Quantum Science and Technology

PAPER • OPEN ACCESS

An updated LLVM-based quantum research compiler with further OpenQASM support

To cite this article: Andrew Litteken *et al* 2020 *Quantum Sci. Technol.* **5** 034013

Quantum Science and Technology

OPEN ACCESS

RECEIVED
25 November 2019

REVISED
13 April 2020

ACCEPTED FOR PUBLICATION
22 April 2020

PUBLISHED
28 May 2020

PAPER

An updated LLVM-based quantum research compiler with further OpenQASM support

Andrew Litteken[1,3], Yung-Ching Fan[1,3], Devina Singh[2], Margaret Martonosi[2] and Frederic T Chong[1]

[1] Department of Computer Science University of Chicago, United States
[2] Department of Computer Science Princeton University, United States
[3] Both authors contributed equally to this work.

E-mail: litteken@uchicago.edu, clairefan@uchicago.edu, devinas@alumni.princeton.edu, mrm@princeton.edu and chong@cs.uchicago.edu

Keywords: quantum computing, programming languages, compilers, quantum compiler, Scaffold, LLVM, quantum programming languages

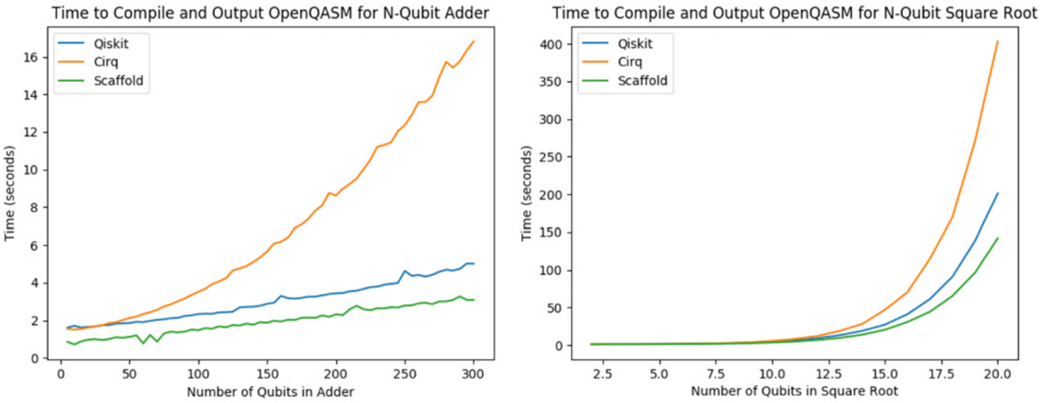
Abstract

Quantum computing is a rapidly growing field with the potential to change how we solve previously intractable problems. Emerging hardware is approaching a complexity that requires increasingly sophisticated programming and control. Scaffold is an older quantum programming language that was originally designed for resource estimation for far-future, large quantum machines, and ScaffCC is the corresponding LLVM-based compiler. For the first time, we provide a full and complete overview of the language itself, the compiler as well as its pass structure. While previous works Abhari *et al* (2015 *Parallel Comput.* **45** 2–17), Abhari *et al* (2012 Scaffold: quantum programming language https://cs.princeton.edu/research/techreps/TR-934-12), have piecemeal descriptions of different portions of this toolchain, we provide a more full and complete description in this paper. We also introduce updates to ScaffCC including conditional measurement and multidimensional qubit arrays designed to keep in step with modern quantum assembly languages, as well as an alternate toolchain targeted at maintaining correctness and low resource count for noisy-intermediate scale quantum (NISQ) machines, and compatibility with current versions of LLVM and Clang. Our goal is to provide the research community with a functional LLVM framework for quantum program analysis, optimization, and generation of executable code.

1. Introduction

Quantum programming languages and compilers will play a critical role in achieving practical quantum computation [3]. In this paper, we detail Scaffold [2], a programming language built for hybrid classical and quantum computing, its architecture and future development. Both the Scaffold compiler infrastructure, ScaffCC, and benchmarks have been used broadly in academic research projects [4–6]. However, there does not exist a single description of the entire Scaffold toolchain. This paper documents the overall infrastructure as well as our tool's most recent updates.

Scaffold exists among a wide set of different programming languages built for quantum programming [3]. As in to classical programming, there are declarative, functional and imperative solutions for quantum programming. Languages, Quipper [7], Microsoft's Q# [8] and QuaFL [9], are functional languages, and, in the case of Quipper and Q# are built on existing languages, simplifying language implementation and initial costs. These functional languages benefit from clearly defined type systems, in the case of Q#, with extensions for quantum gates and guarantees about certain states. Unfortunately, Q# only offers an interface to their own simulator, and not to any sort of quantum assembly languages [10]. Quipper provides similar benefits for checking correctness, but also fails to provide and interface to quantum assembly languages, instead using its own simulator [11]. Declarative frameworks, such as Qiskit [12] and Cirq [13] which are



(a) Increase in time for N-Qubit Adder

(b) Increase in time for N-Qubit Square Root

Figure 1. Framework comparison: an example of how increasing the number of qubits in a circuit stresses Cirq and Qiskit.

both built on top of Python, provide a decent foothold in developing smaller scale quantum circuits. However, these frameworks face problems as the size of the program grows. These frameworks tend to be more focused on how circuits will work in current simulators and on small scale quantum computers, and while they have good support for demonstrating noise models and connectivity between qubits, they struggle to handle larger programs due to the computational overhead associated with increased circuit size. For example, building and analyzing circuits of 100 or greater qubits can take a significant amount of time. This can be seen in figure 1, where building, decomposing and outputting the OpenQASM for an adder and a square root circuit can take significantly longer for Cirq and Qiskit when compared to Scaffold. Finally, QCL and Scaffold follow the imperative model. QCL and Scaffold are both based on C, with QCL relying on matrices and matrix operations to express quantum data and operations. On the other hand, Scaffold was designed to scale and takes a much simpler approach, using arrays of qubits and displaying the operations as gates—a powerful strategy in terms of both program optimization and analysis. Furthermore, since the heuristics of an imperative language are straight forward, it is easy to add further compiler optimizations to the existing compilation framework, which becomes increasingly important as programs grow in size. With these strategies, we can create a much more efficient program with increased compiler scalability and flexibility.

We will give a background introduction of Scaffold, and a thorough examination of the structure of the compiler that gives Scaffold, as a programming language, its capabilities. Finally, we describe an extensive update in which we added more support for compilation to OpenQASM [14] and a compiler restructuring which better utilizes the power of the LLVM infrastructure to give the developers a simpler and more accessible way to enhance this language based to their needs. We also describe how NISQ oriented tools have been added to supplement with the main compiler. The current distribution for Scaffold can be found at https://github.com/epiqc/ScaffCC, which also includes detailed technical aspects, such as how to add a pass, an in-depth user manual, and build instructions.

2. Scaffold history and language features

Several features distinguish Scaffold from other quantum computing languages. Besides supporting several different target quantum assembly languages, Scaffold provides critical quantum computing functionalities to allow for programming in the quantum space.

2.1. History

Scaffold was built with an emphasis on expressing complex quantum algorithms in a concise manner. Complicated components of quantum algorithms, such as a quantum oracle or repeatable sets of quantum circuitry, would be easily expressible and easy to understand. Scaffold programs did not have a strong emphasis on running executable quantum programs, but rather, focused on creating a framework where the program could be analyzed and manipulated. Through the integration of different quantum components in a modular structure, it would be easy to create, model, and manipulate larger programs with many qubits. Additionally, by leveraging the analysis capabilities of Clang and LLVM, the ScaffCC (Scaffold compiler) framework was designed to handle large programs. ScaffCC was designed to perform optimizations under the assumption that most, if not all, program inputs would be known at compilation time.

Since the language was meant to express algorithms in a concise way, the compiler was designed to be flexible for the purpose optimizing and analyzing the program, with the aim of finding, and potentially decreasing, resource usage. This was done through different optimizations, such as inserting uncomputation into the program to save ancilla qubits, or attempting to take advantage of natural commutative circuit features and removing operations accordingly. Like most compiler optimizations, the goal was to make these programs less resource intensive; however, since there was no machine to run programs on, the goal was not for these programs to be run directly on machines. While the Scaffold compiler has the ability to generate quantum assembly code, this feature was not emphasized. For example, quantum error correction will require conditional execution on measurement, which was not originally supported by Scaffold in quantum assembly generation. Beyond conditional measurement on a single qubit, quantum assembly languages support arraywise qubit measurement as well. We remedy such shortcomings in this paper by updating our code generation for IBM's OpenQASM quantum assembly specification.

