A Parallel Tensor Network Contraction Algorithm and Its Applications in Quantum Computation

by

Fang Zhang

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2020

Fang Zhang
fangzh@umich.edu
ORCID iD: 0000-0002-0000-7101

To the entrancing boundary between Truth and trivia

ACKNOWLEDGMENTS

I would like to thank my co-advisor (formerly my advisor), Professor Yaoyun Shi, both for academic guidance throughout the Ph.D. program, and for giving me the opportunity to be a research intern at Alibaba Quantum Lab (AQL), since most of the work in this thesis was done either at AQL or in collaboration with AQL. I would also like to thank my other co-advisor, Professor John Hayes, who gave me a lot of helpful advice during the few months he has been my co-advisor.

I would like to thank my supervisor during my internship at AQL, Jianxin Chen, who has also been a personal friend of mine ever since we met. Dynamic slicing, the very idea underlying the algorithm described in this thesis, has emerged from one of our discussions near the beginning of my internship. I would like to thank my other colleagues at AQL who have contributed to the development of the idea, including Professor Yaoyun Shi (again), Cupjin Huang, Michael Newman, Professor Mario Szegedy, Xun Gao, and many others. I would also like to thank other colleagues in Alibaba Group who helped with our experiment on computer clusters.

I would like to thank my lab mates at U-M, including the aforementioned Cupjin Huang and Michael Newman, as well as Kevin Sung, for valuable discussions even before I started my internship at AQL. Special thanks to Kevin Sung for introducing me to the Open-Fermion project, where I implemented a simple functionality that familiarized me with `numpy.einsum`, which turned out to be fundamental for this thesis.

I would like to thank other people that have helped me in my study of quantum information, including during the Ph.D. program and during my undergraduate study at Tsinghua University. In particular, I would like to thank Professor Xiongfeng Ma, who was my undergraduate thesis advisor, and Professor Giulio Chiribella, who instructed my first formal quantum computation course. I would also like to thank my parents, who raised me up in an environment where I could get to know quantum computation even before going to university.

I would like to thank my committee members Professor John Hayes, Professor Yaoyun Shi, Professor Kai Sun, and Professor Christopher Peikert for their service.

I would like to thank NSF and Alibaba Group USA for funding my research.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Tensors are a natural generalization of matrices, and tensor networks are a natural generalization of matrix products. Despite the simple definition of tensor networks, they are versatile enough to represent many different kinds of "products" that arise in various theoretical and practical problems. In particular, the powerful computational model of quantum computation can be defined almost entirely in terms of matrix products and tensor products, both of which are special cases of tensor networks. As such, (classical) algorithms for evaluating tensor networks have profound importance in the study of quantum computation.

In this thesis, we design and implement a parallel algorithm for tensor network contraction. In addition to finding efficient contraction orders for a tensor network, we also dynamically slice it into multiple sub-tasks with lower space and time costs, in order to evaluate the tensor network in parallel. We refer to such an evaluation strategy as a contraction scheme for the tensor network. In addition, we introduce a local optimization procedure that improves the efficiency of the contraction schemes we find.

We also investigate the applications of our parallel tensor network contraction algorithm in quantum computation. The most ready application is the simulation of random quantum supremacy circuits, where we benchmark our algorithm to demonstrate its advantage over other similar tensor network based simulators. Other applications we found include evaluating the energy function of a Quantum Approximate Optimization Algorithm (QAOA), and simulating surface codes under a realistic error model with crosstalk.

CHAPTER 1

Introduction

1.1 Quantum computation, tensor networks, and parallel algorithms

The role of classical algorithms in quantum computation. Even though it is believed that $BPP \neq BQP$ — i.e. quantum computers will be able to provide superpolynomial speedup over classical computers on various problems, and in particular the problem of simulating quantum systems, which quantum computers themselves are based on — the truth value of this inequality, like many others in complexity theory, is an open problem [1]. In general, we know relatively little about the properties of BQP and other quantum-related complexity classes. As such, the study of classical algorithms for problems that arise in quantum computation always has theoretical significance.

More importantly, though, currently quantum computers are still only in the development phase. There has only been one reported case of "quantum supremacy", i.e. a quantum computer supposedly achieving a performance better than classical (exponential time) algorithms on a specific computational task of quantum circuit simulation [2], and a fairly controversial case at that [3, 4]. Therefore, at this point, classical algorithms for "quantum problems" have practical significance as well. For example, classical simulation of quantum computers may provide guidance to the design of both quantum software and quantum hardware [5], and both may help us realize the goal of useful quantum computation. At least in the short term, classical algorithms would keep playing a major role in the research of quantum computation.

Tensor networks in quantum computation. A tensor network is a versatile mathematical object that can represent many concepts in quantum computation. In its most general form, a tensor network is represented by a hypergraph, where each vertex represents a tensor, and each (hyper)edge a tensor index[1]. Each edge has a *dimension*; whenever not otherwise specified, we assume that all the edge dimensions are 2, meaning that each edge can be assigned two values, either 0 or 1. Each edge can also be *closed* or *open*. The value of a tensor network is a tensor, computed by multiplying the entries of all the tensors for each assignment of values to the edges, and then summing over all possible values of the closed edges. As a special case, if all the edges in a tensor network are closed, then the value of the tensor network is a scalar.

Quantum states (pure or mixed), measurements, unitary gates, and general channels can all be represented as tensors, and their combinations all follow the evaluation rules of appropriately constructed tensor networks. The advantage of representing quantum computation as tensor network is that a tensor network can be evaluated in many ways with varying space and time complexities, so a tensor network representation may reveal a better method to compute a value. Furthermore, representing many different concepts as tensors allows a unified treatment of them, making many evaluation strategies more generally applicable.

Parallelization in classical computation. The idea of parallel algorithms — breaking up a computational task into several sub-tasks that can be completed with little to no communication in-between — probably exists even before digital computers exist. While it can be difficult to increase the computational power of a single computing unit by increasing the frequency, improving the instruction set, etc., it is always relatively easy to put together many processor cores or machines for more computational power. The catch, however, is that parallelizing an algorithm usually requires some structure in said algorithm. Some algorithms are inherently serial because every step depends on the previous one. Some algorithms can only be parallelized on a low level, requiring frequent communication between sub-tasks, which makes such parallel algorithms suitable only for multi-core processors, and not for clusters of computers.

The most ideal cases of parallel algorithms are sometimes called "embarrassingly parallel algorithms", where the sub-tasks take approximately the same amount of time to complete, no communication is needed between the sub-tasks except for the initial input and the final results, and the process to aggregate the final results is simple and has negligible space and time costs. Such algorithms can be run even on clusters with no special infrastructure that allows efficient communication between nodes, which would be needed for some more communication-intensive distributed algorithms. Obviously, algorithms with such a structure is highly desirable in the context of high performance computing.

[1]Note that this is different from the notation in [6], where edges are tensors and vertices are indices.

Overview of the thesis. This thesis describes a parallel tensor network contraction algorithm designed and implemented by the author during the graduate program, together with some variants and extensions of which the author is also involved in design and implementation. It also summarizes existing and potential applications of said algorithm in quantum computation. The thesis can thus be divided into two parts.

The first part of the thesis introduces various aspects of the parallel tensor network contraction algorithm. We give a definition of tensor networks that generalizes some previous definitions, and then define (simple) tensor network contraction, contraction orders, and metrics that characterize "good" contraction orders. We observe that tensor networks can be *sliced* to give multiple easier-to-evaluate tensor networks, which naturally gives rise to a parallel tensor network contraction algorithm. We call the combination of a set of edges to slice and a contraction order of the resulting tensor network a *contraction scheme*; the most difficult part of our algorithm is to find a feasible contraction scheme with as low a time cost as possible (where both algorithmic and hardware considerations may affect the time cost). To this end, we employ and combine various optimization algorithms. Despite our best efforts, and even though the contraction schemes we found already surpass previous strategies by a lot, we believe that there may still exist better contraction schemes, even for the tensor networks for which we spent a lot of time on finding them.

The second part of the thesis concerns applications of tensor network contraction in quantum computation. The most obvious and direct application is "single-amplitude simulation" of a quantum circuit, i.e. computing specific amplitudes of the output state vector of the quantum circuit. We also identify other potential applications to the quantum approximate optimization algorithm (QAOA), and to quantum error correction.

1.2 Overview of results

1.2.1 A parallel tensor network contraction algorithm

In Chapter 3, we introduce the parallel tensor network contraction algorithm which is central to this thesis. *Tensor network contraction* is a common way to evaluate a tensor network, which works by repeatedly eliminating closed edges (i.e. indices that are summed over) and combining adjacent nodes (i.e. tensors) until only a single node with open edges is left. Different *contraction orders* result in different ways to evaluate the same tensor network, usually with different time complexities and space complexities. Finding the optimal contraction order for a tensor network is itself a hard problem, but there are heuristic methods to tackle this problem [7]. However, the naive implementation of tensor network contraction is very difficult to parallelize, because the complicated data dependency between the tensors would usually require a lot of inter-processor communication. This can be a bottleneck when trying to contract larger tensor networks with more classical computational resources.

The basic idea we deal with this problem, introduced in [6], is by *slicing* the tensor network, i.e. splitting it into multiple simpler tensor networks that *sum* to the original tensor network. By slicing k edges, the original tensor network will be split into 2^k tensor networks with identical structure, each with lower space and time requirements to contract than the original. We define a *contraction scheme* as the combination of a list of edges to slice, and a contraction order for the post-slicing tensor network structure. Our main goal, therefore, is to find a *feasible* contraction scheme (i.e. one with a space cost low enough to fit in the available memory) with as low a time cost as possible.

Actually finding such a good contraction scheme is still a hard problem that can only be solved with heuristic methods. The current version of our contraction scheme finding procedure consists of hypergraph partitioning based *initial contraction order finding*, greedy *local optimization*, and greedy *dynamic slicing*. Chapter 4 will show that this procedure indeed finds good contraction schemes in practice, and thus give evidence that efficient parallel tensor network contraction is indeed possible under this framework.

1.2.2 Classical simulation of quantum supremacy circuits

In Chapter 4, we report the results of applying our parallel tensor network contraction algorithm to the task of simulating "quantum supremacy circuits", a class of quantum circuits that are designed to be hard to simulate classically [5, 8, 2]. This is the most direct application of our algorithm, and indeed the one that inspired this work in the first place. The importance of this problem mainly lies in that there have been several previous works that attempt to solve the same task with similar methods, so it works well as a benchmark of our algorithm.

Our experiment results show that our algorithm performs better than all comparable classical algorithms in terms of running time. For the largest circuits considered in [2], we achieve a $2000\times$ speedup over our predecessor, Cotengra [9]. For smaller circuits, our algorithm is able to do the "supremacy task" faster than the quantum device in [2], potentially challenging the validity of their quantum supremacy claim.

In particular, we note that the methods used in both [8] and [9] fall under our paradigm of "contraction schemes", which was first introduced (though not explicitly defined nor named) in [6]. The significant speedup we achieved in [10, 4] compared to those works shows the advantage of dynamic slicing ([10] vs. [8]) and local optimization ([4] vs. [9]).

1.2.3 Applications to the Quantum Approximate Optimization Algorithm

In Chapter 5, we investigate applications of our algorithm to the Quantum Approximate Optimization Algorithm (QAOA) [11], one of the most promising potential applications of near-term quantum computation devices. QAOA is based on quantum adiabatic algorithms, but instead of simulating changing the Hamiltonian slowly, which would require a circuit with exponentially many layers of small rotations, it uses a circuit with a reasonable number p of layers and not necessarily small rotations, with the rotation angles being parameters of the algorithm. By optimizing over those parameters, nontrivial results can be obtained even with a small value of p.

The usually proposed method for optimizing the parameters is with a feedback process, where a quantum processor tries the algorithm with different parameters and a classical algorithm optimizes for the best result. However, due to the inherent randomness in quantum computation, the classical part would have to be a *stochastic* optimizer, which is usually less efficient than optimizers for deterministic functions.

In [12], we directly compute the *expected value* of the objective function using tensor networks. By only considering the *lightcones* of each individual clause, we can efficiently calculate the expected value for some problems and values of p. We find that our implementation of this algorithm performs better than existing software packages that solve the same problem. In the case of small-cycle-free graphs, the speed of our implementation allows doing the classical parameter optimization completely classically.

In addition to computing the expectation value, we also attempt to sample from the final state with tensor networks. However, even though we are able to simulate measuring each qubit individually, we find it infeasible to sample from the joint distribution. Therefore, when quantum devices become available, it may make sense to still implement QAOA as a hybrid quantum-classical algorithm, by optimizing over the parameters completely classically, then using those parameters on a real quantum device to do the actual sampling.

1.2.4 Applications to quantum error correction

In Chapter 6, we use tensor network contraction to study quantum error correction codes. Quantum error correction codes is an essential component for fault-tolerant quantum computation, which is necessary for universal scalable quantum computation in the near future. Hence, the study of the properties of quantum error correction codes has both theoretical and practical importance. Existing theoretical analyses are usually based on simplistic error models such as the Pauli twirling approximation, which can only give crude approximations of the logical error rate. Realistic error models should give more precise estimations of the logical error rate, which may provide guidance on quantum hardware design.

In [13], we study a realistic error model based on the one presented in [14], with an important additional component, *crosstalk* induced by ZZ-interactions that are present between neighboring qubits even when they are idle. As our results show, crosstalk can significantly affect the logical error rate of quantum error correction codes, potentially necessitating measures to specifically mitigate it.

We focus on the quantum error correction code Surface-17, which involves 9 data qubits and 8 ancilla qubits. We simulate 3 rounds of error syndrome extraction, the minimum number of rounds needed for fault tolerant error correction. Therefore, the total number of syndrome bits measured is $8 \times 3 = 24$, and we can use a tensor that fits in the memory to represent the logical error for each of the 2^{24} results. This allows us to figure out the optimal decoder and exactly compute the logical error rate, without resorting to Monte Carlo sampling like is done in [14].

1.3 Dissertation outline

This dissertation is divided into seven chapters. Chapter 2 introduces some basic concepts of quantum information, as well as a general definition of tensor networks. This is supposed to be a minimal set of definitions and theorems to help understand this thesis. A more comprehensive introduction of quantum information can be found in [1].

Chapter 3 introduces the concept of contraction schemes, and describes our parallel tensor network contraction algorithm, which at its core is just a method to find a good contraction scheme. Chapter 4 reports the benchmark results of our algorithm on “quantum supremacy circuits”. Chapter 5 and Chapter 6 investigate applications of our algorithm to QAOA and to quantum error correction, respectively.

Finally, in Chapter 7 we summarize these results, and briefly discuss potential future work that can be done on this topic.

1.3.1 Works appearing

The work in Chapter 3 and Chapter 4 has been developed in a series of papers [6, 10], culminating in [4], which is being submitted for publication. The author of this thesis proposed dynamic slicing in a discussion with Jianxin Chen, and wrote the code for the original algorithm used in [6], which was later refactored by Cupjin Huang. The author of this thesis was also involved in profiling the program, doing benchmark experiments on clusters, discussions regarding the improvements to the original algorithm, and writing the paper.

The work in Chapter 5 is drawn from [12]. The author of this thesis was mainly involved in discussions regarding the design of the experiments, and the interpretation of the results.

The work in Chapter 6 appears in [13]. The author of this thesis wrote the code to generate the quantum circuit for a general surface code, and implemented the realistic error model used in [14].

Other works that the author has contributed to during the Ph.D. program, but do not fit into the theme of this thesis, can be found in [15, 16, 17, 18].

CHAPTER 2

Preliminaries

In this chapter, we review some basic concepts of quantum information and computation. These definitions and theorems will be used throughout the thesis. For a more comprehensive introduction on those topics, we refer to [1]. We will assume that the readers are already familiar with the basics of linear algebra and graph theory.

At the end of this chapter, we will also review the concept of tensors, and give a general definition of tensor networks. We note that our definition of tensor networks coincides with the so-called "Einstein summation convention" evaluated by the `einsum` function in NumPy, as well as the `opt_einsum` package [19]. In fact, this tensor network contraction algorithm can be regarded as a specialized implementation of `einsum`.

2.1 Asymptotic notations

We use the following notations to describe the *growth rates* of functions.

Definition 2.1 (Asymptotic notations)**.** The notations $f(n) = O(g(n))$, $f(n) = \Omega(g(n))$, and $f(n) = \Theta(g(n))$ are defined as follows:

- $f(n) = O(g(n))$ if and only if there exists n_0 and $c > 0$ such that $f(n) \leq c \cdot g(n))$ for all $n \geq n_0$. This means that f grows *at most as fast as* g.
- $f(n) = \Omega(g(n))$ if and only if there exists n_0 and $c > 0$ such that $f(n) \geq c \cdot g(n))$ for all $n \geq n_0$. This means that f grows *at least as fast as* g. (An alternative definition is that $f(n) = \Omega(g(n)) \iff g(n) = O(f(n))$.)
- $f(n) = \Theta(g(n))$ if and only if both $f = O(g(n))$ and $f = \Omega(g(n))$ hold. This means that f grows *at the same rate as* g.

The big Theta notation defines an equivalence relation, and the big O notation defines a partial order of the equivalence classes. In unusual cases, the growth rates of two functions may not be comparable, i.e., neither $f = O(g(n))$ nor $f = \Omega(g(n))$ holds.

Those asymptotic notations are commonly used to describe space and time complexities of algorithms, because even though the detailed implementation of an algorithm may affect the exact amount of memory and numbers of instructions it takes, for reasonable implementations, the difference should be no more than a constant factor, i.e., within the same big Theta equivalence class. Therefore, asymptotic notations allows us to focus on the "big picture" behavior of an algorithm as the input size grows, without worrying about implementation details.

2.2 The circuit model of quantum computation

In this thesis, all instances of quantum computation, except adiabatic quantum computing which is described in Section 5.2.2, are all based on the circuit model. In this section, we will review the basics of quantum computation in terms of the circuit model.

2.2.1 Quantum states

The basic unit of quantum information is a *qubit*, just like the basic unit of classical information is a bit. Unlike in the classical case, however, the state of a quantum system containing multiple qubits cannot always be described by a description of the state of each individual qubit. Therefore, we will first give ways to describe the state of a quantum system of any size (although we restrict ourselves to quantum systems of finite dimension), then discuss how quantum systems combine with each other.

Definition 2.2 (Quantum system of finite dimension and pure state)**.** A quantum system A of *dimension* d can be in any *pure state* $|\psi\rangle$, which is mathematically formalized as a d-dimensional vector with complex entries $(\alpha_0, \dots, \alpha_{d-1})^T$, such that $\sum_{j=0}^{d-1} |\alpha_j|^2 = 1$.[1]

To emphasize the nature of $|\psi\rangle$ as a vector, we may call $|\psi\rangle$ a *state vector*. The complex numbers $\alpha_0, \dots, \alpha_{d-1}$ are called the *amplitudes* of the state vector $|\psi\rangle$.

One small "catch" is that the states represented by $|\psi\rangle$ and $e^{i\theta}|\psi\rangle$ are considered the same, even though $|\psi\rangle$ and $e^{i\theta}|\psi\rangle$ are different vectors. The reason is that, as will be seen in Section 2.2.2, the only way to extract information from quantum states is by measurement, and the reader can verify that none of the measurements and other operations described below gives a way to distinguish between $|\psi\rangle$ and $e^{i\theta}|\psi\rangle$. Such a scalar prefactor like the $e^{i\theta}$ in $e^{i\theta}|\psi\rangle$ is called a *global phase*.

[1]We avoid using i as an index whenever complex numbers may be involved, due to the potential confusion with the imaginary unit $i = \sqrt{-1}$.

A *qubit* is a quantum system of dimension $d = 2$, and we give special names to the two "basis states" below:

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \qquad |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

The state $|\psi\rangle = (\alpha, \beta)^T$ can therefore also be written as $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$. This can be easily generalized to cases where $d > 2$. This kind of notation would be convenient for sparse quantum states, as well as in many other cases.

Combination of quantum systems. Let A and B be two quantum systems of dimension d_A and d_B, respectively, Then their combination AB is a quantum system with dimension $d_{AB} = d_A d_B$. If the system A and B are in the states $|\psi_A\rangle$ and $|\psi_B\rangle$ respectively, then the state of the system AB is given by taking the *tensor product* of the two states, $|\psi_A\rangle \otimes |\psi_B\rangle$. When there is no danger of confusion, this product is also simply written as $|\psi_A \psi_B\rangle$. For example, $|00\rangle = |0\rangle \otimes |0\rangle = (1, 0, 0, 0)^T$.

A state of the system AB that can be written as the tensor product of a states of A and a state of B is called a *product state*. As alluded to in the beginning of this section, there exist pure states of AB that are not product states. A typical example is the *Bell state*[2] $|\Phi^+\rangle = (|00\rangle + |11\rangle)/\sqrt{2}$. Such states are called *entangled states*.

The reader may wonder what state the system A is in when the system AB is in an entangled state. The answer is that A is in a *mixed state*, which can be regarded as a probabilistic mixture of pure states. An understanding of mixed states is not really necessary in order to understand quantum computation in the circuit model, so we will defer talking about mixed states until Section 2.2.4, in order to focus on more essential components of the circuit model for now.

2.2.2 Computational basis measurements

One of the basic postulates of quantum information is that the state of a quantum system is not directly observable from the outside. Instead, in order to extract information from a quantum system, one must use *measurements*, which would not only generate a classical result, but also affect the quantum state of the system. There is no way to extract information from a quantum system without disturbing its state.

The simplest kind of measurement is known as a *computational basis measurement*. Suppose a system A of dimension d is in the state $|\psi\rangle = \sum_{j=0}^{d-1} \alpha_j |j\rangle$. Then a computational basis measurement on A will give the classical result j with probability $|\alpha_j|^2$, and

[2]There are actually four distinct states of the system composed of two qubits with similar properties that are collectively called the *Bell states*, and $|\Phi^+\rangle$ is one of them.

after measurement, the quantum state of A will also become $|j\rangle$. (Since this state does not contain any information not contained in the classical result j, there is an alternative definition of computational basis measurement where the system A vanishes completely.)

As an aside, there is a way to represent the amplitudes α_j of a state $|\psi\rangle$ without explicitly writing down the decomposition $|\psi\rangle = \sum_{j=0}^{d-1} \alpha_j |j\rangle$. For this, we need to make use of a new notation:

Definition 2.3 ("Bra" notation)**.** The notation $\langle\psi|$ is defined as the conjugate transpose of $|\psi\rangle$. (Remember that $|\psi\rangle$ represents a column vector; thus $\langle\psi|$ represents a row vector.)

The notation $\langle\varphi|\psi\rangle$ represents the inner product of $\langle\varphi|$ and $|\psi\rangle$, with one vertical bar omitted by convention. Below we list some properties of this new notation:

- For any valid quantum state $|\psi\rangle$, we have $\langle\psi|\psi\rangle = 1$.
- For $0 \leq j, k < d$, we have $\langle j|k\rangle = 0$ if $j \neq k$, and $\langle j|k\rangle = 1$ otherwise.
- If $|\psi\rangle = \sum_{k=0}^{d-1} \alpha_k |k\rangle$, then $\langle j|\psi\rangle = \sum_{k=0}^{d-1} \alpha_k \langle j|k\rangle = \alpha_j$.

Therefore, the probability of measuring the state $|\psi\rangle$ and getting the result j can be represented as $|\langle j|\psi\rangle|^2 = \langle j|\psi\rangle\langle\psi|j\rangle$.

In the case where A is a composite system, it is conventional to write the "basis state" $|j\rangle$ as a product state $|j_1 j_2 \ldots j_k\rangle$, and refer to the classical result j in the same way. For example, when A is the system composed of two qubits, then a computational basis measurement has four possible outcomes: 00, 01, 10, and 11. In general, measuring a system composed of n qubits will give a bit string of length n.

In fact, it is possible to measure only a part of a composite system according to the following rule. Without loss of generality, assume that we measure the system A in the composite system AB. We can decompose the state $|\psi_{AB}\rangle$ of the system AB as:

$$|\psi_{AB}\rangle = \sum_{j=0}^{d-1} \alpha_j (|j\rangle \otimes |\psi_{B,j}\rangle), \qquad \langle\psi_{B,j}|\psi_{B,j}\rangle = 1.$$

(A state vector $|\psi\rangle$ such that $\langle\psi|\psi\rangle = 1$ is also known as a *normalized* state vector.) The probability of getting the classical result j is then again $|\alpha_j|^2$, and the quantum state of AB becomes $|j\rangle \otimes |\psi_{B,j}\rangle$.

One intuitive property of the above definition is that in a system composed of qubits, measuring the whole system to get a bit string is equivalent to measuring every qubit in any order, then putting the result bits together in an appropriate order.

Finally, we note that here we have only described computational basis measurements. There are several more complicated kinds of measurements; however, in the circuit model, they can all be represented as a combination of computational basis measurements and other allowed operations. Therefore, they are not essential to the circuit model, and are omitted here.

2.2.3 Unitary operations

One final essential component of quantum computation is *unitary operations*. Recall that a *unitary matrix* is a matrix U such that $U^\dagger U = I$, where $U^\dagger$ is the conjugate transpose of U.

Definition 2.4 (Unitary operation)**.** A *unitary operation* that applies to a quantum system of dimension d is represented as a $d \times d$ unitary matrix U. If the initial state of the system is $|\psi\rangle$, then after applying the unitary operation U, the state becomes $|\psi'\rangle = U|\psi\rangle$.

Note that the conjugate transpose of $|\psi'\rangle$ is $\langle\psi'| = \langle\psi|U^\dagger$, therefore

$$\langle\psi'|\psi'\rangle = \langle\psi|U^\dagger U|\psi\rangle = \langle\psi|\psi\rangle = 1.$$

Thus $|\psi'\rangle$ is also a valid normalized quantum state.

The sequential composition of unitary operations follows the simple rule $U_2(U_1|\psi\rangle) = (U_2U_1)|\psi\rangle$, i.e. first applying U_1 then applying U_2 to the same system is equivalent to applying a single unitary operation, U_2U_1.

Similar to measurements, unitary operations can also be applied to only a part of a composite system. Without loss of generality, assume that we apply a unitary operation U to system A in the composite system AB. Then the rule for updating the state of AB is

$$|\psi'_{AB}\rangle = (U \otimes I_{d_B})|\psi'_{AB}\rangle.$$

In other words, applying U to the system A is equivalent to applying $U \otimes I_{d_B}$ (which is easily seen to be a $d_{AB} \times d_{AB}$ unitary matrix) to the system AB.

An important observation is that

$$(U_A \otimes I_{d_B})(I_{d_A} \otimes U_B) = U_A \otimes U_B = (I_{d_A} \otimes U_B)(U_A \otimes I_{d_B}).$$

Therefore, when we apply unitary operations to distinct components of a composite system, it does not matter which one we apply first. This is consistent with our physical intuition of local operations.

Circuit model. The essence of the circuit model is to build complicated unitary operations on many qubits with simple unitary operations that usually only apply on a few qubits. Such a simple unitary operation used as a building block is also known as a *unitary gate*, or a *gate* for short.

Definition 2.5 (Unitary circuit)**.** A *unitary circuit* on n qubits is an ordered sequence of unitary gates, each applying on a subset of those n qubits. The circuit itself represents a unitary operation on n qubits given by sequentially composing all its gates together.

By the observation above, when two adjacent gates in a circuit apply to disjoint sets of qubits, then those gates can be exchanged without affecting the value of the circuit. Therefore, sometimes we divide a circuit into "layers", or *cycles*, of gates that all apply to disjoint sets of qubits, and only care about the order of the cycles. The number of cycles is sometimes also called the *depth* of the circuit.

Sometimes we also consider quantum circuits in a more general sense, which are allowed to contain:

- Initial states given for some or all of the qubits.
- Computational basis measurements on some or all of the qubits.

Over the course of a circuit, each qubit will have a number of gates applied to it, and conceptually each gate change the "state" of the qubit. Even though in a circuit, the state of all qubits should be considered as a whole, and talking about the "state" of a single qubit may not be meaningful, we will say that each qubit is divided into $m + 1$ *wires* by the m gates applying to it. We define an *input wire* as a wire corresponding to the initial state of a qubit that is not given in the circuit, and an *output wire* corresponding to the final state of a qubit that is not measured in the circuit. The rest of the wires are called *internal wires*.

Below, we list some commonly used gates appearing in this thesis:

$$H = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}, \qquad T = \begin{pmatrix}1 & 0\\ 0 & e^{i\pi/4}\end{pmatrix},$$

$$\sqrt{X} = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & -i\\ -i & 1\end{pmatrix}, \qquad \sqrt{Y} = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & -1\\ 1 & 1\end{pmatrix}, \qquad \sqrt{W} = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & -\sqrt{i}\\ \sqrt{-i} & 1\end{pmatrix},$$

$$CZ = \begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & -1\end{pmatrix}, \qquad CNOT = CX = \begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\\ 0 & 0 & 1 & 0\end{pmatrix},$$

Here the names H, CZ, CNOT, and CX are shorthands for Hadamard, controlled-Z, controlled-NOT, and controlled-X, respectively.

Convention. It is customary to write the names of gates as though they are English words, i.e. to typeset them in roman. In this thesis, we will typeset the names of gates in italic only when we want to explicitly refer to the underlying unitary matrix. Therefore, we will say "a U gate" when we talk about a gate in a circuit, but "$U = \cdots$" when we give its explicit matrix representation.

Commutativity of gates. As we mentioned above, two gates applied in succession on different components (i.e. qubits) of a composite system can switch places without affecting the final result. Sometimes, this property holds for two specific gates even though they are applied to the same qubit or qubits. We say that those two gates *commute*.

In general, two unitary operations U_1 and U_2 on the same system commute with each other if $U_1U_2 = U_2U_1$. Sometimes the commutativity of two gates may depend on *how* the set of qubits they apply to overlap. For example, a CNOT gate commutes with a T gate applied to the first qubit, but not with a T gate applied to the second qubit.

Universality. We will note that the circuit model is a *universal* model of quantum computation, in the sense that there exist finite sets of gates, called *universal gate sets*, such that any computation physically allowed by quantum mechanics can be approximated with gates from any such gate set to an arbitrary precision. One example of such a gate set is $\{\mathrm{CNOT}, \mathrm{H}, \mathrm{T}\}$; there are many others.

2.2.4 Mixed states and density matrices

In some cases, we would want to describe the state of a system A when it is entangled with another system B (i.e. AB is in an entangled state). In other cases, it would be convenient to represent a quantum system that we only have probabilistic information of (i.e. we only know a probability distribution of the (pure) state of the system) with a single mathematical object. It would turn out that those two situations are one and the same, and the mathematical object we need is the *density matrix*.

Definition 2.6 (Density matrix and probabilistic mixture of quantum states)**.** The state of a quantum system can also be represented by its *density matrix*. If a system of dimension d is in the pure state $|\psi\rangle$, then its density matrix is a $d \times d$ matrix given by $\rho = |\psi\rangle\langle\psi|$.

A density matrix can also represent a *probabilistic mixture* of quantum states. If a system has probability p_j to be in the state ρ_j, $\sum_j p_j = 1$, then the overall density matrix of the system is given by $\sum_j p_j \rho_j$.

A quantum state represented by a density matrix ρ that is not a pure state, i.e. that cannot be written in the form $\rho = |\psi\rangle\langle\psi|$, is called a *mixed state*.

All physical operations that can be applied to a state vector can also be applied to a density matrix:

- If we do a computational basis measurement on a density matrix ρ, then the probability of getting the result j is given by the diagonal element $\langle j|\rho|j\rangle$ of ρ.
  - This implies that the diagonal elements of a density matrix are always non-negative, and they also sum to 1, i.e. $\mathrm{Tr}(\rho) = 1$.
  - More generally, if a composite system AB is in the state ρ_{AB}, then the effect of measuring the system A can be found out as follows: First write ρ in the block matrix form
    $$\rho_{AB} = \sum_{j=1}^{d_A} \sum_{k=1}^{d_A} |j\rangle\langle k| \otimes \rho_{jk}.$$
    The probability of getting the result j is then $\mathrm{Tr}(\rho_{jj})$, and after measurement, the state of the system B becomes $\rho_{jj} / \mathrm{Tr}(\rho_{jj})$.
- Applying a unitary U to a density matrix ρ results in $\rho' = U\rho U^\dagger$.

