

THE UNIVERSITY OF CHICAGO

LARGE-SCALE TENSOR NETWORK QUANTUM ALGORITHM SIMULATOR

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES DIVISION
IN CANDIDACY FOR THE DEGREE OF
MASTERS

DEPARTMENT OF COMPUTER SCIENCE

BY
DANYLO LYKOV

CHICAGO, ILLINOIS

FEBRUARY 2024

TABLE OF CONTENTS

LIST OF FIGURES	v
LIST OF TABLES	viii
ABSTRACT	ix
1 INTRODUCTION	1
1.1 Tensor networks introduction	2
1.1.1 Tensors and quantum states	2
1.1.2 Tensor networks	4
1.2 Quantum computing introduction	5
1.3 Structure	7
2 TENSOR NETWORK SIMULATION OF QUANTUM CIRCUITS	9
2.1 Introduction	9
2.2 Related work	10
2.3 Quantum circuit simulation algorithm	12
2.3.1 Tensor networks and graphical models	12
2.3.2 Simulation of quantum circuits	16
2.4 Batch circuit simulation	19
2.4.1 Simulation of multiple amplitudes	19
2.4.2 Node ordering and chordal graphs	21
2.4.3 Finding restricted elimination orders	24
2.4.4 Numerical examples	27
2.5 Conclusion and comparison	29
3 DIAGONAL GATES APPROACH FOR OPTIMIZING QUANTUM CIRCUIT SIM- ULATION	31
3.1 Abstract	31
3.2 Introduction	31
3.3 QAOA algorithm	33
3.4 Methodology	35
3.4.1 Tensor network approach	35
3.4.2 Tensor network contraction	36
3.5 Optimization techniques	37
3.5.1 Optimization of QAOA circuit structure	37
3.5.2 Diagonal gate simplification	37
3.6 Results	40
3.7 Conclusions	42

4	PARALLEL COMPUTATION	44
4.1	Introduction	44
4.2	Related Work	44
4.3	Methodology	46
4.3.1	QAOA introduction	46
4.3.2	Description of quantum circuits	47
4.4	Overview of simulation algorithm	48
4.4.1	Quantum circuit as tensor expression	48
4.4.2	Graph model of tensor expression	49
4.5	Simulation of a single amplitude	52
4.5.1	Ordering algorithm	52
4.6	Parallelization algorithm	54
4.6.1	Description of hardware and software	55
4.6.2	Single-node parallelization	56
4.6.3	Multinode parallelization	59
4.6.4	Step-dependent slicing	62
4.7	Simulation of several amplitudes	65
4.8	Results	66
4.9	Conclusions	67
4.10	Acknowledgements	68
5	GPU ACCELERATION OF TENSOR NETWORK CONTRACTION	70
5.1	Introduction	70
5.2	Methodology	71
5.2.1	QAOA Overview	71
5.2.2	Tensor Network Contractions	73
5.2.3	Merged Indices Contraction	74
5.2.4	CPU-GPU Hybrid Backend	75
5.2.5	Datasets for Synthetic Benchmarks	76
5.3	Results	79
5.3.1	Single CPU-GPU Backends	79
5.3.2	Merged Backend Results	81
5.3.3	Mix CPU-GPU Backend Results	82
5.3.4	Mixed Merged Backend Results	83
5.3.5	Synthetic Benchmarks	84
5.4	Conclusions	88
6	CONCLUSIONS AND OUTLOOK	91
	REFERENCES	93

LIST OF FIGURES

2.1	Example of quantum circuit drawn as a tensor network. The state of i -th qubit at t -th clock cycle is denoted by $\{s\}_i^t$ (only unique states are shown, e.g. $s_1^2 = s_1^1$ is omitted). The time flows from right to left.	14
2.2	The mapping between two graph-based notations of tensor networks.	15
2.3	Alternative representation (graphical model) of the circuit in Fig. 2.1. Gate tensors are shown in red, self-loops are omitted.	16
2.4	Contraction of a tensor network from Eq. 2.5 in graphical form. The sequence of contractions π is the same as in Eq. 2.6. Labels of tensors are shown in red. . .	17
2.5	Alternative contraction of a tensor network in Eq. 2.5. The maximal clique of size 4 is highlighted in red. This sequence of contractions is not optimal. . . .	18
2.6	Evaluation of amplitude subsets. Top - extended amplitude expression to evaluate all amplitudes of qubits 2, 3, 4; Bottom - resulting amplitude tensor.	20
2.7	Building a chordal graph from the elimination order	22
2.8	Treewidth dependence on the size of a random quantum circuit. Left - dependence of treewidth on the depth of a random circuit, Right - dependence of treewidth on the number of qubits.	28
2.9	Total flop requirements for the simulation of a typical random circuit of varying size. Estimated number of floating point operations for the simulation of the full subset of amplitudes of $ C $ qubits (there are $2^{ C }$ amplitudes). In case of one amplitude at a time simulation a combined cost of all tasks is drawn.	28
2.10	Minimal memory requirement for the simulation of a typical random circuit of varying size	29
2.11	Floating point to memory access ratio during the simulation of random circuits of varying size	30
3.1	Quantum circuit that generates QAOA ansatz state for MaxCut problem on a 4-node complete graph. This widely used decomposition of QAOA into common set of basis gates is not optimal for the classical simulation of the output state. .	32
3.2	Line graphs of tensor networks for calculating QAOA ansatz state using different optimizations. "Default" and "diagonal" show line graphs of tensor network for the circuit shown in Figure 3.1, using a full-matrix gates and diagonal gates approach, respectively. "ZZ gates + diagonal" is obtained by using the diagonal gates approach on a simplified quantum circuit obtained by applying Equation 3.4. This figure demonstrates how improving the conversion of a quantum algorithm to a tensor network can reduce the complexity of the network, providing speedups for both finding contraction order and the contraction itself.	33
3.3	Number of FLOPs to calculate a single amplitude of QAOA ansatz state for MaxCut using a different number of QAOA iterations. Each line shows a combination of optimization techniques, with "diagonal + ZZ gates" being the most advanced one. The shaded region shows $1-\sigma$ interval over 5 random graphs.	39

3.4	Number of FLOPs to calculate a single amplitude of QAOA ansatz state with $p=1$ for MaxCut on random 3-regular graphs with 160 nodes. The shaded region shows $1-\sigma$ interval over 5 random graphs.	39
3.5	The number of FLOPs to calculate a single amplitude of QAOA ansatz state for MaxCut on random 3-regular graphs of different sizes. The number of nodes in the graph corresponds to the number of qubits in the quantum circuit. Each line shows a combination of optimization techniques, with "diagonal + ZZ gates" being the most advanced one. The shaded region shows $1-\sigma$ interval over 5 random graphs.	41
4.1	Correspondence of quantum gates and graphical representation.	49
4.2	Graph representation of tensor expression of the circuit in Fig. 3.1. Every vertex corresponds to a tensor index of a quantum gate. Indices are labeled right to left: 0-3 are indices of the output statevector, and 32-25 are indices of the input statevector. Self-loop edges are not shown (in particular $Z^{2\gamma}$, which is diagonal).	50
4.3	Cost of contraction for every vertex for a circuit with 150 qubits. Inset shows the peak magnified and the number of neighbors of the vertex contracted at a given step (right y-axis).	51
4.4	Comparison of different ordering algorithms for single amplitude simulation of QAOA ansatz state	52
4.5	Illustration of our two-level tensor parallelization approach. On the multinode level MPI parallelization we use slicing of a partially contracted full expression. On the lower level of a single node, we use thread-based parallelization with a shared resulting tensor.	57
4.6	Sketch of the parallel bucket elimination algorithm. Part (a) and steps b2–b4 depend only on the structure of a task and can be executed only once for the QAOA algorithm. Steps b1 and b5 are performed serially. The outer loop of the blue region performs the elimination of the remaining buckets; the inner loop corresponds to processing a single bucket. The summation operation at the end of the bucket processing is omitted for simplicity.	60
4.7	Step-based slicing algorithm. The blue boxes are evaluated for each graph node and are the main contributions to time.	64
4.8	Simulation cost for a batch of amplitudes. The calculations are done for 5 random instances of degree-3 random regular graphs and the mean value is plotted. The three plots are calculated for different number of qubits: 100, 150 and 200. . . .	65
4.9	Experimental data of simulation time with respect to the number of Theta nodes. The circuit is for 210 qubits and 1,785 gates.	66
4.10	Distribution of the contraction width (maximum number of neighbors) c for different numbers of parallel indices n . While variance of c is present, showing that it is sensible to the parallelization index s , we are interested in the minimal value of s , which, in turn, generally gets smaller for bigger n	69

5.1	Breakdown of mean time to contract a single bucket by bucket width. The test is performed for expectation value as described in 5.3.1. CPU backends are faster for buckets of width $\leq 13 - 16$, and GPU faster are better for larger buckets. This picture also demonstrates that every contraction operation spends some time on overhead which doesn't depend on bucket width, and actual calculation that scales exponentially with bucket width.	79
5.2	Distribution of bucket width in the contraction of QAOA full circuit simulation. The y-axis is log scale; 82% of buckets have width ≤ 6 , which have relatively large overhead time.	80
5.3	Breakdown of total time spent on bucket of each size in full QAOA expectation value simulation. The y-value on this plot is effectively one in Figure 5.1 multiplied by one in Figure 5.2. This figure is very useful for analyzing the bottlenecks of the simulation. It shows that most of the time for CPU backend is spent on large buckets, but for GPU backends the large number of small buckets results in a slowdown.	80
5.4	Breakdown of total contraction time by bucket width in full expectation value simulation of problem size 30. Lines with the same color use the same type of backends. The solid lines represent the merged version of backends, and the dashed lines denote the baseline backends. The merged GPU backends are better for buckets of width ≥ 20	82
5.5	Breakdown of sum contraction time by bucket width for merged backends. CPU backends are better for buckets of width ≤ 15 , and GPU backends are better for larger buckets. The hybrid backend's GPU backend spends outperforms the regular GPU backend for buckets of width ≥ 15	85
5.6	FLOPs vs. the number of operations for all tasks on the CuPy backend. "circuit unmerged" and "circuit merged" are results of expectation value of the full circuit simulation of QAOA MaxCut problem on a 3-regular graph of size 30 with depth $p = 4$. "tncontract random" tests on tensors of many indices where each index has a small size. "tncontract fixed" uses the contraction sequence "abcd,bcdf \rightarrow acf" for all contractions. "matmul" performs matrix multiplication on square matrices. All groups use <code>complex128</code> tensors in the operation. We use the triangles to denote the data at ~ 100 million operations, which is shown in Table 5.3. . . .	88
5.7	FLOPs vs. the number of operations for all tasks on NumPy backend. Same problem setting as Fig. 5.6. "tncontract random" outperforms "tncontract fixed" as the ops value increases. Merged backend does not have an advantage on CPU compared to the unmerged backend. We use the triangles to denote the data at ~ 100 million operations, which is shown in Table 5.3.	89

LIST OF TABLES

3.1	The maximum number of QAOA iterations p for which one can simulate a single amplitude of ansatz state for MaxCut on a 40-node random regular graph. . . .	42
3.2	The maximum number of nodes of a 3-regular graph for which one can simulate a single amplitude of the MaxCut QAOA ansatz state.	42
4.1	Comparison between different notations of quantum circuits	49
4.2	Hardware and software specifications	56
5.1	Time for full QAOA expectation value simulation using backend that utilize GPUs or CPUs. The expectation value is MaxCut on a 3-regular graph of size 30 and QAOA depth $p = 4$. Speedup shows the overall runtime improvement compared with the baseline CPU backend "NumPy". "Mixed" device means the backend uses both CPU and GPU devices.	83
5.2	Time for full QAOA expectation value simulation using different Merged backends, as described in Section 5.2.3. The expectation value is MaxCut on a 3-regular graph of size 30 and QAOA depth $p = 4$. Speedup shows the overall runtime improvement compared with the baseline CPU backend "NumPy".	84
5.3	Summary of GPU and CPU FLOPs for different tasks at around 100 million operations. Matrix Multiplication and Tensor Contraction tasks are described in Section 5.3.5. "Bucket Contraction" groups record the maximum number of FLOPs for a single bucket. "Lightcone Contraction" groups contain the FLOPs data on a single lightcone where the sum of operations is approximately 100 millions, small and large buckets combined.	86

ABSTRACT

As quantum computing field is starting to reach the realm of advantage over classical algorithms, simulating quantum circuits becomes increasingly challenging as we design and evaluate more complex quantum algorithms. In this context, tensor networks, which have become a standard method for simulations in various areas of physics, from many-body quantum physics to quantum gravity, offer a natural approach. Despite the availability of efficient tools for physics simulations, simulating quantum circuits presents unique challenges, which I address in this work, specifically using the Quantum Approximate Optimization Algorithm as an example.

The main results of this work span several steps of the problem. For the step of creating a tensor network, I demonstrate that applying the diagonal representation of quantum gates leads to a complexity reduction in tensor network contraction by one to four orders of magnitude.

For large-scale contraction of tensor networks, I propose a step-dependent parallelization approach which performs slicing of partially contracted tensor network. Finally, I study tensor network contractions on GPU and propose an algorithm of contraction which uses both CPU and GPU to reduce GPU overhead. In our benchmarks, this algorithm reduces time to solution from 6 seconds to 1.5-2 seconds.

CHAPTER 1

INTRODUCTION

The science of classically simulating quantum many-body physics involves exponential scaling of memory resources. The fact that quantum many-body systems are hard to simulate classically is the basis for the idea of quantum computing as first suggested by Richard Feynman [Feynman, 1982].

Growing interest in quantum computing in recent years led to the increase of size and capabilities of experimental quantum computers. Promising physical realizations of quantum computing devices were implemented in recent years [Intel, 2018; IBM, 2018], which bolsters the expectations that a long thought quantum supremacy will be reached [Harrow and Montanaro, 2017; Boixo et al., 2018; Neill et al., 2018].

Quantum information science has a tremendous potential to speed up calculations of certain problems over classical calculations [Alexeev et al., 2021; Shor, 1994]. To continue the advances in this field, however, often requires classically simulating quantum circuits. Such simulation is done by using classical simulation algorithms that replicate the behavior of executing quantum circuits on classical hardware such as personal computers or high-performance computing (HPC) systems. These algorithms play an important role and can be used to (1) verify the correctness of quantum hardware, (2) help the development of hybrid classical-quantum algorithms, (3) find optimal circuit parameters for hybrid variational quantum algorithms, (4) validate the design of new quantum circuits, and (5) verify quantum supremacy and advantage claims.

Tensor networks are an invaluable tool for the classical simulation of both quantum computers and general physical systems. For example, in the domain of molecular quantum dynamics the Multi-layer multi-configuration time-dependent Hartree (ML-MCTDH) [Wang and Thoss, 2003] algorithm, which is an extension of MCTDH [Meyer et al., 1990] algorithm, achieved significant recognition. This algorithm uses tensor networks to represent quantum

states and then solve the underlying dynamics equations.

Moreover, tensor networks can be used to exactly calculate the partition function of quantum many-body systems [Vanderstraeten et al., 2018]. The partition function can then be used to extract useful information about the system, such as energy or specific heat capacity.

1.1 Tensor networks introduction

1.1.1 Tensors and quantum states

A vector is an entity with magnitude and direction, often used to represent physical quantities such as velocity or force. In contrast, a covector (or one-form) is a linear map from vectors to scalars. Both can be represented as an array of numbers in a given basis, but when the basis is changed they transform oppositely to ensure that all scalars are invariant, a key principle in physics and geometry.

Expanding to two dimensions, we distinguish between bilinear forms and linear maps from one vector space to another. A bilinear form takes two vectors as input and returns a scalar. Interestingly, a linear map can also be viewed as a bilinear map taking a covector and a vector as input, producing a scalar. Similarly to vectors and covectors, both bilinear forms and linear maps can be represented as matrices, and special rules are used to change the representation when changing the basis.

A tensor generalizes these ideas further. A tensor of rank n is a n -linear map from a mix of n vectors and covectors to a scalar. To specify a tensor, one has to provide a representation in some basis and the rules for changing the representation under a change of basis. Examples of tensors in physics include the electromagnetic tensor $F_{\mu\nu}$ and the Levi-Civita tensor, among many others.

In the context of quantum computing and computer science, the basis is often fixed

forever, and a tensor is just a representation: an array of numbers that is indexed by several indices. The number of indices is called *order* or sometimes *rank* or *mode* of a tensor and the index is sometimes called the *dimension*. The number of values that an index can have is usually called the *size* of the index. For instance, a scalar, is a single number is labeled by zero indices, so a scalar is considered to be a 0th-order tensor. A vector is a tensor of first order and a matrix is a tensor of second order. In the case of quantum physics, tensors can be used for the representation of states. For example, a first-order tensor can be used to represent the state vector of a spin- $\frac{1}{2}$ particle in some basis:

$$|\psi_1\rangle = C_0 |0\rangle + C_1 |1\rangle = \sum_{s=0,1} C_s |s\rangle. \quad (1.1)$$

The $|0\rangle$ and $|1\rangle$ are basis vectors that correspond to, for instance, spin-up and spin-down states. The vector C_s is a first-rank tensor that represents this state given the basis $|0\rangle, |1\rangle$. This approach can be extended to a tensor of two particles:

$$|\psi_2\rangle = C_{00} |00\rangle + C_{01} |01\rangle + C_{10} |10\rangle + C_{11} |11\rangle = \sum_{s_1, s_2=0}^1 C_{s_1, s_2} |s_1 s_2\rangle, \quad (1.2)$$

where a common notation is used $|ab\rangle = |a\rangle |b\rangle = |a\rangle \otimes |b\rangle$.

In quantum physics, it is common to use a state vector with a dimension of size 4 to represent a state of such a system. The difference in representing it as a second-order tensor as above is just the way of labeling the numbers, but such notation is the first step towards impressive advantages for classical simulation of tensor networks.

As a natural extension for a system of N spins, one can use the following representation of the state:

$$|\psi_N\rangle = \sum_{s_1 \dots s_N=0}^1 C_{s_1 \dots s_N} |s_1 \dots s_N\rangle. \quad (1.3)$$

Similarly, the tensor $C_{s_1 \dots s_N}$ can be reshaped into a vector of size 2^N . The above ex-

amplitudes were for spin- $\frac{1}{2}$ systems, but the approach is the same for collections of quantum systems with larger state space.

It is important to note that since the basis is fixed, there is no classification of indices into covariant and contravariant, e.g. there is no distinction between indices for bra- and ket-states, which is a common distinction in tensors used in physics.

1.1.2 Tensor networks

As described above it may seem that tensors are not more useful than just vectors, after all the difference lies only in labeling the numbers. It is when tensors are combined with each other into a tensor network, that the true usefulness appears. A tensor network is a product of tensors, which can share indices between each other. For example, the expression $A_{ij}B_{jk}$ is a tensor network. This tensor network can be viewed as a tensor itself, which is indexed by indices ijk . Each value of the tensor is a product of two elements from tensors A and B :

$$T_{ijk} = A_{ij}B_{jk}. \quad (1.4)$$

Tensor networks are widely used to represent a linear mathematical model of the studied system. The fact that the model is linear means that the values of interest are a sum of products of numbers, which are input parameters to the model. For example, when modeling rotation of a solid, to obtain coordinates of the rotated geometry, one uses a linear map on a 2-D vector space. The first component of resulting coordinates will be $u_0 = A_{00}v_0 + A_{01}v_1$, and the second $u_1 = A_{10}v_0 + A_{11}v_1$, where v_i are the initial coordinates and A_{ij} are the rotation parameters. The initial v_i and rotated u_i vectors and the matrix A_{ij} are all tensors, so we can use a summation over a tensor network to represent such a linear model:

$$u_i = \sum_{j=0,1} A_{ij}v_j = \sum_{j=0,1} K_{ij}. \quad (1.5)$$

The example in the Equation (1.4) can be used to represent a matrix multiplication:

$$C_{ik} = \sum_{j=0,1} T_{ijk} = \sum_{j=0,1} A_{ij} B_{jk}. \quad (1.6)$$

In order to compute the values for the tensor C it is not required to store all the elements of tensor T in memory at the same time. Instead it is possible to evaluate each element C_{ij} by evaluating corresponding entries T_{ijk} , then performing the summation, and finally discarding the used entries, thus saving memory. The process of evaluating a sum is called a *contraction* of a tensor network. For example, in equation (1.6) the tensor network is contracted over index j .

More complex tensor networks can have summation over many indices and have hundreds or thousands of tensors and indices. They can be used to represent more complex models, such as hidden Markov chains [Gillman et al., 2020] or probabilistic graphical models [Miller et al., 2021; Carrasquilla et al., 2019]. They are also remarkably efficient at representing quantum circuits and states of many-body quantum systems. In addition to exact representations, tensor networks are widely used in algorithms for approximately calculating quantum many-body systems of large size or infinite size. The most popular algorithm is known as Density Matrix Renormalization Group (DMRG) [Schollwöck, 2011] which uses the representation of Matrix Product State (MPS) in case of a chain of a quantum system to approximately calculate the ground state or arbitrary observables of the systems [Orús, 2014].

1.2 Quantum computing introduction

Quantum computers are physical systems that allow the implementation of arbitrary transformations of a quantum state. A quantum computer consists of several qubits, which can interact with each other. In a classical computer, computation is performed by taking input

information represented as bits of data, then applying some operations to it to calculate some useful output data. The operations are composed of elementary logic gates that calculate some output value from input bits. Examples of classical logic gates are $\text{NOT}(b_0)$ which flips the input bit, and $\text{AND}(b_0, b_1)$ which outputs a binary sum of input bits.

Each qubit in a quantum computer is conceptually similar to a bit in a classical computer. Usually, it is a two-level quantum system, analogous to spin- $\frac{1}{2}$ particle. In order to describe computation on a quantum computer, we use quantum gates. Each quantum gate acts on one or several qubits and transforms the corresponding state. For example, when a one-qubit gate is applied to a single qubit in the state $|0\rangle$, the transformation is described as

$$\hat{U} |s\rangle = U_{0s} |0\rangle + U_{1s} |1\rangle = |\psi\rangle, \quad (1.7)$$

where s is a binary variable that labels the qubit basis. Examples of quantum gates are Pauli rotations $\hat{\sigma}_x$, $\hat{\sigma}_y$, and $\hat{\sigma}_z$. Another widely used gate is called Hadamard gate:

$$\hat{H} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \quad (1.8)$$

If a quantum gate is applied to i -th qubit in a quantum computer with N qubits, the change of state is following:

$$\hat{U}^i |0_0 0_1 \dots s_i \dots 0_N\rangle = c_{0s} |0_0 0_1 \dots 0_i \dots 0_N\rangle + c_{1s} |0_0 0_1 \dots 1_i \dots 0_N\rangle = |\psi\rangle. \quad (1.9)$$

Now we can apply the framework of tensor networks to describe such operation:

$$\hat{U}^i \sum_{s_1 \dots s_N=0}^1 C_{s_1 \dots s_N} |s_1 \dots s_N\rangle = \sum_{s_1 \dots s_N=0}^1 C_{s_1 \dots s_N} \hat{U}^i |s_1 \dots s_N\rangle \quad (1.10)$$

$$= \sum_{s_1 \dots s_N=0}^1 C_{s_1 \dots s_N} \sum_{s_{N+1}} U_{s_{N+1} s_i} |s_1 \dots s_N\rangle \quad (1.11)$$

$$= \sum_{s_m=0, m \in \{1 \dots N+1\}, m \neq i}^1 D_{s_1 \dots s_{N+1}} |s_1 \dots s_N\rangle. \quad (1.12)$$

The last summation is performed over all indices s_i for $i = 1 \dots N + 1$ except index s_i which was contracted. We can omit the basis kets and the summation over those and concentrate on just the tensors that represent the states.

$$\sum_{s_j} U_{s_{N+1} s_j} C_{s_0 s_1 \dots s_j \dots s_N} = D_{s_0 s_1 \dots s_{N+1} \dots s_N}. \quad (1.13)$$

In a case of a two-qubit gate, we can represent the resulting state in a similar way, using a 4-th order tensor instead

$$\sum_{i_j i_k} W_{i_{N+2} i_{N+1} i_j i_k} C_{s_0 s_1 \dots s_j \dots s_k \dots s_N} = D_{s_0 s_1 \dots i_{N+2} \dots i_{N+1} \dots s_N}. \quad (1.14)$$

1.3 Structure

In this work, I start with Chapter 2 on using tensor networks for the classical simulation of quantum circuits. I describe how a quantum circuit may be converted to a tensor network and how its contraction is used to obtain different properties of the quantum circuit. In particular, this chapter describes an efficient method for simulating a batch probability amplitudes in one tensor network contraction.

The conversion of quantum circuit to a tensor network may be tweaked to produce significant reductions in tensor network complexity and its computational cost. In Chapter 3 I

demonstrate a significant improvement when using the diagonal gates approach in application to circuits for Quantum Approximate Optimization Algorithm (QAOA).

Next, in Chapter 4 I discuss how to efficiently contract a tensor network, given a set of classical hardware resources. In particular, I explore various parallelization approaches based on slicing the tensor networks that are crucial for large-scale simulations. In Chapter 5 I explore using GPU for tensor network contraction and propose an algorithm for dynamic balancing of tensor contraction steps between CPU and GPU.

Finally, Chapter 6 summarizes the work in this thesis and provides possible avenues for future research.

CHAPTER 2

TENSOR NETWORK SIMULATION OF QUANTUM CIRCUITS

This chapter is adapted from the publication by Schutski, Lykov, and Oseledets [2020].

2.1 Introduction

Substantial progress has been made in understanding quantum computation and developing classical simulators of quantum circuits. Efficient simulators were developed for highly parallel computers, such as Sunway Taihulight [Li et al., 2018]. At the moment the simulation software is aimed at either one of two tasks. The first one is predicting the probability of measuring a particular binary string as the result of a quantum program, or single amplitude simulation. The second is obtaining the full distribution of quantum circuit outputs, or full state simulation. The first approach was found more economical in terms of memory a classical computer has to use, thus allowing the simulation of few amplitudes of larger quantum circuits on up to 100 qubits [Chen et al., 2018b]. On the other hand, the second approach may be preferred when the full state information is needed, such as in Shor’s algorithm [Shor, 1994].

