Optimizing Quantum Computer Simulations With Data Compression & GPU Acceleration

Bachelor's thesis in Computer science and engineering

Erik Ljung
Darko Petrov
Felix Bråberg
Björn Forssén
Beata Ringmar
Jonas Hedlund

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Bachelor's thesis 2022

Optimizing Quantum Computer Simulations With Data Compression & GPU Acceleration

Erik Ljung
Darko Petrov
Felix Bråberg
Björn Forssén
Beata Ringmar
Jonas Hedlund

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

Optimizing Quantum Computer Simulations With Data Compression & GPU Acceleration

Supervisor: Stavroula Zouzoula, Computer and Network Systems division, Department of Computer Science and Engineering
Examiner: Pedro Petersen Moura Trancoso, Computer and Network Systems division, Department of Computer Science and Engineering Science

Bachelor's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2022

Abstract

Simulating quantum computers involves high memory usage and often long execution times. For that reason the purpose of this project was to analyze whether data compression and GPU acceleration can be used to run simulations with more qubits than previously allowed. Ultimately this project sheds some light on how GPU acceleration and data compression algorithms, ZFP and FPZIP, impact the amount of qubits that are able to be simulated. The simulator tested in this project was a modified version of the Quantum Exact Simulation Toolkit (QuEST). From the results of this project it was found that data compression shows good potential in decreasing the total memory usage per qubit size. However, the use of data compression negatively impacted the execution time, but by using GPU acceleration the impact was reduced.

Sammandrag

Simulering av kvantdatorer har en hög minnesanvändning och ofta långa exekveringstider. Av den anledningen var syftet med detta projektet att analysera ifall datakomprimering och GPU-acceleration kan användas för att köra simuleringar med fler qubits än vad som var tidigare möjligt. Detta projekt belyser hur GPU-acceleration och komprimeringsalgoritmerna, ZFP och FPZIP, påverkar mängden qubits som kan simuleras. Simulatorn som testades i projektet var en modifierad version av Quantum Exact Simulation Toolkit (QuEST). Resultatet av projektet påvisade att datakomprimering visar en god potential för att minska den totala minnesanvändningen per qubit. Användningen av datakomprimering påverkade dock exekveringstiden negativt, men genom att använda GPU-acceleration minskades effekterna.

Acknowledgements

We would like to express our deepest gratitude to our supervisor Stavroula Zouzoula who has granted invaluable insight and support through the course of this project.

Contents

List of Figures

List of Tables

Glossary

Bloch sphere - A geometric representation of a quantum state.

Bottleneck - A part of a program, in this project, that is particularly inhibiting for the overall performance.

CPU - Central Processing Unit, this is where the main calculations happen in a computer.

CUDA - Compute Unified Device Architecture is a parallel programming model designed by Nvidia. It's used for accessing and programming Nvidia GPUs.

Floating-point - A way to represent a large range of decimal numbers with a small number of bits by sacrificing some accuracy.

GPU - Graphics Processing Unit, a hardware component dedicated to perform many parallel calculations, usually relating to graphics, and assist the CPU.

heaptrack - A tool for analyzing heap memory utilization.

heaptrack_print - A tool for reading output files from heaptrack and presenting them in a human readable form.

Lossless compression - Compression technique that will perform reversible compression where no data from the original source is lost.

Lossy compression - Compression technique that will perform irreversible compression where data from the original source is lost.

Negabinary - A number represented in base -2.

Quantum gate - A logic gate representing the quantum logic applied to any number of qubits much like classical logic gates.

RAM - Random-Access Memory, the main memory unit of a computer.

VRAM - Video Random-Access Memory, the main memory unit on a graphics card.

1 Introduction

1.1 Background

Quantum computers as a concept were first mentioned by the mathematician Yuri Manin and later gained traction when the physicist Richard Feynman furthered the concept [2]. Feynman argued that as the complexity of the system increases, simulations of quantum systems become infeasible on classical computers. He even stated during a keynote address at the California Institute of Technology that, "...nature isn't classical, ... and if you want to make a simulation of nature, you'd better make it quantum mechanical, and by golly it's a wonderful problem, because it doesn't look so easy." [3].

The purpose for the development of quantum computers is to solve problems that are otherwise too difficult to solve on classical computers. Through the utilization of specially designed algorithms, this purpose can be achieved. One such algorithm is Shor's algorithm, which, with a quantum computer, can perform prime factorization in polynomial time [4]. This is a significant improvement from classical algorithms that perform prime factorization in non-polynomial time. Quantum computers can, therefore, have a big impact on cyber security since modern cryptography relies on the difficulty of prime factorization.

The main difference between a classic and quantum computer, is the size of the problem space in which they operate. The basic building blocks in classical computers are bits, which can have a value of either 1 or 0. In quantum computers, bits are instead replaced with quantum bits, or qubits. A qubit can fall into one of three different states: 1, 0, or a superposition state of 1 and 0. While classical computers can only represent one specific state at a time, quantum computers are able to represent all possible states, known as quantum superposition [2]. The reason quantum computers can operate in a larger problem space is because they can utilize quantum mechanical properties such as superposition and quantum entanglement. To fully utilize the potential in a quantum computer, specific quantum algorithms are required. These algorithms take advantage of the quantum mechanical properties to simultaneously represent all possible states and then collapse the system into a desired end state.

While, theoretically, quantum computers have a broad range of applications, the ability to construct these machines is still in early development. IBM only recently revealed their quantum computer with 127 qubits [5]. This is when the role of a simulator becomes imperative. Simulators are often used to replicate the functionality of quantum algorithms by simulating the qubits on classical hardware [6]. There are multiple ways of implementing a quantum simulator on a classical computer. One such way is to represent the quantum states as state vectors. Matrices are used to represent quantum gates, and show how the state vectors can be manipulated. Quantum Exact Simulation Toolkit (QuEST) is an example of a simulator that implements pure state-vector representation to simulate a quantum computer [7]. A drawback with the state-vector based simulators is that resource requirements will grow by a factor of 2 for each added qubit, e.g. simulating 30 qubits will require a state-vector of size 2^{30} to represent each possible state [8].

One of the main resource limitations when it comes to simulating quantum computers is the increase in memory requirement for each new qubit [8]. One way to address this problem is to use data compression [9]. Data compression works by compressing larger pieces of data into smaller representations. There are two types of compression techniques, *lossless compression* and *lossy compression*. Lossless compression refers to compression where no data is lost. Although lossless compression is effective there are practical limits to how much the data can be compressed [10]. An alternative compression method is lossy compression. Lossy compression may achieve a higher compression ratio but comes with the risk of data loss. Due to this risk, lossy compression is not inherently reversible [11].

Another common way to improve performance is through hardware acceleration. It can be used to offload some of the work that the Central Processing Unit (CPU) performs and potentially do faster computations. This is because hardware accelerators are typically designed for a specific task and that task only. Graphical Processing Units (GPUs) are well-suited for this role as they are more able to be parallelized than a CPU, thereby allowing a higher level of concurrent executions [12].

1.2 Purpose

When simulating the full state of quantum computer circuits on classical hardware, one of the key issues is the memory and computation requirements that increase exponentially with the number of qubits [8]. For example, a quantum circuit consisting of 25 qubits can be simulated on a laptop, while a 50 qubit circuit will require a supercomputer. Therefore, the purpose of this project is to analyze if data compression and GPU acceleration can be used to allow the simulation of larger quantum circuits at a reasonable speed. Ultimately this project should provide insight into how different compression algorithms and GPU acceleration can impact the number of qubits that can be simulated on classical hardware. The task of testing different compression algorithms included both lossless and lossy techniques. The reason for trying both categories of compression was to provide further knowledge of how data loss may impact the performance, size, and correctness of a quantum circuit.

1.3 Scope

Certain limitations were set in order to narrow the scope of the project. One such limitation was to implement optimization techniques for only one quantum computer simulator. Each simulator on the market uses a different simulation method in order to focus on a different area of research. The choice to study only one simulator, allowed for a thorough investigation into how compression and hardware acceleration affect memory usage and execution time for that specific simulator. Qiskit [13] and CirQ [14] are well-established quantum simulators made by some of the biggest names in quantum computing today, Google and IBM. Both Google and IBM are also developers of actual quantum computing hardware, which may lead to the simulator's libraries containing fewer details of the actual quantum computational procedure [15]. On the other hand, QuEST was built purely with the intention to be used for research purposes and as such was better suited for the needs of this study [15]. The choice to focus on one simulator also resulted in the limitation of one simulation method. The use of QuEST meant limiting the project to the state vector method of simulating general quantum circuits. While there are various methods of simulation, the state vector method is fast and versatile when memory size is sufficient and the number of qubits is low [16].

Another limitation of the project was the choice to study the effects of the optimization techniques using one quantum algorithm. The algorithm in question was Grover's search algorithm which is a well-established quantum algorithm for efficiently finding an element in an unsorted list [17]. The choice to limit the study to only one algorithm was related to the problem statement and scope. The focus of this project was not on the quantum algorithm itself, but instead on the algorithm's performance and whether that performance can be improved using techniques found in classical computing. The study was also limited by two compression algorithms, one lossless and one lossy. While this was partially due to the project's time constraint, the choice to study two different compression algorithms was deliberate. By implementing two contrasting compression methods, one could easily compare which was superior in terms of resource requirements.

2
Theory

This chapter presents the underlying theory necessary to understand the methods of implementation and testing discussed in the subsequent chapters. The topics covered in this chapter range from the basics of quantum computer simulators, data compression, performance metrics, and GPUs.

2.1 Simulating Quantum Circuits

As previously stated, quantum bits differ from classical bits. There exists a way to represent qubits using classical bits to simulate quantum states on classical hardware. One way to do this is to describe quantum states using state vectors. The state vector representing a single qubit contains two probabilities for collapsing into either state 0 or 1. For two qubits, there are four possible states for the system to collapse into, and as such the state vector contains four entries. The depiction of these state vectors is shown below.

$$\underset{\begin{bmatrix} a_0 \\ a_1 \end{bmatrix}}{\text{1 qubit}} \qquad \underset{\begin{bmatrix} a_{00} \\ a_{01} \\ a_{10} \\ a_{11} \end{bmatrix}}{\text{2 qubits}} \qquad \underset{\begin{bmatrix} a_{00...0} \\ a_{00...1} \\ \dots \\ \dots \\ \dots \\ a_{11...1} \end{bmatrix}}{n \text{ qubits}}$$

It is common to utilize Dirac notation when representing individual states inside a state vector. Using Dirac notation simplifies the representation of states with a higher number of qubits [18]. Below is an example of how Dirac notation works for a system of 2 qubits.

$$|00\rangle = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad |01\rangle = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \qquad |10\rangle = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \qquad |11\rangle = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

In conclusion, to fully define the state of a quantum computer, a state vector of 2^n amplitudes is required (n represents the number of qubits to simulate). This exponential growth will also lead to an exponential increase in memory requirements when simulating quantum systems. After a certain number of qubits, it becomes very resource-intensive to simulate a quantum computer [19]. When using QuEST to simulate a circuit on a personal computer, only about 30 qubits can be run as the memory usage reaches approximately 16GB. This is the limit for most personal computers today, due to the limited memory on consumer hardware and the state vector's exponential growth.

