School of Computation, Information and Technology - Informatics
Technical University of Munich

Master's Thesis in Informatics

High Performance Contraction of Brickwall Quantum Circuits for Riemannian Optimization

Fabian Putterer

School of Computation, Information and Technology - Informatics
Technical University of Munich

Master's Thesis in Informatics

High Performance Contraction of Brickwall Quantum Circuits for Riemannian Optimization

Hochleistungsauswertung von Brickwall-Quantenschaltkreisen für riemannische Optimierung

Author: Fabian Putterer
Supervisor: Prof. Dr. Christian Mendl
Advisors: Qunsheng Huang
Submission Date: October 15, 2023

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich diese Master's Thesis selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, October 15, 2023

Fabian Putterer

Abstract

Quantum computers show a lot of promise in enabling the large-scale computational workloads of the future. Their capability scales exponentially with the number of qubits used. Due to this property, they are well suited for Hamiltonian simulation, the simulation of quantum processes itself scaling exponentially. Various solutions for building a quantum circuit for this purpose exist. Those are not suitable for error-prone systems of the NISQ area though. One possible mitigation for this problem is shortening the circuit by using Riemannian optimization. This works well for 1-dimensional systems but breaks down when looking at larger 2-dimensional cases as the complexity of the classical method scales exponentially with the qubit count. Applying it therefore requires efficient brickwall tensor network contraction as well as gradient and Hessian computation performance and memory-wise. Existing methods for contraction and the similar problem of deep learning fail to exploit the specific constrained structure of the given brickwall and thereby fall short of solving the problem quickly. This thesis investigates whether brickwall tensor network contraction can be optimized sufficiently to be run in a reasonable time frame. Algorithms for the various subtasks such as gradient and Hessian computation are proposed. A native implementation of those is provided, tested for correctness, and evaluated for performance improvements. The largest gain is achieved by applying the tensor network to all 1-hot state vectors gate-wise instead of performing a contraction of the layer matrices. The other major improvement is using caching to store intermediate results. The performance of the gradient is increased by a factor of 30, that of the Hessian by 10. The memory requirement is reduced from at least 72 GB to less than 2 GB. The efficiency is L3 cache-bound but given enough space on a server processor scales well with parallelization over cores and distribution over multiple nodes. Using the algorithms developed in this work on a compute cluster, obtaining the gradients and Hessians required for the Riemannian quantum circuit optimization is possible significantly faster in a matter of weeks rather than years.

Acknowledgments

First of all I would like to thank my advisor Qunsheng Huang for always being there and supporting me when I had any questions about this complex topic. The brainstorming sessions with him helped me a lot to develop the main ideas that made this thesis possible. I would also like to thank Prof. Dr. Christian Mendl for guiding me in this work and swiftly pointing out when I was missing the forest for the trees to get me back on track.

I am also thankful for my friends being there when I needed them and always having my back.

Last but definitely not least, I would like to thank my parents for the support they have given me during my studies and other adventures.

Thank you to everyone who shared a part of this journey through life with me.

Contents

Eidesstattliche Erklärung iii

Abstract iv

Acknowledgments v

1 Introduction 1

2 Problem Statement 4
- 2.1 Original Problem . . . 4
- 2.2 Computing the Trace . . . 5
- 2.3 Contracting the Time Evolution Operator . . . 6
- 2.4 Simplified Problem . . . 6
- 2.5 Computing the Gradient and Hessian . . . 7

3 Target Platform 8
- 3.1 Existing Implementation . . . 8
- 3.2 Matrix-free Gate-wise Contraction . . . 8
- 3.3 Target Hardware . . . 9
  - 3.3.1 CPU . . . 10
  - 3.3.2 GPU . . . 11
  - 3.3.3 TPU . . . 11
  - 3.3.4 Compute Cluster . . . 12
  - 3.3.5 Summary . . . 12
- 3.4 Software Platform . . . 12
  - 3.4.1 Libraries . . . 12

4 Brickwall Circuit Contraction 14
- 4.1 Single Gate Application . . . 14
  - 4.1.1 Principle of Operation . . . 14
  - 4.1.2 Implementation . . . 14
  - 4.1.3 Hardware Aspects . . . 15
  - 4.1.4 Results . . . 16
- 4.2 Gate Hole Contraction . . . 17
  - 4.2.1 Principle of Operation . . . 17
  - 4.2.2 Implementation . . . 17
  - 4.2.3 Performance . . . 18
- 4.3 Indexing . . . 19
- 4.4 Matrix-free Contraction . . . 19

5 Problem and Performance Analysis 20
- 5.1 Metrics . . . 20
  - 5.1.1 Baseline System . . . 20

5.1.2 Absolute Time . . . 20
5.1.3 Original Layers Processed . . . 20
5.2 Problem Size . . . 21
5.2.1 Problem Structure . . . 21
5.2.2 Memory Requirement . . . 22
5.2.3 Complexity . . . 22

6 Gradient Computation **24**
6.1 Problem Statement . . . 24
6.2 Naive Implementation . . . 25
6.3 Cached Implementation . . . 26
6.3.1 Algorithm . . . 26
6.3.2 Deep Learning and Backpropagation . . . 28
6.4 Memory Requirements . . . 30
6.5 Evaluation . . . 30

7 Parallelization **33**
7.1 Parallel Speedup . . . 33
7.2 Cache Architecture . . . 34
7.3 Summary . . . 35

8 Hessian Computation **37**
8.1 Problem Statement . . . 37
8.2 Workload Estimation . . . 38
8.3 Naive Implementation . . . 38
8.4 Cached Implementation . . . 40
8.4.1 Algorithm . . . 41
8.5 Splitting . . . 43
8.6 Evaluation . . . 43

9 Future Work **46**
9.1 Dedicated Hardware . . . 46
9.1.1 Graphics Processors . . . 46
9.1.2 Neural Processing Units . . . 47
9.2 Optimization . . . 47
9.3 Large-scale Parallelization . . . 47
9.4 Riemannian Quantum Circuit Optimization . . . 48

10 Conclusion **49**

List of Figures **51**

Bibliography **53**

1 Introduction

A lot of research and engineering in various disciplines is based on computational results obtained from classical computers. Their performance and capacity increase every year enabling more and more complicated scientific discoveries. Even if this trend continues though, there are problems of such gigantic size that even the strongest supercomputers of the future will fail to solve them in any reasonable time frame.

To address this, a lot of effort has been invested into using quantum processes to perform computations that could never be realized on a classical computation device. The field of quantum computing and quantum information has a lot of promising future applications in multiple areas such as medical research, material development, and fundamental research. One potential use case is the simulation of quantum processes which are described by a Hamiltonian operator. This is a challenging task on a classical computer due to the exponential nature of a superposition. Using a quantum computer which is itself based on those fundamental quantum mechanical principles is therefore a prime choice for tackling this hard problem.

To perform the Hamiltonian simulation, a quantum circuit needs to be built based on the given operator. The conventional approach for doing so is to apply Trotter splitting[4]. The issue with this as with a lot of solutions in quantum computing at the moment is that current-day quantum hardware is still in its early stages and suffers from a significant amount of noise. This causes computational results to become unusable after a certain number of gates have been applied. For nearly all real-world applications this means that they cannot be used yet on the quantum systems of the so-called noisy intermediate-scale quantum (NISQ) era. One major point of focus for research at the moment is therefore circuit optimization, reducing the number of gates in a circuit and therefore the execution time while maintaining the same yet less noisy result.

In the case of Hamiltonian simulation and Trotter splitting, one potential optimization has been proposed by Kotil et al.[14]. They suggest constructing a brickwall circuit using a loss function between the Hamiltonian and the effect of the circuit and optimizing such using the Riemannian trust-region algorithm[2, chapter 7], a quadratic version of gradient descent based on gradient as well as Hessian. Their approach requires the ability to efficiently contract a tensor network representing the target loss function being minimized to obtain the gradient and Hessian of each part of the parametric brickwall gates.

Tensor Network Contraction

A lot of strategies for circuit optimization as well as problems in quantum computing in general rely on treating the linear map of the circuit as a tensor network and working with that representation instead. Lots of research has therefore been done on efficient tensor network contraction. The problem is quite tricky as the size of the involved tensors grows exponentially in the number of qubits in the system. Additionally, gates are only applied to a selection of qubits therefore contracting high-dimensional tensors over their various

respective dimensions. This causes memory accesses with changing strides of powers of 2 when iterating over the indices of the two tensors during the contraction. This results in poor performance due to almost perfect non-local cache usage.

To be able to build and optimize a circuit from the ground up using gradient descent or the Riemannian trust-region algorithm, a kind of model for the circuit needs to be chosen. The model should have a simple structure with few gates preferably acting on two qubits each, yet be able to approximate any unitary map. One such option is the Brickwall circuit as used by Kotil et al.[14]. As long as it contains a sufficient number of layers it can be called a unitary function approximator. This is similar to deep learning, which the entire problem itself is as well, where a neural network is used as a universal function approximator and is trained using a loss function, backpropagation, and gradient descent. Due to these properties required for the task of Hamiltonian simulation and its applications in general, this work will focus on efficiently contracting brickwall tensor networks and computing their gradient and Hessian. Even though the setup sounds very similar to deep learning it has multiple distinct differences that can be exploited allowing for significant performance gains when compared to typical machine learning methods.

Riemannian Quantum Circuit Optimization

The algorithms in this work can be used for any application that needs gradient and Hessian computation on brickwall circuits. They have been specifically built for enabling the execution of the method described by Kotil et al.[14] on larger 16-qubit systems though. Performing their method optimizing a circuit for Hamiltonian simulation by approximating it using a loss function and the Riemannian trust-region algorithm[2, chapter 7] requires the gradient and Hessian of said loss function to be obtained. The optimal gates $G_{opt} = \underset{G \in U(m)^{\times n}}{\arg\min} \|W(G) - U\|_F^2$ are determined by minimizing the loss function representing the distance between the real operator and its approximation. Kotil et al.[14] show that the target can be rewritten as $f(G) = -Re\,\mathrm{Tr}[U^\dagger W(G)]$ and represented as a tensor network. Optimization is therefore reduced to computing the gradient and Hessian of the tensor network which they solve by cutting holes into it and contracting and based on them using the Riemannian trust-region algorithm to minimize f. Due to exponential scaling, this problem is easily solved for smaller 4-qubit systems but becomes almost impossible for larger 2-dimensional 16-qubit systems. This work will therefore focus on performing the gradient and Hessian computation of brickwall tensor networks and therefore quantum circuits efficiently.

High Performance Contraction

Traditional approaches for tensor network contraction of quantum circuits rely on using the Kronecker product to combine the individual gate matrices into larger layer matrices. Those are then contracted using well-studied linear algebra operations implemented by a BLAS library that perform the multiplications efficiently. This approach is also used in the original implementation by Kotil et al.[14]. It works fine for smaller systems with 4 qubits and therefore 16 state vector entries and 16×16 matrices. It is unsuited for larger systems though as the size of the tensors scales exponentially resulting in a $2^{16} \approx 65000$ entry state vector and 68 GB layer matrices. Multiplying and storing those is too expensive for current-day hardware. A more efficient method of applying the gates individually

instead of computing the layer effect will be explored in the following chapters.

As already mentioned, the problem also shows quite a few similarities to deep learning and backpropagation. That field has been deeply studied for possible optimizations so some of the algorithms might be transferable to this use case. The problem with using those strategies is that there are subtle differences in the constraints on the optimization problem at hand such as the number of parameters, structure of the function approximator, and lack of non-linearities. Exploiting those properties yields significantly higher performance. Therefore methods from machine learning are not useful in solving this problem but might offer some insights and inspiration.

The main goal of this thesis is to optimize the tensor network contraction of brickwall circuits. The following chapters will investigate whether it is possible to increase the efficiency of this and the gradient and Hessian computation performance- and memory-wise to be able to apply the Riemannian quantum circuit optimization method for 16-qubit systems on current-day hardware in a realistic time frame.

2 Problem Statement

To optimize a quantum circuit for Hamiltonian Simulation using the method proposed by Kotil et al.[14], we need to be able to evaluate the target function $-Re\,Tr[U^{\dagger}W(G)]$, and its gradient and Hessian efficiently. This means we want to be able to contract the circuit $UW(G)$ within a reasonable time frame. This can be easily achieved for smaller systems with e.g. just 4 qubits arranged as a 1D structure by constructing the per-layer matrices and then just contracting them.

In the real world though quantum systems are 3-dimensional. It would therefore be desirable to be able to apply the same optimization method to 2D and 3D systems as well. A 2D system will realistically require at least a 4x4 grid of qubits and therefore 16 qubits in total.

Evaluating a brickwall circuit with $q = 16$ qubits naively will require a state vector with $2^q = 2^{16}$ entries and cause the construction of a $2^q \times 2^q$ matrix with $2^{2q} = 2^{32}$ entries for each layer. Such a state vector $|\psi\rangle \in \mathbb{C}^{2^{16}}$ stored as double precision floating point [1] will therefore have a size of 1 MiB. Each layer matrix $L \in \mathbb{C}^{2^{16} \times 2^{16}}$ applied to it will occupy 68 GB.

While storing this amount of memory is still possible on larger-scale systems, applying a matrix of this scale millions of times over is impossible on current hardware. An alternative approach to performing the contraction and evaluation is therefore needed.

This thesis focuses on achieving this for 16 qubit systems. It does so by investigating, implementing, and optimizing the performance-critical parts of the algorithm, being the contraction, gradient, and Hessian computation. It tries to answer whether and describes how it is possible to compute these values for a 2D system in a reasonable time frame to be usable for research purposes using classical present-day hardware.

2.1 Original Problem

At its core, the problem consists of finding the fastest way to evaluate a brickwall circuit. In this chapter, the original problem will be transformed into a simplified version that will then be implemented and optimized in later chapters.

Figure 2.1 shows a smaller version of the entire circuit of which the gradient and Hessian have to be computed. It is equivalent to the inner part of the trace of the target function:

$$U^{\dagger}W(G) \tag{2.1}$$

It consists of the adjoint of the unitary time evolution operator $U^{\dagger}$ and the brickwall circuit $W(G)$, with $G = \{G_1, G_2, \ldots, G_L\}$. For each layer i the gates share the same matrix G_i.

