frontiers | Frontiers in Physics

TYPE Original Research
PUBLISHED 05 August 2022
DOI 10.3389/fphy.2022.940293

OPEN ACCESS

EDITED BY
Alexander McCaskey,
Nvidia, United States

REVIEWED BY
Andy C. Y. Li,
Fermi National Accelerator Laboratory (DOE), United States
Yongjian Han,
University of Science and Technology of China,

*CORRESPONDENCE
Thomas Lubinski,
tlubinski1@gmail.com

SPECIALTY SECTION
This article was submitted to Quantum Engineering and Technology, a section of the journal Frontiers in Physics

RECEIVED 10 May 2022
ACCEPTED 27 June 2022
PUBLISHED 05 August 2022

CITATION
Lubinski T, Granade C, Anderson A, Geller A, Roetteler M, Petrenko A and Heim B (2022), Advancing hybrid quantum–classical computation with real-time execution.
Front. Phys. 10:940293.
doi: 10.3389/fphy.2022.940293

Advancing hybrid quantum–classical computation with real-time execution

Thomas Lubinski[1]*, Cassandra Granade[2], Amos Anderson[1], Alan Geller[2], Martin Roetteler[2], Andrei Petrenko[1] and Bettina Heim[2]

[1]Quantum Circuits Inc, New Haven, CT, United States, [2]Microsoft Corporation, Quantum Architectures and Computation Group, Redmond, WA, United States

The use of mid-circuit measurement and qubit reset within quantum programs has been introduced recently and several applications demonstrated that perform conditional branching based on these measurements. In this work, we go a step further and describe a next-generation implementation of classical computation embedded within quantum programs that enables the real-time calculation and adjustment of program variables based on the mid-circuit state of measured qubits. A full-featured Quantum Intermediate Representation (QIR) model is used to describe the quantum circuit including its embedded classical computation. This integrated approach eliminates the need to evaluate and store a potentially prohibitive volume of classical data within the quantum program in order to explore multiple solution paths. It enables a new type of quantum algorithm that requires fewer round-trips between an external classical driver program and the execution of the quantum program, significantly reducing computational latency, as much of the classical computation can be performed during the coherence time of quantum program execution. We review practical challenges to implementing this approach along with developments underway to address these challenges. An implementation of this novel and powerful quantum programming pattern, a random walk phase estimation algorithm, is demonstrated on a physical quantum computer with an analysis of its benefits and feasibility as compared to existing quantum computing methods.

KEYWORDS

quantum intermediate representation, quantum algorithm, hybrid quantum computing, hybrid quantum program, advanced quantum program, quantum-classical computation, quantum circuit, phase estimation

1 Introduction

Over the past decade, quantum computers have become more advanced and accessible to users. Quantum applications are theoretically capable of addressing a limited set of computational challenges in an exponentially accelerated time frame [1]. Quantum computing research has resulted in hundreds of algorithms shown to function on near-term quantum computing systems [2–4]. Recent work has shown that these algorithms offer the potential for quantum advantage over classical computing in specific domains [1, 5].

Progress towards quantum advantage has been hindered, however, by significant challenges in the noisy intermediate-scale quantum computing regime (*a.k.a.* NISQ) [2, 6]. Current generation gate-model devices are restricted to a small number of qubits, and can only execute a limited number of instructions before "noise" or gate error dominates. To address these limitations, creative algorithms such as VQE and QAOA have been developed that take advantage of quantum and classical devices working in tandem. In these hybrid approaches, a classical computer program iteratively invokes a quantum processor to execute a small part of the algorithm (with some exponential speedup). Bits of problem data are passed to the quantum processor and results returned to the classical processor, which makes decisions about the next batch of quantum instructions to execute and assembles a solution from parts.

However, there is a fundamental limit to how well this approach to hybridizing quantum and classical computing can scale. Treating quantum and classical processors as disjoint physical instruments, each with their own data transfer pipeline and computational interface, constrains hybrid algorithms to repeatedly switching between contexts and exchanging intermediate data between devices. This high latency means classical decisions can not be made to influence evolution of quantum state before qubits decohere.

In this paper, we describe a new class of hybrid program in which elements of classical computation are embedded directly within the quantum program and execute in the same time domain as the quantum operations. This approach delivers a compelling advantage, reducing latency of data exchange by orders of magnitude and providing flexibility in controlling the quantum state during execution. We delineate the characteristics of this new type of hybrid quantum/classical computation, the hardware and software required to enable it, and the opportunities it affords.

This capability requires a quantum computer that is able to execute a series of quantum operations commingled with some (classical) computation that uses the results of previous operations to compute new results that may affect the next iteration or series of quantum operations. The most important aspect of this implementation is that the classical computation is performed *without terminating execution of the quantum program and discarding the qubit state* or returning to the classical computer for those computations. Many small classically driven adjustments to the quantum state can be made based on measurements performed in the middle of the program, resulting in a program that is adaptive in nature.

Unlike a classical computer with its programming constructs such as variables, arithmetic computation and looping, a quantum computer is typically implemented using highly specialized hardware and firmware optimized to generate complex sequences of high-resolution microwave or laser pulses to manipulate sensitive and fragile quantum states. These control systems have only a limited ability to perform integrated classical computation under tight time constraints [7, 8]. We will discuss enhancements that may be necessary to fully enable this new form of hybrid program.

This paper demonstrates a first step towards fully general, tightly integrated quantum/classical processing, enabling new types of quantum algorithms that have not been possible on prior generations of hardware. This is not only an interesting capability in and of itself, but also provides an impetus for the community to fundamentally rethink what a quantum algorithm can look like and to go beyond the limitations in current quantum algorithms.

The remainder of this paper is structured as follows. In Section 2 we describe the context in which our work is positioned relative to current quantum computing methods. Section 3 introduces the software development methodology that enables this new type of programming and the associated hardware challenges. In Section 4 we outline two quantum computing algorithms that are made possible with this new capability. Finally, in Section 5, we present an early implementation of one of the algorithms on a physical hardware device and review the results of its execution and associated trade-offs.

2 Background

We review here prior efforts on which our work is based. First, we examine characteristics of an established hybrid approach used in many algorithms available to users of today's quantum computers. We then consider algorithmic enhancements that take advantage of recent hardware advances such as mid-circuit measurement and reset along with near-term implementations of real-time classical computation. Taken together, these features of the current and upcoming generations of hardware describe the current state of existing hybrid quantum/classical computation.

2.1 Hybrid quantum applications

We first review two essential hybrid algorithms that interleave classical and quantum processing to reduce

resources such as overall circuit size, depth, and number of qubits as a way to work around the limited coherence time and fidelity of qubits today. However, the constraints of data transfer between classical and quantum processors pose a barrier to leveraging such schemes on sufficiently large devices. Below, we highlight several specific challenges for practical use of these hybrid algorithms.

2.1.1 Variational quantum eigensolver (VQE)

Estimating ground (and excited) state energies with accuracy ε is at the core of many quantum applications in chemistry [9–11] and materials science [12]. No efficient classical algorithms are known that run in time *poly* (log $(1/\varepsilon)$), whereas a quantum computer makes this possible in principle. With VQE, a Hamiltonian describing a physical system is simulated using iterative execution of a quantum circuit which prepares an approximate wavefunction, the so-called "ansatz," and a variational algorithm is used to upper bound its ground state energy [13]. By suitably grouping Hamiltonian terms, the family of quantum operations $U(\theta)$ used in the ansatz and the circuit that simulates Hamiltonian terms can be chosen to have a low number of entangling gates, which is advantageous for execution on near-term quantum devices.

VQE is inherently a hybrid algorithm, as classical code is used to optimize the parameter vector θ in the ansatz $U(\theta)$ by applying a classical method such as gradient descent, SPSA [14], or the gradient-free Nelder–Mead method, while the quantum computer is used to evaluate a cost function (the sum of estimated Pauli expectations provides an approximate upper bound of the minimum energy). Classical computations alternate with quantum computations so that the quantum state does not need to stay coherent while classical optimization takes place. An advantage of this hybrid method over other quantum algorithms is the trade-off of circuit size against total number of repetitions necessary to reach a target accuracy of ε.

2.1.2 Quantum approximate optimization algorithms (QAOA)

Optimization problems such as MAX-CUT are among the most important tasks addressed by classical methods. In order to find solutions more efficiently with quantum computing, we can use the QAOA algorithm [15] by expressing optimization problems in terms of finding the highest energy configuration of a spin Hamiltonian that is diagonal in the computational basis. A variational algorithm is then used to optimize, with suitable classical optimization methods, weights applied to a quantum circuit consisting of alternating evolution under the problem Hamiltonian and a non-diagonal mixing Hamiltonian used to move amplitude between different configurations. An additional parameter defines the number of rounds applied in the QAOA scheme and impacts depth of the resulting circuit. For a large number of rounds, the Trotter–Suzuki decomposition is a limiting case, while for a small number of rounds the method is not more powerful than classical methods [16].

Similar to VQE, this algorithm is hybrid as the classical code used to optimize the parameter vectors is interleaved with quantum processing used to evaluate the cost function. The quantum state need not stay coherent while classical optimization takes place. An advantage of this hybrid method over approaches based on the Trotter–Suzuki decomposition is to trade off depth of circuit size with quality of solution.

A common structural element of both of VQE and QAOA is the alternating execution of classical optimization code with quantum code. While this specific structure makes it possible to perform these algorithms on the current generation of quantum computers, it introduces significant challenges in minimizing the total time to solution. Several solutions have emerged to partially address these.

A non-negligible time is required to compose a quantum circuit and modify its parameters prior to each iteration. Some quantum software frameworks [17–19] support "parameter" values as arguments to quantum operations. A quantum circuit may be created using parameters instead of fixed values and is compiled once with these symbolic values. Prior to each execution, current parameter values are injected by a second compiler pass or in the backend system, resulting in a reduction in time consumed by program composition.

