Compiling Quantum Programs

A Senior Project submitted to
The Division of Science, Mathematics, and Computing
of
Bard College

by
Li-Heng Henry Chang 張立恆

Annandale-on-Hudson, New York
May, 2023

Abstract

This thesis introduces the quantum compilation problem and develops a prototypical compiler. The problem of quantum compiling is, in essence, converting high-level human expressions of quantum programs into low-level hardware executable code. Compilers that target different hardware platforms enable portable code that can be used to benchmark hardware performance, reduce programming work and speed up development. Because quantum systems are subjected to phenomena such as noise, no-cloning and decoherence, the challenge of quantum compiling is tied to the optimization of program runtimes and the lengths of compiled sequences. For near-term intermediate scale quantum (NISQ) computers with limited hardware resources and without error correction, a well-designed compiler that optimizes hardware usage and circuit fidelity is necessary to running applications and tests. To give an introduction to the problem of quantum compiling, this thesis reviews the universality proof for quantum computation and the Solovay-Kitaev theorem, which are both foundational to the topic. A compilation scheme with two components, one following the universality proof and one inspired by the Solovay-Kitaev theorem, is implemented to demonstrate an approach that solves the quantum compiling problem. Finally, I survey state-of-the-art compilation techniques and discuss how to extend this thesis toward building a practical compiler that will be a part of the broader software stack.

Contents

Dedication

I dedicate this senior project to my parents who support me selflessly.

Acknowledgments

I am extremely grateful to my advisor Prof. Paul Cadden-Zimansky for mentoring the quantum computing reading group that I initiated and having his door always open to my endless questions. This is how my curiosity continues to grow in the area of quantum computing. I am thankful to my advisor Prof. Sven Anderson for giving honest and sharp feedback along with some fun banters, Prof. Harold Haggard for providing an overloading amount of insights on math and quantum theories, my advisor Prof. Keith O'Hara for being always supportive, and lastly Prof. Valerie Barr for listening to my questions attentively.

1
Introduction

This thesis introduces the core problem of quantum compilation and demonstrate a prototypical compiler as a solution. The main chapters summarize the fundamentals of quantum compiling, implement a hardware-independent compiler, and survey state-of-the-art quantum compiling schemes. In the end, we briefly discuss how this work can be extended to build a practical compiler. Chapter 2, "Quantum Computation (QC)," reviews the paradigm of quantum computing, and how it is different from classical computing. This provides relevant background and motivation for quantum compilation. Chapter 3, "The Gate Model of Quantum Computing," introduces the mathematical formalism that is used to describe the necessary background of the quantum computing paradigm. All the prerequisites to quantum compiling are introduced in this chapter. Chapter 4, "Quantum Compilation," is the main chapter that reviews two important concepts behind quantum compiling: universal quantum computation and the Solovay-Kitaev theorem. Universal quantum computation connects unit (single- and two-qubit) operations with arbitrary quantum programs. Solovay-Kitaev is a fundamental theorem that shows quantum compiling can be done efficiently and so opens up research in the area. We will build a quantum compiler by applying both universal quantum computation and a Solovay-Kitaev-inspired algorithm. In Chapter 5, "Experiments," the compilation performance of our compiler implementation will be discussed. We will see that, for practical application on NISQ

devices, our prototypical compiler is incomplete and too inefficient. In Chapter 6, "Challenges towards Practical Quantum Compilation", we will survey recent compilation techniques as well as discuss how our compiler can be extended for NISQ and realistic applications.

1.1 Prerequisites

This thesis builds on top of and assumes that the readers are familiar with the following topics:

- Elementary Linear Algebra
- Basic Probability Theory
- Boolean Algebra
- Complexity Analysis and Big-O Notation
- Data Structures and Algorithms

2
Quantum Computation (QC)

The problem of quantum compilation stems from solving the larger problem of realizing quantum computers and quantum algorithms. This chapter will introduce the role and importance of quantum compilation in relation to the broader context of quantum computing, which is a drastically different paradigm than classical computing. Quantum computing (QC) is known to have an advantage over classical computing in solving specific sets of problems such as prime factoring. What quantum properties allow quantum computation to have an advantage over classical computing? What makes quantum computation unique? This chapter will attempt to address these fundamental questions at a high level. To characterize these quantum properties and apply them to algorithmic design, we need a framework to describe their behaviors. For this purpose, the quantum circuit model of quantum computation will be introduced in the next chapter. To enhance cohesive understanding, an overview of quantum mechanics, the operating principles behind quantum computing, will also be given.

2.1 Quantum Advantage, Quantum Weirdness, and the Roadmap to Useful Quantum Computing

Richard Feynman pointed out that classical computers cannot efficiently simulate a general quantum evolution, but other quantum mechanical systems can [6]. This observation has led to a series of discoveries on the so-called quantum supremacy, problems that can be solved expo-

nentially faster with quantum computation than with classical computers. These computational advantages of quantum computers are generally attributed to the natural phenomena of superposition and entanglement. In quantum mechanics, superposition allows a single qubit (short for "quantum bit") state to exist continuously between 0 and 1, so a quantum state can encode infinitely more information than can a discrete classical bit. However, to access the information of a quantum state one must perform a measurement that collapses the state to a binary outcome and destroys most information, so the extra information provided by superposition can be taken advantage of only in certain cases. One of the known cases is entanglement, which is the phenomenon where qubit states are correlated in a multi-qubit system. Their correlation provides more information than the individual qubits do independently. This extra information is accessible by measurement, helps infer state properties, and thus provides computational advantage [12]. The opposite of an entangled state is a product or separable state, and it has been shown that any product state can be decomposed and efficiently simulated by classical computers with a small overhead. Hence, quantum computers have no advantage in computing product states over classical ones [22].

Currently, most efforts to realize a useful quantum computer are focused on developing robust and scalable hardware. At the moment, there is a variety of Near-term Intermediate Scale Quantum (NISQ) computing hardware platforms being developed. However, NISQ quantum machines do not have enough resources to abstract away their physical conditions and act as if they are "virtual." To give programmers the expressive power to articulate quantum algorithms relative to tight NISQ hardware constraints, we will need to create and enhance programming languages and compilation techniques [18]. Thus, the research for implementation, optimization, mapping, and resource management on functionalities between algorithms and devices is important to the NISQ era. A portable and efficient compiler will help with benchmarking and the evolution of QC hardware advancement.

3
The Gate Model of Quantum Computing

In order to describe quantum computation, we need a concise language. The linear algebraic formalism of quantum mechanics is a powerful tool that describes the evolution of a quantum state over time and is used at the core of the Gate Model of Quantum Computing, with a small number of modifications, to describe quantum computation. This chapter will start by introducing quantum states and then operations that change states through evolution.

This chapter provides the background and notation reference for later chapters. The reader can skip this chapter if already familiar with the content and jump back for reminders. The main reference for this chapter is the classic textbook *Quantum Computation and Quantum Information* by Nielsen and Chuang [22]. There are also wonderful resources that the readers can refer to for an introduction to quantum computation [10, 14, 19, 27].

3.1 Single-Qubit States

In classical physics that describes the macroscopic world, a bit can take on two possible values, 0 and 1. The number 0 is often used to represent a state of low electrical voltage 0V and 1 a state of high voltage 5V. By doing so, we can keep track of the state of capacitors, switches, and electrical components in computers.

When looking at the microscopic world of atoms, electrons, and photons, the state of a physical system is described by quantum mechanics. In this framework, the smallest unit of information

to keep track of a quantum state is a "quantum bit" or "qubit". For example, a qubit is used to describe the polarization of a photon or the spin of an electron. We use the **Dirac** notation $|0\rangle$ and $|1\rangle$ to represent physically distinguishable states. While a classical state can be either 0 or 1, a quantum state can be in a **superposition**, or linear combination, of states $|\psi\rangle = a\,|0\rangle + b\,|1\rangle$. In quantum mechanics, a physical system that can exist in any superposition of n distinguishable states is called an **n-level** system, so a qubit is a **two-level** system. In the linear algebra language, the special states $|0\rangle$ and $|1\rangle$ are called the computational basis states that form an orthonormal basis

$$|0\rangle \to \begin{bmatrix} 1 \\ 0 \end{bmatrix}, |1\rangle \to \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

Mathematically, a single-qubit state $|\psi\rangle = a\,|0\rangle + b\,|1\rangle$ can be described by two complex numbers such that they are normalized $|a|^2 + |b|^2 = 1$.

$$|\psi\rangle \to a \begin{bmatrix} 1 \\ 0 \end{bmatrix} + b \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} a \\ b \end{bmatrix}$$

In the context of an electron, this means that if we measure its spin, we have a $|a|^2$ probability of measuring the $|0\rangle$ state and $|b|^2$ probability of measuring the $|1\rangle$ state. For this reason, a and b are called the "probability amplitudes" of state $|\psi\rangle$. The normalization condition ensures that the probabilities of measurement outcomes add up to one. For electron spin, the computational basis state $|0\rangle$ and $|1\rangle$ represent it pointing upwards and downwards respectively. However, in general, the two computational basis states can be mapped to arbitrary states of an experimenter's choice.

As a mental model, it is helpful to think of $|0\rangle$ and $|1\rangle$ as electron spin. However, in general, they can be mapped to arbitrary physical states depending on the experimental setup. For ease of computational thinking and algorithm design, we abstract away the details of what physically makes a qubit and are only interested in the qubit as a mathematical object. We leave the physical details to material scientists and quantum hardware vendors. For this reason, this section provides a brief discussion of the physics of qubits to give context for understanding, but for the most part, we will discuss the mathematical and computational aspects of a qubit.

There are multiple useful representations of qubits. Since $|a|^2 + |b|^2 = 1$, we can rewrite $|\psi\rangle = a\,|0\rangle + b\,|1\rangle$ as $|\psi\rangle = e^{i\gamma}\left(\cos\frac{\theta}{2}|0\rangle + e^{i\varphi}\sin\frac{\theta}{2}|1\rangle\right)$. This parameterizes $|\psi\rangle$ in terms of two angles θ and ϕ that define a point on the unit R^3 sphere. This is called a **Bloch Sphere** that provides a useful way of visualizing a single-qubit state, as shown in Figure 3.1.1. The γ parameter is called the global phase. It is often ignored in the context of single-qubit states because it has no contribution to the probability of measurement outcomes if one takes the norm of the probability amplitudes.

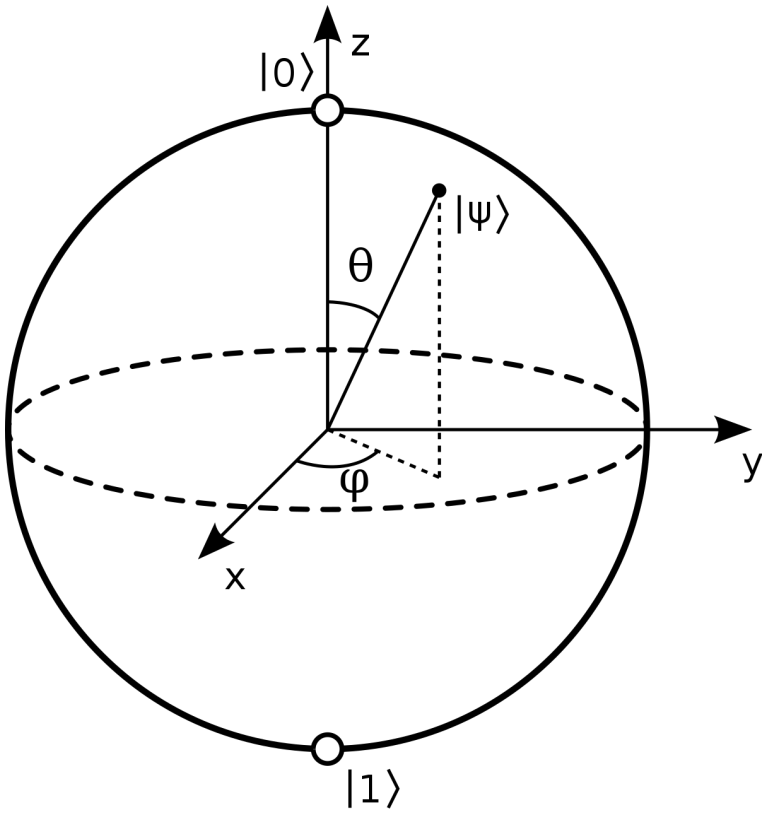

Figure 3.1.1: The Bloch Sphere Visualization of a single-qubit state.

3.2 Multi-Qubit States

By the linear algebra formalism of quantum mechanics in Section 3.4, we can describe an n-qubit state by taking the tensor product of individual qubits. For example, two single-qubit states

can be combined as

$$|ba\rangle = |b\rangle \otimes |a\rangle = \begin{bmatrix} b_0 \times \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} \\ b_1 \times \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} b_0 a_0 \\ b_0 a_1 \\ b_1 a_0 \\ b_1 a_1 \end{bmatrix}$$

If we apply this to the single-qubit computational basis state, we can get the computational basis states for two-qubit systems.

$$|0\rangle \otimes |0\rangle = |00\rangle \rightarrow \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, |0\rangle \otimes |1\rangle = |01\rangle \rightarrow \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, |1\rangle \otimes |0\rangle = |10\rangle \rightarrow \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, |1\rangle \otimes |1\rangle = |11\rangle \rightarrow \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}.$$

A general two-qubit state $|\psi\rangle$ is a linear combination of these basis states $|\psi\rangle = a_{00}|00\rangle + a_{01}|01\rangle + a_{10}|10\rangle + a_{11}|11\rangle$, where all coefficients are complex and the normalization condition applies $|\psi\rangle = |a_{00}|^2 + |a_{01}|^2 + |a_{10}|^2 + |a_{11}|^2$. Here tensor product combined two 2-level systems into a 4-level system, described by 4 complex numbers. Since the state space is the number vectors in the basis, the state space for $n = 2$ qubits is now $2^n = 4$. A general n-qubit state describes a 2^n-level system and the state space has dimension 2^n that grows exponentially to the number of qubits.