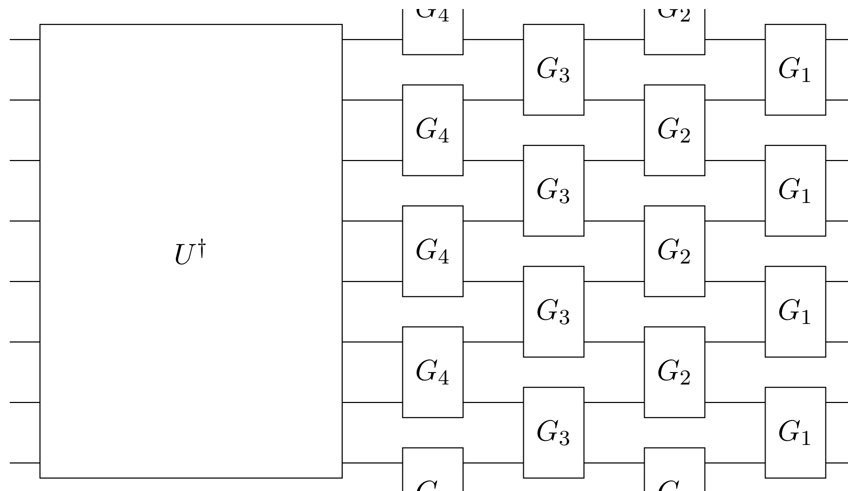


Figure 2.1: Circuit representation of the inner part of the trace as shown in equation 2.1

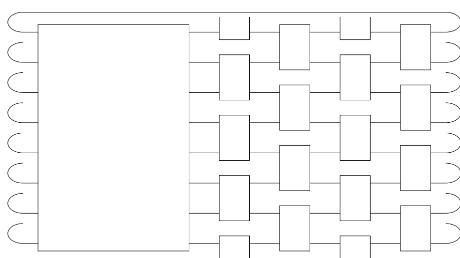
The circuit this thesis will be looking at works on 16 qubits and has around 8 layers. As each gate affects 2 qubits, there are 8 gates per layer, and 64 gates need to be applied in total to contract one circuit.

2.2 Computing the Trace

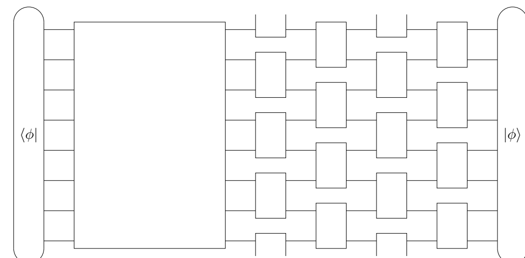
For the optimization procedure we need the real part of the trace of the circuit, and therefore have to determine the gradient and Hessian of:

$$Tr[U^\dagger W(G)]$$

We will represent this using a tensor network as an extension of the original circuit. This way the trace of the contracted circuit is therefore equivalent to a contraction of the tensor network with itself [14, p. 4] as can be seen in figure 2.2a. The trace is implemented by the connections wrapping around the back.



(a) The trace of the base circuit as a tensor network representation.



(b) Obtaining the trace as a sum of contractions with 1-hot state vectors as in equation 2.2.

Figure 2.2: Computing the trace function using tensor network contractions.

We can then insert an identity matrix into the connecting edges wrapping around the back of the circuit. The identity can be represented using the set of all 1-hot state vectors $\mathbb{O}$ as:

$$I = \sum_{|\psi\rangle \in \mathbb{O}} |\psi\rangle \langle\psi|$$

The entire trace can therefore be obtained by contracting the circuit with all possible 1-hot state vectors $|\psi\rangle \in \mathbb{O}$ on both sides and summing the result:

$$\mathrm{Tr}[U^\dagger W(G)] = \sum_{|\psi\rangle \in \mathbb{O}} \langle\psi| U^\dagger W(G) |\psi\rangle \tag{2.2}$$

The tensor network to be evaluated for each possible 1-hot state vector can be seen in figure 2.2b. The number of 1-hot vectors being $|\mathbb{O}| = 2^{16}$ implies that we will have to evaluate the circuit 2^{16} times.

2.3 Contracting the Time Evolution Operator

To evaluate the tensor network from figure 2.2b for a given $|\psi\rangle \in \mathbb{O}$ the unitary time evolution operator U needs to be used. It is a complex $2^{16} \times 2^{16}$ matrix and has a size of 68 GB. It is therefore too large to be stored in memory on a lot of systems, especially when highly parallelized or stored multiple times.

In the tensor network representation $\langle\psi| U^\dagger W(G) |\psi\rangle$, the operator $U^\dagger$ is directly connected to a 1-hot state vector on the left side. It can just be contracted with that in the beginning, therefore just selecting a single row $\langle\psi| U^\dagger = U_i^\dagger$ from the matrix.

As this is only performed once per 1-hot vector and the rest of the process is significantly more expensive, this can easily be solved by streaming the adjoint operator matrix rows from disk drive, thereby eliminating the need to store it in the system's memory. Access will be fastest by storing rows of the operator adjoint $U^\dagger$ instead of U.

This also reduces the contraction problem to

$$U_i^\dagger W(G) |\psi\rangle$$
$$\text{with } i \text{ being the 1-hot index of } |\psi\rangle$$

It effectively becomes just a contraction of a brickwall circuit with two different state vectors.

2.4 Simplified Problem

The main objective of this thesis is to develop a method to evaluate the gradient and Hessian of the target function as efficiently as possible by solving the reduced problem:

$$\sum_{|\psi\rangle \in \mathbb{O}} \langle\phi| W(G) |\psi\rangle \tag{2.3}$$

In this case, $\langle \phi | = U_i^\dagger$ is a row of the matrix, but the evaluation method should work for arbitrary $\langle \phi |$.

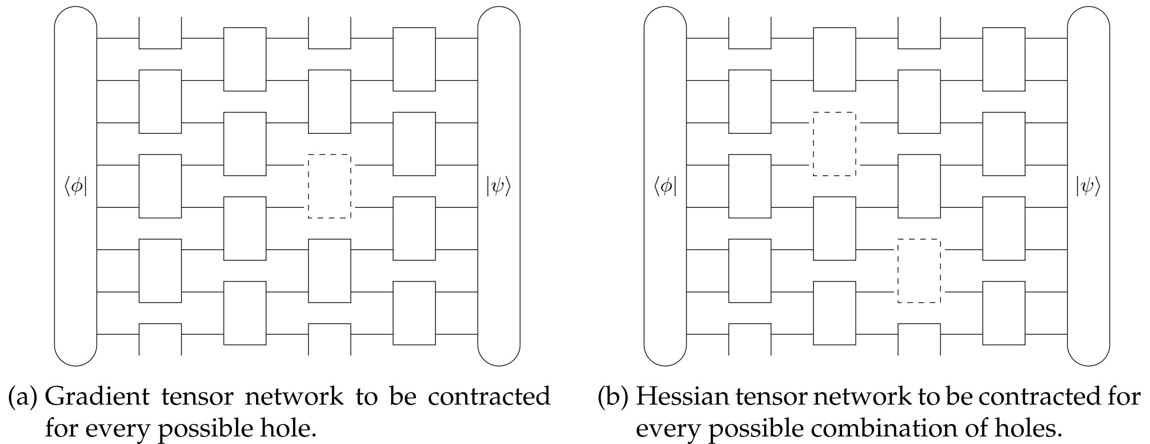
2.5 Computing the Gradient and Hessian

To perform the optimization procedure on the circuit the gradient and Hessian of the target function f and therefore of equation 2.3 have to be determined. The gradient is the derivative of f relative to each individual gate parameter, shared among an entire layer. The Hessian is the second derivative of f relative to each combination of parameters of two gates.

$$\nabla f(G) \in \mathbb{C}^{|G| \times (4\times 4)}$$
$$\mathbf{H}_f \in \mathbb{C}^{|G| \times |G| \times (4\times 4) \times (4 \times 4)}$$

They can be obtained by cutting holes into the tensor network representation and performing the contraction, as proposed by Kotil et al.[14, p. 4]. For the gradient, this is achieved by cutting a single hole for each gate site and summing the results of the individual contractions along the gates and layers. For the Hessian, the same effect can be achieved by cutting two holes, performing 16 contractions in the region between them, and summing the results of all possible hole combinations by the two gates they affect. The tensor network representations of both scenarios can be seen in figure 2.3.

This way the gradient and Hessian can be computed by contracting the tensor network with holes. This can be realized by just applying gates (see section 4.1), splitting at holes (see chapter 8), and performing contractions of the final hole (see section 4.2).



(a) Gradient tensor network to be contracted for every possible hole.

(b) Hessian tensor network to be contracted for every possible combination of holes.

Figure 2.3: The tensor network representations used for computing the gradient and Hessian of the target function. The actual problem has 16 qubits and 64 gates.

In the following sections, a method will be developed to obtain the gradient and Hessian of the expression in equation 2.3 as fast as possible based on cutting holes into the tensor network representation.

3 Target Platform

This chapter investigates the potential technology choices for implementing the algorithms described in the later chapters to solve the problem from chapter 2.

3.1 Existing Implementation

In their work, Kotil et al.[14, p. 5] provide an implementation of their optimization procedure. It uses Python[18] and is based on NumPy[12]. It works by computing the full matrix representation of each layer of the circuit using the Kronecker product of the individual gates and then contracting, in this case just multiplying, the resulting matrices.

As described in chapter 2, this approach works well for smaller systems but fails for larger cases such as 2-dimensional systems with 16 qubits. In those cases the computed matrices would reach a size of 2^{32} entries, occupying 68 GB of memory. They would therefore be too large for today's hardware, as working with them in a time-efficient manner over lots of iterations is computationally too expensive. Additionally, the use of an interpreted high-level programming language such as Python decreases performance. The usage of NumPy and therefore natively implemented mathematical operations such as matrix multiplications alleviates this a bit but there is still an overhead due to the independence of NumPy's optimizations from the actual problem being solved as well as interactions between native and interpreted code.

An additional drawback of this implementation is its lack of support for different hardware platforms and options. Currently, it only supports being executed on regular CPUs that support a Python interpreter and have a version of NumPy available for them.

The native NumPy code also uses parallelization to utilize the full performance available on the CPU by distributing the work of the individual matrix multiplications onto multiple cores. The disadvantage of this implementation is that the parallelization is done at the matrix multiplication level. This causes overhead after each operation due to data sharing, synchronization, and having to redistribute the new work and restart the workers. Parallelizing at the 1-hot encoded state vector level on the other hand would allow for full separation of workers, therefore minimizing synchronization and data sharing as well as enabling full utilization of the individual cores' caches containing only their current workload without any shared data.

The new implementation developed in this work tries to avoid, fix, or mitigate most of these issues.

3.2 Matrix-free Gate-wise Contraction

Forming, storing, and using the entire matrix for each layer during the contraction is too expensive for today's memory architectures. A workaround for this problem would be to

compute the required matrices on the fly. As they are the result of the Kronecker product of a gate with itself, applied 8 times, their entries can easily be determined and updated on the fly while performing the contraction. This way we avoid having to store it in memory. But as can be seen from table 3.1, even though most of the workload has been avoided, we are still performing a lot of unnecessary read operations on the state vector the matrix is being applied to. This is due to inner dependencies between the individual matrix entries that are the result of independence between the gate applications which are not being exploited yet. An alternative approach is therefore needed that avoids the matrix multiplication altogether.

This issue can be sidestepped by treating the gates independently, as if they were on different layers, and applying them sequentially. This way, effects from earlier gates are already contained in the computation chain. The workload is shifted from computing the effect of all gates on all state vector entries to the effect of a single, small gate to all state vector entries, applied multiple times. This "matrix-free", gate-wise contraction substitutes the quadratic scaling of the matrix operation for a linear scaling of having to apply a small matrix for each gate. This achieves the desired exploitation of the cumulative gate effect and avoids the redundancy that would be present in the full matrix.

Figure 3.1 shows this effect by comparing the number of operations performed by both methods.

	Gate-wise	Full Matrix
State vector reads	2^{19}	$\mathbf{2^{28}}$
State vector writes	2^{19}	2^{16}
Operations (mul/add)	2^{21}	$\mathbf{2^{32}}$

Figure 3.1: Comparison of approximate data accesses and operations performed during the two algorithms.

As can be seen, the gate-wise method will read $2^3 = 8$ full state vectors with 2^{16} entries each, while the matrix multiplication would read 4096. As memory and caching are the main bottlenecks in this computation, reducing the number of data accesses is crucial. Therefore, this thesis will be using the gate-wise application. In the case of the first layer, the matrix representation computed on the fly can be used though, as the input is a 1-hot state vector. The result will therefore just be a single column of the matrix which can be computed way cheaper than applying each gate individually. For the rest of the contraction gate-wise application is more efficient.

3.3 Target Hardware

The hardware to run the algorithm on should be chosen based on the requirements and the scale of the problem. As section 5.2 shows, the problem is rather large. An execution on a single system will therefore likely take too long and massive parallelization and/or distribution will be necessary. This has to be taken into account for determining the right system.

Table 3.2 gives an estimate of the computational performance that can be expected from different devices. It only lists computational speed though, for the contraction process, memory access speed also plays a vital role.

Device	FP32	FP64	Ratio	Cores
RTX 3080 (GPU)	30	0.47	1:64	8704
RTX 3060 (GPU)	13	0.20	1:64	3584
GTX 970 (GPU)	4	0.12	1:32	1664
Vega 6 (iGPU)	0.85	0.05	1:16	384
Ryzen 4650U (CPU)	0.03	0.03	1:1	6

Figure 3.2: Comparison of computational performance of several hardware solutions on single (32-bit) and double (64-bit) precision floating point numbers in trillion operations per second (TFLOPS) [17].

Dedicated hardware such as graphics cards (GPUs) show higher raw floating point performance but as the problem requires high precision for convergence of the optimization procedure, double precision floating point numbers will have to be used. Modern GPUs have weaker FP64 performance as they only have one processing unit per 64 FP32 units. Even though they show higher performance in general use cases, in this case, their advantage compared to a CPU cluster might not be significant anymore due to limitations about memory layout, data sharing, and caching.

3.3.1 CPU

The most straightforward choice for a computational device is a general-purpose server or desktop CPU. It is the easiest to work with as many development tools are available and have been optimized for efficiency. Code written for this platform can also be easily compiled for different architectures and ported between them as almost all systems share a similar structure.

The downside of this choice is that CPUs are general-purpose machines that are less suited for very specific problems compared to dedicated hardware. Due to their architectural complexity, they often do not have many cores, with typical consumer systems having between 8 and 16, and most server processors being limited to at most 64 cores per node. This can be alleviated a bit by using SIMD, but applying this to complex-valued matrix multiplications is non-trivial and in most cases way slower compared to a highly parallelized dedicated platform.

Memory bandwidth is another important factor that has to be considered. While a CPU usually has access to more memory than a dedicated system, the access speed is usually slower. A typical DDR4 dual-channel configuration will be able to access memory at a speed of 50 GB/s. Using DDR5 hardware improves this to at most 100 GB/s.

On paper, it looks like a CPU is not as capable as a more specific system at solving this problem due to its weaker capabilities. In reality, the task is highly dependent on how fast state vectors can be read and written from and to memory. This means the execution speed is mostly determined by the distribution and amount of L1, L2, and L3 cache on the chip. In most modern CPUs the L3 cache, reading at up to 1 TB/s, is able to store multiple state vectors and is shared by only 2 to 4 cores. As the algorithm developed in chapters 6 and 8 mostly reads data from one state vector and applies it to another, a modern server CPU with more than 2 MiB of L3 cache available per core will be able to execute it with a low amount of cache misses.