Now, we can also answer the question of what the state of the system B is when the system AB is in a general state that can be entangled (or even mixed). We again write ρ_{AB} in the block matrix form above, and measure the system A, but *immediately forget the result*. The state of B is now a probabilistic mixture[3]:

$$\rho_B = \sum_{j=1}^{n} \mathrm{Tr}(\rho_{jj}) \frac{\rho_{jj}}{\mathrm{Tr}(\rho_{jj})} = \sum_{j=1}^{n} \rho_{jj} = \mathrm{Tr}_A(\rho_{AB}),$$

where the notation Tr_A denotes *partial trace over the system A*. This "measure and forget" procedure can be viewed as discarding the system A, or simply setting the system A aside and doing nothing with it, since physically there should be no way to tell the difference. Therefore, we will just say that $\rho_B = \mathrm{Tr}_A(\rho_{AB})$ is the state of the system B.

[3]The first summation below is only over values of j where $\mathrm{Tr}(\rho_{jj}) \neq 0$; for the second summation it does not matter, because it can be shown that if $\mathrm{Tr}(\rho_{jj}) = 0$, then $\rho_{jj} = 0$.

One final note is that apparently different probabilistic mixtures of pure states may give the same mixed state. For example, for any single-qubit unitary U, the equal probability mixture of $U|0\rangle$ and $U|1\rangle$ is the same as the equal probability mixture of $|0\rangle$ and $|1\rangle$:

$$\frac{1}{2}(U|0\rangle)(\langle 0|U^\dagger) + \frac{1}{2}(U|1\rangle)(\langle 1|U^\dagger) = \frac{1}{2}UIU^\dagger = \frac{I}{2} = \frac{1}{2}|0\rangle\langle 0| + \frac{1}{2}|1\rangle\langle 1|.$$

Operationally, the two mixtures are completely equivalent, so it makes sense to say that they are really the same state.

2.3 Tensors and tensor networks

Tensors are a natural generalization of vectors and matrices. From a computational viewpoint, both vectors and matrices are arrays of numbers indexed by some integer indices — one index for a vector, and two indices for a matrix. The definition of tensors simply generalizes the number of indices to any non-negative integer.

Definition 2.7 (Tensor [7]). An order-k *tensor* $g = [g_{i_1,i_2,\ldots,i_k}]_{i_1,i_2,\ldots,i_k}$ is an array of numbers $g_{i_1,i_2,\ldots,i_k}$, indexed by k indices $i_1, i_2, \ldots, i_k$, which take $m_1, m_2, \ldots, m_k$ values respectively. The number m_t is called the *dimension* of the index i_t.

As an example, scalars, vectors, and matrices can all be regarded as special cases of tensors, with order 0, 1 and 2 respectively. For an $m \times n$ matrix $A = [A_{ij}]_{ij}$, the dimensions of its two indices i and j would be m and n, respectively.

In order to concisely denote a sequence of indices, we may use the notation $\vec{i}$. The range of such a "multi-index" is denoted $\Pi(S)$, where S is a set of abstract indices. For example, in the matrix example above, $\Pi(\{i,j\}) = \{1,2,\ldots,m\} \times \{1,2,\ldots,n\}$.

A tensor network defines a "product" for a collection of tensors, as a generalization of matrix products, etc.

Definition 2.8 (Tensor network). A *tensor network* is defined by a collection of input tensors T, together with a list of output indices S_{out}. Any number of indices in the input tensors and/or output indices can be written as the same symbol, to signify that they are regarded as the same index. (An index must have the same dimension in every tensor it appears in.) The value of a tensor network is a tensor g, and can be evaluated as follows:

$$g_{\vec{i}} = \sum_{\vec{j} \in \Pi(S \setminus S_{\text{out}})} \prod_{t \in T} t_{\vec{i}\vec{j}}, \qquad \forall \vec{i} \in \Pi(S_{\text{out}}).$$

Here S is the set of all indices appearing in the input tensors and/or output indices, and $\overrightarrow{ij} \in \Pi(S)$ is a *concatenation* that contains all indices in S, but $t_{\overrightarrow{ij}}$ should be understood as only taking those indices needed by the tensor t.

This evaluation rule may be easier to understood in the form of a naive algorithm:

1. For every possible combination of all indices appearing in the input tensors and/or output indices, take the *product* of the numbers in each input tensor indexed by those indices.

2. *Sum* over all indices not appearing in the output indices.

As a concrete example, a matrix product $C = AB$ can be represented as a tensor network $C_{ik} = A_{ij}B_{jk}$, which is evaluated as

$$C_{ik} = \sum_j A_{ij}B_{jk}.$$

In the tensor network notation, the summation over the index not appearing in the output indices, j, is implicit.

Most previous works [7] represented a tensor network as a graph, where vertices are tensors and edges are indices. This creates an implicit limitation: Since each edge in a graph can only be associated with two vertices, this would mean each index can only appear twice in the summation. In some practical cases in quantum computation, this may give rise to tensor network representations less concise than desirable.

For example, consider the matrix product $C = ADB$, where D is known to be a diagonal matrix. This kind of expression can easily arise in quantum computation, e.g. when simulating a circuit with diagonal gates. Rather than representing D as an order-2 tensor with off-diagonal elements equal to 0, and write

$$C_{ik} = A_{ij}D_{jj'}B_{j'k},$$

it is more desirable to represent D as an order-1 tensor, and write

$$C_{ik} = A_{ij}D_jB_{jk}.$$

Our definition provides enough flexibility to allow for the second more simple representation.

In order to represent such a general tensor network, it is necessary to use a *hypergraph* instead of a graph.

$$T_{be} = \sum_{a,c,d} A_{ac} B_{abd} C_{cde} D_{bc}$$

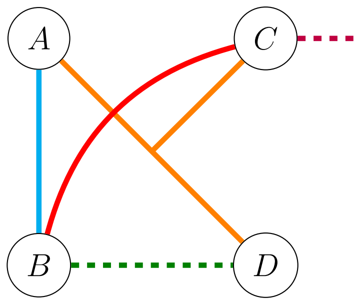


Figure 2.1: Example of a tensor network and its hypergraph representation. Solid lines and dashed lines represent closed and open edges, respectively.

Definition 2.9 (Hypergraph representation of tensor network)**.** The *hypergraph representation* of a tensor network is a hypergraph $G = (V, E)$, where each vertex $v \in V$ corresponds to a tensor in the tensor network, and each (hyper)edge $e \in E$ corresponds to an index. An edge e is associated with a vertex v if and only if the index represented by e appears in the tensor represented by v. Edges corresponding to output indices are called *open edges*; the other edges are called *closed edges*.

A tensor network with no open edge is called a *closed tensor network* for short. By definition, a closed tensor network evaluates to an order-0 tensor, i.e. a scalar.

Figure 2.1 illustrates the correspondence between a simple example tensor network, $T_{be} = A_{ac}B_{abd}C_{cde}D_{bc}$, and its hypergraph representation. Notice that in the hypergraph, the yellow hyperedge (corresponding to the index c) is associated with three vertices, and the purple hyperedge (corresponding to the index e) is only associated with one.

Convention. In the context of quantum computation, it is a common occurrence that the dimension of each edge is 2 (representing the two states of a qubit). Therefore, for the rest of this proposal, unless otherwise mentioned, we assume the dimension of each edge (i.e. each index) is 2.

CHAPTER 3

A parallel tensor network contraction algorithm

In this chapter, we describe a parallel tensor network contraction algorithm that allows efficient evaluation of tensor networks on a computer cluster.

Tensor network contraction is a common way to evaluate a tensor network, which works by repeatedly eliminating closed edges (i.e. indices that are summed over) and combining adjacent nodes (i.e. tensors) until only a single node with open edges is left. Different *contraction orders* result in different ways to evaluate the same tensor network, usually with different time complexities and space complexities. Finding the optimal contraction order for a tensor network is itself a hard problem, but there are heuristic methods to tackle this problem [7]. However, the naive implementation of tensor network contraction is very difficult to parallelize, because the complicated data dependency between the tensors would usually require a lot of inter-processor communication. This can be a bottleneck when trying to contract larger tensor networks with more classical computational resources.

The basic idea we deal with this problem, introduced in [6], is by *slicing* the tensor network, i.e. splitting it into multiple simpler tensor networks that *sum* to the original tensor network. By slicing k edges, the original tensor network will be split into 2^k tensor networks with identical structure, each with lower space and time requirements to contract than the original. We define a *contraction scheme* as the combination of a list of edges to slice, and a contraction order for the post-slicing tensor network structure. Our main goal, therefore, is to find a *feasible* contraction scheme (i.e. one with a space cost low enough to fit in the available memory) with as low a time cost as possible.

Actually finding such a good contraction scheme is still a hard problem that can only be solved with heuristic methods. The current version of our contraction scheme finding procedure consists of hypergraph partitioning based *initial contraction order finding*, greedy *local optimization*, and greedy *dynamic slicing*. Chapter 4 will show that this procedure indeed finds good contraction schemes in practice, and thus give evidence that efficient parallel tensor network contraction is indeed possible under this framework.

3.1 Introduction

3.1.1 Problem and motivation

Simulation of quantum systems is a problem that arose naturally with the discovery of quantum mechanics, and the very idea of quantum computation was born from the observation that solving this problem classically seems hard. However, quantum computers are still very much in the development phase: Only recently have there been plausible claims of programmable quantum computers with a scaling advantage over known classical algorithms on specific problems [2, 20], and it is safe to say that fault-tolerant quantum computers would still require years, if not decades, to develop. Therefore, classical simulators for quantum systems would remain useful at least in the near future, and many approaches to them have been proposed for different kinds of quantum systems.

One promising and mathematically interesting approach is simulating quantum systems with tensor networks, which were first proposed as a model for certain physical systems. A method to model general quantum circuits as tensor networks is given in [7]. However, actually evaluating those tensor networks is still a hard problem.

For the particular task of general quantum circuit simulation, most state-of-the-art simulators make use of some massively parallel computational infrastructure, whether it is a computer cluster [8] or a supercomputer [21]. This is not a coincidence: As far as we know, the computational cost of simulating a general quantum circuit necessarily grows exponentially in the size of the circuit, and parallel computation is the most realistic way to have so much classical computational power.

However, parallelizing quantum circuit simulation is not a trivial task. Many previous proposals suffer from a huge communication cost, sometimes necessitating using a supercomputer instead of a much more affordable computer cluster. In this work, we try to solve the problem of parallelization in a much less communication intensive way.

3.1.2 Tensor network contraction schemes

In Definition 2.8 (tensor networks), a naive algorithm for evaluating a tensor network is given. However, the time complexity of the naive algorithm is often too high to be practical: Since the definition requires iterating over every possible combination of all indices, the time complexity is $\Theta(2^m n)$, where m is the number of edges in the tensor network and n is the number of vertices. In most cases, a lower time complexity can be achieved by evaluating the tensor network using a *contraction algorithm*.

In the (hyper)graph representation of tensor networks, tensor network contraction works by repeatedly "contracting" multiple vertices (i.e. tensors) into a single vertex, until only one vertex is left. Computationally, each contraction step itself consists of evaluating a simpler tensor network. The contraction algorithm is usually faster than the naive algorithm because the numbers of edges and vertices involved in each contraction step are usually significantly smaller than those of the entire tensor network. Detailed definitions of tensor network contraction can be found in Section 3.2.1.

One important point about tensor network contraction is that the *contraction order* (also known as the *contraction path*), i.e. which vertices to contract in a given contraction step, is not unique, and indeed different contraction orders may give rise to very different space and time complexities, which can be measured with the *contraction width* and the *contraction cost* of the contraction order, respectively. Usually, we want to find a contraction order with the lowest cost, with an upper bound on contraction width given by the space constraint. This is a hard optimization problem, and the best we can do is to use heuristic algorithms to find a relatively good order.

For very complex tensor networks, even the best known contraction order still gives contraction width and cost that are far out of reach for single classical processors. As mentioned at the beginning of this chapter, normally tensor network contraction is difficult to parallelize, and thus we *slice* the tensor network in order to divide the contraction task into independent sub-tasks.

Definition 3.1 (Slicing tensor networks; Definition 3.9)**.** In a tensor network G, any edge e with dimension m can be *sliced* to create m tensor networks with the same structure (i.e. the same hypergraph), each simpler than G. The hypergraph of the resulting tensor networks is attained by simply removing the edge e. In the ith resulting tensor network, each vertex v that was associated with e is replaced by v_i; the tensor on v_i is a slice of the tensor on v, by fixing the value of the index represented by the edge e to i.

It is straightforward to slice multiple edges in the original tensor network with dimensions $m_1, \ldots, m_k$, in order to get $\prod_i m_i$ tensor networks, each with those k edges removed. As long as each sub-task fits within the available memory for a single processor, they can be evaluated either in parallel or sequentially depending on the resources available. The value of the original tensor network can then be calculated by summing and/or concatenating the results of those sub-tasks (depending on whether the edges sliced are closed edges or open edges). We call such an evaluation strategy a *contraction scheme* for the tensor network.

Definition 3.2 (Contraction scheme; Definition 3.10)**.** A *contraction scheme* for a tensor network G is defined as a set of edges to slice $S = \{e_1, \ldots, e_k\}$, together with a contraction order p for the tensor network after slicing, $G - S$.

3.1.3 Outline of the contraction scheme finding procedure

The current version of our contraction scheme finding procedure [4] consists of the following components:

- **Initial contraction order finding**, which is currently done with a heuristic algorithm based on hypergraph partitioning, an idea first introduced in [9]. Compared with the implementation in [9], we have identified a smaller set of hyper-parameters that affects the outcome of the heuristic procedure the most, and optimize over those parameters with CMA-ES [22], a more effective hyper-parameter tuner for this problem.

- **Local optimization**, which is done by first finding the *stem* of a given contraction order, then randomly taking a short segment of the stem and finding the optimal contraction order of the corresponding tensor network, which has a small number of nodes but a high contraction cost. We repeat this procedure for a reasonable number of times to efficiently reduce the cost of a given contraction order.

- **Dynamic slicing**, which is a simple but effective greedy algorithm that, given a contraction order, finds one edge to slice that would reduce the contraction cost the most, repeating until the resulting contraction scheme is feasible. In the current version, we also do a round of local optimization after slicing each edge, which keeps the contraction order highly optimized and makes the dynamic slicing procedure more efficient overall.

In addition, we found it necessary in practice to account for peculiarities of the underlying hardware implementation of a single contraction step, especially when trying to leverage the computational power of GPUs. In some cases, it is necessary to make small modification to the final contraction order, sacrificing some "theoretical" time cost to make the best use of GPU kernels, ultimately reducing the real running time of the contraction.

This chapter will focus on describing our algorithm itself, with only a little discussion about its performance on concrete tensor networks that arise in practice. Chapter 4 will give some benchmark results of our algorithm on such tensor networks.

3.2 Preliminaries

3.2.1 Tensor network contraction

To give a general definition of tensor network contraction, we first define sub-networks of tensor networks.

Definition 3.3 (Sub-network)**.** A tensor network $G' = (V', E')$ is a *sub-network* of another tensor network $G = (V, E)$ if the following conditions are satisfied:

- $V' \subseteq V$.
- $E' = \{e \cap V' \mid e \in E\} \setminus \{\emptyset\}$ (i.e. G' contains all edges of G associated with at least one vertex in V', *restricted to* V').
- All closed edges of G' are closed edges of G that falls inside V' completely.

In order to succinctly describe commonly used sub-networks, we also define the concept of induced sub-networks.

Definition 3.4 (Induced sub-network)**.** A sub-network of $G = (V, E)$ *induced* by a set of vertices $V' \subseteq V$ is the sub-network that contains exactly those vertices in V', and as many closed edges as possible. In other words, all closed edges of G that falls inside V' completely are closed edges of the induced sub-network.

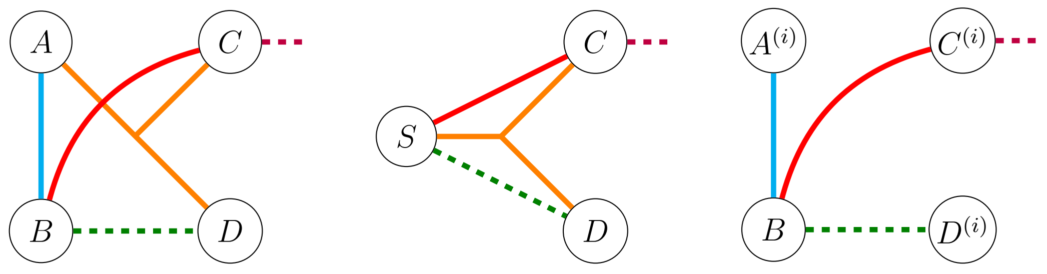
A sub-network of $G = (V, E)$ *induced* by a closed edge $e \in E$ is the smallest sub-network that contains e as the only closed edge. In other words, the induced sub-network contains exactly all vertices associated with e (and all other edges associated with those vertices as open edges).

Now we can define tensor network contraction steps, and prove that they preserve the value of the tensor network.

Definition 3.5 (Contraction step)**.** In a tensor network G, any sub-network H with n vertices can be *contracted*, to create a new tensor network G' with $n - 1$ fewer vertices than G. To contract a sub-network H, simply replace it with a single tensor (i.e. a single vertex) that results from evaluating H. All closed edges of H are removed.

A simple example of tensor network contraction is given in Figure 3.1 (b). We will use the notation $G \xrightarrow{H} G'$ to represent single contraction steps.

Theorem 3.1. *If $G \xrightarrow{H} G'$, then G and G' have the same value.*



$$T_{be} = A_{ac}B_{abd}C_{cde}D_{bc}$$

(a)

$$T_{be} = S_{bcd}C_{cde}D_{bc}$$
$$S_{bcd} = A_{ac}B_{abd}$$

(b)

$$T^{(i)}_{be} = A^{(i)}_{a}B_{abd}C^{(i)}_{de}D^{(i)}_{b}$$
$$T = T^{(0)} + T^{(1)}$$

(c)

Figure 3.1: Examples of tensor network contraction and tensor network slicing. (a) The same example tensor network as in Figure 2.1. (b) Contracting the sub-network induced by the vertex set $\{A, B\}$. The blue edge (i.e. the index a) is eliminated from the main tensor network, as the implicit sum on a is now in the expression for S, rather than in the expression for T. (c) Slicing the yellow edge (i.e. the index c). The structure of the tensor network is greatly simplified, but the hypergraph actually represents two tensor networks $T^{(0)}$ and $T^{(1)}$ that both need to be evaluated.

Proof. Let $G = (V, E)$ and $H = (V_H, E_H)$. Denote the values of G, G' and H as g, g' and h respectively, and noticing that closed edges of H cannot be associated with vertices in $V \setminus V_H$, we have:

$$\begin{aligned}
g_{\vec{i}} &= \sum_{\vec{j}\in\Pi(E_{\text{closed}})} \prod_{v\in V} v_{\vec{i}\vec{j}} \\
&= \sum_{\vec{j}\in\Pi(E_{\text{closed}})} \left(\prod_{v\in V\setminus V'} v_{\vec{i}\vec{j}}\right)\left(\prod_{v\in V'} v_{\vec{i}\vec{j}}\right) \\
&= \sum_{\vec{j}\in\Pi(E_{\text{closed}}\setminus E_{H,\text{closed}})} \left(\prod_{v\in V\setminus V'} v_{\vec{i}\vec{j}}\right)\left(\sum_{\vec{j}'\in\Pi(E_{H,\text{closed}})} \prod_{v\in V_H} v_{\vec{i}\vec{j}\vec{j}'}\right) \\
&= \sum_{\vec{j}\in\Pi(E_{\text{closed}}\setminus E_{H,\text{closed}})} \left(\prod_{v\in V\setminus V'} v_{\vec{i}\vec{j}}\right) h_{\vec{i}\vec{j}} \\
&= g'_{\vec{i}}.
\end{aligned}$$

□

A complete tensor network contraction procedure consists of iteratively contracting relatively small sub-networks until there is only one tensor left. The space and time complexities of a contracting algorithm are generally much lower than those of the naive algorithm, and they depend very much on the *contraction order* (also known as the *contraction path*), i.e. sequence of sub-networks contracted.

Definition 3.6 (Contraction order)**.** A *contraction order* for a tensor network G is a sequence of tensor networks $p = (H_1, \dots, H_t)$, such that

$$G = G_0 \xrightarrow{H_1} G_1 \xrightarrow{H_2} G_2 \xrightarrow{H_3} \cdots \xrightarrow{H_t} G_t$$

is a valid sequence of contraction steps, and G_t only consists of a single tensor.

We have made our definition as general as possible, in order to cover multiple different definitions of contraction used in previous works. For example, the contraction in [6] can be regarded as *edge contraction*, where every H_i (except for the final one) is a sub-network of G_{i-1} induced by an edge.

In this work, we will focus on *pairwise contraction orders*, where each H_i is a sub-network of G_{i-1} induced by a set of exactly two vertices.

Contraction cost. In [7], the *treewidth* of a tensor network is used to give an upper bound on the time complexity of tensor network contraction, $T^{O(1)} \exp[O(d)]$, where d is the treewidth and T is the size of the tensor network. Indeed, in our model, for pure contraction algorithms, and for an appropriate definition of d, $O(2^d)$ is a lower bound for the time complexity, so the upper bound $T^{O(1)}O(2^d)$ is tight up to a polynomial prefactor. However, we find that, in useful practical cases, 2^d is usually already large, so the polynomial prefactor actually matters significantly. Furthermore, using d only does not give a fine-grained enough metric to differentiate between contraction orders with similar, but still noticeably different, contraction costs. Therefore, we define a *contraction cost* that allows better estimation of the time complexity of a particular contraction order.

Definition 3.7 (Contraction cost)**.** The *contraction cost* c of a given contraction order $p = (H_1, \dots, H_t)$ for a tensor network G is defined as

$$c = \sum_{i=1}^{t} 2^{d_i}, \qquad \text{where } d_i = |E_{H_i}|.$$

(Here we make use of the assumption that the dimension of each edge is 2. Otherwise, 2^{d_i} needs to be replaced by the product of the dimensions of all edges in H_i.)

The contraction cost reflects a rough estimate of the time complexity of a general contraction order p, by using the naive algorithm for each step. In particular, when p is a pairwise contraction order, then the estimate is tight up to a constant factor.

We note that the cost function used in [6], which only adds up the sizes of the *output tensor* from each step, has exactly half the value of the contraction cost as defined above. The reason is that, since [6] uses edge contraction, each H_i will have exactly one closed edge (with dimension 2), and the size of the output tensor is the product of the dimensions of all open edges. Even though [6] did not use the naive algorithm for each step (in fact, each step was done with the library function `numpy.einsum`, which itself uses a contraction algorithm on the sub-network), this ensures that the contraction cost is a lower bound on the time complexity of the edge contraction procedure.

Contraction width. Apart from lower bounding the time cost of each contraction step, the size of each output tensor also lower bounds the *space cost* of the contraction step, which is also important in determining the feasibility of a contraction order. In fact, we will estimate the space cost of a contraction order using the *contraction width*, defined as follows:

Definition 3.8 (Contraction width)**.** The *contraction width* w of a given contraction order $p = (H_1, \ldots, H_t)$ for a tensor network G is defined as

$$c = \max_{i=1}^{t} d'_i, \qquad \text{where } d'_i = |E_{H_i,\text{open}}|.$$

(Again, we make use of the assumption that the dimension of each edge is 2. Otherwise, the dimensions would also need to be taken into account; a reasonable approach would be to let d'_i be the sum of the base-2 logarithm of the open edges in H_i.)

Obviously, $\Theta(2^w)$ is a lower bound for the space complexity of the contraction procedure. A more precise estimate could be computed by considering all the tensors at the end of each contraction step that would be needed later. However, in practice, we find it cumbersome and largely unnecessary to consider such details, because for most reasonable contraction orders, the space complexity is dominated by $\Theta(2^w)$. Furthermore, since whenever w increases by 1, 2^w doubles, practically w alone determines the feasibility of a contraction order.

Finally we note that, although sometimes there is a trade-off between the contraction cost c and the contraction width w, in general, a contraction order with a low cost will also have a low width, and vice versa. This is because both values are primarily determined by the sizes of the largest sub-networks involved in the contraction procedure, so methods that help decrease one of them usually would also help decrease the other. Therefore, we may simply refer to a contraction order with a low cost or width as a "good" contraction order, without specifying whether it is optimized with respect to cost, width, or a combination thereof. This is also why the dynamic slicing algorithm described in Section 3.5 can efficiently decrease the contraction width, even though it only considers the contraction cost.

3.2.2 Parallel contraction schemes

A tensor network contraction algorithm is usually difficult to parallelize without incurring a large communication cost. The reasons are:

- Each contraction step will usually depend on most, if not all, of the results of the contraction steps before it, so parallelization on the level of contraction steps is unlikely to be helpful.
- Within a contraction step, the computation may be easily parallelized by having each processor compute a different part of the output tensor. However, subsequent steps are likely to require each processor to have the full data of that tensor, or have the data sliced in a different way. Therefore, a large communication cost would be required.

However, we note that the *naive* tensor network evaluation algorithm is easy to parallelize, either by slicing the output tensor as mentioned in the second point above, or, if the output tensor is small (e.g. when the output is a scalar, it cannot be sliced itself), by having each processor compute and sum up a part of the summands in the definition, and finally taking the sum of the outputs of each processor. Since the output tensor is small, the second approach would only need a small amount of communication.

Of course, those approaches may not necessarily show an advantage over (non-parallel) contraction algorithms, since the time complexity of the naive algorithm may be huge. However, we further observe that, if the task is sliced "along one or more indices", then each resulting sub-task can still be described as a tensor network evaluation task, which can then be solved by contraction. If the time complexity of each sub-task is significantly lower than the original task, then we would have successfully accelerated the evaluation of the original tensor network by parallelization; and if the space complexity is significantly lower, then the parallel algorithm may be feasible even though the original contraction task is not feasible at all due to memory limits.

This idea for parallelization can be easily described in the hypergraph model as "slicing" (i.e. removing) edges. A simple example is given in Figure 3.1 (c).

Definition 3.9 (Slicing tensor networks)**.** In a tensor network G, any edge e with dimension m can be *sliced* to create m tensor networks with the same structure (i.e. the same hypergraph), each simpler than G. The hypergraph of the resulting tensor networks is attained by simply removing the edge e. In the ith resulting tensor network, each vertex v that was associated with e is replaced by v_i; the tensor on v_i is a slice of the tensor on v, by fixing the value of the index represented by the edge e to i.

As mentioned above, a tensor network can be evaluated by evaluating all the tensor networks resulting from a slice, then either concatenating the results (if the slice is of an open edge) or adding the results (if it is of a closed edge). Since m is usually a small number, we usually need to slice the tensor network multiple times, which corresponds to removing multiple edges from the hypergraph. In fact, by our convention, $m = 2$ for each sliced edge, so k slices are needed to divide the original contraction task into 2^k sub-tasks.

We will call such a tensor network evaluation strategy — slicing it into multiple sub-tasks, then using contraction for each sub-task — a *contraction scheme*.

Definition 3.10 (Contraction scheme)**.** A *contraction scheme* for a tensor network G is defined as a set of edges to slice $S = \{e_1, \ldots, e_k\}$, together with a contraction order p for the tensor network after slicing, $G - S$.

The contraction cost of a contraction scheme is defined as $2^k c$, where c is the contraction cost of the final contraction order p. The contraction width of a contraction scheme is the same as that of the contraction order p.

In the above definition, we define the contraction cost of a contraction scheme as the *total* cost of all sub-tasks, which can then be evenly distributed among all available processors. This allows dividing the task into more sub-tasks than available processors, which is frequently necessary because in many cases the space complexity is the bottleneck. The contraction width of a contraction scheme simply quantifies the amount of memory needed for a single sub-task, i.e. on a single processor.

3.2.3 Tree decomposition and treewidth

The *treewidth* of a graph is a number that characterizes how far the graph is from a tree. Many algorithms in graph theory that has exponential time complexity in general — including tensor network contraction — becomes polynomial time if the input graph has low treewidth. It is closely related to the contraction cost and the contraction width defined in the previous sections (See Section 3.3.1 for details), and a previous version of our contraction order finding procedure makes use of it. Therefore, we will give the definition of the treewidth in this section.

The treewidth is usually defined in terms of *tree decompositions* of a graph.

Definition 3.11 (Tree decomposition)**.** A *tree decomposition* of a graph $G = (V, E)$ is a tree T where each vertex is a subset of V (called a "bag"), satisfying the below properties:

- For each vertex $v \in V$, the bags that contain v form a nonempty connected subtree of T.
- For each edge $e = (u, v) \in E$, there exists a bag that contains both u and v.

The *width* of a tree decomposition is the size of the largest bag minus one.

Definition 3.12 (Treewidth)**.** The *treewidth* of a graph G is the minimum width among all tree decompositions of G.

We note that:

- Every graph G has at least one valid tree decomposition. For example, the trivial tree decomposition where a single bag contains all vertices of G is always valid. This means that the treewidth of $G = (V, E)$ is at most $|V| - 1$.
- The treewidth of a graph G with at least one edge is at least 1 (i.e. minimum of largest bag size is 2), and this is achieved when G is a forest.
- More generally, a graph containing a clique of size k has treewidth at least $k - 1$. In fact, the clique must be entirely contained in at least one bag, as proved below.

Theorem 3.2. *If a graph G contains a clique $C = \{v_1, \ldots, v_k\}$ of size k, then in any tree decomposition of G there must be a bag containing all of $v_1, \ldots, v_k$.*

Proof. The proof is by induction on k. The base cases $k = 1$ and $k = 2$ are true by the requirements that the bags that contain v_1 form a *nonempty* connected subtree and that the edge (v_1, v_2) must appear in a single bag, respectively.

Suppose $k \geq 3$. By induction hypothesis, there must exist bags X_1, X_2, X_3 such that

$$C \setminus \{v_1\} \subseteq X_1, \qquad C \setminus \{v_2\} \subseteq X_2, \qquad C \setminus \{v_3\} \subseteq X_3.$$

Therefore, each vertex in C is in at least two of X_1, X_2, X_3. By the properties of the tree decomposition, if $v \in X_i \cap X_j$, then v must also be in all the bags on the path between X_i and X_j. Since the tree decomposition is a tree, the paths between X_1 and X_2, between X_1 and X_3, and between X_2 and X_3 must intersect at at least one bag, X. Thus, X must contain all vertices in C. □

3.3 Initial contraction order finding

Even though when evaluating a tensor network with a parallel contraction scheme we do not need a contraction order for the original tensor network, a good initial contraction order may still be helpful when trying to find a good contraction scheme. In particular, the dynamic slicing procedure described in Section 3.5 requires a good initial contraction order to begin with. Therefore, our contraction scheme finding procedure begins with initial contraction order finding.

One of the problems with tensor network contraction is that it may be hard to find a contraction order with low enough cost. Indeed, it has been shown that a closely related quantity, the treewidth of a graph, is NP-hard to compute [23], and we see no reason that finding the minimum (pairwise) contraction cost or contraction width of a hypergraph would be any easier. Therefore, the best we could hope for is to have a heuristic algorithm that finds a contraction order that is "good enough", similar to what people usually use in the case of treewidth [24, 25].

3.3.1 Initial contraction orders based on treewidth algorithms

In fact, many heuristic treewidth algorithms, such as those in [24, 25], are directly applicable to contraction order finding. These algorithms all find an explicit upper bound of the treewidth of a graph by producing an elimination order or tree decomposition. When they are given the *line graph* of a tensor network, the elimination order or tree decomposition would allow constructing an edge contraction order with contraction width equal to the upper bound of treewidth found for the line graph. In general, the better (i.e. lower) the upper bound is, the better the contraction order found this way will be.

The line graph (also known as the "edge-to-vertex dual") of a hypergraph is defined as follows.

Definition 3.13 (Line graph)**.** The *line graph* of a hypergraph $G = (V, E)$ is an (ordinary) graph $G' = (V', E')$, where

$$V' = E, \qquad E' = \{(e_1, e_2) \in E \times E \mid e_1 \cap e_2 \neq \emptyset\}.$$

This definition gives a transformation on a hypergraph that transforms edges into vertices, which is a natural idea because in tensor network contraction *edges* are eliminated (and vertices are combined), while the treewidth is related to *vertex* elimination orders, as defined below.

Definition 3.14 (Vertex elimination orders; adapted from Definition 2.3 from [24] (perfect elimination ordering))**.** A *vertex elimination order* for a graph G is simply an ordering of all vertices in G. It describes an iterative process where in each round one vertex v in G is removed, and the neighborhood of v replaced by a clique (i.e. edges are added between every pair of neighbors of v). The *final degree* of a vertex v is the degree of v when it is eliminated in this process.

The *width* of a vertex elimination order is the maximum final degree among all vertices.