3.3 State Evolution and Quantum Computation

The changes of a quantum state over time can be described by the language of quantum computation, the **quantum circuit model**, which contains wires to carry around and **quantum gates** to manipulate quantum information. This section provides a brief introduction to elementary quantum gates and simple circuits that describes quantum computation, and therefore quantum algorithms. These topics are fundamental to quantum computation that provide the necessary background and context for quantum compilation.

3.3.1 Single-Qubit Gates

Since our smallest unit of information is a qubit, the atomic operation is a single-qubit gate. For example, the NOT gate X exchanges the amplitudes of the computational basis states.

$$X \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \beta \\ \alpha \end{bmatrix}$$

Using linear algebra, we can find that the X gate operator can be characterized by

$$X \equiv \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

Some other notable single-qubit gates are the Hadamard Gate H, the Z-gate Z, and the T(or $\pi/8$)-gate T.

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}, T = \begin{bmatrix} 1 & 0 \\ 0 & \exp\left(\frac{i\pi}{4}\right) \end{bmatrix}.$$

Since a single-qubit operation maps a single-qubit state to another, we can use a 2x2 matrix to describe them. However, quantum mechanics also requires the normalization condition that the probability of all measurement outcomes sums up to 1. This requires that any quantum gate U to be **unitary**, so that $U^\dagger U = I$, where $U\dagger$ is the transposed complex conjugate of U, also known as the adjoint of U. It turns out that the unitary constraint is the only constraint on quantum gates.

Thus, single-qubit gates are 2x2 unitary matrices U_{2x2} belonging to the two-dimensional unitary group $U(2)$. Using the unitary constraint, single-qubit gates, in their most general form, can be expressed with four real parameters α, β, γ, and δ:

$$U(\alpha, \beta, \gamma, \delta) = e^{i\alpha} \begin{bmatrix} e^{-i\beta/2} & 0 \\ 0 & e^{i\beta/2} \end{bmatrix} \begin{bmatrix} \cos\frac{\gamma}{2} & -\sin\frac{\gamma}{2} \\ \sin\frac{\gamma}{2} & \cos\frac{\gamma}{2} \end{bmatrix} \begin{bmatrix} e^{-i\delta/2} & 0 \\ 0 & e^{i\delta/2} \end{bmatrix}$$

Often the global phase factor α is unimportant, and removing it gives the nice property for U to have determinant one. The group of unitary matrices with determinant one is the **special unitary group**, with the two-dimensional group denoted as $SU(2)$. For convenience, we often ignore the global phase factor and restrict ourselves to $SU(2)$ when looking at single-qubit gates.

3.3.2 Quantum Circuit Diagrams

The matrix representation of a quantum gate has multiple elements and can have much information to read off. Conceptually, it is often easier to express quantum gates in a circuit diagram to show their functionality (the matrix and circuit diagram representation are equivalent and can be converted from one to another but convenient for different purposes). A quantum circuit is composed of wires and gates. Typically, wires lie horizontally, with each representing the state

of a qubit. Information passes from left to right on the wire over time. Gates are blocks marked with symbols. Some examples of single-qubit gates put into a circuit diagram are displayed in Figure 3.3.1.

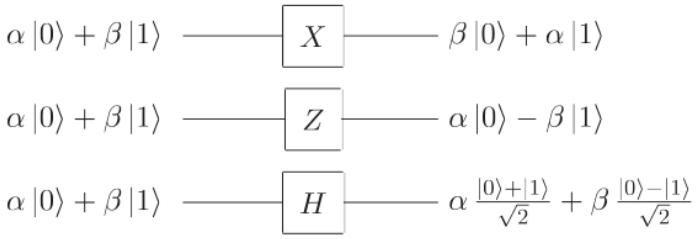

Figure 3.3.1: Circuit diagrams of single-qubit gates.

3.3.3 Multi-Qubit Gates

The most commonly seen and prototypical multi-qubit gate is the controlled-NOT or CNOT gate, which provides a function similar to conditionals in classical computing. Its circuit representation is in Figure 3.3.2, where the $\oplus$ operator is a binary add. The CNOT gate U_{CN} takes two input qubits named the control bit and the target bit. The target bit flips if the control bit is 1; otherwise, it is left unchanged. The basis states are switched as follows:

$$|\psi\rangle = \begin{bmatrix} a_{00} \\ a_{01} \\ a_{10} \\ a_{11} \end{bmatrix}, \quad U_{CN}|\psi\rangle = \begin{bmatrix} a_{00} \\ a_{01} \\ a_{11} \\ a_{10} \end{bmatrix}$$

Note that here the CNOT gate is unitary $U_{CN}^{\dagger}U_{CN} = I$.

In general, an n-qubit operation is a $2^n \times 2^n$ unitary matrix. However, we can also obtain them by taking the tensor product of single-qubit gates. For example, if we have H acting on qubit one $|q_0\rangle$ and X acting on qubit one $|q_1\rangle$, the corresponding two-qubit gate is represented by the matrix $X \otimes H$:

$$X|q_1\rangle \otimes H|q_0\rangle = (X \otimes H)|q_1 q_0\rangle,$$

where the tensor product can be expanded by

$$X \otimes H = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \otimes H = \begin{bmatrix} 0 \times H & 1 \times H \\ 1 \times H & 0 \times H \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \\ 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \end{bmatrix}.$$

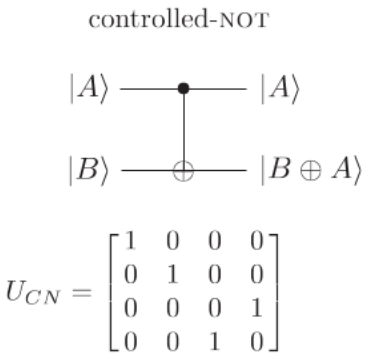

Figure 3.3.2: Circuit diagram and matrix representation of the CNOT gate.

There are many more multi-qubit gates. However, later in Section 4.4 we will show that CNOT gates and single-qubit gates form a universal gate set, meaning that they can approximate any quantum operation up to an arbitrary precision. For compilation purposes, we are interested in universal gate sets, so we will focus on introducing CNOT and single-qubit gates. Readers interested in other multi-qubit gates can refer to [1].

3.3.4 Quantum Circuit and Quantum Computation

We have shown the circuit diagram for single-qubit operations. In general, a quantum circuit is useful in terms of visualizing a quantum algorithm and its functionalities. Note that the same algorithm can have multiple representations, for instance, Figure 3.3.3 shows that a SWAP operation can be represented by three CNOT gates. A rather complete circuit example is shown in Figure 3.3.4. This is the circuit of Grover's algorithm [25], the quantum search algorithm that provides at most a quadratic speedup over the classical solution for unstructured search. In this circuit, the two-qubit state starts in the computational basis $|00\rangle$, goes through a series of single- and two-qubit operations, and is measured along the computational basis in the end, with the measurement outcomes sent into a classical channel $c2$.

We introduce quantum circuits here as an illustration tool that is helpful in terms of visualizing quantum algorithms, their functionalities, and their compositions, but we won't go into every

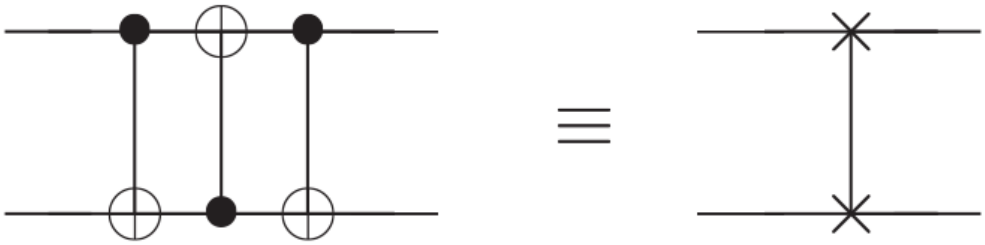

Figure 3.3.3: Two representations of the SWAP gate.

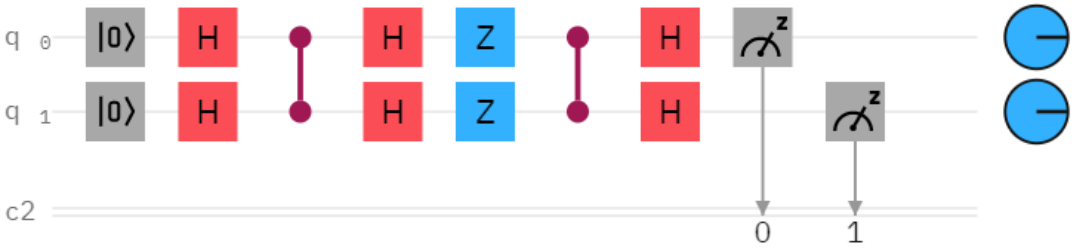

Figure 3.3.4: The circuit diagram of Grover's algorithm from [25].

operation in the circuit. For a more detailed introduction to quantum circuits and algorithms, see sections 1.3 and 1.4 of [22].

In our thesis, when we say "compile an algorithm into a circuit," the "circuit" refers to the part of the circuit before measurement that is made up of quantum wires and gates and excludes the classical channel. Whether the circuit refers to the full circuit or the one without the measurement-related portions is context-dependent. The reader should be aware of this distinction.

A useful metric of a quantum circuit is its **gate depth** l, defined as the maximum number of gates on each qubit in the circuit. For instance, the number of gates on qubits $q0$ and $q1$ in the circuit diagram of Grover's Search algorithm in Figure 3.3.4 is both 6 (the initial states $|0\rangle$ and measurement operators are excluded), so the gate depth is $l = 6$.

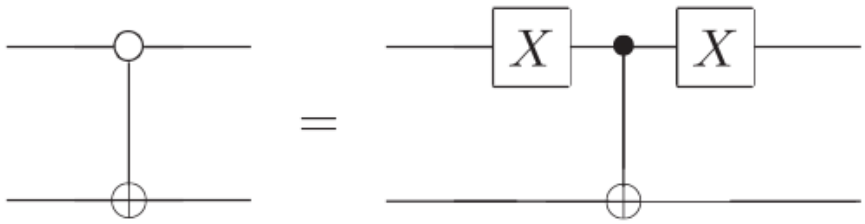

Figure 3.3.5: The counterpart gate of a CNOT gate that flips the target bit if the control bit is 0.

3.3.5 Controlled Operations

Controlled operations are important in computation because they provide conditionals such as IF A THEN B. They are also crucial to the proof for universal quantum computation that will be introduced in Section 4.4.

The most fundamental controlled operation in quantum computation is the controlled-NOT (CNOT) gate we have seen earlier. It flips the target bit based on the control bit, and so has the matrix representation

$$\text{CNOT} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

and circuit diagram as shown in Figure 3.3.2. Note that CNOT is a **permutation matrix** that flips the 3rd and 4th element of a vector on which it acts. Controlled operations often show up in the form of a permutation matrix. The CNOT gate has a counterpart gate that flips the target gate conditioned on the control bit being 0 instead of 1. It can be implemented with a CNOT and two NOT gates, as shown in Figure 3.3.5.

It is useful to have a controlled-U operation. That is, when the control bit is 1, the operation U is applied to the target bit; otherwise the target bit is left unchanged. A CNOT is a special case of a controlled-U where $U = X$. A controlled-U operation is denoted in a circuit as in Figure 3.3.6.

So far we have introduced two-qubit controlled gates, but we can have a controlled operation on n-qubits. An example is the Tiffoli gate, or the controlled-controlled-not (CCNOT) gate,

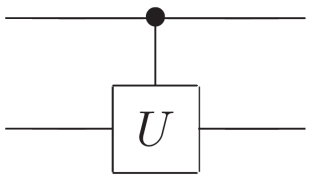

Figure 3.3.6: The circuit symbol of a controlled-U operation.

Inputs			Outputs		
a	b	c	a'	b'	c'
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	1	0	1
1	1	0	1	1	1
1	1	1	1	1	0

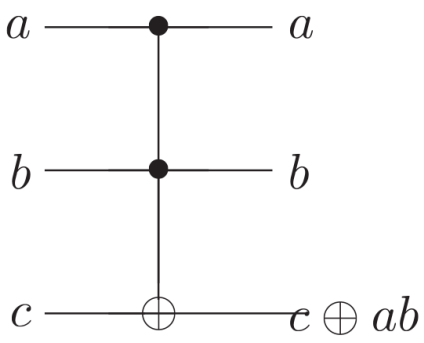

Figure 3.3.7: Tiffoli gate and its truth table. Referenced from [22] figure 1.14.

that acts on a three-qubit state, with two qubits being the control bits and the last one the target bit. The target bit is flipped only if both of the control bits are 1. The truth table and circuit representation of the Tiffoli gate are in Figure 3.3.7.

The most general form of a controlled operation $C^n(U)$ on n-qubits is one that has a unitary operation U acting on k qubits and conditioned on $n-k$ qubits, as denoted in Figure 3.3.9. The Tiffoli, or controlled CNOT gate, is $C^3(X)$ gate with $k=1$ conditioned on 2 qubits, for instance. A general $C^n(U)$ can be implemented following the example in Figure 3.3.8. Notice that in this example we use Tiffoli gates to implement the fully controlled condition, and then add a controlled-U operation to the last working and target bit. Since a Tiffoli gate can be implemented with only $O(1)$ CNOT and single-qubit operations as shown in Figure 3.3.7, a general controlled $C^n(U)$ can be implemented with $O(n)$ CNOT and single-qubit operations taking the same approach as in Figure 3.3.8. A **fully controlled gate** is one where $k=1$, meaning that a single qubit operation is conditioned on the values of all the other qubits.

