SoftwareX 7 (2018) 245–254

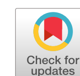
Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

A language and hardware independent approach to quantum–classical computing☆



A.J. McCaskey [a,b,*], E.F. Dumitrescu [a,c], D. Liakh [a,d], M. Chen [e,f], W. Feng [f], T.S. Humble [a,c,g]

[a] *Quantum Computing Institute, Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA*
[b] *Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA*
[c] *Computational Sciences and Engineering Division, Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA*
[d] *Oak Ridge Leadership Computing Facility, Scientific Computing, Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA*
[e] *Department of Physics, Virginia Tech, Blacksburg, VA 24060, USA*
[f] *Department of Computer Science, Virginia Tech, Blacksburg, VA 24060, USA*
[g] *Bredesen Center for Interdisciplinary Research, University of Tennessee, Knoxville, TN 37996, USA*

ARTICLE INFO

Article history:
Received 30 April 2018
Received in revised form 24 July 2018
Accepted 25 July 2018

Keywords:
Quantum computing
Quantum software

ABSTRACT

Heterogeneous high-performance computing (HPC) systems offer novel architectures which accelerate specific workloads through judicious use of specialized coprocessors. A promising architectural approach for future scientific computations is provided by heterogeneous HPC systems integrating quantum processing units (QPUs). To this end, we present **XACC** (*e**X**treme-scale **ACC**elerator*) — a programming model and software framework that enables quantum acceleration within standard or HPC software workflows. XACC follows a coprocessor machine model that is independent of the underlying quantum computing hardware, thereby enabling quantum programs to be defined and executed on a variety of QPUs types through a unified application programming interface. Moreover, XACC defines a polymorphic low-level intermediate representation, and an extensible compiler frontend that enables language independent quantum programming, thus promoting integration and interoperability across the quantum programming landscape. In this work we define the software architecture enabling our hardware and language independent approach, and demonstrate its usefulness across a range of quantum computing models through illustrative examples involving the compilation and execution of gate and annealing-based quantum programs.

© 2018 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

Code metadata

Current code version	v1.0.0
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX_2018_48
Legal Code License	EPL and EDL
Code versioning system used	git
Software code languages, tools, and services used	C++, Python
Compilation requirements, operating environments & dependencies	C++11, Boost 1.59+, OpenSSL 1.0.2, CMake
Link to developer documentation/manual	https://xacc.readthedocs.io
Support email for questions	xacc-dev@eclipse.org, mccaskeyaj@ornl.gov

☆ This manuscript has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The US government retains and the publisher, by accepting the article for publication, acknowledges that the US government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for US government purposes. DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan. (http://energy.gov/downloads/doe-public-access-plan).

* Corresponding author at: Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA.

E-mail address: mccaskeyaj@ornl.gov (A.J. McCaskey).

https://doi.org/10.1016/j.softx.2018.07.007
2352-7110/© 2018 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

1. Introduction

High-performance computing (HPC) architectures continue to make strides in the use of specialized computational accelerators, and future HPC designs are expected to increasingly take advantage of compute node heterogeneity [1]. Quantum processing units (QPUs) represent a unique coprocessor paradigm which leverages the information-theoretic principles of quantum physics for computational purposes. Several small-scale experimental QPUs, including the publicly available IBM quantum computer [2], already exist and their sophistication, capacity, and reliability continues to improve [3]. As a potential HPC accelerator, the emergence of mature QPU technologies requires careful consideration for how to best integrate these devices with conventional computing environments. While the hardware infrastructure for early QPUs is likely to limit their usage to remote access models and state-of-the-art HPC systems [4], there are clear use cases where hybrid algorithms may judiciously leverage both conventional and quantum computational resources for near-term scientific applications [5,6]. A hybrid computing paradigm is poised to broadly benefit scientific applications that are ubiquitous within research fields such as modeling and simulation of quantum many-body systems [7], applied numerical mathematics [8], and data analytics [9].

The generalization of HPC programming paradigms to include new accelerators is not without precedent. Integrating graphical processing units (GPUs) into HPC systems was also a challenge for many large-scale scientific applications because of the fundamentally different way programmers interact with the hardware. Hardware-specific solutions provide language extensions [10] that enable programming natively in the local dialect. Hardware-independent solutions define a hybrid programming specification for offloading work to attached classical accelerators (GPUs, many-integrated core, field-programmable gate array, etc.) in a manner that masks or abstracts the underlying hardware type [11]. These hardware-agnostic approaches have proven useful because they retain a wide degree of flexibility for the programmer by automating those aspects of compilation that are overly complex. Programming models for QPUs will pose additional challenges because of the radically different logical features and physical behaviors of quantum information, such as the no cloning principle and reversible computation. The underlying technology (superconducting, trapped ion, etc.) and models (gate, adiabatic, topological, etc.) will further distinguish QPU accelerators from conventional computing devices. It is therefore necessary to provide flexible classical–quantum programming models and integrating software frameworks to handle the variability of quantum hardware to promote robust application benchmarking and program verification and validation.

Approaches for interfacing domain computational scientists with quantum computing have progressed over the last few years. A variety of quantum programming languages have been developed with a similar number of efforts under way to implement high-level mechanisms for writing, compiling, and executing quantum code. State-of-the-art approaches provide embedded domain-specific languages for quantum program expression. Examples include the languages and tools from vendors such as Rigetti [12], Microsoft [13], Google [14], and IBM [15], which each enable assembly-level quantum programming alongside existing Pythonic code. Individually, these implementations provide self-contained software stacks that optionally target the vendor's unique hardware implementation or simulator backend. The increasing variability in languages and platforms raises concerns for managing multiple programming environments and compilation tool-chains. The current lack of integration between software stacks increases application development time, decreases portability, and complicates benchmarking analysis. Methods that enable cross-compilation for QPUs will support the broad adoption of experimental quantum computing through faster development time and reusable code.

To address these unique challenges, we present a programming model and extensible compiler framework that integrates quantum computing devices into an accelerator-based execution model. The eXtreme-scale ACCelerator (XACC) framework is designed for robust and portable QPU-accelerated application programming by enabling quantum language and hardware interoperability. XACC defines interfaces and abstractions that enable compilation of hybrid programs composed of both conventional and quantum programming languages. The XACC design borrows concepts from existing heterogeneous programming models like OpenCL [11] by providing a hardware-independent interface for off-loading quantum subroutines to a quantum coprocessor. Moreover, XACC enables language interoperability through a low-level quantum intermediate representation.

