

QUANTUM CIRCUIT COMPILATION FROM THE GROUND UP

NATHANIEL STEMEN



Master of Mathematics (MMath)
Department of Applied Mathematics
Faculty of Mathematics
Institute for Quantum Computing
University of Waterloo

April 2022

Nathaniel Stemen: *Quantum Circuit Compilation From The Ground Up*,
Master of Mathematics (MMath), © April 2022

SUPERVISOR:

Joel Wallman

LOCATION:

Seattle, WA (completed remotely during COVID-19)

TIME FRAME:

September 2020—April 2022

Ohana means family.
Family means nobody gets left behind, or forgotten.
— Lilo & Stitch

Dedicated to the loving memory of James Tighe.
1939–2016

ABSTRACT

This thesis details the problem of quantum circuit compilation. Starting from the very definition of compile, we introduce many of the ideas needed to understand the main problem of circuit compilation from the very basics. We cover classical compilers and show how the effort to build effective circuit compilers draws heavily from its classical counterparts. Upon introducing the formalism of quantum computation, we are able to formulate many of the problems related to circuit compilation in a mathematical language, and detail some of the cutting edge efforts. We end by showing how circuit compilation is part of a much larger “quantum stack” that needs to be created to have effective quantum computers.

*They didn't have much trouble
teaching the ape to write poems:
first they strapped him into a chair,
then tied the pencil around his hand
(the paper had already been nailed down).
Then Dr. Bluespire leaned over his shoulder
and whispered into his ear:
You look like a god sitting there.
Why don't you try writing something?*

— James Tate

ACKNOWLEDGMENTS

Many thanks are in place for the successful completion of this thesis. First I would like to thank my academic advisor Joel Wallman for the guidance during my bumpy career as a graduate student. In addition, thank you to the following professors to helping me complete my studies: John Watrous, Achim Kempf, Michael Waite, Brian Ingalls, and Michael Brannan. Whether it was sharing details about your personal career, asking probing questions, or offering time and having supportive conversation despite not having to: thank you.

Thank you to Joel's research group for helping me deal with Joel's departure: Darian McLaren, Anthony Chytros, Matthew Graydon, Stefanie Beale, Sam Ferracin, and Joshua Skanes-Norman. I would also like to thank my many classmates without which remote classes would have been far less interesting and rewarding: Wilson Wu, Chelsea Komlo, Mohammad Ayyash, Nicholas Zutt, and Xiaoran Li. A big thank you is also in order for Overleaf and in particular John Lees-Miller and Ryan Looney for allowing me to work part time and being extremely flexible with my hours. It was great to continue working with the team. . . and to supplement the measly graduate student salary.

Thank you Mom and Dad for letting me live in your house while we endured the brunt of the pandemic. Thank you Diane for always having my back and being supportive throughout my graduate studies. Thank you to my friends who were always open to discuss my struggles and triumphs: Kevin (both of you), Rafael, Ana, and Aimee.

CONTENTS

I Front End

- 1 All Things Classical 3
 - 1.1 What can a computer do? 3
 - 1.2 Compilers 4
 - 1.2.1 Compilation Phases 6
 - 1.2.2 Optimizations 9
 - 1.2.3 Examples 10
 - 1.3 LLVM 11
- 2 Quantum Computation 13
 - 2.1 Historical Development 13
 - 2.2 Quantum Computation 14
 - 2.2.1 Formalism 15
 - 2.2.2 Quantum Gates 16
 - 2.2.3 Quantum Circuits 18
 - 2.2.4 Universal Gate Sets 21
 - 2.3 Fault Tolerance 22
 - 2.4 Mathematics 23
 - 2.4.1 Operator Norms 23
 - 2.4.2 Free Group 23
 - 2.4.3 Dense-ness 24
 - 2.4.4 Fidelity 24

II Back End

- 3 Quantum Hardware 29
 - 3.1 Requirements 29
 - 3.2 Quantum Chips 30
 - 3.3 Hardware Specifications 32
 - 3.4 Errors 34
- 4 Circuit Compilers 37
 - 4.1 Compiling on a ring 39
 - 4.2 Methods 42
 - 4.2.1 Verification 44
 - 4.3 Quantum Stack 45
- 5 Conclusion 47

- Bibliography 49

LIST OF FIGURES

Figure 1.1	von Neumann Architecture	3
Figure 1.2	Action of Compiler	5
Figure 1.3	Compilation Phases	7
Figure 1.4	Compiler with many front and back ends	9
Figure 2.1	Example Quantum Circuit	19
Figure 2.2	Abstract Quantum Circuit	19
Figure 3.1	IBMQ Jakarta Architecture	31
Figure 3.2	Quantum Volume Protocol	34
Figure 4.1	Action of Quantum Compiler	37
Figure 4.2	Modularity of Quantum Compiler	38
Figure 4.3	Circuit to be compiled	40
Figure 4.4	Circuit after compression	40
Figure 4.5	Circuit after peephole optimization	40
Figure 4.6	Ring Topology	41
Figure 4.7	Compiled Circuit	41
Figure 4.8	Compiled Circuit on Directed Ring	42
Figure 4.9	VQE Schematic	44
Figure 4.10	Quantum Stack	45

LIST OF TABLES

Table 1.1	Machine Code	8
Table 1.2	Peephole Optimizations	9
Table 2.1	Common Quantum Gates	17
Table 2.2	Gate Compositions	20

ACRONYMS

CPU	Central Processing Unit
TPU	Tensor Processing Unit
IR	Intermediate Representation
QIR	Quantum Intermediate Representation
NISQ	Noisy Intermediate-Scale Quantum

VQE Variational Quantum Eigensolver
ISA Instruction Set Architecture
CISC Complex Instruction Set Computer
RISC Reduced Instruction Set Computer

LIST OF SYMBOLS

$U(n)$ Group of unitary operators or matrices of dimension $n \times n$. 15
 $\mathcal{M}_n(\mathbb{C})$ Set of $n \times n$ complex matrices. 15
 $PU(n)$ Group of projective unitary operators of dimension $n \times n$. 16
 \mathbb{F}_2 The finite field with two elements. 18
 G^* Kleene star of a finite set G . 19
 $[n]$ Shorthand notation for the integers up to and including n : $\{1, 2, \dots, n\}$. 21
End V The set of endomorphisms, or linear transformations on a vector space V . 23

Part I

FRONT END

To begin this document we will introduce the notion of a compiler and show the foundational role it plays in our modern computing infrastructure. We will cover the main ideas from compilers that are useful for our quest to understand quantum circuit compilers in part [ii](#). We will then switch gears to cover the basics of quantum computation needed to understand the quantum part of our story.

ALL THINGS CLASSICAL

In this chapter we will give a very brief overview of the components of classical computers that will be helpful to further discussions of quantum circuit compilation. A key component to quantum circuit compilation is the word “compilation”, whose origins (in computing) date to the early 1950’s when electronic digital computers were in their early stages. Understanding the historical development of compilation and its techniques will provide ideas and tools necessary to solve the new task of quantum circuit compilation.

This chapter is meant to provide the reader with the basics of some computing terminology and ideas. It is by no means a complete introduction to compilers, nor computer architecture.

1.1 WHAT CAN A COMPUTER DO?

If you’re reading this, I’m sure you can imagine something your computer is capable of. Maybe reading this document online, sending messages/email, browsing the internet, writing documents, etc. These are very high-level operations our computer can perform, but under the hood much more primitive operations are taking place. It is these primitive operations that we wish to understand, and will have many similarities with modern-day quantum hardware.

A simplified model of computer architecture, known as the von Neumann Architecture (fig. 1.1) shows what we now call a Central Processing Unit (CPU) which is the workhorse of the computer.¹

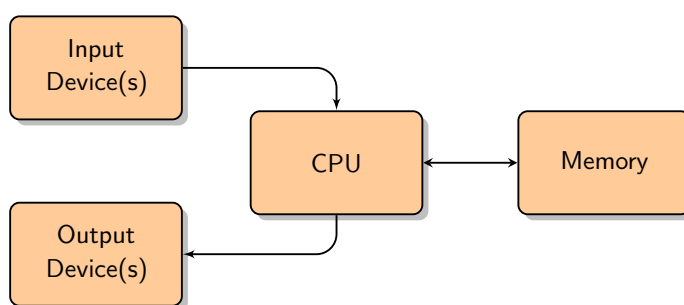


Figure 1.1: von Neumann Architecture

Since the CPU is the computational component of the computer, what can *it* do? Modern CPUs are built on the Instruction Set Architecture (ISA), which means that the CPU has a finite set of operations (also

¹ At least in this *very simplified* model.

known as instructions) that it can perform. Every operation the computer can perform must be built up from these primitive instructions. Some examples of these primitive operations are:

- put a value into memory;
- add two values in memory together and store the result in a new location;
- perform the bitwise negation on a value;
- compute the square root of a value.

One can then use these primitives to build up complex functionality that eventually implement the capabilities we know and love (and hate) computers for.

Choosing an *ISA* results in the creation of a *complexity class* which is a collection of problems that can be solved using a polynomial number of primitive instructions/operations. In practice, most *ISAs* implement the same complexity class, and we denote it by *P*. The formal definition of *P* is “decision problems solvable by a deterministic Turing machine in a polynomial amount of time”, but the picture one should have in mind is “problems for which we have efficient algorithms”. For more details on complexity classes, and computational complexity in general consult [NC10, Chapter 3] for the material with an eye towards quantum, and [AB09] for a more detailed exposition.

The *ISA* architecture style has seen major success, but it suffers from the drawback of requiring the programmer to work at the very low-level of machine instructions. To work at a higher level of abstraction, and hence to have a higher level of productivity, computer scientists and programmers created new languages which were easier to read, write, and reason about. This necessitated new languages to be “translated” into the instruction set after the code was written. The software responsible for translating these higher level ideas into a machine’s instruction set are known as **compilers**.

1.2 COMPILERS

While compilers have their origins in the aforementioned translation of higher-level code into lower-level code, they have grown considerably to perform many more tasks. Before we dive into all of the capabilities of modern compilers, let’s take a step back and recall what the word compile means.

Merriam-Webster [Mer] defines the word *compile* to mean

to compose out of materials from other documents.

In the context of programming language compilers, “other documents” might mean the code itself, as well as configuration files and environment variables. This definition is reflected in *Compilers: Principles,*

*techniques & tools*² [Aho+07] where the authors introduce compilers through the process of transforming software.

[B]efore a program can be run, it first must be translated into a form in which it can be executed by a computer.

The software systems that do this translation are called *compilers*.

Hence we can view compilers as a function taking software written at one level of abstraction and bringing it down to a lower level that a computer's CPU can understand.



Figure 1.2: Action of Compiler

The term compiler was first used in the context computers by Grace Hopper in the early 1950's while working on a system that could translate symbolic mathematics into a machine language. Initially Hopper's new idea was met with resistance as it was thought to be unrealistic.

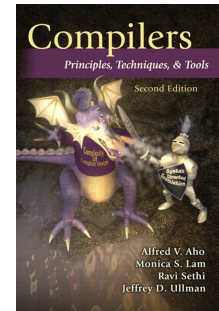
I had a running compiler, and nobody would touch it because, they carefully told me, computers could only do arithmetic; they could not do programs. It was a selling job to get people to try it. I think with any new idea, because people are allergic to change, you have to get out and sell the idea. (Grace Hopper [Hop52])

In the end, Hopper succeeded in selling the idea and compilers have become a ubiquitous piece of modern computing infrastructure. While Hopper's compiler focused solely on code translation, a modern compiler might perform all of line reconstruction, preprocessing, lexical analysis, syntax analysis, semantic analysis, conversion to an Intermediate Representation (IR), optimization (and there are many different types!), and finally code generation. Thankfully we will not need to understand *all* of these parts in full, but rather will focus on Intermediate Representations, optimizations, and code generation.

RESOURCES Before jumping into processes that make up a compiler, we will first detail some of the hardware restrictions that compilers must be aware of while performing their job. Modern digital computers are built on the transistor: a small³ device models a bit (0 or 1) as

² Colloquially known as "The Dragon Book" because of the cover, and likely the most famous book on (classical) compilers. This is also where the logo of the LLVM project originates from which we will discuss in section 1.3.

³ They are today, but they were not always small!



The Dragon Book



LLVM Logo

the absence or flow of electricity. Since transistors provide the basic building block of the bit, the transistor count is an effective measure of how much memory the computer has. Hence when a compiler is performing some sort of optimizations, it must be aware of the amount of memory it can make use of. While modern computers have an abundance of memory, not all memory is created equally. The CPU can talk to the computer's long-term storage (hard drive), however it is a slow communication that is not ideal to perform frequently and cause bottlenecks in many computations. Instead, CPUs have their own (smaller) internal memory which is often referred to as a collection of registers. These registers provide fast access to variables during computation. Hence during the compilation of a program, the compiler's knowledge of the target hardware's CPU allows the compiler to efficiently use the on-board registers, and make informed decisions as to when to use long-term storage.

The second resource that compilers are often made aware of is the CPU's ability to run parallelized computations. The ability to perform multiple instructions at the same time is often taken advantage of in compilers via techniques like loop unrolling.⁴ However, not all architectures support this mode of optimization, and even if it does, the compiler must be careful to ensure parallelization optimizations do not over-burden registers.

1.2.1 *Compilation Phases*

As in the previous section, a compiler has many different responsibilities. Each responsibility is broken into a separate component so that it can be understood on its own, and later be reused in its own context. A schematic for this can be seen in fig. 1.3 on page 7 showing the main steps that we will be concerned with in this document.

SYNTAX ANALYZER This phase is for ensuring the code is syntactically well formed (that is, that it abides by the specification of the language). If one is writing code in a binary alphabet with characters 0 and 1, then the "program" 00011 is syntactically valid, while 1102 is not because a 2 appears in the code. Many compilers transform the code into a syntax tree to complete the verification.

