CS 269Q Lecture Notes (Lecture 12)

Robert S. Smith[*]

These are notes for the 8 May 2019 lecture for Stanford's CS 269Q course. Questions, corrections, and comments are welcome!

Contents

[*]email: `robert@rigetti.com`

1 Introduction

On the face of it, "compilation" is a very simple idea. I, a programmer, want to write my programs as easily as possible. My stubborn computer, however, only executes a very low-level machine code. Regardless of whether the computer is classical, quantum, or otherwise, a program called a *compiler* translates what I've written into something that can be run.

Most programming language ecosystems, even notionally "interpreted" languages, have some sort of compiler. The `cpython` implementation of Python, for instance, has a so-called *bytecode compiler*, that translates the Python into a linear sequence of instructions to be run on a special machine made out of software called the Python interpreter.

More familiar compiled languages, like C or Fortran, generally require the explicit use of a compiler. You use something like `gcc` to translate the C code into machine code your computer runs natively.

I'd like to share a little bit about compilers for quantum programming languages. Very excitingly, in the world of quantum software engineering, we are seeing compilers slowly moving the goal posts of quantum advantage closer to us. Industrial-strength quantum compilers are relatively new, but are paying great dividends in our ability to both write and run code on quantum computers. I am one of the authors of `quilc`—Rigetti's portable, open-source[1], optimizing, and fully automatic compiler for quantum computation. It has become an essential element of our software development kit for all of the reasons described hitherto.

Compilation of quantum programs serves two critical roles. First, it allows us as programmers express programs with a broader set of primitives, since quantum computers expose only a limited set of operations. Second, compilation provides means for optimizing the program, generally making the program shorter or make use or more efficient operations.

Optimizing compilers for *classical* programs are merely a convenience. An optimized classical program may run in less time or consume less memory. This, in turn, means that I, the programmer, have to wait less, or use a less expensive computer, in order to accomplish the task I set out to do with the classical computer. The optimization (usually) does not affect the correctness or accuracy of the program.

However, in the case of quantum programs, optimization serves a much more important role outside of providing convenience. While a quantum computer is running, it *decoheres* due to natural effects, which causes the machine to produce incorrect results with higher probability over time. For superconducting qubit processors, there's an associated measure for this decoherence, called the T_1 and T_2 times. Roughly speaking, these times put a bound on the

[1]`https://github.com/rigetti/quilc`

amount of time a program can run before it produces bad results more than 50% of the time. Currently, T_1 times sit around 1 to $100\,\mu$s.

I won't delve into the details as to *why* this time limit exists, except to say one of the top areas of research is how to build machines that ever increase this limit. Since the development of the first superconducting qubits, this time has been improved exponentially, for a quantum-Moore's-like law.

Anyway, a compiler has the role of optimizing your program, to potentially take a long program and make it shorter, giving the programmer more freedom and certainty that their program will run as best it can. We are going to first look at the problem of compilation through a very "pure", mathematical lens for which theorems about their performance exist.

2 The Strong Compilation Problem

Relatively straightforwardly, we can encode the problem of compilation in the language of mathematics of unitary operators. The usual compilation problem goes something like this:

Problem 1 (The Strong Compilation Problem) *Let G be a discrete and finite subset of $SU(2^n)$ and let $U \in SU(2^n)$. Given a fixed but arbitrary $\epsilon > 0$, find a finite sequence of ℓ elements of G*

$$V := g_\ell \cdots g_2 g_1$$

such that

$$\max_{|\psi\rangle} \|(U - V)\,|\psi\rangle\,\| < \epsilon.$$

This problem is called "strong" because it must account for maximum error across *all* possible states. It turns out that Solovay and Kitaev determined that it is efficiently solvable, provided that the group generated by G is dense in $SU(2^n)$ and that if $g \in G$, then $g^\dagger \in G$ too.

Theorem 1 (Solovay–Kitaev[4]) *The Strong Compilation Problem can be solved with sequences of length $\ell \in O([\log(1/\epsilon)]^c)$ where $c > 3$ in time complexity[a] $O([\log(1/\epsilon)]^c)$ where $c > 2$.*

[a]These complexities are known as *polylog*, because they are polynomials of logarithms.