The structure of this work is as follows: first, we present related work with regards to quantum programming and detail inherent unique challenges that XACC seeks to address; second, we define the XACC software architecture, including platform, programming, and memory models; finally, we detail unique demonstrations of the model's flexibility through demonstrations using both gate and annealing quantum computing models.

2. Related Work

Programming, compilation, and execution of quantum programs on physical hardware and simulators has progressed rapidly over the last few years. During this time, much research and development has gone into exploring high-level programming languages and compilers [16–19]. Moreover, there has been a recent surge in the development of embedded domain specific languages that enable high-level problem expression and automated reduction to low-level quantum assembly languages [12,14,15]. However, despite progress there are still numerous challenges that currently impede adoption of quantum computing within existing classical scientific workflows [20]. Most approaches that target hardware executions are implemented via Pythonic frameworks that provide data structures for the expression of one and two qubit quantum gates; essentially providing a means for the programming of low-level quantum assembly (QASM). Compiler tools provided as part of these frameworks enable the mapping of an assembly representation to a hardware-specific gate set as well as mapping logical to physical connectivity. The arduous task of complex compiler workflow steps, including efficient instruction scheduling, routing, and robust error mitigation are left as a manual task for the user. This hinders broad adoption of quantum computation by domain computational scientists whose expertise lies outside of quantum information.

Higher-level languages exist, but do not explicitly target any physical hardware. Therefore, users can compile these high-level languages to a representative quantum assembly language, but such instructions must be manually mapped to the set of instructions specified by a given hardware gate set. This translation process is often performed by re-writing the assembly code in terms of a Pythonic execution frameworks targeting a specific device. Moreover, high-level languages have in the past assumed a fault-tolerant perspective of quantum computation. However, this interpretation is at odds with practical near-term noisy computations, for which the user must provide robust compilation tools to enable a variety of error mitigation strategies. To this end, domain specific languages enabling problem expression at higher levels of abstraction [21–23] for non-fault-tolerant quantum computing have recently been developed. These represent promising pathways for enabling a broad community of computational scientists to benefit from quantum computation.

Overall, currently available quantum languages and compilers are not well integrated with each other. Research and development efforts that offer quantum programming, compilation, and execution mechanisms often target a single simulator or physical hardware instance (i.e. see Pythonic frameworks above). This leads to poor quantum code portability and disables any effort attempting to benchmark various hardware types (superconducting, ion trap, etc.) against each other. Furthermore, there are currently a number of quantum computing models that various research efforts are targeting. A majority of these efforts are targeting the gate model of quantum computation, while others have implemented a noisy form of adiabatic quantum computation. Moreover, there are other models that researchers are becoming increasingly interested in (one-way, topological, etc.). The differences in computational paradigm and hardware specific gate sets negatively affect code portability, verification and validation, and benchmarking efforts.

Our work seeks to address the drawbacks associated with near-term quantum hardware execution and programming models, thus enabling the integration of quantum and classical computation through an extension of the classical coprocessor-computing model. Our aim is to provide a model framework that is extensible to a wide variety of important, practical quantum programming workflow steps. In this way, we can provide a truly integrating quantum compiler and execution framework that works across quantum computing models, languages, and physical (virtual) hardware types. Our efforts aim to benefit quantum code portability, tightly-coupled quantum access models, and classical–quantum application benchmarking efforts.

3. XACC Architecture

The XACC framework is designed to enable the expression and integration of a wide spectrum of accelerator (quantum) algorithms alongside existing classical code in a manner independent of the accelerator language, hardware, and computational model. Here we define and detail the XACC architecture which we decompose into constituent platform, memory, and programming models. The XACC platform model describes the hardware components at play in the compilation and execution of hybrid programs and how these components behave in relation to one another, while the memory model details the management and movement of data between these components. These model abstractions drive the design and implementation of the XACC programming model, which specifies an application programming interface (API) for offloading computations to an attached quantum accelerator.

3.1. Platform and Memory Model

XACC treats a general quantum processing unit (QPU) as described in Ref. [4], whereby the QPU is composed of a register of quantum bits (qubits), a quantum control unit (QCU), and a classical memory space for the storage of quantum program results. Ref. [4] puts forth a variety of classical–quantum integration strategies that promote a range of quantum accelerated use cases. For example, one could imagine a loosely-coupled coprocessor model with one or many classical compute nodes accessing a single, remotely hosted QPU. On the other hand, one may consider a tightly-coupled coprocessor model, in which one or many compute nodes have access to an in-memory, in-process device driver API for QPU control and execution (no remote access required). There are, of course, many variants on these models that interpolate between the two extremes. The XACC platform model attempts to take this spectrum into account and enable a variety QPU access models, both local and remote.

Building on this QPU definition, we define a classical–quantum platform model that enables a range of quantum integration types via the classic *client–server model* [24], in which programmers of the conventional computing system are on the client side and the quantum accelerator system is on the server side. XACC defines three non-trivial components in this model: (1) the host CPU, (2) the accelerator system (and further sub-types), and (3) the accelerator buffer (see Fig. 1). The host CPU drives the interaction of classical applications with the attached accelerator system by executing classical applications and delegating quantum computer executions to the attached accelerator system. The role of the accelerator system is to listen for execution requests, and then drive the execution of a quantum program compiled according to the vendor-supplied quantum computer API specifications. The accelerator buffer component forms the underlying hybrid classical–quantum memory space sharing the accelerator execution results.

We have designed XACC to facilitate both serial classical–quantum computing and massively-parallel, distributed high-performance computing enhanced with quantum acceleration, therefore, the cardinality of the host CPU component is one to many (1..*). One could have one or many host CPUs as part of a given hybrid execution corresponding to the many cores available in a HPC application. Likewise, one may consider a computation involving multiple quantum coprocessors. In the very near term this will be unlikely due to QPU infrastructure requirements, but given modest hardware advances the collections of modestly sized QPUs could become available to multiple compute nodes, as is the case with GPUs in classical heterogeneous computing. Therefore, the cardinality of the accelerator system is also one to many (1..*). This platform model allows for the inclusion of multiple classical threads having access to multiple accelerators.