Theorem 3.3. *For any graph G, if G has a vertex elimination order with width w, then G also has a tree decomposition with width at most w, and vice versa.*

Proof. The proof in both directions is by induction on the number of vertices. Both directions are trivially true for the null graph.

Suppose that a vertex elimination order of G with width w begins with the vertex v, and the maximum final degree among all vertices other than v is w'. Then by induction hypothesis, there exists a tree decomposition of the graph G', which is the graph that arises after eliminating v from G, with width w'. Now, since $N(v)$, the neighborhood of v, forms a clique in G', there exists a bag X' in said tree decomposition such that $N(v) \subseteq X'$. Create a new bag $X = \{v\} \cup N(v)$, and attach it to X' in the tree decomposition. The result is a tree decomposition of G with width $\max(|X| - 1, w') = \max(|N(v)|, w') = w$.

Conversely, consider a tree decomposition of G with width w, and take any leaf X from the tree. Let X' be the only neighbor of X, if any; otherwise (when the tree decomposition only consists of one bag) let $X = \emptyset$. There are two cases:

- $X \subseteq X'$. In this case, removing the bag X from the tree decomposition still results in a valid tree decomposition of G, and we can repeat this process.
- There exists a v such that $v \in X$ and $v \notin X'$. Since the set of bags v is in must be connected, it follows that X is the only bag that contains v, and thus $\{v\} \cup N(v) \subseteq X$. Therefore, the degree of v is at most w, and after eliminating v from G, simply deleting v from X gives a valid tree decomposition for the new graph.

By induction hypothesis, after eliminating one vertex, we can find a vertex elimination order for the rest of the graph with width at most w. Therefore, this process gives a vertex elimination order for G with width at most w. ☐

It is straightforward to see the relation between edge contraction orders of a closed tensor network and vertex elimination orders of its line graph. Contracting an edge e in the tensor network creates a new vertex associated with all edges that share a vertex with e, which is equivalent to eliminating the corresponding vertex in the line graph and changing its neighborhood into a clique. The contraction width trivially corresponds with the width of the vertex elimination order.

The algorithms in [24, 25] outputs a vertex elimination order and a tree decomposition respectively. Since all the proofs above are constructive, it is easy to transform those into contraction orders for closed tensor networks. In fact, this approach also works for open tensor networks with a simple modification, by regarding the designation of open edges as an extra vertex in the hypergraph, i.e. connecting each pair of vertices in the line graph corresponding to open edges. We omit the detailed derivation, which is similar to the proofs above.

Limitations of initial contraction orders based on treewidth algorithms. We used initial contraction orders based on treewidth algorithms in [6, 10], with fairly good results. However, in addition to being limited by the performance of treewidth algorithms, this approach has some inherent problems.

- First, it optimizes the contraction width w, but not the contraction cost c. Even though in general (especially for edge contraction orders) those metrics agree with each other, there is the danger of ending up with a "flat" contraction order with more contractions involving large tensors than necessary.
    - Empirically, the treewidth algorithms we used are actually surprisingly good in this regard, producing solutions with the ratio $c/2^w$ significantly lower than "regular" orders, such as the vertical order used in [5]. We conjecture the reason is that the min-fill heuristic [24] used in those implementations for the initial solution already tends to give solutions with low contraction costs. However, in this regard, they still compare unfavorably with a specialized contraction cost optimizer like the one in [9].
- Second, it produces an edge contraction order, for which the contraction cost is only a rough estimate of the running time. Furthermore, there are examples of tensor networks where edge contraction cannot give the best contraction width (whereas pairwise contraction can), meaning that this approach is not necessarily good in terms of the space complexity either.

3.3.2 Initial contraction orders based on hypergraph partitioning

The shortcomings mentioned above motivate researchers to find heuristic optimization algorithms specifically designed for contraction orders that does not depend on treewidth algorithms. One such algorithm based on hypergraph partitioning is introduced in [9].

The algorithm in [9] is essentially a greedy algorithm, where the idea is to first (approximately) optimize the cost of the last contraction step or steps, then consider how to get to the tensors involved in those last steps. Notice that those tensors must come from contracting disjoint sets of tensors (i.e. vertices) in the original tensor network, so in fact we can use the same idea on each of those tensors. This gives rise to a top-down divide-and-conquer strategy that we can keep using all the way down to the input tensors — or in practice, until the tensor network to contract is so small that its cost is almost negligible compared to the original tensor network, at which point we can just use a simple bottom-up greedy strategy to get a reasonable contraction order.

Of course, if we allow all potential "last steps", then the ones with the lowest cost are usually the ones where a small tensor is contracted with the rest of the network, from which most of the closed edges are already eliminated. That would mean many potential choices would be tied for the lowest cost, and a greedy strategy would not guide our choice. Moreover, that would also go against the intuition of a top-down strategy, since we would start with the easy contractions and leave the difficult problems for later.

As shown in [9], a heuristic way to solve this problem is by adding a *balance constraint*. More precisely, at each step, when we partition the set V of all vertices of the tensor network into k partitions $V_1, \ldots, V_k$, we set a parameter $\varepsilon \in [0, k-1]$, and ask that

$$|V_i| \leq \left\lceil (1+\varepsilon)\frac{|V|}{k} \right\rceil, \qquad i = 1, \ldots, k.$$

Not only does this force a top-down structure, the resulting problem for each step also coincides with the existing problem of hypergraph partitioning [26, 27]. Of course, the latter problem is still a hard problem, but again heuristic algorithms are given in [26, 27] that gives reasonably good solutions for typical hypergraphs. Those algorithms are exactly what the contraction order finding procedure in [9] is based on.

There are still a few details that need to be filled in for the above approach. First, when the partitioning algorithm divides the hypergraph into $k > 2$ partitions, we need to find a way to contract those k tensors together in order to get a pairwise contraction order. Since k is chosen to be relatively small, this task can be delegated to `opt_einsum` [19], with either an exhaustive search with the help of dynamic programming when k is really small, or a simpler greedy approach when k is larger.

Second, in a “real-life” tensor network, there are usually a lot of small tensors (e.g. diagonal matrices) that can be easily “absorbed” into adjacent tensors without increasing their tensor order or complicating the tensor network in any other way. Such small tensors make the hypergraph partitioning procedure slower, and also may skew the balance constraint. Therefore, we follow [9] in *simplifying* the tensor network beforehand, by contracting such tensors with their neighbors before beginning the partitioning procedure.

Third, and most importantly, the values of the parameters k and ε need to be determined for each partitioning step. This is where our initial contraction order finding procedure diverges from [9] the most. In the implementation of [9], the following rules are used to determine the values of k and ε at each step:

- The number of parts k is initially k_0, and decays in a complex way according to the size of the tensor network currently considered, the size of the original tensor network, and a decay parameter β. The minimum value of k is 2.
- The imbalance parameter ε is determined by a global imbalance parameter $\varepsilon_0 \in [0, 1]$, with the simple formula $\varepsilon = k\varepsilon_0$.

The hyper-parameters k_0, β, and ε_0 are then tuned using “a combination of randomization and Bayesian optimization [28]”. The optimization also involves a number of other parameters, like the cutoff at which to stop partitioning and use a simpler greedy algorithm, which variant of the partitioning algorithm to use ([26] vs. [27]), and how to take edge dimensions into account. We find that those other parameters are either irrelevant for a typical tensor network arising in quantum computation (which usually has dimension 2 for all the edges), or does not end up matter much in terms of minimizing the contraction cost.

In our experiments, we also noticed that partition step at the top-level is significantly different from all the others. The reason is that the tensor network we want to evaluate is usually either a closed tensor network, or one with only a few open edges. On the other hand, after the first partition step, there are usually many open edges in the larger parts of the tensor network that needs further partitioning. That makes it clear which small tensors can be stripped without affecting the overall structure of contraction, so the balance constraint is less necessary. Therefore, we use different rules to determine k and ε for the top-level partition step and all the others:

- For the top level, k is still set to k_0, and ε is calculated as $\varepsilon = (k-1)\varepsilon_0$. (This is because, as mentioned above, the reasonable range of ε is $[0, k-1]$ rather than $[0, k]$.)

- For all other partition steps, k is fixed at 2, i.e. we do bi-partitioning for all steps except for the top-level one. We also use a different imbalance parameter $\varepsilon = \varepsilon_1$.

Furthermore, to optimize over ε_0 and ε_1, we use CMA-ES [22], which is a more recent and more effective stochastic optimizer than the Bayesian optimizer used in [9]. We also decide that, since k_0 is a discrete parameter with relatively few reasonable values, it is not suitable to be tuned with the hyper-parameter tuner. Instead, for each run of the algorithm, we enumerate small values of k_0 starting from 3, and find the one that gives the most optimal contraction cost. We take advantage of the computer we run our algorithm on, which has 28 CPU cores, to run the partitioning procedure in parallel for those different values of k.

3.4 Local optimization

The heuristic approach described in the last section still only gives a rough contraction order that may be easily improvable, even locally. We regard local optimization as an individual component of our algorithm, because it is used not only to refine the initial contraction order, but also during the dynamic slicing procedure in order to refine the contraction order for the sub-tasks.

In order to describe our local optimization algorithm, it is conceptually easier to consider the contraction order as a *contraction tree*, a representation which is actually already implicit in the partition-based initial order finding algorithm.

Definition 3.15 (Contraction tree)**.** Every contraction order for a tensor network G can be transformed into a *contraction tree*, a rooted tree where each leaf represents an input tensor, and each internal node represents a contraction step. It can be defined recursively:

- If the contraction order is trivial (i.e. consists of zero steps), then the contraction tree has only one node representing the only input tensor.
- If the contraction order ends with the contraction $G_{t-1} \xrightarrow{H_t} G_t$, where $H_t = G_{t-1}$, then the root of the contraction tree represents the contraction of H_t, and its children represents the tensors in H_t. Each immediate subtree is a contraction tree for all the contraction steps needed to get the corresponding tensor in H_t.

In particular, for a pairwise contraction order, the corresponding contraction tree will be a regular binary tree.

Compared with the contraction order, the contraction tree loses some information in that the order between independent contraction steps (i.e. those in disjoint subtrees) is not specified. This is not very important: The exact amount of memory used may be slightly affected, but the contraction cost and the contraction width remains the same.

The main advantage of the contraction tree is that "local" segments of the contraction procedure is more evident. Take any connected subgraph of the contraction tree that forms a regular binary tree. The subgraph can be viewed as a contraction tree. In fact, the original contraction order can be easily reordered such that all contraction steps corresponding to internal nodes of the subgraph are performed consecutively. Therefore, if we can find a contraction order for the underlying smaller tensor network with a lower cost, then we can reduce the contraction cost of the original contraction order.

For finding the optimal contraction order for the smaller tensor network, we again use the dynamic programming method implemented in `opt_einsum` [19]. It can find the optimal contraction order for tensor networks with up to 14 vertices. Therefore, when finding subgraphs from the original contraction tree, we limit ourselves to regular binary trees with up to 13 internal vertices and 14 leaves.

One final problem is which subgraphs to do local optimization on. For a tensor network with hundreds to thousands of tensors, there are many such subgraphs possible, and it would be infeasible to run the dynamic programming method on them all. In choosing the subgraphs, we take into consideration that:

- The parts of the contraction tree we want to optimize the most is the ones that contribute the most to the contraction cost. There is no need to optimize a part that has negligible cost compared to the total cost to begin with.

- We want those subgraphs to cover as much of the contraction tree as possible. There may be local segments with as few as 2 internal vertices and 3 leaves that has a more cost-efficient contraction order, and the more we cover, the more likely we are to find those.

- We do not want those subgraphs to overlap. On one hand, for a fixed number of subgraphs, the less the overlap, the more the coverage. On the other hand, non-overlapping subgraphs are also more convenient to implement, because the program can first find all the subgraphs, then local optimize all of them (potentially in parallel) without worrying that the results will interfere with each other.

Based on those considerations, we use the following procedure to find the subgraphs for local optimization:

- We scan through the contraction order in reverse. For each contraction step s we encounter:

    - Build an initial subgraph T that contains s as the only internal node, and the two children of s as the only leaves.
    - Choose a leaf from T that corresponds to the largest tensor. If the order of that tensor is larger or equal to a threshold (chosen to be 7 in our program), and that leaf is an internal node in the original contraction tree, then we change the leaf into an internal node, and add both its children into T as new leaves.
    - Repeat the previous step until the size of T is already the maximum that can be handled (i.e. 13 internal nodes and 14 leaves), or we cannot change the leaf into an internal node according to the previous rule.

- Continue scanning through the contraction order, ignoring any contraction step that has already been chosen as an internal node in a previous subgraph.
- Do local optimization on all subgraphs found this way.

We call the above procedure a *local optimization round*, and do multiple such rounds to locally optimize a contraction order. In order to cover as much of the contraction tree as possible, especially when the previous local optimization round results in little or no change, we further modify the procedure by choosing a random offset between 0 and 12, and skip over that many leaves to change to internal nodes at the beginning of each local optimization round. That helps each local optimization round to find different subgraphs that cover the contraction tree more thoroughly.

The number of local optimization rounds we use varies throughout the overall algorithm. Namely, we do 20 rounds before greedily slicing each edge as described in Section 3.5 (including immediately after finding the initial contraction order), and then 100 more rounds for the contraction order to be used in the final contraction scheme.

3.5 Dynamic slicing

The two previous sections have been focused on finding contraction orders for a single tensor network. In order to make use of our parallel tensor network contraction algorithm, we need a *contraction scheme*, which consists of not only a contraction order for the tensor network after slicing, but also a set of edges to slice. Therefore, finding good choices of edges to slice is as important as finding a contraction order for the resulting tensor networks.

One basic point to keep in mind is that slicing an edge will not *save* time complexity, only divide it into more manageable pieces.

Theorem 3.4. *Let G be a tensor network, and $G - e$ be the tensor network resulting from slicing an edge e (with edge dimension 2) in G. Let c and c' be the minimum pairwise contraction cost for G and $G - e$, respectively. Then $2c' \geq c$.*

Proof. Suppose that p' is a contraction order for $G - e$ with contraction cost c'. Then we can construct a contraction order p for G by simply contracting the same tensors as p'. Each contraction step in p will either involve one more edge than the corresponding contraction step in p' (if e is involved), or the same number of edges (if e is not involved). Therefore, the cost of each single step in p would either be twice the cost of the corresponding step in p', or the same cost. Let the cost of p be c_0, then $c_0 \leq 2c'$, and since c is the minimum pairwise contraction cost, we have $c \leq c_0$. Thus, we have $2c' \geq c$. □

Corollary 3.1. *Let G be a tensor network, and c be the minimum pairwise contraction cost for G. Then every pairwise contraction scheme for G has contraction cost at least c.*

Proof. Suppose that the set of edges to slice is S in a given contraction scheme, and let $k = |S|$ be the number of edges sliced. For $i = 1, \ldots, k$, let c_i be the minimum pairwise contraction cost of the tensor network resulting from removing the first i edges of S from G. Then we have $2^k c_k \geq 2^{k-1} c_{k-1} \geq \cdots \geq 2c_1 \geq c$. □

Therefore, the best we could hope for is that $2^k c_k$ be as close to c as possible. In practice, we cannot know the actual minimum cost, so we usually take c to be the cost of the initial contraction order we found, and define the *efficiency* of a contraction scheme as c'/c, where c' is the total cost of the contraction scheme. This is not a very scientific definition, and in unusual cases, the efficiency calculated this way may be higher than 100% because the initial contraction order is not optimal, but it does give a way to assess how efficient it is to slice edges according to a given contraction scheme.

Finding the optimal set of edges to slice is probably as difficult as, if not more difficult than, finding the optimal contraction order. One strategy is to manually inspect the tensor network, and choose edges to slice according to a pattern that seems to help reduce the contraction cost and the contraction width. In fact, this is the strategy used in [8]. Since the edges to slice are chosen manually beforehand, we call this strategy *static slicing*. The problems with static slicing are:

- By definition, it involves human work, and cannot be fully automated. This may be fine when evaluating many tensor networks with the same structure, but would be problematic if one want to contract tensor networks with different structures.

- It is difficult for a human to visualize the optimal contraction order for a hypergraph, and thus even more difficult for a human to choose good edges to slice. In Chapter 4, we will show that the contraction schemes used in [8] is not close to the most efficient possible for that tensor network.

In [6], we proposed using a greedy algorithm to heuristically and *dynamically* find good edges to slice. Later papers like [10, 9, 4] all adapted the same basic idea of greedy dynamic slicing, with only minor modifications.

The basic idea is simple: Starting from a tensor network G, we try removing each single edge e, find out the optimal contraction cost for each resulting graph $G - e$, and select the edge that give the lowest contraction cost to slice. Repeat this process to slice more edges, until the final contraction order is feasible in terms of contraction width.

Of course, in practice, we cannot find out the objectively optimal contraction cost for anything. We can only heuristically find good contraction orders, and even a reasonably good contraction order may take a moderately long time; therefore, doing so separately for each edge in a graph may be impractical.

The strategy we actually use is to just find a contraction order for G, and use it for every $G - e$, like in the proof of Theorem 3.4. Empirically, we observe that a good contraction order for G is usually also a good enough contraction order for $G - e$. This is exactly why our algorithm needs an initial contraction order to begin with.

In summary, the procedure we use now for finding contraction schemes for tensor network G is as follows:

1. Use the partition-based algorithm described in Section 3.3.2 to find an initial contraction order p for G.

2. Repeat the following steps until the desired number of sub-tasks is reached, or until the contraction width for each sub-task is low enough:

   (a) Run 20 rounds of local optimization to improve the current contraction order p.

   (b) For each edge e in G, find the contraction cost of $G - e$, using the current contraction order p to determine which tensors to contract in each step.

   (c) Select an edge e_0 such that $G - e_0$ gives the lowest contraction cost. Remove e_0 from G.

3. Finally, run 100 more rounds of local optimization to get a final contraction order that is actually used for the sub-tasks.

Reducing the contraction width. In many cases, a tensor network is infeasible to contract even after slicing it into a number of sub-tasks equal to the number of processors available, on grounds of the contraction width being too large. In this case, we simply keep slicing until the contraction width becomes small enough. It may seem that this could be inefficient because the dynamic slicing procedure only tries to optimize the contraction cost, and not the contraction width. However, as noted at the end of Section 3.2.1, just optimizing the contraction cost would in fact help reduce the contraction width, too. As the algorithm slices edges that are involved in the most expensive contraction steps, the contraction width will naturally go down.

The main reason we focus on the contraction cost during the dynamic slicing procedure is that the contraction width is too discrete, and does not provide enough granularity to guide the greedy slicing procedure. In many cases, slicing a single edge would not suffice to reduce the contraction width by 1, and every edge would be the same to the greedy algorithm if we only focus on the contraction width. Using the contraction cost ensures that this “tie” is usually broken and a good edge to slice is chosen. The fact that it also makes every single slice more efficient is a nice bonus.

3.6 GPU implementation of tensor network contraction

In order to implement a complete parallel tensor network contraction algorithm, we need to not only find a good contraction scheme, but actually implement the contraction process itself. Of course, we want our underlying implementation to be competitive in terms of performance compared to other similar software packages. Nowadays, GPUs are commonly present on supercomputers and high performance computing clusters, and for floating-point computation, they provide a much better performance than CPUs in terms of raw FLOPS. Therefore, we focus our effort on optimizing our GPU implementation.

Following the implementation in [9], we use the just-in-time compilation capability of the JAX library [29] to compile the sub-tasks before running them on the GPU. The compilation process takes a noticeable amount of time. However, since all the sub-tasks in a contraction scheme are the same except for the input tensors, this compilation process only needs to be executed once on each GPU node of the cluster[1]. In use cases where each node needs to do a large number of sub-tasks, the compilation time can therefore be ignored.

[1]We do not know if JAX is capable of outputting “compiled functions” in a format that can be transferred to other computers with the same GPU, which would allow us to only compile once and use the result on all GPU nodes, but this does not matter much since the compilation processes are happening in parallel anyway.

Inefficient contraction steps and branch merging. On the GPU, a pairwise tensor contraction step is usually compiled into an invocation of the GEMM (general matrix multiplication) algorithm. Consider the sub-network H with two vertices u and v that corresponds with a pairwise contraction step. In most cases, the closed edges of H are exactly the ones that is shared between its two vertices. Suppose that there are m open edges associated with u, n open edges associated with v, and k closed edges shared between both. Let $M = 2^m$, $N = 2^n$, and $K = 2^k$. Then the contraction step can be compiled as follows:

1. Transpose and reshape the tensor represented by u to an $M \times K$ matrix A.
2. Transpose and reshape the tensor represented by v to a $K \times N$ matrix B.
3. Invoke GEMM to compute the matrix product $A \times B$.

The theoretical time complexity of this is $O(MK + KN + MKN) = O(MKN) = O(2^{m+k+n})$, which agrees with the contraction cost for this contraction step, according to Definition 3.7. Therefore, the running time is supposed to be approximately proportional to the contraction cost.

However, experiment shows that not all GEMM operations have the same efficiency on the GPU. The reason is that GEMM operations on GPU is usually computed with *kernel functions*, which are usually designed to optimize the case where M, N, and K are all large. In particular, the GPU we used (Nvidia Tesla V100) only achieves its peak efficiency when M, N, and K are all multiples of 32.

Meanwhile, our contraction scheme finding procedure is likely to output a contraction order where a number of small tensors are sequentially "absorbed" into a large tensor one by one. Let the large tensor be u and the small tensor be v, then this is a case where M is very large, while both N and K are usually 16 or smaller. Therefore, this kind of contraction steps have high cost but low efficiency, which negatively affects the performance of our contraction algorithm.

We try to deal with this problem by *branch merging*, i.e. changing the contraction order such that the small tensors are merged together before being absorbed into the large tensor. As an example, consider the following tensor network:

$$T_{ijkl_1l_2\dots l_m} = A_{i'j'k'l_1l_2\dots l_m} B_{ii'} C_{jj'} D_{kk'}, \qquad m \gg 5.$$

If we absorb the small tensors B, C and D sequentially into A, then the contraction cost is $2^{m+4} + 2^{m+4} + 2^{m+4} = 3 \cdot 2^{m+4}$. Meanwhile, if we contract B, C, and D together and then contract the result with A, then the contraction cost is $16 + 64 + 2^{m+6} = 4 \cdot 2^{m+4} + 80$. Therefore, in terms of contraction cost, the first contraction strategy is better.

However, in reality, on the GPU due to the way the kernel functions are designed, absorbing an 8×8 tensor into A takes no more time than absorbing one 2×2 tensor. Meanwhile, because B, C, and D are small in all dimensions, the time cost for contracting them together can be ignored. Therefore, the second contraction strategy actually performs better on the GPU.

We scan through our final contraction order to detect contraction structures of the first kind, and change it into the second kind. This usually goes against the direction that local optimization takes the contraction order to, so obviously this procedure needs to be done after the last round of local optimization.

We note that branch merging is only an ad hoc solution to the problem of inefficient contraction steps. In fact, it is likely that specialized kernel functions can be designed in order to do the "small tensor into large tensor" type of GEMM operations without so much loss of efficiency, which would render the use of branch merging unnecessary, and also significantly improve the overall performance of our algorithm.

3.7 Discussion

As can be seen from this chapter, designing and implementing an efficient parallel tensor network contraction algorithm that makes use of dynamic slicing is a huge project. The core of our algorithm, the contraction scheme finding procedure, has three components, and each component has a number of plausible alternatives. With only a limited amount of benchmarking, it is usually hard to say which alternatives are better. Even peripheral modules like the implementation of single contraction steps can be tricky, as evidenced by the inefficiency problem described in Section 3.6. There is almost certainly room for improvement on our algorithm; if nothing else, the factor of randomness introduced by the hypergraph partitioning algorithm, by the hyper-parameter tuner CMA-ES, *and* by the random offset in the local optimization procedure leaves something to be desired.

Above all, we feel that our intuitive understanding of tensor network contraction is still somewhat lacking. It is difficult to visualize tensor network contraction, so at times it can be hard to tell what our algorithm is doing under the hood. Many key ideas in our algorithm were found out essentially by trial-and-error on a specific kind of tensor network (see Chapter 4), rather than with an underlying intuition that applies to general tensor networks. We hope that this would be changed in the future.

CHAPTER 4

Classical simulation of quantum supremacy circuits

In this chapter, we report the results of applying our parallel tensor network contraction algorithm to the task of simulating “quantum supremacy circuits”, a class of quantum circuits that are designed to be hard to simulate classically [5, 8, 2]. This is the most direct application of our algorithm, and indeed the one that inspired this work in the first place. The importance of this problem mainly lies in that there have been several previous works that attempt to solve the same task with similar methods, so it works well as a benchmark of our algorithm.

Our experiment results show that our algorithm performs better than all comparable classical algorithms in terms of running time. For the largest circuits considered in [2], we achieve a $2000\times$ speedup over our predecessor, Cotengra [9]. For smaller circuits, our algorithm is able to do the “supremacy task” faster than the quantum device in [2], potentially challenging the validity of their quantum supremacy claim.

In particular, we note that the methods used in both [8] and [9] fall under our paradigm of “contraction schemes”, which was first introduced (though not explicitly defined nor named) in [6]. The significant speedup we achieved in [10, 4] compared to those works shows the advantage of dynamic slicing ([10] vs. [8]) and local optimization ([4] vs. [9]).

4.1 Introduction

4.1.1 Overview of quantum supremacy

Even though the idea of quantum computers having an exponential advantage over classical computers dates back all the way to the 1980s, when the concept of “quantum computers” was first introduced [30], the phrase “quantum supremacy”, a concrete goal meant to capture the same idea, was only coined a few years ago in [31]. Basically, quantum supremacy means a *programmable*[1] quantum device solving a specific computational task that arguably cannot be feasibly solved on any existing classical computing device, not even costly supercomputers. Whether the specific computational task is actually *useful* is not taken into consideration in the definition of quantum supremacy.

The obvious candidate for this easy-for-quantum computational task is the simulation of a quantum system — even easier, simulation of the quantum computer itself. Even then, the task is not entirely trivial on the quantum side. To make the computational task well-defined, there needs to be an ideal model of the quantum computer, which the behavior of the real quantum device would unavoidably deviate from. Therefore, it is necessary to allow some margin of error in the definition of the computational task, but that would make the task easier to classical computers, too.

From the other side, it is surprisingly difficult to argue that simulating a specific class of quantum circuits that can be executed on a quantum device is infeasible for classical computers. Quantum devices that can be built in the near term would all have some constraints, such as connectivity constraints, on the class of quantum circuits that can be executed. The existence of noise also means that the circuit cannot be too large, as otherwise any information about the output distribution of the circuit would be drowned in the noise. Specifically designed classical algorithms may be able to exploit those constraints, and solve the task in a reasonable time even though the theoretical time complexity is exponential.

4.1.2 Random quantum supremacy circuits

In [5], it is suggested that *random* quantum circuits should be used for the purpose of demonstrating quantum supremacy. More precisely, the topology of the circuit is fixed, as dictated by hardware considerations, but each individual gate in the circuit is randomly chosen according to certain rules. The advantage of random circuits is that, in terms of gate pattern, they do not have any apparent structure that could potentially be exploited by a classical simulator. This is also based on the intuition that quantum simulation is a hard problem in general, and not only for a small fraction of hard instances. Later, [8, 2] give revised definitions of random circuits, both to reflect the topology of quantum chips built and in response to potential classical simulation strategies, but they are all based on the same principle.

A problem with defining quantum supremacy tasks based on random circuits is that it is difficult to measure how well a (quantum or classical) device is performing the task. Unlike "conventional" computational tasks like integer factoring (which could be solved by

[1]This specification is necessary so that one cannot claim, for example, that a chemical molecule achieves quantum supremacy.

Shor's algorithm [32] if fault tolerant quantum computers existed) which has well-defined outcomes of success and failure, the output of sampling an n-qubit random circuit is a probability distribution spread over the set of all bit strings of length n, and typically no single bit string has a non-negligible probability. This makes it hard to define the computational task so that it can be verified that a quantum computer can indeed do it.

To solve this problem, [5] proposes using the *cross entropy* between the experimental distribution and the ideal distribution to assess how close the former is to the latter. Given a sequence of experimental samples, the cross entropy can be estimated if *single-amplitude simulation* of the ideal circuit is feasible, i.e. the probability of each individual bit string can be computed efficiently. When this is not possible, the best one can do is to extrapolate the cross entropy from small-scale experiments, and validate the extrapolation results through theoretical models of quantum device fidelity.

4.1.3 Solving "supremacy tasks" with tensor network contraction

Despite not having any apparent structure, it turns out that random circuits also have a weakness that can be exploited by the classical computer. The weakness is related to the problem mentioned in the previous section: As demonstrated in [33], since the output distribution of a random circuit is typically not concentrated on any single string, it can be efficiently sampled with negligible error using the technique of frugal rejection sampling, if only *single-amplitude simulation* for the random circuit is available. Therefore, the same algorithm that would allow estimating the cross entropy without resorting to extrapolation would also allow the "supremacy task" to be solved classically, invalidating the claim of quantum supremacy.

Of course, even single-amplitude simulation of quantum circuits is far from easy for intermediate-size quantum circuits that may be possible to execute on current quantum devices. The most promising framework for single-amplitude simulation of quantum circuit is tensor network evaluation, the very problem that our algorithm in Chapter 3 tries to solve. Each amplitude of the output state of a quantum circuit can be represented as a closed tensor network, which can be evaluated with our parallel contraction algorithm on a classical supercomputer or computer cluster, with space and time complexities depending on the width and cost of the contraction scheme. The probability of a bit string can then be easily evaluated as the square of the absolute value of the amplitude.

Furthermore, the capability to evaluate open tensor networks may also help accelerate the speed the "supremacy task" can be solved classically. From a quantum circuit, we can construct open tensor networks with k open edges that correspond to a "batch" of 2^k

amplitudes. As noted in [8], the time required to evaluate such an open tensor network is often not much more than the time required to evaluate a closed tensor network for a single amplitude, while having the batch of amplitude instead of a single amplitude can remove a $10\times$ overhead incurred by the frugal rejection sampling method in [33]. Therefore, by evaluating open tensor networks instead of closed ones, we can solve the "supremacy task" even faster using our algorithm.

Supremacy circuits as a tensor network contraction benchmark. Thanks to the prominence of the quantum supremacy proposal in [5], there have been many attempts to simulate the random supremacy circuits classically, and many of them make use of some variant of tensor network contraction, including [6, 10, 4] by us and [34, 8, 35, 9] by other researchers. Therefore, those circuits and their associated tensor networks work well as a benchmark, both for optimizing the performance of our algorithm and for comparing with other algorithms. Because the class of circuits are designed to be as hard to simulate classically as possible, we believe that methods that perform well on those circuits would also perform well in general.

4.2 Preliminaries

4.2.1 Rules for random circuit generation

Several versions of random supremacy circuits has been proposed. In this chapter, we report the benchmark results of different versions of our parallel tensor network contraction algorithm on three different versions of those circuits. They differ in topology, in the gate set, and also in the rule to randomly choose each gate from the gate set. Some of those changes reflect the architecture details of existing or prospective quantum devices, and others are meant to defeat previously proposed classical simulation strategies.

Rectangular circuits. In [6], we simulate an early version of random circuits inspired by [5][2]. The rules for circuit generation are as follows:

- The qubits are arranged in a regular rectangular lattice, and all initialized to $|0\rangle$.
- There are $1+m$ *cycles* (i.e. layers) of gates arranged according to the following rules:

[2]The circuits are not exactly the same because the description of them in [5] was somewhat ambiguous, and the circuit files are never published.