They proved this theorem constructively, meaning that the actual sequence of gates to reconstruct U is computed. That method is called the *Solovay–Kitaev algorithm*, and—in the grand scheme of all things quantum—is quite simple.

This theorem should be understood appropriately because it has a hidden but nonetheless crucial detail. In the problem and the theorem, the value of n—the number of qubits—is fixed. That is to say, it makes a statement about the length of approximating n-qubit operators with a set of 1- to n-qubit operators. If n is *not* fixed, then the complexity becomes $O(2^n[\log(1/\epsilon)]^c)$. In general, this is to be expected. The size of Hilbert space grows exponentially with the number of qubits, so approximating many-qubit operators with few-qubit operators requires an exponential number of them, and then some to accommodate our bound on errors.

With that caveat in mind, this is a landmark theoretical result, as it shows that the compilation problem is tractable on a classical computer. However, the Strong Compilation Problem and Solovay–Kitaev find most of their popularity in quantum error correction circles. A huge benefit of the Solovay–Kitaev algorithm is that it only needs a finite set of gates. This is beneficial in the study of quantum error correction because one can devise schemes to error-correct exactly the gates in the gate set.

Solovay–Kitaev is more foreign to the scientists and engineers of NISQ[2] devices, however, because Solovay–Kitaev doesn't take into account the facilities and detriments of a NISQ system. On the plus side, many NISQ systems have continuous families of gates available to them, such as Z-rotations, with virtually perfect fidelity. On the minus side, NISQ devices and the associated gates have noise, and the compilation procedure doesn't take any of that into account. As such, we file Solovay–Kitaev away as an important theorem that provides a bright glimmer of hope for the aspiring quantum compiler developer.

From here, we will focus on understanding compilation that is perhaps more appropriate for NISQ devices, and to do this, we will want the machinery of the QAM.

3 Compilation of Quantum Abstract Machines

In the third lecture, you learned about the *quantum abstract machine*. This is the funny six-tuple of information:

$$(|\Psi\rangle, C, G, G', P, \kappa).$$

Here, $|\Psi\rangle$ is our quantum state, C is our classical state, G is our set of static gates, G' is our set of parametric gates, P is a sequence of instructions comprising our program, and κ is where we are in the program. Because only $(|\Psi\rangle, C, \kappa)$ change during the execution of P, we call that the *state part* of the QAM. Suppose we have the following Quil program:

[2] "Near-term Intermediate Scale Quantum"

```
RZ(pi/2)  0
RX(pi/2)  0
RZ(-pi/2) 1
RX(pi/2)  1
CZ        0 1
RZ(-pi/2) 0
RX(-pi/2) 1
RZ(pi/2)  1
HALT
```

Loaded onto one of Rigetti's freshly initialized eight-qubit Rigetti quantum computers, this would represent the following quantum abstract machine, which we will call M:

$$\begin{aligned}
|\Psi\rangle &= |000000\rangle \\
C &= \texttt{000000} \\
G &= \left\{\mathrm{RX}(\pm\tfrac{\pi}{2})_{\{0,1,2,3,4,5,6,7\}}, \mathrm{CZ}_{\{01,12,23,34,45,56,67,70\}}\right\} \\
G' &= \{\theta \mapsto \mathrm{RZ}(\theta)_{\{0,1,2,3,4,5,6,7\}}\} \\
P &= \left\langle \mathrm{RZ}(\tfrac{\pi}{2})_0, \mathrm{RX}(\tfrac{\pi}{2})_0, \mathrm{RZ}(-\tfrac{\pi}{2})_1, \mathrm{RX}(\tfrac{\pi}{2})_1 \mathrm{CZ}_{01}, \mathrm{RZ}(-\tfrac{\pi}{2})_0, \mathrm{RX}(-\tfrac{\pi}{2})_1, \mathrm{RZ}(\tfrac{\pi}{2})_1, \mathrm{HALT}\right\rangle \\
\kappa &= 0.
\end{aligned}$$

