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 $T^{abc...}_{ijk...}$ 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

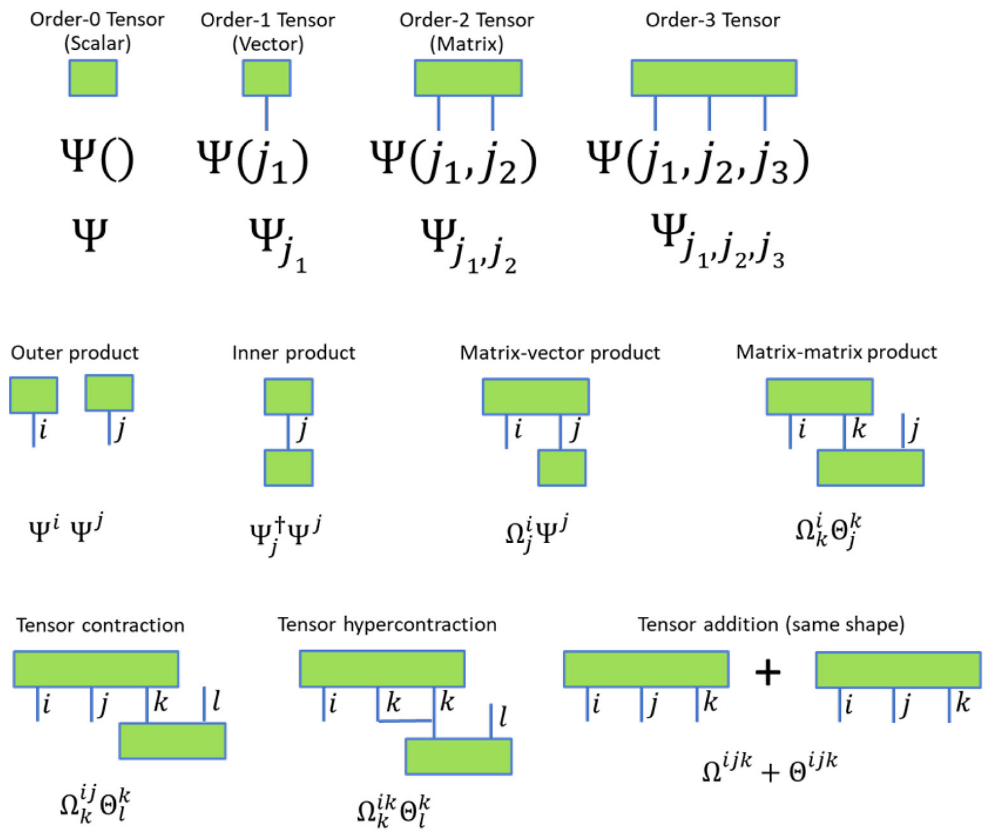

FIGURE 1 | Graphical diagrams representing tensors and tensor operations.

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.

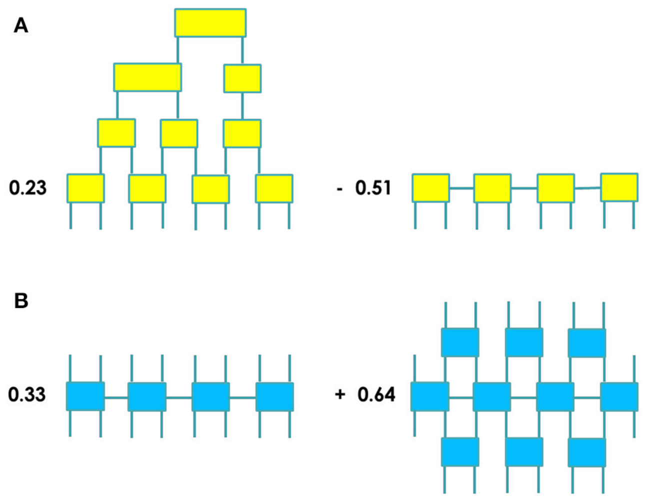

FIGURE 2 | (A) An example of a tensor network expansion as a linear combination of two tensor networks of the same order; **(B)** An example of a tensor network operator as a linear combination of two tensor networks with open legs establishing a map between the ket and bra spaces.

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.

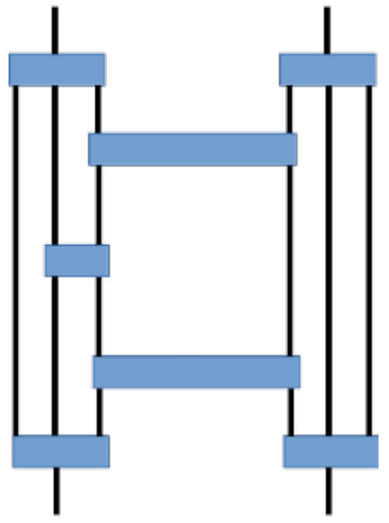
FIGURE 3 | Graphical diagram depicting a fragment of the 3:1 MERA tensor network.