The XACC memory model ensures that client-side applications can retrieve quantum execution results through the `AcceleratorBuffer` concept, which models a register of bits and stores ensembles of measurement results. Clients (the host CPU) create instances of the `AcceleratorBuffer` that are then passed to the accelerator system upon execution. It is the responsibility of the accelerator system to keep track of all measurement results and

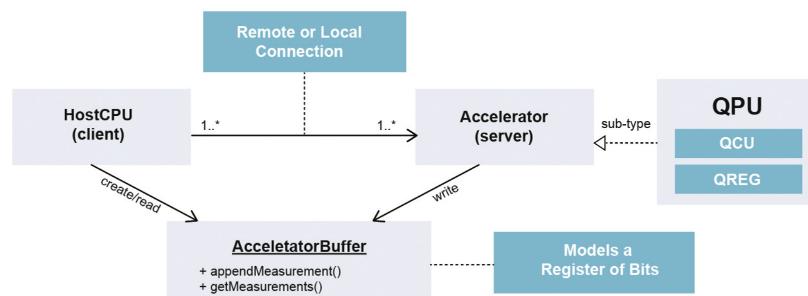


Fig. 1. The XACC platform model defines the interplay between the host CPU, accelerator system, and accelerator buffer memory space. The host CPU is charged with judiciously delegating work to a QPU which is controlled by an accelerator system. Results are stored in, and shared by, the accelerator buffer.

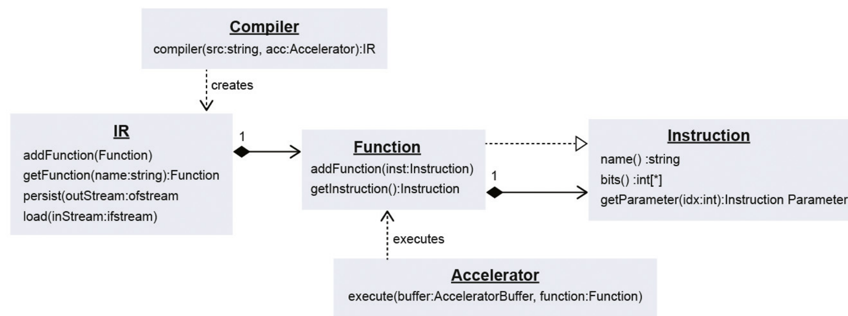


Fig. 2. The XACC interface architecture. Its core is the `IR`, `Function`, and `Instruction` interfaces and their mutual relationships. `Compilers` generate compiled `IR` instances, and `Accelerators` execute `IR`-contained `Functions`. Here solid arrows with a diamond origin imply a composition (`IR` is composed of `Functions`), and a dotted line with open-faced arrow implies interface realization (`Function` *is an* `Instruction`).

store them in the `AcceleratorBuffer`. Since clients keep reference to the created `AcceleratorBuffer` handle throughout the execution process, the data it contains after execution is available to be post-processed in order to compute expectation values or other statistical quantities of interest, thus influencing the rest of the hybrid computation.

3.2. Programming Model

The XACC programming model is designed to enable the expression of quantum algorithms alongside existing code in a quantum language-independent manner. Furthermore, the compiled result of the expressed quantum algorithm is designed to be amenable to execution on any quantum hardware through appropriately implemented device drivers. To achieve this, XACC defines six main concepts: (1) accelerator intermediate representation, (2) quantum kernels, (3) transformations on the intermediate representation, (4) compilers, (5) accelerators, and (6) programs. These concepts enable an expressive API for offloading computational tasks to an attached quantum accelerator. Clients express quantum algorithms via quantum kernel source code expressions in a similar way to OpenCL or CUDA for GPUs. These kernels may express quantum algorithms in any quantum programming language for which there exists a valid compiler implementation, thereby enabling a wide variety of programming approaches and techniques (high-level and low-level programmatic abstractions). Compilers map source kernels to a core intermediate representation that enables hardware dependent and independent program analysis, transformations, and optimization. This generally transformed or optimized representation is then mapped to hardware-native assembly code and executed on available physical or virtual hardware instances.

3.2.1. Intermediate Representation

To promote interoperability and programmability across the wide range of available accelerators and programming languages (embedded or stand-alone), there must exist a low-level program representation that is easy to understand and manipulate. An illustrative example can be found in the LLVM compiler infrastructure which maps various high-level classical programming languages (C, C++, Objective-C, Fortran, etc.) to a common intermediate representation (IR). The IR is then used to perform hardware (dependent and independent) analysis and optimizations in order to generate efficient hardware-specific executable code [25]. A standard IR for quantum computation should enable a wide range of programming tools and provide early users the benefit of programming their domain-specific algorithms in a manner that best suits their research and application. To date, there have been no efforts regarding the development of a unified intermediate representation for quantum computing that can span a number of different quantum compute models (e.g., adiabatic, gate). Compiler tools that are currently available take circuit-level programmatic expressions and map them to a hardware-specific quantum assembly (QASM) language, with different efforts providing QASM representations that differ in format and grammar. There is a strong need for a polymorphic set of extendable interfaces that span and support differing quantum accelerator types, thus enabling a retargetable compiler infrastructure for quantum computing across compute models and hardware types. Such an infrastructure sits at a slightly higher level of abstraction than typical assembly representations and therefore enables a unified API that integrates multiple high-level languages with multiple hardware architectures. The goal of this quantum intermediate representation is to provide an assembly-level language and API for quantum program analysis, transformation, and optimization.

XACC defines a polymorphic IR architecture that integrates programming languages and techniques with concrete (physical or virtual) hardware realizations. The XACC IR is designed to adhere to four primary requirements: (1) IR should provide a manipulable in-memory representation and API, (2) IR should be persistable to an on-disk file representation, (3) IR should provide a human-readable, assembly-like representation, and (4) IR should provide a graph representation. The architecture governing the IR interfaces is shown in Fig. 2 using the Unified Modeling Language (UML). The foundation of the XACC IR is the `Instruction` interface, which abstracts the concept of an executable instruction (e.g., a quantum gate or program). `Instructions` have a unique name and reference the accelerator qubits operated upon. `Instructions` can operate on one or many qubits and can be enabled or disabled for use in classical conditional branching. `Instructions` can also be parameterized — each `Instruction` can optionally keep track of one or many `InstructionParameters`, which are represented as a variant data structure that can be of type `float`, `double`, `complex`, `int`, or `string`. Importantly, the `InstructionParameter` concept allows a natural representation of instructions in variational quantum algorithms [26], which are among some of the most promising candidates for near-term speedups. Next, XACC defines a `Function` interface to express source code as compositions of `Instructions`. The `Function` interface is a derivation of the `Instruction` interface that itself contains `Instructions`. This `Instruction`/`Function` combination is an implementation of the composite design pattern, a common software design that models part-whole hierarchies [27,28]. Via this pattern, XACC models compiled programs as an *n-ary* tree with `Function` instances as nodes and `Instruction` instances as leaves (see Fig. 3 depicting the mapping between kernel source code and `Function`/`Instruction` trees). Executions, transformations, and optimizations of these `Function` instances are handled via a pre-order tree traversal, whereby walking each node involves walking each child node first, from left to right. For the IR tree in Fig. 3, this

