Extending C++ for Heterogeneous Quantum-Classical Computing

ALEXANDER MCCASKEY, THIEN NGUYEN, ANTHONY SANTANA,
DANIEL CLAUDINO, and TYLER KHARAZI, Oak Ridge National Laboratory, United States
HAL FINKEL, Argonne National Laboratory, United States

We present qcor—a language extension to C++ and compiler implementation that enables heterogeneous quantum-classical programming, compilation, and execution in a single-source context. Our work provides a first-of-its-kind C++ compiler enabling high-level quantum kernel (function) expression in a quantum-language agnostic manner, as well as a hardware-agnostic, retargetable compiler workflow targeting a number of physical and virtual quantum computing backends. qcor leverages novel Clang plugin interfaces and builds upon the XACC system-level quantum programming framework to provide a state-of-the-art integration mechanism for quantum-classical compilation that leverages the best from the community at-large. qcor translates quantum kernels ultimately to the XACC intermediate representation, and provides user-extensible hooks for quantum compilation routines like circuit optimization, analysis, and placement. This work details the overall architecture and compiler workflow for qcor, and provides a number of illuminating programming examples demonstrating its utility for near-term variational tasks, quantum algorithm expression, and feed-forward error correction schemes.

CCS Concepts: • **Hardware → Quantum computation**; • **Software and its engineering → Compilers;**

Additional Key Words and Phrases: Quantum computing, compilers, domain specific languages, quantum programming

ACM Reference format:
Alexander McCaskey, Thien Nguyen, Anthony Santana, Daniel Claudino, Tyler Kharazi, and Hal Finkel. 2021. Extending C++ for Heterogeneous Quantum-Classical Computing. *ACM Trans. Quantum Comput.* 2, 2, Article 6 (July 2021), 36 pages.
https://doi.org/10.1145/3462670

This work has been supported by the US Department of Energy (DOE) Office of Science Advanced Scientific Computing Research (ASCR) Quantum Computing Application Teams (QCAT), Quantum Testbed Pathfinder (QTP), and Accelerated Research in Quantum Computing (ARQC). ORNL is managed by UT-Battelle, LLC, for the US Department of Energy under contract no. DE-AC05-00OR22725. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357. This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.

Authors' addresses: A. McCaskey, T. Nguyen, A. Santana, D. Claudino, and T. Kharazi, Oak Ridge National Laboratory, 1 Bethel Valley Rd, Oak Ridge, Tennessee, 37831; emails: mccaskeyaj@ornl.gov, nguyentm@ornl.gov, santanaam@ornl.gov, daniel.pchem@gmail.com, tylerdkharazi@gmail.com; H. Finkel, Argonne National Laboratory, 1 Thørväld Circle, Lemont, Illinois; email: Hal.Finkel@science.doe.gov.

2643-6817/2021/07-ART6 $15.00
https://doi.org/10.1145/3462670

1 INTRODUCTION

The recent availability of programmable quantum computers over the cloud has enabled a number of small-scale experimental demonstrations of algorithmic execution for pertinent scientific computing tasks [7, 11, 33, 35, 40]. These demonstrations point toward a future computing landscape whereby classical and quantum computing resources may be used in a hybrid, heterogeneous manner to continue to progress the state of the art with regards to simulation capability and scale. A future post-exascale, heterogeneous computing architecture enhanced with quantum accelerators or co-processors in a tightly integrated manner could enable large-scale simulation capabilities for a number of scientific fields such as chemistry, nuclear and high-energy physics, and machine learning. However, the novelty and utility of heterogeneous quantum-classical compute models will only be effective if there is an enabling software infrastructure that promotes efficiency, programmability, and extensibility. There is a strong need to promote novel software frameworks, programming languages, compilers, and tools that will enable tight integration of existing HPC resources and applications with future quantum computing hardware.

C++ has proven itself as a leading language within the high-performance scientific computing community for its portability, scalability and performance, multi-paradigm capabilities (generic, object-oriented, imperative), integration with other languages, and community support. It has been leveraged to enable a number of programming models for classical accelerated computing [1, 8, 17, 23]. We anticipate that this trend will continue, and one will require models, compilers, and tools that promote node-level quantum acceleration via extensions or libraries for C++. Moreover, as tighter integration models become possible, quantum-classical programs that require a feed-forward capability (e.g., quantum error correction schemes) will require performant languages with low overhead. Here by feed-forward we mean a model whereby mid-circuit measurements are possible and these measurement results conditionally trigger further quantum circuit execution.

As of this writing, a number of approaches for programming quantum computers have been introduced, and one can classify most of these as either low-level intermediate or assembly languages, circuit construction software frameworks, or high-level languages and compilers. Low-level intermediate languages like OpenQASM 2.0 [13], Quil [44], and Jaqal [15] have been proposed that enable circuit definition at the gate or pulse level and most provide some form of hierarchical function (subroutine) definition, composition, and control-flow. Each of these provide its own set of benefits and drawbacks, but all are at a low-level of abstraction and are primarily meant to be generated by higher-level compilers and frameworks. Moving up the stack, there have been a number of Pythonic circuit construction frameworks developed (Qiskit [20], PyQuil [27], Cirq [5], JaqalPaq [14], and ProjectQ [45]) that make it easier for users to generate hardware-specific intermediate language representations for ultimate execution on remotely hosted backends. As hardware progresses and tighter CPU-QPU integration is enabled, we anticipate that this remote Pythonic programming and execution model will not be sufficient for enabling a performant interplay between classical and quantum resources. At the highest level, a few approaches have enabled high-level stand-alone, as well as embedded, domain specific languages and associated compilers for quantum-classical programming. We specifically look to Q# [25] and Scaffold [10] as prototypical examples that have seen adoption and success. These approaches enable high-level expressibility as well as quantum-classical control flow. Unfortunately, both of these currently lack in some form with regards to tight integration of HPC resources with quantum co-processors. Q# leverages the Microsoft .NET infrastructure and integrates with the C# language, both of which are not easily adopted or accessed by existing HPC applications and resources. Scaffold extends C, a popular HPC language, but lacks direct integration with QPU resources, relying on manual processes for mapping compiler assembly output to appropriate Pythonic circuit-construction frameworks.

Here we describe a mechanism that seeks to fill this void in the quantum scientific computing software stack. Specifically, we detail the qcor compiler, which enables a language extension to C++ through high-level Clang plugin implementations promoting quantum function expression alongside standard classical code. Our approach targets both near-term, remotely hosted quantum computing models as well as future fault-tolerant, tightly integrated quantum-classical architectures with feed-forward capabilities. We enable quantum code expression in a language agnostic manner as well as the ability to compile to most available quantum computing backends (including simulators). Furthermore, we provide a compiler runtime library that exposes a robust API for leveraging quantum kernels (functions) as standard callables, which can be used as input to algorithmic implementations as needed. Ultimately, the qcor compiler paves the way for direct integration with existing applications, toolchains, and techniques common to scientific HPC, and is the first platform that allows programming hybrid quantum-classical algorithms in a single-source C++, general, and deployable manner.

This article is outlined as follows: First, we provide a quick discussion of a typical qcor program in an effort to guide the reader through the rest of the architectural details. We then provide the necessary background information required for a proper discussion of the qcor implementation (the specification, XACC, and Clang). Next, we provide the architectural details of the qcor runtime library and compiler implementation and workflow. The runtime library provides crucial utilities underpinning the language extension and compiler, as well as data structures and API calls for typical quantum algorithmic expression and execution. We detail the novel extensions to Clang we have developed for mapping general quantum kernel domain specific languages to valid C++ API calls. We end with a robust demonstration of qcor, and detail the programming of prototypical use cases, as well as its capability and performance as an optimizing, retargetable quantum compiler.

2 ANATOMY OF A QCOR PROGRAM

Figure 1 demonstrates a simple qcor-enabled C++ program—the programming and execution of the GHZ state. This straightforward case demonstrates the single-source programming model qcor provides, without going into all the complexity in the rest of the qcor/XACC framework for common algorithmic tasks. We will go into the full details of the qcor implementation in the following sections, but here we show the model and the philosophy proposed by the language extension.

The qcor compiler provides a C++ language extension that enables the use of a primitive qreg type, quantum kernel definition, quantum instruction programming, and quantum-classical control flow. In the code snippet, one notices there are no header files included, everything in the source code is provided by the language extension. Programmers begin by defining a quantum kernel, which is just a standard C++ function annotated with the __qpu__ attribute. Kernels can take arbitrary function arguments, but must take at least one reference to an allocated qubit register (qreg). The function body itself is language-agnostic, i.e., programmers can use any quantum programming language (for which there is an appropriate TokenCollector implementation, see Section 4.2.1). The current version of qcor enables one to program in the XASM [3], IBM OpenQASM 2.0 [13], and unitary matrix decomposition languages. Notice that low-level quantum instruction invocation is allowed as part of the language extension itself, and that we are free to use existing C++ control flow statements like the for loop to condense long sequences of quantum instruction calls. Once the kernel is defined, one simply allocates a register of qubits of a desired size (similarly to the C malloc call but for qubits, qalloc). To execute the kernel on the targeted quantum co-processor, one just invokes the quantum kernel function, providing the correct arguments (here the qubit register). Execution results (bit strings and counts) are persisted to the qreg instance and are available for use in the rest of the program.

```
// No includes needed, we are using the
// language extension

// Quantum Kernels are just C++ functions
// annotated with __qpu__. Can take any
// arguments must provide a qreg to run on.
__qpu__ void ghz_n(qreg q) {
  // Kernels can be expressed in any available
  // quantum language, here XACC XASM.
  // The language extension allows quantum
  // instruction expression as part of the
  // language. We also get C++ control flow
  // as part of the language
  H(q[0]);
  for (auto i : range(q.size()-1)) {
    CX(q[i], q[i+1]);
  }
  for (auto i : range(q.size())) {
    Measure(q[i]);
  }
}
int main() {
  // Language extension gives us the
  // qalloc() quantum buffer allocator.
  // q is a qreg, a primitive type provided
  // by the language extension
  auto q = qalloc(10);

  // Execute the quantum kernel
  ghz_n(q);

  // Query results on the allocated qreg
  auto counts = q.counts();
  for (auto [bits, count] : counts) {
    print(bits, ": ", count);
  }
}
// Run on remote IBM Paris backend with
// qcor -qpu ibm:ibmq_paris -shots 1024 \
//        ghz.cpp -o ghz.x
// ./ghz.x
```

Fig. 1. A simple qcor program, expressing a quantum kernel that executes the standard GHZ state on 10 qubits.

To compile and run this program, one uses the qcor compiler, indicating the quantum backend being compiled to and any other pertinent execution information (like shots). The qcor compiler provides all of the same compiler command line arguments as Clang and GCC, i.e., one can build up complex source codes that require extra header and library search paths, specific libraries to link, and other compiler and link flags. After compilation, the programmer is left with a binary executable or object file.

Figure 1 is a simple example of programming with qcor. There is of course much more that one could do, including kernel composition (kernels that call other kernels), auto-generated adjoint

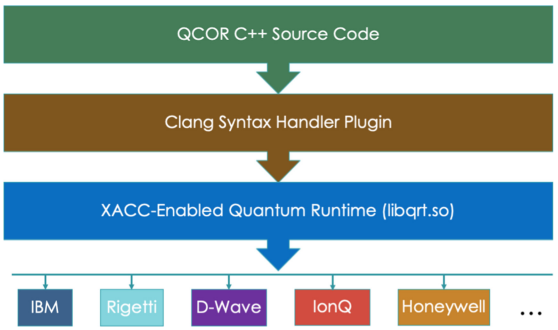

Fig. 2. qcor provides a single-source C++ programming model through plugin extensions to Clang and an XACC-enabled quantum runtime library implementation, enabling execution on a number of popular quantum backends.

and control versions of the defined quantum kernel, kernel construction with complex control flow, kernel definition at the unitary matrix level via extensible circuit synthesis algorithms, and the use of qcor provided data structures for the expression of complex hybrid quantum-classical algorithms. The rest of this work will describe these key abilities in the following sections.

3 BACKGROUND

qcor implements the specification introduced in Reference [41] by building upon the XACC quantum programming framework. Moreover, quantum kernel compilation is accomplished via extension of core Clang plugin interfaces. Here we describe pertinent details about XACC, the QCOR specification, and Clang to provide a foundation to describe the qcor compiler implementation. Figure 2 gives a high-level view of the overall relationship between qcor, Clang, and XACC. qcor kernel expressions are mapped to appropriate XACC types via domain-specific language preprocessing provided by novel plugins to the Clang infrastructure. The incorporation of XACC implies a retargetable compiler workflow, with backends provided by the main quantum computing hardware vendors.

3.1 XACC

The XACC quantum programming framework is a system-level, C++ infrastructure enabling language and hardware agnostic quantum programming, compilation, and execution [39]. XACC adopts a dual-source programming model, whereby quantum kernels are defined as separate source strings and compiled to a core, polymorphic **intermediate representation (IR)** via an appropriate API library call. XACC builds upon the CppMicroServices framework [6] to provide a native implementation of the Open Services Gateway Initiative [38], and promotes a service oriented architecture that provides extensibility at all points of the quantum-classical programming workflow. We leave a detailed overview of XACC to Reference [39], but here we highlight a few core service interfaces that are pertinent for our discussion of qcor.

XACC employs a layered architecture that decomposes the framework into extensible frontend, middle-end, and backend layers. The frontend exposes a service interface, the `Compiler`, that maps kernel source strings to instances of the IR, in a language-specific manner. The middle-end exposes extension points defining the quantum intermediate representation, which is a polymorphic object model for representing compiled quantum kernels. It is composed of `Instruction` and `CompositeInstruction` service interfaces that, for gate model computing, are specialized for concrete quantum gates and composites of those gates, respectively. The middle-end also

exposes an IRTransformation service interface that enables the general transformation of CompositeInstructions, important for quantum compilation tasks such as general circuit optimization, low-level synthesis, analysis, and circuit placement. Finally, the backend layer exposes an extensible interface for injecting physical and virtual quantum computing backends—the Accelerator. XACC puts forward another critical concept for modeling an allocation of quantum memory (a register of qubits) called the AcceleratorBuffer. This data structure spans the three architectural layers and is instantiated by programmers and passed to backend Accelerators for execution—we say Accelerators execute CompositeInstructions on a given AcceleratorBuffer. The results of execution are persisted to the buffer and immediately available to the programmer via the original instantiated reference to the buffer.