SEMANTIC ANALYZER Now that the code is syntactically valid, we can ensure it has meaning. This phase usually consists of type checking and scope validation (ensuring the code does not access variables outside of scope). In many compiled languages the operation

⁴ https://en.wikipedia.org/wiki/Loop_unrolling

'hello' * 5 would pass syntax analysis, but fail semantic analysis because a string multiplied by an integer is not a valid operation.⁵

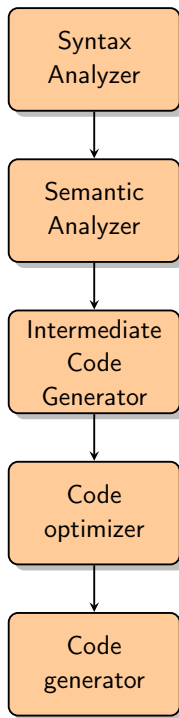


Figure 1.3: Compilation Phases

INTERMEDIATE CODE GENERATOR The code is now ensured to be well formed and can begin to be transformed into something the hardware is capable of running. Passing directly to the code generator is possible from here, but the end product will be slower as no optimizations will take place. Instead, the existing code (sometimes the syntax tree created in the previous steps is used) will be transformed into an Intermediate Representation (IR). This is a mid-level representation of the code in that it is typically thought of as somewhere between the high-level of abstraction of the programming language, and the low-level instruction set.

This is best seen with a simple example. Suppose we have the following snippet to calculate the final location of a moving object after 5 seconds.

```
x_final = x_initial + velocity * 5
```

Upon transforming this code to an IR, it takes on a more basic form.

```
t1 = inttofloat(5)
t2 = velocity * t1
t3 = x_initial + t2
x_final = t3
```

The power here comes from the fact that the Intermediate Representation (IR) can be language agnostic, and hence many languages can compile into the same IR. This design allows for the use of an optimizer for many languages.

CODE OPTIMIZER Once the code is in the IR, the optimizer will attempt to “improve” it using many different methods. Improve can mean many different things, but usually refers to runtime and memory use. Optimizations that occur during this step are constant propagation, dead code elimination, removing unnecessary code from loops, and loop unrolling. Optimizing the above example our code is still “bulkier” than originally written, but compressed in comparison to the original IR-form.

```
t1 = velocity * 5.0
x_final = x_initial + t1
```

⁵ It is completely valid in other languages like Python, but Python is not a compiled language.

Here we have skipped the call to `inttfloat` and instead immediately converted the integer 5 to the float 5.0. We have also combined two of the steps to reduce the number of temporary variables we have to create and store in memory. As you can see the task of the optimizer is not only to try and speed up the code, but reduce its memory usage as well. Some of the other problems the code optimizer must tackle are instruction selection, register allocation, and instruction scheduling all of which have analogs we will see in chapter 4.

CODE GENERATOR Finally we have an optimized IR and we can generate code for hardware. This requires us to know which hardware it is we'd like to run our code on as each chip might have a different ISA. This is a very difficult step as many of the sub-problems that are required to be solved are themselves NP-complete such as register allocation [Cha+81]. Further, generating mathematically optimal machine code has also been shown to be undecidable [Aho+07]! Hence this step uses effective heuristics to solve the problem at hand in tractable amounts of time. Typically this step is broken down into first optimizing the IR for the hardware that has been chosen, followed by the actual code generation. If this occurs the optimizer is typically referred to as a hardware-independent optimizer, and a later stage of optimizations is performed in a hardware-dependent optimizer. We will see later that the distinct phases of optimization are of crucial importance when compiling quantum circuits.

Again following the above code example, upon code generation we may end with the following generic hardware instruction code.

	Function	Meaning
LDF R2, velocity	LDF	Load float
MULF R2, R2, #5.0	MULF	Multiply floats
LDF R1, x_initial		
ADDF R1, R1, R2	ADDF	Add floats
STF x_final, R1	STF	Store float

Table 1.1: Machine Code

Here anything beginning with R is a register.

The phases described here are often grouped into three larger categories. The syntax and semantic analysis, as well as the generation of an IR fall under the umbrella of "front end", the optimizer is the optimizer, and everything else that follows is the "back end". The implications of this design is that an optimizer and backend can be paired with many different front ends as long as the front end can generate the optimizer's preferred IR flavor.

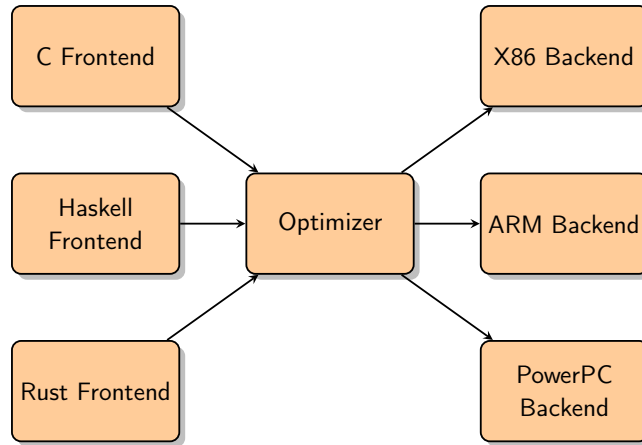


Figure 1.4: Compiler with many front and back ends

1.2.2 Optimizations

Before moving on to some examples of compilers, it's important to understand the separation of concerns in the two types of optimizations we've seen. The main optimizer we see in fig. 1.3 as “Code optimizer” and again the “Optimizer” in fig. 1.4 are typically where the majority of optimizations take place in classical compilers and are performed on an IR. One interesting class of examples are peephole optimizations [McK65]. These are optimizations that take advantage of small patterns found in code that can be simplified in some way. Some examples are seen in table 1.2. Other examples include dead

Instruction	Optimized Instruction
Read value into a register, then immediately store it in memory.	Do nothing
$a \cdot x + b \cdot x$	$(a + b) \cdot x$
$x - x$	0
$(A^T B^T)^T$	BA

Table 1.2: Peephole Optimizations

code elimination, common subexpression elimination, and inlining. The optimizations done here—usually to the ends of faster runtime and smaller memory use—are performed in the hopes that once the code is compiled into machine code it *will* run faster. The intuitive optimizations often remove duplication, but many other optimizations that are not so clear take advantage of the commonalities among CPU design to produce code that will run faster on any CPU.

With an optimized IR, and a chosen backend, or hardware, the code can be modified to suit the instruction set, as well as other restrictions the hardware may place on computation. For example, most CPUs

have a small number of registers, and hence must use them wisely throughout the computation so as to use *all* of them where possible, but not slow down computation by waiting for a register to be available. Another example is instruction scheduling, where the compiler must figure out an optimal ordering to the computation, again to maximize the CPUs compute power while not causing bottlenecks. There are many other examples of hardware-dependent optimizations, but as you might imagine, many require an intimate knowledge of the hardware's particular design. All this transformation occurs while maintaining the same semantic meaning of the original program.

In summary the first hardware-independent optimization should be thought of as optimizing the implementation theoretically, and the hardware-dependent optimization as ensuring the optimized algorithm runs as fast as possible in its final implementation. Many more examples of optimizations (both hardware-independent and hardware-dependent) can be found in [Rod20, Chapter 8].

1.2.3 Examples

We've now seen what a compiler is and what we typically use it for. A few examples are in order to help understand how compilers work in the real world, and just how varied they can be.

CLANG: Short for C Language, this is a compiler frontend for the C/C++ languages. It takes in C/C++ code and produces an LLVM IR which we will learn about in section 1.3. It then lets LLVM handle the rest of the compilation processing.

LATEX: While perhaps not very obvious, L^AT_EX is indeed a compiler as it takes high-level formatting code, and produces a lower level representation of what the user wants to typeset. Usually that comes in the form of postscript which is another programming language that is read by printers (hardware) to produce the requested document. Postscript can also be read by PDF readers and browsers which then display content as the author desired (maybe).

TENSORFLOW: TensorFlow is a library for machine learning that has drawn on the design principles of compilers in attempts to speed up and ensure the accuracy of models. Indeed it has a frontend where the user builds their model and compiles it into an IR known as HLO IR or High Level Operations. Typical optimizations then occur and again using the LLVM compiler infrastructure this code can be brought to many backends such as the browser, mobile, and specialized compute infrastructure (such as Google's Tensor Processing Unit (TPU)). This is all before we talk about TensorFlow Quantum which allows for hybrid quantum-classical machine learning models [Bro+20].

1.3 LLVM

The LLVM⁶ project [LA04] is one of the largest open source compiler projects in existence and much of the compiler architecture we've discussed here come from its design. The founder of the project Chris Lattner has characterized compilers succinctly in [Lat19] as

the art of allowing humans to think at a level of abstraction
that they want to think about.

As an interesting historical note, once the ISA scheme had become commonplace, chip designers began to implement more and more complex instructions on CPUs so that machine code became higher level. At the same time, compilers became more popular, especially as their optimizations became more robust, and useful. This led to a distinction between chip architectures known as Complex Instruction Set Computer (CISC) and Reduced Instruction Set Computer (RISC). At the time of writing, CISC processors are dominant in desktop computers, while RISC processors emphasize efficiency and can be found in phones and many other portable computing hardware. Some examples include Intel's x86 and x64 chips which are built in the CISC style, while ARM is major designer of RISC chips (including the most recent Apple Bionic A15 chip). Today RISCs are sometimes referred to using the backronym "Relegate Interesting Stuff to the Compiler".

With the growth of LLVM, developers have pushed the compiler to extend its use to "heterogeneous hardware" [Lat+21], which already includes new types of computing hardware like TPUs and could in the future encompass a Quantum Processing Unit (QPU). This is exciting not only because classical computer designers are beginning to consider quantum technologies as coprocessors, but because the monumental classical computing infrastructure can then be leveraged to aid in the solutions to quantum problems. With the futurism, hype, and unknowns surrounding quantum technologies, it often seems that fundamentally new and ingenious ideas are needed to forward the field. Projects such as the above show there are serious possibilities of recycling, or at the very least, learning from what has come before us.

⁶ The project, while originally an acronym for Low Level Virtual Machine, now goes solely by LLVM. The original name reflects the fact that the compiler targets low-level IR code that runs on some theoretical (hence the term virtual) machine. Since the inception virtual machines have come to mean something different, hence the abandonment of the acronym.

In this chapter we will lay the groundwork for the necessary ideas from quantum computation. We will not attempt to introduce quantum computation from the ground up, but instead introduce and emphasize the ideas needed for compiling quantum circuits. The notation used here will mostly follow [Wat18] and we recommend [NC10] for a more thorough introduction to the material.

2.1 HISTORICAL DEVELOPMENT

One of the core tenets of quantum theory is that, at this scale, nature is reversible. Hence, when physicist Charles H. Bennett began investigating reversible Turing machines [Ben73] we might say the field of quantum computing was *just* getting started. Since Turing machines are the mathematical and theoretical foundation for modern computers, it makes sense that a reversible Turing machine might lay the groundwork as the foundation for a computer that operators under quantum mechanical law. More than 6 years later, Paul Benioff extended this work to describe a fully quantum mechanical version of a Turing machine in his paper “The computer as a physical system: A microscopic quantum mechanical Hamiltonian model of computers as represented by Turing machines” [Ben80].

Once the theoretical foundation had been laid by Bennett and Benioff, Richard Feynman brought the idea mainstream when he proposed using these new computers to simulate quantum mechanics itself. This idea was very attractive at the time (1981) since our classical computers were not powerful enough to simulate large quantum systems,¹ and since Feynman was such a popular figure the idea finally took hold. Feynman motivated the need for a new paradigm in computing as such.

Nature isn't classical, dammit, and if you want to make a simulation of nature, you'd better make it quantum mechanical, and by golly it's a wonderful problem, because it doesn't look so easy. (Richard P. Feynman [Fey82])

Even with one of the most famous physicists popularizing the idea, it took another 10 years to see the next major development which came when David Deutsch and Richard Jozsa gave an example of a problem that is solved exponentially faster on a quantum computer

¹ In fact, they still aren't!

than a classical one [DJ92]. If there was any hesitancy from the academic community at this point about the theoretical usefulness of a quantum computer, this result showed real potential for the emerging technology. More applications start rolling in with quantum teleportation [Ben+93] and famously Peter Shor’s polynomial time algorithm to factor integers (and hence break many modern cryptosystems) [Sho94].

The latter caught the eye of the US Government and within the year of Shor’s publication the National Institute of Standards and Technology (NIST) organized the first government-funded conference on quantum computation.² Since then ambitions have risen and technological progress has allowed for more and more qubits and quantum computers today have even been shown to complete tasks that classical ones cannot in any feasible amount of time. In particular a team at China’s Hefei National Laboratory used their 66-qubit computer³ to complete a task in 4 hours that would take state of the art programs tens of thousands of years [Zhu+22].

In 2018 John Preskill coined the term Noisy Intermediate-Scale Quantum (NISQ) as a characterization of quantum computers with a relatively small number of noisy qubits (50–100) with limited connectivity: i. e. machines that have dominated the past decade, and will likely continue to for the next few years [Pre18]. The problem presented in this document is relevant to quantum computers past the NISQ-era, but are especially important as we attempt to squeeze every ounce of computation out of them in the NISQ-era.

2.2 QUANTUM COMPUTATION

In this section we will go over the basics of quantum computation. Before continuing I would like to recommend [NC10] as well as <https://quantum.country> as great resources to learn the basics of quantum computing.

² It’s likely this is when quantum computation was put on the radar of other US government agencies. In 2014 leaked documents showed the National Security Agency had begun a project dubbed “Owning The Net” whose purpose was to use a quantum computer to break internet cryptography and to “gain access to and securely return high value target communications”. The status of the project—which also goes by the moniker “Penetrating Hard Targets”—is unknown.

³ Affectionately named Zuchongzhi after Chinese mathematician Zu Chongzhi whose computation of π was more accurate than any other for more than 800 years.

2.2.1 Formalism

A quantum bit, or **qubit** for short, is a vector $|\psi\rangle$ in 2-dimensional complex space \mathbb{C}^2 such that $\| |\psi\rangle \| = 1$. Often the following canonical basis is chosen and referred to as the computational basis.

$$|0\rangle := \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad |1\rangle := \begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad (2.1)$$

In this basis, a qubit is represented as

$$|\psi\rangle = \alpha |0\rangle + \beta |1\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \quad (2.2)$$

with the normalization condition that $|\alpha|^2 + |\beta|^2 = 1$. In the case of eq. (2.2) the state $|\psi\rangle$ is said to be in a **superposition** of state $|0\rangle$ and $|1\rangle$.