After undergoing the transition induced by $P_0 = \mathrm{RZ}(\frac{\pi}{2})_0$, namely

$$(|\Psi\rangle, C, \kappa) \to \left(\mathrm{RZ}(\tfrac{\pi}{2})_0 |\Psi\rangle, C, \kappa+1\right),$$

we have

$$\begin{aligned}
|\Psi\rangle &= \left(\tfrac{1}{\sqrt{2}} - \tfrac{1}{\sqrt{2}}i\right)|000000\rangle \\
C &= \texttt{000000} \\
G &= \left\{\mathrm{RX}(\pm\tfrac{\pi}{2})_{\{0,1,2,3,4,5,6,7\}}, \mathrm{CZ}_{\{01,12,23,34,45,56,67,70\}}\right\} \\
G' &= \{\theta \mapsto \mathrm{RZ}(\theta)_{\{0,1,2,3,4,5,6,7\}}\} \\
P &= \left\langle \mathrm{RZ}(\tfrac{\pi}{2})_0, \mathrm{RX}(\tfrac{\pi}{2})_0, \mathrm{RZ}(-\tfrac{\pi}{2})_1, \mathrm{RX}(\tfrac{\pi}{2})_1 \mathrm{CZ}_{01}, \mathrm{RZ}(-\tfrac{\pi}{2})_0, \mathrm{RX}(-\tfrac{\pi}{2})_1, \mathrm{RZ}(\tfrac{\pi}{2})_1, \mathrm{HALT}\right\rangle \\
\kappa &= 1.
\end{aligned}$$

We of course would continue to transition this machine until it reaches a HALT[3]. Let's write this idea of "running a machine until it HALTs" with the function $\mathcal{R}$.

[3] Recall that HALT induces the state transition

$$(|\Psi\rangle, C, \kappa) \to (|\Psi\rangle, C, \kappa),$$

which is to say that the machine does not continue execution, since κ does not change.

So, if you'll take my word for it, $\mathcal{R}M$ is

$$\begin{aligned}|\Psi\rangle &= \frac{1}{\sqrt{2}}(|00000000\rangle + |00000011\rangle)\\ C, G, G', P &= \cdots\\ \kappa &= 8.\end{aligned}$$

Given this is how we model computation, we are prepared to state another kind of compilation problem in the context of a QAM. Roughly speaking, we want to convert one QAM into another one with a different set of gates G, G', roughly having otherwise equivalent semantics. Ultimately, this gives rise to the following problem:

Problem 2 (The Weak Compilation Problem[a]) *Let*

$$M_{source} := (|\Psi\rangle, C, G_{source}, G'_{source}, P_{source}, \kappa)$$

and

$$M_{target}(x) := (|\Psi\rangle, C, G_{target}, G'_{target}, x, \kappa)$$

both be QAMs. Also let

$$M^*_{source} := \mathcal{R}M_{source} \qquad \text{and} \qquad M^*_{target}(x) := \mathcal{R}M_{target}(x)$$

whose quantum states are $|\Psi^*_{source}\rangle$ and $|\Psi^*_{target}(x)\rangle$ respectively. Then for an arbitrary but fixed $\epsilon > 0$, find a program x such that the respective classical states are identical and

$$\left\| |\Psi^*_{src}\rangle - |\Psi^*_{target}(x)\rangle \right\| < \epsilon.$$

[a]This statement of the problem is actually not entirely robust. What does non-determinism mean, say with a measurement? How do we cope with it? In order to answer these questions, and consequently make this problem statement more robust, we would need to provide *formal operational semantics* to the quantum abstract machine, so that we can reason about the equivalence of the non-determinism of quantum mechanics. This also extends to reasoning about programs that have non-linear control flow, e.g., loops. Despite its fragility, however, it'll still serve useful for reasoning about different compilation heuristics.

We can see that this problem is a lot more relaxed than the Strong Compilation Problem. In particular, not only do we have to compile for *just* the specific state of the QAM, we also (possibly!) get entire families of parametric gates at our disposal. Both of these facts will prove to be helpful in compilation. Note that we could strengthen the conditions of this problem to revert it to the Strong Compilation Problem by universally quantifying over all $|\Psi\rangle$ and forcing $G' = \varnothing$.

