THE UNIVERSITY OF CHICAGO

LARGE-SCALE CLASSICAL SIMULATIONS FOR DEVELOPMENT OF QUANTUM OPTIMIZATION ALGORITHMS

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

DEPARTMENT OF COMPUTER SCIENCE

BY
DANYLO LYKOV

CHICAGO, ILLINOIS
MAY 2024

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

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.

These improvements allow to predict the performance of Quantum Approximate Optimization Algorithm (QAOA) on MaxCut problem without running the quantum optimization. This predicted QAOA performance is then used to systematically compare to the classical MaxCut solvers. It turns out that one needs to simulate a deep quantum circuit in order to compete with classical solvers.

In search for provable speedups, I then turn to a different combinatorial optimization problem, Low-Autocorrelation Binary Sequences (LABS). This study involves simulating ground state overlap for deep circuits. This task is more suited for a state vector simulation,

which requires advanced distributed algorithms when done at scale.

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:

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

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:

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

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:

$$|\psi_N\rangle = \sum_{s_1 \ldots s_N=0}^{1} C_{s_1 \ldots s_N} |s_1 \ldots s_N\rangle . \tag{1.3}$$

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

amples 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:

$$T_{ijk} = A_{ij}B_{jk}. \tag{1.4}$$

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:

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

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

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

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 $\texttt{NOT}(b_0)$ which flips the input bit, and $\texttt{AND}(b_0, b_1)$ 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

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

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:

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

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

$$\hat{U}^i\,|0_0 0_1 \ldots s_i \ldots 0_N\rangle = c_{0s}\,|0_0 0_1 \ldots 0_i \ldots 0_N\rangle + c_{1s}\,|0_0 0_1 \ldots 1_i \ldots 0_N\rangle = |\psi\rangle\,. \tag{1.9}$$

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

$$\hat{U}^i \sum_{s_1 \ldots s_N=0}^{1} C_{s_1 \ldots s_N} |s_1 \ldots s_N\rangle = \sum_{s_1 \ldots s_N=0}^{1} C_{s_1 \ldots s_N} \hat{U}^i |s_1 \ldots s_N\rangle \tag{1.10}$$

$$= \sum_{s_1 \ldots s_N=0}^{1} C_{s_1 \ldots s_N} \sum_{s_{N+1}} U_{s_{N+1} s_i} |s_1 \ldots s_N\rangle \tag{1.11}$$

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

The last summation is performed over all indices s_i for $i = 1 \ldots N+1$ 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.

$$\sum_{s_j} U_{s_{N+1} s_j} C_{s_0 s_1 \ldots s_j \ldots s_N} = D_{s_0 s_1 \ldots s_{N+1} \ldots s_N}. \tag{1.13}$$

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

$$\sum_{i_j i_k} W_{i_{N+2} i_{N+1} i_j i_k} C_{s_0 s_1 \ldots s_j \ldots s_k \ldots s_N} = D_{s_0 s_1 \ldots i_{N+2} \ldots i_{N+1} \ldots s_N}. \tag{1.14}$$

1.3 Structure

In this work, I start with Chapter **??** 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

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

Next, in Chapter 2 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 3 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

PARALLEL COMPUTATION

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

2.1 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 5.6 we start by discussing related work. In Section 2.4 we describe tensor networks and the bucket elimination algorithm. Simulations of a single amplitude of QAOA ansatz state are described in Section 2.5. 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 2.7.

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 4.2. In Section 2.9 we summarize our results and draw conclusions.

2.2 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].

2.3 Methodology

2.3.1 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 parametirized by $2p$ variational parameters β and γ. The ansatz state obtained after p layers of the QAOA is:

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

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 optimizaiton 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.

2.3.2 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. **??**. The generated circuit were converted to tensor networks as described in Section 2.4.1. The resulting tensor network for the circuit in **??** is shown in Fig. 2.2. 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.

2.4 Overview of simulation algorithm

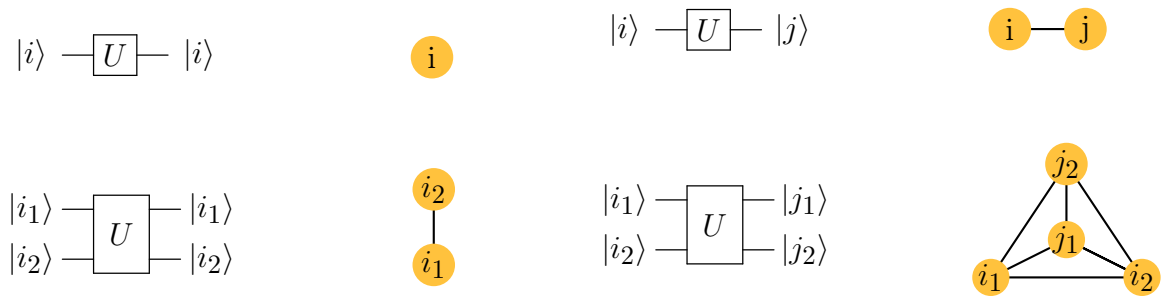
In this section, we briefly introduce the reader to the tensor network contraction algorithm. It is described in much more detail in the paper by Boixo et al. Boixo et al. [2017], and the interested reader can refer to work by Detcher et al. [Dechter, 2013] and Marsland et al. [Marsland, 2011] to gain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

2.4.1 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 2.1.

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



(a) Diagonal gates

(b) Non-diagonal gates

Figure 2.1: 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].

2.4.2 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 2.1 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	$\|\phi\rangle = \hat{X}_0 \otimes \hat{I}_1 \|\psi\rangle$	$\phi_{i'j} = X_{i'i}\psi_{ij}$
product state	$\|\psi\rangle = \|a\rangle \|b\rangle$	$\psi_{ij} = a_i a_j$
with Bell state	$\|\phi\rangle = \hat{X}_0 \otimes \hat{I}_1(\|00\rangle + \|11\rangle)$	$\phi_{i'j} = X_{i'i}\delta_{ij}$

Table 2.1: Comparison between different notations of quantum circuits

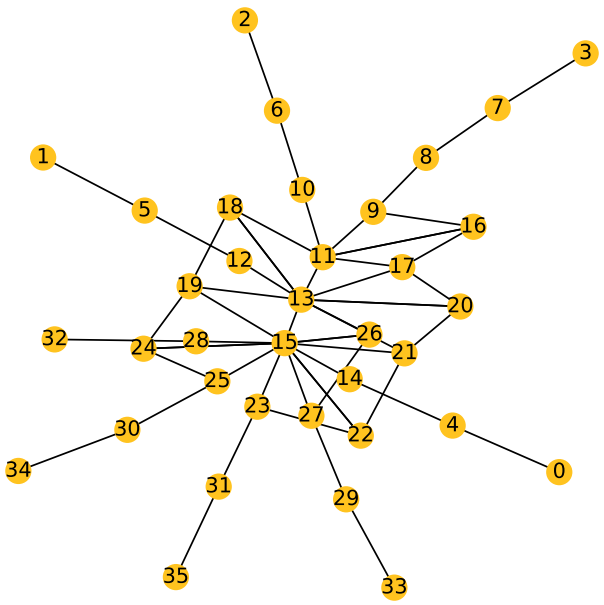


Figure 2.2: Graph representation of tensor expression of the circuit in Fig. **??**. Every vertex corresponds to a tensor index of a quantum gate. Indices are labeled right to left: 0-3 are indices of the output statevector, and 32-25 are indices of the input statevector. Self-loop edges are not shown (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 2.3 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.

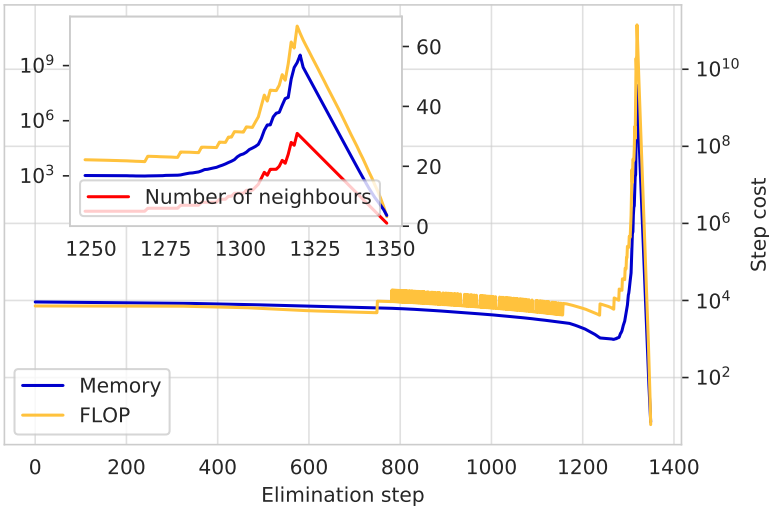


Figure 2.3: Cost of contraction for every vertex for a circuit with 150 qubits. Inset shows the peak magnified and the number of neighbors of the vertex contracted at a given step (right y-axis).

Note that the majority of contraction is very cheap, which corresponds to the low-degree nodes from Figure 2.2. 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:

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

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

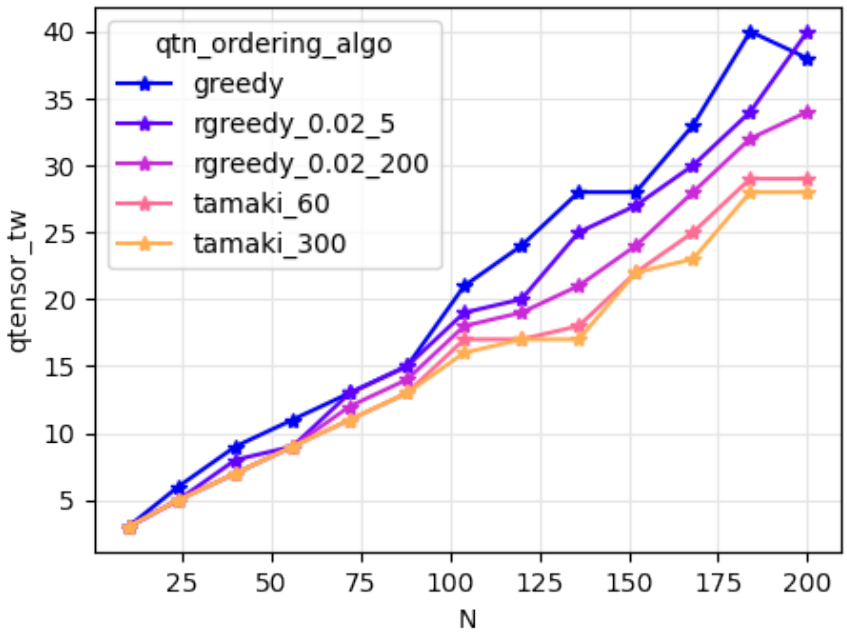


Figure 2.4: Comparison of different ordering algorithms for single amplitude simulation of QAOA ansatz state

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

2.5 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.

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

2.5.1 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.

2.6 Parallelization algorithm

We use a two-level parallelization architecture to couple the simulation structure and hardware constraints. Our approach is shown at Fig. 2.5. 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 2.6.3. A high-level picture of our algorithm is shown in Fig. 2.6.

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 2.6.2.

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 2.6.3.

2.6.1 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 2.2.

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 2.2: 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.

2.6.2 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

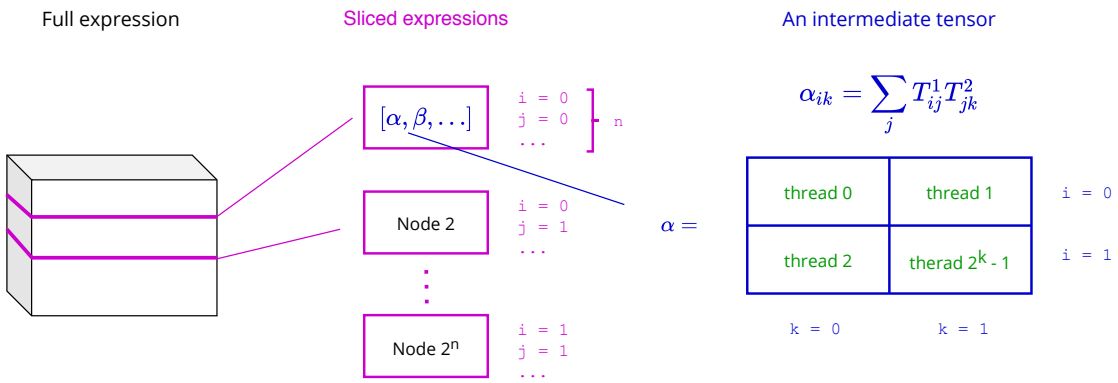


Figure 2.5: Illustration of our two-level tensor parallelization approach. On the multinode level MPI parallelization we use slicing of a partially contracted full expression. On the lower level of a single node, we use thread-based parallelization with a shared resulting tensor.

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. 2.1). 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 2.6.

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:

$$t = \max(\lfloor \frac{r-22}{2} \rfloor, 1), \tag{2.1}$$

where r is rank of the resulting tensor.

2.6.3 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:

$$\sum_{m_1 \dots m_n} \Big(\sum_{V \setminus \{m_i\}} T^1 T^2 \dots T^N \Big), \tag{2.2}$$

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

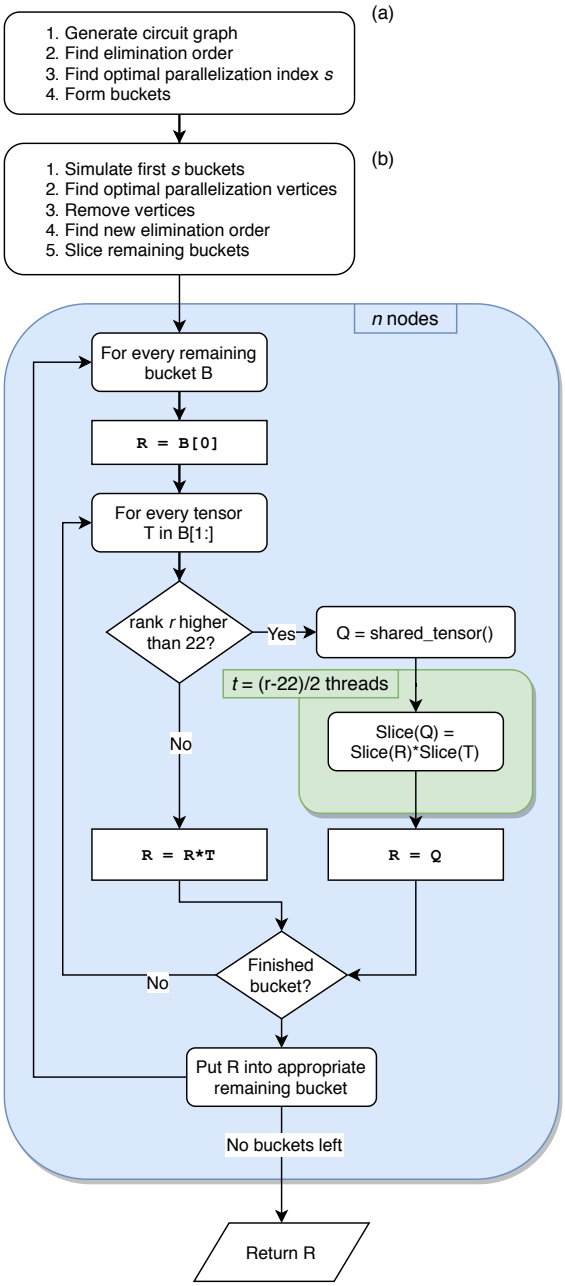


Figure 2.6: Sketch of the parallel bucket elimination algorithm. Part (a) and steps b2–b4 depend only on the structure of a task and can be executed only once for the QAOA algorithm. Steps b1 and b5 are performed serially.
The outer loop of the blue region performs the elimination of the remaining buckets; 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. 2.2 for a small circuit. As can be seen in Fig. 2.3, 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 2.8. 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 2.6, 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 1. 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. 2.1 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.

2.6.4 Step-dependent slicing

The QAOA circuit tensor expression results in a graph that has many low-degree vertices, as demonstrated in Fig. 2.2 for a small circuit. As can be seen in Fig. 2.3, 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. 2.7.

