A MLIR Dialect for Quantum Assembly Languages

Alexander McCaskey*† and Thien Nguyen*†

*Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN, 37831, USA
†Quantum Science Center, Oak Ridge National Laboratory, Oak Ridge, TN, 37831, USA

Abstract—We demonstrate the utility of the Multi-Level Intermediate Representation (MLIR) for quantum computing. Specifically, we extend MLIR with a new quantum dialect that enables the expression and compilation of common quantum assembly languages. The true utility of this dialect is in its ability to be lowered to the LLVM intermediate representation (IR) in a manner that is adherent to the quantum intermediate representation (QIR) specification recently proposed by Microsoft. We leverage a `qcor`-enabled implementation of the QIR quantum runtime API to enable a retargetable (quantum hardware agnostic) compiler workflow mapping quantum languages to hybrid quantum-classical binary executables and object code. We evaluate and demonstrate this novel compiler workflow with quantum programs written in OpenQASM 2.0. We provide concrete examples detailing the generation of MLIR from OpenQASM source files, the lowering process from MLIR to LLVM IR, and ultimately the generation of executable binaries targeting available quantum processors.

Index Terms—quantum computing, quantum programming, quantum simulation, programming languages

I. Introduction

The availability of noisy quantum processing units (QPUs) from a variety of hardware vendors has raised new research and development questions into application use cases, programming model [9], [13], [19], [26]. It is anticipated that QPUs may serve as co-processors or accelerators within existing classical scientific computing workflows in a post-exascale computing era [17]. The heterogeneous quantum-classical characteristic of this model of computation necessitates the development of quantum languages, compilers, and associated frameworks and tooling that tightly

integrate with existing classical infrastructures. For example, the LLVM compiler infrastructure [15] has proven critical to existing classical accelerator-based computing workflows, and we anticipate that building upon this infrastructure for quantum co-processing will precipitate required tight integration between classical and quantum compute resources. A number of efforts are leveraging LLVM and Clang to provide GPU-based programming models, compilers, and tools [7], [23], and quantum computing may be well-suited to build upon these common ideas and concepts.

Recently, a novel infrastructure for domain specific compilation has been put forward under the LLVM project umbrella. The Multi-Level Intermediate Representation (MLIR) represents a unique approach to the expression of compiler intermediate representations that promotes a variety of abstraction levels (close to the source language all the way down to machine level IR), enables progressive lowering from high-level IR abstractions to lower ones, and promotes easy and efficient extensibility and reusability of IR components [16]. It has primarily been leveraged in classical heterogeneous computing workflows to enable common machine learning application compilation to available classical co-processors (GPUs, TPUs, FPGAs, etc.) [11], [14]. As a domain-specific compiler infrastructure, the MLIR is be well suited for the development of compiler infrastructures mapping quantum assembly languages to executables or object code targeting available quantum co-processors. The development of a novel MLIR extension for quantum computing would enable an integration platform for vendor-specific languages and promote compilation in a quantum-backend-retargetable manner.

In an effort to move quantum co-processing compiler tools towards integration with LLVM, researchers at Microsoft have recently introduced a novel specification called the quantum intermediate representation (QIR) [2]. This work defines a number of low-level LLVM callable IR instructions and opaque data types that are necessary for typical quantum computing workflows and tasks. The overall goal is to provide
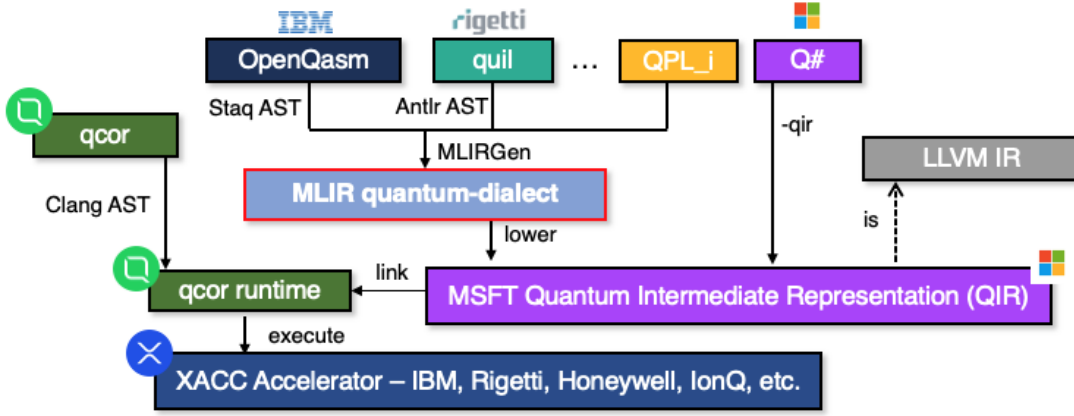
Figure 1: Our goal is to enable progressive lowering of quantum assembly languages to executable binaries via the quantum intermediate representation (QIR). We provide a new dialect to MLIR for quantum assembly languages that can be lowered to LLVM IR adherent to the QIR specification. Linking to a `qcor`-enabled, QIR runtime library implementation will enables a retargetable compiler that produces executable binaries and object code for available quantum co-processors.

a unified representation that quantum languages and compilers can map to in an effort to promote reusability of common quantum compilation tools, strategies, and backend runtime implementations. The integration of the MLIR with the recently proposed QIR specification would directly enable quantum code optimization and progressive lowering to executable code in a quantum language and hardware agnostic fashion. Extending MLIR for quantum assembly languages would enable existing languages to quickly map to the LLVM IR, and then through appropriate library implementation linkage, execute on available quantum processors. This whole workflow would be directly integrated with popular classical compiler infrastructures and promote a hybrid quantum-classical workflow that is tightly integrated.

In this work, we present a novel MLIR dialect for quantum assembly languages and implement lowering routines to map this representation to an instance of the LLVM IR adherent to the QIR specification. We demonstrate this workflow by providing a parser for the OpenQASM language that generates a corresponding MLIR instance. Ultimately we provide a compiler for quantum assembly languages (with OpenQASM as a first example) that takes quantum language source files and produces executables or library code that targets a desired quantum co-processor. Our work implements the QIR specification with a quantum runtime implementation backed by the `qcor` quantum-classical compiler platform, enabling execution on currently available quantum computers.