In this chapter, we present a unified approach to quantum circuit simulation. The user can choose the number of probabilities of bitstrings to simulate in a single pass. Our algorithm allows one to balance between the amount of available computational resources and the overall time of the simulation. We build our work on the connection between graphical models and quantum circuits introduced by Markov and Shi [2008] and later developments by [Boixo et al., 2017] and other authors. A relevant research work was presented by Pednault et al. [2017]. We find that our approach is more straightforward, as it disentangles the problem of multiple amplitude simulation from the parallelization. We defer a more detailed comparison to a later section. An overview of the chapter is as follows.

In Section 2.2 we compare our approach to existing techniques. In Sec. 2.3, we review the connection of quantum circuits, tensor diagrams, and statistical graphical models. We then proceed by describing a basic algorithm for circuit simulation based on [Markov and Shi, 2008; Boixo et al., 2017]. In Sec. 2.4 we formulate the main problem solved in this work, which is batch simulation of amplitudes. To solve it, we recall the tree decomposition of graphs and its connection to the problem of ordering of graph nodes. We then propose a new algorithm to transform graph orderings while preserving treewidth (which is the measure of quality) of the given ordering. To achieve a proper transformation we use the connection of tree decomposition and chordal graphs, as explained in 2.4.2 and 2.4.3. Numerical experiments are listed in Sec. 2.4.4. Finally, we conclude in Sec. 2.5 with final remarks and outline possible future research.

2.2 Related work

The problem of efficient tensor contraction was approached multiple times in the field of many-body physics and quantum computing. Some older works are based on the sequential application of sparse matrices to the state vector, such as in [De Raedt et al., 2007]. The authors issued a follow-up paper recently [De Raedt et al., 2019]. Their simulator can evaluate both full sets and subsets of amplitude tensor. This direct simulation procedure, however, requires a lot of non-trivial techniques to make it efficient, especially if parallel operation is considered. Another problem is that it is hard to analyze the effectiveness of the algorithm compared to theoretical bounds on the numerical cost [Aaronson and Chen, 2016]. The latter fact has lead to the previously believed threshold of 50 qubits for “quantum supremacy”.

The seminal work of Markov and Shi [2008] introduced tensor networks for quantum algorithm simulations and showed that treewidth is a natural measure of simulation hardness. The graph-based notation became standard in tensor network literature a decade ago [Bridge-

man and Chubb, 2017]. Following Markov and Shi, several groups developed highly efficient algorithms for quantum circuit simulation based on this representation, see [Pfeifer et al., 2014; Chen et al., 2018c; Pednault et al., 2017; Li et al., 2018] for more details. The previous threshold of 50 qubits was raised, as is demonstrated by multiple authors [Chen et al., 2018c; Pednault et al., 2017; Li et al., 2018]. Usually, these simulators are capable of evaluating full state vectors as well as some subsets of the amplitudes. A similar program was created for contraction of tensors emerging in the many-body physics community [Pfeifer et al., 2014]. The drawback of the approaches based on tensor diagrams is the hardness of the development of efficient codes and the theoretical performance analysis, especially if parallelization is involved. To see why, let us note that classical tensor networks were developed to represent pairwise contractions. Quantum circuits often involve multiple diagonal gates, which allows for significant computational savings. The treewidth of classical diagram’s graphs is higher than optimal (see Appendix in [Boixo et al., 2017]). Traditional network notation can be understood as a hypergraph to eliminate this drawback, as was done in [Pednault et al., 2017]. However, the theory of hypergraphs is less known to the general scientific community.

Recently Boixo et al. [2017] proposed to consider line graphs of the classical tensor networks, which has multiple benefits. First, it establishes the connection of quantum circuits with probabilistic graphical models, allowing for knowledge transfer between the fields. Second, these graphical models avoid the overhead of traditional diagrams for diagonal tensors. Moreover, treewidth is a universal measure of complexity for these models, and links the complexity of quantum states to the well-studied problems in graph theory, a topic we hope to explore in future works. Additionally, simple parallelization of the simulator is possible, as demonstrated in the work of Chen et al. [2018b]. The only disadvantage of the line graph approach was that it is limited usability to simulate subtensors of amplitudes, which we are going to address in this chapter.

Lastly, we have to mention that multiple approximate methods are currently being de-

veloped for circuit simulation. Very recent work of Carrasquilla et al. [2019] presents a neural-network based approach. Pan et al. [2019] devised an approximate algorithms based on tensor network transformation. Extension of our approach with approximation techniques may be a prospective direction of research.

2.3 Quantum circuit simulation algorithm

In this section we describe a procedure for efficient quantum circuit evaluation. We first set up the notation, and then review the current state of the art method for numerical simulation of quantum circuits.

2.3.1 Tensor networks and graphical models

A quantum program describes an evolution of the initial state $|0\rangle$ of a system of n qubits. Any evolution of a physical system corresponds to a unitary operator. Thus, the result of a quantum circuit is a state $|\psi\rangle$, which is a linear transformation of the input state: $|\psi\rangle = \mathcal{U}|0\rangle$. Usually, the transformation \mathcal{U} is performed in several steps corresponding to clock cycles of a quantum computer. Suppose the transformation is described by a depth d circuit. We introduce the following notation:

$$\begin{aligned}\mathcal{U}|0\rangle &= \mathcal{U}^d \dots \mathcal{U}^2 \mathcal{U}^1 |0\rangle \\ |s^{t+1}\rangle &= \mathcal{U}^t |s^t\rangle, \quad |s^0\rangle = |0\rangle\end{aligned}\tag{2.1}$$

Here \mathcal{U}^t are unitary matrices acting at the t -th clock cycle and $|s^t\rangle$ is the state vector. In the simplest case the initial state is taken to be a product of single qubit states $|0\rangle = |0_0\rangle \otimes \dots \otimes |0_n\rangle$. A naive simulation algorithm would take the initial vector $|0\rangle$ and apply matrices \mathcal{U}^t to it. This procedure lays behind full state circuit simulation. To calculate an

amplitude of a bit string x , one would evaluate a dot product $\langle x | s^d \rangle$:

$$\sigma(x) = \langle x | s^d \rangle = \sum_{i=1}^n \langle x_i | s^d \rangle \quad (2.2)$$

The probability of x is then the modulus squared of the amplitude. Note, however, that it is hard to perform full state simulation efficiently. A naive algorithm would need to operate on vectors of size 2^n . Also, the matrices \mathcal{U}^t are highly sparse, at least if they represent transformations achievable with single and two-qubit gates in modern experimental hardware. Here and later in the chapter, we chose to work with the following universal set of one and two-qubit gates: $\{X^{1/2}, Y^{1/2}, CZ, T, H\}$; the same reasoning, however, applies to any quantum gates.

An alternative to full state simulation would be the evaluation of one or several amplitudes from Eq. 2.2 without explicitly forming $|s^d\rangle$. The latter approach provides several benefits. First of all, we can avoid storing the high dimensional state vector $|s^d\rangle$ in computer memory. Second, it may be easier to use the internal structure of the operators \mathcal{U}^t to perform calculations efficiently. Let us introduce a set of variables to denote the state at different cycles of the circuit.

$$\{s\}_i^t, \quad s \in [0, 1], \quad i \in [1, n], \quad t \in [0, d] \quad (2.3)$$

We slightly abuse notation here, as $|s_i^t\rangle$ denotes a state of the i -th qubit at t -th cycle, and s_i^t is a binary variable indexing this state. The same notation is used for the initial and final states, e.g. $|s_i^0\rangle = |0_i\rangle$ and $|s_i^d\rangle = |x_i\rangle$. Consider a circuit shown in Fig. 2.1.

We start with a product state $|0\rangle$ on the right. As the program proceeds the states of individual qubits are changed by gates application. Note that the gates T and cZ do not change the basis of the single-qubit subspaces they act on (they only multiply basis vectors by constants), and hence $|s_i^t\rangle = |s_i^{t+1}\rangle$ for those qubits. In contrast, non-diagonal gates

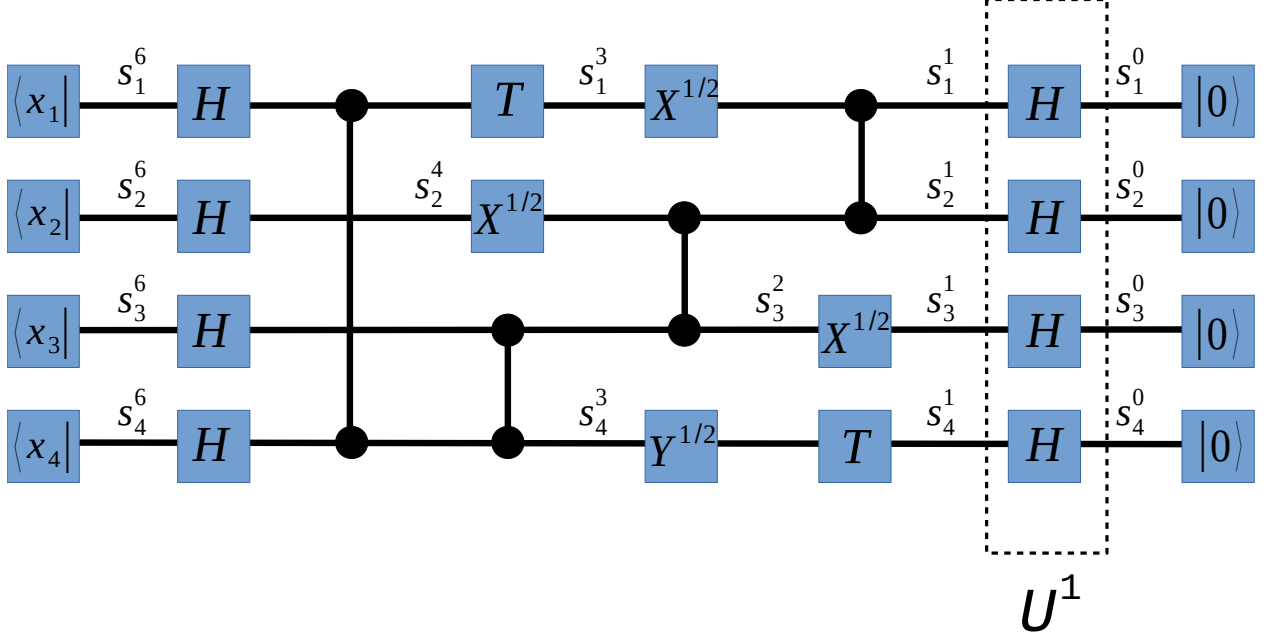


Figure 2.1: Example of quantum circuit drawn as a tensor network. The state of i -th qubit at t -th clock cycle is denoted by $\{s_i^t\}$ (only unique states are shown, e.g. $s_1^2 = s_1^1$ is omitted). The time flows from right to left.

$\{X^{1/2}, Y^{1/2}, H\}$ mix basis vectors of the appropriate qubit subspaces, and new variables $|s_i^{t+1}\rangle$ have to be introduced for the resulting bases. On Fig. 2.1 only unique variables are shown. The expression for the single amplitude in Eq. 2.2 can be rewritten as

$$\begin{aligned} \sigma(x) &= \langle x | \mathcal{U} | 0 \rangle = \\ &= \sum_{\{s_i^t\}} \langle x_i | \mathcal{G}_i^d | s_i^{d-1} \rangle \dots \langle s_i^{t+1} s_j^{t+1} | \mathcal{G}_{ij}^t | s_i^t s_j^t \rangle \dots \langle s_i^1 | \mathcal{G}_i^1 | 0_i \rangle \\ & \mathcal{G}_i^t \in \{X^{1/2}, Y^{1/2}, T, H\}, \quad \mathcal{G}_{ij}^t = cZ \end{aligned} \quad (2.4)$$

The Eq. 2.4 can be interpreted as a discrete Feynman path integral as it represents a sum over all ‘paths’ to go from $|0\rangle$ to $|x\rangle$. On the other hand, one can easily see that the evaluation of the amplitude $\sigma(x)$ in Eq. 2.4 is equal to the contraction of a tensor network shown in Fig. 2.1. For the introduction to the graphical notation used for tensor networks please refer to [Cichocki et al., 2016]. It is well known, however, that the numerical cost of tensor

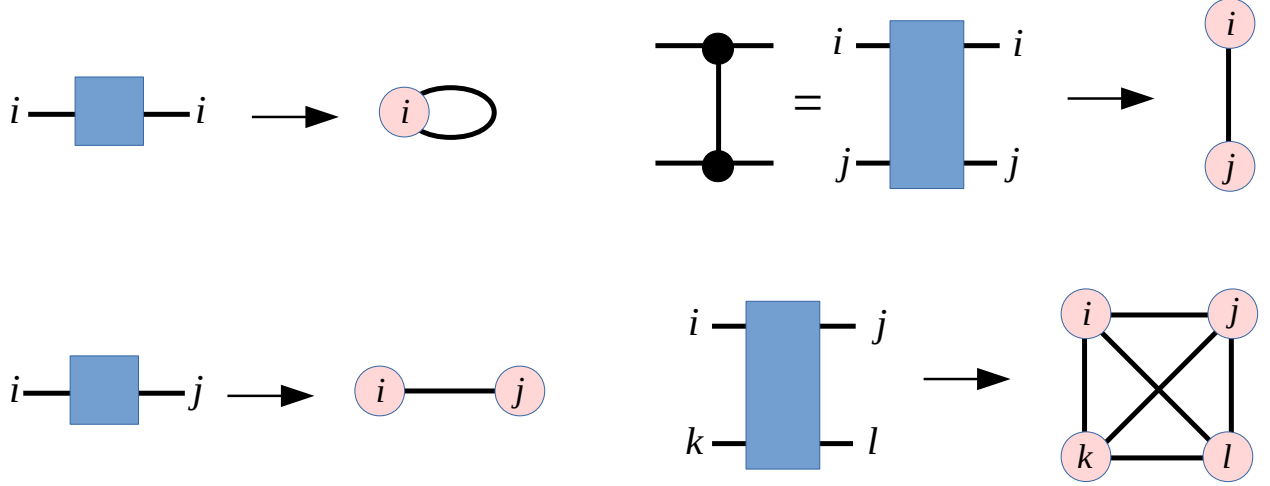


Figure 2.2: The mapping between two graph-based notations of tensor networks.

contractions dramatically depends on the order of operations. Following Markov and Shi [2008] let us introduce another type of graphical models to denote quantum circuits, which better suits for the estimation of numerical costs.

In traditional notation, a tensor network is represented by a graph with nodes standing for tensors and edges denoting their indices. In the new notation, we use nodes to denote unique indices, and tensors are denoted by cliques (fully connected subgraphs). Note that tensors, which are diagonal along some of the axes and hence can be indexed with fewer variables, are depicted by cliques of size lower than the dimension of the corresponding tensor. For a special case of vectors or diagonal matrices, self-loop edges are used. Fig. 4.1 lists the notation for the gates used in this work.

A graphical model, which is equivalent to the circuit in Fig. 2.1, is shown in Fig. 2.3 (self loops are omitted for simplicity). As was pointed out by Boixo et al. [2017] this representation of tensor contractions is traditional in Bayesian network literature. Notice that provided a quantum circuit in a traditional form, one can easily build its probabilistic model representation. To do that, one has to replace all edges carrying non-equivalent single-qubit states with nodes, and all gates with cliques. The diagonal structure of CZ gate tensors leads to significant simplification of the resulting graphs. This is discussed in more detail in

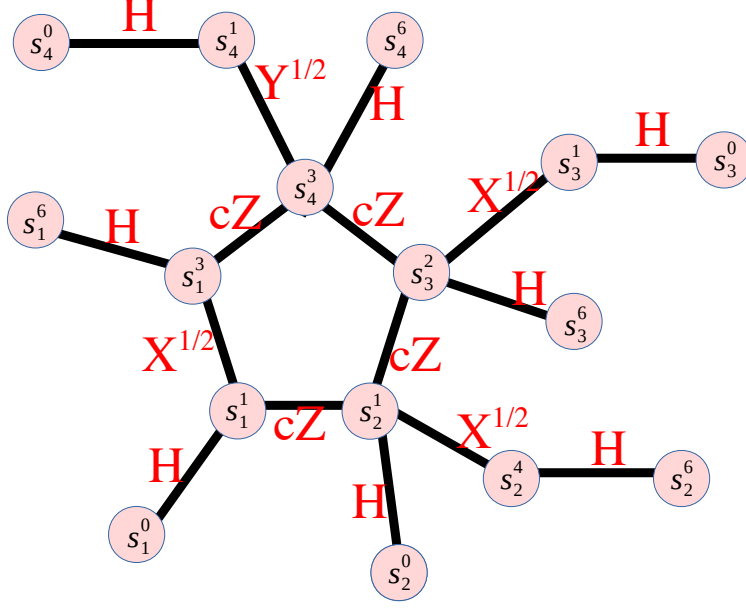


Figure 2.3: Alternative representation (graphical model) of the circuit in Fig. 2.1. Gate tensors are shown in red, self-loops are omitted.

Chapter 3

2.3.2 Simulation of quantum circuits

Having set up the notation, let us proceed with a description of a basic procedure for the evaluation of tensor networks. This algorithm was developed in the context of probabilistic models under the names of bucket elimination [Dechter, 2013] or the variable elimination algorithms [Marsland, 2011].

As an example, let us consider the contraction of a simple tensor network:

$$\sum_{ijklmn} A_{ij} B_{jk} C_{ikl} D_{km} E_{ln} F_{mn} = \sigma \quad (2.5)$$

The graphical model of this network is shown in Fig. 2.4. We choose the order of indices as $\pi = \begin{pmatrix} i & j & k & l & m & n \\ 1 & 2 & 3 & 4 & 5 & 6 \end{pmatrix}$, e.g. i is first, j is second etc. In the bucket elimination procedure the indices are contracted one at a time in order fixed by π , until no indices is left. The sequence

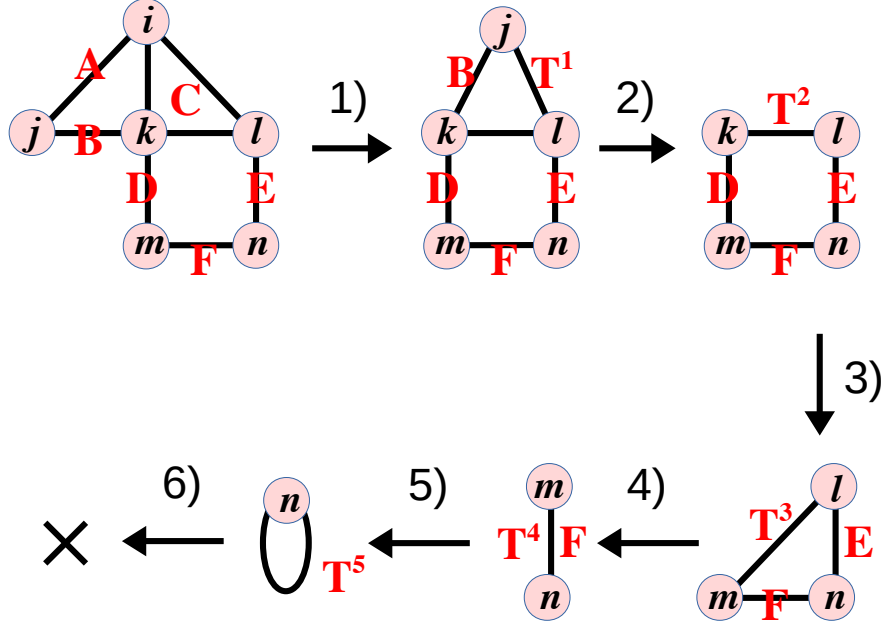


Figure 2.4: Contraction of a tensor network from Eq. 2.5 in graphical form. The sequence of contractions π is the same as in Eq. 2.6. Labels of tensors are shown in red.

of the expressions evaluated in the algorithm is listed below. Assuming the dimensions of all indices is L , we also list numerical costs of the operations.

$$\begin{aligned}
 1) \quad & \sum_i A_{ij} C_{ikl} = T_{jkl}^1 \quad \mathcal{O}(L^4) \\
 2) \quad & \sum_j B_{jk} T_{jkl}^1 = T_{kl}^2 \quad \mathcal{O}(L^3) \\
 3) \quad & \sum_k D_{km} T_{kl}^2 = T_{ml}^3 \quad \mathcal{O}(L^3) \\
 4) \quad & \sum_l E_{ln} T_{ml}^3 = T_{nm}^4 \quad \mathcal{O}(L^3) \\
 5) \quad & \sum_m T_{nm}^4 = T_n^5 \quad \mathcal{O}(L^2) \\
 6) \quad & \sum_n T_n^5 = \sigma \quad \mathcal{O}(L)
 \end{aligned} \tag{2.6}$$

The sequence of transformations of the graphical model corresponding to Eq. 2.6 is shown in Fig. 2.4 2) - 6).

At each step, the contracted variable is removed from the graph, and all its neighbors form a clique. This clique corresponds to the next intermediate in the sequence. Note that the order of the cliques formed at each step corresponds to the exponent of the scaling of numerical cost.

The computational cost of the tensor network contraction is highly dependent on the order of operations. To illustrate this let us consider an alternative order $\tilde{\pi} = \begin{pmatrix} k & j & i & l & m & n \\ 1 & 2 & 3 & 4 & 5 & 6 \end{pmatrix}$ for evaluating the Eq. 2.5. The corresponding sequence of graphical models is shown in Fig. 2.5. Note that the size of the maximal clique corresponding to order $\tilde{\pi}$ is four, which translates to the intermediate of order four and the overall scaling $\mathcal{O}(L^5)$ of the numerical effort.

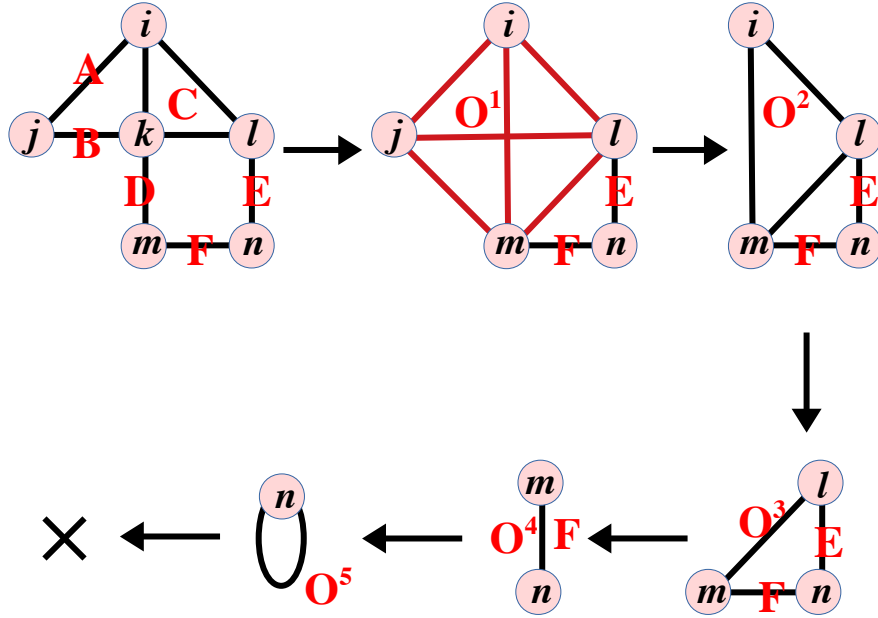


Figure 2.5: Alternative contraction of a tensor network in Eq. 2.5. The maximal clique of size 4 is highlighted in red. This sequence of contractions is not optimal.

Finding the elimination order of a graph is equivalent to the calculation of its tree decomposition; the size of the maximum clique of an order π is $\text{treewidth} + 1$. Tree decomposition is NP-hard for general graphs [Bodlaender, 1994], and a similar hardness result is known for the optimal tensor contraction problem [Chi-Chung et al., 1997]. However, several exact and

approximate algorithms for tree decomposition were developed in graph theory literature; for references, please see [Gogate and Dechter, 2004; Bodlaender et al., 2006; Kloks, 1994; Bodlaender, 1994; Kloks et al., 1993]. For our simulations, we used an exact algorithm of V. Gogate[Gogate and Dechter, 2004]. Having reviewed the procedure for calculation of a single amplitude, let us consider the case of multiple amplitudes, which is the main topic of this chapter.

2.4 Batch circuit simulation

2.4.1 *Simulation of multiple amplitudes*

The procedure we used to calculate single amplitude can be easily extended to calculate any subtensor of the full amplitude tensor. Suppose we are interested in amplitudes of two bitstrings differing only in the value of the first qubit, e. g. $x^0 = (0, s_2^d, \dots, s_n^d)$ and $x^1 = (1, s_2^d, \dots, s_n^d)$. Let us note that the expressions for the amplitudes of σ^0 and σ^1 differ only by the value of the state vector of the first qubit, which is $|0\rangle$ and $|1\rangle$ respectively. One could merge both expressions and introduce an additional variable s_1^{d+1} to index the result $\sigma(s_1^{d+1})$ (which is a vector of size two). The same procedure can be implemented for any combination of output qubits; thus, any subtensor of the full amplitude tensor can be encoded. A graphical representation of the extended amplitude expression is shown in Fig. 2.6. We have to mention that the same procedure can be used not only to evaluate the probabilities of multiple output states, but also the evolution of multiple input states. This approach can be used to simulate the dynamics of mixed states, although we will not elaborate on this in the current chapter.