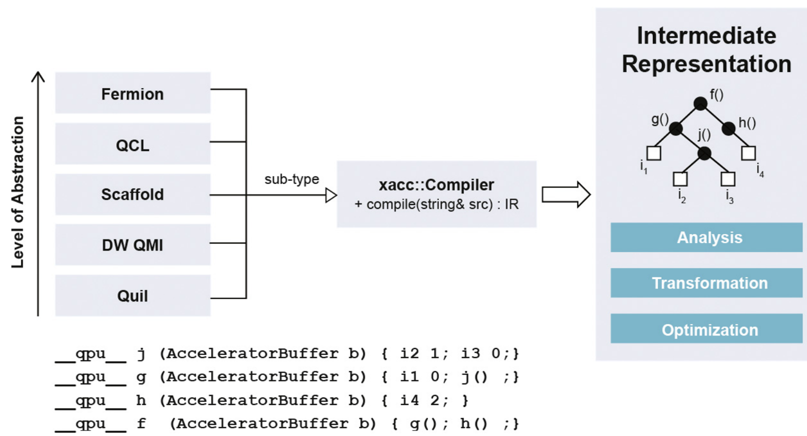


Fig. 3. The XACC Compiler workflow demonstrating quantum language extensibility. Compilers take quantum kernel source code written in an available language and map it to the XACC intermediate representation for program analysis, transformation, and optimization.

implies nodes are visited in the following order: $f \to g \to i_1 \to j \to i_2 \to i_3 \to h \to i_4$, where f, g, j, h are `Function` instances, and i_1, i_2, i_3, i_4 are general `Instruction` instances. Finally, XACC defines the `IR` interface which serves as a container for `Functions`. `IR` contains a list of `Functions` instances, with an exposed API that enables the mapping of those `Functions` to both an assembly-like, human-readable string and a graph data structure. For digitized computations, the graph can model the quantum circuit and provides a convenient data structure for program transformation and analysis. For quantum annealing, the graph structure can model the Ising Hamiltonian and scheduling parameters that form the machine-level instructions for the quantum computation. To provide an on-disk representation, the `IR` interface exposes load and persist methods that take a file path to read in, and to write to, respectively. In this way, `IR` instances that are generated from a given set of kernels can be persisted and reused, enabling faster ahead-of-time or just-in-time compilation.

3.2.2. IR Transformations

A key aspect of any compilation workflow is the ability to implement optimizations and transformations, which could be general or hardware dependent. There has been great progress in the development of quantum program transformation, optimization, and optimal instruction scheduling techniques for quantum programs over the last few years [29–32], and we have designed XACC to incorporate such optimizations into its overall compilation workflow. The goal of any program manipulation is to ensure that all compiled instructions are amenable to execution on the desired accelerator in an optimal or near-optimal manner. To handle optimizations and transformations, XACC defines an `IRTransformation` interface. This interface provides an extension point for taking an `IR` instance and generating a modified, optimized, or more generally transformed `IR` instance. The transformed IRs are logically equivalent, i.e., producing equivalent results in an idealized noise-free setting.

More general IR modifications are particularly well suited to handling error mitigation tasks which are crucial for near-term quantum computations. The basic idea is that on can generate a new IR, or set of transformed IRs, which gather additional information as needed to mitigate against some source of error. In this case, a proper post-execution processing mechanism must be in place to ensure that users retrieve the results they expect. To handle this situation, XACC defines an `IRPreprocessor` instance to take in the `IR` instance and modify it in a non-isomorphic manner, but return an `AcceleratorBufferPostprocessor` instance that knows the details of this modification and can adjust accelerator results accordingly. An example of the utility of this mechanism is in qubit measurement error-mitigation [7], whereby an `IRPreprocessor` can be implemented that adds measurement kernels to an `IR` instance. The execution of these additional kernels characterizes readout error rates and can be used by a corresponding `AcceleratorBufferPostprocessor` implementation, provided by the `IRPreprocessor` instance, to correct accelerator results. Other mitigation techniques such as noiseless extrapolations and quasi-probability methods [33,34] can likewise be handled within the construct of IR pre-processing and transformations.

After mapping kernel source code to an IR instance, the XACC model specifies that `IRTransformations` transform the `IR` instance before accelerator execution. Following these transformations, all requested or default `IRPreprocessors` are run and resultant `AcceleratorBufferPostprocessors` are stored and executed on resultant `AcceleratorBuffers` after execution.

3.2.3. Accelerators

The inevitable near-term variability in quantum hardware types forces any heterogeneous quantum–classical programming model to be extensible in the hardware it interacts with. XACC is no exception to this and therefore defines an `Accelerator` interface for injecting physical and virtual (i.e. simulator) QPU backends. The `Accelerator` interface (shown in Fig. 2) provides an `initialize` operator for sub-types to handle any start-up or loading procedures that are needed before execution on the device. This includes the retrieval of hardware specifications, such as connectivity information, that could influence kernel compilation and `IR` transformations. `Accelerators` expose a mechanism for creating `AcceleratorBuffer` instances, which provide programmers with a handle on `Accelerator` measurement results. Moreover, `Accelerator` realizations provide an implementation of a `getIRTransformations` operation to provide the necessary, low-level hardware-dependent transformations on the logically compiled `IR` instances.

Most crucially, `Accelerators` expose an `execute` operation that takes as input the `AcceleratorBuffer` to be operated on and the `Function` instance representing the kernel to be executed. Realizations of the `Accelerator` interface are responsible for leveraging these input data structures to affect execution on their target hardware or simulator. It is intended that `Accelerator` implementations leverage vendor- or library-supplied APIs to perform this execution. All `execute` implementations are responsible for updating the `AcceleratorBuffer` with measurement results.