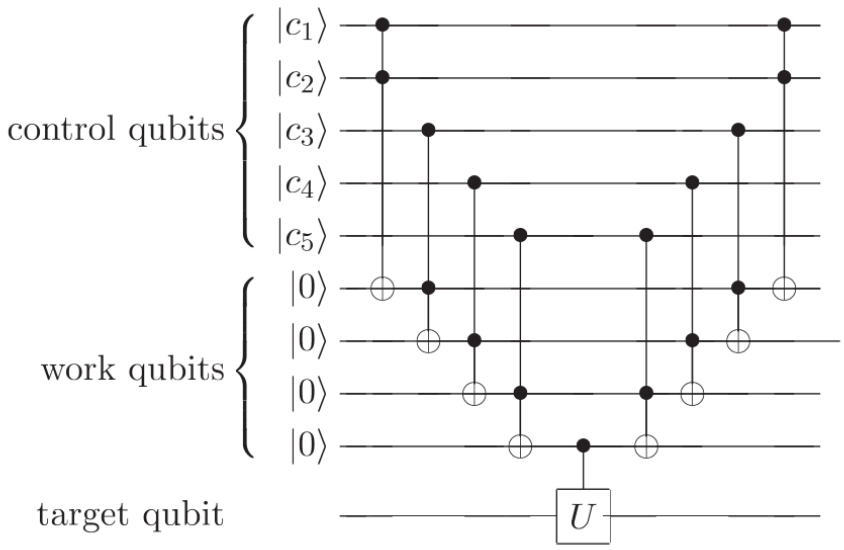

Figure 3.3.8: Circuit implementation of the $C^n(U)$ operation. Referenced from [22] figure 4.10.

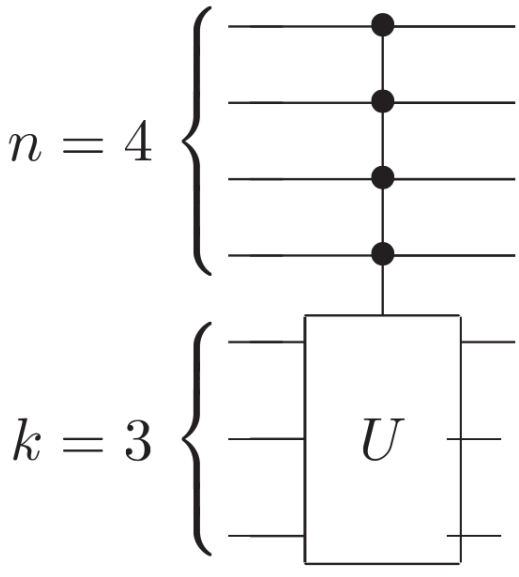

Figure 3.3.9: The circuit of a general controlled operation $C^n(U)$, where U is a gate that acts on k qubits for $n = 4$ and $k = 3$. Referenced from [22] figure 4.7.

The definitions above for various controlled operations will become useful when we introduce Universal Quantum Computation in section 4.4.

3.4 Postulate of Quantum Mechanics

Quantum Mechanics is essentially a set of mathematical postulates that describe the behaviors of the microscopic world of atoms, electrons, and photons over time. As far as we know, physicists have been unable to in any way violate Quantum Mechanics, which is the operating principle behind quantum computation! Here we will introduce the postulates of quantum mechanics to help the reader understand the unusual paradigm it has and better make sense of quantum computation. Quantum mechanics is known to be unintuitive and difficult to understand, but that comes from the application of the set of its postulates to physical contexts; the postulates themselves are a simple set of rules that should not scare away the reader. We take all four postulates of Quantum Mechanics from [20] and create connections by providing examples in quantum computation.

> **Postulate 1**: Associated with any physical system is a complex vector (Hilbert) space known as the *state space* of the system. If the system is isolated, then the system is completely described by its *state vector*, which is a unit vector in the system's state space.

Postulate 1 corresponds to our example of a state vector $|\psi\rangle = \begin{bmatrix} a \\ b \end{bmatrix}$, where $\langle\psi|\psi\rangle = \begin{bmatrix} a^2 & b^2 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = |a|^2 + |b|^2 = 1$. Complex vector space says that the probability amplitudes are complex numbers, and the unit vector condition is equivalent to the normalization condition.

> **Postulate 2**: The evolution of an isolated quantum system is described by a unitary matrix acting on the state space of the system. That is, the state $|\psi\rangle$ of the system at a time t_1 is related to the state $|\psi'\rangle$ at a later time t_2 by a unitary matrix, $U : |\psi'\rangle = U|\psi\rangle$. That matrix U may depend on the times t_1 and t_2, but does not depend on the states $|\psi\rangle$ and $|\psi'\rangle$.

From postulate 2, we can show why being unitary is a condition of quantum gates. If we have a state evolution described by U, then the state vector $|\psi'\rangle = U|\psi\rangle$ of the state at t_2 should still satisfy the normalization condition and remain a unit vector, so $\langle\psi'|\psi'\rangle = \langle\psi|U^\dagger U|\psi\rangle = 1 = \langle\psi|\psi\rangle$. It turns out that this "norm preserving" property of operators on complex vectors is unitary. Since $|\psi\rangle$ is any complex vector with $\langle\psi|\psi\rangle = 1$, it must be the case that $U^\dagger U = I$ in order for $\langle\psi|U^\dagger U|\psi\rangle = \langle\psi|\psi\rangle$.

> **Postulate 3**: Quantum measurements are described by a collection $\{M_m\}$ of measurement operators. Each M_m is a matrix acting on the state space of the system being measured. The index m takes values corresponding to the measurement outcomes that may occur in the experiment. If the state of the quantum system is $|\psi\rangle$ immediately before the measurement then the probability that result m occurs is given by $p(m) = \langle\psi|M_m^\dagger M_m|\psi\rangle$, and the state of the system after the measurement, often called the posterior state, is
>
> $$\frac{M_m|\psi\rangle}{\sqrt{\left\langle\psi\left|M_m^\dagger M_m\right|\psi\right\rangle}}$$
>
> (It's worth noting that: (a) the denominator is just the square root of the probability $p(m)$; and (b) this is a properly normalized quantum state.) The measurement operators satisfy the completeness relation, $\sum_m M_m^\dagger M_m = I$

For a single qubit, the measurement operators along the computational basis are

$$M_0 = |0\rangle\langle 0| = \begin{bmatrix}1 & 0\\ 0 & 0\end{bmatrix} \text{ and } M_1 = |1\rangle\langle 1| = \begin{bmatrix}0 & 0\\ 0 & 1\end{bmatrix} \tag{3.4.1}$$

Measuring a state $|\psi\rangle$ with M_0, the probability of getting back $|0\rangle$ is

$$p(0) = \langle\psi|M_0^\dagger M_0|\psi\rangle = \begin{bmatrix}a* & b*\end{bmatrix}\begin{bmatrix}1 & 0\\ 0 & 0\end{bmatrix}\begin{bmatrix}1 & 0\\ 0 & 0\end{bmatrix}\begin{bmatrix}a\\ b\end{bmatrix} = |a|^2$$

with the posterior state being

$$\frac{M_0|\psi\rangle}{\sqrt{|a|^2}} = \frac{|0\rangle\langle 0|\psi\rangle}{|a|} = \frac{a|0\rangle}{|a|}.$$

Similarly, the probability of getting back $|1\rangle$

$$p(1) = \langle\psi| M_1^\dagger M_1 |\psi\rangle = \begin{bmatrix} a* & b* \end{bmatrix} \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = |b|^2$$

with the posterior state being

$$\frac{M_1 |\psi\rangle}{\sqrt{|b|^2}} = \frac{|1\rangle \langle 1|\psi\rangle}{|b|} = \frac{b |1\rangle}{|b|}.$$

It is worth noting that here the measurement operators are NOT unitary! They are a special type of operator that is treated differently. The focus of this thesis will be on the operators that describe state evolution in postulate 2.

> **Postulate 4**: The state space of a composite physical system is the tensor product of the state spaces of the component physical systems. Moreover, if we have systems numbered 1 through n, and system number j is prepared in the state $|\psi_j\rangle$, then the joint state of the total system is just the tensor product of the individual states, $|\psi_1\rangle \otimes |\psi_2\rangle \otimes \ldots \otimes |\psi_n\rangle$.

Postulate 4 connects with Section 3.2. The important consequence here of the tensor product is that the size of the composite vector space is the product of the sizes of its component vector spaces, so the size grows *exponentially* with the number of qubits. It requires a number of parameters exponential to the number of qubits in order to describe a quantum state, whereas, in classical computing, we only need a binary string of length n to describe a computational state with n bits. This is why we say Hilbert space is large! The state space size difference between n qubits and n bits is one of the reasons quantum computers are more powerful than classical computers. At a high-level, quantum computation evolves a state over this complex vector space (larger than the state space of its classical counterpart) to search through different parameters.

4
Quantum Compilation

From the perspective of an application designer or programmer, we want programs and programming languages to be intuitive, expressive, and concise in terms of algorithmic thinking, thus at a high level of abstraction. However, quantum hardware usually only provides a limited instruction set that is made up of small transformations at a lower level of abstraction. Thus, a translation from high-level programs to low-level hardware instructions is required to create executable programs. This translation process is called compilation, and the algorithm that performs compilation is called a compiler. In classical computing, a compiler converts high-level programming language into low-level assembly or machine code. Similarly, a quantum compiler, takes the input of an algorithm U and hardware instruction set $\mathcal{G}$, and outputs a sequence S that approximates U using instructions from $\mathcal{G}$.

To utilize hardware resources and satisfy performance needs, modern compilers have evolved to be large and complicated. Since physical qubits easily decohere under the influence of environmental noise, we care that the compiler generates an efficient output sequence S as short as possible, so that the program finishes execution before noise affects the outcome of computation. In general, the design of a compiler needs to consider hardware topology (qubit connectivity), decoherence time, and fidelity (1 - error) constraints. For the same high-level programs to run on different hardware platforms, compilers can have the same frontend that translates a high-level

language into intermediate representations (IRs) and multiple back ends that convert IRs into machine instructions supported by different hardware. For all the above design considerations, compilers often involve multiple sub-pieces that handle conversion to intermediate representations, optimization and more. These topics are beyond the scope of this thesis but will be briefly discussed in the final chapter.

The goal of this chapter is to provide clarity about the foundational compilation concepts that are ubiquitous across quantum compilers, meanwhile ignoring the complications of creating a fully realistic compiler. This chapter will build a simple quantum compiler using two pieces —Universal Decomposition and the Solovay-Kitaev algorithm—and introduce the theory behind them. Universal Decomposition highlights the role of linear algebraic decomposition in quantum compilers; the Solovay-Kitaev algorithm highlights the importance of approximation in quantum compilers as well as how better approximations are generated from an initial rough approximation. As we shall see, the compiler built in this thesis is conceptually important, but too inefficient for practical purposes. In Chapter 6, we provide a perspective on how the compiler concepts used in this thesis can be extended to create practical programs, review state-of-the-art compilation methodologies and discuss compiler design in the NISQ era.

4.1 What is Quantum Compilation?

To build a quantum compiler, we first need a good understanding of the compilation problem. The core of quantum compiling is solving the problem of implementing an arbitrary unitary transformation on a quantum machine. Mathematically, a quantum algorithm is a unitary transform U on a quantum state $|\psi\rangle$ that produces a computed state $|\psi'\rangle = U|\psi\rangle$. The compilation problem is posed as follows: Given a limited discrete set of gates $\mathcal{G}$ provided by physical hardware. How can we realize U using $\mathcal{G}$?

A simple example would be that the algorithm or target gate $U = R_x(\frac{3\pi}{5})$ is a rotation of $\frac{3\pi}{5}$ radians around the x-axis of the Bloch Sphere and the hardware instruction $\mathcal{G} = \{R_x(\frac{\pi}{4}), R_x(\frac{\pi}{3})\}$ set includes operators that give rotation around the x-axis by smaller angles. For this specific

example, one possible compilation is:

$$R_x(\frac{3\pi}{5}) = R_x(\frac{36\pi}{60}) \approx R_x(\frac{\pi}{4})R_x(\frac{\pi}{3}) = R_x(\frac{35\pi}{60})$$

Here the compiled sequence $R_x(\frac{\pi}{4})R_x(\frac{\pi}{3})$ has two operators from $\mathcal{G}$ and thus a length $l = 2$. This quantity l is an important metric for compilation since we want our compiled program to be short and efficient. The formal definition of gate depth l for a quantum program is the maximum gate count on a single qubit in its quantum circuit. One can visualize gate depth easily from a circuit diagram that can be found in Section 3.3.4. Here we see that the approximation is off by $\frac{36\pi}{60} - \frac{35\pi}{60} = \frac{\pi}{36}$ radians rotation around the Bloch x-axis. How does the approximation error change the probability of measurement outcome that we can observe? Here's where quantifying error with vector and matrix norms comes in handy, as we shall see in Section 4.3 "Approximating Quantum Circuits".

Along with the discussion of error, we make a distinction between what we mean by decomposition and compilation. **Decomposition** refers to the perfect translation of a gate from a single matrix representation U into a product of other matrices $\prod_i M_i$, where $||U - \prod_i M_i|| = 0$. In this case, M_i are matrices that serve as an intermediate-level representation that will be further decomposed or compiled. On the other hand, **compilation** is the approximation of a single matrix representation U using a product of other matrices $\prod_i M_i$, where $||U - \prod_i M_i|| < \epsilon$ and $M_i \in \mathcal{G}$ is from a small discrete set of matrices that represent the gates $\mathcal{G}$ available from hardware.

Another example compilation from [28] involving multiple qubits is shown in the following circuit diagram in Figure 4.1.1. Here the instruction set is $\mathcal{G} = \{H, T, CNOT\}$ that involves the Hadamard, T, and CNOT gate, and the target gate U is the Tiffoli gate on the left-hand side of the figure. On the right-hand side of the figure is the compiled circuit, with a gate depth of 12 (since some operations can be performed in parallel). Here the compilation is a full decomposition, and the approximation error is $\epsilon = 0$.