In order to evaluate multiple amplitudes, the resulting extended expressions have to be contracted only partially (the indices of the amplitude subtensors should not be summed over). Partial contraction can be achieved by stopping the bucket elimination algorithm [Dechter,

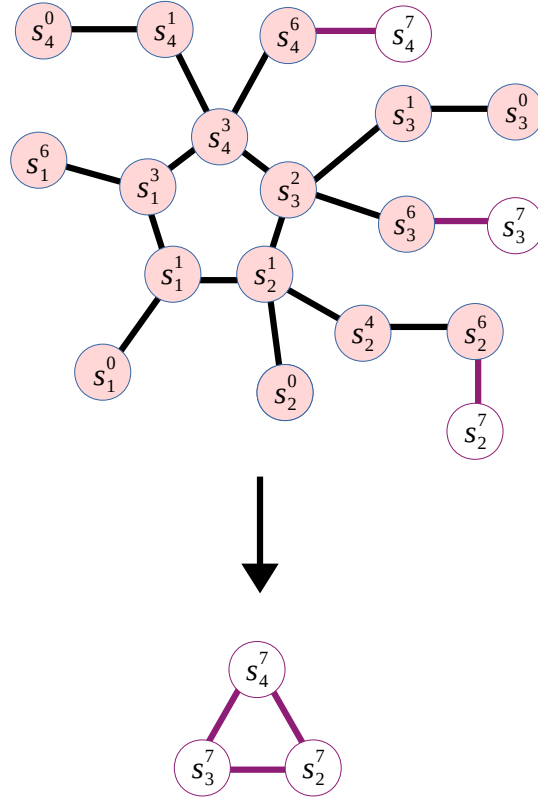


Figure 2.6: Evaluation of amplitude subsets.

Top - extended amplitude expression to evaluate all amplitudes of qubits 2, 3, 4; Bottom - resulting amplitude tensor.

2013; Marsland, 2011] when all necessary indices are eliminated and (possibly) merging the final set of tensors. Notice that the result of the evaluation of all amplitudes for c qubits will result in a tensor with 2^c elements, which will be mirrored by a clique (a fully connected subgraph) with c nodes in our graphical notation (Fig. 2.6, bottom).

After selecting a subset of nodes to leave in the result, one still faces a problem of choosing an optimal order of variable elimination to implement partial contractions. Let us turn to the discussion of a possible solution.

2.4.2 Node ordering and chordal graphs

To properly introduce the procedure of finding elimination orders for partial contractions, let us first highlight the connection of elimination orders and chordal graphs. Chordal graphs (also called triangulated graphs) are the ones that do not have cycles of length higher than 3. Many problems that are hard on general graphs can be solved on chordal graphs in polynomial time (for example, the Maximum Clique problem[Gavril, 1972]). An extensive introduction to the properties of chordal graphs and related algorithms can be found in [Blair and Peyton, 1993].

We will employ chordal graphs because of their relation to node orderings. Consider the bucket elimination procedure described before, but without node removal. Specifically, given a graph G and a node order π , one would loop over the nodes according to π and for each node connect all of its neighbors who have higher order in π . It can be shown[Blair and Peyton, 1993] that this procedure will always produce a chordal graph. Indeed, if the initial graph had any cycle with four or more nodes, connecting the neighbors of any node in the cycle will introduce a chord (a link between nodes in the cycle), thus breaking a cycle into smaller, three node cycles. The resulting chordal graph is also called a fill-in graph in this context.

A formal algorithm for building a fill-in graph H given an initial graph G and an elimina-

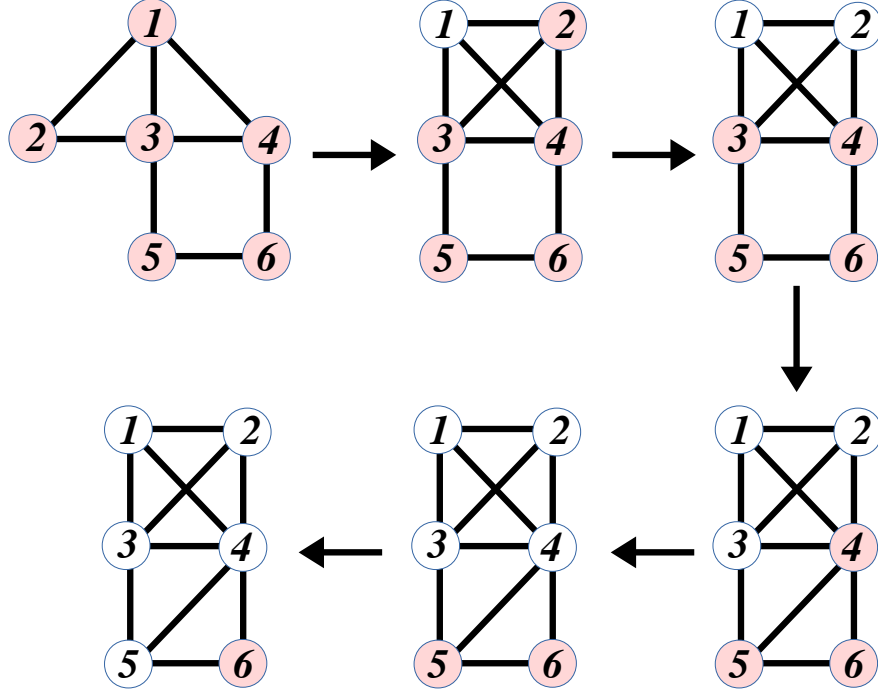


Figure 2.7: Building a chordal graph from the elimination order
The graph corresponds to the tensor network in Eq. 2.5. The nodes are labeled according to their order.

tion order π is listed in Alg. 1. A corresponding graphical representation of the algorithm is provided in Fig. 2.7. An important remark has to be made here. Any elimination order π will produce a chordal graph, but this does not imply that there is a one-to-one correspondence. Multiple orders can result in the same fill-in graph [Tarjan and Yannakakis, 1984]; we will employ this fact in the next section.

Algorithm 1 Building chordal graph from the elimination order

Input: $G = (V, E), \pi : V \rightarrow \mathcal{N}, \pi = \{(v_i, i)\}_{i=1}^{|V|}$

Output: $H = (V, \tilde{E}), H$ is chordal

```
1: function BUILD_CHORDAL_GRAPH( $G, \pi$ )
2:    $\tilde{E} \leftarrow E$ 
3:   for  $i \in [1, \dots, |V|]$  do
4:      $v \leftarrow \pi^{-1}(i)$ 
5:      $U = \emptyset$ 
6:     for  $w$  in neighbors( $v$ ) do
7:       if  $\pi(w) > i$  then
8:          $U \leftarrow U \cup w$ 
9:       end if
10:    end for
11:    for  $x, y$  in pairs( $U$ ) do
12:       $\tilde{E} \leftarrow \tilde{E} \cup (x, y)$ 
13:    end for
14:  end for
15: end function
```

The size of the maximum clique in the fill-in graph equals *treewidth* by construction [Bodlaender et al., 2006]. The problem of searching the best elimination order for a graph G thus can be formulated in terms of the search of an optimal fill-in graph. Formally, given a graph $G = (V, E)$ the task of finding an elimination order π with minimal treewidth is equivalent to finding a chordal graph $H = (V, \tilde{E}), \tilde{E} \in E$, such that the size of its maximum clique is minimized.

Provided a chordal graph H is found, any of its elimination orders that does not introduce

additional edges, and hence does not change the graph H , will have the same treewidth. Chordal graphs thus provide means of building equivalent (in terms of treewidth) elimination orders. In contrast with arbitrary graphs, finding elimination orders of chordal graphs can be done in linear time [Tarjan and Yannakakis, 1984]. We now turn to the description of the procedure of building of equivalent elimination orders of chordal graphs.

2.4.3 Finding restricted elimination orders

Let us now find an optimal elimination order for multiple amplitude evaluation, as described in Sec. 2.4.1. In essence, we want to find an order with minimal treewidth, such that some set of nodes will be at the end of this order. Putting it formally, for a graph $G = (V, E)$ and a subset of nodes $C \in V$ we want to find an order π with minimal treewidth, such that for any nodes $v \in C$ and $w \in V \setminus C$ the order of v is higher than the order of w : $\pi(v) > \pi(w)$.

Our idea is to calculate an optimal unrestricted elimination order $\tilde{\pi}$ (not necessary having C at the end), and then to use the connection between the elimination orders and chordal graphs to transform it to the desired order π . Essentially, we employ the result of Bodlaender [Bodlaender et al., 2006], to devise a procedure for building π . Our approach is outlined below:

1. Check if C induces a clique in G . If $G[C]$ is not a clique, turn $G[C]$ into a fully connected subgraph. This step ensures that the condition stated in (Ref. [Bodlaender et al., 2006], Lemma 10) is satisfied: if C is a clique, then there always exists an elimination order with C at the end. A graph \tilde{G} is produced as a result of (possibly) turning C into a clique.
2. Find an elimination order $\tilde{\pi}$ of \tilde{G} using an exact (NP-hard) or a heuristic algorithm. We use the branch and bound algorithm of Gogate [Gogate and Dechter, 2004] with the time limit of 60 seconds (to obtain an exact solution the algorithm has to be given a very long time).

3. Build a chordal graph H using Alg. 1.
4. Provided with a set C and a chordal graph H , construct a new order π with the help of the Restricted Maximum Cardinality Search (MCS) algorithm. The order π has same treewidth as the order $\tilde{\pi}$ and nodes in C are placed at the end in π .

Essentially, in our approach, we transform an arbitrary solution to the Tree decomposition problem to the one that satisfies our restrictions (places all nodes in C to the end) and has the same quality (same treewidth). The last ingredient to complete the procedure is the Restricted Cardinality Search algorithm. We modified the original algorithm from Ref. [Tarjan and Yannakakis, 1984]. The resulting pseudocode is provided in Alg. 2.

Algorithm 2 Computing an elimination order with a set of nodes C at the end

Input: $H = (V, E)$, H is chordal, $C \in V$, C is clique**Output:** $\pi : V \rightarrow \mathcal{N}$, $\pi = \{(v_i, i)\}_{i=1}^{|V|}$

```
1: function RESTRICTED-MCS( $H, C$ )
2:   for  $v \in V$  do
3:     cardinality( $v$ )  $\leftarrow 0$ 
4:   end for
5:   for  $i \in [|V|, |V| - 1, \dots, 1]$  do
6:     if  $C \neq \emptyset$  then
7:       pick  $v \in C$ ,  $C \leftarrow C \setminus \{v\}$ 
8:     else
9:       pick  $v \in V$  with maximum cardinality
10:       $V \leftarrow V \setminus \{v\}$ 
11:    end if
12:     $\pi \leftarrow \pi \cup (v, i)$ 
13:    for  $w \in \text{neighbors}(v)$ ,  $w \notin \pi$  do
14:      cardinality( $w$ )  $\leftarrow \text{cardinality}(w) + 1$ 
15:    end for
16:  end for
17: end function
```

In the Alg. 2 each node v of the graph H is assigned a counter "cardinality", which holds the number of labeled neighbors of v . At each step, we label the next node with maximal cardinality, breaking ties arbitrarily. The order is built in a reversed form. In the beginning, nodes in C are labeled (to be last), and then the rule stated before is applied. Note that if at step i a node v is selected, then in the next steps all neighbors of v , which belong to the

maximal clique K , $v \in K$ will be labeled. Overall, the procedure in Alg. 2 is polynomial in the number of nodes $|V|$.

Let us now demonstrate the benefits of the proposed approach with numerical examples.

2.4.4 Numerical examples

The methods developed in previous sections were used to implement a quantum circuit simulator. As numerical examples we use the simulation of random quantum circuits from the work of Boixo et al. The qubits are arranged in a square grid of size $k \times k$ and a set of gates $\{X^{1/2}, Y^{1/2}, cZ, T, H\}$ is applied in a predefined pattern. This circuit choice can be implemented in superconducting quantum processors. [Chen et al., 2014] The reader is referred to [Boixo et al., 2017] to learn more details about the motivation of these random circuits. The dataset with random circuits of Boixo et al., which we used in this work, is available online [Boixo, 2019]. Here we are interested only in the numerical cost and the memory usage to evaluate amplitudes.

In Fig. 2.8, the dependence of treewidth ν on the size and depth of the random circuits is shown. Let us recall that the number of floating-point operations scales as $O(2^{\nu+1})$ and the required memory as $O(2^\nu)$. The complexity of simulation grows exponentially with the volume of random quantum programs, which was the original motivation for considering them as a test bench for demonstrations of quantum supremacy [Boixo et al., 2018].

In Fig. 2.9, the advantage of batch simulation comparing to single amplitude at a time is shown. The steep growth of the flop cost is significantly ameliorated. We recall that a clique C is introduced into the computational graph when the evaluation of all amplitudes of $|C|$ qubits is performed. While this clique is less than the treewidth of the computational graph, there is only a negligible increase in the computational cost. Batch simulation, however, requires copious amounts of memory, as shown in Fig. 2.10. The results we obtain illustrate a usual CPU/memory trade-off seen in numerical algorithms. Notice also that the curves for

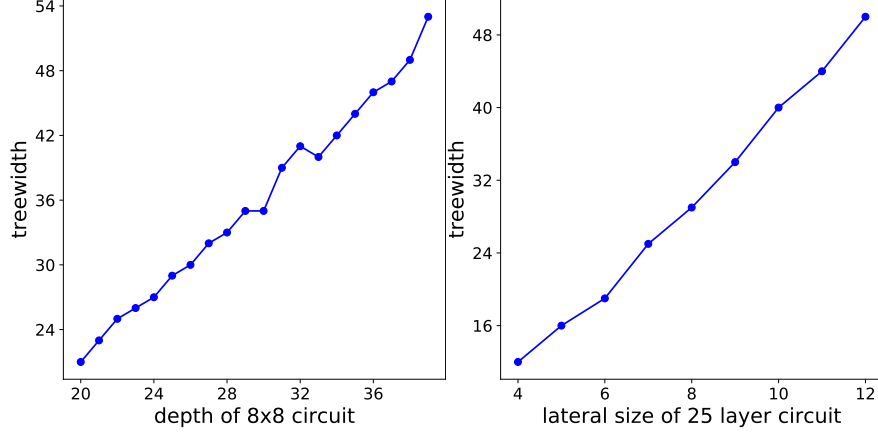


Figure 2.8: Treewidth dependence on the size of a random quantum circuit. Left - dependence of treewidth on the depth of a random circuit, Right - dependence of treewidth on the number of qubits.

total amount of memory and flop are almost indistinguishable. This result is caused by the fact that during the evaluation of the amplitudes one needs to contract high order tensors over an index of size 2: the flop cost of the most expensive contraction equals the size of the largest tensor times 2.

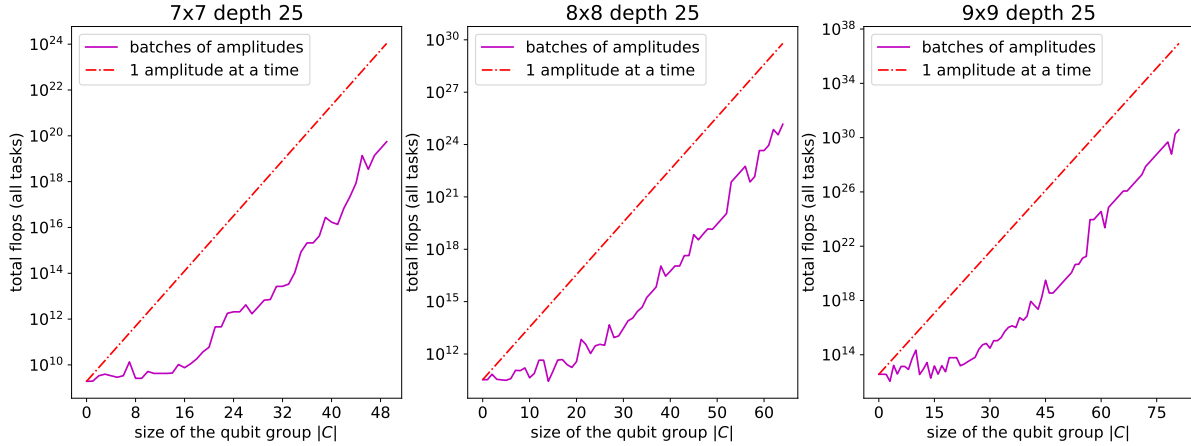


Figure 2.9: Total flop requirements for the simulation of a typical random circuit of varying size. Estimated number of floating point operations for the simulation of the full subset of amplitudes of $|C|$ qubits (there are $2^{|C|}$ amplitudes). In case of one amplitude at a time simulation a combined cost of all tasks is drawn.

Lastly, we provide the dependence of the number of operations per memory access for circuits of different sizes in Fig. 2.11. In all cases, the values are in the range of $O(L)$, where

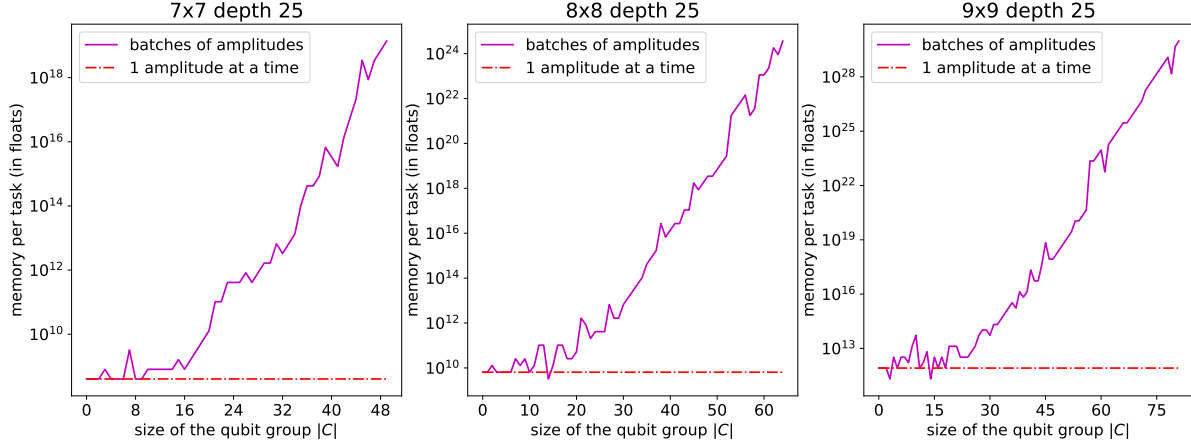


Figure 2.10: Minimal memory requirement for the simulation of a typical random circuit of varying size

Shown is the predicted number of memory in floating point units per single task (simulation of either 1 amplitude or a batch of amplitudes). Notice the high similarity with Fig. 2.9: the ratio of flop to memory access is almost constant in logarithmic scale.

$L = 2$ for qubits. This dependence demonstrates that despite a potential for massive parallelism [Chen et al., 2018b], the problem of tensor contraction is essentially memory bound, and an efficient algorithm has to very carefully overlap data transmission and computations. Comparing to the contraction of matrices, where extremely efficient algorithms were developed [Dongarra et al., 1988] using CPU cache and vectorized operations, a general tensor contraction is more challenging for optimization.

2.5 Conclusion and comparison

We introduced a novel way to optimize graphical model algorithms for quantum circuits simulation. Our approach allows the user to select between the amount of memory consumed and the speed of the calculation; thus, the code can be adapted to the available hardware. We emphasize that our approach is not restricted to quantum circuit simulation, but can be used to evaluate partial contractions of general tensor networks. To our best knowledge, this is a first of a kind method which evaluates partial contractions efficiently, e.g., its resource requirements depend only on the treewidth of the expression’s graph.

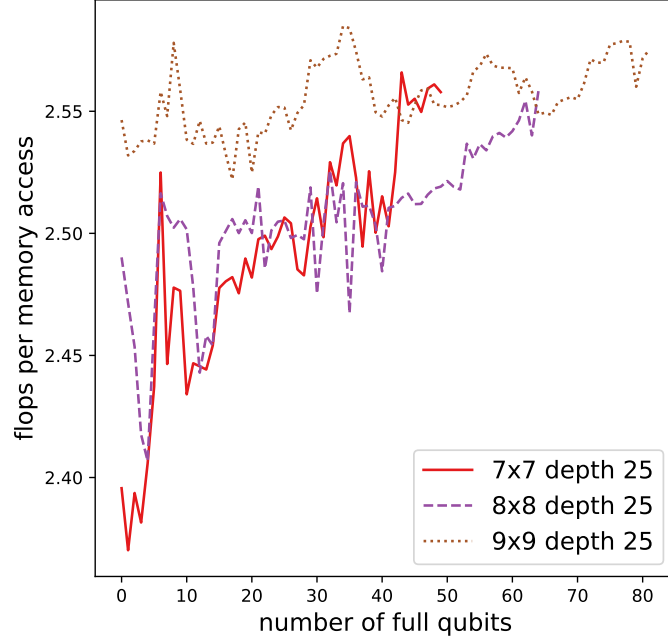


Figure 2.11: Floating point to memory access ratio during the simulation of random circuits of varying size

Many more improvements to the tensor contraction strategy can be proposed. Here we explicitly avoided the discussion of parallel simulation, and defer it to a future work. Also, since the structure of modern computer memories combines devices with varying latency and size, some research is needed on the proper scheduling of the operations, especially in the parallel case.

CHAPTER 3

DIAGONAL GATES APPROACH FOR OPTIMIZING QUANTUM CIRCUIT SIMULATION

This chapter is adapted from Lykov and Alexeev [2021].

3.1 Abstract

In this work we present two techniques that increase the performance of tensor-network based quantum circuit simulations. The techniques are implemented in the QTensor package and benchmarked using Quantum Approximate Optimization Algorithm (QAOA) circuits. The techniques allowed us to increase the depth and size of QAOA circuits that can be simulated. In particular, we increased the QAOA depth from 2 to 5 and the size of a QAOA circuit from 180 to 244 qubits. Moreover, we increased the speed of simulations by up to 10 million times. Our work provides important insights into how various techniques can dramatically speed up the simulations of circuits.

3.2 Introduction

Several approaches have been employed to simulate quantum circuits. The major types include the state-vector evolution approach [De Raedt et al., 2007; Smelyanskiy et al., 2016; Häner and Steiger, 2017; Wu et al., 2019, 2018b,a], linear algebra open system simulation [Otten, 2020], and tensor network contractions [Markov and Shi, 2008; Pednault et al., 2017; Boixo et al., 2017; Lykov et al., 2020b]. All these simulators have various advantages and disadvantages. For example, the state-vector evolution approach, while being relatively easy to implement, has an exponential memory requirement with respect to the number of qubits in the circuit, which is a major bottleneck preventing quantum simulations beyond approximately 46 qubits on modern supercomputers.

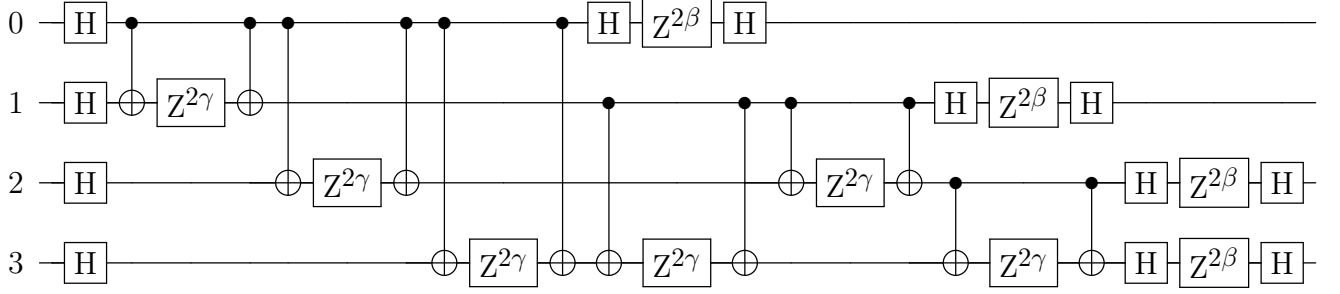


Figure 3.1: Quantum circuit that generates QAOA ansatz state for MaxCut problem on a 4-node complete graph. This widely used decomposition of QAOA into common set of basis gates is not optimal for the classical simulation of the output state.

In our opinion the most promising type of simulator is the tensor network contraction approach. It is especially efficient for simulating shallow-depth circuits. This approach can be sensitive to the connectivity of a quantum circuit and the types of gates. In this chapter we describe the tensor network simulator implementation and show two optimization techniques that enable dramatic speedup of simulations. We use quantum circuits from the Quantum Approximate Optimization Algorithm (QAOA) algorithm since it is a promising candidate for demonstrating quantum advantage and benchmarking quantum devices.

All simulations in our work used QTensor [Lykov, 2021], developed at Argonne National Laboratory. It is a quantum circuit simulator that uses a tensor network contraction approach with a special focus on the simulation of QAOA circuits. It supports simulating both probability amplitudes and energy expectation values.

In the following section we introduce the tensor network contraction approach and describe the QAOA quantum circuits. In particular, we show how the usage of the Feynman path formalism provides the possibility for optimization. We then describe the optimization techniques and the resulting speedup of simulations. The final section contains our conclusions and further directions of research.

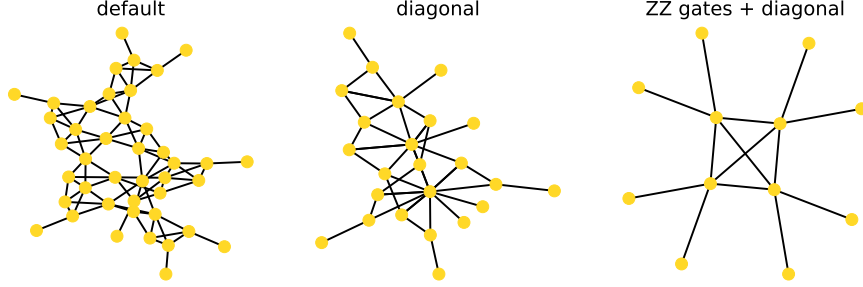


Figure 3.2: Line graphs of tensor networks for calculating QAOA ansatz state using different optimizations. “Default” and “diagonal” show line graphs of tensor network for the circuit shown in Figure 3.1, using a full-matrix gates and diagonal gates approach, respectively. “ZZ gates + diagonal” is obtained by using the diagonal gates approach on a simplified quantum circuit obtained by applying Equation 3.4. This figure demonstrates how improving the conversion of a quantum algorithm to a tensor network can reduce the complexity of the network, providing speedups for both finding contraction order and the contraction itself.

