105/joss.00819
33. Rocklin M. Dask: parallel computation with blocked algorithms and task scheduling. In: *Proceedings of the 14th Python in Science Conference*. Vol. 130. Austin, TX: Citeseer (2015). p. 136. doi: 10.25080/Majora-7b98e3ed-013
34. Bradbury J, Frostig R, Hawkins P, Johnson MJ, Leary C, Maclaurin D, et al. *JAX: Composable Transformations of Python+NumPy Programs*. (2018). Available online at: https://github.com/google/jax
35. Kossaifi J, Panagakis Y, Anandkumar A, Pantic M. TensorLy: tensor learning in python. *J Mach Learn Res*. (2019) 20:1-6. doi: 10.5555/3322706.3322732
36. Roberts C, Milsted A, Ganahl M, Zalcman A, Fontaine B, Zou Y, et al. TensorNetwork: a library for physics and machine learning. *arXiv[Preprint].arXiv:190501330*. (2019). doi: 10.48550/arXiv.1905.01330
37. Lyakh DI, McCaskey AJ, Nguyen T. *ExaTN: Exascale Tensor Networks*. (2018-2022). Available online at: https://github.com/ORNL-QCI/exatn.git
38. Gray J, Kourtis S. Hyper-optimized tensor network contraction. *Quantum*. (2021) 5:410. doi: 10.22331/q-2021-03-15-410
39. Kalachev G, Panteleev P, Yung MH. Recursive multi-tensor contraction for XEB verification of quantum circuits. *arXiv[Preprint].arXiv:210805665*. (2021). doi: 10.48550/arXiv.2108.05665
40. Karypis G, Kumar V. Multilevel algorithms for multi-constraint graph partitioning. In: *SC '98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*. Dallas, TX: ACM; IEEE (1998). p. 28. doi: 10.1109/SC.1998.10018
41. Dmitry I Lyakh. *TAL-SH: Tensor Algebra Library for Shared-Memory Platforms*. (2014–2022). Available online at: https://github.com/DmitryLyakh/TAL_SH
42. Lyakh DI. An efficient tensor transpose algorithm for multicore CPU, Intel Xeon Phi, and NVidia Tesla GPU. *Comput Phys Commun*. (2015) 189:84–91. doi: 10.1016/j.cpc.2014.12.013
43. Hynninen AP, Lyakh DI. cutt: A high-performance tensor transpose library for cuda compatible gpus. *arXiv[Preprint].arXiv:170501598*. (2017). doi: 10.48550/arXiv.1705.01598
44. Troyer M, Wiese UJ. Computational complexity and fundamental limitations to fermionic quantum Monte Carlo simulations. *Phys Rev Lett*. (2005) 94:170201. doi: 10.1103/PhysRevLett.94.170201