II. Background

Here we provide some necessary background on the underlying technologies and specifications we leverage to provide a mechanism for compiling quantum languages through the MLIR, lowered to the LLVM IR, and ultimately to binary executables or object code. At a high-level, our approach can be seen in Figure 1. We put forward an extension to the MLIR infrastructure specifically for quantum computing, with the intention that any and all quantum languages are able to be parsed and mapped to an instance of this representation. We then provide the implementation necessary to map that representation to the LLVM IR which can then be linked with a backend-agnostic quantum runtime, thus enabling any quantum language to be compiled to an executable binary. Specifically, we detail at a high-level the internal structure of the MLIR, the new quantum intermediate representation (QIR) specification, and the QCOR quantum-classical compiler platform recently put foward at ORNL, which serves as the underlying quantum runtime enabling execution of the generated QIR code.

MLIR*:* The Multi-Level Intermediate Representation (MLIR) is a novel approach for constructing new domain specific compilers in a reusable and extensible manner. It puts forward a novel single-static assignment (SSA)-based intermediate representation for compiler developers that enables the encoding of higher levels of language abstraction, and the ability to progressively lower this representation to other levels of IR abstraction, ultimately targeting a machine-level IR (e.g. LLVM IR). For a full description of the framework, we invite the interested reader to see [3], [16]. Here

we detail a few key points that are important for the discussion of the work presented here.

The key abstraction promoted by the MLIR is an *operation*, represented in the framework as an `Op` class. Everything in the language (instructions, functions, composites of functions) is modeled as an `Op`, which is itself an extension point to the framework (developers can contribute new operations to define new semantics and functionality). Each `Op` has a unique string identifier (the operation name), can take and return zero or more values (the class `mlir::Value`), and retains a unique dictionary of constant information related to the `Op` called `Attributes`. Since these `Attributes` must be constant, they are used to represent compile-time information about the operation.

Developers contribute new `Ops` via the introduction of a custom `Dialect` subclass. `Dialects` represent a logical grouping of `Ops` under a domain-specific and unique namespace string identifier. Each new `Dialect` registers at creation its provided set of operations, and in this way promotes modularity and extensibility of language parsing and lowering within the MLIR system. MLIR provides a builtin `Dialect` for expressing common concepts such as functions and modules (composites of functions). Other useful `Dialects` have been contributed as well - one representing the LLVM IR, others for GPU-accelerated programming models, and still others for operating on vectors and tensors common to machine-learning application workflows.

Critically, MLIR provides a robust infrastructure for transformations on operations that enable progressive lowering of the representation to lower levels of IR abstraction. The key concepts put forward here are the `ConversionTarget and rewrite patterns` represented as `ConversionPattern` class instances. First, one specifies a `ConversionTarget` sub-type that enforces the requirement that all transformed or rewritten `Ops` are *legal* with respect to that target `Dialect` (i.e. the `LLVMConversionTarget` enforces the requirement that all resultant operations are valid LLVM dialect operations). Next, for each `Op` in the custom `Dialect` that should be lowered, one can define a `ConversionPattern` sub-type that exposes a mechanism for specifying a replacement operation while marking the original operation as *erased*. Ultimately, these extensions can be contributed to the MLIR framework and leveraged as part of a built-in `mlir::PassManager` pipeline that affects lowering of the high-level custom `Dialect` down to the conversion target.

***QIR*:** The quantum intermediate representation (QIR) specification has recently been published and promoted by researchers at Microsoft and is intended to provide a common, unified compiled representation for quantum languages targeting any gate-based quantum computing platform. Critically, QIR is based on the LLVM IR (ultimately it is the LLVM IR) and specifies key concepts and rules for representing quantum instructions, register allocation, qubit addressing, and measurement retrieval. This approach promotes the utility of the full classical capabilities that LLVM provides, thereby enabling a mechanism for robust classical computation in tandem with quantum co-processing or acceleration.

For the full specification, we direct the reader to [5], however here we wish to highlight a few key points that are critical for the work described here. The key concepts in the specification are register allocation, qubit addressing, and instruction invocation, and we provide an example QIR code in Figure 2 to demonstrates these concepts. First note that the specification leaves qubits as an opaque type so that implementations of an associated runtime library can define it in a way that best fits the implementation. Similarly, a measurement result type is also marked as opaque. Next, all runtime functions are declared but not implemented and are intended to be provided externally and linked in later. This design decision enables maximum flexibility to swap out various quantum runtime implementations for specific hardware backends. The specification defines a qubit register allocation function - `__quantum__rt__qubit_allocate_array` - which takes as input the number of qubits to allocate and returns a pointer to the `Qubit` array. Individual qubits are queried or addressed via a declared `__quantum__rt__array_get_element_ptr_1d` function, which takes an allocated qubit register and a qubit element index integer and returns a pointer to that `Qubit`. Quantum instructions are exposed as declared `__quantum__qis__INSTNAME`, where `INSTNAME` is the name of the instruction (h for Hadamard, cnot for CNOT, etc.). The remainder of the code snippet follows any typical LLVM IR disassembled code (an LLVM *.ll file) where each line is an instruction call in the SSA form. A `call` is made to the quantum runtime functions with given input values and return results are stored to new value names. Figure 2 allocates a register of size 2, gets the 0th qubit, executes a Hadamard on that qubit to place it in superposition, gets the 1st qubit, executes a CNOT to entangle the qubits, and finally measures both to produce single-bit `Results`.

These are only a few of the quantum runtime functions detailed by the QIR specification, but this should give an overall perspective on the structure of a com-

```
%Array = type opaque
%Result = type opaque
%Qubit = type opaque

declare %Result* @__quantum__qis__mz(%Qubit* %0)
declare void @__quantum__qis__cnot(%Qubit* %0, %Qubit* %1)
declare void @__quantum__qis__h(%Qubit* %0)

declare %Array* @__quantum__rt__qubit_allocate_array(i64 %0)
declare void @__quantum__rt__qubit_release_array(%Array* %0)
declare i8* @__quantum__rt__array_get_element_ptr_1d(%Array* %0, i64 %1)

define i32 @main(i32 %0, i8** %1)  {
  %4 = call %Array* @__quantum__rt__qubit_allocate_array(i64 2)
  %5 = call i8* @__quantum__rt__array_get_element_ptr_1d(%Array* %4, i64 0)
  %6 = bitcast i8* %5 to %Qubit**
  %7 = load %Qubit*, %Qubit** %6, align 8
  call void @__quantum__qis__h(%Qubit* %7)
  %8 = call i8* @__quantum__rt__array_get_element_ptr_1d(%Array* %4, i64 1)
  %9 = bitcast i8* %8 to %Qubit**
  %10 = load %Qubit*, %Qubit** %9, align 8
  call void @__quantum__qis__cnot(%Qubit* %7, %Qubit* %10)
  %11 = call %Result* @__quantum__qis__mz(%Qubit* %7)
  %12 = call %Result* @__quantum__qis__mz(%Qubit* %10)
  call void @__quantum__rt__qubit_release_array(%Array* %4)
  ret i32 0
}
```