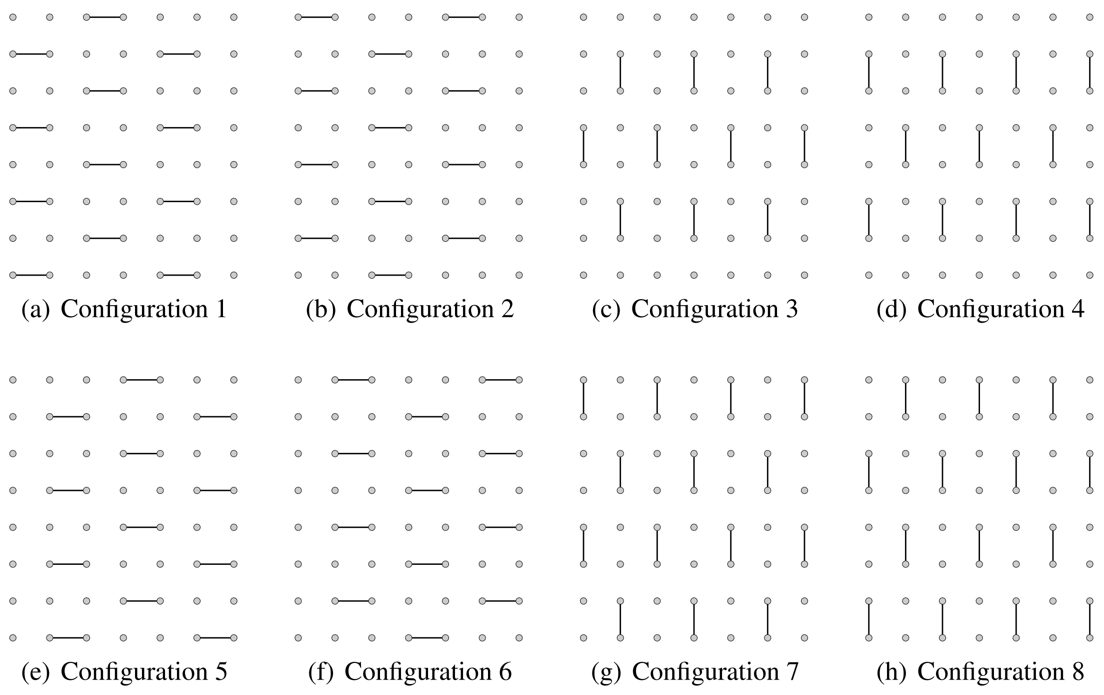


Figure 4.1: The 8 configurations of CZ gates for rectangular circuits [5]. This example is for a circuit with $8 \times 7 = 56$ qubits, but the pattern is the same for all circuit sizes.

1. The first cycle consists of one Hadamard gate applied to each of the qubits. (The different nature of this cycle from the other cycles is why we do not count it in the cycle count m.)
2. For each of the rest of the cycles, first place two-qubit CZ gates in one of 8 possible configurations, which repeat in a fixed sequence through the cycles, as shown in Figure 4.1. Each cycle covers approximately $1/8$ of all pairs of (horizontally or vertically) adjacent qubits. Note that each qubit is only involved in one CZ gate per cycle, and furthermore no two adjacent qubits are involved in *different* CZ gates in the same cycle. This is due to a hardware restriction: In the proposed quantum device architecture, adjacent CZ gates would suffer from large crosstalk noise.
3. Place single-qubit gates in those same m cycles. For each qubit, we place a single-qubit gates only in cycles where the qubit is not occupied by a CZ gate, but the *previous* cycle does has a CZ gate on that qubit. (The intuition is that two consecutive single-qubit gates does not make classical simulation harder, but increases the noise level in the quantum device.) The single-qubit gate is chosen according to the following rules:

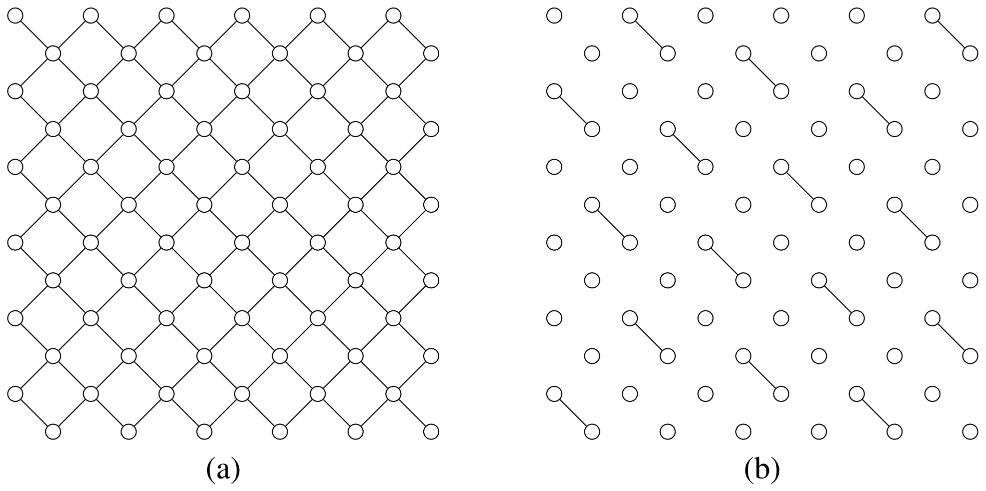


Figure 4.2: (a) Qubit layout of 72-qubit random Bristlecone circuits. Notice that this layout can be seen as a rectangular lattice rotated by 45 degrees and trimmed. Bristlecone circuits with less qubits use a sub-lattice of this lattice; for example, 70-qubit Bristlecone circuits has the two qubits in the upper left corner and the lower right corner removed. See [8] for all the lattices used. (b) An example configuration of CZ gates. Notice the similarity with the configurations in Figure 4.1.

(a) If there is no single-qubit gate on the same qubit in previous cycles (except for the initial Hadamard gate), place a T gate.

(b) Otherwise, choose the gate uniformly at random from $\{\sqrt{\mathrm{X}}, \sqrt{\mathrm{Y}}, \mathrm{T}\}$.

The rule to force T gates is meant to prevent simulation strategies based on Clifford circuit simulation [36].

- Finally, a computational basis measurement is applied to all qubits.

The final measurement will generate a bit string of length n equal to the number of qubits in the circuit. The probability distribution of this bit string will vary depending on the random choices of gates in the circuit. The “supremacy task” is, loosely speaking, to sample from this ideal distribution.

Bristlecone circuits. In [8], a second version of supremacy circuits is proposed based on the architecture of the Google Bristlecone GPU. The main difference between this version of supremacy circuits and the rectangular circuits described above are:

- The qubits are now arranged in a “Bristlecone lattice”, where only half of the qubits in a rectangular lattice is present, and the connections between qubits are diagonal, instead of horizontal and vertical, as shown in Figure 4.2. Alternatively, the new lattice can also be regarded as a regular rectangular lattice, trimmed into a diamond shape by cutting off all four corners, then rotated by 45 degrees.

- There are now an additional cycle of Hadamard gates after all the other cycles, before the final measurement.

- The sequence of CZ gate configurations is adjusted. The new order is (b), (g), (f), (d), (a), (h), (e), (c) from Figure 4.1. Note that now the layers alternate between horizontal and vertical CZ gates.

- The rule for adding single-qubit gates are changed as follows:

  1. In each cycle, if a qubit is not occupied by a CZ gate, but the previous cycle has a CZ gate on the qubit, then add a gate chosen uniformly at random from $\{\sqrt{\mathrm{X}}, \sqrt{\mathrm{Y}}\}$.
  2. In each cycle, if a qubit is not occupied by a CZ gate, but the previous cycle has a $\sqrt{\mathrm{X}}$, $\sqrt{\mathrm{Y}}$, or H gate on the qubit, then add a T gate.

  This change is in response to the revelation in [6] that, since the T gate is a diagonal gate, it can give rise to tensor networks that are easier to evaluate, especially when it appears following (or preceding) a CZ gate.

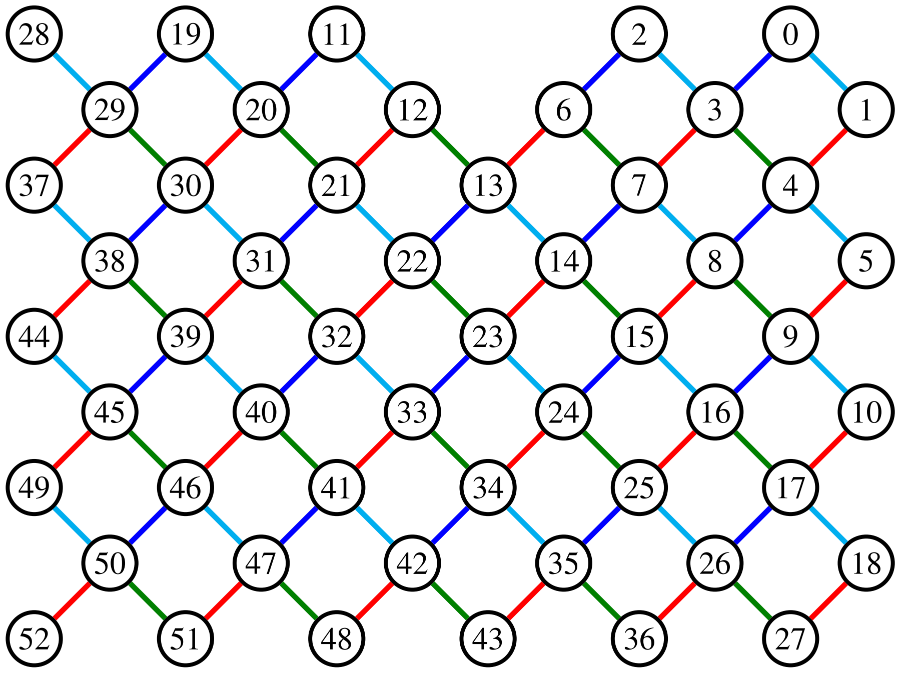
The circuit files generated by this set of rules are published by Google [37], and we directly use those published circuits to benchmark our algorithm.

A variant of the Bristlecone circuits is also proposed in [8], where iSWAP gates given by the unitary $|00\rangle\langle 00| + |11\rangle\langle 11| + i|01\rangle\langle 10| + i|10\rangle\langle 01|$ are used instead of CZ gates, but the circuit structure is otherwise the same. However, neither [8] nor [10] used this variant for benchmarking.

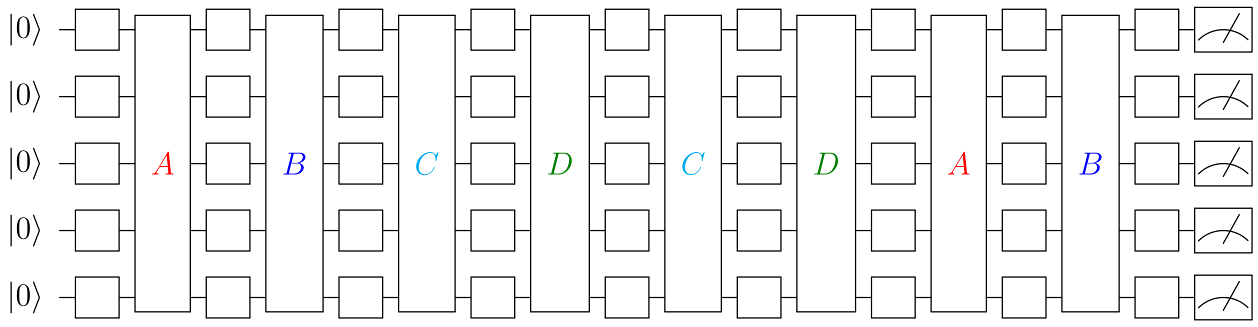
Sycamore circuits. In [2], the latest version of supremacy circuits is proposed based on the architecture of the Google Sycamore GPU, and it is claimed that quantum supremacy is achieved using those circuits. The qubit arrangement of the Sycamore GPU is actually pretty similar to that of the Bristlecone GPU, as shown in Figure 4.3 (a), with one qubit removed from a regular 54-qubit lattice due to a hardware defect. Apart from the qubit arrangement, this version of supremacy circuits also has several important differences from its predecessors:

- The circuit now consists of $m + 1$ cycles of single-qubit gates and m cycles of two-qubit gates in alternation. The circuit itself is still called an "m-cycle" circuit, where each cycle can be thought of as being composed of two "sub-cycles".

- Thanks to the previous modification, every cycle of single-qubit gates now can contain one gate applied to each qubit.



(a)



(b)

Figure 4.3: Structure of the 53-qubit random Sycamore circuits. (a) Layout of the 53 qubits and the 4 configurations of two-qubit gates, represented by lines in different colors. (b) Schematic diagram of an 8-cycle circuit.

1. In the first cycle, each single-qubit gate is chosen uniformly at random from $\{\sqrt{\mathrm{X}}, \sqrt{\mathrm{Y}}, \sqrt{\mathrm{W}}\}$.
2. In subsequent cycles, each single-qubit gate is chosen uniformly at random from $\{\sqrt{\mathrm{X}}, \sqrt{\mathrm{Y}}, \sqrt{\mathrm{W}}\} \setminus \mathrm{U}$, where U is the gate applied to the same qubit in the previous cycle of single-qubit gates.

  Since $\sqrt{\mathrm{W}}$ is neither a diagonal gate nor a Clifford gate, this helps make the circuit harder for simulation strategies based on (computational basis) tensor networks as well as those based on Clifford circuit simulation.

- The Sycamore architecture allows two-qubit gates in the same cycle adjacent to each other. As a result, each configuration of two-qubit gates can now cover approximately $1/4$ of the adjacent qubit pairs, and we only need 4 of them, as shown in Figure 4.3. The four configurations are named A (red in the figure), B (blue), C (cyan), D (green), and they repeat in the sequence ABCDCDAB.

- Finally, the two-qubit gates used are the so-called "fSim gates" (short for "fermionic simulation gates"), which are given by the unitary

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & e^{i(\Delta_+ + \Delta_-)} \cos\theta & -ie^{i(\Delta_+ - \Delta_{-,\text{off}})} \sin\theta & 0 \\ 0 & -ie^{i(\Delta_+ + \Delta_{-,\text{off}})} \sin\theta & e^{i(\Delta_+ - \Delta_-)} \cos\theta & 0 \\ 0 & 0 & 0 & e^{i(2\Delta_+ - \phi)} \end{pmatrix}.$$

  Those gates are the "natural two-qubit gates" for the Sycamore architecture. In addition, unlike the CZ gates, fSim gates are not diagonal, which also adds to the difficulty of classical simulation.

Using those circuits, [2] did experiments on a Sycamore quantum device, and claimed that this achieved quantum supremacy. The circuits used are released with [2], and again we use those circuits for benchmarking.

4.2.2 Frugal rejection sampling based on single-amplitude simulation

We distinguish between *single-amplitude simulation* and *true simulation* of a circuit.

Definition 4.1 (Single-amplitude simulation and true simulation)**.** Let $|\psi\rangle$ be a pure state of n qubits.

- *Single-amplitude simulation* of $|\psi\rangle$ means finding the amplitude $\alpha_s = \langle s|\psi\rangle$, given any single n-bit string s.

- *True simulation* $|\psi\rangle$ means sampling from the probability distribution arising from doing a computational basis measurement on $|\psi\rangle$, $P(X = s) = |\alpha_s|^2$.

Given a unitary circuit C, single-amplitude simulation (resp. true simulation) of C is defined as single-amplitude simulation (resp. true simulation) of the output state of C on a given input, $C|0\rangle$ (without loss of generality, we usually assume that the given input is $|0\rangle$).

In the literature, single-amplitude simulation is sometimes also known as "strong simulation", and true simulation "weak simulation". We find those terms misleading and confusing in the context of this thesis, because the capability to (efficiently) "strongly simulate" a circuit does not necessarily imply the capability to (efficiently) "weakly simulate" the same circuit (and neither is the converse true). In fact, as described in Section 4.3, tensor network based methods are better at single-amplitude simulation than true simulation. Meanwhile, for a fair comparison with quantum devices, it is necessary to have a way to do true simulation.

The "missing link" between single-amplitude simulation and true simulation is given by [33], which shows that true simulation can be reduced to single-amplitude simulation with a *frugal rejection sampling* procedure.

The validity and efficiency of frugal rejection sampling rely on the properties of the output distribution of a random circuit. As shown in [5], for large enough random circuits, the distribution of output probabilities $p(s) = |\alpha_s|^2$ approaches the *Porter-Thomas distribution* Ne^{-Np}, where $N = 2^n$ is the number of all possible bit strings. In other words,

$$\Pr_{s \sim \mathcal{U}(\{0,1\}^n)}[p(s) > p_0] \approx \int_{p_0}^{\infty} Ne^{-Np} dp = e^{-Np_0}.$$

When $p_0 \gg 1/N$, the right-hand side is a very small number. Therefore, we can use the following frugal rejection sampling procedure to sample from a distribution $\bar{p}(s)$ that is very close to $p(s)$:

1. Choose a parameter M. The larger M is, the closer $\bar{p}(s)$ is to $p(s)$, but the less efficient the sampling procedure will be.

2. Sample $s_0 \sim \mathcal{U}(\{0,1\}^n)$, and compute $p(s_0)$ with single-amplitude simulation.

3. Accept s_0 with probability $\min\{1, p(s_0)N/M\}$. Otherwise, repeat this process until one s_0 is accepted.

The probability P that we accept s_0 in one round of this process can be estimated as follows:

$$\begin{aligned}
P &= \sum_{s_0 \in \{0,1\}^n} \frac{\min\{1, p(s_0)N/M\}}{N} \\
&= \sum_{s_0 \in \{0,1\}^n} \frac{p(s_0)}{M} - \sum_{s_0 \in \{0,1\}^n}^{p(s_0) > M/N} \frac{p(s_0) - M/N}{M} \\
&= \frac{1}{M} - \frac{N}{M} \int_{M/N}^{\infty} \Pr_{s \sim \mathcal{U}(\{0,1\}^n)} [p(s) > p_0] dp_0 \\
&\approx \frac{1}{M} - \frac{N}{M} \int_{M/N}^{\infty} e^{-Np_0} dp_0 \\
&= \frac{1}{M} - \frac{e^{-M}}{M}.
\end{aligned}$$

We then have $\bar{p}(s_0) = \min\{1, p(s_0)N/M\}/NP$. Since $P \leq 1/M$, the only case where $p(s_0) > \bar{p}(s_0)$ is when $p(s_0) > 1/NP$. The statistical distance between $\bar{p}(s)$ and $p(s)$ can then be estimated as:

$$\sum_{s_0 \in \{0,1\}^n}^{p(s_0) > \bar{p}(s_0)} (p(s_0) - \bar{p}(s_0)) = \sum_{s_0 \in \{0,1\}^n}^{p(s_0) > 1/NP} (p(s_0) - 1/NP) \approx e^{-1/P} = \exp\left(-\frac{M}{1 - e^{-M}}\right).$$

If we take $M = 10$, then this statistical distance is about 4.5×10^{-5}, which is low enough to be negligible compared to the noise level in current quantum devices. The overhead caused by this sampling progress (i.e. the expected number of times single-amplitude simulation needs to be used to truly simulate a sample) is $1/P$. Since P is very close to $1/M$, $1/P$ is very close to $M = 10$. Therefore, we conclude that true simulation can be satisfactorily reduced to single-amplitude simulation with a $10\times$ overhead.

Batch of amplitudes. Suppose that we have a procedure that is slightly stronger than single-amplitude simulation, in that it can compute a "batch" of N_C amplitudes instead of one amplitude at a time. The amplitudes in the same batch correspond to bit strings that are related (e.g. same except for the last $\log_2 N_C$ bits). It is argued in [8] that this would give a more efficient way to do the frugal rejection sampling. The idea is simple: When we reject one s_0, we do not choose another s_0 uniformly at random from $\{0,1\}^n$, but pick the new s_0 from the same batch of bit strings. We change batch when we accept a bit string, so that there is no correlation between the samples.

Assuming that amplitudes in a batch are independent of each other, the probability that there is at least one bit string accepted from a batch is $1-(1-P)^{N_C}$, which is approximately

99.88% when $M = 10$ and $N = 64$. This is high enough that the overhead of sampling is now negligible. Of course, for this estimate to be sound, and for this procedure to not skew the distribution too much, the correlation between amplitudes in a batch must be low enough, which has been empirically tested in [8].

The bottom line is that, if we can compute a batch of 64 amplitudes as fast as we compute a single amplitude, then we can get rid of the $10\times$ overhead incurred by frugal rejection sampling. In Section 4.3, we will show how we could do this.

Sampling with lower fidelity. As can be seen in this section, the rejection sampling procedure actually gives a distribution with negligible distance to the ideal distribution. This is not the case for current quantum devices, which still suffer from a high level of noise, causing the fidelity between the experimental distribution and the ideal distribution to be low. It can be argued that this is unfair in the context of quantum supremacy.

In order to make the comparison fair, the classical algorithm should be allowed to do true simulation just with the same fidelity $f \ll 1$ as the quantum device, and this should translate to a time save. A simple method to do this is to sample from a *statistical mixture* of $p(s)$ and the uniform distribution:

$$\bar{p}(s) = fp(s) + (1 - f)\frac{1}{N}.$$

This can be done by doing the frugal sampling procedure only with probability f, and otherwise just outputting an s chosen uniformly at random. This reduces the average time needed by a factor of $1/f$. For example, when $f = 0.2\%$, the classical algorithm can be 500 times faster compared to sampling with fidelity 1.

4.3 Transforming circuits into tensor networks

As we alluded to in Section 1.1, it is easy to transform quantum circuits into tensor networks, because most basic concepts in quantum computation can be described as either matrix product or tensor product, both of which are easy to represent as tensor networks. Below, we give a general procedure for doing this transformation.

- Each input qubit in the circuit becomes an order-1 tensor, i.e. a 2-dimensional vector, corresponding to the initial state vector $|\psi\rangle$ of the qubit.
- In general, each unitary gate applied to k qubits becomes an order-$2k$ tensor, i.e. a $2^k \times 2^k$ matrix, corresponding to the unitary matrix of the gate.

  - However, diagonal gates (i.e. gates with a diagonal matrix) can be handled specially to simplify the tensor network. A diagonal gate applied to k qubits becomes an order-k tensor, i.e. a 2^k-dimensional vector, corresponding to the diagonal elements of its matrix.
  - Block diagonal gates, which usually represent "controlled" versions of gates (such as the CNOT gate), can also be simplified. If the $2^k \times 2^k$ matrix of a k-qubit gate can be written as a matrix of $2^l \times 2^l$ blocks that is block diagonal, then the gate becomes an order-$(k+l)$ tensor, corresponding to the 2^{k-l} blocks of size $2^l \times 2^l$.

- Each internal wire of the circuit becomes a closed edge, and they are connected to the aforementioned tensors in a natural way. When diagonal gates (or block diagonal gates) are involved, multiple different internal wires may become the same edge.
- Similarly, each output wire of the circuit becomes an open edge.

The above procedure does not take measurements into account, and it returns a tensor network with n open edges, whose value is an order-n tensor corresponding to the output state of the circuit. For large n, such a tensor network would usually be inherently too costly to evaluate in terms of both space and time.

The difficulty of computing whole state vectors is the reason why we consider single-amplitude simulation an easier problem to solve classically than true simulation. In single-amplitude simulation, we only need to evaluate a single amplitude α_s, so we can consider each measurement result known in advance. Therefore, we can add the following rule:

- A computational basis measurement on a qubit that is known to yield the result $i \in \{0, 1\}$ becomes an order-1 tensor, corresponding to the row vector $\langle i|$.

When we add such "measurement" tensors to all the output wires of the circuit, the result becomes a closed tensor network, with value equal to the amplitude α_s. In many cases, this closed tensor network will have a much lower contraction width and cost, making it feasible to do single-amplitude simulation on circuits where full state vector simulation is out of reach.

Special handling of computational basis vectors. In the above "measurement" rule, we make use of a row vector $\langle 0|$ or $\langle 1|$. Furthermore, in supremacy circuits, all qubits are initialized to the state $|0\rangle$. Those are all 2-dimensional *computational basis vectors*, with one component being 1 and the other component being 0. Such a tensor v can be handled specially to further simplify the tensor network.

The easiest way to explain the simplification is to consider what happens if we slice the only edge connected to v. The order-1 tensor v would be sliced into two scalars, one being 1 and the other being 0. Since tensor network is essentially a type of multiplication, one zero tensor in the inputs will make the whole result zero. Therefore, one of the slices can be discarded, and the other slice becomes a simplification of the original tensor network. This can be done for each computational basis vector in the tensor network.

The effectiveness of this simplification depends highly on the structure of the circuit. If the computational basis state or measurement is connected to a non-diagonal gate, then only that one tensor is sliced, and the effect on the contraction width and cost of the tensor network would be negligible. On the other hand, if the computational basis state or measurement is connected to one or more diagonal gates (since tensor network edges "go through" diagonal gates), not only will all of those tensors be sliced, but the connection between them would also be broken, which can significantly lower the contraction width and cost. The rectangular circuits described in Section $4.2.1$ is especially susceptible to this simplification, because the final measurements are often preceded with CZ and/or T gates.

Computing a batch of amplitudes. In fact, there is no need to add measurement tensors to *all* the output wires. Instead, we can leave out k of the measurement tensors, making the result of the corresponding measurements "unknown". The entire tensor network will then have k open edges, and evaluating it gives all 2^k amplitudes corresponding to varying the measurement results of those k measurements and fixing the rest. Therefore, this gives a way to compute a "batch" of amplitudes, which, as shown in Section $4.2.2$, can remove the $10\times$ overhead associated with frugal rejection sampling.

"Opening" edges of the tensor network like this will make it harder to evaluate. However, when k is relatively small, those open edges would not become a bottleneck, and the increase in contraction width and cost may be negligible. This is confirmed in our experiments in Section $4.4.3$, where we use $k = 6$ to compute a batch of 64 amplitudes at a time.

4.4 Benchmark experiments

In [6, 10, 4], we did "complete" benchmark experiments that start from the raw circuits, and end with running sub-tasks. The only caveat is that in most cases we did not run all the sub-tasks needed to calculate an amplitude or a batch of them. Instead, we extrapolated the running time for a complete task (here a "task" means an amplitude or a batch, not the entire supremacy task) by multiplying the average running time for one sub-task by the number of sub-tasks that need to be run on each node of the cluster. This is justified because each sub-task has exactly the same structure, and takes almost exactly the same time to finish. This is also necessary because the cost of the computational power needed to do a complete task, we must admit, is still high enough that we would not be able to afford to do it for every one of our experiments.

We did run a few experiments that really went the full distance and computed actual amplitudes on a computer cluster, as described in Section 4.4.2. We believe that those experiments are proof enough that, under our framework, the costs of scheduling and communication, as well as other potential hidden costs (such as loss of efficiency when the cluster is overloaded), are manageable.

We also do not include the time spent on preprocessing in the reported total time for a task, although in each case we separately give a bound on the preprocessing time. The justification is that, once we found a contraction scheme, it can be used to simulate multiple circuits with the same structure, or calculate multiple amplitudes or batches for the same circuit. This approach is also consistent with other papers that use similar methods for the same task, such as [34, 9].

4.4.1 Rectangular circuits

In [6], we did single-amplitude simulation on the rectangular version of supremacy circuits. Since there was no version of supremacy circuit publicly available at the time of [6], we generated the circuits ourselves according to the rules described in Section 4.2.1. Therefore, there is unfortunately no other work that we can compare our results in [6] directly with.

The rectangular circuits also causes an extra complication compared to the other variations of supremacy circuits, because the hypergraph structure of tensor networks for those circuits varies randomly even for the same circuit size. The reason is that each single-qubit gate is randomly chosen from the set $\{\sqrt{\mathrm{X}}, \sqrt{\mathrm{Y}}, \mathrm{T}\}$; the first two are non-diagonal gates, while the T gate is a diagonal gate. Therefore, the contraction width and cost for circuits of the same size can vary wildly depending on the number and positions of T gates.

In order to reflect the typical performance of our algorithm, we randomly generated 1000 circuits for each size, and reported the 80th percentile of running time, i.e. the time in which 80% of random circuits of a size could be simulated. This means that, for the remaining 20% of random circuits, either the running time exceeded the reported time, or the simulator ran out of memory. (We sliced each circuit a fixed number of times regardless of the contraction width, so running out of memory was normal.)

For this experiment, we used an early version of our contraction scheme finding proce-

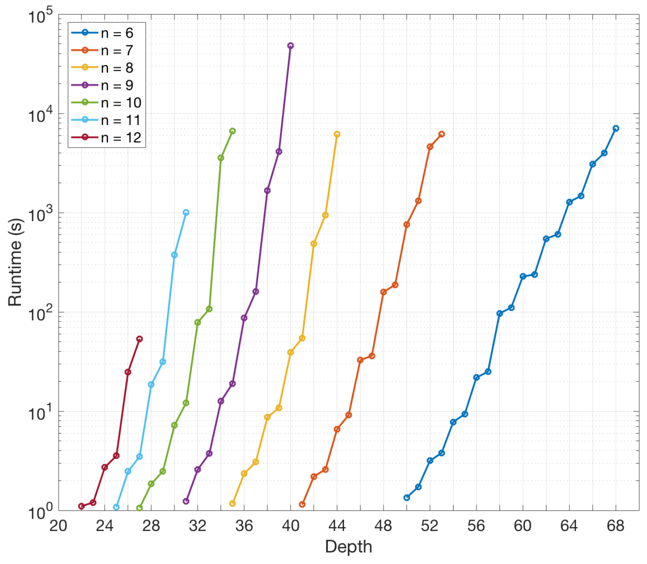


Figure 4.4: Estimated 80th percentile running times for rectangular random circuits. Here the circuit depth is defined as the value of m in the circuit definition in Section 4.2.1, i.e. not counting the initial layer of Hadamard gates.

dure, which implemented dynamic slicing similar to the current version, but the initial contraction order was based on a stock implementation of QuickBB [24], a heuristic treewidth algorithm, and there was no local optimization. Instead, we ran QuickBB again on the circuit after all rounds of dynamic slicing in order to find a (usually) better contraction order for the final sliced tensor network. We set a time limit of 60 seconds for both runs of QuickBB, which dominated the preprocessing time; therefore, the preprocessing time for each circuit is slightly longer than 2 minutes.

We designed our experiments based on having $2^{17} = 131\,072$ cores available, each core with 8 GB of memory ($1\,048\,576$ GB in total). This assumption is based on the size of the cluster we had access to at Alibaba group (which has $960\,000$ cores and $5\,120\,000$ GB of memory in total). As mentioned above, in each trial of the experiment, we fixed the number of edges to slice, which was set to 17 in the beginning. If it turned out that the algorithm ran out of memory on more than 20% of the circuit instances, then we increased this number until at least 80% of the circuits can be simulated.

The results of those experiments are shown in Figure 4.4. As can be seen, the largest depth simulated for each circuit size (subject to the 80% rule) are $6 \times 6 \times 68$, $7 \times 7 \times 53$, $8 \times 8 \times 44$, $9 \times 9 \times 40$, $10 \times 10 \times 35$, $11 \times 11 \times 31$, and $12 \times 12 \times 27$. Among those sizes, the hardest in terms of running time is $9 \times 9 \times 40$, where the 80th percentile of running time is about $50\,000$ seconds.

4.4.2 Bristlecone circuits

In [10], we did single-amplitude simulation on Bristlecone circuits published by Google [37], which are the same circuits as the ones simulated in [8]. The contraction scheme finding procedure we used was similar to that used in Section 4.4.1, except that we ran QuickBB after slicing every edge, instead of only at the beginning and the end. In a sense, QuickBB played the same role as local optimization plays in the current version of our algorithm. In addition, we also set a longer time limit for each QuickBB invocation. Those modifications increased the preprocessing time to about one day for each circuit size. However, this is justified by the fact that the tensor network structure for each circuit size is now exactly the same, so the contraction scheme found can now be used for any number of random circuits of the same size.

Notably, for this class of circuits, we actually computed amplitude values in addition to estimating the running time by extrapolation. In some experiments, we computed multiple random amplitudes individually, in order to reduce the communication overhead per amplitude. The experiments were run on 1449 Alibaba Cloud Elastic Computing Service (ECS) instances, each with 88 Intel Xeon(Skylake) Platinum 8163 vCPU cores @ 2.5 GHz and 160 GB of memory. The total number of cores is $127\,512$, and the total amount of memory is $231\,840$ GB. As a comparison, the experiments in [8] used two different clusters, each with an assortment of different nodes. The total number of cores and total amount of memory are respectively $245\,536$ and $932\,864$ GB for the Pleiades system, and $124\,416$ and $589\,824$ GB for the Electra system.

For the full experiment, we used the OSS (Object Storage Service), which can be understood as a shared storage that can be accessed by all nodes on the cluster, for communication between them. We used an agent node (not included in the 1449 nodes mentioned above) to find the contraction scheme, serialize it into a file, and upload it onto the OSS, which all worker nodes would query at the frequency of once per second. When the worker node saw the contraction scheme file, it would use its ID to figure out which sub-tasks in the contraction scheme were assigned to it, and begin contraction. Similarly, the agent node would then begin repeatedly querying the OSS, where the worker nodes would upload their result files (named according to the ID of each worker node) to. The final summation was performed on the agent node. This model of communication is not particularly efficient, but our point is that the communication cost is negligible when the number of amplitudes to compute is large enough.