These core abstractions—kernel Compilers, Instructions and Composite Instructions, IRTransformations, Accelerators, and AcceleratorBuffers—make up the key elements that will be leveraged in our single-source C++ programming model and language extension implementation. High-level quantum kernels in qcor will have a corresponding CompositeInstruction instance that will be generated by variants of the Compiler service. Quantum compilation optimization and placement routines will be injected as implementations of the IRTransformation. The retargetability of the compiler will be due to the interchangeable characteristic of backend Accelerators. The language extension representation of a register of qubits, or *qreg*, will be represented under the hood as an AcceleratorBuffer.

3.2 QCOR Specification

The language extension specification proposed in Reference [41] defines a single-source programming model for heterogeneous quantum-classical quantum computing that leverages a shared memory model and an asynchronous task-based execution model. Moreover, it puts forward a data model that provides a set of abstractions for describing general hybrid quantum-classical variational algorithms for near-term quantum computation. The qcor compiler implementation, in tandem with XACC, implements this specification for the case of extending the C++ programming language. The data model specification puts forward the Operator, Optimizer, and ObjectiveFunction abstractions for composing hybrid variational algorithms, and the taskInitiate() call for asynchronous execution. Operators represent quantum mechanical operators or compositions of operators that can observe unmeasured quantum kernels (if the Operator is Hermitian), returning a list of measured quantum kernels. An example of this would be an Operator subtype representing Pauli operators or sums of Pauli tensor products. The Optimizer class abstraction represents a multi-variate function optimization strategy (COBYLA [43], L-BFGS [16], Adam [34], etc.). We have provided implementations of the Operator and Optimizer as part of the latest release of XACC [39]. The ObjectiveFunction class represents a multi-variate function that returns a scalar value, and evaluation of the function to produce that scalar requires quantum co-processor execution. An example of this would be the **variational quantum eigensolver (VQE)** workflow, where one has a parameterized circuit and would like to execute the circuit and evaluate the expectation value of some Operator.

3.3 Clang Plugins

We base our qcor compiler implementation upon the Clang compiler frontend infrastructure due to its excellent support and utility in academia and industry, its overall extensibility and modularity, and its ability to enable the injection of custom plugin implementations for various aspects of the compiler frontend workflow. Clang is the C++ frontend for the LLVM compiler infrastructure [36], responsible for converting C++ source code into LLVM's intermediate representation. Clang uses LLVM to compile C++ source code to executable objects, and in addition, can perform tasks such

as static analysis and source rewriting. At a high-level, the Clang infrastructure puts forward a robust object model for lexing, parsing, preprocessing, **abstract syntax tree (AST)** generation, and LLVM-IR code generation. Clang supports several plugin interfaces that can be used, in arbitrary combination, to enhance Clang's ability to process C++ source code. Existing plugin interfaces are the `ASTConsumer`, allowing a plugin to monitor the creation of AST nodes, and the `PragmaHandler`, allowing a plugin to process custom pragma directives. Plugins, in general, have access to Clang's AST data structures and the state describing how an individual C++ source file is being compiled.

An early design goal of this work that separates it from others in the field is to ensure that all Clang extensions for enabling `qcor` functionality and features are contributed as separate plugin implementations. We explicitly avoid making permanent modifications to the core of Clang or LLVM. Doing so would force us to maintain a separate fork of these huge code-bases. We adopt the simpler route: Extend key points of the preprocessing workflow with custom plugin implementations, and ultimately enable users to build `qcor` off existing Clang/LLVM binary installs. To this effect, `qcor` makes use of a newly proposed plugin interface: the syntax handler, `SyntaxHandler` [31]. The syntax handler allows embedding of domain-specific languages into C++ function definitions. Each syntax handler implementation (see Appendix, Figure 23) registers to handle a specific, named syntax tag. Functions with the C++ attribute `[[clang::syntax(tag)]]` are processed by Clang's parser in a special way. First, the function body is extracted by collecting all tokens prior to the closing "}" using balanced-delimiter matching. Thus, while the text in the body of the function does not need to be valid C++ code, it is subject to C++ preprocessing and cannot contain unbalanced "{" and "}" characters. The token stream is then provided to the syntax-handler plugin along with information about the already-parsed function declarator. The declarator contains information about the function's name and arguments. The plugin provides, in return, a replacement text stream for the function. This text stream is then subjected to tokenization, much in the same way as an included source file might be handled, and parsing continues using the replacement text instead of the original function body. As described in Section 4.2.1), we leverage this plugin interface to translate our quantum kernel expressions to valid C++ API calls. More details on the `SyntaxHandler` can be found in the Appendix.

4 QCOR

Ultimately, the `qcor` compiler implementation is composed of a runtime library as well as a Clang `SyntaxHandler` implementation enabling compilation of quantum kernels to valid C++ API calls (specifically, calls to the runtime library, and ultimately XACC). The runtime library puts forward a number of key abstractions that implement the original specification. Specifically, the runtime library provides a `QuantumKernel` class abstraction, implementations of `ObjectiveFunction`, `Operator`, and `Optimizer`, and a novel quantum runtime library API. The compiler provides a Clang `SyntaxHandler` that ensures quantum kernel domain specific languages (invalid code with respect to other compilers) are mapped to appropriate and valid subtypes of the `QuantumKernel` abstraction, as well as other utility functions. This mechanism ensures the quantum language-agnostic characteristic of our specification and implementation. The compiler module of `qcor` currently enables programming in XASM, OpenQASM, and a custom language for expression unitary matrices to be decomposed into native one and two qubit quantum gates.

4.1 Runtime

4.1.1 Quantum Kernel. The QCOR specification stipulates that the quantum kernel must be some functionlike object with a function body composed of quantum code provided in some domain specific language, and execution of the function affects execution of that quantum expression on the quantum co-processor. Beyond that, the specification currently allows language extension

implementations to freely describe the kernel object model in a way that best suits the language being extended. For this C++ extension, we specify quantum kernels as standard functions that are annotated with a `__qpu__` attribute, return void, and can take any function arguments, with at least one `qreg` argument. The function body can contain quantum code expressions written in any available quantum language. Here, the word *available* implies the compiler has an appropriate token analysis implementation (a `SyntaxHandler`) for the quantum domain specific language.

To represent this kernel concept as part of the runtime library, `qcor` exposes a `QuantumKernel` class that follows the familiar curiously recurring template pattern (CRTP) [26] and is intended to serve as a super-type for concrete kernel implementations. It takes the type of the subclass as its first template argument (`Derived` in Figure 3), followed by a variadic template parameter pack describing the quantum kernel function argument types (`Args...` in Figure 3). The class keeps a reference to a `std::tuple` of the variadic types and stores concrete function argument instances in the tuple upon construction (the constructor in Figure 3). Crucially, the class also keeps reference to an `xacc::CompositeInstruction` pointer (the `_parent_kernel` member) — an internal representation of this quantum kernel as an XACC IR instance. This is used for ultimate submission to the quantum co-processor (an instance of the XACC `Accelerator`). To promote quantum kernel composition (kernels that call other kernels), `QuantumKernel` exposes a second constructor that takes an upstream `xacc::CompositeInstruction` pointer. So an entry-point kernel (a quantum kernel called from a classical function) can work to fill its `_parent_kernel` instance and then pass that to another kernel instance for it to use as its internal `_parent_kernel`. This pattern directly enables quantum kernel composition. The `QuantumKernel` class is never intended for use on its own, but rather it is meant to be subclassed by concrete quantum kernel representations. The design strategy for subtypes is to inherit from `QuantumKernel`, passing the subtype itself as the first template argument, followed by the kernel function argument types, then provide an implementation of the subtype destructor that ultimately affects execution of the quantum code. Figure 3 demonstrates this, where we have a parameterized quantum kernel, `ansatz`, that takes a `qreg` and `double` parameter. We subclass `QuantumKernel<ansatz, qreg, double>` and provide a means for execution at destruction. Specifically, the subtype should fill the `_parent_kernel CompositeInstruction` and submit for execution. By doing this, one can see that instantiating a temporary instance of `ansatz` looks like quantum kernel function evaluation.

By doing it this way, we allow ourselves the opportunity to provide extra functionality for quantum kernels that one could not get through a standard function alone. For example, defining the `QuantumKernel` class gives us an opportunity to define extra public class methods that enable pertinent analysis tasks, like printing the kernel to an output stream or viewing depth, number of gates, or other circuit-specific information. Moreover, this gives us the opportunity to automatically generate related circuits, and implement common patterns such as compute-action-uncompute [46] via appropriate class attributes on the `QuantumKernel` subtype.

The `QuantumKernel` class is primarily intended to serve as an internal representation of the quantum kernel function that enables high-level programmability, as well as provide extra internal features for compiler and library developers. Therefore, the primary goal of the `qcor` compiler is to *automatically* map quantum kernel functions to appropriate definitions of `QuantumKernel` subtypes.

4.1.2 Quantum Runtime. The `qcor QuantumRuntime` exposes a class API for compiler and runtime developers to execute low-level quantum gate instructions on a specified quantum backend. This class represents a critical piece of the `qcor` runtime library architecture in that it provides an extensible hardware abstraction layer enabling typical quantum instruction execution. Moreover, it promotes the utility of different models of quantum-classical integration—remote, near-term

```
 ... user-defined quantum kernel ...
__qpu__ void ansatz(qreg q, double x) {
  X(q[0]);
  Ry(q[1], x);
  CX(q[1], q[0]);
}

... QuantumKernel definition ...
template<typename Derived, typename... Args>
class QuantumKernel {
 std::tuple<Args...> args_tuple
public:
 QuantumKernel(Args... args);
};

... user kernel as QuantumKernel subtype ...
class ansatz :
        public QuantumKernel<ansatz,
                             qreg, double> {
protected:
  void operator()(qreg q, double x) {
    // fill _parent_kernel
    // add x, ry, cx using QuantumRuntime
  }
public:
  ~ansatz() {
    auto [q,x] = args_tuple;
    operator()(q,x);
    // submit _parent_kernel via QuantumRuntime
  }
}
... instantiating performs like execution ...
ansatz(q, 2.2);
... can get auto-generated static methods ...
ansatz::adjoint(q,2.2);
ansatz::ctrl(1, q, 2.2);
```

Fig. 3. Mapping a user-programmed quantum kernel to a `QuantumKernel` subtype. Ultimately execution of quantum kernels is affected by destruction of temporary `QuantumKernel` instances.

models as well as tightly integrated feed-forward models. Here, by feed-forward, we mean models of execution whereby the quantum co-processor is local to the driver CPU, and one has the ability to execute quantum instruction sequentially, and make mid-circuit measurements that enables conditional execution of further quantum instructions.

For near-term applications, the `QuantumRuntime` can be implemented to queue the appropriate gate instructions as each corresponding API call is invoked. This execution paradigm keeps track of an internal representation of the low-level quantum circuit, and for each gate-level API invocation in a given quantum kernel execution context, the internal representation is built up. At the end of this construction or queuing period, the API exposes a `submit()` call that flushes the internal representation, sending the entirety of its contents to be executed on the desired backend. This is demonstrated in Figure 4 as the `NISQ` subtype, and is the default `QuantumRuntime` backend in `qcor`. Specifically, this default implementation of the `QuantumRuntime` API keeps track

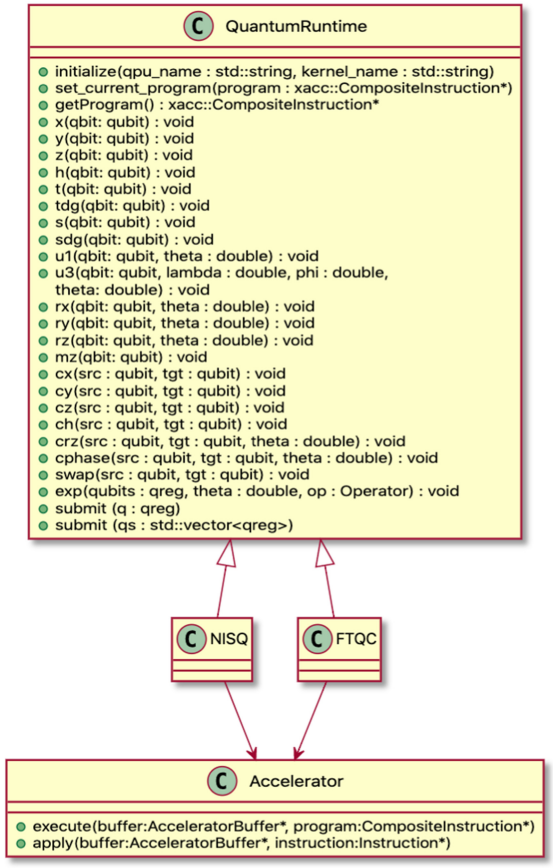

Fig. 4. The class diagram for the `QuantumRuntime` class. We provide implementations of this that enable both remotely hosted QPU execution (NISQ), as well as future fault-tolerant models that stream instruction execution on a tightly integrated quantum backend (FTQC).

of a `xacc::CompositeInstruction` member that it populates upon each invocation of a quantum gate function call. Ultimately, the `QuantumRuntime` API represents a public interface for constructing XACC IR instances programmatically. The `QuantumRuntime` exposes methods for all common single qubit (Hadamard, T, S, Tdg, Sdg, Rx, Ry, Rz, U3, U1, X, Y, Z), two qubit (CX, CY, CZ, CH, CPhase, CRz, Swap), and measurement gates, as well as more complicated circuit synthesis routines like a function for first order trotterization of a provided `qcor Operator` (`exp()` function call). The default `submit` call takes a `qreg` instance and configures the execution of the `xacc::CompositeInstruction` on the backed `xacc::Accelerator` specified at compile time.