Figure 2: Bell state measurement expressed in the QIR. Qubits and Results are opaque types, while quantum runtime functions are declared and implementations are expected to be linked in later.

piled QIR program. Since this is just LLVM IR, we get standard program control flow structures and semantics for free, and we are free to integrate these quantum API calls with existing classical workflows.

qcor: Recently, researchers from Oak Ridge National Laboratory have put forward a novel language extension to C++ for heterogeneous quantum-classical computing [21]. The qcor compiler platform promotes a single-source programming model that enables programmers to write quantum functions natively in C++ alongside classical code and compile to available quantum processors in a hardware-agnostic manner. qcor extends C++ through a novel extension to the Clang plugin system [10] that enables the parsing of invalid quantum function bodies and their translation to valid C++ API calls at compile time. To achieve compiler retargetability, qcor ultimately delegates to the XACC quantum programming framework [18], which provides a system-level C++ approach to quantum-classical programming, compilation, and execution in a service-oriented fashion. XACC exposes a number of interfaces that span the quantum compilation and execution workflow that enable quantum language parsing, quantum intermediate representation generation, circuit optimization, and backend execution on QPUs from IBM, Rigetti, D-Wave, Honeywell, among others. Ultimately, XACC provides qcor with an API for ubiquitous quantum compilation and execution tasks.

To promote backend extensibility, qcor puts forward an extensible QuantumRuntime interface that exposes an API for quantum gate execution and can be implemented for both remotely hosted QPU protocols as well as future, tight integration models enabling CPU-QPU fast-feedback. These implementations directly delegate to the XACC Accelerator infrastructure to enable overall quantum compiler retargetability. For a complete background on the qcor compiler, we refer the reader to [1], [4], [21], but for the purposes of this work, the QuantumRuntime infrastructure is the primary background needed to detail the execution of MLIR-generated QIR code on available QPUs.

Quantum MLIR Dialect

Opaque Types: Qubit | Result | Array | ArgvType

QRTInitOp
- Attributes: N/A
- Operands: argc: I32, argv: ArgvType
- Return: void

InstOp
- Attributes: name : str, params: Vec<Float64>
- Operands: qubits: Variadic<Qubit>
- Return: Optional<Result>

QallocOp
- Attributes: name : str, params: Vec<Float64>
- Operands: N/A
- Return: Array

QubitExtractOp
- Attributes: N/A
- Operands: qubits: Array, idx : I64
- Return: qubit : Qubit

DeallocOp
- Attributes: N/A
- Operands: qubits: Array
- Return: void

QRTFinalize
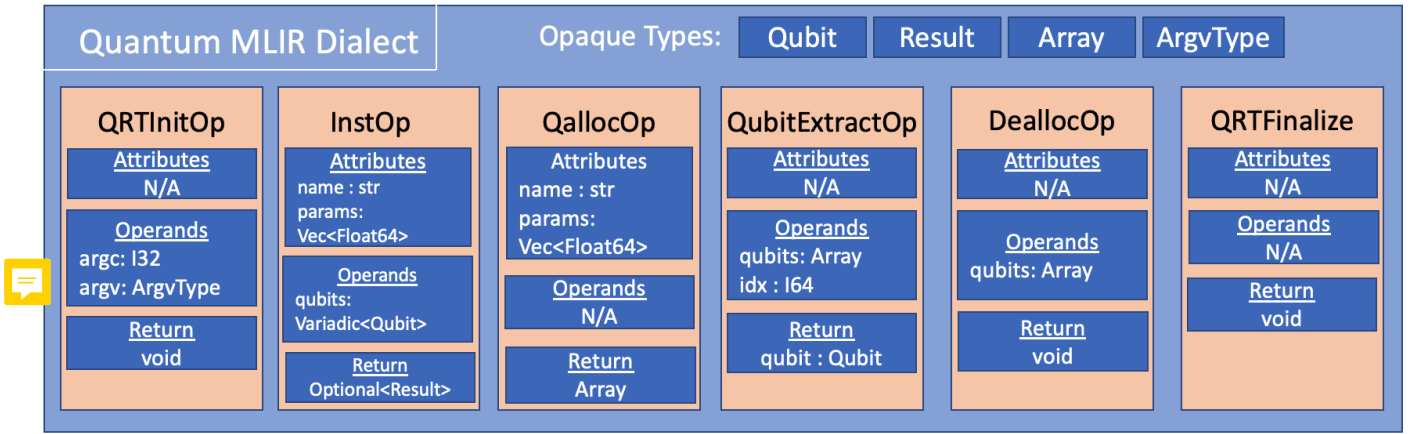- Attributes: N/A
- Operands: N/A
- Return: void

Figure 3: Graphical representation of the operations contributed from the quantum MLIR dialect. These operations enable qubit array allocation and deallocation, qubit addressing, quantum instruction invocation, and a mechanism for initializing and finalizing the QIR runtime implementation. Each operation keeps track of constant attributes, and can take operands as input and return specific values.

III. A Quantum MLIR Dialect

To enable executable code generation targeting quantum co-processors from available quantum assembly languages, we seek to leverage the progressive lowering capabilities put forward by the MLIR compiler framework. This infrastructure provides a mechanism for mapping language-level intermediate representations down to classical assembly and object code via the LLVM IR. To properly interface quantum assembly languages with MLIR, we must provide a custom MLIR Dialect that exposes new operations (Op sub-classes) enabling the expression of common quantum assembly semantics. Specifically, we envision this work (this Dialect extension) as a first pass or initial prototype to demonstrate the overall utility of the MLIR for quantum code compilation and execution. Therefore, we keep our required feature-set small and define a Dialect that enables (1) quantum register allocation and deallocation, (2) qubit addressing within an allocated register, (3) simple quantum instruction invocation on addressed qubits, and (4) composition of quantum functions. These features enable one to perform simple circuit compilation and execution, and other higher-level languages should be able to map down to this lower-level of abstraction.