A CPU, even though it has less parallelism and memory bandwidth available, may therefore be able to compute the result of the given problem faster if it can exploit its larger

hardware caches well. A server CPU will have an advantage at this as it has more cores with way more cache available to them.

3.3.2 GPU

A prominent choice for large, repetitive, highly parallelizable workloads is using a graphics processing unit (GPU), as it was built for a similar task, performing the same computation for millions of pixels in parallel. It therefore offers many parallel computation units, called shader units, of which modern GPUs have up to 10000 each. Even though they use lower clock speeds, due to their number of parallel threads, their overall computation capability for specialized, highly parallelizable tasks is way higher, as can be seen in figure 3.2. One downside of their specialization is that consumer cards are usually built for low-precision applications such as rendering, which only use single-precision (FP32) floating point numbers. This means the number of available double-precision (FP64) units is significantly lower, in modern GPUs at a ratio of 1:64. As can be seen in figure 3.2, this means that for the quantum optimization problem, which requires high precision, raw computational GPU performance is only one order of magnitude larger than a laptop CPU.

To supply all of those cores with data GPUs have faster memory with higher bandwidths. A modern GPU can read up to 1 TB/s [17] from its memory. As the given problem is very data-intensive, this might help a lot with the computation speed. Due to its performance, memory is smaller than usual system memory. This does not pose an issue for the optimization problem though, as applying layers gate-wise reduces memory requirements considerably. The downside of this is that due to its specific application area, a GPU does not have a significant amount of cache, which is also shared along all of its compute units.

Developing an algorithm for a graphics card is comparably harder to a CPU as it needs to be written in a device-specific language, such as Nvidia's CUDA[15], OpenCL[9] or Vulkan Compute[10]. Additionally, the parallelization and memory layout of a GPU are way more complicated and need to be taken into account. Deploying the software requires specific, cost- and energy-intensive hardware.

3.3.3 TPU

The most performant solution for an algorithmic problem is purpose-built hardware. Building such a circuit or using a (slower) FPGA is more cost- and/or labor-intensive though, and therefore in most situations not feasible. This can be solved by using one of the already existing, readily available dedicated hardware implementations for a different, yet similar problem.

Due to the recent interest in the applications of machine learning in real-world scenarios, a lot of consumer hardware now includes dedicated circuits for evaluating those models. For developers working on the go, specific compute sticks for training are also available which achieve similar results as way more expensive and energy-hungry GPUs. They are called Tensor Processing Units (TPUs) and Neural Processing Units (NPUs), as they mostly perform matrix/tensor multiplications to evaluate the given machine learning model.

As the quantum optimization problem can be reduced to a tensor network contraction, using a TPU to solve this problem seems promising. One issue of this approach is that

those devices target larger tensor contractions. The most efficient way to solve the problem at hand though, as shown in section 3.2, is to apply layers gate-wise. It is unclear how a TPU, which was not built for this specific purpose, can be used to implement this.

Due to complexity and time constraints, the application of TPUs was not investigated and evaluated in this thesis. It might be worth looking into in future work though.

3.3.4 Compute Cluster

No matter which hardware is chosen, due to the significant size of the problem, a single device will likely not be enough to obtain a result in a reasonable time frame. Multiple devices in a compute cluster will therefore have to be used in parallel to compute the desired output. As the problem is trivially parallel, this can be done using just a few lines of MPI[5] or an equivalent framework without having to modify any other part of the existing code base.

In the case of a CPU cluster, due to the heavy use of the on-chip caches when evaluating the optimization problem, using server CPUs is preferred as they not only contain more cores but also have more cache available for each of those cores.

3.3.5 Summary

The problem is mostly cache-bound as can be seen in chapter 7. At the high precision required to solve it, the computational power of dedicated hardware is greatly reduced and does not outperform a CPU by a large margin anymore. Due to this in combination with the scope of this work, the further chapters will focus on using and implementing the developed algorithms on a CPU or CPU cluster. Exploring more specialized, dedicated options will be left to future research.

3.4 Software Platform

The software developed in this thesis targets CPUs on the x86-64 architecture. To achieve the performance and control over memory management required to solve the optimization problem, the algorithms have been implemented natively.

C++ was chosen as the programming language due to its low-level features as well as widespread use and support. It allows writing performance-oriented, memory-efficient code and has lots of optimization and profiling tools available. Its popularity also allows for other developers and researchers to use and interact with the software developed in this work. The C++ compiler of the GNU Compiler Collection[6] was used to compile the code on optimization level three.

3.4.1 Libraries

Due to the low-level, highly specialized nature of the optimization problem and code, only a few libraries have been used. This allows for precisely controlling the flow and processing of data. Due to its simplistic nature and better optimization support by the

compiler, the `complex<>` data type from the standard library has been used for representing complex values.

Parallelization

The optimization problem is trivially parallel and does not require any fine control over the exact distribution of the workload onto the different worker threads. Using a library such as OpenMP[3] is therefore sufficient for multithreading and has been chosen for parallelization in this case. To further distribute the load onto multiple machines in a compute cluster, the best option is probably using MPI[5] or an equivalent framework.

Unit Testing

Unit tests have been created to verify the results of the implementation against either a trivial approximation or the original Python implementation. GoogleTest[13] has been selected as the testing framework for creating and running those test cases.

4 Brickwall Circuit Contraction

4.1 Single Gate Application

As described in section 3.2, applying each layer as a large matrix is too expensive. Layers will therefore be applied gate-wise, applying one at a time sequentially to exploit the compounding effects that exist in the final result.

4.1.1 Principle of Operation

Each 2-qubit gate is described as a 4×4 matrix. To be able to apply this matrix to our larger 2^{16}-entry state vector, we have to fix all other unaffected qubits and perform the operation for each of their possible states. The resulting output state vector $|\psi\rangle^{out}$ of a single gate application of G can be described in terms of its input $|\psi\rangle^{in}$ as:

$$\forall l \in L, \forall s \in S, \forall o \in \{00, 01, 10, 11\} : \quad |\psi\rangle^{out}_{l \oplus o \oplus s} = \sum_{i \in \{00,01,10,11\}} G_{i,o} \, |\psi\rangle^{in}_{l \oplus i \oplus s} \tag{4.1}$$

Here, $L = \{0,1\}^{Q-2-|S|}$ is the set of all fixed larger / more significant qubit indices, and $S = \{0,1\}^{Q-2-|L|}$ analogous for all smaller / less significant qubit indices. $l \oplus o \oplus s$ is therefore the combination of the indices of the upper and lower fixed qubits and the index on the qubits the gate is being applied to, forming an index into the entire state vector. Indices are treated and combined as binary representations of their fixed qubit states. The resulting index therefore has a length of $Q = 16$ bits for a 16-qubit system.

4.1.2 Implementation

Implemented correctly, this mathematical dependency can be achieved by reading each entry of the input state vector and writing each entry of the output state vector just once.

To apply the gate, all indices for all unaffected fixed qubits are generated. They are computed for all more significant qubits and all less significant qubits compared to the application site. The indices are then combined using the logical OR operation and sorted so that the least significant bits are the fastest running. This way, memory indices are as efficient as possible.

For each "other" index that has been generated, the matrix multiplication with the single gate matrix is then performed by obtaining all four inputs and combining them into the four written outputs.

The pseudocode for a single gate application is shown in listing 4.1.

Listing 4.1: Computing the output state vector of a single gate application

```
site: Int
input: ComplexVector[2^16]
output: ComplexVector[2^16]
gate: ComplexMatrix[4x4]


S = site // smaller/lower qubits
L = 16 - 2 - s // larger/upper qubits

for(largerIndex in 0 .. 2^L) {
   for(smallerIndex in 0 .. 2^S) {

      FixedIndex:
      otherIndex = combine(smallerIndex, largerIndex)

      MatMul:
      for(outputIndex in [00, 01, 10, 11]) {
         outputEntry = 0

         for(inputIndex in [00, 01, 10, 11]) {
            gateFactor = gate[outputIndex][inputIndex]
            inputEntry = input[combine(otherIndex, inputIndex)]
            outputEntry += gateFactor * inputEntry
         }

         output[combine(otherIndex, outputIndex)] = outputEntry
      }
   }
}
```

The indexing is generated separately for each group of qubits, "lower", "upper", and "applied to", and is then combined using bit shifts and logical OR operations.

The algorithm presented here would read the same input in line 22 of listing 4.1 for each of the four outputs. In practice, this will be optimized away by a low-level compiler and kept in register or memory as much as possible. In cases where this does not happen the implementation should take care of that.

4.1.3 Hardware Aspects

There are multiple aspects of the hardware the algorithm is being run on that need to be taken into consideration, especially regarding memory accesses. As CPUs usually operate using a cache-based memory architecture that loads multiple bytes, an entire cache line, at once, data accesses should be as sequential as possible to reduce cache misses. This way the highest bandwidth and therefore execution speed is achieved.

Memory access order is non-trivial for this problem as neighboring state vector entries which only differ in a single qubit are a stride of 2^i removed from each other due to the state vector's construction. Performance of the gate application and thereby the entire contraction therefore mostly depends on the indexing that is used to access the entries. The fastest iteration should preferably be done over the least significant bits (LSB) instead of the most significant bits (MSB). This is done by the algorithm in listing 4.1.

In case the gate is applied to the fastest running qubits in the state vector, the four entries loaded at once for the matrix multiplication will be neighboring each other. As they each occupy 16 bytes as double precision complex values, all four will exactly fill the cache line which is 64 bytes in size. Memory accesses in that scenario will be perfectly local, therefore.

When the gate is being applied to some faster running qubits on the other side, accesses won't be perfectly local though. The iteration is performed on the other qubits and therefore on the fastest running index, but each iteration will require four values to perform the matrix multiplication which might be spread out quite a lot. Luckily, this only means that data will be read in four spread-out but individually local streams. As all caches are able to store more than four cache lines, all loaded data will still be consumed and performance should be equivalent to a single sequential read.

The same principle applies to the output state vector that is being written. As data is also stored in the cache until the cache line is flushed this means that data output is also cache-local in four streams and therefore almost as efficient as maximally possible.

The gate matrix also has to be stored. It is very small though, only having 16 entries, and can therefore be permanently held in register or low-level cache.

The algorithm shown in listing 4.1 should be very cache-efficient, due to the locality of the four streams, and probably achieves the maximum performance possible on this architecture without building dedicated hardware.

4.1.4 Results

The algorithm described in section 4.1.2 was implemented natively as part of this work.

Running it for a single gate for 16 qubits takes around 5 Megacycles, 1 ms on a modern laptop. If only a single gate is applied there are 750k cache references, out of which around 7 % are cache misses. If the algorithm is run for 8 layers, 8 gates per layer, and 32 iterations, subsequent applications benefit from each other and the cache miss ratio drops to 4 %. This should be sufficient for efficient execution as part of the larger algorithms designed later on. Branch misses also only make up 0.1 % of all branches, which further drops to 0.04 % if multiple gates are applied sequentially.

In practice, this allows running 66 layers per second ($\frac{L}{s}$) with 8 gates per layer using complex values on a single thread. Assuming the computation of a single gradient takes 67 ML (mega layers), computing the gradient for a single iteration naively would take more than 11 days. When parallelized perfectly onto 12 threads, which is not possible due to cache limitations as seen in chapter 7, this still leaves a run time of at least one day. The Hessian requires this computation not just for each possible hole but also for each combination of those as well as having to track 16 state vectors in parallel in the center as described in chapter 8.

The implementation also contains unit tests to verify the gate application. The native gate-wise code applying multiple layers of gates produces the same numeric results as a naive, NumPy-based approach computing the Kronecker product and multiplying the results. The numerical difference to the layer-matrix forming contraction approach is less than 10^{-13}.

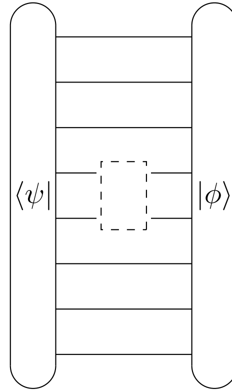


Figure 4.1: The remaining tensor network before the final gate hole contraction.

4.2 Gate Hole Contraction

To obtain the final gradient and Hessian from the contraction required for the Riemannian optimization procedure, we have to contract the two state vectors neighboring the hole that was cut into the network. An algorithm to do this efficiently needs to be developed.

4.2.1 Principle of Operation

After all gates have been contracted, the remaining tensor network will be of the form shown in figure 4.1. The gradient ∇G resulting from the contraction of the left vector $\langle\phi|$ and right vector $|\psi\rangle$ over the hole will be of the form:

$$\nabla G_{o,i} = \sum_{l \in L,\, s \in S} |\psi\rangle_{l \oplus i \oplus s} \, |\phi\rangle_{l \oplus o \oplus s} \tag{4.2}$$

where o is the row, i the column of the gate gradient and L and S the sets of possible other qubit indices that the hole is not located on. The index combination operation $\oplus$ is defined analogously to equation 4.1.

4.2.2 Implementation

The expression from equation 4.2 can be evaluated by iterating over all possible other lower and higher indices. Then, all gate gradient entries can be computed by substituting i and o and using the combined indices to access and take the product of the correct state vector entries in $|\psi\rangle$ and $|\phi\rangle$. The entire algorithm can be seen in listing 4.2.

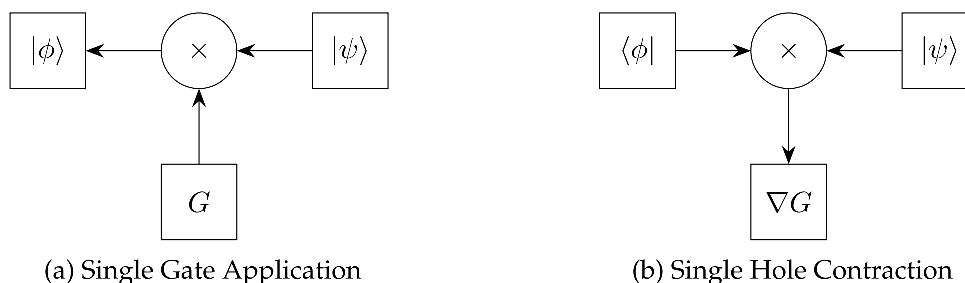
Listing 4.2: Computing the gradient of two state vectors over a hole

```
site: Int
left: ComplexVector[2^16]
right: ComplexVector[2^16]
gradient: ComplexMatrix[4x4]


S = site // smaller/lower qubits
```

```
8  L = 16 - 2 - s // larger/upper qubits
9
10 for(largerIndex in 0 .. 2^L) {
11    for(smallerIndex in 0 .. 2^S) {
12
13       FixedIndex:
14       otherIndex = combine(smallerIndex, largerIndex)
15
16       Contraction:
17       for(leftIndex in [00, 01, 10, 11]) {
18          for(rightIndex in [00, 01, 10, 11]) {
19             leftEntry = left[combine(otherIndex, leftIndex)]
20             rightEntry = right[combine(otherIndex, rightIndex)]
21
22             gradient[leftIndex][rightIndex] += leftEntry * rightEntry
23          }
24       }
25    }
26 }
```

This algorithm is very similar to the gate application shown in listing 4.1. It generates the same indexing for unaffected qubits and then proceeds to iterate over the gate matrix or its gradient. Apart from the actual read, write, and arithmetical operations it is identical. This is due to both algorithms performing a tensor contraction on the same structure with different inputs and outputs. Additionally, due to open legs in both scenarios, the summation dimension is different, summing over the outer loops (the unaffected qubits) instead of the gate row of the matrix multiplication. The difference in data flow in the inner operation is visualized in figure 4.2.