We often need to understand more complicated systems than just simple qubits, and to do so we use the **tensor product** to build up systems from subsystems. E. g. if $|\psi\rangle \in \mathbb{C}^2$ and $|\phi\rangle \in \mathbb{C}^2$ represent two distinct physical qubits, we can represent the combined system as a single vector $|\psi\rangle \otimes |\phi\rangle$ in a larger complex Euclidean space $\mathbb{C}^2 \otimes \mathbb{C}^2 \cong \mathbb{C}^4$. In many cases it is customary to drop the tensor product \otimes symbol and write $|\psi\rangle |\phi\rangle$ or even $|\psi\phi\rangle$ when the underlying complex Euclidean spaces are understood. In the computational basis we can expand this tensor product as

$$|\psi\rangle \otimes |\phi\rangle = (\alpha |0\rangle + \beta |1\rangle) \otimes (\gamma |0\rangle + \delta |1\rangle) \quad (2.3)$$

$$= \alpha\gamma |00\rangle + \alpha\delta |01\rangle + \beta\gamma |10\rangle + \beta\delta |11\rangle \quad (2.4)$$

where $\alpha, \beta, \gamma, \delta \in \mathbb{C}$.

With the objects of the theory defined, we must now understand the dynamics, or choreography of the theory. As stated in section 2.1, we take the theory of quantum mechanics to be reversible, and hence any operation we perform on a qubit $|\psi\rangle$ must be undo-able. Thankfully linear algebra has just the tool to transform complex vectors in a reversible, and general way: unitary matrices!

Definition 2.2.1. An $n \times n$ complex matrix A is called unitary if

$$AA^\dagger = A^\dagger A = \mathbb{1} \quad (2.5)$$

where † denotes the conjugate transpose. The collection of unitary matrices form a group known as the unitary group and is denoted $\mathbf{U}(n)$. This definition can also be stated simply using set builder notation;

$$\mathbf{U}(n) := \left\{ A \in \mathcal{M}_n(\mathbb{C}) : AA^\dagger = \mathbb{1} = A^\dagger A \right\} \quad (2.6)$$

where $\mathcal{M}_n(\mathbb{C})$ is the set of all $n \times n$ complex matrices.

Hence when we have a qubit $|\psi\rangle$ and perform some action on it, the new state is modeled by $|\phi\rangle = U|\psi\rangle$ where U represents whatever action we performed. The condition shown in eq. (2.5) is quite restrictive: where a general $n \times n$ matrix has $2n^2$ real degrees of freedom, an element of $U(n)$ only has n^2 .⁴ In fact for a general element of $U(2)$ we can decompose it into pieces that look much more familiar.

Example 2.2.2. *Let A be an arbitrary element of $U(2)$. Then the following decomposition holds for $\alpha, \beta, \gamma, \delta \in \mathbb{R}$.*

$$A = e^{i\alpha} \begin{bmatrix} e^{-i\beta} & 0 \\ 0 & e^{i\beta} \end{bmatrix} \begin{bmatrix} \cos \gamma & -\sin \gamma \\ \sin \gamma & \cos \gamma \end{bmatrix} \begin{bmatrix} e^{-i\delta} & 0 \\ 0 & e^{i\delta} \end{bmatrix} \quad (2.7)$$

As we can see the middle matrix is simply a 2-dimensional rotation matrix, and the other two are of a simple diagonal form. Lastly we have the global phase $e^{i\alpha}$.

This is a particularly important example as the idea of decomposing unitary matrices into simpler pieces is something we will need heavily in circuit compilation tasks. This decomposition also shows that each unitary in $U(2)$ has a global phase ($e^{i\alpha}$ in example 2.2.2), which in quantum computation is often irrelevant as it is not experimentally measurable. For that reason we also often work in the following group where phases are removed.

Definition 2.2.3. *Define the projective unitary group by taking the quotient of the unitary group $U(n)$ by matrices of the form $\alpha \cdot \mathbb{1}$ where α is a unit length complex number. This is often written as follows, using a slight abuse of notation.*

$$\text{PU}(n) := U(n)/U(1) \quad (2.8)$$

Representative elements of the projective unitary group are the smallest physically realizable set of operations in quantum computation, and hence they make sense as our starting point in the formalization process.

2.2.2 Quantum Gates

A **quantum gate** is a physically realizable, and unambiguous mathematical transformation. More formally, a quantum gate on n qubits is an element $g \in \text{PU}(2^n)$. In this document we will mainly discuss quantum gates acting on 1 and 2 qubits as that is the capability of most modern hardware we will discuss in chapter 3. Table 2.1 outlines some of the common gates we will encounter throughout this document, and their associated notations both mathematically, and diagrammatically.

⁴ This is to say $\dim_{\mathbb{R}} U(n) = n^2$.

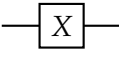
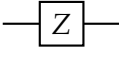
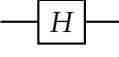
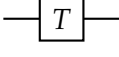
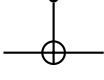
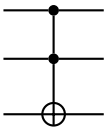
Name	Notation	Circuit Diagram	Matrix
Pauli X	X		$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$
Pauli Z	Z		$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$
Hadamard	H		$\frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$
$\frac{\pi}{8}$ -gate ⁵	T		$\begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}$
Controlled Not	CNOT		$\begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 0 & 1 \\ & & 1 & 0 \end{bmatrix}$
Toffoli	CCNOT		$\begin{bmatrix} 1 & & & & & & & \\ & 1 & & & & & & \\ & & 1 & & & & & \\ & & & 1 & & & & \\ & & & & 1 & & & \\ & & & & & 1 & & \\ & & & & & & 1 & \\ & & & & & & & 0 & 1 \\ & & & & & & & 1 & 0 \end{bmatrix}$

Table 2.1: Common Quantum Gates

Along with the examples in table 2.1 we have **parametric gates** which we view as gates that are dependent on some number of parameters, although we will often just use one. Parametric gates are modeled by functions $g : \mathbb{R} \rightarrow \text{PU}(2^n)$ and in all technicality are not gates in and of themselves, but rather a function whose images are gates. Two important parametric gates are the X and Z rotations.

$$R_X(\theta) = \begin{bmatrix} \cos(\theta/2) & -i \sin(\theta/2) \\ -i \sin(\theta/2) & \cos(\theta/2) \end{bmatrix} \quad R_Z(\theta) = \begin{bmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{bmatrix} \quad (2.9)$$

These two can also be represented more compactly as $R_X(\theta) = e^{-iX\theta/2}$ and similarly $R_Z(\theta) = e^{-iZ\theta/2}$ where X and Z are the Pauli operators as in table 2.1. Note that these two functions may seem to have period 4π , but due to the quotient structure on the image space $\text{PU}(2^n)$, all phases are modded out to ensure $R_X(\theta) = R_X(\theta + 2\pi)$.

Example 2.2.4. *As we will later see, due to the limited connectivity of qubits on modern hardware, the ability to move qubits around on a chip is paramount. While some hardware can physically move qubits, many cannot. In the latter case a strategy must be devised to perform some sort of swap operation between qubits using quantum gates. That is a gate $\text{SWAP} \in \text{PU}(2^2)$ is desired that acts on two qubit systems as*

$$\text{SWAP}(|\psi\rangle \otimes |\phi\rangle) = |\phi\rangle \otimes |\psi\rangle. \quad (2.10)$$

⁵ The name $\frac{\pi}{8}$ -gate comes from the way the gate was first introduced in the literature, where it was written $T = e^{i\pi/8} \begin{bmatrix} e^{-i\pi/8} & 0 \\ 0 & e^{i\pi/8} \end{bmatrix}$.

This operation as defined can be seen to be unitary by taking the conjugate transpose of eq. (2.10) and taking the forming the inner product again with eq. (2.10).

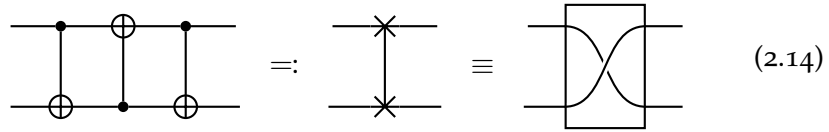
$$\langle \psi | \otimes \langle \phi | \text{SWAP}^\dagger \text{SWAP} | \psi \rangle \otimes | \phi \rangle = \langle \phi | \otimes \langle \psi | | \phi \rangle \otimes | \psi \rangle \quad (2.11)$$

$$= \langle \phi | \phi \rangle \langle \psi | \psi \rangle \quad (2.12)$$

$$= 1 \quad (2.13)$$

Since $|\phi\rangle, |\psi\rangle$ were arbitrary, we must have $\text{SWAP}^\dagger \text{SWAP} = \mathbb{1}$ and a similar argument can be used to show $\text{SWAP} \text{SWAP}^\dagger = \mathbb{1}$, and hence SWAP is a valid unitary operation.

As for the decomposition of the SWAP gate, we have the following equivalence.



Where the first equality shows us how to perform the swap with 3 CNOT gates, and the last equality is an equivalence of notation.

We can show this using the fact that the CNOT gate is defined to act as $\text{CNOT}[|x\rangle \otimes |y\rangle] = |x\rangle \otimes |x \oplus y\rangle$ where $x, y \in \mathbb{F}_2$ and \oplus is binary addition. With this we can explicitly compute the action of this circuit. Here we use the notation CNOT_b^a to mean a CNOT gate acting from qubit a (the control qubit) to qubit b (the target qubit). Then the 3 CNOT gates in eq. (2.14) act under the following manipulations.

$$\begin{aligned} |x\rangle \otimes |y\rangle &\xrightarrow{\text{CNOT}_2^1} |x\rangle \otimes |x \oplus y\rangle \\ &\xrightarrow{\text{CNOT}_1^2} |x \oplus (x \oplus y)\rangle \otimes |x \oplus y\rangle = |y\rangle \otimes |x \oplus y\rangle \\ &\xrightarrow{\text{CNOT}_2^1} |y\rangle \otimes |(x \oplus y) \oplus y\rangle = |y\rangle \otimes |x\rangle. \end{aligned}$$

Exactly as desired. The SWAP gate also has the following matrix representation in the computational basis.

$$\text{SWAP} = \begin{bmatrix} 1 & & & \\ & 0 & 1 & \\ & 1 & 0 & \\ & & & 1 \end{bmatrix} \quad (2.15)$$

2.2.3 Quantum Circuits

We are now ready to put these pieces together to build larger structures. Since it is common that a quantum computer can perform a multitude of gates, we collect them together to form a **quantum gate set**.

Definition 2.2.5. A quantum gate set is a (typically finite) subset $G \subsetneq \text{PU}(2^n)$. An element of G is called a quantum gate.

Just as we had gates and parametric gates in the previous section, some authors also like to define another set keeping track of said parametric gates. A **parametric quantum gate set** G' is a finite collection of parametric gates. While we only have a finite collection of parametric gates, this usually means an infinite amount of quantum gates. From these gates, we can construct a **quantum circuit** by applying a sequence of elements from the gate set.

Definition 2.2.6. Let G be a quantum gate set, and let G^* denote the set of finite length words over G (and the empty word which we take to mean identity).⁶ A quantum circuit is an element of G^* .

Thus if our gate set $G = \{a, b, c\}$, then the following are example circuits: $aacba$, $cccbba$, $cbbbab$, and ab .

Something to note here is that in this abstraction, all of our quantum gates are assumed to act on all qubits. With a 2 qubit quantum chip and the ability to perform a Pauli X gate on either qubit, our gate set is $\{\mathbb{1} \otimes \mathbb{1}, \mathbb{1} \otimes X, X \otimes \mathbb{1}, X \otimes X\}$.⁸ Sometimes this gate set is denoted $\{\mathbb{1}, X_0, X_1, X_0X_1\}$, but we will try to use more explicit notation here.

Circuits are often drawn as in fig. 2.1 where each horizontal “wire” represents a qubit, and boxes and other gadgets represent quantum gates. That said, the way our theoretical model sees this circuit is more

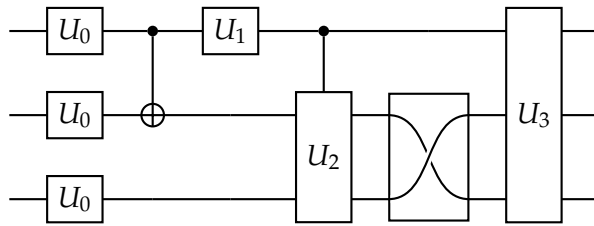


Figure 2.1: Example Quantum Circuit

like that of fig. 2.2 where each gate acts on the entirety of the qubits. In table 2.2 we see what each one of these circuits are under the hood,

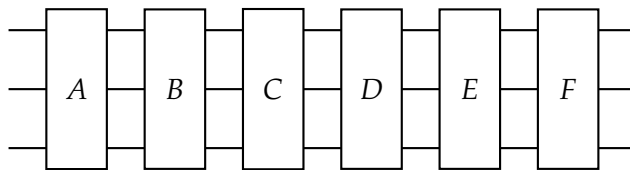


Figure 2.2: Abstract Quantum Circuit

⁶ This $*$ operation is known as the Kleene⁷ star.

⁷ Technically the author for which this operation is named after is Stephen Cole Kleene in which Kleene is pronounced *KLAY-nee*, yet most people say this operation as *clean* star.

⁸ We don't always think of the identity gate $\mathbb{1}$ as a gate that needs to be included, but doing nothing to a qubit is no easy task, so it's important to remember to treat it just like any other gate and understand its error rates as well.

and we can know that all of them are in the gate set for the above circuit.

Gate Name	Composition
A	$U_0 \otimes U_0 \otimes U_0$
B	$\text{CNOT} \otimes \mathbb{1}$
C	$U_1 \otimes \mathbb{1} \otimes \mathbb{1}$
D	Controlled- U_2
E	$\mathbb{1} \otimes \text{SWAP}$
F	U_3

Table 2.2: Gate Compositions

We now have the machinery for circuits, and one of the important questions we need to ask is *when are two circuits the same?* Surely we can compare the circuits as strings in G^* , but if $G = \{\mathbb{1}, X\}$, it will not tell us that $C = \mathbb{1}$ and $C' = XX$ are logically the same despite corresponding to different physical processes. To this end we wish to understand how the combinations of gates come together to form the entire process. Following [Amy19] we define a map $\llbracket - \rrbracket : G^* \rightarrow \text{PU}(2^n)$ which takes a quantum circuit, or sequence of gates, and multiplies them together to obtain a single unitary operator: $\llbracket g_1 g_2 \cdots g_m \rrbracket = g_m \cdot g_{m-1} \cdots g_1$.⁹ With this notation we can say two circuits C and C' implement the same operation if $\llbracket C \rrbracket = \llbracket C' \rrbracket$.

This formalism also allows us to frame the following important question about unitary synthesis.