Algorithm 1 Parallel bucket elimination

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

Output:

contract_first(s, B_i) ▷ Serial part: contract first s buckets
for $i = 0, n$ **do** ▷ Find best index to slice along
    $p_i = \max\_degree\_vertex(G)$
    remove_vertex(G, p_i)
end for
$\vec{v} \leftarrow$ binary_repr(mpi_get_rank())
for $j = 0, n$ **do** ▷ Slice the expression
    $B_i \leftarrow B_i|_{p_j=v_j}$
end for
for $i = s, |V|$ **do**
    $v \leftarrow \pi^{-1}(i)$
    $R \leftarrow B_i[0]$
    for $T \in B_i[1:]$ **do** ▷ Process next bucket
        $r \leftarrow |T.\text{indexes} \cup R.\text{indexes}|$ ▷ Determine resulting size
        $t \leftarrow floor(\frac{r-22}{2})$
        if t>0 **then** ▷ Contract in thread pool
            $Q \leftarrow$ shared_tensor(r)
            $\vec{w} \leftarrow$ binary_repr$(get_thread_num())$
            $\vec{k} \leftarrow$ indices_of$(Q)[:t]$
            $Q_{v\ldots}|_{k_j=w_j} \leftarrow (Q_{v\ldots}T_{v\ldots})|_{k_j=w_j}$
            $R \leftarrow Q$ ▷ R now points to shared memory tensor
        else
            $R \leftarrow RT$
        end if
    end for
    $R \leftarrow \sum_v R$ ▷ Parallel sum can be implemented in same fashion as contraction above
    if R is scalar **then**
        result $\leftarrow$ result $\cdot R$
    else
        $k = \pi(w)$, w is the earliest index of R w.r.t π
        $B_k \leftarrow B_k \cup R$
    end if
end for
result $\leftarrow$ mpi_reduce_sum(result) ▷ Gather the results
return result

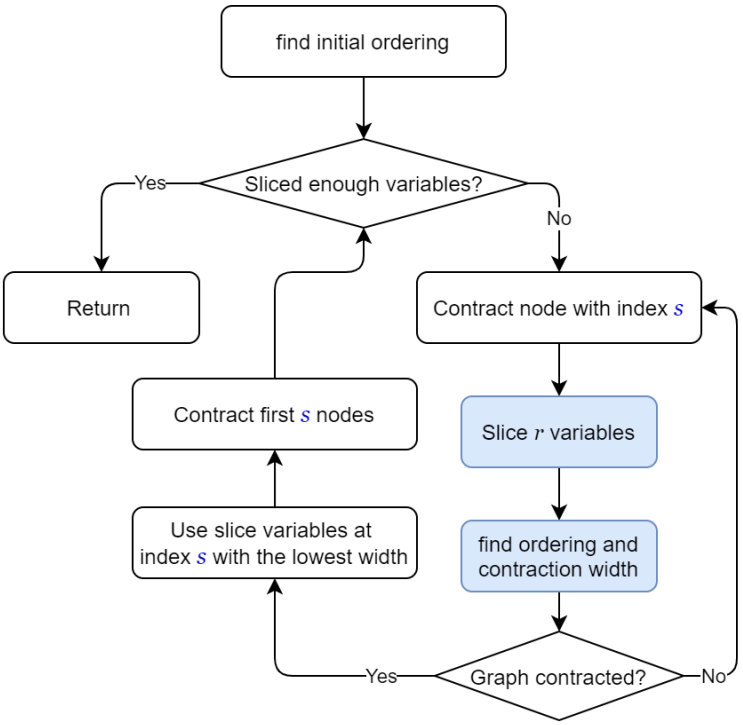


Figure 2.7: Step-based slicing algorithm. The blue boxes are evaluated for each graph node and are the main contributions to time.

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. 2.10.

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 has its own slice

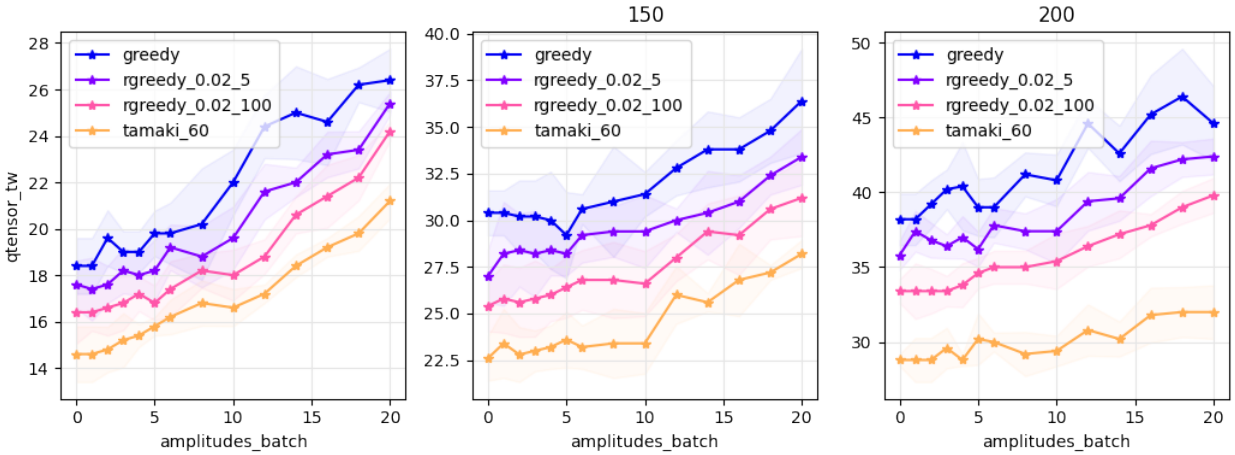


Figure 2.8: Simulation cost for a batch of amplitudes. The calculations are done for 5 random instances of degree-3 random regular graphs and the mean value is plotted. The three plots are calculated for different number of qubits: 100, 150 and 200.

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.

2.7 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

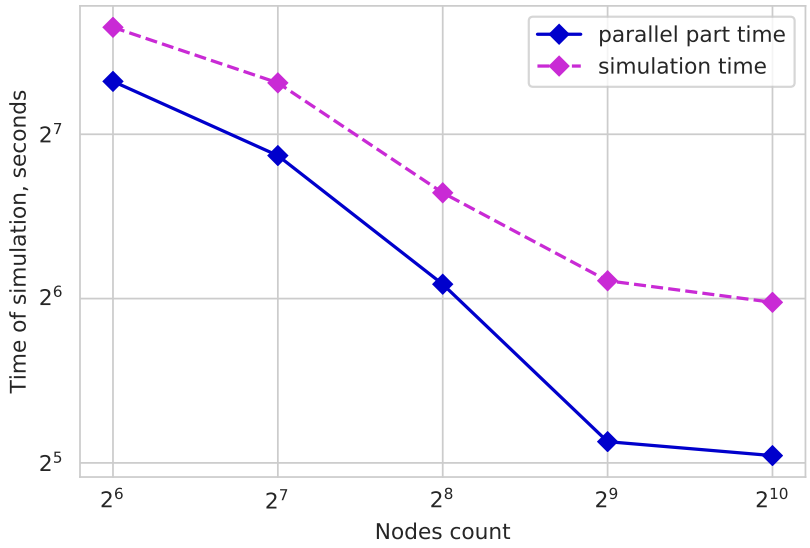


Figure 2.9: Experimental data of simulation time with respect to the number of Theta nodes. The circuit is for 210 qubits and 1,785 gates.

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

2.8 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 2.10 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. 2.9. 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.

2.9 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.

2.10 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

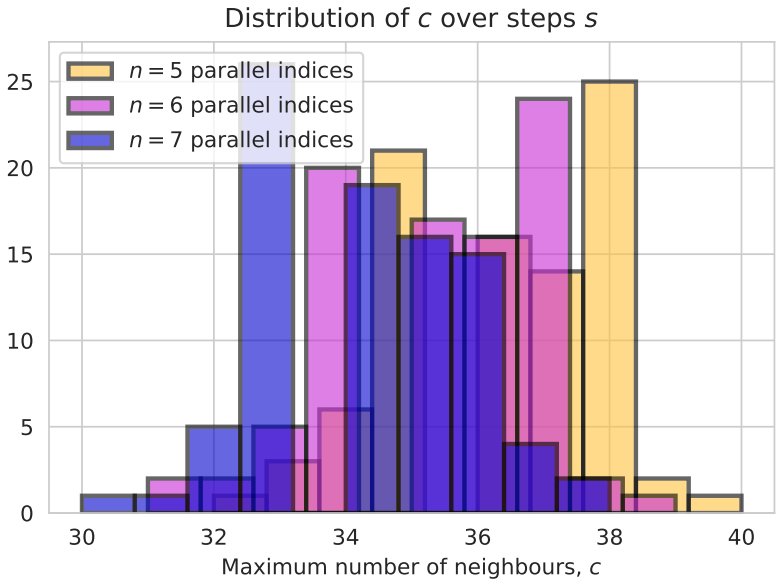


Figure 2.10: Distribution of the contraction width (maximum number of neighbors) c for different numbers of parallel indices n. While variance of c is present, showing that it is sensible to the parallelization index s, we are interested in the minimal value of s, which, in turn, generally gets smaller 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 3

GPU ACCELERATION OF TENSOR NETWORK CONTRACTION

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

3.1 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 $p = 4$.

3.2 Methodology

3.2.1 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.

$$\hat{C} = \frac{1}{2} \sum_{(ij)\in E} 1 - \hat{\sigma^z}_i \hat{\sigma^z}_j \tag{3.1}$$

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 $\hat{B}$.

$$|\vec{\gamma}, \vec{\beta}\rangle_p = \prod_{k=1}^{p} e^{-i\beta_k \hat{B}} e^{-i\gamma_k \hat{C}} |+\rangle \tag{3.2}$$

The parameterized quantum circuit (3.2) 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 $\langle C \rangle_p$.

3.2.2 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

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

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 $\hat{\sigma^z}_j \hat{\sigma^z}_k$ 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 $jk \in E$.

3.2.3 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.

3.2.4 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 3.1. 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.

3.2.5 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 3.3.5 and Figures 3.6 and 3.7.

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.

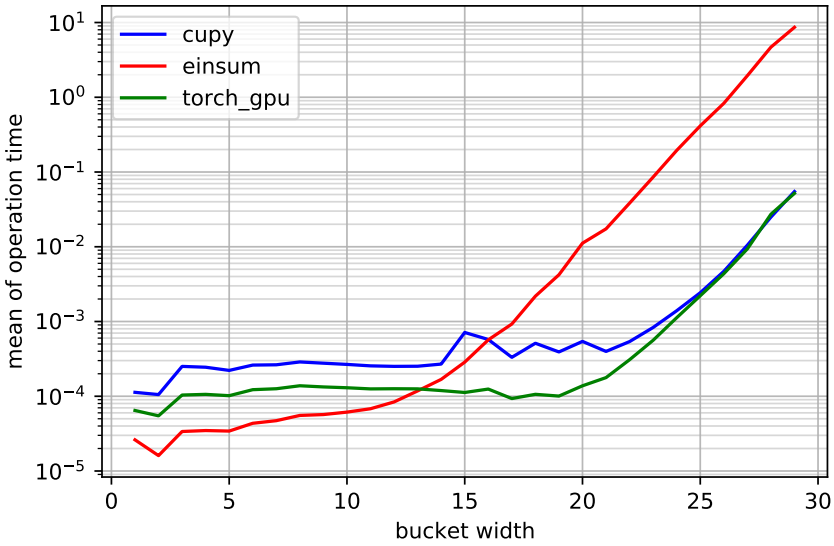


Figure 3.1: Breakdown of mean time to contract a single bucket by bucket width. The test is performed for expectation value as described in 3.3.1. CPU backends are faster for buckets of width $\leq 13 - 16$, and GPU faster are better for larger buckets. This picture also demonstrates that every contraction operation spends some time on overhead which doesn't depend on bucket width, and actual calculation that scales exponentially with bucket width.

3.3 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).

3.3.1 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 3.1, 3.2, and 3.3. This corresponds to contraction of 20 tensor networks, one network per each lightcone. Our GPU implementation of the simulator using PyTorch (labeled "torch_gpu")

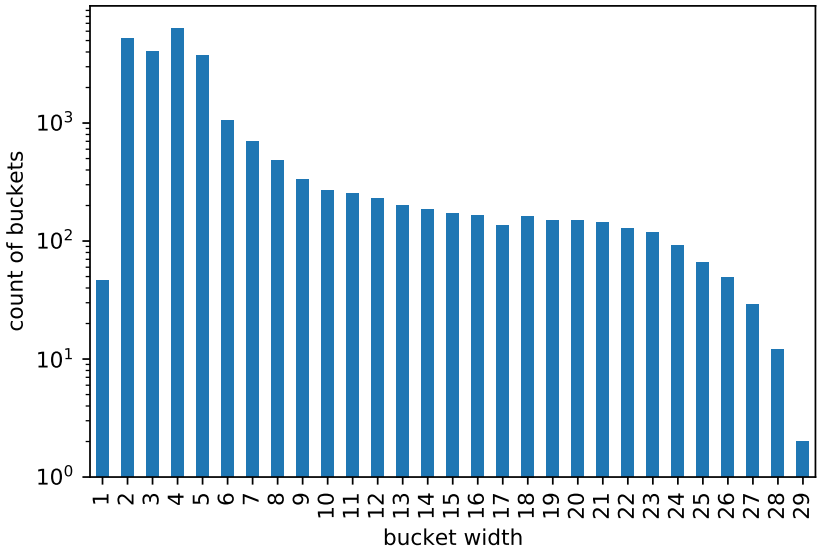


Figure 3.2: Distribution of bucket width in the contraction of QAOA full circuit simulation. The y-axis is log scale; 82% of buckets have width ≤ 6, which have relatively large overhead time.

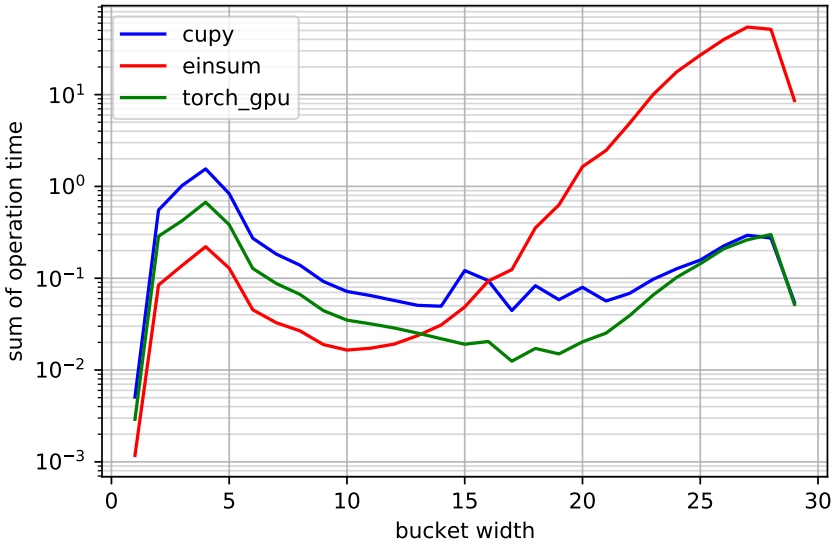


Figure 3.3: Breakdown of total time spent on bucket of each size in full QAOA expectation value simulation. The y-value on this plot is effectively one in Figure 3.1 multiplied by one in Figure 3.2. This figure is very useful for analyzing the bottlenecks of the simulation. It shows that most of the 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× speedup over the CPU baseline and 1.92× speedup over CuPy.

Figure 3.1 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 3.3 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 3.2 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 3.3 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.

3.3.2 Merged Backend Results

The "merged" groups merge the indices before performing contractions. In Fig. 3.4, 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

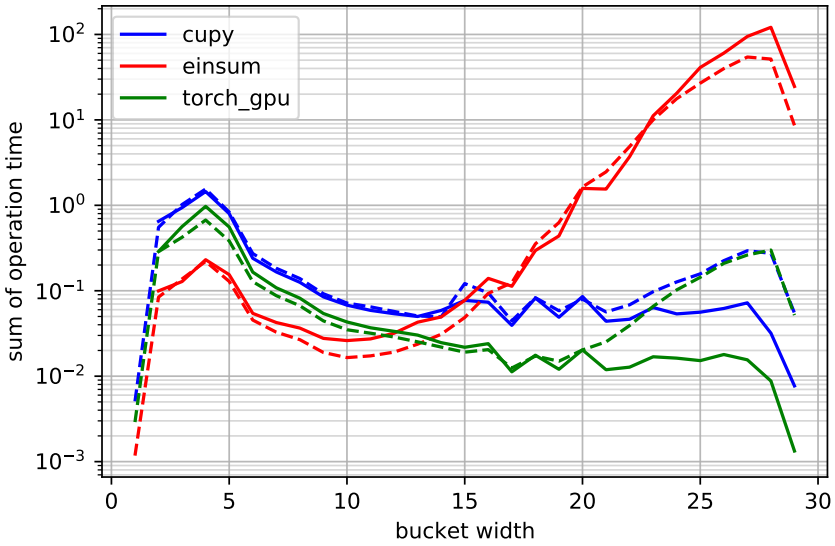


Figure 3.4: Breakdown of total contraction time by bucket width in full expectation value simulation of problem size 30. Lines with the same color use the same type of backends. The solid lines represent the merged version of backends, and the dashed lines denote the baseline backends. The merged GPU 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× speedup) for the CuPy merged group and 8.8 ms (31.82× speedup) for the PyTorch backend. But we do not observe much improvement for the merged CPU backend.

3.3.3 Mix CPU-GPU Backend Results

From Figure 3.1 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 3.3 shows that for GPU backends small buckets occupy approximately 90% of the total simulation time. The results for this approach are shown in Table 3.1 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×
NumPy (baseline)	CPU	246	1.00×
CuPy	GPU	6.7	36.7×
Torch_GPU	GPU	3.5	70.3×
Torch_CPU + Torch_GPU	Mixed	2.6	94.8×
NumPy + CuPy	Mixed	2.1	**117**×

Table 3.1: 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×, and for CuPy the improvement is 3×. 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× 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.

3.3.4 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 3.2: Time for full QAOA expectation value simulation using different Merged backends, as described in Section 3.2.3. 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 3.2, 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. 3.5, 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.