Table 4.1 shows the estimated and actual running times for each experiment we ran this way. The estimated times were measured on a single worker node, using approximately

m	k	N_s	t_{estimate}/N_s (s)	t_{actual}/N_s (s)	$t_{\text{estimate}}/t_{\text{actual}}$
28	8	200 000	0.03	0.04	75%
32	10	1 000	0.36	0.43	84%
36	16	10	4.56	7.6	60%
				6.6	69%
		100		5.6	81%
				5.9	77%
40	22	1	480.17	580.7	83%

Table 4.1: Estimated and actual running times for Bristlecone-70 circuits. In each experiment, we sliced k edges of a Bristlecone-70 $\times$ $(1 + m + 1)$ circuit, and computed N_s random amplitudes. For the circuit with depth 36, we ran four separate experiments, two with $N_s = 10$ and two with $N_s = 100$.

the same number of sub-tasks that each node would need to do for the full experiment. As can be seen, apart from the cases where $m = 36$ and $N_s = 10$ (which has artificially low efficiency because the estimated time was estimated using $N_s = 100$), the full experiment had around 80% the efficiency compared to the estimated time. This efficiency loss is acceptable in the context of challenging quantum supremacy, and we suspect that it is caused by the computers in the cluster lowering the CPU frequency in response to the power overload. On the other hand, the results showed a large advantage over [8], which reported running times of 104 and 128 seconds per amplitude on the HPC Pleiades and Electra systems, respectively.

4.4.3 Sycamore circuits

In [4], we simulated Sycamore circuits that were used to claim quantum supremacy in [2]. For this class of circuit, we used the most recent version of our contraction scheme finding procedure described in Chapter 3.

Since our ultimate goal is true simulation with the rejection sampling method in Section 4.2.2, following the precedent in [2], we calculate batches of $2^6 = 64$ amplitudes instead of single amplitudes. The edges we leave open are the ones corresponding to the output wires of either qubits $\{0, 1, 2, 3, 4, 5\}$ or qubits $\{10, 17, 18, 26, 27, 36\}$ in Figure 4.3. This choice is based on the following considerations:

- We want those qubits to be as close as possible, so that the extra complexity they introduce is localized.
- Furthermore, we want those qubits to be as disconnected as possible from other qubits near the output wires. To this end, we note that qubits $\{0, 1, 2, 3, 4, 5\}$ are

only connected to other qubits in two-qubit gate configurations A and B, while qubits $\{10, 17, 18, 26, 27, 36\}$ are only connected to other qubits in configurations C and D. Since the circuits used in [2] have an even number of (two-qubit gate) cycles, we can choose the "open qubits" so that they are disconnected from other qubits in the last two layers, i.e.,

- $\{0, 1, 2, 3, 4, 5\}$ if the number of cycles is $8t + 4$ or $8t + 6$;
- $\{10, 17, 18, 26, 27, 36\}$ if the number of cycles is $8t$ or $8t + 2$.

Another modification we do in order to match the setting of quantum supremacy is that we only estimate the time to simulate each Sycamore circuits to the same fidelity f achieved by the quantum device in [2] according to cross entropy benchmarking (XEB). As mentioned at the end of Section 4.2.2, this just means multiplying the running time by $f < 1$.

This group of experiments are also the first ones that we run on GPU. In order to match the estimations for qFlex in [2] which assumed running the classical simulation on the Summit supercomputer, we used an Nvidia Tesla V100 SMX2 GPU with 16 GB of GPU memory, which matches the specifications of the GPUs in Summit. We assume that we have 27 648 of those GPUs, the same number that Summit has. We also do branch merging in order to improve the GPU efficiency, as described in Section 3.6.

In Figure 4.5, we compare the performance of our algorithm with various other algorithms that are also run on Sycamore circuits, including qFlex, the Schrödinger-Feynman simulator [2], and Cotengra [9]. Since the experiments in [9] are run on a different GPU, we redo their experiment using the same methodology. This means that we do not attempt to apply Cotengra to open tensor networks in order to reduce the rejection sampling overhead. However, we do give a lower bound on the time that may be achieved with open tensor networks, by assuming the time needed to evaluate a batch of amplitudes is the same as that for a single amplitude. The contraction costs we reported for our own algorithm are the ones achieved before branch merging (which, as mentioned before, increases the contraction costs), and the FLOPS efficiency is computed with this contraction cost too.

As can be seen, for circuit depths $m = 12, 14, 16, 18, 20$, our algorithm consistently outperforms every other classical algorithm that we are comparing with. For shallow circuits with $m = 12$ or 14, our algorithm takes less than 90 seconds to do the sampling task with the same fidelity as the quantum device, surpassing even the quantum device itself, which takes 200 seconds regardless of the circuit depth. For the largest circuit depth $m = 20$, we can still do the sampling task in 20 days, improving upon the second best classical algorithm with a concrete implementation, Cotengra, by a factor of over 2000.

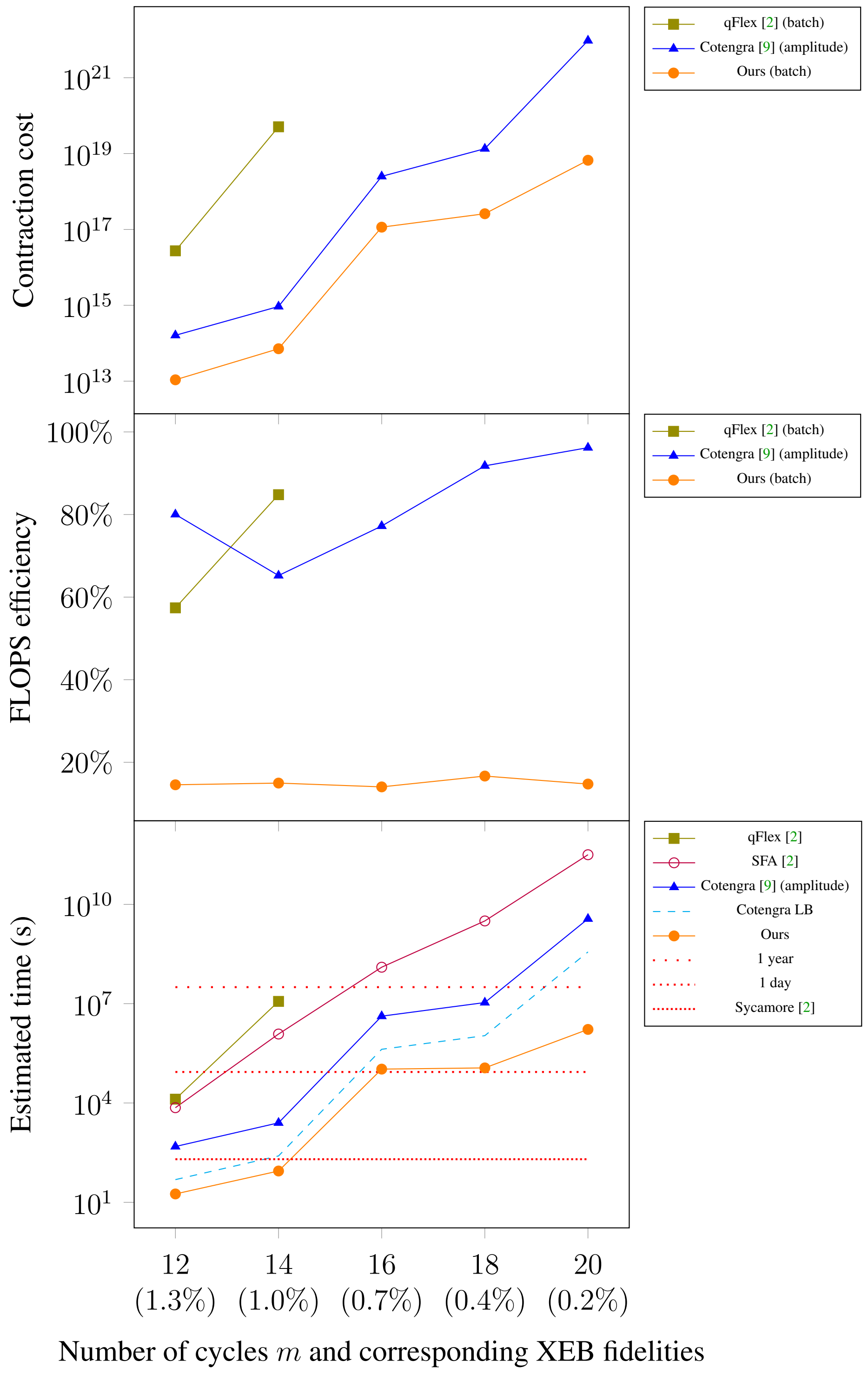


Figure 4.5: Contraction costs, FLOPS efficiencies, and estimated times to do the "supremacy task" for Sycamore circuits. We note that the version of Cotengra [9] we used cannot evaluate open tensor networks (i.e. batches of amplitudes), which the other tensor network based algorithms both use. "Cotengra LB" assumes that Cotengra can evaluate a batch of 64 amplitude as fast as it evaluates a single amplitude.

The shape of the running time curve of our algorithm is close to that of Cotengra, which is expected, since those algorithms are similar. The consistent advantage of our algorithm over Cotengra shows the advantage of the improvements we have done to the partition-based initial order finding algorithm, as well as our local optimization procedure.

However, we note that the FLOPS efficiency of our algorithm is consistently lower than 20%, which reduced our advantage in running time somewhat. In fact, they would have been even lower without the branch merging procedure. The FLOPS efficiency computed with the contraction cost *after* branch merging is somewhat higher, but still does not come close to those achieved by Cotengra. This is disappointing, especially since we do not yet fully understand why our FLOPS efficiency is so low, but we also see this as an opportunity. If we could solve the problem of low GPU efficiency — especially if in a way that no longer require us to use branch merging which is inefficient in theory — then the running time of our algorithm may be further improved.

4.5 Discussion

The concept of quantum supremacy is a controversial one, and rightfully so. Above all, it is difficult to convincingly argue that a computational task is infeasible for *any* classical algorithm to solve. Statements of this type that are generally accepted usually either are based on the failure of people to come up with an efficient solution — *for decades, despite researchers' best efforts* — or rely on complexity theory reductions to such problems. Complexity theory reductions tend to prove worst-case hardness, rather than average-case hardness. Plus, they are less strong in regimes where exponential time algorithms may be acceptable, since even a polynomial change in input size may exponentially affect the running time.

Meanwhile, not only is quantum supremacy a relatively recent concept, but the details of the supremacy tasks themselves change a lot as new prospective architectures for quantum devices are developed. This gives researchers little time to tackle each specific iteration of the task. Furthermore, the fact that the task does not necessarily have any practical use also limits the motivation for researchers to solve it. Therefore, even if there were no good classical algorithms at the time the quantum experiment was run, it would be quite a stretch to claim that the problem is classically infeasible. Indeed, our results in this chapter, especially compared with [9] which came out just a little earlier, demonstrates exactly how shaky the basis of any current "quantum supremacy" claim is.

Regardless, the prominence of the quantum supremacy proposal does give us a good benchmark for quantum circuit simulation as well as tensor network contraction. While analyzing the performance of our algorithm and trying to find improvements, we have also noticed some interesting phenomena. For example, the contraction trees we found usually have a “stem”, a path on which most of the heavy computations happen (see [4] for details). It would be interesting to see whether this is a peculiarity of our own algorithm, an idiosyncrasy of the random supremacy circuits (which are supposed to have as little structure as possible), or a general fact that holds for a wide range of tensor networks.

CHAPTER 5

Applications to the Quantum Approximate Optimization Algorithm

In this chapter, we investigate applications of our algorithm to the Quantum Approximate Optimization Algorithm (QAOA) [11], one of the most promising potential applications of near-term quantum computation devices. QAOA is based on quantum adiabatic algorithms, but instead of simulating changing the Hamiltonian slowly, which would require a circuit with exponentially many layers of small rotations, it uses a circuit with a reasonable number p of layers and not necessarily small rotations, with the rotation angles being parameters of the algorithm. By optimizing over those parameters, nontrivial results can be obtained even with a small value of p.

The usually proposed method for optimizing the parameters is with a feedback process, where a quantum processor tries the algorithm with different parameters and a classical algorithm optimizes for the best result. However, due to the inherent randomness in quantum computation, the classical part would have to be a *stochastic* optimizer, which is usually less efficient than optimizers for deterministic functions.

In [12], we directly compute the *expected value* of the objective function using tensor networks. By only considering the *lightcones* of each individual clause, we can efficiently calculate the expected value for some problems and values of p. We find that our implementation of this algorithm performs better than existing software packages that solve the same problem. In the case of small-cycle-free graphs, the speed of our implementation allows doing the classical parameter optimization completely classically.

In addition to computing the expectation value, we also attempt to sample from the final state with tensor networks. However, even though we are able to simulate measuring each qubit individually, we find it infeasible to sample from the joint distribution. Therefore, when quantum devices become available, it may make sense to still implement QAOA as a hybrid quantum-classical algorithm, by optimizing over the parameters completely classically, then using those parameters on a real quantum device to do the actual sampling.

5.1 Introduction

5.1.1 Problem and motivation

The utility of current quantum computing devices are still extremely limited, not only due to the presence of noise, but also due to the engineering challenge of scaling up. As such, they are also known as "noisy intermediate-scale quantum" (NISQ) devices [38]. While some researchers consider it meaningful to demonstrate quantum supremacy with such devices, others strive to find more practical applications for them. Quantum algorithms inspired by physical models are usually robust against noise to some degree. For example, simulation of quantum dynamics is likely to give useful insights even on NISQ devices, and adiabatic quantum computing [39] also works to some extent in the NISQ regime (where it is also known as "quantum annealing").

The original proposal of adiabatic quantum computation works by explicitly varying the Hamiltonian of a physical system over time. Of course, by the universality of the circuit model, it can always be translated into a quantum circuit. However, the evolution time needed for adiabatic quantum computation is usually big, potentially even exponential. In order to translate such a physical process into the circuit model, the depth of the circuit must be large to limit the approximation error. Meanwhile, NISQ devices usually can only handle a small depth of circuit before the accumulated noise becomes too large. This greatly limits the potential of adiabatic quantum computation on NISQ devices based on the circuit model.

The Quantum Approximate Optimization Algorithm (QAOA) [11] is an interesting variant of adiabatic computing that works in the circuit model. In order to limit the noise, it only uses a small number of layers in the circuit, but each layer is parameterized by two rotation angles, γ_l and β_l. For certain values of those parameters, QAOA is known to approximate adiabatic quantum computation; however, the idea is that adjusting those parameters may give a better approximation, and potentially even an improvement on the time complexity of adiabatic quantum computation. Of course, there are also values for those parameters that gives completely wrong results. As such, the performance of QAOA depends heavily on the values of those parameters, which are usually supposed to be optimized for the specific instances of the problem.

The fact that QAOA is designed for NISQ devices means that it may be in the reach of our tensor network based circuit simulator, and the need to optimize for $\vec{\gamma}$ and $\vec{\beta}$ also makes classical simulation of QAOA more meaningful. Therefore, it is a good problem to try applying our algorithm to.

5.1.2 Applications of classical simulation to QAOA

In this chapter, we apply our parallel tensor network contraction algorithm to two related tasks: evaluating the QAOA energy function, and sampling from the QAOA output distribution.

Evaluating the energy function. Essentially, QAOA is an optimization algorithm with the goal of minimizing an objective function on the output bit string resulting from measuring the final state. Since quantum measurement is inherently random, the value of the objective function achieved would vary randomly even when the number of layers p and the parameters $\vec{\gamma}$ and $\vec{\beta}$ — and thus the final state — remain the same. The QAOA *energy function* $F(\vec{\gamma}, \vec{\beta})$ is defined as the expectation value of the objective function:

$$F(\vec{\gamma}, \vec{\beta}) = \langle \vec{\gamma}, \vec{\beta} | C | \vec{\gamma}, \vec{\beta} \rangle,$$

where $|\vec{\gamma}, \vec{\beta}\rangle$ is the final state of the QAOA algorithm, and C is the objective function represented as an *observable operator*, a diagonal matrix in this particular case.

The energy function is named as such because it corresponds to the energy of the physical system in adiabatic quantum computation. When p is large enough, there exists values of $\vec{\gamma}$ and $\vec{\beta}$ such that QAOA approximates an adiabatic quantum computation process that is slow enough to evolve to the final ground state, and the energy function will be equal to the minimum value of the objective function, i.e. any bit string that results from measuring the final state achieves this minimum. When p is smaller, such a goal may not be achievable, but we still want to come as close as possible. Therefore, it is desirable to choose $\vec{\gamma}$ and $\vec{\beta}$ so as to minimize $F(\vec{\gamma}, \vec{\beta})$.

The usual proposal for optimizing $\vec{\gamma}$ and $\vec{\beta}$ is to run QAOA on a quantum processor with different values of $\vec{\gamma}$ and $\vec{\beta}$, and use a classical algorithm to tune the values of those parameters in order to get better results. This would be a complex feedback process between the quantum device and the classical computer, as well as a difficult classical problem to solve. However, if there is a way to compute the value of $F(\vec{\gamma}, \vec{\beta})$ classically, then we can get around the feedback process, and we also only need an ordinary optimizer for deterministic functions in order to minimize $F(\vec{\gamma}, \vec{\beta})$.

Evaluating $F(\vec{\gamma}, \vec{\beta})$ becomes easier when we take into account that QAOA also requires the objective function C to have a particular structure. Namely, C must be the sum of a number of *clauses* C_j, each only depending on a constant number of bits in the bit string, so that the unitary operators $e^{-i\gamma_j C}$ needed by QAOA can be implemented efficiently in the circuit model. This means that the energy function can also be written as the sum of multiple functions, each only depending on a constant number of qubits in the final state.

As we will see in Section 5.3, rather than trying to evaluate the energy function as a whole tensor networks, it is much easier to evaluate it as a sum of multiple tensor networks, each corresponding to a clause. Those tensor networks are easy to evaluate with a "lightcone trick". In fact, the energy function evaluating algorithm can have *linear* or even *constant* time complexity, depending on the structure of the problem and the value of p. In such cases, it becomes possible to efficiently optimize $F(\vec{\gamma}, \vec{\beta})$ entirely on classical computers, which would help a lot with demonstrating QAOA on NISQ devices.

Sampling from the output distribution. Of course, a more ambitious goal for a classical simulator is to *truly simulate* QAOA, i.e. to sample from its output distribution. Unlike in Chapter 4, single-amplitude simulation would not help much, because in this case the ideal distribution is concentrated on one or few bit strings that minimizes the objective function. Indeed, if we want to solve a combinatorial optimization problem, simulating QAOA with a method like the one in Section 4.2.2 would have no advantage over simply computing the value of the *objective function* on randomly chosen bit strings.

As we will see in Section 5.4, there is a method to truly simulate a circuit with tensor networks, at the cost of doubling the depth of the circuit. It is actually rather similar to the way we evaluate each component of the energy function, at least for the first qubit measured. We did not use this method in Chapter 4 because doubling the depth would increase the contraction width and cost of most supremacy circuits too much, putting them out of reach for classical simulation. However, for shallow QAOA circuits, the "lightcone trick" allows us to efficiently compute, and thus sample from, the probability distribution of any single qubit.

Unfortunately, in order to sample an entire bit string from the output distribution, we need to also be able to sample the probability distribution of a qubit *conditioned on observed values of all previously measured qubits.* The only way we know of dealing with previously measured qubits is by adding them to the lightcone, which increases the complexity of the resulting tensor network quickly. We conclude that sampling from the output distribution of an instance of QAOA may not be feasible classically, even when evaluating the energy function is.

5.2 Preliminaries

5.2.1 Observable operators and Hamiltonian

In order to describe the idea of adiabatic quantum computation, it is necessary to introduce a model of quantum computation that is different from the circuit model. We first define the concept of observable operators.

Definition 5.1 (Observable operator)**.** An *observable operator* of a quantum system of dimension d is represented as a $d \times d$ matrix A that is Hermitian, i.e. such that $A^\dagger = A$.

If a state $|\psi\rangle$ is an eigenvector of A associated with eigenvalue a, i.e. $A|\psi\rangle = a|\psi\rangle$ for a scalar number a, then we say the value of A on $|\psi\rangle$ is a.

Otherwise, A only has an *expected value* on $|\psi\rangle$, given by $\langle\psi|A|\psi\rangle$.

By the spectral theorem in linear algebra, for any Hermitian matrix A, there always exist a unitary matrix U and a diagonal matrix $D = \mathrm{diag}(a_0, \ldots, a_{d-1})$, such that $A = UDU^\dagger$. Then the eigenvectors of A are given by $|a_j\rangle = U|j\rangle$ for $j = 0, \ldots, d-1$, each associated with the eigenvalue a_j. Furthermore, the value of A can be *measured* by applying $U^\dagger$ to a state, doing a computational basis measurement, then applying U. If the result of the computational basis measurement is j, then the measured value of A is said to be a_j. It can be easily verified that the expected value of this measured value is indeed $\langle\psi|A|\psi\rangle$. This is where the name "observable" comes from.

In quantum mechanics, there is a special observable operator H of every quantum system called the *Hamiltonian*. When the Hamiltonian is fixed, the state of the system evolves according to the rule

$$|\psi(t)\rangle = e^{-iHt/\hbar}|\psi(0)\rangle,$$

where

$$e^{-iHt/\hbar} = \sum_{j=0}^{d-1} e^{-iE_j t/\hbar}|E_j\rangle\langle E_j|$$

is a unitary operator, $|E_j\rangle$ are the eigenvectors of H, E_j are the corresponding eigenvalues, and $\hbar$ is the reduced Planck constant. We note that units of time and of the Hamiltonian can be chosen so that $\hbar = 1$; therefore, we will omit the factor $1/\hbar$ hereafter.

The eigenstates $|E_j\rangle$ are also known as *stationary states* because they do not evolve over time. (Remember that a global phase does not change the quantum state.) The expected value of the Hamiltonian is called the *energy*, and is conserved when the Hamiltonian is not time-dependent. The state with the lowest energy (which is easily seen to be a stationary state) is called the *ground state*.

When the Hamiltonian does vary over time, but slowly, the evolution of the system is known as *adiabatic*, and obeys the following theorem.

Theorem 5.1 (Adiabatic theorem [39])**.** *If the initial state of a system is the ground state of its initial Hamiltonian $H(0)$, the time-dependent Hamiltonian $H(t)$ varies slow enough, and there always exists a non-zero gap between the two lowest eigenvalues of $H(t)$ for $t < T$, then the state of the system at time T is the ground state $H(T)$.*

Essentially, the system remains in the ground state of its Hamiltonian at all times. This theorem is the basis of adiabatic quantum computation.

5.2.2 Adiabatic quantum computation

Adiabatic quantum computation is a general approach to solve combinatorial optimization problems (and thus in theory, combinatorial search problems) by the adiabatic evolution of a quantum system. Let the solution space of a combinatorial optimization problem be the set of n-bit strings, and the objective function $C(z)$ be defined as the sum of m clauses

$$C(z) = \sum_{j=1}^{m} C_j(z), \qquad z \in \{0,1\}^n,$$

where each clause $C_j(z)$ is a function whose value depends only on a constant number of bits of z. The goal is to find a z that (approximately) minimizes $C(z)$.

In order to apply the adiabatic theorem, we want to view $C(z)$ as a Hamiltonian of the system of n qubits, which is easily achieved by making it a diagonal matrix $C = C(z)|z\rangle\langle z|$. In addition, we also need an initial Hamiltonian with a non-zero spectral gap (i.e. gap between the two lowest eigenvalues) that we know the ground state of. For reasons that would become apparent soon, we will use the Hamiltonian

$$B = -\sum_{k=1}^{n} X_k,$$

where

$$X_k = I^{\otimes k-1} \otimes X \otimes I^{\otimes n-k}, \qquad X = |0\rangle\langle 1| + |1\rangle\langle 0| = |+\rangle\langle +| - |-\rangle\langle -|,$$

$$|+\rangle = H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle), \qquad |-\rangle = H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle).$$

The unique ground state of B can be shown to be $|+\rangle^{\otimes n}$.

Now, we define $H(t) = (1-s)B + sC$, where $s = t/T$ to be a Hamiltonian that changes from B to C when t goes from 0 to T. Note that the off-diagonal elements of $H(t)$ is equal to that of $(1-s)B$, and thus using the Perron-Frobenius theorem on $-H(t)$, it can be proved that the gap between the two lowest eigenvalues of $H(t)$ is non-zero for all $0 \leq t < T$. Therefore, the adiabatic theorem applies as long as T is large enough.

To summarize, in order to find a z that minimizes $C(z)$:

1. Initialize a system of n qubits into the state $|+\rangle^{\otimes n}$.
2. Vary the Hamiltonian of the system as $H(t) = (1-s)B + sC$, where $s = t/T$, as the time t goes from 0 to T.
3. Measure the final state in the computational basis to get the bit string z.

We note that the value of T that is "large enough" depends on the spectral gap of $H(t)$. Furthermore, even if T is not large enough to ensure that the system ends up at the ground state, the measurement result is still likely to be one of the lowest energy eigenstates of $H(T) = C$. Therefore, adiabatic quantum computation is capable of (and, in practice, is usually used for) approximate optimization. The detailed proofs are beyond the scope of this thesis.

5.2.3 Quantum Approximate Optimization Algorithm

The Quantum Approximate Optimization Algorithm is an algorithm inspired by adiabatic quantum computation that can be applied to the same problems, but works in the circuit model. Depending on its parameters, it can simulate adiabatic quantum computation when the circuit depth p is large, but may be able to give approximate solutions to an optimization problem even when p is small.

The form of QAOA can be derived by considering the *Trotter approximation* of an adiabatic quantum computation process. The Trotter approximation is given by:

$$e^{A_1+A_2} = \lim_{r\to\infty} \left[e^{A_1/r}e^{A_2/r}\right]^r,$$

where A_1 and A_2 are matrices. In the context of quantum state evolution, this is usually applied in the form of

$$e^{-i(H_1+H_2)t} = \lim_{r\to\infty} \left[e^{-iH_1t/r}e^{-iH_2t/r}\right]^r,$$

meaning that applying a Hamiltonian $H = H_1 + H_2$ for time t can be approximated by applying H_1 and H_2 alternately in r time steps, so that *each* of H_1 and H_2 is applied with *total* time t. The more time steps we use, the more precise the approximation is.

In order to use the Trotter approximation, we first divide the process into q time steps, fixing the Hamiltonian in each time step, then apply the Trotter approximation in each step, dividing each time step further into r sub-steps:

$$\begin{aligned}
|\psi(T)\rangle &= \lim_{q\to\infty}\left(\prod_{j=0}^{q-1} e^{-iH(jT/q)T/q}\right)|\psi(0)\rangle \\
&= \lim_{q\to\infty}\left(\prod_{j=0}^{q-1} e^{-i((q-j)B+jC)T/q^2}\right)|+\rangle^{\otimes n} \\
&= \lim_{q\to\infty}\lim_{r\to\infty}\left(\prod_{j=0}^{q-1}\left[e^{-i(q-j)BT/q^2r}e^{-ijCT/q^2r}\right]^r\right)|+\rangle^{\otimes n}.
\end{aligned}$$

Even though the coefficients are slightly complicated, the above expression has the following simple form:

$$|\psi(T)\rangle \approx |\vec{\gamma},\vec{\beta}\rangle = e^{-i\beta_p B}e^{-i\gamma_p C}\cdots e^{-i\beta_1 B}e^{-i\gamma_1 C}|+\rangle^{\otimes n}.$$

Here $p = qr$, β_j and γ_j are small numbers, and the order is reversed to show the order in which they appear in the quantum circuit (i.e. the order in which the unitaries are applied to the initial state). Both $e^{-i\beta B}$ and $e^{-i\gamma C}$ can be implemented efficiently with local gates:

$$e^{-i\beta B} = \bigotimes_{k=1}^{n} e^{-i\beta X_k}, \qquad e^{-i\gamma C} = \prod_{j=1}^{m} e^{-i\gamma C_j}.$$

Note that, since all C_j correspond to diagonal matrices, the decomposition above is valid, and indeed the gates $e^{-i\gamma C_j}$ can be applied in any order.

The above expression for $|\psi(T)\rangle$ gives a faithful approximation of adiabatic quantum computation in the circuit model. However, in order to simulate changing the Hamiltonian slow enough, the number of layers p needed is too large to be feasible in NISQ devices. Therefore, QAOA uses a different strategy, where p is chosen to be small, and the rotation angles $\vec{\gamma}$ and $\vec{\beta}$ are parameters that are not defined *a priori*. The resulting $|\vec{\gamma},\vec{\beta}\rangle$ is regarded as an ansatz that may or may not correspond to good solutions of the combinatorial optimization problem.

In order to use QAOA to actually solve a problem, we need to find values for parameters $\vec{\gamma}$ and $\vec{\beta}$ such that $|\vec{\gamma},\vec{\beta}\rangle$ does correspond to good solutions. The usual metric is the expectation value of the objective function C, also known as the *energy function*:

$$F(\vec{\gamma}, \vec{\beta}) = \langle \vec{\gamma}, \vec{\beta} | C | \vec{\gamma}, \vec{\beta} \rangle.$$

The goal of Section 5.3 is to evaluate the energy function classically, so that good values of $\vec{\gamma}$ and $\vec{\beta}$ can be found with a classical optimizer.

5.3 Evaluating the energy function

In order to evaluate the energy function $F(\vec{\gamma}, \vec{\beta}) = \langle \vec{\gamma}, \vec{\beta} | C | \vec{\gamma}, \vec{\beta} \rangle$ using our tensor network contraction algorithm, we need to represent the value in terms of one or more tensor networks. One naive idea is to use a single tensor network built as follows:

1. Using the method in Section 4.3, build a tensor network with n open edges that represents the output of the QAOA circuit, $|\vec{\gamma}, \vec{\beta}\rangle$.
2. Make a copy of the above tensor network, but take the complex conjugate of every input tensor. The result represents $\langle \vec{\gamma}, \vec{\beta}|$.
3. Build a third tensor network with n open edges representing the diagonal matrix C.
4. Connect the aforementioned tensor networks together by identifying open edges with each other (according to the qubits in the output state $|\vec{\gamma}, \vec{\beta}\rangle$ they correspond to), and changing them into closed edges.

It can be verified that such a tensor network would indeed have the value $\langle \vec{\gamma}, \vec{\beta} | C | \vec{\gamma}, \vec{\beta} \rangle$.

However, the third step above is the problematic part. It is difficult to build a tensor network for C, because C is defined as the *sum* of multiple clauses C_j, and a tensor network represents a kind of *product* of the input tensors. However, this does hint that maybe we should represent the energy function as the sum of multiple tensor networks.

Indeed, the energy function, which is the expectation value of the objective function, can be computed as the sum of the expectation values of all clauses:

$$F(\vec{\gamma}, \vec{\beta}) = \sum_{j=1}^{m} F_j(\vec{\gamma}, \vec{\beta}), \text{ where } F_j(\vec{\gamma}, \vec{\beta}) = \langle \vec{\gamma}, \vec{\beta} | C_j | \vec{\gamma}, \vec{\beta} \rangle.$$

Now, each F_j can indeed be represented as a tensor network using the idea described at the beginning of this section. Since C_j is a simple function of the value of only a few qubits, it can be simply represented with a single tensor.

Furthermore, there is actually a big advantage of focusing on a single clause. In order to illustrate the trick we can use to significantly reduce the cost to compute F_j, we write down tensor network representation of F_j as follows:

$$(\langle +|^{\otimes n}) e^{i\gamma_1 C} e^{i\beta_1 B} \cdots e^{i\gamma_p C} e^{i\beta_p B} \cdot C_j \cdot e^{-i\beta_p B} e^{-i\gamma_p C} \cdots e^{-i\beta_1 B} e^{-i\gamma_1 C} (|+\rangle^{\otimes n}).$$

We note that, unlike C, C_j is a tensor that is localized on a few qubits. Therefore, when building the tensor network, there are many open edges from $e^{i\beta_p B}$ that are *only* connected to the corresponding open edges from $e^{-i\beta_p B}$. In addition, $e^{i\beta_p B}$ and $e^{-i\beta_p B}$ themselves are composed of localized X gates. Therefore, there are many sub-networks of the above tensor network that are of the form $e^{i\beta_p X_k} e^{-i\beta_p X_k}$, which evaluates to I. Those sub-networks can be contracted and removed from the tensor network.