Question 2.2.7. *Given a quantum gate set G over n qubits, and unitary operator $U \in \text{PU}(2^n)$, does there exist a circuit $C \in G^*$, such that $\llbracket C \rrbracket = U$?*

If the answer is yes, we say a gate set G **synthesizes** U . We also say that G synthesizes U if there is a collection SWAP operations that can be composed before and after G that implement U . More formally, if there is a circuit $C \in G^*$ and unitaries S_0 and S_1 that are made solely of SWAP operations such that $S_0 \cdot \llbracket C \rrbracket \cdot S_1 = U$, then we also say G synthesizes U . This additional level of equivalence is particularly useful in quantum circuit compilation as it comes from simply relabeling qubits.

This question is answered, at least in part, through the Solovay-Kitaev theorem first published in [Kit97] with further proofs/elucidations in [NC10; DN05; KSV02]. The theorem, stated in our terminology is as follows.

⁹ Notice here on the left we have string concatenation, and on the right matrix multiplication. Also note the fact that when doing the multiplication we reverse the order. This is an artifact of the way we draw quantum circuits from left to write, but apply gates mathematically right to left.

Theorem 2.2.8 (Solovay-Kitaev). *Let G be a quantum gate set on n qubits such that*

- $g^\dagger \in G$ for all $g \in G$, and
- the free group $\langle G \rangle$ is dense in $\text{PU}(2^n)$.

Then with $\varepsilon > 0$, there is a constant $c > 3$, such that for any $U \in \text{PU}(2^n)$, there exists a circuit $C \in G^$ of length $\mathcal{O}(\log^c(\frac{1}{\varepsilon}))$ that approximates U with error less than ε : that is $\|[[C]] - U\| < \varepsilon$.*

Not only does this theoretical result provide some insight into 2.2.7, but it's constructive and hence provides an algorithm¹⁰ to approximate arbitrary elements of $\text{PU}(2^n)$ using gates from $\text{PU}(2^m)$ for any $m \in [n]$. This was, a very important result in the field of quantum computing because it was the first to show that with the right gate set, one can theoretically perform any desired unitary.

With at least a partial answer to Question 2.2.7 we can begin to refine further questions. If the answer to 2.2.7 is positive, we can then ask the following.

Question 2.2.9. *If G synthesizes U , and if $f : G^* \rightarrow \mathbb{R}$ is a cost function, can we find*

$$C_{\min} = \arg \min_{C \in G^*} \{f(C) : [[C]] = U\}?$$

Some examples of common cost functions are given below, and multiple can be used in the case of tie-breaking.

- $f(C) = \text{length}(C)$ (commonly referred to as the depth of the circuit)
- $f(C) = \#$ of uses of a particular gate in C
- $f(C) = \text{duration}(C)$ (by this we mean the total elapsed time the circuit takes)¹¹

2.2.4 Universal Gate Sets

We slightly danced around the idea of universality in theorem 2.2.8, but we will make it clear now. In order to harness the full power of a quantum computer, it must be able to perform arbitrary unitary operations.

Definition 2.2.10. *A gate set G on n qubits is called universal if for all $U \in \text{PU}(2^n)$ there exists a circuit $C \in G^*$ such that $[[C]] = U$.*

¹⁰ This result sometimes goes under the name "The Solovay-Kitaev Algorithm".

¹¹ We have not discussed this yet, but each gate $g \in C$ takes a nonzero amount of time, during which the computation may be disturbed by outside forces.

If our gate set is not universal, then we can often find ourselves in a situation where it is more efficient to simulate a given quantum algorithm than to actually run it. E. g. circuits composed of gates from $\{\text{CNOT}, H, S\}$ are known to be efficiently simulable [AGo4] despite not limiting factors typically thought to make quantum computation more powerful such as entanglement.

The question of which gate sets are universal for quantum computation is important both for our theoretical understanding of quantum computation, but also for building physical devices. Some examples that have been shown to be universal are the following.

- CNOT plus $U(2)$ as shown in [Bar+95]
- CNOT, Hadamard, and the $\frac{\pi}{8}$ -gate as shown in [Boy+00]
- Toffoli, Hadamard, and the $\frac{\pi}{8}$ -gate squared as shown in [Kit97]
- CNOT plus any single qubit gate that does not preserve the computational basis and is not the Hadamard gate as shown in [Shio3]

2.3 FAULT TOLERANCE

Introduced in the context of quantum computation by Shor, the idea of fault tolerance is to make quantum computers that can perform meaningful computation despite decoherence and other errors [Sho96]. As it stands, even with basic quantum error correction quantum errors can spread and quickly become unwieldy. Suppose we have a general two-qubit state that we'd like to perform a CNOT gate on, but a bit-flip error occurs on the first qubit before the CNOT can be applied. This single-qubit error is then propagated to the second qubit as follows.

$$\alpha |00\rangle + \beta |01\rangle + \gamma |10\rangle + \delta |11\rangle \quad (2.16)$$

$$\xrightarrow{X \otimes 1} \alpha |10\rangle + \beta |11\rangle + \gamma |00\rangle + \delta |01\rangle \quad (2.17)$$

$$\xrightarrow{\text{CNOT}} \alpha |11\rangle + \beta |10\rangle + \gamma |00\rangle + \delta |01\rangle \quad (2.18)$$

Compare that with the effect of a CNOT on the general two qubit state in eq. (2.16).

$$\alpha |00\rangle + \beta |01\rangle + \gamma |10\rangle + \delta |11\rangle \quad (2.19)$$

$$\xrightarrow{\text{CNOT}} \alpha |00\rangle + \beta |01\rangle + \gamma |11\rangle + \delta |10\rangle \quad (2.20)$$

To solve this problem (and many others like it), fault tolerance encodes single qubits into many to increase information redundancy. Gates are then replaced by **gadgets** which implement the one and two qubit gates on the encoded logical qubits. To prevent errors from spreading, restrictions are placed on the number of two qubit inter-gates. Here inter-gates refer to gates within the encoded qubits, and the restrictions

are in place to ensure errors spread in only a limited capacity. Further explanation and details can be found in [Goto9].

2.4 MATHEMATICS

Before moving on there are a few more bits of mathematics we need to cover. All of our discussions in this section will assume our vector space V is some complex space \mathbb{C}^n .

2.4.1 Operator Norms

VECTOR INDUCED NORMS: Suppose our vector space V has an existing norm defined on it $\|\cdot\| : V \rightarrow \mathbb{R}$. This induces a norm on the space of operators [End \$V\$](#) as

$$\|A\|_{\text{vec}} := \max_{v \in V} \{\|Av\| : \|v\| = 1\}. \quad (2.21)$$

TRACE NORM:

$$\|A\|_{\text{tr}} := \text{tr}\left(\sqrt{A^\dagger A}\right) \quad (2.22)$$

FROBENIUS NORM:

$$\|A\|_{\text{F}} := \sqrt{\text{tr}(A^\dagger A)} = \left(\sum_{i,j \in [n]} |a_{ij}|^2\right)^{1/2} = \|\text{vec}(A)\| \quad (2.23)$$

2.4.2 Free Group

We will not attempt a rigorous definition of the free group and instead opt for something more informal since we will not need to work with the details. Let S be a finite set, and denote by S^{-1} the formal inverse of elements in S . Then the free group of S is $\langle S \rangle := (S \cup S^{-1})^*$ where the asterisk indicates the Kleene star.

As an example take S to be $\{f, g, h\}$, and hence the formal inverses are $S^{-1} = \{f^{-1}, g^{-1}, h^{-1}\}$. Then the free group $\langle S \rangle$ contains elements such as $fg^{-1}hhhh^{-1}$, $fghhhf$, and $h^{-1}gh^{-1}f^{-1}ghgg$. Note that this may appear very similar to definition 2.2.6, however we did not require our “words” to be over the inverses as we have here; only elements of the set itself.¹²

¹² This is done because in practice, being able to perform a quantum gate U , does not always imply one can perform its inverse U^\dagger . In fact, theorem 2.2.8 has been generalized without the need of unitary inverses in the approximation algorithm [BG21].

2.4.3 *Dense-ness*

What does it mean for a gate set G to be dense in $\text{PU}(2^n)$? It means that for every $U \in \text{PU}(2^n)$, and every $\varepsilon > 0$, we have a sequence of gates $C = g_1 g_2 \cdots g_m$ such that $\|C - U\| < \varepsilon$.

2.4.4 *Fidelity*

The **fidelity** of two density operators ρ and σ is a measure of state-similarity and can be calculated as follows.

$$F(\rho, \sigma) := \|\sqrt{\rho}\sqrt{\sigma}\|_{\text{tr}} = \text{tr}\left(\sqrt{\sqrt{\sigma}\rho\sqrt{\sigma}}\right). \quad (2.24)$$

If $\rho = \sigma$, then their fidelity is equal to 1, while if ρ and σ have orthogonal images (i. e. $\rho\sigma = 0$), then the fidelity is equal to 0. In all other cases the value of the fidelity lies in the range $(0, 1)$. This notion can be extended from quantum states to quantum gates to obtain a similarity measure for unitary matrices [HHH99]. Given two unitary operators U and V and a density operator ρ , we can compute $F(U\rho U^\dagger, V\rho V^\dagger)$, but the dependence on a particular state ρ is not ideal for understanding how U and V differ across *all* states. Hence we define the **average gate fidelity** between two quantum gates as

$$F_{\text{gate}}(U, V) := \int F(U\rho U^\dagger, V\rho V^\dagger) d\rho \quad (2.25)$$

where the integral is taken over all density operators which is made possible by a Haar measure. This measure has the downside of being “dimensionally unstable” which means $F_{\text{gate}}(U, V) \neq F_{\text{gate}}(U \otimes \mathbb{1}, V \otimes \mathbb{1})$. This can be amended by using the **process fidelity**¹³ which requires the introduction of a few other terms. Let $|\phi\rangle = \frac{1}{\sqrt{d}} \sum_{x=0}^{d-1} |x\rangle |x\rangle$ be the maximally mixed state on the tensor square of some complex Euclidean space \mathcal{H} . The Choi representation of a quantum channel, or Completely Positive Trace Preserving (CPTP) map $\Phi : \mathbb{C}^{n \times n} \rightarrow \mathbb{C}^{m \times m}$ is defined as follows.

$$J(\Phi) := (\mathbb{1} \otimes \Phi)(|\phi\rangle\langle\phi|) \quad (2.26)$$

The process fidelity is then defined as

$$F_{\text{proc}}(U, V) := \langle\phi| (\mathbb{1} \otimes V^\dagger) J(U) (\mathbb{1} \otimes V) |\phi\rangle \quad (2.27)$$

$$= \text{tr}(J(U)J(V)). \quad (2.28)$$

The process fidelity and average gate fidelity are linearly related [Nie02] where d is the dimension of the system by the relation

$$F_{\text{gate}} = \frac{d F_{\text{proc}} + 1}{d + 1}. \quad (2.29)$$

¹³ This notion goes by *mapping fidelity* in [Wat18], and *entanglement fidelity* in [NC10].

This definition is made more useful when we replace V by a noisy, error-prone implementation of U . This gives us a theoretical tool to examine the accuracy of an implementation of a desired unitary gate that is more relevant than a more naïve measure like the distance or trace norm.

Part II

BACK END

The goal of this part is to familiarize the reader with the realities of quantum hardware. This means understanding their architecture, strengths, weaknesses, and some of the many measures we have to quantify their effectiveness. With an understanding of the limitations of modern-day hardware we can understand the problem of quantum circuit compilation. We will show how the problem is both similar and different from classical compilation and how we can benefit from using existing classical infrastructure.

QUANTUM HARDWARE

The goal of this chapter is twofold. First, we introduce the most common constraints seen in modern quantum hardware, as well as other common tools used to measure the efficacy of a given quantum computer. Second, we will introduce the mathematical formalism needed in order to formulate the problems related to quantum circuit compilation we will see in chapter 4. We will not, however, attempt to give an introduction to the physical implementations of quantum hardware and instead refer the reader to [NC10, Chapter 7] for a more comprehensive introduction.

3.1 REQUIREMENTS

In 2000 David DiVincenzo proposed 5 requirements as being necessary to make an effective quantum information processing device [DiVoo]. His proposed requirements are summarized here.

1. A scalable physical system with well-characterized qubits.
2. The ability to initialize the state of the qubits to a simple fiducial state, such as $|0\rangle^{\otimes n}$.
3. Long decoherence times, much longer than the gate operation time.
4. A universal set of quantum gates.
5. A qubit measurement capability.

While all of these requirements are still under active research, requirements 2., 4. and 5. are completed for NISQ devices, while requirements 1. and 3. keep us in the NISQ-era. However, even if all of these problems were solved completely, there are still many things that can go wrong. Just because your qubits scale doesn't mean you have enough for a specific algorithm. Just because there is a long decoherence time doesn't mean a qubit can't error in some other way. Just because you have a universal gate set doesn't mean you know how to efficiently decompose a gate from the algorithm you are trying to run.

As you can see, these five requirements provide us with the backbone upon which we can build further, but do not guarantee optimal quantum computations. It is some of these secondary questions we wish to understand more deeply to make quantum computers more useful once the bedrock has been established.

3.2 QUANTUM CHIPS

Intuitively a quantum chip is a collection of qubits along with the capability to perform operations on subsets of the qubits. This can be formalized using a graph structure as follows.

Definition 3.2.1. A quantum connectivity graph is an undirected¹ graph $H = (V, E)$ ² such that $(v, v) \in E$ for all $v \in V$.

What we call the connectivity graph is sometimes referred to as a (network) topology in other resources. We ensure the connectivity graph has all self loops as edges are the basis for performing quantum gates and all modern hardware has the capability of performing single qubit gates. Once the connectivity graph has been established, we can consider which gate-sets are allowable by ensuring only qubits which are connected via an edge are acted on in a nontrivial manner. That is, if two nodes are not connected via an edge, there should be no entangling gates operating on them.

In order for an effective definition for a quantum connectivity graph we must first define the following qubit indexing function.

Definition 3.2.2. Let G be a quantum gate set acting on n qubits. Define a function $\text{qubits} : G \rightarrow \mathcal{P}([n])$ which returns a set containing the index of the qubits each gate acts on nontrivially.