Of course, simply using state vectors to represent qubits does not create a functional simulator, there must also be a way to simulate quantum gates. Classical gates provide the basic functionality inside conventional computers, in much the same way as quantum logic gates provide the basic functionality for quantum computers. Quantum gates are represented as matrices, applied to a qubit to perform basic operations. The most common quantum gates operate on one or two qubits, meaning that multiple gates are needed to form a quantum circuit, just as conventional circuits are composed of multiple logic gates [20]. Some of the most imperative single-bit gates are the Pauli gates, *X, Y, Z* which can be represented as matrices [21].

$$X = \sigma_x = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \qquad Y = \sigma_y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \qquad Z = \sigma_z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

The Pauli-X matrix, as understood from linear algebra, is used to represent the X-gate. When the X-gate operates on a qubit, the Pauli-X matrix is multiplied by that qubit's state vector. An example of how this gate operates on $|0\rangle$ is below. The gate can also be expressed as a rotation of π radians along the x-axis of the *Bloch Sphere*. The Pauli-X operator is analogous to the NOT-gate in classical computing. The Y- and Z-gates operate similarly, except that they rotate along the y- and z-axes, respectively [21].

$$X|0\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = |1\rangle$$

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \qquad |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

The Hadamard Gate, *H*, is also a fundamental operation in quantum computers. When given a basis state, the Hadamard Gate creates a superposition of the states. The matrix, shown below, is used to represent the gate. The Hadamard Gate and the Pauli Gates are both implemented in Grover's search algorithm [20].

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

2.1.1 Grover's Search Algorithm

Grover's search algorithm answers a classic computing problem, namely searching for an element within a list. Grover's search algorithm is a clear demonstration of when the performance of a quantum computer significantly outperforms that of a classical computer. With classical computation, the time complexity of finding the correct element in an unsorted list is O(N) [17]. However, when using a quantum computer the time complexity is only $O(\sqrt{N})$. This decrease in complexity is achieved by utilizing an oracle and a reflection operator. Together, these operations define a technique known as amplitude amplification. Simply, this technique amplifies the probability of the state collapsing into the winning state, w by reducing the likelihood of it collapsing into all the other states.

The first operation in the algorithm is the oracle. The oracle works by taking several inputs and changing the winning state's sign. The inputs are encoded as the basis states of a specified number of qubits. For example, for two qubits, the inputs will be the states 00, 01, 10, and 11. It flips the sign of the winning state to its negative phase through the use of a unitary matrix. To exemplify the algorithm's functionality, let $w = 11$. When the position of the winning state is still unknown, the probability of the superposition collapsing to any of the basis states has equal probability. The state that represents this is known as the superposition state, or s [22]. The following vectors express the states $|w\rangle$ and $|s\rangle$:

$$|w\rangle = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = |11\rangle \qquad\qquad |s\rangle = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \tfrac{1}{2}$$

The oracle takes $|s\rangle$ as an input and outputs a state where the sign of the winning state has flipped, shown from the following vector:

$$\begin{bmatrix} 1 \\ 1 \\ 1 \\ -1 \end{bmatrix}$$

This oracle operation can also be described graphically, as seen in Figure 2.1. The graph on the left shows the original vectors $|w\rangle$ and $|s\rangle$. It also shows the state $|s'\rangle$ that is orthogonal to $|w\rangle$. The graph on the right shows the transformation of $|s\rangle$ after the oracle has been applied. It has been reflected across $|s'\rangle$ to get a new vector.

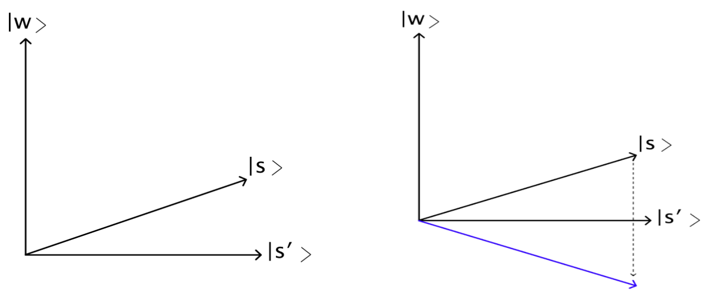


Figure 2.1: The oracle operation applied to $|s\rangle$

The next step is to apply a reflection operator, defined as $2|s\langle s| - 1$, to this new vector. The result of this operation is the reflection of the vector back across $|s'\rangle$. From Figure 2.2, one can see the combination of these two transformations has moved the vector closer to the winning vector, $|w\rangle$. With 2 qubits, it only takes one application of the amplitude amplification operation for our quantum state to arrive at the winning state. However, by increasing the number of qubits, the number of times the amplitude amplification must be applied increases by $\sqrt{N}$ [22].

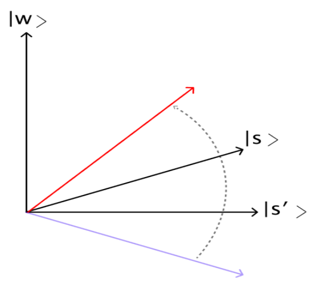


Figure 2.2: The reflection operator

2.1.2 QuEST

QuEST is a simulator for quantum circuits that are built for high performance by using multi-threading, GPU acceleration, and distributed computation [23]. This allows QuEST to run efficient simulations on everything ranging from simple laptops to powerful supercomputers. The QuEST simulator supports both state-vector and density matrix representations, thereby allowing simulations of quantum circuits with pure or mixed states. State-vector representation involves having a vector filled with complex numbers. QuEST solves this by having two separate floating-point arrays in memory, where one of the arrays stores the real parts of the complex number and the other holds the imaginary parts. Together, these two arrays define the complete state of a quantum computer with a fixed number of qubits [7].

2.2 Data Compression

The amount of available data is constantly increasing, although handling it remains a challenge. One method of tackling this issue is through data compression. Data compression is the task of taking a representation of data and determining a smaller data size that still represents the same data. Today, data compression has many purposes, most commonly in the storage of images and audio files [24].

Compressing data can be done in one of two ways, either through lossless or lossy compression. Lossless compression refers to when no data is lost and the quality remains the same. What makes lossless compression advantageous is that after decompression, the data is restored to its original form. This function is essential when compressing confidential documents or digital images where one wants to maintain high quality. The alternative is lossy compression which can achieve a higher compression ratio but at the risk of permanent data loss. In this type of compression, once a file is compressed it cannot be entirely restored to its original form [11].

2.2.1 ZFP

ZFP is a lossy compression algorithm for high compression of multi-dimensional numerical arrays[25]. The algorithm works by splitting the multi-dimensional array, or d-dimensional, into arrays of 4^d values, also known as blocks. These blocks are compressed and decompressed independently of one another [26]. Once the data has been partitioned into blocks, the floating-point values contained in each block are converted to a block-floating-point representation. Block-floating-point is a method that assigns a single exponent to all the values in the block. Essentially, each of the floating-point values becomes a signed integer [26]. The blocks undergo a process of decorrelation using a near orthogonal transform to remove redundant information. This orthogonal transform resembles the discrete cosine transform used in image processing [27]. It transforms data from a spatial domain into a frequency domain, where the higher frequencies are identified as redundant [28].

Next, the signed integer coefficients are reordered in decreasing magnitude resulting in the grouping of ones together and zeros together in the bit plane [27]. After which, the standard representation of integers, or the two's complement signed integers, are converted to their negabinary-representation. The leftmost one-bit in this representation encodes the sign and approximates the magnitude of a value. Negabinary-representation facilitates encoding due to the numbers having many leading zeros regardless of the sign. Currently, the list of 4^d values is sorted by the magnitude of the coefficient. The next step is to reorder them from most to least significant bit on the bit plane [26].

Often, coefficients have many leading zeros. Only a single zero-bit is needed to represent a group consisting only of zeros, which results in less memory usage. As such, the goal is for groups of zero-bits to be encoded together. Each bit plane is losslessly compressed using embedded coding. The embedded coder will emit one bit at a time until it satisfies the criteria for stopping. This stopping criterion is defined by which mode of ZFP compression is being run, namely fixed-rate, fixed-precision, or fixed-accuracy [27].

The three compression modes each determine a different way of compressing data. Fixed-rate compression stores the compressed floating-point number as a set number of bits. In other words, it will always compress the data to a defined limit even if the data has the potential to be represented using a smaller data set. Using fixed-precision compression, the bit size of the compressed data may vary but will never become greater than some fixed limit. The difference between these two modes is that fixed-precision can represent the same amount of data using a smaller compressed data size. Lastly, fixed-accuracy compression limits the compressed data based on a defined accuracy for the floating-point number, maintained throughout the execution. If, for example, the tolerance is three decimal points, fixed-accuracy mode will compress the data to a size that corresponds to three decimal point accuracy [27].

2.2.2 FPZIP

FPZIP is a lossless compression algorithm for two or three-dimensional floating-point arrays. The algorithm parses through the array and predicts each data point based on a subset of data available to the decompressor. The data in this subset is already encoded. Next, the predicted values and the actual floating-point values, are mapped to their signed integer representation which calculates the prediction residuals. If the residuals were to be calculated by subtraction of the original floating-point values it might cause an irreversible loss of information [29].

Then, the residual values are partitioned into entropy codes and raw bits by an entropy coder. Using a method known as range coding, the entropy coder encodes the symbols by taking a range of integers that function as a unique representation of a string. Each symbol in the string has a probability distribution that divides the original range into sub-ranges. The size of these intervals will be proportionate to the probability of the symbol it represents. These portions must be at non-overlapping intervals. The range is then reduced further to a sub-interval which corresponds to the next symbol to be encoded. This process then repeats itself for all the remaining values [29].

2.3 Graphics Processing Unit

Hardware acceleration is the process of running a dedicated set of instructions on a hardware component that is optimized to perform those specific instructions efficiently. One typical example of hardware acceleration is rendering computer graphics, where the rendering task usually is offloaded to a dedicated Graphics Processing Unit, or GPU. The GPU is much better suited for handling tasks that allow for a high level of parallelism, such as calculating the colors of dedicated pixels [30]. GPUs are different from CPUs in that they can perform many computations simultaneously. CPUs, in contrast, typically perform complex instructions that are sequential in nature. GPUs are built to handle a large number of graphics calculations which often involve floating-point arithmetic. GPUs also differ from CPUs on an architectural level. GPUs were created to prevent the CPU from becoming overwhelmed when performing graphics calculations. Today, GPUs have grown to become sophisticated and powerful multi-core systems that can respond to high computational demands. Mainly, GPUs are either integrated on the motherboard of the respective computer or connected via a PCIe port. The latter alternative usually yields better performance [12].