3.3 QAOA algorithm

The QAOA algorithm, introduced by Farhi and Goldstone in 2016 [Farhi and Harrow, 2016], is a seminal hybrid quantum-classical algorithm for approximate optimization. The algorithm can be used to find approximate solutions to NP-complete combinatorial optimization problems. Here we will demonstrate how it works to solve the MaxCut problem and all benchmarking simulations are performed for MaxCut, but this work also applies to other types of tensor network simulations.

The goal of the MaxCut problem is to color all vertices of a given graph $G = (V, E)$, such that the number of edges between the two resulting parts of the graph is maximized. The cost function for MaxCut is $C = \frac{1}{2} \sum_{\langle jk \rangle \in E} -Z_j Z_k + 1$, where Z_i is a label of each vertex. To solve this problem using QAOA, one has to find optimal parameters for the parametrized ansatz state $|\gamma\beta\rangle$ such that the expectation value of the Hamiltonian cost function $\langle\gamma\beta|\hat{H}|\gamma\beta\rangle$ is maximized. The ansatz state depends on two parameter vectors γ and β . The length of the parameter vector, denoted p , is an important parameter that defines the quality of the solution. For a QAOA depth p with MaxCut on graph $G = (V, E)$, the ansatz state is equal

to

$$|\gamma\beta\rangle_p = \prod_{k=1}^p U_C(\gamma_k) U_B(\beta_k) |+\rangle, \quad (3.1)$$

where $U_B(\beta) = e^{-i\beta \sum_{j \in V} X_j}$ and $U_C(\gamma) = e^{-i\frac{\gamma}{2} \sum_{(i,j) \in E} (I - Z_i Z_j)}$. Substituting U_B and U_C into (3.1) and discarding the global phase, we obtain

$$|\gamma\beta\rangle_p = \prod_{q=1}^p \left[\prod_{ij \in E, k \in V} e^{i\gamma_q Z_i Z_j} e^{-i\beta_q X_k} \right] |+\rangle. \quad (3.2)$$

A quantum circuit that generates the ansatz state for MaxCut on a fully connected 4-node graph is shown in Figure 3.1. For a more detailed description of QAOA see [Farhi and Harrow, 2016].

The optimal γ, β parameters correspond to the minimum of the Hamiltonian cost function expectation value, which can be calculated as $E = \langle \gamma\beta | C | \gamma\beta \rangle$. Note that C is a sum of $|E|$ elements, where each element corresponds to an edge of the original graph on which we solve MaxCut. Hence, the expectation value is $E = \sum_k E_k$, where each element is a matrix element of a local operator that acts on two qubits. This locality gives room for efficient optimization by canceling all the conjugate gates that commute through that local operator. This optimization, called lightcone optimization, was introduced by Farhi and Goldstone in [Farhi and Harrow, 2016].

The energy calculation is an important part of the QAOA method, since one can use a classical computer to optimize the γ, β parameters without having to use noisy quantum devices. In this work, however, we focus on simulating a single amplitude of the ansatz state as a benchmark for demonstrating the optimizations. The results for energy simulations will be discussed in future works.

3.4 Methodology

3.4.1 Tensor network approach

Quantum computers operate by applying gates to a quantum state that describes a quantum system consisting of N subsystems (qubits). One way to describe the evolution of the quantum system is to apply quantum gates in matrix form on the wavefunction in the form of a state vector. With the state-vector evolution approach, each gate action should be described as an operator that acts on the whole system, even if the gate is local to a certain subsystem of qubits. As a result, the whole state vector needs to be stored, which is extremely inefficient in terms of computational resources and memory.

The tensor network approach associates a state vector of the system with a tensor of N indices. Each index in this tensor labels the state of a particular subsystem. That is, the dimension of the index is equal to number of states of the subsystem, which is always 2 in our case. Each gate that acts on a subsystem can then be described as a tensor. The amplitudes of the resulting state can be calculated by summing the product of the state tensor and the operator tensor over the index of the subsystem.

For example, given a system of two qubits and operator \hat{X}_0 acting on the first qubit, the resulting state in the state-vector formalism would be $|\phi\rangle = \hat{X}_0 \otimes \hat{I}_1 |\psi\rangle$. In tensor network representation, this equation is $\phi_{i'j} = X_{i'i} \psi_{ij}$.

If the system is in a product state, then the corresponding state-tensor is a product of smaller tensors representing each subsystem state. In particular, a 2-qubit system in a state $|0\rangle$ is represented by $\psi_{ij} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}_i \begin{pmatrix} 1 \\ 0 \end{pmatrix}_j$. Note that the size of this object is $2N$ compared with that of 2^N in the state-vector notation. More details on tensor network formalism are available in [Cichocki et al., 2016].

3.4.2 Tensor network contraction

Simulation of probability amplitudes of a quantum state can be done by contracting a tensor network that represents the quantum circuit that generates the state.

The contraction of a general tensor network can be written by using a line graph approach as following:

$$R_{i_1, \dots, i_p} = \sum_{j_1, \dots, j_q} \prod_{e_i \in F} W_{e_i}^i, \quad (3.3)$$

where tensor indices $i, \dots, j, \dots \in U$ are represented by vertices of a hypergraph $L = (U, F)$, $e \in F$ is a hyperedge of L , edges are tuples of indices $e_i = (v_1, v_2, \dots, v_d), \forall k \ v_k \in U$, and tensors $W_{e_i}^i$ have the number of dimensions d , the same as the number of vertices in a corresponding edge. For two-level quantum systems, where each tensor dimension has size 2, the sum (3.3) has 2^q elements, and each element corresponds to assignment of 0, 1 to each variable.

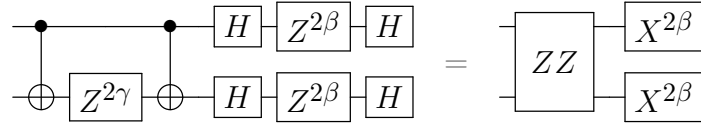
Instead of calculating every element of this sum, one can merge tensors with each other, producing an intermediary tensor after each merge operation. One way to do this is by selecting some vertices j_i from all contracted vertices $\{j_k\}_{k=1}^q$ and evaluating values of a new intermediary tensor by summing over j_i a product of only those tensors that have j_i as their index. The order in which the j_i are selected determines the size of the largest tensor that needs to be stored as an intermediate step in the contraction. Thus the total contraction speed and memory requirements are determined by this intermediate largest tensor, which can reach a very large number of dimensions. More information on ordering tensor networks is available in [Schutski et al., 2020], [Lykov et al., 2020b], [Dechter, 2013].

3.5 Optimization techniques

To speed up and reduce memory requirements of tensor network contraction, we applied two techniques, which we describe in detail in this section. In the first technique we combined gates, and in the second technique we took advantage of the diagonal properties of the gates.

3.5.1 Optimization of QAOA circuit structure

A typical 4-qubit QAOA circuit is shown in Figure 3.1. We note that Pauli-Z gates are enclosed by two CNOT gates or by Hadamard gates. One can merge these gates in a specialized 2-qubit gate with a parameter γ or parameter β respectively, as shown in the equations below.



$$\text{CNOT}_{1,2} \cdot Z^{2\gamma}_2 \cdot \text{CNOT}_{1,2} \cdot H_1 \cdot Z^{2\beta}_1 \cdot H_1 \cdot H_2 \cdot Z^{2\beta}_2 \cdot H_2 = ZZ_{1,2} \cdot X^{2\beta}_1 \cdot X^{2\beta}_2 \quad (3.4)$$

$$\hat{Z}\hat{Z} = e^{i\gamma\hat{Z}_i\hat{Z}_j} \quad (3.5)$$

This gate optimization technique reduces the complexity of the tensor network line graph, as shown in Figure 3.2. It also makes finding an optimal tensor contraction sequence easier since the line graph has fewer vertices.

3.5.2 Diagonal gate simplification

A certain property of tensors $W_{e_i}^i$ can provide an opportunity for optimization in terms of how these tensors are stored and computed. Each index of sum (3.3) can have a value 0 or 1 for an N -qubit system, and each assignment of values to indices corresponds to a single Feynman path which evaluates to an element of the sum (3.3). Since the value of each

Feynman path is a product of values of different tensors, we know in advance that if for some assignment the value of any tensor is 0, the whole contribution is 0 as well.

In particular, if a tensor W^{i_0} from Equation 3.3 is diagonal, in other words $W_{lm}^{i_0} = \alpha_l \delta_{lm}$, then for any assignment of indices $(l, m) = e_{i_0}$, $l, m \in U$ from sum (3.3) in which values of the diagonal tensor indices match, the corresponding element in the sum will be equal to zero. The tensor W^{i_0} can then be safely removed from the tensor network and replaced by α_l without changing the result.

Here is a 2-gate example demonstrating how our diagonalization technique is applied.

$$\begin{aligned}
|\phi\rangle &= \hat{U} \hat{D} |\psi\rangle \phi_i = \sum_{jk} U_{ij} D_{jk} \psi_k \\
&= \sum_{jk} U_{ij} \alpha_j \delta_{jk} \psi_k = \sum_j U_{ij} \alpha_j \psi_j
\end{aligned} \tag{3.6}$$

We note that QAOA circuits have only one type of a 2-qubit gate: $\hat{Z}\hat{Z} = e^{i\gamma} \hat{Z}_i \hat{Z}_j$. The \hat{Z} gate is diagonal, as is $\hat{Z}_i \hat{Z}_j$; therefore, the matrix of the $\hat{Z}\hat{Z}$ gate in the 2-qubit basis will be diagonal.

$$Z\hat{Z}(\gamma) = e^{i\gamma} \hat{Z}_i \hat{Z}_j = \text{diag}(e^{i\gamma}, e^{-i\gamma}, e^{-i\gamma}, e^{i\gamma}) \tag{3.7}$$

We can use this fact to replace the 4-index tensor U_{ijkl} representing a generic 2-qubit gate with a 2-index tensor $U_{ij} = \begin{pmatrix} \rho & \bar{\rho} \\ \bar{\rho} & \rho \end{pmatrix}$, $\rho = e^{i\gamma}$ that represents a diagonal gate. This significantly reduces the computational cost for tensor contraction and the memory requirements to store these tensors.

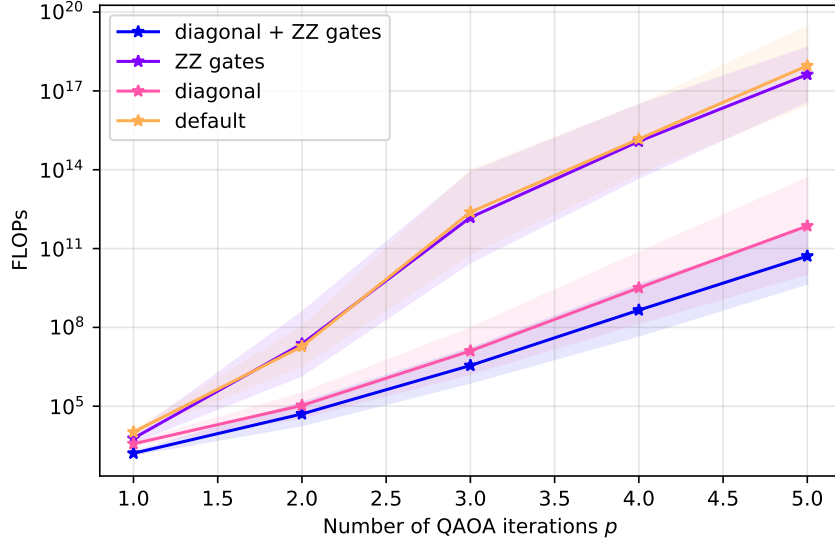


Figure 3.3: Number of FLOPs to calculate a single amplitude of QAOA ansatz state for MaxCut using a different number of QAOA iterations. Each line shows a combination of optimization techniques, with “diagonal + ZZ gates” being the most advanced one. The shaded region shows 1- σ interval over 5 random graphs.

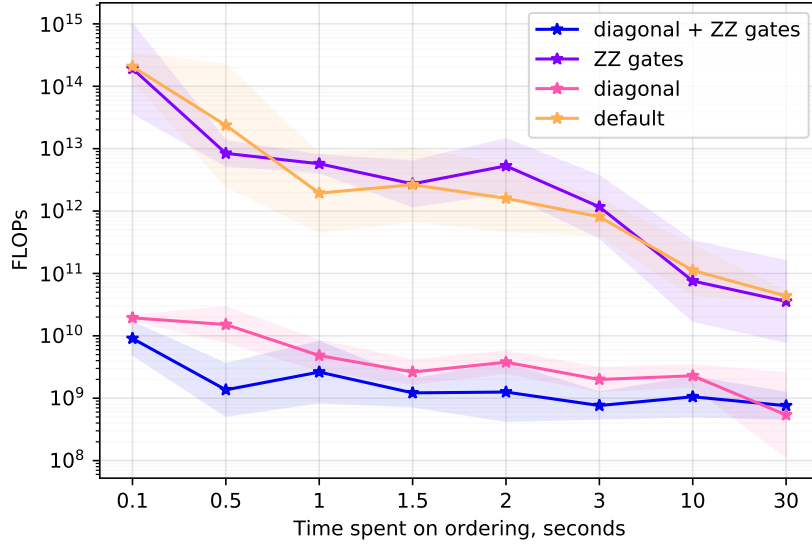


Figure 3.4: Number of FLOPs to calculate a single amplitude of QAOA ansatz state with $p=1$ for MaxCut on random 3-regular graphs with 160 nodes. The shaded region shows 1- σ interval over 5 random graphs.

3.6 Results

In this section we evaluate the simulation complexity of different combinations of quantum circuits. For this work we used the number of FLOPs required for contraction as the main metric of simulation complexity. We used only the FLOPs metric because it is easy to use and FLOPs and memory requirements are highly correlated. The number of FLOPs can be estimated as 2^C , where C is a contraction width, or the number of dimensions of the largest intermediary tensor. Thus, one can safely assume that the maximum memory in bytes for the contraction will be $16 * 2^C$, positing the size of a single complex number to be 16 bytes.

First, we selected 5 random 3-regular graphs, for which we formulated the MaxCut problem by using QAOA. Then, each graph was used to generate quantum circuits that produce QAOA ansatz. There are two types of circuits: one using ordinary decomposition of 2-qubit gates into three gates and another using the $\hat{Z}\hat{Z}$ gate simplification, as described in Section 3.5.1, where the complexity of the circuit is reduced. For each quantum circuit, we constructed a tensor network using two approaches: with diagonal simplification and without it, as described in Section 3.5.2. The tensor network was then sliced to produce the first amplitude of the ansatz state when all its indices are contracted. We then used the **rgreedy** algorithm from the QTensor package with **n_repeats**=10 and **temp**=0.02 to obtain a contraction sequence, which we used to estimate the number of FLOPs required for the contraction. Dependence of the contraction complexity from the contraction ordering time is shown in Figure 3.4. The experiments in this chapter aim to demonstrate the difference between simulation of quantum circuits optimized using different techniques, rather than absolute values of the simulation cost. Hence, we pick relatively modest ordering algorithm parameters that result in 1 second of ordering time.

Further analysis of contraction complexity is shown in Figure 3.3. In this figure, the number of FLOPs for simulating a single amplitude of QAOA ansatz circuit versus the number of QAOA iterations p is shown. The data in the plot is evaluated for five random 3-regular

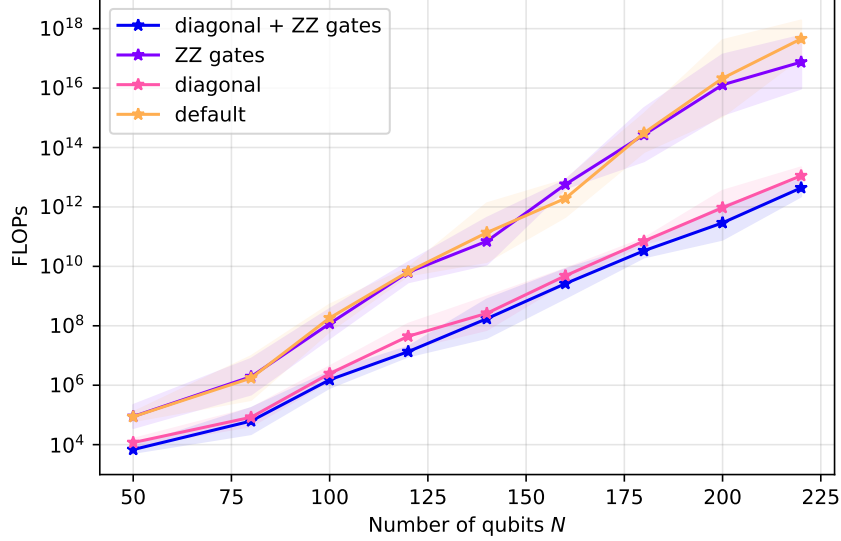


Figure 3.5: The number of FLOPs to calculate a single amplitude of QAOA ansatz state for MaxCut on random 3-regular graphs of different sizes. The number of nodes in the graph corresponds to the number of qubits in the quantum circuit. Each line shows a combination of optimization techniques, with “diagonal + ZZ gates” being the most advanced one. The shaded region shows 1- σ interval over 5 random graphs.

graphs. From the figure one can see that diagonal gates and $\hat{Z}\hat{Z}$ gate optimization techniques dramatically reduce the number of FLOPs compared with the original unoptimized tensor network graphs.

We also analyzed how the number of FLOPs is correlated with the size of the circuit in terms of the number of qubits N . Figure 3.5 shows the number of FLOPs for ansatz simulation of QAOA MaxCut on 3-regular graphs of different sizes and fixed $p=1$. We note that improvement in performance grows with the size of a circuit, showing that the diagonal gate simplification proportionally reduces the contraction width C , while the $\hat{Z}\hat{Z}$ gates simplification produces a similar amount of improvement for all sizes.

From the analysis of Figures 3.3, 3.4, and 3.5, it is surprising to find out that using only $\hat{Z}\hat{Z}$ gate optimization is not enough to get significant computational and memory savings. We note that using the diagonal gates is what really provides dramatically better results, especially when combined with gate optimization.

	Default	With diagonal	With ZZ gates and diagonal
Laptop	2	3	3
Supercomputer	2	4	5

Table 3.1: The maximum number of QAOA iterations p for which one can simulate a single amplitude of ansatz state for MaxCut on a 40-node random regular graph.

	Default	With diagonal	With ZZ gates and diagonal
Laptop	118	156	160
Supercomputer	180	240	244

Table 3.2: The maximum number of nodes of a 3-regular graph for which one can simulate a single amplitude of the MaxCut QAOA ansatz state.

The simulation of quantum circuits is usually a memory-bound task because of memory requirements to store an intermediate tensor. The optimization techniques described in this chapter allow simplification of a tensor network structure with $\hat{Z}\hat{Z}$ and diagonal gates, which in turn helped us find the optimal tensor contraction sequence. As a result, we extended the depth p and the number of qubits N of the largest circuits that is feasible to simulate using a laptop or a supercomputer, as shown in Tables 3.1 and 3.2. A laptop is assumed to have 4 GB of memory, and a supercomputer is assumed to have 800 TB of aggregated memory.

Our optimization techniques also can be applied to the simulation of the QAOA energy expectation value. Thus, it would allow us to find parameters for even larger circuits with higher depth.

3.7 Conclusions

Here, we implemented $\hat{Z}\hat{Z}$ gate optimization and diagonal gate techniques and demonstrated that can lead to dramatic savings in terms of memory and computation requirements. As a result, we were able to simulate much larger QAOA circuits both in size and in depth, as shown in Tables 3.1 and 3.2.

We were able to increase p from 2 to 5 and the size of a QAOA circuit N from 180 to 244 qubits on a supercomputer. These numbers were estimated without actually running on

a supercomputer, but the laptop results have been actually verified.

The tensor network contraction method for simulation of quantum circuits is a powerful approach that has the potential to simulate very large quantum circuits. At the same time, however, a lot of improvements, simplifications, and approximations can be used to improve the simulations and dramatically decrease memory and computational requirements. For example, in this work we sped up quantum simulations by up to 10 million times, as can be seen in Figure 3.3 in the “diagonal+ZZ gates” curve versus “default” curve for $p = 5$. Moreover, there is room to speed up simulations by an even larger factor. This work underscores how one needs to be careful when comparing tensor network simulations against classical solvers and quantum hardware for demonstration of quantum supremacy and advantage. Our work provides important insights into how various optimization techniques can speed up tensor network simulations and what other techniques can be used to achieve this goal.

CHAPTER 4

PARALLEL COMPUTATION

This chapter is adapted from Lykov, Schutski, Galda, Vinokur, and Alexeev [2020b].

4.1 Introduction

In this chapter, we explore the limits of classical computing using a supercomputer to simulate large QAOA circuits, which in turn helps to define the requirements for a quantum computer to beat existing classical computers.

Our main contribution is the development of a novel slicing algorithm and an ordering algorithm. These improvements allowed us to increase the size of simulated circuits from 120 qubits to 210 qubits on a distributed computing system, while maintaining the same time-to-solution.

In Section 4.2 we start by discussing related work. In Section 4.4 we describe tensor networks and the bucket elimination algorithm. Simulations of a single amplitude of QAOA ansatz state are described in Section 4.5. We introduce a novel approach *step-dependent slicing* to finding the slicing variables, inspired by the tensor network structure. Our algorithm allows simulating several amplitudes with little cost overhead, which is described in Section 4.7.

We then show the experimental results of our algorithm running on 64-1,024 nodes of Argonne’s Theta supercomputer. All these results are described in Section 3.6. In Section 4.9 we summarize our results and draw conclusions.

4.2 Related Work

In recent years, much progress has been made in parallelizing state vector [Häner and Steiger, 2017; Smelyanskiy et al., 2016; Wu et al., 2019] and linear algebra simulators [Otten, 2020].

Very large quantum circuit simulations were performed on the most powerful supercomputers in the world, such as Summit [Villalonga et al., 2020], Cori [Häner and Steiger, 2017], Theta [Wu et al., 2019], and Sunway Taihulight [Li et al., 2018]. All these simulators have various advantages and disadvantages. Some of them are general-purpose simulators, while others are more geared toward short-depth circuits.

One of the most promising types of simulators is based on the tensor network contraction technique. This idea was introduced by Markov and Shi [2008] and was later developed by Boixo et al. [2017] and other authors [Schutski et al., 2020]. Our simulator is based on representing quantum circuits as tensor networks.

Boixo et al. [2017] proposed using the line graphs of the classical tensor networks, an approach that has multiple benefits. First, it establishes the connection of quantum circuits with probabilistic graphical models, allowing knowledge transfer between the fields. Second, these graphical models avoid the overhead of traditional diagrams for diagonal tensors. Third, the treewidth is shown to be a universal measure of complexity for these models. It links the complexity of quantum states to the well-studied problems in graph theory, a topic we hope to explore in future works. Fourth, straightforward parallelization of the simulator is possible, as demonstrated in the work of Chen et al. [Chen et al., 2018b]. The only disadvantage of the line graph approach is that it has limited usability to simulate subtensors of amplitudes, which was resolved in the work by Schutski et al. [Schutski et al., 2020]. The approach has been studied in numerous efficient parallel simulations relevant to this work [Chen et al., 2018b; Li et al., 2018; Pednault et al., 2017; Schutski et al., 2020].

4.3 Methodology

4.3.1 QAOA introduction

The combinatorial optimization algorithms aim at solving a number of important problems. The solution is represented by an N -bit binary string $z = z_1 \dots z_N$. The goal is to determine a string that maximizes a given classical objective function $C(z) : \{+1, -1\}^N$. The QAOA goal is to find a string z that achieves the desired approximation ratio:

$$\frac{C(z)}{C_{max}} \geq r$$

where $C_{max} = \max_z C(z)$.

To solve such problems, QAOA was originally developed by Farhi et al. [2014]. In this paper, QAOA has been applied to solve MaxCut problem. It was done by reformulating the classical objective function to quantum problem with replacing binary variables z by quantum spin σ^z resulting in the problem Hamiltonian H_C :

$$H_C = C(\sigma_1^z, \sigma_2^z, \dots, \sigma_N^z)$$

After initialization of a quantum state $|\psi_0\rangle$, the H_C and a mixing Hamiltonian H_B :

$$H_B = \sum_{j=1}^N \sigma_x^j$$

is then used as to evolve the initial state p times. It results in the variational wavefunction, which is parametrized by $2p$ variational parameters β and γ . The ansatz state obtained after p layers of the QAOA is:

$$|\psi_p(\beta, \gamma)\rangle = \prod_{k=1}^p e^{-i\beta_k H_B} e^{-i\gamma_k H_C} |\psi_0\rangle$$

To compute the best possible QAOA solution corresponding to the best objective function value, we need to sample the probability distribution of 2^N measurement outcomes in state $|\gamma\beta\rangle$. The noise in actual quantum computers hinders the accuracy of sampling, resulting in the need of even a larger number of measurements. At the same time, sampling is an expensive process that needs to be controlled. Only a targeted subset of amplitudes need to be computed because sampling all amplitudes will be very computationally expensive and memory footprint prohibitive. As a result, the ability of a simulator like QTensor to effectively sample certain amplitudes is a key advantage over other simulators.

The important conclusion by Farhi et al. [2014] was that to compute an expectation value, the complexity of the problem depends on the number of iterations p rather than the size of the graph. This is a result of what is known as lightcone optimization. It has a major implication to the speed of a quantum simulator computing QAOA energy, but this type of optimization is not applicable for simulating ansatz state, which is the type of simulation we focus in this paper. A more detailed MaxCut formulation for QAOA was provided by Wang et al. [Wang et al., 2018]. It is worth mentioning that there is a direct relationship between QAOA and adiabatic quantum computing, meaning that QAOA is a Trotterized adiabatic quantum algorithm. As a result, for large p both approaches are the same.