Compounding this is latency involved in initiating execution of a quantum circuit and communication between the classical and quantum computers [20], especially pronounced when using a cloud computing service. Rigetti's quantum–classical cloud platform was one of the first systems to offer execution of classical code on a system physically co-located with the quantum system, considerably reducing data transfer latency [21]. The Qiskit Runtime system [22, 23] and Quantum Serverless [24] take this a step farther, co-locating the classical and quantum systems, but also executing all iterations of an iterative application as a single job, eliminating lengthy queuing times introduced when many users share a single system.

Both of these approaches offer faster near-time execution, albeit not real-time execution of classical code while the quantum state remains coherent (terminology introduced by IBM Research [25]); see Figure 1 for a schematic comparison between near-time and real-time quantum execution. A significant latency still remains in the time taken to initialize the quantum circuit and the transfer of data between the classical processor and the quantum processor. To bring execution times down and get closer to real-time, more advanced methods are needed.

2.2 Mid-circuit measurement

In a recent innovation, both IBM [26, 27] and Quantinuum [28, 29] have shown ways to perform measurement in the middle

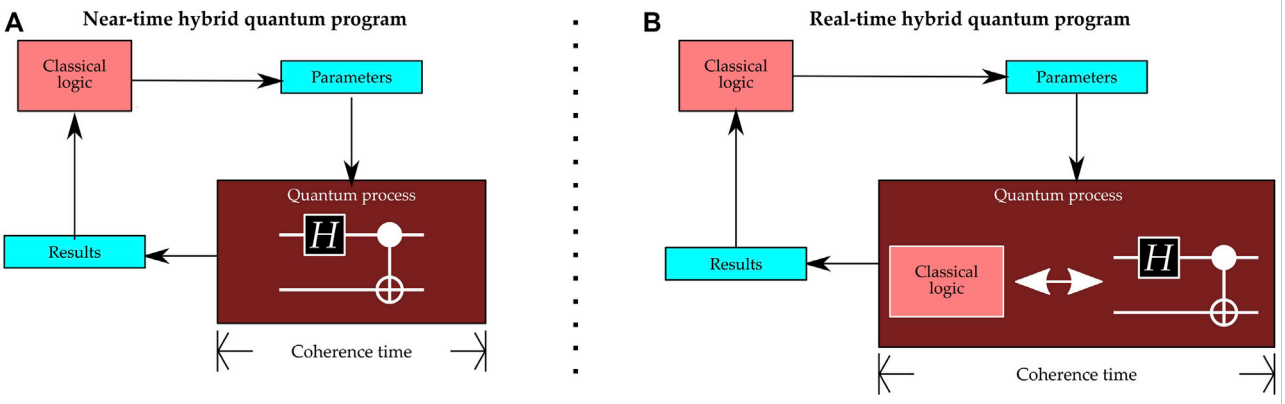

FIGURE 1
A schematic comparison between near-time hybrid programs **(A)** and the real-time hybrid quantum programs considered and implemented in this work **(B)**. Importantly, in the more advanced form we consider here, hybrid quantum–classical programs can make classical decisions based on the results of quantum measurements, and then use those decisions to condition and control future quantum operations within the coherence times of quantum registers.

of executing a quantum circuit, with several benefits. One, a measurement can be taken and qubit state reset afterwards, effectively enabling the reuse of qubits. This facilitates the implementation of algorithms that require fewer qubits by using an ancilla that provides state that is carried forward. An array of the individual measurements can be returned and additional analysis performed by a classical driver program.

Two, each measurement can trigger branching to logically different parts of a circuit. This permits more complex algorithms to be encoded within a single execution of a quantum program. However, the use of branching alone requires that all subsequent computational paths be delineated in the circuit, either in code or using a lookup table, an approach which can grow exponentially with the number of branch points. While this offers an improvement in program control, the benefit is lost with larger problems. Nonetheless, there are a number of algorithms, a few described below, that are able take advantage of the mid-circuit measurement capability to implement programs that have lower depth and require fewer qubits.

2.2.1 Repeat-until-success methods (RUS)

Paetznick and Svore [30] introduced "repeat-until-success" (RUS) circuits, a method that is useful for the ε-approximate synthesis of unitaries over basic gate sets such as the Clifford + T set. The targeted unitary is implemented with a probability p which constitutes the "success" case and fails with probability $1 - p$ which results in the given state being affected by a Clifford gate. This leads to a branching on a classical bit indicating success: if the successful branch is taken, the computation continues and the next gate is applied, if unsuccessful, the affected Clifford gate is undone and another attempt is made for the same gate (up to a maximum since coherence time is limited). It has been shown [31, 32] that RUS circuits can lower the expected cost of approximating R_z rotations from $4\log_2(\varepsilon)$ for deterministic methods [33, 34] to $c\log_2(\varepsilon)$, where the constant c is independent of the rotation angle and $c \approx 1$. See also Kliuchnikov et al. [35] for a recent overview of synthesis methods, including probabilistic methods such as RUS.

2.2.2 Other iterative algorithms

Other quantum algorithms are iterative in nature and can benefit from the interleaving of quantum/classical computation, processing of mid-circuit measurements, or both. For example, iterative phase estimation (IPE) functions with a smaller number of qubits than quantum phase estimation, its non-iterative equivalent. IPE measures and resets the state used to read out phases along the way, relying on classical processing to compute a result. The semi-classical Fourier transform differs from a normal quantum Fourier transform in that a result distribution is obtained using a single qubit that undergoes a sequence of Hadamard operations, measurements and phase corrections that depend on previous measurements [36]. This provides a useful resource tradeoff between the number of qubits required and the classical computation of phase corrections. Two other examples use hybrid computing at the quantum algorithm level: 1. the quantum sieving algorithm introduced by Kuperberg [37] uses hybrid computing to cut down the time-complexity of the dihedral hidden subgroup by re-grouping and further processing quantum registers depending on mid-circuit measurement results. 2. the quantum rejection algorithm [38] uses hybrid computing to implement non-unitary operations based on attaching auxiliary qubits followed by mid-circuit measurements and branching on the outcomes. In addition, there are phase estimation algorithms recently introduced that use a novel technique known as the quantum singular value transformation (QSVT) [39].

In all these cases, the quantum circuits make use of mid-circuit measurement and reset to get more out of the quantum processor, re-using qubits without exiting the program or using branching to adapt the circuit to changes in the quantum state. Combined with interleaving of classical and quantum processes, this represents a second level of hybrid quantum programming. However, this is only a small step towards taking full advantage of the power of the quantum computer.

2.3 Real-time classical computation

There is another level of sophistication in hybrid algorithms that is the subject of our work. Córcoles et al. [7] introduce the concept that an adaptive version of iterative phase estimation, exploiting dynamic circuits, could offer a substantial advantage when noise and latency are low. They explore the effect of conditional branching on qubit measurement and selection of a new angle from a lookup table given the measurement result. This is an important first step, as it is an improved form of branching on measurement data. However, to achieve "real-time" and scalable computation, we need a more efficient solution that does not require programming of all branch paths.

Ideally, the results of one or more measurements could be used as input to arithmetic operations that influence subsequent processing. Rather than branch to specific hard-coded subsequent operations, the rotation angles applied to gates within the program can be the result of a computation in which the angle values change every time. This is effectively an adaptive quantum program, where variables are modified as the execution progresses. Each iteration may perform a progressively refined computation that converges to an answer, re-using the set of parameterized quantum operations embedded in the program.

An early example of this capability was demonstrated in a randomized benchmarking (RB) application in Reinhold [40]. In that experiment, each shot, or repetition, of the RB circuit was comprised of a different sequence of random gates. This sequence was generated with an embedded classical co-processor, which was programmed to calculate pseudo random numbers using a linear-feedback shift register algorithm. These random numbers were then used in real-time to choose which Clifford gate to apply to the qubit. Such an approach to RB has an enormous advantage over pre-computing the entire sequence of random gates prior to execution [41] and highlights one powerful application of real-time classical computation. In another example, Ofek et al. [42] demonstrated the use of real-time feedback as a key component in Quantum Error Correction (QEC).

There are many language and hardware-independent approaches to quantum-classical programming, several of which are reviewed in Smith et al. [20], McCaskey et al. [43]. One effort underway, relevant to our discussion of hybrid quantum programming, is an enhancement of the popular OpenQASM 2.0 specification for quantum circuit definition, called OpenQASM 3.0 [44]. A quantum program defined in OpenQASM 3.0 can include classical computations as part of the definition of a quantum circuit.

Complementing this, the QIR Alliance [45] is working to develop a Quantum Intermediate Representation (QIR) that can represent such programs that may involve arbitrary interleaving of quantum and classical computation within a single program. While OpenQASM 3.0 is a human-readable representation of a quantum program, the role of QIR is to provide a format that is optimally machine-manipulable, compatible with many existing languages and compiler tools. While many of these efforts are early stage, the move to embedding classical computation within a quantum program is an important direction for the future of quantum computing.

3 Enabling a new form of hybrid program

Advancing quantum/classical computational capabilities requires an end-to-end stack where each layer has a clear and distinct purpose, as well as an arsenal of tools that enables integration with such a stack. Ideally, an application program is written in a form that makes it easy for users to represent all elements of the solution concisely and in a portable fashion. It should also be easy for providers of backend systems to convert the intermediate representation of the solution to execute on specific hardware architectures. We propose that both can and need to be accomplished by introducing a compilation stage that targets a language and hardware agnostic holistic program representation to a backend specific profile.

Below, we examine the improvements necessary in the software stack to support a comprehensive form of quantum/classical computation [20]. This includes a discussion about the proposed Quantum Intermediate Representation and a look at challenges to implementation on backend hardware systems.