To support quantum hardware capable of fast feedback between the quantum processor and the classical processor, we also introduce a **fault-tolerant quantum runtime (FTQC)** (`FTQC` subtype as shown in Figure 4). In this execution model, the runtime library will dispatch quantum instructions to the `Accelerator` backend immediately and reflect any measurement results to the classical code as return values of the `QuantumRuntime::mz()` function. This `FTQC` runtime enables flexible control flow of our quantum kernels such as that required for quantum error correction implementations, whereby syndrome decoding is performed in real-time by a classical computer to determine appropriate correction strategies.

Since our provided `QuantumRuntime` implementations default to XACC, `qcor` picks up support for a number of physical quantum computers via the XACC `Accelerator` extension point, which ultimately handles mapping the XACC IR to the appropriate native gate set. However, one further design goal of this interface is to enable developers to extend the `QuantumRuntime` with a more robust level of support for the designated backend. We anticipate that this interface may enable implementations for specific physical backends, or even for lower-level electronic control system APIs, that provide a more efficient IR-translation mechanism for native backend gate sets.

4.1.3 Operator, Optimizer, and Objective Function. The QCOR specification defines a few class interfaces that seek to enable efficient expression of common quantum algorithms, specifically those that are variational and target potential near-term quantum hardware. These types, the `Operator`, `ObjectiveFunction`, and `Optimizer`, provide the necessary abstractions at a familiar level to enable general variational tasks that leverage quantum co-processing. The `qcor` implementation seeks to enable these class abstractions in a manner that is modular and extensible, allowing future `qcor` developers to tailor these classes to their specific workflow.

First, the `Operator` interface represents a general quantum mechanical operator, or composition of operators. The `Operator` should expose appropriate algebra that enables programmers to build up complicated Hamiltonian models that can be leveraged for quantum simulation. Critically, `Operators` must expose some mechanism for the *observation* of quantum states on the quantum co-processor. By this we mean, given some unmeasured quantum kernel, the `Operator` should return a list of measured kernels, dependent solely on its internal structure. The prototypical example of this would be the VQE algorithm, whereby one has an `Operator` that describes the Hamiltonian of interest consisting of a sum of Pauli tensor products, and one requires quantum kernel executions for each term followed by measurements in the basis of the term itself. `qcor` implements the `Operator` abstraction as a class to be subtyped for specific quantum mechanical operator types, each encoding its own operator algebra. The class exposes an interface for algebra (appropriate operator overloads in C++), as well as common methods for operator analysis. Every `Operator` in `qcor` can be instantiated from a string representation, from a site-map (qubit index to operator name), or from a mapping of options.

`qcor` provides subtypes for Pauli and Fermionic operators, as well as more complicated `Operators` that auto-generate themselves from a map of key-value options (e.g., molecular geometry and basis set name to generate a molecular Hamiltonian, for example). `qcor` provides a creation API for `Operators` that enables efficient expression of quantum operators in a way that is familiar for programmers (see Figure 5). The class architecture for the `Operator` and related types is shown in the Appendix, Figure 25. We further define and provide the `OperatorTransform` interface to serve as an extension point for general transformations on `Operators` (e.g., Jordan-Wigner for mapping Fermionic `Operators` to Pauli ones).

The `ObjectiveFunction` interface in `qcor` represents a multi-variate function that returns a scalar value ($y = F(\mathbf{x})$), and evaluation of the function requires quantum co-processor execution (e.g., the VQE workflow, where one has a parameterized circuit and would like to execute the circuit and evaluate the expectation value of some `Operator`). Ultimately, the `ObjectiveFunction` generalizes the notion of preprocessing, circuit evaluation, and post-processing to produce some scalar value given a vector of input scalar parameters. This concept has proven ubiquitous throughout near-term variational quantum-classical algorithm development and utility. To affect that workflow, `ObjectiveFunctions` requires initialization with both the quantum kernel of interest (passed as a callable or function pointer) and the `Operator` dictating measurements on the kernel. For more details on the class architecture for the `ObjectiveFunction`, please see the Appendix. `qcor` provides a creation API for `ObjectiveFunctions`, `createObjectiveFunction` (see Figure 6), with a

```
// Create Operator from string
auto H = createOperator("pauli",
                "2.2 X0 X1 + 3.3 Y0 Y1");
// Create Operator from X, Y, Z, API
auto H = 5.907 - 2.1433 * X(0) * X(1) -
          2.1433 * Y(0) * Y(1) + .21829 * Z(0) -
          6.125 * Z(1);
// Create from a, adag API
auto H = adag(1) * a(0) + adag(0) * a(1);
// Create from Operator Generators
auto H2_chem =
    createOperator("chemistry",
     {{"basis", "sto-3g"}, {"geometry", H2_geom}});

// Create Optimizer based on NLOPT (COBYLA default)
auto optimizer = createOptimizer("nlopt");
// Create Adam from MLPACK
auto optimizer = createOptimizer("mlpack",
                {{"mlpack-optimizer", "adam"}});
```

Fig. 5. Programmers can easily create `Operators` and `Optimizers` for use in general variational algorithms with qcor.

few overloads: (1) take as input a quantum kernel function and an `Operator`, which defaults to evaluating the expectation value of the `Operator` at the given parameters, and (2) take a kernel and an `Operator`, but also the name of a concrete `ObjectiveFunction` subclass for custom pre- and post-processing around quantum circuit execution. Additionally, each of the public creation functions for `ObjectiveFunctions` requires the number of variational parameters in the quantum kernel. Optionally, programmers can provide a heterogeneous map of options that may affect the construction and use of the `ObjectiveFunction` (e.g., gradient computation information).

Finally, the `Optimizer` interface represents a typical classical multi-variate function optimization strategy (COBYLA, L-BFGS, Adam, etc.). `Optimizers` expose an `optimize()` method that takes as input an `ObjectiveFunction`, which, as demonstrated above, is essentially a function or lambda with the signature `double(const std::vector<double>, std::vector<double>&)`. Here the first argument is the parameter set for the current optimization iteration, while the second argument represents the gradient vector as a reference that can be set. Using this function signature, most classical derivative-free or gradient-based optimization routines are able to be implemented. As of this writing, `qcor` provides implementations of this interface that delegate to the popular NLOpt and MLPack libraries.

4.2 Compiler

The `qcor` compiler implementation handles the complexity behind enabling this novel quantum-C++ language extension through simple extensions to Clang and integration with the QCOR runtime library. Here we go into detail behind the compiler implementation. We specifically highlight our novel implementation of the new Clang `SyntaxHandler` plugin, the overall compiler workflow, and the implementation of a compiler pass manager enabling general transformations on the compiled quantum kernel representation (for both optimization and placement). Ultimately, we introduce a `qcor` compiler executable that provides the same compiler flags programmers are used to, in addition to quantum-specific command line arguments.

```
__qpu__ void foo(qreg, double x) {
 .... quantum circuit using x parameter
}
...
auto H = createOperator("pauli", "X0 + Y1");
int n_params = 1;

// Create Objective to
// evaluate <foo(x) | H | foo(x)>
auto objective =
    createObjectiveFunction(foo, H, n_params);

// Evaluate at a concrete vector of parameters.
auto exp_val_H = (*objective)({1.345});

// Perform parameter sweep
for (auto x : linspace(-constants::pi,
                       constants::pi, 20)) {
  std::cout << "Value at " << x << " is " <<
            (*objective)({x}) << "\n";
}
```

Fig. 6. `ObjectiveFunctions` can be created based on information about the system `Operator`, number of variational parameters, and a parameterized quantum kernel. After creation, they can be evaluated in the expected manner. Here we demonstrate the default VQE objective, returning the expected value of the provided `Operator`.

4.2.1 Syntax Handler. The Clang compiler front-end exposes a modular and extensible set of libraries for common tasks found in the mapping of C, C++, and Objective-C source files to LLVM IR. It has a number of plugin interfaces, or extension points, that enable analysis of the AST representation of a C++ source file. This extensibility enables a single Clang binary install to take on new functionality depending on what plugins are loaded at compile time via standard command line arguments. This approach is optimal for us and the `qcor` compiler implementation. We seek to enable quantum-classical programming in C++ without having to modify core Clang/LLVM code bases, forcing a fork of these efforts and increasing the cost of maintainability for `qcor`.

Our approach leverages a recent plugin interface contribution to Clang—the `SyntaxHandler`—which provides a hook for plugin developers to analyze functions written in any **domain specific language (DSL)** and provide a rewritten token stream to Clang that is composed of valid C++ API calls (see Appendix, Figure 23). This replacement occurs after lexing and preprocessing but before the AST is generated. This plugin interface exposes a `GetReplacement()` method that provides the function body tokens for implementation-specific analysis and an output stream that the implementation uses to provide valid C++ replacement code. The `SyntaxHandler` infrastructure will then replace the invalid DSL code with the provided output stream code and restart tokenization at the beginning of the function. Developers are free to update the function body but can also write new code after it. Additionally, the `SyntaxHandler` exposes an `AddToPredefines()` method that can be used by implementations to add to the current source file's header file include statements.

Our goal is to provide a `SyntaxHandler` implementation that enables the `qcor` C++ language extension. Specifically, we want our users to be able to express quantum kernels in a quantum language agnostic manner, while retaining standard C++ control flow statements and variable declaration and utility. To do so, we implement the `QCORSyntaxHandler` (see Appendix, Figure 27), with

name qcor, which analyzes the incoming Clang CachedTokens reference and attempts to perform two tasks: (1) translate the quantum code itself into appropriate QuantumRuntime API calls, and (2) define a QuantumKernel<Derived, Args...> subtype and associated function calls. The first task relies on a further extension point we provide called the TokenCollector, which we implement for the various quantum languages that we support. qcor currently has support (TokenCollector implementations) for XASM, OpenQASM, and a special circuit synthesis language that lets programmers describe their quantum code at the unitary matrix level. The TokenCollector exposes a single collect() method that allows implementations to map incoming clang Tokens to valid QuantumRuntime API calls dependent on the language corresponding to the implementation. Those QuantumRuntime calls are written to a provided std::stringstream that is passed down from the QCORSyntaxHandler. A unique feature of this architectural decomposition is that one can switch TokenCollectors while analyzing a given sequence of CachedTokens. This means that, dependent on some language extension syntax, one can define quantum kernels using multiple quantum languages within the same quantum kernel. In qcor, the default quantum kernel language is XASM, but we permit switching to other languages via a using qcor::LANG; statement. So to switch from the default XASM to OpenQASM for instance, and trigger internally a switch to the OpenQASM TokenCollector, one would simply write using qcor::openqasm; (see Figure 7). This is a useful feature, since some languages do provide more efficient expressability for various quantum programming tasks.

After the token collection phase of the QCORSyntaxHandler workflow, the provided stringstream contains the rewritten QuantumRuntime API code for creating and executing the described quantum kernel. The details of how each TokenCollector implementation works is of critical importance. The most well-supported TokenCollector in qcor is the XASMTokenCollector. This implementation works by leveraging the XACC XASM Compiler implementation on a statement-by-statement basis. Specifically, it will attempt to compile each statement with this Compiler to map the statement to an XACC Instruction instance. If that mapping succeeds, then the Instruction is mapped to a QuantumRuntime API call via an appropriate XACC InstructionVisitor (e.g., the H(q[0]) call mapped to a quantum::h(q[0]) call. If that mapping fails, then the statement string itself is retained and is assumed to be some classical code that must be part of the QuantumRuntime rewritten source string (e.g., the for statement in Figure 7). The OpenQasmTokenCollector collects the incoming Clang Tokens and leverages the XACC Staq Compiler implementation to map each OpenQASM statement to an XACC Instruction instance. Staq [2] is a C++ library providing an abstract syntax tree for the OpenQASM 2.0 language. XACC provides a number of plugins that delegate to this library for mapping the XACC IR to OpenQASM or for performing Staq-provided circuit placement and optimization strategies.

We have also developed a means for programming at the unitary matrix level through an appropriate implementation of the TokenCollector. First, we define the qcor::UnitaryMatrix data structure, which is simply a typedef for a complex matrix provided by the Eigen matrix library [21]. Next, we enable a decompose keyword as part of our quantum kernel language extension, which programmers declare, open a new scope, and define their unitary matrix using the qcor::UnitaryMatrix API. Programmers close that new scope and provide further arguments indicating the qreg to operate on and the name of the specific circuit synthesis algorithm to employ in decomposing the unitary matrix to gate-level quantum instructions. Figure 8 demonstrates how this circuit synthesis mechanism can be leveraged. Effectively, the UnitaryMatrixTokenCollector is invoked when the decompose syntax is observed during token analysis, and the kernel is rewritten to a qcor runtime call that initiates the decomposition of the given unitary matrix using appropriate XACC circuit synthesis routines.

```
__qpu__ void bell(qreg q) {
  H(q[0]);
  using qcor::openqasm;
  cx q[0], q[1];
  using qcor::xasm;
  for (int i = 0; i < q.size(); i++)   {
    Measure(q[i]);
  }
}
-------- After Token Collection ---------
quantum::h(q[0]);
quantum::cx(q[0], q[1]);
for (int i = 0; i < q.size(); i++) {
  quantum::mz(q[i]);
}
```

Fig. 7. Quantum kernel function bodies can be written in a mixture of available languages, enabled via the `TokenCollector` infrastructure.

```
__qpu__ void unitary(qreg q) {
  decompose {
    // Create the unitary matrix
    UnitaryMatrix ccnot_mat =
        UnitaryMatrix::Identity(8,8);
    ccnot_mat(6, 6) = 0.0;
    ccnot_mat(7, 7) = 0.0;
    ccnot_mat(6, 7) = 1.0;
    ccnot_mat(7, 6) = 1.0;
  }
  (q);
}
```

Fig. 8. Programmers can optionally define quantum code a the unitary matrix level. Decomposition to native gates is provided by the compiler and is itself an extension point for the platform.