4.3.2 Description of quantum circuits

A classical application of QAOA for benchmarking and code development is to apply it to Max-Cut problem for random 3-regular graphs. A representative circuit for a single-depth QAOA circuit for a fully connected graph with 4 nodes, is shown in Fig. 3.1. The generated circuit were converted to tensor networks as described in Section 4.4.1. The resulting tensor network for the circuit in 3.1 is shown in Fig. 4.2. Every vertex corresponds to an index of a tensor of the quantum gate. Indices are labeled right to left: 0 – 3 are indices of output statevector, and 32 – 25 are indices of input statevector. Self-loop edges are not shown

(in particular $Z^{2\gamma}$, which is diagonal). We simulated one amplitude of state $|\vec{\gamma}, \vec{\beta}\rangle$ from the QAOA algorithm with depth $p = 1$, which is used to compute the energy function. The full energy function is defined by $\langle \vec{\gamma}, \vec{\beta} | \hat{C} | \vec{\gamma}, \vec{\beta} \rangle$ and is essentially a duplicated tensor expression with a few additional gates from \hat{C} . The full energy computation corresponds to the simulation of a single amplitude of such duplicated tensor expression.

4.4 Overview of simulation algorithm

In this section, we briefly introduce the reader to the tensor network contraction algorithm. It is described in much more detail in the paper by Boixo et al. [Boixo et al., 2017], and the interested reader can refer to work by Detcher et al. [Detcher, 2013] and Marsland et al. [Marsland, 2011] to gain an understanding of this algorithm in the original context of probabilistic models.

4.4.1 Quantum circuit as tensor expression

A quantum circuit is a set of gates that operate on qubits. Each gate acts as a linear operator that is usually applied to a small subspace of the full space of states of the system. State vector $|\psi\rangle$ of a system contains probability amplitudes for every possible configuration of the system. A system that consists of n two-state systems will have 2^n possible states and is usually represented by a vector from \mathbb{C}^{2^n} .

However, when simulating action of local operators on large systems, it is more useful to represent state as a tensor from $(\mathbb{C}^2)^{\otimes n}$. In tensor notation, an operator is represented as a tensor with input and output indices for each qubit it acts upon. The input indices are equated with output indices of previous operator. The resulting state is computed by summation over all joined indices. The comparison between Tensor Network notation and Dirac notation is shown in Table 4.1.

Following tensor notations we drop the summation sign over any repeated indices, that

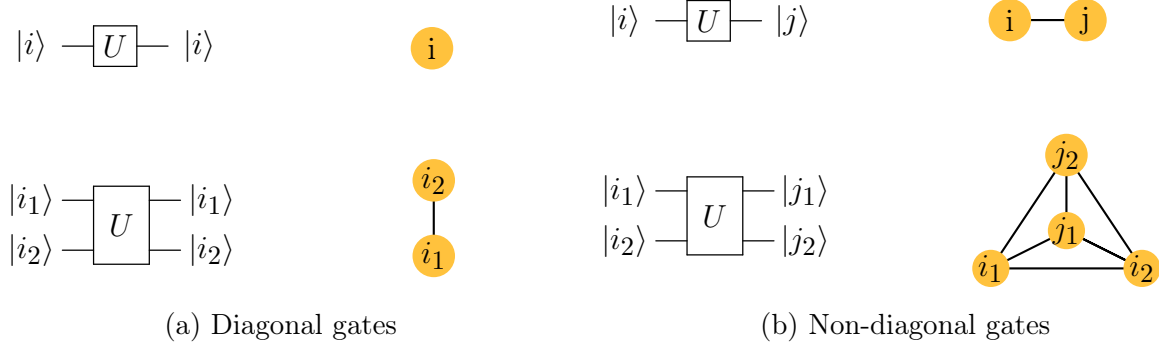


Figure 4.1: Correspondence of quantum gates and graphical representation.

is, $a_i b_{ij} = \sum_i a_i b_{ij}$. For more details on tensor expressions, see [Cichocki et al., 2016].

4.4.2 Graph model of tensor expression

Evaluation of a tensor expression depends heavily on the order in which one picks indices to sum over [Schutski et al., 2020; Markov and Shi, 2008]. The most widely used representation of a tensor expression is a “tensor network,” where vertices stand for tensors and tensor indices stand for edges. For finding the best order of contraction for the expression, we use a line graph representation of a tensor network. In this notation, we use vertices to denote unique indices, and we denote tensors by cliques (fully connected subgraphs). Note that tensors, which are diagonal along some of the axes and hence can be indexed with fewer indices, are depicted by cliques that are smaller than the dimension of the corresponding tensor. For a special case of vectors or diagonal matrices, self-loop edges are used. Figure 4.1 shows the notation for the gates used in this work. For a more detailed description of graph representation, see [Schutski et al., 2020].

	Dirac notation	Tensor notation
general	$ \phi\rangle = \hat{X}_0 \otimes \hat{I}_1 \psi\rangle$	$\phi_{ij} = X_{i'i} \psi_{ij}$
product state	$ \psi\rangle = a\rangle b\rangle$	$\psi_{ij} = a_i a_j$
with Bell state	$ \phi\rangle = \hat{X}_0 \otimes \hat{I}_1 (00\rangle + 11\rangle)$	$\phi_{ij} = X_{i'i} \delta_{ij}$

Table 4.1: Comparison between different notations of quantum circuits

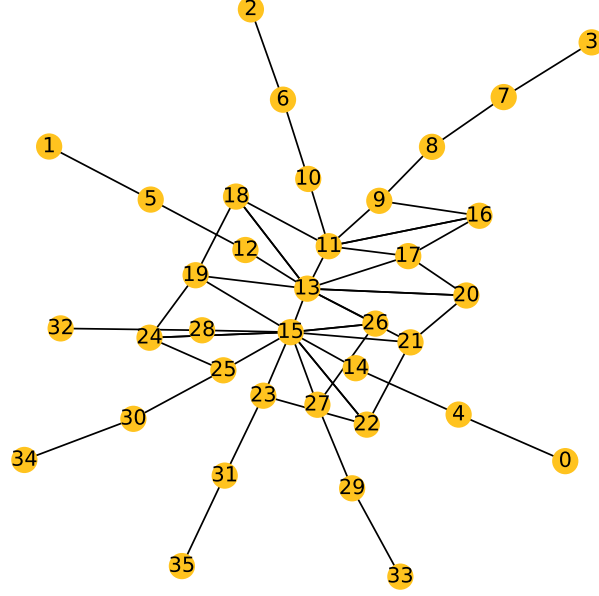


Figure 4.2: Graph representation of tensor expression of the circuit in Fig. 3.1. Every vertex corresponds to a tensor index of a quantum gate. Indices are labeled right to left: 0-3 are indices of the output statevector, and 32-25 are indices of the input statevector. Self-loop edges are not shown (in particular $Z^{2\gamma}$, which is diagonal).

Having built this representation, one has to determine the index elimination order. The tensor network is contracted by sequential elimination of its indices.

The tensor after each index elimination will be indexed by a union of sets of indices of tensors in the contraction operation. In the line graph representation, the index contraction removes the corresponding vertices from the graph. Adding the intermediate tensor afterwards corresponds to adding a clique to all neighbors of index i . We call this step *elimination of vertex (index) i* . An interactive demo of this process can be found at <https://lykov.tech/qg> (works for cZ_v2 circuits from “Files to use”— link).

The memory and time required for the new tensor after elimination of a vertex v from G depends exponentially on the number of its neighbors $N_G(v)$. Figure 4.3 shows the dependence of the elimination cost with respect to the number of vertices (steps) of a typical QAOA quantum circuit. The inset also shows for comparison the number of neighbors for every vertex at the elimination step.

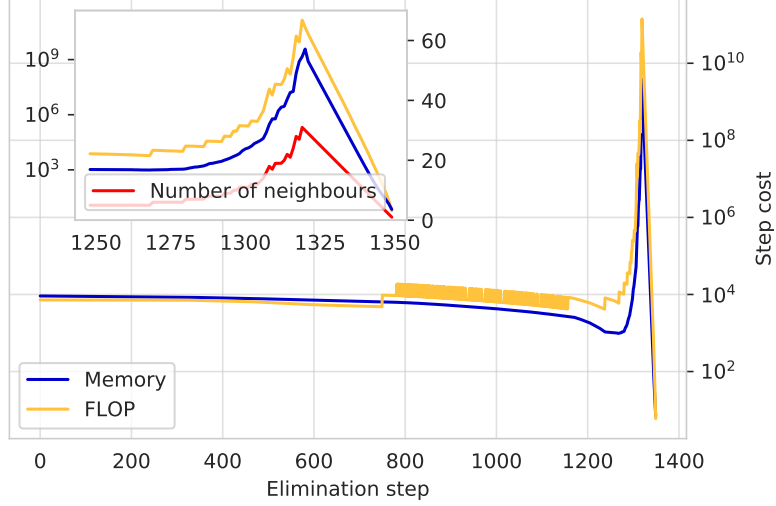


Figure 4.3: Cost of contraction for every vertex for a circuit with 150 qubits. Inset shows the peak magnified and the number of neighbors of the vertex contracted at a given step (right y-axis).

Note that the majority of contraction is very cheap, which corresponds to the low-degree nodes from Figure 4.2. This observation serves as a basis for our *step-dependent slicing* algorithm.

The main factor that determines the computation cost is the maximum $N_G(v)$ throughout the process of sequential elimination of vertices. In other words, for the computation cost C the following is true:

$$C \propto 2^c; c \equiv \max_{i=1 \dots N} N_{G_i}(v_i),$$

where G_i is obtained by contracting $i - 1$ vertices and c is referred to as the *contraction width*. We later use shorter notation for the number of neighbors $N_i(v) \equiv N_{G_i}(v_i)$.

The problem of finding a path of graph vertex elimination that minimizes c is connected to finding the tree decomposition. In fact, the treewidth of the expression graph is equal to $c - 1$. Tree decomposition is NP-hard for general graphs [Bodlaender, 1994], and a similar hardness result is known for the optimal tensor contraction problem [Chi-Chung et al., 1997]. However, several exact and approximate algorithms for tree decomposition were developed

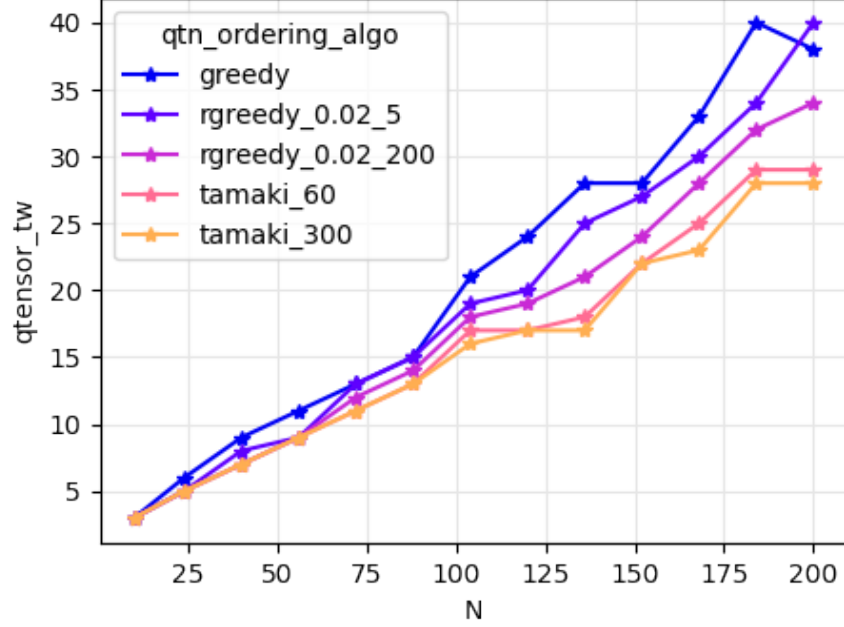


Figure 4.4: Comparison of different ordering algorithms for single amplitude simulation of QAOA ansatz state

in graph theory literature; for references, see [Gogate and Dechter, 2004; Bodlaender et al., 2006; Kloks, 1994; Bodlaender, 1994; Kloks et al., 1993].

4.5 Simulation of a single amplitude

The simulation of a single amplitude is a simple benchmark to use to evaluate the complexity of quantum circuits and simulation performance. We start with N -qubit zero state $|0^{\otimes N}\rangle$ and calculate a probability to measure the same state.

$$\sigma = \langle 0^{\otimes N} | \hat{U} | 0^{\otimes N} \rangle = \langle 0^{\otimes N} | \vec{\gamma}, \vec{\beta} \rangle$$

4.5.1 Ordering algorithm

The ordering algorithm is a dominating part of efficient tensor network contraction. Linear improvement in contraction width results in an exponential speedup of contraction.

There are several ordering algorithms that we use in our simulations. The major criterion to choose one is to maintain a balance between ordering improvement and run time of the algorithm itself.

Greedy algorithm

The greedy algorithm contracts the lowest-degree vertex in the graph. This algorithm is commonly used as a baseline since it provides a reasonable result given a short run-time budget.

Randomized greedy algorithm

The contraction width is very sensitive to small changes in the contraction order. Gray and Kourtis [2020] used this fact in a randomized ordering algorithm, which provided contraction width improvement without prolonging the run time. We use a similar approach in the *rgreedy* algorithm. Instead of choosing the smallest-degree vertex, *rgreedy* assigns probabilities for each vertex using Boltzmann’s distribution:

$$p(v) = \exp(-\frac{1}{\tau}N_G(v))$$

The contraction is then repeated q times, and the best ordering is selected. The τ and q parameters are specified after the name of the *rgreedy* algorithm.

Heuristic solvers

The attempt to use some global information in the ordering problem gives rise to several heuristic algorithms.

QuickBB [Gogate and Dechter, 2004] is a widely-used branch-and-bound algorithm. We found that it does not provide significant improvement in the contraction width in addition

to being much slower than greedy algorithms.

Tamaki’s heuristic solver [Tamaki, 2017] is a dynamic programming approach that provides great results. This is also an “anytime” algorithm, meaning that it provides a solution after it is stopped at any time. The improvements from this algorithm are noticeable when it runs from tens of seconds to minutes. We denote time (in seconds) allocated to this ordering algorithm after its name.

4.6 Parallelization algorithm

We use a two-level parallelization architecture to couple the simulation structure and hardware constraints. Our approach is shown at Fig. 4.5. Multinode-level parallelization uses MPI to share tasks. We slice the partially contracted full expression over n indexes and distribute the slices to 2^n MPI ranks. We use a novel algorithm for determining the slice vertex and step at which to perform slicing, which results in massive expression simplification. This is described in Section 4.6.3. A high-level picture of our algorithm is shown in Fig. 4.6.

Node-level parallelisation over CPU cores uses system threads. For every tensor multiplication and summation we slice the input and output tensors over t indices. Contraction is then performed by 2^t threads writing results to a shared result tensor. This process is described in Section 4.6.2.

To illustrate the two approaches used, we consider a simple expression $C_i = A_{ij}B_j$. There are two obvious ways of parallelization:

1. Parallelization over elements of sum, index j . Every worker computes its version of C_i for some value of j , and the results then are summed.
2. Parallelization over the indices of the result, i . Every worker computes part of the result, C_i , for some value of i .

These two options are intrinsically similar: every worker is assigned a simplified version of

the expression, which is obtained by applying a slicing operation over some indices to every tensor. The difference between the two is that while performing computation using the first option, one must store copies of the result for every worker, which results in higher memory usage that scales linearly with the number of workers. This is not an issue in the second option, where different workers write to different parts of the shared result. The second option is less flexible, however. Usually one has a complex expression on the right-hand side, and the result has a smaller number of dimensions. The crucial part is that one can reduce treewidth of a complex expression using parallelization, which is discussed in Section 4.6.3.

4.6.1 Description of hardware and software

The benchmarks reported in this paper were performed on the Intel Xeon Phi HPC systems in the Joint Laboratory for System Evaluation (JLSE) and the Theta supercomputer at the Argonne Leadership Computing Facility (ALCF) [alc, 2017]. Theta is an 12-petaflop Cray XC40 supercomputer consisting of 4,392 Intel Xeon Phi 7230 processors. Hardware details for the JLSE and Theta HPC systems are shown in Table 4.2.

The Intel Xeon Phi processors used in this work have 64 cores. The cores operate at 1.3 GHz frequency. Besides the L1 and L2 caches, all the cores in the Intel Xeon Phi processors share 16 GBytes of MCDRAM (another name is High Bandwidth Memory) and 192 GBytes of DDR4 memory. The bandwidth of MCDRAM is approximately 400 GBytes/s, while the bandwidth of DDR4 is approximately 100 GBytes/s.

The memory on Xeon Phi processors can be configured in the following modes: flat mode, cache mode, and hybrid mode. In the flat mode, the two levels of memory are treated as separate entities. One can run entirely in MCDRAM or entirely in DDR4 memory. In the cache mode, the MCDRAM is treated as a direct-mapped L3 cache to the DDR4 layer. In the hybrid mode, a part of the MCDRAM is L3 cache and the rest is directly addressable fast MCDRAM, but it does not become part of the (lower bandwidth) DDR4 memory.

Table 4.2: Hardware and software specifications

Intel Xeon Phi node characteristics	
Intel Xeon Phi models	7210 and 7230 (64 cores, 1.3 GHz, 2,622 GFLOPs)
Memory per node	16 GB MCDRAM, 192 GB DDR4 RAM
JLSE Xeon Phi cluster (26.2 TFLOPS peak)	
# of Intel Xeon Phi nodes	10
Interconnect type	Intel Omni-Path TM
Theta supercomputer (11.69 PFLOPS peak)	
# of Intel Xeon Phi nodes	4,392
Interconnect type	Aries interconnect with Dragonfly topology
Cray environment loaded modules	PrgEnv-intel/ 6.0.5, intel/ 19.0.5.281, cray- mpich/ 7.7.10

Besides memory modes, the Intel Xeon Phi processors support five cluster modes: all-to-all, quadrant/hemisphere, and sub-NUMA cluster SNC-4/SNC-2 modes of cache operation. The main idea behind these modes is how to optimally maintain cache coherency depending on data locality.

For the types of problems we are computing here, there is not much difference between various memory configurations [Mironov et al., 2017]. In the calculations presented in this paper, we used the quadrant clustering mode for all quantum circuit simulations on Intel Xeon Phi nodes. We explored the use of different affinity modes and found that there is not much difference in performance between them. For our benchmarks, we used the default affinity, which is set to scatter.

4.6.2 *Single-node parallelization*

Simulation of quantum circuits is an example of a memory-bound task: the main bottleneck of simulation is the storage of intermediate results of a simulation. In a simplistic approach

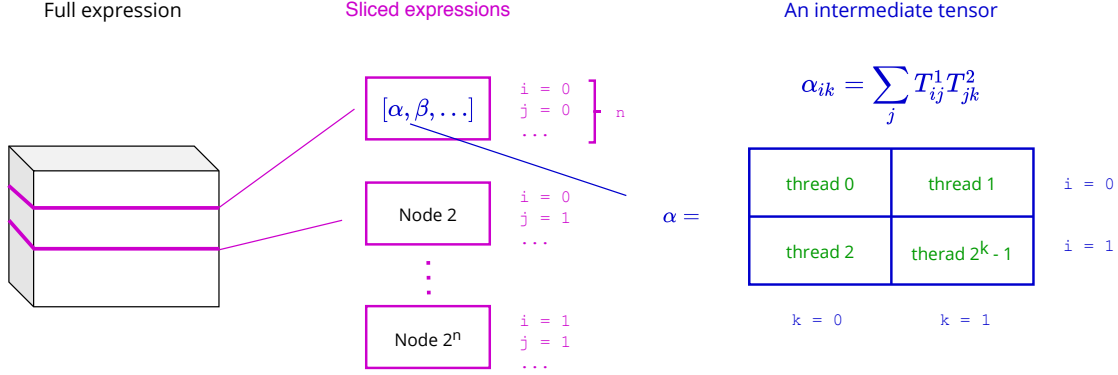


Figure 4.5: Illustration of our two-level tensor parallelization approach. On the multinode level MPI parallelization we use slicing of a partially contracted full expression. On the lower level of a single node, we use thread-based parallelization with a shared resulting tensor.

called the state-vector evolution scheme, the full vector of size 2^n is stored in memory. Thus a circuit containing only 300 qubits will require more memory than there are atoms in the universe. A much more efficient algorithm is the tensor network contraction algorithm described here. But as we show below, it requires use of a complicated parallelization scheme compared with the straightforward linear algebra parallelization scheme used in the state-vector simulators.

Modern high-performance computing (HPC) systems have nodes with a large number of CPU cores. An efficient calculation has to utilize all available CPU cores, using many threads to execute code. The major problem in using MPI-only code is that all of the data structures are replicated across MPI ranks, which results in increased memory usage linearly with respect to the number of MPI ranks. The largest data object in our simulation is the tensor, which is a result of the contraction step. Memory requirements to store such tensor are exponential with respect to its size.

Moreover, every code will inevitably have a part that can be executed only serially. As the number of OpenMP threads or MPI increases, the parallelization becomes less efficient according to Amdahl's law. Thus, following this logic, smaller computations require less time, and the portion of the program that benefits from parallelization will be smaller for

small tensors. As a result, according to Amdahl’s law, this means that for small tensors, we need to use fewer threads.

To address these problems, we share the resulting tensor between 2^t threads. We also use an adaptive thread count determined from task size (Eq. 4.1). A usual approach of splitting matrices in the code is to split into 2^t rows, or columns. This approach is not applicable in our case since tensors have size 2 over each dimension, and it would require reshaping the tensor, so it would be indexed with a multi-index. We choose a similar but more elegant approach. To slice into 2^t parts, we first choose indices that will be our slice dimensions. The slicing operation fixes the value of the index and reduces the number of dimensions by one. We then use a binary form of the thread index (the id of the thread) as a point in space $\{0, 1\}^{\otimes t}$ that defines the slice index values.

Every contraction in the bucket elimination step can be represented by the permutation of indices as

$$C_{ijk} = A_{ij}B_{ik}$$

,

where index i contains indices that A and B have in common and j, k contain indices specific to A, B , respectively. For our simulation, we slice the tensor over the first t indexes of the resulting tensor because this approach results in consistent blocks of resulting tensors assigned to each thread, thereby reducing the memory access time. This part of the algorithm is shown in green in Fig 4.6.

To determine an optimal number of threads to use, we run a series of experiments to estimate the overhead time. We use these experimental results as the basis for an empirical formula for optimal thread count:

$$t = \max(\lfloor \frac{r - 22}{2} \rfloor, 1), \tag{4.1}$$

where r is rank of the resulting tensor.

4.6.3 Multinode parallelization

Every computational node has RAM and a pool of CPU cores. Parallelization over nodes (compared with threads) increases the size of aggregated distributed memory. Thus storing duplicates of tensors is not an issue. For this reason, we use every node to compute a version of a tensor expression evaluated at some values of the tensor indices.

In *graph representation*, the contraction of the full expression is done by consecutive elimination of graph vertices. The elimination of a vertex removes it from the graph and connects all neighbors. An interactive demo of this process can be found at [%link to personal webapp](#), hidden for double-blind review, will be displayed in the final article% (works for cZ_v2 circuits from “Files to use”— link).

The slice of a tensor over an index can be viewed as the function of many variables evaluated at some value of one variable

$$f(x_1, x_2, \dots x_n)|_{x_1=a} = \tilde{f}(x_2, \dots x_n)$$

, where variables can have integer values $v_i \in [0, d-1]$. Slicing reduces the number of indices of the tensor by one, Moreover, in graph representation, this operation results in the removal of the corresponding vertex from the expression graph. Since all sizes of indices we use are equal to 2, removal of n vertices allows us to split the expression into 2^n parts.

This operation is equivalent to decomposition of the full expression into the following form:

$$\sum_{m_1 \dots m_n} (\sum_{V \setminus \{m_i\}} T^1 T^2 \dots T^N), \quad (4.2)$$

where m_i are indices that we slice over and the parts of the expression correspond to the expression in parentheses.

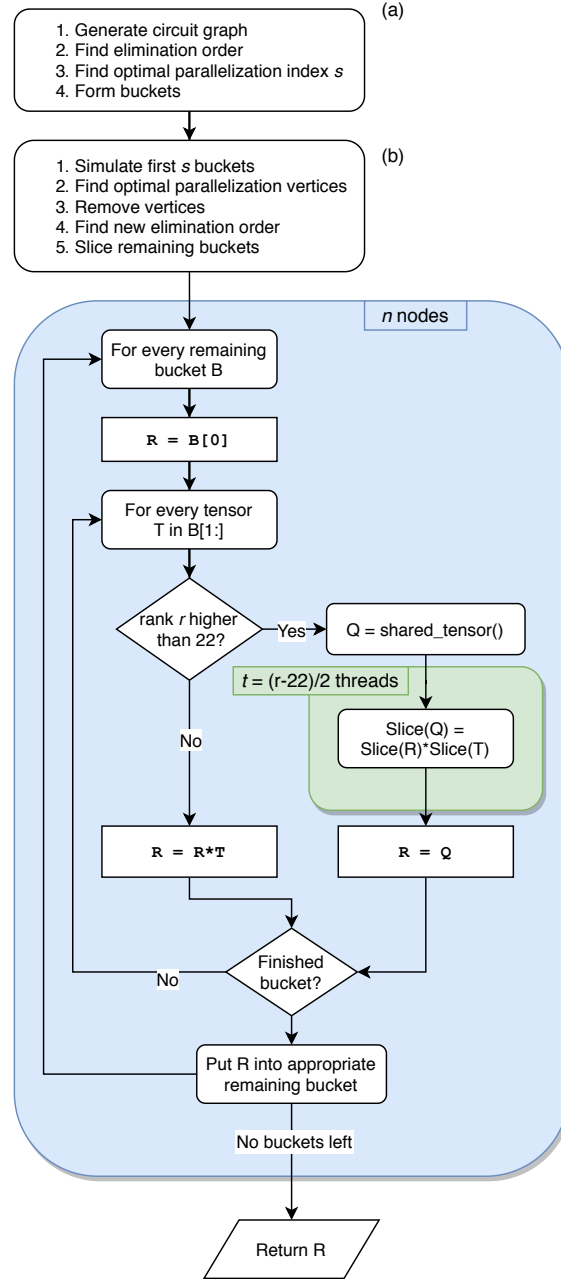


Figure 4.6: Sketch of the parallel bucket elimination algorithm. Part (a) and steps b2–b4 depend only on the structure of a task and can be executed only once for the QAOA algorithm. Steps b1 and b5 are performed serially.