(a) Single Gate Application

(b) Single Hole Contraction

Figure 4.2: Apart from the data flow in the inner operation, the indexing is identical.

4.2.3 Performance

The performance of this procedure is not as critical as that of the gate application from section 4.1 as it is only applied to the final "gate" compared to all gates. Optimizing it is still desirable though to prevent it from bottlenecking the rest of the computation in case those can be executed significantly faster.

Due to the structural similarity of the two algorithms, the efficiency of this computation is comparable to the one discussed in section 4.1.3. The reading of data is not strictly local either, but due to the same indexing, the two input state vectors are also being read in four local streams each. This means two state vectors will be read in parallel but this

should not have any effect on cache performance. Data is being written into the gate gradient matrix which is small enough to fit into cache permanently.

The performance of this operation should be almost identical to that of gate application (section 4.1) which is performed significantly more often and it is therefore sufficient for the overall optimization procedure.

The native hole contraction code was verified for correctness using unit tests. These compare the result of the algorithm to manually constructed symmetric and non-symmetric 3-qubit test cases.

4.3 Indexing

The algorithms shown and discussed in this thesis assume the qubits to which the gates are applied to be neighboring. In the case of a brickwall circuit wrapping around the boundaries or the 2-dimensional case, this assumption breaks down though. To allow for those cases and to enable generalizability, the application of gates to and contraction of arbitrary qubit pairs needs to be supported. This can be achieved using boolean logic as well as bit shifts to determine the correct indices in the algorithms from this chapter. Apart from the indexing the methods stay the same and performance remains about equivalent. This also needs to be taken into account for the structure of the caching described in chapters 6 and 8. Their ordering of the cached state vectors for each gate needs to be adapted but apart from that their core principle remains analogous.

4.4 Matrix-free Contraction

Compared to the original Python implementation[14] which is able to apply $5.3\frac{kL}{s}$ (12-L) this new implementation runs at $9.1\frac{kL}{s}$ (12-L) when fully parallelized. As the non-gate-wise approach is incapable of performing the calculation for 16-qubit systems due to memory limitations, these tests were performed using just 12 qubits. This performance difference does not scale linearly with the number of qubits, therefore a more significant advantage for the matrix-free approach is expected on a 16-qubit system.

As the Python implementation is forming the entire layer matrix and then just executing NumPy[12] the matrix multiplication is already parallelized. Additionally, due to it contracting the matrices first and only then applying the different state vectors, it already benefits from a caching-like effect. The new implementation is not exploiting that redundancy in the contraction evaluation yet. Caching data to reduce redundancy and computational effort will be investigated in chapters 6 and 8.

During the evaluation, the new application uses 100 MB of memory for all threads combined. The Python code is using up to 1 GB. This factor of 10 will also scale with qubit count up to 72 GB for the full-matrix implementation at 16 qubits.

The matrix-free gate-wise application is therefore not significantly faster than forming the entire matrix yet but consumes way less memory in the process making it suitable for larger systems. Additionally, it still has a lot of potential for optimization using caching and other techniques to reduce redundant computation. While it on its own is not capable of making the optimization method realistically feasible and more improvements are therefore needed, it is an integral part to solving the optimization problem.

5 Problem and Performance Analysis

5.1 Metrics

Without optimization the entire problem is too large to be executed and for its run time to be monitored during the optimization process. Therefore, metrics are required that allow measuring and quantifying the amount of work done and the processing speed to allow for efficient development and evaluation of the optimized algorithm. Several metrics were developed and used during this work to achieve this.

5.1.1 Baseline System

Any measurement system for speed requires a baseline for time. Most metrics used here are based on real time when running on a constant system. The system used in this work was a Laptop with an AMD Ryzen 4650U processor and 32 GB of DDR4 memory. It has six physical cores running twelve hardware threads through hyperthreading. The parallelization speedup is limited by this to a factor of at most six or twelve. The L3 cache has a size of 8 MB, with 4 MB being shared on each island of three cores.

5.1.2 Absolute Time

The easiest approximation of performance can be obtained by measuring the execution time of the program. This can become tricky though if the run time exceeds a realistic time frame or the memory requirement of a to-be-optimized baseline implementation overflows the available system memory.

The estimates given in this thesis for absolute execution time are based on measuring a tiny sample workload and then extrapolating to the larger total workload to get an approximate result. This process should only be used for looking at the order of magnitude though, as a lot of memory efficiency speedups as well as parallel scaling to distributed systems are hard to correctly estimate. The big advantage of this approach is that it yields immediate feedback about the current feasibility of actually executing the entire optimization procedure. It therefore gives an easy-to-grasp estimation of the requirements on still missing optimizations during the development process and the speedup they have to achieve to enable that goal.

5.1.3 Original Layers Processed

The metric most frequently used in this work is layers per second ($\frac{L}{s}$). It provides a direct way to compare the performance of two different versions of the algorithm during development and an estimation of the exact speedup one of them achieves over the other. This helps guide development and optimization.

This metric is based on the most simple implementation of the optimization method. It assumes evaluating one circuit takes as many layers as the circuit has, with each layer having usually six or eight gates. This means that it regards computing a single gradient as 2^Q evaluations of the circuit and therefore $2^{Q+3}L$ at 8 layers. Evaluating a gradient therefore will consume the same amount of processing power for each possible hole, computing the Hessian analogous for each combination of holes. This mimics a naive implementation which is used as a baseline and just contracts the entire tensor network for each state vector and hole.

In reality, more optimized algorithms will use caching and different paths of execution to achieve significantly higher performance on the problem. They therefore reduce the effective number of circuits that need to be evaluated. Using this metric, their performance is specified by the number of layers that the naive implementation would have taken to achieve the same result, even though the better algorithm did not actually compute all of it. In those contexts, the notion of equivalent layers per second ($eq\frac{L}{s}$) will be used.

Naively computing a single gradient requires obtaining the gradient of each gate and therefore hole. In the default case, there are 64 holes in a 16-qubit, 8-layer circuit and $2^{16} \approx 65000$ 1-hot vectors and therefore circuits to be contracted, each containing 8-layers. The number of layers that would have to be evaluated is $64 \cdot 2^{16} \cdot 8L = 2^{26}L \approx 70ML$ in that case. An algorithm that solves the same problem in $700ks$, even when in reality applying fewer layers, is assigned a performance metric of $\frac{70ML}{700ks} = 100k\frac{L}{s}$.

Qubit Scaling

One problem with this metric is that it is based on the assumption that the workload of a layer is constant. This might not be the case as running 16-qubit systems is desirable for monitoring the performance evolution during the development process. On the other hand, for drawing a comparison to the Python implementation, a test using only 12 qubits is required as the original implementation is incapable of computing 16-qubit systems due to its memory requirements. In those cases the metric will be marked with "12-L" to indicate the smaller 12 qubit layers that are faster to process and the full version will be marked with "16-L". Scaling from 12 to 16 qubits might not cause a linear performance change between implementations though but more optimized versions usually perform way better on larger systems than their naive counterparts when being scaled up.

5.2 Problem Size

The given optimization problem consists of a lot of operations that have to be performed on top of each other for all other possibilities. To get a feeling for the scale the algorithm is operating on this section will give an overview of the structure and computational complexity of the problem at hand.

5.2.1 Problem Structure

The structure of the computation performed by a naive implementation determining only the gradient of a single iteration can be seen in the following illustration:

for all layers with hole	$\to$ gradient of all layers	$\cdot 8$	2^{45}
for all holes in a layer	$\to$ gradient of layer	$\cdot 8$	2^{42}
for all 1-hot statevecs	$\to$ gradient of one hole	$\cdot 2^{16}$	2^{39}
for all layers	$\to$ circuit applied	$\cdot 8$	2^{23}
for gates in layer	$\to$ layer applied	$\cdot 8$	2^{20}
for every other state	$\to$ single gate applied	$\cdot 2^{14}$	2^{17}
for every output state	$\to$ four statevec entries	$+4$	2^{3}
read 4 inputs + combine	$\to$ one statevec entry	4	2^{2}

It shows the individual, stacked loops that need to be iterated over to solve a part of the problem as well as their respective outputs. Each iteration level scales the problem by its own factor yielding an overall order of magnitude of $2^{45} \approx 30$ trillion relative to the number of reads and writes performed.

The result of all those operations is only the gradient for a single iteration of the Riemannian optimization procedure. Solving the entire problem would require doing this for every one of the 64 second derivative holes (yielding the Hessian) as well. Additionally, 16 state vectors have to be computed in parallel between the two holes. Then this process is repeated about 100 times obtaining the gradient and hessian for each step of the optimization method. This results in an estimated number of reads and writes to be performed in the range of $2^{45} \cdot 64 \cdot 16 \cdot 100 \approx 2^{63}$. This is equivalent to 10 Exa operations and shows that significant improvements will have to be developed to solve a computational problem of this size.

5.2.2 Memory Requirement

Those 10 quintillion memory accesses to 16-byte complex-valued entries would therefore cause memory read and write operations totaling around 150 EB (ExaByte), almost filling the entire 64-bit address space.

A CPU using DDR4 memory will be able to access memory at 25 GB/s, with DRR5 at 50 GB/s. Even four graphics cards working in parallel, each using GDDR6X memory will only be able to read and write at 700 GB/s. At those speeds, accessing 0.5 PB, representing a gradient computation will take 15 minutes. The total amount of 150 EB will take more than 2000 days. A more memory-efficient solution is therefore needed.

5.2.3 Complexity

Looking at the proposed single gate application naively yields an advantage due to no quadratic scaling in the qubit size as only state vectors are being scaled, not matrices. The problem has a terrible complexity class regarding the number of input qubits. Using 8 qubits yields a $2^8 \times 2^8$ matrix with ≈ 65000 entries. Increasing this just to 12 yields an increase in matrix size to 16 million entries, a factor of 256 for a 50% increase in input size. Doing the same again to get to 16 qubits results in 4 billion entries. Doing the same to the state vector used by the gate-wise application, the factor is only 16, going from 256 entries at 8 qubits, to 4096 and finally 65000 on 12 and 16 qubits respectively.

While this is substantially better than the matrix multiplication ($\in O((2^n)^2)$), the scaling is still exponential with a base of 2 instead of 4. An approximate scale of the approach is given by:

$$A \cdot 2^n = L \cdot \frac{gates}{layer} \cdot 2^n = 2^3 \cdot \frac{n}{2} \cdot 2^n \sim n \cdot 2^n \tag{5.1}$$

where A is the number of applications of gates onto the state vector, L the number of layers and n the number of qubits. This complexity class of $n \cdot 2^n < (2^n)^2$ is still better than the matrix multiplication, but not as much as would be expected by the naive estimation of $2^n << (2^n)^2$.

The difference in complexity between the two methods therefore is not as expected linear vs quadratic. Both methods are exponential, just with different bases being 2 or 4. A comparison of the scaling of those complexities can be seen in figure 5.1.

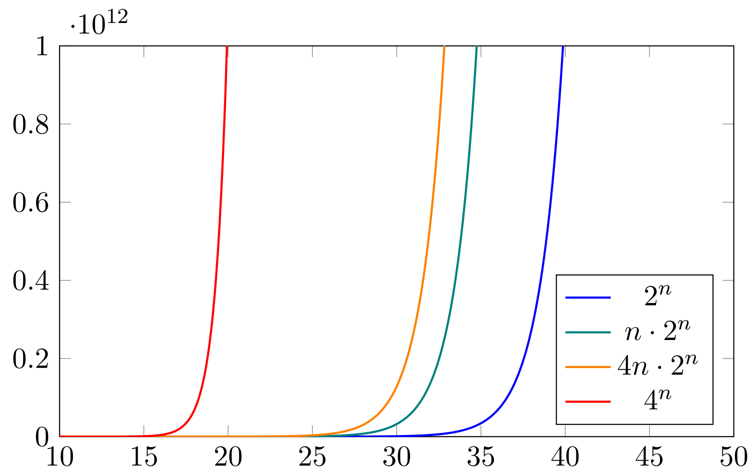


Figure 5.1: The scaling of the different complexity classes relative to the Terra (10^{12}) order of magnitude.

As can be seen in the figure, the difference between the classes is not as pronounced as one could hope for. It instead manifests itself as a shift in achievable qubit count to the right until the better algorithm also exceeds the realm of the physically possible.

The new, single-gate application based approach represented by the plot of $4n \cdot 2^n$ therefore does not provide an opportunity for arbitrary larger qubit systems but it moves the problem of running the optimization procedure on 16 qubits from the impossible into the realm of barely possible.

How it is possible to use this single-gate approach to run the entire optimization method on current hardware in a realistic time frame will be explored in the following chapters.

6 Gradient Computation

The Riemannian optimization procedure for the circuit works by building a quadratic approximation of the target function in the neighborhood of the current gate parameters. This step requires obtaining the gradient and Hessian of said function relative to the gate parameters. This is achieved by contracting the tensor network representation as described in chapter 2. This chapter focuses on optimizing the computational process for that contraction.

6.1 Problem Statement

The problem of obtaining the gradient can be reduced to a contraction of the tensor network shown in figure 6.1. This process is repeated for each possible hole and yields the gradient for the gate that would be located where the hole is. Combining all gradients in a layer leads to the gradient of the gate, repeating this process for all layers returns the gradients of all gates for the state vector, and therefore by repeating for and summing over all 1-hot vectors, the gradient of all parameters of the circuit.

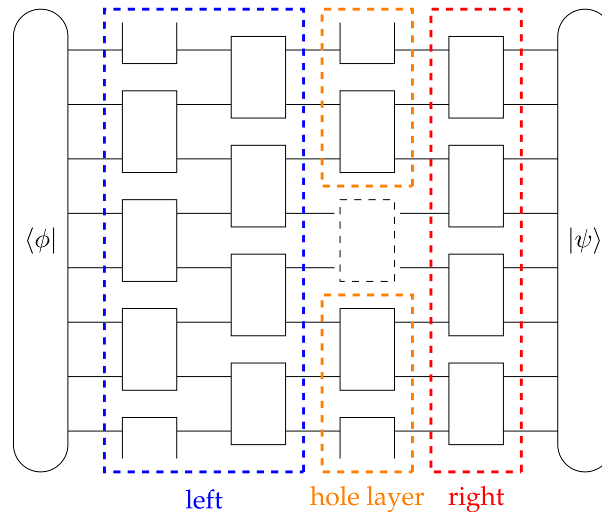


Figure 6.1: The tensor network to be contracted for the gradient computation.