The second task for the `QCORSyntaxHandler` is to rewrite the quantum kernel function and define a new `QuantumKernel<Derived, Args...>` subtype, incorporating the results of the first task - the rewritten `QuantumRuntime` code. Rewriting the function call as a `QuantumKernel` subtype gives us auto-generated `adjoint/ctrl` methods and provides an avenue for future kernel extensions enabling novel functionality. Our rewrite strategy is as follows: (1) Rewrite the original function to forward declare a `__internal_call_function_KERNELNAME` function and immediately call that function (its implementation will follow the `QuantumKernel` subtype declaration); (2) define the `QuantumKernel` subtype, and implement its `operator()(Args...)` method with the `QuantumRuntime` code generated from the token handling phase; and (3) define the internal function call we forward declared in the original function, with an implementation that simply instantiates a temporary instance of the new `QuantumKernel` subtype (immediately calling the destructor that affects quantum backend execution of the quantum code). An example of this rewrite is given in Figure 9. We also add a function after the subtype definition that takes a `CompositeInstruction` as its first argument, which is used internally to enable kernel composition.

Programmers see quantum kernel functions, but at compile time, these function are expanded into a new subclass definition of the `QuantumKernel`. The first subclass constructor takes as input the original function arguments, and calls the corresponding constructor on the superclass. This configures the kernel to be callable (`is_callable` equals `true`). In the case of a `NISQ` `QuantumRuntime`, instantiation and destruction of a kernel constructed this way will build up the internal `CompositeInstruction` via the `QuantumRuntime` API calls, and invoke `submit()` to execute on the backend `Accelerator`. For the `FTQC` `QuantumRuntime`, instantiations and destruction invokes the `QuantumRuntime` calls, which immediately affect execution of the single instruction on the backend `Accelerator`. Note that if the kernel has not been called, then the `_parent_kernel` is null, so the first task of `operator()(Args...)` is to create it. It is then given to the `QuantumRuntime` API and used for construction, or immediate execution, of the circuit. The second constructor takes as its first argument an already constructed `_parent_kernel`, which is set on the new instance's `_parent_kernel` attribute. Now when `operator()(Args...)` is called, a new `_parent_kernel` is not created, and the incoming one from instantiation is used. This directly enables kernel composition—the second constructor is always used for quantum kernels called from other quantum kernels. If this second constructor is used, then `is_callable` equals

```
void bell(qreg q) {
  void __internal_call_function_bell(qreg);
  __internal_call_function_bell(q);
}
class bell :
    public qcor::QuantumKernel<class bell_multi,
                               qreg> {
  friend class
    qcor::QuantumKernel<class bell, qreg>;

protected:
  void operator()(qreg q) {
    if (!parent_kernel) {
      parent_kernel = qcor::__internal__::
          create_composite("bell");
    }
    quantum::set_current_program(parent_kernel);
    quantum::h(q[0]);
    quantum::cnot(q[0], q[1]);
    for (int i = 0; i < q.size(); i++) {
      quantum::mz(q[i]);
    }
public:
  bell(qreg q) : QuantumKernel<bell, qreg>(q) {}
  bell(std::shared_ptr<qcor::CompositeInstruction>
              _parent, qreg q)
      : QuantumKernel<bell, qreg>(_parent, q) {}
  virtual ~bell() {
    auto [q] = args_tuple;
    operator()(q);
    if (is_callable) {
      quantum::submit(q.results());
    }
  }
};
void bell(
    std::shared_ptr<qcor::CompositeInstruction>
                    parent, qreg q) {
  class bell_multi k(parent, q);
}
void __internal_call_function_bell(qreg q) {
  class bell_multi k(q);
}
```

Fig. 9. The `QCORSyntaxHandler` translates quantum kernels (like the kernel in Figure 1) into new function calls and a `QuantumKernel<Derived,Args...>` subclass definition.

`false`, and `submit()` is never called on the kernel. For remote execution, submission to the backend is only ever invoked for entry-level quantum kernels.

4.2.2 Pass Manager. As mentioned, the `QuantumRuntime` API exposes a `submit()` call that affects execution of the constructed `CompositeInstruction` on the desired backend `Accelerator`. Upon invocation of this call, the runtime-resolved quantum IR tree is completely flattened and only

Table 1. Descriptions of Hardware Placement Strategies that are Implemented for qcor

Pass Name	Description
circuit-optimizer	A collection of simple pattern-matching-based circuit optimization routines.
single-qubit-gate-merging	Combines adjacent single-qubit gates and finds a shorter equivalent sequence if possible.
two-qubit-block-merging	Combines a sequence of adjacent one and two-qubit gates operating on a pair of qubits and tries to find a more optimal gate sequence via Cartan decomposition if possible.
rotation-folding	A wrapper of the Staq's RotationOptimizer [2] that implemented the rotation gate merging algorithm.
voqc	A wrapper of the VOQC OCaml library [24], which implements generic gate propagation and cancellation optimization strategy.

contains simple quantum assembly instructions to be submitted to the specified QPU. Therefore, this submission API is ammenable for the implementation of a **just-in-time (JIT)** quantum circuit optimization and transformation sub-system that utilizes best-known techniques in the field of circuit optimization to further simplify the circuit before sending it to the target QPU. Since qcor is built upon the XACC framework, it is well positioned to serve as an integration framework for state-of-the-art quantum compilation strategies coming from experts in the field. We specifically design our JIT quantum compilation system to build upon XACC's plugin extensibility to enable a diverse set of quantum compilation strategies.

Adopting the ubiquitous LLVM optimization framework pattern for user-contributed IR transformation strategies, we structure runtime circuit optimization tasks into *passes* that simplify the input circuit in terms of gate count and depth. The application of runtime optimization passes is handled by a class called `PassManager`, and passes are implemented as subtypes of the XACC `IRTransformation`, and are invoked by the `PassManager`. This approach enables the `qcor PassManager` to inherit a well-established set of circuit optimizers from XACC, such as the implementations of the rotation folding and the phase polynomial optimization algorithms. Table 1 provides the default circuit optimizer passes (`xacc::IRTransformations`) that `qcor` leverages.

Based on internal profiling, we further define optimization *levels* that dictate the set of passes and their execution order. The goal here is to strike a balance between the potential gate count reduction and the optimization time. For example, invoking the `qcor` compiler with "`-opt 1`" command-line option will activate optimization level 1. It is worth noting that since this option controls the final JIT optimization of the quantum kernel before remote execution, it will not impact the compile time of top-level classical-quantum code. The produced executable will contain the selected optimization level to pass over to the `PassManager` that then selects and loads appropriate `IRTransformation` modules to optimize the quantum IR tree. Once all passes have completed, the simplified circuit will be sent to the QPU for execution.

More advanced users can also specify an ordered list of passes to be executed by using the `qcor`'s "`-opt-pass`" option. External developers can thus develop in-house passes adhering to the `IRTransformation` API and integrate them into the `qcor` compilation and execution workflow using this compile option. For example, we have made available two `IRTransformation` plugins that wrap the C++ Staq rotation folding [2] and the OCaml-based **Verified Optimizer for Quantum Circuits (VOQC)** [24] optimizers, thereby demonstrating the cross-language extensibility of the `qcor` circuit optimization sub-system.

Table 2. Descriptions of Hardware Placement Strategies That Are Implemented for qcor

Name	Description
Swap shortest path	Implement permutation-based mapping for uncoupled qubits [2].
Noise Adaptive	Optimize a noise-adaptive layout [28] based on backend calibration data (gate errors.)
Sabre	Implement SWAP-based BidiRectional heuristic search algorithm (SABRE) [37].
QX Mapping	Implement the IBM-QX contest-winning technique [48].

For diagnostic purposes, the `PassManager` analyzes detailed statistics about each pass, such as the execution time, the gate count distribution before and after the pass, which could be retrieved for analysis. In Section 5.7, we will show some statistics of the passes that are currently available in the qcor-XACC ecosystem.

4.2.3 Placement. When qcor compiles the executable for a target accelerator backend, it also takes into account the qubit connectivity as well as any user-defined mappings to project the logical qubit indices as defined in the quantum kernel onto the actual physical qubit indices on hardware. This hardware placement functionality often involves (1) permutations of gates and qubits, e.g., by inserting SWAP gates, so that the resulting circuit satisfies the device topology constraints and (2) direct logical-physical qubit mapping to take advantage of best-performing qubits.

To address the first task, qcor defaults to an `xacc::IRTransformation` implementation delegating to the Staq [2] library providing a generic shortest path permutation algorithm (`swap-shortest-path`) whereby two-qubit gates between uncoupled qubits are swapped to satisfy the coupling graph. Figure 10 demonstrates such mapping when we compile the same kernel source for two different IBMQ device targets, namely the Ourense and Yorktown 5-qubit backends. Since their connectivity graphs are different, the resulting circuits after placement are also different. Specifically, the sequence of CNOT gates was permutated to match the backend topology and the measure gates are also swapped accordingly. It is worth noting that this propagating permutation approach is more efficient than a SWAP gate-based solution, since we do not need to swap the qubits back and forth. Besides `swap-shortest-path`, Table 2 provides the details of hardware placement strategies that are available in qcor.

Manual qubit-to-qubit mapping functionality is also available in qcor. In particular, by supplying a "`-qubit-map`" option along with a sequence of qubit indices to qcor, the runtime placement service will map logical qubits to the physical ones according to this map. For example, depending on the readout and gate error information of the backend, we may want to use qubit 5 and 6 for a two-qubit quantum kernel that was written in terms of `q[0]` and `q[1]` by simply compiling with "`-qubit-map 5,6`."

4.2.4 Automated Error Mitigation. XACC enables automated error mitigation via decoration of the `Accelerator` backend [39]. The `AcceleratorDecorator` service interface inherits from `Accelerator` but also contains an `Accelerator` member reference, enabling an `Accelerator::execute()` override that provides an opportunity for pre- and post-processing around execution of the decorated `Accelerator`. For error mitigation, this is used to analyze or update the incoming compiled circuit, execute it, and analyze and mitigate the results based on

```
// Create a multi-qubit entangled state
__qpu__ void entangleQubits(qreg q) {
  H(q[0]);
  for (int i = 1; i < q.size(); i++) {
    CX(q[0],q[i]);
  }
  for (int i = 0; i < q.size(); i++) {
    Measure(q[i]);
  }
}

int main() {
  // Create a 4-qubit register
  auto q = qalloc(4);
  // Execute the kernel
  entangleQubits(q);
  // Expect: ~50-50 for "0000" and "1111"
  q.print();
}
```

```
// Target ibmq_ourense backend:
// qcor -qpu aer:ibmq_ourense
H q0              ------------------------
CNOT q1,q0        | Ourense Connectivity |
CNOT q0,q1        |  (0) -- (1) -- (2)   |
CNOT q1,q2        |          |           |
CNOT q1,q3        |         (3)          |
Measure q1        |          |           |
Measure q0        |         (4)          |
Measure q2        ------------------------
Measure q3
```

```
// Target ibmqx2 (ibmq_5_yorktown) backend
// qcor -qpu aer:ibmqx2
H q0              ------------------------
CNOT q0,q1        | ibmqx2 Connectivity  |
CNOT q2,q0        |        (1)           |
CNOT q0,q2        |      /  |            |
CNOT q2,q3        |    (0)-(2)-(3)       |
Measure q2        |          | /         |
Measure q1        |        (4)           |
Measure q0        ------------------------
Measure q3
```

Fig. 10. qcor automates qubit placement of logical program connectivity to physical hardware connectivity. (Top) qcor source code and final circuits after placement for the IBMQ's (middle) Ourense and (bottom) Yorktown backends.

the subtype's implemented strategy. Since qcor builds upon XACC and ultimately targets backend `Accelerators`, this mechanism should also be readily available to users of the qcor language extension and compiler.

We have added this capability to qcor via an `-em` command line option. This option flag lets users specify the name of a decorator to use to automatically apply error mitigation to kernel

```
__qpu__ void noisy_zero(qreg q) {
    for (int i = 0; i < 100; i++) {
        X(q[0]);
    }
    Measure(q[0]);
}
int main() {
  auto q = qalloc(1);
  noisy_zero(q);
  printf("Expectation: %f\n", q.exp_val_z());
}
-----------------------------------------------
$ qcor -qpu aer[noise-model:noise_model.json] \
    -shots 4096 -o noisy.x zne_test.cpp
$ ./noisy.x
Expectation: 0.895996
$ qcor -qpu aer[noise-model:noise_model.json] \
    -shots 4096 -em mitiq -o mitiq_noise.x \
    zne_test.cpp
$ ./mitiq_noise.x
Expectation: 1.02295
```

Fig. 11. qcor optionally enables various error mitigation strategies from the command line. Here we show error mitigation via the Mitiq library providing zero-noise extrapolation.

invocations. The code snippet in Figure 11 demonstrates this, whereby we have a quantum kernel that applies a large, even number of X gates on a single qubit, theoretically resulting in the $|0\rangle$ state. Due to the presence of noise, this will not be the case, and we should observe an expectation value with respect to Z measurements that drifts from the true value of 1.0. The bottom half of this snippet shows how one would use this error mitigation flag. Here we compile to the IBM noise-aware Aer simulation backend, providing a custom noise model file as an option. Execution of this compiled executable results in a noisy expectation value, as expected. We next compile with the same noise model but additionally indicate that qcor should apply error mitigation from the Mitiq library [29, 42], which provides routines for zero-noise extrapolation [30]. Executing the compiled executable this time we see that the result has been shifted closer to the true value of 1.0. qcor enables one to stack these decorators by passing more than one -em flag, and the order with which they are seen on the command line will represent the order they will be executed.

