Department of Informatics

Technische Universität München

Master's Thesis

Heterogeneous Quantum Computing with OPENQASM 3.0

Omar Ibrahim

Department of Informatics

Technische Universität München

Master´s Thesis

Heterogeneous Quantum Computing with OPENQASM 3.0

Author:	Omar Ibrahim
1st examiner:	Univ.-Prof. Dr. Christian Mendl
Assistant advisor:	Msc. Martin Knudsen
Submission Date:	March 15th, 2023

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

March 15th, 2023 Omar Ibrahim

Acknowledgments

If someone helped you or supported you through your studies, this page is a good place to tell them how thankful you are.

"Everything should be made as simple as possible.
But to do that you have to master complexity."

-Butler Lampson

Abstract

This document will serve as an example to you, of how to use LaTeXto write your CSE Master's Thesis. It will have examples and recomendations, and hopefully a few laughs. Because this is the abstract, it will have to convince you that this template is something you want to use. It has been proven, that without using this template, writing your thesis will be much more dificult. The template is based on previous work, and has been improved apon and updated. The result of this template is a modern latex template that everyone can contribute to and use for their studies of CSE @ TUM.

Some more great abstract tips can be found here: Great Abstract tips

Contents

1 Introduction

Combination of quantum and classical computation as a form of heterogeneous computing allows us to solve very interesting problems. Many quantum algorithms and applications relies deeply on this combination. Whether near-time heterogeneous computing (i.e: Variational quantum eigensolver), or real-time (i.e: quantum error correction algorithms). While near-time heterogeneous computing is already achievable with current technology, albeit on a small scale, real-time heterogeneous computing is still a challenge, due to the issue of decoherence time of quantum systems, and the requirement for very low classical data transfer latency to account for this issue. In either of these cases, we can use a quantum processing unit (QPU) as an accelerator alongside a central processing unit (CPU), where the QPU can solve problems that QPUs excel in, and the CPU can solve problems that CPUs excel in, such as conditionals, loops...etc. This allows us to apply quantum algorithms that use a CPU as a coprocessor, but also to improve the performance of a QPU as in the case of quantum error correction. This concept of heterogeneity can also be extended to include other types of accelerators such as graphical processing units (GPUs), field programmable gate arrays (FPGAs), and application-specific integrated circuits (ASICs), to benefit from them in domains where they excel. In this thesis we provide a compilation framework for OPENQASM 3.0, a quantum programming language that supports many classical instructions, allowing for heterogeneous computing. Previous workflows have been provided towards this, notably the QCOR framework [4], which was a great inspiration for this work. QCOR, and after that this work, created dialects for lowering of OPENQASM 3.0 to MLIR, apply optimizations and then execute this intermediate representation on a QPU paired with a classical accelerator, or just execute it on a quantum simulator. In this work we approach this problem similarly, in a couple of aspects. Moreover, since the work on QCOR was discontinued, and the OPENQASM 3.0 spec was changing frequently, and we wanted to keep up with the latest changes in the spec at the time of development. The differences in our approach are the following:

1. Adhere to the latest spec as of time of development
2. Use the latest MLIR version
3. Make use of the latest MLIR dialects useful for our purposes.
4. Provide an explicit lowering step to a restricted quantum gates set, matching quantum gates available on desired hardware. We did this step as a proof of concept of the possibility to lower a generic quantum dialect to a restricted quantum dialect that

has a specific set of quantum gates. We used quantum gates supported by the Walter Meiner Institute (WMI)'s quantum computers, as an example.

In the next section we provide a background about the different topics relevant for this thesis. Followed by the body, where we showcase our approach to this compilation pipeline, including the different steps involved in MLIR generation, and finally we present some MLIR generated code and their corresponding stages in the compilation pipeline.

The different MLIR generation step ranges from showing the OPENQASM 3.0 code parse stage, to showing how we generate the corresponding MLIR code in the quantum dialect -among other MLIR dialects- to applying optimizations - predefined classical ones and explicitly defined quantum optimizations, to lowering this optimized MLIR code to have quantum gates only in the restricted quantum dialect, to optionally generate a classical analog of all the quantum operations for our quantum operations to be simulated, and finally to lowering the MLIR code to LLVM IR for execution by LLVM's Just-In-Time (JIT) compiler. In the last section, we showcase a live-run of our quantum simulator along with its corresponding quantum circuit, and provide some final closing remarks.

2 Background

2.1 Heterogeneous Computing

Heterogeneous computing refers to the use of multiple types of processing units or cores, with the aim of enhancing the overall performance and energy efficiency of the system. When mixing different types of processing units, we are able to utilize the strengths of each processing unit, while mitigating the weaknesses. These processing units can include central processing units (CPUs), but also other processing units - sometimes referred to as accelerators - like graphical processing units (GPUs), field programmable gate arrays (FPGAs), and application-specific integrated circuits(ASICs). Each of these processing units, can perform specific types of computations with varying degrees of efficiency. For our purposes we will focus on heterogeneous computing that combines classical computing using CPUs and quantum computing using quantum processing units (QPUs), quantum-classical heterogeneous computing. In this kind of heterogeneous computing we face the challenge of quantum decoherence. Quantum decoherence is the phenomenon where quantum systems interact with the environment, causing them to lose their quantum behavior and fallback to a classical one. These two challenges are different depending on whether we are applying near-time or real-time quantum computing. Real-time quantum computing requires calculations to complete within - or faster than - the qubit coherence time, while near-time quantum computing can tolerate high latency [2].

2.2 Quantum Computing Programming

Quantum computing programming languages are designed to help developers write and execute quantum algorithms on quantum computers. Although quantum computers are still in the early stages of development, there has been significant progress in the development of quantum computing programming languages. In this thesis we are only interested in OPENQASM 3.0. OPENQASM - short for Open quantum assembly - is a low level, quantum circuit-like language for quantum computing. With its newest edition - OPENQASM 3.0 - OPENQASM is no longer a strictly quantum programming language, but rather a Heterogeneous language. It allows for strictly quantum operations provided in OPENQASM 2.0, but also allows classical operations such as control operations (i.e: for and while loops, if statements...etc), extensive arithmetic operations, among many other operations. This heterogeneity of the language, being an open standard, and the large adoption by many scientists are all reasons for focusing on OPENQASM 3.0 in this thesis.

