Hybrid Programming for Near-term Quantum Computing Systems

Alexander McCaskey
Computer Science and Mathematics
Oak Ridge National Laboratory
Oak Ridge, TN, U.S.
mccaskeyaj@ornl.gov

Eugene Dumitrescu
Computational Sciences and Engineering
Oak Ridge National Laboratory
Oak Ridge, TN, U.S.
dumitrescuef@ornl.gov

Dmitry Liakh
National Center for Computational Sciences
Oak Ridge National Laboratory
Oak Ridge, TN, U.S.
liakhdi@ornl.gov

Travis Humble
Computational Sciences and Engineering
Oak Ridge National Laboratory
Oak Ridge, TN, U.S.
humblets@ornl.gov

***Abstract*—Recent computations involving quantum processing units (QPUs) have demonstrated a series of challenges inherent to hybrid classical-quantum programming, compilation, execution, and verification and validation. Despite considerable progress, system-level noise, limited low-level instructions sets, remote access models, and an overall lack of portability and classical integration presents near-term programming challenges that must be overcome in order to enable reliable scientific quantum computing and support robust hardware benchmarking. In this work, we draw on our experience in programming QPUs to identify common concerns and challenges, and detail best practices for mitigating these challenges within the current hybrid classical-quantum computing paradigm. Following this discussion, we introduce the XACC quantum compilation and execution framework as a hardware and language independent solution that addresses many of these hybrid programming challenges. XACC supports extensible methodologies for managing a variety of programming, compilation, and execution concerns across the increasingly diverse set of QPUs. We use recent nuclear physics simulations to illustrate how the framework mitigates programming, compilation, and execution challenges and manages the complex workflow present in QPU-enhanced scientific applications. Finally, we codify the resulting hybrid scientific computing workflow in order to identify key areas requiring future improvement.**

***Index Terms*—Quantum Computing, Quantum Programming Models**

I. Introduction

Currently available quantum processing units (QPUs) consisting of tens of qubits are providing a unique capability for understanding hybrid classical-quantum algorithms and associated speedups for future scientific computing applications. Such applications range across scientific domains, and small-scale demonstrations of quantum programming have been developed in fields such as nuclear and high-energy physics, machine learning, chemistry, and materials science [1]. While QPU hardware development is progressing at a rapid pace, these near-term quantum computing systems are far from ideal [2]. Low-level unitary quantum instruction noise, readout errors, decoherence patheways, and remote programming access models limit the scalability of these devices to research applications. Recent results in hybrid classical-quantum variational algorithms demonstrate the potential ability to mitigate some of these challenges, specifically QPU noise errors, but there is an overall lack of awareness of the software tooling needed by programmers and domain scientists to leverage such computing systems in a robust and coherent way.

State of the art demonstrations of hybrid scientific quantum computations on gate-model QPUs, e.g. devices offered by vendors such as IBM, Rigetti, and Google, have had varied success and limited simulation accuracy [3]–[6]. For example, the first variational hybrid quantum computation via remote cloud resources reached an overall simulation accuracy of 3 percent [5]. This is a computation that can be performed by a classical computer in microseconds, yet it took months of work to map it to a quantum computer and execute via a remote cloud access model with job queue constraints. These types of near-term hybrid programming challenges must be overcome to enable reliable and reproducible quantum computing applications and to support the continued testing and characterization of quantum computer performance.

Can a robust software platform and workflow improve the usability and accessibility of near-term quantum computers for scientific applications? In this work, we attempt to answer this question in the affirmative by providing a model workflow for near-term hybrid quantum computations that enables useful scientific applications and makes quantum computing technologies accessible to a broader community. We will detail

the primary challenges present in these hybrid, noisy quantum computations and discuss best practices for addressing them. We describe the XACC quantum compilation and execution framework that provides a hardware and language independent solution for many of these challenges and supports extensible methodologies that provide a strategy for managing programming concerns across an increasingly diverse set of QPU tools. As an example, we discuss recent demonstrations of scientific applications built on XACC for nuclear physics.

This work is organized as follows: first, we provide a discussion of hybrid classical-quantum computing systems, including high-level discussions of requisite programming and execution models. We then discuss the near-term challenges present in programming, compiling, and executing hybrid scientific computing applications. Afterwards we introduce the XACC quantum programming framework and detail how it addresses these unique challenges. Finally, we conclude by demonstrating its utility with the example of computing the binding energy of the deuteron bound state.

II. Hybrid Computing Systems

We broadly define hybrid computing systems as a class of abstract machine models that combine different computational paradigms. However, we specialize our analysis to the case of realizable architectures that integrate a conventional classical Turing machine with a quantum Turing machine. Research into these different machine models has clarified that they are not equivalent with respect to computational power [7], and we address some of the unique considerations that arise by using the conventional model to program and control the quantum model.

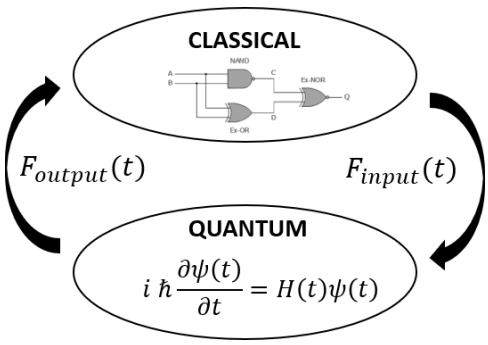

Fig. 1. A hybrid computing infrastructure that integrates classical systems of Boolean logic into the control of quantum dynamics requires an interface for input-output (IO) based on the transmission and reception of physical fields $F_{input}(t)$ and $F_{output}(t)$.

As shown in Fig. 1, the two-way interaction between a classical and a quantum machine model imposes a constraint that requires the machines to share a common understanding of language. In practice, this language is expressed as the physical fields that define the operations issued by the conventional model and implemented within the quantum domain [8]. Electric, magnetic, and optical fields are prominent examples by which the current technology controls the quantum physical systems. It is notable that this description is inherently analog due to the continuity of the time-dependent Schrodinger equation. Translation between the system of Boolean logic characterizing the conventional Turing machine into the analog fields is necessarily limited by the available computational power. In practice, this amounts to constraints imposed by the available digital-to-analog converters and the range of the arbitrary waveform generators as well as the speed at which the logical network can be processed and the connectivity of the control system.

The outstanding concern for programming such a hybrid computing system is the controllability of the conventional and quantum machines [9], [10]. That is to say, for those scenarios in which the quantum machine must execute a series of issued instructions, how accurately are these instructions realized and how precisely does the result reflects the effect of the intended instructions?