We define a QuantumDialect subclass of mlir::Dialect which serves as an aggregator for custom Ops exposing these four quantum assembly features. Figure 3 provides a graphical representation of this dialect and the operations it provides. The QuantumDialect exposes a unique namespace identifier - quantum - and registers six Op subclasses under the quantum namespace: the QRTInitOp, QRTFinalizeOp, QallocOp, DeallocOp, QubitExtractOp, and InstOp. The QRTInitOp and QRTFinalizeOp enable the initialization and finalization of a provided QIR runtime library implementation. Specifically, QRTInitOp expects the typical argc, argv input arguments coming from main(). We define an mlir::OpaqueType to describe the incoming char** argument.

The QallocOp performs qubit register allocation and exposes a unique name - qalloc - as well as a dictionary of compile-time attributes indicating the size and unique variable name for the register. It returns an Array, and opaque type that maps directly to the QIR array concept. The Array is assumed to be an array of the Qubit OpaqueType. The QallocOp does not take any input argument operands since we treat the register size as a compile-time constant, however, future iterations may wish to treat the register size as an input operand. To address individual qubits, the quantum dialect exposes a QubitExtractOp which takes and input Array value and a 64-bit integer describing the index of the array to retrieve. It returns an instance of Qubit. In order to ensure allocated qubit registers are deallocated upon going out of scope, the dialect provides a DeallocOp which takes as input the Array of Qubits to deallocate.

Finally, we define the InstOp to model any operation that affects execution of a specific quantum instruction on a provided qubit or qubits. This Op exposes the unique inst name and a compile-time attribute dictionary indicating the name of the instruction and any potential double parameters (gate rotation angles, for example). Once again, for this first prototype implementation, we treat potential parameters as compile-

```
module {
  func @main(%arg0: i32, %arg1: !quantum.ArgvType) -> i32 {
    "quantum.init"(%arg0, %arg1) : (i32, !quantum.ArgvType) -> ()
    %0 = "quantum.qalloc"() {name = "q", size = 1 : i64} : () -> !quantum.Array
    %c0_i64 = constant 0 : i64
    %1 = "quantum.qextract"(%0, %c0_i64) : (!quantum.Array, i64) -> !quantum.Qubit
    %2 = "quantum.inst"(%1) {name = "h"} : (!quantum.Qubit) -> none
    "quantum.dealloc"(%0) : (!quantum.Array) -> ()
    "quantum.finalize"() : () -> ()
    %c0_i32 = constant 0 : i32
    return %c0_i32 : i32
  }
}
```

Figure 4: Example MLIR instance using the quantum dialect. This code allocates a register of size 1, gets reference to that single qubit, executes a hadamard gate, and then deallocates the register.

time constants, but future iterations may choose to move this to input operands. Qubits that the instruction operates on are treated as input operands, and are modeled as a variadic list of type Qubit, allowing one to invoke single or multi-qubit operations. InstOps return an optional Result - most instructions do not return anything, while measurement instruction calls will return a binary result (qubit measured 0 or 1).

Figure 4 gives a simple example of a printed MLIR instance leveraging operations from the quantum dialect. This example affects the application of a hadamard gate on a single qubit. The example demonstrates the utility of the various operations and shows runtime initialization and finalization, quantum register allocation and deallocation, and qubit addressing and gate invocation.

IV. MLIR Generation

Now that we have discussed the overall architecture of the new MLIR quantum dialect, we turn our attention to an extensible mechanism for the generation of this representation. The goal here is to enable a unique extension point that language developers can implement to take source strings to an instance of the mlir::ModuleOp containing one or many mlir::FuncOp instances each composed of quantum dialect operations.

We put forward the QuantumMLIRGenerator interface in Figure 5 and intend it to be implemented by quantum assembly language developers to take in a language source string and output an MLIR module. The interface is simple, but enables sub-types to perform any initialization, language parsing to tree-like representations, tree-to-MLIR translation, and any finalization routines. Clients of this interface can initialize the

generator and indicate whether to produce an MLIR instance with a main() function entry point. This feature is useful for the purposes of compiling assembly representations to object code versus executables. The mlirgen() method is meant to be implemented by sub-types to generate language specific abstract syntax trees and map that tree to the corresponding MLIR quantum dialect operations. We demonstrate the implementation of this interface for OpenQASM in Section VII, and leave implementations for other languages like Quil [24] and Jaqal [20] for future work.

V. Lowering to QIR

With a valid MLIR dialect for quantum assembly languages in place, we now turn our attention to the specific mechanisms for lowering quantum operations to valid LLVM dialect operations. Specifically, we seek a mechanism to lower to the LLVM IR in a manner that is adherent to the recently defined quantum intermediate representation (QIR). Fortunately, MLIR currently has a built-in dialect for the LLVM IR and puts forward a robust and general mechanism for translating operations from one dialect to another. We leverage this functionality, and provide translation mechanisms that map our custom Op data structures to valid operations in the LLVM MLIR Dialect in a way that is QIR specification adherent.

The basic workflow for lowering one MLIR Dialect to the LLVM Dialect is demonstrated in Figure 6. MLIR provides a mlir::PassManager data structure that orchestrates the execution of custom and default passes that act to transform the input mlir::ModuleOp. Developers can contribute new passes as subclasses of the mlir::PassWrapper template type, here we provide one called QuantumToLLVMLoweringPass. This

```cpp
class QuantumMLIRGenerator {
 public:
  QuantumMLIRGenerator(mlir::MLIRContext& ctx);
  virtual void initialize_mlirgen(bool add_entry_point = true,
                                  const std::string file_name = "") = 0;
  virtual void mlirgen(const std::string& src) = 0;
  mlir::OwningModuleRef get_module();
  virtual void finalize_mlirgen() = 0;
};
class OpenQasmMLIRGenerator : public QuantumMLIRGenerator {...}
class QuilMLIRGenerator : public QuantumMLIRGenerator {...}
class JaqalMLIRGenerator : public QuantumMLIRGenerator {...}
...
```

Figure 5: Class definition for the Quantum MLIR generation extension point. Developers can sub-type this class to provide a mechanism to map language-specific source strings to an instance of the MLIR leveraging the quantum dialect.

pass sets the conversion target to the LLVM dialect and contribute a custom `Op` lowering pattern for one of our quantum dialect operations. Once the pass manager workflow is run, the resultant `ModuleOp` should entirely consist of `Op`s from the LLVM Dialect. Finally, MLIR provides an API function for translating that `ModuleOp` to an LLVM `Module`, which is the top-level abstraction for LLVM IR. From there, one can readily lower to object code for the target machine.