Note the generality of this `Accelerator` interface. Subclasses can provide an `execute` implementation that targets either physical or virtual hardware. In this way we enable available quantum

```
auto src = R"src(__qpu__ foo(AcceleratorBuffer qreg, double theta) {...})src";
auto qpu = xacc::getAccelerator("ibm");
auto buffer = qpu->createBuffer("qreg", 2);
xacc::Program program(qpu, c
program.build();
auto kernel = program.getKernel<double>("foo");
for (auto& theta : {-3.14...3.14}) kernel(buffer, theta);
```

Listing 1: Example usage of foundational XACC API and the interplay between conventional and quantum programs.

programming languages to target simulated hardware which provides a mechanism for fast feedback on hybrid quantum–classical algorithmic execution. For example, `Accelerator` developers could provide an `execute` implementation for a variety of high-performance and specialty simulators [35–37]. In the absence of a preferred simulation methodology, we have provided a default implementation for the `Accelerator` that enables gate model quantum computer simulation via tensor network theory, specifically through a wave function decomposition leveraging the matrix product state ansatz (Tensor Network Quantum Virtual Machine, TNQVM) [38,39]. This enables users of XACC that target gate model quantum computation to study large systems of qubits before execution on physical hardware (with an upper bound dependent on the level of entanglement in the system).

3.2.4. Kernels, Compilers, and Programs

XACC requires that code intended for an `Accelerator` be provided in a manner similar to code intended for GPU acceleration within CUDA or OpenCL. That is, code must be expressed via stand-alone *kernels*. A kernel is a programmatic representation of accelerator operations applied to a register of bits. At its core, an XACC kernel is represented by a C-like function, however, this function must take as its first argument the `AcceleratorBuffer` instance representing the accelerator bit register (qubits) that this kernel operates on. It is in this way that kernels connect classical code with a handle to accelerator measurement results. XACC kernels do not specify a return type; all information about the results of a kernel's operation are gathered from the `AcceleratorBuffer's` ensemble of bit measurements. Kernels in XACC must be differentiated from conventional library function calls using the *__qpu__* keyword. This annotation can enable static, ahead-of-time compilation of XACC kernels by providing an abstract syntax tree search mechanism. We leave this static, ahead-of-time compiler as future work. Currently, all XACC `Compilers` are executed at runtime and therefore only enable just-in-time compilation. Finally, Kernels can take any number of kernel arguments that drive the overall execution of the quantum code. These parameters are modeled as the aforementioned `InstructionParameter` variant type. This enables parameterized compiled `IR` instances that can be evaluated at runtime.

The function body of an XACC kernel can be expressed in any *available* language. An *available* language is one for which there is a valid `Compiler` implementation for the language. The `Compiler` interface architecture is shown in Fig. 2, and its extensibility and connection to the XACC IR is shown in Fig. 3. This interface provides a `compile` method that takes kernel source code as input and produces a valid instance of the XACC `IR`. Derived `Compilers` are free to perform compilation in any way they see fit, as long as they return a valid `IR` instance. Moreover, the `compile` operation can optionally take the targeted accelerator as input, which enables hardware-specific details to be present at compile time and thus influence the way compilation is performed.

This compilation extension point provides a mechanism for the mapping of high-level constructs to lower-level quantum assembly, and therefore facilitates quantum program decomposition methods that map domain specific programmatic expressions to the XACC IR. An example of this would be a domain specific language that expresses a molecular Hamiltonian, and an associated `Compiler` realization that maps this Hamiltonian to quantum assembly via a Jordan–Wigner or Bravyi–Kitaev transformation [40,41]. Note this design also facilitates general gate decomposition techniques through the overall extensibility of the `compile` method. One could imagine a domain specific language for the expression of general unitaries that are expressed as an XACC kernel and passed to a `Compiler` implementation that decomposes the unitary into a native low-level gate set [42].

The XACC compilation concept also defines a kernel source code `Preprocessor` extension point. `Preprocessors` are executed before compilation and take as input the source code to analyze and process, the `Compiler` reference for the kernel language, and the target accelerator. Using this data, `Preprocessors` can perform operations on the kernel source string to produce a modified source code that enhances or simplifies a computation. An example of the `Preprocessor's` utility would be quantum language macro expansion, or searching kernel source code for certain keywords describing a desired algorithm and replacing that line of code with a source-code representation of the algorithm. In this way, `Preprocessors` can be used to alleviate tedious programming tasks.

The primary entry point for interaction with the XACC compilation infrastructure is the concept of a `Program`. The `Program` orchestrates the entire kernel compilation process and provides users with an executable functor to execute the compiled kernel on the desired `Accelerator`. `Programs` are instantiated with reference to the kernel source code and targeted `Accelerator`, and provide programmers with a `build()` operation that applies requested kernel `Preprocessors`, selects and executes the correct `Compiler` to produce the `IR` instance, and then executes all desired (or default) `IRTransformations` and `IRPreprocessors`. Finally, the `Program` exposes a `getKernel` operation returning an executable functor that executes the compiled `Function` on the target `Accelerator`.

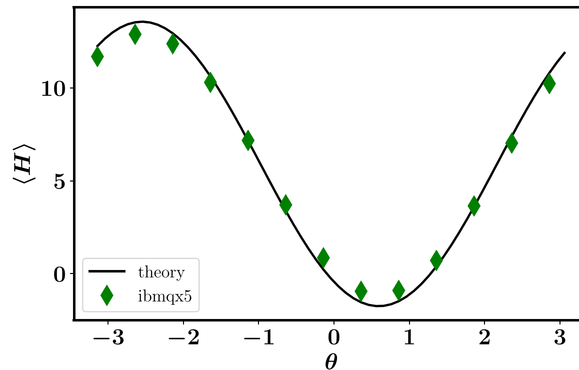
The interplay of conventional and quantum programs is demonstrated in Listing 1. Users describe their source code as an XACC kernel (note this kernel is parameterized by a `double` parameter), request a reference or handle to the desired `Accelerator`, and allocate a buffer of qubits. Next, a `Program` object is instantiated and the XACC compilation workflow is initiated through the `build` invocation. At this point the appropriate `Compiler` has mapped the source code to the XACC IR, and all transformations, optimizations, and preprocessors have been invoked to provide an executable functor or lambda that will enable user execution on the desired `Accelerator`. This executable kernel reference can then be used as part of some parameterized loop, enabling hybrid quantum–classical variational algorithms.

```
__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]
}
```

Listing 2: XACC Kernels for Deuteron VQE

```
auto qpu = xacc::getAccelerator("ibm");
auto buffer = qpu->createBuffer("qreg", 2);
xacc::Program program(qpu, deuteronSrc);
program.build();
std::vector<double> energies, coeffs{.218291,
        -6.125, -2.143304, -2.143304};
auto kernels = program.getKernels();
for (auto theta : thetaRange) {
    double energy = 5.906709;
    for (int i = 0; i < kernels.size(); i++) {
        kernels[i](buffer, theta);
        energy += coeffs[i] *
                buffer->getExpectationValueZ();
        buffer.resetBuffer();
    }
    energies.push_back(energy);
}
```

(a)



(b)

Fig. 4. Parameter variation for deuteron. The code snippet in (a) follows from the code in Listing 1. Users request an Accelerator and compile XACC kernel source code to XACC IR. Executable kernel functors can be requested and used as part of a parameterized loop. The `energies` vector as a function of the variational parameter θ computed via the TNQVM (virtual) Accelerator and IBM QX5 16 qubit QPU are shown in (b).