The algorithms developed in this chapter compute the contraction of two state vectors with a brickwall circuit, omitting the application of the unitary time evolution operator. As the operator is just combined with a 1-hot state vector, the left input state vector to the brickwall part can be easily determined by picking a single row from the matrix.

6.2 Naive Implementation

The process of computing the gradient of a single hole naively can be broken up into multiple steps. Those are shown in figure 6.1. First, all complete layers not affected by the hole are contracted. This can be done by contracting with the state vector and applying layer by layer gate-wise from the left and right until the layer containing the hole is reached. The contraction from the left uses the inverse of the gate matrices. Those are computed and stored in advance to improve efficiency. Then, all gates still present in the hole layer are applied, as they affect different qubits and are therefore independent, their application order can be chosen arbitrarily. In the end, the two remaining state vectors on both sides are contracted over the hole using the procedure described in section 4.2.

The result of this process is a 4×4 gradient matrix. Summing this along the layer and iterating over all layers gives the gradient of the entire circuit for a single 1-hot vector. This is then repeated for each 1-hot state vector to obtain the total gradient of each of the circuit's parameters.

The following pseudocode illustrates the structure of the algorithm:

Listing 6.1: Naive algorithm computing the full gradient of a circuit

```
Q: int // qubits
gates: ComplexMatrix[8][4x4] // input
gradients: ComplexMatrix[8][4x4] // output

L = gradients.size
gatesPerLayer = Q / 2

gradients = 0

for(oneHotIndex in 0 .. 2^Q) {
   for(layer in 0 .. L) {
      for(hole in 0 .. gatesPerLayer) {
         left, right: ComplexVector[2^Q]

         initWithOneHot(right)
         initWithMatrixRow(left)

         // right
         for(l in 0 .. layer) {
            contractLayer(right, gates[l])
         }

         // left
         for(l in (L - 1) .. layer) {
            contractLayer(left, gates[l])
         }

         // center
         contractLayer(right, gate[layer], exclude: hole)

         gradients[layer] += contractHole(left, right, hole)
      }
   }
}
```

An optimized version of this algorithm has been natively implemented using C++. Parallelization is done over the outer loop on the 1-hot state vectors, the results are summed. The algorithm can be executed on a laptop with six cores in 15 hours. This results in a total runtime for the naive Hessian calculation of about 30 days per iteration, not taking into account a factor of 16 due to the state vector splitting between the holes. At 100 iterations, some improvements are needed to make the computation realistically possible.

6.3 Cached Implementation

In the naive implementation, the entire circuit is contracted for each hole. Apart from its length, the computational path taken by both state vectors until they end up at the hole is always identical though. There is therefore a lot of duplicate computation for the intermediate results after applying each layer, which is done repeatedly for each hole. To avoid unnecessary work and improve performance this redundancy should therefore be eliminated as much as possible.

This section describes an algorithm storing intermediate results to speed up computation. This concept will be referred to as caching. To distinguish, the CPU's physical L1, L2, and L3 caches will be referred to as hardware cache.

There are two ways to reduce redundancy. One option is reordering the operations so that intermediate results are used for multiple operations. The other is caching the intermediate results in advance or during the computation and accessing them during later steps. This optimization method will focus on the solution using caching and will try to reduce the amount of stored and accessed data for efficiency.

6.3.1 Algorithm

Observing the algorithm's contraction steps, it can be easily seen that the contraction process from the left and right to the hole is always the same apart from its final stopping point. The intermediate state vectors are computed over and over again. Optimally, those intermediate results should be cached.

One idea that comes to mind for achieving this is to split the process into two steps, cache generation and evaluation of each individual hole. Building the cache in advance and then just retrieving the required data for each hole is desirable compared to dynamically caching during the gradient computation due to cache management overhead and complicated lookups. As each hole just needs information about its neighboring state vectors after contraction, this can be done efficiently by doing just two cache passes through the brickwall circuit, one forward and one backward, and storing the intermediate state vector after each layer application. Afterward, the gradient of each hole can be easily computed by picking the correct state vectors from the cache and applying the missing layer. Then, the hole contraction can be performed as usual. This way, the computation of 7 out of 8 layers for each of the 64 holes is replaced by evaluating 16 layers once in advance.

The cache generation procedure and indexing can be seen in figure 6.2. The index of the layer and hole site in the cache always equals the indices of the next gate that would have to be applied to the stored state vector in sequential order.

This approach should already yield a substantial performance improvement, it does still

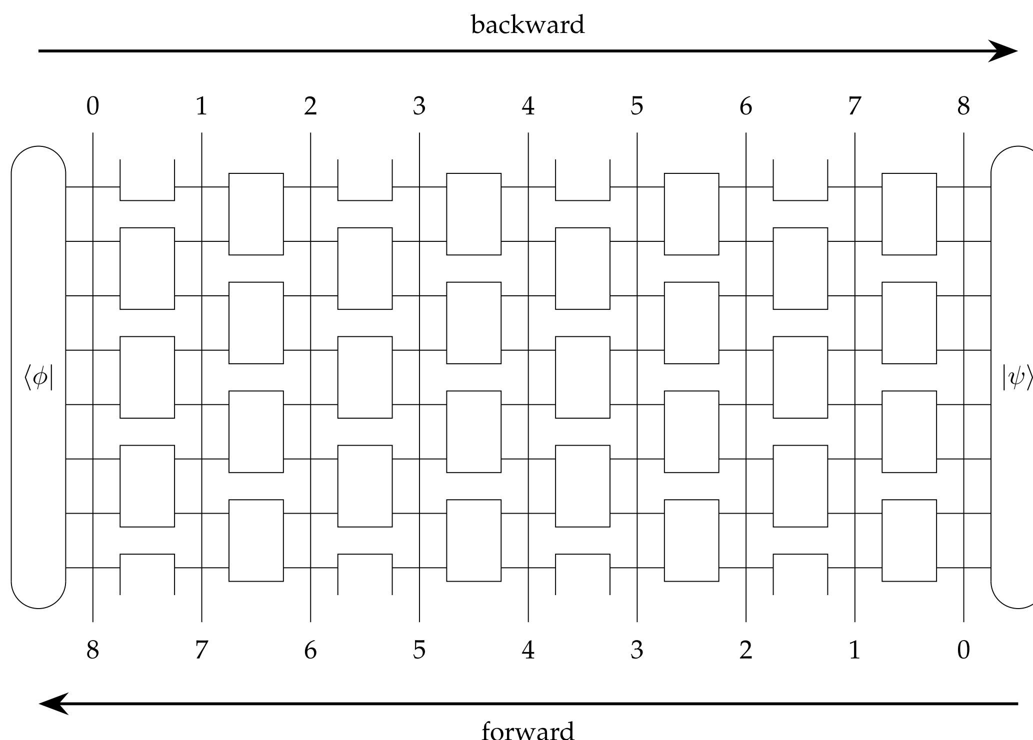


Figure 6.2: The indexing of the two cache generation passes when applied at a layer level. In this case, the last layers in both directions could be skipped, in the site-wise caching the speedup gained by that is negligible.

perform quite a lot of unnecessary computation though. For all the holes in one layer, the process of evaluating that last layer is pretty similar. Instead of caching state vectors layer-wise, they can also be stored site-wise for every application site in the layer. This way, in exchange for higher memory usage, even more performance is achieved. Each hole can now simply access its two neighboring state vectors from the cache and perform its hole contraction without having to apply any more gates.

Listing 6.2: Efficient algorithm using caching to compute the full gradient of a circuit

```
Q: int // qubits
gates: ComplexMatrix[8][4x4] // input
gradients: ComplexMatrix[8][4x4] // output

L = gradients.size
gatesPerLayer = Q / 2

gradients = 0

for(oneHotIndex in 0 .. 2^Q) {

   // Generate cache
   // run circuit both ways, store one statevec per gate
   cache = {
      right: applyRightAndStoreEachGate(oneHotIndex),
      left: applyLeftAndStoreEachGate(oneHotIndex)
   }

```

```
19    // Compute gradients for each hole
20    for(layer in 0 .. L) {
21       for(hole in 0 .. gatesPerLayer) {
22 
23          right = cache.right[layer][hole]
24          left = cache.left[layer][hole]
25 
26          gradients[layer] += contractHole(left, right, hole)
27       }
28    }
29 }
```

6.3.2 Deep Learning and Backpropagation

As described in section 6.3 an alternative approach to caching is to reorder the computational process and to perform parts of the computation on the fly. This can be achieved by only caching the forward pass and then performing contractions with the respective forward cache entry on the fly while running the backward pass. This reduces memory usage and potentially access in exchange for algorithmic complexity.

The procedure is quite similar to the problem of training deep learning models in machine learning. This is done using an algorithm called backpropagation. The model is evaluated in a forward pass, storing all intermediate results on the way. Then, originating from the target function, a backward pass is performed, computing the gradient of each parameter on the fly using the stored result from the forward pass in conjunction with the chain rule. The resulting gradient for model parameters is then utilized to optimize for the target function using gradient descent.

The optimization problem on the quantum circuit is structurally similar due it evaluating a tensor network in two directions and then using the obtained gradient for the optimization of a target function as well. The main difference here is that in the brickwall circuit case, the optimization method is more elaborate. It uses a quadratic approximation based on gradient and Hessian for the Riemannian trust-region algorithm[14, eq. 13] instead of the linear approximation utilized by a gradient descent based approach.

Another significant difference between the two problems is the amount of model or circuit parameters. Deep Learning models usually start at hundreds of thousands of parameters and can reach up to a few hundred million or even billion in modern large language models. The tensor network representing the quantum circuit $W(G_1, G_2, \ldots G_l)$ in contrast only has 16 parameters per layer as the same gate matrix is shared by all gates on a layer. This leads to 128 optimizable parameters in total.

The quantum optimization problem is a very specific, restricted version of the training problem solved by Deep Learning methods. Due to this, the algorithms developed here are quite different as they exploit the special structure of the repeated gate application to two qubits at a time to achieve a significant advantage in performance.

One inspiration that can be drawn from the Deep Learning view and be applied to the problem being investigated is the on-the-fly computational characteristic of the backward pass. As mentioned at the start of the section, the current algorithm can be modified to compute the gradients during the backward pass, therefore avoiding having to store the immediate state vectors in one of the two passes.

The downside of this approach is that during the hole contraction, which is now performed during the backward pass, the state vector needed for continuing afterward might be flushed out of the cache. During the contraction, two 1 MB 2^{16}-entry state vectors are read fully and a resulting 4×4-entry gradient matrix is written. As the backward pass itself needs two 1 MB state vector buffers for the gate application, this exceeds the 2 MB threshold per core. This might cause the state vector of the gate application pass to be written out to memory during the final contraction and to be read in again afterward. This memory access is equivalent in amount to the previous approach performing and caching two passes in advance and reading them back in later on. The following pseudocode example illustrates this issue:

Listing 6.3: Computing the gradients during the backward pass

```
generateForwardCache()

while( backwardPass() ) {
   getForward()
   contractHole() // flushes cache
   readBackwardFromMemory()
}
```

Listing 6.4: Caching forward and backward pass and computing gradients later

```
generateForwardCache()
generateBackwardCache()

for(hole in gates) {
   getForward()
   getBackward()
   contractHole()
}
```

As can be seen, on CPUs with small caches, the backward state vector will be flushed out of the L3 cache during `contractHole()`. The number of forward and backward state vector reads is then the same in both versions. Due to this requirement in memory actively used during the entire pass, this solution is only really viable on systems with large L3 caches such as server processors.

If there is enough cache available, the `readBackwardFromMemory()` call in listing 6.3 will be eliminated yielding a performance improvement over the fully cached version. As the `getForward()` call still has to be performed though, the speedup from this optimization is limited to < 2.

No other possible exploitation of the similarity between this optimization problem and Deep Learning was discovered during this work due to the higher optimization potential when using more efficient algorithms on the structurally more restricted problem. Drawing inspiration from backpropagation and therefore performing the gradient computation on the fly during the backward pass might be useful for a performance improvement, but will not yield an order of magnitude due to the speedup being limited to up to 2.0.

6.4 Memory Requirements

Lower memory usage is one of the main targets of the implementation developed in this work. The original Python-based implementation by Kotil et al.[14] constructs the full matrix representation for each layer using the Kronecker product. For 16 qubits those matrices would have a size of 68 GB each. To be able to apply the method to larger 2D systems as well, one of the goals was to reduce this memory footprint.

With 16 qubits and therefore $2^{16} = 65536$ state vector entries each using 16 bytes for a double precision complex value, each state vector will have a size of exactly 1 MB. As there are two buffers needed for input and output during the gate application, the normal contraction process actively uses 2 MB. With layer-wise caching doing two passes, each containing 8 layers, 16 MB will therefore be stored in the cache. Parallelizing over the 1-hot state vectors on 12 threads will therefore consume around 200 MB.

When using site-wise caching for even more efficiency, a state vector is stored on both sides of each gate. With 64 gates this yields 128 MB used for the cache per thread. With 12 threads this is equivalent to 1536 MB. In reality, a memory allocation of 1.6 GB can be observed, accounting for additional overhead due to computational buffers, the gate matrices, and the gradients.

Using the backpropagation-inspired approach of only caching the forward pass in advance and doing the backward pass on the fly as described in section 6.3.2 would cut this resource usage in half. But even with caches for the forward and backward pass, this implementation still uses significantly less memory than the original matrix multiplication-based implementation and is suitable for real-world use.

6.5 Evaluation

Performing the layer application gate-wise is significantly faster than constructing the matrix representation of the entire layer. Additionally, an improvement in performance of orders of magnitude is achieved by combining this with caching and parallelization. This section tries to estimate the speedup compared with the original implementation.

On the laptop described in chapter 3, the original program achieves a performance of around $5k\frac{L}{s}$ (12-L) according to the metrics described in section 5.1. The gate application on its own and therefore the naive implementation from section 6.2 runs at $2k\frac{L}{s}$ (12-L) but still has a lot of potential in parallelization and redundancy that is already being exploited by the full matrix multiplication. In parallel, the gate-wise application can execute $9k\frac{L}{s}$ (12-L).