The outer loop of the blue region performs the elimination of the remaining buckets; the inner loop corresponds to processing a single bucket. The summation operation at the end of the bucket processing is omitted for simplicity.

Each part is represented by a graph with lower connectivity than the original one. This dramatically affects optimal elimination path and, respectively, the cost of contraction. Depending on the expression, we observed that using only two computational nodes can allow for speedups of an order of 2^5 .

The QAOA circuit tensor expression results in a graph that has many low-degree vertices, as demonstrated in Fig. 4.2 for a small circuit. As can be seen in Fig. 4.3, most contraction steps are computationally cheap, and connectivity of a graph is low. Vertices can be removed at any step of contraction, giving rise to a completely new problem of finding an optimal step for slicing the expression. We use a simple brute-force algorithm to determine the optimal step at which to perform parallelization. First, we find the ordering for the full graph and analyze the number of neighbors in the contraction path at each step. Any step after the step with the peak number of neighbors is out of consideration since our goal is to lower this peak, and we have to contract the initial expression before parallelizing. For every step of K steps before the peak, we remove from the graph n vertices with the biggest number of neighbors and rerun the ordering algorithm to determine the new contraction width. The vertices could be any vertices in the graph, including “free” (nonrepeated) indices. Since the removed vertices have the biggest number of neighbors, they usually index several tensors, and the expression includes a sum over them. We found that this new width can be lower than the original by more than n , providing freedom for massive reduction in the contraction cost, as discussed in Section 4.8. Step s at which the width is minimal is to be used in the main run of the simulation.

To the best of our knowledge, this approach of *late parallelization* was never described in previous work of this field.

In the first part of the full simulation, labeled (a) in Figure 4.6, we read the circuit, create the expression graph, find the elimination order, and form buckets. We also find the best parallelization step s and the corresponding index used in the parallel bucket elimination.

The simulation starts with contracting the first s buckets, which is computationally cheap. After this we have some other tensor expression network, which also is represented by a partially contracted graph. This expression is conceptually no different from the one we started with; however, its graph representation has much higher connectivity.

The pseudo-code for the next stage, parallel bucket elimination, is listed in Algorithm 3. We first select n vertices with the most number of neighbors and use corresponding indices to slice the remaining expression over. To determine values for slices, we use the binary representation of the MPI rank of the current node. We find a new ordering for the sliced expression to identify a better elimination path with removed vertices taken into account. After reordering the sliced buckets, we run our bucket elimination algorithm with parallel tensor contraction. For every pair of tensors in the bucket, we determine the size of the resulting tensor as a union of the set of indices of both tensors. We then determine whether it is reasonable to use parallel contraction by checking that t calculated by Eq. 4.1 is greater than 0. To run multiplication or summation in parallel, we first allocate a shared tensor, then perform the computation for slices of input and output tensors. The final result is obtained by summing the results from different nodes.

4.6.4 *Step-dependent slicing*

The QAOA circuit tensor expression results in a graph that has many low-degree vertices, as demonstrated in Fig. 4.2 for a small circuit. As can be seen in Fig. 4.3, most contraction steps are computationally cheap, and the connectivity of a graph is low.

Each partially-contracted tensor network is a perfectly valid tensor network and can be sliced as well. From a line graph representation perspective, vertices can be removed at any step of contraction, giving rise to a completely new problem of finding an optimal step for slicing the expression. We propose a *step-dependent slicing* algorithm that uses this fact and determines the best index to perform slicing operation, shown in Fig. 4.7.

Algorithm 3 Parallel bucket elimination

Input: Ordered buckets B_i containing tensors, parallelization step s , number of parallel vertices n vertex ordering $\pi : V \rightarrow \mathcal{N}$, $\pi = \{(v_i, i)\}_{i=1}^{|V|}$

Output:

```
1: contract_first( $s, B_i$ )                                ▷ Serial part: contract first  $s$  buckets
2: for  $i = 0, n$  do                                       ▷ Find best index to slice along
3:    $p_i = \text{max\_degree\_vertex}(G)$ 
4:    $\text{remove\_vertex}(G, p_i)$ 
5: end for
6:  $\vec{v} \leftarrow \text{binary\_repr}(\text{mpi\_get\_rank}())$ 
7: for  $j = 0, n$  do                                       ▷ Slice the expression
8:    $B_i \leftarrow B_i|_{p_j=v_j}$ 
9: end for
10: for  $i = s, |V|$  do
11:    $v \leftarrow \pi^{-1}(i)$ 
12:    $R \leftarrow B_i[0]$ 
13:   for  $T \in B_i[1 :]$  do                                   ▷ Process next bucket
14:      $r \leftarrow |T.\text{indexes} \cup R.\text{indexes}|$              ▷ Determine resulting size
15:      $t \leftarrow \text{floor}(\frac{r-22}{2})$ 
16:     if  $t > 0$  then                                       ▷ Contract in thread pool
17:        $Q \leftarrow \text{shared\_tensor}(r)$ 
18:        $\vec{w} \leftarrow \text{binary\_repr}(\text{get\_thread\_num}())$ 
19:        $\vec{k} \leftarrow \text{indices\_of}(Q)[t]$ 
20:        $Q_{v...|_{k_j=w_j}} \leftarrow (Q_{v...}T_{v...})|_{k_j=w_j}$ 
21:        $R \leftarrow Q$                                        ▷  $R$  now points to shared memory tensor
22:     else
23:        $R \leftarrow RT$ 
24:     end if
25:   end for
26:    $R \leftarrow \sum_v R$  ▷ Parallel sum can be implemented in same fashion as contraction above
27:   if  $R$  is scalar then
28:      $\text{result} \leftarrow \text{result} \cdot R$ 
29:   else
30:      $k = \pi(w)$ ,  $w$  is the earliest index of  $R$  w.r.t  $\pi$ 
31:      $B_k \leftarrow B_k \cup R$ 
32:   end if
33: end for
34:  $\text{result} \leftarrow \text{mpi\_reduce\_sum}(\text{result})$              ▷ Gather the results
35: return result
```

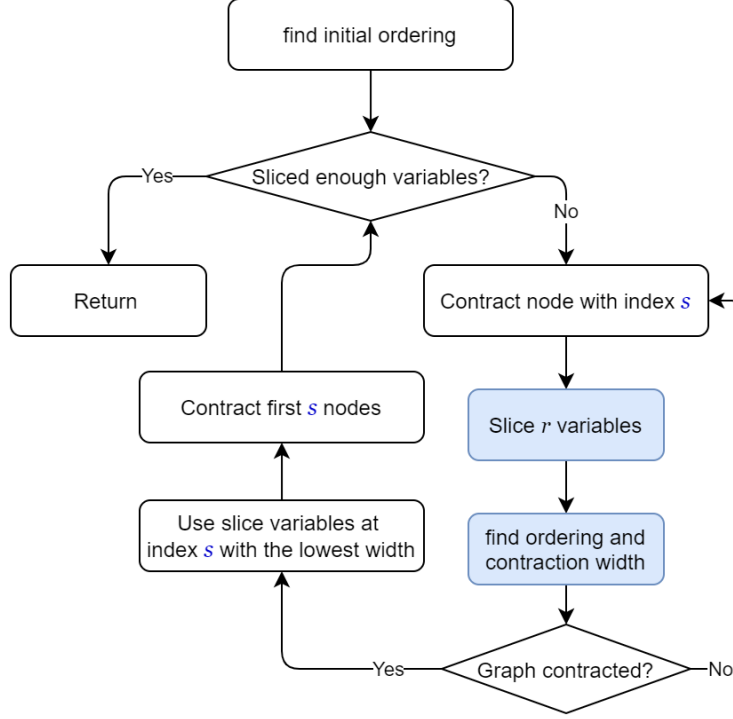


Figure 4.7: Step-based slicing algorithm. The blue boxes are evaluated for each graph node and are the main contributions to time.

We start with finding the ordering for the full graph. Our algorithm then selects consideration only those steps that come before the peak. For every such contraction step s , we remove r vertices with the biggest number of neighbors from the graph and re-run the ordering algorithm to determine the contraction after slicing. The distribution of contraction width is shown on Fig. 4.10.

The step s at which slicing produces best contraction width and contraction order before that is then added to a contraction schedule. This process can be repeated several times until n indices in total are selected - each r of them having their optimal step s . This algorithm requires $\frac{n}{2r}\mathcal{N}$ runs of an ordering algorithm, where \mathcal{N} is the number of nodes in the graph, which is usually of the order of 1000. Only greedy algorithms are used in this procedure due to its short run time.

The value of r can be used to slightly tweak the quality of the results. If $r = n$, all the n variables are sliced at a single step. If $r = 1$, each slice variable can have its own slice

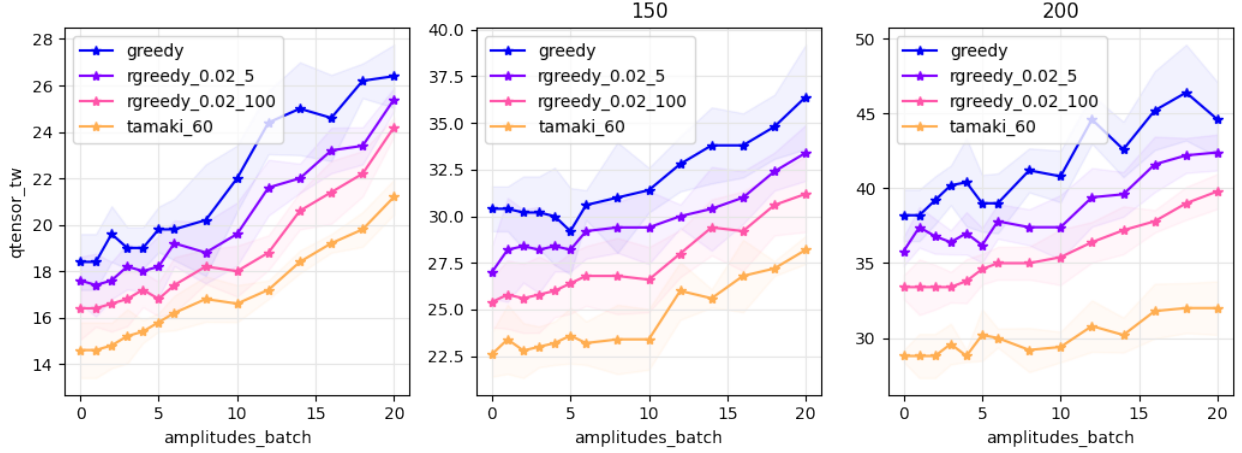


Figure 4.8: Simulation cost for a batch of amplitudes. The calculations are done for 5 random instances of degree-3 random regular graphs and the mean value is plotted. The three plots are calculated for different number of qubits: 100, 150 and 200.

step s , which gives better results for larger n .

We observed that using $n = 1$ already provides contraction width reduction by 3, which converts to 8x speedup in simulation.

To the best of our knowledge, this approach of *step-dependent parallelization* was never described in previous work in this field.

4.7 Simulation of several amplitudes

The QAOA algorithm in its quantum part requires sampling of bit-strings that are potential solutions to a Max-Cut problem. It is possible to emulate sampling on a classical computer without calculating all the probability amplitudes. To obtain such samples, one can use *frugal rejection sampling* [Villalonga et al., 2019] which requires calculating several amplitudes.

Our tensor network approach can be extended to simulate a batch of variables. If we contract all indices of a tensor network, the result will be scalar - a probability amplitude. If we decide to leave out some indices, the result will be a tensor indexed by those indices.

This tensor corresponds to a clique on left-out indices. If a graph contains a clique of size

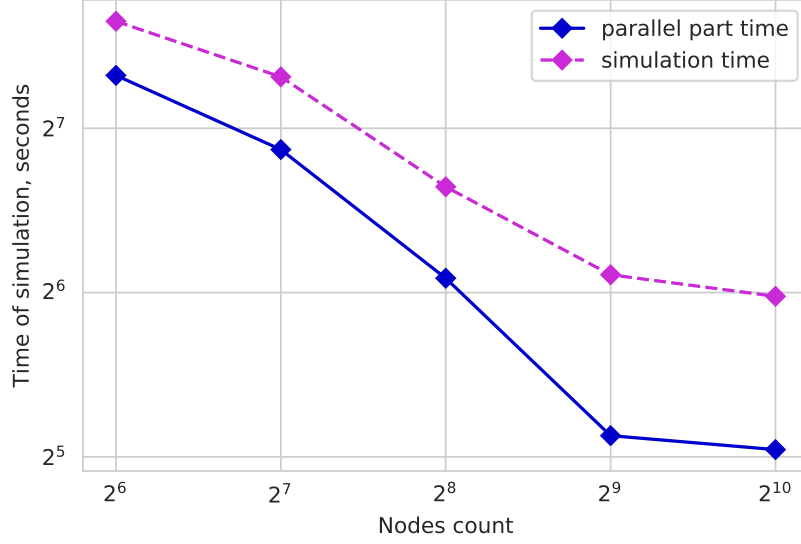


Figure 4.9: Experimental data of simulation time with respect to the number of Theta nodes. The circuit is for 210 qubits and 1,785 gates.

a , its treewidth is not smaller than a . And if we found a contraction order with contraction width c , during the contraction procedure we will have a clique of size c . If $a < c$ then adding a clique to the original graph does not increase contraction width. This opens a possibility to simulate a batch of 2^a amplitudes for the same cost as a single amplitude. This is discussed in great detail in [Schutski et al., 2020].

Figure 4.8 shows contraction width for simulation of batch of amplitudes for different values of a , ordering algorithms and graph sizes.

4.8 Results

We used the Argonne’s Cray XC40 supercomputer called Theta that consists of 4,392 computational nodes. Each node has 64 Intel Xeon Phi cores and 208 GB of RAM. The combined computational power of this supercomputer is about 12 PFLOP/sec. The aggregated amount of RAM across nodes is approximately 900,000 GB.

For our main test case, a circuit with 210 qubits, the initial contraction was calculated

using a greedy algorithm and resulted in contraction width 44. This means that the cost of simulation would be ≥ 70 TFLOPS and 281 TB, respectively. Using our *step-dependent slicing* algorithm with $r = n$ on 64 computational nodes allows us to remove 6 vertices and split the expression into smaller parts that have a contraction width of 32, which easily fits into RAM of one node. The whole simulation, in this case, uses 60% of 13 TB cumulative memory of 64 nodes, more than 35x less than a serial approach uses.

Figure 4.10 shows how the contraction width c of the sliced tensor expression depends on step s for several values of numbers of sliced indices n . The notable feature is the high variance of c with respect to s —the difference between the smallest and the largest values goes up to 9, which translates to a 512x cost difference. However, the general pattern for different QAOA circuits remains similar: increasing n by one reduces $\min_s(c(s))$ by one.

Computational speedup provided by 64 nodes is on the order of $4096 = 2^{44-32}$ which is more than the theoretical limit of 64x for any kind of straightforward parallelization. Using 512 nodes drops the contraction width to 29 and reduces the simulation time 3x compared with that when using 64 nodes.

The experimental results for 64–1,024 nodes are shown in Fig. 4.9. Simulation time includes serial simulation of the first small steps before step s , which takes 40 s for a 210-qubit circuit, or 25–50% of total simulation time, depending on the number of nodes.

4.9 Conclusions

We have presented a novel approach for simulating large-scale quantum circuits represented by tensor network expressions. It allowed us to simulate large QAOA quantum circuits up to 210 qubit circuits with a depth of 1,785 gates on 1,024 nodes and 213 TB of memory on the Theta supercomputer.

As a demonstration, we applied our algorithm to simulate quantum circuits for QAOA ansatz state with $p = 1$, but our algorithm also works for higher p also. To reduce memory

footprint, we developed a *step-dependent slicing* algorithm that contracts part of an expression in advance and reduces the expensive task of finding an elimination order. Using this approach, we found an ordering that produces speedups up to 512x, when compared with other parallelization steps s for the same expression.

The unmodified tensor network contraction algorithm is able to simulate 120-140 qubit circuits, depending on the problem graph. By using a randomized greedy ordering algorithm, we were able to raise this number to 175 qubits. Furthermore, using a parallelization based on *step-dependent slicing* allows us to simulate 210 qubits on the supercomputer Theta. Another way to obtain samples from the QAOA ansatz state is to use density matrix simulation, but it is prohibitively computationally expensive and memory demanding. The largest density matrix simulators known to us can compute 100 qubit problems [Fried et al., 2018] and 120 qubit problems [Zhao et al., 2020] using high-performance computing.

The important feature of our algorithm is applicability to the QAOA algorithm: the contraction order has to be generated only once and then can be reused for additional simulations with different circuit parameters. As a result, it can be used to simulate a large variety of QAOA circuits.

We conclude that this work presents a significant development in the field of quantum simulators. To the best of our knowledge, the presented results are the largest QAOA quantum circuit simulations reported to date.

4.10 Acknowledgements

This research used the resources of the Argonne Leadership Computing Facility, which is a U.S. Department of Energy (DOE) Office of Science User Facility supported under Contract DE-AC02-06CH11357. We gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory. This research was also supported by the U.S. Department of Energy, Office of

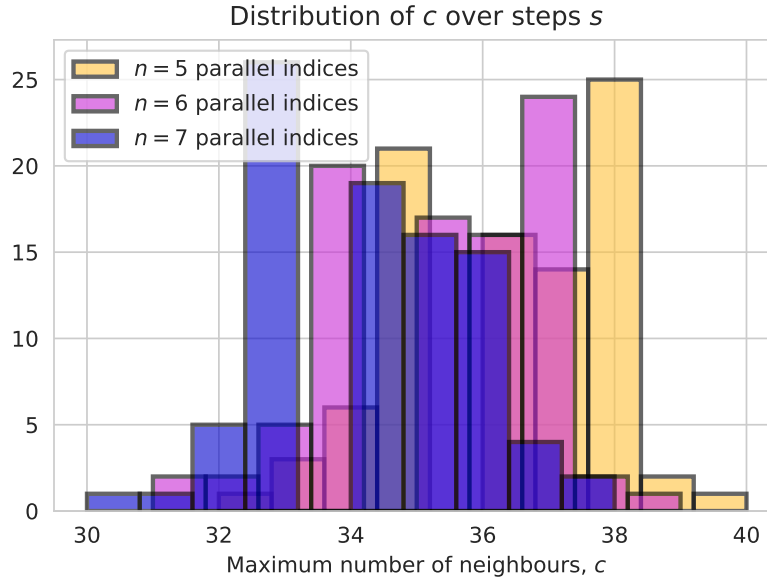


Figure 4.10: Distribution of the contraction width (maximum number of neighbors) c for different numbers of parallel indices n . While variance of c is present, showing that it is sensible to the parallelization index s , we are interested in the minimal value of s , which, in turn, generally gets smaller for bigger n .

Science, Basic Energy Sciences, Materials Sciences and Engineering Division, and by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation's exascale computing imperative.

CHAPTER 5

GPU ACCELERATION OF TENSOR NETWORK CONTRACTION

This chapter is adapted from Lykov, Chen, Chen, Keipert, Zhang, Gibbs, and Alexeev [2021].

5.1 Introduction

Quantum information science (QIS) has a great potential to speed up certain computing problems such as combinatorial optimization and quantum simulations [Alexeev et al., 2021]. The development of fast and resource-efficient quantum simulators to classically simulate quantum circuits is the key to the advancement of the QIS field. For example, simulators allow researchers to evaluate the complexity of new quantum algorithms and to develop and validate the design of new quantum circuits. Another important application is to validate quantum supremacy and advantage claims.

One can simulate quantum circuits on classical computers in many ways. The major types of simulation approaches are full amplitude-vector evolution [De Raedt et al., 2007; Smelyanskiy et al., 2016; Häner and Steiger, 2017; Wu et al., 2019], the Feynman paths approach [Bernstein and Vazirani, 1997], linear algebra open system simulation [Otten, 2020], and tensor network contractions [Markov and Shi, 2008; Pednault et al., 2017; Boixo et al., 2017]. These techniques have advantages and disadvantages. Some are better suited for small numbers of qubits and high-depth quantum circuits, while others are better for circuits with a large number of qubits but small depth. Some are also tailored toward the accuracy of simulation of noise in quantum computers.

For shallow quantum circuits the state-of-the-art technique to simulate quantum circuits is currently arguably the tensor network contraction method because of the memory efficiency for the method relative to state vector methods that scale by 2^N , where N is the

number of qubits. This effectively limits the state vector methods to quantum circuits with less than 50 qubits. The challenge with the tensor network methods is determining the optimal contraction order, which is known to be an NP-complete problem [Markov and Shi, 2008]. We choose to focus on the simulation of the Quantum Approximate Optimization Algorithm (QAOA) [Farhi and Harrow, 2016] given its importance to machine learning and its suitability for the current state of the art with noisy intermediate state quantum computers that generally work with circuits of short depth.

In this work we ported and optimized the tensor network quantum simulator QTensor to run efficiently on GPUs, with the eventual goal to simulate large quantum circuits on the modern and upcoming supercomputers. In particular, we benchmarked QTensor on a NVIDIA DGX-2 server with a V100 accelerator using the CUDA version 11.0. The performance is shown for the full expectation value simulation of the QAOA MaxCut problem on a 3-regular graph of size 30 with depth $p = 4$.

5.2 Methodology

5.2.1 QAOA Overview

The Quantum Approximate Optimization Algorithm is a variational quantum algorithm that combines a parameterized ansatz state preparation with a classical outer-loop algorithm that optimizes the ansatz parameters. QAOA is used for approximate solution of binary optimization problems [Farhi et al., 2014]. A solution to the optimization problem is obtained by measuring the ansatz state on a quantum device. The quality of the QAOA solution depends on the depth of the quantum circuit that generated the ansatz and the quality of parameters for the ansatz state.

A binary combinatorial optimization problem is defined on a space of binary strings of length N and has m clauses. Each clause is a constraint satisfied by some assignment of

the bit string. QAOA maps the combinatorial optimization problem onto a 2^N -dimensional Hilbert space with computational basis vectors $|z\rangle$ and encodes $C(z)$ as a quantum operator \hat{C} diagonal in the computational basis. One of the most widely used benchmark combinatorial optimization problems is MaxCut, which is defined on an undirected unweighted graph. The goal of the MaxCut problem is to find a partition of the graph's vertices into two complementary sets such that the number of edges between the sets is maximized. It has been shown in [Farhi et al., 2014] that on a 3-regular graph, QAOA with $p = 1$ produces a solution with an approximation ratio of at least 0.6924.

A graph $G = (V, E)$ of $N = |V|$ vertices and $m = |E|$ edges can be encoded into a MaxCut cost operator over N qubits by using m two-qubit gates.

$$\hat{C} = \frac{1}{2} \sum_{(ij) \in E} 1 - \hat{\sigma}_i^z \hat{\sigma}_j^z \quad (5.1)$$

The QAOA ansatz state $|\vec{\gamma}, \vec{\beta}\rangle$ is prepared by applying p layers of evolution unitaries that correspond to the cost operator \hat{C} and a mixing operator $\hat{B} = \sum_{i \in V} \hat{\sigma}_i^x$. The initial state is the equally weighted superposition state and maximal eigenstate of \hat{B} .

$$|\vec{\gamma}, \vec{\beta}\rangle_p = \prod_{k=1}^p e^{-i\beta_k \hat{B}} e^{-i\gamma_k \hat{C}} |+\rangle \quad (5.2)$$

The parameterized quantum circuit (5.2) is called the *QAOA ansatz*. We refer to the number of alternating operator pairs p as the *QAOA depth*.

The solution to the combinatorial optimization problem is obtained by measuring the QAOA ansatz. The expected quality of this solution is an expectation value of the cost operator in this state.

$$\langle C \rangle_p = \langle \vec{\gamma}, \vec{\beta} |_p C | \vec{\gamma}, \vec{\beta} \rangle_p$$

The expectation value can be minimized with respect to parameters $\vec{\gamma}, \vec{\beta}$. The optimization of $\vec{\gamma}, \vec{\beta}$ can be performed by using classical computation or by varying the parameters and sampling many bitstrings from a quantum computer to estimate the expectation value. Acceleration of the optimal parameters search for a given QAOA depth p is the focus of many approaches aimed at demonstrating the quantum advantage. Examples include such methods as warm- and multistart optimization [Egger et al., 2021; Shaydulin et al., 2019a], problem decomposition [Shaydulin et al., 2019b], instance structure analysis [Shaydulin et al., 2020], and parameter learning [Khairy et al., 2020].

In this paper we focus on application of a classical quantum circuit simulator QTensor to the problem of finding the expectation value $\langle C \rangle_p$.

5.2.2 Tensor Network Contractions

Calculation of an expectation value of some observable in a given state generated by some quantum circuit can be done efficiently by using a tensor network approach. In contrast to state vector simulators, which store the full state vector of size 2^N , QTensor maps a quantum circuit to a tensor network. Each quantum gate of the circuit is converted to a tensor. An expectation value $\langle \phi | \hat{C} | \phi \rangle = \langle \psi | \hat{U}^\dagger \hat{C} \hat{U} | \psi \rangle$ is then simulated by contracting the corresponding tensor network. For more details on how a quantum circuit is converted to a tensor network, see [Schutski et al., 2020; Lykov et al., 2020a].

A tensor network is a collection of tensors, which in turn have a collection of indices, where tensors share some indices with each other. To contract a tensor network, we create an ordered list of tensor buckets. Each bucket (a collection of tensors) corresponds to a tensor index, which is called *bucket index*. Buckets are then contracted one by one. The contraction of a bucket is performed by summing over the bucket index, and the resulting tensor is then appended to the appropriate bucket. The number of unique indices in aggregate indices of all bucket tensors is called a *bucket width*. The memory and computational resources of a bucket

contraction scale exponentially with the associated bucket width. For more information on tensor network contraction, see [Lykov and Alexeev, 2021; Lykov et al., 2020b; Schutski et al., 2020]. If some observable $\hat{\Sigma}$ acts on a small subset of qubits, most of the gates in the quantum circuit \hat{U} cancel out when evaluating the expectation value. The cost QAOA operator \hat{C} is a sum of m such terms, each of which could be viewed as a separate observable. Each term generates a *lightcone*—a subset of the problem that generates a tensor network representing the contribution to the cost expectation value.