2.3.1 CUDA

Compute Unified Device Architecture (CUDA) is a programming model created by Nvidia to provide programming access to their line of GPUs used in consumer software. CUDA can improve a program's performance by carrying-out calculations on the GPU instead of the CPU. A developer can use CUDA with several programming languages such as C/C++, OpenCL, Fortran, etc. The CUDA programming model uses an abstraction hierarchy. This hierarchy consists of warps, grids, threads, and blocks, of which threads are the fundamental execution units. Multiple threads make up a block and multiple blocks make up a so-called grid. A warp is a composite of threads executing in Single-Instruction-Multiple-Thread (SIMT) mode. SIMT is when the multiple threads in a warp can simultaneously perform the same instructions. There are two types of functions, those executed on the GPU, referred to as *kernel functions*, and those executed on the CPU. However, it is the responsibility of the CPU to transfer the data between the host (CPU) and the device (GPU). The CPU is also responsible for the invocation of the said kernel functions and the setting up of the dimensions for the blocks and grids of the device. Parameters such as these are to be defined by the programmer [31].

The architecture of a GPU, consisting of thousands of threads compared to the tens of threads on a CPU, promotes the parallel execution of tasks. While the single thread speed on a CPU is usually higher than that on a GPU, the throughput of the GPU is higher as a result of the high parallelism that it offers. Not all Nvidia GPUs support CUDA, but some that do are the GeForce, TitanX series GPUs as well as a collection of the Quadro and RTX series [32].

2.4 Measuring Performance

To achieve a formal and measurable quantity of performance, 'performance' is defined as the *absolute execution time of a program* per the definition stated by Patterson and Hennessy [33]. Equation 2.1 and equation 2.2 both illustrate the underlying equation. For equation 2.2, I is the number of instructions of a program, CPI is the average number of cycles per instruction and T is the period of the system clock.

$$Execution\ Time = \frac{Instructions}{Program} \times \frac{Clock\ Cycles}{Instruction} \times \frac{Seconds}{Clock\ Cycle} \tag{2.1}$$

$$CPU\ Time = I \times CPI \times T \tag{2.2}$$

Not only is defining a metric for execution time important so is finding a definition of memory utilization since it is the core tool for analyzing the efficacy of the compression algorithms. The metric used for measuring memory usage is compression ratio. Equation 2.3 below shows the specific formula for calculating the data compression ratio.

$$Compression\ Ratio = \frac{Uncompressed\ Size}{Compressed\ Size} \tag{2.3}$$

2.5 Fidelity

When information travels, it is vulnerable to changes, both expected and unexpected. One way to describe the overall similarity between the sent and received information is fidelity. In quantum computing, fidelity is the measure of the error rate produced between the expected output and the actual output.

In Quantum computer simulator when compression is applied data loss can happened. To determine the data loss, fidelity can be used to represent the similarity between the original data and the data after decompression. When changes occur in the state vectors, potential errors can arise in the simulation. State vectors determinate how the qubits will collapse to get the resulting calculation and high fidelity minimizes the risk of qubits collapsing to the wrong state.

Using Equation 2.4 fidelity can be calculated mathematically by turning into density matrix and comparing the original state vectors as ρ_1 and the state vectors after using compression as ρ_2. This produces a number between 0 and 1 that represent how similar the two state vectors are.

$$F(\rho_1, \rho_2) = \{trace[\sqrt{\sqrt{\rho_1}\rho_2\sqrt{\rho_1}}]\}^2 \tag{2.4}$$

Figure 2.3 shows visual representation of how fidelity is calculated. Comparing $\rho 1$ as the original data and $\rho 2$ as the data after compression. Fidelity is represented in the intersecting area between the two graphs.

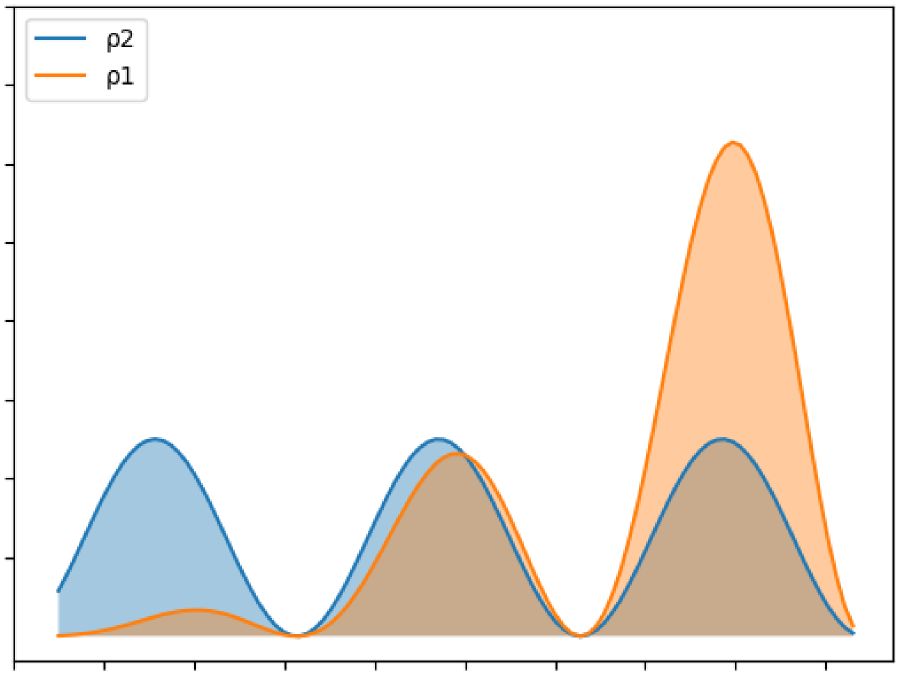


Figure 2.3: Visual representation of fidelity shown as the intersection between two state vectors ρ_0 and ρ_1 [1]

3

Implementation

This chapter details the methods used to integrate data compression into QuEST, implement GPU acceleration, and test these new techniques. Describing the methodology of the study is crucial when contextualizing the final results.

3.1 Adapting QuEST to Compressed Data

As mentioned in Chapter 2, the QuEST simulator used two arrays of floating-point numbers to represent the current state of the quantum computer. When performing operations, QuEST directly fetched and modified the floating-point numbers in these two arrays, thereby changing the underlying state of the quantum system. Given this approach, a new memory structure was created to support the compression of the state vector. If the memory structure remained the same, one would run into the same problem as when no data compression is used. Which is to say, if the entire state vector were to be decompressed at one time, it would require a memory size as large as the original state vector, defeating the purpose of using data compression.

3.1.1 Defining a Memory Structure

The new memory structure took advantage of a block-style implementation. The two state vector arrays were split into multiple blocks where each block contained a fixed number of floating-point values. Through this approach, only a fixed number of blocks remained decompressed in memory at one time. This allowed the remaining blocks to remain compressed in memory and decreased the overall memory usage. Figure 3.1 illustrates how the memory block structure works. As can be seen, two decompressed blocks are used to fetch and modify the state vector's floating-point values.

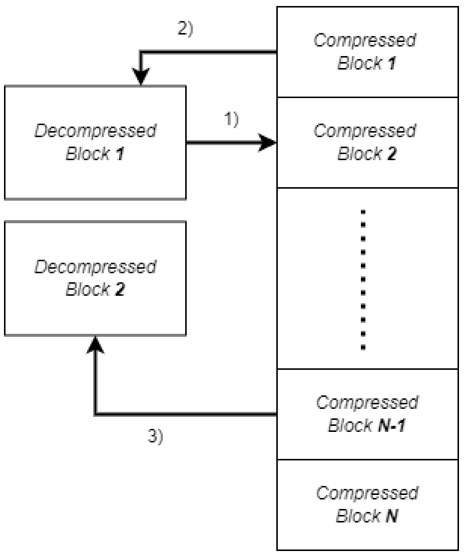


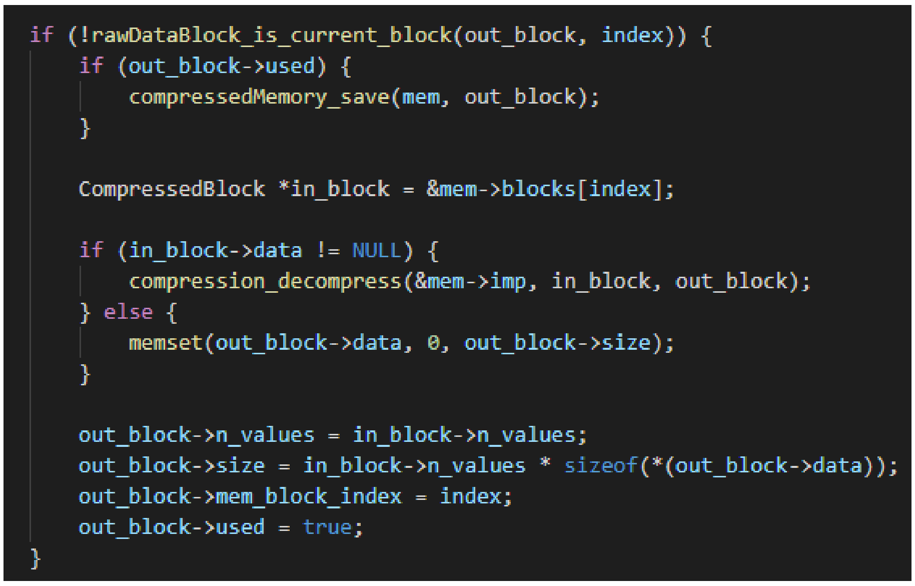
Figure 3.1: The memory block structure 1) saving decompressed state to memory 2) loading compressed state from memory 3) loading compressed block into second decompressed block

For a specific index in the state vector to be fetched or modified, the QuEST simulator recalculated the index into two separate parts, one block index and one internal index. As the names imply, the block index determined which block was to be accessed while the internal index determined the index of the value within the specific block.

3.1.2 Fetching & Modifying the State Vector

By implementing a new memory block structure, the process of accessing specific values within the state vector became more complex. More explicitly, the new memory structure resulted in the handling of three potential scenarios to ensure the compression and decompression of the correct memory block.

The first of these scenarios was when no compressed block was already decompressed in the memory. In this case, QuEST needed to find the correct block and decompress it before performing the fetch or modify operation. In the second scenario, the decompressed block in memory had the same block index as the index currently being accessed. As a result, no further action was necessary as QuEST had previously accessed the block. In the final scenario, the decompressed block in memory did not share the same block index as the current index, and thus QuEST had to perform two actions to access the correct value. Firstly, it had to compress the existing decompressed block and store it in the correct location in the memory structure. Secondly, QuEST had to find the correct block corresponding to the current index, decompress the block, and store it in memory, thereby allowing access to the correct value. Figure 3.2 shows the specific code for the handling of these three scenarios.