4 The Instruction Set Architecture

Physical quantum processors have physical constraints. For example, in superconducting qubit systems, if two physical qubits are not near one another, then it's generally not possible for them to interact in a single, discrete operation. So while we might have a quantum computer that supports the `CNOT` gate, we aren't really being precise by saying that. Instead, we ought to say something like "our computer supports a `CNOT` gate between qubits 1 and 0, as well as between qubits 2 and 1." More exotic gates, such as the Mølmer–Sørensen gate[5] can operate on more than two qubits simultaneously. This gate has the action of producing something like a GHZ-state:

$$|0\rangle^{\otimes n} \mapsto \frac{e^{i\phi_0}|0\rangle^{\otimes n} + e^{i\phi_1}|1\rangle^{\otimes n}}{\sqrt{2}}.$$

The role of G and G' in the QAM is to define precisely what the machine supports. It turns out, "what the machine supports" also provides a certain discrete structure to that machine. This structure is called the instruction set architecture or ISA. Understanding the ISA gives rise to a topological understanding of the machine, and provides excellent supporting theory in the construction of data structures a compiler might use. Moreover, knowing the ISA makes formulating certain compilation questions a lot easier.

Because qubits can support operations, qubit pairs can support operations, qubit triplets etc., we formally consider the ISA of a quantum abstract machine to be a *hyper-multigraph* whose vertices are qubits and whose edges are possible operations on qubits. It is a *hypergraph* because we consider generalizations of edges, called "hyperedges", which can connect multiple vertices. It is a *multigraph* because there can be more than one hyperedge present between vertices.

More specifically, we define an ISA as such:

Definition 1 (Instruction Set Architecture) *An* instruction set architecture *is a set of qubit labels V called the* vertices*, and a set of $|V|$ hyperedges which are maps*[a]

$$E_k : \binom{V}{k} \to (\textit{sets of } k\textit{-qubit gates}).$$

We call E_k a hyperedge of valence *k. For a given set $S \subseteq V$, the gate $g \in E(S)$ if and only if g acts on the qubits of S non-trivially.*

[a] The notation $\binom{S}{k}$ is the set of k-element subsets of S.

In Quil, for n qubits, the vertex labels are $V := \{0, 1 \ldots, n-1\}$. So we fix that convention moving forward.

As shorthand, we might write this graph as (V, E) where E is understood to be the $|V|$-tuple of hyperedges. For superconducting qubits, the valence is usually limited to 2, that is, we only have one- and two-qubit gates[4].

As a remark, for qubits p and q, if $g_1 \in E_1(\{p\})$ and $g_2 \in E_1(\{q\})$, we generally do *not* regard $g_1 \otimes g_2 \in E_2(\{p, q\})$, even though $g_1 \otimes g_2$ is a perfectly synthesizable operation.

The hyper-multigraph presentation of an ISA is useful for understanding at a glance what a quantum abstract machine can do. It's also nice because it makes for drawing nice pictures; sometimes we talk about the *qubit graph*, which is a traditional graph whose vertices are qubit labels and whose edges represent the existence of *any* two-qubit gate between them. When we *don't* care so much about the topological structure, it's useful to just bag everything together. In fact, we recover the content of a QAM from an ISA:

$$G \cup G' = \bigcup_{\substack{1 \le i \le |V| \\ H \in \operatorname{image} E_i}} H. \tag{1}$$

What's important to note—in the context of the specification of a QAM's ISA—is that there is no "ordained" CNOT gate. There are only CNOT gates that act on vertices. Of course I'm speaking of CNOT as a placeholder for any usually named gate. Consider the following three-qubit ISA whose only hyperedge is

$$E_2 = \begin{cases} \{0, 1\} \mapsto \{\mathtt{CNOT}_{10}\} \\ \{1, 2\} \mapsto \{\mathtt{CNOT}_{12}\}. \end{cases}$$

If qubits 0, 1, and 2 are residents of the Hilbert spaces B_0, B_1, and B_2 respectively, then we can fix a space for our system and associated basis. The space we fix is

$$B_2 \otimes B_1 \otimes B_0$$

along with its usual basis, for reasons described in [3]. We can then write the gates in terms of matrices. These would be

$$\mathtt{CNOT}_{10} = I_2 \otimes \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

$$\mathtt{CNOT}_{12} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \otimes I_0.$$

[4]This isn't strictly true for superconducting qubits, and is certainly not true for other qubit technologies like ion traps.