3.1 The quantum software stack

To our knowledge, applications executed on quantum hardware so far have been limited by the inability to execute classical computations while the quantum state remains coherent. This is evidenced by the prevalence of algorithms such as VQE and QAOA, outlined in Section 2.1. These algorithms consist of an outer loop that alternates classical optimization of parameters and execution of a quantum cost function that uses those parameters, requiring the quantum state to remain persistent only for the duration of an iteration. Even so, the practicality of leveraging such algorithms is limited by the added latency due to the required data exchange.

Practicality can be improved by reducing compilation times using symbolic representation of parameters or minimizing latency by co-locating the classical and quantum processors. In principle, advanced multi-processor systems could achieve very low-latency communication between the classical computer and the quantum control and readout logic. However, to go beyond this and enable tightly integrated classical processing within the quantum application requires additional support throughout the entire hardware and software stack.

While several dedicated quantum programming languages have been developed [44, 46, 47], the predominant approach within the ecosystem largely relies on leveraging popular classical languages such as *Python* to generate a quantum circuit [48–51]. Such code generation or metaprogramming frameworks rely on the host language to provide the convenience and expressiveness to concisely and comprehensively articulate the program intent, and can present a comprehensive API to a quantum compiler. The actual quantum circuit is defined by invoking API calls to build a program abstraction in the form of a data structure. This intermediate data structure can be transformed and optimized by the framework before it ultimately generates native hardware instructions to be executed by the targeted quantum processor, a process that requires a significant amount of logic and sophistication.

Another approach is to enhance the semantics of a high-level language, such as C or *Python*, with syntax to specify that certain loops, variables, and arithmetic computations are to be executed within the quantum program that is produced. Recently, full-featured languages such as Q# and extended program representations such as OpenQASM 3.0 have emerged that embed these constructs directly into the language so that a user is able to program using a unified abstraction and the compiler is able to perform the necessary transformations seamlessly.

With any approach, support for classical processing while qubits remain live requires representing the logic for data exchange and processing within an integrated intermediate program representation. Whether the quantum application is expressed using a domain specific language or a metaprogramming framework, both its program abstraction and its intermediate representation must not be limited to capturing merely quantum operations but need to include classical computation and control flow as well.

3.2 Quantum Intermediate representation

Challenges related to maximizing utility of a dedicated accelerator working in concert with a central processing unit are not unique to quantum computing. The use of GPUs in modern computing inspired strategies for data exchange between different processors and for facilitating code portability and integration with existing tools and technologies. Quantum processors, however, are early in their development and to promote and accelerate innovation it is crucial that we do not standardize on a representation for quantum programs that is specific to a particular backend or default to a least common denominator approach to deal with diverse hardware technologies.

To address this challenge in quantum computing, and specifically for the demonstration in this paper, we identified these goals for an effective quantum intermediate representation:

1. Reduce and accelerate the development effort for software frontends and hardware backends.
2. Permit application and library code to take advantage of novel and unique backend capabilities while maintaining code portability and interoperability.
3. Enable incremental progress in how different subprocessors or processor components interact and communicate.

A quantum program written in a high-level language is compiled to an intermediate representation that can be executed on a variety of backend systems. While quantum computing may be unique in many regards, a large part of the required functionality to leverage advanced compilation techniques is not. To accelerate advancement in quantum, our efforts take advantage of the decades of experience at our disposal on program and dependency analysis, powerful tools for code transformations and optimization, as well as versatile infrastructure for linking and machine code generation.

For these reasons, we chose to build on top of an LLVM-based quantum intermediate representation (QIR). LLVM is a mature collection of modular and reusable compiler and toolchain technologies [52]. QIR is a language and hardware agnostic format that allows for full interoperability between quantum languages and libraries, rewrite steps and optimization passes, and code generation for quantum hardware. In particular, QIR builds on the design goals of LLVM IR and extends those into the quantum domain:

> LLVM is a Static Single Assignment (SSA) based representation that provides type safety, low-level operations, flexibility, and **the capability of representing 'all' high-level languages cleanly**. It is the common code representation used throughout all phases of the LLVM compilation strategy [53].

Using a common intermediate representation allows the software stack to support different source languages and execution platforms without large amounts of redundant development, to keep pace with a significant evolution of the quantum processor architecture over time. To that end, QIR is an integrated program IR representing not only quantum instructions (e.g.: gate calls and measurements), but classical

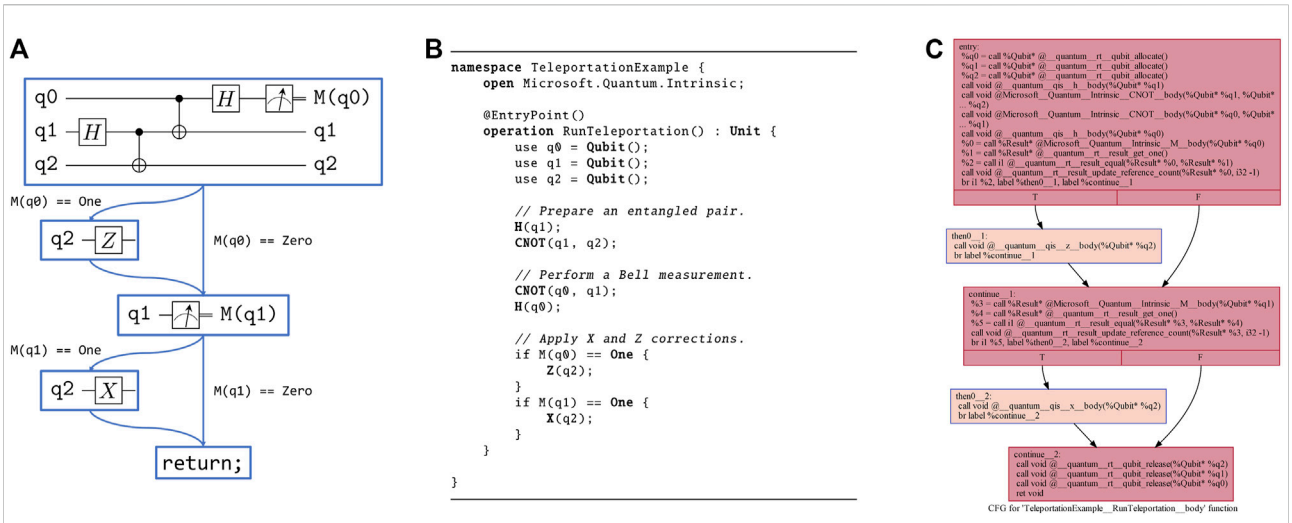

FIGURE 2
An example of compiling a simple Q# program to QIR **(A)** A schematic representing quantum teleportation as control flow between circuit-like blocks **(B)** Q# implementation of quantum teleportation **(C)** The control flow graph for the QIR generated from the Q# source code. Note that the conditional branching in the schematic at left and in the control flow graph at right is represented in the control flow graph as arrows between *blocks* of quantum and classical instructions. Within each block, quantum gates such as `__quantum__qis__x__body` can be called using the `call` instruction in QIR.

logic concepts such as branching, memory management and variables.

As an illustration of QIR, Figure 2 shows a brief example of how a simple program such as quantum teleportation can be thought of as a flow through different blocks of quantum instructions with classical branching on measurements (left pane). To the right of this is shown the equivalent Q# program and the QIR/LLVM code that is generated.

QIR specifies how to represent quantum subroutines using a subset of the LLVM IR, following a similar approach as the NVVM compiler IR [54], designed to represent GPU compute kernels. A set of QIR profiles is defined, each of which imposes additional rules that restrict the IR to contain only those constructs that will be executed on a specific QPU target.

Rather than require each frontend language to compile into a processor specific profile, we introduce a compilation stage which maps a QIR program to a targeted QIR profile. An initial implementation for this stage is provided by the Quantum Adaptor Tool (QAT) [55] with some custom tooling to map the intermediate representation to a specific backend architecture. This permits the development of hardware-targeted capabilities without needing to specialize quantum languages to depend on specific features of different devices.

The QIR profile for a specific hardware device defines its quantum gate set, its measurement capabilities, the control flow constructs and classical computations that it can reasonably support. Any program logic that cannot be reduced to leverage only the supported profile will need to be executed as pre- and post-processing steps much like the common practice today. Programs that inherently require a unique hardware feature can execute only on hardware that supports that feature, yet the choice of representation does not add portability constraints to those fundamental to the algorithm. Conversely, a hardware feature that is not represented in the QIR will not be accessible to users. A vital step towards making quantum computing practical will require agreement in the community about the operations supported in a QIR as well as the transformations and optimizations applied at compile time for both the quantum operations and the classical computations within any program.

3.3 Hardware challenges

Any quantum program, written in a user-level programming language, is typically converted to a hardware-specific set of instructions that executes on a quantum computer system. To enable quantum programs that use a new form of hybrid quantum-classical computation, we must account for the limitations that are inherent in this generation of quantum computing system and consider how these systems may evolve.

The quantum elements (or 'qubits') assembled into a quantum computing system are manipulated using classical control electronics to generate sophisticated sequences of microwave or laser pulses depending on the technology used in the system. The

nature of these quantum elements dictates that pulses on them are defined on a nanosecond timescale, and the relative timing of pulses on different quantum elements must be precisely coordinated. Furthermore, to enable any useful control of a qubit, programmable control flow that operates on the same timescale is a requirement. Since a general-purpose CPU is not sufficient for this, a quantum control system is often constructed with specialized hardware such as an arbitrary waveform generator (AWG) or a field programmable gate array (FPGA) that can be utilized to meet these stringent requirements. At scale, a quantum computer is built with many of these. Newer generations are becoming more sophisticated, for example, all on a single chip.

Unsurprisingly, the advantages of special purpose hardware come with loss of general purpose computing features including smaller instruction sets and less runtime memory. While specific tradeoffs may be redressable, we describe here some possible limitations when working with such specialized quantum control systems.