Gate depth is one measure of circuit complexity. However, circuit complexity and quality can also be measured by the counts of gates that take longer to execute and are more prone to error

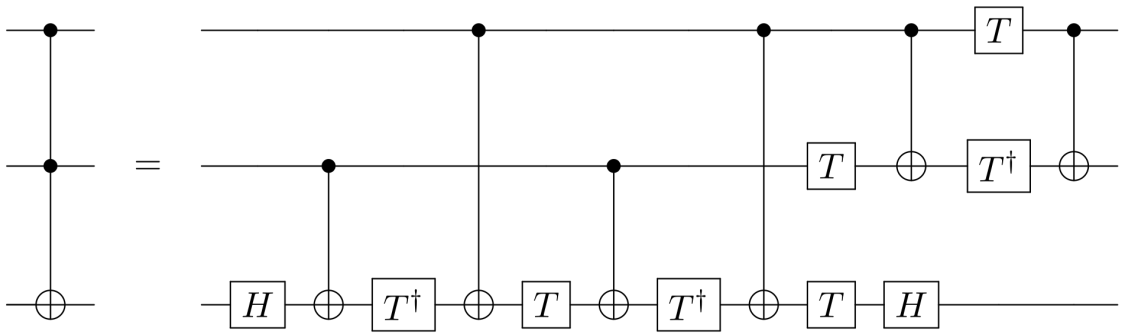

Figure 4.1.1: The optimal decomposition of the Tiffoli Gate using 6 CNOT gates.

than other gates. Because CNOT is ubiquitous across quantum circuits, proportional to gate depth, and costly compared to other gates, CNOT count is another useful measure for circuit complexity. In the example of Tiffoli compilation into H and T in Figure 4.1.1, the CNOT count is 6, which is exactly half the gate depth.

From the above design concerns, compilers have optimization targets such as minimizing CNOT count, circuit depth, and error rate. However, without the help of error correction, executable programs on NISQ devices are subjected to decoherence time, connectivity, and fidelity constraints. These requirements motivate compilers to take into account hardware resources and limitations as well as take advantage of optimization methods to minimize circuit depth or maximize fidelity. These methods are important for a working compiler but are too detailed for this chapter to cover. For now, we will neglect them and focus on the abstract framework of treating compilation as a matrix decomposition and approximation problem, which is at the heart of quantum compilation.

The assumption that we make about hardware is that it will be able to provide us a small and discrete gate set $\mathcal{G}$, that is usually limited to one- or two-qubit operations at the hardware level because of physical and fabrication constraints. In short, compilation lies in the challenge of using a small discrete gate set of 2-level and 4-level unitary operations to cover the full space of general D-level unitary operations. In general, D can be large, and the compilation problem has been shown to be NP-hard. It is often computationally less expensive to rewrite U as a

product of d-level ($d < D$) matrices (which can be thought of as $d \times d$ matrices) with linear algebraic decompositions, and then compile each d-level matrix to hardware instructions $\mathcal{G}$.

Analogous to the classical universal gates set { NAND } or {AND, NOT, OR} that can be combined to compute any classical function, the property that $\mathcal{G}$ can compute any quantum function is called universal, and such $\mathcal{G}$ is called a universal gate set. This section will show that such a universal gate set $\mathcal{G}$ exists as well as introduce an implementation that compiles any arbitrary U with $\mathcal{G}$. We will call this compilation scheme **UDSK** since it has two components Universal Decomposition and Solovay-Kitaev algorithm. As we will see later in this section, UDSK produces a compiled sequence with a length that grows exponentially in the number of qubits involved in the algorithm. Although UDSK is too inefficient for practical applications, we introduce it here as a conceptual appetizer to quantum compilation because its two components are fundamental to the topic.

Universal Decomposition breaks up U as a combination of CNOT and single-qubit operations. In other words, the CNOT and some single-qubit operations together form a universal set for quantum computation. This is a great step forward since hardware operations are usually single-qubit or two-qubit operations! A significant difference between classical and quantum compilation is that the set of unitary operators U in quantum computation is continuous. However, the hardware provides only a discrete set of operations, so we can approximate any arbitrary single-qubit unitary U down to only a given precision such that the corresponding error ϵ is acceptable. We want to know whether such an approximation exists and whether it can be done efficiently. Fortunately, the Solovay-Kitaev theorem provides the theoretical ground to show that, for certain gate sets, it is guaranteed that we can find a compiled sequence with a length that is polylogarithmic (efficient) in the error. A compilation scheme inspired by the theorem [5] has also been developed to implement a Solovay-Kitaev algorithm that can approximate any single-qubit unitary U with a hardware gate set $\mathcal{G}$ to within an error tolerance ϵ.

Geared with Universal Decomposition and the Solovay-Kitaev theorem, we have an inefficient, yet fully functional compilation scheme that can approximate any quantum operation U with a

given hardware gate set $\mathcal{G}$. This chapter will walk the readers through the theories behind and the implementations of Universal Decomposition and the Solovay-Kitaev algorithm, in the hope of providing a concrete layout of how quantum compilation works conceptually and algorithmically.

4.2 Note on Math for Quantum Compiling

The core theories behind quantum computation that will be introduced later in Chapter 4 are universal quantum computation and the Solovay-Kitaev theorem, which build on top of the prerequisites:

- Linear algebraic formalism of quantum circuit model
- The behaviors, identities, and decomposition of single-qubit gates
- Decomposition of D-level unitary matrices into a product of d-level unitary matrices where $d < D$.
- Group theory, Special Unitary Group (2) and Lie Algebra

If the reader is unfamiliar with the math foundation behind quantum computation and compilation, Chapter 4 provides the necessary background for all prerequisites except the last one, which goes beyond the scope of this thesis.

4.3 Approximating Quantum Circuits

Since we will be generating a matrix A, the output of a compiler, to approximate our target unitary matrix U, we want to know how good our approximation is. The linear algebra formalism includes vector and matrix norms that come in handy for us to measure the difference between A and U. There are many useful metrics to measure the quality of matrix approximation, and we will introduce two commonly used ones here: the maximum vector norm and the spectral matrix norm (also commonly known as the l-2 norm) [30].

The **maximum vector norm** is defined as

$$E(U, A) \equiv \max_{|\psi\rangle} \|(U - A)|\psi\rangle\|.$$

It is an intuitive choice that relates to physical quantities. Suppose we act U and A on a state $|\psi\rangle$ and make measurements along the computational basis. Let P_U and P_A be the probabilities of measurement outcomes if U and A are performed. Their difference can be described by

$$|P_U - P_A| = |\langle\psi|U^\dagger M U|\psi\rangle - \langle\psi|A^\dagger M A|\psi\rangle|.$$

Relating the difference between the probability of measurement outcomes to the maximum vector norm using the Cauchy-Schwarz inequality, we see that the error is an upper bound to possible measurement differences if the state were to be acted on by U and A. Let $|\Delta\rangle \equiv (U - A)|\psi\rangle$.

$$\begin{aligned} |P_U - P_A| &= |\langle\psi|U^\dagger M|\Delta\rangle + \langle\Delta|MA|\psi\rangle| \\ &\leq |\langle\psi|U^\dagger M|\Delta\rangle| + |\langle\Delta|MA|\psi\rangle| \\ &\leq \||\Delta\rangle\| + \||\Delta\rangle\| \\ &\leq 2E(U, A). \end{aligned}$$

Thus, if $E(U, A)$ is small, then measurement outcomes on a state acted on by U and A will produce similar probabilities. If we choose a tolerance $\Delta = |P_U - P_A|$ that probabilities of different measurement outcomes obtained from the approximate circuit A have to be within, then we know we have to find A such that $\frac{\Delta}{2} \geq E(U, A)$. The maximum vector norm is useful in quantifying error and directly connects with the physical meaning of differences in probabilities of measurement outcomes.

Another useful measure for matrix approximation is the **spectral norm**, defined as $\|D\| = \sqrt{\lambda_{\max}(D^\dagger D)}$. If we define $D = A - U$, then the spectral norm can be a measure of approximation error $\epsilon = \|D\|$ by definition of matrix norms. The meaning of spectral norm is rather abstract. A way to look at it is to see that since $D^\dagger D$ is Hermitian, it can be thought of as an observable that also describes the value of the difference between A and U. The maximum eigenvalue $\lambda_{\max}\left(D^\dagger D\right)$ is the maximum possible expectation value that we can extract if we were to measure the error on any quantum state. This interpretation of the spectral norm is rather descriptive and needs further rigor to be verified, so take it with a grain of salt. With that said, the spectral norm is a still useful mathematical quantity to characterize approximation error.

4.4 Universal Quantum Computation

Since we have the goal of approximating any arbitrary U up to an error ϵ, we would like to know if a hardware gate set $\mathcal{G}$ has the property to achieve this goal. A set of gates is **universal** for quantum computation if any unitary matrix can be approximated to arbitrary accuracy by a quantum circuit built with only those gates. In other words, a universal gate set can compute any quantum algorithm to any desired tolerance level.

Since we want to run any arbitrary program, It is of interest for hardware to provide a universal instruction set, and so compilation is focused on compiling programs to a universal gate set. The immediate question is whether such a gate set exists. It has been shown that there are plenty of them. For example, 1) the standard gate set: Hadamard, phase, controlled-NOT, and $\pi/8$ gates, and 2) Hadamard, phase, controlled-NOT, and the Tiffoli gate are both universal gate sets. We refer to the fact that there exists a universal gate set as the **universality of quantum computation**.

This section shows the universality of quantum computation, which follows from combining the proofs of the three statements below:

- Statement 1: An arbitrary unitary operator U of d dimensions may be expressed *exactly* as a product of 2-level unitary operators. An n-qubit system may be written as a product of $O(4^n)$ two-level unitary matrices.
- Statement 2: Single qubit and CNOT gates together can be used to implement an arbitrary two-level unitary operation on the state space of n qubits.
- Statement 3: Hadamard and the $\pi/8$ gate can be used to approximate any single-qubit unitary operation to arbitrary accuracy.

We will prove that each statement is true (other versions of proof can be found in [1] and Section 4.5 of [22]). The first two statements of the universality proof are important because, in practice, hardware instruction sets usually provide single-qubit or two-qubit (CNOT) gates, and we want to combine these single- and two-qubit operations to approximate multi-qubit

operations that represent a quantum algorithm. Since we are interested in hardware-gate-set independent compiling and the last statement is specific to the standard gate set, we will leave the last out for further reading in section 4.5.3 of [22]. Instead, we will apply the Solovay-Kitaev theorem, which is gate set independent, to replace the role of the last statement. The universality proof can also be implemented to become a crude compiler, and incorporating the Solovay-Kitaev algorithm allows the compilation to be hardware gate set independent.

4.4.1 Proof of Statement 1: 2-level Systems are Universal

Statement 1 essentially says that 2-level systems are universal. The construction for Statement 1 effectively breaks down a multi-qubit operation into a series of single-qubit operations that are represented by 2-level unitaries. Consider a unitary matrix U which acts on a d-dimensional Hilbert space and represents a quantum algorithm. In this section, we explain how U may be decomposed into a product of two-level unitary matrices; that is, unitary matrices which act non-trivially only on two-or-fewer vector components.

$$U = \prod_i A_i, \text{ where } A_i \text{ are 2-level unitaries.}$$

The approach is matrix reduction with algebra, applied iteratively. Suppose in U there are neighboring elements a and b in the first row at the ith and $(i+1)$th position such that $ab \neq 0$. We multiply it by a 2-level unitary matrix A_i such that we eliminate the element at the $(i+1)$th element.

$$\begin{bmatrix} \dots & a & b & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \end{bmatrix} A_i = \begin{bmatrix} \dots & c & 0 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \end{bmatrix}$$

To see that we can always find such a two-level unitary matrix A_i given a, b, and c, we choose A_i to be acting on elements $(i, i+1)$ with non-trivial unitary 2×2 submatrix A', where we can simplify the condition on A_i to be:

$$\begin{bmatrix} a & b \end{bmatrix} A' = \begin{bmatrix} c & 0 \end{bmatrix}$$

In general, a 2-level special unitary matrix can be expressed using three parameters θ, λ, and μ.

$$A' = \begin{bmatrix} \cos\theta e^{i\lambda} & \sin\theta e^{i\mu} \\ -\sin\theta e^{-i\mu} & \cos\theta e^{-i\lambda} \end{bmatrix}$$

Suppose $ab \neq 0$ and substitute in the parameters, we get

$$\begin{cases} a\cos\theta e^{i\lambda} - b\sin\theta e^{i\mu} = c \\ a\sin\theta e^{-i\mu} + b\cos\theta e^{-i\lambda} = 0 \end{cases}$$

Solving them, we get

$$\theta = \arctan\left(\left|\frac{b}{a}\right|\right); \lambda = -\arg(a); \mu = \pi + \arg(b)$$

Note the special case $ab = 0$ that would give undefined values. If $b = 0$, we don't have to do anything since there is nothing to eliminate. If $a = 0$ and $b \neq 0$, we switch the ith and $(i+1)$th column by taking $U' = X$ and continue. This moves non-zero elements to the left in the first row. The above procedure guarantees us to find the 2-level unitary A_i that eliminates one element at a time.

Now we apply this procedure repeatedly. Let $U = U_0$ for ease of indexing. The first step $U_0 A_{d-1} = U_1$ gives us a matrix U_1 with the dth element being 0. Repeating $d-1$ times $U_{i-1}A_{d-i} = U_i$ for $i = 1...(d-1)$ with the final step $U_{d-2}A_1 = U_{d-1}$, we will be able to reduce all elements in the first row to until only the first element is non-zero. Expanding all steps, we have $U_{d-1} = (U_{d-2})A_1 = (U_{d-3}A_2)A_1 = ... = U_0 A_{d-1}...A_2A_1$. Since U and A_i's are unitary, it is guaranteed that U_{d-1} is also unitary. Since U_{d-1} have only the first element non-zero in the first row and also $U_{d-1}^\dagger U_{d-1} = I$, the first column of U_{d-1} will all be zeros except for the first element. The unitariness of U_{d-1} guarantees that the first diagonal element is one. In short, the reduction of the 1st row through $U_{d-1} = U_0 A_{d-1}...A_2A_1$ produces a $(d-1)$-level matrix U_{d-1} in the form

$$\begin{bmatrix} 1 & 0 & ... & 0 \\ 0 & * & * & * \\ ... & * & * & * \\ 0 & * & * & * \end{bmatrix}$$

Let's call the elimination for the first row $E_1 = A_{d-1}...A_2A_1$ producing a first-row-eliminated matrix U_{r1}, such that $U_{d-1} = U_0E_1 \equiv U_{r1}$. We repeat row reduction $U_{ri} = U_{r(i-1)}E_i$ for $i = 1...(d-2)$. We will have all non-diagonal elements zero and diagonal elements one up until the $(d-2)$th row and column. After all the row reductions, the 4 special non-trivial elements will be in the bottom right corner, forming a 2x2 unitary matrix. Call this final eliminated

matrix A_f. The following relation connects the elimination of all rows:

$$U_0 E_1 E_2 ... E_{d-2} = A_f.$$

For a matrix U with $d = 4$, the elimination process looks like:

$$\begin{bmatrix} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & * & * & * \\ 0 & * & * & * \\ 0 & * & * & * \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & * & * \\ 0 & 0 & * & * \end{bmatrix}$$

Now if we multiply the conjugates of the elimination matrices to the right side of the equation, we can express our target matrix $U = U_0$ as a product of A_f and E_i's.

$$U = U_0 = A_f E_{d-2}^{\dagger} E_{d-3}^{\dagger} \dots E_1^{\dagger}$$

Since E_i's are products of two-level matrices, U is now expressed as a product of two-level matrices, and we have shown Statement 1 to be true. Note that each E_i is a product of $(d - i)$ two-level unitaries, so in total, we can express U as a product of $1+[2+3+...+(d-2)+(d-1)] = \frac{d(d-1)}{2} \rightarrow O(d^2) = O(4^n)$ two-level unitaries.