Scaffold was developed as a C-style language so that its abstraction was not too far removed from the hardware and quantum circuitry that has been traditionally used to describe quantum programs. This assisted in making the translation from Scaffold to quantum assembly more viable and acted as a stepping stone to programs that could run on future quantum machines. In line with the attempts to integrate Scaffold into a C-style language, it was built on top of Clang, allowing for the C syntax to be leveraged in a new quantum context. While this allows for any aspect of C-code to be used, only the basic aspects needed for quantum algorithms, such as simple conditionals, for loops, and declarations have been thoroughly tested.

Previously, Scaffold circuits relied on simple hardware gates. However, here are many benefits to control flow constructs that can be used to more easily design circuits. As a simple example, the addition of for-loops and conditionals allow for classical components to alter program results. Additionally, control flow constructs, specifically module calls, allow for the removal of several different boilerplate operations. By taking advantage of these constructs, Scaffold is able to minimize arithmetic operations, reducing the amount of computation associated with each operation in a quantum circuit and making it easier to ensure that the results are correct. Furthermore, the insertion of reversible computations into Scaffold programs with simple function annotations, greatly improves the ability to design and test algorithms that require these operations.

As quantum computing has evolved into the NISQ-era, Scaffold too is evolving to meet the necessary requirements of new quantum algorithms, especially those that need to be run on actual quantum computers (with eventual error correction functionality planned for the future). This has meant finding any deficiencies in the current quantum assembly code generation, improving the existing optimization algorithms, and developing new algorithms to ensure that Scaffold is able to both maintain its current functionality as well as work on a smaller scale for this new era of quantum computing.

2.2. Syntax

A critical feature of Scaffold is its similarity to the C family language, which is widely used in classical programming, in terms of program structure, standard library availability and use of classical data types. Quantum computing components in Scaffold also mimic C-style variable declarations and function calls. This make the implementation of quantum computing algorithms a smoother, more familiar programming experience. Scaffold developers are also able to create and add modules and gates based on their needs for research or practical use and are not restricted to the built in library.

2.2.1. Data types

Quantum registers. Scaffold provides several native data types for quantum programming. In Scaffold, we have `qbit` and `abit` quantum registers, each with a specific use. As in C and C++ arrays, a `qbit` or `abit` variable indicates a single qubit, but can be adapted with bracket notation, as in example 4 to represent an one-dimensional or multidimensional array of qubits. These data types act in the same way except that an `abit` is provisioned for representing an ancilla quantum register. These qubits are only used during a part of computation process, whereas the `qbit` is used for registers that hold data for the computation throughout the entire circuit. While not an explicit requirement, separate `qbit` and `abit` data types allow programmers to differentiate which registers hold the data that is integral to the circuit from the registers that can be used as a 'scratchpad' of sorts. Later on, we see how that this has advantages when performing optimizations.

Classical registers. Apart from the `qbit` and `abit`, Scaffold provides the `cbit` data type to represent classical bits which are used to store the result of measurement of quantum registers. The `cbit` is classical i.e. it can ultimately only hold a 0 or 1 value in each register, making it different from the quantum focused

Example 1. All-gates.scaffold.

```
int main (){
    qbit q[3];
    cbit c[1];

    X (q[0]);
    Y (q[0]);
    Z (q[0]);
    H (q[0]);
    T (q[0]);
    S (q[0]);
    Tdag (q[0]);
    Sdag (q[0]);
    Rx (q[0], 3.14159);
    Ry (q[0], 3.14159);
    Rz (q[0], 3.14159);
    PrepX (q[0], 0);
    PrepZ (q[0], 0);
    c[0] = MeasX (q[0]);
    c[0] = MeasZ (q[0]);

    CNOT (q[0], q[1]);
    Toffoli (q[0], q[1], q[2]);
    Fredkin (q[0], q[1], q[2]);

    return 0;
}
```

qbits and abits. Since it serves a critical function in quantum computing process as the measurement destination, we give it this special classification.

C data types. Scaffold also allows programmers declare classical data types to do classical computation or pass as arguments in classical/quantum functions. Valid primitive data types cover those commonly used in C language, such as int, float, char. Casting between different data type is also viable in Scaffold with the intention of providing C familiarity, but is constrained to classical data types.

2.3. Quantum gates

Besides quantum registers, another unique feature in Scaffold is a library of built-in quantum gates which are specific actions on quantum registers to perform quantum computations. A quantum gate can take both quantum and classical type parameters. Such an example are the rotation gates Rx, Ry, and Rz which take a quantum register and a value to rotate the register by around a specific axis. Example 1 lays out all the built-in gates supported by the Scaffold compiler, which incorporates the mostly common used quantum gates. In the future, as more control over quantum hardware is developed, we expect to develop more complex gates. Scaffold also provides flexibility to users who wish to create and define a new gate by adding a module to the Scaffold infrastructure.

Built-in Gates. Pauli-X gate, Pauli-Y gate, Pauli-Z gate, Hadamard gate, T gate (phase shift gate where $\Phi = \pi/4$), S gate (phase shift gate where $\Phi = \pi/2$), T dagger gate, S dagger gate, Arbitrary rotation gate from X, Y, Z axis, single bit preparation gate (initialize to 0), Measurement gate, CNOT gate, Toffoli gate and Fredkin (Control Swap) gate.

Measurement. Measurement in quantum programs allow us to determine the state of the quantum bits within a program since they cannot be observed directly. Scaffold includes the ability to both measure a single qubit value, or multiple qubit values at once. When performing these measurements, the output is a cbit, or array of cbits that matches the dimensions of the qubits measured by the gate. A simple example of a Scaffold program that performs examples of both of these styles of measurement after being run through an H gate are shown in example 2. It should be noted that we must 'dereference' both the qbit and cbit arrays to perform this operation. This dereferencing is performed due to our handling of the underlying LLVM instructions. If dereferencing did not occur, it is much more difficult to discern the difference between measuring a single bit versus measuring the entire array. This design detail allows for much simpler compilation strategies later on.

When measuring the qubit, a 1 or 0 is written into each cbit. It is important to note that once these classical bits are written to once, they cannot be written to again due to the fact that it will overwrite the data in the quantum circuit.

Example 2. Measurement on quantum datatypes to classical datatypes.

```
int main (){
    qbit one_qbit[1];
    cbit one_cbit[1];

    H(one_qbit);
    one_cbit[0] = MeasZ(one_qbit[0]);

    qbit many_qbits[4];
    cbit many_cbits[4];

    H(many_qbits);
    *many_cbits = MeasZ(*many_qbits);

    return 0;
}
```

Example 3. An example of using measurement in conditional operations in Scaffold.

```
int main(){
    qbit one_qbit[1];
    cbit one_cbit[1];
    H(one_bit);

    one_cbit[0] = MeasZ(one_qbit);
    if(one_cbit[0] == 1) X(one_qbit);

    qbit many_qbits[3];
    cbit many_cbits[3];

    H(many_qbits);
    *many_cbits = MeasZ(*many_qbits);
    if(*many_cbits == 1) X(many_qbits);
    if(*many_cbits == 2) Y(many_qbits);
    if(*many_cbits == 3) Z(many_qbits);
}
```

2.4. C constructs in Scaffold

As previously mentioned, Scaffold is modeled on C, which allows for the integration of the C control flow into the program.

Loops. For loops are almost directly comparable to their C counter parts. It is as simple as iterating over an array of qubits or classical bits by increasing some index and applying gates to the qubits. A For loop is the main style of loop that Scaffold uses due to limitations in the compilation framework. Loop unrolling is used extensively to optimize the program; while loops are not as conducive to this effort and may not act as expected within a Scaffold program as they can be difficult to accurately unroll.

Conditionals. We can use conditionals on any normal C data type as would normally be done in a C program. But, there are special cases when dealing with quantum data types. In Scaffold we can only use conditionals on classical bits. Once we measure the quantum data, as mentioned before, we can perform conditional operations depending on the determined value.

Conditionals can be performed on both single `cbits` and arrays of `cbits`. When performing on a single `cbit` it is obvious that the value can either be a simple 0 or 1 and we can condition on this check. However, for arrays, it becomes more complicated. We take the sum of the values across the array and condition on value of the sum. In example 3 we measure a 1 by 3 two-dimensional array which can have values of integer 0 to 7 (2 to the power of 3). We can then perform different operations accordingly, in this case apply an X, Y, or Z gate to the entire set of quantum registers. Once again, we pay attention to the fact that when we are interacting with the `qbit` and `cbit` arrays we dereference the array to accurately identify and optimize for this situation.