Arithmetic operations performed on such systems may be different from those of a general-purpose processor. To minimize memory usage and execution time, fixed point numerical representations are commonly used: an integer of some number of bits with an implicit decimal point. Other than division, basic arithmetic is fast, the periodic 2's complement format provides an effective representation of angles, and interpolation may be used to implement operations such as sine and cosine. However, fixed point representations are sensitive to both overflow and underflow. When not used to represent periodicity, the numeric range is quite limited, and for example, multiplication by 0.5 results in the loss of 1 bit of precision. The developer of a hybrid quantum program that includes such arithmetic operations will need to be aware of any constraints imposed by the target hardware system.

There is also a challenge in the generation of hardware level instructions for a hybrid program that includes classical computation. As long as the arguments to quantum gates are known at compile-time, a transpiler can generate hardware-specific code and pulse sequences to perform those operations efficiently. However, to enable variable arguments to quantum gates, the compiler must generate a more complex sequence of code. A common way to implement support for variable arguments on current architectures is to use a RZ basis gate since it may be implemented virtually [56], i.e., as only a phase change on subsequent pulses instead of as its own pulse. This means it has effectively perfect fidelity and has zero run-time cost. For this gate to be "virtual" but still accept variable arguments at run-time, the classical processor must support arithmetic and trigonometry.

There are many other challenges associated with this new form of quantum program that adapts its execution to changes in variable state. A program that executes a fixed series of operations returns a dataset with a predictable structure, but if the paths are modified during execution the structure of the return data can vary across executions. Another complexity stems from the fact that qubits are often manipulated in parallel, using what is essentially a network of small classical processors, and the bandwidth, latency, and connectivity of such networking could be rate limiting factors at scale. In these early stages, the introduction of classical computation to quantum programs will be constrained by these multiple challenges. Future generations of quantum control systems should take these challenges into account.

4 The path to scalable and reliable applications

With a quantum software stack that supports classical computation and a portable intermediate representation, we have the ingredients to enable a compelling advance in quantum programming. Existing hybrid algorithms integrate quantum and classical computation, but in a restricted and disjoint fashion. We break new ground here by describing a form of adaptive hybrid programming in which quantum and classical computational primitives may be tightly interwoven, rendering optional the need for quantum measurement data to be repeatedly transferred across computer interfaces.

This capability inspires development of an entirely new class of quantum algorithm, one in which reliability can be improved and the breadth of applications extended. We illustrate this idea with an example of a simple quantum algorithm that uses both quantum and classical operations, followed by a discussion of the range of features essential to making this new capability complete. Later, we describe the workings of an advanced algorithm that we execute on both a quantum simulator and on quantum hardware (results presented in Section 5).

4.1 Resetting a quantum system

To show how classical computation can be used to enhance a quantum program, we highlight an algorithm designed to reset a quantum bit from an unknown state in the shortest amount of time required to achieve a desired "fidelity". The probabilistic nature of a quantum state makes it challenging to implement a quantum reset protocol that has 100% certainty of success in a single operation [57]. Several tutorial examples [8, 58] demonstrate how multiple qubit reset operations are required to achieve a high probability of success and how the number used can affect the fidelity of the operation.

While this example is primitive, it serves to highlight the use of program variables, classical loop execution, and simple arithmetic during the time domain of quantum program execution. Algorithm 1 describes its program logic.

Algorithm 1. Active qubit reset.

```
1:  req_successes ← 2
2:  num_successes ← 0
3:  for counter ← 0, 4 do
4:      value ← measure(qubit)                ▷ measure qubit
5:      if value == 0 then
6:          num_successes ← num_successes + 1
7:      else
8:          X(qubit)                          ▷ flip qubit state
9:          num_successes ← 0
10:     end if
11:     if num_successes == req_successes then
12:         break                             ▷ done, exit loop
13:     end if
14: end for
```

The algorithm succeeds when it measures 0 twice in a row on the qubit. If not successful after 4 measurements, the loop exits and the reset has failed (not shown). Multiple variables are defined and used within a classically controlled loop and a counter is updated using simple classical arithmetic. Operations such as these can be constructed using various hardware-specific libraries [8] that support classical operations and pulse control in the same program. However, OpenQASM-level APIs used to develop portable algorithms and applications typically permit branching on mid-circuit measurement but little else that is classical in nature (Section 2.2). Our work is specifically targeting general solutions with open specifications, available for implementation across multiple target platforms.

With classical computation embedded in the quantum program, flow control logic and values applied to gate operations may be computed "on-the-fly." Performing an arithmetic computation based on captured measurement values to modify the current variable values results in a program is "adaptive," i.e. it changes its conditional behavior in response to measurements of a continually changing quantum state.

4.2 Scaling up the software stack

Several enhancements to the software stack and target hardware are required to enable this new form of hybrid programming. To illustrate, we focus on a particular model of integrated classical and quantum computation: a parameterized series of quantum operations executed repeatedly, interleaved with classical computation of the next set of variable values and control flow, with some iterations occurring while the qubits are kept coherent and the quantum state maintained. This is sufficient to allow us to run algorithms such as random walk phase estimation, described in Section 4.3 below.

To accomplish this, our implementation of a quantum/classical hybrid program supports the following programming constructs in addition to features already available:

4.2.1 Parameters and variables

In Section 2.1, we showed how existing hybrid algorithms minimize latency of circuit composition through the use of parameterization, by which angles used in quantum gates are defined symbolically and the actual values supplied at the time of execution. In these algorithms, the classical values provided at initialization, the "parameters," are used as constants and not modified during execution.

In our advanced real-time hybrid programs, we support classical values that may be changed during the course of execution. These values, or "variables," may be used as rotation angles in some systems, but could also be used as a loop counter or as computed values to be returned to a calling program. The quantum firmware and hardware is sophisticated enough to adjust the execution of the circuit to a new variable value, and to perform this adjustment during the course of its execution. Optimal performance can be achieved if the quantum program does not need to exit in order for new values to be provided.

4.2.2 Variable arithmetic

Computing new variable values within the quantum program requires sufficient arithmetic computational capability in the quantum firmware to calculate new values as a function of mid-circuit measurement results or other variables, between execution of quantum operations. The calculation, including the routing of the measurement results, must happen quickly enough so that the qubits don't decohere in the meantime.

In current hybrid algorithms, the classical code used to execute a quantum circuit and iteratively converge on a solution is typically implemented in a high-level language like *Python*, C#, or Julia. This will not work for arithmetic operations that are to be executed within the context and time domain of the quantum program. Instead, these instructions need to be converted to low-level assembly or bit codes that can execute in the control system (FPGA or other) and must be synchronized with quantum code execution, requiring advanced control features not exposed in most of today's quantum computers.

4.2.3 Conditional looping

Running the parameterized circuit repeatedly (looping) requires support in the high-level quantum software for defining the body of the loop (which may include quantum and classical parts) and specifying the loop exit condition (either a variable or a direct measurement). The upper levels of the software stack, such as the compiler and the intermediate representation, must be capable of recognizing variables, loop constructs, and arithmetic computations that will be executed within the quantum control system, and generating the associated low-level firmware instructions in the assembly language specific to a target hardware system.

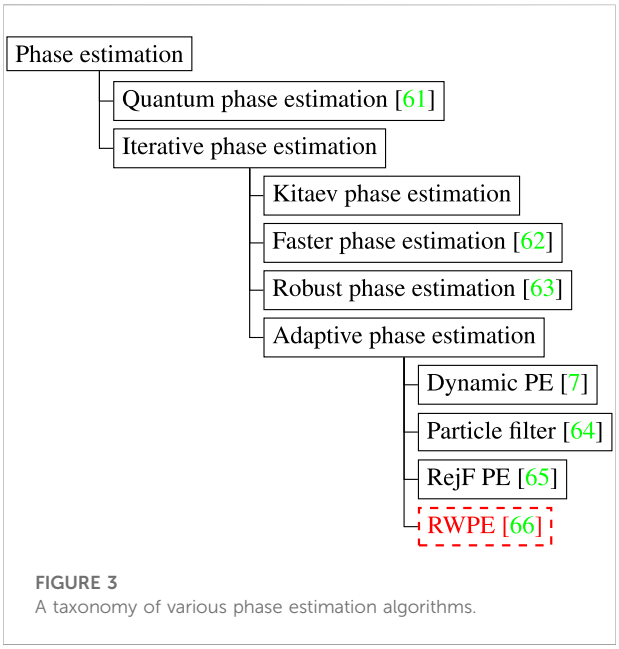

FIGURE 3
A taxonomy of various phase estimation algorithms.

Taken together, these features make possible a new class of hybrid quantum/classical program that can exploit the full potential of both types of computing hardware working in tandem. The integrated quantum program used in our demonstration below was implemented in the Q# programming language [59] and made use of the Quantum Intermediate Representation (QIR) described in Section 3.2. The Q# compiler follows the approach described in QAT documentation [55] to separate the classical and quantum portions of the QIR and generate the instructions required by the backend hardware. These represent the first set of programming tools capable of representing this advanced form of hybrid computation.

4.3 Random walk phase estimation

For our demonstration, we focus on the example problem of quantum phase estimation. A variety of algorithms are available to determine the phase inherent in a given quantum operation, as shown in the summary in Figure 3. We contrast several of these algorithms with our approach, the RWPE algorithm, that uses the new form of hybrid quantum/classical computation introduced in this paper.

Consider the problem in which have a particular quantum subroutine U whose action is represented by the unitary U whose eigenvalues we would like to learn. For instance, in quantum chemistry, U may be a step in an algorithm to simulate the Hamiltonian of a given chemical system such that the eigenvalues of U represent energy levels of that system.

In the case that $U = e^{iH\tau}$ for some Hamiltonian of interest H and for some time interval τ, then one can naïvely approach the problem of finding the minimum eigenvalue E_0 of H by rephrasing as a minimization problem

$$E_0 = \min_{\vec{x}} \langle \psi(\vec{x}) | H | \psi(\vec{x}) \rangle, \tag{1}$$

for some parameterized set of state preparations $|\vec{x}\rangle$ known as an *ansatz*. Approximating this expectation from measurements of a quantum device yields the variational quantum eigensolver (VQE) algorithm [13] described in Section 2.1.