We can see that in this view, the notion of there being “a” CNOT gate melts away. As a corollary, this also means that $\mathtt{CZ}_{01}$ and $\mathtt{CZ}_{10}$ are perfectly indistinguishable[5].

We’ve defined what compilation means, and we’ve provided a few ways to organize the structure of our gates, so we are prepared to attack some of the problems of compilation.

5 Compilation Heuristics

Unlike the Strong Compilation Problem, there is no algorithm that solves the Weak Compilation Problem wholesale. Instead, one uses a lot of heuristics, especially making use of the ISA in said heuristics. This is not too different with the state of affairs with usual classical compilers. A compiler like gcc employs lots of different techniques to achieve compilation of C code to machine code.

In general, a heuristic function is a function that takes us from one QAM to another. Solving the Weak Compilation Problem amounts to successively selecting heuristics until we reach the final QAM. It might look something like this:

$$M_{\text{source}} \xrightarrow{f_0} M_1 \xrightarrow{f_1} M_2 \xrightarrow{f_3} \cdots \xrightarrow{f_n} M_{\text{target}}.$$

Each heuristic f should be designed to either take us a few steps “closer” to our target QAM, or to bring us to a more efficient QAM.

5.1 Decomposition

In essence, *all* compilation problems are decomposition problems; we wish to express objects in one semantic domain as objects with equivalent properties in another semantic domain. However, here, we mean something more specific, decomposition in a “truer” sense of the word. In the hyper-multigraph view of a QAM, it might be that our source QAM has hyperedges whose valence is larger than any of those in the target QAM. For instance, we might write a program which uses the Toffoli gate

$$\mathtt{CCNOT} := (I \otimes I) \oplus \mathtt{CNOT}$$

whose valence is 3, but our target QAM only has one- and two-qubit gates. The goal of decomposition is to re-express our high-valence gates in terms of products of gates whose valences match the target gate set, without any other specific regard to the target gate set. The heuristic would have the effect of both transforming the program of the QAM, as well as the gate set. We move

[5]We will break away from this philosophy slightly when we talk about a compilation subproblem called “routing”.

"closer" to our target QAM because we've removed gates from our gate set that perhaps don't exist in the target.

There are many ways to perform these decompositions:

1. Algebraic identities,
2. Matrix factorization methods, and
3. The Solovay–Kitaev algorithm.

Algebraic identities are the simplest to explain, and often are extremely effective in efficient compilation[6]. Fig. 1 shows an example decomposition of $\mathtt{CCNOT}_{012}$ into the set

$$\{\mathtt{CZ}_{\{01,02,12\}}, \mathrm{RX}(\pm\pi/2)_{\{0,1,2\}}, \mathrm{RZ}(\pm\pi/4)_{\{0,1,2\}}\}.$$

Note the all-to-all connectivity of the qubits.

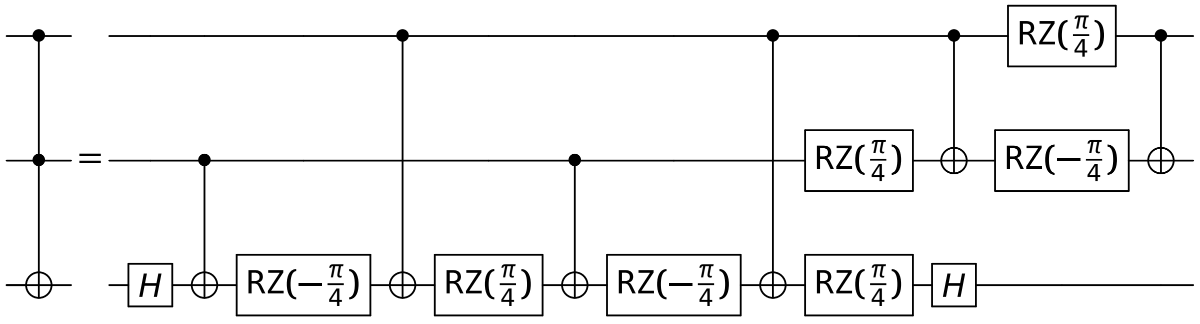

Figure 1: A decomposition of CCNOT into six CNOT gates [6].