2.4.1. Modules

Quantum data types and quantum gates are the building blocks to writing a quantum computing program. The structure of a Scaffold program is similar to that of a C program with some additional features to more adeptly handle quantum programming.

Example 4. GHZ State with 4 Qubits algorithm.

```
scaff_module GHZN (qbit *qbits, const int n){
    H(qbits[0]);
    for (int i = 1; i < n; i++){
        CNOT(qbits[i - 1], qbits[i]);
    }
}

int main (){
    qbit qbits[4];
    GHZN(qbits, 4);
    return 0;
}
```

A Scaffold program is comprised of modules, which wrap a sequence of classical and quantum instructions, and gate prototypes that define placeholders for quantum gates. In the context of a resulting quantum circuit, prior to any sort of decomposition, a module can be thought of as a function, but is used as a more complex quantum gate.

Like a C program, every Scaffold program has a main module, which is the entry point for the Scaffold compiler. A module is the Scaffold equivalent of a function call. When compiled to a target assembly language, these modules are either referenced via a specific instruction or inlined as necessary. Within a Scaffold program, a module defined with the keyword `scaff_module` before the name of the module, and any arguments needed. Like a C function, these modules can be declared with a prototype before their definition later in the program.

Example 4 is a simple example of a quantum computing algorithm in Scaffold using modules.

These modules allow for the creation of more complex sets of instructions throughout the circuit, reducing the visual complexity of a program and lends modularity to the program so that common pieces could be used throughout the circuit and between programs.

3. ScaffCC

In a higher level quantum computing program, we expect to interweave quantum computing components with classical computing methods. Up to the point where instructions are sent to a quantum machine, the programming and compilation process takes place on a classical machine. To aid this process, ScaffCC, the compiler for Scaffold, harnesses the classical LLVM open-source infrastructure to compile the quantum program to a desired representation. Through LLVM intermediate representations and modifications to the Clang code parsing functions, the compiler is able to distinguish quantum instructions from classical instructions. The classical Scaffold elements are handled by the classical machine and compiler, whereas the quantum operations will be translated into low level quantum machine language and sent to the targeted quantum hardware.

This transformation is performed included in a set of compiler commands that are accessed through the scaffold.sh script included in the repository. Running `scaffold.sh <scaffold_file>` will generate QASM file as well as a resource file after passing the Scaffold file through the various optimizations and transformations necessary to create the resources and assembly files. Various flags, such as `-b`, `-f`, `-o`, `-T`, `-R` for OpenQASM generation, flattened QASM, optimized QASM, Toffoli decomposition and rotation decomposition respectively can be denoted as compilation options.

3.1. ScaffCC functionality

Quantum computing is still at an infant stage in the sense that there is no clear path toward scalable, fault-tolerant quantum hardware. However, ScaffCC works to balance the priorities of both near term and long term quantum programs. In the near term, NISQ (noisy intermediate scale quantum) [15] machines provide our first look into physically realized quantum machines and can be used to develop higher level quantum computing components. However, NISQ machines are heavily resource constrained. Besides translating the language, the ability to analyze and optimize a program to not only use less error prone patterns, but also use less resources is crucial. ScaffCC utilizes the LLVM toolchain to perform code analysis and optimization on the Scaffold language, which provides a handy gadget to its developers, especially for research purposes. In the future, there will hopefully be larger scale quantum computers that are able to handle the algorithms that Scaffold was initially designed to express and analyze. As this becomes more viable, there will be opportunity to expand upon the initial features of Scaffold so that it continues functioning as a valid optimizer and code generator for many types of quantum programs.

3.1.1. Transformation

Similar to a compiler for classical computing, a well-designed quantum compiler should be able to adapt to generating code for different hardware targets. Support for complex gate decomposition, such as Toffoli and Fredkin gates, are built into ScaffCC since not every quantum computer has the same set of built in gates [16]. Decomposition of multidimensional quantum registers are also inherited in some passes to adapt to the rules of different quantum low level languages. This is also necessary since multidimensional registers are not supported in all quantum assembly languages or architectures.

3.1.2. Optimization

Due to the scarcity of resources associated with quantum hardware in NISQ era, optimization in quantum computing has been a popular area of research [17, 18]. Since some target quantum back ends do not have support for external gate definitions, Scaffold also offers the ability to integrate optimization methods for quantum computing in addition to classical C/C++ optimization routines, such as loop unrolling and function inlining.

3.1.3. Resource count

As mentioned above, it is critical to know how many resources are consumed by an algorithm, as each quantum operation is expensive and error prone. There are mainly two kinds of resources to consider when writing quantum programs, number of quantum registers and number of quantum gates. Scaffold provides resources estimation for both of them from several different perspectives as a built in output option for the compiler. In the latest ScaffCC update, we also have the option to leverage IBMQ resource counting to estimate the resources used by a particular program. Analysis of these quantum resources from multiple perspectives would not only be useful to algorithm builders but would also benefit researchers working in quantum hardware to examine the difference between theoretical measurement and actual count from a real machine's perspective. Section 4 provides further explanation of this process.

3.1.4. QASM generation

One of the main feature of Scaffold is generating quantum machine assembly language. Currently, Scaffold targets QASM and OpenQASM. Due to and broad use of OpenQASM, we enhance OpenQASM generation in this update to support conditional statements and multidimensional variables for quantum registers. We have a detailed documentation of the transformation strategy in sections 4 and 6.

3.2. LLVM, Clang and ScaffCC

A critical component of ScaffCC is its connection to the LLVM infrastructure. LLVM [19] provides a clean solution and toolchain for building compilers for higher level programming languages. Clang [20], one of LLVM's main subprojects, provides a language front end framework for the C family of languages that uses LLVM for the following compilation. Scaffold is built using Clang as a foundation, blending in quantum computing components with classical structures, which not only offers familiarity of C-style languages to developers but also provides a powerful foundation to explore opportunities of combining classical computing method in quantum computing.

3.2.1. Clang and LLVM adjustments

Scaffold-specific keywords have been added into Clang and LLVM has been adapted to support new quantum data types, structures and gates. ScaffCC uses various LLVM features to express the quantum data types, built in gates and reversible arithmetic gates that have been added, allowing for specific LLVM IR to be generated for later analysis by the passes. Building ScaffCC on top of Clang allows for Scaffold to act as a C variant and perform overloading of functions. Thus, various quantum data types can be molded to act similar to classical data types such as classical bits interacting with if statements.

Since LLVM does not contain support for quantum data types, we have defined LLVM representations for `qbits`, `abits`, and `cbits` throughout the entire LLVM infrastructure. To avoid collisions with frequently used data type, the memory size keywords `i1` is used to represent `cbits`, `i8` to represent `abits` and `i16` for `qbits`. This design makes it easier for developers to debug program memory allocation.

Building on these data types, we also have the set of gates that can operate on different integer types. Having these gates defined within the LLVM IRs allows for both traditional optimizations and for custom optimizations for programs resulting in a smoother compilation framework. Rather than having specially defined headers to define the gate set, the gates are built directly into the program allowing the gates remain in place between a Scaffold program it is associated quantum assembly. This also allows the resource counting to occur after compilation.

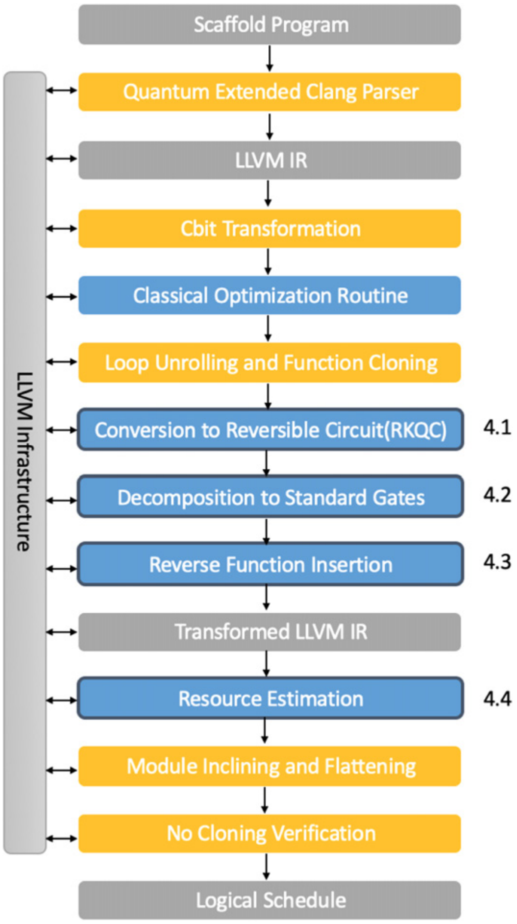

Scaffold Program
Quantum Extended Clang Parser
LLVM IR
Cbit Transformation
Classical Optimization Routine
Loop Unrolling and Function Cloning
Conversion to Reversible Circuit(RKQC)
4.1
Decomposition to Standard Gates
4.2
Reverse Function Insertion
4.3
Transformed LLVM IR
Resource Estimation
4.4
Module Inclining and Flattening
No Cloning Verification
Logical Schedule
LLVM Infrastructure

Figure 2. Scaffold routine: yellow passes are mandatory, the blue passes are optional, passes with numbers on the right feature Scaffold and will be further discussed in section 4.