VQE has a number of limitations that make it difficult to apply for larger problems. In particular, the state preparation ansatz must be called repeatedly as the measurement implied by (1) consumes $O(1/\epsilon^2)$ copies of the state prepared by the ansatz operation at each iteration in order to reach an accuracy of ϵ [60]. On the other hand, if we can prepare a register of qubits in an eigenstate $|\phi\rangle$ of U with eigenvalue $e^{i\phi}$, such as by using adiabatic state preparation, then we can use phase estimation to learn ϕ.

In particular, quantum phase estimation (QPE) uses the inverse quantum Fourier transform (QFT) to prepare a register in the state $|b_0 b_1 \ldots b_{n-1}\rangle$ where $\phi = 0.b_0 b_1 \ldots b_{n-1}$ is an expression of ϕ as a fixed-point binary number using n classical bits [61]. To avoid the need for an additional register of n auxiliary qubits, iterative phase estimation (IPE) methods improve upon QPE by estimating ϕ using one classical bit at a time [62, 63, 67]. Critically, the ideal action of each IPE measurement leaves the eigenstate $|\phi\rangle$ invariant, such that initial state preparation can be reused between iterations up to the extent allowed by noise.

Algorithm 2. Iterative PE (single iteration).

Require: `target` must be in an eigenstate $|\phi\rangle$ of `U`
procedure IterativePEStep(`U`, `t`, ϕ_{inv}, `target`)
  `q` ← fresh qubit
  `H(q)`
  `Rz(`$-\phi_{\text{inv}} \cdot$ `t, q)`
  `Controlled(q) U(t, target)`
  ▷ `q` should now be in the state $(|0\rangle + e^{i(\phi - \phi_{\text{inv}})t}|1\rangle)/\sqrt{2}$.
  `H(q)`
  return the result of measuring `q` in the Z-basis
end procedure

By contrast with VQE, both QPE and IPE use $O(1/\epsilon)$ time to estimate ϕ, roughly corresponding to the difference between the standard quantum and Heisenberg limits for metrology. This quadratic advantage together with the ability to reuse state preparations allows for PE methods to be much more efficient at estimating eigenvalues than VQE.

In practice, however, noise in near- and medium-term devices can make running IPE challenging due to the large gate depths introduced by calling U for a variety of different evolution times while qubits remain coherent. To mitigate this, one can consider resetting the eigenstate register when needed; making this decision online, however, requires an online estimate $\hat{\phi}$ of ϕ to detect when inconsistent measurement results are

observed. The problem of estimating a parameter conditioned on a partial data record is a natural fit for Bayesian inference [64], such that we consider Bayesian PE methods in this section. Adopting a Bayesian approach also allows extending adaptivity to include online experiment design as well as the heuristics used by dynamic phase estimation [7].

If we consider a single IPE iteration of the form listed in Algorithm 2, then the probability of getting a 1 at the end of the iteration is given by $\Pr(1|\phi; t) = \cos^2(\phi t/2)$. This forms a *likelihood function*, such that we can use Bayes' rule to compute $\Pr(\phi|d_0, d_1, \ldots, d_{n-1})$ from a sequence of measurements $\{d_0, d_1, \ldots, d_{n-1}\}$ collected at evolution times $\{t_0, t_1, \ldots, t_{n-1}\}$ [68]. The expectation value over this distribution then minimizes the error in our online estimate of ϕ [69].

Performing exact Bayesian inference can be prohibitively expensive, however, especially within qubit lifetimes. Online approximation methods such as particle filtering [64, 70] can reduce Bayesian inference to a Markov chain conditioned on experimental measurements, diminishing the cost for Bayesian PE. Rejection filtering phase estimation (RejF PE) [65] further reduces costs by using rejection sampling to implement a reduced form of particle filtering, allowing adaptive resets to be performed with fewer classical arithmetic operations.

Recently, the random walk phase estimation (RWPE) algorithm [66] was introduced to allow for computing online estimates using only a few arithmetic expressions per iteration, making it practical to use PE methods that are iterative on near- and medium-term devices subject to noise.

Algorithm 3. Basic random walk phase estimation algorithm of Granade and Wiebe [66].

function RandomWalkPhaseEst(μ_0, σ_0, U)
  μ_0: initial mean
  σ_0: initial standard deviation
  U: oracle whose eigenvalues are to be estimated
  `target`: A register of qubits prepared in an eigenstate of U.

  ■ *Initialization*
  $\mu \leftarrow \mu_0$
  $\sigma \leftarrow \sigma_0$

  ■ *Main body*
  for $i_{\text{exp}} \in \{0, 1, \ldots n_{\text{exp}} - 1\}$ **do**
    $\phi_{\text{inv}} \leftarrow \mu - \pi\sigma/2$
    $t \leftarrow 1/\sigma$
    Sample d from IterativePEStep(U, t, ϕ_{inv}, `target`)
    ▷ $\Pr(d = 0|\phi; \phi_{\text{inv}}, t) = \cos^2(t(\phi - \phi_{\text{inv}})/2)$.
    if $d = 0$ **then**
      $\mu \leftarrow \mu + \sigma/\sqrt{e}$
    else
      $\mu \leftarrow \mu - \sigma/\sqrt{e}$
    end if
    $\sigma \leftarrow \sigma\sqrt{(e-1)/e}$
  end for

  ■ *Final estimate*
  return $\hat{\phi} \leftarrow \mu$
end function

Critical to the execution of RWPE is that the update of μ and σ happens during execution so that the new values of each variable can be used as inputs to U. This requires us to not only branch based on measurement outcomes, but to maintain a continually changing program state without returning to a remote classical processor. This is the advantage of the hybrid, adaptive algorithm of this type.

In contrast, to implement the equivalent program logic with existing methods (no classical computation inside the quantum code), one would re-write Algorithm 3 to use lookup tables rather than floating-point variables—in particular, if we know the whole history of quantum measurements made throughout an RWPE run, then we can reconstruct μ and σ. This table grows exponentially in size with the number of measurements made, however. Practical applications may require between 20 and 60 iterations of RWPE (yielding respective relative accuracies of 10^{-2} and 10^{-6}), requiring the storage of prohibitively large lookup tables.

The RWPE algorithm is a clear example of the methods available to a developer with access to this new form of hybrid and adaptive quantum programming. In the following section, our efforts were focused on demonstrating the practical viability of the approach by executing the RWPE program on a specific quantum computing system.

5 Execution on quantum computing systems

For this work, we executed the Random Walk Phase Estimation program in Section 4.3 on both a quantum simulator and a physical quantum computing system. We discuss how this was accomplished, focusing on the interpretation of the intermediate representation RWPE program and the specific parameters used in the program to control its execution. Execution of this program was performed on a quantum simulator enhanced with classical computation capability and a next-generation quantum computing system provided by Quantum Circuits Inc (QCI) [71, 72]. The QCI system is one of the first quantum computers designed with a control system that provides the novel capabilities integral to the enhanced hybrid quantum-classical programming model and necessary for execution of the RWPE program that was selected as the primary example.

5.1 Compiling RWPE and interpreting QIR

The quantum intermediate representation (QIR) discussed in Section 3.2 enables the abstract definition of an advanced quantum program in a form that is independent of any specific target system. The QIR can be produced from a variety of higher-level programming languages. For this demonstration, we chose to define the RWPE program in the Q# language (source code shown in section A1), taking advantage of QIR generation support provided by the Q# compiler. We then utilized the QAT tool [55] to apply transformations during compilation to produce

QIR compatible with hardware (e.g. by assigning static qubit indices). Submission and execution of the QIR program is all managed within the Azure Quantum service [59].

The RWPE program in its targeted QIR form can be transformed to the native program representation required for execution on a specific backend quantum computing system. QCI used a pre-release version of the PyQIR package for *Python* [73] to parse the QIR representation of RWPE and to perform the transformation within the QCI quantum program compiler. The mapping from QIR program features to QCI's intermediate representation was mostly a one-to-one translation, with a few exceptions. Specifically, the quantum gates that are generated after transpilation are unique to the QCI hardware and the behavior of division with respect to numeric data types is limited as described below.

Integer and Boolean values are mapped to 18 bit signed integers, while doubles map to Q2.16 fixed-point integers (with range $[-2, 2-2^{-16}]$). For a numerical value used as an angle, we assume a convention that its value is in units of π for a range of two full periods. Native addition, subtraction, and multiplication are supported, but division is implemented with an approximation using interpolation table techniques. Control flow is implemented with hardware level if and goto statements and quantum gates are transpiled to the native gate set, described in Section 5.4, using internal methods. The system has the ability to return intermediate values, including measurements, to the user.

The QCI compiler will raise an error if a variable is set to a value outside its allowed range by the QIR, although there is no run-time check on hardware if a value underflows or overflows. Indeed, in this RWPE program, sigma will underflow after about 20 iterations. The program variable 1/sigma is used as an angle and its run-time overflow behavior of wrapping around the bounds is acceptable. These behaviors were validated for RWPE using simulations. For many problems of interest, sufficient accuracy may be obtained with these constraints.

These limitations highlight two consequences of using a quantum intermediate representation. First, data types provided in hardware may not match exactly what is specified in the QIR representation of the program. Second, since the same QIR may be used with different hardware, the end user does not need to change their algorithm in order to target different backend systems. For this reason, it is important for the user to have available a simulator subject to the same classical limitations as the hardware in order to validate program behavior prior to scaling the program up to larger numbers of qubits. See Section 5.3 for details about QCI's simulator.