2.3 Compilation

2.3.1 Compilation Overview

Compilation in computer science is the process of translating a source language to a target language. Typically, this is done in an effort to translate a high-level language, to a lower level language, or to machine code to be executed by the computer. In this section we provide an overview of the compilation process, and the different stages of compilation, based on a more detailed description by [1, p. 4-11]. Normally, a compiler has two main parts, the frontend and the backend. The frontend applies the analysis phase of a compiler, while the backend applies the synthesis phase. The analysis phase, among possibly other steps, has three main steps:

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis

Lexical analysis, also called tokenization, is the process of splitting the input file into multiple tokens. If invalid tokens are found a tokenization error is generated. Syntax analysis, also known as parsing, is the process of verifying that a token stream is syntactically well-formed, and constructs a syntax tree from a valid set of tokens. This essentially verifies that the input program abides to our language's grammar syntax, and provides an intermediate representation for the next stage of analysis. The program that performs the parsing is called the parser. Finally, semantic analysis is the process of verifying that our program is sound in regard to our language's semantics. For example, in a strictly typed language an int variable can't be assigned a string. During semantic analysis, the syntax tree can be annotated to provide useful information (i.e: implicit type casting information) and produces what is known as an abstract syntax tree.

The next stage, the synthesis phase, consists also of three main phases:

- Intermediate Code Generation
- Program Optimization
- Code Generator

Intermediate code generation is the process of generating a low-level intermediate representation (IR) code. This IR usually has the benefit of having a small instruction set, and simple syntax and semantics. This makes the next stage, the program optimization stage, much simpler. Program optimization is the process of transformation from a less efficient program to a more efficient one. This can be done by removing redundant instructions, or instructions with more efficient ones. Intuitively, this stage is optional yet very essential. It can be split into two stages, machine independent and machine dependent optimization, where the former is done before target code generation, and the latter is done after. Finally, target code generation is the process of generating the target code to be executed on the machine.

2.3.2 ANTLR

ANTLR, which stands for Another Tool for Language Recognition, is a tool for generating parsers that can be utilized for different tasks relevant to components of the compilation process, and text analysis and processing in general. It is frequently utilized in developing programming languages, tools, and frameworks. It's often used in the creation of programming languages, tools, and frameworks [8]. ANTLR generates a parser from a grammar defined in a domain specific language (DSL), designed by ANTLR. ANTLR provides runtimes in multiple programming languages. You can choose which runtime you want your parser to be written in, and then you can construct and walk parse trees from input programs using ANTLR's runtime library.

2.3.3 LLVM and MLIR

LLVM

LLVM, which stands for Low Level Virtual Machine, is a compiler framework that enables comprehensive program analysis and transformation for various programs throughout their entire lifespan. It does this by providing high-level information to compiler transformations at compile-time, link-time, run-time, and in idle time between runs. LLVM uses a common, low-level code representation in static single assignment (SSA) form, which includes a simple, language-independent type-system that exposes the primitives commonly used to implement high-level language features. Additionally, LLVM features an instruction for typed address arithmetic and a simple mechanism that can be used to implement the exception handling features of high-level languages and `setjmp/longjmp` in C uniformly and efficiently[3].

2.3.4 MLIR

Multi-level intermediate representation (MLIR) is a novel approach for creating reusable and adaptable domain specific compilers. It is designed to provide an extensible common IR of SSA form, that can be used to represent multiple Domain specific languages (DSLs). This elevates the need for a new IR for each new DSL language, and allows for the reuse of existing IRs, especially within the same domain. Moreover, MLIR can be then lowered to LLVM IR, and utilize the LLVM tool-chain providing multiple lower-level optimizations and the ability to target different kinds of hardware accelerators with different architectures. The structure of MLIR's IR has few key abstractions - types, attributes, and operations. These main three abstractions can sufficiently extend the IR to its users needs. There are two more abstractions, regions and blocks, but they are extension of operations, where a block is a list of operations, and a region is a list of blocks. A Dialect refers to a unique namespace that encompasses a coherent set of operations, types, and attributes that are logically grouped together. MLIR provides a lot of public dialects catering to different domains, such as the `Arith` Dialect, the `Math` Dialect, the `MemRef` (Memory Reference) Dialect, and the Vector Dialect...etc. One can also create their own dialect for their own domain specific application. MLIR also provides a set of tools to help with the creation of new dialects, such as the MLIR `TableGen` tool, which allows for the creation of new dialects by writing a simple DSL. However, it should be noted that if one desires to lower down to LLVM IR, they will have to lower their dialect's operations in terms of the dialects provided by MLIR (include the LLVM IR). There are multiple approaches and defined classes that can aid with that.

2.3.5 Efforts to Leverage MLIR for heterogeneous quantum computing

Similar work to the one in this thesis was done by a team at Oak Ridge National Laboratory. QCOR, a tool created by this team, is a language extension specification for creating quantum kernels (in C++ and Python), and a compiler platform for heterogeneous computing [6][7]. To cater for other publicly available quantum DSL's, mainly OPENQASM 2.0 and OPENQASM 3.0, they created an MLIR dialect, Quantum Dialect, that allows for representation of the quantum operations in MLIR [4][5]. They also used MLIR's pattern rewriter to perform optimizations on the quantum IR. Finally, they lowered it to LLVM IR in a specific format that can be executed by their own runtime, that can target different quantum simulators and quantum hardware, alongside classical hardware. This effort was a great inspiration for this thesis, where we similarly created an MLIR dialect for quantum operations, and also shared some similarities in the quantum operations and the structure of lowering to the `quantum` dialect. However, there were also many areas that exhibited vast differences. Ranging from major differences (additions and omissions) in the operations that were defined in the quantum dialect, as well as the differentiation between a generic quantum dialect and a more restricted one, the way quantum operations were optimized, the construction of a quantum simulator written purely in MLIR, and in general differences in the way we lowered OPENQASM 3.0 all the way down to LLVM IR. Most noticeably as mentionend earlier we used the latest MLIR version allowing us to incorporate dialects like the `vector` dialect to be able to represent classical registers, for storing measurements or otherwise, which allows for vectorization and more efficient classical computation. Moreover, we used the latest OPENQASM 3.0 grammar defined at the time of our development, which led to differences in the way we lowered to the `quantum` dialect. Because of these differences in strategy, in the underlying library used and in OPENQASM 3.0 grammar, this project is naturally a complete rewrite, but was definitely vastly inspired by QCOR.