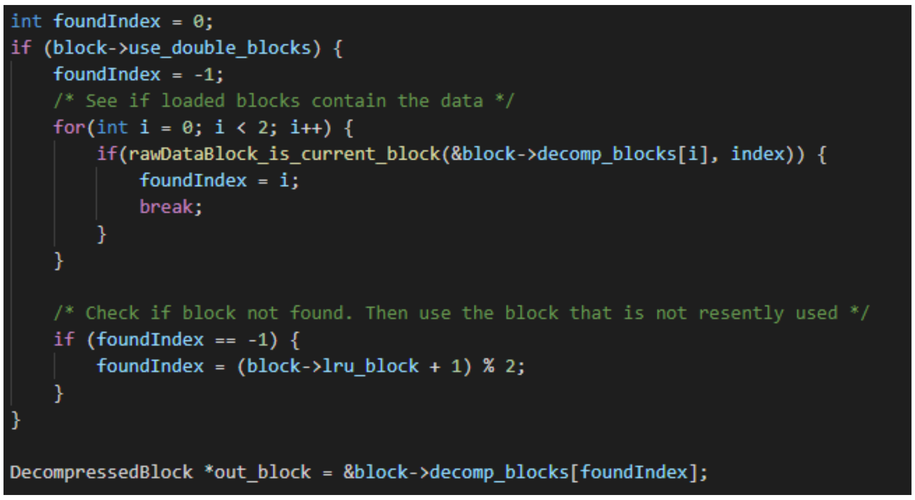


```c
if (!rawDataBlock_is_current_block(out_block, index)) {
    if (out_block->used) {
        compressedMemory_save(mem, out_block);
    }

    CompressedBlock *in_block = &mem->blocks[index];

    if (in_block->data != NULL) {
        compression_decompress(&mem->imp, in_block, out_block);
    } else {
        memset(out_block->data, 0, out_block->size);
    }

    out_block->n_values = in_block->n_values;
    out_block->size = in_block->n_values * sizeof(*(out_block->data));
    out_block->mem_block_index = index;
    out_block->used = true;
}
```

Figure 3.2: Code for handling the loading of compressed memory blocks

3.1.3 Moving Data Across Two Different Blocks

The quantum gates used in QuEST involved moving data from one block of memory to another. This was done in the modified QuEST simulator by allowing two blocks to remain decompressed in memory at one time to avoid the need for repeated compression and decompression operations. The simple logic of the least recently used (LRU) determined which decompressed block was to be replaced when accessing a compressed block. In other words, the memory block that was accessed previously remained decompressed while the other memory block was replaced. Figure 3.3 below shows the underlying code used for finding which of the decompressed blocks to replace.



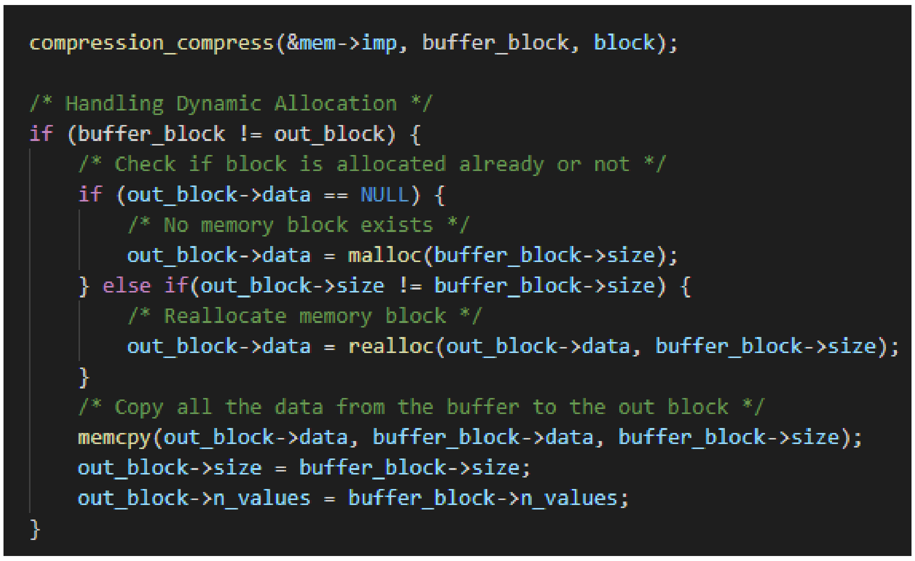
```c
int foundIndex = 0;
if (block->use_double_blocks) {
    foundIndex = -1;
    /* See if loaded blocks contain the data */
    for(int i = 0; i < 2; i++) {
        if(rawDataBlock_is_current_block(&block->decomp_blocks[i], index)) {
            foundIndex = i;
            break;
        }
    }

    /* Check if block not found. Then use the block that is not resently used */
    if (foundIndex == -1) {
        foundIndex = (block->lru_block + 1) % 2;
    }
}

DecompressedBlock *out_block = &block->decomp_blocks[foundIndex];
```

Figure 3.3: Code for handling the process of determining which block needs to be replaced in memory

3.1.4 Static & Dynamic Memory Allocation

The block memory structure allowed the memory to be allocated in one of two ways, either statically at the beginning of the program or dynamically when memory was first accessed. These two types of memory allocation resulted in different optimization effects. Static memory allocation had the potential to allow for faster execution time as there was no temporary storage required during the compression of the original state vector. Yet, dynamic memory allocation could result in a more effective compression since the space allocated at any given moment was only that which was indispensable for storing the current compressed data block.

During static memory allocation, a fixed amount of space was allocated at compile time, regardless of whether it was fully utilized. However, in dynamic memory allocation, a temporary storage block was allocated during compile-time and used as intermediary storage while a data block was being compressed. After compression, the size of the compressed data was used to allocate memory, and the data was copied into the correct memory block. Through this method, only the size of the currently compressed data was allocated for any given memory block. Figure 3.4 below shows the code for handling the dynamic allocation when saving data to a memory block.



```c
compression_compress(&mem->imp, buffer_block, block);

/* Handling Dynamic Allocation */
if (buffer_block != out_block) {
    /* Check if block is allocated already or not */
    if (out_block->data == NULL) {
        /* No memory block exists */
        out_block->data = malloc(buffer_block->size);
    } else if(out_block->size != buffer_block->size) {
        /* Reallocate memory block */
        out_block->data = realloc(out_block->data, buffer_block->size);
    }
    /* Copy all the data from the buffer to the out block */
    memcpy(out_block->data, buffer_block->data, buffer_block->size);
    out_block->size = buffer_block->size;
    out_block->n_values = buffer_block->n_values;
}
```

Figure 3.4: Code for handling the dynamic allocation when saving data to a compressed memory block

Whether to use static or dynamic memory allocation depended on the compression algorithm itself. In this case, ZFP supported both static and dynamic memory allocation, while FPZIP only worked with dynamic allocation. This limitation is because FPZIP does not provide a maximum estimate for the final size of the compressed memory, rendering it impossible to efficiently utilize static memory allocation without allocating the same size as the uncompressed data.

3.2 Testing of Data Compression

To be able to understand if data compression was a valid alternative for simulating larger quantum circuits several tests had to be performed. The tests were performed by simulating an implementation of Grover's search algorithm on the modified instance of QuEST. Memory usage and execution time were the main factors used when evaluating the effectiveness of the compression algorithms. Memory usage showed whether data compression allowed for the simulation of larger quantum circuits. It could also easily compare the performance of the two algorithms based on the ability to compress the state vector. The feasibility of integrating compression into QuEST was measured using execution time. If execution time was too large, the simulation of higher qubit circuits would be impractical, thereby deeming the integration of ZIP and FPZIP as infeasible in practice. A test script was created to collect the memory usage and execution time data. The test script's purpose was to ensure the tests were performed in controlled settings and with similar inputs. For memory usage, the test script used *heaptrack* to generate an output file. Next, *heaptrack_print* could extract the specific information about QuEST's peak memory utilization. To collect the data on execution time, the *perf_counter_ns* function within Python's *time* package was used. This tool calculated the difference between the start and the end of a test run in nanoseconds. The results collected from the tests were copied to a file to be parsed by a data processing script.

The new memory block structure also made it relevant to test a new independent variable, namely block size. The range of block sizes to be tested depends on the number of qubits used. If a block size was too big, the entire state vector would remain decompressed in memory, which led to no call to the compression algorithm. For example, for 15 qubits the maximum block size was $2^{15} - 1 = 32767$. Any value above the maximum size would cause the entire state vector to fit within the decompressed block. The effect block size had on memory usage was also considered. An increase in block size means that the decompressed memory block would also increase in size and potentially result in higher memory usage. As stated in Chapter 1, one of the main resource limitations when simulating quantum computers was the increase in memory requirement for each new qubit. For this reason, the number of qubits used in the quantum circuit was another independent variable tested. In order to test a higher number of qubits, one also has to test higher block sizes. For example, 10 different block sizes were tested for a quantum circuit running 16 qubits. While limiting the number of tests run, it still provided a clear account of how block size impacts execution time and memory usage. From Table 3.1 and Table 3.2 it is possible to see the different tests performed, and that for 16 qubits ten different block sizes were used.

Another aspect when testing the quantum simulator was the modes and compression sizes of ZFP and FPZIP. For ZFP, the tested modes were fixed-rate (-r), fixed-precision (-p), and fixed-accuracy (-a). fixed-rate meant that the compression algorithm would compress the original floating-points (64 bits) down to fixed data size. In this case, 32-bit or 16-bit representation. The fixed-precision parameter worked similarly to fixed-rate, the difference being that it would be able to compress the data even lower than the set limit. fixed-precision was tested with the same compression size as used for fixed-rate (32-bit and 16-bit). Lastly, fixed-accuracy compressed the data down to a representation that guaranteed a fixed level of accuracy. In this case, the tested parameters were four decimal accuracy (1e-4) or six decimal accuracy (1e-6). Table 3.1 illustrates the different modes and compression sizes used to test the ZFP implementation. ZFP was tested using both static and dynamic allocation (-d).

Table 3.1: Test parameters for testing ZFP compression. Here -p refers to precision, -r to rate and -a to accuracy.

Qubits	**Block Size**	**Allocation Type**	**Parameters**
10	32, 64, 128, 256	static, dynamic	-r 32, -r 16, -p 32, -p 16, -a 1e-4, -a 1e-6
11	64, 128, 256, 512	static, dynamic	-r 32, -r 16, -p 32, -p 16, -a 1e-4, -a 1e-6
12	128, 256, 512, 1024	static, dynamic	-r 32, -r 16, -p 32, -p 16, -a 1e-4, -a 1e-6
13	256, 512, 1024, 2048	static, dynamic	-r 32, -r 16, -p 32, -p 16, -a 1e-4, -a 1e-6
14	512, 1024, 2048, 4096	static, dynamic	-r 32, -r 16, -p 32, -p 16, -a 1e-4, -a 1e-6
15	1024, 2048, 4096, 8192	static, dynamic	-r 32, -r 16, -p 32, -p 16, -a 1e-4, -a 1e-6
16	32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384	static, dynamic	-r 32, -r 16, -p 32, -p 16, -a 1e-4, -a 1e-6

FPZIP had only one mode to test, namely precision mode (-p). In this case, the precision was set to 64-bit to ensure that FPZIP remained lossless. Table 3.2 illustrates the modes and settings used for the testing of the FPZIP. The testing of FPZIP was limited to only dynamic allocation.