5.2 Parameterizing RWPE program for execution

When a program such as RWPE is executed, a user may want to select options to analyze variations in the run-time behavior. In this case, the RWPE program shown in Supplementary Appendix defines an oracle, $U(t) = R_z(-0.5t)$, for which $\phi = \pm 0.5$ (in units of π) are the true eigenvalues. The RWPE algorithm will preferentially converge on the eigenvalue at 0.5 because it is closer to the initial value for the prior estimate mu. For this demonstration, the parameter value −0.5 to the R_z gate is hard-coded in the program source. Alternatively, this could have been passed as a program parameter, but this was not in place at the time this program was executed. Consequently, the program is always expected to produce the result $\hat{\phi} \approx 0.5$.

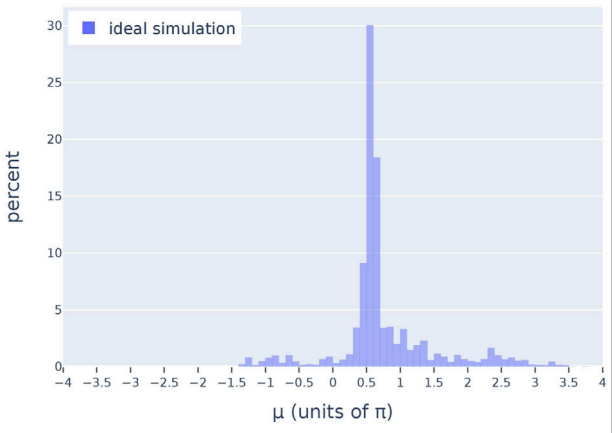

FIGURE 4
Ideal simulation of 10,000 shots, each resulting in one value of μ included in this histogram. Each value of μ shows where the RWPE algorithm ended after 24 iterations, and the success of the RWPE algorithm is exhibited in the highest peak appearing at the eigenvalue $\mu = 0.5$. The other values in the histogram highlight the random walk aspect of the RWPE algorithm and show that the algorithm can "fail" even if perfectly executed. The asymmetry of the distribution reflects the starting value, μ_0. The height of the peak depends on the number of bins, which was 100 in this paper.

For each program execution, the inner code loop is executed *nIter* = 24 times, as the algorithm can make no progress beyond this due to underflow. Throughout this paper, we refer to such executions as *shots*, and use the term to include all embedded loops and intermediate measurements. In this case, each shot returns a single datum, the estimated value of mu. Since the eigenstate will decohere across iterations within a single shot, the program includes a parameter specifying a subset of iterations to run before the eigenstate is refreshed (reset and prepared anew). In general, if the eigenstate is expensive to prepare, it should be refreshed less often as each refresh will increase execution time; the faster the eigenstate decoheres, the more often it should be refreshed to improve accuracy. In this program, the eigenstate is simply $|1\rangle$, prepared with a reset and an X gate, which favors a frequent refresh. For our demonstration, we chose to refresh every other iteration to highlight the existence of this user-selectable trade-off.

The algorithm required a final calculation of mu * 2, which we did in post processing instead of in the Q# program, meaning the range of resultant values of mu was effectively doubled relative to the range of the fixed-point data type. We could obtain a better estimate of the eigenvalue by refitting all measurements as a post-processing

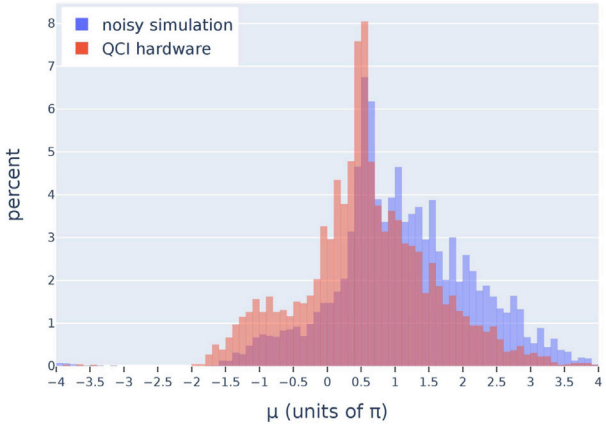

FIGURE 5
Results on hardware compared with a noisy simulation. We collected 5,000 shots on hardware and 10,000 shots in the simulator. This histogram is comparable with Figure 4 because it uses the same *x*-axis bounds and number of bins. The correctness of the result is evident in the position on the *x*-axis of the highest peak, not directly its height. The peak at $\mu = 0.5$ is lower in this histogram because the noise made it less likely to converge there.

step as described in Section 4, but we do not do that here. Instead we only show the result estimated at run-time by the program.

5.3 Execution on an enhanced quantum simulator

Prior to executing on quantum hardware, it is important to validate the program by running it on a simulator that mimics closely the computational behavior of the target system. Existing simulators that execute only quantum operations or which don't match the capabilities of the target hardware are not adequate.

To address this, QCI developed a custom simulator in which control-flow and classical operations are executed in *Python* and quantum operations on a statevector or a density matrix from Qiskit Terra's [48] `quantum_info` module. Classical registers are simulated using native *Python* int and float data types or, to model the hardware closely, the fixedpoint *Python* package. The simulator can also leverage Qiskit Aer's `noise_model` module to include quantum gate and readout noise. Three sources of infidelity can be modeled: a finite number of shots, quantum noise, and hardware-specific classical computation.

We executed the RWPE program on the simulator with no noise configured, i.e. an "ideal simulation," to validate that the algorithm performs as designed. Results are shown in Figure 4. To mimic an ideal quantum computer, the statevector simulator is seeded with a pseudo random number generator (RNG). It includes no quantum noise, and classical operations use full precision registers. Each repetition, or shot, of the RWPE protocol generates a single value for mu corresponding to a different RNG seed. The only source of error in this simulation is associated with the RNG in the finite number of shots executed. The result of the simulation is a distinct peak in the histogram at the expected eigenvalue of 0.5.

We then ran the RWPE simulation with a noise model that approximates the characteristics of the target hardware to predict the behavior to be expected when the program is executed on that system. Data were obtained from execution of the RWPE program on this "noisy" simulator and compared against results obtained when running on hardware, as discussed in Section 5.4 below.

5.4 Execution on quantum hardware

In this section, we present results obtained from executing the RWPE program on a quantum computing system provided by Quantum Circuits Inc (QCI) [71, 72]. For its work with Microsoft Azure, QCI had deployed for testing and validation a quantum computer designed around superconducting 3-D resonator technology and which provides a hardware-efficient platform for the development of advanced quantum algorithms. The control system that manages the components of this system has been implemented with many of the quantum/classical computational features described in this paper.

In Figure 5 we present results from execution of the RWPE program on the QCI quantum hardware alongside results from the quantum simulator described above Section 5.3). Execution on the simulator was performed using a QCI-specific noise model along with classical computation which models the fixed-point precision of the system.

For both the noisy simulation and the hardware execution, there is a prominent peak in the data at $\mu = 0.5$ which corresponds to the eigenvalue described in Section 5.2, matching closely what was seen in the ideal simulation. The correctness of these results is suggested by the location of the highest peak in this histogram. Both the hardware and simulator results include shoulders in the data, loosely corresponding to where we see population in the ideal simulation in Figure 4. Detailed error analysis and quantitative comparison of these results was deferred to future work. These results may be considered sufficient to confirm a successful translation of the quantum program defined in the QIR representation, resulting in nearly equivalent results on the both the simulator and the QCI hardware.

The native gate set used in the QCI hardware system includes the H, $\sqrt{X}$, X, and RZ single qubit gates. As discussed

in Section 3.3, the RZ gate can accept variable arguments at run-time and is executed in a small fraction of the time it takes to execute a pulse-based gate. The native entangling gate in the QCI hardware is the exponential-SWAP [74], or ESWAP, which can also accept a run-time variable argument, and which has the unitary

$$U_{\mathrm{eSWAP}}(\theta) = \begin{pmatrix} e^{-i\theta/2} & 0 & 0 & 0 \\ 0 & \cos(\theta/2) & -i\sin(\theta/2) & 0 \\ 0 & -i\sin(\theta/2) & \cos(\theta/2) & 0 \\ 0 & 0 & 0 & e^{-i\theta/2} \end{pmatrix} = e^{-i\frac{\theta}{2}(11+XX+YY+ZZ)}. \quad (2)$$

Useful for larger programs than the one demonstrated here, the SWAP gate is a special case of the ESWAP corresponding to $\theta = \pi$. Near synonyms of this gate are referred to as the SwapPowGate [50] and SWAP$^{\alpha}$ [75]. Gate fidelity is likely the largest factor influencing the height of the peak in the results for both the noisy simulator and hardware, however, and these fidelities were sufficient for the RWPE algorithm to produce a well-defined solution.

6 Future work

The RWPE algorithm presented here could be enhanced with additional study. We limited our consideration only to those inferences made on timescales significantly shorter than qubit lifetimes. Even under such severe constraints, the deep integration of classical and quantum computation allows us to derive estimates of the phase in real-time. With this unique capability, we can validate hypotheses that mitigate the impact of noise and other errors on phase estimation.

Generally, we could relax this restriction by re-analyzing intermediate measurements after the fact. The sequence $\{(t_i, \phi_{\mathrm{inv},i}, d_i)\}$ of evolution times, inversion angles, and intermediate measurement results is sufficient to capture the inferential effect of the RWPE protocol in the likelihood function:

$$\Pr(\mathrm{data}|\phi) = \prod_i \begin{cases} \cos^2\left(\left[\phi - \phi_{\mathrm{inv},i}\right]t_i/2\right) & \text{if } d_i = 0 \\ \sin^2\left(\left[\phi - \phi_{\mathrm{inv},i}\right]t_i/2\right) & \text{if } d_i = 1. \end{cases} \quad (3)$$

Taking an expectation over a prior distribution updated by (3) yields an estimate that minimizes the average mean squared error in ϕ [69]; such expectation values can be readily computed in postprocessing using software packages such as PyMC3 [76] or QInfer [77]. For example, Granade and Wiebe [66] used a QInfer particle filter to post-process results of RWPE execution and observed a reduction in the impact of outliers on overall performance, allowing recovery from some approximation failures observed in Figure 4.