4.2.5 Compiler Workflow. The architecture described above ultimately puts forward C++ libraries that provide pertinent qcor runtime and compile-time capabilities. In order for programmers to interface with this novel infrastructure, we provide a qcor compiler command-line executable. This executable is meant to directly mimic existing compilers like clang++ and g++, but with the addition of quantum-pertinent command line options. We provide this compiler as an executable Python script, which delegates to a clang++ sub-process call configured with all necessary include paths, library link paths, libraries, and compiler flags required for executing the qcor compilation workflow. Of critical importance is the loading of the QCORSyntaxHandler plugin library, which enables the underlying clang++ call to operate on defined quantum kernels and transform them to valid C++ code. Additionally, the qcor compiler exposes a -qpu compiler flag that lets users dictate what quantum backend this source file should be compiled to. The quantum backend name provided follows the XACC syntax for specifying Accelerators (e.g., *accelerator_name:backend_name*). As seen in Section 4.2.2, the compiler also exposes -opt LEVEL

and `-opt-pass PASSNAME` arguments to turn on quantum circuit optimization. Just like existing classical compilers, `qcor` can be used in compile-only mode (`-c SOURCEFILE.cpp`) as well as in link-mode.

The overall compiler workflow is fairly simple, and can be described as follows: (1) invocation of `qcor` on a quantum-classical C++ source file, indicating the backend QPU to target, (2) `clang++` is invoked and loads the `QCORSyntaxHandler` plugin library, (3) usual Clang preprocessing and lexing occurs, (4) the `QCORSyntaxHandler` is invoked on all `__qpu__` annotated functions, translating them to a set of new functions and a `QuantumKernel` definition, as in Figure 9, and (5) finally, classical compilation proceeds with this rewrite (AST generated, LLVM IR CodeGen executed). The user is left with a classical binary executable or object file (depending on whether `-c` was used). Invocation of the executable proceeds as it would normally (`./a.out`, or whatever the executable was named).

5 DEMONSTRATION

Now we turn to some illustrative examples of using the `qcor` compiler infrastructure. Specifically, we detail code snippets demonstrating the level of quantum-classical programmability that `qcor` provides, as well as novel library data structures and API calls for affecting execution of useful quantum algorithms (VQE [12], QAOA [47], **quantum phase estimation (QPE)** [18], etc.).

5.1 Quantum Phase Estimation

The QPEalgorithm is a seminal quantum subroutine that computes the eigenvalue of a unitary matrix for a given eigenvector. From a programming perspective, this algorithm demonstrates some intriguing aspects of the composability and synthesis of quantum programs. In particular, the input to the algorithm is a black box operation U (*oracle*) that we must be able to apply conditioned on a qubit. Hence, the compiler needs to figure out the decomposition in terms of basic gates to implement that arbitrary controlled-U operation. In `qcor`, each user-defined quantum kernel has intrinsic `adjoint()` and `ctrl()` extensions, which automatically generate the adjoint and controlled circuits.

We demonstrate the programmability of the QPE algorithm in Figure 12. The oracle is expressed as a `qcor` kernel (annotated with `__qpu__`) named `compositeOracle` that only contains a single T gate operating on the last qubit of the provided quantum register. It is worth noting that the oracle can be an arbitrarily complex circuit or even be specified as a unitary matrix using the `qcor` unitary decompose extension. Given this oracle kernel, the QPE algorithm requires the application of controlled-U^k operations (where k is the number of applications of the controlled U).

Thanks to the `for` loop and the built-in `ctrl` kernel extension, the algorithm is expressed in a very succinct manner yet generic for arbitrary oracles. There is another language feature that we also want to point out in this example. We take advantage of the **Inverse Quantum Fourier Transform (`iqft`)** kernel that is pre-defined in the `qcor` standard library by simply including the appropriate header file (`qft.hpp`). The algorithm is implemented for generic cases allowing us to specify a subset of the qubit register to act upon and to control whether or not we need to add SWAP gates at the beginning of the circuit.

5.2 GHZ State on a Physical Backend

To demonstrate `qcor`'s ability to compile to physical backends, here we demonstrate a simple GHZ experiment on a 5-qubit physical backend from IBM. The logical connectivity of this problem will not directly map to the physical connectivity of the backend we target (`ibmq_vigo`), but `qcor` handles this by applying an appropriate placement strategy, as described in Section 4.2.2. The code snippet in Figure 13 (top) shows a simple kernel that runs the GHZ state on 5 qubits. In `main()`,

```
// QCOR standard library
#include "qft.hpp"
// The Oracle: a T gate
__qpu__ void compositeOracle(qreg q) {
  int last_qbit = q.size() - 1;
  T(q[last_qbit]);
}
__qpu__ void QuantumPhaseEstimation(qreg q) {
  const auto nQbits = q.size();
  // Prepare eigenstate (|1>)
  X(q[nQubits - 1]);

  // Apply Hadamard gates to the counting qubits:
  for (auto qIdx : range(nQbits-1)) {
    H(q[qIdx]);
  }
  // Apply Controlled-Oracle
  auto bitPrecision = nQbits - 1;
  for (auto i : range(bitPrecision)) {
    const int nbCalls = 1 << i;
    for (auto j : range(nbCalls)) {
      int ctlBit = i;
      // Controlled-Oracle:
      // in this example, Oracle is T gate;
      compositeOracle::ctrl(ctlBit, q);
    }
  }
  // Inverse QFT on the counting qubits:
  int startIdx = 0, shouldSwap = 1;
  iqft(q, startIdx, bitPrecision, shouldSwap);
  // Measure counting qubits
  for (auto qIdx : range(bitPrecision)) {
    Measure(q[qIdx]);
  }
}
int main(int argc, char **argv) {
  // Allocate 4 qubits, i.e. 3-bit precision
  auto q = qalloc(4);
  QuantumPhaseEstimation(q);
  // print results, expect "100" bitstring
  q.print();
}
```

Fig. 12. Through standard control flow and auto-generated `ctrl` circuits, one is able to put together complex quantum-classical algorithms like quantum phase estimation.

```
__qpu__ void ghz(qreg q) {
    H(q[0]);
    for (int i = 0; i < q.size()-1; i++) {
        CX(q[i], q[i+1]);
    }
    for (int i = 0; i < q.size(); i++) {
        Measure(q[i]);
    }
}
// helper to show histogram of counts
void plot_counts(auto&& counts) {...}
int main() {
    auto q = qalloc(5);
    ghz::print_kernel(std::cout, q);
    ghz(q);
    plot_counts(q.counts());
}
```

```
$ qcor -qpu ibm:ibmq_vigo ghz.cpp ; ./a.out
H q0            --------------------
CNOT q0,q1      | Vigo Connectivity |
CNOT q1,q2      | (0) -- (1) -- (2) |
CNOT q2,q1      |         |         |
CNOT q1,q2      |        (3)        |
CNOT q2,q1      |         |         |
CNOT q1,q3      |        (4)        |
CNOT q3,q4      --------------------
Measure q0
Measure q2
Measure q1
Measure q3
Measure q4
```

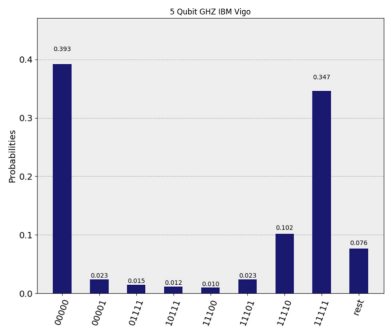

Fig. 13. (top) Preparing a GHZ state on n-qubits with `qcor`. (middle) Standard out from code in (top). The default placement strategy has been applied to enable all CNOTs in the logical program. (bottom) Results of running the code in (top) on the `ibm_vigo` physical backend. Execution on Aug. 27, 2020. 11:03 AM EDT, IBM Job-Id: 5f47cb10654c28001b53b144.

we allocate the 5-qubit `qreg`, print the kernel to see the results of placement on the `ibm_vigo` backend, run the kernel, and output the bit strings and corresponding counts observed.

The results of compilation and execution of this code are shown in Figure 13 (middle) and (bottom), where one can clearly see the presence of the `SWAP` to enforce the logical connectivity of the program (introduced by the default Staq `swap-shortest-path` placement strategy). The results

```
// Measure Z0Z1 and Z1Z2 syndromes
// and recover from a bit-flip error.
__qpu__ void correctLogicalQubit(qreg q,
                int logicalIdx, int ancIdx) {
  int physicalIdx = logicalIdx * 3;
  // Step 1: Measure Z0Z1
  CX(q[physicalIdx], q[ancIdx]);
  CX(q[physicalIdx + 1], q[ancIdx]);
  // Measure ancilla to determine the syndrome.
  const bool parity01 = Measure(q[ancIdx]);
  if (parity01) {
    // Reset ancilla qubit for reuse
    X(q[ancIdx]);
  }
  // Step 2: Measure Z1Z2
  CX(q[physicalIdx + 1], q[ancIdx]);
  CX(q[physicalIdx + 2], q[ancIdx]);
  // Measure ancilla to determine the syndrome.
  const bool parity12 = Measure(q[ancIdx]);
  if (parity12) {
    // Reset ancilla qubit for reuse
    X(q[ancIdx]);
  }
 ... function continued on next pannel --> ...
```

```
  // Step 3: Correct bit-flip errors
  // based on parity results:
  //     Error | Z0Z1 | Z1Z2
  //     ==================
  //       Id  |False |False
  //       X0  |True  |False
  //       X1  |True  |True
  //       X2  |False |True
  if (parity01 && !parity12)
        {X(q[physicalIdx]);}
  if (parity01 && parity12)
        {X(q[physicalIdx+1]);}
  if (!parity01 && parity12)
        {X(q[physicalIdx+2]);}
}
// Run a full QEC cycle on bit-flip
// code encoded qubit register.
__qpu__ void runQecCycle(qreg q) {
  int nbLogicalQubits = q.size() / 3;
  int ancBitIdx = q.size() - 1;
  for (int i = 0; i < nbLogicalQubits; ++i) {
    correctLogicalQubit(q, i, ancBitIdx);
  }
}
```

Fig. 14. Code snippet demonstrating bit-flip quantum error correction code. The `runQecCycle` kernel iterates over all *logical* qubits (encoded as three consecutive physical qubits) and performs syndrome detection and correction (using `correctLogicalQubit` helper kernel). Compilation requires the `-qrt ftqc` flag.

indicate the typical noise present in execution on NISQ hardware, but one can see the dominant observed configurations of `00000` and 11111, as expected.

5.3 Feed-Forward Error Correction

In this demonstration, we seek to illustrate the utility of the FTQC runtime to implement **quantum error correction (QEC)**, which is a crucial aspect of fault-tolerant quantum computation. Specifically, we examine the implementation of the canonical QEC feedback (syndrome) and feed-forward (correction) loop of a toy three-qubit bit-flip encoding scheme, as shown in Figure 14. The syndrome signatures (`parity01` and `parity12` Boolean variables) detected by measurement operations are used to infer the most probable bit-flip location for correction. Albeit its simplicity, this model of error correction immediately generalizes to other codes that could require much more complex decoding mechanisms such as the Blossom [32] or maximum-likelihood [4] algorithms for the surface code [9].

5.4 Multi-Language Kernel Development

The `SyntaxHandler` and `TokenCollector` architecture gives us a unique opportunity for general embedded domain-specific language processing in C++ for quantum programming. Moreover, as implemented, it gives us the ability to program kernels in multiple `qcor`-supported quantum languages. Here we demonstrate this capability using an example that leverages both gate-level and unitary matrix-level programming approaches side-by-side.

Figure 15 demonstrates the generation of the truth table for the Toffoli gate using three distinct languages in a single quantum kernel definition. The example starts off by defining a **controlled-CNOT quantum kernel (ccnot)** that takes a `qreg` and a `vector<int>` describing the initial qubit state configuration (some combination of 0s and 1s). The kernel starts by using the XASM language

```
__qpu__ void ccnot(qreg q,
            std::vector<int> bit_config) {
  // Setup the initial bit configuration
  // This is using XASM language
  for (auto [i, bit] : enumerate(bit_config)) {
    if (bit) {
      X(q[i]);
    }
  }

  // Use the Unitary Matrix DSL for
  // creating the Toffoli matrix to decompose
  decompose {
    UnitaryMatrix ccnot_mat =
                UnitaryMatrix::Identity(8, 8);
    ccnot_mat(6, 6) = 0.0;
    ccnot_mat(7, 7) = 0.0;
    ccnot_mat(6, 7) = 1.0;
    ccnot_mat(7, 6) = 1.0;
  }(q);

  // Switch to OpenQASM and Measure all
  using qcor::openqasm;
  creg c[3];
  measure q -> c;
}

// Helper functions
std::vector<std::vector<int>>
            generate(int size) {...}
void print_result(auto& bit_config,
            auto counts) {...}

int main() {
  // Loop over all configs and print out
  // the Toffoli truth table
  for (auto &bit_config : generate(3)) {
    auto q = qalloc(3);
    ccnot(q, bit_config);
    auto counts = q.counts();
    print_result(bit_config, counts);
  }
}
------- compile and run with ---------
$ qcor -qpu qpp -shots 1024 ccnot.cpp && ./a.out
000 -> 000
001 -> 001
010 -> 010
011 -> 011
100 -> 100
101 -> 101
110 -> 111
111 -> 110
```