Table 3.2: Test parameters for testing FPZIP compression

Qubits	**Block Size**	**Allocation Type**	**Parameters**
10	32, 64, 128, 256	dynamic	-p 64
11	64, 128, 256, 512	dynamic	-p 64
12	128, 256, 512, 1024	dynamic	-p 64
13	256, 512, 1024, 2048	dynamic	-p 64
14	512, 1024, 2048, 4096	dynamic	-p 64
15	1024, 2048, 4096, 8192	dynamic	-p 64
16	32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384	dynamic	-p 64

3.3 Determining Error Rate for Compression

Other than memory usage and execution time, one key aspect of evaluating the most effective compression algorithm was determining the error rate. The error rate refers to the fidelity between the state vectors resulting from QuEST without compression and QuEST with compression integration. While lossy compression may lead to lower memory usage, it may also cause data loss, altering the final results. The error rate was a tool to depict the severity of this loss. If fidelity was too low, the benefits of improved memory usage would be irrelevant as the data would be ineffectual. Calculating the error rate also helped to compare the performance of the lossless and lossy compression algorithms. The fidelity function took two vectors and returned a number between 0 and 1. This value indicated the similarity between the two vectors in percentage form. Fidelity as a measure of error rate allowed one to determine the similarity between the state vectors of the unmodified QuEST compared to the modified version.

3.4 GPU Acceleration of the Simulator

Integrating compression logic into the simulator could increase execution time, and therefore GPU acceleration was a way to mitigate this problem. It involved offloading all of the qubit operations performed by the simulator to the GPU, allowing for a higher level of parallel executions. In the original implementation of the QuEST simulator support already existed for GPU acceleration through CUDA. Because of this the new GPU acceleration code would build on the existing implementation within QuEST.

When integrating GPU acceleration, the existing code was modified to handle the new memory structure, and to be able to compress and decompress the data every time a new memory block needed to be accessed. Due to the new block memory structure, the maximum number of parallel executions was thereby limited to the total number of values within a memory block. The reason for this is that the qubit operations could only be performed on decompressed values of the state vector.

In addition to the qubit operations, compressing and decompressing blocks were also offloaded to the GPU. This was integrated into QuEST easily, as CUDA already supported ZFP with an implementation to accelerate compression and decompression. However, this implementation only supported fixed-rate compression. Due to the lack of support, fixed-precision mode, fixed-accuracy mode, as well as FPZIP were not tested using GPU acceleration. While only testing the fixed-rate compression did not provide the complete picture, it still allowed one to see the impact of GPU accelerated code on execution time.

To run the GPU accelerated implementation it was only a matter of compiling the modified QuEST simulator with a specific flag that compiled the CUDA implementation of the simulator. The CUDA implementation was also limited to only using the VRAM of the GPU for storing the state vector. This was because the existing implementation in QuEST used older version of the CUDA API that did not utilize unified memory (where data is shared between RAM and VRAM).

3.4.1 Testing GPU Acceleration

Tests allowed one to study the impact of GPU acceleration on execution time. The tests ran ZFP with a fixed-rate mode of 32- and 16-bit representation. Through the results, it was possible to determine the speedup that occurred when using GPU acceleration as compared to using the original fixed-rate mode. Table 3.3 shows the list of parameters used to test the GPU accelerated ZFP implementation. As can be seen, the number of qubits tested increased to 18 qubits to show how GPU acceleration handles larger qubit sizes.

Table 3.3: Test parameters for testing ZFP compression with GPU acceleration

Qubits	Block Size	Allocation Type	Parameters
10	32, 64, 128, 256	static, dynamic	-r 32, -r 16
11	64, 128, 256, 512	static, dynamic	-r 32, -r 16
12	128, 256, 512, 1024	static, dynamic	-r 32, -r 16
13	256, 512, 1024, 2048	static, dynamic	-r 32, -r 16
14	512, 1024, 2048, 4096	static, dynamic	-r 32, -r 16
15	1024, 2048, 4096, 8192	static, dynamic	-r 32, -r 16
16	32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384	static, dynamic	-r 32, -r 16
17	4096, 8192, 16384, 32768	static, dynamic	-r 32, -r 16
18	8192, 16384, 32768, 65536	static, dynamic	-r 32, -r 16

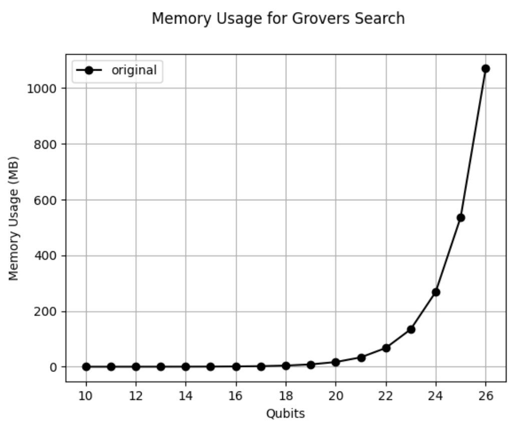
All the tests ran on a dedicated computer, to ensure the same hardware configuration. The computer was a Linux machine running Ubuntu 20.04.04 LTS, on an AMD Ryzen 5 1600x processor with 16 GB of RAM. The same hardware configuration was used for the GPU accelerated simulator, alongside an Nvidia GTX980 Ti graphics card.

4
Results & Discussion

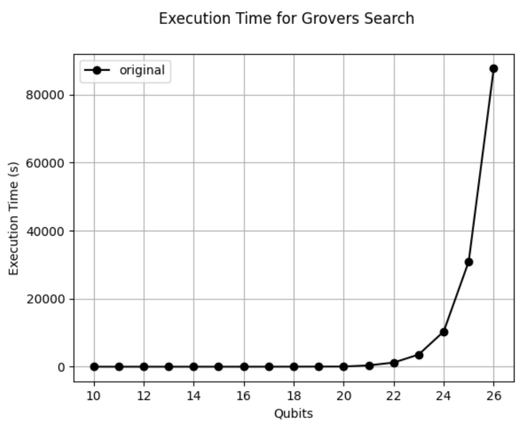
This chapter will provide the data resulting from tests of QuEST integrated with data compression algorithms (ZFP and FPZIP) and QuEST with GPU acceleration. Throughout this chapter, the different compression modes will be explored and compared to determine the optimal settings with regard to memory utilization, execution time, and fidelity.

4.1 Original QuEST

In order to fully comprehend the results derived from the integration of the two compression algorithms as well as GPU acceleration, one must first look at the performance of the original QuEST without any integrated optimization techniques. Figure 4.1 displays how both the amount of memory utilized and execution time scales exponentially with the number of simulated qubits. When considering the results from using data compression and GPU acceleration, this is the benchmark for determining the effectiveness of the solutions.



(a)



(b)

Figure 4.1: Memory usage (a) and execution time (b) of running Grover's search algorithm on QuEST with no data compression

4.2 ZFP

As discussed in Chapter 2, to integrate compression into QuEST, a new block-structured memory was defined. The new block structure introduced a new independent variable, namely block size. Figure 4.2 and Figure 4.3 show the effect block size has on ZFP compression and ultimately how that influences the performance of Grover's search algorithm in terms of execution time and memory usage.

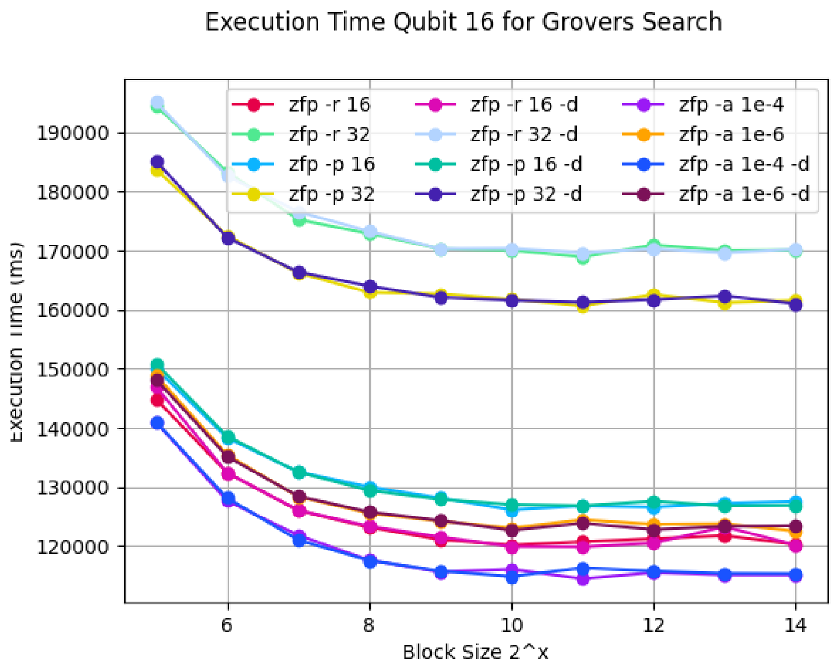


Figure 4.2: Execution time at 16 qubits over the block sizes 32-16384 floating-points values for ZFP compression modes, fixed rate (-r), fixed precision (-p), fixed accuracy (-a) and dynamic allocation (-d).

Figure 4.2 illustrates how the execution time changes with respect to the increase in block size for all ZFP modes at 16 qubits. As shown, the execution time decreases until around 2^{10}, after which it remains almost constant as a result of the increased overhead from more calls to compress and decompress blocks. For example, at a block size of 2^5, there are a total of 4096 blocks (2048 for each part of the complex state vector), while at a block size of 2^{10}, the total number of blocks is only 128. This difference impacts how many calls to compression and decompression occur during execution. One can also see from the graph that the pattern for execution time is the same across all modes of ZFP, further supporting the theory that execution time is affected by the increased number of blocks resulting from low block sizes. Another conclusion that can be drawn, is that there is very little difference in execution time whether one chooses to use dynamic or static memory allocation. The reason for this is described in more detail when discussing figure 4.7 below.

Other than the effect on execution time, the block size may also impact total memory usage. Figure 4.3 depicts this relationship and shows that optimal memory usage is achieved at a block size of 2^9. Two factors can explain the U-shaped relationship between block size and memory usage. While the decompressed memory blocks are small for block sizes from 2^5 to 2^7, the overhead when creating these memory blocks is significant. For each newly created memory block, 32 bytes of data are allocated, irrespective of block size. For example, at block size 2^5 there are 4096 compressed memory blocks which result in an overhead of 131072 bytes. The overhead grows considerably when many blocks are needed to represent the compressed data. The second factor that impacts memory usage is the size of the decompressed memory block. As the block size increases, so does the memory requirement necessary to keep a block decompressed in memory. For example, at block size 2^{13}, the total memory required to represent the decompressed data is 262144 bytes (each floating-point value is 8 bytes), while at a block size of 2^9, the decompressed data size is only 16384 bytes. Therefore, the increase in memory block size negatively impacts the simulator's total memory requirement.

From analyzing the results, one can see that the optimal block size for 16 qubits is 2^9 and 2^{10}. At both these block sizes, the execution time and memory usages are at an optimal level for the different modes.