4.4.2 Single-qubit and CNOT Gates are Universal.

Statement 1 says that 2-level unitaries are universal. Statement 2 says that single-qubit and CNOT gates can implement any arbitrary 2-level unitary matrix. Combining Statement 1 and Statement 2, we have that single-qubit and CNOT gates are universal. Now let us see why Statement 2 is needed and how it is true.

From the proof of Statement 1, we have a decomposition of our 2^n-dimensional target matrix U on an n qubit quantum computer into two-level matrices A_i that act on the pairs of basis states with $(i, i + 1)$. Let A_i' be the 2x2 unitary submatrix of A_i; then A_i' can be thought of as a single-qubit gate. Thinking from the perspective of the circuit model and single-qubit operations provided by hardware, operations on the ith and $(i+1)$th state are not straightforward to execute. Instead, acting on a pair of states that differ only on the ith bit is; that is, as simple as acting A' on the ith qubit among all n qubits. For example, the binary strings 000 and 001

can be mapped to the bases $|000\rangle$ and $|001\rangle$ that represent the 3rd qubit in a 3 qubit quantum computer when the 1st and 2nd qubits are in the $|00\rangle$ state. The idea is that if we have two binary strings representing basis states differing in exactly one bit, say the jth bit, we can apply a fully controlled-A' operation on the jth qubit, conditioned on all the other qubits being the same. Effectively, this means that we can apply a single-qubit operation, or a 2-level unitary, in a n-qubit state space, affecting only the jth qubit while leaving others unchanged. This is an operation that is straightforward for hardware to execute.

Gray coding can transform the basis such that $(i, i+1)$ becomes $\left(i, i \oplus 2^k\right)$, where $\oplus$ means binary add. Gray code is a sequence of codes that connect two binary strings s and t such that neighbors in the sequence differ in exactly one bit ($s = i$ and $t = i + 1$ in our case). For any n-qubit quantum computer with 2^n basis states indexed $0, 1, 2..., 2^n - 1$, we can find their Gray code through the binary-reflected Gray code permutation [8], given by

$$\pi_i = i \oplus \lfloor i/2 \rfloor, \text{ where } i = 0, 1, 2..., 2^n - 1.$$

For example, the Gray code for $n = 3$ is $(0, 1, 3, 2, 6, 7, 5, 4)$, which in binary, is (000, 001, 011, 010, 110, 111, 011, 010). Note that each neighboring string only differs in one bit as desired.

An example shows how Gray code is applied with fully controlled-NOT operations to perform basis transformation then introduce the general framework in the matrix notation. Suppose our goal is to implement the two-level unitary transformation

$$A = \begin{bmatrix} a & 0 & 0 & 0 & 0 & 0 & 0 & c \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ b & 0 & 0 & 0 & 0 & 0 & 0 & d \end{bmatrix}$$

Here, a, b, c and d are any complex numbers such that $A' \equiv \begin{bmatrix} a & c \\ b & d \end{bmatrix}$ is a unitary matrix .

Now A acts only on states $|000\rangle$ and $|111\rangle$, whose binary strings can be connected by the following Gray code:

A	B	C
0	0	0
0	0	1
0	1	1
1	1	1

From the Gray code table, we can create the desired circuit, shown in Figure 4.4.1. The first two gates shift basis so that $|000\rangle$ gets swapped with $|011\rangle$. Then, the operation A' is applied to the first qubit that differs in the states $|011\rangle$ and $|111\rangle$, conditioned on the second and third qubits being in the state $|11\rangle$. Finally, we invert the basis transformation, so that $|011\rangle$ becomes $|000\rangle$ again.

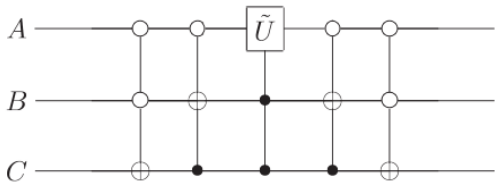

Figure 4.4.1: Circuit implementing the two-level unitary operation. This is also Figure 4.16 from [22].

From this example, we see that a two-level unitary A can be expressed as a circuit that consists of $2(n-1) \rightarrow O(n)$ fully controlled-NOT operations plus a fully controlled single-qubit A' operation. In Section 3.3.5, we showed that a general controlled operation $C^n(A')$ can be implemented with $O(n)$ CNOT and single-qubit operations. Here our fully controlled operations are $C^n(A')$ in the case where $k = 1$, so each of them can also be implemented with $O(n)$ CNOT and single-qubit gates. It follows that a two-level unitary A can be implemented with $O(n^2)$ CNOT and single-qubit gates. Since our target gate U can be expressed as a product of $O(4^n)$ two-level unitaries, it can be implemented with $O(4^n n^2)$ CNOT and single-qubit gates. This is quite inefficient!

To see how the decomposition of U into CNOT and single-qubit operations can be done more generally, we start with the matrix formulation. Consider the permutation matrix $P \in \mathrm{U}(2^n)$, where $P_{ij} = \delta_{i,\pi_j}$, that permutes the ith element of a vector with permutation π_j. The expression

$P^\dagger \tilde{A}_i P$ simultaneously permutes rows and columns of matrix $\tilde{A}_i$ with permutation π. If $\tilde{A}_i$ was a two-level matrix acting on states $(i, i+1)$, then $A_i = P^\dagger \tilde{A}_i P$ will be a two-level matrix acting on states (π_i, π_{i+1})—exactly what we wanted.

Our objective is to express $U = \prod_i A_i$ as a product of two-level matrices A_i acting on states differing in one bit. Using our permutation, we have $U = \prod_i A_i = \prod_i (P^\dagger \tilde{A}_i P) = P^\dagger (\prod_i \tilde{A}_i) P$. If we define $D = PUP^\dagger$, and apply two-level decomposition $D = \prod_i \tilde{A}_i$ as in Statement 1, then we will have obtained our desired decomposition of $U = \prod_i A_i$ into two-level unitary matrices $A_i = P^\dagger \tilde{A}_i P$ acting on states differing in one bit.

Here we introduced the Gray code permutation matrix P. Its function is the same as the fully controlled-NOT gates in Figure 4.4.1—to perform basis transformation, but represented with a matrix. From Figure 4.4.1, any two-level unitary acting on states differing in one bit can be implemented with fully controlled gates which, in turn, can be implemented by CNOT gates and single-qubit gates. Thus, combined with Statement 1 we have shown that CNOT and single-qubit gates are universal.

4.4.3 Proof for Statement 3

Using Statements 1 and 2, if we are given CNOT gates, the compilation problem has been reduced to the problem of implementing an arbitrary single-qubit gate given a native single-qubit gate set $\mathcal{G}$. This is what we will need to show to prove Statement 3. There are documented gate-set-specific proofs of Statement 3. Interested readers can find the universality of Hadamard + phase + CNOT $+\pi/8$ gates in section 4.5.3 of [22] and the universality of rotation gates around the Bloch basis in [7] for instance. However, this thesis is interested in gate set independent compilation, so we will replace the role of Statement 3 to compile single-qubit gates with the Solovay-Kitaev algorithm.

4.5 Universal Decomposition

Universal Decomposition refers to the algorithm that applies Statement 1 and 2 to decompose an arbitrary matrix U. The proofs of Statement 1 and Statement 2 are constructive, so we can simply use their procedures as our algorithm to decompose any arbitrary unitary matrix U into CNOT gates and single-qubit gates. We mainly follow the implementation of [7], but the procedures are also described in [15] and [1]. The function of Statement 3 in the compiler will be replaced by the Solovay-Kitaev algorithm.

4.6 The Solovay-Kitaev theorem

The Solovay-Kitaev theorem states that for any unitary gate U on a single qubit, and given any precision $\epsilon > 0$, it is possible to approximate U to the precision using $\Theta(\log^c(1/\epsilon))$ gates from a fixed finite set $\mathcal{G}$, where c is a small constant approximately equal to 2. It is known that c cannot be smaller than 1, but what value it takes between 1 and 2 is an open problem [38]. The Solovay-Kitaev theorem guarantees that we can find an efficient compiled sequence for a single-qubit gate U to a desired approximation ϵ! We will give a high-level overview of the proof of the Solovay-Kitaev theorem. For steps of the proof that require understanding of Lie Algebra and Lie Group theory, we will simply state their result and leave reference to details for interested readers. This Section mainly references [23] and appendix 3 of [22].

4.6.1 Useful Definitions

Here we introduce some useful definitions to help us state the Solovay-Kitaev theorem precisely and walk through why the theorem is true. Since we are introducing the Solovay-Kitaev theorem in the context of compiling single-qubit gates into hardware gates, which are all 2-level unitaries, we can represent our compiled program as a sequence of gates from the hardware gate set. We say that a word of length l from $\mathcal{G}$ is a product $g_1 g_2 \dots g_l \in SU(2)$, where $g_i \in \mathcal{G}$ for each i. Define $\mathcal{G}_l$ to be the set of all words of length at most l, and $\langle \mathcal{G} \rangle$ to be the set of all words of finite length.

Similar to Section 4.3, we need a distance measure to quantify the approximation quality. Here we follow [22] and use the **trace distance**, defined as $D(U,V) \equiv tr|U-V|$, where $|X| \equiv \sqrt{X^\dagger X}$. The reason for choosing this distance is that it is helpful to think of elements in $SU(2)$ as points in space, and around the identity $D(U,I)$ is a good approximation to the Euclidean distance. We are interested in finding gate sets that are universal—gates that cover the operator space within an error. In group theory terminology, we want to find a group that is **dense**. A subset S of $SU(2)$ is said to be dense in $SU(2)$ if for any $U \in SU(2)$ and $\epsilon > 0$ there exists an element $s \in S$ such that $D(s,U) < \epsilon$. Suppose S and W are subsets of $SU(2)$. Then S is said to form an ϵ-net for W, where $\epsilon > 0$ if every point in W is within a distance ϵ of some point in S. For compilation, we are interested in how quickly $\mathcal{G}_l$ covers $SU(2)$ as l increases. In other words, given l, how small an ϵ is $\mathcal{G}_l$ an ϵ-net of $SU(2)$? The Solovay-Kitaev theorem states that ϵ gets small rapidly as l is increased. However, it builds on the assumption that $\langle \mathcal{G} \rangle$ is **dense**. Formally stated, the Solovay-Kitaev theorem is,

> Let $\mathcal{G}$ be a finite set of elements in $SU(2)$ containing their inverses, such that $\langle \mathcal{G} \rangle$ is dense in $SU(2)$. Let $\epsilon > 0$ be given. Then $\mathcal{G}_l$ is an ϵ-net in $SU(2)$ for $l = O\left(\log^c(1/\epsilon)\right)$, where c is some constant.

4.6.2 Outline of Proof

The high-level overview of the Solovay-Kitaev theorem involves two main pieces: the **shrinking lemma** that guarantees a longer $\mathcal{G}_l$ sequence generates denser dense nets around the identity and the **translation step** that applies shrinking lemma iteratively to increase sequence length with better approximation from an initial crude approximation $\mathcal{G}_{l_0}$. The two main pieces are visualized in Figure 4.6.2 and 4.6.1, which are helpful in terms of getting an intuitive feeling of how the proof works.

4.6.3 Shrinking Lemma

Formally stated, the shrinking lemma says that there exists a universal constant ϵ_0 such that for any $\mathcal{G}$ and $\epsilon \leq \epsilon_0$, we have:

$$\mathcal{G}_l \text{ is an } \epsilon^2\text{-net for } S_\epsilon \implies \mathcal{G}_{5l} \text{ is an } s\epsilon^3\text{-net for } S_{\sqrt{s\epsilon^3}}, \text{ for some constant } s.$$

Effectively, the shrinking lemma says that by scaling the length of a sequence G_l by a factor of five, we are guaranteed an exponentially denser net. Notice that the parameters are mapped as follows after applying the shrinking lemma: $(l, \epsilon^2, \epsilon) \mapsto (5l, s\epsilon^3, \sqrt{s\epsilon^3})$. Thus, if we apply the shrinking lemma iteratively k times on an initial set of parameters $(l_0, \epsilon_0^2, \epsilon_0) \mapsto (5^k l_0, (s\epsilon_0)^{3^k}/s^2, (s\epsilon_0)^{(\frac{3}{2})^k}/s)$. This result give us the **iterated shrinking lemma** for some integer k:

$$\mathcal{G}_{l_0} \text{ is an } \epsilon_0^2\text{-net for } S_{\epsilon_0} \implies \mathcal{G}_{l_k} \text{ is an } \epsilon_k^2\text{-net for } S_{\epsilon_k}, \text{ where } l_k = 5^k l_0 \text{ and } \varepsilon_k = (s\varepsilon_0)^{(3/2)^k}/s.$$

Since $\langle \mathcal{G} \rangle$ is dense in $SU(2)$, we make l_0 sufficiently large such that G_{l_0} is an ϵ_0^2-net for $SU(2)$. It follows from the iterated shrinking lemma that we can cover the entire operator space around the identity within an error as arbitrarily small as ϵ_k, as long as we increase the sequence length l_k. Let's see how the translation step applies the shrinking lemma iteratively to cover the entire operator space, instead of only around the identity.