3 Compiling OpenQASM using MLIR

3.1 Overview

OPENQASM 3.0 compilation goes through a few phases. The first one being the frontend compilation phase. Namely, the OPENQASM 3.0 code is parsed and converted into a concrete syntax tree (CST), using ANTLR4 parser generator, to a c++ runtime target. The CST is traversed using the visitor pattern through generated visitor methods. These methods are then overridden to generate MLIR code. Generation of MLIR code for classical OPENQASM 3.0 instructions is quite straight forward, it is done using MLIR built-in dialects. However, for quantum instructions, we created our own quantum dialect, named `quantum`, introducing types for qubits, and a set of quantum operations. Some quantum optimizations are then performed on theses quantum operations, such as the elimination of identity gates. This is then lowered to another dialect called `restquantum`, which represents a restricted set of quantum gates. These set of gates are the ones supported by the Walter Meiner Institute(WMI) quantum computer, as of time of writing. This is of course not the only possible lowering step, and can be extended to support other hardware. Now, since for the scope of this thesis we were not able to test our code on the WMI quantum computer, we created a quantum simulator written purely in MLIR, which can be used to test our code. Therefore, the next step for our compilation pipeline is to convert operations of the `restquantum` dialect to other built-in MLIR dialects. Finally, we lower all the MLIR code to LLVM IR, and then to machine code by LLVM's Just-In-Time (JIT) compiler, to be executed. An overview of the stages of the pipeline are shown in Figure 3.1.

3.2 Dialects Created

The quantum dialect has been created to represent quantum operations, and qubits. Both types and operations are opaque. This means that they don't have an inherent interpretation and can be lowered dependent on the target quantum hardware or simulator. Some operations and types are shared between the two dialects. The class diagram for the two dialects are shown in Figures 3.2 and 3.3. In the next two subsections we will describe the operations and types created in each dialect.

3.2.1 Quantum Dialect

The main types created are:

- `quantum.Array` - represents a quantum register, which is a collection of qubits.
- `quantum.Qubit` - represents a qubit, which is a quantum bit.

Both of these types are also used in the `restquantum` dialect. The main operations created are:

- `quantum.qalloc` - allocates a quantum register of size `size`.
- `quantum.dealloc` - frees a quantum register.
- `quantum.qextract` - extracts a qubit from a quantum register, at index `index`.
- `quantum.mz` - measures a qubit in the standard basis.
- `quantum.print_global_vector` this is a utility function to print the global quantum state vector in our simulator.
- `quantum.SingleSimpleGate` - A class representing a single non-parametrized qubit gate. This is reused to create many other gates.
- `quantum.RotationGate` - A class representing a single qubit gate, parametrized with an angle operand. This is reused to create other rotation gates.
- `quantum.cx` - represents a controlled x gate.
- `quantum.gen_gate` - represents a generic gate, that takes in an arbitrary number of qubits, and arbitrary number of parameters.

3.2.2 RestQuantum Dialect

The `restquantum` dialect is a restricted set of quantum gates, which are supported by the WMI quantum computer. All these gates, except for the iSWAP gate, perform a rotation around an axis in the Bloch sphere, at certain angles. These gates are:

- `restquantum.rx90` - represents a rotation around the x axis by 90 degrees.
- `restquantum.rx180` - represents a rotation around the x axis by 180 degrees - equivalent to a Pauli X gate up to a global phase.
- `restquantum.ry90` - represents a rotation around the y axis by 90 degrees.
- `restquantum.ry180` - represents a rotation around the y axis by 180 degrees - equivalent to a Pauli Y gate up to a global phase.
- `restquantum.rz90` - represents a rotation around the z axis by 90 degrees.

- `restquantum.rz180` - represents a rotation around the z axis by 180 degrees - equivalent to a Pauli Z gate up to a global phase.
- `restquantum.sqrtx` - represents a square root of the Pauli X gate.
- `restquantum.iSWAP` - represents an iSWAP gate.

3.3 Pipeline

3.3.1 Frontend

Using the ANTLR4 grammar live spec (now stable) on OPENQSM 3.0's website we created a parser for a subset of OPENQASM 3.0. This subset can certainly be extended to fully support the language, but for the scope of this thesis we focused mainly on quantum gates, quantum measurement, some control flow operations - namely if statements and while loops -, some arithmetic (bitwise or otherwise) operations, logical operations, and main type declarations(quantum or classical). To do this lowering we used ANTLR's parser generator JAR file, which generated a c++, tokenizer(`qasmLexer`) and parser(`qasmParser`), along with a visitor class(`qasmBaseVisitor`) having methods for each of the grammar rules. We then overrode these methods in the class `Visitor` to generate MLIR code. The parsing starts by passing an input `.qasm` file to the `MLIRGenerator` class, which then tokenizes it with ANTLR's generated tokenizer, and parses the tokens with ANTLR's generated parser. Starting at the start rule of OPENQASM 3.0's grammar (which is `program`), the visitor traverse the CST and builds MLIR code upon visiting nodes of supported OPENQASM 3.0 constructs.

3.3.2 Translation to MLIR

To first inialize MLIR code building, we first need to create an `MLIRContext` pointer, and load all the desired dialects in the MLIR code building. This can be done as follows:

```
    // create an MLIR context pointer
    auto context = std::make_unique<MLIRContext>();

    // load quantum dialect
    context->loadDialect<quantum::QuantumDialect();

    // load restquantum dialect
    context->loadDialect<restquantum::RestrictedQuantumDialect>();

    // load vector dialect
    context->loadDialect<vector::VectorDialect>();

    // load arithmetic dialect
    context->loadDialect<arith::ArithDialect>();

    // ...etc
```

Next we initialize an Operation Builder, of class `OpBuilder`, which is used to build MLIR operations. We pass a source code (.qasm) location to the `ModuleOp` builder, to construct a module operation, which is the root of the MLIR code. We finally construct a `FuncOp` of name `main` with no arguments, and an Integer 64 (I64) return type, and append it to the module operation. This is an analogue to the main function in C/C++, and is the entry point of the MLIR code. We set this function to be the operation builder's insertion point, to ensure that created operation are appended to the function. After all OPENQASM to MLIR translations are done, we build a final `ReturnOp` operation with an `arith::ConstantOp` of value 0, and append it to the function. In the next sections we will discuss different compiler passes we applied to transform our MLIR code. Compiler passes are essentially traversal of an application's Directed Acyclic Graph (DAG), and applying a set of transformations to the graph.