As an example, if $G = \{\mathbb{1} \otimes \mathbb{1}, \mathbb{1} \otimes X, X \otimes \mathbb{1}, X \otimes X\}$, then $\text{qubits}(\mathbb{1} \otimes \mathbb{1}) = \emptyset$, $\text{qubits}(\mathbb{1} \otimes X) = \{1\}$, and $\text{qubits}(X \otimes X) = \{0, 1\}$.

Definition 3.2.3. Let $H = (V, E)$ be a quantum connectivity graph. A gate set G is said to be amenable to H if

- G acts on $|V|$ qubits, and
- $\text{qubits}(g) \in E$ for all gates $g \in G$.

We can now combine the connectivity graph and an amenable gate set to form the model of quantum hardware.

Definition 3.2.4. A quantum chip $T = (H, G)$ is a quantum connectivity graph $H = (V, E)$ together with an amenable gate set G . The gates $g \in G$ are often called native to T .

While this formalism does have the drawback of restricting our gate sets to single and two qubit gates, this model applies to the majority of hardware today. Multi-qubit gates can be allowed using the notion of a hypergraph in definition 3.2.1, but doing so introduces complexity

¹ There are some hardware which are better modeled by a directed graph, and we will see an example in section 4.1, but for most cases undirected is simpler and provides the intuition.

² V is a finite set which we refer to as vertices, and E is a collection of pairs of vertices, i.e. $E \subseteq V \times V$.

without a clear advantage. This slightly simplified notion still encompasses universal quantum computation as three (and higher) qubit gates are not needed as we saw in section 2.2.4.

As an example, the connectivity graph of IBM's 7-qubit quantum computer `ibmq_jakarta` shown in fig. 3.1 and the gate set is as follows [IBM].

$$\{\text{CNOT}, \mathbb{1}, R_Z, S_X, X\} \quad (3.1)$$

Vertex "3" being connected to "1" means that we can apply a 2-qubit unitary targeting both of those qubits, however the hardware does not support 2-qubit gates between qubits "2" and "6" natively.

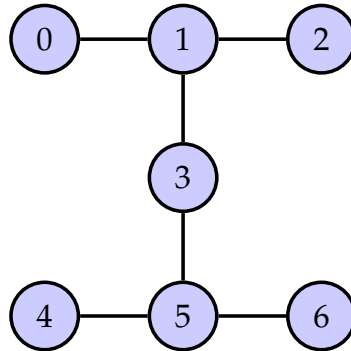


Figure 3.1: IBMQ Jakarta Architecture

Definition 3.2.5. Let $T = (G, H)$ be a quantum chip with graph $H = (V, E)$, and $C \in G^*$ be a circuit. The quantum chip T can run C if for all $g \in C$ we have $\text{qubits}(g) \in E$. In this case we say that C is executable on T .

We can now define the main problem of quantum circuit compilation: that of the qubit mapping problem.³

Question 3.2.6. Let $C \in A^*$ be a circuit over quantum gate set A , and $T = (B, H)$ a quantum chip. Is there a T -executable circuit $C' \in B^*$ such that $\llbracket C' \rrbracket = \llbracket C \rrbracket$?

In the case when the number of qubits required in C is greater than the number of vertices in the connectivity graph, the answer is no.⁴ On the other hand when the number of qubits required for C is fewer than the number we have access to $|V|$, then the answer is yes, provided

1. B is a universal gate set, and
2. (V, E) is connected.⁵

³ This sometimes also goes by the name of the qubit routing problem, or qubit scheduling problem although sometimes these mean slightly different things.

⁴ There are however specific cases when this *is* possible. If the algorithm only requires $n < |V|$ qubits to be entangled at once, there are clever scheduling tactics one can employ to implement such an algorithm. There are also "quantum autoencoders" which attempt to implement compressed versions of circuits on smaller numbers of qubits [ROA17].

⁵ That is for any two vertices, there is a path between them.

Just as the unitary synthesis problem (question 2.2.7) was turned into an optimization problem by the use of cost functions (question 2.2.9), we can ask for the optimal version of question 3.2.6.

Question 3.2.7. Let $T = (B, H)$ be a quantum chip and $C \in B^*$ be a T -executable circuit that implements $C' \in A^*$ (i.e. $\llbracket C \rrbracket = \llbracket C' \rrbracket$). Let $f : B^* \rightarrow \mathbb{R}$ be a cost function. Can we find

$$C_{min} = \arg \min_{C \in B^*} \{f(C) : \llbracket C \rrbracket = \llbracket C' \rrbracket \text{ and } C \text{ is } T\text{-executable}\} \quad (3.2)$$

Despite formulating the key problem we'd like to understand in this document, there are many contributing factors that effect solutions to this problem. For that reason we need to not just understand a theoretical model of quantum hardware, but some of the implementation details as well.

3.3 HARDWARE SPECIFICATIONS

Here we will briefly cover the most important topics discussed in quantum circuit compilation when it comes to optimizations on NISQ-era hardware.

PARALLELIZABILITY Just as many classical algorithms can be sped up with parallelization, so too can many quantum algorithms. That said some architectures, such as ion traps and cold atoms, do not easily support parallelization while superconducting quantum computers do. Since this is typically a fact compilers can exploit to speed up computation, the non-parallelizability can sometimes means different compilation techniques must be employed, or just skipped entirely. This is especially important since gates are often grouped based on non-overlapping sets of qubits.

3.3.0.1 Relaxation and Dephasing Times

As qubits are two-state systems, they are often implemented experimentally using some physical system (e.g. an atom) that has a ground state, and an excited state. Excited states often have a tendency to “decay” into ground states, especially so when interacting with the environment. Hence we define the **relaxation time**.⁶ T_1 as the lifetime for the state $|1\rangle$ decaying into $|0\rangle$. This value can be experimentally found using the following methodology.

1. Prepare the state $|0\rangle$
2. Apply a Pauli X gate to obtain $|1\rangle$

⁶ This value also goes by the following names: coherence time, amplitude damping, longitudinal coherence time, spin lattice time, and spontaneous emission time.

3. Wait some time t (during this time the qubit may decay into $|0\rangle$)
4. Measure the qubit

Each time we measure the qubit in the ground state we record the amount of time t we waited. This process is then modelled with an exponential decay of the form e^{-t/T_1} .

The second important factor we need to understand is the **dephasing time**⁷ This time, instead of watching for the the bit flip from $|1\rangle$ to $|0\rangle$ we will watch for a phase flip from $|+\rangle$ to $|-\rangle$ via the following procedure.

1. Prepare the state $|0\rangle$
2. Apply a Hadamard H gate to obtain $|+\rangle$
3. Wait some time t (during this time a phase might appear on either qubit)
4. Apply another Hadamard H
5. Measure the qubit

Again, this experiment is modeled by an exponential decay with lifetime which we denote T_2 . This decoherence time is a measure of how quickly a superposition ($|+\rangle$) will decay into a classical mixture. In both the definition of T_1 and T_2 step 3 requires the experimenter to “wait”, meaning apply identity gates until time t . That said while the waiting occurs, if *other* qubits are acted upon, this may change the experimental results due to crosstalk (section 3.4). Since T_1 is a measure of how robust the qubit is against bit flips, and T_2 is a measure of how robust the qubit is against becoming probabilistic, these two quantities are important metrics to track the progress of quantum computers.

3.3.0.2 Quantum Volume

While transistor count has long served as an effective single number metric for the power of classical computers, qubit count does not have the same descriptive power due to quantum computers’ difference in connectivity, and error rates. In attempt to devise a single number metric effective in quantifying a quantum computers capabilities Cross, Bishop, Sheldon, Nation, and Gambetta introduced the notion of **quantum volume** which takes into account the number of qubits, connectivity, gate and measurement errors, and crosstalk. While understanding the full method to measure a quantum computer’s quantum volume is beyond the scope of this document, the

⁷ Again, this value also goes by the following names: phase coherence time, phase damping, spin-spin relaxation time, transverse coherence time, and elastic scattering time.

process consists of applying randomized circuits shown in fig. 3.2 where π is a permutation of the qubits, and $\text{PU}(4)$ denotes a random two-qubit gate. After the gates are applied a measurement is performed and the resulting bit-string is stored, and the process is repeated many times. A statistical analysis is then run to compare the computers performance with an ideal implementation of random circuits of this form. The largest quantum volume achieved to date is 1024 and was done by Honeywell’s System Model H1; a 10-qubit trapped-ion computer [Sol21].

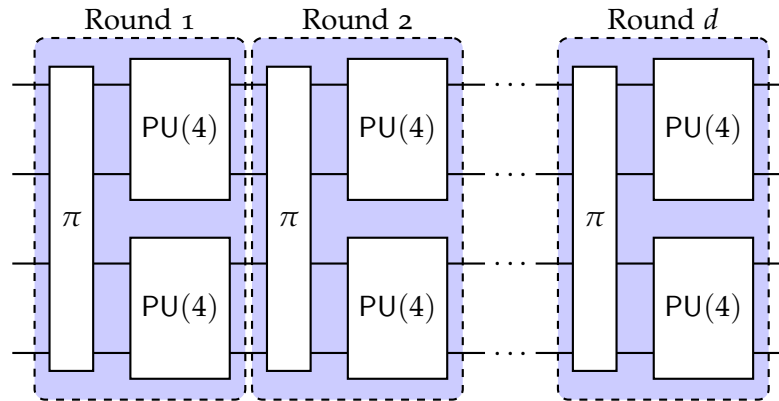


Figure 3.2: Quantum Volume Protocol

3.3.0.3 Gate Duration

As qubits are finnickly beasts that don’t want to retain their quantumness, how quickly we can perform gates is a very important measure and one tracked across many quantum computers. This measure usually comes under the guise of Circuit Layer Operations per Second (CLOPS) first introduced in [Wac+21].

3.4 ERRORS

Errors are ubiquitous in quantum computing and for the near future there is almost certainly no getting around them. Not only are errors abundant, but they can vary across the chip, and they can vary in type. The first error that is often encountered is that of **gate errors**. Some examples of gate errors might be

- performing $R_X(\theta + \varepsilon)$ when you intended to do $R_X(\theta)$, or
- performing $H + \varepsilon X$ when you intended to apply H .

This first type of error is sometimes mitigated experimentally if ε is either fixed, or coupled in some way to θ . However it may be the case that a more complex coupling is taking place dependent on the surrounding state of the qubit that the gate is acting on. Since we

represent a quantum chip as a graph, one way to quantify errors is to attach a number to each node and edge. The node error rate represents the computers error rate on performing a single qubit unitary, and the edge represents the computers error rate for performing a 2-qubit unitary.

The next type of error that can be introduced into a quantum computation is through **State Preparation and Measurement (SPAM)** errors. These—as you might have guessed—are introduced during state preparation and measurement. Despite these being one of the largest sources of errors on modern quantum hardware, quantum circuit compilation cannot aid in mitigating these errors.

Finally, the last major source of noise that is seen in quantum computers is that of **crosstalk**. Crosstalk corrupts information in our system when multiple gates are performed simultaneously. This is unfortunate as parallelizing computation drastically decreases the runtime and keeps the total computation time below decoherence times discussed in section 3.3.0.1. These errors arise as qubits are not perfectly isolated from each other and hence can interact especially when control pulses (i. e. the gate implementations) bleed into nearby qubits.

CIRCUIT COMPILERS

We can now return to the topic of compilers. It should now be clear that the level of abstraction we work at when designing quantum algorithms (i. e. quantum circuits possibly with some some classical computation mixed in) is much higher than the capabilities of our current, and likely near-future hardware. Hence, just as we saw in chapter 1, we are in need of a tool to translate this description down to a lower level of abstraction that embodies the restrictions of the hardware. As in fig. 1.3 which detailed the phases of a compiler, there are syntax and semantic analyses that are performed to ensure circuits are well formed, but we will not go any further into this topic here. The most interesting, and complicated portions of circuit compilation occur in transforming a circuit to an IR, optimizing it, and generating machine level instructions. Naïvely this is three phases, but because current quantum hardware is so restrictive this can often be broken down into the following four phases.

1. Conversion of quantum algorithm to a Quantum Intermediate Representation (QIR).
2. Optimization of the QIR.
3. Compilation of the QIR to a specific quantum chip, resulting in an instruction set.
4. Optimization of the instruction set.

This is reflected in the following diagram.

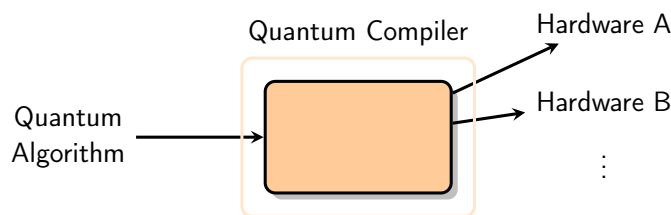


Figure 4.1: Action of Quantum Compiler

This reflects the structure of a classical compiler very closely in part because the phased approach works well, but as we will see later it suits our needs well for hybrid quantum-classical computations that are expected to be the dominant near-term use of quantum computers. This approach also allows the design of components to be easily reused just as we saw with classical compiler in fig. 1.4. A similar figure can

be drawn for some of the many players in the quantum landscape and can be seen in fig. 4.2.

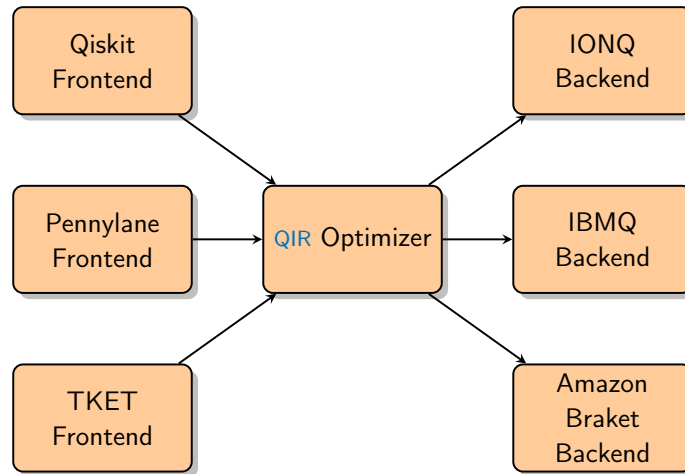


Figure 4.2: Modularity of Quantum Compiler

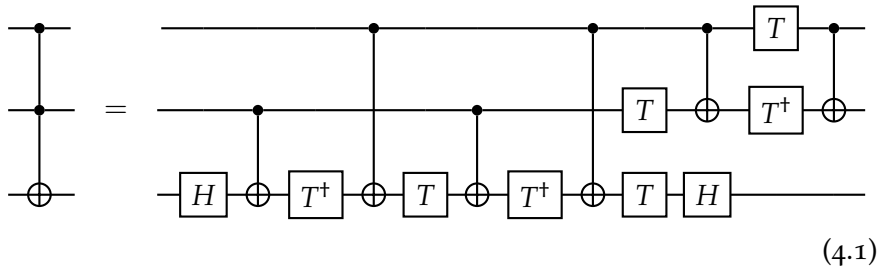
One of the benefits of the modular compiler structure seen in fig. 4.2 is that once the optimizer is made, backends can be written as new hardware arrive, *and* a backend can be written to take the circuit to a classical CPU. In effect what this provides is an optimized quantum simulator.