In a similar fashion, future work could incorporate real-time hybrid applications more deeply into data processing workflows. Intermediate measurements can be used to inform online experiment design heuristics and approximations, while the full power of classical data processing can be used outside of qubit lifetimes to refine estimates offline.

Beyond the RWPE algorithm, an obvious next step is to explore other quantum algorithms that might benefit from the ability to do mathematical computation within the quantum program. One use case for this is in the quantum algorithms designed to implement error correction. Any quantum algorithm that requires some computation of classical variables within the algorithm itself could conceivably be implemented using classical arithmetic operations rather than with quantum gates. Other known complex algorithms could be candidates for efficiency gains, such as in chemistry simulation [78]. Developing entirely new and novel algorithms that use this capability at their core is another area to be explored.

7 Summary and conclusion

In this paper, we have demonstrated an initial step towards fully general and tightly integrated quantum and classical processing, backed by QIR, an intermediate representation that can be used to express hybrid quantum–classical programs in a form that can be translated to the unique assembly language of specific hardware targets. We then used a newly developed suite of supporting software tools together with advanced quantum simulation and physical quantum hardware capabilities to demonstrate random walk phase estimation, a recent algorithm that effectively exploits real–time hybrid quantum–classical computation within a quantum program.

The Quantum Algorithm Zoo [3] lists 64 quantum computing algorithms—by comparison, Knuth's unfinished multi-volume compendium of classical algorithms [79] has more than 64 chapters. One hypothesis for the enormous gap between the number of quantum and classical algorithms is that we have a better language for the latter, as it is easier to conceive of new algorithms when we have a language in which to express them [80] easily, along with a robust software and hardware ecosystem in which they can be exercised.

If it is the case that progress towards development of new and more efficient quantum algorithms has been limited by our computational models, then the work described in this paper offers substantial progress towards resolving that gap. We hope that the results that we demonstrated in this paper provide an impetus to fundamentally rethink and expand what a quantum algorithm can look like and to go beyond the limitations seen in the current algorithms.

The success of quantum computing hinges not only on progress in qubit fidelities and lifetimes and in electronics for controlling qubits, but also on the development of innovative software components that support the hardware in creative and

extensible ways. Our work may enable us to narrow the gap, and accelerate the development of new techniques.

The section on Code Availability below VII provides information about various public repositories containing tools related to the Quantum Intermediate Representation (QIR). We are early in the evolution of this new form of hybrid and adaptive quantum programming and it is likely that additional resources and examples will soon become available.

8 Code availability

The QIR Alliance [45] provides information and tools to support the development of a specification for Quantum Intermediate Representation (QIR) [81]. Source code for tools such as "PyQIR" and "QAT" mentioned in the text of this paper is available online [55, 73].

For more information about the OpenQASM 3.0 specification, please see Cross et al. [44].

Data availability statement

The raw data supporting the conclusions of this article will be made available by the authors, without undue reservation.

Author contributions

All the authors were substantial contributors not only to content but also to review. TL, CG, AA, and BH contributed the largest part of the text, while MR, AP, and AG each were responsible for a subsection or two. Review was conducted regularly by the entire group.

Conflict of interest

TL, AA, and AP are employed by Quantum Circuits Inc. CG, AG, MR, and BH are employed by Microsoft 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.

Supplementary material

The Supplementary Material for this article can be found online at: https://www.frontiersin.org/articles/10.3389/fphy.2022.940293/full#supplementary-material

References

1. Frank A, Arya K, Ryan B, Bacon D, Joseph CB, Barends R, et al. Quantum supremacy using a programmable superconducting processor. *Nature* (2019) 574(7779):505–10. doi:10.1038/s41586-019-1666-5

2. Preskill J. Quantum computing in the NISQ era and beyond. *Quantum* (2018) 2:79. doi:10.22331/q-2018-08-06-79

3. Jordan S. *Algebraic and number theoretic algorithms*. Available from: https://quantumalgorithmzoo.org/. (Accessed February, 2022).

4. Bharti K, Cervera-Lierta A, Kyaw TH, Haug T, Alperin-Lea S, Anand A, et al. *Noisy intermediate-scale quantum (nisq) algorithms* (2021).

5. Zhong HS, Wang H, Deng YH, Chen MC, Peng LC, Luo YH, et al. Quantum computational advantage using photons. *Science* (2020) 370(6523):1460–3. doi:10.1126/science.abe8770

6. Ferracin S, Kapourniotis T, Datta A. Accrediting outputs of noisy intermediate-scale quantum computing devices. *New J Phys* (2019) 21(11):113038. doi:10.1088/1367-2630/ab4fd6

7. Córcoles AD, Takita M, Inoue K, Scott L, Minev ZK, Chow JM, et al. Exploiting dynamic quantum circuits in a quantum algorithm with superconducting qubits. *Phys Rev Lett* (2021) 127(10):100501. doi:10.1103/physrevlett.127.100501

8. Lior E. *How to dramatically increase the initialization fidelity of your qubits with qua* (2022). Available from: https://www.quantum-machines.co/blog/increasing-qubit-initialization-fidelity-with-qua/. (Accessed February, 2022).

9. McArdle S, Endo S, Aspuru-Guzik A, Benjamin SC, Yuan X. Quantum computational chemistry. *Rev Mod Phys* (2020) 92:015003. doi:10.1103/revmodphys.92.015003

10. Reiher M, Wiebe N, Svore KM, Wecker D, Troyer M. Elucidating reaction mechanisms on quantum computers. *Proc Natl Acad Sci U S A* (2017) 114(29):7555–60. doi:10.1073/pnas.1619152114

11. Burg VV, Low GH, Häner T, Steiger DS, Reiher M, Roetteler M, et al. Quantum computing enhanced computational catalysis. *Phys Rev Res* (2021) 3:033055. doi:10.1103/physrevresearch.3.033055

12. Bauer B, Wecker D, Millis AJ, Hastings MB, Troyer M. Hybrid quantum-classical approach to correlated materials. *Phys Rev X* (2016) 6:031045. doi:10.1103/physrevx.6.031045

13. Alberto P, McClean J, Shadbolt P, Man-Hong Y, Zhou X-Q, Love PJ, et al. A variational eigenvalue solver on a photonic quantum processor. *Nat Commun* (2014) 5(1):4213. doi:10.1038/ncomms5213

14. Spall JC. An overview of the simultaneous perturbation method for efficient optimization. *John Hopkins APL Tech Dig* (1998) 19(4):482–92.

15. Farhi E, Goldstone J, Gutmann S. *A quantum approximate optimization algorithm* (2014).

16. Hastings MB. *Classical and quantum bounded depth approximation algorithms* (2019).

17. Qiskit Tutorials. *Advanced circuits*. San Jose, California, USA: IBM Quantum Lab (2022). Available from: https://quantum-computing.ibm.com/lab/docs/iql/advanced-circuits. (Accessed February, 2022).

18. Quantum AI. *Cirq basics*. Santa Barbara, California: Quantum AI (2022). Available from: https://quantumai.google/cirq/tutorials/basics. (Accessed February, 2022).

19. Quantum AI. *Quantum circuits on rigetti devices*. Santa Barbara, California: Quantum AI (2022). Available from: https://quantumai.google/cirq/tutorials/rigetti/getting_started. (Accessed February, 2022).

20. Smith RS, Curtis MJ, Zeng WJ. A practical quantum instruction set architecture. *arXiv:1608.03355 [quant-ph]* (2016).

21. Karalekas PJ, Tezak NA, Peterson EC, Ryan CA, da Silva MP, Smith RS, et al. A quantum-classical cloud platform optimized for variational hybrid algorithms. *Quan Sci Technol* (2020) 5(2):024003. doi:10.1088/2058-9565/ab7559

22. Johnson B, Faro I. *Ibm quantum delivers 120x speedup of quantum workloads with qiskit runtime* (2021). Available from: https://research.ibm.com/blog/120x-quantum-speedup?lnk=ushpv18re2. (Accessed February, 2022).

23. Runtime Q. *IBM quantum lab*. San Jose, California, USA: IBM Quantum Lab (2021). Available from: https://quantum-computing.ibm.com/lab/docs/iql/runtime/. (Accessed February, 2022).

24. Faro I, Johnson B, Behrendt M, Gambetta J. *Introducing Quantum Serverless, a new programming model for leveraging quantum and classical resources*. San Jose, California, USA: IBM Research (2021). Available from: https://research.ibm.com/blog/quantum-serverless-programming/. (Accessed February, 2022).

25. Faro I, Johnson B, Gambetta J. *Rethinking quantum systems for faster, more efficient computation*. San Jose, California, USA: IBM Research (2020). Available from: https://research.ibm.com/blog/near-real-time-quantum-compute/. (Accessed February, 2022).

26. Nation and Johnson. *How to measure and reset a qubit in the middle of a circuit execution* (2021). Available from: https://www.ibm.com/blogs/research/2021/02/quantum-mid-circuit-measurement. (Accessed February, 2022).

27. Mid-Circuit Measurements Tutorial. *Mid-circuit measurements tutorial*. San Jose, California, USA: IBM Quantum Lab (2021). Available from: https://quantum-computing.ibm.com/lab/docs/iql/manage/systems/midcircuit-measurement/. (Accessed February, 2022).

28. Gaebler JP, Baldwin CH, Moses SA, Dreiling JM, Figgatt C, Foss-Feig M, et al. *Suppression of mid-circuit measurement crosstalk errors with micromotion* (2021).

29. Moore S. *Honeywell's ion trap quantum computer makes big leap* (2020). Available from: https://spectrum.ieee.org/searchContent?q=quantum%2Bsupremacy. (Accessed February, 2022).

30. Paetznick A, Svore K. *Repeat-until-success: Non-deterministic decomposition of single-qubit unitaries* (2013). Available from: http://arxiv.org/abs/1311.1074. (Accessed February, 2022).