4. Demonstration

Near-term quantum computing devices provide a relatively small quantum register and lack sufficient error correction capabilities to implement fault-tolerant computations. Nevertheless, these pre-threshold devices demonstrate sufficient hardware control to support programmable sequences of (imperfect) operations known as quantum circuits. Devices executing these quantum circuits may be used as primitive quantum accelerators within a hybrid computing scheme [43]. Only a few of these early QPUs are publicly available, and all are remotely located with respect to the end user, matching the client–server platform model described in Section 3.1. In this section, we demonstrate the utility of the XACC framework through demonstrations programming both gate and annealing quantum computers, using the unified XACC API.

4.1. Example program for nuclear binding energy calculations

Here we demonstrate using XACC to compose a scientific application for calculating the binding energy of an atomic nuclei. The accuracy of this program was reported previously for the example of deuteron [5], and we use this example to describe the technical details for how this program is constructed. The general structure of this XACC hybrid program derives from the variational quantum eigensolver (VQE) algorithm [6], which is a quantum–classical algorithm for recovering the lowest energy eigenstate of a quantum mechanical Hamiltonian. The minimal form of the system Hamiltonian, whose lowest eigenvalue is related to the binding energy, is given by

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

where X_i, Y_i, Z_i denote Pauli operators acting on the ith qubit.

The VQE algorithm searches for the ground state energy of a given Hamiltonian by optimizing the expectation value of the Hamiltonian with respect to a parameterized quantum wavefunction encoded into the qubit register of an accelerator QPU. For large system sizes, wavefunctions are easily represented in qubit registers but require exponential classical resources to store. At each iteration of this optimization, the QPU is evolved by a quantum circuit parameterized by the current iterate's parameters, and multiple measurements are performed for non-commuting sets of

```
src = R"src(__qpu__ factor15() {
   0 0 20;
   1 1 50;
   ...
   1 6 -128;
   2 6 -128;
})src";
```

(a)

```
auto qpu = xacc::getAccelerator("dwave");
auto qubitReg = qpu->createBuffer("q");
xacc::Program program(qpu, src);
program.build();
auto factor15 = program.getKernel("factor15");
factor15(qubitReg);
... analyze qubitReg measurement bit strings
```

(b)

Fig. 5. A simple example demonstrating the flexibility and utility of the XACC framework in executing a program on the D-Wave QPU. (a) represents an XACC kernel written for the D-Wave quantum machine instruction compiler, while (b) demonstrates using the XACC API to compile and execute this code. This code factors 15 into 3 and 5 using the D-Wave QPU.

Hamiltonian terms. Expectation values are then evaluated with respect to the ensemble of measurement samples, and the weighted sum of all these expectation values determines the system energy at a given parameterization. This optimization continues until convergence.

We now demonstrate how to program this algorithm, in a QPU-independent manner, using the XACC framework. First, we define the kernel source code initializing a trial wavefunction, known as an *ansatz*, on a QPU. This ansatz is defined subsequently in Listing 2, with the `__qpu__ ansatz(···){···}` kernel. This kernel implements three logical operations. Using the ansatz kernel as a building block, we append additional gates and measurement instructions, as needed, to evaluate the expectation values of the Hamiltonian terms from Eq. (1). The Z_0, Z_1 terms in Eq. (1) can be evaluated with respect to the QPU state after the initialization ansatz, so only measurement instructions are appended. To evaluate the other terms, involving X and Y operators, local change of basis rotations are applied, that is, a Hadamard gate for all X operators and an $X(\frac{\pi}{2})$ rotation for all Y operators. The XACC quantum kernel source code in Listing 2 has been written in Quil [12]. Note, however, that kernels can be written in any gate model quantum language supported by the framework (OpenQASM, Scaffold, etc.). Because the XACC `IR` behaves as an *n-ary* tree of `Instruction` instances, previously defined kernels can be reused as Instructions in other kernels. Recursive circuits, such as the quantum Fourier transform, can easily be defined in this manner.

To compile and execute these kernels, we leverage the XACC API, as shown in Fig. 4(a). Note that users requesting an `Accelerator` from the framework simply provide the `string` name corresponding to the desired `Accelerator`. This returns a polymorphic `Accelerator` reference that points to the desired implementation. Running this code on the TNQVM `Accelerator` amounts to simply modifying the `getAccelerator` `string` argument to **`tnqvm`**. The results of running this code on the TNQVM `Accelerator` and the IBMQX5 16 qubit QPU are shown in Fig. 4(b). Raw timing information for this execution is not very illuminating as a large majority of the time is spent waiting in the IBM Quantum Experience job queue or suspect to network lags due to remote HTTPS invocations. We can however estimate a lower bound on the execution times for this example program by considering the circuit length, measurement, and refresh timescales. Let us consider a quantum program consisting of $n_{l(e)}$ layers of local (entangler) quantum gates which may be implemented in parallel (in our example $n_l = 2, n_e = 1$). Given typical superconducting gate timescales of $t_l = 20$ ns, $t_e = 200$ ns, along with a $t_m = 2$ µs measurement timescale, and a refresh time of $t_R \approx 10 * T_1 \approx 500$ µs needed to re-initialize the qubit registers by natural relaxation mechanisms. The minimum device time needed to evaluate all four kernels in Listing 2, given an ensemble size of 10^4 samples per term, would be 20 s for each function evaluation at a given parameter θ. Note that the sample rate can be partially alleviated by parallelization (a topic to be detailed in future work), but one can already see that the overall time resources may become prohibitive (e.g. exponentially costly) for programs which require a significant number of samples in order to optimize over noisy cost function evaluations [44,45]. It is therefore necessary to improve the scalability of quantum optimization algorithms in order to reduce the significant cost of quantum optimization.