Layer-wise Caching

Using layer-wise caching on its own, a performance of $14eq\,k\frac{L}{s}$ (12-L) is achieved. In parallel, $56eq\,k\frac{L}{s}$ (12-L) can be reached. For 12-qubit systems, this yields a speedup of 6.3. On a 16-qubit system as in the target problem, the parallel, cached performance is $2.7eq\,k\frac{L}{s}$ (**16-L**). That implies that the computation of an entire gradient would take around 3.5 hours, a Hessian would take around 140 days. The estimated run time of 100 iterations of the optimization algorithm without further improvements would therefore be about 40 years. The original Python implementation can only be estimated as it cannot

be executed for 16 qubits due to memory requirements. Its approximate computation time would be within the range of 500 to 1000 years.

The naive algorithm applies 8 layers for 64 holes for a total of $8 \cdot 64 = 512$ layers applied. The layer-wise caching implementation contracts a layer for each hole and an additional 16 cached layers in advance yielding a total of $64 + 16 = 80$ layers. The difference in workload between those two computation paths gives a theoretical maximum speedup of $\frac{512}{80} \approx 6.4$. This upper bound is almost realized by the actual program at 6.3 suggesting that the implementation is as efficient on actual hardware as it could possibly be given the theoretical algorithm.

Site-wise Caching

The performance gains from caching can be further improved by caching the state vectors at each application site as described in section 6.3. The site-wise cached implementation obtains the same result as the layer-wise and naive versions. It does so while requiring significantly fewer gates to be applied. The following table shows the actual run times of various tasks executed on the three implementations.

qubits, layers, vectors	non cached	layer caching	site caching
16 Q, 8 L, 1 vec	5.2 s	0.81 s	0.34 s
8 Q, 8 L, All vecs	1.3 s	0.23 s	0.12 s
10 Q, 8 L, All vecs	32 s	5.0 s	2.0 s
12 Q, 8 L, All vecs	750 s	110 s	43 s

When using multithreading the runtime of 12 qubits, 8 layers for all state vectors shrinks even further from 43 s to 9.9 s. Executing the equivalent of $1.6ML$ (12-L) in this time frame is equal to a parallel performance of $159eq\,k\frac{L}{s}$ (12-L). The resulting speedup over the layer-wise version at $56eq\,k\frac{L}{s}$ (12-L) is 2.8 for a 12-qubit system. At 16 qubits the performance is $4.8eq\,k\frac{L}{s}$ **(16-L)** at a speedup of just 1.8.

This reduces the required time for the computation of a gradient to 2 hours. The total run time of the entire optimization procedure is reduced to 23 years compared with the layer-wise version.

The maximum theoretical speedups from the caching-based optimizations can be estimated based on the number of gates that have to be applied. There are eight gates per 16-qubit layer so the naive implementation will be evaluating $8 \cdot 64 \cdot 8 = 4096$ gate contractions. The layer-wise cached approach replaces most layers with 16 layers cached in advance giving $(64 + 16) \cdot 8 = 640$ gate applications. This results in a speedup of 6.4 as seen before. Using site-wise caching optimizes this again to $64 + 16 \cdot 8 = 192$ gates. To maximum possible speedup for site-wise caching compared to layer-wise is therefore $\frac{640}{192} \approx 3.3$.

Interestingly, this speedup is not reached by the implementation on 12 qubits which is only able to reach 2.8. In the case of 16 qubits, the speedup observed shrinks to 1.8. This is related to hardware restrictions due to parallelization and will be investigated in detail in chapter 7.

Summary

The gradient computation procedure described in this chapter was verified using unit testing. The test cases employ finite difference approximations to obtain an estimate of the real gradient and compare this to the result computed by the optimized version.

A comparison of the layers per second performance of all previous versions of the algorithm can be seen in figure 6.3. The caching methods developed in this chapter speed up the execution time of the gradient and therefore optimization procedure significantly. The theoretical speedup of the caching process is $\frac{4096}{192} \approx 21.3$, in reality, 17 to 19 was observed.

Version	Performance (kL/s) (12-L)
Original Python, parallel	5
Naive	2
Naive, parallel	9
Layer-wise	14
Layer-wise, parallel	56
Site-wise	37
Site-wise, parallel	159

Figure 6.3: Performance comparison of the different versions of the gradient algorithm.

Compared to the original Python implementation the speedup of gate-wise application, caching, and parallelization is even more significant and the current implementation of the gradient computation determines the correct result around 30 times faster.

7 Parallelization

The clock speed of processors used to grow exponentially with them being able to process more and more instructions on a single core each year. As this trend has largely come to a halt, with only minor improvements to the number of instructions per cycle being made, the focus has shifted on building parallelized chips. Those use the still growing number of transistors on scaling the number of cores that compute in parallel instead of drastically improving the individual cores. In the realms of dedicated computation like graphics and machine learning, highly parallel hardware accelerators are available that have thousands of cores.

Given the large problem size of the quantum circuit optimization method, multiple systems working together in a computer cluster, each with multiple cores themselves, all working in parallel, will have to be used to solve the problem. Parallelizing the algorithms as efficiently as possible has therefore been a major focus of this thesis.

7.1 Parallel Speedup

The problem consists of computing tensor network contractions for each possible 1-hot state vector to obtain the overall gradient and Hessian for all circuit parameters. As those computations are each expensive and independent of each other the problem becomes trivally-parallel by distributing batches over cores and systems along the outermost iteration loop, the 1-hot vectors. At the end of each full gradient and Hessian computation, and therefore only once per Riemannian optimization iteration, the results are collected and combined by summation.

The issue with simple multiple-core parallelization is not posed by the problem itself but instead by the amount of data it requires and the way it is accessed. The algorithms have to constantly read and write the same data in very distributed strides across the memory. The bottleneck on computational speed and parallelization is therefore not just dependent on the speed and number of cores but also mainly on the cache architecture, size, and bandwidth.

The following table shows the speedup of parallelizing the full (all 1-hot state vectors) cached gradient algorithm from chapter 6 for 12 qubits and 8 layers.

Threads	Time	Speedup
1	113 s	1
2	60 s	1.9
4	36 s	3.1
6	27 s	4.2
12	28 s	4.0
48	29 s	3.9

The investigation in this chapter is based on a laptop CPU with 6 cores and 12 threads.

Therefore, any speedup larger than 12 is only obtainable in computation-low cases and would not be observed in this work.

As can be seen, the speedup due to parallelization is already imperfect at 4 threads. It does not scale any further than ≈ 4 when using all 6 threads. It does not seem to benefit from hyper-threading at all. Any larger thread count reduces performance likely due to scheduling and memory/cache access overhead.

The sample above used 12 qubits and achieved a speedup of 4 due to parallelization. Looking back at section 6.5, the change from layer-wise to site-wise caching yielded a lesser speedup on 16 qubits compared to 12 qubits. The following table illustrates the various runtimes of the algorithm based on the qubit count it is operating on when run on 12 threads.

Qubits	Speedup
18	3.0
16	3.0
14	3.9
12	4.2
10	5.2

Why does this decrease significantly with larger system sizes compared to the expected speedup of at least 6? The speedup is relative to the same problem with the same size being run non-parallelized. As the scaling of computational power using more cores is linear and therefore constant for a fixed core count, the change in speedup cannot be due to the change in workload of the larger problems. The run time is clearly not bound by computational power but by some other factor.

The main bottleneck for the execution therefore has to be the data access. The algorithm is constantly applying gates for contracting the tensor network. This gate application, as shown in chapter 4, reads a full state vector in, applies a gate to it, and writes out an entire state vector. For each gate application, at least two state vectors are therefore constantly needed and used over and over again. To speed up execution and prevent reading and writing the same addresses from and to memory over and over again CPUs use a layered memory architecture with storage and caches being stacked on top of each other. The smaller the cache, the faster its response time and the higher its bandwidth.

The problem is therefore not just about core count and memory bandwidth, it is mostly about how much of the repeatedly accessed data can fit into the processor's low-level hardware caches. In this case, at a state vector size in the range of megabytes, the limitation is due to the L3 cache inside the CPU.

7.2 Cache Architecture

Most CPUs have 3 levels of caches apart from their registers, memory, and permanent storage, called L1, L2, and L3. In the case of the AMD Ryzen 5 4650U, there are 384 KB of L1 cache, 3 MB of L2 cache, and 8 MB of L3 cache available[16]. This cache is not shared across all cores though. Each core has its own L1 and L2 caches, the L3 caches are shared by multiple cores on each island. This section will explore the exact cache structure of the AMD Renoir 3 architecture and its implications and therefore that of hardware caches in general on the algorithms developed in this work.

AMD Renoir 3 Architecture

The architecture of the AMD Ryzen 5 4650U chip is based on the Zen 2 microarchitecture. Detailed die shots have been published by Flickr user "Fritzchens Fritz" [7] and have been annotated by Twitter user "Nemez" [8].

As can be seen, the base architecture has 8 cores which are separated into two compute clusters. It is worth noting, that the 4650U being investigated here only has six cores and therefore only three out of four cores on each cluster are enabled. Their L1 and L2 caches are contained in the individual cores, but there are two L3 caches with a size of 4 MB each. All four cores of one of the two clusters therefore share an entire L3 cache with each other. All threads on each computer cluster therefore have to share 4 MB fitting at most four state vectors with each other. Given that the network contraction process uses two buffers containing one state vector each as input and output, two parallel computation chains can therefore fit onto one cluster.

With two clusters and 8 MB L3 cache in total, four threads performing the gradient and hessian computation can be running in parallel at most. This theory is confirmed by the observations from the tables earlier in the chapter showing that a speedup of just 3.0 is realized with 16 qubits. If only 4 threads are running, the gains are not fully efficient probably due to efficiency gaps in the algorithm where the cache is not completely used. Running 12 threads at once overfilled the caches, therefore leading to minor performance losses. Using six threads to solve the problem seemed optimal but with the given architecture will not be able to achieve a speedup greater than 4.

Some further investigation into the native implementation revealed some unnecessary copy and memory allocation operations. Removing those improved the speedup from 3.0 to 3.8 when using 16 qubits. This pushes parallelization to the theoretically possible limit due to hardware restrictions on this chip and further improvements will not be possible in this category.

7.3 Summary

The main limiting factor for the parallelization of the algorithms developed in this thesis is therefore the size of the caches on the hardware being used.

One option to improve this constraint would be to reduce the state vector size by switching from double- to single-precision floating point numbers. This reduces precision though and might therefore lead to the optimization procedure not converging and therefore not terminating with a satisfactory result. Reducing precision was therefore considered unsuited as a solution in this work.

The other solution to this problem is to choose hardware having large L3 caches. One option for such a system is using server processors which contain a lot of cores but still more L3 cache per core. This means that all cores can run at full load and still be efficient during the distributed contraction process. At those sizes, using hyper-threading or oversubscribing using more software than hardware threads could yield even more performance gains as the problem is not computationally bound and the CPU therefore not fully loaded. Server processors also contain more L2 cache and could therefore fit an entire state vector at even higher access speeds.

As shown in this chapter, parallelization is an essential part of optimizing the algorithms

for this quantum circuit optimization method and allowing the problem to be solved on real-world hardware. The workload is memory and most importantly cache bound. A computational device such as a server processor containing a sufficiently large L3 cache should therefore be used. The resulting overall speedup of gradient caching and parallelization over the original Python implementation is 30.

8 Hessian Computation

Based on the gradient obtained in chapter 6, a gradient descent could be performed to optimize the gate parameters to approximate the unitary time evolution operator. Instead, Kotil et al.[14] propose to use an optimization procedure based on the Riemannian trust-region algorithm[2]. This additionally uses the Hessian of the target function in conjunction with the gradient to build a local quadratic approximation of said function and thereby improves the accuracy of the optimization. Before each optimization step is performed, the Hessian of the target function and therefore tensor network representation needs to be obtained. This chapter will explore how to do that as efficiently and as fast as possible.

8.1 Problem Statement

As described in chapter 2 the Hessian can be obtained by cutting two holes into the tensor network representation and contracting the circuit. The resulting $(4 \times 4) \times (4 \times 4)$ tensor specifies the second derivative for each combination of parameters of the gates that would be in those holes. The problem network to be contracted can be seen in figure 8.1.

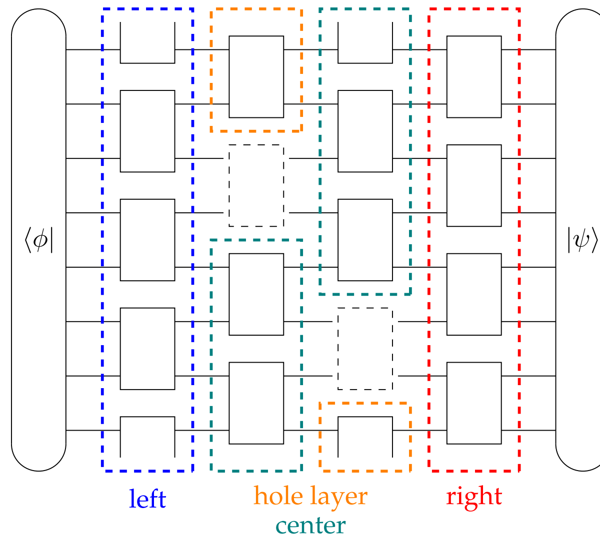


Figure 8.1: The tensor network to be contracted for the Hessian computation.

The straightforward computation for this problem is identical to the gradient from section 6 for the left and right parts. In the center, a state vector needs to be applied for every parameter of the first gate, 16 state vectors therefore need to be contracted there in parallel. This increases the computational workload significantly.

8.2 Workload Estimation

The workload of the Hessian computation is larger than that of the gradient computation as a contraction of the entire network needs to be performed for every combination of gate holes (factor 64) and 16 state vectors need to be applied in the center (factor 8). This results in a workload increase by a factor of approximately 500.

To investigate the exact scale of the workload, a simulation counting the number of layers to be contracted by the gradient and Hessian computations was performed.

	naive	cached
Full Hessian	6700 ML	530 ML
Full Gradient	34 ML	1.6 ML

The maximum theoretical speedup of the gradient computation confirmed by this as already described in chapter 6 is $\frac{34}{1.6} \approx 21.3$. For the Hessian, this works out to $\frac{6700}{530} \approx 12.6$. The naive Hessian method already includes the trivial optimization of mirroring the result to avoid computing swapped holes twice. It is therefore missing a factor 2.

As can be seen from the table, the workload of the Hessian computation is significantly larger than that of the gradient as expected. As both have to be obtained for the Riemannian trust-region algorithm, the gradient's performance sinks in comparison to the potential run time of the Hessian computation. Using the caching algorithm from chapter 6, it should already be fast enough not to be a bottleneck for the Hessian, even if the latter is optimized significantly. The focus should therefore be on improving the efficiency of the Hessian computation. The following sections will investigate how to compute it in the first place and how to reduce redundancies to make it as fast as possible.