3.2.2. ScaffCC pass structure

Each piece of the transformation from Scaffold code to quantum assembly in ScaffCC is achieved through carefully scheduled LLVM compiler passes. ScaffCC utilizes on the LLVM framework to perform resources estimation, optimization and QASM generation. Each intermediate transformation of the code is a pass, allowing the compiler to move from step to step, and the programmer to choose which transformations they believe to be best for the program. This classical structure also provides more autonomy to Scaffold developers and thus more possibilities to extend the language.

Throughout the ScaffCC toolchain, all the Scaffold code is compiled to LLVM IR for further analysis and transformation. The LLVM IR inherits the relationship of elements in various instructions and lays out the program flow from a general hardware's perspective. The order of these passes can be seen in figure 2.

Following the initial conversion to the LLVM IR through the general C and C++ compilation pipeline with no optimizations, ScaffCC conducts several transformations based on the targeted machine language. The first creates new calls to handle storing classical bit values. This is necessary so that the classical bits will not be regarded as dead code as the rest of the compilation continues. Following this initial preprocessing, the program is passed several standard operations including function cloning, loop unrolling, instruction combining, constant propagation and dead code elimination until no further changes are made to the program. This is important for compilation since most quantum assembly languages do not have the ability to make conditional jumps, or even jumps, when executing. They expect a single flow of instructions, so it is necessary to flatten the program before translation. When they are passed through the compilation script, a program uses the information in the program to design the resulting assembly, in order for all the information to be present at time of translation, constant propagation must also be in place.

From here, there are various transformation passes that remove the LLVM Scaffold features and replace them with their prototype with both an implementation and an actual function call that will be translated later on in the program. At this stage, various optimizations for the quantum program can also be used.

Finally, we are able to use the resulting version of the LLVM IR to produce quantum assembly language, or a resource count for the program. Both of these processes will be detailed further so that we can get a clearer sense of how the intermediate representation can be converted into executable quantum assembly.

4. Scaffold passes implementations

As mentioned in the previous section there are many possible options for combining of Scaffold passes. We gave a short description of the ScaffCC passes, but will be digging in much deeper throughout this section to describe a more in-depth compilation pipeline for a quantum program.

4.1. RKQC

The RevKit for quantum compilation (RKQC) pass was developed in order to remove a lot of the boilerplate for general arithmetic needed for quantum circuits. Many classical operations can be decomposed into reversible operations using only NOT, CNOT and Toffoli gates, and are utilized by many quantum algorithms. These decompositions can be tedious, and can be easily expanded through this pass. This pass encompasses the following arithmetic functions: assign value of 0 to register `a`, assign value of 1 to register `a`, assign value of register `b` to register `a`, swap two registers, assign register `a` to the sum of registers `a` and `b`.

To support RKQC integration into Scaffold as a set of built in operations, there are intrinsics baked into the ScaffCC variant of LLVM that represent these arithmetic functions. Once an RKQC intrinsic is found, it is replaced with the a call to decomposed version of the function. These decomposed versions are defined as functions elsewhere in the program. However, for many of these operations, the computation cannot be performed in place, that is, they require ancilla values. For example, when the value of a qubit `b` is assigned to a qubit `a` the target qubit `a` must be reset to 0 by canceling out itself. This is done by copying out to an ancilla qubit with a CNOT, perform a CNOT from the ancilla to qubit `a`, then performing a CNOT from `b` to `a` with another CNOT gate. These three operations, with the creation of the ancilla, are defined within the generated LLVM IR, replacing the declaration of the RKQC function. These functions will then be inlined with the rest of the user created modules later on in the compilation process.

4.2. Decomposition

Just as not all types of instructions are available across every classical processor, not all operations are available in the same set up across all of the different variations of quantum computers. For example, IBM's superconducting machine supports a gate library of X, Y, Z, H, S, CNOT and T gates; whereas some trapped ion machines support XX gates and rotation gates. As decompositions for complex gates must be defined accordingly [16]. As more gates are introduced into the built in Scaffold library, understanding the various decompositions needed for each type of hardware when compiling to quantum assembly is necessary.

This pass runs over the program following any loop unrolling and function cloning so that we have a mostly flattened version of the program, except for functions that are outlined rather than embedded in the program. Once a decomposable gate is found, ScaffCC retrieves a valid decomposition, and using the same values that were given to the gate, it reconstructs a new version of that gate that uses simpler components that are available on the hardware. This newly constructed set of gates replaces the original gate that was in the program. Contrary to the RKQC pass, there is no need to insert extra ancilla into the circuit. Additionally, the instructions are directly replaced, there is no outlining of the decomposition. While potentially expensive, it more directly mirrors the ultimate result of the compiled program, reduces complexity in code generation later on, and provides more accurate resource counts.

At present, ScaffCC has built in supports Toffoli and Fredkin gate decompositions, and are the main gates that are decomposed by these passes. They also each have one main decomposition into CNOTs which are generally supported on current quantum machines. In the future, more gates will be supported by similar passes. Additionally, different decomposition options will be made available, or potentially provided by the user as a compiled Scaffold program in order to make decomposition even more flexible and applicable to different quantum architectures.

4.3. Function reversal

The ability to uncompute after computation can be an important optimization for a quantum program. By reversing computation, we can free ancilla qubits, that is, return them to their initial state, to be used again. However, inserting these reversible sections can be difficult, since we must maintain correctness of the program and use the correct reversible operations, even within user generated modules.

Once again this pass walks through the program tree looking for the `_reverse_` prefix on called functions. This can be done naively using a visitor from the LLVM framework and looking for the prefix as a substring of the name at the front of the function call. Once one of these is found, it is another matter of inserting the reversed set of instructions into the circuit.

For each function, the reversed function is created when it is needed, and then cached for later use. The creation of these reverse functions is performed by first cloning a function and iterating over each instruction within the function. Since the user defined functions are constructed from the built in Scaffold instrinsics and user generated modules, each of the constituent calls to these instructions must also have an inverse to be inserted. For the built in calls, this is simply a matter of defining what operation is the inverse. For most, such as the X, Y, Z or CNOT gates, it simply the same gate again. For the parameterized gates, such as the rotation gates, it is the same gate, but the parameter is inverted. For a small number of gates, S and T, we use the Sdag and Tdag respectively as the inverse. When a user defined function is encountered, we recurse, and perform the same process to develop the reverse module for the user defined function. Once each inverse is created, the original instruction is removed from the cloned function, leaving only the inverse instruction.

While a relatively simple construction, it significantly reduces the tediousness of creating inverses, especially if many different instructions need to be reversed or the uncomputation is complicated.

4.4. Resource estimation

This pass is run after the loop unrolling and function cloning pass, and if needed, the reverse pass. Starting from the leaf functions, we keep track of the number of resources used in each call to a gate. When these are built in Scaffold intrinsics, this is easy, we are able to simply increase a counter. For user defined modules, this becomes more difficult, each external call must be tracked as well. By performing this analysis from the bottom up, we are able to most efficiently capture the called modules, since, whenever they are encountered, we can simply add the resources in the module to the program as a whole.

The resource counter is also able to track other metrics about the program, such as the number of overall qubits used, the total number of ancilla used—counting ancilla that have been uncomputed and reused as two separate ancilla (gross ancilla count), and the number of ancilla used where these are not counted as separate qubits (net ancilla count). To search for the number of qubits used in the circuit, we simply look for allocations of `i16`, where have been designated for the standard qubit type. If it is an array, we also track the size of the array. We can then extrapolate the number of new qubits needed per function. We can do the same for the gross ancilla qubit count, except look for the `i8` type. For the instance where we track the number of ancilla, where we want to reuse the ancilla, we search for the prefix `afree` which lets us know that the function also frees the ancilla qubits, and we can subtract this value from the current net qubit count at that point.