4.2. Simple integer prime factorization on D-Wave

In an effort to demonstrate the polymorphic nature of the XACC IR, here we provide an example of programming a simple problem targeting the D-Wave QPU. This example leverages the exact same API calls as in Listing 1 and the deuteron demonstration (see Fig. 5(b)). Specifically, we demonstrate the use of an XACC quantum annealing IR implementation (with corresponding `Function` and `Instruction` subtypes for quantum annealing) by using the D-Wave QPU to factor 15 into 5 and 3. The quantum kernel for factoring 15 on the D-Wave QPU, and the associated code required to compile and execute it using the XACC API are shown in Fig. 5 (kernel code trimmed for brevity).

We have implemented a `Compiler` implementation, the `DWQMICompiler` [46], which takes as input kernels structured as a new-line separated list of D-Wave quantum machine instructions (Ising Hamiltonian coefficients). The compilation and execution workflow starts by getting reference to the D-Wave Accelerator, which gives the user access to all remotely hosted D-Wave Solvers (physical and virtual resources). Next, users request that an `AcceleratorBuffer` be allocated, which gives them a reference to the D-Wave QPU qubits, as well as all resultant data after execution. Then, a `Program` is created and built (compiled) with reference to the `Accelerator` and source code. This, in turn begins the minor graph embedding and parameter setting steps as part of the `DWQMICompiler` workflow (for full details on the D-Wave programming workflow, see [47]). Users execute the kernel lambda which populates the `AcceleratorBuffer` instance with the resultant data (energies, measurement bit strings, etc.). The bit string corresponding to the minimum energy can then be used to reconstruct the binary representation of the factors of 15.

5. Discussion

We have presented a programming, compilation, and execution framework enabling the integration of quantum computing within standard and HPC workflows in a language and hardware independent manner. We have demonstrated a high-level set of interfaces and programming concepts that support QPU acceleration reminiscent of existing GPU acceleration. These interfaces enable domain computational scientists to migrate existing scientific computing code to early QPU devices while retaining prior programming investments.

This work opens up interesting avenues for the development of benchmarking, verification, and profiling software suites for near-term quantum computing hardware. As domain computational scientists start leveraging these quantum technologies as part of existing software workflows, the ability to quickly swap out virtual and physical `Accelerator` instances will enable quick verification of actual QPU results. Benchmarking suites that compare and contrast high-level algorithm executions across the varied quantum hardware types will provide a mechanism for intuiting which hardware best fits the problem at hand. In this regard, XACC provides a unified API for quickly swapping out these hardware instances, thus enabling a write once and run QPU benchmarking and verification mentality.

Finally, note the generality of the framework's core interfaces. We have focused on the quantum acceleration of classical heterogeneous architectures, but one could easily imagine fitting other post-Moore's law hardware types, such as neuromorphic accelerators, into the XACC framework. This is a direction for future work we intend to pursue.

Acknowledgments

This work has been supported by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory, United States, 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, United States, 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, United States for the US Department of Energy under contract no. DE-AC05-00OR22750. The US government retains and the publisher, by accepting the article for publication, acknowledges that the US government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for US government purposes. DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan.