Many proposals for a QIR are built on top of the LLVM IR because of the success it has had in classical computing. In particular the QIR Alliance [QIR21] has been formed in order to formalize a specification for a QIR that will describe quantum and classical computation. This project has already had some success as a Multi-Level Intermediate Representation (MLIR) has already been made that lowers into the LLVM IR in a way that is adherent to the QIR specification put forth [MN21]. As we will see in section 4.2 many near-term applications of quantum computers will use quantum computers as a coprocessor of information, rather than operating independently. Thus having a unified IR that is capable of describing quantum and classical computation is compulsory. This reinforces the benefits of building a QIR on top of an existing IR.

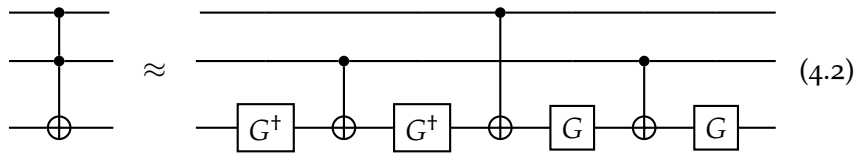
FAULT TOLERANCE As we saw in section 2.3, fault tolerance is a key method for encoding qubits and gates to prevent the spread of errors in a quantum circuit. This is done by restricting where entangling gates can be applied. Thus when compiling a fault tolerant circuit, the compiler needs to understand not only the restrictions that may be in place due to the quantum chips connectivity, but *also* the entangling gate restriction that fault tolerance places on the circuit. Not only this, but it is hoped that we may also be able to use compilers to take

circuits and compile them *into* a fault tolerant form if the quantum chip allows for it.

Example 4.0.1 (Compiling the Toffoli Gate). *Since most hardware are not capable of 3 qubit operations we must decompose the Toffoli gate into something more manageable. This is typically done using CNOT's, Hadamard's (H), and $\pi/8$ (T) gates [NC10].*



This is an important decomposition as the CCNOT gate appears in the modular exponentiation problem which is a core part of Shor's factoring algorithm [Sho94]. Hence if there are smaller decompositions than shown above that would be ideal as one CCNOT gate becomes 14! Barenco et al. show a more compact decomposition of CCNOT using only 3 CNOT gates if the phase of one of the qubits is allowed to change [Bar+95]. Let $G = R_Y(\frac{\pi}{4})$ in the following circuit.



However the question of "how many CNOT gates does it take to decompose a CCNOT?" was answered in 2009 when it was shown that a true equality preserving decomposition requires a minimum of 6 CNOT gates [SM09].¹

4.1 COMPILING ON A RING

In this section we will see an example that will take us through some of the many difficulties one might face while attempting to come up with a general purpose algorithm/method for compiling quantum circuits. This example is drawn from [Cow+19] with modifications.

To begin, suppose we'd like to run the quantum circuit shown in fig. 4.3. The first step we can take is to compress the diagram into a fewer number of layers. To do this we group operations on nonoverlapping qubits since they can be performed at the same time.² This is vital as decoherence times (section 3.3.0.1) are so short. This "compressed" version of the circuit is seen in fig. 4.4.

¹ This result shows that a minimum of 6 CNOT gates must be used, if they are being used. Other decompositions not using CNOT gates might still be more compact.
² This is not always an option as some implementations of quantum hardware (e.g. trapped ion), and hence the grouping might not be as compact.

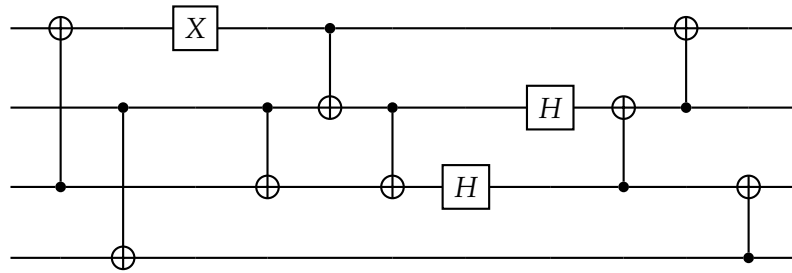


Figure 4.3: Circuit to be compiled

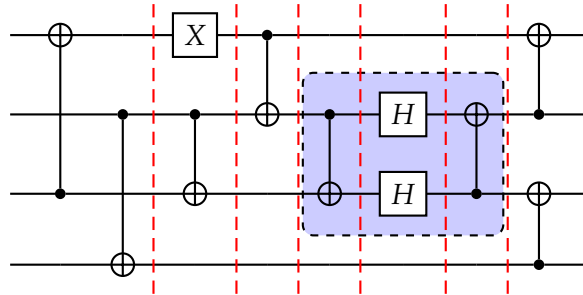


Figure 4.4: Circuit after compression

We can now apply a type of “device independent optimization” known as “peephole optimization” just as we saw in section 1.2.2, using the fact that $\text{CNOT}_1^2 \cdot (H \otimes H) \cdot \text{CNOT}_2^1 = H \otimes H$. This minor optimization, and many others can be found in [Siv+20]. Hence we can drop the two CNOT gates in the blue box to obtain the figure seen in fig. 4.5.

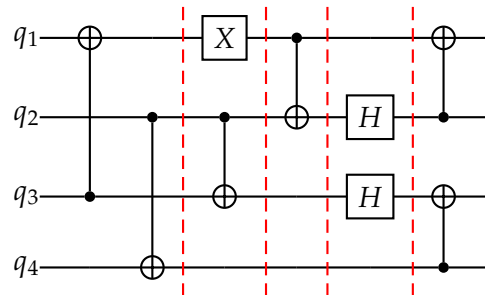


Figure 4.5: Circuit after peephole optimization

To continue with the problem we must now choose hardware we would like to run this circuit on. As the section title suggest, we will be choosing a qubit network topology of a ring. The first problem we need to tackle is placing the qubits from the circuit onto the ring. The first slice of the circuit contains CNOTs connecting $q_1 \leftrightarrow q_3$ and $q_2 \leftrightarrow q_4$ so placing them together to prevent additional SWAPs from being added is the first task. There are many configurations to satisfy

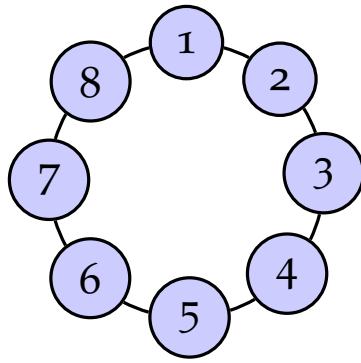


Figure 4.6: Ring Topology

this, but only one³ that satisfy the requirements that no SWAP gates are added in the second slice as well! That mapping is

$$q_1 \rightarrow 1 \quad q_2 \rightarrow 3 \quad q_3 \rightarrow 2 \quad q_4 \rightarrow 4. \quad (4.3)$$

Hence the first two slices of the circuit can be computed without any additional SWAP gates being added.

Executing the gates in slice 3 however will require a SWAP as qubits q_1 and q_2 are no longer adjacent. To make these qubits adjacent we can either swap qubits q_1 and q_3 or q_2 and q_3 . Looking ahead to slice 5 we see we need adjacency of $q_1 \leftrightarrow q_2$ and $q_3 \leftrightarrow q_4$. Swapping q_1 and q_3 would mean two additional SWAP gates before slice 5, but swapping q_2 and q_3 leaves the qubits in their desired positions for slice 5. Hence our compiled circuit in its final form:

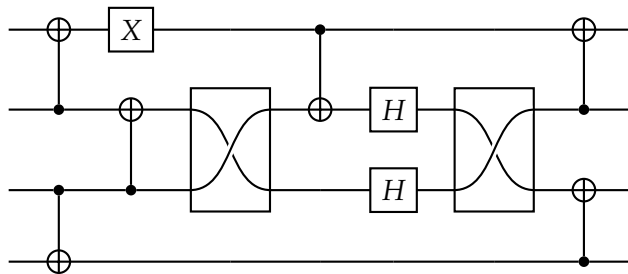


Figure 4.7: Compiled Circuit

If the quantum chip has the further restriction that its network topology is a directed graph and all the edges point clockwise, we can no longer use the typical SWAP decomposition we are used to as in eq. (2.14). Instead we must use

$$\begin{array}{c} \oplus \\ | \\ \bullet \end{array} = \begin{array}{c} \text{---} [H] \text{---} \bullet \text{---} [H] \text{---} \\ | \\ \text{---} [H] \text{---} \oplus \text{---} [H] \text{---} \end{array} \quad (4.4)$$

³ Modulo ring rotations/reflections.

in eq. (2.14) to decompose SWAP using only CNOT gates that go in one direction.

$$\text{SWAP} = \text{CNOT}_{12} H_1 \text{CNOT}_{21} H_2 \text{CNOT}_{12} \quad (4.5)$$

With this addition the compiled circuit begins to grow very quickly (fig. 4.8)

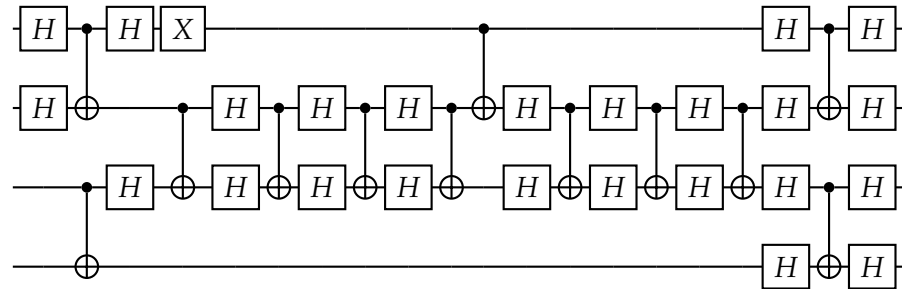


Figure 4.8: Compiled Circuit on Directed Ring

While this was a relatively simple example of some of the tasks a circuit compiler must complete, it did not begin to touch on the problem of gate decomposition or unitary synthesis (question 2.2.9). In the above example all gates applied were taken to be native to the hardware.

4.2 METHODS

Current research on compilation methods can be benchmarked in many ways, and compilation techniques often arise to improve on a given benchmark. Benchmarks are typically performed with respect to the most prominent compiler, which at the time of writing seems to be that of IBM's Qiskit [ANI+21]. Just as we saw in section 1.2.1 many of the subproblems required to be solved in classical compilation are NP-complete, or more difficult. Unfortunately the situation seems as bad in quantum compilation as the problem of assigning logical qubits to physical ones is equivalent to the subgraph isomorphism problem which is known to be NP-complete. Again finding the optimal number, and position, of SWAP gates is equivalent to another problem known to be at least NP-hard. Thus, as before we must look to heuristic solutions.

The procedure we encountered in section 4.1 was loosely based on methods proposed in [Cow+19] where first a circuit is sliced by timesteps, an initial mapping of qubits is made to the connectivity graph, routing gates from the original circuit onto the new architecture is performed, finally ending with gate synthesis for the gates

that the quantum chip may not support.⁴ In [Nan+21] considers solely the problem of finding optimal solutions to the qubit assignment, and routing problems. Despite this being an NP-complete problem the authors make the simplifying assumption that the circuit is already decomposed into one and two qubit gates that are native to the hardware. The problem is then encoded via a complex integer programming problem, with similarly encoded cost functions such as minimizing the total error rate, minimizing circuit depth, and minimizing crosstalk. Once encoded, the optimization problem can then be solved by one of the many integer programming libraries. The authors report a decrease in CNOT gates and higher fidelity when run on real hardware when compared to Qiskit’s compiler.

A slightly different approach is taken through [LDX19; Mur+19; Niu+20] where compilers are designed specifically for NISQ-era devices. In particular [Mur+19; Niu+20] use calibration data collected from hardware to inform the compilation process. This means if a particular qubit has a very high error rate, the compiler attempts to route computation around it, or use it as infrequently as possible. This allows the compiler to generate circuits optimized for the hardware at particular times of day as calibration data changes intra-day.

Deep reinforcement learning has also made its way into quantum circuit compilation in attempt to perform unitary synthesis [Mor+21]. This approach is well-suited for real-time quantum computation where the additional time required to compile a circuit is unavailable and hence a more immediate solution is required.

In the following two paragraphs we will see examples of how compilers can use knowledge about a general problems circuit/solution to improve compilation methods.

QAOA The Quantum Approximate Optimization Algorithm (QAOA) is a combinatorial optimization algorithm that is intended to be run on NISQ-era devices [FGG14]. Focusing in this particular problem, a 23% reduction in gate count, and 53% reduction in circuit depth was achieved [AAG20]. In the future we might hope to build these problem-specific compilers into a more general purpose one that can diagnose and understand when to use problem-specific compilers on demand.

VQE Another hybrid quantum-classical algorithm that has seen much attention due to its near-term applications in quantum chemistry is that of the Variational Quantum Eigensolver (VQE) [WHT15; Per+14]. This algorithm is used to calculate the ground state of a molecular Hamiltonian using a parametrized quantum circuit as a cost function, and the classical compute nodes as an optimizer. E. g. let $\theta \in \mathbb{R}^n$ be a vector of numbers that our circuit U depends on, i. e.

⁴ Followed by peephole optimizations if they are available.