After removing those sub-networks, there will be some gates from $e^{i\gamma_p C}$ and $e^{-i\gamma_p C}$ that are now directly connected to each other, and can be removed. Then, some gates from $e^{i\beta_{p-1} B}$ and $e^{-i\beta_{p-1} B}$ can be removed in the same way. When n is large and p is small, such cancellations can get all the way to the "outermost layers" $e^{i\gamma_1 C}$ and $e^{-i\gamma_1 C}$. The resulting tensor network would be much smaller, and thus easier to evaluate, than the original.

This phenomenon can be explained by the physical intuition of "lightcone". In a quantum circuit, information can only be passed between qubits through multi-qubit gates. For a large but shallow circuit, the "past lightcone" of each output qubit — the part of circuit which that output qubit can receive information from — only actually covers a small fraction of gates and input qubits. Since each F_j only depends on a few output qubits, it makes sense that only a small portion of the tensor network is needed to evaluate each F_j.

In summary, by using the "lightcone trick", we can evaluate each $F_j(\vec{\gamma}, \vec{\beta})$ by only contracting a small tensor network, and then compute $F(\vec{\gamma}, \vec{\beta}) = \sum_{j=1}^{m} F_j(\vec{\gamma}, \vec{\beta})$. This gives an efficient method to evaluate the energy function of a QAOA algorithm when p is small.

The partial trace interpretation. We note that the way we evaluate the expected value of a clause F_j and implement the "lightcone trick" can also be regarded as an instance of *density matrix simulation*, which we will introduce in Section 6.5.1. The reason why density matrix simulation is useful for this problem is that it allows discarding a qubit by taking the partial trace over it, which can be implemented by performing a computational basis measurement on the qubit, but keeping the edge corresponding to the measurement result closed rather than making it open. When a qubit falls out of the "past lightcone" of the clause F_j, we can discard it, so that the size of the resulting tensor network is manageable.

It can be shown that this generates a tensor network that is trivially equivalent to the one generated by the cancellation process described above.

5.4 Sampling from the output distribution

In general, the method described in the previous section is an efficient method for evaluating the expected value of not only C_j, but any localized observable operator on the output state of a shallow quantum circuit. In particular, this method can be adapted to the problem of sampling from the output distribution of such a circuit.

First consider the problem of sampling the output distribution arising from a computational basis measurement on *a single qubit* in the output state $|\vec{\gamma}, \vec{\beta}\rangle$. This distribution is a Bernoulli distribution, so we only need to know the expected value of the outcome (which is 0 or 1). Importantly, the result of this measurement can be represented as an observable operator M, and the expectation value is then $\langle\vec{\gamma}, \vec{\beta}|M|\vec{\gamma}, \vec{\beta}\rangle$. Without loss of generality, assume that the qubit in question is the first qubit in the system. Then

$$M = (0 \cdot |0\rangle\langle 0| + 1 \cdot |1\rangle\langle 1|) \otimes I^{\otimes n-1} = |1\rangle\langle 1| \otimes I^{\otimes n-1}.$$

Similar to C_j, this observable operator is localized on a single qubit, so the "lightcone trick" can be applied. The resulting tensor network can be simplified a little further by representing M as a tensor network consisting of two tensors $|1\rangle$ and $\langle 1|$, and applying the simplification in Section 4.3 for computational basis vectors. This simplification only moderately reduces the complexity of the tensor network, since the main way M introduces connectivity in the tensor network is actually with the $I^{\otimes n-1}$ factor, which connects two pieces of the "lightcone" together.

The above gives an efficient procedure for simulating the computational basis measurement on any single qubit. However, this alone does not allow true simulation of the circuit, which required each qubit to be measured *sequentially*. In other words, we need to sample the distribution of some qubits while taking previous measurement results into account.

An obvious solution is to add those previous measurement results to the observable operator. Suppose that the first k qubits are already measured, and the measurement results form the bit string $s \in \{0, 1\}^k$. Then we take the next observable operator as

$$M = |s\rangle\langle s| \otimes |1\rangle\langle 1| \otimes I^{\otimes n-k-1}.$$

The expectation value of M reflects the *a priori* probability that measuring the first $k + 1$ qubits give the bit string $s|1$ (where the vertical bar stands for string concatenation), $p_{s|1}$.

Previously in the sampling process, we should already know the probability p_s. Then the conditional probability of measuring the current qubit to be 1 is $p_{s|1}/p_s$. Knowing this probability would allow us to sample the current qubit (and compute $p_{s|0} = p_s - p_{s|1}$ if necessary).

However, as k grows larger, the operator M becomes less localized, and its lightcone becomes larger. Even though larger k also makes the tensor network of M less connected (for example, when measuring the last qubit, the tensor network for $\langle \vec{\gamma}, \vec{\beta} | M | \vec{\gamma}, \vec{\beta} \rangle$ becomes two disjoint pieces, and we essentially only need to calculate a single amplitude for the circuit), it cannot nearly make up for the increase of size of the lightcone. The hardest cases are when k is large enough that the lightcone covers nearly the entirety of the circuit, but not so large that the connection between the two pieces of the lightcone becomes insignificant.

We note that we can adjust the order in which we measure the qubits, in an attempt to make the tensor networks that we need to contract during this procedure simpler. However, we do not know of a good method to adjust this order, and experimental evidence suggests that the effect of adjusting this order is limited. As far as we know, there are many QAOA circuits where evaluating the energy function for hundreds of values of $\vec{\gamma}$ and $\vec{\beta}$ is feasible, but sampling even one bit string from the output distribution is out of reach for classical computers.

5.5 Experiment results

In [12], we tested evaluating the energy function and optimizing for the parameters $\vec{\gamma}$ and $\vec{\beta}$ on MAX-CUT problems on unweighted regular graphs. The MAX-CUT instance on a graph $G = (V, E)$ is defined by the following objective function:

$$C(s) = \sum_{e \in E} C_e(s), \qquad C_{(u,v)}(s) = \begin{cases} -1, & s_u \neq s_v, \\ 0, & s_u = s_v. \end{cases}$$

Here we identify positions in the bit string $s \in \{0,1\}^n$ with vertices $v \in V$. As can be seen from this formulation, $n = |V|$, $m = |E|$, and each clause corresponds to an edge in E. In other words, the goal is to find a way to assign $s_v \in \{0,1\}$ to each vertex v, such that as many edges possible have different value assigned to both its endpoints.

MAX-CUT is one of the earliest known NP-complete problems [40], and its representation as a combinatorial optimization problem is about the simplest possible, since each clause only depends on the values of two bits in s. As such, it is likely to be one of the first problems usefully solved on a NISQ device with QAOA. Therefore, MAX-CUT is the problem of choice for our experiments.

All experiments below are run on a single Alibaba Cloud ECS instances with 24 Intel Xeon Gold 6149 CPU cores @ 3.1 GHz and 96 GB of memory.

5.5.1 Energy function evaluation on random regular graphs

We tested evaluating the energy function on random d-regular graphs with up to 1000 vertices, for $d = 3, 4, 5$, and with the number of layers in the QAOA $p \leq 5$. The random d-regular graphs are generated with the `random_regular_graph` function from the NetworkX library, which uses the definition of "random regular graph" and the algorithm from [41]. The graphs generated do not have any self-loops or parallel edges.

We note that, even when the number of vertices n is large, this size of each lightcone is still relatively small, as it is primarily determined by d and p rather than n. Therefore, we opt not to slice each tensor network further into sub-tasks, but to simply regard the $m = nd/2$ tensor networks themselves as sub-tasks and evaluate them in parallel.

For comparison, we also do the same task on three other libraries with quantum simulation functionalities: Cirq [42], Qiskit [43], and qTorch [44]. In each case, we use the implementation given as an example in the documentation of the library. Cirq and Qiskit are both general purpose quantum circuit simulation libraries, and their examples are based on state vector simulation. Therefore, we expect that they would not be able to handle instances with large n. Meanwhile, qTorch is based on tensor network contraction, similar to our work, so we expect that the time scaling of qTorch should be similar to ours.

We first compare the performance of the four algorithms on 3-regular graphs with numbers of vertices $n = 10, 20, 30, 50, 100, 1000$. The results are shown in Figure 5.1. As expected, both Cirq and Qiskit run out of memory when $n \geq 30$, which would require them to store a state vector with at least 2^{30} complex numbers. When $n \leq 20$, the performance of our algorithm is either better than or comparable to those algorithms.

Meanwhile, qTorch is much less robust than we expected. We find that it is very prone to random crashes on QAOA circuits with $p \geq 2$. Among 5 random instances for each number of vertices and each $p \geq 2$, qTorch only managed to produce an answer for two of them without crashing, one with $n = 10, p = 2$ and one with $n = 30, p = 2$. In the cases where it succeeds, it constantly takes more than $10\times$ the time our algorithm takes to compute one value of the energy function.

We also observe that, for smaller n, sometimes the running time may *decrease* a little when n increases. For example, when $d = 3$ and $p = 4$, the case $n = 100$ seems to run faster than $n = 50$. This is easily explained by the fact that when n is small, the

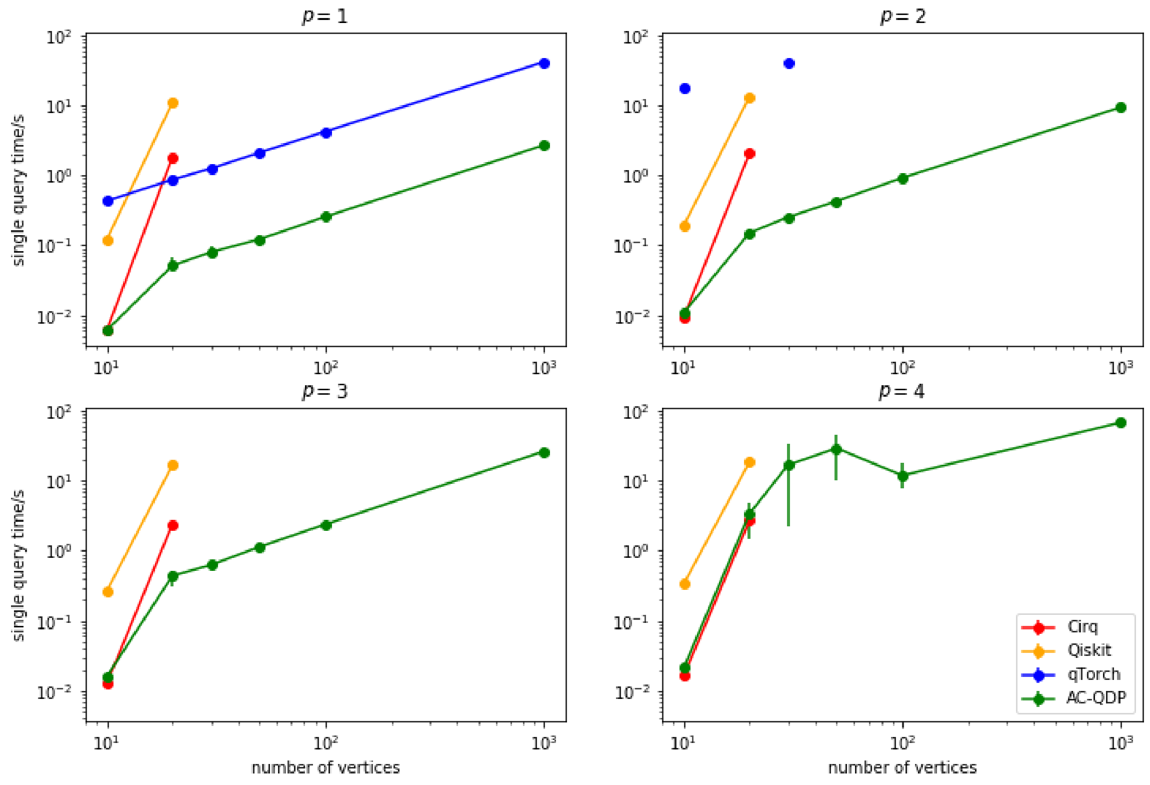


Figure 5.1: Comparison of the average time to evaluate the energy function on a single set of parameters, $\vec{\gamma}$ and $\vec{\beta}$, for 3-regular MAX-CUT instances. For each problem size, 5 random MAX-CUT instances are drawn. Both Cirq and Qiskit run out of memory when $n \geq 30$; qTorch crashed on all but two instances when $p \geq 2$.

	Time (s)			Vertices in a tree		
	$d = 3$	$d = 4$	$d = 5$	$d = 3$	$d = 4$	$d = 5$
$p = 1$	2.707	4.318	6.789	6	8	10
$p = 2$	9.433	20.455	36.913	14	26	42
$p = 3$	26.450	80.816	*	30	80	170
$p = 4$	68.651	*	*	62	242	682
$p = 5$	409.022	*	*	126	728	2730

Table 5.1: Time to evaluate the energy function on a single set of parameters, for d-regular MAX-CUT instances with $n = 1000$ vertices. Numbers in the columns labeled "vertices in a tree" is the number of vertices that would be involved in the lightcone of a single clause if the subgraph induced by those vertices is a tree.

random regular graph is more likely to contain small cycles, which causes tensor networks for single clauses to have a more complex structure, unlike in the large n limit where the tensor networks are more tree-like. This also explains why the running time in some cases varies much more with the choice of the random graph. For example, in the case $d = 3, p = 4, n = 30$, the running time has a large variance since the local structures of the random graphs vary greatly.

Next, we study the effects of d and p to the running time. We use d-regular graphs with 1000 vertices for $d = 3, 4, 5$, and increase the value of p until the evaluation task becomes infeasible. The results are shown in Table 5.1.

As can be seen, when feasible, the running time roughly scales proportionally to d^p, which can be explained by the hypothesis that most of the lightcones correspond to trees in the original graph, in which case the number of vertices involved in a lightcone would be given by

$$\frac{2(d-1)^{p+1}-1}{d-2}.$$

The case $d = 3, p = 5$ is an outlier, as well as some of the "*"s in the table that would be feasible if they followed the scaling. This is probably because that in those cases, the number of vertices in a "tree lightcone" is too large relative to the total number of vertices, so it is likely that some of the vertices coincide with another, and the lightcone is no longer tree-like.

5.5.2 Optimization for small-cycle-free graphs

The above observation naturally gives rise to the idea to study small-cycle-free graphs specifically. If a graph does not contain cycles with at most $2p + 1$ vertices, then all the lightcones for QAOA with p layers will have exactly the same tree structure, making the

d	p	Vertices	Time (s)	`dlib`	`de`	`FOURIER`	Procession	Grid search
3	1	6	0.0015	**0.692**	**0.692**	**0.692**	**0.692**	**0.692**
3	2	14	0.0048	0.738	**0.756**	**0.756**	**0.756**	**0.756**
3	3	30	0.0129	0.782	0.782	0.782	**0.792**	**0.792**
3	4	62	0.0349	**0.817**	0.805	0.811	0.806	-
3	5	126	0.0945	**0.822**	-	0.819	0.800	-
4	1	8	0.0019	**0.662**	**0.662**	**0.662**	**0.662**	**0.662**
4	2	26	0.0076	**0.716**	**0.716**	**0.716**	**0.716**	**0.716**
4	3	80	0.0306	0.737	0.737	0.739	0.735	**0.749**
4	4	242	0.1432	0.760	**0.761**	0.751	0.753	-
5	1	10	0.0021	**0.643**	**0.643**	**0.643**	0.357	**0.643**
5	2	42	0.0113	**0.691**	**0.691**	**0.691**	0.682	**0.691**
5	3	170	0.0666	0.709	0.709	**0.720**	0.711	**0.720**
5	4	682	0.6436	0.731	-	0.736	**0.739**	-
6	1	12	0.0025	**0.629**	**0.629**	**0.629**	**0.629**	**0.629**
6	2	62	0.0160	**0.673**	**0.673**	**0.673**	**0.673**	**0.673**
6	3	312	0.1422	0.690	0.690	**0.699**	0.692	**0.699**
7	1	14	0.0027	**0.619**	**0.619**	**0.619**	**0.619**	**0.619**
7	2	86	0.0218	**0.659**	**0.659**	**0.659**	**0.659**	**0.659**
7	3	518	0.2663	**0.674**	-	0.667	0.667	-

Table 5.2: Optimized expected value of each clause for small-cycle-free d-regular graphs, using various optimizers. Since the expected value is always in the range $-1 \leq C_j \leq 0$, we omit the minus sign for all of them; therefore a larger value is better. We also list the number of vertices in the lightcone and the average time per query.

expectation values of each clause equal and independent of n. In fact, the idea to optimize $\vec{\gamma}$ and $\vec{\beta}$ specifically for small-cycle-free graphs is also proposed in [45].

We used various classical numerical optimization algorithms to optimize $\vec{\gamma}$ and $\vec{\beta}$:

- The library function `dlib.find_min_global`.
- The library function `scipy.optimize.differential_evolution`, which we abbreviate as `de`.
- The `FOURIER` heuristic from [46].
- The "procession" heuristic from [45], where the parameters gotten from optimizing $p = p_0$ is used to initialize a local optimizer for $p = p_0 + 1$.
- A grid search over the parameter space, with the most promising result used to initialize a local optimizer.

All of them queries our tensor network contraction algorithm as a subroutine to evaluate the energy function for specific values of $\vec{\gamma}$ and $\vec{\beta}$. The results are shown in Table 5.2. As shown in the table, the largest values of parameters we could handle are $d = 3, p = 5$; $d = 5, p = 4$; and $d = 7, p = 3$. Within the parameter regime that we could handle, each query takes less than one second.

As expected, grid search always gives the best result when it is feasible. However, it is also the slowest numerical optimization algorithm, and becomes infeasible quickly when d and p increase. In general, all methods give similar results, in most cases within 0.02 of each other, although there are a few unexplained outliers such as 0.357 for the procession heuristic when $d = 5$ and $p = 1$.

5.6 Discussion

In this chapter, we attempt to apply tensor network contraction to two different tasks that arise in the study of QAOA, evaluating the energy function and sampling from the output distribution. For the former, we gave a method that solves the task satisfactorily for small values of d and p. Importantly, depending on the nature of the optimization problem, the scaling of the time complexity may be only linear in the problem size n, or even independent of n. We also demonstrated how to use this solution as a subroutine in order to find good values of the parameters $\vec{\gamma}$ and $\vec{\beta}$.

For the latter task, however, our solution is far from satisfactory. It can sample the measurement results of a few qubits efficiently, but if we want to sample a significant portion of qubits, then the time complexity becomes superpolynoimial in n. It seems that for classical simulators, evaluating is easier than sampling, since there are many instances where evaluating the energy function is trivial in terms of space and time costs, yet sampling with this method is not feasible.

While disappointing, this also gives an interesting idea for a hybrid quantum-classical implementation of QAOA. Notice that, in evaluating the energy function, it is the *quantum computer* that needs a large overhead, since many samples are required to get a good estimation of the energy function, which may be needed so that the optimization on $\vec{\gamma}$ and $\vec{\beta}$ is stable. Therefore, even when NISQ devices become available, it makes sense to optimize for $\vec{\gamma}$ and $\vec{\beta}$ using energy function values evaluated classically, and only use the quantum device for sampling when a good energy function value is found.

CHAPTER 6

Applications to quantum error correction

In this chapter, we use tensor network contraction to study quantum error correction codes. Quantum error correction codes is an essential component for fault-tolerant quantum computation, which is necessary for universal scalable quantum computation in the near future. Hence, the study of the properties of quantum error correction codes has both theoretical and practical importance. Existing theoretical analyses are usually based on simplistic error models such as the Pauli twirling approximation, which can only give crude approximations of the logical error rate. Realistic error models should give more precise estimations of the logical error rate, which may provide guidance on quantum hardware design.

In [13], we study a realistic error model based on the one presented in [14], with an important additional component, *crosstalk* induced by ZZ-interactions that are present between neighboring qubits even when they are idle. As our results show, crosstalk can significantly affect the logical error rate of quantum error correction codes, potentially necessitating measures to specifically mitigate it.

We focus on the quantum error correction code Surface-17, which involves 9 data qubits and 8 ancilla qubits. We simulate 3 rounds of error syndrome extraction, the minimum number of rounds needed for fault tolerant error correction. Therefore, the total number of syndrome bits measured is $8 \times 3 = 24$, and we can use a tensor that fits in the memory to represent the logical error for each of the 2^{24} results. This allows us to figure out the optimal decoder and exactly compute the logical error rate, without resorting to Monte Carlo sampling like is done in [14].

6.1 Introduction

6.1.1 Quantum error correction and surface codes

From a theoretical viewpoint, the ultimate goal of quantum computing would be to build a universal, scalable quantum computer, so that any problem in BQP could be solved in polynomial time. Currently, since all physical implementations of qubits are noisy, i.e. error-prone, the most promising approach to building a universal, scalable quantum computer is by utilizing quantum error correction codes. The famous quantum threshold theorem [47] states that, as long as the error of each individual quantum operation is below a certain threshold, then by encoding each logical qubit with an error correction code and carefully doing error correction, quantum circuits of arbitrary size can be evaluated fault-tolerantly in principle.

In practice, fault-tolerant quantum computation is a little more complicated. The proof of the threshold theorem in [47] uses concatenated codes, which are not very efficient, requiring a large number of qubits for any reasonable error parameters, more than what is feasible with current technologies. A more efficient family of codes would certainly help to make fault-tolerant quantum a reality sooner. One example of such a family of codes is the family of surface codes.

One peculiar characteristic of the surface codes is that their error correcting capacities are more than their code distance would suggest: Since they are defined on a lattice, whether a particular set of errors is correctable depends less on the error weight, and more on the layout of the error on the lattice. Unfortunately, this also means that not much is known about the actual error correcting capacities of surface codes. Even finding the optimal correction for a set of error syndromes is a nontrivial problem.

6.1.2 Realistic error models

Most existing studies on quantum error correcting codes are based on simple error models — either specific ones that does not consider many of the subtle effects on real hardware, or overly general ones to make sure the results are valid. Therefore, the results of such studies are either too optimistic or too pessimistic, and they also do not provide much insight on what could be done with the hardware designs to make them work better with error correction.

One of the studies that does use a more realistic error model is [14]. It simulated Surface-17, a code with 9 data qubits and 8 ancilla qubits. Due to the specific way the gates in the error correction circuit is arranged, it is possible to reorder the computation so that only one ancilla qubit is active at any time, making a density-matrix simulation feasible since only a 10-qubit density matrix needs to be maintained. Even then, the size of surface codes that could be simulated is limited, and some potentially important sources of errors on real hardware, like crosstalk and leakage, are left out. Moreover, the resulting density matrix is only for a single error syndrome, so the results are subject to sampling error, necessitating a large number of samples.

In this chapter, we aim to do *exact simulation* of Surface-17 by a tensor network that computes the effect of the noise together with the error correction process on the logical qubit, for each possible error syndrome. This means that our approach cannot handle too many syndrome extraction rounds: The most we can handle is 24 syndrome bits, which on Surface-17 corresponds to 3 rounds, the bare minimum needed for fault tolerant error correction. In return, we can incorporate some more complicated errors into our error model. We will focus on *crosstalk*, which is likely to be a significant source of logical error in near-term implementations of surface codes.

6.2 Preliminaries

6.2.1 Pauli matrices

We start with an introduction to the *Pauli matrices*, which play a major role in the theory of quantum error correction.

Definition 6.1 (Pauli matrices)**.** The *Pauli matrices* are

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \qquad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \qquad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}.$$

The Pauli matrices are both unitary matrices and Hermitian matrices. They have the following properties:

$$X^2 = Y^2 = Z^2 = I, \quad XY = -YX = iZ, \;\; YZ = -ZY = iX, \;\; ZX = -XZ = iY.$$

An *n-qubit Pauli operator* is defined as the tensor product of n matrices in $\{I, X, Y, Z\}$, with a scalar prefactor in $\{\pm 1, \pm i\}$. From the properties of Pauli matrices above, it can be seen that the set of all n-qubit Pauli operators is a group, called the *n-qubit Pauli group*. Furthermore, any two n-qubit Pauli operators P and Q either *commute* or *anticommute*, i.e. either $PQ = QP$ or $PQ = -QP$, which can be determined as follows:

- Two single-qubit Pauli operators in $\{I, X, Y, Z\}$ commute if and only if one of them is I, or they are the same.
- If $P = P_1 \otimes P_2$, and $Q = Q_1 \otimes Q_2$, then P and Q commute if any only if the pairs P_1, Q_1 and P_2, Q_2 either both commute or both anticommute.
- The scalar prefactor does not affect the commutativity of two Pauli operators.

Convention. When there is no danger of confusion, sometimes we omit the tensor product sign between Pauli matrices. For example, the notation XY usually means $X \otimes Y = X_1 Y_2$, which is a two-qubit operator (i.e. a 4×4 matrix) where a Pauli X is applied to the first qubit and a Pauli Y is applied to the second qubit. This is different from the "XY" in $XY = iZ$, where both Paulis are applied to the same qubit sequentially.

6.2.2 Basics of stabilizer codes

We define an $[[n, k]]$ *quantum code* as a way to encode a state $|\psi_L\rangle$ of a k-qubit system into a state $|\psi\rangle$ of a larger n-qubit system where $n > k$. The state $|\psi\rangle$ can only be in a limited subset, known as the *codespace*, of all possible n-qubit states. A *stabilizer code* is a quantum code where the codespace can be described by a set of n-qubit Pauli operators.

Definition 6.2 (Stabilizer code)**.** A *stabilizer code* is an $[[n, k]]$ quantum code where the code space is defined as

$$\{|\psi\rangle \mid \forall P \in S : P|\psi\rangle = |\psi\rangle\},$$

where S is a set of n-qubit Pauli operators. Without loss of generality, we can take S to be a subgroup of the n-qubit Pauli group, called the *stabilizer group*. Each element P of S is called a *stabilizer*.

There are several properties that S must satisfy. Obviously, S cannot contain $-I$, which implies that S cannot contain any anti-Hermitian Pauli operator (i.e. one with a prefactor in $\{\pm i\}$ rather than $\{\pm 1\}$), and that every pair of stabilizers in S must commute. It can also be shown that $|S| = 2^{n-k}$, i.e. S can be generated by a set of $n - k$ generators.

A stabilizer P can be regarded as an observable operator, as defined in Section 5.2.1. As such, it can be measured[1]. The outcome can be $+1$ or -1, but it should always be $+1$ for a state $|\psi\rangle$ in the codespace. Furthermore, it suffices to measure the $n - k$ stabilizers in any generator of the stabilizer group S to verify that $|\psi\rangle$ is indeed in the codespace. In general, such a set of $n - k$ measurement results is known as the *error syndrome*.

We do not discuss in detail the theoretical error correcting capability of stabilizer codes, which has been studied in a number of previous works, such as [48]. For our purpose, it suffices to say that any n-qubit state can be "corrected" into the codespace by measuring the error syndrome, then only applying Pauli gates on individual qubits. For any given error syndrome, there are multiple ways of doing such "correction", and they do not necessarily

[1]In practice, we also want the measurement not to otherwise disturb the code state, i.e. to yield no information other than whether the result is $+1$ or -1, so the method of measurement mentioned in Section 5.2.1 is not applicable. Section 6.2.3 gives a method of measuring specific stabilizers without otherwise disturbing the code state.

give the same code state. The best one to choose is usually the one that needs the fewest Pauli gates (i.e. that assumes that fewest qubits are affected by the error), but may depend on the actual error model.

Regardless, it is always possible to define a stabilizer code so that different "corrections" differ by only Pauli gates on the *logical* state. In Section 6.5.2, we will use this fact to find an *optimal decoder* under our error model with tensor networks.

6.2.3 Surface code design

We will focus on the family of surface codes and implementation proposed in [49]. The surface code consists of three kinds of qubits with a grid-like layout: data qubits, X-ancilla qubits, and Z-ancilla qubits. The data qubits store information about the logical qubit encoded, and the ancilla qubits are used to measure the error syndrome without otherwise disturbing the data qubits. More specifically, we focus on a specific code in this family, Surface-17, which is a $[[9, 1]]$ quantum code.

In a surface code, there are two types of stabilizers, X-stabilizers and Z-stabilizers, that form a generator of the stabilizer group. As the name indicates, an X-stabilizer is a tensor product of Pauli X operators on a subset of the qubits, and the same goes for a Z-stabilizer and Pauli Z operators. Furthermore, the qubits are laid out on a 2-dimensional surface, with each stabilizer in the generator only acting on a *local* subset of qubits. The qubit layout for Surface-17 is shown in Figure 6.1.

In order to use the surface code to store a logical qubit and protect it against errors, we first encode it into the data qubits, then repeatedly measure the error syndrome in order to detect and correct errors. At the end of Section 6.2.2, we have described how "ideal" error correction works, but in practice it is more complicated because the procedure to measure the error syndrome is also subject to errors. In theory, the error correcting process can be made *fault tolerant*, i.e. resistant to a single error occurring in *any* one component of the circuit[2], only if we combine the error syndromes measured in at least three consecutive rounds to determine how to correct the error.

On the other hand, applying Pauli gates to correct errors can be implemented in a way that is *not* subject to error, because no physical circuit is needed for those Pauli gates. Instead, we note that the effect of a Pauli gate P on the measurement results of a stabilizer Q thereafter can be predicted: The measurement result will be unchanged if P and Q commute, and flipped if P and Q anticommute. Therefore, we can apply Pauli gates conceptually by adjusting the measurement results thereafter. For preserving the value of a

[2]The actual definition is a little trickier; see [47] for a rigorous treatment of fault tolerance.

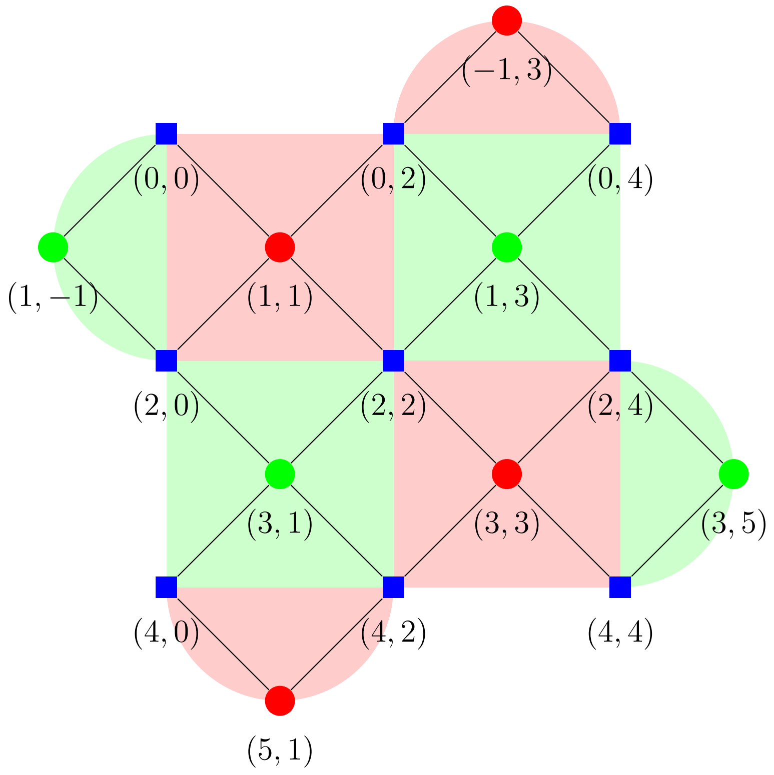


Figure 6.1: Qubit layout of Surface-17. Data qubits are represented by blue squares, and X-ancilla and Z-ancilla qubits by red and green circles respectively. Each ancilla qubit is used to measure a stabilizer, as shown by the shaded red and green regions. For example, the X-ancilla at $(-1,3)$ measures $X_{(0,2)}X_{(0,4)}$, and the Z-ancilla at $(3,1)$ measures $Z_{(2,0)}Z_{(2,2)}Z_{(4,0)}Z_{(4,2)}$.

logical qubit, the only operation that needs to be implemented on the hardware is extracting the error syndrome.

The circuit for a round of error syndrome extraction is conceptually simple:

1. For each Z-ancilla qubit, apply CNOT gates to it, with each adjacent data qubit as the control qubit for each CNOT gate. Then do a computational basis measurement to measure a Z-stabilizer. (0 and 1 correspond to $+1$ and -1 respectively.)

2. Apply Hadamard gates to all data qubits to switch to the Hadamard basis.

3. For each X-ancilla qubit, apply CNOT gates to it, with each adjacent data qubit as the control qubit for each CNOT gate. Then do a computational basis measurement to measure an X-stabilizer.

4. Apply Hadamard gates to all data qubits to switch back to the computational basis.

Note that in this implementation, ancilla qubits are not reset after each measurement (which may be difficult on hardware). Instead, we assume that measurement simply causes the qubit to decohere into the computational basis, and its value stays as is, barring errors, for the next round.