45. White SR. Density matrix formulation for quantum renormalization groups. *Phys Rev Lett*. (1992) 69:2863–6. doi: 10.1103/PhysRevLett.69.2863
46. Schollwöck U. The density-matrix renormalization group. *Rev Modern Phys*. (2005) 77:259–315. doi: 10.1103/RevModPhys.77.259
47. Cirac JI, Verstraete F. Renormalization and tensor product states in spin chains and lattices. *J Phys A*. (2009) 42:504004. doi: 10.1088/1751-8113/42/50/504004
48. Orús R. Advances on tensor network theory: symmetries, fermions, entanglement, and holography. *Eur Phys J B*. (2014) 87:280. doi: 10.1140/epjb/e2014-50502-9
49. Vidal G. Entanglement renormalization. *Phys Rev Lett*. (2007) 99:220405. doi: 10.1103/PhysRevLett.99.220405
50. Vidal G. Efficient classical simulation of slightly entangled quantum computations. *Phys Rev Lett*. (2003) 91:147902. doi: 10.1103/PhysRevLett.91.147902
51. White SR, Feiguin AE. Real-time evolution using the density matrix renormalization group. *Phys Rev Lett*. (2004) 93:076401. doi: 10.1103/PhysRevLett.93.076401
52. Daley AJ, Kollath C, Schollwöck U, Vidal G. Time-dependent density-matrix renormalization-group using adaptive effective Hilbert spaces. *J Stat Mech*. (2004) 2004:P04005. doi: 10.1088/1742-5468/2004/04/P04005
53. Vidal G. Classical simulation of infinite-size quantum lattice systems in one spatial dimension. *Phys Rev Lett*. (2007) 98:070201. doi: 10.1103/PhysRevLett.98.070201
54. Evenbly G, Vidal G. Tensor network renormalization. *Phys Rev Lett*. (2015) 115:180405. doi: 10.1103/PhysRevLett.115.180405
55. Nishino T, Okunishi K. Corner transfer matrix renormalization group method. *J Phys Soc Jpn*. (1996) 65:891–4. doi: 10.1143/JPSJ.65.891
56. Pfeifer RNC, Haegeman J, Verstraete F. Faster identification of optimal contraction sequences for tensor networks. *Phys Rev E*. (2014) 90:033315. doi: 10.1103/PhysRevE.90.033315
57. Li X, Parrish RM, Liu F, Kokkila Schumacher SIL, Martínez TJ. An *ab initio* exciton model including charge-transfer excited states. *J Chem Theory Comput*. (2017) 13:3493–504. doi: 10.1021/acs.jctc.7b00171
58. Sisto A, Glowacki DR, Martinez TJ. *Ab initio*. nonadiabatic dynamics of multichromophore complexes: a scalable graphical-processing-unit-accelerated exciton framework. *Acc Chem Res*. (2014) 47:2857–66. doi: 10.1021/ar500229p
59. Parrish RM, Hohenstein EG, McMahon PL, Martínez TJ. Quantum computation of electronic transitions using a variational quantum eigensolver. *Phys Rev Lett*. (2019) 122:230401. doi: 10.1103/PhysRevLett.122.230401
60. Nguyan T, Lyakh D, Dumitrescu E, Clark D, Larkin J, McCaskey A. Tensor network quantum virtual machine for simulating quantum circuits at exascale. *arXiv [Preprint]*. (2021). arXiv: 2104.10523. doi: 10.48550/ARXIV.2104.10523
61. McCaskey AJ, Lyakh DI, Dumitrescu EF, Powers SS, Humble TS. XACC: a system-level software infrastructure for heterogeneous quantum–classical computing. *Quant Sci Technol*. (2020) 5:024002. doi: 10.1088/2058-9565/ab6bf6
62. Schutski R, Khakhulin T, Oseledets I, Kolmakov D. Simple heuristics for efficient parallel tensor contraction and quantum circuit simulation. *Phys Rev A*. (2020) 102:062614. doi: 10.1103/PhysRevA.102.062614
63. Arute F, Arya K, Babbush R, Bacon D, Bardin JC, Barends R, et al. Quantum supremacy using a programmable superconducting processor. *Nature*. (2019) 574:505–10. doi: 10.1038/s41586-019-1666-5
64. Hrinchuk O, Khrulkov V, Mirvakhabova L, Orlova E, Oseledets I. Tensorized embedding layers for efficient model compression. *arXiv[Preprint].arXiv:190110787*. (2020). doi: 10.18653/v1/2020.findings-emnlp.436

Conflict of Interest: AM is currently employed by NVIDIA Corporation.

The remaining authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's Note: All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher.

Advantages of publishing in Frontiers

OPEN ACCESS
Articles are free to read for greatest visibility and readership

FAST PUBLICATION
Around 90 days from submission to decision

HIGH QUALITY PEER-REVIEW
Rigorous, collaborative, and constructive peer-review

TRANSPARENT PEER-REVIEW
Editors and reviewers acknowledged by name on published articles

Frontiers
Avenue du Tribunal-Fédéral 34
1005 Lausanne | Switzerland

Visit us: www.frontiersin.org
Contact us: frontiersin.org/about/contact

REPRODUCIBILITY OF RESEARCH
Support open data and methods to enhance research reproducibility

DIGITAL PUBLISHING
Articles designed for optimal readership across devices

FOLLOW US
@frontiersin

IMPACT METRICS
Advanced article metrics track visibility across digital media

EXTENSIVE PROMOTION
Marketing and promotion of impactful research

LOOP RESEARCH NETWORK
Our network increases your article's readership