4.5. Combining passes

From the descriptions of these passes, there are instances where it becomes clear that the use of these passes is not stateless. There is a particular order that must be employed to ensure that we maintain correctness. At present, the toolchain constructed such that each pass removes another piece of Scaffold abstraction. There is nothing, except logical checks within each pass that gleams necessary information as necessary, that would prevent each the passes from being run in a different order. For example, we could run the decomposition pass prior to the RKQC pass, but this would result in potentially unperformable instructions in the resulting code since the expansion of RKQC passes can involve gates that would normally be decomposed. Similarly, for function reversal, it is possible that if decomposition is not done prior to inserting the uncomputed version of the circuit, there could be instances where invalid gates are introduced into the assembly language that are not covered by the hardware.

It is important to keep the interplay between the different passes in mind as new features are introduced. A different order can drastically change the result of a program, and potentially lead to erroneous results.

5. Pass creation

ScaffCC is designed to be expanded by the research and user community. In particular, the LLVM infrastructure provides ready support to incorporate their own optimization strategies for quantum compilation in the Scaffold toolchain through the use of LLVM passes. This flexibility,and infrastructure already in place greatly reduces the complexity of the system. Using LLVM as a jumping off points provides many classical program analysis tools that do not have to be rewritten from scratch. That being said, developers can set their own rules to deal with instructions in an LLVM pass and add them into the compilation routine. Below we provide an overview of creating LLVM passes for Scaffold. First, we explain the relationship between LLVM, Clang and ScaffCC.

Example 5. dataRepresentation struct with its member functions.

```
struct dataRepresentation{
        Value * instPtr;
        argtype type;
        bool isPtr;
        vector<int> index; // in the reverse order
        vector<int> dimsize; // sizes of dimensions vector

        /* Classical inst. */
        int intvalue;
        double doubvalue;
        bool isclassical();
        string val();

        /* Quantum inst. */
        bool isqbit();
        bool iscbit();
        void printQregistername();
        string registertype();
        string cbitarraystring();

        dataRepresentation() : instPtr(NULL), type(undef), isPtr(false), index({}),
            dimsize({}){}

        string getname(){
                string name = instPtr- > getname();
                replace(name.begin(), name.end(), ’.’, ’_’);
                return name; }
        void printdebugmode();
};
```

5.1. Adding a pass to the ScaffCC toolchain

Based on what kind of transformation is needed, an LLVM pass, based on the same visitor structure in the LLVM library can be used to walk over the Scaffold program to find information and transform information accordingly. How to create a particular pass is more closely detailed in the GitHub documentation for Scaffold. In general, a transformation or analysis is performed by iterating over the instructions in the program after defining a class that inherits from the ModulePass class. In order for the pass to actually be run on the program, the `RunOnModule` method for the class must be defined. The passes described in section 4 are good examples for how we can use Scaffold program features to determine how to make decisions for what to optimize based on the purpose of each pass.

Each pass must be added to the LLVM Scaffold library, and integrated into the build system as well. This is a matter of registering the pass with the pass manager via the LLVM API, adding the source program file into the `CMakeLists.txt` as well as adding this file in the Scaffold section of the LLVM Transforms file. Then using the `opt`, the pass can be loaded to analyze a piece of LLVM IR: `opt -S -load LLVMScaffold.dylib -PassName in.ll -o out.ll`. Once again, more detailed instructions can be found on the GitHub page. This is nearly identical to the methods used today for LLVM transformations, allowing optimizations to be easily added and reworked in a compilation strategy as necessary.

5.2. Data representation in ScaffCC

During compilation, the compiler does not discern quantum computing tasks until Scaffold associated LLVM passes. To avoid duplicate code for discerning between classical and quantum computing instructions in the pass, we designed a `dataRepresentation` structure, it is definition can be seen in example 5. The `dataRepresentation` structure can represent either classical or quantum allocated register(s). Its member functions can detect the kind of data type it belongs to, store its value and return the variable name. Developers who first work with Scaffold might also find it helpful in understanding the structural representation of Scaffold data types since this struct provides the API foundation of Scaffold pass creation.

5.3. Debug mode

As algorithms and programs increase in size and complexity, it is easy to get lost in the intricacies of LLVM, and debugging whether the compiler is working as expected gets trickier. Therefore, to make the development process a smooth experience, we have debug mode in every built-in Scaffold pass. In this update, we especially make the debug mode of GenOpenQASM pass more detailed and better informed.

Example 6. Here we demonstrate how C-Style control flow, such as for loops, and modules, can greatly reduce the difficulty of code written and needed, while increasing readability when compared to the generated OpenQASM code.

```
scaff_module maj(qbit *a){
  Toffoli(a[0], a[1], a[2]);
  CNOT(a[2], a[0]);
  CNOT(a[0], a[1]);
}

scaff_module uma(qbit *a){
  CNOT(a[2], a[1]);
  CNOT(a[2], a[0]);
  Toffoli(a[0], a[1], a[2]);
}

int main(){
  int adder_size = 2;
  qbit data[2*2 + 1];
  int i;
  for(i = 0;i < 2*adder_size-2;i+ = 2)
    maj(&data[i]);
  CNOT(data[i], data[i + 1]);
  for(i = i − 2;i>=0;i− = 2)
    uma(&data[i]);
}
```

```
OPENQASM 2.0;
include ‘qelib1.inc’;
qreg data[5];
h data[2];
t data[0];
t data[1];
t data[2];
cx data[1], data[0];
cx data[2], data[1];
cx data[0], data[2];
tdg data[1];
t data[2];
cx data[0], data[1];
tdg data[0];
tdg data[1];
cx data[2], data[1];
cx data[0], data[2];
cx data[1], data[0];
h data[2];
cx data[2], data[0];
cx data[0], data[1];
cx data[2], data[3];
cx data[2], data[1];
cx data[2], data[0];
h data[2];
t data[0];
t data[1];
t data[2];
cx data[1], data[0];
cx data[2], data[1];
cx data[0], data[2];
tdg data[1];
t data[2];
cx data[0], data[1];
tdg data[0];
tdg data[1];
cx data[2], data[1];
cx data[0], data[2];
cx data[1], data[0];
h data[2];
```

When writing a pass for Scaffold, it is a good start to rely on debug mode and try to figure out how ScaffCC works through every LLVM instruction. The debug modes attempt to give a more detailed description to the programmer as to what each step the pass is taking to change or analyze the program. By analyzing these debug statements, it is the hope that the transformations are more explicit and easier to understand to an outside user. To turn on debug mode, an environmental variable can be set to debug all Scaffold passes, or a separate variable for each pass can be set to debug an individual single pass.

6. Translating Scaffold to OpenQASM

OpenQASM [14] is one of the most broadly used quantum computing assembly languages due to the fact that IBM Q experience—a popular online platform that runs circuits on quantum backends—requires OpenQASM executable programs. Analogous to LLVM IR in classical computing, OpenQASM is an intermediate representation of a series of instructions related to a quantum circuit. While OpenQASM allows general classical computations, it does so in a limited way. For example, a loop with quantum computations is not permitted in OpenQASM. In example 6, we can see how we can greatly reduce the construction of an adder through the use of loops. Thus, Scaffold is a higher level quantum computing language in the sense that it allows more complex syntax and its compiler generates corresponding complex executable code. We motivate this with example 6, which greater reduces the necessary effort to build the Cucarro Adder [21]. In this Scaffold update, we add support to multidimensional quantum registers and conditional statements, a vital part in error correction, and which is also integrated into the compiler.

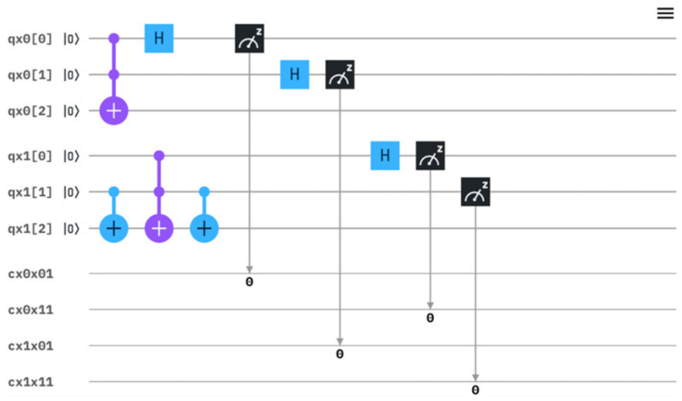


Figure 3. Multidimensional Scaffold program as circuit.

6.1. Qubit and classical bit declaration

There are disparities in quantum register declaration between OpenQASM and Scaffold. The defined LLVM IR structure is used in this case to resolve the difference. Although the naming rules are more restricted in OpenQASM than in Scaffold, such as limited ability to use special character in OpenQASM, we still manage to preserve the variable name and pass them through various transformations and optimizations as well as throughout the program. Lastly, based on the final IR, ScaffCC generates corresponding OpenQASM for declaring essential quantum registers with 1D size, as OpenQASM only supports quantum registers of arbitrary single dimension size.