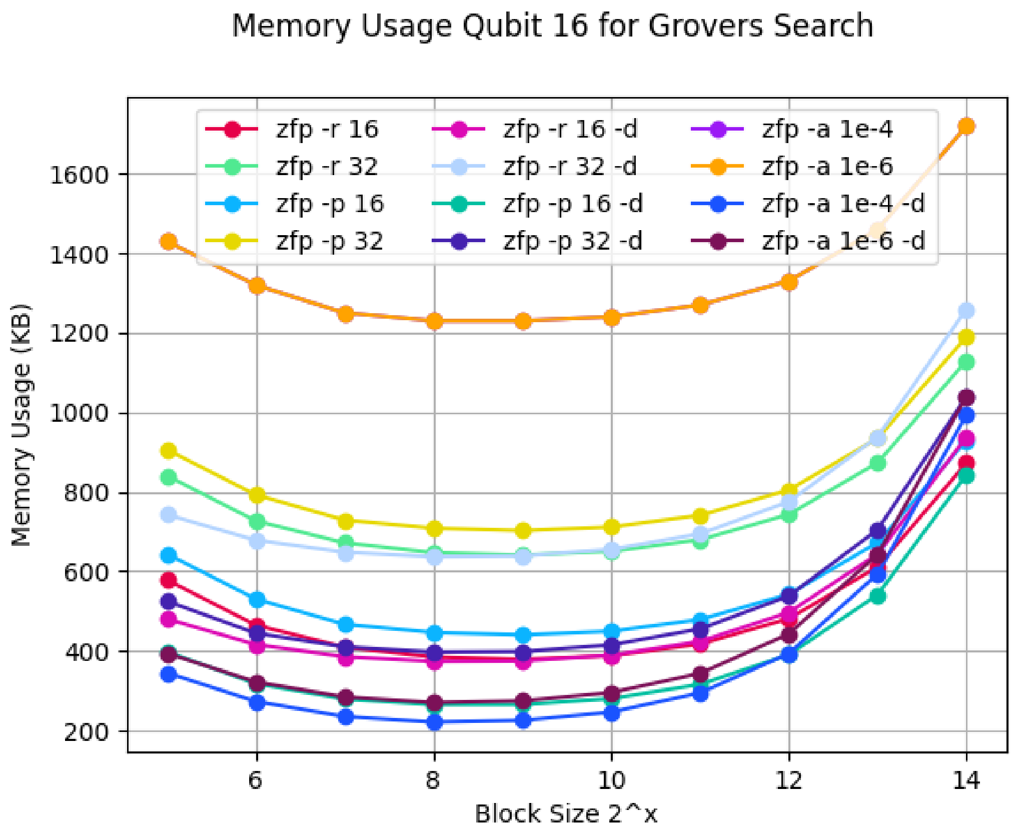
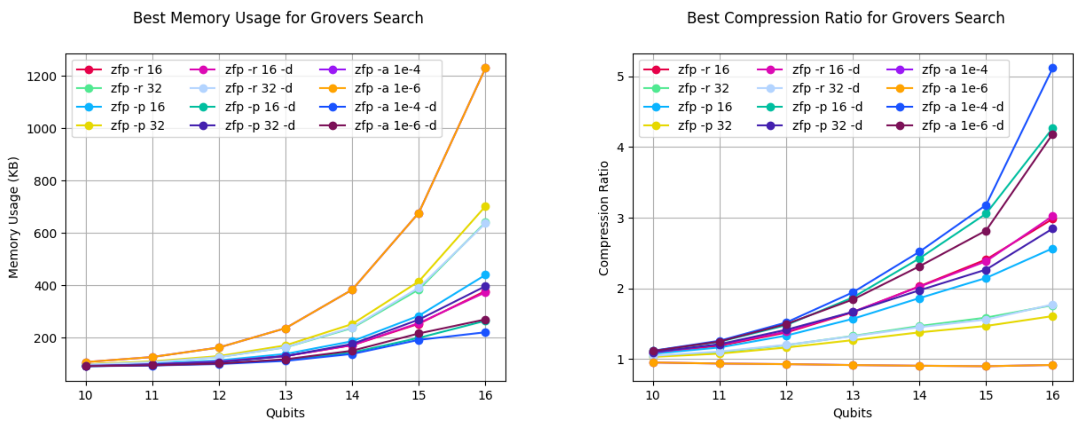


Figure 4.3: Memory used at 16 qubits over the block sizes 32-16384 floating-points for ZFP compression modes, fixed rate (-r), fixed precision (-p), fixed accuracy (-a) and dynamic allocation (-d).

4.2.1 ZFP & Number of Qubits

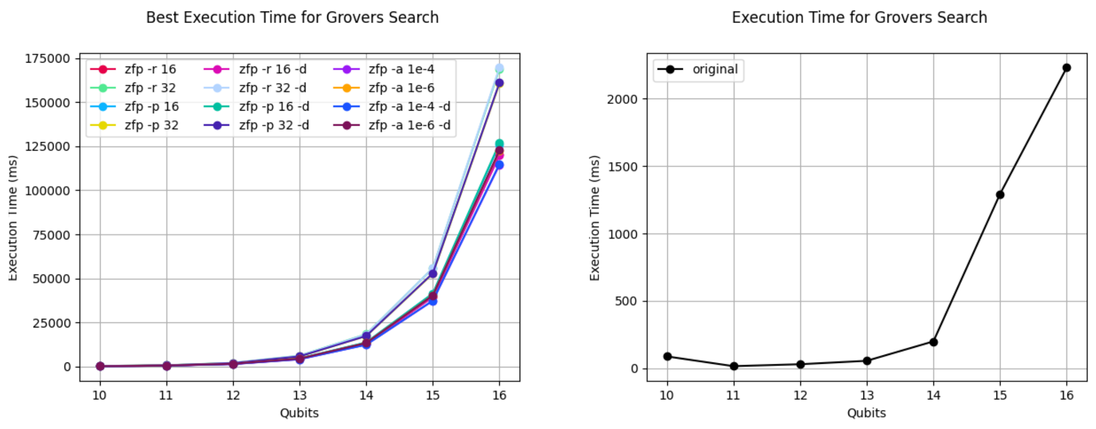
Figure 4.4 illustrates how the number of qubits affect the memory usage and the compression ratio for different modes of ZFP. By analysing the compression ratio in Figure 4.1 (b), one can see that all modes except *zfp -a 1e-6* achieve improved memory usage compared to the original implementation of Grover's search algorithm. The modes that achieve the best level of compression are *zfp -a 1e-4 -d*, *zfp -a 1e-6 -d*, and *zfp -p 16 -d*. These modes are able to achieve a compression ratio above 4 when running 16 qubits, which indicates that it may allow for the simulation of quantum circuits with 2 more qubits on the same hardware.



(a) (b)

Figure 4.4: Memory usage in KB (a) and compression ratio (b) per qubit when running Grover's search algorithm. Fixed rate (-r), fixed precision (-p), fixed accuracy (-a) and dynamic allocation (-d)

Figure 4.5 (a) shows how execution time is affected by the number of qubits. Following a similar pattern of growth as seen in Figure 4.4 (a), the execution time for all the ZFP modes increases exponentially. From the graph, one can also see that the best performing solution is to use *zfp -a 1e-4 -d*. Figure 4.5 (b) shows the execution time of the original implementation of Grover's search algorithm. By comparison, one can see that the execution time when running with compression is much higher than the original. In most cases, this difference is quite significant. For example, for *zfp -r 32 -d* the execution time increased from around 2200 to 170000 milliseconds for 16 qubits, which is approximately 77 times greater than the original. This may indicate that even if compression can decrease memory usage, it may lead to infeasible execution times for simulations with a high number of qubits. Infeasible, in this context, meaning an execution time that takes days or even weeks to complete a simulation.



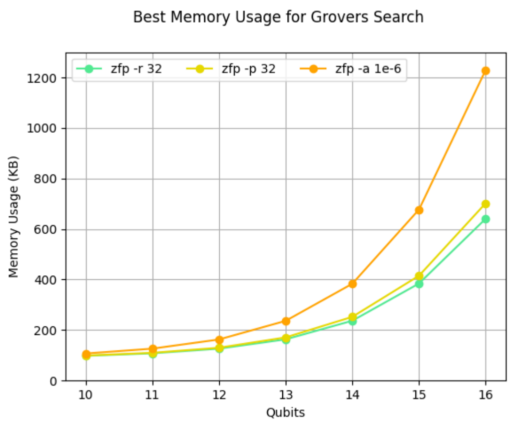
(a) (b)

Figure 4.5: Execution time in ms (a) and original execution time in ms (b) per qubit when running Grover's search algorithm. Fixed rate (-r), fixed precision (-p), fixed accuracy (-a) and dynamic allocation (-d)

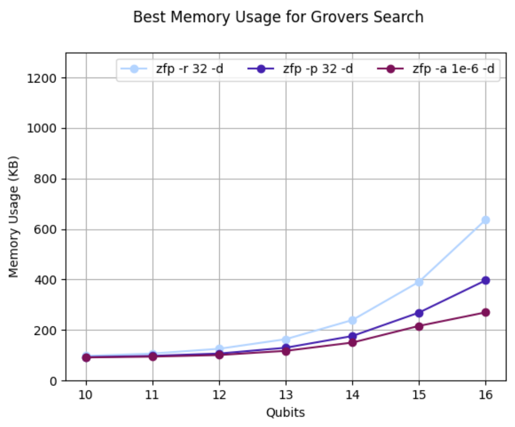
4.2.2 Comparing modes of ZFP

As observed, the modes of ZFP are able to achieve varying degrees of good memory utilization and execution time. In order to choose decide which is the best performing mode, one must further analyze and compare the various compression methods. Figure 4.6 shows how the different modes impact the memory usage. The graphs reveal that if one wants to use static memory allocation, ZFP fixed-rate mode achieves the best memory usage. However, if instead dynamic memory allocation is used, fixed-precision and fixed-accuracy both outperform fixed-rate compression. One explanation is that fixed-rate mode always compresses the data down to a fixed size, even if it has the potential to be represented using a smaller data size. Contrary to this, fixed-precision and fixed-accuracy mode are able to dynamically compress the data to a smaller representation when it is more uniform. An example of uniform data is when the values are all zeros. Given the improved compression ratio for dynamic allocation the underlying state vector must provide certain level of uniformity. Because of this uniformity in the data, both fixed-precision and fixed-accuracy were able to compress the data further than fixed-rate compression. From this, it can be concluded that it is optimal to use fixed-precision and fixed-accuracy modes when using dynamic allocation.

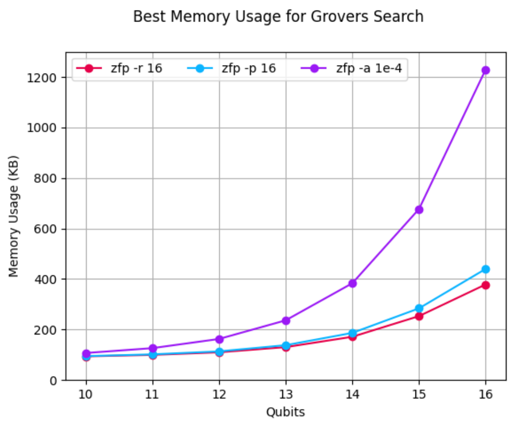
Besides the benefit of dynamic allocation, one can also see that different accuracy levels and compression sizes may also positively impact memory usage. For example, Figure 4.6 shows that a compression size of 16-bit (*zfp -r 16 -d* and *zfp -p 16 -d*) and an accuracy of 4 decimals (*zfp -a 1e-4 -d*) are able to achieve a much higher level of compression compared to using 32-bit compression size or 6 decimals accuracy. This is expected, given that a lower compression size and accuracy rate will also lead to lower memory usage. When simulating 16 qubits, fixed-rate compression can decrease the compression size from 638KB down to 374KB, fixed-precision compression size decreases from 397KB to 264KB, and fixed-accuracy compression size decreases from 270KB to 220KB. The largest improvement is seen for fixed-rate mode, 32-bit to 16-bit representation. On the other hand, for fixed-accuracy has no significant reduction in memory usage when the accuracy is lower. A lower compression size and accuracy may also lead the underlying error rate of the data to increase. Therefore the best possible solution with regards to memory usage might be to use *zfp -a 1e-6 -d* given the effective compression ratio and the potential for maintaining a higher accuracy.