3.3.3 Optimization

In this thesis optimization wasn't a main focus, but we did implement simple quantum optimization using quantum gate identities. These were only applied to the `quantum` dialect, and not the `restquantum` dialect. To do this we first defined our rewriter patterns simply in a TableGen file (`Optimize.td`). The defined patterns included the following identities:

These patterns are defined in the `Optimize.td` file as follows:

```
    // RZ(theta1) RZ(theta2) -> RZ(theta1 + theta2)
    def RzRz: Pat<(RzOp (RzOp $input, $theta2), $theta1),
        (RzOp $input, (Arith_AddFOp $theta1, $theta2,
        (createFastMathNone)))>;

    // HYH -> Y
    def HYH: Pat<(HOp (YOp (HOp $input))), (YOp $input)>;

    // HZH -> Z
    def HZH: Pat<(HOp (ZOp (HOp $input))), (ZOp $input)>;

    // HH -> I
    def HH: Pat<(HOp (HOp $input)), (IOp $input)>;

    // ... more patterns
```

To be able to apply these patterns in a simple manner, we employed them as parts of MLIR's `Conanicalizer` pass. To do so we have to set the `hasCanonicalizer` flag to 1 in the quantum dialect's definition, in its tablegen file. Next we need to implement MLIR `Dialect`'s class override method, `getCanonicalizationPatterns`. This is done as follows:

```
// #include ...
#include "quantum-mlir/Conversion/Optimize.h.inc"
using namespace mlir::quantum;

void QuantumDialect::getCanonicalizationPatterns(
    ::mlir::RewritePatternSet& results) const {
        populateWithGenerated(results);
}
```

The `populateWithGenerated` function defined in the generated `Optimize.h.inc` populates `results` with the patterns defined in `Optimize.td`. Finally, we initialize `MLIR`'s pass manager, and add MLIR's `Canonicalizer` pass to it. We also add the `DCE` pass, which is a dead code elimination pass, and common subexpression elimination (CSE) pass. The DCE pass is essential as the rewrite patterns defined in TableGen syntax only rewrites the last quantum gate (in direction of the quantum circuit), but leaves the remaining gates. By passing the `NoMemoryEffect` trait to our quantum gates, we allow MLIR's `DCE` pass to remove these remaining quantum gates as they are longer used by any other upcoming operations. This setting is predicated on the assumption that qubits have to be measured for the operations on them to be useful and not removed, which is fine for our purposes. Moreover, this wouldn't cause issue for entagled qubits because if either of them is measured both stay due to their joint use in a CNOT gate. Note that using Canoincalizer pass here is as it is more intended for canonicalization of syntax, however since optimization isn't the main goal of this thesis, this was a quick and simple way to test our idea, but in the future this should be refactored into its own pass.

3.3.4 Lowering to Restricted Quantum

In order to lower our code to the `restquantum` dialect, we use multiple quantum gate identities, some of which are shown in Figure ??.

To implement this lowering in MLIR, we create a `LowerToRestQuantum` pass. We add this in a `Passes.td` file where we defined all necessary passes, including relevant passes for the quantum simulator. We define it as follows:

```
def LowerToRestQuantum: Pass<"lower-to-restquantum", "ModuleOp"> {
    let summary =
        "Lower quantum dialect instructions to restrict"
      + "quantum dialect instructions";
    let constructor =
        "mlir::quantum::createLowerToRestQuantumPass()";
    let description = [{ }];
}
```

Then in a `Passes.h` file we declare a function `createLowerToRestQuantumPass` as follows:
`std::unique_ptr<OperationPass<ModuleOp>>`
`createLowerToRestQuantumPass();`
Finally, we implement the pass in a `LowerToRestQuantum.cpp` file. There we define a `LowerToRestQuantum` class, which inherits from `LowerToRestQuantumBase`, a class generated by MLIR's TableGen. There we set a `ConversionTarget target` object with all the quantum dialect gates as illegal operations. We also pass rewrite pattern classes, that we defined, to a `RewritePatternSet` object. These rewrite pattern classes transform all quantum gates in terms of `restquantum` gates.
We then call MLIR's `applyFullConversion` function, which applies all the defined patterns to the current MLIR code, and lowers all illegal operations to legal ones. Each of the rewriter pattern classes is defined as follows:

```
class ConvertQOp : public OpConversionPattern<quantum::QOp> {
    using OpConversionPattern::OpConversionPattern;
    LogicalResult matchAndRewrite(quantum::QOp op,
                    PatternRewriter& rewriter) const override {
        // Create relevant restquantum gates
        // to match a quantum identity.
        // Use rewriter.replaceOpWithNewOp
        // to replace the quantum gate
        // with needed operations.
        // Use rewriter.create
        // to add remaning necessary operations.
    }
};
```

We finally implement the `createLowerToRestQuantumPass` function, to inialize a `unique_ptr` of type `LowerToRestQuantum` and return it. To use this pass, like we did with the `Optimize` pass, we pass it to the `PassManager` object, and run it. We also add the `CSE` pass.

3.3.5 Quantum simulator (optional) pass

The quantum simulator pass is an optional pass that can be used to simulate the quantum circuit. It is declared similar to the `LowerToRestQuantum` pass, and defined in its own `QuantumSimulator.cpp` file. However, for convenience, we created a prior pass that does a forward declaration of all quantum allocations at the start of the MLIR code. Then, in the quantum simulator we sum the number of allocations multiplied by their qubit size, to get the totable number of qubits needed for a global state vector. We initialize a state vector of $||0_{n-1}...0_0\rangle$ where n is the number of qubits. Note, we assume the 0^{th} qubit is the right-most qubit in the bra-ket notation, thus accessing the 0^{th} element in the first qubit array accesses the right-most qubit, and accessing the last element in the last qubit array accesses the left-most qubit. We use MLIR's `VectorType` to represent the state vector, and we initialize a vector of shape $2^n \times 2$, where each element of the vector is a 2d vector denoting a complex number, this is because complex numbers are not supported yet as types for vector elements. Similar to what we did in the `LowerToRestQuantum` pass, we create different rewrite patterns to replace all quantum operations (gates, allocations, and measurements) with classical counterparts, manipulating with the initialized global state vector. For simulating quantum gates, instead of doing a matrix multiplication that would involve a lot of tensoring in the case of applying a gate to a subset of qubits. For example if we apply a gate to the first qubit of a quantum system that can be represented by a tensor product (non-entangled), the transformation would be as follows:

$$|\psi_{n-1}\rangle \otimes \psi_{n-2}\rangle \otimes ...|\psi_0\rangle \rightarrow I|\psi_{n-1}\rangle \otimes I|\psi_{n-2}\rangle \otimes ...X|\psi_0\rangle \tag{3.1}$$

To avoid this excessive tensoring, we reshape our vector to a tensor in a way that we can isolate the qubit we want to apply the gate to in its own dimension, then apply the matrix of this gate to the element in that dimension in the reshaped tensor, and finally reshape the tensor back to a vector of the original shape. For the case of the two-qubit gate $iSWAP$, we just swap the phases for the quantum states $|01\rangle$ and $|10\rangle$, and multiply them by i.