4.6.4 Translation Step

Suppose we have $U_0 \in \mathcal{G}_{l_0}$, an $\epsilon(0)^2$-approximation to U (this will always be true if we choose sufficiently large l_0 since $\mathcal{G}$ is dense). Define $\Delta_1 \equiv U U_0^\dagger$ to represent the "difference" between U and U_0. It follows that $\Delta_1 \in S_{\epsilon_1}$ by $\|\Delta_1 - I\| = \left\|(U - U_0) U_0^\dagger\right\| = \|U - U_0\| < \epsilon_0^2 < \epsilon_1$, where ϵ_1 is the universal constant from the shrinking lemma. Now we apply the iterated shrinking lemma with $k = 1$ to approximate Δ_1, the leftover difference. The lemma says that there exists an approximation $U_1 \in \mathcal{G}^{l_1}$ such that $\|\Delta_1 - U_1\| = \left\|U U_0^\dagger - U_1\right\| = \|U - U_1 U_0\| < \epsilon_1^2$. Notice that $U_1 \in \mathcal{G}^{l_1}$ is a $l_1 = 5l_0$ sequence. By adding $5l_0$ gates (U_1) to the initial l_0 gates (U_0), we

have improved our approximation from ϵ_1 (or ϵ_0^2) to ϵ_1^2. We call this process the translation step, which can be visualized in 4.6.1.

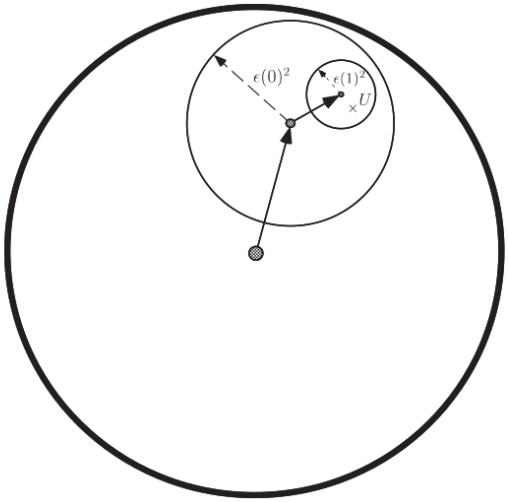

Figure 4.6.1: The translation step used in the proof of the Solovay-Kitaev theorem. To approximate a single-qubit gate U we first approximate to within a distance $\epsilon(0)^2$ using l_0 gates from $\mathcal{G}$. Then we improve the approximation by adding $5l_0$ more gates, for a total accuracy better than $\epsilon(1)^2$, and continue on this way, quickly converging to U.

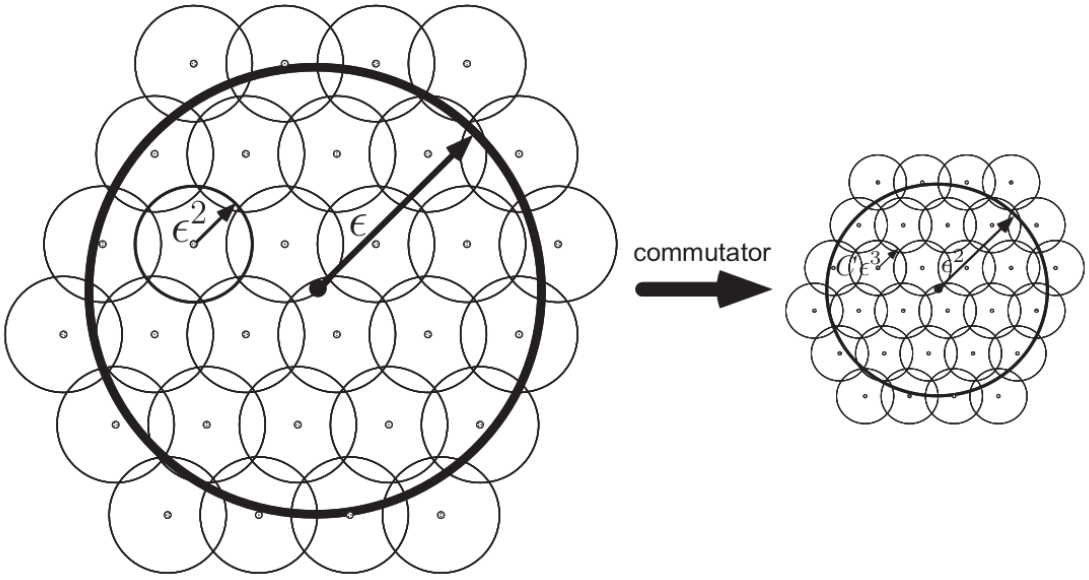

Figure 4.6.2: The main idea of the shrinking lemma. Taking group commutators of elements U_1 and U_2 dense in ϵ-net fills in a denser ϵ^2-net.

4.6.5 Proof of Solovay-Kitaev theorem

From our initial approximation, we repeat the translation step on the leftover difference until our error is small enough. Following the same technique, define $\Delta_2 \equiv \Delta_1 U_1^\dagger = U U_0^\dagger U_1^\dagger$. Then $\|\Delta_2 - I\| = \left\|(U - U_1 U_0) U_0^\dagger U_1^\dagger\right\| = \|U - U_1 U_0\| < \epsilon_1^2 < \epsilon_2$, and so $\Delta_2 \in S_{\epsilon_2}$. Again, we apply the iterated shrinking lemma with $k = 2$ to obtain $U_2 \in \mathcal{G}^{l_2}$ such that $\|\Delta_2 - U_2\| = \left\|U U_0^\dagger U_1^\dagger - U_2\right\| = \|U - U_2 U_1 U_0\| < \epsilon_2^2$. Continuing k steps this will result in $U_k \in \mathcal{G}^{l_k}$ such that

$$\|U - U_k U_{k-1} \cdots U_0\| < \varepsilon_k^2$$

Each $U_i \in \mathcal{G}^{l_i}$, where $l_i = 5^i l_0$. Thus, our overall approximation sequence has L gates, where

$$L = \sum_{m=0}^{k} l_m = \sum_{m=0}^{k} 5^m l_0 = \frac{5^{k+1} - 1}{4} l_0 < \frac{5}{4} 5^k l_0,$$

and is up to accuracy ϵ_k^2. To find the value of k, we set $\epsilon_k^2 = \left((s\epsilon_0)^{(3/2)^k} / s\right)^2 = \epsilon$ and solve for k:

$$\left(\frac{3}{2}\right)^k = \frac{\log\left(1/s^2\epsilon\right)}{2\log\left(1/s\epsilon_0\right)}$$

Note that we can always choose ϵ_0 slightly smaller so that the obtained value of k is an integer. Let $c = \log 5 / \log(3/2) \approx 3.97$ so that $5^k = \left(\frac{3}{2}\right)^{kc}$. Then

$$L < \frac{5}{4} 5^k l_0 = \frac{5}{4} \left(\frac{3}{2}\right)^{kc} l_0 = \frac{5}{4} \left(\frac{\log\left(1/s^2\epsilon\right)}{2\log\left(1/s\epsilon_0\right)}\right)^c l_0$$

Hence, for any $U \in SU(2)$ there is a sequence of $L = O\left(\log^c(1/\epsilon)\right)$ gates that approximates U to accuracy ϵ, and we have shown the Solovay-Kitaev theorem.

4.6.6 Proof of the Shrinking Lemma

We have only used the result of the Shrinking Lemma but not proved it. To fully understand the proof of the shrinking lemma requires a good knowledge of Lie Group and Lie Algebra that goes beyond the scope of this thesis, but a good high-level description from Harrow [9] of how they work is as follows (the group commutator of unitary gates V and W is $[V, W]_{gp} = VWV^\dagger W^\dagger$):

> Specifically, near the identity the group commutator (which is easily expressible with strings of matrices) approaches the algebra commutator (which is easily calculable), and by moving back and forth between these two ways of looking at operators we can express precise matrices near the identity as strings of less precise matrices that are farther from the identity" [9].

Figure 4.6.2 also offers a geometric view of how the shrinking lemma works. The main function of the shrinking lemma is that for each iteration it is applied, we increase the length of the approximation sequence by a factor of 5, meanwhile decreasing the error exponentially.

$$l_1 \to 5l_0$$
$$\epsilon_1 \to \epsilon_0^{2/3}$$

We are guaranteed to get better approx $U_2 \approx U$ with a longer sequence by the property of group commutators of first approximations U_1 and U_0. Why this works requires an understanding of Lie Algebra and Lie Group theory, and is explained in [23] and appendix 3 of [22].

4.7 The Solovay-Kitaev Algorithm

Although the Solovay-Kitaev theorem says that we can find an efficient compilation, it does not tell us how to find it. Dawson and Nielsen noticed that the proof of the Solovay-Kitaev theorem shares a similar structure with a recursive program. They recursively apply the properties of group commutators that are used in the proof to develop a Solovay-Kitaev algorithm [5]. This section will be an implementation guide that introduces the conceptual pieces and algorithmic steps of the Solovay-Kitaev algorithm while skipping the details of why it is proven to work. References to details will be provided for interested readers. We will also discuss the complexity of the Solovay-Kitaev algorithm. An implementation will be run and analyzed in the Experiments Section. For now, let us start with the pseudo-code of the Solovay-Kitaev theorem:

Function Solovay-Kitaev (Gate U , depth d)

If $(d == 0)$

Return BasicApproximation(U)

Else

Set $U_{d-1} =$ Solovay-Kitaev$(U, d-1)$

$WVW^\dagger V^\dagger =$ **GCDecompose**$(UU_{d-1}^\dagger)$

Set $V_{d-1} =$ Solovay-Kitaev$(V, d-1)$

Set $W_{d-1} =$ Solovay-Kitaev$(W, d-1)$

Return $U_d = V_{d-1}W_{d-1}V_{d-1}^\dagger W_{d-1}^\dagger U_{d-1}$

As we see from the pseudo-code, the function Solovay-Kiteav theorem calls itself. This means that it is a recursive algorithm. The interesting parts are then the other portions of the non-recursive code that do most of the heavy lifting: Basic Approximation and GCDecompose. We follow the method described in [5], and we will summarize how both steps are implemented.

Before we go into the details, notice the subtleties of the inputs of the Solovay-Kitaev function interface. For a general quantum compiler program the inputs are: the gate set available from hardware $\mathcal{G}$, the target gate that describes a quantum program U, and a tolerable error ϵ. Notice that $\mathcal{G}$ and ϵ are not explicitly stated in the Solovay-Kitaev pseudo code. $\mathcal{G}$ is implied in BasicApproximation through a preprocessing step and ϵ is implied together by the recursive depth d and also the BasicApproximation. We use the spectral norm introduced in 4.3 to evaluate the error between the approximation A and the target matrix U.

$$\|D\| = \sqrt{\lambda_{\max}\left(D^\dagger D\right)}, \text{ where } D \equiv U - A$$

4.7.1 Basic Approximation

BasicApproximation corresponds to the translation step in the Solovay-Kitaev theorem. The goal is to find an initial crude approximation $U_0 \approx U$ within an error ϵ_0 using l_0 gates from $\mathcal{G}$. A compilation problem typically asks for an error ϵ tolerable by the program as input. As we

shall see, ϵ can also be implied with the input d and a choice of l_0. Given the gate set $\mathcal{G}$, we can find a relation between ϵ_0 and l_0 for $SU(2)$:

$$l_0 \geq O\left(\frac{3}{\log|\mathcal{G}|}\log\left(1/\epsilon_0\right)\right),$$

so instead we can use l_0 as a parameter to control the initial error. We can enumerate all gate sequences of length $\leq l_0$ from $\mathcal{G}$, and we are guaranteed by the theorem to find an approximation $U_0 \approx U$ to any gate U within error ϵ_0, which can be quantified but will not be done explicitly here. The tolerable error ϵ is related to the initial error ϵ_0 through the shrinking lemma, and so we are guaranteed a small ϵ as long as we apply the shrinking lemma sufficient times.

The implementation of **BasicApproximation** is a lookup table of gate sequences with length $\leq l_0$. The table stores a total of $(|\mathcal{G}|^1+|\mathcal{G}|^2+|\mathcal{G}|^3+...+|\mathcal{G}|^{l_0}) = |\mathcal{G}|^{l_0+1}-1$ sequences, each resulting in a product of a 2x2 unitary matrix. Since a 2x2 unitary matrix can be parameterized by 3 real numbers, we can use a KDTree [35], which is efficient for querying neighbors in n-dimensional data, as our lookup table. KDTree provides an average query complexity of $O(\log(|\mathcal{G}|^{l_0+1}))$. Practically, once we have chosen l_0, this query time becomes a constant $O(c)$ for a specific gate set.

4.7.2 Group Commutator Decomposition

GCDecompose, corresponding to the shrinking lemma in the theorem, is a procedure of decomposing a precise unitary matrix into a product of less precise balanced group commutators. Through the decomposition, the length of the compiled sequence is increased but improvements in the approximation are guaranteed. Each decomposition decreases the error by $\epsilon_d \equiv c_{\text{approx}}\,\epsilon_{d-1}^{3/2}$, for some small constant c_{approx}. The algorithmic procedures of GCDecompose(U) involve trivial math and rotations of matrices around the Bloch Sphere. Why the group commutator decomposition works is highly mathematical. We will skip the details for both the algorithmic steps as well as the proof. Interested readers can refer to section 4.1 of [5].