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 $\sim$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,

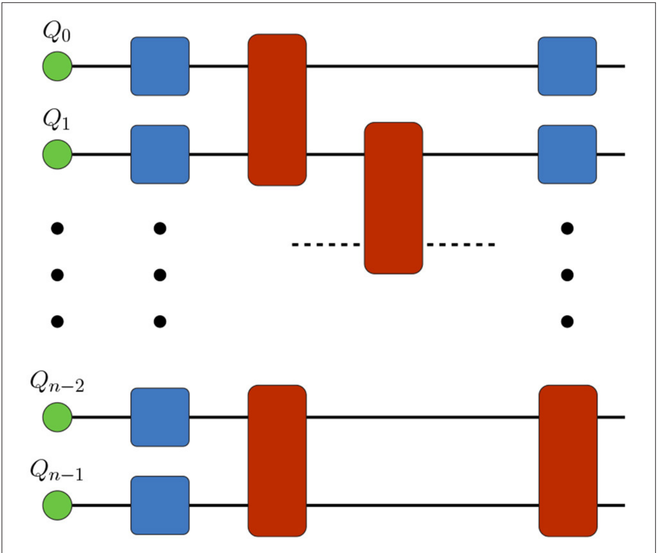

FIGURE 4 | Quantum circuit simulation by direct contraction of the circuit tensor network with ExaTN. Qubits are represented as order-1 tensors. Tensors describing quantum gates are appended to the qubit register according to the quantum circuit specification. Full contraction of the tensor network produces the complete wave-function.

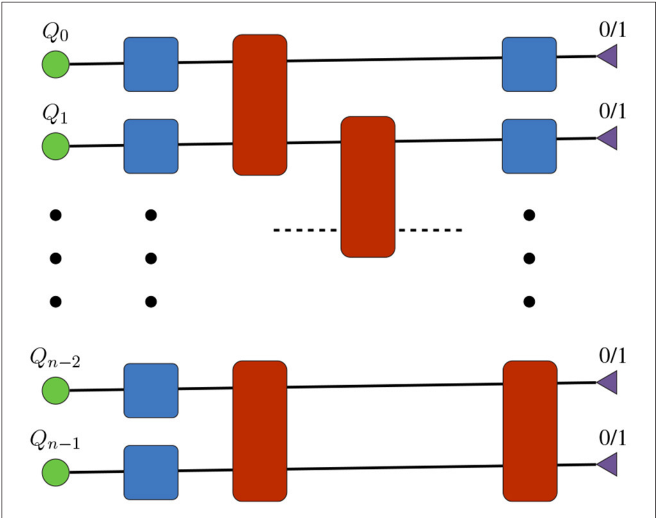

FIGURE 5 | ExaTN tensor network for bit-string amplitude computation. The tensor network representing the input quantum circuit is constructed as described in **Figure 4**. Triangular-shaped order-1 tensors are 0/1 tensors, representing the projected state.

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

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

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.

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,

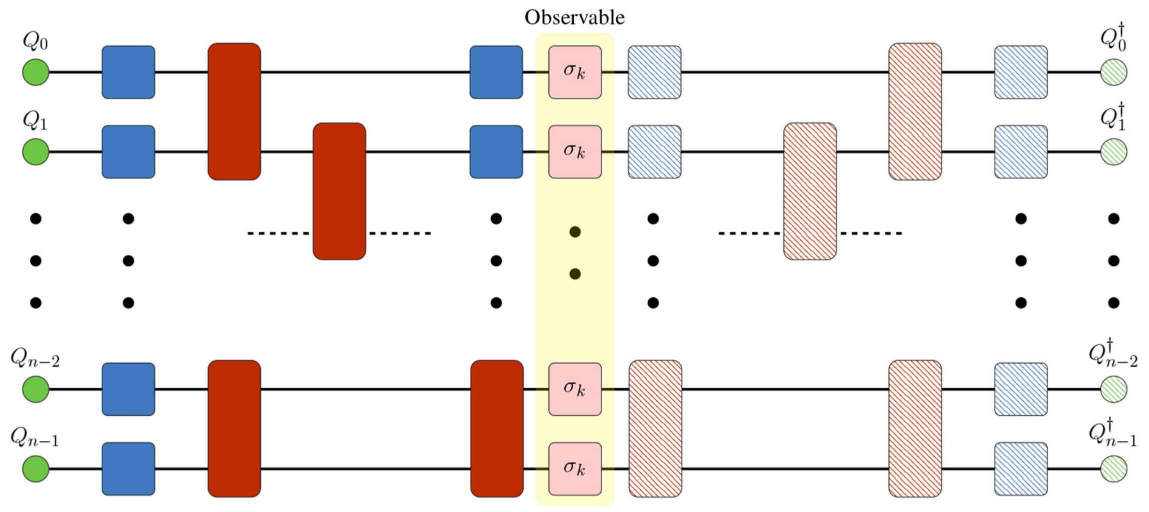

FIGURE 6 | ExaTN tensor network for expectation value evaluation using the circuit conjugation technique. Hashed-filled tensors represent the complex conjugates of the solid-filled counterparts.

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

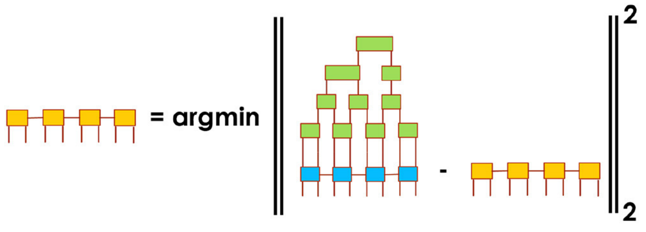

FIGURE 7 | An example of reconstruction of one (more complex) tensor network as another (simpler) tensor network by minimizing the Euclidean norm of the difference.

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

This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

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.

Copyright © 2022 Lyakh, Nguyen, Claudino, Dumitrescu and McCaskey. This is an open-access article distributed under the terms of the Creative Commons Attribution License (CC BY). The use, distribution or reproduction in other forums is permitted, provided the original author(s) and the copyright owner(s) are credited and that the original publication in this journal is cited, in accordance with accepted academic practice. No use, distribution or reproduction is permitted which does not comply with these terms.