The expectation value of the cost for the graph G and MaxCut QAOA depth p is then

$$\begin{aligned}
\langle C \rangle_p(\vec{\gamma}, \vec{\beta}) &= \langle \vec{\gamma}, \vec{\beta} | \hat{C} | \vec{\gamma}, \vec{\beta} \rangle \\
&= \langle \vec{\gamma}, \vec{\beta} | \sum_{jk \in E} \frac{1}{2} (1 - \hat{\sigma}_j^z \hat{\sigma}_k^z) | \vec{\gamma}, \vec{\beta} \rangle \\
&= \frac{|E|}{2} - \frac{1}{2} \sum_{jk \in E} \langle \vec{\gamma}, \vec{\beta} | \hat{\sigma}_j^z \hat{\sigma}_k^z | \vec{\gamma}, \vec{\beta} \rangle \\
&\equiv \frac{|E|}{2} - \frac{1}{2} \sum_{jk \in E} e_{jk}(\vec{\gamma}, \vec{\beta}),
\end{aligned}$$

where e_{jk} is an individual edge contribution to the total cost function. Note that the observable in the definition of e_{jk} is local to only two qubits; therefore most of the gates in the circuit that generates the state $|\vec{\gamma}, \vec{\beta}\rangle$ cancel out. The circuit after the cancellation is equivalent to calculating $\hat{\sigma}_j^z \hat{\sigma}_k^z$ on a subgraph S of the original graph G . These subgraphs can be obtained by taking only the edges that are incident from vertices at a distance $p - 1$ from the vertices j and k . The full calculation of $E_G(\vec{\gamma}, \vec{\beta})$ requires evaluation of $|E|$ tensor networks, each representing the value $e_{jk}(\vec{\gamma}, \vec{\beta})$ for $jk \in E$.

5.2.3 Merged Indices Contraction

Since the contraction in the bucket elimination algorithm is executed one index at a time, the ratio of computational operations to memory read/write operations is small. This ratio is

also called the operational intensity or arithmetic intensity. Having small arithmetic intensity hurts the performance in terms of FLOPs: for each floating-point operation calculated there are relatively many I/O operations, which are usually slower. For example, to calculate one element of the resulting matrix in a matrix multiplication problem, one needs to read $2N$ elements and perform $4N$ operations. The size of the resulting matrix is similar to the input matrices. In contrast, when calculating an outer product of two vectors, the size of the resulting matrix is much larger than the combined size of the input vectors; each element requires two reads and only one floating-point operation.

To mitigate this limitation, we develop an approach for increasing the arithmetic intensity, which we call merged indices. The essence of the approach is to combine several buckets and contract their corresponding indices at once, thus having smaller output size and larger arithmetic intensity. We have a group of circuit contraction backends that all use this approach.

For the merged backend group, we order the buckets first and then find the mergeable indices before performing the contraction. We list the set of indices of tensors in each bucket and then merge the buckets if the set of indices of one bucket is a subset of the other. We benchmark the sum of the total time needed for the merged indices contraction and compare it with the unmerged baseline results. We call this group the “merged” group and the baseline the “unmerged” group.

5.2.4 *CPU-GPU Hybrid Backend*

The initial tensor network contains only very small tensors of at most 16 elements (4 dimensions of size 2). We observe that the contraction sequence obtained by our ordering algorithm results in buckets of small width for first 80% of contraction steps. Only after all small buckets are contracted, sequence we start to contract large buckets. The GPUs usually perform much better when processing large amount of data. We observe this behaviour in

our benchmarks on Figure 5.1. We therefore implement a mix backend which uses both CPU and GPU. It combines a CPU backend and a GPU backend by dispatching the contraction procedure to appropriate backend.

The mix or the hybrid backend uses the bucket width, which is determined by the number of unique indices in a bucket, to allocate the correct device for such a bucket to be computed. The threshold between the CPU backend and the GPU backend is determined by a trial program. This program runs a small circuit, which is used for all backends for testing, separately on a GPU backend and a CPU backend. After the testing is complete, it iterates through all bucket widths and checks whether at this bucket width the GPU takes less time or not. If it finds the bucket width at which the GPU is faster, it will output that bucket width, and the user can use this width when creating the hybrid backend in the actual simulation. In the actual simulation, if the bucket width is smaller than the threshold, the hybrid backend will allocate this bucket to the CPU and will allocate it to the GPU if the width is greater.

Since we don't contract buckets of large width on CPU, the resulting tensors are rather small, on the order of 1,000s of bytes. The time for data transfer in this case is considered negligible and is not measured in our code. The large tensors start to appear from contractions that combine these small tensors after all the data is moved to GPU.

5.2.5 *Datasets for Synthetic Benchmarks*

Tensor network contraction is a complex procedure that involves many inhomogeneous operations. Since we are interested in achieving the maximum performance of the simulations, it is beneficial to compare the FLOPs performance to several more relevant benchmarking problems. We select several problems for this task:

1. Square matrix multiplication, the simplest benchmark problem which serves as an upper bound for our FLOP performance;

2. Pairwise tensor contractions with a small number of large dimensions and fixed contraction structure;
3. Pairwise tensor contractions with a large number of dimensions of size 2 and permuted indices;
4. Bucket contraction of buckets that are produced by actual expectation value calculation;
5. Full circuit contraction which takes into account buckets of large and small width.

By gradually adding complexity levels to the benchmark problems and evaluating the performance on each level, we look for the largest reduction in FLOPs. The corresponding level of complexity will be at the focus of our future efforts for optimisation of performance. The results for these benchmarks are shown in Section 5.3.5 and Figures 5.6 and 5.7.

Matrix Multiplication

We perform the matrix multiplications for the square matrices of the same size and record the time for the operation for the CPU backend Numpy and the GPU backends PyTorch and CuPy. We use the built-in `random()` function of each backend to randomly generate two square matrices of equal size as our input, and we use the built-in `matmul()` function to produce the output matrix. The size of the input matrices ranges from 10×10 to 8192×8192 , and the test is done repeatedly on four different data types: `float`, `double`, `complex64`, and `complex128`. For the multiplication of two $n \times n$ matrices, we define the number of complex operations to be n^3 , and we calculate the number of FLOPs for complex numbers as $8 \times \frac{\text{number_of_operations}}{\text{operation_time}}$.

Tensor Network Contraction

We have two experiment groups in benchmarking the tensor contraction performance: tensor contractions with a fixed contraction expression and tensor contractions with many indices where each index has a small size. We call the former group “tncontract fixed” because we fix the contraction expression as “abcd,bcdf→acf,” and we call the latter one “tncontract random” because we randomly generate the contraction expression. In a general contraction expression, we sum over the indices not contained in the result indices. In this fixed contraction expression, we sum over the common index “b” and “d” and keep the rest in our result indices. We generate two square input tensors of shape $n \times n \times n \times n$ and output a tensor of shape $n \times n \times n$, where n is a size ranging from 10 to 100. For the “tncontract random” group, we randomly generate the number of contracted indices and the number of indices in the results first and then fill in the shape array with size 2. For example, a contraction formula “dacb,ad→bcd” (index “a” is contracted) needs two input tensors: the first one with shape $2 \times 2 \times 2 \times 2$ and the second one with shape 2×2 . We use the formula $2^{\text{number_of_different_indices}}$ to calculate the number of operations, and we record the contraction time and compute the FLOPs value based on the formula used in matrix multiplication. Following the same procedure in matrix multiplication, we use the backends’ built-in functions to randomly generate the input tensors based on the required size and the four data types.

Circuit Simulation

For numerical evaluations, we benchmark the full expectation value simulation of the QAOA MaxCut problem for a 3-regular graph of size 30 and QAOA depth $p = 4$. We have two properties for evaluating the circuit simulation performance: unmerged vs. merged backend and single vs. mixed backend.

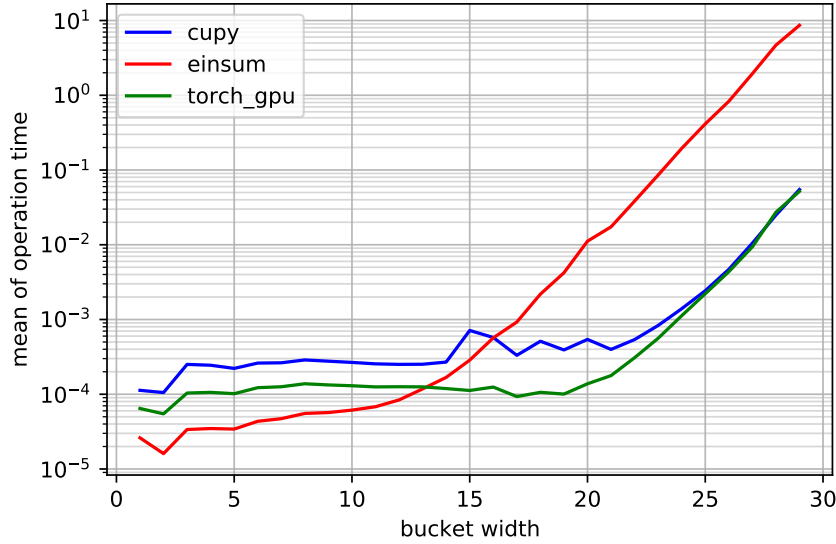


Figure 5.1: Breakdown of mean time to contract a single bucket by bucket width. The test is performed for expectation value as described in 5.3.1. CPU backends are faster for buckets of width $\leq 13 - 16$, and GPU faster are better for larger buckets. This picture also demonstrates that every contraction operation spends some time on overhead which doesn’t depend on bucket width, and actual calculation that scales exponentially with bucket width.

5.3 Results

The experiment is performed on an NVIDIA DGX-2 server (provided by NVIDIA corporation) with a V100 accelerator using the CUDA version 11.0. The baseline NumPy backend is executed only on a CPU and labeled “einsum” in our experiment since we use `numpy.einsum()` for the tensor computation. We also benchmark the GPU library CuPy (on the GPU only) and PyTorch (on both the CPU and GPU).

5.3.1 Single CPU-GPU Backends

We benchmark the performance of the full expectation value simulation of the QAOA Max-Cut problem on a 3-regular graph of size 30 with depth $p = 4$, as shown in in Figures 5.1, 5.2, and 5.3. This corresponds to contraction of 20 tensor networks, one network per each lightcone. Our GPU implementation of the simulator using PyTorch (labeled “torch_gpu”)

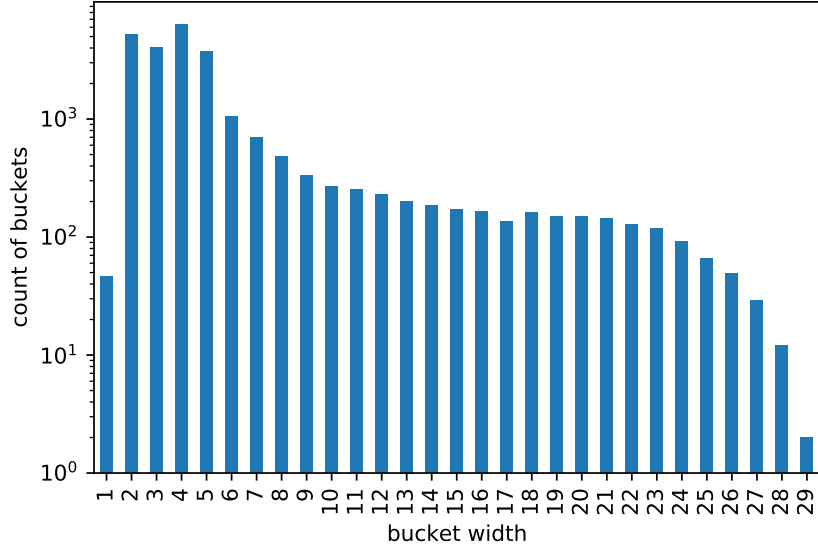


Figure 5.2: Distribution of bucket width in the contraction of QAOA full circuit simulation. The y-axis is log scale; 82% of buckets have width ≤ 6 , which have relatively large overhead time.

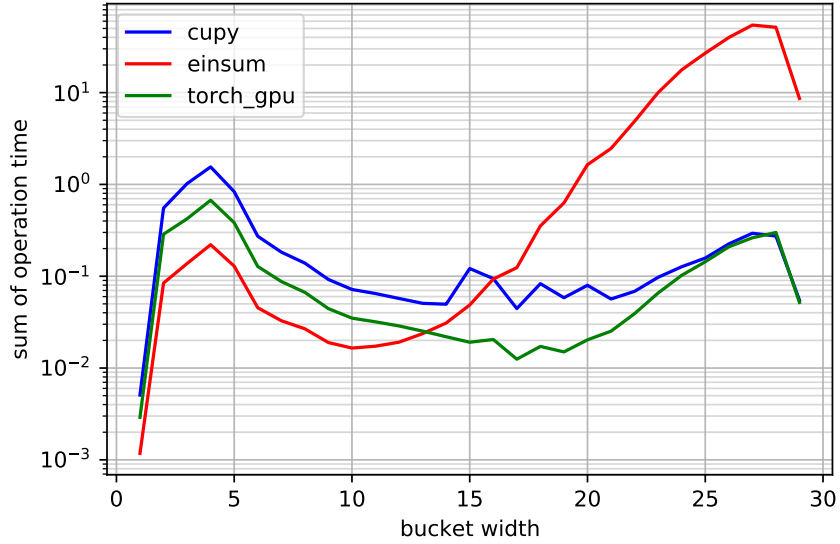


Figure 5.3: Breakdown of total time spent on bucket of each size in full QAOA expectation value simulation. The y-value on this plot is effectively one in Figure 5.1 multiplied by one in Figure 5.2. This figure is very useful for analyzing the bottlenecks of the simulation. It shows that most of the time for CPU backend is spent on large buckets, but for GPU backends the large number of small buckets results in a slowdown.

achieves $70.3\times$ speedup over the CPU baseline and $1.92\times$ speedup over CuPy.

Figure 5.1 shows the mean contraction time of various bucket widths in different backends. In comparison with "cupy" backend, the "einsum" backend spends less total time for bucket width less than 16, and the threshold value changes to around 13 when being compared to "torch_gpu" backend. Both GPU backends have similar and better performance for larger bucket widths. However, this threshold value can fluctuate when comparing the same pair of CPU and GPU backends. This is likely due to the fact that the benchmarking platform are under different usage loads.

Figure 5.3 provides a breakdown of the contraction times of buckets by bucket width. This distribution is multimodal: A large portion of time is spent on buckets of width 4. For CPU backends the bulk of the simulation time is spent on contracting large buckets. Figure 5.2 shows the distribution of bucket widths, where 82% of buckets have width less than 7. This signifies that simulation has an overhead from contracting a large number of small buckets.

This situation is particularly noticeable when looking at the total contraction time of different bucket widths. Figure 5.3 shows that the distribution of time vs bucket width has two modes: for large buckets that dominate the contraction time for CPU backends and for small buckets where most of the time is spent on I/O and other code overhead.

5.3.2 Merged Backend Results

The “merged” groups merge the indices before performing contractions. In Fig. 5.4, the three unmerged (baseline) backends are denoted by dashed lines, while the merged backends are shown by solid lines. For the GPU backends CuPy and PyTorch, the merged group performs significantly better for buckets of width ≥ 20 . The CuPy merged backend always has a similar or better performance compared with the CuPy unmerged group and has much better performance for buckets of larger width. For buckets of width 28, the total operation

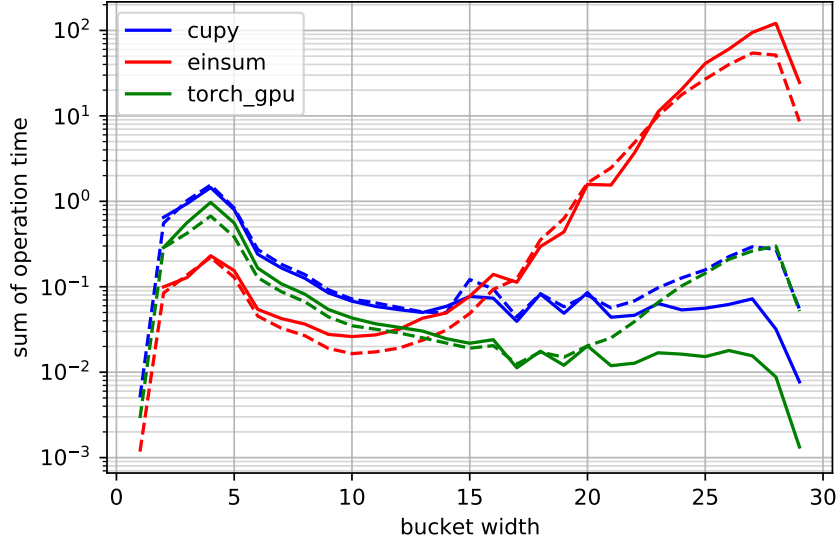


Figure 5.4: Breakdown of total contraction time by bucket width in full expectation value simulation of problem size 30. Lines with the same color use the same type of backends. The solid lines represent the merged version of backends, and the dashed lines denote the baseline backends. The merged GPU backends are better for buckets of width ≥ 20 .

time of the unmerged GPU backends is about 0.28 seconds, compared with 32 ms ($8.75\times$ speedup) for the CuPy merged group and 8.8 ms ($31.82\times$ speedup) for the PyTorch backend. But we do not observe much improvement for the merged CPU backend.

5.3.3 Mix CPU-GPU Backend Results

From Figure 5.1 one can see that GPU backends perform much better for buckets of large width, while CPU backends are better for smaller buckets. We thus implemented a mixed backend approach, which dynamically selects a device (CPU or GPU) on which the bucket should be contracted. We select a threshold value of 15 for the bucket width; any bucket that has a width larger than 15 will be contracted on the GPU. Figure 5.3 shows that for GPU backends small buckets occupy approximately 90% of the total simulation time. The results for this approach are shown in Table 5.1 under backend names “Torch_CPU + Torch_GPU” and “NumPy + CuPy.” Using a CPU backend in combination with Torch_GPU improves

Backend Name	Device	Time (second)	Speedup
Torch_CPU	CPU	347	$0.71\times$
NumPy (baseline)	CPU	246	$1.00\times$
CuPy	GPU	6.7	$36.7\times$
Torch_GPU	GPU	3.5	$70.3\times$
Torch_CPU + Torch_GPU	Mixed	2.6	$94.8\times$
NumPy + CuPy	Mixed	2.1	$117\times$

Table 5.1: Time for full QAOA expectation value simulation using backend that utilize GPUs or CPUs. The expectation value is MaxCut on a 3-regular graph of size 30 and QAOA depth $p = 4$. **Speedup** shows the overall runtime improvement compared with the baseline CPU backend “NumPy”. “Mixed” device means the backend uses both CPU and GPU devices.

the performance by $1.2\times$, and for CuPy the improvement is $3\times$. These results suggest that using a combination of NumPy + Torch_GPU has the potential to give the best results.

We have evaluated the GPU performance of tensor network contraction for the energy calculation of QAOA. The problem is largely inhomogeneous with a lot of small buckets and a few very large buckets. Most of the improvement comes from using GPUs on large buckets, with up to $300\times$ speed improvement. On the other hand, the contraction of smaller tensors is faster on CPUs. In general, if the maximum bucket width of a lightcone is less than ~ 17 , the improvement from using GPUs is marginal. In addition, large buckets require a lot of memory. For example, a bucket of width 27 produces a tensor with 27 dimensions of size 2, and the memory requirement for `complex128` data type is 2 GB. In practice, these calculations are feasible up to width ~ 29 .

5.3.4 Mixed Merged Backend Results

Since the performance of the NumPy-CuPy hybrid backend is the best among all implemented hybrid backends, cross-testing between merged backends and hybrid backends focuses on the combination of the NumPy backend and CuPy backend. Because of the API constraint, the hybrid of a regular NumPy backend and a merged CuPy backend was not implemented.

Backend Name	Device	Time (seconds)	Speedup
NumPy_Merged	CPU	383	0.64×
NumPy (baseline)	CPU	246	1.00×
CuPy	GPU	6.7	36.7×
CuPy_Merged	GPU	5.6	43.9×
NumPy + CuPy	Mixed	2.1	117×
NumPy_Merged + CuPy_Merged	Mixed	1.4	176×

Table 5.2: Time for full QAOA expectation value simulation using different Merged backends, as described in Section 5.2.3. The expectation value is MaxCut on a 3-regular graph of size 30 and QAOA depth $p = 4$. **Speedup** shows the overall runtime improvement compared with the baseline CPU backend “NumPy”.

In Table 5.2, merging buckets provide a performance boost for the CuPy backend and Numpy + CuPy hybrid backend but not the NumPy backend. CuPy_Merged is 20% faster than CuPy, and NumPy_Merged + CuPy_Merged is 50% faster than its regular counterpart. However, NumPy_Merged has an significant slowdown compared with the baseline NumPy, suggesting that combining the regular NumPy backend with the merged CuPy backend can provide more speedup for the future.

In Fig. 5.5, CPU performance is better than GPU performance when the bucket width is approximately less than 15. After 15, GPU performance scales with width much better than that of CPU performance, providing a significant speed boost over the CPU in the end. GPU performance of the hybrid backend is better than that of pure GPU backend for buckets of width ≥ 15 . This speedup of the hybrid backend is likely caused by less garbage handling for the GPU since most buckets aren’t stored on GPU memory.

5.3.5 Synthetic Benchmarks

We also benchmark the time required for the basic operations: matrix multiplication, tensor network contraction with fixed contraction indices, and tensor network contraction with random indices, as well as circuit contractions.

The summary of the results is shown in Table 5.3, which compares FLOPs count for

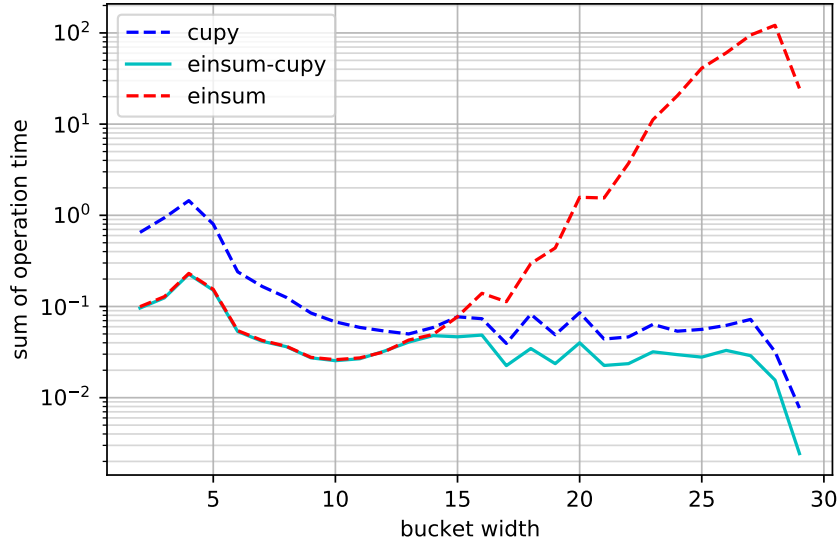


Figure 5.5: Breakdown of sum contraction time by bucket width for merged backends. CPU backends are better for buckets of width ≤ 15 , and GPU backends are better for larger buckets. The hybrid backend’s GPU backend spends outperforms the regular GPU backend for buckets of width ≥ 15 .

similar-sized problems of different types. Figures 5.6 and 5.7 show dependence of FLOPs vs problem size for different problems. We observe 80% of theoretical peak performance on GPU for matrix multiplication. Switching to pairwise tensor network contraction shows similar FLOPs for GPU, while for CPU, it results in $10\times$ FLOPs decrease. A significant reduction in performance comes from switching from pairwise tensor network contractions of a tensor with few dimensions of large size to tensors with many permuted dimensions and small size. This reduction in performance is about $10\times$ for both CPU and GPU. This observation suggests that further improvement can be achieved by reformulating the tensor network operations in a smaller tensor by transposing and merging the dimensions of participating tensors. It is partially addressed in using the merged indices approach, where the contraction dimension is increased. The “Bucket Contraction Merged” task shows 45% of theoretical peak performance, which significantly improves compared to the unmerged counterpart.

The significant reduction of performance comes when we compare bucket contraction and

Task	CPU FLOPs	GPU FLOPs
Matrix Multiplication	50.1G	2.38T
Tensor Network Fixed Contraction	5.53G	1.36T
Tensor Network Random Contraction	640M	97.5G
Bucket Contraction Unmerged	241M	61.9G
Bucket Contraction Merged	542M	1.14T
Lightcone Contraction Unmerged	326M	4.92G
Lightcone Contraction Merged	177M	3.1G
Circuit Contraction Mixed	30.7G	

Table 5.3: Summary of GPU and CPU FLOPs for different tasks at around 100 million operations. Matrix Multiplication and Tensor Contraction tasks are described in Section 5.3.5. “Bucket Contraction” groups record the maximum number of FLOPs for a single bucket. “Lightcone Contraction” groups contain the FLOPs data on a single lightcone where the sum of operations is approximately 100 millions, small and large buckets combined.

full circuit contraction. It was explained in detail in Section 5.3.1 and is caused by overhead from small buckets. It is evident from Figure 5.3 that most of the time in GPU simulation is spent on overhead from small bucket contraction. This issue is addressed by implementing the mixed backend approach.

It is also notable that the merged approach does not improve the performance for CPU backends which is probably due to an inefficient implementation of original `numpy.einsum()`.

Matrix Multiplication

The multiplication of square matrices of size 465 needs approximately 100 million complex operations according to our calculation of operations value. The average operation time for the multiplication of two randomly generated `complex128` square matrices of size 465 is 0.3 ms on the GPU, which achieves 50× speedup compared with the operation time of 16 ms on the CPU; NumPy produces 50G FLOPs on CPU, and the GPU backend CuPy reaches 2.38T FLOPs for this operation. We observe that the CPU backend has an advantage in performing small operations: for matrices of size 10×10 , the CPU backend NumPy spends only 5.8 μ s for the multiplication, while the best GPU backend PyTorch spends 27 μ s on the

operation. When the matrix size is less than 2000×2000 for the GPU backends, PyTorch outperforms CuPy, and CuPy is slightly better for much larger operations. Moreover, the operation time for both CPU and GPU backends decreases slightly when the size of matrices increases from 1000 to 1024 and from 4090 to 4096.