$U : \mathbb{R}^n \rightarrow U(2^m)$ for some number of qubits m . A compiler specific

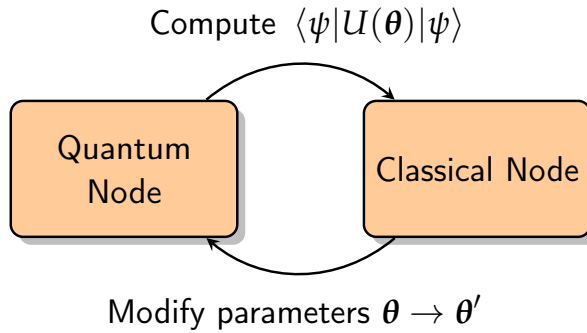


Figure 4.9: VQE Schematic

to this problem has been created, and generalized to other quantum-classical algorithms [Kha+22] leveraging much of the existing infrastructure brought forth by the LLVM project discussed in section 1.3. This allows the classical optimizations to be handled by the robust LLVM system, while using new circuit compilation techniques that take advantage of the fact that variational circuits have a particular form. The structure of variational circuits has been further taken advantage of by pre-compiling specific blocks of gates which resulted in 1.5–3 times improvement over existing systems [Gok+19].

CROSSTALK Due to crosstalk’s prevalence on nearly all hardware, compilers have been developed to mitigate this problem by utilizing both commutation identities and physical gate timing [XZZ21; Mur+20].

4.2.1 Verification

While retaining the semantic meaning of a circuit is one of the highest priorities during circuit compilation, it is possible it has changed. Thus, just as chip manufacturers use verification techniques to ensure electronics are built to specification, circuit compilation can also benefit from such techniques. With smaller circuits, it’s possible to ensure the correctness of compilers by simulation, but this is not a scalable approach to due to the inherent complexities in simulating quantum systems. To this end various methods have been developed such as formal proof [RPZ18], diagrammatic methods [DL13], equivalence checking [YM10] and functional verification [Amy18]. There have also been circuit optimizers written in formal languages like Coq [Tea22] using the semantics of matrices to only perform optimizations it has formally verified to be correct [Hie+21].

4.3 QUANTUM STACK

A “full quantum stack” is a collection of tools and components required to make a fully functioning quantum computer. As we’ve seen in this thesis, a compiler is an important, but single piece of this stack. In this section we will give an overview of *some* of the many existing tools and technologies that exist. For a more exhaustive of productionized list of software projects, visit the following webpages.

https://wikipedia.org/wiki/quantum_programming

<https://github.com/desireevl/awesome-quantum-computing>

Before diving into examples of existing infrastructure, it is useful to have a picture of what a quantum full stack may look like. There are many possibilities (each with different amounts of complexity), but a basic example is shown in fig. 4.10.

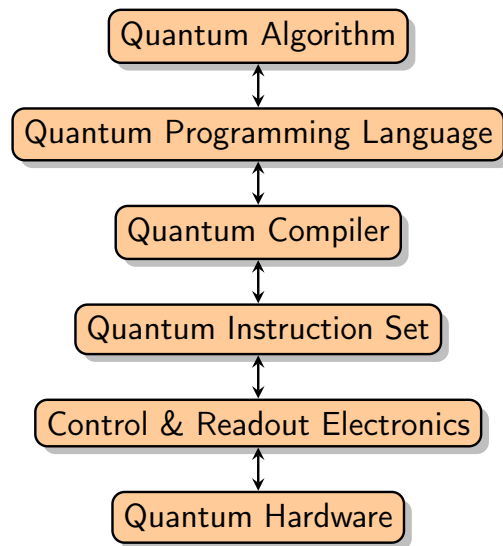


Figure 4.10: Quantum Stack

Starting at the top, we’ve seen the common language for specifying quantum algorithms is typically that of a circuit diagram. This is slightly complicated when there is quantum *and* classical coprocessing required in which case quantum circuits are often used as subroutines within classical pseudocode.

With a specification of an algorithm, one can then code this into a computer programming language using Qiskit (IBM) [ANI+21], PennyLane (Xanadu) [Ber+18], Q# (Microsoft) [Qua], Cirq (Google) [Dev21] and many others. A more detailed overview of the capabilities of each language along with the trade-offs can be found in [LaR19]. However, one example not discussed there is OpenQASM [Cro+17; Cro+21] which doesn’t fit as neatly into fig. 4.10. OpenQASM was first introduced to be a language used for specifying, and then drawing

quantum circuits. It has since grown to be a low-level language that can handle hybrid quantum-classical computation and can serve as both a quantum programming language and quantum instruction set.

Moving on to compilers, there are many examples of research compilers (like those we saw in section 4.2), but there is also the compiler built into Qiskit along with `qcor` [McC+21] which is a C++ compiler for hybrid quantum-classical computing built on `clang` (C++ compiler mentioned in section 1.2.3). There is also ScaffCC is a a scalable compilation and analysis framework based on LLVM [Jav+15; Lit+20]. `staq` [AG20] is a compiler specifically designed to compile programs written in OpenQASM.

Post-compilation the quantum algorithm exists in the form of a QIR, or quantum instruction set. In 2020 Microsoft introduced a QIR [QIR21] which has started to see some adoption among compilers, and in [SCZ16] the authors introduced the `quil` language [Smiz0] which is intended to serve the single purpose of being a quantum instruction set.

Finally the electronics and hardware play the foundational aspect in fig. 4.10, but are not the focus of this thesis and instead recommend [LaR19] which details some hardware currently available to internet users.

CONCLUSION

In this document we have covered the basics needed to understand the problem of quantum circuit compilation. In doing so we have introduced the prerequisite ideas from classical computing such as compilers, and the basics of computer architecture insofar as to understand the necessity of compilers. With a brief introduction to both quantum computation and the capabilities/limitation of modern-day quantum hardware we formalized one of the main questions in circuit compilation: that of the qubit mapping problem. We also covered multiple secondary questions that arise in circuit compilation, along with detailing current research to solve these problems. Finally we introduced some of the many tools under current construction to aid in the creation of a fully functioning quantum stack.

BIBLIOGRAPHY

- [AAG20] Mahabubul Alam, Abdullah Ash-Saki, and Swaroop Ghosh. “Circuit Compilation Methodologies for Quantum Approximate Optimization Algorithm.” In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, Oct. 2020. DOI: [10.1109/micro50266.2020.00029](https://doi.org/10.1109/micro50266.2020.00029) (cit. on p. 43).
- [AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. en. Cambridge, England: Cambridge University Press, Apr. 2009 (cit. on p. 4).
- [AG04] Scott Aaronson and Daniel Gottesman. “Improved simulation of stabilizer circuits.” In: *Physical Review A* 70.5 (Nov. 2004). DOI: [10.1103/physreva.70.052328](https://doi.org/10.1103/physreva.70.052328) (cit. on p. 22).
- [AG20] Matthew Amy and Vlad Gheorghiu. “staq—A full-stack quantum processing toolkit.” In: *Quantum Science and Technology* 5.3 (June 2020), p. 034016. DOI: [10.1088/2058-9565/ab9359](https://doi.org/10.1088/2058-9565/ab9359) (cit. on p. 46).
- [Aho+07] Alfred V. Aho, Jeffrey D. Ullman, Ravi Sethi, and Monica S. Lam. *Compilers: Principles, techniques & tools*. 2nd ed. Boston, MA, USA: Pearson Addison-Wesley, 2007 (cit. on pp. 4, 5, 8).
- [Amy18] Matthew Amy. “Towards Large-scale Functional Verification of Universal Quantum Circuits.” In: (May 2018). arXiv: [1805.06908 \[quant-ph\]](https://arxiv.org/abs/1805.06908) (cit. on p. 44).
- [Amy19] Matthew Amy. “Formal Methods in Quantum Circuit Design.” PhD thesis. Feb. 2019. URL: <http://hdl.handle.net/10012/14480> (cit. on p. 20).
- [ANI+21] MD SAJID ANIS et al. *Qiskit: An Open-source Framework for Quantum Computing*. 2021. DOI: [10.5281/zenodo.2573505](https://doi.org/10.5281/zenodo.2573505). URL: <https://github.com/Qiskit/qiskit> (cit. on pp. 42, 45).
- [Bar+95] Adriano Barenco et al. “Elementary gates for quantum computation.” In: *Phys. Rev. A* 52 (5 Nov. 1995), pp. 3457–3467. DOI: [10.1103/PhysRevA.52.3457](https://doi.org/10.1103/PhysRevA.52.3457) (cit. on pp. 22, 39).
- [Ben+93] Charles H. Bennett et al. “Teleporting an unknown quantum state via dual classical and Einstein-Podolsky-Rosen channels.” In: *Physical Review Letters* 70.13 (Mar. 1993), pp. 1895–1899. DOI: [10.1103/physrevlett.70.1895](https://doi.org/10.1103/physrevlett.70.1895) (cit. on p. 14).

- [Ben73] C. H. Bennett. “Logical Reversibility of Computation.” In: *IBM Journal of Research and Development* 17.6 (1973), pp. 525–532. DOI: [10.1147/rd.176.0525](https://doi.org/10.1147/rd.176.0525) (cit. on p. 13).
- [Ben80] Paul Benioff. “The computer as a physical system: A microscopic quantum mechanical Hamiltonian model of computers as represented by Turing machines.” In: *Journal of Statistical Physics* 22.5 (May 1980), pp. 563–591. DOI: [10.1007/bf01011339](https://doi.org/10.1007/bf01011339) (cit. on p. 13).
- [Ber+18] Ville Bergholm et al. “PennyLane: Automatic differentiation of hybrid quantum-classical computations.” In: (Nov. 2018). arXiv: [1811.04968](https://arxiv.org/abs/1811.04968) [quant-ph] (cit. on p. 45).
- [BG21] Adam Bouland and Tudor Giurgica-Tiron. “Efficient Universal Quantum Compilation: An Inverse-free Solovay-Kitaev Algorithm.” In: (Dec. 2021). arXiv: [2112.02040](https://arxiv.org/abs/2112.02040) [quant-ph] (cit. on p. 23).
- [Boy+00] P. Oscar Boykin, Tal Mor, Matthew Pulver, Vwani Roychowdhury, and Farrokh Vatan. “A new universal and fault-tolerant quantum basis.” In: *Information Processing Letters* 75.3 (2000), pp. 101–107. DOI: [10.1016/S0020-0190\(00\)00084-3](https://doi.org/10.1016/S0020-0190(00)00084-3) (cit. on p. 22).
- [Bro+20] Michael Broughton et al. *TensorFlow Quantum: A Software Framework for Quantum Machine Learning*. Mar. 2020. arXiv: [2003.02989](https://arxiv.org/abs/2003.02989) [quant-ph] (cit. on p. 10).
- [Cha+81] Gregory J. Chaitin et al. “Register allocation via coloring.” In: *Computer Languages* 6.1 (1981), pp. 47–57. DOI: [10.1016/0096-0551\(81\)90048-5](https://doi.org/10.1016/0096-0551(81)90048-5) (cit. on p. 8).
- [Cow+19] Alexander Cowtan et al. “On the Qubit Routing Problem.” In: *14th Conference on the Theory of Quantum Computation, Communication and Cryptography (TQC 2019)*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany, May 2019. DOI: [10.4230/LIPIcs.TQC.2019.5](https://doi.org/10.4230/LIPIcs.TQC.2019.5) (cit. on pp. 39, 42).
- [Cro+17] Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. “Open Quantum Assembly Language.” In: (July 2017). arXiv: [1707.03429](https://arxiv.org/abs/1707.03429) [quant-ph] (cit. on p. 45).
- [Cro+19] Andrew W. Cross, Lev S. Bishop, Sarah Sheldon, Paul D. Nation, and Jay M. Gambetta. “Validating quantum computers using randomized model circuits.” In: *Phys. Rev. A* 100 (3 Sept. 2019), p. 032328. DOI: [10.1103/PhysRevA.100.032328](https://doi.org/10.1103/PhysRevA.100.032328) (cit. on p. 33).
- [Cro+21] Andrew Cross et al. “OpenQASM 3: A Broader and Deeper Quantum Assembly Language.” In: *ACM Transactions on Quantum Computing* (Dec. 2021). DOI: [10.1145/3505636](https://doi.org/10.1145/3505636) (cit. on p. 45).

- [Dev21] Cirq Developers. *Cirq*. Version v0.12.0. See full list of authors on Github: <https://github.com/quantumlib/Cirq/graphs/contributors>. Aug. 2021. DOI: [10.5281/zenodo.5182845](https://doi.org/10.5281/zenodo.5182845) (cit. on p. 45).
- [DiVoo] David P. DiVincenzo. “The Physical Implementation of Quantum Computation.” In: *Fortschritte der Physik* 48.9-11 (Sept. 2000), pp. 771–783. DOI: [10.1002/1521-3978\(200009\)48:9/11<771::aid-prop771>3.0.co;2-e](https://doi.org/10.1002/1521-3978(200009)48:9/11<771::aid-prop771>3.0.co;2-e) (cit. on p. 29).
- [DJ92] David Deutsch and Richard Jozsa. “Rapid solution of problems by quantum computation.” In: *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences* 439.1907 (Dec. 1992), pp. 553–558. DOI: [10.1098/rspa.1992.0167](https://doi.org/10.1098/rspa.1992.0167) (cit. on p. 14).
- [DL13] Ross Duncan and Maxime Lucas. “Verifying the Steane code with Quantomatic.” In: (June 2013). arXiv: [1306.4532](https://arxiv.org/abs/1306.4532) [quant-ph] (cit. on p. 44).
- [DN05] Christopher M. Dawson and Michael A. Nielsen. “The Solovay-Kitaev algorithm.” In: (May 2005). arXiv: [0505030](https://arxiv.org/abs/0505030) [quant-ph] (cit. on p. 20).
- [Fey82] Richard P. Feynman. “Simulating physics with computers.” In: *International Journal of Theoretical Physics* 21.6-7 (June 1982), pp. 467–488. DOI: [10.1007/bf02650179](https://doi.org/10.1007/bf02650179) (cit. on p. 13).
- [FGG14] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. “A Quantum Approximate Optimization Algorithm.” In: (Nov. 2014). DOI: [10.48550/ARXIV.1411.4028](https://doi.org/10.48550/ARXIV.1411.4028). arXiv: [1411.4028](https://arxiv.org/abs/1411.4028) [quant-ph] (cit. on p. 43).
- [Gok+19] Pranav Gokhale et al. “Partial Compilation of Variational Algorithms for Noisy Intermediate-Scale Quantum Machines.” In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO '52*. Columbus, OH, USA: Association for Computing Machinery, 2019, pp. 266–278. DOI: [10.1145/3352460.3358313](https://doi.org/10.1145/3352460.3358313) (cit. on p. 44).
- [Got09] Daniel Gottesman. “An Introduction to Quantum Error Correction and Fault-Tolerant Quantum Computation.” In: (Apr. 2009). arXiv: [0904.2557](https://arxiv.org/abs/0904.2557) [quant-ph] (cit. on p. 23).
- [HHH99] Michał Horodecki, Paweł Horodecki, and Ryszard Horodecki. “General teleportation channel, singlet fraction, and quasidistillation.” In: *Phys. Rev. A* 60 (3 Sept. 1999), pp. 1888–1898. DOI: [10.1103/PhysRevA.60.1888](https://doi.org/10.1103/PhysRevA.60.1888) (cit. on p. 24).
- [Hie+21] Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. “A Verified Optimizer for Quantum Circuits.” In: *Proc. ACM Program. Lang.* 5.POPL (Jan. 2021). DOI: [10.1145/3434318](https://doi.org/10.1145/3434318) (cit. on p. 44).