A. Client-Server Model for Hybrid Computing Systems

Current state-of-the-art hybrid computing systems integrate existing CPU-based clients with QPU-based servers [11], [12]. The latter represent the online availability of a programmable infrastructure to access the field generators that drive the control of a quantum physical device. The applied fields are shaped and scheduled to control the dynamics of an addressable array of quantum physical subsystems, which for convenience we denote generically as the quantum register. Similarly, the response fields emitted by the register elements are collected and discriminated to generate binary representations that characterize the state of the register.

A QPU-based server often includes conventional CPUs for purposes of parsing the digital programming instructions that generate the shape and timing of the control fields as well as the detection and discrimination of the response fields. Access to the QPU-based server requires an interface that may adhere to conventional logic, for example, as found in modern networking communication technology, to accept instructions from and return results to a CPU-based client. Currently, the conventional client-server model shown in Fig. 2 dominates the access method to QPU systems due in large part to the experimental nature of these machines.

Within the client-server model for hybrid computing systems, the QPU represents a layer of language parsers that translate the Boolean logic of the client to the control fields required to drive the quantum register. As detailed elsewhere [8], the QPU is partitioned into a control unit, execution units, and the register itself. The control unit parses the client instructions received by the server into the local instruction set architecture for generating and applying control fields. Application of the fields are carried out by the execution units, which in practice represent waveform generators for electric, magnetic and optical fields. A similar signal flow occurs for detecting output fields and generating a digital response.

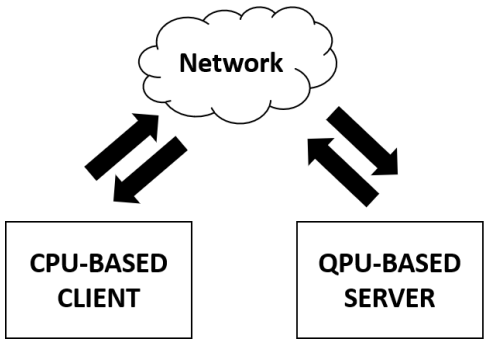

Fig. 2. Currently, most hybrid computing systems rely on a CPU-based client interacting with a QPU-based server over a network. The QPU-based server presents a classical control interface to the client that can be customized to a variety of programming styles. Internally, the QPU-based server must parse these instructions into a local representation that carries out the requested sequence of control fields.

B. Programming and Execution Model

The client-side API for the hybrid computing system dictates how users access the quantum physical devices, while the functionality of the QPU-based server is restricted to the local interpretation of these transmitted instructions. In this programming model, a client may select which instructions to send from a predefined instruction set for the QPU. The instruction set architecture (ISA) defines the functionality of the QPU and the language for the control unit [13]. These instructions are generally transmitted as character strings that are mapped by the QPU-based server into pre-compiled functions that execute the instruction. Calls to these libraries initiate the cascade of logic required to trigger the execution units, which subsequently apply the necessary fields to the register elements.

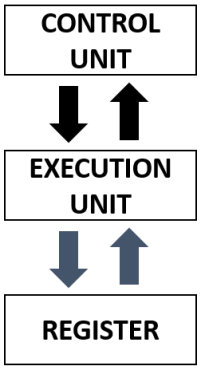

Fig. 3. Components in a QPU-based server include a control unit that expresses the instruction set architecture (ISA), multiple execution units that translate instructions into applied fields, and the quantum register which stores the computational state. This layered, hierarchical structure provides a natural separation of concerns but introduces challenges to programming near-term, noisy QPUs.

As shown in Fig. 3, the two-way information flow within the QPU-based server provides a natural separation of concerns for device programming. The logic dictating computational functionality is isolated within the control unit and determined by programs expressed within the ISA. Similarly, the execution unit hosts the translation of individual instructions into the fields that carry out the intended logical transformations on the register. The information unique to the implementation of either layer is notionally not required in the opposing layer. However, as we explore in the subsequent section, this design of the current QPU-based servers leads to several challenges for application programming.

III. Programming Challenges for Noisy QPUs

Despite the natural separation of concerns that arises in a hybrid computing system using a QPU-based server, this design presents a number of challenges when programming currently available noisy quantum processors. Advances in our understanding and engineering of QPUs have enabled demonstrations with register capacities up to 20 addressable elements, yet these realizations still demonstrate a non-trivial amount of noise relative to the underlying decoherence rates, greatly limiting the depth of quantum circuits with even modest reliability. This raises a need for repeated sampling of the circuit execution, but existing client-server access models hamper this interaction and greatly limit the overall utility and accessibility of the QPUs.

Code portability is also a growing concern as many QPU vendors trend toward standalone programming frameworks that interact with their tightly controlled and proprietary ISA. These barriers to portability impede efforts to perform verification of software and benchmarking of hardware as well as methods to validate hardware behavior using numerical simulation. Finally, the individual workflow steps required to program, compile, and execute hybrid applications with near-term QPUs are tightly coupled. The lack of inter-operability across available software and hardware platforms raises a concern that the users programming hybrid computing systems will face artificial barriers to adoption and performance.

We provide a more detailed description of each of these challenges with an analysis for how hybrid computation is affected. While several references provide background for additional technical details, we identity how each of these challenges must be overcome in order to advance utilization of noisy QPU systems.

A. Gate Noise and Execution Errors

In a QPU, gate noise represents a lack of control over the fundamental physical processes by which program instructions are executed. Among many measures of instruction accuracy, the quantum state fidelity quantifies the accuracy by which an observed state meets the instruction design requirements. For this characterization, a noisy gate applied to a quantum register induces a logical transformation to a state which is some distance away from the expected outcome. Assuming only pure states, the fidelity is defined as the squared magnitude of the inner product between the observed and expected register states ($F(\rho,\sigma) = |\langle\psi_\rho|\psi_\sigma\rangle|^2$) and, for imperfect gates, it is always less then unity. A similar measure can be defined for mixed states in terms of the trace distance.

Noisy gate operation obviously influences program execution by diverting the computational state away from the intended algorithmic design. Quantum error correction and fault-tolerant gate protocols may be expected to mitigate this noise eventually, but such techniques are beyond the scope of current hardware devices. Instead, programming noisy QPUs must directly address the presence of errors within the application logic. This may be as simple as repeated execution of the program as a method of sampling the computer outputs, or it may be a more sophisticated redesign of the compiled circuit to mitigate against known errors. For such methods, the burden of understanding the noise that arises from the execution of these instructions lies on the programmer. However, as noted in the previous section, the separation of concerns for current QPU-based servers isolates the physics of the execution unit and register from the user. The ISA alone provides the interface for programming and controlling the computational logic.

So far the application developers have relied on external characterization of the gate noise outside of the programming workflow. This requires consideration of errors during the algorithmic design stage which is largely based on manual analysis [14]. Such strategies are untenable as circuit complexity increases. Programming models that are device-aware are currently lacking, but would be necessary to automate circuit rewriting techniques to compensate for noisy gates.