The goal of each conversion operation is to erase the `Op` from the quantum dialect and replace it with an `Op` from LLVM Dialect. Specifically, we replace each with its corresponding QIR runtime function call represented as a `mlir::CallOp` with input operand represented as a `LLVM::FuncOp`. Figure 7 demonstrates the same code in Figure 4 lowered to the LLVM MLIR dialect via our contributed lowering operations. For the `QRTInitOp` we provide a rewrite pattern that erases the operation and introduces a declaration and call to the function `__quantum__rt__initialize()` and set its input operands to the block arguments coming from the parent `main()` function. For the `QallocOp`, we extract the size attribute and create a new LLVM `ConstantOp` to return a 64-bit integer representation of the size, which then serves as the input operand of a call to `__quantum__rt__qubit_allocate_array()`. The return value for this new LLVM operation is an LLVM `PointerType` to the opaque `Array` structure. Extracting a qubit from that array, the `QubitExtractOp`, is rewritten as a call to `__quantum__rt__array_get_element_ptr_1d()` with the allocated array pointer serving as the input operand. This function (as per the QIR specification) returns an `i8` pointer, so we therefore also introduce

LLVM `BitcastOp` and `LoadOp` instances to map it to a qubit pointer (the opaque `Qubit` type). `InstOp` is mapped to a `__quantum__qis__INSTNAME` where `INSTNAME` is extracted from the instruction operation's attributes dictionary. The input operand for this LLVM function operation is the casted qubit pointer. Finally, the `DeallocOp` is mapped to a call to `__quantum__rt__qubit_release_array()`

```cpp
// Create the PassManager for lowering
// to LLVM MLIR and run it
mlir::PassManager pm(&context);
auto q_to_llvm =
 std::make_unique<QuantumToLLVMLoweringPass>();
pm.addPass(q_to_llvm);
auto module_op = module.getOperation();
if (mlir::failed(pm.run(module_op))) {
   std::cout << "Pass Manager Failed\n";
   return 1;
}

//! module_op is now totally in LLVM Dialect !

// Now lower MLIR to LLVM IR
llvm::LLVMContext llvmContext;
auto llvmModule =
   mlir::translateModuleToLLVMIR(module,
                                 llvmContext);
```

Figure 6: Code snippet demonstrating how one may apply the MLIR pass management system to lower representations in the quantum dialect to equivalent expressions using the LLVM dialect.

```
module  {
  llvm.func @__quantum__rt__qubit_release_array(!llvm.ptr<struct<"Array", opaque>>)
  llvm.func @__quantum__qis__h(!llvm.ptr<struct<"Qubit", opaque>>)
  llvm.func @__quantum__rt__array_get_element_ptr_1d(!llvm.ptr<struct<"Array", opaque>>, i64) \
                                                                      -> !llvm.ptr<i8>
  llvm.func @__quantum__rt__qubit_allocate_array(i64) -> !llvm.ptr<struct<"Array", opaque>>
  llvm.func @__quantum__rt__finalize()
  llvm.func @__quantum__rt__initialize(i32, !llvm.ptr<ptr<i8>>) -> i32
  llvm.func @main(%arg0: i32, %arg1: !llvm.ptr<ptr<i8>>) -> i32 {
    %0 = llvm.call @__quantum__rt__initialize(%arg0, %arg1) : (i32, !llvm.ptr<ptr<i8>>) -> i32
    %0 = llvm.mlir.constant(1 : i64) : i64
    %1 = llvm.call @__quantum__rt__qubit_allocate_array(%0) : (i64) \
                                            -> !llvm.ptr<struct<"Array", opaque>>
    %2 = llvm.mlir.constant(0 : i64) : i64
    %3 = llvm.call @__quantum__rt__array_get_element_ptr_1d(%1, %2) : \
                              (!llvm.ptr<struct<"Array", opaque>>, i64) -> !llvm.ptr<i8>
    %4 = llvm.bitcast %3 : !llvm.ptr<i8> to !llvm.ptr<ptr<struct<"Qubit", opaque>>>
    %5 = llvm.load %4 : !llvm.ptr<ptr<struct<"Qubit", opaque>>>
    %6 = llvm.call @__quantum__qis__h(%5) : (!llvm.ptr<struct<"Qubit", opaque>>) -> !llvm.void
    %7 = llvm.call @__quantum__rt__qubit_release_array(%1) : \
                              (!llvm.ptr<struct<"Array", opaque>>) -> !llvm.void
    %8 = llvm.call @__quantum__rt__finalize() : () -> !llvm.void
    %9 = llvm.mlir.constant(0 : i32) : i32
    llvm.return %9 : i32
  }
}
```

Figure 7: Translated MLIR code in the LLVM dialect corresponding to the quantum dialect code in Figure 4.

with input operand pointing to the original allocated qubit array value. We end with a call to `__quantum__rt__finalize()` and add a return operation to terminate the block. These conversion operations are all contributed to the MLIR pass management system via our custom `QuantumToLLVMLoweringPass`.

Once the quantum assembly code is represented using the LLVM MLIR dialect, we are able to lower that representation even further to the LLVM IR using the built-in `mlir::translateModuleToLLVMIR()` function. This returns an instance of the `llvm::Module` which we are then able to persist to a human-readable, LLVM assembly language *.ll file. We can then use existing tools to lower this even further to object code. Specifically, LLVM provides the `llvm-as` and `llc` tools which can lower this file to machine-specific object code. We can then link this object file with a custom QIR runtime API implementation to output a binary executable that will affect execution of the compiled quantum assembly code on a target quantum co-processor.

VI. QIR Runtime Implementation

We provide an implementation of the QIR runtime API that is backed by the `qcor` quantum-classical compiler platform. This implementation stipulates that the `Array` opaque type maps to a `std::vector<int8_t*>`, and the `Qubit` and `Result` map to a `uint64_t`. We implement `__quantum__rt__initialize()` to take in any input command line arguments and initialize the `qcor` runtime. This initialization sets the target quantum co-processor, and any related options, and selects the runtime mode. This mode can be one of two choices - the fault-tolerant, tightly integrated `ftqc` mode, or the remotely-hosted, loosely-coupled `nisq` mode (for currently available quantum processors). The mode can be configured from the command line as this initialization runtime function reads input command line arguments.