3.3.5 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 3.3, which compares FLOPs count for

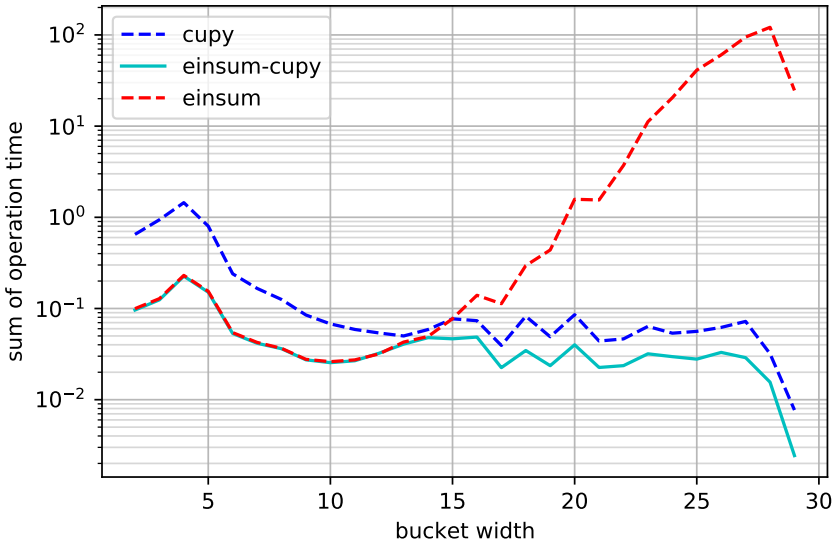


Figure 3.5: Breakdown of sum contraction time by bucket width for merged backends. CPU backends are better for buckets of width $\leq$ 15, and GPU backends are better for larger buckets. The hybrid backend's GPU backend spends outperforms the regular GPU backend for buckets of width $\geq$ 15.

similar-sized problems of different types. Figures 3.6 and 3.7 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 3.3: Summary of GPU and CPU FLOPs for different tasks at around 100 million operations. Matrix Multiplication and Tensor Contraction tasks are described in Section 3.3.5. "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 3.3.1 and is caused by overhead from small buckets. It is evident from Figure 3.3 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 µs for the multiplication, while the best GPU backend PyTorch spends 27 µs 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→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→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. 3.6, 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. 3.7. 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

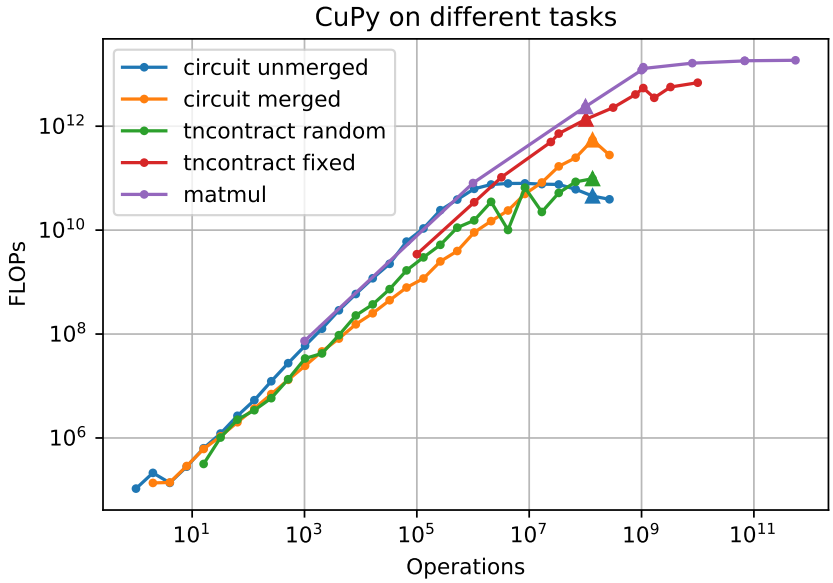


Figure 3.6: FLOPs vs. the number of operations for all tasks on the CuPy backend. “circuit unmerged" and “circuit merged" are results of expectation value of the full circuit simulation of QAOA MaxCut problem on a 3-regular graph of size 30 with depth $p = 4$. “tncontract random” tests on tensors of many indices where each index has a small size. “tncontract fixed" uses the contraction sequence “abcd,bcdf→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 $\sim$ 100 million operations, which is shown in Table 3.3.

break down the circuit simulation to tensor contraction operations, so it overlaps with the results from the "bucket unmerged" group in Fig. 3.6. 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).

3.4 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

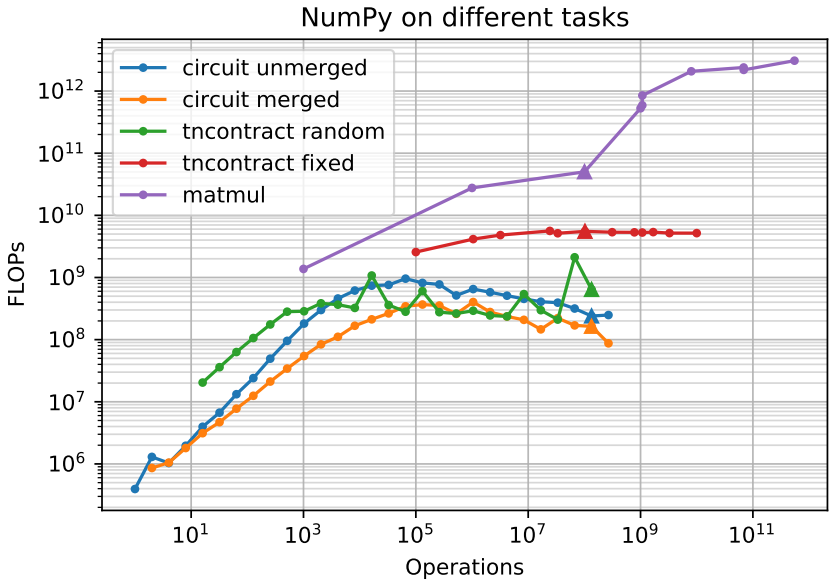


Figure 3.7: FLOPs vs. the number of operations for all tasks on NumPy backend. Same problem setting as Fig. 3.6. “tncontract random" outperforms “tncontract fixed" as the ops value increases. Merged backend does not have an advantage on CPU compared to the unmerged backend. We use the triangles to denote the data at $\sim$ 100 million operations, which is shown in Table 3.3.

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× improvement in time to solution.

We observe up to 300× 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 4

MAXCUT QAOA PERFORMANCE STUDY

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

4.1 Introduction

Quantum computing promises enormous computational powers that can far outperform any classical computational capabilities [**?**]. In particular, certain problems can be solved much faster compared with classical computing, as demonstrated experimentally by Google for the task of sampling from a quantum state [Arute et al., 2019]. Thus, an important milestone [Arute et al., 2019] in quantum technology, so-called 'quantum supremacy', was achieved as defined by Preskill [**?**].

The next milestone, 'quantum advantage', where quantum devices solve useful problems faster than classical hardware, is more elusive and has arguably not yet been demonstrated. However, a recent study suggests a possibility of achieving a quantum advantage in runtime over specialized state-of-the-art heuristic algorithms to solve the Maximum Independent Set problem using Rydberg atom arrays [**?**]. Common classical solutions to several potential applications for near-future quantum computing are heuristic and do not have performance bounds. Thus, proving the advantage of quantum computers is far more challenging [**???**]. Providing an estimate of how quantum advantage over these classical solvers can be achieved is important for the community and is the subject of this paper.

Most of the useful quantum algorithms require large fault-tolerant quantum computers, which remain far in the future. In the near future, however, we can expect to have noisy intermediate-scale quantum (NISQ) devices [**?**]. In this context variational quantum algorithms (VQAs) show the most promise [**?**] for the NISQ era, such as the variational quantum eigensolver (VQE) [**?**] and the Quantum Approximate Optimization Algo-

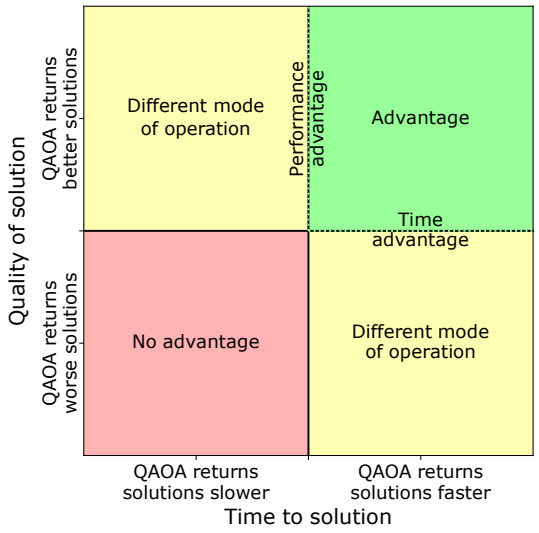


Figure 4.1: Locus of quantum advantage over classical algorithms. A particular classical algorithm may return some solution to some ensemble of problems in time T_C (horizontal axis) with some quality C_C (vertical axis). Similarly, a quantum algorithm may return a different solution sampled in time T_Q, which may be faster (right) or slower (left) than classical, with a better (top) or worse (bottom) quality than classical. If QAOA returns better solutions faster than the classical, then there is clear advantage (top right), and conversely no advantage for worse solutions slower than the classical (bottom left).

rithm (QAOA) [Farhi et al., 2014]. Researchers have shown remarkable interest in QAOA because it can be used to obtain approximate (i.e., valid but not optimal) solutions to a wide range of useful combinatorial optimization problems [**???**].

In opposition, powerful classical approximate and exact solvers have been developed to find good approximate solutions to combinatorial optimization problems. For example, a recent work by Guerreschi and Matsuura [**?**] compares the time to solution of QAOA vs. the classical combinatorial optimization suite AKMAXSAT. The classical optimizer takes exponential time with a small prefactor, which leads to the conclusion that QAOA needs hundreds of qubits to be faster than classical. This analysis requires the classical optimizer to find an exact solution, while QAOA yields only approximate solutions. However, modern classical heuristic algorithms are able to return an approximate solution on demand. Allowing for

worse-quality solutions makes these solvers extremely fast (on the order of milliseconds), suggesting that QAOA must also be fast to remain competitive. A valid comparison should consider both solution quality and time.

In this way, the locus of quantum advantage has two axes, as shown in Fig. 4.1: to reach advantage, a quantum algorithm must be both faster and return better solutions than a competing classical algorithm (green, top right). If the quantum version is slower and returns worse solutions (red, bottom left) there is clearly no advantage. However, two more regions are shown in the figure. If the QAOA returns better solutions more slowly than a classical algorithm (yellow, top left), then we can increase the running time for the classical version. It can try again and improve its solution with more time. This is a crucial mode to consider when assessing advantage: heuristic algorithms may always outperform quantum algorithms if quantum time to solution is slow. Alternatively, QAOA may return worse solutions faster (yellow, bottom right), which may be useful for time-sensitive applications. In the same way, we may stop the classical algorithm earlier, and the classical solutions will become worse.

One must keep in mind that the reason for using a quantum algorithm is the scaling of its time to solution with the problem size N. Therefore, a strong quantum advantage claim should demonstrate the superior performance of a quantum algorithm in the large-N limit.

This paper focuses on the MaxCut combinatorial optimization problem on 3-regular graphs for various problem size N. MaxCut is a popular benchmarking problem for QAOA because of its simplicity and straightforward implementation. We propose a fast fixed-angle approach to running QAOA that speeds up QAOA while preserving solution quality compared with slower conventional approaches. We evaluate the expectation value of noiseless QAOA solution quality using tensor network simulations on classical hardware. We then find the time required for classical solvers to match this expected QAOA solution quality. Surprisingly, we observe that even for the smallest possible time, the classical solution qual-

ity is above our QAOA solution quality for $p = 11$, our largest p with known performance. Therefore, we compensate for this difference in quality by using multishot QAOA and find the number of samples K required to match the classical solution quality. K allows us to characterize quantum device parameters, such as sampling frequency, required for the quantum algorithm to match the classical solution quality.

4.2 Results and discussion

This section will outline the results and comparison between classical optimizers and QAOA. This has two halves: Sec. 4.2.1 outlines the results of the quantum algorithm, and Sec. 4.2.2 outlines the results of the classical competition.

4.2.1 Expected QAOA solution quality

The first algorithm is the quantum approximate optimization algorithm (QAOA), which uses a particular ansatz to generate approximate solutions through measurement. We evaluate QAOA for two specific modes. The first is single shot fixed angle QAOA, where a single solution is generated. This has the the benefit of being very fast. The second generalization is multi-shot fixed angle QAOA, where many solutions are generated, and the best is kept. This has the benefit that the solution may be improved with increased run time.

In Section 4.3.3 we find that one can put limits on the QAOA MaxCut performance even when the exact structure of a 3-regular graph is unknown using fixed angles. We have shown that for large N the average cut fraction for QAOA solutions on 3-regular graphs converges to a fixed value f_{tree}. If memory limitations permit, we evaluate these values numerically using tensor network simulations. This gives us the average QAOA performance for any large N and $p \leq 11$. To further strengthen the study of QAOA performance estimations, we verify that for the small N, the performance is close to the same value f_{tree}. We are able to numerically verify that for $p \leq 4$ and small N the typical cut fraction is close to f_{tree}, as

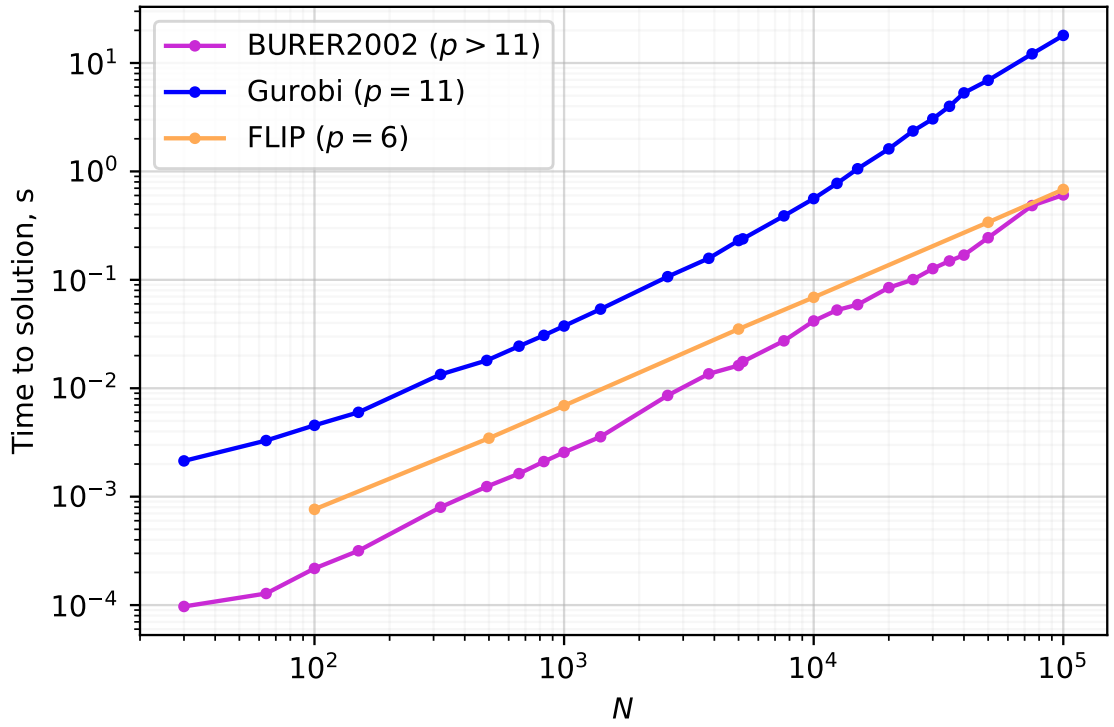


Figure 4.2: Time required for a single-shot QAOA to match classical MaxCut algorithms. The blue line shows time for comparing with the Gurobi solver and using $p = 11$; the yellow line shows comparison with the FLIP algorithm and $p = 6$. Each quantum device that runs MaxCut QAOA can be represented on this plot as a point, where the x-axis is the number of qubits and the y-axis is the time to solution. For any QAOA depth p, the quantum device should return at least one bitstring faster than the Y-value on this plot.

shown on Fig. 4.6.

Combining the large-N theoretical analysis and small-N heuristic evidence, we are able to predict the average performance of QAOA on 3-regular graphs for $p \leq 11$. We note that today's hardware can run QAOA up to $p \leq 4$ [?] and that for larger depths the hardware noise prevents achieving better QAOA performance. Therefore, the $p \leq 11$ constraint is not an important limitation for our analysis.

4.2.2 Classical solution quality and time to solution

The second ensemble of algorithms are classical heuristic or any-time algorithms. These algorithms have the property that they can be stopped mid-optimization and provide the best solution found so far. After a short time spent loading the instance, they find an initial 'zero-time' guess. Then, they explore the solution space and find incramentally better solutions until stopping with the best solution after a generally exponential amount of time. We experimentally evaluate the performance of the classical solvers Gurobi, MQLib using BURER2002 heuristic, and FLIP in Sec. 4.3.2. We observe that the zero-time performance, which is the quality of the fastest classical solution, is above the expected quality of QAOA $p = 11$, as shown in Fig. 4.3. The time to first solution scales almost linearly with size, as shown in Fig. 4.2. To compete with classical solvers, QAOA has to return better solutions faster.

4.2.3 Multi-shot QAOA

To improve the performance of QAOA, one can sample many bitstrings and then take the best one. This approach will work only if the dispersion of the cut fraction distribution is large, however. For example, if the dispersion is zero, measuring the ansatz state would return only bitstrings with a fixed cut value. By analyzing the correlations between the qubits in Section 4.3.3, we show that the distribution of the cut fraction is a Gaussian with the standard deviation on the order of $1/\sqrt{N}$. The expectation value of maximum of K samples is proportional to the standard deviation, as shown in Equation 4.7. This equation determines the performance of multishot QAOA. In the large N limit the standard deviation is small, and one might need to measure more samples in order to match the classical performance.