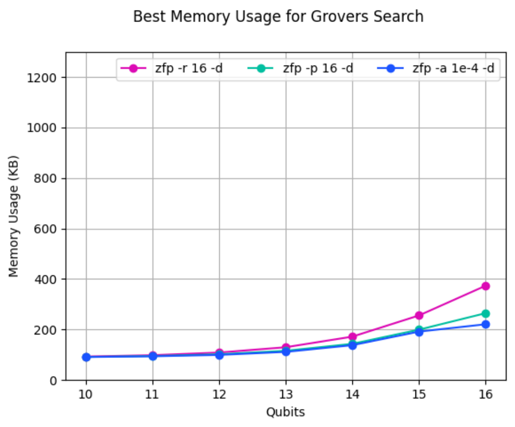
(a)



(b)



(c)



(d)

Figure 4.6: Memory in KB per qubit comparing the ZFP modes, fixed-rate, fixed-precision, and fixed-accuracy compression (a) static and (b) dynamic allocation for 32-bit precision and 6 decimal point accuracy, (c) static and (d) dynamic allocation for 16-bit precision and 4 decimal point accuracy

Figure 4.7 shows the results of execution time when running the different modes of ZFP. One can observe that, independently of fixed-rate, fixed-precision, or fixed-accuracy mode, the execution time when running dynamic and static memory allocation is almost identical. This indicates that the added execution step of copying data from the temporary storage block into the memory block during dynamic allocation has only a negligible impact on the overall execution time. As such, one can conclude that dynamic allocation is most appropriate due to its ability to decrease the overall memory usage without causing a negative impact on the total execution time.

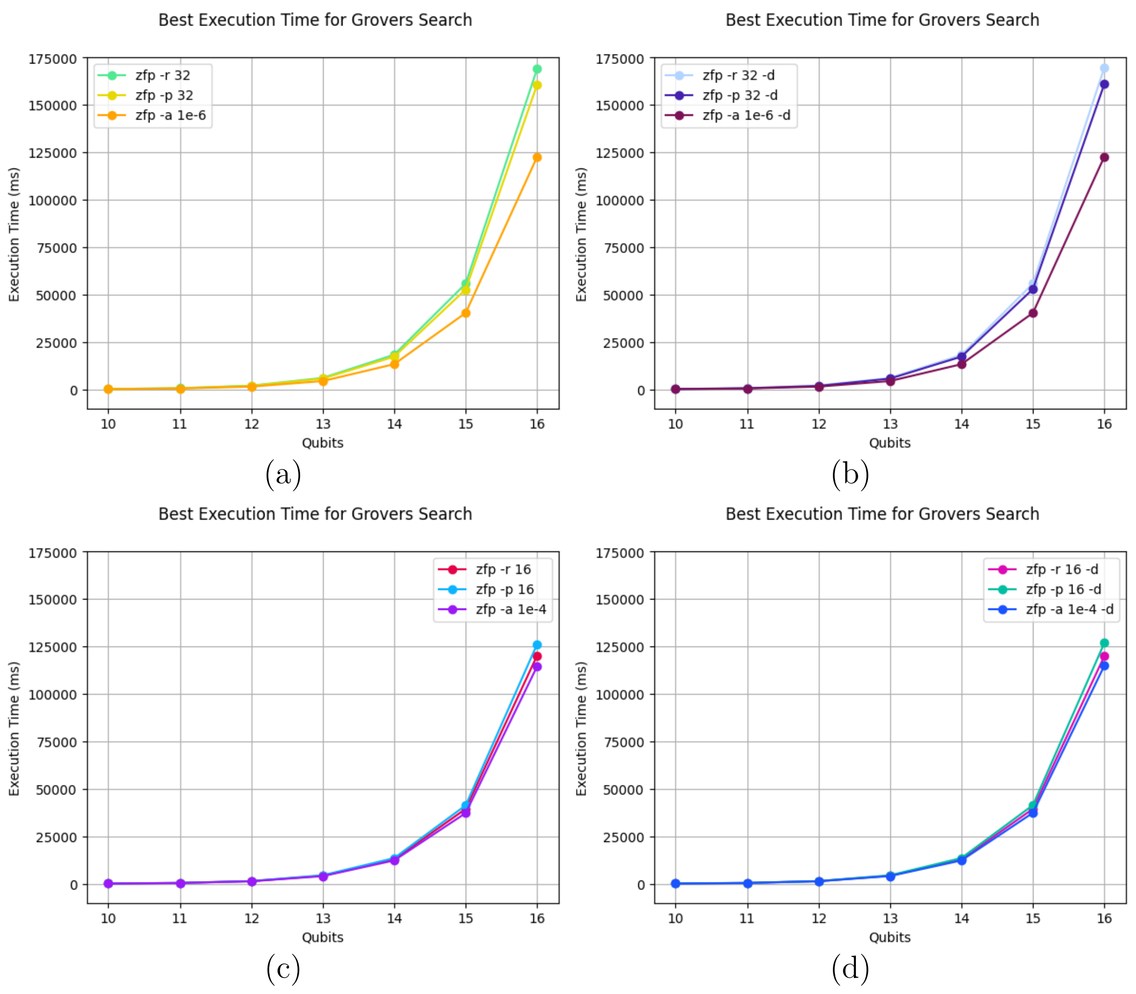
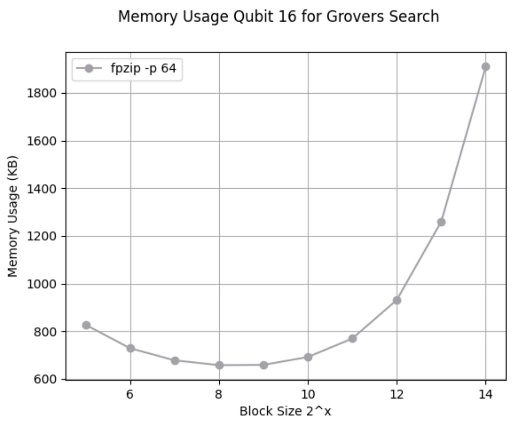


Figure 4.7: Time in ms per qubit comparing the ZFP's precision compression to fixed-rate using dynamic memory allocation for (a) 32 bit, (b) 16 bit floating-point were (c) and (d) use dynamic memory allocation

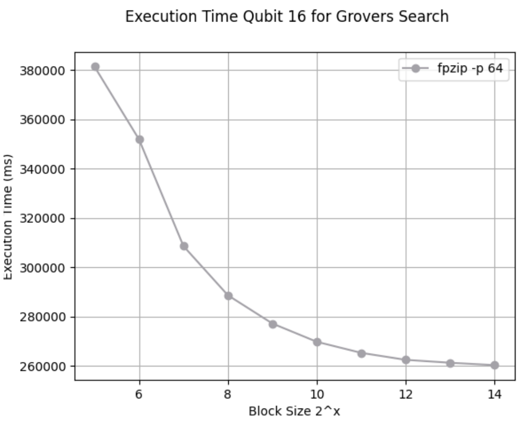
Figure 4.7 shows that fixed-accuracy mode performs the most effectively of the three modes, no matter whether static or dynamic memory allocation is used. Figure 4.7 (a) and (b) reveals that at a compression size of 32-bit and an accuracy of 6 decimals, the difference between the execution time of fixed-accuracy versus fixed-rate and fixed-precision mode is more significant. For example, when running with 16 qubits, fixed-accuracy mode takes approximately 122 000 milliseconds to execute, while fixed-rate mode takes about 170 000 milliseconds to finish execution. From Figure 4.7 (c) and (d), one can also observe that using 16-bit compression size or an accuracy of 4 decimals leads to a lower execution time. However, all modes of ZFP compression increase the execution time significantly, seen in Figure 4.5 (a) and (b). For example, at 16 qubits, the execution time for *zfp -r 32 -d* is 170 000 milliseconds compared with 2200 milliseconds in the original implementation. This increase in execution time indicates that running ZFP compression might not be a feasible solution in practice. At a high number of qubits, the execution time might be too large to justify using compression to decrease memory usage.

4.3 FPZIP

The following results describe how the use of lossless compression affects the performance of Grover's search algorithm in terms of total memory utilization and execution time. Figure 4.8 (a) shows how memory usage and execution time change as the block size increases. In the same way as ZFP, the increase in block size has resulted in a kind of U-shaped trend. When the block size is 2^5, the amount of memory used is just above 800KB. As the block size increases, memory usage decreases until about 2^9, which is the optimal block size, using approximately 650KB of memory. The decrease in memory usage can be explained using the same logic as described when discussing how block size affects memory in ZFP. While a small block size means smaller blocks in the decompressed memory, it also means a greater amount of blocks are created. The creation of the blocks results in a significant amount of overhead. From 2^9, the increasing block size begins to have a negative impact on memory usage. For example, at block size 2^{14}, about 1950KB of data is utilized, or three times as much as when the block size is 2^9. FPZIP only uses dynamic allocation, so the size of the temporary compression block stored in memory is the same as the block size, meaning that as the block size increases, so does the memory usage.



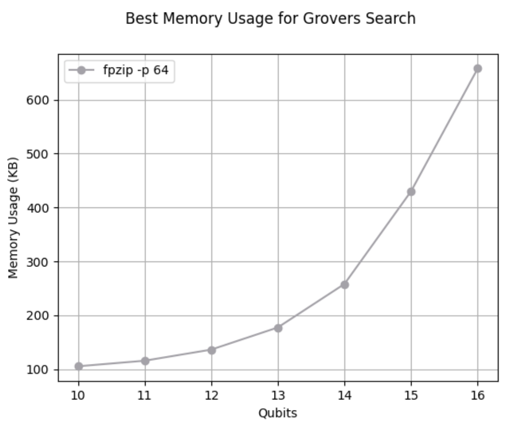
(a)



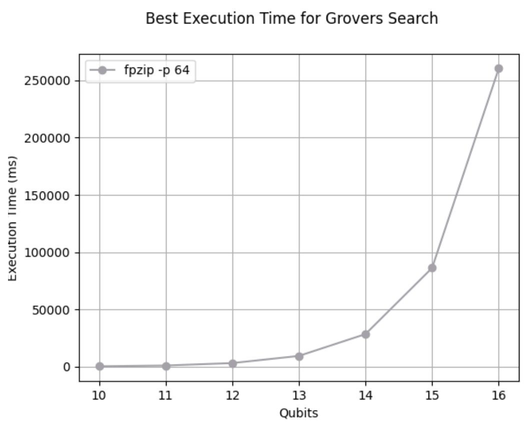
(b)

Figure 4.8: Memory usage (a) and execution time (b) of running Grover's search algorithm on QuEST with FPZIP compression in range of 2048-16384 floating-points per block size for 16 quits

Figure 4.8 (b) shows execution time in relation to the increasing block size. The impact block size has on the execution time for FPZIP does not follow the same linear pattern as in ZFP. The graph shows that execution time decreases as the block size increases. One explanation may be that the FPZIP algorithm can optimize the compression of larger quantities of data and is, therefore, more efficient at compressing the data. In conclusion, the optimal block size for lossless compression is not as apparent as for ZFP. Simply by looking at how block size affects memory usage, one would choose 2^9 as the most effective size. In terms of execution time, 2^{14} would be the optimal choice, however it has a significant negative impact on memory utilization.