3.4 Examples - MLIR generated code

For different stages of the compilation pipeline, different MLIR code is generated. In this section we showcase the MLIR code generated by the compiler at different stages. We start with the following OPENQASM code:

```
OPENQASM 3.0;
qubit[2] q;
H q[0];
Y q[0];
H q[0];
X q[1];
```

```
    PRINT_GLOBAL_VECTOR();
    bit[2] c;
    c[0] = measure q[0];
    c[1] = measure q[1];
    print(c);
```

This code declares two qubits in a qubit array identified by `q`, applies a Hadamard gate to the first qubit, a Pauli-Y gate to the first qubit, followed by another Hadamard gate, and finally an X gate to the second qubit. Then it prints the global state vector, and finally measures the first and second qubits, and prints the result. Note that both `PRINT_GLOBAL_VECTOR` and `print` are custom functions that we defined as utility functions. The former was defined as part of the `quantum` dialect, and the latter is an addition to OPENQASM 3.0 grammar.

3.4.1 Translating to the quantum dialect

The first stage of the compilation pipeline is translating the OPENQASM code to the `quantum` dialect, as it was shown earlier in the 3.1 section, as well as in figure 3.1. An internal MLIR structure is built at that stage, but the corresponding IR assembly can be printed to show:

```
module {
  func.func @main() -> i64 {
    %0 = "quantum.qalloc"() {name = "q", offset = 0 : i32, size = 2 : i64} :
     () -> !quantum.QubitArray

    %c0_i64 = arith.constant 0 : i64

    %1 = "quantum.qextract"(%0, %c0_i64) :
    (!quantum.QubitArray, i64) -> !quantum.Qubit

    %2 = "quantum.HOp"(%1) : (!quantum.Qubit) -> !quantum.Qubit

    %3 = "quantum.YOp"(%2) : (!quantum.Qubit) -> !quantum.Qubit

    %4 = "quantum.HOp"(%3) : (!quantum.Qubit) -> !quantum.Qubit

    %c1_i64 = arith.constant 1 : i64

    %5 = "quantum.qextract"(%0, %c1_i64) :
    (!quantum.QubitArray, i64) -> !quantum.Qubit

    %6 = "quantum.XOp"(%5) : (!quantum.Qubit) -> !quantum.Qubit
    "quantum.print_global_vector"() : () -> ()

    %cst = arith.constant dense<false> : vector<2xi1>

    %7 = "quantum.mz"(%4) : (!quantum.Qubit) -> i1

    %c0 = arith.constant 0 : index

    %8 = vector.insertelement %7, %cst[%c0 : index] : vector<2xi1>

    %9 = "quantum.mz"(%6) : (!quantum.Qubit) -> i1

    %c1 = arith.constant 1 : index

    %10 = vector.insertelement %9, %8[%c1 : index] : vector<2xi1>

    vector.print %10 : vector<2xi1>

    %c0_i64_0 = arith.constant 0 : i64

    return %c0_i64_0 : i64
  }
}
```

In the above code, we can see the `quantum` dialect operations that were generated from the OPENQASM code, along with some standard MLIR operations. Firstly, an MLIR module is created, and a `main` function is defined in it, which returns an integer of value 0 in the end. The `quantum.qalloc` operation allocates a qubit array of size 2 with identifier `q`, and assigns it to the variable `q`. Then, the `quantum.qextract` operation extracts the first qubit from the qubit array, and assigns it to the variable `q[0]`. Similarly, the second qubit is extracted and assigned to the variable `q[1]`. Then, the `quantum.HOp` operation is applied to the first qubit, followed by the `quantum.YOp` operation, and then another `quantum.HOp` operation. The `quantum.XOp` operation is applied to the second qubit. The `quantum.print_global_vector` operation prints the global state vector, as explained earlier. The `quantum.mz` operation measures the qubit in the computational basis, and returns the result as a boolean. The `vector.insertelement` operation inserts the result of the measurement into the vector of bits, and the `vector.print` operation, translated from the defined `print` function, prints the vector of bits.

3.4.2 Applying Optimizations

The next stage of the compilation pipeline is applying optimizations on the code. This utilizes a few MLIR defined optimization passes, as well as the quantum identifies defined in the `QuantumOpt` pass, as explained earlier in the 3.3.3 section. Here is the MLIR code after applying the optimizations:

```
module {
  func.func @main() -> i64 {
    %c1 = arith.constant 1 : index

    %c0 = arith.constant 0 : index

    %cst = arith.constant dense<false> : vector<2xi1>

    %c1_i64 = arith.constant 1 : i64

    %c0_i64 = arith.constant 0 : i64

    %0 = "quantum.qalloc"() {name = "q", offset = 0 : i32, size = 2 : i64} :
    () -> !quantum.QubitArray

    %1 = "quantum.qextract"(%0, %c0_i64) :
    (!quantum.QubitArray, i64) -> !quantum.Qubit

    %2 = "quantum.YOp"(%1) : (!quantum.Qubit) -> !quantum.Qubit

    %3 = "quantum.qextract"(%0, %c1_i64) :
    (!quantum.QubitArray, i64) -> !quantum.Qubit

    %4 = "quantum.XOp"(%3) : (!quantum.Qubit) -> !quantum.Qubit

    "quantum.print_global_vector"() : () -> ()

    %5 = "quantum.mz"(%2) : (!quantum.Qubit) -> i1

    %6 = vector.insertelement %5, %cst[%c0 : index] : vector<2xi1>

    %7 = "quantum.mz"(%4) : (!quantum.Qubit) -> i1

    %8 = vector.insertelement %7, %6[%c1 : index] : vector<2xi1>

    vector.print %8 : vector<2xi1>

    return %c0_i64 : i64
  }
}
```

Applying the optimizations, we can see that the `quantum.HOp` operations are removed by the identity $HYH \equiv Y$(up to a global phase).