In addition to the effects of gate noise, execution errors for a quantum program also arise from state-preparation and measurement (SPAM) errors. These errors correspond to faulty initialization or measurement of a register element. For example, an instruction to initialize the computational state $|0\rangle$ may inadvertently prepare the state $|1\rangle$ or a superposition of these two possibilities. Similarly, measurement of the state $|0\rangle$ may instead project into $|1\rangle$, which would be interpreted as the 1. It is notable that SPAM errors currently dominate QPU performance in multiple technologies with error rates nearly an order of magnitude above typical gate errors.

Several strategies exist for mitigating SPAM errors such as using a series of repeated program executions to decode the correct result based on the maximum likelihood statistics. However, this example of statistical detection based on estimates of the output measurement values not only adds to the complexity of the program execution but also requires accurate characterization of the error mechanisms. Because of the relatively high SPAM error rates, a large number of samples are typically required to gain high confidence in the program behavior. But the separation of concerns between the physical and logical layers in the QPU details hides these physical errors from the programmer. Inline calibrations for initialization and measurement gates may be able to bridge this gap when the number of register elements is very small and the errors are independent, but cross-talk during measurement may invalidate the latter model. Moreover, such calibrations are intractable as the size of the register reaches ever larger sizes.

B. Access Models

The overall system infrastructure required to operate current QPU systems is based largely on sensitive, experimental devices that cannot be easily distributed. Many vendors and laboratories therefore enable users to access these complex computing systems via remote, online server. Examples of these remote access models include QPU-based servers that support REST APIs that delegate requests to a job queue service, which then schedules program execution on the QPU. In addition, many systems provide web portals that permit manual input methods for programming. These may also be operated via the REST API or Pythonic frameworks by creating batch-style program executions.

This remote user access infrastructure suffices for small-scale program executions, but production-level computing, including scientific computing applications based on quantum acceleration, are not amenable to remote access models. As described in the subsequent sections, many current demonstrations of hybrid variational quantum algorithms employ repeated program executions [15]–[18]. These methods require many consecutive serial executions in which each execution influences the next iteration of the algorithm. A remote network connection is therefore an impractical access model for these types of iterative hybrid algorithms because of the additional overhead associated with the remote calls, the communication parsing, and the queuing latency.

A related challenge for QPU-based server programming is that the job queue service employed by most servers is based on individual serial QPU executions. This introduces a hardware bottleneck for the variational sampling algorithms that use multiple circuit executions to estimate a single observable. Related circuit executions are not collated within the queue, which slows down the overall application execution time. In contrast, job schedulers for typical high-performance computing systems operate by queuing a complete application execution. That is to say, when a job reaches the top of the queue, the associated applications are executed completely to remove unnecessary uncertainty in both the application and machine state.

C. Portability

A persistent concern for any application developer is the portability of existing code onto new platforms. Despite the major conceptual shift offered by the quantum computational model, this concern is also faced by the nascent hybrid application developer community. Many QPU vendors support standalone programming frameworks that interact only with their tightly controlled and proprietary ISA. Although tight control over the system may offer advantages to the vendor, a major disadvantage to users is a need to retool for each QPU. Efforts to retool often slow down overall productivity and may, eventually, impede efforts to adopt new software or hardware.

The lack of portability also presents a challenge for verification of software and benchmarking of hardware as well as for validation of the hardware behavior based on numerical simulations. In particular, there is a concern that test cases

and benchmarks devised for one hardware platform will be incompatible with similar efforts developed for other platforms. In essence, developers cannot begin to address quantum benchmarking concerns without some form or mechanism of quantum code portability. Code verification and program validation represent important steps in certifying an application as correct. The above challenges for current quantum programming tools directly impact the ability for application developers to perform verification and validation. Noise and errors undermine the testability of the application, while non-portable codes challenge the ability to use different platforms for comparative analysis.

D. Classical-Quantum Integration

Primarily due to the remote access models employed and an overall lack of code portability, currently available quantum computing resources lack direct integration with conventional software and processing workflows. This integration is not necessary for experimental proof-of-concept demonstrations, in which the focus is on manual, device-level benchmarking and validation of the QPU physical behavior [19]. But integration is necessary to make such devices available for development and testing of classical-quantum hybrid algorithms. Currently, these interactions are largely avoided or treated as separate stages of a manual workflow.

There are several efforts underway to integrate quantum and classical workflows. Examples of domain-specific languages have appeared including Quipper [20], ProjectQ [21], Liquid [22], and Q# [23] among many others. However, these languages target programmers with detailed knowledge and understanding of quantum logic, especially in the gate or circuit model. The integration of these languages with existing programming methods, including well-known languages such as C/C++ and Python, are established by using externally reference library functions or embedding in a host language. In both cases, the understanding of quantum functionality and device behavior is ambiguous to the high-level language user and challenges the understanding of information flow and error analysis needed for profiling an application on a noisy QPU.

IV. Quantum Compilation Workflow

The quantum compilation workflow, like any compiler workflow, can be decomposed into discrete steps that, at a black-box level, read in quantum source code and produce machine-level instructions for execution. However, along the way, many individual tasks need to be executed in order to ensure that the computation is amenable for the chosen hardware, is resource-efficient, and is at least partly resilient to system noise and errors. This includes allocation of quantum registers and instruction scheduling that minimizes errors and noise, as well as pre- and post-processing of erroneous inputs and outputs.

These individual compiler layers for quantum computation must be tightly coupled, i.e. the layers closer to the high-level source code directly influence layers at a lower level closer to the hardware. Compilation layers can have a multiplicity greater than 1, meaning that multiple processes of the same type may be executed for a given layer, and these processes may or may not commute. Also, various compilation layers may dictate whether or not any post-execution actions must take place.

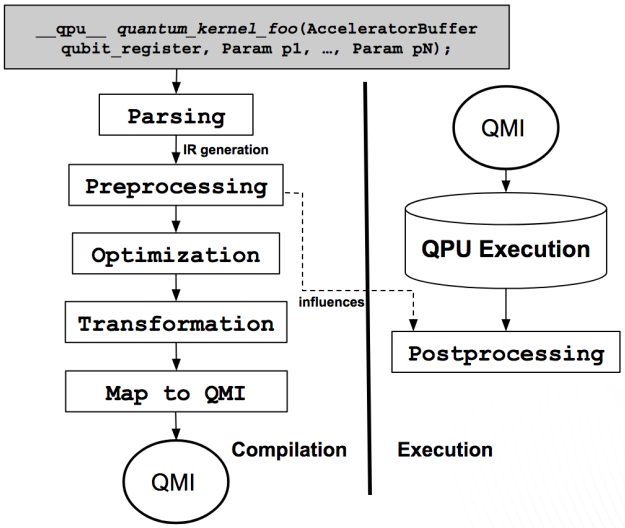

Fig. 4. The XACC compilation workflow is a series of processing layers amenable to near-term quantum computer programs. These layers include language parsing, source code pre-processing, optimization, and transformation. For near-term program behavior, the pre-processing layer is strongly tied to the post-processing methods that follow QPU execution, e.g., SPAM error mitigation strategies.