While this might not be a sufficient compilation for CCNOT for all target QAMs, it certainly moves us a step closer[7] to a final compilation.

Algebraic identities don't provide a general means for decomposition, however, so we rely on other methods. Matrix factorization methods are deterministic ways to recursively breaks larger matrices into smaller ones. In the one-qubit case, one might use Euler decomposition, which is a factorization of any unitary matrix into a sequence of three rotations. In its fundamental form, one has

$$\mathrm{RZ}(\gamma)\mathrm{RY}(\beta)\mathrm{RZ}(\gamma).$$

[6]They are so effective that `quilc` has easy ways to write and incorporate them them.

[7]We could formally state the transition as mapping a gate set G to

$$(G \setminus \{\mathtt{CCNOT}_{012}\}) \cup \{\mathtt{CZ}_{\{01,02,12\}}, \mathrm{RX}(\pm\pi/2)_{\{0,1,2\}}, \mathrm{RZ}(\pm\pi/4)_{\{0,1,2\}}\},$$

and mapping CCNOTs of the program to their respective expansions.

In our Toffoli decomposition, there are a couple Hadamard gates that need to be converted. The Hadamard gates have an Euler decomposition of

$$\mathsf{H} = \mathsf{RZ}(\pi/2)\mathsf{RX}(\pi/2)\mathsf{RZ}(\pi/2).$$

This formula needn't be seen as magic; it can be understood perfectly in analog to rotations in SO(3).

For more qubits, one has more complex methods. The *cosine-sine decomposition* takes a unitary $U \in SU(2^n)$ and computes

$$U = (L \oplus L')e^{iY \otimes \Delta}(R \oplus R').$$

All L, L', R, R', Δ are matrices of half the dimension of U and Δ is diagonal. Recursive application of this factorization to the left- and right-factors leads to an expression which ultimately can be written as two-qubit controlled rotations.

An even simpler factorization, called *quantum Shannon decomposition*, follows a similar line of thinking. Both of these methods and more are described in [7]. One should note that these methods compile n-qubit operators into $O(2^n)$ two-qubit operators.

Provided we've used a collection of identities and decomposition methods, we still aren't in a form which will generally respect a QAM of interest. We do have a program whose gates have the right target valences, but they're perhaps not gates that are available to us. This leads to the next problem of quantum compilation.

5.2 Routing

Many QPUs have ISAs which are *not* a hodgepodge of gates. Often, there's regularity in the ISA. Rigetti's chips, for example, support the same two-qubit gates across each valence-2 edge, and the same one-qubit gates on each qubit. This opens up a variety of compilation techniques, including routing.

Given a program in a QAM with all-to-all connectivity, we wish to compile the program so it conforms to the given connectivity. This problem is relatively simple when the program has gates whose matrices are found in the edges—for one can often use a chain[8] of SWAP gates—but more difficult when they're not. Suppose our gate set of our target QAM is

$$G \cup G' = \{\mathsf{RX}(\pm\pi/2)_{\{0,1,2\}}, \mathsf{RZ}(\theta)_{\{0,1,2\}}, \mathsf{CNOT}_{\{01,12\}}\}.$$

In the Toffoli gate example, there are two CNOT_{02} gates that need to be routed. Routing the CNOT gate is also quite simple, though for different reasons, with

[8]More specifically, if one is compiling an operator U_{pq} where there is no p–q edge, then one can perform a series of SWAPs on qubits p and q so that U in effect operates on an edge that *does* exist.

the identity:

$$\texttt{CNOT}_{02} = \texttt{CNOT}_{12}\texttt{CNOT}_{01}\texttt{CNOT}_{12}\texttt{CNOT}_{01}.$$

This identity can be used recursively for a longer-range CNOTs. All of these identities, expansions, SWAP gates, etc. usually lead to redundant and inefficient code. This is where optimization is helpful.