3.4.3 Lowering to restquantum

In the next stage of the compilation pipeline, we lower our quantum code to the `restquantum` dialect, per the identities defined in the figure ??. Here is the MLIR code after lowering to `restquantum`:

```
module {
  func.func @main() -> i64 {
    %c1 = arith.constant 1 : index

    %c0 = arith.constant 0 : index

    %cst = arith.constant dense<false> : vector<2xi1>

    %c1_i64 = arith.constant 1 : i64

    %c0_i64 = arith.constant 0 : i64

    %0 = "quantum.qalloc"() {name = "q", offset = 0 : i32, size = 2 : i64} :
     () -> !quantum.QubitArray

    %1 = "quantum.qextract"(%0, %c0_i64) :
     (!quantum.QubitArray, i64) -> !quantum.Qubit

    %2 = "restquantum.ry180"(%1) :
    (!quantum.Qubit) -> !quantum.Qubit

    %3 = "quantum.qextract"(%0, %c1_i64) :
    (!quantum.QubitArray, i64) -> !quantum.Qubit

    %4 = "restquantum.rx180"(%3) :
     (!quantum.Qubit) -> !quantum.Qubit

    "quantum.print_global_vector"() : () -> ()
    %5 = "quantum.mz"(%2) : (!quantum.Qubit) -> i1

    %6 = vector.insertelement %5, %cst[%c0 : index] : vector<2xi1>

    %7 = "quantum.mz"(%4) : (!quantum.Qubit) -> i1

    %8 = vector.insertelement %7, %6[%c1 : index] : vector<2xi1>

    vector.print %8 : vector<2xi1>

    return %c0_i64 : i64
  }
}
```

3.4.4 Quantum Simulator

The next stage of the compilation pipeline is to convert all quantum operations into classical ones that can be executed by a quantum simulator. Here is the MLIR code after lowering to the simulator:

```
module {
  func.func @main() -> i64 {
    %c1 = arith.constant 1 : index
    %c0 = arith.constant 0 : index
    %cst = arith.constant dense<false> : vector<2xi1>
    %c0_i64 = arith.constant 0 : i64
    %cst_0 = arith.constant dense
    <[1.000000e+00, 0.000000e+00, 0.000000e+00, 0.000000e+00,
     0.000000e+00, 0.000000e+00, 0.000000e+00, 0.000000e+00]> : vector<8xf64>
    %0 = vector.shape_cast %cst_0 : vector<8xf64> to vector<4x2xf64>
    %alloc = memref.alloc() : memref<vector<4x2xf64>>
    memref.store %0, %alloc[] : memref<vector<4x2xf64>>
    %1 = memref.load %alloc[] : memref<vector<4x2xf64>>
    %2 = vector.shape_cast %1 : vector<4x2xf64> to vector<2x2x1x2xf64>
    %cst_1 = arith.constant 0.000000e+00 : f64
    %cst_2 = arith.constant -1.000000e+00 : f64
    %3 = vector.extract %2[0, 0] : vector<2x2x1x2xf64>
    %4 = vector.extract %2[0, 1] : vector<2x2x1x2xf64>
    %5 = vector.extract %4[0, 0] : vector<1x2xf64>
    %6 = vector.extract %4[0, 1] : vector<1x2xf64>
    %7 = complex.create %5, %6 : complex<f64>
    %8 = complex.create %cst_1, %cst_2 : complex<f64>
    %9 = complex.mul %7, %8 : complex<f64>
    %10 = complex.re %9 : complex<f64>
    %11 = complex.im %9 : complex<f64>
    %12 = vector.insert %10, %4 [0, 0] : f64 into vector<1x2xf64>
    %13 = vector.insert %11, %12 [0, 1] : f64 into vector<1x2xf64>
    %14 = vector.insert %13, %2 [0, 0] :
    vector<1x2xf64> into vector<2x2x1x2xf64>
    %15 = vector.extract %3[0, 0] : vector<1x2xf64>
    %16 = vector.extract %3[0, 1] : vector<1x2xf64>
    %17 = complex.create %15, %16 : complex<f64>
    %cst_3 = arith.constant 1.000000e+00 : f64
    %18 = complex.create %cst_1, %cst_3 : complex<f64>
    %19 = complex.mul %17, %18 : complex<f64>
    %20 = complex.re %19 : complex<f64>
    %21 = complex.im %19 : complex<f64>
    %22 = vector.insert %20, %3 [0, 0] : f64 into vector<1x2xf64>
    %23 = vector.insert %21, %22 [0, 1] : f64 into vector<1x2xf64>
    %24 = vector.insert %23, %14 [0, 1] : vector<1x2xf64> into vector<2x2x1x2
    %25 = vector.extract %24[1, 0] : vector<2x2x1x2xf64>
    %26 = vector.extract %24[1, 1] : vector<2x2x1x2xf64>
    %27 = vector.extract %26[0, 0] : vector<1x2xf64>
    %28 = vector.extract %26[0, 1] : vector<1x2xf64>
    %29 = complex.create %27, %28 : complex<f64>
    %30 = complex.mul %29, %8 : complex<f64>
    %31 = complex.re %30 : complex<f64>
    %32 = complex.im %30 : complex<f64>
    %33 = vector.insert %31, %26 [0, 0] : f64 into vector<1x2xf64>
    %34 = vector.insert %32, %33 [0, 1] : f64 into vector<1x2xf64>
    %35 = vector.insert %34, %24 [1, 0] :
    vector<1x2xf64> into vector<2x2x1x2xf64>
    %36 = vector.extract %25[0, 0] : vector<1x2xf64>
    %37 = vector.extract %25[0, 1] : vector<1x2xf64>
    %38 = complex.create %36, %37 : complex<f64>
    %39 = complex.mul %38, %18 : complex<f64>
    %40 = complex.re %39 : complex<f64>
    %41 = complex.im %39 : complex<f64>
    %42 = vector.insert %40, %25 [0, 0] : f64 into vector<1x2xf64>
    %43 = vector.insert %41, %42 [0, 1] : f64 into vector<1x2xf64>
    %44 = vector.insert %43, %35 [1, 1] : vector<1x2xf64> into vector<2x2x1x2
    %45 = vector.shape_cast %44 :
     vector<2x2x1x2xf64> to vector<4x2xf64>
    memref.store %45, %alloc[] : memref<vector<4x2xf64>>
    %46 = memref.load %alloc[] : memref<vector<4x2xf64>>
    %47 = vector.shape_cast %46 :
    vector<4x2xf64> to vector<1x2x2x2xf64>
    %48 = vector.extract %47[0, 0] : vector<1x2x2x2xf64>
    %49 = vector.extract %47[0, 1] : vector<1x2x2x2xf64>
    %50 = vector.insert %49, %47 [0, 0] :
    vector<2x2xf64> into vector<1x2x2x2xf64>
    %51 = vector.insert %48, %50 [0, 1] :
    vector<2x2xf64> into vector<1x2x2x2xf64>
    %52 = vector.shape_cast %51 : vector<1x2x2x2xf64> to vector<4x2xf64>
    memref.store %52, %alloc[] : memref<vector<4x2xf64>>
    %53 = memref.load %alloc[] : memref<vector<4x2xf64>>
    vector.print %53 : vector<4x2xf64>
    %54 = memref.load %alloc[] : memref<vector<4x2xf64>>
    %55 = vector.extract %54[0, 0] : vector<4x2xf64>
    %56 = vector.extract %54[0, 1] : vector<4x2xf64>
    %57 = complex.create %55, %56 : complex<f64>
    %58 = complex.abs %57 : complex<f64>
    %59 = arith.mulf %58, %58 : f64
    %60 = arith.addf %cst_1, %59 : f64
    %61 = vector.extract %54[1, 0] : vector<4x2xf64>
    %62 = vector.extract %54[1, 1] : vector<4x2xf64>
    %63 = complex.create %61, %62 : complex<f64>
    %64 = complex.abs %63 : complex<f64>
    %65 = arith.mulf %64, %64 : f64
    %66 = arith.addf %cst_1, %65 : f64
    %67 = vector.extract %54[2, 0] : vector<4x2xf64>
    %68 = vector.extract %54[2, 1] : vector<4x2xf64>
    %69 = complex.create %67, %68 : complex<f64>
    %70 = complex.abs %69 : complex<f64>
    %71 = arith.mulf %70, %70 : f64
    %72 = arith.addf %60, %71 : f64
    %73 = vector.extract %54[3, 0] : vector<4x2xf64>
    %74 = vector.extract %54[3, 1] : vector<4x2xf64>
    %75 = complex.create %73, %74 : complex<f64>
    %76 = complex.abs %75 : complex<f64>
    %77 = arith.mulf %76, %76 : f64
    %cst_4 = arith.constant 0.9492427099970786 : f64
    %78 = arith.cmpf ole, %cst_4, %72 : f64
    %false = arith.constant false
    %true = arith.constant true
    %79 = arith.select %78, %false, %true : i1
    %80 = vector.insertelement %79, %cst[%c0 : index] : vector<2xi1>
    %81 = arith.addf %60, %65 : f64
    %82 = arith.addf %cst_1, %71 : f64
    %cst_5 = arith.constant 0.14277997637369078 : f64
    %83 = arith.cmpf ole, %cst_5, %81 : f64
    %84 = arith.select %83, %false, %true : i1
    %85 = vector.insertelement %84, %80[%c1 : index] : vector<2xi1>
    vector.print %85 : vector<2xi1>
    return %c0_i64 : i64
  }
}
```