Another difference between OpenQASM and Scaffold is that we do not denote the ancilla type in OpenQASM. Since ancilla qubits are normal qubits, we do not require this special notation after our optimizations have been completed. When dealing with classical registers, we can handle them in the same way we do for the quantum registers. We simply look for the classical bit allocations in the LLVM IR, and emit the classical register declaration as needed in the OpenQASM language.

6.2. Multidimensional quantum registers

As quantum algorithms get more complicated, high dimensional variables for quantum registers will be needed to make the program neat and more organized. Due to OpenQASM only allowing one-dimensional arrays of quantum registers and the specificity of conditional use on `cbit`, we decided to apply different decomposition methods to `qbits` and `cbits`. Rather than relying on the assembly language to have these multidimensional arrays, we are able to create special structures that keep track of each multidimensional value. As OpenQASM only allows for 1D arrays, we deconstruct the larger, higher dimensional arrays into smaller 1D arrays following a special naming scheme to be followed throughout the program. For example, in a two by two multidimensional `qbit` array `qarray`, we call the first subarray `qarrayx0` and the second by `qarrayx1`. We use *x* rather than a character such as an underscore since OpenQASM does not support special characters in the variable names. Example 7 shows functional examples of from a Scaffold program to its compiled OpenQASM equivalent. Additionally, an example of the resulting circuit can bee seen in figure 3. We can see how this feature can be uniquely helpful if there are blocks of qubits in a circuit that need to be acted on in the same way. In example 8 we demonstrate how we can efficiently represent a 2D architecture, and apply operations across each connection according to the coupling activation concept developed by Google [22] when compared to the necessary verbosity needed without multidimensional arrays. Coupler activation is performed by applying operations to every other connection horizontally,then vertically, and then repeating with the missing connections. With one-dimensional arrays, this could get very tedious, as each row of the architecture is declared separately and expressed with a different data structure. With this new feature, such architectures are more clearly expressed. While in our example we could conceivably achieve much of these same results by performing each of these operations, with some amount of index calculation, within a function, having multidimensional arrays allows us to bypass function inlining step and only worry about constant propagation within loop unrolling. Additionally, to achieve a similar result with only one-dimensional arrays of qubits, each row must be defined individually outside of any function if they are to be used multiple times. These arrays cannot be defined inside the function to reduce the code size, as they cannot be returned from the function and reused. The multiple definition scenario is not scalable for large architectures, and multidimensional arrays makes addressing many sets of qubits much simpler.

Example 7. Here we have a two by three `qbit` array and two by two `cbit` array, and in OpenQASM, we decompose to two arrays with three elements and give the first one the name of qx0 and the second one qx1 to represent the first 3 `qbit` array and the second 3 `qbit` array in Scaffold program. Note that for `cbit`, because we have element-wise measurement in the source code, in OpenQASM, it is necessary to decompose to single element array for each `cbit`.

```
int main (){
  qbit q[2][3];
  cbit c[2][2];

  Toffoli(q[0][0], q[0][1], q[0][2]);
  Fredkin(q[1][0], q[1][1], q[1][2]);

  c[0][0] = MeasX(q[0][0]);
  c[0][1] = MeasX(q[0][1]);
  c[1][0] = MeasX(q[1][0]);
  c[1][1] = MeasZ(q[1][1]);
  return 0;
}
```

```
OPENQASM 2.0;
include ‘qelib1.inc’;
qreg qx0[3];
qreg qx1[3];
creg cx0x0[1];
creg cx0x1[1];
creg cx1x0[1];
creg cx1x1[1];
ccx qx0[0], qx0[1], qx0[2];
//Decompose Fredkin(q0, q1, q2)
cx qx1[1], qx1[2];
ccx qx1[0], qx1[1], qx1[2];
cx qx1[1], qx1[2];
h qx0[0];
measure qx0[0] -> cx0x0[0];
h qx0[1];
measure qx0[1] -> cx1x0[0];
h qx1[0];
measure qx1[0] -> cx0x1[0];
measure qx1[1] -> cx1x1[0];
```

6.3. Basic gates

The C-style design of Scaffold, and compilation strategy employed by ScaffCC makes implementing basic gates an easy task. By the time we have completed our optimizations, the LLVM IR is very similar to single gate instructions with the addition of loads and stores that are not reduced. OpenQASM does not require these loads and stores, so we are able to go over any basic call to a gate and emit it as the instruction in OpenQASM in keeping with our naming scheme for the generated qubits. This is not a particularly difficult problem, we simply must make sure that we have the correct translation from Scaffold syntax and semantics to OpenQASM.

6.4. User defined modules

Compiling the user defined modules is only slightly a step up in complexity from the basic quantum gates. We must ensure that we inline each of these modules into their respective locations from the bottom up. Since we performed the necessary constant propagation in the static case and will have generated the proper amount of levels in the dynamic case, this will resolve recursive issues, allowing for the entire program to be inlined and outputted as OpenQASM as detailed above. This is necessary only because it streamlines the more universal use of the OpenQASM code as not all versions of OpenQASM support user defined gates.

6.5. Conditional statements

Due to the constraints of OpenQASM only allowing conditional statements on array-wise measurement, consistency in the usage of `cbits` between bitwise and arraywise measurement is important. This is why we do not allow using both bit-wise and array-wise measurement for the same `cbit` variable in a given program. Having conditional statements as whole is important as it is the basis for error correction; however, by having support for both array-wise and bit-wise measurement, we offer the user both fine-grained access off of the value of a measurement on a particular qubit, or a more broadly scoped version if a certain is taken based on the result of an array of qubits.

Condition on bit-wise measurement. As mentioned above, OpenQASM only allows array-wise measurement, so in the compilation phase, ScaffCC decomposes `cbit` arrays that are measured on the bitwise level to single element `cbit` arrays in OpenQASM so that they can be measured. Example 9 gives an demonstration of how Scaffold translates these bitwise measurements to OpenQASM, with the circuit generated depends on the measured value of a specific bit in a `cbit` array.

Condition on array-wise measurement. Although OpenQASM supports array-wise quantum operations, we were especially careful when designing this feature in Scaffold. While it might be intuitive to use pointers to reference the entire array, like in C, pointer variables represent the address of the variable. However, quantum registers do not exist on classical machines, so traditional classical memory addresses may not exist. Currently, we come up with a solution which overloads the pointer to a quantum data type. This overloading is to recognize the different pointer styles, so that we do misinterpret the qubits and classical bits being passed into the measurement or conditional. The downside of this approach is that the

Example 8. This Scaffold code demonstrates how the newly added multidimensional registers can greatly reduce the amount of copy-paste programming necessary to produce the same program. Here, we can represent a 2D architecture, and very simply demonstrate the staggered coupler activation performed by Google [22] both with and without multidimensional arrays for our quantum registers.

```
int main(){
  int width = 10;
  int height = 10;
  qbit mesh[10][10];

  int i, j;
  for(i = 0; i < height; i++)
    for(j = 0; j < width − 1; j+ = 2)
      CNOT(mesh[i][j], mesh[i][j + 1]);

  for(i = 0; i < height − 1; i+ = 2)
    for(j = 0; j < width; j++)
      CNOT(mesh[i][j], mesh[i + 1][j]);

  for(i = 0; i < height; i++)
    for(j = 1; j < width; j+ = 2)
      CNOT(mesh[i][j], mesh[i][j + 1]);

  for(i = 1; i < height; i+ = 2)
    for(j = 0; j < width; j++)
      CNOT(mesh[i][j], mesh[i + 1][j]);
}
```

```
scaff_module width_wise(qbit *mesh,
                        int start,
                        int end){
  int j;
  for(j = start; j < end; j+ = 2)
    CNOT(mesh[j], mesh[j + 1]);
}
scaff_module height_wise(qbit *mesh_0,
                         qbit *mesh_1,
                         int width){
  int i;
  for(i = 0; i < width; i++)
    CNOT(mesh_0[i], mesh_1[i]);
}
int main(){
  qbit mesh_0[10];
  qbit mesh_1[10];
  qbit mesh_2[10];
  qbit mesh_3[10];
  qbit mesh_4[10];
  qbit mesh_5[10];
  qbit mesh_6[10];
  qbit mesh_7[10];
  qbit mesh_8[10];
  qbit mesh_9[10];
  width_wise(mesh_0, 0, 9);
  width_wise(mesh_1, 0, 9);
  width_wise(mesh_2, 0, 9);
  width_wise(mesh_3, 0, 9);
  width_wise(mesh_4, 0, 9);
  width_wise(mesh_5, 0, 9);
  width_wise(mesh_6, 0, 9);
  width_wise(mesh_7, 0, 9);
  width_wise(mesh_8, 0, 9);
  width_wise(mesh_9, 0, 9);
  height_wise(mesh_0, mesh_1, 10);
  height_wise(mesh_2, mesh_3, 10);
  height_wise(mesh_4, mesh_5, 10);
  height_wise(mesh_6, mesh_7, 10);
  height_wise(mesh_8, mesh_9, 10);
  width_wise(mesh_0, 1, 8);
  width_wise(mesh_1, 1, 8);
  width_wise(mesh_2, 1, 8);
  width_wise(mesh_3, 1, 8);
  width_wise(mesh_4, 1, 8);
  width_wise(mesh_5, 1, 8);
  width_wise(mesh_6, 1, 8);
  width_wise(mesh_7, 1, 8);
  width_wise(mesh_8, 1, 8);
  width_wise(mesh_9, 1, 8);
  height_wise(mesh_1, mesh_2, 10);
  height_wise(mesh_3, mesh_4, 10);
  height_wise(mesh_5, mesh_6, 10);
  height_wise(mesh_7, mesh_8, 10);
}
```