Examining the pseudo-code of the Solovay-Kitaev theorem, we observe that at each recurrence: the error ϵ reduces exponentially by property of balanced group commentator near the identity,

the gate depth l increases fivefold through group commutator decomposition, and the runtime t is triple the runtime of the previous recurrence (since the function calls itself three times). In mathematical language, we have:

$$\epsilon_d = c_{\text{approx}}\, \epsilon_{d-1}^{3/2}$$

$$l_d = 5 l_{d-1}$$

$$t_d \leq 3 t_{d-1} + \text{ const.}$$

By recursive relations, we can establish the complexity bounds for the error, gate depth, and runtime for the Solovay-Kitave algorithm by:

$$\epsilon_d = \frac{1}{c_{\text{approx}}^2} \left(\epsilon_0 c_{\text{approx}}^2\right)^{\left(\frac{3}{2}\right)^d}$$

$$l_d = O\left(5^d\right)$$

$$t_d = O\left(3^d\right)$$

In this chapter, we have gone through Universal Decomposition and Solovay-Kitaev algorithm as well as how they can be combined to make a working compiler that we name UDSK. We have discussed their function and complexity, but have not shown how they work in practice. In the following chapter, Chapter 5, "Experiments", we will give an overview of an implementation of the UDSK compiler, including Universal Decomposition and the Solovay-Kitaev algorithm, and demonstrate examples of compilation results using USDK.

5
Experiments

The previous chapter provided all the conceptual foundations required to build a quantum compiler. Specifically, the compiler UDSK that combines Universal Decomposition and the Solovay-Kitaev algorithm is introduced. To verify that our assumptions and analysis about USDK hold, I implemented the UDSK compiler (code along with explanation and example files can be found online at the GitHub repository `https://github.com/Hazarre/quantum-compilation`). In this chapter, we will discuss the implementation of the UDSK compiler, run compilation examples on it, and analyze its complexity.

5.1 Notes on Implementation

The UDSK compiler implementation can be found in the `UDSKcompiler.ipynb` file in the project repository. I modified and combined the solution from [7] for Universal Decomposition and the SolovayKitaevSynthesis interface from Qiskit [24] for the Solovay-Kitaev algorithm. We build our UDSK compiler on [7] and [24] for ease of implementation and generating circuit diagrams. However, the SolovayKitaevSynthesis interface follows Qiskit conventions and is rather difficult to unpack. For conceptual clarity and ease of analysis, another version of the Solovay-Kitaev algorithm and Universal Decomposition are separately implemented from scratch in the `SKalgo.ipynb` and the `UD.ipynb` file.

A subtlety in our implementation is that Universal Decomposition breaks a general unitary matrix into a product of two-level unitaries—that can be represented with fully controlled gates—but does not further break the fully controlled gates into CNOT plus single-qubit gates as was described in the previous chapter. Breaking down fully controlled gates into CNOT and single-qubit gates can be implemented with a not-so-interesting fixed routine shown in Figure 3.3.8. We leave it out for simplicity.

5.2 Compilation of Notable Gates

Here, an example of compilations of notable gates into $\mathcal{G}$ gates will be illustrated to showcase the function of UDSK. If not otherwise specified, it is implied that the hardware gate set $\mathcal{G} = \{H, T\}$ contains the Hadamard and T-gate. Note that we have discussed the parameters, initial approximation length l_0 and d recursive depth, in the context of the Solovay-Kitaev theorem. These parameters will be analyzed separately. Choosing l_0 and d also allows us to bound error ϵ. However, quantifying error requires math that goes beyond this thesis.

The compiled circuit of when the target gate U is the Swap gate can be found in Figure 5.2.1 and when U is the Tiffoli gate in Figure 5.2.2. From gate identities, the sequence $HTTTTH = X$ is a quantum NOT gate. Then the two compiled circuits are exactly equivalent to the circuits of the Swap gate in Figure 3.3.3 that exchanges the coefficients of two qubits and the Tiffoli gate that flips the target bit if the other two control bits are both one. These results verify the correctness of the UDSK compiler.

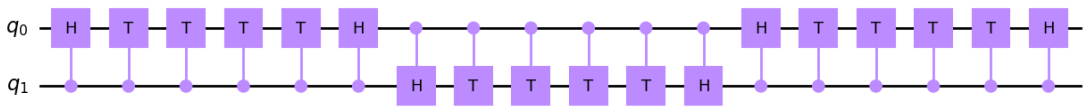

Figure 5.2.1: Compiling the Swap gate into H and T gates.

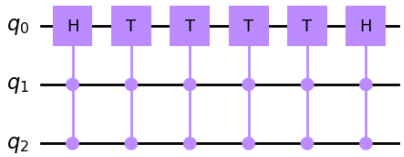

Figure 5.2.2: Compiling the Tiffoli gate into H and T gates.

5.3 Complexity Analysis

5.3.1 Universal Decomposition

Universal Decomposition is executed for randomly generated unitary matrices acting on n-qubit state space, for various n. The results are plotted in Figure 5.3.1, with an average compiled sequence length $L = .44 \cdot 4^n = O(4^n)$ and average runtime of $T \approx 6 \cdot 10^{-7} \cdot 11^n = O(11^n)$. The result confirms the analysis in Section 4.4.1 that Universal Decomposition decomposes arbitrary unitaries acting on n-qubit state space into a product of $O(4^n)$ two-level unitaries. In principle, the runtime is also $O(4^n)$, since computing each two-level unitary matrix takes the same fixed row reduction procedure. How we obtained $O(11^n)$ requires further analysis.

5.3.2 Solovay-Kitaev algorithm

In Section 4.7, we discussed that the recursive depth d and the initial approximation length l_0 can implicitly control the quality of the approximation. The parameter l_0 is used in Basic Approximation so it will be analyzed separately from the full Solovay-Kitaev algorithm. I run **BasicApproximation** on a randomly generated single-qubit U with l_0 ranging from 1 to 23. Name the output approximation A. The error $\epsilon_0 = |U - A|$ will be recorded using the spectral norm. Additionally, the average initialization time to generate the KDTree lookup table and the average query to look up an item from the KDTree are also recorded. The result is shown in Figure 5.3.2. All lines are plotted with the vertical axis log scaled. The data points follow a straight line for initialization time, average error, and average query time, so they are all exponential to l_0. Here we can see that the best-fit line for the error has a negative slope so it

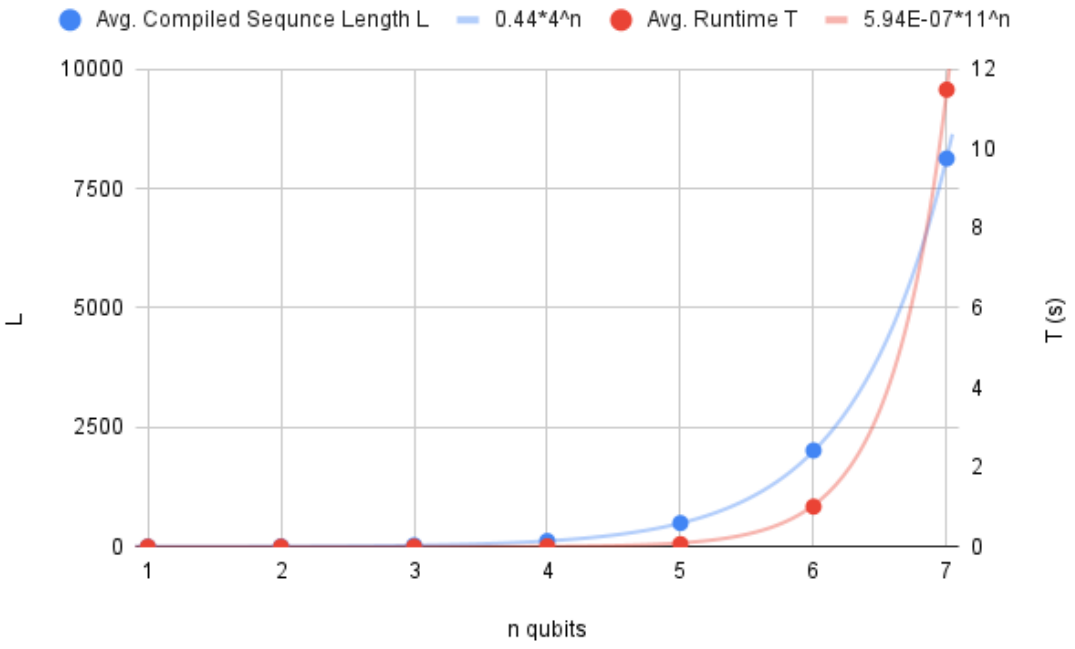

Figure 5.3.1: The Average Compiled Sequence Length and Average Runtime of Universal Decomposition versus the number of qubits n that defines the size of the unitary operator state space. Dots indicate data points and curves the best fit lines.

decreases exponentially as l_0 grows. This is consistent with our analysis of the relation between the initial approximation error ϵ_0 and the initial approximation sequence length l_0

$$l_0 \geq O\left(\frac{3}{\log|\mathcal{G}|}\log\left(1/\epsilon_0\right)\right).$$

From Section 4.7, the complexities of error, runtime, and compiled sequence length for the Solovay-Kitaev algorithm are as follows:

$$\epsilon_d = \frac{1}{c_{\text{approx}}^2}\left(\epsilon_0 c_{\text{approx}}^2\right)^{\left(\frac{3}{2}\right)^d}$$
$$t_d = O\left(3^d\right)$$
$$l_d = O\left(5^d\right)$$

I ran the Solovay-Kitaev algorithm with $l_0 = 22$ fixed and d ranging from 1 to 10 on randomly generated single-qubit unitary matrices. The experimental results are displayed in Figure 5.3.3. The runtime and compiled sequence length obtained from the experiment are $t_d \approx O(3.28^d)$ and $l_d \approx O(5^d)$ respectively. These results are consistent with the analysis in Section 4.7. The error

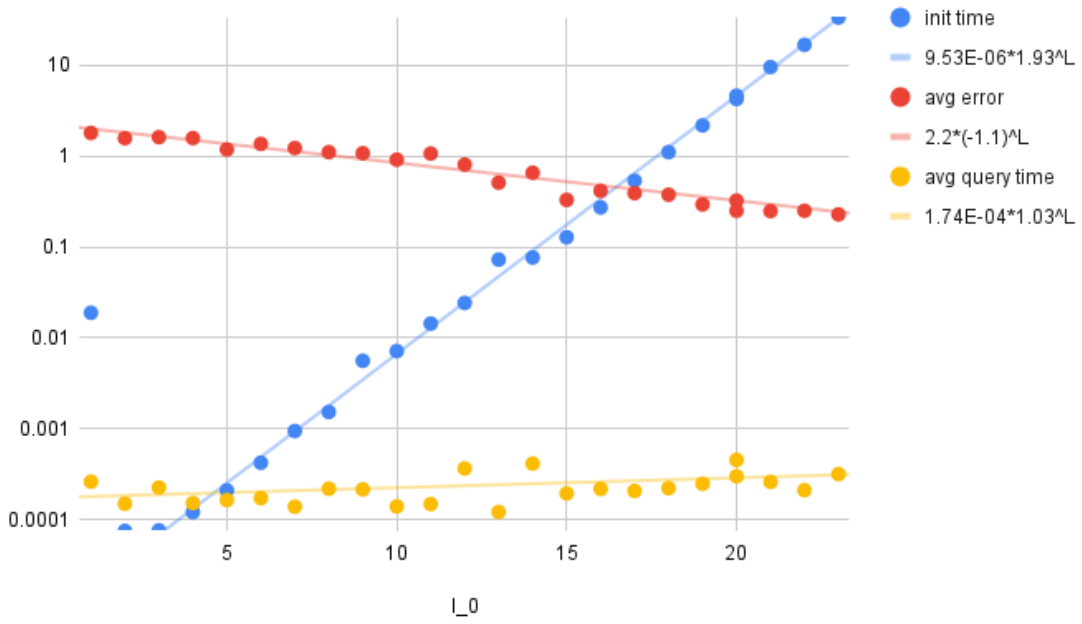

Figure 5.3.2: The log plot of average initialization time, query time, and error over the initial approximation length l_0 (in the plot this is L) of Basic Approximation. Although the initialization is expensive, since it only needs to be done once, we are rather interested in the complexity of each query and error in terms of l_0.

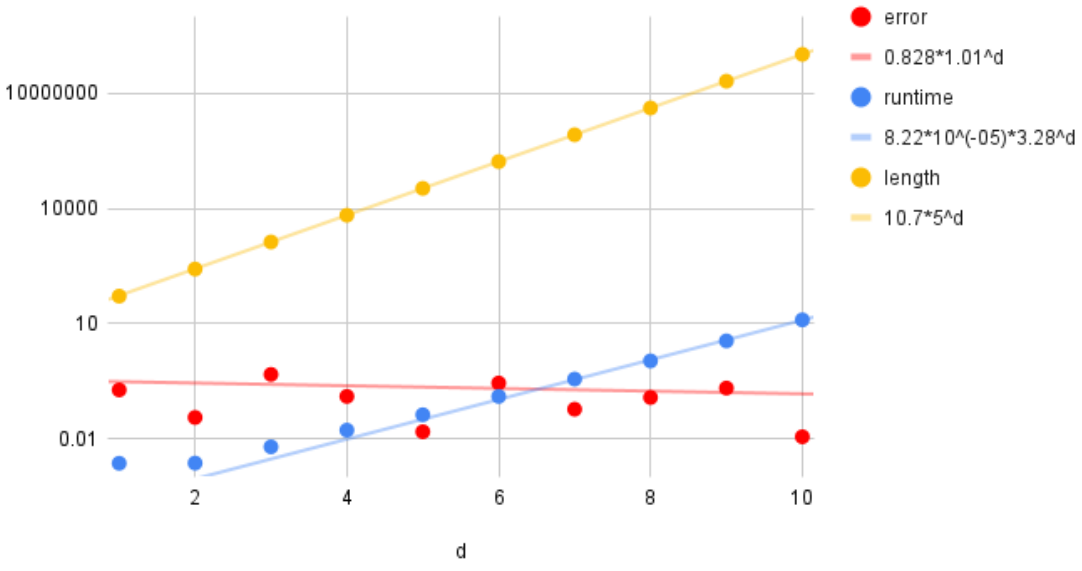

Figure 5.3.3: The log plot of average error ϵ_d, runtime t_d, and output compiled sequence length l_d over the recursive depth d of the Solovay-Kitaev algorithm.

also has a decreasing trend as desired. However, the experimental result is rather messy so we cannot be certain about the shape it takes. From analysis, we expect the error to be $C^{\frac{3}{2}^d}$, where C is some constant, meaning the *log* plot of the error over d should yield some exponential over d. It is hard to see this from the figure. Potentially, collecting data points for $d \geq 10$ (which is computationally expensive) can help us verify this curve.