5.3 Optimization

There are many optimization techniques for quantum programs. We will just mention a few:

- Allocating qubits so that gates run more favorably, due to either their connectivity to neighboring qubits.
- Collecting groups of instructions and searching for simple reducing identities, such as HH = I, RX(α)RX(β) = RX($\alpha + \beta$), and the like. These are the quantum equivalent of *peephole optimizations*, but requires careful attention because of the large amount of instruction parallelism in quantum programming languages.
- Taking advantage of known quantum state preparations, so that e.g. when an operator is acting on an eigenstate, that operator can be eliminated, as with Z and the $|0\rangle$ state.

5.4 Approximate Compilation

Quantum computers are imperfect, and one can take advantage of these imperfections to do a better job compiling. While I might want the quantum computer to execute a sequence of gates $U := \prod_i U_i$, it will in reality execute the gates with some amount of error E_i, giving us the “actual” execution

$$V := \prod_i (U_i + E_i).$$

We can estimate the total error of the process with some selection of norm. If $\|E_i\| < \varepsilon_i$, then we can say that the total error we will incur will be bounded above by $r := \sum_i \varepsilon_i$.

In some sense, this means that our “target” operation U has a sort of radius r in which we will invariably land. If we can supply a different sequence of gates that *deliberately does not* calculate U, but instead calculates some other operator U', so long as U' has a smaller radius and is contained in V, we have done better. More specifically, if we can supply some alternative set of operations $U' := \prod_i U'_i$ such that its physical realization is

$$V' := \prod (U'_i + E'_i),$$

and if

$$\|U - U'\| + \sum_i \|E'_i\| < r,$$

then we have succeeded in optimizing U.

This isn't some theoretical result, and an algorithm for doing so for two-qubit gates is described in [2]. It is also implemented in `quilc`, which can use measures of *average gate fidelity* of the available gate set.

Industrial-strength quantum optimizing compilers, such as `quilc`, contain all of these methods and more, arranged cleverly so that you can convert just about any QAM into a large collection of other QAMs, including all currently known superconducting architectures.

6 Programs Below Quil

I want to end my section by talking a little bit about what happens to Quil code when it's about to be run. While it may seem like so, Quil isn't *quite* an assembly language, even if it's written for the ISA of the physical device. As such, there's one more level of compilation that's very close to traditional assembling.

Superconducting qubits, at the end of the day, operate off of microwave pulses and electrical currents. So any code written needs to turn into a sequence of pulse-firings or current-drives. These are controlled by a plurality of processors which can—and do—operate on the QPU in parallel.

The waveforms are generally not as sophisticated as one might think. An $\mathrm{RX}(\pi/2)$ gate corresponds to a waveform that looks something like $\sin(\omega t)e^{t^2}$, where ω is a frequency in the 5 GHz range. The Quil code

```
RX(pi/2) 3
RX(pi/2) 5
```

would assemble into two of these sinusoidal pulses that are fired synchronously at qubits 3 and 5.

As a rule of thumb, one-qubit gates usually have a duration of around 20 to 50 ns, while two-qubit gates have a duration of around 80 to 200 ns. To me, it is very remarkable that these gates-measured-in-nanoseconds can effect change in a quantum state that cannot be stored on my laptop feasibly.

References

[1] Robert S. Smith, Michael J. Curtis, William J. Zeng. *A Practical Quantum Instruction Set Architecture*. arXiv.org preprint. arXiv:1608.03355. 2016.

[2] Eric C. Peterson, Gavin E. Crooks, Robert S. Smith. *Fixed-Depth Two-Qubit Circuits and the Monodromy Polytope* arXiv.org preprint. arXiv:1904.10541. 2019.

[3] Robert S. Smith. *Someone Shouts* $|01000\rangle$*! Who's Excited?* arXiv.org preprint. arXiv:1711.02086. 2017.

[4] Christopher M. Dawson, Michael A. Nielson. *The Solovay–Kitaev Algorithm*. Quantum Information and Computation. 2005.

[5] Klaus Mølmer, Anders Sørensen. *Multi-particle entanglement of hot trapped ions*. Physical Review Letters 82. 1999.

[6] Vivek V. Shende, Igor L. Markov. *On the CNOT-cost of TOFFOLI gates*. Quantum Information and Computation. 2009.

[7] Vivek V. Shende, Stephen S. Bullock, Igor L. Markov. *Synthesis of Quantum Logic Circuits*. IEEE Transactions on Computer-Aided Design, vol. 25, no. 6. 2006.