If we have the mean performance of a classical algorithm, we can estimate the number of samples K required for QAOA to match the classical performance. We denote the difference

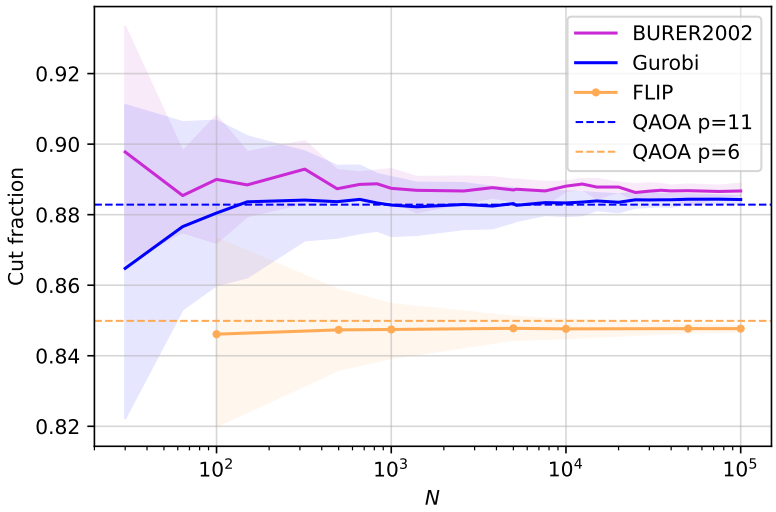


Figure 4.3: Zero-time performance for graphs of different size N. The Y-value is the cut fraction obtained by running corresponding algorithms for minimum possible time. This corresponds to the Y-value of the star marker in Fig. 4.4. Dashed lines show the expected QAOA performance for $p = 11$ (blue) and $p = 6$ (yellow). QAOA can outperform the FLIP algorithm at depth $p > 6$, while for Gurobi it needs $p > 11$. Note that in order to claim advantage, QAOA has to provide the zero-time solutions in faster time than FLIP or Gurobi does. These times are shown on Fig. 4.2.

between classical and quantum expected cut fraction as $\Delta_p(t)$, which is a function of the running time of the classical algorithm. Moreover, it also depends on p, since p determines QAOA expected performance. If $\Delta_p(t) < 0$, the performance of QAOA is better, and we need only a $K = 1$ sample. In order to provide an advantage, QAOA would have to measure this sample faster than the classical algorithm, as per Fig. 4.1. On the other hand, if $\Delta_p(t) > 0$, the classical expectation value is larger than the quantum one, and we have to perform multisample QAOA. We can find K by inverting Equation 4.7. In order to match the classical algorithm, a quantum device should be able to run these K samples in no longer than t. We can therefore get the threshold sampling frequency.

$$\nu_p(t) = \frac{K}{t} = \frac{1}{t} \exp\left(\frac{N}{2\gamma_p^2}\Delta_p(t)^2\right) \tag{4.1}$$

The scaling of $\Delta_p(t)$ with t is essential here since it determines at which point t we will have the smallest sampling frequency for advantage. We find that for BURER2002, the value of $\Delta(t)$ is the lowest for the smallest possible $t = t_0$, which is when a classical algorithm can produce its first solution. To provide the lower bound for QAOA we consider t_0 as the most favourable point, since classical solution improves much faster with time than a multi-shot QAOA solution. This point is discussed in more detail in the Supplementary Methods.

Time t_0 is shown on Fig. 4.2 for different classical algorithms. We note that in the figure the time scales polynomially with the number of nodes N. Figure 4.3 shows the mean cut fraction for the same classical algorithms, as well as the expectation value of QAOA at $p = 6, 11$. These two figures show that a simple linear-runtime FLIP algorithm is fast and gives a performance on par with $p = 6$ QAOA. In this case $\Delta_6(t_0) < 0$, and we need to sample only a single bitstring. To obtain the $p = 6$ sampling frequency for advantage over the FLIP algorithm, one has to invert the time from Fig. 4.2. If the quantum device is not capable of running $p = 6$ with little noise, the quantum computer will have to do multishot QAOA. Note that any classical prepossessing for QAOA will be at least linear in

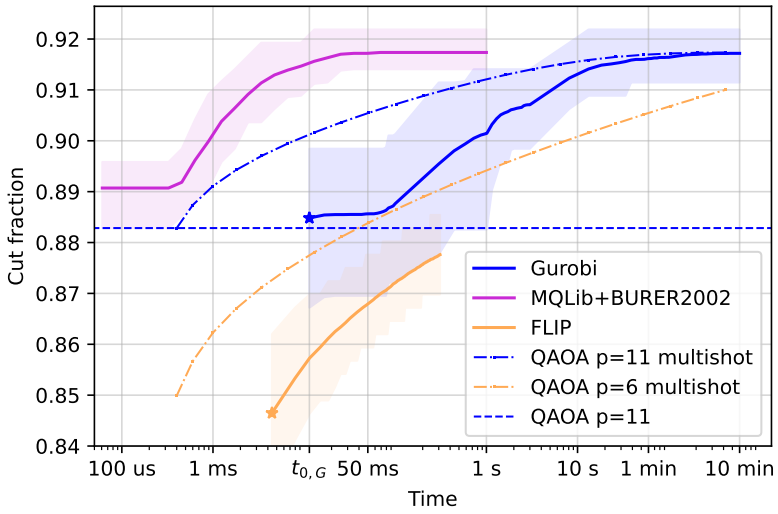


Figure 4.4: Evolution of cut fraction value in the process of running the classical algorithms solving 3-regular MaxCut with N=256. The shaded area shows 90-10 percentiles interval, and the solid line shows the mean cut fraction over 100 graphs. The dashed lines show the expectation value of single-shot QAOA for $p = 6, 11$, and the dash-dotted lines show the expected performance for multishot QAOA given a sampling rate of 5 kHz. Note that for this $N = 256$ the multi-shot QAOA with $p = 6$ can compete with Gurobi at 50 milliseconds. However, the slope of the multi-shot line will decrease for larger N, reducing the utility of the multi-shot QAOA.

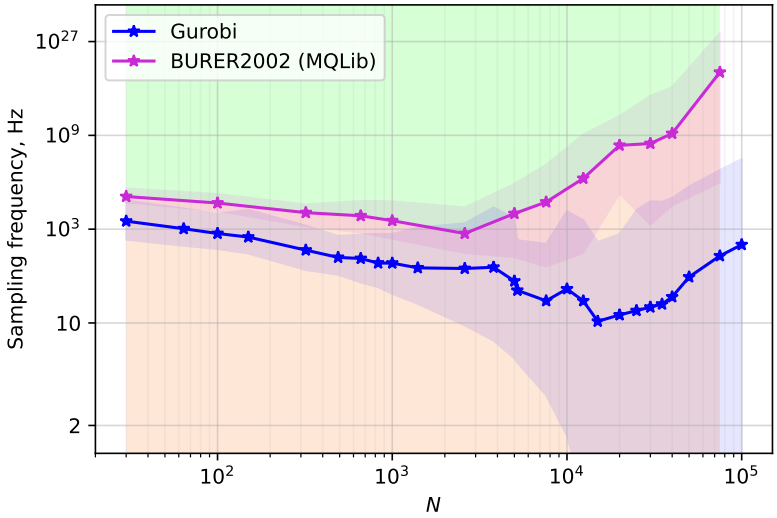


Figure 4.5: Sampling frequency required to achieve MaxCut advantage using QAOA $p = 11$. The shaded area around the solid lines corresponds to 90-10 percentiles over 100 seeds for Gurobi and 20 seeds for BURER2002. The background shading represents comparison of a quantum computer with BURER2002 solver corresponding to modes in Fig. 4.1. Each quantum device can be represented on this plot as a point, where the x-axis is the number of qubits, and the y-axis is the time to solution. Depending on the region where the point lands, there are different results of comparisons. QAOA becomes inefficient for large N, when sampling frequency starts to grow exponentially with N.

time since one must read the input and produce a quantum circuit. Therefore, for small $p < 6$ QAOA will not give significant advantage: for any fast QAOA device one needs a fast classical computer; one might just run the classical FLIP algorithm on it.

The Gurobi solver is able to achieve substantially better performance, and it slightly outperforms $p = 11$ QAOA. Moreover, the BURER2002 algorithm demonstrates even better solution quality than does Gurobi while being significantly faster. For both Gurobi and BURER2002, the $\Delta_{11}(t_0) > 0$, and we need to either perform multishot QAOA or increase p. Figure 4.5 shows the advantage sampling frequency $\nu_{11}(t_0)$ for the Gurobi and BURER2002 algorithms; note that the vertical axis is doubly exponential.

The sampling frequency is a result of two factors that work in opposite directions. On the one hand, the time to solution for a classical algorithm grows with N, and hence ν

drops. On the other hand, the standard deviation of distribution vanishes as $1/\sqrt{N}$, and therefore the number of samples K grows exponentially. There is an optimal size N for which the sampling frequency is minimal. This analysis shows that there is a possibility for advantage with multi-shot QAOA for moderate sizes of $N = 100..10\,000$, for which a sampling frequency of $\approx$ 10kHz is required. These frequencies are very sensitive to the difference in solution quality, and for $p \geq 12$ a different presentation is needed, if one quantum sample is expected to give better than classical solution quality. This is discussed in more detail in Supplementary Methods.

For large N, as expected, we see a rapid growth of sampling frequency, which indicates that QAOA does not scale for larger graph sizes, unless we go to higher depth $p > 11$. The color shading shows correspondence with Fig. 4.1. If the quantum device is able to run $p \geq 11$ and its sampling frequency and the number of qubits N corresponds to the green area, we have a quantum advantage. Otherwise, the quantum device belongs to the red area, and there is no advantage.

It is important to note the effect of classical parallelization on our results. Despite giving more resources to the classical side, parallel computing is unlikely to help it. To understand this, one has to think on how parallelization would change the performance profile as shown on Figure 4.4. The time to the first classical solution is usually bound from below by preparation tasks such as reading the graph, which are inherently serial. Thus, parallelization will not reduce t_0 and is in fact likely to increase it due to communication overhead. Instead, it will increase the slope of the solution quality curve, helping classical algorithms to compete in the convergence regime.

4.2.4 Discussion

As shown in Fig. 4.1, to achieve quantum advantage, QAOA must return better solutions faster than the competing classical algorithm. This puts stringent requirements on the speed

of QAOA, which previously may have gone unevaluated. If QAOA returns a solution more slowly, the competing classical algorithm may 'try again' to improve its solution, as is the case for anytime optimizers such as the Gurobi solver. The simplest way to improve the speed of QAOA is to reduce the number of queries to the quantum device, which we propose in our fixed-angle QAOA approach. This implementation forgoes the variational optimization step and uses solution concentration, reducing the number of samples to order 1 instead of order 100,000. Even with these improvements, however, the space of quantum advantage may be difficult to access.

Our work demonstrates that with a quantum computer of $\approx$ 100 qubits, QAOA can be competitive with classical MaxCut solvers if the time to solution is shorter than 100 μs and the depth of the QAOA circuit is $p \geq 6$. Note that this time to solution must include all parts of the computation, including state preparation, gate execution, and measurement. Depending on the parallelization of the architecture, there may be a quadratic time overhead. However, the required speed of the quantum device grows with N exponentially. Even if an experiment shows advantage for intermediate N and $p \leq 11$, the advantage will be lost on larger problems regardless of the quantum sampling rate. Thus, in order to be fully competitive with classical MaxCut solvers, quantum computers have to increase solution quality, for instance by using $p \geq 12$. Notably, $p = 12$ is required but not sufficient for achieving advantage: the end goal is obtaining a cut fraction better than ≥ 0.885 for large N, including overcoming other challenges of quantum devices such as noise.

These results lead us to conclude that for 3-regular graphs (perhaps all regular graphs), achieving quantum advantage on NISQ devices may be difficult. For example, the fidelity requirements to achieve quantum advantage are well above the characteristics of NISQ devices.

We note that improved versions of QAOA exist, where the initial state is replaced with a preoptimized state [**?**] or the mixer operator is adapted to improve performance [**??**].

One also can use information from classical solvers to generate a better ansatz state [?]. These algorithms have further potential to compete against classical MaxCut algorithms. Also, more general problems, such as weighted MaxCut, maximum independent set, and 3-SAT, may be necessary in order to find problem instances suitable for achieving quantum advantage.

When comparing with classical algorithms, one must record the complete time to solution from the circuit configuration to the measured state. This parameter may be used in the extension of the notion of quantum volume, which is customarily used for quantum device characterization. Our work shows that QAOA MaxCut does not scale with graph size for at least up to $p \leq 11$, thus putting quantum advantage for this problem away from the NISQ era.

4.3 Methods

Both classical solvers and QAOA return a bitstring as a solution to the MaxCut problem. To compare the algorithms, we must decide on a metric to use to measure the quality of the solution. A common metric for QAOA and many classical algorithms is the approximation ratio, which is defined as the ratio of cut value (as defined in Eq. (4.3)) of the solution divided by the optimal (i.e., maximum possible) cut value for the given graph. This metric is hard to evaluate heuristically for large N, since we do not know the optimal solution. We therefore use the cut fraction as the metric for solution quality, which is the cut value divided by the number of edges.

We analyze the algorithms on an ensemble of problem instances. Some instances may give advantage, while others may not. We therefore analyze ensemble advantage, which compares the average solution quality over the ensemble. The set of 3-regular graphs is extremely large for large graph size N, so for classical heuristic algorithms we evaluate the performance on a subset of graphs. We then look at the mean of the cut fraction over the ensemble, which

is the statistical approximation of the mean of the cut fraction over all 3-regular graphs.

4.3.1 QAOA Methodology

Usually QAOA is thought of as a hybrid algorithm, where a quantum-classical outer loop optimizes the angles γ, β through repeated query to the quantum device by a classical optimizer. Depending on the noise, this process may require hundreds or thousands of queries in order to find optimal angles, which slows the computation. To our knowledge, no comprehensive work exists on exactly how many queries may be required to find such angles. It has been numerically observed [**??**], however, that for small graph size $N = 12$ and $p = 4$, classical noise-free optimizers may find good angles in approximately 100 steps, which can be larger for higher N and p. Each step may need order 10^3 bitstring queries to average out shot noise and find expectation values for an optimizer, and thus seeking global angles may require approximately 100 000 queries to the simulator. The angles are then used for preparing an ansatz state, which is in turn measured (potentially multiple times) to obtain a solution. Assuming a sampling rate of 1 kHz, this approach implies a QAOA solution of approximately 100 seconds.

Recent results, however, suggest that angles may be precomputed on a classical device [**?**] or transferred from other similar graphs [**?**]. Further research analytically finds optimal angles for $p \leq 20$ and $d \to \infty$ for all large-girth d-regular graphs, but does not give angles for finite d [**?**]. Going a step further, a recent work finds that evaluating regular graphs at particular fixed angles has good performance on all problem instances [**?**]. These precomputed or fixed angles allow the outer loop to be bypassed, finding close to optimal results in a single shot. In this way, a 1000 Hz QAOA solution can be found in milliseconds, a speedup of several ordesr of magnitude.

For this reason we study the prospect for quantum advantage in the context of fixed-angle QAOA. For d-regular graphs, there exist particular fixed angles with universally good

performance [?]. Additionally, as will be shown in Section 4.3.5, one can reasonably expect that sampling a single bitstring from the fixed-angle QAOA will yield a solution with a cut fraction close to the expectation value.

The crucial property of the fixed-angle single-shot approach is that it is guaranteed to work for any graph size N. On the other hand, angle optimisation could be less productive for large N, and the multiple-shot (measuring the QAOA ansatz multiple times) approach is less productive for large N, as shown in Section 4.3.6. Moreover, the quality of the solution scales with depth as $\sqrt{p}$ [?], which is faster than with the number of samples $\sqrt{\log K}$, instructing us to resort to multishot QAOA only if larger p is unreachable. Thus, the fixed-angle single-shot QAOA can robustly speed up finding a good approximate solution from the order of seconds to milliseconds, a necessity for advantage over state-of-the-art anytime heuristic classical solvers, which can get good or exact solutions in approximately milliseconds. Crucially, single-shot QAOA quality of solution can be maintained for all sizes N at fixed depth p, which can mean constant time scaling, for particularly capable quantum devices.

To simulate the expectation value of the cost function for QAOA, we employ a classical quantum circuit simulation algorithm QTensor [Lykov et al., 2020b; ?; ?]. This algorithm is based on tensor network contraction and is described in more detail in Supplementary Methods. Using this approach, one can simulate expectation values on a classical computer even for circuits with millions of qubits.

4.3.2 Classical Solvers

Two main types of classical MaxCut algorithms exist: approximate algorithms and heuristic solvers. Approximate algorithms guarantee a certain quality of solution for any problem instance. Such algorithms [??] also provide polynomial-time scaling. Heuristic solvers [??] are usually based on branch-and-bound methods [?] that use branch pruning and heuristic rules for variable and value ordering. These heuristics are usually designed to run well on

graphs that are common in practical use cases. Heuristic solvers typically return better solutions than do approximate solvers, but they provide no guarantee on the quality of the solution.