8.3 Naive Implementation

The naive implementation of the Hessian is quite similar to that of the gradient from chapter 6. As the problem is the same except for the presence of a second hole, the computation is identical until at first, performing a contraction of the left and right parts until the first hole is encountered. For both sides, the part of the layer before the hole is then applied. In this work, the right one will always be treated as the first hole. As the Hessian is the second derivative relative to all parameters of the two hole gates, a representation of the first hole derivative needs to be obtained that will yield the desired $(4 \times 4) \times (4 \times 4)$ sized second derivative at the other hole. The final contraction will compute the 4×4 gradient for that hole, therefore it needs to be performed 16 times for each possible derivative at the first hole. This can be achieved by representing the first hole as a split effectively applying all 16 1-hot matrices and therefore splitting the input into 16 state vectors.

The main difference to the gradient computation is the presence of a state vector for each parameter of the first gate and therefore 16 state vectors that need to be contracted in the central part between the two holes. At the end, a normal contraction is performed over the second hole for each state vector, yielding the 16 gradients for all parameters of the second gate and therefore the Hessian specifying the second derivative for each combination of parameters of the two holes.

Afterward, the results are combined by summation along the gates on the layers as well as all 1-hot input state vectors. The final result is a Hessian for each combination of gates/layers. The entire procedure can be seen in figure 8.2.

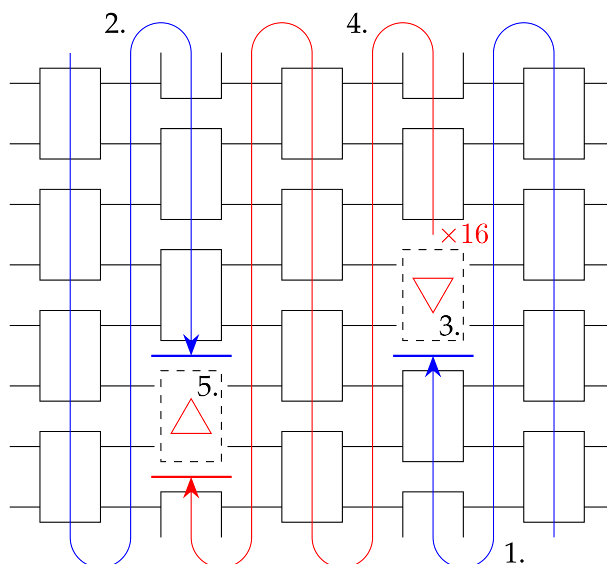


Figure 8.2: Naive Hessian computation. Splitting over a hole is indicated by an upside-down triangle, contraction with 16 state vectors by a normal triangle. Parallel contraction of 16 state vectors is marked using red color. The entire procedure is repeated for each combination of holes.

The following listing shows pseudocode illustrating the structure of the algorithm.

Listing 8.1: Naive algorithm computing the full Hessian of a circuit

```
Q: int // qubits
gates: ComplexMatrix[8][4x4] // input
hessians: ComplexMatrix[8][8][4x4] // output

L = hessians.size
gatesPerLayer = Q / 2

hessians = 0

for(oneHotIndex in 0 .. 2^Q) {
   for(layer1 in 0 .. L) {
      for(hole1 in 0 .. gatesPerLayer) {
         for(layer2 in layer1 .. L) {
            for(hole2 in 0 .. gatesPerLayer) {
               if(layer1 == layer2 && hole2 <= hole1) {
                  skip
               }

               left, right: ComplexVector[2^Q]

               initWithOneHot(right)
               initWithMatrixRow(left)

               // 1: right
```

```
25                 for(l in 0 .. layer1) {
26                    contractLayer(right, gates[l])
27                 }
28                 contractLayer(right, gate[layer1], upTo: hole1)
29 
30                 // 2: left
31                 for(l in (L - 1) .. layer2) {
32                    contractLayer(left, gates[l])
33                 }
34                 contractLayer(left, gate[layer2], downTo: hole2)
35 
36                 // 3: split
37                 vecs = splitOnHole(right, layer1, hole1)
38 
39                 // 4: center
40                 for(vec in vecs) {
41 
42                    contractLayer(vec, gate[layer1], upFrom: hole1)
43                    for(l in (layer1 + 1) .. layer2) {
44                       contractLayer(vec, gates[l])
45                    }
46                    contractLayer(vec, gate[layer2], upTo: hole2)
47 
48                    // 5: contract hole
49                    hessians[layer1][layer2][index(vec)] += contractHole(
                           left, vec, hole2)
50                 }
51              }
52           }
53        }
54     }
55  }
```

The application developed in this work implements this algorithm natively and efficiently. Not parallelized, the gradient computation takes 5 seconds for a single 1-hot state vector on 16 qubits. The Hessian computation on the other hand uses more than 15 minutes for a single state vector. Obtaining the entire Hessian this way would therefore take more than 200 days, the entire optimization procedure more than 50 years, even with parallelization. A more efficient algorithm is clearly needed to realistically solve this problem on current-day hardware.

8.4 Cached Implementation

As with the gradient, there are a lot of redundancies and duplicate computations of intermediate state vectors when using this approach. To optimize the algorithm, these should be avoided.

One strategy to achieve this is to take inspiration from the gradient computation and employ caching to store results that will be reused multiple times. The drawback of this method in the cache of the Hessian is that due to the 2-hole nature and the 16 parallel state vectors in between, a lot of data would have to be stored. Writing and reading this data takes time and reduces performance. Additionally, it significantly increases the memory footprint of the algorithm, especially in multi-threaded scenarios. Therefore, a

hybrid approach using caching and on-the-fly hole contraction was chosen.

8.4.1 Algorithm

First, the cache that will be used in the further computation is built. Using the on-the-fly hybrid approach, cache generation is identical to the one used in the gradient computation. The circuit is run once forward and once backward and all the intermediate results are stored.

Then, the algorithm proceeds by computing all Hessians for all second gate holes, grouped by the first hole. For each first hole, it starts by picking a state vector from the right cache and performing the split through the hole. All of the resulting 16 state vectors are then run to the end of the circuit in parallel. During this process, a hole contraction (for all 16 vectors) is performed on the fly at each possible hole that is encountered. This is achieved in conjunction with reading the state vector on the other side of the potential hole from the left cache. The resulting Hessian for the combination of holes is then saved. The algorithm continues to the next potential hole and the hole contraction and application process is repeated until the end of the circuit is reached.

The algorithm always runs forward from the selected hole and the first hole is therefore always assumed to be the right one, the second the left one. But as the Hessian tensor is symmetric relative to swapping the two gates and therefore holes, after mirroring and summing along the gates on a layer and the 1-hot state vectors, the full Hessian of the tensor network representation and therefore the target function is obtained.

The fundamental structure of the algorithm is shown in figure 8.3.

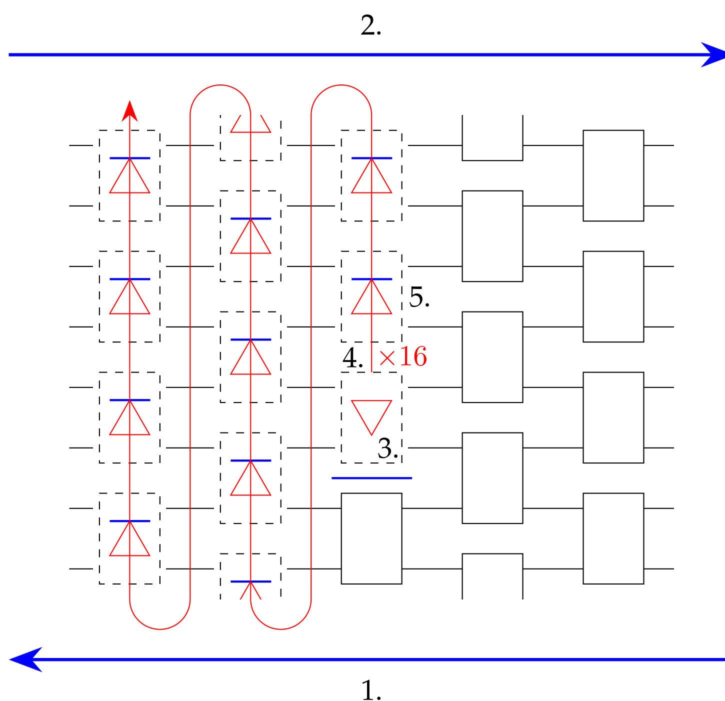


Figure 8.3: Cached Hessian computation. Splitting over a hole is indicated by an upside-down triangle, contraction with 16 state vectors by a normal triangle. Parallel contraction of 16 state vectors is marked using red color. The entire procedure (steps 3-5) is repeated for each first hole.

The exact construction of the algorithm can be seen in the following pseudocode.

Listing 8.2: Efficient cached algorithm computing the full Hessian of a circuit

```
Q: int // qubits
gates: ComplexMatrix[8][4x4] // input
hessians: ComplexMatrix[8][8][4x4] // output

L = hessians.size
gatesPerLayer = Q / 2

hessians = 0

for(oneHotIndex in 0 .. 2^Q) {

   // 1, 2: Generate cache
   cache = {
      right: applyRightAndStoreEachGate(oneHotIndex),
      left: applyLeftAndStoreEachGate(oneHotIndex)
   }

   for(layer1 in 0 .. L) {
      for(hole1 in 0 .. gatesPerLayer) {

         // 3: pick right and split
         right = cache.right[layer1][hole1]
         vecs = splitOnHole(right, layer1, hole1)

         for(layer2 in layer1 .. L) {
            for(hole2 in 0 .. gatesPerLayer) {
               if(layer1 == layer2 && hole2 <= hole1) {
                  skip
               }

               // 4: run forward until end
               for(vec in vecs) {
                  applyGate(vec, gates[layer2])
               }

               // 5: for each -> pick left, hole contract
               left = cache.left[layer2][hole2]

               for(vec in vecs) {
                  hessians[layer1][layer2][index(vec)] += contractHole(
                      left, vec, hole2)
               }
            }
         }
      }
   }
}
```

The implementation developed in this work obtains a numerically identical result using the cached Hessian algorithm as the naive implementation does. It achieves this in 8 seconds for 12 state vectors on 12 qubits while the naive algorithm takes 82 seconds for the same task. With this performance, it can compute an entire Hessian for 1 out 100

iterations in 15 days. The performance of the algorithm will be investigated in detail in section 8.6. It represents a significant improvement over the previous versions.

8.5 Splitting

As described in chapter 2, the contraction of the contraction of the network is achieved by slotting in an identity matrix and representing it as a sum of 1-hot vectors. Those are easier to run through the entire network instead of performing the contraction on the multi-dimensional tensors themselves. This requires the ability to run the contracting vector through the entire circuit. In the case of the Hessian computation, there are two state vectors, one on the right and one on the left. Contracting with them will stop at the two holes present due to the second derivative and will therefore be prevented in the center between them. A solution to allow the 1-d tensor-based contraction to continue through the entire circuit therefore needs to be found.

This can be achieved by considering the effect of each single parameter at the first hole encountered and splitting the incoming state vector into multiple vectors accordingly. The rest of the contraction is then performed for each of the resulting vectors separately thereby showing the influence of each of the parameters at the second hole for each parameter of the first. The resulting 16×16 parameters are the desired Hessian of the two gate sites replaced by the holes.

This can be done by treating the hole as filled with each of the possible 1-hot matrices having a single 1 in the place of the parameter to be investigated. This eliminates any effects from the other parameters and yields the required 16 state vectors. Instead of actually performing the matrix multiplications they can be efficiently obtained using the following equation:

$$|\phi\rangle^{a,b}_{o \oplus \hat{a} \oplus o} = \begin{cases} |\psi\rangle_{o \oplus b \oplus o} & \text{if } a = \hat{a} \\ 0 & \text{else} \end{cases} \tag{8.1}$$

Here, $|\psi\rangle$ is the input and $|\phi\rangle^{a,b}$ the output state vector for the parameter in row a and column b. o is the index representation of the unaffected qubits. This is implemented using a loop over the entries of the input state vector distributing them to the right places in the output state vectors. This way, the input is only read once in a sequential, hardware cache-friendly order.

The native implementation of the splitting algorithm is verified using a test case that simulates the multiplication with 1-hot matrices.

8.6 Evaluation

The cached algorithm for calculating the Hessian described in section 8.4 is a substantial improvement over the naive implementation described beforehand. As with the gradient, it has also been tested against a finite difference approximation of the true values and returns the correct result. The following table shows the total run time of both implementations for various problems.

qubits, layers, vectors	non-cached	cached
8 Q, 8 L, All vecs	33 s	5.2 s
10 Q, 8 L, 12 vecs	11 s	1.5 s
12 Q, 8 L, 12 vecs	83 s	8.2 s
14 Q, 8 L, 12 vecs	n/a	45 s
16 Q, 8 L, 12 vecs	n/a	240 s

As can be seen, the more sophisticated implementation is significantly faster. The speedup compared to the non-cached version is around 10 for a 12-qubit system. It will likely be higher on larger systems such as 16 qubits. It achieves a performance of $85eq\frac{kL}{s}$ (12-L) compared to $8.4eq\frac{kL}{s}$ (12-L) with the naive implementation. For 16 qubits, computing 12 state vectors is equivalent to around $1.2ML$ in 240 seconds yields a layer performance of $5.1eq\frac{kL}{s}$ (16-L).

As the Hessian tensor is symmetric relative to two gates and therefore the holes being computed being swapped, half the computation is avoided by just mirroring the result at the end. The first hole is always assumed to be the right one. This means that all Hessian performance approximations lack a factor of 2. Due to this the performance of around $84eq\frac{kL}{s}$ (12-L) should be almost equivalent to the performance on the gradient when scaled up. The speedup compared to a totally naive implementation would therefore also rise to a factor of 20, similar to the gradient case. As this optimization is trivial it was not excluded from the naive version.

As described in section 8.2, the expected maximum speedup due to workload reduction is 12.6. This means the speedup achieved by the implementation at 10 nearly realizes the full potential. There is therefore nearly no loss due to inefficient memory access. This can mostly be attributed to performing the parallel part with 16 flowing state vectors on the fly instead of cached in advance. That way the memory footprint of the application is reduced and the potential for relevant state vectors to be held in hardware cache is maximized, especially on server systems with larger caches.

At this speed, the algorithm will take 15 days to compute an entire 16-qubit Hessian. Running 100 iterations of the optimization procedure will therefore take less than five years on a laptop.

Total Run Time

This run time seems quite large but can be reduced significantly by employing large-scale parallelization. Using server processors with more cores and way larger caches can increase performance by up to a factor of 4. One server year is therefore needed to solve the entire problem. By utilizing 20 of such equivalent nodes in a compute cluster, the total execution time of the implementation is reduced to 2.5 weeks. If even more performance is needed, a larger cluster will scale the time required to solve the problem down linearly. Another option is to exploit one of the strategies that could not be explored in this work due to time constraints. It might be possible to exploit the translational symmetry of the brickwall circuit. Using dedicated hardware such as GPUs might yield a high speedup depending on whether the cache architecture can be used to achieve that. Those possibilities will be described further in chapter 9.