The most compelling reason behind the error that fluctuates instead of going downwards with respect to the recursion depth is because we use the SolovayKitaevSynthesis interface from Qiskit transpiler as our algorithm to approximation single-qubit gates. SolovayKitaevSynthesis returns an approximation that can be off by a global phase factor. The global phase factor difference results in a large error, but once removed, leaves out a good approximation with a small error. We can see this in the `qiskitSK.ipynb` file. Combining UDSK with our implementation of the Solovay-Kitaev algorithm may be able to resolve the issue. Other possible reasons for the messiness in the error data points may stem from incorrect implementation or error propagation during matrix arithmetic, which can add up linearly or multiply exponentially to the length of the compiled sequence. In short, the messiness in the error data points requires further investigation.

5.3.3 UDSK Compiler

In the UDSK compiler, Universal Decomposition decomposes n-qubit unitary into $O(4^n)$ 2-level unitaries, each is then fed into the Solovay-Kitaev algorithm to be decomposed into sequences of hardware gates. Therefore, the complexities of the runtime and compiled sequence length of the UDSK compiler are,

$$T_{UDSK} = T_{UD}T_{SK} = O(4^n 3^d)$$

$$L_{UDSK} = L_{UD}L_{SK} = O(4^n 5^d).$$

The complexity of the error requires an intricate analysis but is not the focus of this thesis. A crude approximation can be obtained by the multiplication $\epsilon_{SK} \cdot 4^n$.

5.4 Environment

All the experiments are run on a Lenovo T490 laptop with the following specification:

- RAM: 16 Gb
- Architecture: x86_64
- Core(s): 4
- CPU(s): 8
- Model Intel(R) Core(TM) i5-8365U CPU @ 1.60GHz

Programs are implemented in Python language, with dependencies on the following packages:

- numpy (1.23.3) for matrix operations
- scipy.stats for generating random unitary matrix
- qiskit (0.42.0) for circuit diagram generation and SolovayKitaevSynthesis

The consistency between analysis and experiments confirms the correctness of our implementation. The small error also verifies that our UDSK compiler can approximate a general unitary to arbitrary precision. However, deviations from expectations and messy data points prompt further examination into the runtime for the Universal Decomposition and the error for the Solovay-Kitaev algorithm. Detailed analyses on these topics are good future directions.

6
Challenges towards Practical Quantum Compilation

Up to this point we have built a prototypical compiler that can compile any arbitrary quantum gate into hardware gates. However, for a fully functional quantum computer to read a program and execute its functions, a larger ecosystem of tool chains called the **software stack** is needed to convert application programs into hardware control signals. These toolchains transform programs into lower levels of abstraction and include many compilers. We will put our UDSK compiler into the context of the software stack, and touch on pieces other than UDSK that are required to execute a program on an actual quantum computer.

6.1 Software Stack

Translating a quantum program into hardware control signals for execution typically involve many steps to lower abstraction one layer at a time. Each layer of abstraction is notated by some intermediate representation (IR), for example, the quantum assembly language (QASM) [3]. These IRs are useful in exposing specific configurations while remaining hardware agnostic. The division of the layers of abstraction to designated functions is called the architecture of a quantum device, and the pieces of software that regulate the architecture is the software stack.

There are no standardized ways of dividing the layers of abstraction. However, it is typically broken down into five or six layers, each with a designated purpose, as is introduced in quantum programming and compilation literature [2, 11, 17]. We follow Jones [13] and break up the architecture into five layers: the Application (5), Logical (4), Quantum Error Correction (3), Virtual (2), and Physical (1) layer. There is a compiler in between each layer to convert to a less abstract layer one level at a time.

The Physical layer contains hardware gates and qubits that are noisy and have connectivity constraints. Since each hardware qubit is noisy, decoheres, and loses its state quickly, we use a large number of them to simulate a virtual qubit that is almost noiseless, has a longer lifetime, and conforms to our mathematical model. This forms the Virtual Layer. To reduce the overall aggregate error of the entire quantum computing system, error correction is performed at the Quantum Error Correction Layer to produce logical qubits that are seemingly noiseless and hardware-independent. Above the Logical Layer, a programmer can treat a quantum computer as a "clean" mathematical model. Typically a quantum program is written at the Logical Layer to create an application.

The UDSK compiler presented in Chapter 4 is a high-level compiler that takes the position between the Application and Logical layers only! There are lower-level compilers between all other layers that incorporate error correction and consider hardware constraints. In short, much work is to be done for a quantum program to run on actual hardware. We will survey recent compilation methodologies in the next section to demonstrate how UDSK can be extended to become more powerful and compile to layers beyond the Logical layer.

6.2 Overview of Quantum Compilation and Synthesis

The UDSK compiler that we built is prototypical, incomplete, and not scalable to a quantum computer with a large number of qubits. With the design challenges for NISQ and the broader context of the software stack introduced, we want to show more advanced techniques that create rather realistic solutions and provide a perspective into how this thesis can be extended. The

following will be a survey of recent quantum compilation literature. The field evolves quickly with the development of quantum devices so the survey is brief and in no way comprehensive. (Note that in literature, synthesis often refers to the lowest level of compilation that targets hardware gates. However, it is often also used interchangeably with compilation, depending on the context. Here we use the more general meaning of referring to compilation.)

Quantum compilation is built on the foundation of linear algebra decomposition methods in complex vector space. Conventional fixed compilation routines are derived from mathematical decompositions. For example, gate identities can be used as replacement rules that are applied iterating over gates in a circuit. General decomposition rules are also developed. Barenco et al. [1] proved that an arbitrary quantum circuit for a n qubit quantum computer can be expressed by compositions of a set of single-qubit and CNOT gates, resulting with a $O(n^2 4^n)$. This is the universality proof in Chapter 4.4 shown for the first time. Other more advanced fixed routines yield shorter compiled circuit lengths.

Particularly useful are decomposition methods stemming from the Cartan Decomposition from Lie group theory. One of them is the Cosine-Sine-Decomposition (CSD) [31]. An application instance of the CSD is Tucci [33] which uses CSD to construct a binary tree whose product is the approximation gate sequence. The other one is the KAK decomposition. Vatan [36] shows that KAK decomposition produces optimal compiled circuit depth for two-qubit operations with respect to the family of CNOT, y-rotations, z-rotations, and phase gates. Since KAK is optimal, advanced compilers often decompose the target matrix into two-qubit operations and then feed them into KAK as a final step. A good introduction to KAK can be found in Tucci [34], where the decomposition is proven constructively with only linear algebra, as opposed to the conventional derivation through Lie Group Theory and Lie Algebra. Aside from KAK, a great routine to reduce dimensions is the Quantum Shannon Decomposition (QSD) [29], which breaks an n-qubit unitary into four $(n-1)$-qubit unitaries and three multi-controlled rotations.

Fixed routine methods typically are not aware of hardware topology and suffer from long execution times for quantum computers with a larger number of qubits. Computational methods

such as search and heuristics are adopted to improve topology awareness and scalability to compile for NISQ and large quantum devices. QSearch [4] minimizes error-prone CNOT counts while accounting for connectivity of NISQ superconducting devices using an A-star-inspired algorithm. SABRE [16] is a SWAP-based BidiRectional heuristic search algorithm that solves the qubit mapping problem on NISQ devices with arbitrary connections between physical qubits.

More recently, machine learning are used for quantum compiling. Swaddle [32] uses two neural networks, trained on a generated data set, to decompose a three-qubit system. However, in their formulation, the size of the neural network scales 2^n to the number of qubits, so their solution is not scalable to a machine with a large number of qubits. Moro [21] proposes a deep reinforcement learning method that learns a general strategy to approximate single-qubit unitaries via a single precompilation procedure. By doing so, the overall execution time is reduced, potentially allowing real-time operations. Both Swaddle and Moro's methods provide better execution times and small errors but are not aware of hardware topology.

The most successful and complete compilers combine a multitude of search, optimization and transformation rules to meet the design criteria. QFAST [37] includes search and numerical optimization to perform topology-aware quantum synthesis, trading off optimality for increased compilation speed that scales up to seven qubits. Rakyta and Zimboras [26] approximate a general unitary through optimization over continuous variables and iterations of adaptive circuit compression, which is achieved by sequential removal of controlled two-qubit gates from the initial circuit. They obtained lower CNOT counts than QFAST [37], QSEARCH [4], and the Qiskit transpiler [12].

In conclusion, hardware topology, error correction, scalability and optimization all have to be considered to built a practical compiler. Topology awareness and optimization are especially important to utilize the tight hardware resources on NISQ devices. These are important features that will need to be added to our USDK compiler for it to create code executable on hardware.

Bibliography

[1] Adriano Barenco, Charles H. Bennett, Richard Cleve, David P. DiVincenzo, Norman Margolus, Peter Shor, Tycho Sleator, John A. Smolin, and Harald Weinfurter, *Elementary gates for quantum computation*, Physical Review A **52** (1995nov), no. 5, 3457–3467.

[2] Frederic T Chong, Diana Franklin, and Margaret Martonosi, *Programming languages and compiler design for realistic quantum hardware*, Nature **549** (September 2017), no. 7671, 180–187 (en).

[3] Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta, *Open quantum assembly language*, 2017.

[4] Marc G. Davis, Ethan Smith, Ana Tudor, Koushik Sen, Irfan Siddiqi, and Costin Iancu, *Towards optimal topology aware quantum circuit synthesis*, 2020 ieee international conference on quantum computing and engineering (qce), 2020, pp. 223–234.

[5] Christopher M. Dawson and Michael A. Nielsen, *The solovay-kitaev algorithm*, 2005.

[6] Artur Ekert, *Quantum interferometers as quantum computers*, Physica Scripta **1998** (1998jan), no. T76, 218.

[7] Dmytro Fedoriaka, *Decomposition of unitary matrix into quantum gates* (201906).

[8] F. Gray, *Pulse code communication*, Google Patents, 1953. US Patent 2,632,058.

[9] A. Harrow, *Quantum compiling*, Massachusetts Institute of Technology, Department of Physics, 2001.

[10] Jack D Hidary, *Quantum computing: An applied approach*, 1st ed., Springer Nature, Cham, Switzerland, 2019.

[11] Thomas Häner, Damian S Steiger, Krysta Svore, and Matthias Troyer, *A software methodology for compiling quantum programs*, Quantum Science and Technology **3** (2018feb), no. 2, 020501.

[12] IBM, *Qiskit: An open-source framework for quantum computing*, 2021.

[13] N. Cody Jones, Rodney Van Meter, Austin G. Fowler, Peter L. McMahon, Jungsang Kim, Thaddeus D. Ladd, and Yoshihisa Yamamoto, *Layered architecture for quantum computing*, Phys. Rev. X **2** (2012Jul), 031007.

[14] Phillip Kaye, Raymond Laflamme, and Michele Mosca, *An introduction to quantum computing*, Oxford University Press, London, England, 2006 (en).

[15] Chi-Kwong Li, Rebecca Roberts, and Xiaoyan Yin, *Decomposition of unitary matrices and quantum gates*, arXiv, 2012.

[16] Gushu Li, Yufei Ding, and Yuan Xie, *Tackling the qubit mapping problem for nisq-era quantum devices*, CoRR **abs/1809.02573** (2018), available at `1809.02573`.

[17] Marco Maronese, Lorenzo Moro, Lorenzo Rocutto, and Enrico Prati, *Quantum compiling*, 2021.

[18] Margaret Martonosi and Martin Roetteler, *Next steps in quantum computing: Computer science's role*, CoRR **abs/1903.10541** (2019), available at 1903.10541.

[19] Andy Matuschak and Michael Nielsen, 1970.

[20] ———, *Quantum mechanics distilled*, 1970.

[21] Lorenzo Moro, Matteo G. A. Paris, Marcello Restelli, and Enrico Prati, *Quantum compiling by deep reinforcement learning*, Communications Physics **4** (2021aug), no. 1.

[22] Michael A. Nielsen and Isaac L. Chuang, *Quantum computation and quantum information*, Cambridge University Press, 2000.

[23] Maris Ozols, *The solovay-kitaev theorem*, Essay at University of Waterloo (2009).

[24] IBM Qiskit, *Solovaykitaevsynthesis*.

[25] IBM Quantum, *Grover's algorithm*, 2023.

[26] Péter Rakyta and Zoltán Zimborás, *Efficient quantum gate decomposition via adaptive circuit compression*, 2022.

[27] Eleanor G Rieffel and Wolfgang H Polak, *Quantum computing*, Scientific and Engineering Computation, MIT Press, London, England, 2014.

[28] Vivek V. Shende and Igor L. Markov, *On the cnot-cost of toffoli gates* (2008).

[29] V.V. Shende, S.S. Bullock, and I.L. Markov, *Synthesis of quantum-logic circuits*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **25** (2006jun), no. 6, 1000–1010.

[30] Gilbert Strang, *Introduction to linear algebra*, Wellesley-Cambridge Press, 2003.

[31] Brian D. Sutton, *Computing the complete cs decomposition*, 2008.

[32] Michael Swaddle, Lyle Noakes, Harry Smallbone, Liam Salter, and Jingbo Wang, *Generating three-qubit quantum circuits with neural networks*, Physics Letters A **381** (2017oct), no. 39, 3391–3395.

[33] Robert R. Tucci, *A rudimentary quantum compiler(2cnd ed.)*, 1999.

[34] ———, *An introduction to cartan's kak decomposition for qc programmers*, arXiv, 2005.

[35] SciPy v1.10.1 Manual, *Kdtree*.

[36] Farrokh Vatan and Colin Williams, *Optimal quantum circuits for general two-qubit gates*, Physical Review A **69** (2004mar), no. 3.

[37] Ed Younis, Koushik Sen, Katherine Yelick, and Costin Iancu, *Qfast: Conflating search and numerical optimization for scalable quantum circuit synthesis*, arXiv, 2021.

[38] Kitaev A Yu, *Quantum computations: algorithms and error correction*, UMN, 1997.