pointer data type, representing a quantum register, from a classical computing perspective, the quantum register is not using any stored values, and may be optimized out, causing and incorrectly generated program. On the other hand, if we had just used the OpenQASM variable names to refer to an array—there might be collisions with gate names such as X, Y and Z, thus limiting Scaffold's naming conventions. Due to this dissimilarity in array referencing between the two computing methods, we create a pointer to represent the entire array of quantum registers. Furthermore, this method simplifies the ability to distinguish array-wise pointer use and element-wise pointer use (*indextoanelementinanarray*) in the LLVM intermediate representation.

We can see an example of this array-wise measurement from Scaffold to OpenQASM in example 10 and the corresponding circuit in figure 4.

Example 9. Transformation of bitwise conditional measurement from Scaffold to OpenQASM. For an if condition on single a `cbit`(line 11, 12), in order to get the `cbit` array values, the array must be decomposed at the start of the compilation phase. The compiler would know which quantum register is to be used before compiling the program to assembly language so that it is able to generate the correct allocation instructions.

```
1   int main () {
2       qbit q[3];
3       cbit c[2];
4
5       X(q[0]);
6       Z(q[1]);
7
8       c[0] = MeasX(q[0]);
9       c[1] = MeasZ(q[1]);
10
11      if(c[0] == 1) X(q[0]);
12      if(c[1] == 0) Z(q[1]);
13
14      return 0;
15  }
```

```
1   OPENQASM 2.0;
2   include ‘qelib1.inc’;
3   qreg q[3];
4   creg cx0[1];
5   creg cx1[1];
6   x q[0];
7   z q[1];
8   h q[0];
9   measure q[0] -> cx0[0];
10  measure q[1] -> cx1[0];
11  if(cx0 == 1 ) x q[0];
12  if(cx1 == 0 ) z q[1];
13
```

Example 10. In this example, we show the transformation of array-wise measurement in Scaffold into array-wise measurement in OpenQASM. We perform measurement on `qbit` array q, and store the result in `cbit` array `syn`—both of which are arrays with 3 registers. Then we call an *X* gate on a `qbit` element in q based on the measurement result of `syn`.

```
int main() {
    qbit q[3];
    qbit a[2];
    cbit c[3];
    cbit syn[3];

    X(q[0]);

    CNOT(q[0], a[0]);
    CNOT(q[1], a[0]);
    CNOT(q[1], a[1]);
    CNOT(q[2], a[1]);

    *syn = MeasZ(*q);

    if(*syn == 1) X(q[0]);
    if(*syn == 2) X(q[1]);
    if(*syn == 3) X(q[2]);

    *c = MeasZ(*q);
}
```

```
OPENQASM 2.0;
include ‘qelib1.inc’;
qreg q[3];
qreg a[2];
creg c[3];
creg syn[3];
x q[0];
cx q[0], a[0];
cx q[1], a[0];
cx q[1], a[1];
cx q[2], a[1];
measure q -> syn;
if(syn == 1) x q[0];
if(syn == 2) x q[2];
if(syn == 3) x q[1];
measure q -> c;
```

7. Scaffold-NISQ

As we enter the noisy intermediate-scale (NISQ) era involving quantum devices equipped with 50–100 qubits, increasing emphasis has been placed on developing small-scale versions of NISQ programs. In particular, the focus has been on designing NISQ algorithms and compilation techniques that may eventually be able to perform faster calculations than classical computers, with applications of these algorithms ranging from simulating the physics of entanglement to machine learning. This involves both taking into account certain properties of machines such as connectivity [23], noisy qubits and connections [5, 24]. In line with this focus, we have created ScaffCC-NISQ—a lightweight version of ScaffCC designed specifically with NISQ applications in mind, with the goal of allowing faster and easier distribution, installation and usage of the compiler. Streamlining much of the original ScaffCC's functionality, the NISQ version compiles Scaffold programs to flattened QASM, hierarchical QASM and OpenQASM, as it does with the original version, but also features a series of NISQ benchmarks and a new resource estimator pass. These features are a built-in pipeline within the Scaffold scripts. It features a subset of the existing ScaffCC compilation passes and an updated set of compilation steps in accordance with the recent ScaffCC update. These particular features have been added to not only make the compiler easier to use, but provide an

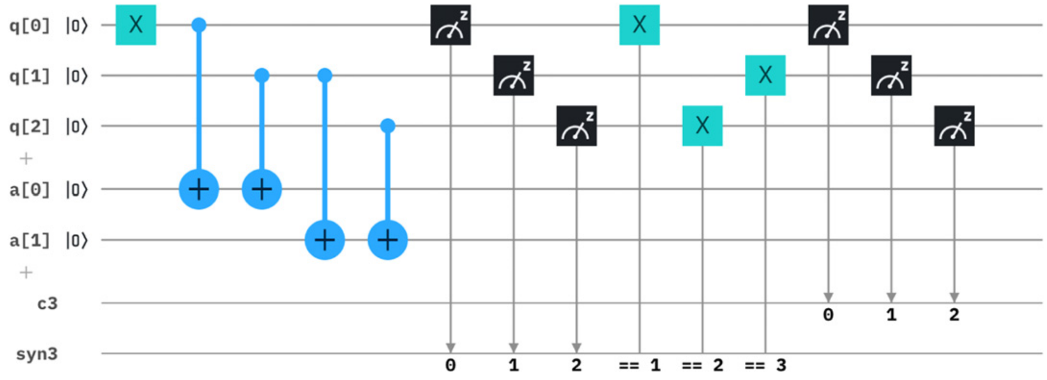


Figure 4. Arraywise conditionals as a circuit.

environment where NISQ strategies can be developed by providing these external resources to accurately describe and check the new developments.

7.1. NISQ benchmarks and IBM Q simulator

Similar to ScaffCC, the NISQ version allows for further optimizations and passes to be added to the LLVM framework, providing developers and researchers with increased flexibility. In order to ensure that these compiler modifications do not impact the reliability or accuracy of its functionality, ScaffCC-NISQ features a series of Scaffold NISQ benchmarks. A provided testing script first compiles these deterministic benchmarks to OpenQASM after which, it connects to IBM Q Experience's quantum simulator back-ends to run the circuits and generate output that is checked to ensure correctness. If all the NISQ tests pass, developers can be assured that their compiler optimizations and additions have not changed the existing compiler capabilities. Additionally, comparison against some of these basic benchmarks allows a programmer to further ensure that any pass additions had adversely affected certain resources in the compiled program.

7.2. IBMQ backend simulator

IBM's high powered simulator allows for compiled OpenQASM code to be tested and run by simulating quantum behavior on classical computers. Through IBM Q Experience, these quantum simulators are provided as a cloud service available to the public, making it easy to prototype circuits and estimate their corresponding noise responses. ScaffCC-NISQ's resource estimation pass and the test script that validates the existing NISQ benchmarks both rely on IBM's quantum simulator to run OpenQASM circuits and return results. As a result, in order to avail of these features, a user must create and provide their IBMQ credentials by saving them in the `nisq_benchmarks/config/Config_IBMQ_experience.py` file. ScaffCC-NISQ then uses these credentials to connect to one of IBMQ's available backend simulators. It does so by converting the compiled OpenQASM file to a quantum object before passing it to IBMQ's simulator which returns the output associated with the circuit.