The comparison of QAOA with classical solvers thus requires making choices of measures that depend on the context of comparison. From a theory point of view, guaranteed performance is more important; in contrast, from an applied point of view, heuristic performance is the measure of choice. A previous work [?] demonstrates that QAOA provides better performance guarantees than does the Goemans–Williamson algorithm [?]. In this paper we compare against heuristic algorithms since such a comparison is more relevant for real-world problems. On the other hand, the performance of classical solvers reported in this paper can depend on a particular problem instance.

We evaluate two classical algorithms using a single node of Argonne's Skylake testbed; the processor used is an Intel Xeon Platinum 8180M CPU @ 2.50 GHz with 768 GB of RAM.

The first algorithm we study is the Gurobi solver [?], which is a combination of many heuristic algorithms. We evaluate Gurobi with an improved configuration based on communication with Gurobi support [1]. We use `Symmetry=0` and `PreQLinearize=2` in our improved configuration. As further tweaks and hardware resources may increase the speed, the results here serve as a characteristic lower bound on Gurobi performance rather than a true guarantee. We run Gurobi on 100 random-regular graphs for each size N and allow each optimization to run for 30 minutes. During the algorithm runtime we collect information about the process, in particular the quality of the best-known solution. In this way we obtain a performance profile of the algorithm that shows the relation between the solution quality and the running time. An example of such a performance profile for $N = 256$ is shown in Fig. 4.4. Gurobi was configured to use only a single CPU, to avoid interference in runtime between different Gurobi optimization runs for different problem instances. In order to speed

1. `https://support.gurobi.com/hc/en-us/community/posts/4403570181137-Worse-performance-for-smaller-problem`

up collection of the statistics, 55 problem instances were executed in parallel.

The second algorithm is MQLib [**?**], which is implemented in C++ and uses a variety of different heuristics for solving MaxCut and QUBO problems. We chose the BURER2002 heuristic since in our experiments it performs the best for MaxCut on random regular graphs. Despite using a single thread, this algorithm is much faster than Gurobi; thus we run it for 1 second. In the same way as with Gurobi, we collect the performance profile of this algorithm.

While QAOA and Gurobi can be used as general-purpose combinatorial optimization algorithms, this algorithm is designed to solve MaxCut problems only, and the heuristic was picked that demonstrated the best performance on the graphs we considered. In this way we use Gurobi as a worst-case classical solver, which is capable of solving the same problems as QAOA can. Moreover, Gurobi is a well-established commercial tool that is widely used in industry. Note, however, that we use QAOA fixed angles that are optimized specifically for 3-regular graphs, and one can argue that our fixed-angle QAOA is an algorithm designed for 3-regular MaxCut. For this reason we also consider the best-case MQLib+BURER2002 classical algorithm, which is designed for MaxCut, and we choose the heuristic that performs best on 3-regular graphs.

4.3.3 QAOA performance

Two aspects are involved in comparing the performance of algorithms, as outlined in Fig. 4.1: time to solution and quality of solution. In this section we evaluate the performance of single-shot fixed-angle QAOA. As discussed in the introduction, the time to solution is a crucial part and for QAOA is dependent on the initialization time and the number of rounds of sampling. Single-shot fixed-angle QAOA involves only a single round of sampling, and so the time to solution can be extremely fast, with initialization time potentially becoming the limiting factor. This initialization time is bound by the speed of classical computers, which perform calibration and device control. Naturally, if one is able to achieve greater

initialization speed by using better classical computers, the same computers can be used to improve the speed of solving MaxCut classically. Therefore, it is also important to consider the time scaling of both quantum initialization and classical runtime.

The quality of the QAOA solution is the other part of performance. The discussion below evaluates this feature by using subgraph decompositions and QAOA typicality, including a justification of single shot sampling.

QAOA is a variational ansatz algorithm structured to provide solutions to combinatorial optimization problems. The ansatz is constructed as p repeated applications of an objective $\hat{C}$ and mixing $\hat{B}$ unitary:

$$|\gamma, \beta\rangle = e^{-i\beta_p \hat{B}} e^{-i\gamma_p \hat{C}} (\cdots) e^{-i\beta_1 \hat{B}} e^{-i\gamma_1 \hat{C}} |+\rangle, \tag{4.2}$$

where $\hat{B}$ is a sum over Pauli X operators $\hat{B} = \sum_i^N \hat{\sigma}_x^i$. A common problem instance is MaxCut, which strives to bipartition the vertices of some graph $\mathcal{G}$ such that the maximum number of edges have vertices in opposite sets. Each such edge is considered to be cut by the bipartition. This may be captured in the objective function

$$\hat{C} = \frac{1}{2} \sum_{\langle ij \rangle \in \mathcal{G}} (1 - \hat{\sigma}_z^i \hat{\sigma}_z^j), \tag{4.3}$$

whose eigenstates are bipartitions in the Z basis, with eigenvalues that count the number of cut edges. To get the solution to the optimization problem, one prepares the ansatz state $|\vec{\gamma}, \vec{\beta}\rangle$ on a quantum device and then measures the state. The measured bitstring is the solution output from the algorithm.

While QAOA is guaranteed to converge to the exact solution in the $p \to \infty$ limit in accordance with the adiabatic theorem [Farhi et al., 2014; **?**], today's hardware is limited to low depths $p \sim 1$ to 5, because of the noise and decoherence effects inherent to the NISQ era.

A useful tool for analyzing the performance of QAOA is the fact that QAOA is local [Farhi et al., 2014; **?**]: the entanglement between any two qubits at a distance of $\geq 2p$ steps from each other is strictly zero. For a similar reason, the expectation value of a particular edge $\langle ij \rangle$

$$f_{\langle ij \rangle} = \frac{1}{2}\langle \vec{\gamma}, \vec{\beta} | 1 - \hat{\sigma}_z^i \hat{\sigma}_z^j | \vec{\gamma}, \vec{\beta} \rangle \tag{4.4}$$

depends only on the structure of the graph within p steps of edge $\langle ij \rangle$. Regular graphs have a finite number of such local structures (also known as subgraphs) [**?**], and so the expectation value of the objective function can be rewritten as a sum over subgraphs

$$\langle \hat{C} \rangle = \sum_{\text{subgraphs } \lambda} M_\lambda(\mathcal{G}) f_\lambda. \tag{4.5}$$

Here, λ indexes the different possible subgraphs of depth p for a d regular graph, $M_\lambda(\mathcal{G})$ counts the number of each subgraph λ for a particular graph $\mathcal{G}$, and f_λ is the expectation value of the subgraph (e.g., Eq. (4.4)). For example, if there are no cycles $\leq 2p + 1$, only one subgraph (the tree subgraph) contributes to the sum.

With this tool we may ask and answer the following question: What is the typical performance of single-shot fixed-angle QAOA, evaluated over some ensemble of graphs? Here, performance is characterized as the typical (average) fraction of edges cut by a bitstring solution returned by a single sample of fixed-angle QAOA, averaged over all graphs in the particular ensemble.

For our study we choose the ensemble of 3-regular graphs on N vertices. Different ensembles, characterized by different connectivity d and size N, may have different QAOA performance [**??**].

Using the structure of the random regular graphs, we can put bounds on the cut fraction by bounding the number of different subgraphs and evaluating the number of large cycles.

These bounds become tighter for $N \longrightarrow \infty$ and fixed p since the majority of subgraphs become trees and 1-cycle graphs. We describe this analysis in detail in Supplemental methods, which shows that the QAOA cut fraction will equal the expectation value on the tree subgraph, which may be used as a 'with high probability' (WHP) proxy of performance. Furthermore, using a subgraph counting argument, we may count the number of tree subgraphs to find an upper and lower WHP bound on the cut fraction for smaller graphs. These bounds are shown as the boundaries of the red and green regions in Fig. 4.6.

4.3.4 QAOA Ensemble Estimates

A more straightforward but less rigorous characterization of QAOA performance is simply to evaluate fixed-angle QAOA on a subsample of graphs in the ensemble. The results of such an analysis require an assumption not on the particular combinatorial graph structure of ensembles but instead on the typicality of expectation values on subgraphs. This is an assumption on the structure of QAOA and allows an extension of typical cut fractions from the large N limit where most subgraphs are trees to a small N limit where typically a very small fraction of subgraphs are trees.

Figure 4.6 plots the ensemble-averaged cut fraction for $p = 2$ and various sizes of graphs. For $N \leq 16$, the ensemble includes every 3-regular graph (4,681 in total). For each size of $N > 16$, we evaluate fixed-angle QAOA on 1,000 3-regular graphs drawn at random from the ensemble of all 3-regular graphs for each size $N \in (16, 256]$. Note that because the evaluation is done at fixed angles, it may be done with minimal quantum calculation by a decomposition into subgraphs, then looking up the subgraph expectation value f_λ from [**?**]. This approach is also described in more detail in [**?**]. In this way, expectation values can be computed as fast as an isomorphism check.

From Fig. 4.6 we observe that the median cut fraction across the ensemble appears to concentrate around that of the tree subgraph value, even for ensembles where the typical

graph is too small to include many tree subgraphs. Additionally, the variance (dark fill) reduces as N increases, consistent with the fact that for larger N there are fewer kinds of subgraphs with non-negligible frequency. Furthermore, the absolute range (light fill), which plots the largest and smallest expectation value across the ensemble, is consistently small. While the data for the absolute range exists here only for $N \leq 16$ because of complete sampling of the ensemble, 0ne can reasonably expect that these absolute ranges extend for all N, suggesting that the absolute best performance of $p = 2$ QAOA on 3-regular graphs is around ≈ 0.8.

We numerically observe across a range of p (not shown) that these behaviors persist: the typical cut fraction is approximately equal to that of the tree subgraph value $f_{\text{p-tree}}$ even in the limit where no subgraph is a tree. This suggests that the typical subgraph expectation value $f_\lambda \approx f_{\text{p-tree}}$, and only an atypical number of subgraphs have expectation values that diverge from the tree value. With this observation, we may use the value $f_{\text{p-tree}}$ as a proxy for the average cut fraction of fixed-angle QAOA.

These analyses yield four different regimes for advantage vs. classical algorithms, shown in Fig. 4.6. If a classical algorithm yields small cut fractions for large graphs (green, bottom right), then there is advantage in a strong sense. Based only on graph combinatorics, with high probability most of the edges participate in few cycles, and thus the cut fraction is almost guaranteed to be around the tree value, larger than the classical solver. Conversely, if the classical algorithm yields large cut fractions for large graphs (red, top right), there is no advantage in the strong sense: QAOA will yield, for example, only ~ 0.756 for $p = 2$ because most edges see no global structure. This analysis emphasizes that of [**?**], which suggests that QAOA needs to 'see' the whole graph in order to get reasonable performance.

Two additional performance regimes for small graphs exist, where QAOA can reasonably see the whole graph. If a classical algorithm yields small cut fractions for small graphs

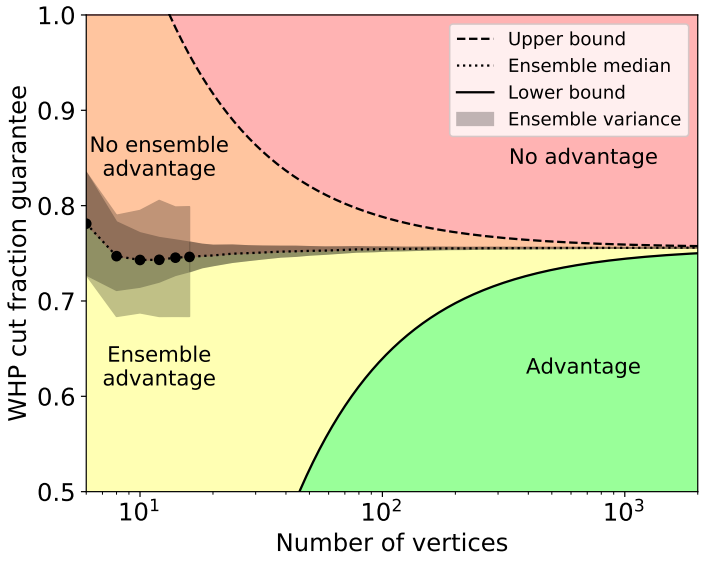


Figure 4.6: $p = 2$ QAOA cut fraction guarantees under different assumptions. Dashed and solid lines plot with high probability the lower and upper bounds on cut fractions, respectively, assuming only graph theoretic typicality on the number of subgraphs. Dotted plots are the ensemble median over an ensemble of 3-regular graphs; for $N \leq 16$ (dots); this includes all graphs, while for $N > 16$ this is an ensemble of 1,000 graphs for each size. We used 32 sizes between 16 and 256. Dark black fill plots the variance in the cut fraction over the ensemble, and light black fill plots the extremal values over the ensemble. The median serves as a proxy of performance assuming QAOA typicality. Given a particular cut from a classical solver, there may be different regions of advantage, shown by the four colors and discussed in the text.

(yellow, bottom left), then there is advantage in a weak sense, which we call the 'ensemble advantage'. Based on QAOA concentration, there is at least a 50% chance that the QAOA result on a particular graph will yield a better cut fraction than will the classical algorithm; assuming that the variance in cut fraction is small, this is a 'with high probability' statement. Conversely, if the classical algorithm yields large cut fractions for small graphs (orange, top left), there is no advantage in a weak sense. Assuming QAOA concentration, the cut fraction will be smaller than the classical value, and for some classical cut fraction there are no graphs with advantage (e.g., > 0.8 for $p = 2$).

Based on these numerical results, we may use the expectation value of the tree subgraph $f_{\text{p-tree}}$ as a high-probability proxy for typical fixed-angle QAOA performance on regular graphs. For large N, this result is validated by graph-theoretic bounds counting the typical number of tree subgraphs in a typical graph. For small N, this result is validated by fixed-angle QAOA evaluation on a large ensemble of graphs.

4.3.5 Single-shot QAOA Sampling

A crucial element of single-shot fixed-angle QAOA is that the typical bitstring measured from the QAOA ansatz has a cut value similar to the average. This fact was originally observed by Farhi et al. in the original QAOA proposal [Farhi et al., 2014]: because of the strict locality of QAOA, vertices a distance more than $> 2p$ steps from each other have a ZZ correlation of strictly zero. Thus, for large graphs with a width $> 2p$, by the central limit theorem the cut fraction concentrates to a Gaussian with a standard deviation of order $\frac{1}{\sqrt{N}}$ around the mean. As the variance grows sublinearly in N, the values concentrate at the mean, and thus with high probability measuring a single sample of QAOA will yield a solution with a cut value close to the average.

However, this result is limited in scope for larger depths p, because it imposes no requirements on the strength of correlations for vertices within distance $\leq 2p$. Therefore, here

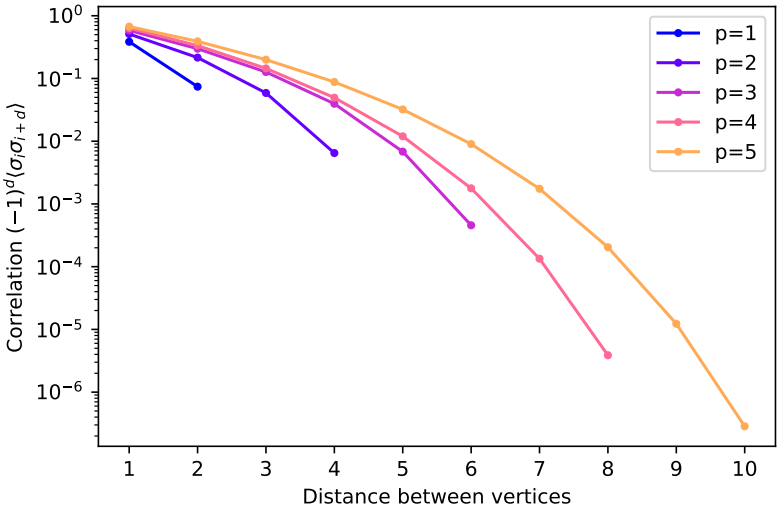


Figure 4.7: Long-range antiferromagnetic correlation coefficient on the 3-regular Bethe lattice, which is a proxy for an $N \to \infty$ typical 3-regular graph. Horizontal indexes the distance between two vertices. QAOA is strictly local, which implies that no correlations exist between vertices a distance $> 2p$ away. As shown here, however, these correlations are exponentially decaying with distance. This suggests that even if the QAOA 'sees the whole graph', one can use the central limit theorem to argue that the distribution of QAOA performance is Gaussian with the standard deviation of $\propto 1/\sqrt{N}$

we strengthen the argument of Farhi et al. and show that these concentration results may persist even in the limit of large depth p and small graphs N. We formalize these results by evaluating the ZZ correlations of vertices within $2p$ steps, as shown in Fig. 4.7. Expectation values are computed on the 3-regular Bethe lattice, which has no cycles and thus can be considered the $N \to \infty$ typical limit. Instead of computing the nearest-neighbor correlation function, the x-axis computes the correlation function between vertices a certain distance apart. For distance 1, the correlations are that of the objective function $f_{\text{p-tree}}$. Additionally, for distance $> 2p$, the correlations are strictly zero in accordance with the strict locality of QAOA. For distance $\leq 2p$, the correlations are exponentially decaying with distance. Consequently, even for vertices within the lightcone of QAOA, the correlation is small; and so by the central limit theorem the distribution will be Gaussian. This result holds because the probability of having a cycle of fixed size converges to 0 as $N \to \infty$. In other words, we know that with $N \to \infty$ we will have a Gaussian cost distribution with standard deviation $\propto \frac{1}{\sqrt{N}}$.