We provide working examples of the layers required by the compilation workflow for noisy QPUs in Fig. 4. The first step in this workflow is source code parsing, which maps quantum code to an intermediate representation (IR) of the input program. This representation then passes through a compilation layer that provides generic pre-processing of the intermediate representation. The subsequent two layers target IR optimization and code transformation. Optimization may include simplification and enhancements to the programmed quantum logic which is isomorphic with the original program, while transformations modify the intermediate representation structure to satisfy logical constraints exposed by the hardware topology. The next layer maps the IR to the low-level quantum machine instructions (QMI) for the target hardware. The resulting representation is an executable that will be executed on the QPU-based server. Following execution, additional post-processing steps may be included to take advantage of the pre-processing layer.

These various layers of the compilation workflow must be tightly coupled, ideally through the use of a common IR to minimize overhead. Pre-processing directly influences the optimizations and transformations that may be implemented as well as the post-analysis of the QPU measurement results. At each compilation layer, multiple invocations of individual, yet different, pre-processors, optimizers, or transformations, and the execution of these processes may not commute with one another. For example, one preprocessor may update the IR

instance in a way that makes a future pre-processor execution invalid. Maintaining logic across these different instances of the IR is a challenge for evaluating near-term noisy QPUs.

V. XACC

We have put forth a new hybrid classical-quantum programming model, XACC (eXtreme-scale ACCelerator), that attempts to address many of the challenges discussed above [24]. XACC has been specifically designed for enabling near-term quantum acceleration within existing classical computing applications and workflows. This model, and associated open-source reference implementation, follows the traditional classical co-processor model, akin to OpenCL or CUDA for GPUs, but takes into account the subtleties and complexities inherent to the interplay between classical and quantum hardware. XACC provides a high-level API that enables classical applications to offload work represented as quantum kernels to an attached quantum accelerator in a manner that is independent of both the quantum programming language and the quantum hardware. This enables one to write quantum code once for a given model of quantum computation (adiabatic or gate), and perform benchmarking, verification and validation, and performance studies for a set of virtual (simulators) or physical quantum hardware. Here we detail the XACC architecture and in subsequent sections describe how it begins to mitigate against the aforementioned hybrid programming challenges.

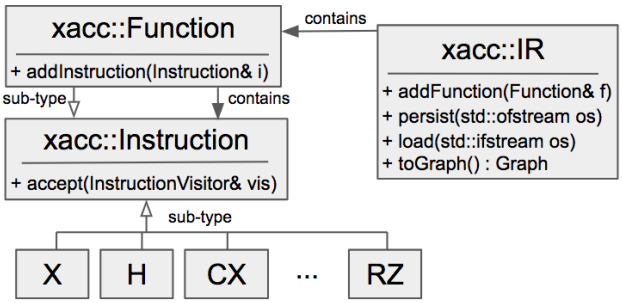

Fig. 5. The XACC intermediate representation architecture demonstrating the relationship between the Instruction, Function, and IR interfaces.

A. Intermediate Representation

Suppose one wishes to program a quantum algorithm in language A and target QPU B. This necessitates the development of a compiler that maps A to the native assembly understood by B. Now suppose that one wishes to benchmark, profile, or otherwise validate the program across a variety of QPU types. This would require the development of N QPU compilers, one for each QPU. Furthermore, if that same user would like to also vary the programming language used we would require $N * M$ compiler implementations, for M available languages.

In classical compiler theory this $N * M$ scaling is overcome through the design and implementation of a robust *intermediate representation* (IR) for available classical instruction sets. This IR sits at a slightly higher level of abstraction than concrete instructions sets, and therefore enables a type of polymorphism across various instructions sets. With an available IR, global compiler implementations can be decomposed into extensible frontend, IR transformation/optimization, and backend components. A prime example of this type of architecture is the LLVM compiler infrastructure, which defines a classical IR that enables the mapping of C++, Fortran, Objective-C, etc. to x86, ARM, MIPS, etc. [25]. Extensions of the frontend map source languages to the IR, and extensions of the backend map the IR to native instruction sets. In this way, mapping M languages to N target hardware types requires M compiler frontend extensions and N backend extensions, but not $N * M$ total compiler implementations.

We are seeing something similar currently in the quantum programming landscape. The recent emergence of various quantum programming mechanisms such as formal languages, domain specific languages, and Pythonic frameworks and libraries, coupled with the emergence of a number of quantum computing hardware types has introduced major issues with regards to application portability, unified compilation strategies, and current and future benchmarking activities. Mapping one language or programming mechanism to a hardware type that it was not directly implemented for introduces a high-cost for scientific quantum-classical application testing and benchmarking.

To alleviate these challenges, XACC builds off of approaches from classical compiler theory and defines an polymorphic intermediate representation for quantum computing that enables the mapping of various programming approaches to available quantum computing backends (physical or virtual). The XACC IR is the key architectural design that enables the integration of a number of high-level and low-level programming abstractions targeting gate and adiabatic quantum computing model implementations. The IR is composed of three core interfaces, shown in Figure 5. First, the Instruction interface abstracts the notion of an operation to be executed on a quantum computer. Instructions have a unique name (such as Hadamard, CNOT, DW-QMI, etc.) and keep reference to the quantum bit indices that they operate on. Instruction can also be parameterized with one or many parameters, modeling gate instructions such as general qubit rotations about a given axis. Next, the Function interface is itself an Instruction sub-type but also contains a list of further Instructions. This design forms a familiar *n-ary* tree of Instructions, and enables executions, transformations, and optimizations of this tree via pre-order traversal. Finally, the IR interface serves as a container for Function instances, and provides methods for serialization and producing graphical representations of the compiled quantum program (quantum circuit graph or Ising Hamiltonian graphs for gate and adiabatic models, respectively).