Fig. 15. qcor enables the definition of quantum kernels in a variety of quantum languages. Here we show the mixing of XASM, UnitaryMatrix, and OpenQASM languages to create a Toffoli gate with a general initial binary state.

```
---------- grover.qasm ----------
OPENQASM 2.0;
include "qelib1.inc";
qreg qubits[9];
creg c[9];
x qubits[5];
h qubits[0];
h qubits[1];
h qubits[2];
h qubits[3];
h qubits[4];
x qubits[4];
x qubits[0];
x qubits[1];
ccx qubits[0],qubits[1],qubits[6];
ccx qubits[2],qubits[6],qubits[7];
ccx qubits[3],qubits[7],qubits[8];
h qubits[5];
ccx qubits[4],qubits[8],qubits[5];
ccx qubits[3],qubits[7],qubits[8];
ccx qubits[2],qubits[6],qubits[7];
ccx qubits[0],qubits[1],qubits[6];
x qubits[4];
x qubits[0];
x qubits[1];
h qubits[0];
h qubits[1];
h qubits[2];
h qubits[3];
h qubits[4];
x qubits[4];
ccx qubits[0],qubits[1],qubits[6];
... missing for brevity, file has 164 lines
---------- grover.cpp ------------
__qpu__ void grover(qreg q) {
   using qcor::openqasm;
   #include "grover.qasm"
   using qcor::xasm;
   for (int i : range(q.size()))
     Measure(q[i]);
}
int main() {
  auto q = qalloc(9);
  grover::print_kernel(std::cout, q);
  grover(q);
}
```

Fig. 16. Programmers can take advantage of the existing C++ pre-processing capabilities to load existing OpenQASM source files as qcor quantum kernels.

to operate X gates on qubits with a corresponding bit configuration of 1 in the `bit_config` vector. Next, the kernel leverages the unitary matrix decomposition DSL for describing the Toffoli interaction as a matrix. This tells qcor to decompose the corresponding unitary matrix with an internal circuit synthesis algorithm (QFAST [19] by default). Finally, the kernel uses the OpenQASM language to apply measure gates to all qubits in the `qreg`. The `main()` implementation loops over all

```
__qpu__ void ansatz(qreg q, double theta) {
  X(q[0]);
  Ry(q[1], theta);
  CX(q[1], q[0]);
}
int main(int argc, char **argv) {
  // Create the Deuteron Hamiltonian
  auto H = 5.907 - 2.1433 * X(0) * X(1)
           - 2.1433 * Y(0) * Y(1) + .21829 * Z(0)
           - 6.125 * Z(1);
  // Create the ObjectiveFunction
  auto objective =
        createObjectiveFunction(ansatz, H, 1);
  // Create the Optimizer
  auto optimizer = createOptimizer("nlopt");
  // Optimize the ObjectiveFunction
  auto [energy, params] = optimizer->optimize(objective);
  // Print the optimal value.
  printf("<H> = %f\n", energy);
}
--------- compile/run with -----------
$ qcor -qpu qpp qcor_api_example.cpp
$ ./a.out
```

Fig. 17. qcor enables an expressive API for general variational tasks.

bit configurations, each time allocating a three-qubit `qreg`, executing the kernel, and printing the resultant truth table entry.

5.5 Incorporating Pre-Existing OpenQASM Codes

A large number of benchmarks and application-level quantum programs are written as stand-alone OpenQASM files—standard text files containing OpenQASM quantum code. Integration of these pre-existing codes with the `qcor` quantum kernel expression mechanism is straightforward, and we demonstrate it here. The top part of the code snippet in Figure 16 shows the contents of an OpenQASM file called `grover.qasm`. The bottom part demonstrates a `qcor` C++ file that incorporates this OpenQASM code into the usual quantum kernel function definition. Programmers simply note that the kernel language to be used is OpenQASM via the `using qcor::openqasm` statement, and then leverage the existing C++ preprocessor to include the contents of the `grover.qasm` file within the function body. One can then add any other kernel code using any of the available kernel languages (e.g., adding measurements using XASM as seen in the code snippet). Programmers can then invoke the kernel on an appropriately sized `qreg` instance or print the kernel qasm to see that the OpenQASM was appropriately incorporated.

5.6 Variational Algorithms with the QCOR API

Here we demonstrate the utility of the public API and data structures defined by the QCOR specification, and specifically its application to hybrid variational algorithms. The code snippet in Figure 17 provides an example of computing the ground state energy of the two qubit deuteron Hamiltonian using the `qcor` `Operator`, `ObjectiveFunction`, `Optimizer`, and `taskInitiate()`. The example starts with a quantum kernel definition describing the variational quantum circuit,

Table 3. Circuit Optimization Results for Staq Benchmarks [2] Using (1) Individual qcor Passes and (2) qcor's Level-1 Optimization Sequence

Pass Name	Gate Count Reduction		
	Min	**Max**	**Avg.**
rotation-folding	0.6%	34.4%	18.2%
single-qubit-gate-merging	0.0%	41.3%	6.2%
circuit-optimizer	0.0%	12.9%	5.8%
voqc	8.2%	38.6%	22.6%
Level 1	8.8%	42.0%	23.2%

in this case a simple kernel leveraging the XASM language using a single `double` parameter. `main()` begins with the definition of the `Operator` describing the Hamiltonian for this system, which is extremely natural when leveraging the qcor `X`, `Y`, `Z` function calls. Next, the programmer creates an `ObjectiveFunction`, giving it the quantum kernel, `Operator`, and the number of variational parameters in the problem. Note that when one does not provide the name of the `ObjectiveFunction` subtype, `vqe` is assumed. Next, the `Optimizer` is created, specifically an implementation backed by the NLOpt library, defaulting to the COBYLA derivative-free algorithm. The optimization task is launched on a separate execution thread via the `taskInitiate()` call, returning a `Handle` that is kept and used later to synchronize the host and execution threads. Finally, after synchronization, the optimal value can be retrieved from the `ResultsBuffer`.

5.7 Overall Compiler Performance

Here, we demonstrate the overall effectiveness of `qcor` as an quantum compiler. To start, we demonstrate the performance of our JIT circuit optimization procedure (described in Section 4.2.2) by running the `qcor` compiler with flag `-opt 1` on a collection of common benchmark circuit files. We compare our optimization passes to existing approaches from the Staq compiler executable. Since we have also wrapped the Staq rotation-folding optimization as a pass that `qcor` can use (an `xacc::IRTransformation`), we are able to directly compare the performance between passes. More importantly, as mentioned in Section 4.2.2, we have bundled those passes into a custom level, which instructs the `PassManager` to execute passes in series. In particular, the overall level-1 optimization performance in Table 3 is the result of the `rotation-folding`, `single-qubit-gate-merging`, `circuit-optimizer`, and `voqc` (see Table 1 for descriptions) sequence.

Not only does `qcor` offer an effective quantum circuit optimization solution, it also incorporates state-of-the-art qubit placement techniques, as described in Section 4.2.3. For near-term quantum devices with limited connectivity, efficient qubit placement is of great importance to the fidelity and success rate of circuit execution. In Table 4, we show a comparison in terms of gate count between some of the placement options that are available in `qcor`. In these test cases, we have processed the input circuits through `qcor` circuit optimization passes before performing hardware placement. The target device is the 65 qubit IBM `ibmq_manhattan` backend, which has a heavy-hexagon lattice topology. We measured the improvement percentage of built-in placement strategies by comparing the number of two-qubit gates present in the benchmark circuit after placement against that of the `qcor` default Staq `swap-shortest-path` [2] strategy. As can be seen in Table 4, `qcor's` placement improves the number of added two-qubit gates from 17% up to 48% compared to that of Staq.

Table 4. The Number of Two-qubit Gates After Placement Using Various Placement Strategies

Name	n	N	N_{ssp}	N_{sabre}	N_{QX}	Imp. [%]
barenco10	19	190	883	**526**	724	40.4
barenco5	9	70	211	**175**	250	17.1
grover5	9	248	1158	**686**	1100	40.8
hwb6	7	110	445	**305**	449	31.5
hwb8	12	6741	39988	**22716**	31263	43.2
mod5_4	5	28	106	**55**	70	48.1
qft4	5	46	154	112	**109**	29.2
tof3	5	16	55	**31**	40	43.6
tof5	9	36	171	**108**	144	36.8
vbe3	10	58	220	**127**	202	42.3

For each benchmark case, the best result among Sabre (N_{sabre}), `swap-shortest-path` (N_{ssp}), and QX-mapping (N_{QX}) is shown in boldface. n is the number of qubits and N is the number of two-qubit gates before placement. The improvement percentage is relative to that of `swap-shortest-path`.

```
__qpu__ void trotter_evolve(qreg q,
           std::vector<Operator> &exp_args,
           int n_steps) {
  for (int i = 0; i < n_steps; ++i) {
    for (auto &exp_arg : exp_args) {
      exp_i_theta(q, 1.0, exp_arg);
    }
  }
}
```

```
from qiskit import QuantumCircuit
from qiskit.aqua.operators import EvolvedOp,
                    PauliTrotterEvolution
def trotter_evolve(q, exp_args, n_steps):
  qc = QuantumCircuit(q)
  for i in range(n_steps):
    for sub_op in exp_args:
      qc += PauliTrotterEvolution()
            .convert(EvolvedOp(sub_op))
            .to_circuit()
  return qc
```

Fig. 18. Constructing Trotter-decomposed Hamiltonian evolution circuit with qcor (left) and Qiskit (right). The kernel function arguments are the qubit register, the list of terms in the Hamiltonian sum (Equation (1)) and the number of Trotter steps.

5.8 Circuit Composition Performance

One key differentiation of the qcor compilation infrastructure is its native C++ implementation, which is supposedly fast and resource efficient. The majority of existing quantum programming frameworks, on the other hand, are based on script languages, prominently Python. Despite its flexibility, implementations in Python can carry substantial runtime overhead for large-scale circuits.

For comparison, we time the quantum circuit composition time of qcor and Qiskit for a Trotter-decomposed Hamiltonian evolution algorithm, as shown in Figure 18. The Hamiltonian that we consider is a generic Heisenberg model of the form,

$$H = -J_z \sum_{i}^{N} Z_i Z_{i+1} - h \sum_{i}^{N} X_i, \tag{1}$$

where N is the number of qubits. To approximate the evolution (e^{-iHt}) of this Hamiltonian, we discretize the time horizon into steps and repeat those time-stepping circuits, i.e., the QCOR's

Table 5. Runtime Data for Trotter-decomposed Circuit Generation for a Various Number of Qubits (N)

N	Gate Count	qcor (s)	Qiskit (s)
5	2700	0.080705	4.776733875274658
10	5700	0.174581	19.232121229171753
20	11700	0.372422	93.73427820205688
50	29700	1.09762	916.3527870178223
100	59700	2.42994	—

The number of steps is fixed at 100. For qcor, the compilation and linking time to generate the executable binary is not included. For Qiskit, data for $N = 100$ is not available due to the excessive running time.

`exp_i_theta` function and the Qiskit's equivalent in Figure 18. The accuracy of the algorithm is inversely proportional to the step size. Therefore, we often need a large number of steps.

Table 5 shows the circuit composition time for up to $N = 100$ qubits. For `qcor`, the reported time is the wall-clock time from kernel invocation to the generation of a flattened IR tree (gate-by-gate). This task is equivalent to the circuit builder pattern employed by Qiskit. As we can see, the time it takes for a `qcor` kernel to be composed at runtime is orders of magnitude faster than an equivalent Pythonic implementation. When taking into account the initial `qcor` compilation time of (4 to 5 s), users can still expect a substantial gain in a wide variety of use cases. For instance, constructing large-scale quantum circuits beyond the NISQ-regime for resource estimation can benefit from this efficient circuit realization.

5.9 QCOR-enabled Library Development

Finally, we turn our attention to future design goals with regards to the `qcor` compiler and runtime library. We wish to demonstrate how one might leverage the infrastructure and compiler defined in this work for high-level quantum algorithmic library development. It is our intention that the work described here will form the basis for the creation of scientific libraries that hide or abstract away the low-level machinery required for quantum-classical algorithm implementation. Specifically, here we introduce a prototype library called `qcor_hybrid` that provides high-level data structures for common hybrid, variational quantum-classical algorithms. We demonstrate how this library enables the integration of the VQE [12] and **Adaptive Derivative Assembled Problem Tailored (ADAPT)** [22] algorithms within existing C++ applications.

5.9.1 VQE. `qcor_hybrid` provides a `VQE` data structure that hides the complexity of the `qcor` data model and asynchronous execution API. Programmers simply instantiate this data structure, invoke its `execute()` method, and retrieve the optimal energy and associated parameters. Moreover, one can use the data structure without the full optimization loop, and simply invoke an `operator()(std::vector<double>)` method to evaluate the expectation value of the given `Operator` at the provided parameters.