31. Bocharov A, Rötteler M, Svore KM. Efficient synthesis of universal repeat-until-success quantum circuits. *Phys Rev Lett* (2015) 114:080502. See also arXiv preprint arXiv:1404.5320. doi:10.1103/physrevlett.114.080502

32. Bocharov A, Rötteler M, Svore KM. Efficient synthesis of probabilistic quantum circuits with fallback. *Phys Rev A (Coll Park)* (2015) 91:052317. See also arXiv preprint arXiv:1409.3552. doi:10.1103/physreva.91.052317

33. Kliuchnikov V, Maslov D, Mosca M. *Asymptotically optimal approximation of single qubit unitaries by Clifford and T circuits using a constant number of ancillary qubits* (2012). Available from: http://arxiv.org/abs/1212.0822. (Accessed February, 2022).

34. Ross N, Selinger P. *Optimal ancilla-free Clifford+T approximation of z-rotations* (2014). Available from: http://arxiv.org/abs/1403.2975. (Accessed February, 2022).

35. Kliuchnikov V, Lauter K, Minko APR, Petit C. *Shorter quantum circuits* (2022). Available from: http://arxiv.org/abs/2203.10064. (Accessed February, 2022).

36. Griffiths RB, Niu CS. Semiclassical Fourier transform for quantum computation. *Phys Rev Lett* (1996) 76(17):3228–31. doi:10.1103/physrevlett.76.3228

37. Kuperberg G. *A subexponential-time quantum algorithm for the dihedral hidden subgroup problem* (2003). Available from: https://arxiv.org/abs/quant-ph/0302112. (Accessed February, 2022).

38. Ozols M, Roetteler M, Roland J. Quantum rejection sampling. *ACM Trans Comput Theor* (2013) 5(3):11–33. doi:10.1145/2493252.2493256

39. Martyn JM, Rossi ZM, Tan AK, Chuang IL. Grand unification of quantum algorithms. *PRX Quan* (2021) 2:040203. doi:10.1103/prxquantum.2.040203

40. Reinhold P. *Controlling error-correctable bosonic qubits*. Schoelkopf Lab Ph.D. Thesis (2019).

41. Granade C, Ferrie C, Cory DG. Accelerated randomized benchmarking. *New J Phys* (2015) 17(1):013042. doi:10.1088/1367-2630/17/1/013042

42. Ofek N, Petrenko A, Heeres R, Reinhold P, Leghtas Z, Vlastakis B, et al. *Demonstrating quantum error correction that extends the lifetime of quantum information* (2016). Available from: https://arxiv.org/abs/1602.04768. (Accessed February, 2022).

McCaskey AJ, Dumitrescu EF, Liakh D, Chen M, Feng W, Humble TS, et al. A language and hardware independent approach to quantum–classical computing. *SoftwareX* (2018) 7:245–54. doi:10.1016/j.softx.2018.07.007

44. Cross AW, Javadi-Abhari A, Alexander T, de Beaudrap N, Bishop LS, Heidel S, et al. *Openqasm 3: A broader and deeper quantum assembly language* (2021).

45. QIR Alliance. *GitHub repository*. QIR Alliance. (2022). Available from: https://github.com/. (Accessed February, 2022).

46. Q#: Enabling scalable quantum computing and development with a high-level DSL.

47. Bichsel B, Baader M, Gehr T, Vechev M. Silq: A high-level quantum language with safe uncomputation and intuitive semantics. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2020. New York, NY, USA: Association for Computing Machinery (2020). p. 286–300. ISBN 978-1-4503-7613-6. doi:10.1145/3385412.3386007

48. Qiskit website. *Qiskit open source quantum development*. Armonk, New York: IBM Research (2021). Available from: https://qiskit.org. (Accessed February, 2022).

49. GitHub repository. *PyQuil: Quantum programming in Python* (2022). Available from: GitHub repository https://github.com/rigetti/pyquil.. (Accessed February, 2022).

50. Google Cirq website. *Google cirq qantum AI*. Mountain View, CA: Google (2021). Available from: https://quantumai.google/cirq. (Accessed February, 2022).

51. Sivarajah S, Dilkes S, Alexander C, Simmons W, Edgington A, Duncan R, et al. t| ket⟩: A retargetable compiler for NISQ devices. *Quan Sci Technol* (2021) 6(1):014003. doi:10.1088/2058-9565/ab8e92

52. *The LLVM compiler infrastructure*. Available from: https://llvm.org/. (Accessed February, 2022).

53. *LLVM language reference manual — LLVM 15.0.0git documentation*. Available from: https://llvm.org/docs/LangRef.html. (Accessed February, 2022).

54. *NVVM IR specification*. Available from: https://docs.nvidia.com/cuda/nvvm-ir-spec/. (Accessed February, 2022).

55. QIR Alliance. *QAT*. Delhi, India: QIR Alliance (2022).

56. McKay DC, Wood CJ, Sheldon S, Jerry Chow M, Gambetta JM. Efficient Z gates for quantum computing. *Phys Rev A (Coll Park)* (2017) 96(2):022330. doi:10.1103/physreva.96.022330

57. Navascués M. Resetting uncontrolled quantum systems. *Phys Rev X* (2018) 8(3):031008. doi:10.1103/physrevx.8.031008

58. *Conditional reset on ibm quantum systems*. Available from: https://quantum-computing.ibm.com/admin/docs/admin/manage/systems/reset/backend_reset. (Accessed February, 2022).

59. Microsoft Q# website. *Microsoft Q# and the quantum development kit*. Redmond, WA: Microsoft (2021). Available from: https://azure.microsoft.com/en-us/resources/development-kit/quantum-computing/. (Accessed February, 2022).

60. Note1. This is completely general for *any* algorithm that uses independent measurements of a quantum state, and follows immediately from that the Fisher information for independent measurements simply adds. Thus, by the Cramèr–Rao bound [?], if a single VQE measurement yields Fisher information I_0, the variance after N shots is bounded by $\sigma^2 \geq 1/NI_0$, such that $1/\epsilon^2$ shots are required to ensure that $\sigma \leq \epsilon$.

61. Yu Kitaev A. Quantum measurements and the abelian stabilizer problem. *arXiv:quant-ph/9511026* (1995).

62. Svore KM, Hastings MB, Freedman M. Faster phase estimation. *arXiv:1304.0741* (2013).

63. Kimmel S, Low GH, Theodore Yoder J. Robust calibration of a universal single-qubit gate set via robust phase estimation. *Phys Rev A (Coll Park)* (2015) 92(6):062315. doi:10.1103/PhysRevA.92.062315

64. Granade CE, Ferrie C, Nathan W, Cory DG. Robust online Hamiltonian learning. *New J Phys* (2012) 14(10):103013. doi:10.1088/1367-2630/14/10/103013

65. Nathan W, Granade C. Efficient Bayesian phase estimation. *Phys Rev Lett* (2016) 117(1):010503. doi:10.1103/PhysRevLett.117.010503

66. Granade C, Nathan W. *Using random walks for iterative phase estimation (in preparation)* (2022).

67. Kitaev AY. Quantum computations: Algorithms and error correction. *Russ Math Surv* (1997) 52:1191–249. doi:10.1070/RM1997v052n06ABEH002155

68. Note2. This expression of the likelihood function also allows for the calculation of Cramèr–Rao bounds for IPE, as in the work of ?], confirming that IPE requires $O(1/\epsilon)$ time to reach an accuracy of ϵ.

69. Banerjee A, Guo X, Wang H. On the optimality of conditional expectation as a Bregman predictor. *IEEE Trans Inf Theor* (2005) 51(7):2664–9. doi:10.1109/TIT.2005.850145

70. Doucet A, Johansen AM. *A tutorial on particle filtering and smoothing: Fifteen years later* (2011).

71. Quantum Circuits Partnership with Microsoft on Azure Quantum. *Quantum circuits partnership with Microsoft - press release 2021* (2021). Available from: https://quantumcircuits.com/news-and-publications/quantum-circuits-partners-with-microsoft-on-azure-quantum. (Accessed February, 2022).

72. Quantum Circuits Inc. *Quantum circuits Inc. Website*. New Haven, Connecticut: Quantum Circuits Inc. Website (2022). Available from: https://quantumcircuits.com. (Accessed February, 2022).

73. QIR Alliance. *PyQIR*. Delhi, India: QIR Alliance (2022).

74. Gao YY, BrianLester J, Chou KS, Frunzio L, Devoret MH, Jiang L, et al. Entanglement of bosonic modes through an engineered exchange interaction. *Nature* (2019) 566(7745):509–12. doi:10.1038/s41586-019-0970-4

75. Fan H, Roychowdhury V, Szkopek T. Optimal two-qubit quantum circuits using exchange interactions. *Phys Rev A (Coll Park)* (2005) 72(5):052323. doi:10.1103/physreva.72.052323

76. Salvatier J, Wiecki TV, Fonnesbeck C. Probabilistic programming in python using pymc3. *PeerJ Comput Sci* (2016) 2:e55. doi:10.7717/peerj-cs.55

77. Granade C, Ferrie C, Hincks I, Casagrande S, Alexander T, Gross J, et al. QInfer: Statistical inference software for quantum applications. *Quantum* (2017) 1:5. doi:10.22331/q-2017-04-25-5

78. Reiher M, Nathan W, Svore KM, Wecker D, Troyer M. Elucidating reaction mechanisms on quantum computers. *Proc Natl Acad Sci U S A* (2017) 114:201619152. doi:10.1073/pnas.1619152114

79. Ervin Knuth D. *The art of computer programming: Volumes 1-4A boxed set*. Boston, MA: Addison-Wesley Publ. (2011).

80. Iverson KE. Notation as a tool of thought. *Commun ACM* (1980) 23(8):444–65. doi:10.1145/358896.358899

81. QIR Alliance. Delhi, India: QIR Alliance (2022)