On actual superconducting hardware, the native two-qubit gate is CZ rather than CNOT. Therefore, the CNOT gates are implemented by sandwiching CZ gates between Hadamard gates (only two Hadamard gates are needed for each ancilla qubit each round). By the same token, Hadamard gates can be replaced by $R_y(\pm\pi/2) = e^{\pm i\pi Y/4} = (ZH)^{\pm 1}$, which achieves the same goal of switching between the computational basis and the Hadamard basis, and is easier to implement on the hardware.

Another detail is that, since the CZ gates all commute with each other, there is some freedom in the orders of all CZ gates on the Z-ancilla qubits, and all CZ gates on the X-ancilla qubits. Some of them may also be applied simultaneously. In the superconducting architecture considered in [49], the CZ gates are implemented by shifting frequencies of qubits so that the two qubits to apply the CZ on has the same frequency, different from that of all other adjacent qubits. It requires some careful design to do this in an efficient and scalable way. For detailed circuit diagrams, we refer to [49] and [14].

6.2.4 Quantum channels

Errors in a quantum device usually happen randomly, and therefore they can take a quantum system from a pure state to a mixed state. An alternative viewpoint is that errors cause loss of information, and loss of quantum information can change a pure state into a mixed state. This means that errors are not unitary operations. However, they can be described in the circuit model as the interaction between a system and an external environment. In general, the description will be in the following form:

1. Starting from a quantum system A that stores the quantum information we are interested in, add another system E representing the environment, conventionally initialized in a fixed state $|0\rangle_E$.
2. Apply a unitary circuit U on the composite system AE.
3. Discard the system E.

This process can also be written as $\rho'_A = \mathrm{Tr}_E[U(\rho_A \otimes |0\rangle\langle 0|_E)]$.

We note that such a process can be used to describe more than just errors. For example, the entire error correction process, where we measure the error syndrome through ancilla qubits and apply Pauli gates according to the measurement outcomes, can be described in this form too. In fact, this process can be slightly generalized to give the definition of a *quantum channel*.

Definition 6.3 (Quantum channel)**.** A *quantum channel* $\mathcal{C}$ is a mapping from $d_A \times d_A$ density matrices to $d_{A'} \times d_{A'}$ density matrices that can be written in the form

$$\mathcal{C}(\rho_A) = \mathrm{Tr}_{E'}[U(\rho_A \otimes |0\rangle\langle 0|_E)].$$

Where AE and $A'E'$ are two decompositions of the same composite quantum system.

As can be seen, the generalization is that the system E' to discard is not necessarily the system E added in the beginning. This allows quantum channels to represent operations that change the dimension of the system, such as the encoding process and the decoding process of a quantum code.

From the definition, it can also be seen that a quantum channel $\mathcal{C}$ is a *linear* mapping from $d_A \times d_A$ matrices to $d_{A'} \times d_{A'}$ matrices. As such, it can be represented as a $d_A \times d_A \times d_{A'} \times d_{A'}$ tensor, conventionally written as a $d_A d_{A'} \times d_A d_{A'}$ matrix known as the *Choi matrix* of $\mathcal{C}$. For example, when $d_A = d_{A'} = 2$, the Choi matrix of $\mathcal{C}$ is the following block matrix:

$$C = \left(\begin{array}{c|c} \mathcal{C}\begin{pmatrix}1 & 0\\ 0 & 0\end{pmatrix} & \mathcal{C}\begin{pmatrix}0 & 1\\ 0 & 0\end{pmatrix} \\ \hline \mathcal{C}\begin{pmatrix}0 & 0\\ 1 & 0\end{pmatrix} & \mathcal{C}\begin{pmatrix}0 & 0\\ 0 & 1\end{pmatrix} \end{array}\right).$$

Pauli transfer matrices. There are other ways of representing a quantum channel. For quantum channels that apply to a single qubit, i.e. when $d_A = d_{A'} = 2$, one such representation is the *Pauli transfer matrix*, which has the advantage of being a real matrix, and also makes some quantities of interest easier to compute.

The Pauli transfer matrix representation is based on the fact that the single-qubit Pauli operators $\{I, X, Y, Z\}$ form a basis for the linear space of 2×2 matrices. Indeed, if we multiply each of them by $1/\sqrt{2}$, and define the inner product of two matrices A and B as $\mathrm{Tr}(A^\dagger B)$, then they form an *orthonormal* basis. As such, the Pauli transfer matrix R for a quantum channel $\mathcal{C}$ can be defined as

$$\begin{pmatrix} R_{00} & R_{01} & R_{02} & R_{03} \\ R_{10} & R_{11} & R_{12} & R_{13} \\ R_{20} & R_{21} & R_{22} & R_{23} \\ R_{30} & R_{31} & R_{32} & R_{33} \end{pmatrix} = \frac{1}{2} \begin{pmatrix} \mathrm{Tr}[\mathcal{C}(I)] & \mathrm{Tr}[\mathcal{C}(X)] & \mathrm{Tr}[\mathcal{C}(Y)] & \mathrm{Tr}[\mathcal{C}(Z)] \\ \mathrm{Tr}[X\mathcal{C}(I)] & \mathrm{Tr}[X\mathcal{C}(X)] & \mathrm{Tr}[X\mathcal{C}(Y)] & \mathrm{Tr}[X\mathcal{C}(Z)] \\ \mathrm{Tr}[Y\mathcal{C}(I)] & \mathrm{Tr}[Y\mathcal{C}(X)] & \mathrm{Tr}[Y\mathcal{C}(Y)] & \mathrm{Tr}[Y\mathcal{C}(Z)] \\ \mathrm{Tr}[Z\mathcal{C}(I)] & \mathrm{Tr}[Z\mathcal{C}(X)] & \mathrm{Tr}[Z\mathcal{C}(Y)] & \mathrm{Tr}[Z\mathcal{C}(Z)] \end{pmatrix}.$$

Note that we omitted the complex conjugate because I, X, Y, Z are all Hermitian. When referring to individual entries of R, it is conventional to index the rows and columns with $0, 1, 2, 3$ instead of $1, 2, 3, 4$, as shown above.

A simple property of Pauli transfer matrices follows from the fact that any valid quantum channel must be *trace-preserving*, since they should map normalized density matrices to normalized density matrices. Even though I, X, Y, Z are not normalized density matrices, the linearity of the quantum channel means that their traces are still preserved, so

$$\mathrm{Tr}[\mathcal{C}(I)] = \mathrm{Tr}(I) = 2,$$

$$\mathrm{Tr}[\mathcal{C}(X)] = \mathrm{Tr}[\mathcal{C}(Y)] = \mathrm{Tr}[\mathcal{C}(Z)] = \mathrm{Tr}(X) = \mathrm{Tr}(Y) = \mathrm{Tr}(Z) = 0.$$

Therefore, the first row of a Pauli transfer matrix for a valid quantum channel must be $(R_{00}, R_{01}, R_{02}, R_{03}) = (1, 0, 0, 0)$.

6.3 Base error model

The realistic error model we use in our simulation experiments is mostly based on the one in [14], described in their supplementary material. It is intended to describe the errors that happen in a superconducting quantum device. Whereas [14] uses Pauli transfer matrices to represent error channels, we will use Choi matrices, both for the sake of variety and because they are closer to the tensor representations used in our program.

6.3.1 Idle error

Even when a qubit is idling, i.e. when we do not intend to apply any gate or measurement to the qubit, interaction with the environment will cause an error on the state of the qubit, which is the very reason why we need to do error correction just to preserve the value of a logical qubit. This error is described by the *amplitude-phase damping model*. As the name suggests, it consists of two components:

- Amplitude damping, which causes the amplitude of the excited state $|1\rangle_A$ to decay over time. This happens because a qubit in the state $|1\rangle_A$ can emit a photon into the environment, and relax to the ground state $|0\rangle_A$. Suppose that over a timespan of length t, the emission happens with probability p_1. Then the amplitude damping channel can be modeled as follows:

    1. Add an ancillary qubit $|0\rangle_E$.

2. Apply a unitary operation U to the system AE, such that

$$U|00\rangle_{AE} = |00\rangle_{AE}, \qquad U|10\rangle_{AE} = \sqrt{1-p_1}|10\rangle_{AE} + \sqrt{p_1}|01\rangle_{AE}.$$

3. Discard the ancilla E.

One interpretation of this model is that $|0\rangle_E$ is the vacuum state of the environment electromagnetic field, $|1\rangle_E$ is the environment state with a photon emitted, and whether a photon is emitted is determined by the unitary process U.

- Phase damping, which causes the qubit to *dephase*, i.e. lose information about the relative phase between the states $|0\rangle_A$ and $|1\rangle_A$. Suppose that over a timespan of length t, complete dephasing happens with probability p_ϕ. Then the phase damping channel can be modeled as follows:

  1. Add an ancillary qubit $|0\rangle_E$.
  2. Apply a unitary operation U to the system AE, such that

  $$U|00\rangle_{AE} = |00\rangle_{AE}, \qquad U|10\rangle_{AE} = \sqrt{1-p_\phi}|10\rangle_{AE} + \sqrt{p_\phi}|11\rangle_{AE}.$$

  3. Discard the ancilla E.

  This can be interpreted as describing interactions between the qubit and the environment that are too weak to flip the qubit from one energy eigenstate to another, but strong enough to change the environment state depending on the qubit state.

In the Choi matrix representation:

$$C_{\Lambda_{T_1}} = \left(\begin{array}{cc|cc} 1 & 0 & 0 & \sqrt{1-p_1} \\ 0 & 0 & 0 & 0 \\ \hline 0 & 0 & p_1 & 0 \\ \sqrt{1-p_1} & 0 & 0 & 1-p_1 \end{array}\right), \quad C_{\Lambda_{T_\phi}} = \left(\begin{array}{cc|cc} 1 & 0 & 0 & \sqrt{1-p_\phi} \\ 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ \sqrt{1-p_\phi} & 0 & 0 & 1 \end{array}\right).$$

For each of the two processes, the probabilities that an error *does not* happen, $1-p_1$ and $1-p_\phi$ respectively, decay exponentially over time:

$$1 - p_1 = e^{-t/T_1}, \qquad 1 - p_\phi = e^{-t/T_\phi}.$$

This ensures that each satisfy $\Lambda(t_1) \circ \Lambda(t_2) = \Lambda(t_1 + t_2)$. Furthermore, the two channels Λ_{T_1} and Λ_{T_ϕ} commute with each other, so $\Lambda_{T_1}(t) \circ \Lambda_{T_\phi}(t) = \Lambda_{T_\phi}(t) \circ \Lambda_{T_1}(t)$ is the channel that describe the effect of both over a timespan of length t.

Quantum operation with a duration. We note that, even though we call the amplitude-phase damping channel the "idle error", in our error model, it is actually applied to the qubit all the time, no matter whether the qubit is idling. In fact, a quantum operation with duration t is usually modeled as follows:

1. Idle for a timespan of length $t/2$.
2. Apply the quantum operation instantaneously.
3. Idle for a timespan of length $t/2$.

In other words, the operation is modeled as happening at the middle point of its time interval, and the amplitude-phase damping channel is applied before and after the operation.

6.3.2 Unitary gate errors

As mentioned above, even though the gates used in the syndrome extraction circuit have non-zero durations, we model them as instantaneous, and apply the "idle error" channel before and after it. However, each gate also has a gate-specific error depending on how the gate is applied physically, which is modeled as happening at the same time point as the gate itself.

There are two different kinds of gates in the syndrome extraction circuit, the $R_y(\pm\pi/2)$ gates and the CZ gates. Below we will give the gate-specific error channel for them.

$R_y(\pm\pi/2)$ **gates.** The gate-specific error of $R_y(\pm\pi/2)$ gates is modeled as a *depolarizing noise*, meaning that an X, Y, or Z gate is applied to the qubit with a certain probability for each. It is conventional to quantify the strength of a depolarization channel using its effect on the expectation values of the Pauli operators. The gate-specific error of $R_y(\pm\pi/2)$ shrinks the expectation values of X and Z by a factor of $1 - p_{xz}$, and the expectation value of Y by a factor of $1 - p_y$, where $p_{xz} > p_y$. In Choi matrix representation[3]:

$$C_{\text{dep}} = \left(\begin{array}{cc|cc} 1 - p_{xz}/2 & 0 & 0 & 1 - p_{xz}/2 - p_y/2 \\ 0 & p_{xz}/2 & -p_{xz}/2 + p_y/2 & 0 \\ \hline 0 & -p_{xz}/2 + p_y/2 & p_{xz}/2 & 0 \\ 1 - p_{xz}/2 - p_y/2 & 0 & 0 & 1 - p_{xz}/2 \end{array}\right).$$

[3]In [14], p_{xz} and p_y are named p_{plane} and p_{axis} respectively, referring to the x-z plane and the y axis of the Bloch sphere.

CZ gates. In [14], the gate-specific error of CZ gates is modeled as a unitary phase error that is "quasi-static", meaning that the magnitude of the phase error on each adjacent qubit pair is regarded as constant in a single run of the experiment, but will vary randomly over multiple runs.

Since the quasi-static error model is inconvenient for our approach that aims to do exact simulation, we choose not to incorporate it in our error model. This choice is also justified because the error is modeled as unitary, so it can be compensated for with techniques like rapid qubit calibrations [50] or coherent error cancellations [51].

6.3.3 Measurement error

In a superconducting quantum device, measurement is implemented by introducing photons into a readout resonator, which will dephase the qubit completely and yield a measurement result. Then, the photon must be allowed to deplete from the resonator by waiting for some time before the qubit can be used again.

In [14], measurement is modeled as a "butterfly gate", where the joint probability of each classical outcome in $\{0, 1\}$ and each output qubit state in $\{|0\rangle, |1\rangle\}$ is given for each input qubit state in $\{|0\rangle, |1\rangle\}$. However, they also note that the experimentally observed parameters are explained well by the following simple assumptions:

- The measurement happens instantaneously at the middle point of the measurement period, and amplitude-phase damping applies before and after it.
- The classical outcome is further subject to a declaration error ϵ_{RO}, which is itself independent of the outcome.

In our experiments, for simplicity, we use this model instead of the "butterfly gate" model for measurement operations in the circuit.

The effect of leftover photons. During the photon depletion period, the amount of photons in the resonator decays over time, but does not decrease to a level that is completely negligible. When the qubit is put into a superposition state of $|0\rangle$ and $|1\rangle$ again, those leftover photons will cause the qubit to slightly dephase. The underlying mechanics for this effect is fairly complex, so we just use the expression given in [14]:

$$p_{\phi,\text{photon}} = \exp\left(2\chi\alpha(0)\exp(\kappa(t_m - t_g))\left[\frac{e^{-\kappa t}}{4\chi^2+\kappa^2}[-\kappa\sin(2\chi t) - 2\chi\cos(2\chi t)]\right]_{t_1-t_g}^{t_2-t_g}\right),$$

where $[t_1, t_2]$ is the time period during which this dephasing applies, t_m is the time at which the measurement starts, t_g is the time of the first $R_y(-\pi/2)$ gate after the measurement, and κ and χ are constant parameters.

The value $p_{\phi,\text{photon}}$ has the same meaning as p_ϕ in the phase damping channel Λ_{T_ϕ} described in Section 6.3.1. Therefore, in order to take the effect of leftover photons into account, we use a similar channel with $p_{\phi,\text{photon}}$ in place of p_ϕ, *in addition to* the amplitude-phase damping channel for that time period.

6.4 Modeling crosstalk

6.4.1 Overview of crosstalk

In the context of quantum computation, *crosstalk* [52] refers to any unwanted coupling between qubits. Obviously, coupling between adjacent qubits is needed to implement CZ gates, but in an ideal world, we should be able to turn this coupling on and off at will. That is not the case in practice, and indeed, in this chapter we will focus on the crosstalk between adjacent qubits when no CZ gate is being applied to them.

Since CZ is the native two-qubit gate on superconducting quantum qubits, it is not a surprise that the form of the crosstalk channel is related to the CZ gate. In fact, the crosstalk in our error model originates from the ZZ coupling between adjacent qubits, and has the form

$$e^{ik(Z\otimes Z)} = e^{ik(|00\rangle\langle 00|+|01\rangle\langle 01|+|10\rangle\langle 10|+|11\rangle\langle 11|)} = e^{ik(-I\otimes I+Z\otimes I+I\otimes Z+4|11\rangle\langle 11|)}.$$

The right-hand side decomposes $e^{ik(Z\otimes Z)}$ into four factors that commute with each other. Among them, e^{-ik} is a global phase and can be ignored, $e^{ik(Z\otimes I)}$ and $e^{ik(I\otimes Z)}$ are both single-qubit gates (known as PHASE gates), and $e^{4ik|11\rangle\langle 11|}$ is known as a CPHASE gate, which the CZ gate is a special case of, as $CZ = e^{i\pi|11\rangle\langle 11|}$.

It is also possible to ignore the PHASE gates by taking them into account during calibration. In fact, because of the energy difference between $|0\rangle$ and $|1\rangle$, superconducting qubits are constantly undergoing a change of relative phase between them, which would manifest as a PHASE gate. In calibrating the frequencies of the qubits, we account for this phase rotation, and apply the gates that do not commute with it ($R_y(\pm\pi/2)$ gates) with specific timings so that they do what they need to do. If we calibrate each qubit while other qubits are in the ground state $|0\rangle$, then we are essentially assuming that crosstalk does not affect the qubit when other qubits are in the state $|0\rangle$, which means that we should model the crosstalk error as CPHASE gates rather than $e^{ik(Z\otimes Z)}$.

Of course, when calibrating the CZ gates, it is easy to also calibrate the crosstalk strength, and adjust the qubit frequencies so that the $e^{ik(Z\otimes Z)}$ model is applicable. Regardless, in this chapter, we will stick to the CPHASE model.

6.4.2 Crosstalk in the circuit model

Like amplitude-phase damping described in Section 6.3.1, crosstalk should affect all pairs of adjacent qubits at all time. In fact, we will start from the assumption that the crosstalk strength is a constant, independent of the qubit pair and of time. This is likely not the case in practice because circumstances like the frequency of both qubits would affect the crosstalk strength, but it works as a rough model.

However, unlike the phase damping channel, the CPHASE gate (or $e^{ik(Z\otimes Z)}$) does not commute with the amplitude damping channel. Therefore, in order to even just simulate the effect of both on a pair of idling qubits, we would need to make use of the Trotter approximation or something similar. Doing this on the whole syndrome extraction circuit would leave us with a very complex tensor network that probably cannot be feasibly evaluated. Therefore, we have to discretize the effect of crosstalk, even at the cost of simulation accuracy.

To this end, we note that the CPHASE gate is a diagonal gate, so it does commute with other CPHASE gates and CZ gates regardless of the configuration. While it does not commute with amplitude-phase damping, the effect of amplitude-phase damping is itself small, so the commutator of it and crosstalk is a “second-order inaccuracy” that we may be able to ignore. The only component in the circuit that we absolutely cannot commute a CPHASE gate through is the $R_y(\pm\pi/2)$ gates. Therefore, as a first-order approximation, we can move CPHASE gates anywhere within regions delineated by $R_y(\pm\pi/2)$ gates. (We also ignore any crosstalk that happens *while* applying the $R_y(\pm\pi/2)$ gates.)

In the circuit of a surface code, data qubits are only adjacent to ancilla qubits, and vice versa. Furthermore, for each pair of adjacent qubits, the $R_y(\pm\pi/2)$ gates divide the circuit into two types of alternating regions:

- CZ regions, during each of which one CZ gate is applied to that qubit pair, and more CZ gates are applied to the data qubit and other ancilla qubits of the same type (X-ancilla or Z-ancilla).

- Measurement regions, during each of which the ancilla qubit is measured, and CZ gates are applied to the data qubit and ancilla qubits of the other type.

In those regions, crosstalk can be handled in different ways.

- In CZ regions, we do not incorporate crosstalk into the circuit at all, because as a first-order approximation all crosstalk can be moved to the same time point as the CZ gate. We can therefore calibrate the CZ gates in such a way to compensate for all the crosstalk in the CZ region that the gate is in. (In fact, when calibrating CZ gates normally, we probably will already account for the crosstalk that happens during the CZ gate itself; we just need to "overcompensate" so that the rest of the crosstalk in the same CZ region is also accounted for.)

- In measurement regions, we move all crosstalk to the same time point, which we choose to be the end of the measurement region, i.e. just before the next round of $R_y(\pm\pi/2)$ gates are applied (more precisely, just before the amplitude-phase damping channel associated with the duration of the $R_y(\pm\pi/2)$ gates). This way, all the crosstalk involving the same ancilla qubit (but different data qubits) are moved to the same time point, which simplifies the tensor network we will need to simulate since the amplitude-phase damping channel on the ancilla qubit is only split into two parts. Furthermore, there will be one edge in the tensor network associated with many tensors, which may help slicing to decrease the contraction cost.

In summary, at the cost of simulation accuracy, we reduce the number of CPHASE gates that represent crosstalk to one per syndrome extraction round, per adjacent pair of qubits. In Section 6.6.2, we will estimate the magnitude of the inaccuracy caused by this approach.

6.5 Simulating the surface code with tensor networks

6.5.1 Simulating noisy circuits

When we take the error model into account, some unitary gates as well as "idle wires" in the syndrome extraction circuit become non-unitary channels. Simulating such a noisy quantum "circuit" is slightly different from simulating an ideal quantum circuit, as we have done in Chapter 4.

It is still possible to represent a noisy "circuit" as a quantum circuit in the strict sense as defined in Section 2.2.3, since every quantum channel can be described as a process of "add ancilla, apply unitary, discard ancilla". However, a circuit generated in this way would have many measurements of ancilla qubits representing the environment. Since the method in Section 4.3 can only do single-amplitude simulation, we would need to assign *a priori* values to the measurement results of those ancilla qubits (as well as ancilla qubits in the surface code itself) before we could calculate the probability of those measurement results or anything else. Even if we have a way of sampling those measurement results (probably with a method like the one in Section 5.4), anything that we care about (in our case, the logical error rate) would still be subject to the variance caused by sampling, and it would take many samples to get a reliable result.

Therefore, in this chapter we take a different approach, in that we will do a *density matrix simulation* instead of a state vector simulation. This will certainly lead to a tensor network with a higher contraction width and cost, since the number of entries in a density matrix is the *square* of the number of entries in the corresponding state vector. However, the density matrix also contains all information about classical randomness, so we would no longer need to worry about sampling. This method allows us to take full advantage of our parallel tensor network contraction algorithm.

Compared to the simulation method in Section 4.3, a density matrix simulation is different in the following ways:

- Each input qubit becomes an order-2 tensor corresponding to the density matrix ρ of the initial state, instead of the state vector $|\psi\rangle$.
- Each quantum channel applied to k qubits becomes an order-$4k$ tensor corresponding to the $d^2 \times d^2$ Choi matrix of the channel, where $d = 2^k$.
- All edges (including closed edges and open edges) are doubled, so that they match the orders of the tensors they are connected to, but with one exception:
  - If we measure a qubit, then we identify the two edges corresponding to the wire the measurement is on, i.e. let the edge remain single, and connect it to each adjacent tensor *twice*. Furthermore, we make that edge an open edge in the final tensor network, representing the measurement result.

Some of the aforementioned tensors may represent pure objects, i.e. pure states and unitary gates. Such tensors can be further decomposed into the product of two tensors in order to simplify the tensor network. In particular, if we apply the above process to a circuit consisting entirely of pure objects, then the resulting tensor network would consist of two disjoint parts with identical structure that are the complex conjugate of each other. Intuitively, non-pure objects in the circuit connect those two parts together.

With the above procedure, we can generate a tensor network that represents the noisy syndrome extraction circuit as a quantum channel. The next section will discuss what we do with the inputs and the outputs.

6.5.2 Defining the logical channel with the optimal decoder

The ultimate goal of our surface code simulation is to compute the *logical channel* of the error correction process, which conceptually is a quantum channel applied to the underlying logical qubit while we try to protect it against errors. In order to rigorously define the logical channel, it is unavoidable to consider the encoding and decoding processes of the surface code.

However, we note that "encoding" and "decoding" are really just imaginary processes we use to describe the surface code. In fault-tolerant quantum computation, we would never actually handle a logical qubit, so it usually makes no sense to implement encoding or decoding. Instead, the qubits are fault-tolerantly initialized to fixed initial states such as $|0\rangle$, fault-tolerantly operated on, and fault-tolerantly measured to get classical results. The process we focus on here, *preserving* the value of a logical qubit, can be seen as a special case of a fault-tolerant operation, namely the identity operation.

Since our focus is how well the surface code preserves quantum information, rather than how to fault-tolerantly prepare and measure a logical qubit, we will define an encoding process and a decoding process, but in a simplistic way. In particular, we assume that both encoding and decoding are *error-free*.

Encoding. We first use the following (unrealistic) procedure to get $V|0\rangle$, the code state encoding the logical state $|0\rangle$[4]:

1. Initialize all data qubits in the state $|0\rangle$.
2. Measure the values of all four X-stabilizers in the syndrome, using part of the syndrome extraction circuit (but without error channels), and *assume that the results are all* $+1$, i.e., just take the corresponding block from the density matrix.
    - The probability of each X-stabilizer measurement yielding $+1$ is $1/2$, independent of the other measurement results. Therefore, the probability that all four measurements yield $+1$ is $1/16$.
3. Multiply the resulting density matrix by 16 so that it is normalized.

The resulting state must be a code state, because the value of all Z-stabilizers are already $+1$ on the all-$|0\rangle$ state, and the X-stabilizer measurements will not change this because the stabilizers all commute.

[4]Here V is an *isometry*, a matrix that is *not* a square matrix but satisfies $V^\dagger V = I$.

In order to fully characterize the encoding channel, we also need to know $V|1\rangle$, including the relative phase between $V|0\rangle$ and $V|1\rangle$. To this end, we make use of a *logical X operator*, which can be applied to a code state in order to achieve the same effect as applying an X gate to the underlying logical qubit. There are multiple such operators, and the one we use is $X_{(0,0)}X_{(2,0)}X_{(4,0)}$, where the qubits are labeled as in Figure 6.1.

In fact, we compute both $V|0\rangle$ and $V|1\rangle$ with the following trick:

1. Generate the state $V|0\rangle$ as above.
2. Add a new "dummy qubit" Q to the system, and initialize it to the state $|+\rangle = H|0\rangle = (|0\rangle + |1\rangle)/\sqrt{2}$.
3. Apply CX gates on the following qubit pairs: $(Q,(0,0))$; $(Q,(2,0))$; $(Q,(4,0))$. In each case, Q is the control qubit.

This procedure generates the state $|\Psi\rangle = |0\rangle_Q \otimes V|0\rangle + |1\rangle_Q \otimes V|1\rangle$, which is essentially a representation of the encoding channel. We can then take its density matrix $|\Psi\rangle\langle\Psi|$, and apply the noisy syndrome extraction circuit to it, leaving the qubit Q unchanged to represent the input of the channel.

Decoding. The decoding process is even more complicated than encoding process, because after undergoing a noisy syndrome extraction circuit, the state of the code qubits is likely to be no longer in the codespace. Furthermore, the syndrome extraction circuit will output error syndromes, which are supposed indicate what errors may have happened, so the decoder should make use of those to recover the logical qubit as well as possible.

In order to simplify the problem, we will first contrive a process to move the code qubits back into the codespace. To this end, after r rounds of the noisy syndrome extraction circuit, we do one more round of *error-free* syndrome extraction. If the results happen to be all $+1$, then we will know that the state is now in the codespace, and we can continue with the next step. Otherwise, we apply simple "default corrections" by applying Pauli gates to data qubits to change the value of each stabilizer in the syndrome to $+1$.

We design a "default correction" for each stabilizer P in the syndrome, which is a Pauli operator that anti-commutes with P, but commutes with all other stabilizers in the syndrome. For example, the "default correction" for the X-stabilizer $X_{(0,2)}X_{(0,4)}$ may be $Z_{(0,4)}$. (Again, refer to Figure 6.1 for the label of each qubit.) Such a Pauli operator, when applied, will flip only the value of P in the syndrome. Therefore, it suffices to do the "default correction" for each stabilizer in the syndrome with value -1.

We note that the "default correction" for each stabilizer is not unique, and it does not necessarily give the "correction" with the fewest gates, especially when there are multiple instances of -1 in the syndrome. However, as we have mentioned at the end of Section 6.2.2, if we restrict ourselves to "corrections" by Pauli gates, then all such "corrections" will differ by at most one Pauli gate on the logical qubit. Therefore, we will apply the "default corrections" for now, and maybe switch to a better correction after we have gotten a logical qubit.

Decoding a state ρ that is known to be in the codespace is relatively simple. Above we already have the state $|\Psi\rangle$ that represents the encoding process V, and it suffices to apply the reverse process: $\rho_L = V^\dagger \rho V$ is a state of the logical qubit that satisfies $V\rho_L V^\dagger = \rho$.

At this point, there are still two qubits in our system. One of them is of course the logical qubit L. The other one is the "input" qubit Q, which we introduced in the encoding process and have not touched since then. In addition, we also have $8(r+1)$ classical bits, which are measurement results from r rounds of noisy syndrome extraction plus one round of error-free syndrome extraction. Therefore, the total number of open edges in our tensor network is $8(r+1)+4$ — one for each classical bit and two for each qubit. *This is the tensor network we will actually evaluate with our parallel tensor network contraction algorithm.*

The value of the tensor network can be regarded as $2^{8(r+1)}$ matrices of size 4×4. Each 4×4 matrix ρ_s corresponds to a combination of error syndromes $s \in \{+1, -1\}^{8(r+1)}$, and is an *unnormalized density matrix* of the two-qubit system QL. Ideally, the sum of those matrices should be the density matrix $|\Phi^+\rangle\langle\Phi^+|$, where $|\Phi^+\rangle = (|00\rangle + |11\rangle)/\sqrt{2}$. However, due to the error indicated by the error syndromes, some of those matrices may be closer to one of $|\Psi^+\rangle\langle\Psi^+|$, $|\Psi^-\rangle\langle\Psi^-|$, or $|\Phi^-\rangle\langle\Phi^-|$, where

$$|\Psi^+\rangle = (I \otimes X)|\Phi^+\rangle, \qquad |\Psi^-\rangle = (I \otimes Y)|\Phi^+\rangle, \qquad |\Phi^-\rangle = (I \otimes Z)|\Phi^+\rangle.$$

Those cases mean that the error syndromes indicate that an X, Y, or Z error is likely to have occurred on the logical qubit, and the corresponding Pauli gate should be applied to correct it.

In order to measure the closeness between a density matrix ρ and a pure state $|\psi\rangle$, we use the *fidelity* between them defined as $F = \langle\psi|\rho|\psi\rangle$. Therefore, for each $s \in \{+1, -1\}^{8(r+1)}$, we choose a Pauli correction P_s from $\{I, X, Y, Z\}$ according to the highest fidelity among $\langle\Phi^+|\rho_s|\Phi^+\rangle$, $\langle\Psi^+|\rho_s|\Psi^+\rangle$, $\langle\Psi^-|\rho_s|\Psi^-\rangle$, $\langle\Phi^-|\rho_s|\Phi^-\rangle$. Pauli corrections chosen this way will maximize the *logical fidelity* $\langle\Psi^+|\rho|\Psi^+\rangle$, where

$$\rho = \sum_{s \in \{+1,-1\}^{8(r+1)}} P_s \rho_s P_s^\dagger.$$

We will call the list of Pauli matrices P_s the *optimal decoder*. The state ρ defined above represents the *logical channel* after applying the optimal decoder, which can be extracted easily.

To summarize, after r rounds of the noisy syndrome extraction circuit, we do the following to get the logical channel:

1. Do one last round of error-free syndrome extraction.
2. For each -1 in the syndrome, apply the "default correction" for that stabilizer to turn it into $+1$.
    - This can be done in the tensor network with controlled Pauli gates.
3. Apply the reverse of the encoding process V to get the logical state $\rho_L = V^\dagger \rho V$.
4. Evaluate the tensor network to get $2^{8(r+1)}$ unnormalized 4×4 density matrices ρ_s, where $s \in \{+1, -1\}^{8(r+1)}$.
5. Find the optimal decoder by choosing the best Pauli correction from $\{I, X, Y, Z\}$ according to the highest among $\langle \Phi^+ | \rho_s | \Phi^+ \rangle$, $\langle \Psi^+ | \rho_s | \Psi^+ \rangle$, $\langle \Psi^- | \rho_s | \Psi^- \rangle$, $\langle \Phi^- | \rho_s | \Phi^- \rangle$ for each $s \in \{+1, -1\}^{8(r+1)}$.
6. Find the logical channel from the two-qubit state $\rho = \sum_{s \in \{+1,-1\}^{8(r+1)}} P_s \rho_s P_s^\dagger$.