The `qcor` quantum runtime exposes API calls for executing specific instructions on the backend quantum co-processor. For each implementation of `__quantum__qis__INSTNAME`, we delegate execution to the appropriate `qcor` runtime API call (e.g. `__quantum__qis__h(Qubit* q)` maps to `::quantum::h(qubit& q)`). qcor represents quantum

registers using a `qreg` data type that delegates to the `xacc::AcceleratorBuffer` concept. We implement `__quantum__rt__qubit_allocate_array()` to create this underlying `AcceleratorBuffer` of the given register size. Calls to measure persist bit configurations to this allocated buffer, and return the measured bit configuration in the case of FTQC execution.

Internally, `qcor` circuit execution builds upon the `xacc::Accelerator` subsystem. The `Accelerator` interface exposes a mechanism for execution of the XACC quantum intermediate representation as well as methods for querying backend-specific information like qubit connectivity, noise parameters, gate set, etc. XACC provides implementations of this interface for targeting all IBM quantum computing backends, the Rigetti QCS platform, Honeywell ion trap quantum computers, and a number of simulators that scale on Summit-like [12] architectures. Our QIR runtime implementation inherits this extensibility and retargetability by delegating to the `qcor` runtime execution infrastructure. Assembly languages lowered to the QIR are therefore able to be lowered to executable binaries that target a wide range of quantum co-processors.

VII. Evaluation

Here we evaluate utility and efficacy of this quantum compiler infrastructure built upon the MLIR framework. We demonstrate its utility for the OpenQASM quantum assembly language, and describe integration efforts for compiled OpenQASM libraries with exiting `qcor` C++ application codes. Our work provides a mechanism for compiling libraries of quantum codes written in OpenQASM that can then be included and linked to in future `qcor` programs. We end with a discussion and demonstration of the runtime of this language-to-object code workflow, and compare with the compiler workflow provided by Q#.

A. OpenQASM Compiler

OpenQASM is a popular assembly language for currently available IBM quantum processors [8], is supported by a large number of software toolchains, and exposes language features like register allocation, qubit addressing, and instruction invocation, among others. For this work, we leverage an off-the-shelf parser for OpenQASM provided by the `staq` project [6]. This project provides a Clang-inspired OpenQASM parser in C++ that translates language strings to a custom abstract syntax tree (AST) representation. Moreover, the project provides a robust visitor pattern on top of this representation that makes `staq` well-suited for key circuit transformation, optimization, and analysis tasks. We leverage this visitor pattern to walk the

```
OpenQASM 2.0;
include "qelib1.inc";
qreg q[2];
creg c[2];                  |
h q[0];                     | staq ast-gen
cx q[0], q[1];              V
measure q -> c;
```

```
|- Program
  ... qelib.inc (not included for brevity)...
  |- Register Decl(q[2], quantum)
  |- Register Decl(c[2])
  |- Declared(h)
   |- Var(q[0])
  |- Declared(cx)        key staq ast nodes:
   |- Var(q[0])              - RegisterDecl
   |- Var(q[1])              - DeclaredGate
  |- Measure                 - CNOTGate
   |- Var(q[0])              - UGate
   |- Var(c[0])              - MeasureStmt
  |- Measure
   |- Var(q[1])
   |- Var(c[1])
```

Figure 8: (top) A simple OpenQASM code to create a 2-qubit bell state followed by a measurement of all qubits. (bottom) `staq` AST representation demonstrating key nodes that need to be visited to map to an equivalent MLIR representation.

OpenQASM AST and map key nodes to corresponding MLIR `Ops` from our quantum dialect.

The `staq AST` exposes a top-level `ASTNode` type that all node-types derive from. The primary nodes that we care about in the generation of a `ModuleOp` composed of operations from the quantum dialect are the `RegisterDecl`, `UGate`, `CNOTGate`, `DeclaredGate`, and `MeasureStmt`. We provide a visitor implementation (`OpenQasmMLIRGenerator` that additionally sub-types the `QuantumMLIRGenerator` interface (see Section IV) and implements `mlirgen()` to walk the AST and visit these specific nodes to create corresponding operations from our quantum dialect. Statements like `qreg q[2]` ultimately map to a `RegisterDecl` in the `staq` AST and visitation of that node provides the register variable name and size information (q and 2 for `qreg q[2]`). With this information, our visit implementation can construct a new `QallocOp` and set its constant name and size attributes and insert it into an pre-initialized `ModuleOp` class member on the visitor class implementation. Note that downstream `InstOps` and `QubitExtractOps` will need reference to the return

```
$ qcor bell.qasm
$ ./a.out -qrt nisq -qpu ibm:ibmq_paris
```

Figure 9: Compile the OpenQASM code in Figure 8 (first line). Run the resultant executable targeting an IBM physical backend (second line).

register from the `QallocOp` as its input operands, therefore, we store all `QallocOp` instances in a symbol table stored on as a visitor class member.

Next, for gates that are not `U` or `CNOT` (these are separated out in the tree since they represent the common basis gate set on the IBM processor), we visit the `DeclaredGate` node (`h`, `rx`, `cx`, etc. - these gates ultimately lower to `U` and `CNOT`). This node provides information about the name of gate instruction and the qubits being operated on - specifically their address in the pre-allocated quantum register (h q[0] gives us the name h, the register q, and the qubit index 0). Within this visit implementation we leverage this information to construct a new `QubitExtractOp` that takes the correct `QallocOp` return operand retrieved from the symbol table, and a `mlir::ConstantOp` for generating the constant qubit index integer. The return operand from the `QubitExtractOp` can then be used as an input operand on a constructed `InstOp`, which is also seeded with corresponding constant attributes for the name of the instruction and any constant gate rotation parameters.

We integrate this workflow within the `qcor` compiler platform. We provide a `qcor-mlir-tool` executable that takes an OpenQASM file as input and outputs an LLVM IR assembly file. Optionally, users can pass `-emit=mlir`, `-emit=mlir-llvm`, or `-emit=llvm` to print out the MLIR quantum, MLIR LLVM, or LLVM IR representations to standard out, respectively. Users may also use this tool to compile to LLVM IR with or without a `main()` entry point (the `-no-entrypoint` argument causes the workflow to output the LLVM IR without a `main` function). Note this is useful in mapping OpenQASM source files to executables or libraries for linking later. We have updated the `qcor` compiler driver to delegate to this tool when passed OpenQASM source files. Users can compile and execute OpenQASM codes as in Figure 9.