B. Kernels and Compilers

The XACC programming model follows the familiar co-processor model leveraged in heterogeneous CPU-GPU programming. XACC requires users to express code intended for quantum computation as *quantum kernels*, similar to CUDA

```
__qpu__ quantum_kernel_foo(AcceleratorBuffer
qubit_register, Param p1, …, Param pN);
```

Fig. 6. A typical XACC quantum kernel definition.

and OpenCL kernels for GPUs. XACC kernels are classic C-like functions that have a unique function name and a typical function argument structure (see Figure 6). XACC kernels must take as their first argument the buffer (for more on the AcceleratorBuffer, see [24]) of qubits that the kernel is to operate on, and then any further runtime parameters (such as parameters in a variational algorithm). Furthermore, XACC kernels must be annotated with the *__qpu__* attribute which can enable ahead-of-time compilation of hybrid quantum-classical source code (a topic for future work).

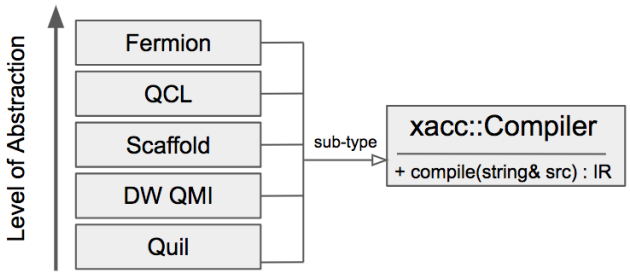

Fig. 7. The XACC Compiler interface mapping languages of varying degrees of abstraction to the XACC IR.

The XACC kernel function body can be written in any quantum programming language supported by the framework. Supported languages are those that provide a Compiler implementation. The Compiler interface makes up the XACC compilation frontend component discussed in the previous section by providing a way to map general kernel source code to the XACC IR. The generality of this interface allows for the inclusion of a hierarchy of quantum programming language abstractions (see Figure 7). Compilers can be implemented for translating low-level quantum assembly to IR, thus enabling low-level source-to-source translation and thereby overall hardware interoperability. Compilers can also be implemented for translating high-level constructs such as problem-specific Hamiltonians to low-level quantum gate instructions. Currently, XACC provides support for a number of low-level quantum programming languages including Scaffold, Quil, D-Wave QMI, and OpenQasm, as well as high-level domain-specific languages encoding second-quantized fermionic Hamiltonians.

C. Operations on the Intermediate Representation

When decomposing a compiler workflow into extensible frontend and backend components via an intermediate representation, it is often useful to provide a middle layer that executes an extensible set of transformations and optimizations on the IR before backend execution. The XACC infrastructure provides such a compile step via the IRTransformation interface definition. This interface takes as input IR instances and outputs a new IR instance whose execution is isomorphic to the previous IR, just simplified our otherwise transformed.

Furthermore, XACC defines an IRPreprocessor interface that is executed before any transformations or optimizations and enables custom modifications to the IR in a non-isomorphic way. This can be useful for error mitigation strategies where one usually requires extra information from the QPU in the form of further circuit executions.

Finally, XACC defines a visitor pattern on the IR tree that enables the mapping of the IR to hardware native gate sets and virtual (simulation) execution [26]. This design pattern enables a dynamic, double-dispatch mechanism that adds virtual functionality to types in the hierarchy dynamically and at runtime. It enables the translation of the XACC IR into native instruction sets via a pre-order tree traversal of the IR. Iterating through the IR nodes and invoking this visitor mechanism invokes the node-specific `visit` method. These methods affect the generation of node-specific, native machine code. In this way, one can see a clear mechanism for source-to-source translation. Translation between languages requires the translation of the source language to the XACC IR, which can then be translated to the target language. The first translation (code to IR) is achieved through implementations of the Compiler interface, while the final translation is achieved through implementations of this visitor pattern. In this way, one is able to translate between Scaffold and Quil, for example, enabling true application portability across various hardware types.

D. Accelerators

Finally, the backend component of XACC is composed of the Accelerator interface, which provides a way to read in instances of the generally pre-processed, transformed, or optimized IR instance and affect execution on the backend physical or virtual QPU. Most Accelerator implementations delegate to vendor supplied APIs for the execution of compiled quantum programs expressed in the native gate set. Accelerators also provide any required transformations on the IR that must be executed before overall Accelerator execution. This provides a mechanism for the mapping of compiled IR to a format that is amenable for execution on the given hardware type (such as program-to-qubit connectivity checks).

Currently, XACC provides unified access to the Rigetti Forest infrastructure (simulators and physical QPUs), the IBM Quantum Experience (simulators and physical QPUs), and the D-Wave quantum annealer infrastructure (simulators and physical QPUs). This breadth of hardware types amenable for XACC compilation and execution is enabled through implementations of the XACC IR interfaces for both gate and adiabatic model quantum computation.

VI. Mitigating Programming Challenges with XACC

A. XACC and Portability

Overall code portability is at the heart of the XACC design. XACC is the first platform to provide a robust and polymorphic intermediate representation object model that enables programming across quantum computing models (gate or adiabatic). This IR allows XACC to be language-agnostic, such that a code written in one language can be compiled to an IR instance and then mapped to a representation amenable for execution on a completely different QPU. This mapping step can be accomplished with user-specified IR transformation implementations. For example, imagine one writes quantum code for one architecture with a given qubit connectivity and wants to run it on another architecture with a different connectivity structure. XACC handles the execution of appropriate transformations on the IR that insert swap gate instructions to insure that two qubit interactions can be executed on the new architecture. This general IR transformation infrastructure makes sure that true code portability can be achieved in the case of hybrid classical-quantum computing. It provides a mechanism for future benchmarking of quantum computers via a number of different application types.

B. XACC and Error Mitigation

The XACC interfaces can also be leveraged to provide certain error mitigation strategies. Error mitigation schemes involve some sort of classical pre-processing of the quantum program, followed by execution and post-processing of the result based on the pre-processing action. As classical pre-processing can be achieved through an implementation of the IRPrerocessor interface implementation, the error mitigation workflow fits in very well with the XACC IR infrastructure. The IRPreprocessor interface takes an IR instance as input, pre-processes or otherwise modifies it, and then outputs a functional instance of some post-processing step that is stored by the XACC framework and executed after QPU execution. This is an ideal setup for mitigating the QPU qubit readout errors discussed in Section III-A as a readout characterization IR can easily be defined on the set of qubits participating in the computation. To this end, XACC provides a pre-processor implementation that prepends the IR instance with additional quantum kernels preparing classical bit strings in order to characterize the bit flip error rates [4]. Those results are stored and used by a post-processor functional instance, which is applied to the qubit measurement results after execution. This post-processor uses the bit flip probabilities to shift and scale resultant observable expectation values to more accurate values.

Clearly the existence of a standard intermediate representation aides in the mitigation of certain types of noise and errors. We classify these errors into hardware dependent and independent types. Hardware independent errors are especially well suited for mitigation at the IR level due to the fact that the IR spans the available set of QPUs. Qubit readout measurement errors are a prime example in this regard, and preprocessing, transformation, and postprocessing steps on the IR provide a novel way for mitigating these types of errors. Hardare dependent error mitigation strategies can also be handled via implementations of the Accelerator backend, specifically through contributed IR Transformations that are QPU-type aware (see discussion on Accelerator functionality in Section V-D).