When considering small N graphs, ones that have cycles of length $\leq 2p+1$, we can reasonably extend the argument of Section 4.3.4 on typicality of subgraph expectation values. Under this typicality argument, the correlations between close vertices is still exponentially decaying with distance, even though the subgraph may not be a tree and there are multiple short paths between vertices. Thus, for all graphs, by the central limit theorem the distribution of solutions concentrates as a Gaussian with a standard deviation of order $\frac{1}{\sqrt{N}}$ around the mean. By extension, with probability $\sim 50\%$, any single measurement will yield a bitstring with a cut value greater than the average. These results of cut distributions have been found heuristically in [**?**].

The results are a full characterization of the fixed-angle single-shot QAOA on 3-regular graphs. Given a typical graph sampled from the ensemble of all regular graphs, the typical

cut fraction from level p QAOA will be about that of the expectation value of the p-tree $f_{\text{p-tree}}$. The distribution of bitstrings is concentrated as a Gaussian of subextensive variance around the mean, indicating that one can find a solution with quality greater than the mean with order 1 samples. Furthermore, because the fixed angles bypass the hybrid optimization loop, the number of queries to the quantum simulator is reduced by orders of magnitude, yielding solutions on potentially millisecond timescales.

4.3.6 Mult-shot QAOA Sampling

In the preceding section we demonstrated that the standard deviation of MaxCut cost distribution falls as $1/\sqrt{N}$, which deems impractical the usage of multiple shots for large graphs. However, it is worth verifying more precisely its effect on the QAOA performance. The multiple-shot QAOA involves measuring the bitstring from the same ansatz state and then picking the bitstring with the best cost. To evaluate such an approach, we need to find the expectation value for the best bitstring over K measurements.

As shown above, the distribution of cost for each measured bitstring is Gaussian, $p(x) = G(\frac{x-\mu_p}{\sigma_N})$. We define a new random variable ξ which is the cost of the best of K bitstrings. The cumulative distribution function (CDF) of the best of K bitstrings is $F_K(\xi)$, and $F_1(\xi)$ is the CDF of a normal distribution. The probability density for ξ is

$$p_K(\xi) = \frac{d}{d\xi}F_K(\xi) = \frac{d}{d\xi}F_1^K(\xi) = KF_1^{K-1}(\xi)p(\xi), \tag{4.6}$$

where $F_1(\xi) = \int_{-\infty}^{\xi} p(x)dx$ and F_1^K is the ordinary exponentiation. The expectation value for ξ can be found by $E_K = \int_{-\infty}^{\infty} dx\, xp_K(x)$. While the analytical expression for the integral can be extensive, a good upper bound exists for it: $E_K \leq \sigma\sqrt{2\log K} + \mu$.

Combined with the $1/\sqrt{N}$ scaling of the standard deviation, we can obtain a bound on improvement in cut fraction from sampling K times:

$$\Delta = \gamma_p \sqrt{\frac{2}{N} \log K}, \tag{4.7}$$

where γ_p is a scaling parameter. The value Δ is the difference of solution quality for multishot and single-shot QAOA. Essentially it determines the utility of using multishot QAOA. We can determine the scaling constant γ_p by classically simulating the distribution of the cost value in the ansatz state. We perform these simulations using QTensor for an ensemble of graphs with $N \leq 26$ to obtain $\gamma_6 = 0.1926$ and $\gamma_{11} = 0.1284$.

It is also worthwhile to verify the $1/\sqrt{N}$ scaling, by calculating γ_p for various N. We can do so for smaller $p = 3$ and graph sizes $N \leq 256$. We calculate the standard deviation by $\Delta C = \sqrt{\langle C^2 \rangle - \langle C \rangle^2}$ and evaluate the $\langle C^2 \rangle$ using QTensor. This evaluation gives large light cones for large p; the largest that we were able to simulate is $p = 3$. From the deviations ΔC we can obtain values for γ_3. We find that for all N the values stay within 5% of the average over all N. This shows that they do not depend on N, which in turn signifies that the $1/\sqrt{N}$ scaling is a valid model. The results of numerical simulation of the standard deviation are discussed in more detail in the Supplementary Methods.

To compare multishot QAOA with classical solvers, we plot the expected performance of multishot QAOA in Fig. 4.4 as dash-dotted lines. We assume that a quantum device is able to sample at the 5kHz rate. Today's hardware is able to run up to $p = 5$ and achieve the 5 kHz sampling rate [**?**]. Notably, the sampling frequency of modern quantum computers is bound not by gate duration, but by qubit preparation and measurement.

For small N, reasonable improvement can be achieved by using a few samples. For example, for $N = 256$ with $p = 6$ and just $K = 200$ shots, QAOA can perform as well as single-shot $p = 11$ QAOA. For large N, however, too many samples are required to obtain substantial improvement for multishot QAOA to be practical.

4.3.7 Classical performance

To compare the QAOA algorithm with its classical counterparts, we choose the state-of-the art algorithms that solve the similar spectrum of problems as QAOA, and we evaluate the time to solution and solution quality. Here, we compare two algorithms: Gurobi and MQLib+BURER2002. Both are anytime heuristic algorithms that can provide an approximate solution at arbitrary time. For these algorithms we collect the 'performance profiles'—the dependence of solution quality on time spent finding the solution. We also evaluate performance of a simple MaxCut algorithm FLIP. This algorithm has a proven linear time scaling with input size. It returns a single solution after a short time. To obtain a better FLIP solution, one may run the algorithm several times and take the best solution, similarly to the multishot QAOA.

Both algorithms have to read the input and perform some initialization step to output any solution. This initialization step determines the minimum time required for getting the initial solution—a 'first guess' of the algorithm. This time is the leftmost point of the performance profile marked with a star in Fig. 4.4. We call this time t_0 and the corresponding solution quality 'zero-time performance'.

We observe two important results.

1. Zero-time performance is constant with N and is comparable to that of $p = 11$ QAOA, as shown in Fig. 4.3, where solid lines show classical performance and dashed lines show QAOA performance.

2. t_0 scales as a low-degree polynomial in N, as shown in Fig. 4.2. The y-axis is t_0 for several classical algorithms.

Since the zero-time performance is slightly above the expected QAOA performance at $p = 11$, we focus on analyzing this zero-time regime. In the following subsections we discuss the performance of the classical algorithms and then proceed to the comparison with QAOA.

4.3.8 Performance of Gurobi Solver

In our classical experiments, as mentioned in Section 4.3.2, we collect the solution quality with respect to time for multiple N and graph instances. An example averaged solution quality evolution is shown in Fig. 4.4 for an ensemble of 256 vertex 3-regular graphs. Between times 0 and $t_{0,G}$, the Gurobi algorithm goes through some initialization and quickly finds some naive approximate solution. Next, the first incumbent solution is generated, which will be improved in further runtime. Notably, for the first 50 milliseconds, no significant improvement to solution quality is found. After that, the solution quality starts to rise and slowly converge to the optimal value of ~ 0.92.

It is important to appreciate that Gurobi is more than just a heuristic solver: in addition to the incumbent solution, it always returns an upper bound on the optimal cost. When the upper bound and the cost for the incumbent solution match, the optimal solution is found. It is likely that Gurobi spends a large portion of its runtime on proving the optimality by lowering the upper bound. This emphasizes that we use Gurobi as a worst-case classical solver.

Notably, the x-axis of Fig. 4.4 is logarithmic: the lower and upper bounds eventually converge after exponential time with a small prefactor, ending the program and yielding the exact solution. Additionally, the typical upper and lower bounds of the cut fraction of the best solution are close to 1. Even after approximately 10 seconds for a 256-vertex graph, the algorithm returns cut fractions with very high quality ~ 0.92, far better than intermediate-depth QAOA.

The zero-time performance of Gurobi for $N = 256$ corresponds to the Y-value of the star marker on Fig. 4.4. We plot this value for various N in Fig. 4.3. As shown in the figure, zero-time performance goes up and reaches a constant value of ~ 0.882 at $N \sim 100$. Even for large graphs of $N = 10^5$, the solution quality stays at the same level.

Such solution quality is returned after time $t_{0,G}$, which we plot in Fig. 4.2 for various N.

For example, for a 1000-node graph it will take ~ 40 milliseconds to return the first solution. Evidently, this time scales as a low-degree polynomial with N. This shows that Gurobi can consistently return solutions of quality ~ 0.882 in polynomial time.

4.3.9 Performance of MQLib+BURER2002 and FLIP Algorithms

The MQLib algorithm with the BURER2002 heuristic shows significantly better performance, which is expected since it is specific to MaxCut. As shown in Fig. 4.4 for $N = 256$ and in Fig. 4.2 for various N, the speed of this algorithm is much better compared with Gurobi's. Moreover, t_0 for MQLib also scales as a low-degree polynomial, and for 1,000 nodes MQLib can return a solution in 2 milliseconds. The zero-time performance shows the same constant behavior, and the value of the constant is slightly higher than that of Gurobi, as shown in Fig. 4.3.

While for Gurobi and MQLib we find the time scaling heuristically, the FLIP algorithm is known to have linear time scaling. With our implementation in Python, it shows speed comparable to that of MQLib and solution quality comparable to QAOA $p = 6$. We use this algorithm as a demonstration that a linear-time algorithm can give constant performance for large N, averaged over multiple graph instances.

4.4 Acknowledgements

This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions and/or findings expressed are those of the author and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. Y.A.'s and D.L.'s work at Argonne National Laboratory was supported by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357. The work at UWM was also supported by the U.S. Department of Energy, Office of Science, National Quantum Information Science Research Centers.

4.5 Data availability

The code, figures and datasets generated during the current study are available in a public repository `https://github.com/danlkv/quantum-classical-time-maxcut`. See the `README.md` file for the details on the contents of the repository.

CHAPTER 5

DISTRIBUTED STATEVECTOR SIMULATION

This chapter is adapted from Lykov, Shaydulin, Sun, Alexeev, and Pistoia [2023].

Until high-fidelity quantum computers with a large number of qubits become widely available, classical simulation remains a vital tool for algorithm design, tuning, and validation. We present a simulator for the Quantum Approximate Optimization Algorithm (QAOA). Our simulator is designed with the goal of reducing the computational cost of QAOA parameter optimization and supports both CPU and GPU execution. Our central observation is that the computational cost of both simulating the QAOA state and computing the QAOA objective to be optimized can be reduced by precomputing the diagonal Hamiltonian encoding the problem. We reduce the time for a typical QAOA parameter optimization by eleven times for $n = 26$ qubits compared to a state-of-the-art GPU quantum circuit simulator based on cuQuantum. Our simulator is available on GitHub: `https://github.com/jpmorganchase/QOKit`

5.1 Introduction

Quantum computers offer the prospect of accelerating the solution of a wide range of computational problems [**?**]. At the same time, only a small number of quantum algorithmic primitives with provable speedup have been identified, motivating the development of heuristics. Due to the limited availability and imperfections of near-term quantum computers, the design and validation of heuristic quantum algorithms have been largely performed in classical simulation. Additionally, classical simulators are commonly used to validate the results obtained on small-scale near-term devices. As a consequence, fast, high-performance simulators are a crucial tool for algorithm development.

Quantum Approximate Optimization Algorithm (QAOA) [Farhi et al., 2014; **?**] is one

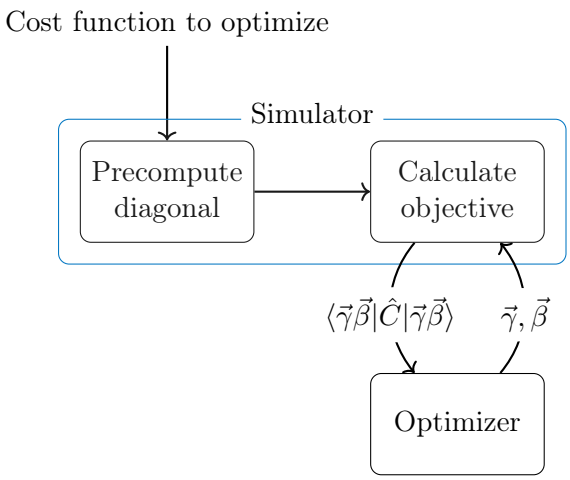


Figure 5.1: Overview of the simulator. Precomputing and storing the diagonal cost operator reduces the cost of both simulating the phase operator in QAOA as well as evaluating the QAOA objective.

of the most promising quantum algorithms for combinatorial optimization. QAOA approximately solves optimization problems by preparing a parameterized quantum state such that upon measuring it, high quality solutions are obtained with high probability.

Due to the difficulty of theoretical analysis, QAOA performance is commonly analyzed numerically. Recently, Boulebnane and Montanaro demonstrated numerically that QAOA scales better than state-of-the-art classical solvers for random 8-SAT [**?**]. The demonstrated potential of QAOA as an algorithmic component that enables quantum speedups motivates the development of tools for its numerical study. Since QAOA performance increases with circuit depth p, it is particularly interesting to simulate high-depth QAOA. For example, Ref. [**?**] only observes a quantum speedup with QAOA for $p \gtrsim 14$ and Ref. [**?**] demonstrates that $p \geq 12$ is needed for QAOA to be competitive with classical solvers for the MaxCut problem on 3-regular graphs.

We implement a fast state-vector simulator for the study of QAOA. Our simulator is optimized for simulating QAOA with high depth as well as repeated evaluation of QAOA objective, which is required for tuning the QAOA parameters. To accelerate the simulation, we first precompute the values of the function to be optimized (see Fig. 5.1). The result of

precomputation is reused during the parameter optimization. The precomputation algorithm is easy to parallelize, making it amenable to GPU acceleration. The precomputation requires storing an exponentially-sized vector, increasing the memory footprint of the simulation by only 12.5%. Our technique is general, and we implement transverse-field and Hamming-weight-preserving XY mixers. We achieve orders of magnitude speedups over state-of-the-art state-vector and tensor-network simulators and demonstrate scalability to 1,024 GPUs [**?**]. We implement the developed simulator in `QOKit` framework, which also provides optimized parameters and additional tooling for a set of commonly studied problems

We use the developed simulator to simulate QAOA with up to 40 qubits, enabling a scaling analysis of QAOA performance on the LABS problem. The details of the observed quantum speedup over state-of-the-art classical solvers are described in detail in Ref. [**?**].

5.2 Background

Consider the problem of minimizing a cost function $f : \mathcal{F} \to \mathbb{R}$ defined on a subset $\mathcal{F}$ of the Boolean cube $\mathbb{B}^n$. The bijection $\mathbb{B} \cong \{-1, 1\}$ is used to express the cost function f as a polynomial in terms of spins

$$f(\mathbf{s}) = \sum_{k=1}^{L} w_k \prod_{i \in \mathbf{t}_k} s_i, \qquad s_i \in \{-1, 1\}. \tag{5.1}$$

The polynomial is defined by a set of terms $\mathcal{T} = \{(w_1, \mathbf{t_1}), (w_2, \mathbf{t_2}), \dots (w_L, \mathbf{t_L})\}$. Each term consists of a weight $w_k \in \mathbb{R}$ and a set of integers $\mathbf{t_k}$ from 1 to n, i.e., $\mathbf{t}_k \subseteq \{i \mid 1 \leq i \leq n\}$. Constant offset is encoded using a term $(w_{\text{offset},\emptyset})$.

We present numerical results for QAOA applied to the following two problems. First, we consider the commonly studied MaxCut problem. The cost function for the MaxCut problem is given by $\sum_{i,j \in E} \frac{1}{2} s_i s_j - \frac{|E|}{2}$ where $G = (V, E)$ is the problem graph, $\mathcal{T} = E$, and $s_i \in \{-1, 1\}$ are the variables to be optimized. Second, we consider the Low Autocorrelation

Binary Sequences (LABS) problem. The cost function for the LABS problem with n variables is given by $2\sum_{i=1}^{n-3} s_i \sum_{t=1}^{\lfloor \frac{n-i-1}{2} \rfloor} \sum_{k=t+1}^{n-i-t} s_{i+t}s_{i+k}s_{i+k+t} + \sum_{i=1}^{n-2} s_i \sum_{k=1}^{\lfloor \frac{n-i}{2} \rfloor} s_{i+2k}$.

The QAOA state is prepared by applying phase and mixing operators in alternation. The phase operator is diagonal and adds phases to computational basis states based on the values of the cost function. The mixing operator, also known as the mixer, is non-diagonal and is used to induce non-trivial dynamics. The phase operator is created using the diagonal problem Hamiltonian given by $\hat{C} = \sum_{\mathbf{x}\in\mathcal{F}} f(\mathbf{x}) |\mathbf{x}\rangle\langle\mathbf{x}|$, where $|\mathbf{x}\rangle$ is a computational basis quantum state. The spectrum of the operator matches the values of the cost function to be optimized, thus the ground state $|\mathbf{x}^*\rangle$ of such Hamiltonian corresponds to the optimal value $f(\mathbf{x}^*)$. The goal of QAOA is to bring the quantum system close to a state such that upon measuring it, we obtain $\mathbf{x}^*$ with high probability.

The QAOA circuit is given by

$$|\vec{\gamma}\vec{\beta}\rangle = \prod_{l=1}^{p} \left(e^{-\mathrm{i}\beta_l \hat{M}} e^{-\mathrm{i}\gamma_l \hat{C}} \right) |s\rangle \,.$$

If $\mathcal{F} = \mathbb{B}^n$, the standard choices are the transverse-field operator $\hat{M} = \sum_i \mathrm{X}_i$ as the mixer and uniform superposition $|+\rangle^{\otimes n}$ as the initial state. The free parameters γ_l and β_l are chosen to minimize the expected solution quality $\langle\vec{\gamma}\vec{\beta}|\hat{C}|\vec{\gamma}\vec{\beta}\rangle$, typically using a local optimizer.