The code snippet in Figure 19 demonstrates the use of `qcor_hybrid` for sweeping the variational parameter for a prototypical state preparation circuit and computing the associated expectation value of the given `Operator`. Programmers begin by including the library header file, followed by the definition of a parameterized quantum kernel. Programmers instantiate an `Operator` representation of the Hamiltonian in the same way as previous examples. The `VQE` data structure is instantiated, taking a reference to the quantum kernel and Hamiltonian. Extra options can be provided to influence the execution, and here we demonstrate requesting that each point be computed

```
#include "qcor_hybrid.hpp"
__qpu__ void ansatz(qreg q,
            std::vector<double> p) {
  X(q[0]);
  auto exp_arg = X(0) * Y(1) - Y(0) * X(1);
  exp_i_theta(q, p[0], exp_arg);
}
int main(int argc, char **argv) {
  // Define the Hamiltonian using the QCOR API
  auto H = 5.907 - 2.1433*(X(0)*X(1)-Y(0)*Y(1))
           + .21829 * Z(0) - 6.125 * Z(1);
  // Create a VQE instance, give it the kernel,
  // the Hamiltonian, and an extra option to run
  // each point 10 times to gather statistics
  VQE vqe(ansatz, H,
            {{"vqe-gather-statistics",10}});

  // Loop over 20 points in [-1., 1.]
  // and compute the energy at that point
  for (auto [iter, x] :
            enumerate(linspace(-1., 1., 20))) {
    std::cout << iter << ", " << x << ", "
              << vqe({x}) << "\n";
  }
  // Dump the data to file for processing
  vqe.persist_data("param_sweep_data.json");
}
--------- compile/run with -----------
// Exact execution
$ qcor -qpu qpp vqe.cpp && ./a.out
// Noisy execution
$ qcor -qpu aer[noise-model:custom_noise.json]
    vqe.cpp && ./a.out
// Error mitigated execution
// (apply readout error mitigation)
$ qcor -qpu aer[noise-model:custom_noise.json]
    vqe.cpp -em ro-error && ./a.out
```

Fig. 19. qcor quantum kernels are just functions, and can therefore be passed around via function pointers. This enables one to define quantum-classical libraries for pertinent algorithmic tasks.

```
// QCOR hybrid algorithms library
#include "qcor_hybrid.hpp"

// Define the state preparation kernel
__qpu__ void initial_state(qreg q) {
  X(q[0]);
  X(q[1]);
  X(q[4]);
  X(q[5]);
}

int main() {

  // Define the Hamiltonian using the QCOR API
  auto H = 0.111499 * Z(0) * Z(6) + ...;

  // optimizer
  auto optimizer = createOptimizer(
      "nlopt", {{"nlopt-optimizer", "l-bfgs"}});

  // Create ADAPT-VQE instance
  ADAPT adapt(initial_state, H, optimizer,
              {{"sub-algorithm", "vqe"},
              {"pool", "singlet-adapted-uccsd"},
              {"n-electrons", 4},
              {"gradient_strategy", "central"}});

  // Execute and print
  auto energy = adapt.execute();
  std::cout << energy << "\n";
}
--------- compile/run with -----------
$ qcor -qpu tnqvm adapt-vqe.cpp
$ ./a.out
```

Fig. 20. One could leverage the quantum kernel programming model as well for algorithmic libraries that require steadily growing a parameterized circuit from an initial one.

multiple times to gather appropriate statistics. Computation of the expected value is affected via the `operator()(std::vector<double>)` method on the VQE class. At the command line, one can specify which backend this code should be compiled for. We demonstrate the compilation and execution of this code for a noise-free, exact backend, a noisy simulation backend, and a noisy simulation backend with readout-error mitigation applied. The results for these three executions are shown in Figure 21.

5.9.2 ADAPT. The `qcor_hybrid` library provides a high-level data structure implementing the popular ADAPT algorithm, which builds an adaptive circuit ansatz on the fly that varies according to the complexity of the problem at hand. The ADAPT algorithm provides an iterative loop that checks for the most relevant operator (Pauli or fermionic), updates the ansatz, and proceeds by calling either the VQE or QAOA routine, depending on the problem of interest. Detailed accounts on these two instances can be found elsewhere [22, 47]. The code snippet in Figure 20 illustrates how to instantiate and run an ADAPT-VQE simulation of a chain of four hydrogen atoms taking advantage of the `qcor_hybrid` library. This is followed by the definition of the quantum kernel representing the initial state. The `main()` function body contains the definitions for the problem Hamiltonian, shortened here for the sake of clarity, and the desired

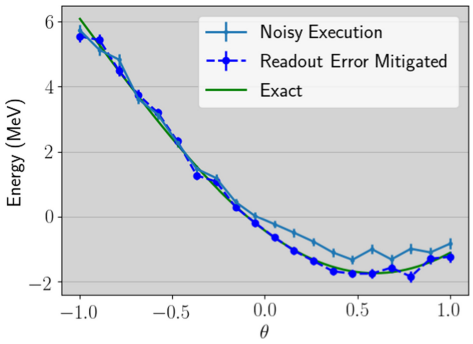

Fig. 21. Results of running the code in Figure 19 with differing qcor command line arguments (shown at bottom of Figure 19).

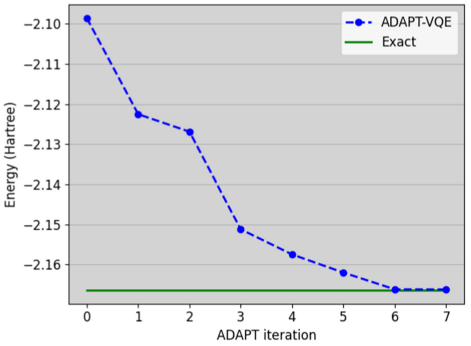

Fig. 22. Results of running the code in Figure 20 with the TNQVM as the noiseless numerical simulator.

optimizer, which are followed by problem- and sub-algorithm-specific parameters. In this case, we need to pass to the ADAPT instance the variational algorithm it will employ to optimize the circuit, the number of electrons, and the set of fermionic operators associated with the variational parameters. Because the chosen optimization strategy here supports gradients (L-BFGS), to aid in updating the variational parameters, we also provide the algorithm with a strategy for its computation (numerical central finite differences). The first three arguments in the constructor of the `ADAPT` class are the necessary components shared by both VQE and QAOA, namely initial state, `Operator`, and `Optimizer`, while the last argument is an options map that is responsible for passing the problem- and sub-algorithm-specific parameters. A simulation exemplifying the code snippet in Figure 20 is presented in Figure 22.

6 CONCLUSION

We have presented `qcor`, a language extension to C++ and associated compiler executable that enables heterogeneous quantum-classical computing in a single-source C++ context. Our approach leverages a novel domain specific language preprocessing plugin from Clang (the `SyntaxHandler`) and enables general quantum DSL integration as part of quantum kernel expression. This has enabled a Clang-based compiler for quantum-classical source files, however we envision implementation avenues that extend this capability to other existing classical compiler drivers (e.g., GCC, Intel, NVCC, etc.). Moreover, our compiler builds upon the XACC quantum programming framework, thereby enabling a hardware-agnostic retargetable compiler, in addition to an integration mechanism for common quantum compiling, optimization, and qubit placement tasks. We believe that `qcor` will ultimately promote tight integration of future quantum co-processors with existing high-performance computing application software stacks. Finally, we note that our work is completely open source and available at https://github.com/ornl-qci/qcor.

APPENDICES

A CLANG SYNTAX HANDLER

The Clang Syntax Handler has been introduced in Reference [31] and provides a novel plugin interface to the Clang platform enabling the embedding of DSL within standard C++ functions. The embedded DSLs need not be syntactically similar to C++; however, it should be balanced with respect the right and left brace delimiters. Functions with embedded DSL code should be annotated with the name of a specific Syntax Handler implementation that can

```
[[clang::syntax(sh_name)]] void foo() {
  ... Embedded DSL here
  ... SyntaxHandler with name sh_name will
  ... translate this to standard C++ code
}
-------------------------------------------------
using namespace clang;
using namespace llvm;
class MySyntaxHandler : public SyntaxHandler {
public:
  MySyntaxHandler() : SyntaxHandler("sh_name") {}
  void GetReplacement(Preprocessor& PP,
                Declarator& D,
                CachedTokens& Toks,
                raw_string_ostream& OS) override
  {
     ... analyze Toks, write new code to OS
  }
  void AddToPredefines(raw_string_ostream& OS) {
    ... add any #includes here
  }
};
```

Fig. 23. Programmers annotate a function indicating the `SyntaxHandler` to be used in parsing and transforming the function body Tokens. `SyntaxHandler` subtypes implement `GetReplacement` to analyze function body tokens and output new code to the provided output stream reference.

handle, analyze, and understand the DSL syntax and semantics. The proper annotation is `[[clang::syntax(SYNTAX_HANDLER_NAME)]]`, where `SYNTAX_HANDLER_NAME` corresponds to the provided `llvm::StringRef` name attribute for a given Syntax Handler subtype. New Syntax Handler subtypes provide implementations of the `GetReplacement()` and `AddToPredefines()` methods. The first takes the seen function body tokens (as a `CachedTokens` reference) as well as an output stream reference. The overall goal for implementations is to take in these tokens, parse and analyze them, and write new valid C++ code to the output stream that will serve as a replacement of the original DSL code. Implementations also take reference to the `Preprocessor` and `Declarator` instances and can use this for token-handling tasks and reconstruction of the original function prototype, respectively. The `AddToPredefines` method takes an output stream and enables implementations to add any pertinent `#include` headers that are required for compilation of the rewritten code. Figure 23 demonstrates the overall design and subtype implementation pattern for the Syntax Handler infrastructure. Functions are annotated indicating the handler name and implemented with the specific DSL. The corresponding Syntax Handler subtype is implemented, passing its name to the super constructor, and implementing the two key methods mapping tokens to valid C++ code and providing any necessary headers for inclusion.

B DETAILS OF THE QCOR SPECIFICATION IMPLEMENTATION

The class architecture for the `ObjectiveFunction` is shown in Figure 26, which we decompose into user-level `ObjectiveFunction` and internal `ObjectiveFunctionImpl` classes. The latter class is a variadic template on the quantum kernel argument types that keeps reference to an internal *helper* `ObjectiveFunction` and implements the `operator()(std::vector<double>)` method to map the incoming parameter vector to appropriate quantum kernel function arguments. It then invokes

```
// assume a kernel like this
__qpu__ void foo(qreg, std::vector<double> gamma,
                       std::vector<double> beta) {
 .... quantum circuit using gamma, beta params
}
...
const int mid_point = 4;
auto args_translator =
    ArgsTranslator<std::vector<double>,
                          std::vector<double>>(
    [&](const std::vector<double> x) {
    // split x into gamma and beta sets
    std::vector<double> gamma(x.begin(),
                    x.begin() + mid_point),
        beta(x.begin() + mid_point, x.end());
    return std::make_tuple(q, gamma, beta);
});
```

Fig. 24. Custom `ArgsTranslator` instances can be provided to map a vector of parameters to specific kernel function argument structures.

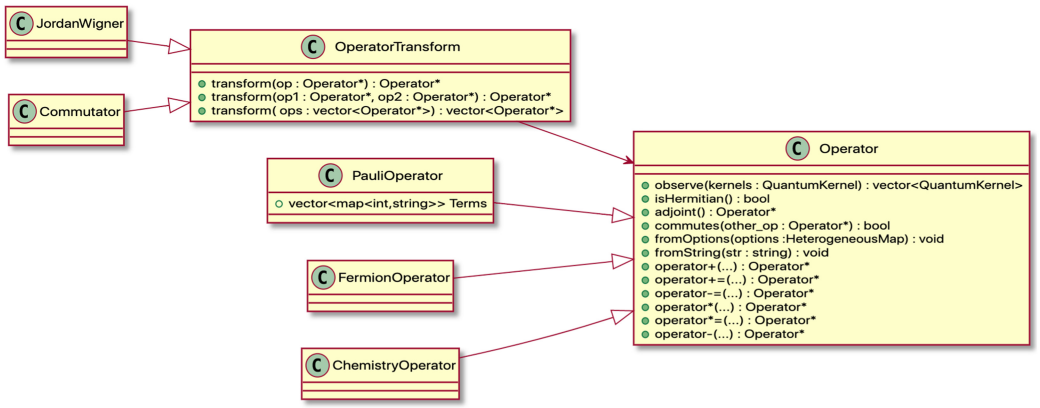

Fig. 25. The class diagram for the `Operator` class. `Operator` exposes an API for algebraic operations, which subtypes implement.

the protected `operator()(qreg, std::vector<double>&)` method of its `ObjectiveFunction` helper reference, passing the internal `qreg` instance and the reference to a vector for gradients, and returns the result of that call.

Conceptually, programmers request an `ObjectiveFunction` (see Figure 6) of a given name (`vqe` for example) and an `ObjectiveFunctionImpl` is constructed internally, templated on the quantum kernel arguments, and given reference to the corresponding `ObjectiveFunction` instance as its `obj_func_helper`. The `ObjectiveFunctionImpl` is solely responsible for evaluation of the quantum kernel, with pre- and post-processing left as a job for the internal helper `ObjectiveFunction`. The `ObjectiveFunctionImpl` instance is returned to programmers as an `ObjectiveFunction` pointer, removing the need for users to know any information about the template types or internal implementation.

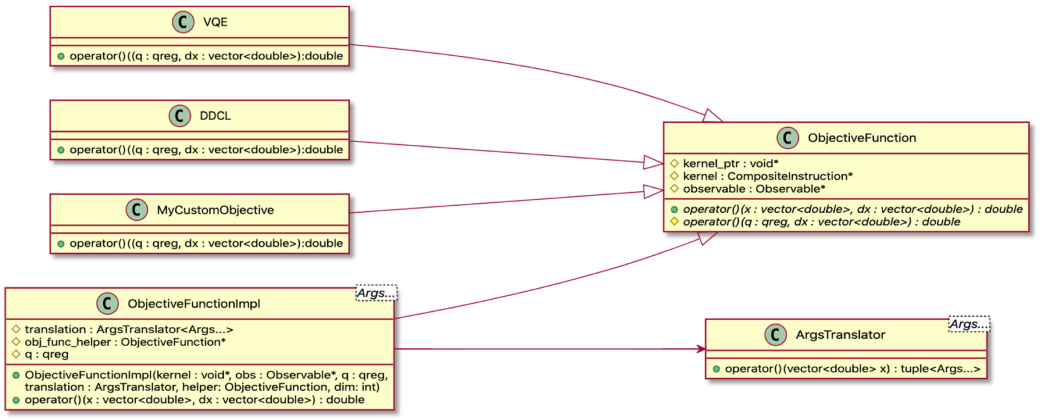

Fig. 26. The class diagram for the `ObjectiveFunction` template class. subtypes provide custom `ObjectiveFunction` evaluation workflows.

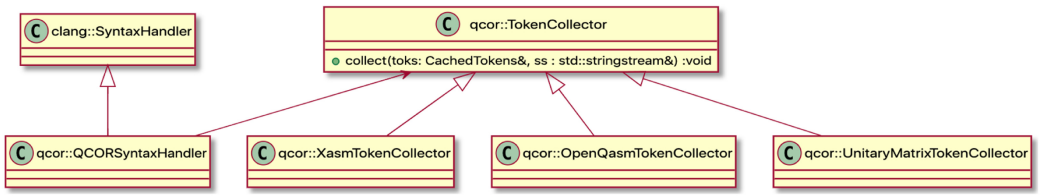

Fig. 27. The class diagram for the `QCORSyntaxHandler` class.