References

[1] Hack J, Riley K, Gerber R. Crosscut report: Exascale requirements reviews, technical report; 2018. URL https://science.energy.gov/media/ascr/pdf/programdocuments/docs/2018/DOE-ExascaleReport-CrossCut.pdf.
[2] IBM Builds Its Most Powerful Universal Quantum Computing Processors. URL https://phys.org/news/2017-05-ibm-powerful-universal-quantum-processors.html.
[3] Kelly J, O'Malley P, Neeley M, Neven H, Martinis JM. Physical Qubit Calibration on a Directed Acyclic Graph, arXiv:1803.03226.
[4] Britt KA, Humble TS. High-performance computing with quantum processing units. J Emerg Technol Comput Syst 2017;13(3):39:1–39:13 http://dx.doi.org/10.1145/3007651. http://doi.acm.org/10.1145/3007651.
[5] Dumitrescu EF, McCaskey AJ, Hagen G, Jansen GR, Morris TD, Papenbrock T, et al. Cloud quantum computing of an atomic nucleus. Phys Rev Lett 2018;120:210501. https://link.aps.org/doi/10.1103/PhysRevLett.120.210501.
[6] McClean JR, Romero J, Babbush R, Aspuru-Guzik A. The theory of variational hybrid quantum-classical algorithms. New J Phys 2016;18(2):023023 http://dx.doi.org/10.1088/1367-2630/18/2/023023. http://stacks.iop.org/1367-2630/18/i=2/a=023023.
[7] Kandala A, Mezzacapo A, Temme K, Takita M, Brink M, Chow JM, et al. Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets. Nature 2017;549:242. http://dx.doi.org/10.1038/nature23879.
[8] Harrow AW, Hassidim A, Lloyd S. Quantum algorithm for linear systems of equations. Phys Rev Lett 2009;103:150502. http://dx.doi.org/10.1103/PhysRevLett.103.150502.
[9] Otterbach JS, Manenti R, Alidoust N, Bestwick A, Block M, Bloom B, et al. Unsupervised machine learning on a hybrid quantum computer, arXiv E-prints, arXiv:1712.05771.
[10] Nickolls J, Buck I, Garland M, Skadron K. Scalable parallel programming with CUDA. Queue 2008;6(2):40–53. http://dx.doi.org/10.1145/1365490.1365500.
[11] Stone JE, Gohara D, Shi G. OpenCL: A parallel programming standard for heterogeneous computing systems. IEEE Des Test 2010;12(3):66–73. http://dx.doi.org/10.1109/MCSE.2010.69.
[12] Smith RS, Curtis MJ, Zeng WJ. A Practical Quantum Instruction Set Architecture, arXiv e-prints arXiv:1608.03355.
[13] Svore K, Geller A, Troyer M, Azariah J, Granade C, Heim B, et al. Q#: enabling scalable quantum computing and development with a high-level dsl. In: Proceedings of the real world domain specific languages workshop 2018. New York, NY, USA: ACM; 2018. p. 7:1–7:10. http://dx.doi.org/10.1145/3183895.3183901.
[14] Cirq, Available from: https://github.com/quantumlib/Cirq. [Accessed 19 July 2018].
[15] Cross AW, Bishop LS, Smolin JA, Gambetta JM. Open Quantum Assembly Language, arXiv e-prints, arXiv:1707.03429.
[16] QCL: A Programming Language for Quantum Computers. Available from: http://tph.tuwien.ac.at/~oemer/qcl.html. [Accessed 21 June 2018].
[17] Green AS, Lumsdaine PL, Ross NJ, Selinger P, Valiron B. Quipper: A scalable quantum programming language. In: Proceedings of the 34th ACM SIGPLAN conference on programming language design and implementation. New York, NY, USA: ACM; 2013. p. 333–42. http://dx.doi.org/10.1145/2491956.2462177.
[18] Svore K, Geller A, Troyer M, Azariah J, Granade C, Heim B, et al. Q#: Enabling scalable quantum computing and development with a high-level DSL. In: Proceedings of the real world domain specific languages workshop 2018. New York, NY, USA: ACM; 2018. p. 7:1–7:10. http://dx.doi.org/10.1145/3183895.3183901.
[19] JavadiAbhari A, Patil S, Kudrow D, Heckey J, Lvov A, Chong FT, et al. ScaffCC. Parallel Comput 2015;45(C):2–17. http://dx.doi.org/10.1016/j.parco.2014.12.001.
[20] McCaskey A, Dumitrescu E, Liakh D, Humble T. Hybrid Programming for Near-term Quantum Computing Systems, arXiv e-prints arXiv:1805.09279.
[21] OpenFermion. Available from: https://github.com/quantumlib/OpenFermion. [Accessed 20 June 2018].
[22] IBM Acqua. Available from: https://github.com/Qiskit/qiskit-acqua. [Accessed 18 July 2018].
[23] XACC-VQE. Available from: https://github.com/ornl-qci/xacc-vqe. [Accessed 21 June 2018].
[24] Sinha A. Client-server computing. Commun ACM 1992;35(7):77–98. http://dx.doi.org/10.1145/129902.129908.
[25] Lattner C, Adve V. 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. Washington, DC, USA: IEEE Computer Society; 2004. p. 75 URL http://dl.acm.org/citation.cfm?id=977395.977673.
[26] Peruzzo A, McClean J, Shadbolt P, Yung MH, Zhou XQ, Love PJ, et al. A variational eigenvalue solver on a photonic quantum processor. Nature Commun 2014;5(May): http://dx.doi.org/10.1038/ncomms5213. arXiv:1304.3061.
[27] Gamma E, Helm R, Johnson R, Vlissides J. Design patterns: Elements of reusable object-oriented software. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.; 1995.
[28] A Look at the Composite Design Pattern. Available from: https://www.javaworld.com/article/2074564/learn-java/a-look-at-the-composite-design-pattern.html. [Accessed 12 March 2018].
[29] Venturelli D, Do M, Rieffel E, Frank J. Compiling quantum circuits to realistic hardware architectures using temporal planners.
[30] Booth KEC, Do M, Beck JC, Rieffel E, Venturelli D, Frank J. Comparing and integrating constraint programming and temporal planning for quantum circuit compilation, arXiv e-prints, arXiv:1803.06775.
[31] Nam Y, Ross NJ, Su Y, Childs AM, Maslov D. Automated Optimization of Large Quantum Circuits with Continuous Parameters.
[32] Zulehner A, Paler A, Wille R. An Efficient Methodology for Mapping Quantum Circuits to the IBM QX Architectures.
[33] Endo S, Benjamin SC, Li Y. Practical Quantum Error Mitigation for Near-Future Applications, arXiv:1712.09271.
[34] Temme K, Bravyi S, Gambetta JM. Error mitigation for short-depth quantum circuits. Phys Rev Lett 2017;119(18):180509 http://dx.doi.org/10.1103/PhysRevLett.119.180509. arXiv:1612.02058.
[35] Zulehner A, Wille R. Advanced Simulation of Quantum Computations, http://dx.doi.org/10.1109/TCAD.2018.2834427, arXiv:1707.00865.
[36] Haner T, Steiger DS, Smelyanskiy M, Troyer M. High performance emulation of quantum circuits. In: International conference for high performance computing, networking, storage and analysis, SC. IEEE; 2017. p. 866–74 http://dx.doi.org/10.1109/SC.2016.73. arXiv:1604.06460.
[37] Smelyanskiy M, Sawaya NPD, Aspuru-Guzik A. qHiPSTER: The quantum high performance software testing environment, arXiv:1601.07195.

[38] McCaskey A, Dumitrescu E, Chen M, Lyakh D, Humble TS. Validating quantum-classical programming models with tensor network simulations, arXiv e-prints, arXiv:1807.07914.

[39] McCaskey A, Chen M. TNQVM - Tensor Network Quantum Virtual Machine, GitHub Repository, GitHub, https://github.com/ORNL-QCI/tnqvm.

[40] Jordan P, von Neumann J, Wigner EP. On an algebraic generalization of the quantum mechanical formalism. In: Wightman AS, editor. The collected works of Eugene Paul Wigner: Part A: The scientific papers. Berlin, Heidelberg: Springer Berlin Heidelberg; 1993. p. 298–333.

[41] Andrew T, Sarah S, Jake S, Michael K, Jarrod M, Ryan B et al. The Bravyi–Kitaev transformation: Properties and applications. Int J Quantum Chem 115 (19): 1431–1441. http://dx.doi.org/10.1002/qua.24969, https://onlinelibrary.wiley.com/doi/abs/10.1002/qua.24969.

[42] Vartiainen JJ, Möttönen M, Salomaa MM. Efficient decomposition of quantum gates. Phys Rev Lett 2004;92:177902. http://dx.doi.org/10.1103/PhysRevLett.92.177902.

[43] Aspuru-Guzik A, et al. ASCR workshop on quantum computing for science, tech. rep. Department of Energy Office of Science Advanced Scientific Computing Research Program; 2015.

[44] Kandala A, Temme K, Corcoles AD, Mezzacapo A, Chow JM, Gambetta JM. Extending the Computational Reach of a Noisy Superconducting Quantum Processor, arXiv:1805.04492.

[45] Hempel C, Maier C, Romero J, McClean J, Monz T, Shen H, et al. Quantum Chemistry Calculations on a Trapped-Ion Quantum Simulator, arXiv:1803.10238.

[46] XACC D-Wave Plugins. Available from: https://github.com/ornl-qci/xacc-dwave. [Accessed 23 July 2018].

[47] Humble TS, McCaskey AJ, Bennink RS, Billings JJ, D'Azevedo EF, Sullivan BD, et al. An integrated programming and development environment for adiabatic quantum optimization. Comput Sci Discov 2014;7(1):015006. http://stacks.iop.org/1749-4699/7/i=1/a=015006.