3.4.5 Lowering to LLVM

Finally, we lower the `restquantum` dialect to LLVM, and generate the LLVM IR code.

3.5 Simulation

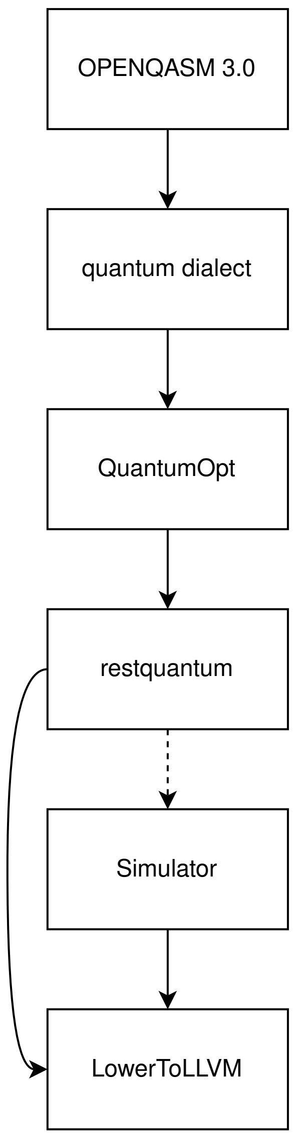


Figure 3.1: Compilation Pipeline Stages

Note: the dotted line is to show that the quantum simulator is an optional lowering step and can be circumvented when deploying on real quantum hardware.