It should be noted that in advanced use cases, the quantum kernel argument signature may in general be much more complex than the argument signature of the `ObjectiveFunction` callable. In this case, we need a mechanism for mapping `std::vector<double> x` parameters to the argument structure of the provided quantum kernel. To achieve this, `qcor` defines the `ArgsTranslator<Args...>` variadic class. This data structure is templated on the argument types of the quantum kernel, and takes at construction a lambda or function of signature `std::tuple<Args...>(const std::vector<double>)`. The goal of this lambda is to map the incoming parameter vector to the kernel function arguments, taking advantage of any lambda capture variables required. A concrete example of this would be in the definition of a quantum kernel that takes two separate parameter vectors that, if concatenated together, would form the single parameter vector required for `ObjectiveFunction`. In this case, one would define a `ArgsTranslator` like in the code snippet provided in Figure 24.

C JUST-IN-TIME QUANTUM KERNEL COMPILATION

Another architectural point of note for the compiler is the addition of data structures and utilities to perform just-in-time compilation of quantum kernels. We foresee use cases whereby developers may wish to build up quantum circuits at runtime based on pertinent runtime information. This is difficult with quantum kernel function declarations, as these are defined at compile time. We have therefore introduced a new data type, `QJIT`, which provides quantum kernel JIT. The code snippet in Figure 28 demonstrates how one might use this utility. `QJIT` exposes a `jit_compile()` method that takes as input the quantum kernel as a source string. This method will then

```
#include "qcor_jit.hpp"
int main() {

  // QJIT is the entry point to QCOR quantum kernel
  // just in time compilation
  QJIT qjit;

  // Define a quantum kernel string dynamically
  const auto kernel_src = R"#(
    __qpu__ void bell(qreg q) {
        using qcor::openqasm;
        h q[0];
        cx q[0], q[1];
        creg c[2];
        measure q -> c;
    })#";

  // Use qjit to compile this at runtime
  qjit.jit_compile(kernel_src);

  // Now, one can get the compiled kernel as a
  // function to execute, must provide the kernel
  // argument types as template parameters
  auto bell = qjit.get_kernel<qreg>("bell");

  // Allocate a qreg and run the kernel function
  auto q = qalloc(2);
  bell(q);
  q.print();

  // Or, one can call the QJIT invoke method
  // with the name of the kernel function and
  // the necessary function arguments.
  auto r = qalloc(2);
  qjit.invoke("bell", r);
  r.print();
}
```

Fig. 28. Code snippet demonstrating qcor quantum kernel just in time compilation.

programmatically run the `QCORSyntaxHandler` on that source string to produce the source string containing the `QuantumKernel` subtype definition plus additional utility functions (as in Figure 9). This new source string is then compiled to an LLVM IR `Module` instance using the Clang `CodeGenAction` programmatically. The resultant `Module` is then passed to the LLVM JIT utility data structures (`ExecutionSession`, `IRCompileLayer`) for just-in-time compilation.

Finally, a pointer to the representative function for the quantum kernel is stored and returned via the `QJIT::get_kernel<Args...>()` call, or leveraged in the `QJIT::invoke()` call. In this way, programmers can compile source string dynamically at runtime, and get a function pointer reference to the JIT compiled function for future execution. This workflow also incorporates `Module` caching so that the same quantum kernel source code is not re-compiled every time it is encountered (or the executable running this workflow is run).

REFERENCES

[1] Aksel Alpay and Vincent Heuveline. 2020. SYCL beyond OpenCL: The architecture, current state and future direction of HipSYCL. In *Proceedings of the International Workshop on OpenCL (IWOCL'20)*. Association for Computing Machinery, New York, NY. https://doi.org/10.1145/3388333.3388658

[2] Matthew Amy and Vlad Gheorghiu. 2019. staq—A full-stack quantum processing toolkit. arXiv:1912.06070. Retrieved from https://arxiv.org/abs/1912.06070.

[3] Matthias Anlauff. 2000. XASM - An extensible, component-based ASM language. In *Proceedings of the International Workshop on Abstract State Machines, Theory and Applications*, 69–90.

[4] Sergey Bravyi, Martin Suchara, and Alexander Vargo. 2014. Efficient algorithms for maximum likelihood decoding in the surface code. *Phys. Rev. A* 90, 3 (Sep. 2014). https://doi.org/10.1103/PhysRevA.90.032326

[5] Cirq Contributors. 2020. Cirq. Retrieved from https://github.com/quantumlib/Cirq.

[6] CppMicroServices. 2020. CppMicroServices. Retrieved August 28, 2020 from https://github.com/CppMicroServices/CppMicroServices.

[7] 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. 2018. Cloud quantum computing of an atomic nucleus. *Phys. Rev. Lett.* 120, 21 (May 2018), 210501. https://doi.org/10.1103/PhysRevLett.120.210501

[8] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *J. Parallel Distrib. Comput.* 74, 12 (2014), 3202–3216. https://doi.org/10.1016/j.jpdc.2014.07.003

[9] Austin G. Fowler et al. 2012. Surface codes: Towards practical large-scale quantum computation. *Phys. Rev. A* 86, 3 (2012). https://doi.org/10.1103/PhysRevA.86.032324

[10] Ali Javadi-Abhari et al. 2014. ScaffCC: A framework for compilation and analysis of quantum computing programs. In *Proceedings of the 11th ACM Conference on Computing Frontiers (CF'14)*. https://doi.org/10.1145/2597917.2597939

[11] Abhinav Kandala et al. 2017. Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets. *Nature* 549 (Sep. 2017), 242.

[12] Alberto Peruzzo et al. 2014. A variational eigenvalue solver on a photonic quantum processor. *Nat. Commun.* 5, 4213 (Jul. 2014), 1–7. https://doi.org/10.1038/ncomms5213

[13] Andrew W. Cross et al. 2017. Open Quantum Assembly Language. arXiv:1707.03429. Retrieved from https://arxiv.org/abs/1707.03429.

[14] Benjamin C. A. Morrison et al. 2020. JaqalPaq. Retrieved from https://gitlab.com/jaqal/jaqalpaq.

[15] Benjamin C. A. Morrison et al. 2020. Just another quantum assembly language (Jaqal). arXiv:quant-ph/2008.08042. Retrieved from https://arxiv.org/abs/2008.08042.

[16] Ciyou Zhu et al. 1997. Algorithm 778: L-BFGS-B: Fortran subroutines for large-scale bound-constrained optimization. *ACM Trans. Math. Softw.* 23, 4 (Dec. 1997), 550–560. https://doi.org/10.1145/279232.279236

[17] D. A. Beckingsale et al. 2019. RAJA: Portable performance for large-scale scientific applications. In *Proceedings of the 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC'19)*. 71–81. https://doi.org/10.1109/P3HPC49587.2019.00012

[18] Edward Farhi et al. 2000. Quantum computation by adiabatic evolution. arXiv:quant-ph/0001106. Retrieved from https://arxiv.org/abs/quant-ph/0001106.

[19] Ed Younis et al. 2020. QFAST: Quantum synthesis using a hierarchical continuous circuit space. arXiv:2003.04462. Retrieved from https://arxiv.org/abs/2003.04462.

[20] Gadi Aleksandrowicz et al. 2019. Qiskit: An Open-source Framework for Quantum Computing. https://doi.org/10.5281/zenodo.2562110

[21] Gaël Guennebaud et al. 2010. Eigen v3. Retreived from http://eigen.tuxfamily.org.

[22] Harper R. Grimsley et al. 2019. An adaptive variational algorithm for exact molecular simulations on a quantum computer. *Nat. Commun.* 10, 1 (08 Jul. 2019), 3007. https://doi.org/10.1038/s41467-019-10988-2

[23] John Nickolls et al. 2008. Scalable parallel programming with CUDA. *Queue* 6, 2 (Mar. 2008), 40–53. https://doi.org/10.1145/1365490.1365500

[24] Kesha Hietala et al. 2021. A verified optimizer for quantum circuits. *Proc. ACM Program. Lang.* 5, Article 37 (Jan. 2021), 29 pages. https://doi.org/10.1145/3434318

[25] Krysta Svore et al. 2018. Q#: Enabling scalable quantum computing and development with a high-level DSL. In *Proceedings of the Real World Domain Specific Languages Workshop 2018 (RWDSL'18)*. Association for Computing Machinery, New York, NY, Article 7, 10 pages. https://doi.org/10.1145/3183895.3183901

[26] Peter Canning et al. 1989. F-bounded polymorphism for object-oriented programming. In *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture*, 273–280. https://doi.org/10.1145/99370.99392

[27] Peter J. Karalekas et al. 2020. A quantum-classical cloud platform optimized for variational hybrid algorithms. *Quant. Sci. Technol.* 5, 2 (Apr. 2020), 024003. https://doi.org/10.1088/2058-9565/ab7559

[28] Prakash Murali et al. 2019. Noise-adaptive compiler mappings for noisy intermediate-scale quantum computers. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*. 1015–1029. https://doi.org/10.1145/3297858.3304075

[29] Ryan LaRose et al. 2020. Mitiq: A software package for error mitigation on noisy quantum computers. arXiv:2009.04417. Retrieved from https://arxiv.org/abs/2009.04417.

[30] Tudor Giurgica-Tiron et al. 2020. Digital zero noise extrapolation for quantum error mitigation. In *Proceedings of the 2020 IEEE International Conference on Quantum Computing and Engineering (QCE'20)*. 306–316. https://doi.org/10.1109/QCE49297.2020.00045

[31] H. Finkel, A. McCaskey, T. Popoola, D. Lyakh, and J. Doerfert. 2020. Really embedding domain-specific languages into C++. In *Proceedings of the 2020 IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar'20)*. 65–73. https://doi.org/10.1109/LLVMHPCHiPar51896.2020.00012

[32] Austin G. Fowler. 2015. Minimum weight perfect matching of fault-tolerant topological quantum error correction in average O (1) parallel time. *Quant. Inf. Comput.* 15, 1–2 (2015), 145–158. https://dl.acm.org/doi/10.5555/2685188.2685197

[33] Kathleen E. Hamilton, Eugene F. Dumitrescu, and Raphael C. Pooser. 2019. Generative model benchmarks for superconducting qubits. *Phys. Rev. A* 99, 6 (Jun. 2019), 062323. https://doi.org/10.1103/PhysRevA.99.062323

[34] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. arXiv:1412.6980. Retrieved from https://arxiv.org/abs/1412.6980.

[35] N. Klco, E. F. Dumitrescu, A. J. McCaskey, T. D. Morris, R. C. Pooser, M. Sanz, E. Solano, P. Lougovski, and M. J. Savage. 2018. Quantum-classical computation of Schwinger model dynamics using quantum computers. *Phys. Rev. A* 98, 3 (Sep. 2018), 032331. https://doi.org/10.1103/PhysRevA.98.032331

[36] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. IEEE Computer Society, 75.

[37] Gushu Li, Yufei Ding, and Yuan Xie. 2019. Tackling the qubit mapping problem for NISQ-era quantum devices. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*. 1001–1014.

[38] Dave Marples and Peter Kriens. 2002. The open services gateway initiative: An introductory overview. *IEEE Commun. Mag.* 39, 12 (Jan. 2002), 110–114. https://doi.org/10.1109/35.968820

[39] Alexander J. McCaskey, Dmitry I. Lyakh, Eugene F. Dumitrescu, Sarah S. Powers, and Travis S. Humble. 2020. XACC: A system-level software infrastructure for heterogeneous quantum–classical computing. *Quant. Sci. Technol.* 5, 2 (Feb. 2020), 024002. https://doi.org/10.1088/2058-9565/ab6bf6

[40] Alexander J. McCaskey, Zachary P. Parks, Jacek Jakowski, Shirley V. Moore, Titus D. Morris, Travis S. Humble, and Raphael C. Pooser. 2019. Quantum chemistry as a benchmark for near-term quantum computers. *npj Quant. Inf.* 5, 1 (2019), 99. https://doi.org/10.1038/s41534-019-0209-0

[41] Tiffany M. Mintz, Alexander J. Mccaskey, Eugene F. Dumitrescu, Shirley V. Moore, Sarah Powers, and Pavel Lougovski. 2019. QCOR: A language extension specification for the heterogeneous quantum-classical model of computation. arXiv:1909.02457. Retrieved from https://arxiv.org/abs/1909.02457.

[42] mitiq. 2020. mitiq. Retrieved September 2, 2020 from https://github.com/unitaryfund/mitiq.

[43] M. J. D. Powell. 1998. Direct search algorithms for optimization calculations. *Acta Numer.* 7 (Jan. 1998), 287–336. https://doi.org/10.1017/S0962492900002841

[44] Robert S. Smith, Michael J. Curtis, and William J. Zeng. 2017. A practical quantum instruction set architecture. arXiv:1608.03355. Retrieved from https://arxiv.org/abs/1608.03355.

[45] Damian S. Steiger, Thomas Häner, and Matthias Troyer. 2018. ProjectQ: An open source software framework for quantum computing. *Quantum* 2 (Jan. 2018), 49. https://doi.org/10.22331/q-2018-01-31-49

[46] Damian S. Steiger, Thomas Häner, and Matthias Troyer. 2019. Advantages of a modular high-level quantum programming framework. *Microprocess. Microsyst.* 66 (2019), 81–89. https://doi.org/10.1016/j.micpro.2019.02.003

[47] Linghua Zhu, Ho Lun. Tang, George S. Barron, Nicholas J. Mayhall, Edwin Barnes, and Sophia E. Economou. 2020. An adaptive quantum approximate optimization algorithm for solving combinatorial problems on a quantum computer. arXiv:2005.10258. Retrieved from https://arxiv.org/abs/2005.10258.

[48] Alwin Zulehner, Alexandru Paler, and Robert Wille. 2018. An efficient methodology for mapping quantum circuits to the IBM QX architectures. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 38, 7 (2018), 1226–1236.

Received October 2020; revised March 2021; accepted April 2021