This workflow could be applied to other forms of error mitigation, requiring simple interface implementations that preprocess quantum IR and post-process QPU qubit measurement results.

C. XACC and Access Models

XACC enables multiple forms of user access via the host language that the framework is written in and its overall platform, or system context, model. XACC is written in C++, a foundational language that enables bindings to many other programming languages. This language binding extensibility facilitates user access to available QPUs since users are not tied to a language they are not familiar with. As of this writing, XACC provides Python bindings and has planned support for Fortran due to its wide adoption in classical high-performance scientific computing.

XACC implements a client-server model designed to support both remote and local user access models, as discussed in Section II-A. Implementations of the XACC accelerator construct are currently available for QPUs with access to a REST client that enables HTTP POST/GET operations to affect execution of quantum programs on the remotely hosted service. The XACC client-server model also enables access to a local access model by using service invocations that can be routed to the local host computing system. XACC is currently ready to enable local QPU-access models simply by redirecting the accelerator URL to the locally hosted QPU driver server, or further Accelerator implementations that leverage available in-memory, in-process QPU APIs.

D. XACC Compilation Workflow

The XACC compilation process directly implements the workflow layers described in Fig. 4. The XACC Compiler interface processes input quantum kernels and returns a representative IR instance. The IR instance may then be passed through any requested IR Preprocessors - which could implement readout-error mitigation, or map logical qubits to more suitable physical qubits, for example. This layer is extensible for future preprocessing implementation steps, and takes as input an IR instance and produces a new post-processing function instance that is queued up for execution after QPU execution.

Next, the IR is passed to any requested optimization routines. An example implementation for optimization is a mechanism (i.e. a set of rules) reducing the IR complexity by removing redundant instructions via CNOT cancellations and compositions of local rotations [27]. Next, transformations are executed that make the program amenable for execution on the

hardware (to ensure requested two-qubit gates are available in hardware, and, if not, apply swap gate transformations, for example). Finally, the IR is mapped to the appropriate low-level machine instruction set implemented in hardware. IR Transformations enable the XACC instruction scheduling and layout steps in that they enable the efficatious assignment of resources with respect to gate fidelities and overall connectivity.

This compiled result is sent off for execution on the QPU, and the resultant bit strings are brought back and passed through any post-processing functional instances produced by the preprocessing layer. This can be, for example, a post-processor that updates observable expectation values based on error probabilities added to the overall computation via a previous IR Preprocessor.

VII. Application Examples

In this section, we review a recent example application programmed and executed using XACC that was hardware agnostic and implemented the various layers of the quantum compilation workflow. This example made use of the variational quantum eigensolver (VQE) algorithm, which relies on the variational principles of quantum mechanics to find the minimal energy quantum state under a given Hamiltonian. The algorithm is relatively simple and has the advantage of permitting even short-depth circuits to address interesting application scenarios. Here we provide a brief overview of this application and detail how XACC addresses the various challenges detailed in this work.

A. Nuclear Binding Energy

We recently undertook the computation of the binding energy of the deuteron via cloud quantum computing resources using the variational quantum eigensolver hybrid algorithm [5]. Using XACC, we were able to program the problem in a hardware-agnostic manner and target available superconducting circuit gate model quantum computers from IBM and Rigetti [28], [29]. Via the compiler workflow discussed in Sections IV,VI-D, we were also able to provide minimal error mitigation that corrected for qubit measurement readout errors. This work represented the first variational quantum eigensolver computation done through a remote access model, and highlighted the need for future local access models that take advantage of application job queues instead of individual QPU execution queues. This remote access model severely hindered the work and demonstrates the need for vendors to research and implement local access models that enhance or enable variational scientific quantum simulation.

We leave the low-level technical details of the deuteron work to [5], but at a high level, we leveraged a pionless effective field theory in a discrete variable representation using the familiar harmonic oscillator basis. We considered cutoffs of that basis at $N = 2, 3$. Here we discuss the $N = 2$ case for brevity, which employed the following Hamiltonian

$$H_2 = 5.906709I + 0.218291Z_0 - 6.125Z_1 - 2.143304\left(X_0X_1 + Y_0Y_1\right). \quad (1)$$

Dictated by this Hamiltonian, we performed measurements of our QPU after application of a unitary coupled cluster circuit composed of a single variational parameter θ. Computing the ground state energy required looping over various θ parameters as part of a classical non-linear optimization scheme until convergence criteria were met.

Listing 1
XACC Kernels for Deuteron VQE

```
__qpu__ ansatz(AcceleratorBuffer b,
                double t0) {
    X 0
    RY(t0) 1
    CNOT 1 0
}
__qpu__ z0(AcceleratorBuffer b, double t0) {
    ansatz(b,t0)
    MEASURE 0 [0]
}
__qpu__ z1(AcceleratorBuffer b, double t0) {
    ansatz(b,t0)
    MEASURE 1 [1]
}
 __qpu__ x0x1(AcceleratorBuffer b, double t0) {
    ansatz(b,t0)
    H 0
    H 1
    MEASURE 0 [0]
    MEASURE 1 [1]
}
__qpu__ y0y1(AcceleratorBuffer b, double t0) {
    ansatz(b,t0)
    RX(1.57079) 0
    RX(1.57079) 1
    MEASURE 0 [0]
    MEASURE 1 [1]
}
```

The code for this work was written in as XACC quantum kernels in the Quil language from Rigetti [30] and is shown in Listing 1. Through XACC, this code was immediately portable to IBM (as well as a number of simulators), thus overcoming the portability challenge for near-term quantum programming and computation.

One common source of error discussed in Section III-A are due to systematic errors in reading out the state of an individual qubit. These types of errors were discussed in the supplemental information of [4]. For this work we automated this error mitigation strategy as part of the XACC compiler workflow. We implement the XACC IR Preprocessor extension interface to append measurements of each qubit that provide probabilities that the qubit was in a state of 0 when a 1 was expected, and vice versa. This IR Preprocessor implementation then returns a post processing function instance that is executed after QPU execution that leverages these probabilities to shift and scale observable expectation values. The plot in Figure 8 shows the energy as a function of the variational parameter, with and without this readout error mitigation preprocessor execution. Clearly, automating this sort of error mitigation will provide more reliable results with minimal costs to those adopting quantum computing as part of their

scientific computing workflows.