Parallel Efficiency

To maximize performance as with the gradient the Hessian computation was fully parallelized by distributing work along the 1-hot state vectors and summing the results. The tables in figure 8.4 show the efficiency of the parallelization relative to qubit and thread count.

Qubits	Speedup
16	3.9
14	4.1
12	4.2
10	4.5
8	4.4

(a) Speedup of parallelization by qubit count.

Threads	Time	Speedup
1	32 s	1
2	17 s	1.8
4	10 s	3.1
6	7.5 s	4.2
12	7.8 s	4.0
48	7.8 s	4.0

(b) Speedup of parallelization by thread count.

Figure 8.4: Impact of parallelization on performance.

As can be seen, the effect of parallelization achieved on 16 qubits matches the maximum possible performance improvement of 4. The parallel speedup is also limited by the L3 cache size as described in chapter 7. Similar to the gradient computation, the best performance on this machine is achieved using six threads when loading each core fully and not oversubscribing or using hyper-threading. This might change on a system with larger caches such as a server processor.

The memory usage of the Hessian computation is also similar to the gradient as apart from the same two-pass cache, no other data is permanently stored. Compared to the original Python implementation which would have to allocate 72 GB for a single layer matrix on a single thread and could therefore not be executed on most machines, the matrix-free gate-wise Hessian computation with caching only uses 2.0 GB on 12 threads. The 300 MB increase from the gradient computation is likely due to the storage of the 16 parallel state vectors flowing in the center held by each thread.

Using specialized large-scale hardware such as a compute cluster, the algorithm and implementation are therefore now sufficiently optimized to be able to solve the problem and apply the entire quantum circuit optimization procedure on a 16-qubit system in a realistic time frame.

9 Future Work

This thesis has shown that it is possible to reduce the run time of the brickwall contraction and gradient and Hessian computation so that they become feasible on current-day hardware for 16-qubit systems. This requires using a distributed compute cluster though which might not be available. Additionally, it still requires multiple weeks of computation using the entire cluster. Further optimization of the performance of the algorithms might therefore be desirable.

There are multiple approaches to doing this that were not explored in this work. They focus on different hardware choices as well as algorithmic decisions and strategies that when applied correctly might yield a few more minor speedups.

9.1 Dedicated Hardware

As already described in chapter 3, this thesis has focused on developing and implementing the basic algorithms for solving the contraction problem while focusing on a normal CPU as the target hardware. Future research might want to explore alternative choices such as graphics cards, tensor processing units, field-programmable gate arrays (FPGAs), and purpose-built circuits. The best results would probably be obtained by using purpose-built hardware and FPGAs but as those are quite expensive and complicated to use, the following sections will give some insight on the two other options.

9.1.1 Graphics Processors

One option that immediately comes to mind when talking about highly parallelizable computation at a large scale is using graphics processing units (GPUs). Those are designed to perform the same computation on different data at the same time, usually shading and determining the color of pixels. Using their compute capabilities in conjunction with frameworks like CUDA[15], OpenCL[9], and Vulkan Compute[10] they can be used for parallelized general-purpose computation as well. Due to this, they play a vital role in the current surge in machine learning which is the reason a lot of effort has gone into figuring out how to perform tensor operations on them.

Given the similarity of the given tensor network contraction problem to deep learning and backpropagation in particular, one could assume that they would also be well suited for the task at hand. The issue with this approach is that while the problems look similar on the surface, the brickwall structure of the circuit constrains the tensor network to a pretty limited and specialized version of the problem. There are only a few highly organized parameters per layer which can be exploited and allow for optimizations that are way faster than even the best full matrix or tensor multiplication could ever be. As layers in deep learning are usually drastically more dense, most of the research and libraries for linear algebra operations and tensor contraction cannot be used for the algorithms

designed in this work.

As described in chapter 7, the problem is bound by the size of the L3 cache. Graphics cards have way higher memory bandwidths and contain thousands of compute units, but as they usually only access specific data from a texture or a model parameter once, they do not contain a lot of hardware cache. The cache they have is built to hold one or two textures used by the computation and is therefore shared by a lot of units. The algorithms designed in chapters 6 and 8 on the other hand read and write the same large state vector over and over again using varying large and small strides that are common for tensor network contractions in quantum computing. They therefore require at least 2 MB of cache / fast memory per computational thread and therefore per compute unit for full performance.

While GPUs offer a lot of potential performance-wise, the efficiency of any future implementation of the algorithms presented in this work depends mainly on how well they can be adapted to and use the different memory architecture present in those devices.

9.1.2 Neural Processing Units

Due to the increasing demand for predictive computation systems using machine learning, Neural Processing Units (NPUs), also known as Tensor Processing Units (TPUs), have become a common occurrence in specialized hardware as well as consumer devices, especially mobile phones. They suffer from the same obstacles as GPUs and deep learning frameworks as they are not built to be able to exploit the characteristics of the brickwall tensor network contraction problem. It might be possible though to adapt one of them to be able to execute the algorithms presented in the previous chapters. This would probably be quite complex but might yield a significant speedup closer to what purpose-built hardware and FPGAs could achieve. It would also lower costs and reduce energy usage in comparison to employing more generalized devices such as GPUs. Whether it is possible to implement the gate-wise application on such a device at all and if the memory architecture is suitable would also have to be determined by future work.

9.2 Optimization

There might be possible optimizations remaining that would improve the performance of the presented algorithms even further. One candidate for this would be translational invariance. The brickwall circuits used in the optimization procedure use the same gate for all sites in a layer and are therefore invariant to translation. This could be exploited for the Hessian computation as the result for any vertical translation of two holes maintaining the same offset should yield the same result. The potential speedup that is obtainable using this and other techniques and their limitations were not investigated in this thesis and will be left to future research.

9.3 Large-scale Parallelization

The implementation created uses OpenMP[3] for parallelization. This allows distributing work over multiple cores on a single system. To achieve the run times given at the end of chapter 8, multiple, preferably server processor nodes, will have to be used. This will

require creating some parallelization code for multi-node scaling using a framework such as MPI[5].

9.4 Riemannian Quantum Circuit Optimization

The contraction algorithms developed in this work for evaluating brickwall quantum circuits as well as determining their gradient and Hessian work well on their own. The implementation created in this thesis computes the results significantly faster using less memory and has been verified for correctness using unit testing based on full matrix multiplications and finite difference approximations.

The task was originally motivated by the desire to apply the Riemannian quantum circuit optimization procedure developed by Kotil et al.[14] to larger 16-qubit systems. Due to time constraints, the other parts of that method have not been implemented though and the various programs have not been integrated.

Future work would therefore have to either reimplement the optimization method natively or integrate the native algorithms from this thesis with the original Python implementation to be able to use the gradient and Hessian computation methods developed in this work for that purpose.

10 Conclusion

The main goal of this thesis was to investigate and implement potential solutions and optimizations for the fast contraction of brickwall tensor networks. The focus was on obtaining the gradient and Hessian of the function represented by it using holes cut into the network and algorithms contracting around those.

As part of this work, a native C++ based implementation was developed that achieves the specified performance on the various tasks. All its desired functionalities have been verified using unit testing. The full source code has been published at `https://github.com/putterer/quantum-brickwall-contraction`.

Target Platform and Parallelization

As shown in section 5.2 the problem is quite large. A lot of computational power will there have to be used to solve it. This requires parallelizing the problem over multiple cores on a single node but will likely also require distributing the workload over multiple nodes to achieve any reasonable completion time.

The algorithms will benefit from choosing hardware with a high degree of parallelization if certain criteria are met. The performance is L3 cache-bound due to lots of repeated memory accesses. Due to this and the specialized nature of the problem structure, a GPU and TPU might or might not be a suitable choice. The details and effect of transferring the algorithms to a graphics processing device could not investigated as part of this work due to time constraints. If a high number of threads processing in parallel is desired a server processor should be chosen over a regular CPU due to its higher core count and cache size. This will mitigate the impact of cache misses.

As the benchmarks in the various chapters have shown, gate-wise application is significantly faster than determining the entire transformation matrix for each layer. Additionally, it uses way less memory and allows for further optimizations using parallelization, reordering, and caching. Even though memory access is not strictly sequential, it occurs in multiple streams that are each cache local individually. The same applies to hole contraction which is fundamentally an identical problem indexing-wise with different inputs and outputs. The gate application should therefore be as efficient as possible if sufficient large hardware caches are available.

Gradient

The central goal was finding a solution to determine the gradient efficiently for use in the optimization method. A naive as well as a significantly faster cached algorithm and corresponding implementation were developed. The advanced version determines the gradient of a tensor network and therefore the target function 30 times faster and uses only a small amount of memory when compared to the original baseline Python-based implementation. It uses the gate-wise application as well as parallelization to achieve

this. Caching is done by performing a forward and backward pass over the circuit and storing the intermediate result at each individual gate.

Deep Learning

This caching process when done on the fly is similar to deep learning models being trained using backpropagation. One of the main differences between the two problems is that in this case the approximated linear function is known exactly while in the case of statistical modeling like machine learning usually only sampled probabilistic data is available. Additionally, neural networks are designed with non-linearities while quantum circuits and the target function in this work are completely linear. The brickwall circuit configuration also uses complex values instead of real ones and contains only 2-qubit gates and therefore 16 parameters per layer, compared to hundreds to millions in larger machine learning models. All these properties allow for optimization that will yield higher efficiencies when solving this problem compared to conventional machine learning methods. While the setup is quite similar to deep learning and it can be used as an inspiration the algorithms developed in this work yield significantly better performance.

Hessian

The workload of computing the Hessian is substantially larger than that of the gradient and for optimizing the entire procedure the focus should be set on investigating and optimizing it. As with the gradient, the naive implementation contains quite a lot of redundancies. Those have been mostly eliminated with the way more efficient cached implementation developed in this work. It allows obtaining the Hessian around 10 times faster than the naive version. Like the gradient, it has a significantly lower memory profile compared to the original implementation while offering lower execution times. It accesses memory efficiently and scales well with parallelization and distribution if sufficient L3 cache is available.

Distribution and Performance

Running the quantum circuit optimization procedure for 100 iterations will still take multiple years using the final versions of the algorithms on a laptop. To obtain more reasonable run times larger parallelization and distribution onto multiple server processors organized in a compute cluster is therefore needed. By utilizing a few dozen server CPUs, a total run time in the range of weeks should be achievable.

Using the methods described and the implementation developed in this thesis, it should therefore be possible to compute the gradient and Hessian of brickwall circuits efficiently and therefore to apply the Riemannian quantum circuit optimization procedure for 2-dimensional 16-qubit systems on current-day hardware in a reasonable time frame.

List of Figures

2.1 Circuit representation of the inner part of the trace as shown in equation 2.1 5
2.2 Computing the trace function using tensor network contractions. 5
2.3 The tensor network representations used for computing the gradient and Hessian of the target function. The actual problem has 16 qubits and 64 gates. 7

3.1 Comparison of approximate data accesses and operations performed during the two algorithms. 9
3.2 Comparison of computational performance of several hardware solutions on single (32-bit) and double (64-bit) precision floating point numbers in trillion operations per second (TFLOPS) [17]. 10

4.1 The remaining tensor network before the final gate hole contraction. . . . 17
4.2 Apart from the data flow in the inner operation, the indexing is identical. 18

5.1 The scaling of the different complexity classes relative to the Terra (10^{12}) order of magnitude. 23

6.1 The tensor network to be contracted for the gradient computation. 24
6.2 The indexing of the two cache generation passes when applied at a layer level. In this case, the last layers in both directions could be skipped, in the site-wise caching the speedup gained by that is negligible. 27
6.3 Performance comparison of the different versions of the gradient algorithm. 32

8.1 The tensor network to be contracted for the Hessian computation. 37
8.2 Naive Hessian computation. Splitting over a hole is indicated by an upside-down triangle, contraction with 16 state vectors by a normal triangle. Parallel contraction of 16 state vectors is marked using red color. The entire procedure is repeated for each combination of holes. 39
8.3 Cached Hessian computation. Splitting over a hole is indicated by an upside-down triangle, contraction with 16 state vectors by a normal triangle. Parallel contraction of 16 state vectors is marked using red color. The entire procedure (steps 3-5) is repeated for each first hole. 41
8.4 Impact of parallelization on performance. 45

Listings

4.1 Computing the output state vector of a single gate application 15
4.2 Computing the gradient of two state vectors over a hole 17

6.1 Naive algorithm computing the full gradient of a circuit 25
6.2 Efficient algorithm using caching to compute the full gradient of a circuit . 27
6.3 Computing the gradients during the backward pass 29
6.4 Caching forward and backward pass and computing gradients later . . . 29

8.1 Naive algorithm computing the full Hessian of a circuit 39
8.2 Efficient cached algorithm computing the full Hessian of a circuit 42

Bibliography

[1] IEEE standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019.

[2] P.-A. Absil, R. Mahony, and Rodolphe Sepulchre. *Optimization Algorithms on Matrix Manifolds*. Princeton University Press, Princeton, 2008.

[3] OpenMP Architecture Review Board. OpenMP application programming interface version 5.2. 2021.

[4] Andrew M. Childs, Yuan Su, Minh C. Tran, Nathan Wiebe, and Shuchen Zhu. Theory of trotter error with commutator scaling. *Phys. Rev. X*, 11:011020, Feb 2021.

[5] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 4.0. 2021.

[6] Free Software Foundation. Using the GNU Compiler Collection - for gcc version 11.3.0. 2021.

[7] Fritzchens Fritz. AMD Ryzen 3 4300U - Die Photography. `https://flic.kr/p/2jcNmuz`, 2020.

[8] Nemez [@GPUsAreMagic]. AMD Renoir annotation. `https://twitter.com/GPUsAreMagic/status/1274024392469741569`, 2020.

[9] The Khronos Group. OpenCL. `https://www.khronos.org/opencl/`, 2023.

[10] The Khronos Group. Vulkan. `https://www.vulkan.org/`, 2023.

[11] Michael Grupp. Tum thesis template. `https://github.com/michaelGrupp/TTT`, 2017.

[12] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[13] Google Inc. GoogleTest. `https://github.com/google/googletest`, 2023.

[14] Ayse Kotil, Rahul Banerjee, Qunsheng Huang, and Christian B. Mendl. Riemannian quantum circuit optimization for hamiltonian simulation, 2022.

[15] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA: Is CUDA the parallel programming model that application developers have been waiting for? *Queue*, 6(2):40–53, mar 2008.

[16] TechPowerUp. CPU Specs Database. `https://www.techpowerup.com/cpu-specs/`, 2023.

[17] TechPowerUp. GPU Specs Database. `https://www.techpowerup.com/`

gpu-specs/, 2023.

[18] Guido Van Rossum, Fred L Drake, et al. *Python reference manual*, volume 111. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[19] www.7 cpu.com. LZMA Benchmark - AMD Zen2. https://www.7-cpu.com/cpu/Zen2.html, 2023.