It is interesting to note that the same parsing process can be performed for other quantum assembly languages. We anticipate that existing parsing mechanisms for languages such as quil, jaqal, and others rely on some sort of parse tree representation that can be traversed with specific general actions being invoked for differing node types. We observe that a number of languages provide an extended Bachus-Naur form grammar specification that could easily generate parse tree data structures through tools and frameworks like Antlr [22]. Frameworks such as this also provide listener and visitor patterns which can be leveraged to create quantum `Ops` and effectively map the language to our MLIR quantum dialect.

B. Linking to Compiled Assembly Code

The ability to compile assembly languages with `qcor` using the `-no-entrypoint` command line option enables one to produce libraries of quantum functions that one should be able to link to and invoke from existing C++ application codes. To do this, one would need some means to retrieve execution results from the linked OpenQASM function. In `qcor`, execution results are persisted to an allocated `qreg` instance, so what we need is a mechanism for passing an allocated `qreg` to the quantum function provided by the compiled object file. To achieve this, we extend the quantum dialect with a `SetQregOp` operation that takes as input a value of opaque type `qreg`. Runtime implementations are free to define the `qreg` and provide it

```
$ qcor -no-entrypoint bell.qasm
$ ls
    bell.o bell.qasm
-----------------------------------------------
#include "qcor.hpp"

// Macro that maps to
// extern "C" void bell(qreg);
include_qcor_qasm(bell)

int main() {
    auto q = qalloc(2);
    // Function from bell.o
    bell(q)
    for (auto [bit, count] : q.counts()) {
        print(bit, ":", count);
    }
    return 0;
}
-----------------------------------------------
$ qcor bell.o test.cpp -o test.x
$ ./test.x -qrt nisq -shots 2048 -qpu aer
00 : 1025
11 : 1023
```

Figure 10: Code snippet demonstrating the compilation of an OpenQASM code and subsequent linking within a standard `qcor` C++ code.

at the link phase, and we map it to the qcor `qreg` data type. The `SetQregOp` operation lowers to a new runtime function we define in the QIR API specification `__quantum__rt__set_qreg(qreg*)`, and is injected at the start of the top-level, entry-point quantum MLIR function. Our implementation provides this function and takes the `qreg` for use in persisting measurement results.

This functionality promotes the ability for programmers to define quantum code in stand-alone OpenQASM source files, compile them with `qcor` (delegating to our `qcor-mlir-tool` to generate LLVM IR) using the `-no-entrypoint` flag, and link them into other `qcor` codes for invocation of the pre-compiled quantum assembly code. Figure 10 demonstrates this workflow, where users compile OpenQASM to an object file, call a pre-defined macro `include_qcor_qasm()` to setup the external linkage, and then call the function like any other C++ function.

C. Compiler Benchmarks

In this evaluation, we compare the compilation time of the QCOR compiler using the MLIR-based compilation workflow vs. the code-rewrite via syntax handler plugin approach. Specifically, in the hierarchical MLIR compilation workflow, the compilation runtime involves QASM to MLIR to QIR lowering and LLVM IR compilation (via `llvm-as` and `llc -filetype=obj`. On the other hand, when using the syntax handler, we embed the QASM source into the `.cpp` source file as a quantum kernel and use QCOR to compile the mixed quantum-classical source.

For comparison, we transpile the QASM test cases into Q# and time the Q# compilation[1]. The result is shown in Figure 11 for a set of OpenQASM source files [25] for common arithmetic quantum algorithms, such as adder, quantum Fourier transform, etc.

The benefit of hierarchical lowering via MLIR to compilation time is significant (up to 1 to 2 order of magnitude) compared to the Clang syntax-hander plugin (CSP) approach [21]. It is worth noting that the CSP approach comprises two steps: (1) transformation of embedded QASM into C++ data structures and (2) compilation of the rewritten source code. The second step is equivalent to the compilation workflow of the Q# compiler since the QASM sources have been transpiled into Q# code.

Both the Q# and QCOR-CSP compilers can handle higher-level language constructs, such as conditional statements, loops, function calls, etc., thus, are gearing towards quantum kernels written in that manner.

[1]Command-line Q# compiler (`qsc`) from Microsoft Quantum SDK version 0.14.2011120240 for Linux
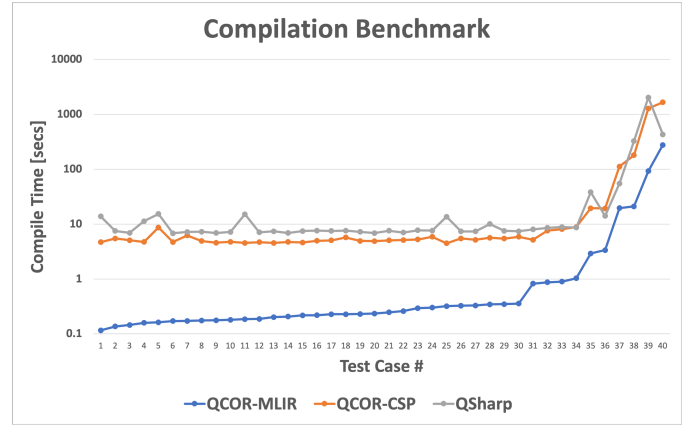

Figure 11: Compilation time comparison between MLIR-based (QCOR-MLIR) and syntax handler plugin-based (QCOR-CSP) QCOR compilation and the Q# compiler (QSharp). The test cases are flattened OpenQASM source files for the QCOR compiler and transpiled Q# sources for the Q# compiler. In this plot, the x-axis represents individual test cases sorted by the runtime of the QCOR-MLIR compiler.

On the other hand, direct QASM to MLIR then QIR lowering and external linkage, as demonstrated in subsection VII-B, is a more efficient compilation approach to integrate pure QASM kernels into the heterogeneous quantum-classical programming workflow of QCOR.

VIII. Conclusion

We have presented a compilation infrastructure for quantum assembly languages that builds upon the extensible and reusable MLIR infrastructure and enables progressive lowering to the LLVM IR in a manner that is adherent to the Microsoft quantum intermediate representation (QIR). We extend the MLIR with a new dialect for quantum computing exposing operations for qubit memory management, addressing, and quantum instruction invocation. We have provided lowering routines for taking this operations to valid operations in the LLVM dialect. Our LLVM representation adheres to the QIR, and we provide a runtime API implementation that delegates execution to the `qcor` runtime library. We demonstrate the utility of this approach for a prototypical assembly language - OpenQASM. Our work enables quantum-classical compilation to executables or libraries, and for the latter can be leveraged within existing application use cases.