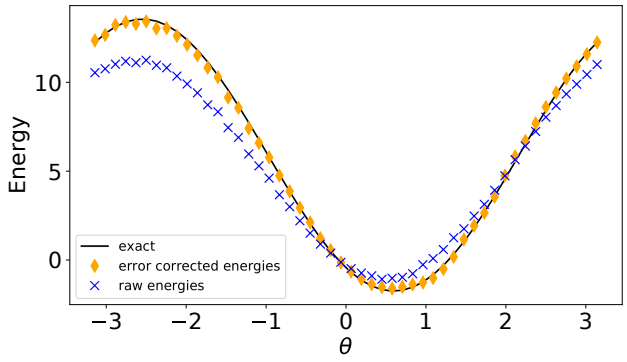

Fig. 8. Energy as a function of the variational parameter for the deuteron $N = 2$ Hamiltonian with and without readout error correction.

B. Higher Levels of Abstraction

The previous section demonstrated the programmability of the $N = 2$ deuteron problem through XACC quantum kernels written in the low-level Quil assembly language. However, XACC Compilers enable the expressibility of problems at higher levels of abstraction as well. Here we take the same problem and program it in a language parseable by a Compiler implementation that abstracts the notion of a fermionic second quantized Hamiltonian.

We can express the above deuteron problem as the following second quantized fermionic Hamiltonian

$$H_2 = -.4365811 a_0^\dagger a_0 - 4.28660705 a_0^\dagger a_1 - 4.28660705 a_1^\dagger a_0 + 12.25 a_1^\dagger a_1. \quad (2)$$

We have developed a Compiler implementation (the Fermion-Compiler [31]) that reads in quantum kernel source code that represents the above Hamiltonian (see Listing 2). Each line of this language represents a single term in the Hamiltonian. Each line begins with the term coefficient followed by pairs of integers indicating the operator site and whether it is a creation or annihilation operator (a 1 or 0 respectively). This Compiler maps the fermionic representation to spins via the Jordan-Wigner or Bravyi-Kitaev transformation, and produces the equivalent XACC quantum kernels shown in Listing 1.

Listing 2
XACC KERNEL FOR DEUTERON VQE TARGETING FERMIONCOMPILER

```
__qpu__ H2(AcceleratorBuffer b) {
    -0.43658111 0 1 0 0;
    -4.28660705 0 1 1 0;
    -4.28660705 1 1 0 0;
    12.25       1 1 1 0;
}
```

C. A Portable API

The previous section provide the reader with examples of the extensibility of XACC with regards to error mitigation and overall quantum program levels of abstraction. Here we seek to demonstrate the portability of the framework via its application programming interface (API). The programming, compilation, and execution of quantum kernels in C++ is demonstrated in Listing 3 for a general quantum kernel `foo`.

Listing 3
PORTABLE XACC API

```
auto src = R"src(
__qpu__ foo(AcceleratorBuffer b,
            double t) {...}) src";

// Get the target Accelerator
auto qpu = xacc::getAccelerator("ibm");
auto qbits = qpu->createBuffer(3);

// Compile the src against qpu
xacc::Program p(src,qpu);
p.build();

// Get executable lambda
auto kernel = p.getKernel<double>("foo");

// Loop over parameterized
// compiled kernel
for (auto& t : thetas) kernel(qbits, t);
```

Note the generality of this API and its portability to available accelerators. Users begin by defining their quantum algorithm as an XACC kernel, and then get reference to the desired Accelerator (here the IBM Accelerator targeting the remote IBM Quantum Experience). Then an allocation of qubits is requested, and Program object is constructed and built, which kicks off the XACC compilation workflow. This includes mapping the source code to the XACC IR and executing all preprocessors, optimizations, and transformations. Users can then get reference to a lambda or functor that affects execution of the compiled result on the requested Accelerator. This code snippet is general and portable to available backend Accelerators. The name of the desired Accelerator can even be elevated to a command line option that one can modify at runtime.

VIII. CONCLUSION

Hybrid computing systems offer novel platforms to integrate emerging QPU with conventional programming methods. However, there are several challenges that arise from these noisy devices whose performance not yet well understood. In this contribution, we have outlined many of the technical issues faced by quantum program developers adopting to client-server model for remote access across multiple technologies and vendors. In addition to new needs for device-level information, current programmers also face obstacles in code portability, tool integration, program validation, and workflow development.

In the context of these challenges, we have described how the XACC programming framework provides new methods for inter-operable program and tool development as well as support for new access client-server access models based on local QPU systems. The framework itself is hardware agnostic and, therefore, meant to provide a generalized approach to

quantum programming. This contrast with the diversity of vendor-specific stacks and domain-specific languages under-development. We anticipate that both efforts, specialized and generalized, are needed to ensure strong and robust growth of the quantum computing ecosystem. The ongoing co-design of hardware and software to mitigate gate noise and execution errors will continue to require close coordination.

ACKNOWLEDGMENT

This work has been supported by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory, the US Department of Energy (DOE) Office of Science Advanced Scientific Computing Research (ASCR) Early Career Research Award, and the DOE Office of Science ASCR quantum algorithms and testbed programs, under field work proposal numbers ERKJ332 and ERKJ335. This work was also supported by the ORNL Undergraduate Research Participation Program, which is sponsored by ORNL and administered jointly by ORNL and the Oak Ridge Institute for Science and Education (ORISE). ORNL is managed by UT-Battelle, LLC, for the US Department of Energy under contract no. DE-AC05-00OR22725. ORISE is managed by Oak Ridge Associated Universities for the US Department of Energy under contract no. DE-AC05-00OR22750.

REFERENCES

[1] A. Aspuru-Guzik, W. van Dam, E. Farhi, F. Gaitan, T. Humble, S. Jordan, A. Landahl, P. Love, R. Lucas, J. Preskill, R. Muller, K. Svore, N. Wiebe, and C. Williams, *ASCR Workshop on Quantum Computing for Science*. Department of Energy, June 2015.

[2] J. Carter, D. Dean, G. Hebner, J. Kim, A. Landahl, P. Maunz, R. Pooser, I. Siddiqi, and J. Vetter, *ASCR Report on a Quantum Computing Testbed for Science*. Department of Energy, February 2017.

[3] N. M. Linke, D. Maslov, M. Roetteler, S. Debnath, C. Figgatt, K. A. Landsman, K. Wright, and C. Monroe, "Experimental comparison of two quantum computing architectures," *Proceedings of the National Academy of Sciences*, p. 201618020, 2017.

[4] A. Kandala, A. Mezzacapo, K. Temme, M. Takita, M. Brink, J. M. Chow, and J. M. Gambetta, "Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets," *Nature*, vol. 549, pp. 242–246, Sep. 2017.