Fixed Tensor Network Contraction

We use the fixed contraction formula “abcd,bcdf \rightarrow acf” and control the size of the tensor indices from 10 to 100. Even for the smallest case when the number of operations is 100,000 with indices of size 10, the slowest GPU backend is faster than the CPU backend Numpy, which spends 0.3 ms on the contraction. For the GPU backends, we achieve 1.36T FLOPs for this fixed contraction, which is 57% of the recorded peak performance. In accordance with the matrix multiplication results, the CuPy backend performs better than the PyTorch backend in the fixed tensor network contractions only when the number of operations is greater than 1G.

Random Tensor Network Contraction

We let the number of indices be any number between 4 and 25, and we set the size of each mode to be 2. For example, we have 5 indices in total, and we randomly generate a contraction sequence “caedb,eab \rightarrow cde,” so the sizes of the input tensors are 2^5 and 2^3 , resulting in an output tensor of size 2^3 . We reach 97.5 G FLOPs for the GPU backends and 640 M for the CPU backend only when performing this random contraction. As shown in Fig. 5.6, the mean FLOPs drop significantly when we use random contraction (in green) instead of fixed contraction (in red) on the CuPy backend. On the CPU, the gap increases with the increasing number of operations according to Fig. 5.7. Therefore, contractions on tensors with small numbers of indices of large size have better performance than contractions on tensors with many indices of small size. The "tncontract random" group is designed to

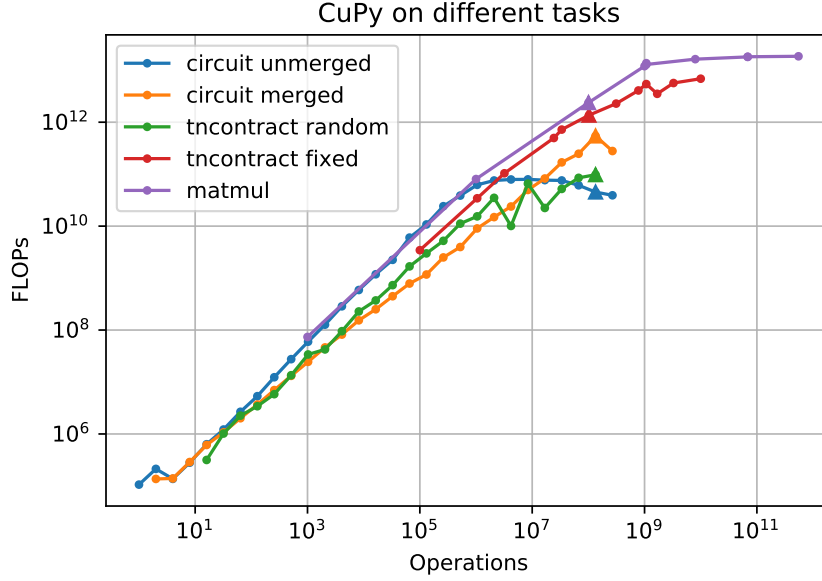


Figure 5.6: FLOPs vs. the number of operations for all tasks on the CuPy backend. “circuit unmerged” and “circuit merged” are results of expectation value of the full circuit simulation of QAOA MaxCut problem on a 3-regular graph of size 30 with depth $p = 4$. “tncontract random” tests on tensors of many indices where each index has a small size. “tncontract fixed” uses the contraction sequence “abcd,bcdf \rightarrow acf” for all contractions. “matmul” performs matrix multiplication on square matrices. All groups use `complex128` tensors in the operation. We use the triangles to denote the data at ~ 100 million operations, which is shown in Table 5.3.

break down the circuit simulation to tensor contraction operations, so it overlaps with the results from the “bucket unmerged” group in Fig. 5.6. From the difference in performance of the random and the fixed tensor contraction group, we design the merged bucket group to improve the performance of contractions. Our goal is to make the bucket simulation curve close to the tensor contraction fixed group (the red curve).

5.4 Conclusions

This work has demonstrated that GPUs can significantly speed up quantum circuit simulations using tensor network contractions. We demonstrate that GPUs are best for contracting large tensors, while CPUs are slightly better for small tensors. Moving the computation onto

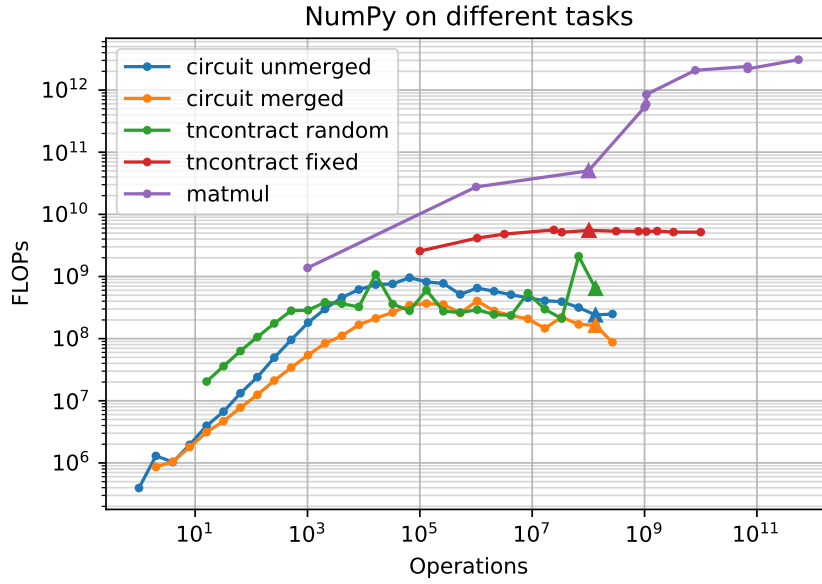


Figure 5.7: FLOPs vs. the number of operations for all tasks on NumPy backend. Same problem setting as Fig. 5.6. “tncontract random” outperforms “tncontract fixed” as the ops value increases. Merged backend does not have an advantage on CPU compared to the unmerged backend. We use the triangles to denote the data at ~ 100 million operations, which is shown in Table 5.3.

GPUs can dramatically speed up the computation. We propose to use a contraction backend that dynamically assigns the CPU or GPU device to tensors based on their size. This mixed backend approach demonstrated a $176\times$ improvement in time to solution.

We observe up to $300\times$ speedup on GPU compared to CPU for individual large buckets. In general, if the maximum bucketwidth of a lightcone is less than ~ 17 , the improvement from using GPUs is marginal. It underlines the importance of using a mixed CPU/GPU backend for tensor contraction and using device selection for the tensor at runtime to achieve the maximum performance. On NVIDIA DGX-2 server we found out that the threshold is ~ 15 , but it may change for other computing systems.

We also demonstrated the performance of the merged indices approach, which improves the arithmetic intensity and provides a significant FLOP improvement. Our synthetic benchmarks for various tensor contraction tasks suggest that additional improvement can be obtained by transposing and reshaping tensors in pairwise contractions.

The main conclusion of this chapter is that we found that GPUs can dramatically increase the speed of tensor contractions for large tensors. The smaller tensors need to be computed on a CPU only because of overhead to move on and off data to a GPU. We show that the approach of merged indices allows to speed up large tensors contraction, but it does not solve the problem completely. Where to compute tensors leads to the problem of optimal load balancing between CPU and GPU. This potential issue will be the subject of our future work, as well as testing of the performance of the code on new NVidia DGX systems and GPU supercomputers using cuTensor and cuQuantum software packages developed by NVidia.

CHAPTER 6

CONCLUSIONS AND OUTLOOK

A naïve approach to simulating observables of an arbitrary quantum system scales exponentially with the system size. However, by utilizing the structure of interactions between system components, it is possible to simulate the system more efficiently. This approach can reduce the exponential factor or in some cases change the scaling to linear in system size. The latter case is possible due to lightcone optimization for calculation of energy expectation values. In the case of calculation of probability amplitudes, the tensor network approach allows calculating a batch of several amplitudes for the same cost as a single amplitude. If the contraction width of the tensor network is w , then one can calculate the batch of 2^w probability amplitudes without significant increase in computational cost. In the application to the quantum circuit simulation, it is possible to exploit the structure of quantum gates in the circuit. For gates that have non-zero elements only on their diagonal, the “Diagonal gates” optimization is possible and provides a significant performance improvement.

A key part of the process of tensor network contraction is the contraction ordering algorithm. The cost for contraction depends exponentially on the quality of the contraction order. It, therefore, promises to be a rewarding task to study different contraction order algorithms. The tensor networks used for most quantum many-body simulation problems have a lot of small indices. This setup is not beneficial for the numerical efficiency of tensor network contraction, since the contraction of two tensors is dominated by read/write operations, not arithmetic operations. This puts an additional constraint on the optimization of the contraction procedure of tensor networks. As discussed above, multiple approaches can be utilized to improve the performance of parallelized GPU-accelerated tensor network contraction.

To further improve the scaling of the tensor network approach, it is possible to borrow the idea of approximate simulations using the tensor decomposition, as used in the DMRG

algorithm. This approach can dramatically reduce the cost for simulation for a small error in the resulting value.

REFERENCES

- Argonne National Laboratory Leadership Computing Facility, 2017.
- U.S. Department of Energy Office of Science Innovative and Novel Computational Impact on Theory and Experiment (INCITE) program, 2017.
- Scott Aaronson and Lijie Chen. Complexity-theoretic foundations of quantum supremacy experiments. *arXiv preprint arXiv:1612.05903*, 2016.
- Yuri Alexeev, Dave Bacon, Kenneth R Brown, Robert Calderbank, Lincoln D Carr, Frederic T Chong, Brian DeMarco, Dirk Englund, Edward Farhi, Bill Fefferman, Alexey Gershkov, Andrew Houck, Jungsang Kim, Shelby Kimmel, Michael Lange, Seth Lloyd, Mikhail Lukin, Dmitri Maslov, Peter Maunz, Christopher Monroe, John Preskill, Martin Roetteler, Martin Savage, and Jeff Thompson. Quantum computer systems for scientific discovery. *PRX Quantum*, 2(1):017001, 2021.
- 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.
- Ethan Bernstein and Umesh Vazirani. Quantum complexity theory. *SIAM Journal on computing*, 26(5):1411–1473, 1997.
- Jacob Biamonte and Ville Bergholm. Tensor networks in a nutshell, 2017.
- Jean RS Blair and Barry Peyton. An introduction to chordal graphs and clique trees. In *Graph theory and sparse matrix computation*, pages 1–29. Springer, 1993.
- Hans L Bodlaender. A tourist guide through treewidth. *Acta cybernetica*, 11(1-2):1, 1994.
- Hans L Bodlaender, Fedor V Fomin, Arie MCA Koster, Dieter Kratsch, and Dimitrios M Thilikos. On exact algorithms for treewidth. In *European Symposium on Algorithms*, pages 672–683. Springer, 2006.
- Sergio Boixo. Random circuits dataset. <https://github.com/sboixo/GRCS.git>, 2019. Accessed: 2019-09-16.
- 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.
- 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.

- Jacob C Bridgeman and Christopher T Chubb. Hand-waving and interpretive dance: an introductory course on tensor networks. *Journal of Physics A: Mathematical and Theoretical*, 50(22):223001, 2017.
- Thang Nguyen Bui and Curt Jones. A heuristic for reducing fill-in in sparse matrix factorization. Technical report, Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1993.
- Juan Carrasquilla, Di Luo, Felipe Pérez, Ashley Milsted, Bryan K Clark, Maksims Volkovs, and Leandro Aolita. Probabilistic simulation of quantum circuits with the transformer. *arXiv preprint arXiv:1912.11052*, 2019.
- Jianxin Chen, Fang Zhang, Mingcheng Chen, Cupjin Huang, Michael Newman, and Yaoyun Shi. Classical simulation of intermediate-size quantum circuits. *arXiv preprint arXiv:1805.01450*, 2018a.
- Jianxin Chen, Fang Zhang, Cupjin Huang, Michael Newman, and Yaoyun Shi. Classical simulation of intermediate-size quantum circuits. *arXiv*, may 2018b.
- Yu Chen, C Neill, P Roushan, N Leung, M Fang, R Barends, J Kelly, B Campbell, Z Chen, B Chiaro, et al. Qubit architecture with high coherence and fast tunable coupling. *Physical review letters*, 113(22):220502, 2014.
- Zhao-Yun Chen, Qi Zhou, Cheng Xue, Xia Yang, Guang-Can Guo, and Guo-Ping Guo. 64-qubit quantum circuit simulation. *Science Bulletin*, 63(15):964–971, 2018c.
- Lam Chi-Chung, P Sadayappan, and Rephael Wenger. On optimizing a class of multi-dimensional loops with reduction for parallel execution. *Parallel Processing Letters*, 7(02):157–168, 1997.
- Andrzej Cichocki, Namgil Lee, Ivan Oseledets, Anh-Huy Phan, Qibin Zhao, Danilo P Mandic, et al. Tensor networks for dimensionality reduction and large-scale optimization: Part 1 – low-rank tensor decompositions. *Foundations and Trends® in Machine Learning*, 9(4-5):249–429, 2016.
- Hans De Raedt, Fengping Jin, Dennis Willsch, Madita Willsch, Naoki Yoshioka, Nobuyasu Ito, Shengjun Yuan, and Kristel Michielsen. Massively parallel quantum computer simulator, eleven years later. *Computer Physics Communications*, 237:47–61, 2019.
- Koen De Raedt, Kristel Michielsen, Hans De Raedt, Binh Trieu, Guido Arnold, Marcus Richter, Th Lippert, H Watanabe, and N Ito. Massively parallel quantum computer simulator. *Computer Physics Communications*, 176(2):121–136, 2007.
- Rina Dechter. Bucket elimination: A unifying framework for several probabilistic inference. *CoRR*, abs/1302.3572, 2013.

- Jack J Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 14(1):1–17, 1988.
- Daniel J Egger, Jakub Mareček, and Stefan Woerner. Warm-starting quantum optimization. *Quantum*, 5:479, 2021.
- Paul Erdős and Alfréd Rényi. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci.*, 5(1):17–60, 1960.
- Edward Farhi and Aram W Harrow. Quantum supremacy through the quantum approximate optimization algorithm. *arXiv preprint arXiv:1602.07674*, 2016.
- Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. A quantum approximate optimization algorithm. *arXiv preprint arXiv:1411.4028*, 2014.
- Richard P. Feynman. Simulating physics with computers. 21(6-7):467–488, June 1982. doi:10.1007/bf02650179.
- E. Schuyler Fried, Nicolas P. D. Sawaya, Yudong Cao, Ian D. Kivlichan, Jhonathan Romero, and Alán Aspuru-Guzik. qtorch: The quantum tensor contraction handler. *PLOS ONE*, 13(12):e0208510, Dec 2018. ISSN 1932-6203. doi:10.1371/journal.pone.0208510.
- Fănică Gavril. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *SIAM Journal on Computing*, 1(2):180–187, 1972.
- Edward Gillman, Dominic C. Rose, and Juan P. Garrahan. A tensor network approach to finite markov decision processes, 2020.
- 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.
- Vibhav Gogate and Rina Dechter. A complete anytime algorithm for treewidth, 2012.
- Johnnie Gray and Stefanos Kourtis. Hyper-optimized tensor network contraction, 2020.
- Michael Hamann and Ben Strasser. Correspondence between multilevel graph partitions and tree decompositions. *Algorithms*, 12(9):198, 2019.
- Thomas Häner and Damian S Steiger. 0.5 petabyte simulation of a 45-qubit quantum circuit. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 33. ACM, 2017.
- Aram W Harrow and Ashley Montanaro. Quantum computational supremacy. *Nature*, 549(7671):203, 2017.

- IBM. Ibm q experience, 2018.
- Intel. 2018 ces: Intel advances quantum and neuromorphic computing research, 2018.
- Sami Khairy, Ruslan Shaydulin, Lukasz Cincio, Yuri Alexeev, and Prasanna Balaprakash. Learning to optimize variational quantum circuits to solve combinatorial problems. In *Proceedings of the Thirty-Fourth AAAI Conference on Artificial Intelligence (AAAI)*, 2020.
- Ton Kloks. *Treewidth: computations and approximations*, volume 842. Springer Science & Business Media, 1994.
- Ton Kloks, H Bodlaender, Haiko Müller, and Dieter Kratsch. Computing treewidth and minimum fill-in: All you need are the minimal separators. In *European Symposium on Algorithms*, pages 260–271. Springer, 1993.
- Riling Li, Bujiao Wu, Mingsheng Ying, Xiaoming Sun, and Guangwen Yang. Quantum supremacy circuit simulation on Sunway Taihulight. *arXiv preprint arXiv:1804.04797*, 2018.
- Norbert M Linke, Dmitri Maslov, Martin Roetteler, Shantanu Debnath, Caroline Figgatt, Kevin A Landsman, Kenneth Wright, and Christopher Monroe. Experimental comparison of two quantum computing architectures. *Proceedings of the National Academy of Sciences*, 114(13):3305–3310, 2017.
- Danil Lykov, Roman Schutski, Valerii Vinokur, and Yuri Alexeev. Large-Scale Parallel Tensor Network Quantum Simulator. In *under review of Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '20*, New York, New York, USA, 2020a. ACM Press.
- Danylo Lykov. QTensor. <https://github.com/danlkv/qtensor>, 2021.
- Danylo Lykov and Yuri Alexeev. Importance of diagonal gates in tensor network simulations, 2021.
- Danylo Lykov, Roman Schutski, Alexey Galda, Valerii Vinokur, and Yurii Alexeev. Tensor network quantum simulator with step-dependent parallelization. *arXiv preprint arXiv:2012.02430*, 2020b.
- Danylo Lykov, Angela Chen, Huaxuan Chen, Kristopher Keipert, Zheng Zhang, Tom Gibbs, and Yuri Alexeev. Performance evaluation and acceleration of the qtensor quantum circuit simulator on gpus. In *2021 IEEE/ACM Second International Workshop on Quantum Computing Software (QCS)*, pages 27–34, 2021. doi:10.1109/QCS54837.2021.00007.
- Igor L Markov and Yaoyun Shi. Simulating quantum computation by contracting tensor networks. *SIAM Journal on Computing*, 38(3):963–981, 2008.
- Stephen Marsland. *Machine learning: an algorithmic perspective*. Chapman and Hall/CRC, 2011.

- H.-D. Meyer, U. Manthe, and L.S. Cederbaum. The multi-configurational time-dependent hartree approach. 165(1):73–78, January 1990. doi:10.1016/0009-2614(90)87014-i. URL [https://doi.org/10.1016/0009-2614\(90\)87014-i](https://doi.org/10.1016/0009-2614(90)87014-i).
- Jacob Miller, Geoffrey Roeder, and Tai-Danae Bradley. Probabilistic graphical models and tensor networks: A hybrid framework, 2021.
- Vladimir Mironov, Yuri Alexeev, Kristopher Keipert, Michael D’Amello, Alexander Moskovsky, and Mark S. Gordon. An efficient MPI/OpenMP parallelization of the Hartree-Fock method for the second generation of Intel Xeon Phi processor. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC ’17*, pages 1–12, New York, New York, USA, 2017. ACM Press.
- Uwe Naumann and Olaf Schenk. *Combinatorial scientific computing*. CRC Press, 2012.
- Charles Neill, Pedran Roushan, K Kechedzhi, Sergio Boixo, Sergei V Isakov, V Smelyanskiy, A Megrant, B Chiaro, A Dunsworth, K Arya, et al. A blueprint for demonstrating quantum supremacy with superconducting qubits. *Science*, 360(6385):195–199, 2018.
- Lars Onsager. Crystal statistics. i. a two-dimensional model with an order-disorder transition. *Phys. Rev.*, 65:117–149, Feb 1944. doi:10.1103/PhysRev.65.117. URL <https://link.aps.org/doi/10.1103/PhysRev.65.117>.
- Román Orús. A practical introduction to tensor networks: Matrix product states and projected entangled pair states. *Annals of Physics*, 349:117–158, Oct 2014. ISSN 0003-4916. doi:10.1016/j.aop.2014.06.013. URL <http://dx.doi.org/10.1016/j.aop.2014.06.013>.
- Matthew Otten. QuaC (quantum in c) is a parallel time dependent open quantum systems solver, 2020.
- Feng Pan, Pengfei Zhou, Sujie Li, and Pan Zhang. Contracting arbitrary tensor networks: general approximate algorithm and applications in graphical models and quantum circuit simulations. *arXiv preprint arXiv:1912.03014*, 2019.
- Edwin Pednault, John A Gunnels, Giacomo Nannicini, Lior Horesh, Thomas Magerlein, Edgar Solomonik, and Robert Wisnieff. Breaking the 49-qubit barrier in the simulation of quantum circuits. *arXiv preprint arXiv:1710.05867*, 2017.
- Robert NC Pfeifer, Jutho Haegeman, and Frank Verstraete. Faster identification of optimal contraction sequences for tensor networks. *Physical Review E*, 90(3):033315, 2014.
- Dorit Ron, Ilya Safro, and Achi Brandt. Relaxation-based coarsening and multiscale graph organization. *Multiscale Modeling & Simulation*, 9(1):407–423, 2011.
- Ilya Safro, Dorit Ron, and Achi Brandt. Multilevel algorithms for linear ordering problems. *Journal of Experimental Algorithmics (JEA)*, 13:1–4, 2009.

- Ulrich Schollwöck. The density-matrix renormalization group in the age of matrix product states. *Annals of Physics*, 326(1):96–192, Jan 2011. ISSN 0003-4916. doi:10.1016/j.aop.2010.09.012. URL <http://dx.doi.org/10.1016/j.aop.2010.09.012>.
- Roman Schutski, Danil Lykov, and Ivan Oseledets. Adaptive algorithm for quantum circuit simulation. *Phys. Rev. A*, 101:042335, Apr 2020. doi:10.1103/PhysRevA.101.042335.
- Ruslan Shaydulin and Yuri Alexeev. Evaluating quantum approximate optimization algorithm: A case study. In *Proceedings of the 2nd International Workshop on Quantum Computing for Sustainable Computing*, 2019.
- Ruslan Shaydulin, Ilya Safro, and Jeffrey Larson. Multistart methods for quantum approximate optimization. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8. IEEE, 2019a.
- Ruslan Shaydulin, Hayato Ushijima-Mwesigwa, Christian FA Negre, Ilya Safro, Susan M Mniszewski, and Yuri Alexeev. A hybrid approach for solving optimization problems on small quantum computers. *Computer*, 52(6):18–26, 2019b.
- Ruslan Shaydulin, Stuart Hadfield, Tad Hogg, and Ilya Safro. Classical symmetries and QAOA. *arXiv preprint arXiv:2012.04713*, 2020.
- 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.
- Mikhail Smelyanskiy, Nicolas PD Sawaya, and Alán Aspuru-Guzik. qHiPSTER: the quantum high performance software testing environment. *arXiv preprint arXiv:1601.07195*, 2016.
- Hisao Tamaki. Positive-Instance Driven Dynamic Programming for Treewidth. In Kirk Pruhs and Christian Sohler, editors, *25th Annual European Symposium on Algorithms (ESA 2017)*, volume 87 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 68:1–68:13, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-049-1. doi:10.4230/LIPIcs.ESA.2017.68.
- Robert E Tarjan and Mihalis Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on computing*, 13(3):566–579, 1984.
- Laurens Vanderstraeten, Bram Vanhecke, and Frank Verstraete. Residual entropies for three-dimensional frustrated spin systems with tensor networks. 98(4), October 2018. doi:10.1103/physreve.98.042145. URL <https://doi.org/10.1103/physreve.98.042145>.
- Benjamin Villalonga, Sergio Boixo, Bron Nelson, Christopher Henze, Eleanor Rieffel, Rupak Biswas, and Salvatore Mandrà. A flexible high-performance simulator for verifying and benchmarking quantum circuits implemented on real hardware. *NPJ Quantum Information*, 5:1–16, 2019.

- Benjamin Villalonga, Dmitry Lyakh, Sergio Boixo, Hartmut Neven, Travis S Humble, Rupak Biswas, Eleanor Rieffel, Alan Ho, and Salvatore Mandrà. Establishing the quantum supremacy frontier with a 281 Pflop/s simulation. *Quantum Science and Technology*, 2020.
- Haobin Wang and Michael Thoss. Multilayer formulation of the multiconfiguration time-dependent hartree theory. 119(3):1289–1299, July 2003. doi:10.1063/1.1580111. URL <https://doi.org/10.1063/1.1580111>.
- Zhihui Wang, Stuart Hadfield, Zhang Jiang, and Eleanor G Rieffel. Quantum approximate optimization algorithm for maxcut: A fermionic view. *Physical Review A*, 97(2):022304, 2018.
- Xin-Chuan Wu, Sheng Di, Franck Cappello, Hal Finkel, Yuri Alexeev, and Frederic Chong. Memory-efficient quantum circuit simulation by using lossy data compression. In *Proceedings of the 3rd International Workshop on Post-Moore Era Supercomputing (PMES) at SC18*, Denver, CO, USA, 2018a.
- Xin-Chuan Wu, Sheng Di, Franck Cappello, Hal Finkel, Yuri Alexeev, and Frederic T Chong. Amplitude-aware lossy compression for quantum circuit simulation. In *Proceedings of 4th International Workshop on Data Reduction for Big Scientific Data (DRBSD-4) at SC18*, 2018b.
- Xin-Chuan Wu, Sheng Di, Emma Maitreyee Dasgupta, Franck Cappello, Hal Finkel, Yuri Alexeev, and Frederic T Chong. Full-state quantum circuit simulation by using data compression. In *Proceedings of the High Performance Computing, Networking, Storage and Analysis International Conference (SC19)*, Denver, CO, USA, 2019. IEEE Computer Society.
- Ya-Qian Zhao, Ren-Gang Li, Jin-Zhe Jiang, Chen Li, Hong-Zhen Li, En-Dong Wang, Wei-Feng Gong, Xin Zhang, and Zhi-Qiang Wei. Simulation of quantum computing on classical supercomputers, 2020.