We see this work as a foundational layer for continued development of quantum-classical compilers, frameworks, and tools. We envision the current OpenQASM workflow as a template that other developers may follow to extend this MLIR compiler workflow to

other quantum assembly languages. This internal MLIR representation, and the lower QIR representation, can serve as a unified representation that enables integration across a number of quantum software research and development activities. We also note that a new version of OpenQASM (version 3.0) has recently been released that provides a more rich feature set enabling a hybrid quantum-classical approach at a higher-level of abstraction then low-level assembly. We belive this infrastructure could serve as a base foundation for the development of a true ahead-of-time compiler for OpenQASM 3.0 codes, enabling backend execution on available quantum co-processors.

Acknowledgment

References

[1] Aide-qc documentation. http://docs.aide-qc.org. Accessed: 2021-01-15.

[2] Introducing quantum intermediate representation (qir). https://devblogs.microsoft.com/qsharp/introducing-quantum-intermediate-representation-qir/. Accessed: 2021-01-15.

[3] Mlir documentation. https://mlir.llvm.org. Accessed: 2021-01-15.

[4] Qcor github. https://github.com/ornl-qci/qcor. Accessed: 2021-01-15.

[5] Quantum intermediate representation (qir) specification. https://github.com/microsoft/qsharp-language/tree/main/Specifications/QIR. Accessed: 2021-01-15.

[6] Matthew Amy and Vlad Gheorghiu. staq – A full-stack quantum processing toolkit. *arXiv e-prints*, page arXiv:1912.06070, December 2019.

[7] Samuel F. Antao, Alexey Bataev, Arpith C. Jacob, Gheorghe-Teodor Bercea, Alexandre E. Eichenberger, Georgios Rokos, Matt Martineau, Tian Jin, Guray Ozen, Zehra Sura, Tong Chen, Hyojin Sung, Carlo Bertolli, and Kevin O'Brien. Offloading support for openmp in clang and llvm. In *Proceedings of the Third Workshop on LLVM Compiler Infrastructure in HPC*, LLVM-HPC '16, page 1–11. IEEE Press, 2016.

[8] Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. Open Quantum Assembly Language. *arXiv e-prints*, page arXiv:1707.03429, Jul 2017.

[9] E. F. Dumitrescu, A. J. McCaskey, G. Hagen, G. R. Jansen, T. D. Morris, T. Papenbrock, R. C. Pooser, D. J. Dean, and P. Lougovski. Cloud quantum computing of an atomic nucleus. *Phys. Rev. Lett.*, 120:210501, May 2018.

[10] Hal Finkel, Alexander McCaskey, Tobi Popoola, Dmitry Lyakh, and Johannes Doerfert. Really embedding domain-specific languages into c++, 2020.

[11] Tobias Gysi, Christoph Müller, Oleksandr Zinenko, Stephan Herhut, Eddie Davis, Tobias Wicky, Oliver Fuhrer, Torsten Hoefler, and Tobias Grosser. Domain-specific multi-level ir rewriting for gpu, 2020.

[12] J. Hines. Stepping up to summit. *Computing in Science Engineering*, 20(2):78–82, 2018.

[13] Ali JavadiAbhari, Shruti Patil, Daniel Kudrow, Jeff Heckey, Alexey Lvov, Frederic T. Chong, and Margaret Martonosi. Scaffcc: Scalable compilation and analysis of quantum programs. *Parallel Computing*, 45:2 – 17, 2015. Computing Frontiers 2014: Best Papers.

[14] Tian Jin, Gheorghe-Teodor Bercea, Tung D. Le, Tong Chen, Gong Su, Haruki Imai, Yasushi Negishi, Anh Leu, Kevin O'Brien, Kiyokuni Kawachiya, and Alexandre E. Eichenberger. Compiling onnx neural network models using mlir, 2020.

[15] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. pages 75–88, San Jose, CA, USA, Mar 2004.

[16] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: A compiler infrastructure for the end of moore's law, 2020.

[17] Alexander McCaskey, Eugene Dumitrescu, Dmitry Liakh, and Travis Humble. Hybrid programming for near-term quantum computing systems. In *2018 IEEE International Conference on Rebooting Computing (ICRC)*, pages 1–12. IEEE, 2018.

[18] Alexander J McCaskey, Dmitry I Lyakh, Eugene F Dumitrescu, Sarah S Powers, and Travis S Humble. XACC: a system-level software infrastructure for heterogeneous quantum–classical computing. *Quantum Science and Technology*, 5(2):024002, feb 2020.

[19] Alexander J. McCaskey, Zachary P. Parks, Jacek Jakowski, Shirley V. Moore, Titus D. Morris, Travis S. Humble, and Raphael C. Pooser. Quantum chemistry as a benchmark for near-term quantum computers. *npj Quantum Information*, 5(1):99, 2019.

[20] Benjamin C. A. Morrison, Andrew J. Landahl, Daniel S. Lobser, Kenneth M. Rudinger, Antonio E. Russo, Jay W. Van Der Wall, and Peter Maunz. Just another quantum assembly language (Jaqal). *arXiv e-prints*, page arXiv:2008.08042, August 2020.

[21] Thien Nguyen, Anthony Santana, Tyler Kharazi, Daniel Claudino, Hal Finkel, and Alexander McCaskey. Extending c++ for heterogeneous quantum-classical computing, 2020.

[22] T. J. Parr and R. W. Quong. Antlr: A predicated-ll(k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

[23] H. Peng and J. J. Shann. Translating openacc to llvm ir with spir kernels. In *2016 IEEE/ACIS 15th International Conference on Computer and Information Science (ICIS)*, pages 1–6, 2016.

[24] Robert S Smith, Michael J Curtis, and William J Zeng. A practical quantum instruction set architecture, 2016.

[25] softwareQ Inc. Benchmarking qasm source files, 2020. https://github.com/softwareQinc/staq/tree/main/demos/benchmarks.

[26] Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. Q#: Enabling scalable quantum computing and development with a high-level dsl. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*, RWDSL2018, New York, NY, USA, 2018. Association for Computing Machinery.