[5] E. F. Dumitrescu, A. J. McCaskey, G. Hagen, G. R. Jansen, T. D. Morris, T. Papenbrock, R. C. Pooser, D. J. Dean, and P. Lougovski, "Cloud Quantum Computing of an Atomic Nucleus," *ArXiv e-prints*, Jan. 2018.

[6] P. OâĂŹMalley, R. Babbush, I. Kivlichan, J. Romero, J. McClean, R. Barends, J. Kelly, P. Roushan, A. Tranter, N. Ding *et al.*, "Scalable quantum simulation of molecular energies," *Physical Review X*, vol. 6, no. 3, p. 031007, 2016.

[7] D. R. Simon, "On the power of quantum computation," *SIAM journal on computing*, vol. 26, no. 5, pp. 1474–1483, 1997.

[8] K. A. Britt, F. A. Mohiyaddin, and T. S. Humble, "Quantum accelerators for high-performance computing systems," in *2017 IEEE International Conference on Rebooting Computing (ICRC)*, Nov 2017, pp. 1–7.

[9] K. R. Brown, J. Kim, and C. Monroe, "Co-designing a scalable quantum computer with trapped atomic ions," *npj Quantum Information*, vol. 2, p. 16034, 2016.

[10] H. Corrigan-Gibbs, D. J. Wu, and D. Boneh, "Quantum operating systems," in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. ACM, 2017, pp. 76–81.

[11] S. J. Devitt, "Performing quantum computing experiments in the cloud," *Phys. Rev. A*, vol. 94, p. 032329, Sep 2016. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevA.94.032329

[12] K. A. Britt and T. S. Humble, "High-performance computing with quantum processing units," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 13, no. 3, p. 39, 2017.

[13] ——, "Instruction set architectures for quantum processing units," in *High Performance Computing*, J. M. Kunkel, R. Yokota, M. Taufer, and J. Shalf, Eds. Cham: Springer International Publishing, 2017, pp. 98–105.

[14] K. Temme, S. Bravyi, and J. M. Gambetta, "Error mitigation for short-depth quantum circuits," *Phys. Rev. Lett.*, vol. 119, p. 180509, Nov 2017. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.119.180509

[15] A. Peruzzo, J. McClean, P. Shadbolt, M.-H. Yung, X.-Q. Zhou, P. J. Love, A. Aspuru-Guzik, and J. L. OâĂŹBrien, "A variational eigenvalue solver on a photonic quantum processor," *Nat Commun*, vol. 5, Jul 2014. [Online]. Available: http://dx.doi.org/10.1038/ncomms5213

[16] J. R. McClean, J. Romero, R. Babbush, and A. Aspuru-Guzik, "The theory of variational hybrid quantum-classical algorithms," *New Journal of Physics*, vol. 18, no. 2, p. 023023, 2016.

[17] Y. Li and S. C. Benjamin, "Efficient variational quantum simulator incorporating active error minimization," *Phys. Rev. X*, vol. 7, p. 021050, Jun 2017. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevX.7.021050

[18] Y. Shen, X. Zhang, S. Zhang, J.-N. Zhang, M.-H. Yung, and K. Kim, "Quantum implementation of the unitary coupled cluster for simulating molecular electronic structure," *Phys. Rev. A*, vol. 95, p. 020501, Feb 2017. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevA.95.020501

[19] K. Michielsen, M. Nocon, D. Willsch, F. Jin, T. Lippert, and H. De Raedt, "Benchmarking gate-based quantum computers," *Computer Physics Communications*, vol. 220, pp. 44–55, 2017.

[20] A. S. Green, P. L. Lumsdaine, N. J. Ross, P. Selinger, and B. Valiron, "Quipper: A scalable quantum programming language," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: ACM, 2013, pp. 333–342. [Online]. Available: http://doi.acm.org/10.1145/2491956.2462177

[21] D. S. Steiger, T. Häner, and M. Troyer, "ProjectQ: an open source software framework for quantum computing," *Quantum*, vol. 2, p. 49, Jan. 2018. [Online]. Available: https://doi.org/10.22331/q-2018-01-31-49

[22] D. Wecker and K. M. Svore, *LIQUID: A Software Design Architecture and Domain-Specific Language for Quantum Computing*, 2014, http://arxiv.org/pdf/1402.4467v1.pdf.

[23] K. Svore, A. Geller, M. Troyer, J. Azariah, C. Granade, B. Heim, V. Kliuchnikov, M. Mykhailova, A. Paz, and M. Roetteler, "Q#: Enabling scalable quantum computing and development with a high-level dsl," in *Proceedings of the Real World Domain Specific Languages Workshop 2018*, ser. RWDSL2018. New York, NY, USA: ACM, 2018, pp. 7:1–7:10. [Online]. Available: http://doi.acm.org/10.1145/3183895.3183901

[24] A. McCaskey, E. Dumitrescu, D. Liakh, M. Chen, W. Feng, and T. Humble, "A language and hardware independent approach to quantumâĂŞclassical computing," pp. 245 – 254, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2352711018300700

[25] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 75–. [Online]. Available: http://dl.acm.org/citation.cfm?id=977395.977673

[26] A. McCaskey, E. Dumitrescu, M. Chen, D. Lyakh, and T. S. Humble, "Validating Quantum-Classical Programming Models with Tensor Network Simulations," Jul. 2018.

[27] Y. Nam, N. J. Ross, Y. Su, A. M. Childs, and D. Maslov, "Automated optimization of large quantum circuits with continuous parameters," *npj Quantum Information*, vol. 4, no. 1, p. 23, 2018. [Online]. Available: https://doi.org/10.1038/s41534-018-0072-4

[28] "IBM builds its most powerful universal quantum computing processors." [Online]. Available: https://phys.org/news/2017-05-ibm-powerful-universal-quantum-processors.html

[29] J. S. Otterbach, R. Manenti, N. Alidoust, A. Bestwick, M. Block, B. Bloom, S. Caldwell, N. Didier, E. Schuyler Fried, S. Hong, P. Karalekas, C. B. Osborn, A. Papageorge, E. C. Peterson, G. Prawiroatmodjo, N. Rubin, C. A. Ryan, D. Scarabelli, M. Scheer, E. A. Sete, P. Sivarajah, R. S. Smith, A. Staley, N. Tezak, W. J. Zeng, A. Hudson, B. R. Johnson, M. Reagor, M. P. da Silva, and C. Rigetti, "Unsupervised Machine Learning on a Hybrid Quantum Computer," *ArXiv e-prints*, Dec. 2017.

[30] R. S. Smith, M. J. Curtis, and W. J. Zeng, "A Practical Quantum Instruction Set Architecture," *ArXiv e-prints*, Aug. 2016.
[31] "Xacc-vqe," https://github.com/ornl-qci/xacc-vqe, accessed: 2018-06-21.