5.3 Simulation of QAOA using `QOKit`

The convention of representing a quantum program as a sequence of quantum gates may pose limitations when simulating QAOA classically. In standard gate-based simulators such as Qiskit and QTensor, the phase operator must be compiled into gates. The number of these gates typically scales polynomially with the number of terms in the cost function $|\mathcal{T}|$. The overhead is especially large when considering objectives with higher order terms, such as k-SAT with $k > 3$ and Low Autocorrelations Binary Sequences (LABS) problem.

Existing state-vector simulators primarily work by iterating over each gate in the circuit and modifying the state vector. By exploiting the structure of the circuit, our state-vector simulator recognizes that each application of the phase operator involves the same set of gates and that the set acts as a diagonal operator, reducing the cost of simulation.

5.3.1 Precomputation of the cost vector

`QOKit` precomputes the diagonal elements in the operator $\hat{C}$, which are the values of the cost function f for each assignment of the input. The values are stored as a 2^n-sized cost vector, which encodes all the information about the problem Hamiltonian. `QOKit` provides simple high-level API which supports both cost functions defined as a polynomial on spins (see Listing 5.1), as well as a Python lambda function. For precomputation using polynomial terms (Eq. 5.1), we start by allocating an array of zeroes, and iterate over terms in $\mathcal{T}$, applying a GPU kernel in-parallel for each element of the array. The binary representation of an index of a vector element corresponds to qubit values in a basis state. This allows us to calculate the value of the term using bitwise-XOR and "population count" operations. The kernel calculates the term value and adds it to a single element of the vector in-place. This has the advantage of locality, which is beneficial for GPU parallelization and distributed computing.

To apply the phase operator with parameter γ_l, we perform an element-wise product of the state vector and $e^{-\mathrm{i}\gamma_l \vec{C}}$, where $\vec{C}$ is the cost vector and the exponentiation is applied element-wise. After simulating the QAOA evolution, we reuse the precomputed $\vec{C}$ to evaluate the expected solution quality $\langle \vec{\gamma}\vec{\beta}|\hat{C}|\vec{\gamma}\vec{\beta}\rangle$ by taking an inner product between $\vec{C}$ and the QAOA state.

5.3.2 Mixing operator

Application of the mixing operator is more challenging than that of the phase operator, and accounts for the vast majority of computational cost in our simulation. We briefly discuss the implementation using the example of the transverse-field mixer. Other mixers are implemented similarly. The transverse-field mixer can be decomposed into products of local gates as $U_M = e^{-\mathrm{i}\beta \sum_i \mathrm{X}_i} = \prod_i e^{-\mathrm{i}\beta \mathrm{X}_i}$. Each gate $e^{-\mathrm{i}\beta \mathrm{X}} = \cos(\beta)\mathrm{I} - \mathrm{i}\sin(\beta)\mathrm{X}$ "mixes" two probability amplitudes, and all n gates mix all 2^n probability amplitudes. Classical simulation of this operation requires all-to-all communication, where each output vector element depends on every entry of the input vector. For example, for $\beta = \pi/2$ the phase operator implements the Walsh-Hadamard transform, which is a Fourier transform on the Boolean cube $\mathbb{B}^n$. The definition of QAOA mixing operator via Walsh-Hadamard transform was known for a long time, see e.g., Refs. [**??**]. In fact, the ability of quantum computers to efficiently perform Walsh-Hadamard transform [**??**], which is the central building block for the famed Grover's algorithm [**?**], was the inspiration for the definition of the QAOA mixer [**?**].

Our GPU simulator implements each $e^{-\mathrm{i}\beta \mathrm{X}_i}$ of the mixing operator by applying a GPU kernel which modifies two elements of the state vector. Since these calculations do not interfere with each other, the updates on all pairs of elements in the state vector can be done in place and in parallel, hence well-utilizing the parallelization power of the GPU. The algorithm for simulating a single $e^{-\mathrm{i}\beta \mathrm{X}_i}$ is described in Algorithm 2. To simulate the full mixer, Algorithm 2 is applied to each qubit $i \in [n]$, as shown in Algorithm 3. Both algorithms modify the state vector in-place without using any additional memory.

The full QAOA simulation algorithm in `QOKit` is described in Algorithm 4. Furthermore, we implement the simulation using NVIDIA cuQuantum framework, by replacing Algorithm 3 with calls to the cuStateVec library. We refer to this implementation as QOKit (cuStateVec). In addition to the conventional transverse-field mixing Hamiltonian $M = \sum_i \mathrm{X}_i$,

we implement Hamming-weight-preserving XY mixer whose Hamiltonian is given by a set of two-qubit operators $M = \sum_{\langle i,j \rangle} \frac{1}{2}(\mathrm{X}_i\mathrm{X}_j + \mathrm{Y}_i\mathrm{Y}_j)$ for $\langle i, j \rangle$ corresponding to the edges of ring or complete graphs. The implementation leverages the observation that Algorithms 2 and 3 can be easily extended to SU(4) operators.

Algorithm 2 Fast SU(2) On A State Vector

Input: Vector $\boldsymbol{x} \in \mathbb{C}^N$ with $N = 2^n$, a unitary matrix $U_\star = \begin{pmatrix} a & -b^* \\ b & a^* \end{pmatrix} \in \mathrm{SU}(2)$ and a positive integer $d \in [n]$ **Output:** Vector $\boldsymbol{y} = U\boldsymbol{x}$, where $U = \mathrm{I}^{\otimes(d-1)} \otimes U_\star \otimes \mathrm{I}^{\otimes(n-d)}$ and I is the 2-dimensional identity matrix Create a reference $\boldsymbol{y}$ to input vector $\boldsymbol{x}$ $k_1 = 1$ **to** 2^{n-d} $k_2 = 1$ **to** 2^{d-1} Compute indices: $l_1 \leftarrow (k_1 - 1)2^d + k_2$ $l_2 \leftarrow (k_1 - 1)2^d + k_2 + 2^{d-1}$ Simultaneously update $\boldsymbol{y}_{l_1}$ and $\boldsymbol{y}_{l_2}$: $\boldsymbol{y}_{l_1} \leftarrow a\boldsymbol{y}_{l_1} - b^*\boldsymbol{y}_{l_2}$ $\boldsymbol{y}_{l_2} \leftarrow b\boldsymbol{y}_{l_1} + a^*\boldsymbol{y}_{l_2}$ **return** $\boldsymbol{y}$

Algorithm 3 Fast Uniform SU(2) Transform (Single-node)

Input: Vector $\boldsymbol{x} \in \mathbb{C}^N$, a unitary matrix $U \in \mathrm{SU}(N)$ decomposable into tensor product of n unitary matrices in SU(2), i.e. $U = \bigotimes_{i=1}^n U_i = U_n \otimes \cdots \otimes U_2 \otimes U_1$, where $U_i = \begin{pmatrix} a_i & -b_i^* \\ b_i & a_i^* \end{pmatrix} \in \mathrm{SU}(2)$ and $N = 2^n$ **Output:** Vector $\boldsymbol{y} = U\boldsymbol{x}$ Create a reference $\boldsymbol{y}$ to input vector $\boldsymbol{x}$ $i = 1$ **to** n Apply Algorithm 2 with $U_\star \leftarrow U_i$ and $d \leftarrow i$ **return** $\boldsymbol{y}$

Algorithm 4 Fast Simulation of QAOA

Input: Initial vector $\boldsymbol{x} \in \mathbb{C}^N$, QAOA circuit parameters $\boldsymbol{\beta}, \boldsymbol{\gamma} \in \mathbb{R}^p$, cost function $f : \mathbb{Z}_2^n \to \mathbb{R}$, where $N = 2^n$ **Output:** State vector after applying the QAOA circuit to $\boldsymbol{x}$ Pre-compute (and cache) cost values for all binary strings into a vector $\boldsymbol{c} \in \mathbb{C}^N$ Initialize output vector $\boldsymbol{y} \leftarrow \boldsymbol{x}$ $l = 1$ **to** p Apply phase operator: **for** $k = 1$ **to** N **do** $y_k \leftarrow e^{-\mathrm{i}\gamma_l c_k} y_k$ **end for** Apply mixing operator: Apply Algorithm 3 on $\boldsymbol{y}$ with $a_i \leftarrow \cos\beta_l$, $b_i \leftarrow \sin\beta_l$ $\forall\ i \in [n]$ **return** $\boldsymbol{y}$

5.3.3 Distributed simulation

A typical supercomputer consists of multiple identical compute nodes connected by a fast interconnect. Each node in turn consists of a CPU and several GPUs. Since GPUs are much faster in our simulation tasks, we do not use CPUs in our distributed simulation. Each

of K GPUs holds a slice of the state vector, which corresponds to fixing the values of a set of $k = \log_2(K)$ qubits. For example, for $K = 2$ GPUs, the first GPU holds probability amplitudes for states with the first qubit in state $|0\rangle$, while the second GPU holds states with first qubit in the state $|1\rangle$. In general, using K GPUs allows us to increase the simulation size by k qubits.

During the precomputation, the cost vector $\vec{C}$ is sliced in the same way as the state vector. Due to the locality discussed above, the precomputation and the phase operator application do not require any communication across GPUs. The most expensive part of the simulation is the mixing operator, since it requires an all-to-all communication pattern. In our simulation we distribute the state vector by splitting it into K chunks, which corresponds to fixing first k qubits, which we call *global qubits*. Bits of the binary representation of the node index determine the fixed qubit values. The remaining $n - k$ qubits are referred to as *local qubits*.

The mixer application starts by applying the $e^{-\mathrm{i}\beta \mathrm{X}_i}$ gates that correspond to local qubits. To apply X rotations on global qubits, we reshape the distributed state vector using the `MPI_Alltoall` MPI collective. This operation splits each local state vector further into K subchunks and transfers subchunk A of process B into subchunk B of process A. If each subchunk consists of one element and we arrange the full state vector in a matrix with process id as column index and subchunk id as the row index, then the call to `MPI_Alltoall` performs a transposition of this matrix. For a n-qubit simulation the algorithm requires $2k \leq n$ to ensure that there is at least one element in each subchunk. Consider the state vector reshaped as a tensor V_{abc} with a being the process id representing the k global qubits, b being a multi-index of first k local qubits, and c being a multi-index of the last $n - 2k$ qubits. Then the `MPI_Alltoall` operation corresponds to a transposition of the first two indices, i.e., $V_{abc} \to V_{bac}$. Thus, after this transposition, the global qubits become local and we are free to apply operations on those k global qubits locally in each process. The

algorithm concludes by applying the `MPI_Alltoall` once again to restore the original qubit ordering. This algorithm is described in Algorithm 5.

Algorithm 5 Fast Uniform SU(2) Transform (Multi-node)

Input: Vector $\boldsymbol{x} \in \mathbb{C}^N$ distributed over K nodes, a unitary matrix $U = \bigotimes_{i=1}^{n} U_i = U_n \otimes \cdots \otimes U_2 \otimes U_1$, where $U_i \in \mathrm{SU}(2)$ and $N = 2^n$ **Output:** Distributed vector $\boldsymbol{y} = U\boldsymbol{x}$ Create a reference $\boldsymbol{y}$ to the local slice of input vector $\boldsymbol{x}$ $i = 1$ **to** $n - \log_2 K$ Apply Algorithm 2 with $U_\star \leftarrow U_i$ and $d \leftarrow i$ to the local slice $\boldsymbol{y}$. Run in-place `MPI_AlltoAll` on the local slice $\boldsymbol{y}$. $i = n - \log_2 K + 1$ **to** n Apply Algorithm 2 with $U_\star \leftarrow U_i$ and $d \leftarrow i - \log_2 K$ to the local slice $\boldsymbol{y}$. Run in-place `MPI_AlltoAll` on the local slice $\boldsymbol{y}$. **return** $\boldsymbol{y}$

The `MPI_Alltoall` is known be a challenging collective communication routine, since it requires the total transfer of the full state vector K times. There exist many algorithms for this implementation [**??**], each with its own trade-offs. Furthermore, the same communication problem occurs in applying distributed Fast Fourier Transform (FFT), which has been studied extensively [**????**]. In this work, we use the out-of-the-box MPI implementation Cray MPICH. Utilizing the research on distributed FFT may help further improve our implementation.

5.4 Examples of use

`QOKit` consists of two conceptual parts:

9999Low-level simulation API defined by an abstract class `qokit.fur.QAOAFastSimulatorBase`

Easy-to-use one-line methods for simulating MaxCut, LABS and portfolio optimization problems

9999 The low-level simulation API is designed to provide more flexibility in terms of inputs, methods and outputs of simulation. The simulation inputs can be specified by providing either terms $\mathcal{T}$ or existing pre-computed diagonal vector. The simulation method is specified by using a particular subclass of `qokit.fur.QAOAFastSimulatorBase` or by using

a shorthand method `qokit.fur.choose_simulator`. This simulator class is the main means of simulation, with input parameters being passed in the constructor, the simulation done in `simulate_qaoa` method, and outputs type specified by choosing a corresponding method of the simulator ob

in Listing 5.1.

```python
import qokit
simclass = qokit.fur.choose_simulator(name='auto')
n = 28  # number of qubits
# terms for all-to-all MaxCut with weight 0.3
terms = [(.3, (i, j)) for i in range(n) for j in range(i+1, n)]
sim = simclass(n, terms=terms)
# get precomputed cost vector
costs = sim.get_cost_diagonal()
result = sim.simulate_qaoa(gamma, beta)
E = sim.get_expectation(result)
```

Listing 5.1: Evaluating the QAOA objective for weighted MaxCut problem on an all-to-a graph using `QOKit`.

`QOKit` implements five different simulator classes that share the same API:

1. `python` – A portable CPU `numpy`-based version
2. `c` – Custom CPU simulator implemented in C
3. `nbcuda` – GPU simulator using `numba`
4. `gpumpi` – A distributed version of the GPU simulator
5. `cusvmpi` – A distributed GPU simulator with cuStateVec as backend

To choose from the simulators, one may use one of the following three methods, depending on the choice of mixer type:

1. `qokit.fur.choose_simulator()`
2. `qokit.fur.choose_simulator_xyring()`

3. `qokit.fur.choose_simulator_xycomplete()`

Each of these simulators accepts an optional `name` parameter. The default simulator is chosen based on existence of GPU or configured MPI environment. An example of using a custom mixer for simulation is provided in Listing 5.2.

```
import qokit
simclass = qokit.fur.choose_simulator_xycomplete()
n = 40
terms = qokit.labs.get_terms(n)
sim = simclass(n, terms=terms)
result = sim.simulate_qaoa(gamma, beta)
E = sim.get_expectation(result)
```

Listing 5.2: Using `QOKit` with a different mixing operator: $M = \sum_{\langle i,j \rangle} \frac{1}{2}(\mathrm{X}_i\mathrm{X}_j + \mathrm{Y}_i\mathrm{Y}_j)$ for tuples $\langle i, j \rangle$ from a complete graph on qubits.

The constructor of each simulator class accepts one of `terms` or `costs` argument. The `terms` argument is a list of tuples $(w_k, \mathbf{t}_k)$, where w_k is the weight of product defined by $\mathbf{t}_k$, whih is a tuple of integers specifying the indices of Boolean variables involved in this product, as described in Equation 5.1. The simulation method returns a `result` object, which is a representation of the evolved state vector. The data type of this object may change depending on simulator type, and for best portability it is advised to use the output methods instead of directly interacting with this object. The output methods all have `get_` prefix, accept the `result` object as their first argument, and return CPU values. These methods are:

1. `get_expectation(result)`
2. `get_overlap(result)`
3. `get_statevector(result)`
4. `get_probabilities(result)`

When evaluating the expectation and overlap with the ground state, the cost vector from the phase operator is used by default. This vector is precomputed at the class instantiation and can be retrieved using `get_cost_diagonal()` method. Alternatively, the user may specify a custom cost vector by passing it as the `costs` argument when calling `get_expectation` or `get_overlap`.

The output methods may accept additional optional arguments depending on the type of the simulator. For example, GPU simulators' `get_probabilities` method has `preserve_state` argument (default `True`) which specifies whether to preserve the statevector for additional calculations; otherwise, the norm-square operation will be applied in-place. In both cases, the method returns a real-valued array of probabilities. Distributed GPU simulators accept `mpi_gather` argument (default `True`) that signals the method to return a full state vector on each node. Specifying `mpi_gather = True` guarantees that the same code will produce the same result if the hardware-specific simulator class is changed. An example of using `QOKit` for distributed simulation is provided in Listing 5.3.

```
import qokit
simclass = qokit.fur.choose_simulator(name='cusvmpi')
n = 40
terms = qokit.labs.get_terms(n)
sim = simclass(n, terms=terms)
result = sim.simulate_qaoa(gamma, beta)
E = sim.get_expectation(result, preserve_state=False)
```

Listing 5.3: Evaluating the QAOA objective for LABS problem using MPI on a distributed computing system using `QOKit`. The `preserve_state` argument is used to reduce memory usage when evaluating the expectation value.