From the logical channel, it is easy to calculate logical error rates of various types of errors.

In the above process, the output of our tensor network is an order-$(8(r+1)+4)$ tensor, which greatly limits the number of rounds that can be simulated with this approach. In our experiments, we take $r = 2$, giving an order-28 tensor which can still be stored in the main memory of a single computer, and allows the final steps to be done relatively quickly. If we increase r to even 3, this will become an order-36 tensor that needs to be sliced to even fit in the memory, and the corresponding tensor network contraction task would be infeasible. Therefore, in our experiments, we will stick to $r = 2$.

Even though we only simulate two rounds of noisy syndrome extraction with $r = 2$, the decoder makes use of three rounds of error syndromes to determine how to correct the error, so in theory the decoder can be used to achieve fault-tolerant quantum computation. Since it can be said that our simulation is a two-round simulation *and* a three-round simulation in different senses, hereafter we will refer to it as a "$(2+1)$-round" simulation.

Category	Parameter	Symbol	Value
Scheduling	Single-qubit gate time	$T_{g,1\mathrm{Q}}$	20 ns
Scheduling	Two-qubit gate time	$T_{g,2\mathrm{Q}}$	40 ns
Scheduling	Coherent step time	τ_c	200 ns
Scheduling	Depletion time	τ_d	300 ns
Scheduling	Measurement time	τ_m	300 ns
Idle error	Qubit relaxation time	T_1	30 µs
Idle error	Qubit dephasing time	T_ϕ	60 µs
$R_y(\pm\pi/2)$ gates	In-axis rotation error	p_y	10^{-4}
$R_y(\pm\pi/2)$ gates	In-plane rotation error	p_{xz}	5×10^{-4}
Measurement error	Readout infidelity	ϵ_{RO}	0.15%
Measurement error	Photon relaxation time	$1/\kappa$	250 ns
Measurement error	Dispersive shift	χ/π	-2.6 MHz
Crosstalk	Crosstalk strength	k	0.03 to 0.05

Table 6.1: Parameters used in our surface code simulation.

6.6 Experiment results

In [13], we did experiments with the methods described in previous sections, with the goal of roughly assessing the effect of crosstalk to the performance of surface codes. We adopt all circuit parameters and error parameters used in [14]. The only newly introduced parameter is the strength k of the crosstalk in each measurement region, represented by the CPHASE gate $e^{4ik|11\rangle\langle 11|}$. In the experiments with crosstalk, we set k to a value between 0.03 and 0.05. Table 6.1 summarizes the values of the parameters used in our experiments.

6.6.1 Effect of crosstalk on the logical channel

We first qualitatively analyze the effect of crosstalk by fixing the value of the crosstalk strength k to 0.03. Figure 6.2 shows two logical channels represented as Pauli transfer matrices, one without crosstalk and one with crosstalk.

It can be clearly seen that the main difference between those two matrices is in the middle 2×2 sub-matrix. For the logical channel without crosstalk, R_{12} and R_{21} are both zero. Meanwhile, for the logical channel with crosstalk, $R_{12} \approx -R_{21}$ has a large absolute value. This indicates that a *coherent Z rotation* $R_z(\theta)$ is taking place, as the Pauli transfer matrix

$$R_{R_z(\theta)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & \sin(\theta) & 0 \\ 0 & -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 1.00 & -4.07\times10^{-18} & 0.00 & -9.42\times10^{-18} \\ 2.88\times10^{-7} & 9.92\times10^{-1} & 0.00 & -2.29\times10^{-13} \\ 0.00 & 0.00 & 9.85\times10^{-1} & 0.00 \\ 6.14\times10^{-6} & -2.31\times10^{-13} & 0.00 & 9.92\times10^{-1} \end{pmatrix}$$

(a) without crosstalk

$$\begin{pmatrix} 1.00 & -2.44\times10^{-18} & -5.37\times10^{-20} & 5.96\times10^{-18} \\ 3.47\times10^{-7} & 9.88\times10^{-1} & 2.65\times10^{-3} & 2.99\times10^{-8} \\ 4.12\times10^{-9} & -2.63\times10^{-3} & 9.80\times10^{-1} & -7.43\times10^{-6} \\ 8.89\times10^{-6} & -6.75\times10^{-10} & 7.95\times10^{-6} & 9.91\times10^{-1} \end{pmatrix}$$

(b) with crosstalk

$$\begin{pmatrix} 0.00 & -1.62\times10^{-18} & 5.37\times10^{-20} & -1.54\times10^{-17} \\ -5.86\times10^{-8} & 4.43\times10^{-3} & -2.65\times10^{-3} & -2.98\times10^{-8} \\ -4.12\times10^{-9} & 2.63\times10^{-3} & 4.84\times10^{-3} & 7.43\times10^{-6} \\ -2.74\times10^{-6} & 6.75\times10^{-10} & -7.95\times10^{-6} & 6.54\times10^{-4} \end{pmatrix}$$

(c) difference between (a) and (b)

Figure 6.2: Comparison of logical Pauli transfer matrices for $2 + 1$ rounds of Surface-17 error correction, without and with crosstalk. The crosstalk strength k is set to 0.03. We note that the first row is supposed to be $(1, 0, 0, 0)$ for all Pauli transfer matrices, but here we show the actual results output by our program, in order to demonstrate that the numerical error is negligible.

has the same properties.

In the above matrix, setting $\sin(\theta) \approx 2.64 \times 10^{-1}$, we get $\cos(\theta) \approx 1 - 3.51 \times 10^{-6}$, which does not explain the large difference between the Pauli transfer matrices at entries R_{11} and R_{22}. Therefore, there is also an extra *dephasing channel* applied to the logical qubit, which decreases R_{11} and R_{22}.

Most of the difference between the Pauli transfer matrices can be explained by the coherent Z-rotation and the dephasing. In particular, the difference between the Pauli transfer matrices at R_{33} is one order of magnitude smaller than the differences in the middle 2×2 sub-matrix as well as $1-R_{33}$, which means that crosstalk only slightly contributes to logical bit flip errors.

Next, we study the effect of the crosstalk strength k on the logical channel. We quantify the three types of errors mentioned above as follows:

- The probability of a *phase flip* is defined as $(1 - R_{11})/2$. (Here we divide by 2 because a phase flip channel with probability 1 would have $R_{11} = -1$.)

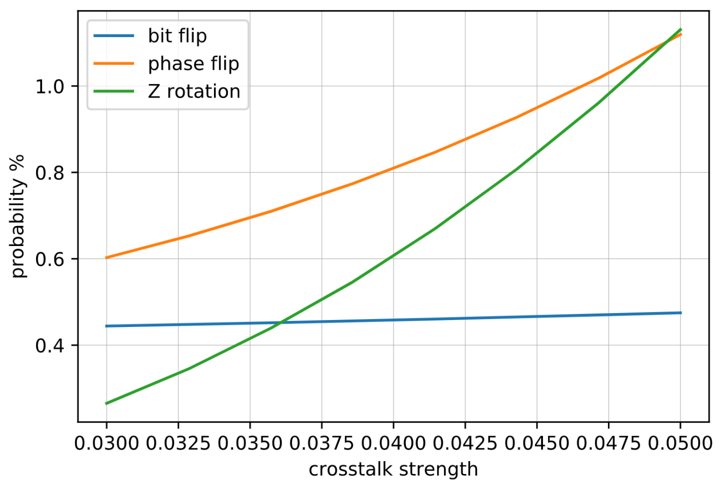


Figure 6.3: Logical error rates of various types as a function of the crosstalk strength.

- The probability of a *bit flip* is defined as $(1 - R_{33})/2$.
- The amount of a *coherent Z rotation* is defined as $|R_{12} - R_{21}|/2$.

We vary the crosstalk strength k from 0.03 to 0.05, and plot the probability or amount of each logical error type in Figure 6.3. The results agree with the above findings: The two types of errors that increase the most with the crosstalk strength are coherent Z rotation and phase flip errors, while the probability of a logical bit flip almost does not change with the crosstalk strength.

Interestingly, despite the apparent symmetry between X and Z in the design of the surface code, the effect of crosstalk on phase flip errors and bit flip errors turns out to be very different. It remains to be seen what the cause of this asymmetry is.

6.6.2 Estimating the discretization inaccuracy

In Section 6.4.2, we discretized the effect of crosstalk into instantaneous CPHASE gates by noting that the commutator of amplitude-phase damping and crosstalk is only a "second-order inaccuracy". In this section, we give numerical evidence that this inaccuracy is indeed negligible compared to the logical error rate.

The discretization we used can be thought of as moving an infinite number of infinitesimal CPHASE gates in a region all to the same time point. Intuitively, moving a CPHASE gate within a CZ region should cause an inaccuracy that is approximately proportional with the distance of the movement. Therefore, we choose a qubit pair where the CZ gate is applied early in the CZ region, move the CPHASE gate which has been combined with the CZ gate to the end of the CZ region, and evaluate the difference of the logical channel caused

$$\begin{pmatrix} 1.00 & -1.22\times10^{-17} & -1.57\times10^{-21} & 8.56\times10^{-17} \\ 9.90\times10^{-8} & 9.99\times10^{-1} & -3.24\times10^{-6} & -2.19\times10^{-17} \\ -5.52\times10^{-11} & 3.17\times10^{-6} & 9.98\times10^{-1} & 1.52\times10^{-21} \\ 9.50\times10^{-8} & -1.24\times10^{-11} & -6.29\times10^{-21} & 9.99\times10^{-1} \end{pmatrix}$$

(a) without moving the error

$$\begin{pmatrix} 1.00 & -1.49\times10^{-17} & -2.90\times10^{-21} & -3.31\times10^{-17} \\ 9.90\times10^{-8} & 9.99\times10^{-1} & -3.24\times10^{-6} & -4.91\times10^{-14} \\ -5.52\times10^{-11} & 3.17\times10^{-6} & 9.98\times10^{-1} & 6.04\times10^{-10} \\ 9.50\times10^{-8} & -1.24\times10^{-11} & -6.11\times10^{-10} & 9.99\times10^{-1} \end{pmatrix}$$

(b) with the error moved

$$\begin{pmatrix} 8.88\times10^{-16} & -2.67\times10^{-18} & -1.33\times10^{-21} & -1.19\times10^{-16} \\ 1.18\times10^{-13} & -8.57\times10^{-9} & -1.90\times10^{-12} & -4.91\times10^{-14} \\ 9.01\times10^{-17} & 1.01\times10^{-12} & -2.21\times10^{-8} & 6.04\times10^{-10} \\ 4.15\times10^{-13} & -3.43\times10^{-16} & -6.11\times10^{-10} & -1.53\times10^{-8} \end{pmatrix}$$

(c) difference between (a) and (b)

Figure 6.4: Comparison of logical Pauli transfer matrices for $1+1$ rounds of Surface-17 error correction, without and with one of the CPHASE gates moved to a different place within the same CZ region. The crosstalk strength k is set to 0.03, and the CPHASE gate moved is between the qubits $(3,3)$ and $(2,4)$. It was moved from the time point the corresponding CZ gate is at to the end of the CZ region.

by the movement. This difference can be seen as an upper bound of the inaccuracy caused by discretizing the crosstalk in that CZ region on that qubit pair, because the CPHASE gate is actually composed of an infinite number of infinitesimal parts, and none of the parts needed to move a longer distance during discretization.

Figure 6.4 shows the result of one such experiment, where the qubit pair chosen is $(3,3)$ and $(2,4)$ (see Figure 6.1 for the labeling). Since moving the CPHASE gate makes the tensor network slightly more complicated, in order to ensure that we can evaluate the tensor networks, we only simulate $1+1$ rounds of syndrome extraction in this experiment.

As shown in Figure 6.4, the difference caused by moving the CPHASE gate is on the order of 10^{-8}. Even if multiplied by the number of qubit pairs and the number of regions, such a small inaccuracy is negligible compared to the logical error caused by crosstalk, which is on the order of 10^{-3}. This shows that our approximation, as well as the proposal to compensate for the crosstalk in the CZ region when calibrating CZ gates, is indeed valid.

6.7 Discussion

Our experiment results show that the effect of crosstalk to the performance of surface codes can be significant. In particular, a large coherent Z-rotation is applied to the logical qubit, which does not exist at all in the logical channel without crosstalk. The probability of a logical phase flip error also increases from about 0.4% to about 0.6%. Therefore, when evaluating the prospect of fault-tolerant computation on near-term quantum devices, it would be necessary to take crosstalk into account.

Fortunately, the error caused by the type of crosstalk considered in this chapter is coherent (i.e. unitary), similar to the quasi-static error of CZ gates, so it can be compensated for with similar techniques. Indeed, we have already shown that, in CZ regions, crosstalk can be compensated for by combining them with CZ gates applied on the same qubit pair, with only a small second-order error. Crosstalk in measurement regions can potentially be compensated for by additional CZ gates, or by resetting the ancilla qubits to $|0\rangle$ after measurement, etc. It may also be possible to correct the coherent effect on the logical qubit directly by a logical operation, if the rotation angle could be computed with a quantitative model.

An interesting by-product of the work in this chapter is an "optimal decoder" for $2 + 1$ rounds of Surface-17 on this error model. Admittedly, the value of such a decoder is dubious because the last round of syndrome extraction is modeled as error-free, possibly making the decoder place more importance on it and less importance on the first two rounds. It may also be argued that in practice, a good decoder should combine information from more than three rounds of syndrome measurements. Still, it is possible that decisions made by such a decoder reflect general principles that can improve a decoder against a realistic error model, and such principles may be found by analysis and guide the design of other types of decoders.

In any case, the work in this chapter demonstrates again the flexibility of the tensor network model and our parallel tensor network contraction algorithm. In general, the behaviour of quantum circuits with non-pure components (like amplitude-phase damping) is an interesting and potentially useful problem. Even if the exponential space and time complexities of tensor network contraction restrict it to small problem sizes, it can serve as a convenient research tool to find patterns that may generalize to larger cases.

CHAPTER 7

Summary and conclusions

In this thesis, we have introduced a parallel tensor network contraction algorithm based on tensor network slicing. We applied it to various classical simulation tasks, with different settings and goals, that arise in quantum computation. Below, we summarize each major part of this thesis.

7.1 A parallel tensor network contraction algorithm

7.1.1 Summary

In Chapter 3, we looked at the core of our parallel tensor network contraction algorithm, which is the contraction scheme finding procedure. We identified three components of the procedure: initial contraction order finding, local optimization, and dynamic slicing. Both initial contraction order finding and dynamic slicing are essential to our algorithm, and repeated local optimization helps dynamic slicing to reach its maximum potential.

Within each component, there are various design choices that can affect the performance of the whole algorithm. In particular, we described two different methods to find initial contraction orders, one based on the treewidth and one based on hypergraph partitioning. The latter method has a recursive structure, but the top-level problem is significantly different from the lower-level problems, which inspired us to use different parameters for the top level and the lower levels when hyper-optimizing the parameters.

7.1.2 Future work

Hardware-oriented optimizations. In the current implementation of our algorithm, we specifically optimized the contraction scheme for GPU because the GPU does floating-point operations much faster than the CPU, and also to compare performance with other algorithms that has been benchmarked on GPU. However, the FLOPS efficiency on GPU, as benchmarked in Chapter 4, still leaves much to be desired. This can likely be improved with a deeper understanding of the time cost on GPU, and also by making use of advanced GPU functionalities such as Nvidia Tensor Cores.

Moreover, there are still many computers and clusters on which GPU is not available, so it would also be meaningful to optimize the contraction process for CPU. There are many parameters, like BLAS kernel sizes and cache sizes, on which the CPU differs from the GPU, and in particular something as simple as transposing one of the input tensors may greatly affect the time needed for a contraction step.

Approximate contraction. A more ambitious, but potentially useful, feature we could try to support is approximate contraction. Approximate contraction algorithms are commonly used for "natural" physical systems which gives rise to tensor networks with special structures and/or properties (e.g. translation invariance), but there has not been much research on adapting them to generic tensor networks. Hopefully, even though the core contraction algorithm may be very different, the idea of parallelizing by slicing edges can still apply.

7.2 Classical simulation of quantum supremacy circuits

7.2.1 Summary

In Chapter 4, we benchmarked our parallel tensor network contraction algorithm on a simulation task for quantum supremacy circuits, which are designed to be as hard to classically simulate as possible. Frugal rejection sampling was used to reduce true simulation to either single-amplitude simulation or batched-amplitude calculation, the latter reduction being more efficient for tensor network based methods.

We compared the running times with other similar methods, and the results demonstrated the advantage of our contraction scheme finding procedure. In particular, on 20-cycle Sycamore circuits, we achieved an estimated running time about 100 to 1000 times lower than other known tensor network based methods, even with a FLOPS efficiency lower than 15%.

7.2.2 Future work

Dynamically choosing open edges. Batched amplitude simulation with open tensor networks can reduce the overhead of rejection sampling, but open tensor networks themselves can incur a little overhead compared to closed tensor networks. In the context of rejection sampling, this overhead can be minimized by a good choice of the set of open edges.

In this thesis, we chose open edges both by following precedents [2] and by manually inspecting the circuit structure. It may be useful if this can instead be done automatically and dynamically, which would reduce human work that needs to be done for each class of random circuit, and may also find better choices of open edges than humans could.

Better way to take advantage of the low fidelity requirement. The current supremacy task only requires sampling the circuits at a relatively low fidelity (1.3% to 0.2%) which the quantum device could achieve, as estimated with cross entropy benchmarking. Currently, we restrict ourselves to sampling a mixture of the uniform distribution and the ideal distribution (disregarding the small error caused by rejection sampling), which means that a low fidelity requirement would only linearly reduce the running time.

However, there is no convincing evidence that the output distribution of the quantum device is such a mixture, therefore for fairness, the classical distribution should also be allowed to be any distribution with the same fidelity. It would be interesting to see whether we can use this extra freedom to design a sampling procedure that further reduces the running time.

7.3 Applications to the Quantum Approximate Optimization Algorithm

7.3.1 Summary

In Chapter 5, we applied our parallel tensor network contraction algorithm to the Quantum Approximate Optimization Algorithm (QAOA) to classically evaluate the energy function. Using the "lightcone trick", we were able to evaluate the energy function in linear or even constant time with regard to the problem size, for problems with a sparse and/or regular structure, and a low number of layers p. This allowed us to experiment with various methods to classically optimize the energy function.

We also tried to classically sample the output distribution of QAOA. With the same "lightcone trick", we were able to sample the distribution of any one qubit, or the joint distribution of any small subset of all qubits. However, the need to account for the correlation between a large number of qubits makes true simulation of *all* qubits out of reach.

7.3.2 Future work

Simulation of noisy quantum devices. QAOA is usually regarded as a promising potential application of "noisy intermediate-scale quantum" (NISQ) devices, but the most important characterization of NISQ devices is that they are noisy. Although QAOA tries to reduce the influence of noise by using a small number of layers, the actual effect of noise is still largely unknown. Using methods similar to the ones used in Chapter 6, we may be able to better understand the effect of noise, which in turn may help us determine whether QAOA would truly be feasible and useful on NISQ devices.

Evaluating the gradient of the energy function. Many efficient numerical optimization algorithms make use of the gradient of the objective function, in addition to the value of the function itself. In our experiments, since we can only evaluate the energy function itself, the gradient must be estimated numerically, which may be inefficient and/or imprecise. If we can represent the gradient of the energy function also in terms of tensor networks, we can potentially extract a better performance from the numerical optimization algorithms.

7.4 Applications to quantum error correction

7.4.1 Summary

In Chapter 6, we studied the effect of crosstalk in surface codes using our parallel tensor network contraction algorithm. In addition to a base error model we adopted from [14], we modeled crosstalk as CPHASE gates on adjacent qubit pairs. Even though in our initial model the strength of crosstalk is uniform over time, we ignored the commutator between CPHASE gates and amplitude damping in order to move all the crosstalk in a region to a single time point.

In order to simulate the noisy syndrome extraction circuit as a tensor network, we made use of density matrix simulation, and also defined simplistic models for encoding and decoding, which allowed us to find the optimal decoder for $2 + 1$ rounds of syndrome extraction by evaluating an order-28 tensor. The results showed that crosstalk has a significant effect on logical error rates of certain types of errors, and also that ignoring the commutator between CPHASE gates and amplitude damping is a valid approximation.

7.4.2 Future work

Different modeling of crosstalk. Our modeling of crosstalk was very rough, and there are a number of refinements and alternatives that may be adopted. For example, in the text we mentioned that crosstalk can also be modeled as $e^{ik(Z \otimes Z)}$ depending on how we calibrate the frequencies of the qubits, and those calibration methods may actually give different error rates.

Another simplifying assumption we made is that the crosstalk strength is not only uniform over time, but also the same between all adjacent qubit pairs. This assumption can be easily refined by taking into account factors such as qubit frequencies.

Tensor network based decoder. Our approach of finding the optimal decoder based on all error syndromes means that we could only simulate surface codes in a very limited parameter regime. In order to do surface code simulations with more qubits or more rounds, it is necessary to find different ways to handle decoding.

A natural idea is to use a decoder that is itself based on tensor network, like the one introduced in [53]. Such a decoder will be able to be incorporated into the tensor network for the entire error correction process, without introducing very large tensors, and thus potentially leading to a tensor network with a low contraction width and cost.

BIBLIOGRAPHY

[1] Michael A Nielsen and Isaac L Chuang. *Quantum computation and quantum information*. Cambridge University Press, 10th anniversary edition, December 2010.

[2] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando GSL Brandao, David A Buell, et al. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, 2019.

[3] Edwin Pednault, John A Gunnels, Giacomo Nannicini, Lior Horesh, and Robert Wisnieff. Leveraging secondary storage to simulate deep 54-qubit sycamore circuits. *arXiv preprint arXiv:1910.09534*, 2019.

[4] Cupjin Huang, Fang Zhang, Michael Newman, Junjie Cai, Xun Gao, Zhengxiong Tian, Junyin Wu, Haihong Xu, Huanjun Yu, Bo Yuan, et al. Classical simulation of quantum supremacy circuits. *arXiv preprint arXiv:2005.06787*, 2020.

[5] Sergio Boixo, Sergei V Isakov, Vadim N Smelyanskiy, Ryan Babbush, Nan Ding, Zhang Jiang, Michael J Bremner, John M Martinis, and Hartmut Neven. Characterizing quantum supremacy in near-term devices. *Nature Physics*, 14(6):595, 2018.

[6] Jianxin Chen, Fang Zhang, Cupjin Huang, Michael Newman, and Yaoyun Shi. Classical simulation of intermediate-size quantum circuits. *arXiv preprint arXiv:1805.01450*, 2018.

[7] Igor L Markov and Yaoyun Shi. Simulating quantum computation by contracting tensor networks. *SIAM Journal on Computing*, 38(3):963–981, 2008.

[8] Benjamin Villalonga, Sergio Boixo, Bron Nelson, Christopher Henze, Eleanor Rieffel, Rupak Biswas, and Salvatore Mandrà. A flexible high-performance simulator for the verification and benchmarking of quantum circuits implemented on real hardware. *arXiv preprint arXiv:1811.09599*, 2018.

[9] Johnnie Gray and Stefanos Kourtis. Hyper-optimized tensor network contraction. *arXiv preprint arXiv:2002.01935*, 2020.

[10] Fang Zhang, Cupjin Huang, Michael Newman, Junjie Cai, Huanjun Yu, Zhengxiong Tian, Bo Yuan, Haihong Xu, Junyin Wu, Xun Gao, et al. Alibaba cloud quantum development platform: Large-scale classical simulation of quantum circuits. *arXiv preprint arXiv:1907.11217*, 2019.

[11] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. A quantum approximate optimization algorithm. *arXiv preprint arXiv:1411.4028*, 2014.

[12] Cupjin Huang, Mario Szegedy, Fang Zhang, Xun Gao, Jianxin Chen, and Yaoyun Shi. Alibaba cloud quantum development platform: Applications to quantum algorithm design. *arXiv preprint arXiv:1909.02559*, 2019.

[13] Cupjin Huang, Xiaotong Ni, Fang Zhang, Michael Newman, Dawei Ding, Xun Gao, Tenghui Wang, Hui-Hai Zhao, Feng Wu, Gengyan Zhang, et al. Alibaba cloud quantum development platform: Surface code simulations with crosstalk. *arXiv preprint arXiv:2002.08918*, 2020.

[14] TE O'Brien, B Tarasinski, and L DiCarlo. Density-matrix simulation of small surface codes under current and projected experimental noise. *npj Quantum Information*, 3(1):39, 2017.

[15] Fang Zhang, Cupjin Huang, Michael Newman, Kevin Sung, and Yaoyun Shi. Limitations on testing quantum theory. *QIP 2018, poster session*, 2017.

[16] Jarrod R McClean, Ian D Kivlichan, Kevin J Sung, Damian S Steiger, Yudong Cao, Chengyu Dai, E Schuyler Fried, Craig Gidney, Brendan Gimby, Pranav Gokhale, et al. OpenFermion: the electronic structure package for quantum computers. *arXiv preprint arXiv:1710.07629*, 2017.

[17] Jianxin Chen, Zhengfeng Ji, David W Kribs, Bei Zeng, and Fang Zhang. Minimum entangling power is close to its maximum. *Journal of Physics A: Mathematical and Theoretical*, 52(21):215302, 2019.

[18] Fang Zhang and Jianxin Chen. Optimizing T gates in Clifford+T circuit as $\pi/4$ rotations around Paulis. *arXiv preprint arXiv:1903.12456*, 2019.

[19] Daniel Smith and Johnnie Gray. opt_einsum — A python package for optimizing contraction order for einsum-like expressions. *Journal of Open Source Software*, 3(26):753, 2018.

[20] Andrew D King, Jack Raymond, Trevor Lanting, Sergei V Isakov, Masoud Mohseni, Gabriel Poulin-Lamarre, Sara Ejtemaee, William Bernoudy, Isil Ozfidan, Anatoly Yu Smirnov, et al. Scaling advantage in quantum simulation of geometrically frustrated magnets. *arXiv preprint arXiv:1911.03446*, 2019.

[21] Riling Li, Bujiao Wu, Mingsheng Ying, Xiaoming Sun, and Guangwen Yang. Quantum supremacy circuit simulation on sunway taihulight. *arXiv preprint arXiv:1804.04797*, 2018.

[22] Nikolaus Hansen, Youhei Akimoto, and Petr Baudis. CMA-ES/pycma on Github. Zenodo, DOI:10.5281/zenodo.2559634, February 2019.

[23] Stefan Arnborg, Derek G Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal on Algebraic Discrete Methods*, 8(2):277–284, 1987.

[24] Vibhav Gogate and Rina Dechter. A complete anytime algorithm for treewidth. In *Proceedings of the 20th conference on Uncertainty in artificial intelligence*, pages 201–208. AUAI Press, 2004.

[25] Hisao Tamaki, Hiromu Ohtsuka, Takuto Sato, and Keitaro Makii. PACE 2017 Track A submissions. `https://github.com/TCS-Meiji/PACE2017-TrackA`, 2017.

[26] Sebastian Schlag, Vitali Henne, Tobias Heuer, Henning Meyerhenke, Peter Sanders, and Christian Schulz. k-way hypergraph partitioning via n-level recursive bisection. In *18th Workshop on Algorithm Engineering and Experiments, (ALENEX 2016)*, pages 53–67, 2016.

[27] Yaroslav Akhremtsev, Tobias Heuer, Peter Sanders, and Sebastian Schlag. Engineering a direct k-way hypergraph partitioning algorithm. In *19th Workshop on Algorithm Engineering and Experiments, (ALENEX 2017)*, pages 28–42, 2017.

[28] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando De Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2015.

[29] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, and Skye Wanderman-Milne. JAX: composable transformations of Python+NumPy programs, 2018.

[30] Richard P Feynman. Simulating physics with computers. *Int. J. Theor. Phys*, 21(6/7), 1982.

[31] John Preskill. Quantum computing and the entanglement frontier. *arXiv preprint arXiv:1203.5813*, 2012.

[32] Peter W Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*, pages 124–134. Ieee, 1994.

[33] Igor L Markov, Aneeqa Fatima, Sergei V Isakov, and Sergio Boixo. Quantum supremacy is both closer and farther than it appears. *arXiv preprint arXiv:1807.10749*, 2018.

[34] Sergio Boixo, Sergei V Isakov, Vadim N Smelyanskiy, and Hartmut Neven. Simulation of low-depth quantum circuits as complex undirected graphical models. *arXiv preprint arXiv:1712.05384*, 2017.

[35] Benjamin Villalonga, Dmitry Lyakh, Sergio Boixo, Hartmut Neven, Travis S Humble, Rupak Biswas, Eleanor G Rieffel, Alan Ho, and Salvatore Mandrà. Establishing the quantum supremacy frontier with a 281 pflop/s simulation. *Quantum Science and Technology*, 5(3):034003, 2020.

[36] Scott Aaronson and Daniel Gottesman. Improved simulation of stabilizer circuits. *Physical Review A*, 70(5):052328, 2004.

[37] Sergio Boixo. Random quantum circuits for circuit sampling with superconducting qubits. https://github.com/sboixo/GRCS, 2019.

[38] John Preskill. Quantum computing in the NISQ era and beyond. *Quantum*, 2:79, 2018.

[39] Edward Farhi, Jeffrey Goldstone, Sam Gutmann, and Michael Sipser. Quantum computation by adiabatic evolution. *arXiv preprint quant-ph/0001106*, 2000.

[40] Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[41] Angelika Steger and Nicholas C Wormald. Generating random regular graphs quickly. *Combinatorics, Probability and Computing*, 8(4):377–396, 1999.

[42] The Cirq Developers. Cirq: A python framework for creating, editing, and invoking Noisy Intermediate Scale Quantum (NISQ) circuits, 2019.

[43] Héctor Abraham, AduOffei, Ismail Yunus Akhalwaya, Gadi Aleksandrowicz, Thomas Alexander, Gadi Alexandrowics, Eli Arbel, Abraham Asfaw, Carlos Azaustre, AzizNgoueya, et al. Qiskit: An open-source framework for quantum computing, 2019.

[44] E Schuyler Fried, Nicolas PD Sawaya, Yudong Cao, Ian D Kivlichan, Jhonathan Romero, and Alán Aspuru-Guzik. qtorch: The quantum tensor contraction handler. *PloS one*, 13(12), 2018.

[45] Michael Streif and Martin Leib. Training the quantum approximate optimization algorithm without access to a quantum processing unit. *arXiv preprint arXiv:1908.08862*, 2019.

[46] Leo Zhou, Sheng-Tao Wang, Soonwon Choi, Hannes Pichler, and Mikhail D Lukin. Quantum approximate optimization algorithm: performance, mechanism, and implementation on near-term devices. *arXiv preprint arXiv:1812.01041*, 2018.

[47] John Preskill. Fault-tolerant quantum computation. In *Introduction to quantum computation and information*, pages 213–269. World Scientific, 1998.

[48] Daniel Gottesman. Stabilizer codes and quantum error correction. *arXiv preprint quant-ph / 9705052*, 1997.

[49] Richard Versluis, Stefano Poletto, Nader Khammassi, Brian Tarasinski, Nadia Haider, David J Michalak, Alessandro Bruno, Koen Bertels, and Leonardo DiCarlo. Scalable quantum circuit and control for a superconducting surface code. *Physical Review Applied*, 8(3):034021, 2017.

[50] MA Rol, F Battistel, FK Malinowski, CC Bultink, BM Tarasinski, R Vollmer, N Haider, N Muthusubramanian, A Bruno, BM Terhal, et al. Fast, high-fidelity conditional-phase gate exploiting leakage interference in weakly anharmonic superconducting qubits. *Physical review letters*, 123(12):120502, 2019.

[51] Dripto M Debroy, Muyuan Li, Michael Newman, and Kenneth R Brown. Stabilizer slicing: Coherent error cancellations in low-density parity-check stabilizer codes. *Physical review letters*, 121(25):250502, 2018.

[52] Mohan Sarovar, Timothy Proctor, Kenneth Rudinger, Kevin Young, Erik Nielsen, and Robin Blume-Kohout. Detecting crosstalk errors in quantum information processors. *arXiv preprint arXiv:1908.09855*, 2019.

[53] Sergey Bravyi, Martin Suchara, and Alexander Vargo. Efficient algorithms for maximum likelihood decoding in the surface code. *Physical Review A*, 90(3):032326, 2014.