7.3. Resource count

The resources associated with a program, such as the number of qubits and the error correction required by a high number of gates (which require more qubits), significantly affect the cost of running that program on a physical device. As a result, ScaffCC-NISQ first relies on known classical compiler algorithms, such as loop unrolling and procedure cloning to flatten out Scaffold programs, and then builds upon IBMQ's resource estimation capabilities in order to approximate the depth, width, size, number of operations and number of tensor factors associated with a Scaffold program. IBMQ estimates these resources by building a directed acyclic graph from the quantum circuit, decomposing instructions and expanding operation nodes. This updated ScaffCC-NISQ pass is much simpler than its predecessor and is able to provide a more granular understanding of the resources required such as number of tensor factors unlike ScaffCC's pass which only provides information about number of gates and time-steps. The aim behind this pass is to help reduce these significant costs linked to program resources as well as allow for an early comparison between different programs. Furthermore, the addition of this pass helps make certain that the resources associated with a quantum algorithm running on a limited NISQ processor are mostly being dedicated to solving the main problem instead of on overhead associated with hardware mapping [13].

8. LLVM version update

Both ScaffCC and ScaffCC-NISQ have undergone a version upgrade to the latest LLVM 8.0.1 version. As a result, both compilers now benefit from updated LLVM bug fixes as well as increased hardware and software compatibility. The most significant changes to arise from LLVM's version upgrade involve the ability to run a single pass of the code generation pipeline, and to stop or start the code generation pipeline at a given point—allowing for more sophisticated debugging techniques. This does not cause any serious issues or changes in the usage of Scaffold or ScaffCC except that it brought about the changes from a `module` keyword to `scaff_module` as `module` become a keyword within Clang. Additionally it necessitated the update of several passes, mostly when referring to the LLVM Contexts of certain instructions. Furthermore, both ScaffCC and ScaffCC-NISQ are now built using CMake as opposed to the traditional Make installation methodology that LLVM used to rely on. We hope that in the future, this large update will allow us to stay more in-keeping with the current LLVM version with minimal effort.

9. Future directions

Scaffold, as it stands, is a decent proof of concept that allows for the manipulation of quantum systems within a familiar C-style environment. It sits between declarative models such as IBM's Qiskit or Google's Cirq, and the more high level abstractions with more robust type checking, such as Microsoft's Q#. The main benefit to Scaffold over these other languages is the native ability to produce OpenQASM and optimize over several passes. But we do have a vision for how we can continue to improve the compiler for general use.

9.1. Improvement of the pass system

Since we have the benefit of a dynamic back-end where we are able to implement several different optimizations available to the compiler, it makes the most sense to leverage that advantage. The most obvious way to do this is to make creating a pass for Scaffold much more accessible.

While helpful, the LLVM APIs are not so concerned with quantum features of the system, but rather program structure. Creating a general pass that surfaces more familiar features of a quantum system to a programmer rather using than the LLVM APIs may allow for simpler optimization experience for quantum programs. If we can find a way to abstract these structural features as quantum analogs, or conversely, quantum features to a more traditional programming analog it could be useful in expanding the optimization passes for the compiler in the future.

9.2. Dynamic optimization application

As new passes are added, we must find a way to combine these various optimizations such that they do not counteract or detract from one another. A secondary piece to additional passes would be a method to determine which passes might be more helpful through various heuristics of the program. This could be implemented as a non-code changing pass, simply to collect information about the circuit in order to gain general information about which optimizations to run rather than simply running them all in sequence.

Another possible solution is running this pass multiple times over the resulting code, providing different pre and post conditions that must be met for various passes. As certain conditions are met, we can mark passes as available to be run with once again with various heuristics for which passes could be more beneficial to the overall performance of the program on the quantum hardware.

9.3. Debugging

Due to the nature of quantum computing, no framework has any real ability to step through a program, especially as they are being simulated. There have been some efforts through using assertions [4]. If we able to properly annotate the generated QASM code as it is compiled, in combination with other quantum debugging tools, we may be able to offer insight into programs as they are run or simulated in order to identify potential bugs or missed assertions in the running program.

10. Conclusion

Just as the power of quantum computing is difficult to comprehend prior to having tangible results, it is difficult to create abstractions and tools for a technology that is still being developed. Nevertheless, we hope that Scaffold provides a sound abstraction to begin programming quantum algorithms. Beyond being a sound abstraction, Scaffold's compiler has been designed to be flexible, and incorporate new algorithms and ideas as they are developed without having to upend the compiler. Using the LLVM infrastructure ScaffCC uses classical tools to bridge a quantum gap to create a step-by-step chain that helps develop a better programming experience and a more optimized program.

Acknowledgments

This work is funded in parts by EPiQC, an NSF Expedition in Computing, under Grant CCF-1730449/1730082, by the NSF STAQ project under Grant NSF Phy-1818914, and by DOE Grants DE-SC0020289 and DE-SC0020331.

ORCID iDs

Andrew Litteken https://orcid.org/0000-0001-5676-1747

References

[1] Abhari A J *et al* 2015 ScaffCC: scalable compilation and analysis of quantum programs *Parallel Comput.* **45** 2–17
[2] Abhari A J *et al* 2012 Scaffold: quantum programming language https://cs.princeton.edu/research/techreps/TR-934-12
[3] Chong F T, Franklin D and Martonosi M 2017 Programming languages and compiler design for realistic quantum hardware *Nature* **549** 180–7
[4] Huang Y and Martonosi M 2019 Statistical assertions for validating patterns and finding bugs in quantum programs *Proc. of the 46th Int. Symp. on Computer Architecture-ISCA '19*
[5] Prakash M *et al* 2019 Formal constraint-based compilation for noisy intermediate-scale quantum systems *Microprocess. Microsyst.* **66** 102–12
[6] Ding Y *et al* 2018 Magic-state functional units: mapping and scheduling multi-level distillation circuits for fault-tolerant quantum architectures *51st Annual IEEE/ACM Int. Symp. on Microarchitecture (MICRO)* pp 828–40
[7] Green A S *et al* 2013 Quipper: a scalable quantum programming language (arXiv:1304.3390)
[8] Svore K *et al* 2018 Q#: enabling scalable quantum computing and development with a high-level DSL *Proc. of the Real World Domain Specific Languages Workshop 2018. RWDSL2018* (Vienna, Austria: ACM) 1–10
[9] Lapets A *et al* 2013 QuaFL: a typed DSL for quantum programming *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP* 19–26
[10] The Q programming language-microsoft quantum https://docs.microsoft.com/en-us/quantum/language/?view=qsharp-preview
[11] Eisenberg R *et al* 2019 The quipper system https://mathstat.dal.ca/ selinger/ quipper/doc/
[12] Abraham H *et al* 2019 *Qiskit: An Open-Source Framework for Quantum Computing*
[13] Ho A and Bacon D 2018 Announcing Cirq: an open source framework for NISQ algorithms https://ai.googleblog.com/2018/07/announcing-cirq-open-source-framework.html
[14] Cross A W *et al* 2017 Open quantum assembly language (arXiv:1707.03429 [quant-ph])
[15] Preskill J 2018 Quantum computing in the NISQ era and beyond *Quantum* **2** 79
[16] Linke N M *et al* 2017 Experimental comparison of two quantum computing architectures *Proc. Natl Acad. Sci. USA* **114** 3305–10
[17] Shi Y *et al* 2019 Optimized compilation of aggregated instructions for realistic quantum computers *Proc. of the 24th Int. Conf. on Architectural Support for Programming Languages and Operating Systems-ASPLOS '19*
[18] Gokhale P *et al* 2019 Partial compilation of variational algorithms for noisy intermediate-scale quantum machines *Proc. of the 52nd Annual IEEE/ACM Int. Symp. on Microarchitecture-MICRO '52*
[19] Lattner V A C 2004 LLVM: a compilation framework for lifelong program analysis & transformation https://llvm.org/
[20] Lattner V A C 2004 Clang: a C language family frontend for LLVM http://clang.llvm.org/
[21] Cuccaro S A *et al* 2004 A new quantum ripple-carry addition circuit (arXiv:[quant-ph]/0410184)
[22] Babbush R *et al* 2019 Quantum supremacy using a programmable superconducting processor *Nature* **574** 505–10
[23] Prakash M *et al* 2019 Noise-adaptive compiler mappings for noisy intermediate-scale quantum computers *Proc. of the 24th Int. Conf. on Architectural Support for Programming Languages and Operating Systems. ASPLOS '19* (Providence, RI, USA: Association for Computing Machinery) pp 1015–29
[24] Alexander C *et al* 2019 Phase gadget synthesis for shallow circuits (arXiv:1906.01734 [quant-ph])