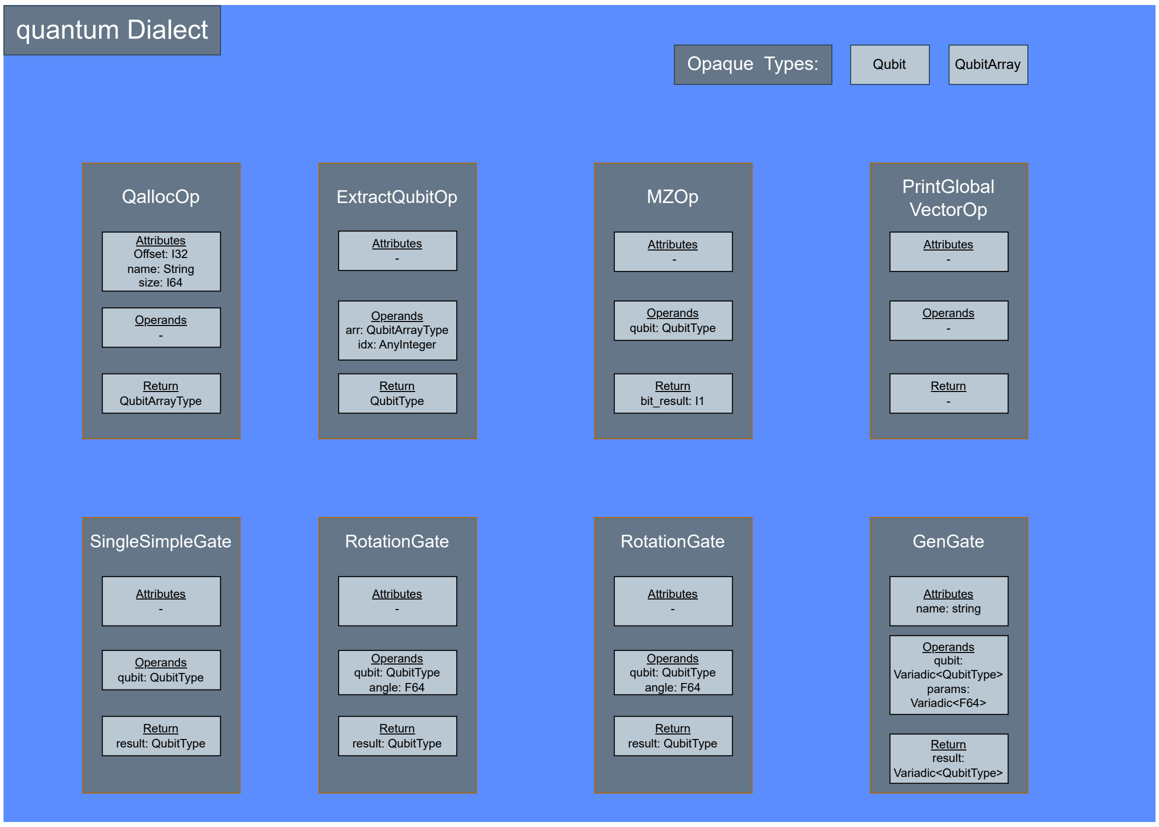


Figure 3.2: Class diagram of quantum dialect

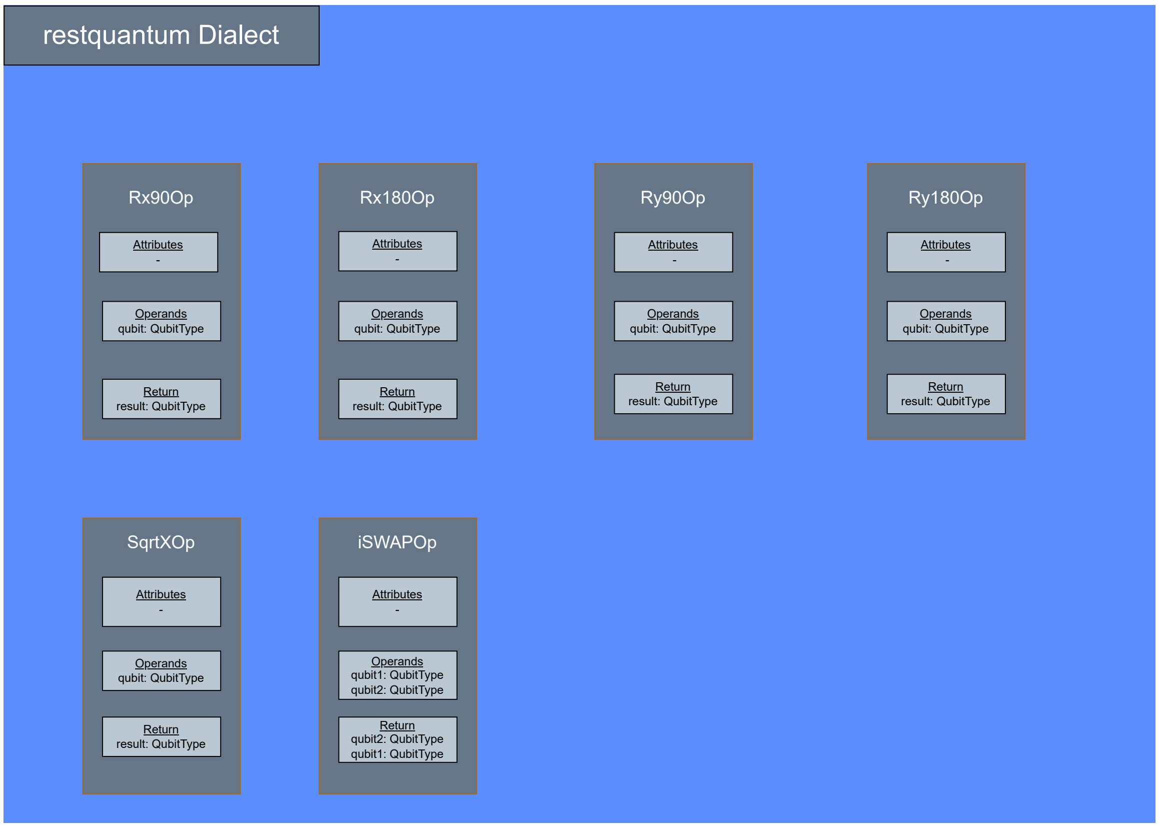
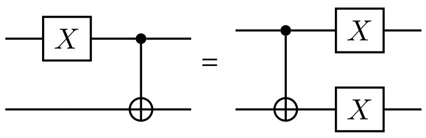


Figure 3.3: Class diagram of restquantum dialect



4 Conclusion

Due to limitations faced by the quantum computing team in WMI, we were not able to execute the quantum heterogeneous generated MLIR code on their hardware. However, as also another proof of concept, we created a quantum simulator written purely in MLIR to test out our compilation pipeline. This was also useful to showcase the possibility to write a quantum simulator purely in MLIR, which could have superior performance than calling external c++ (or other languages) code from within our MLIR code.

Bibliography

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[2] International Business Machines Corporation (IBM). Rethinking quantum systems for faster, more efficient computation. https://research.ibm.com/blog/near-real-time-quantum-compute.

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

[4] Alexander McCaskey and Thien Nguyen. A mlir dialect for quantum assembly languages, 2021.

[5] Alexander J. McCaskey, Thien Nguyen, Eugene Dumitrescu, Dmitry Liakh, and Anthony Santana. Quantum-classical compilation with the mlir. 2021.

[6] Tiffany M. Mintz, Alexander J. McCaskey, Eugene F. Dumitrescu, Shirley V. Moore, Sarah Powers, and Pavel Lougovski. QCOR: A language extension specification for the heterogeneous quantum-classical model of computation. *CoRR*, abs/1909.02457, 2019.

[7] Thien Nguyen and Alexander J. McCaskey. Extending python for quantum-classical computing via quantum just-in-time compilation, 2021.

[8] Terence Parr and Sam Harwell. Another tool for language recognition. https://www.antlr.org/about.html.