5.5 Performance of `QOKit`

We now present a comparison of `QOKit` performance to state-of-the-art state-vector and tensor-network quantum simulators. We show that our framework has lower runtimes and

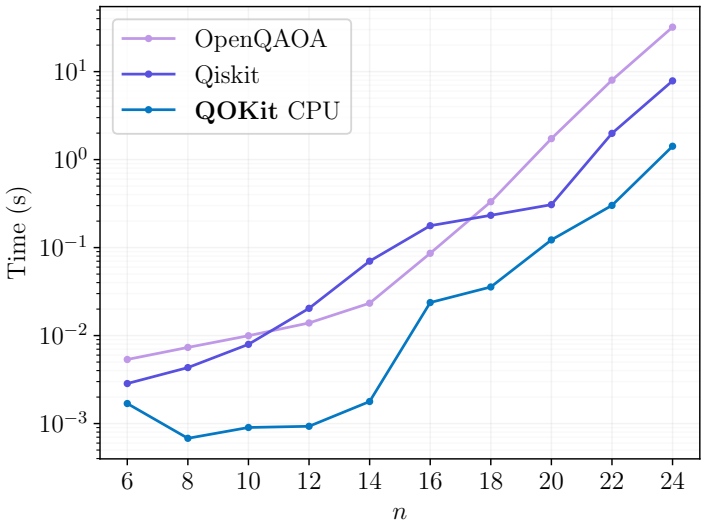


Figure 5.2: Runtime of end-to-end simulation of QAOA expectation value with $p = 6$ on MaxCut problem on 3-regular graphs with commonly-used CPU simulators for QAOA. They time plotted is the mean over 5 runs.

scales well to large supercomputing systems. All reported benchmarks are executed on the Polaris supercomputer accessed through the Argonne Leadership Computing Facility. Single-node results are obtained using a compute node with two AMD EPYC 7713 64-Core CPUs with 2 threads per core, 503 GB of RAM and an NVIDIA A100 GPU with 80 GB of memory. In all experiments the state vector is stored with double precision (`complex128` data type).

5.5.1 CPU and GPU simulation

The CPU simulation is implemented in two ways: using the NumPy Python library and using a custom C code ("`c`" simulator above). The latter is more performance, so we only report the results with `c` simulator. We evaluate the CPU performance by simulating QAOA with $p = 6$ on MaxCut random regular graphs.

Figure 5.2 shows a comparison of runtime for varying number of qubits for commonly-used CPU simulators. We use `QOKit c` simulator, Qiskit Aer state-vector simulator version 0.12.2, and OpenQAOA "vectorized" simulator version 0.1.3. We observe $\approx 5-10\times$ speedup

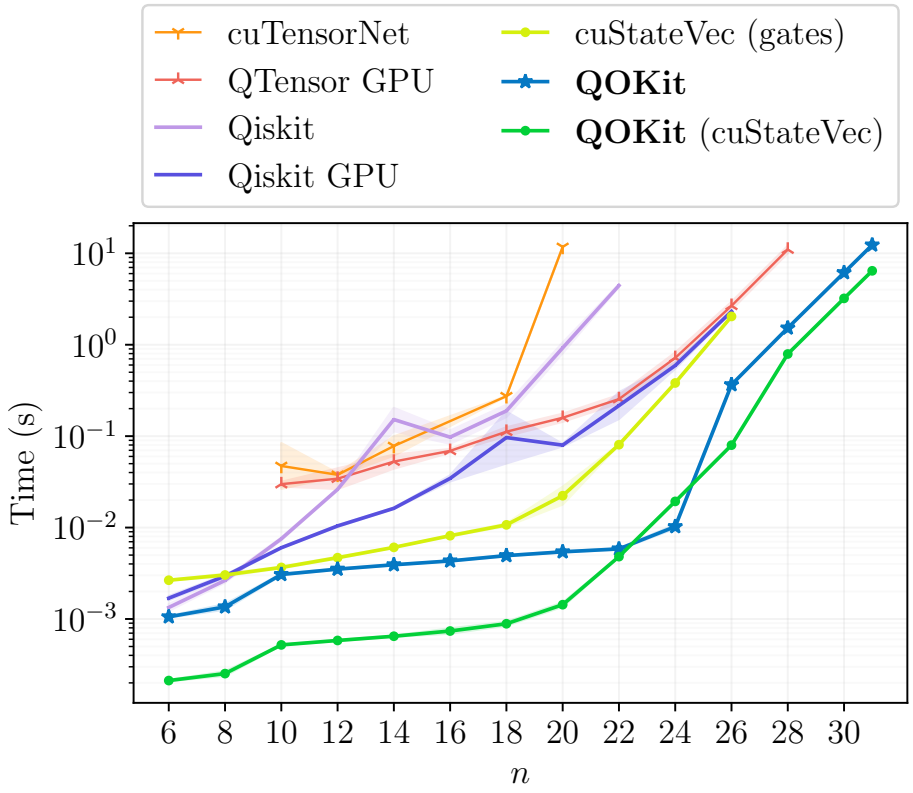


Figure 5.3: Time to apply a single layer of QAOA for the LABS problem with commonly-used CPU and GPU simulators. `QOKit` simulator uses the precomputation which is not included in current plot. The precomputation time is amortized, as shown on Figure 5.4. `QOKit` can be configured to use cuStateVec for application of the mixing operator, which provides the best results.

against Qiskit [?] and OpenQAOA [?] across a wide range of values of n. We note that the simulation method in QAOAKit [?] is Qiskit, which is why we do not benchmark it separately.

We evaluate the GPU performance by evaluating time to simulate one layer of QAOA applied to the LABS problem. Fig. 5.3 provides a comparison between `QOKit` and commonly used state-vector (Qiskit [?] version 0.43.3, cuStateVec [?]) and tensor-network (cuTensorNet [?], QTensor [Lykov, 2021]) simulators. We used CuQuantum Python package version 23.6.0 and cudatoolkit version 14.4.4. The tensor network timing is obtained by running calculation of a single probability amplitude for various values of $1 \leq p \leq 15$ and dividing the total contraction time by p. Deep circuits have optimal contraction order that produces

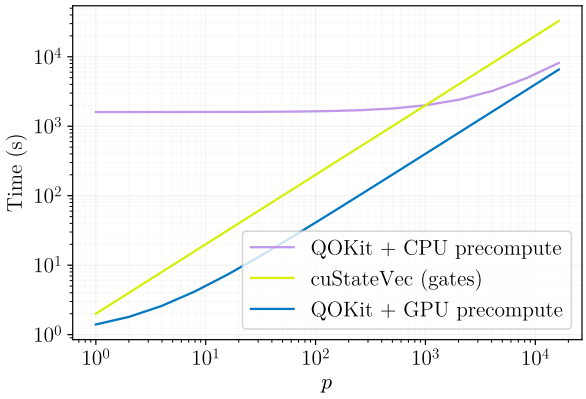


Figure 5.4: Total simulation time vs. number of layers in QAOA circuit for LABS problem with $n = 26$. The GPU precomputation is fast enough to provide speedup over the gate-based state-vector simulation (cuStateVec) even for a single evaluation of the QAOA circuit.

contraction width equal to n. Since obtaining batches of amplitudes does not produce high overhead [**?**], this serves as a lower bound for full state evolution. Note that the so-called "lightcone approach", wherein only the reverse causal cone of the desired observable is simulated, does not significantly reduce the resource requirements due to the high depth and connectivity of the phase operator. For QTensor, "tamaki_30" contraction optimization algorithm is used. CuTensorNet contraction is optimized with default settings. It is possible that the performance can be improved by using diagonal gates [**?**], which are only partially supported by cuTensorNet at this moment.

For $n > 20$, we observe that the precomputation provides orders of magnitude speedups for simulation of a QAOA layer. The LABS problem has a large number of terms in the cost function, leading to deep circuits which put tensor network simulators at a disadvantage. As a consequence, we observe that tensor network simulators are slower than state-vector simulation. We also observe that using cuStateVec as a backend for mixer gate simulation provides additional $\approx 2\times$ speedup, possibly due to higher numerical efficiency achieved by in-house NVIDIA implementation. We do not include the precomputation cost in Fig. 5.3. This cost is amortized over application of a QAOA layer as shown in Fig. 5.4, and is negligible if precomputation is performed on GPU. Simulation of each layer in a deep quantum circuit has

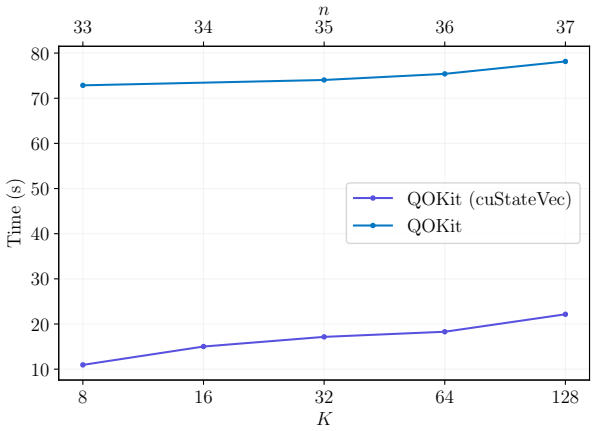


Figure 5.5: Weak scaling results for simulation of 1 layer of LABS QAOA on Polaris supercomputer. We observe that cuStateVec backend has lower communication overhead, leading to lower overall runtime.

the same time and memory cost. Thus, to obtain the time for multiple function evaluations, one can simply use this plot with aggregate number of layers in all function evaluations.

Our best GPU performance for QAOA on LABS problem is ≈ 6 seconds per QAOA layer for $n = 31$ using double precision. This simulation requires the same memory amount as one with $n = 32$ using single precision. In addition to our own implementation, we benchmark the same simulator as in the Ref. [?] on the LABS problem. For smaller $n \leq 26$, `QOKit` with cuStateVec shows a $\approx 20\times$ speedup from our precomputation approach. We discuss the choice of cuStateVec as the baseline as well as other state-of-the-art simulation techniques in Sec. 5.6.

5.5.2 Distributed simulation

Finally, we scale the QAOA simulation to $n = 40$ qubits using 1024 GPUs of the Polaris supercomputer. At $n = 40$, we observe a runtime of ≈ 20 s per layer. The results of the simulation are discussed in detail in Ref. [?]. Here, we focus on the technical aspects of the simulation.

In distributed experiments, we use compute nodes of Polaris with 4 NVIDIA A100 GPUs

with 40GB of memory. The maximum values of f are known for $n < 65$, and they are less then 2^{16}. Therefore, we are able to store the precomputed diagonal as a 2^n vector of `uint16` values, which reduces the memory overhead of the cost value vector. As discussed in Section 5.3.3, the most expensive part of this simulation is communication. We implement two approaches for this simulation, a custom MPI code that uses `MPI_Alltoall` collective and an implementation leveraging the distributed index swap operation in cuStateVec. Weak scaling results in Figure 5.5 demonstrate the advantage of cuStateVec implementation of communication. The GPUs co-located on a single node are connected with high-bandwidh NVLink network. To transfer data between nodes, GPU data need to transfer to CPU for subsequent transfer to another node. This requires the communication to correctly choose the communication method depending on GPU location. MPI has built-in support which can be enabled using `MPI_GPU_SUPPORT_ENABLED` environment variable. However, it shows worse performance than the cuStateVec communication code, which uses direct CUDA peer-to-peer communication calls for local GPU communication. We observe that our performance is comparable to distributed simulation reported in Ref. [**?**], despite having 2× fewer GPUs per node. This is due to the majority of time being spent in communication, which is confirmed by our smaller-scale profiling experiments. Further research, including adapting the communication patterns used in high-efficiency FFT algorithms, may improve our results.

5.6 Related work

Classical simulation of quantum systems is a dynamic field with a plethora of simulation algorithms [**?**Lykov, 2021; **?**; **?**] and a variety of use cases [**??**]. The main approaches to simulation are tensor network contraction algorithms and state-vector evolution algorithms. Tensor network algorithms are able to utilize the structure of quantum circuit and need not store the full 2^n-dimensional state vector when simulating n qubits. Instead, they construct a tensor network and contract it in the most efficient way possible. This approach works

best when the circuit is shallow, since the tensor network contraction can be performed across qubit dimension instead of over time dimension. The main research areas of this approach are finding the best contraction order [???] and applying approximate simulation algorithms [??]. However, while there is no theoretical limitation on simulating deep circuits using tensor networks, it is challenging to implement a performant simulator of deep quantum circuits based on tensor networks, as demonstrated by the numerical experiments above.

The state vector evolution algorithms are more straightforward and intuitive to implement. The main limitation of state-vector simulator is the 2^n size of the state vector. There are many approaches to improve state-vector simulators. Compressing the state vector has been proposed to reduce the memory requirement and enable the simulation of a higher number of qubits [?]. To utilize the structure in the set of quantum gates, some state-vector simulators use the gate fusion approach [????]. The idea is to group the gates that act on a set of F qubits, then create a single F-qubit gate by multiplying these gates together. This approach is often applied for $F = 2$ and provides significant speed improvements. Computing the fused gate requires storing 4^F complex numbers, which is a key limitation. Our approach corresponds to using gate fusion with $F = n$, but the group of gates is known to produce a diagonal gate, which can be stored as a vector of only 2^n elements.

In our comparison in Sec. 5.5.1, we do not enable gate fusion in cuStateVec. While gate fusion may improve the performance of cuStateVec, we believe it is very unlikely to achieve the same efficiency as our method of using the precomputed diagonal cost operator. Our argument is based on examining the results reported in Ref. [?]. The central challenge for gate fusion is presented by the fact that the phase operator for the LABS problem requires many gates to implement. For example, for $n = 31$, the LABS cost function has $\approx 75n$ terms, with many of them being 4-order terms. If decomposed into 2-qubit gates, the circuit for QAOA with $p = 1$ for the LABS problem has $\approx 160n$ gates after compilation. For comparison, QAOA circuit from Ref. [?] for $n = 33$ and $p = 2$ has $50n$ un-fused gates,

which is $\approx 7\times$ fewer. After gate fusion, the circuits from Ref. [?] have $\approx 4n$ fused gates. Our precomputation approach reduces the number of gates to practically only the n mixer gates. Thus, assuming that application of a single gate takes the same amount of time for any statevector simulator, we can expect a speedup in the range of $4 - 160\times$. We note that while this estimate does not take into account various other time factors and variation in gate application times, it provides intuition for why we rule out gate fusion outperforming our techniques.

While there exist a multitude of quantum simulation frameworks, the best results for state-vector GPU simulation that we found in the literature were reported in Ref. [?] (cuQuantum) and Ref. [?] (qsim). Ref. [?] reports ≈ 10 seconds for simulating QAOA with $p = 2$, $n = 33$ qubits and 1650 gates, using the `complex64` data type on a single A100 GPU with 80 GB memory. Ref. [?] presents single-precision simulation results on a A100 GPU with 40 GB memory. The benchmark uses random circuits of depth of 20. The reported simulation time for $n = 32$ is ≈ 6 seconds. Notably, this time may depend significantly on the structure of the circuit since it impacts the gate fusion, as shown in Ref. [?]. Assuming similar gate count, these results show very similar performance, since the simulation in Ref. [?] is two times larger. This motivates our use of cuQuantum (Ref. [?]) as a baseline state-of-the-art state-vector quantum simulator.

Symmetry of the function to be optimized has been shown to enable a reduction in the computational and memory cost of QAOA simulation [???]. While we do not implement symmetry-based optimizations in this work, they can be combined with our techniques to further improve performance.

In addition to the simulation method, many simulators differ in the scope of the project. Some simulators like cuQuantum [?] position themselves as a simulator-development SDK, with flexible but complicated API. On the other hand, there exist simulation libraries that focus on the quantum side and delegate the concern of low-level performance to other li-

braries [??]. Many software packages exist somewhere in the middle, featuring full support for quantum circuit simulation and focusing on end-to-end optimization efforts on a particular quantum algorithm or circuit type [??]. `QOKit` is positioned as one of such packages, as it provides both an optimized low-level QAOA-specific simulation algorithm as well as high-level quantum optimization API for specific optimization problems.

5.7 Conclusion

We develop a fast and easy-to-use simulation framework for quantum optimization. We apply a simple but powerful optimization by precomputing the values of the cost function. We use the precomputed values to apply the QAOA phase operator by a single elementwise multiplication and to compute the QAOA objective by a single inner product. We provide an easy-to-use high-level API for a range of commonly considered problems, as well as low-level API for extending our code to other problems. We demonstrate orders of magnitude gains in performance compared to commonly used quantum simulators. By scaling our simulator to 1,024 GPUs and 40 qubits, we enabled an analysis of QAOA on the Low Autocorrelation Binary Sequences problem that demonstrated a quantum speedup over state-of-the-art classical solvers [?].

After this manuscript appeared on arXiv, we became aware of a serial CPU-only Python implementation of a QAOA simulator that uses diagonal Hamiltonian precomputation and Fast Walsh-Hadamard transform to accelerate QAOA state simulation and QAOA objective evaluation [??]. We note that Ref. [?] requires two applications of fast Walsh-Hadamard transform (forward and inverse) and a diagonal Hamiltonian operation to simulate one layer of QAOA mixer, whereas Algorithms 2, 3 apply the mixer in one step with a cost equivalent to one application of fast Walsh-Hadamard transform. In addition, the implementation of fast Walsh-Hadamard transform in Ref. [?] requires one additional copy of the input state vector, whereas Algorithms 2, 3 applies the mixer in place.

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 Gorshkov, 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.

Danylo Lykov, Ruslan Shaydulin, Yue Sun, Yuri Alexeev, and Marco Pistoia. Fast simulation of high-depth qaoa circuits. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, SC-W 2023. ACM, November 2023. doi:10.1145/3624062.3624216. URL `http://dx.doi.org/10.1145/3624062.3624216`.

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`.

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 Dmello, 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.