- [Hop52] Grace Hopper. 1952. URL: <http://www.computinghistory.org.uk/det/5487/Grace%20Hopper%20completes%20the%20A-0%20Compiler> (visited on 03/05/2022) (cit. on p. 5).
- [IBM] IBM. *IBM Quantum*. URL: <https://quantum-computing.ibm.com> (visited on 03/30/2022) (cit. on p. 31).
- [Jav+15] Ali JavadiAbhari et al. “ScaffCC: Scalable compilation and analysis of quantum programs.” In: *Parallel Computing* 45 (2015). Computing Frontiers 2014: Best Papers, pp. 2–17. DOI: [10.1016/j.parco.2014.12.001](https://doi.org/10.1016/j.parco.2014.12.001) (cit. on p. 46).
- [Kha+22] Pradnya Khalate et al. “An LLVM-based C++ Compiler Toolchain for Variational Hybrid Quantum-Classical Algorithms and Quantum Accelerators.” In: (Feb. 2022). arXiv: [2202.11142 \[quant-ph\]](https://arxiv.org/abs/2202.11142) (cit. on p. 44).
- [Kit97] Alexei Yurievich Kitaev. “Quantum computations: algorithms and error correction.” In: *Russian Mathematical Surveys* 52.6 (Dec. 1997), pp. 1191–1249. DOI: [10.1070/rm1997v052n06abeh002155](https://doi.org/10.1070/rm1997v052n06abeh002155) (cit. on pp. 20, 22).
- [KSV02] A. Kitaev, A. Shen, and M. Vyalyi. *Classical and Quantum Computation*. American Mathematical Society, May 2002. DOI: [10.1090/gsm/047](https://doi.org/10.1090/gsm/047) (cit. on p. 20).
- [LAo4] Chris Lattner and Vikram Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation.” In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO’04)*. Palo Alto, California, Mar. 2004 (cit. on p. 11).
- [LaR19] Ryan LaRose. “Overview and Comparison of Gate Level Quantum Software Platforms.” In: *Quantum* 3 (Mar. 2019), p. 130. DOI: [10.22331/q-2019-03-25-130](https://doi.org/10.22331/q-2019-03-25-130) (cit. on pp. 45, 46).
- [Lat+21] Chris Lattner et al. “MLIR: Scaling Compiler Infrastructure for Domain Specific Computation.” In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2021, pp. 2–14. DOI: [10.1109/CGO51591.2021.9370308](https://doi.org/10.1109/CGO51591.2021.9370308) (cit. on p. 11).
- [Lat19] Chris Lattner. *Compilers, LLVM, Swift, TPU, and ML Accelerators | Lex Fridman Podcast #21*. 2019. URL: <https://youtu.be/yCd3CzGSte8> (visited on 03/05/2022) (cit. on p. 11).
- [LDX19] Gushu Li, Yufei Ding, and Yuan Xie. “Tackling the Qubit Mapping Problem for NISQ-Era Quantum Devices.” In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’19. Providence, RI, USA: Associa-

- tion for Computing Machinery, 2019, pp. 1001–1014. DOI: [10.1145/3297858.3304023](https://doi.org/10.1145/3297858.3304023) (cit. on p. 43).
- [Lit+20] Andrew Litteken, Yung-Ching Fan, Devina Singh, Margaret Martonosi, and Frederic T Chong. “An updated LLVM-based quantum research compiler with further OpenQASM support.” In: *Quantum Science and Technology* 5.3 (May 2020), p. 034013. DOI: [10.1088/2058-9565/ab8c2c](https://doi.org/10.1088/2058-9565/ab8c2c) (cit. on p. 46).
- [McC+21] Alexander Mccaskey et al. “Extending C++ for Heterogeneous Quantum-Classical Computing.” In: *ACM Transactions on Quantum Computing* 2.2 (July 2021). DOI: [10.1145/3462670](https://doi.org/10.1145/3462670) (cit. on p. 46).
- [McK65] W. M. McKeeman. “Peephole Optimization.” In: *Commun. ACM* 8.7 (July 1965), pp. 443–444. DOI: [10.1145/364995.365000](https://doi.org/10.1145/364995.365000) (cit. on p. 9).
- [Mer] Merriam-Webster.com. *compile*. URL: <https://www.merriam-webster.com/dictionary/compile> (visited on 03/26/2022) (cit. on p. 4).
- [MN21] Alexander McCaskey and Thien Nguyen. “A MLIR Dialect for Quantum Assembly Languages.” In: *2021 IEEE International Conference on Quantum Computing and Engineering (QCE)*. 2021, pp. 255–264. DOI: [10.1109/QCE52317.2021.00043](https://doi.org/10.1109/QCE52317.2021.00043) (cit. on p. 38).
- [Mor+21] Lorenzo Moro, Matteo G. A. Paris, Marcello Restelli, and Enrico Prati. “Quantum compiling by deep reinforcement learning.” In: *Communications Physics* 4.1 (Aug. 2021). DOI: [10.1038/s42005-021-00684-3](https://doi.org/10.1038/s42005-021-00684-3) (cit. on p. 43).
- [Mur+19] Prakash Murali, Jonathan M. Baker, Ali Javadi-Abhari, Frederic T. Chong, and Margaret Martonosi. “Noise-Adaptive Compiler Mappings for Noisy Intermediate-Scale Quantum Computers.” In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’19. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 1015–1029. DOI: [10.1145/3297858.3304075](https://doi.org/10.1145/3297858.3304075) (cit. on p. 43).
- [Mur+20] Prakash Murali, David C. McKay, Margaret Martonosi, and Ali Javadi-Abhari. “Software Mitigation of Crosstalk on Noisy Intermediate-Scale Quantum Computers.” In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1001–1016 (cit. on p. 44).

- [Nan+21] Giacomo Nannicini, Lev S Bishop, Oktay Gunluk, and Petar Jurcevic. “Optimal qubit assignment and routing via integer programming.” In: (June 2021). arXiv: [2106.06446](https://arxiv.org/abs/2106.06446) [quant-ph] (cit. on p. 43).
- [NC10] Michael A Nielsen and Isaac L Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2010 (cit. on pp. 4, 13, 14, 20, 24, 29, 39).
- [Nie02] Michael A Nielsen. “A simple formula for the average gate fidelity of a quantum dynamical operation.” In: *Physics Letters A* 303.4 (Oct. 2002), pp. 249–252. DOI: [10.1016/S0375-9601\(02\)01272-0](https://doi.org/10.1016/S0375-9601(02)01272-0) (cit. on p. 24).
- [Niu+20] Siyuan Niu, Adrien Suau, Gabriel Staffelbach, and Aida Todri-Sanial. “A Hardware-Aware Heuristic for the Qubit Mapping Problem in the NISQ Era.” In: *IEEE Transactions on Quantum Engineering* 1 (2020), pp. 1–14. DOI: [10.1109/TQE.2020.3026544](https://doi.org/10.1109/TQE.2020.3026544) (cit. on p. 43).
- [Per+14] Alberto Peruzzo et al. “A variational eigenvalue solver on a photonic quantum processor.” In: *Nature Communications* 5.1 (July 2014). DOI: [10.1038/ncomms5213](https://doi.org/10.1038/ncomms5213) (cit. on p. 43).
- [Pre18] John Preskill. “Quantum Computing in the NISQ era and beyond.” In: *Quantum* 2 (Aug. 2018), p. 79. DOI: [10.22331/q-2018-08-06-79](https://doi.org/10.22331/q-2018-08-06-79) (cit. on p. 14).
- [QIR21] QIR Alliance. *QIR Specification*. 2021. URL: <https://github.com/qir-alliance/qir-spec> (visited on 04/07/2022) (cit. on pp. 38, 46).
- [Qua] Azure Quantum. *The Q# programming language user guide*. URL: <https://docs.microsoft.com/en-us/azure/quantum/user-guide/> (visited on 04/20/2022) (cit. on p. 45).
- [ROA17] Jonathan Romero, Jonathan P Olson, and Alan Aspuru-Guzik. “Quantum autoencoders for efficient compression of quantum data.” In: *Quantum Science and Technology* 2.4 (Aug. 2017), p. 045001. DOI: [10.1088/2058-9565/aa8072](https://doi.org/10.1088/2058-9565/aa8072) (cit. on p. 31).
- [Rod20] Andres Rodriguez. *Deep Learning Systems: Algorithms, Compilers, and Processors for Large-Scale Production*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, Oct. 2020. URL: <https://deeplearningsystems.ai> (cit. on p. 10).
- [RPZ18] Robert Rand, Jennifer Paykin, and Steve Zdancewic. “QWIRE Practice: Formal Verification of Quantum Circuits in Coq.” In: (Mar. 2018). arXiv: [1803.00699](https://arxiv.org/abs/1803.00699) [cs.LO] (cit. on p. 44).

- [SCZ16] Robert S. Smith, Michael J. Curtis, and William J. Zeng. “A Practical Quantum Instruction Set Architecture.” In: (Aug. 2016). arXiv: [1608.03355 \[quant-ph\]](https://arxiv.org/abs/1608.03355) (cit. on p. 46).
- [Shio3] Yaoyun Shi. “Both Toffoli and Controlled-NOT Need Little Help to Do Universal Quantum Computing.” In: *Quantum Info. Comput.* 3.1 (Jan. 2003), pp. 84–92 (cit. on p. 22).
- [Sho94] Peter W. Shor. “Algorithms for quantum computation: discrete logarithms and factoring.” In: *Proceedings 35th Annual Symposium on Foundations of Computer Science*. IEEE Comput. Soc. Press, Nov. 1994. DOI: [10.1109/sfcs.1994.365700](https://doi.org/10.1109/sfcs.1994.365700) (cit. on pp. 14, 39).
- [Sho96] Peter W. Shor. “Fault-tolerant quantum computation.” In: (May 1996). arXiv: [quant-ph/9605011 \[quant-ph\]](https://arxiv.org/abs/quant-ph/9605011) (cit. on p. 22).
- [Siv+20] Seyon Sivarajah et al. “t|ket): a retargetable compiler for NISQ devices.” In: *Quantum Science and Technology* 6.1 (Nov. 2020), p. 014003. DOI: [10.1088/2058-9565/ab8e92](https://doi.org/10.1088/2058-9565/ab8e92) (cit. on p. 40).
- [SMo9] Vivek V. Shende and Igor L. Markov. “On the CNOT-Cost of TOFFOLI Gates.” In: *Quantum Info. Comput.* 9.5 (May 2009), pp. 461–486 (cit. on p. 39).
- [Smiz0] Robert S Smith. *Quil: A Portable Quantum Instruction Language*. Version 20200220. Feb. 2020. DOI: [10.5281/zenodo.3677541](https://doi.org/10.5281/zenodo.3677541) (cit. on p. 46).
- [Sol21] Honeywell Quantum Solutions. *Honeywell Sets Another Record For Quantum Computing Performance*. July 2021. URL: <https://www.honeywell.com/us/en/news/2021/07/honeywell-sets-another-record-for-quantum-computing-performance> (visited on 04/13/2022) (cit. on p. 34).
- [Tea22] The Coq Development Team. *The Coq Proof Assistant*. Version 8.15. Jan. 2022. DOI: [10.5281/zenodo.5846982](https://doi.org/10.5281/zenodo.5846982) (cit. on p. 44).
- [Wac+21] Andrew Wack et al. “Quality, Speed, and Scale: three key attributes to measure the performance of near-term quantum computers.” In: (Oct. 2021). arXiv: [2110.14108 \[quant-ph\]](https://arxiv.org/abs/2110.14108) (cit. on p. 34).
- [Wat18] John Watrous. *The Theory of Quantum Information*. Cambridge University Press, 2018. URL: <https://cs.uwaterloo.ca/~watrous/TQI> (cit. on pp. 13, 24).
- [WHT15] Dave Wecker, Matthew B. Hastings, and Matthias Troyer. “Progress towards practical quantum variational algorithms.” In: *Phys. Rev. A* 92 (4 Oct. 2015), p. 042303. DOI: [10.1103/PhysRevA.92.042303](https://doi.org/10.1103/PhysRevA.92.042303) (cit. on p. 43).

- [XZZ21] Lei Xie, Jidong Zhai, and Weimin Zheng. “Mitigating Crosstalk in Quantum Computers through Commutativity-Based Instruction Reordering.” In: *2021 58th ACM/IEEE Design Automation Conference (DAC)*. Dec. 2021, pp. 445–450. DOI: [10.1109/DAC18074.2021.9586145](https://doi.org/10.1109/DAC18074.2021.9586145) (cit. on p. 44).
- [YM10] Shigeru Yamashita and Igor L. Markov. “Fast Equivalence-Checking for Quantum Circuits.” In: *Quantum Info. Comput.* 10.9 (Sept. 2010), pp. 721–734 (cit. on p. 44).
- [Zhu+22] Qingling Zhu et al. “Quantum computational advantage via 60-qubit 24-cycle random circuit sampling.” In: *Science Bulletin* 67.3 (Feb. 2022), pp. 240–245. DOI: [10.1016/j.scib.2021.10.017](https://doi.org/10.1016/j.scib.2021.10.017) (cit. on p. 14).

DECLARATION

This thesis consists of material all of which I authored.

I understand that my thesis may be made electronically available to the public.

Seattle, WA, April 2022



Nathaniel Stemen

COLOPHON

This document was typeset using the typographical look-and-feel *classicthesis* developed by André Miede and Ivo Pletikosić. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*" and is available at:

<https://bitbucket.org/amiede/classicthesis/>

Hermann Zapf's *Palatino* and *Euler* type faces are used and the "typewriter" text is typeset in *Bera Mono*.