(a)



(b)

Figure 4.9: Memory usage (a) and execution time (b) per qubit running Grover's search algorithm on QuEST with FPZIP compression

Another aspect to analyze from the results of FPZIP is how memory usage and execution time increases with the number of qubits. Figure 4.9 illustrates the results for FPZIP and how it performs in comparison with the original QuEST implementation. Much like ZFP, memory usage and execution time follow an exponential growth pattern as the number of qubits increases. The graphs show that FPZIP can effectively compress the data. For example, at 16 qubits, approximately 658KB of memory is being used, which is a significant decrease compared to the original QuEST that used 1.13MB. FPZIP can decrease the memory usage by 30%, similar to that of *zfp -r 32*. However, the execution time for FPZIP increases at a much faster rate than the original QuEST implementation. Given the risk of too high execution times, FPZIP may prove to be infeasible when running a higher number of qubits.

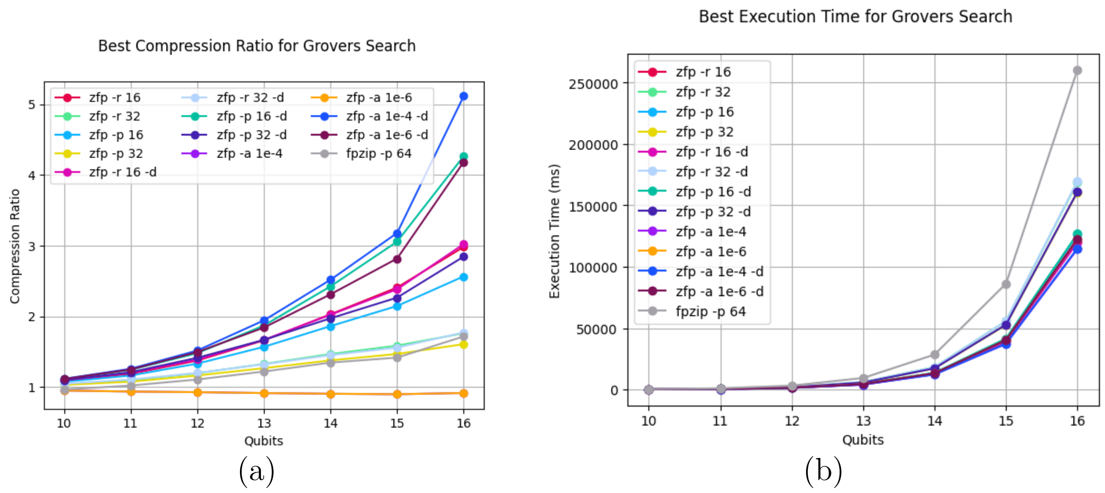


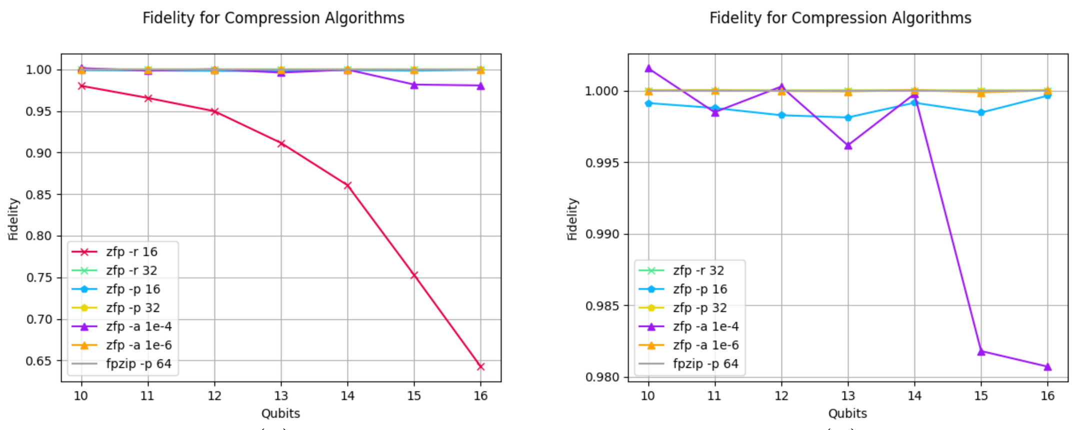
Figure 4.10: Compression ratio (a) and execution time (b) per qubit for all compression algorithms and modes running Grover's search algorithm on QuEST. Fixed rate (-r), fixed precision (-p), fixed accuracy (-a) and dynamic allocation (-d)

Figure 4.10 shows how the performance FPZIP compares to that of ZFP. From Figure 4.10 (a), one can see that FPZIP has around the same compression ratio as ZFP modes using 32 bit fixed-rate compression, but it falls far behind the other better performing ZFP modes. Figure 4.10 (b), reveals that FPZIP has the highest execution time compared to all other modes of ZFP. This, most likely, is due to FPZIP working will the full 64 bit values during compression compared to the modes of ZFP that only have to do computations on 32-bits or 16-bits. Based on this fact alone, one can claim that ZFP is the better option. However, in order to make a final conclusion, the fidelity of ZFP and FPZIP must also be considered.

4.4 Fidelity of ZFP and FPZIP

This section analyzes how well the compression algorithms perform in terms of the fidelity of the state vector. Figure 4.11 (a) illustrates the fidelity with respect to the number of qubits for the different modes of ZFP and FPZIP. From the graphs, one can see that the fidelity of *zfp -r 16* drops significantly for every qubit. In fact, from the downward trend, one can predict that as the qubits continue to increase, the error will also increase. The compression provided by *zfp -r 16* is made irrelevant by the fact that the data changes substantially.

Figure 4.11 (b) shows the behavior of the other compression modes more clearly by excluding the ZFP mode, *zfp -r 16*. There is no visible change in the fidelity as most of the parameters remain close to 100%. Only for *zfp -p 16* and *zfp -a 1e-4* is there a visible fluctuation. This level of fluctuation indicates that *zfp -p 16* and *zfp -a 1e-4* may achieve a lower fidelity for higher qubits. However, even by excluding these three modes, it is possible to see that ZFP can achieve a near similar level of fidelity as FPZIP. It suggests that some modes of ZFP can provide a higher compression ratio with an equal level of fidelity as FPZIP. From this, the optimal solution for memory usage, execution time, and fidelity would be *zfp -a 1e-6 -d*.



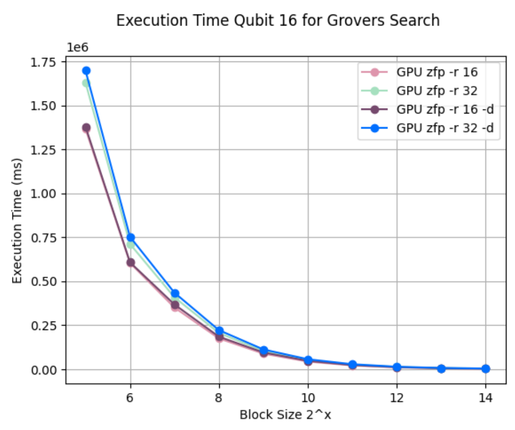
(a) (b)

Figure 4.11: Illustration of the underlying fidelity for the different algorithms (a) all results for fidelity and (b) fidelity results without *zfp -r 16*

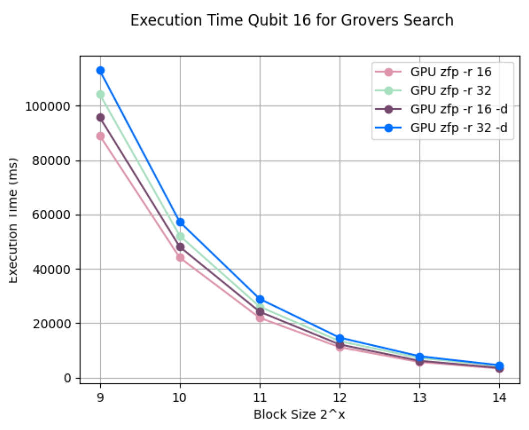
4.5 GPU Acceleration

Throughout this chapter, one concerning aspect has been the execution time. As shown in Figure 4.5, execution time grows more significantly than the original QuEST implementation. In turn, the risk of unreasonable execution times for running a high number of qubits increases. One possible solution to address this problem would be to offload the calculations from the CPU to the GPU.

Firstly, Figure 4.12 shows the correlation between block size and execution time and how it is impacted by GPU acceleration. The increase in block size leads to a decrease in the execution time. This trend is due to an increase in block size which allows for a higher level of parallelism, as more values are decompressed in memory and thereby can be computed in parallel.



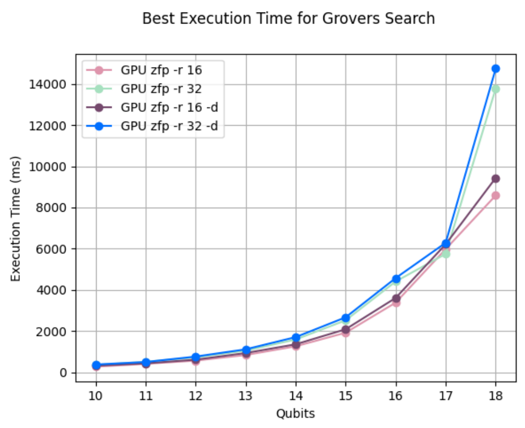
(a)



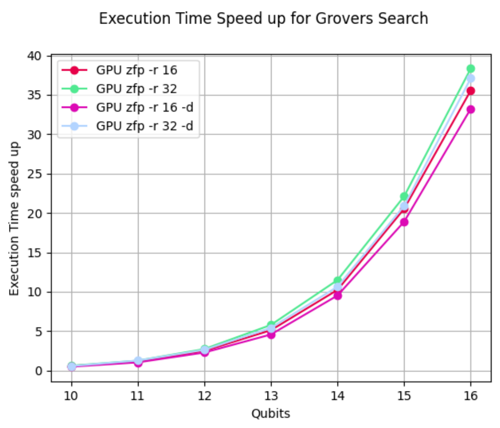
(b)

Figure 4.12: Execution time on the GPU as function of block size for 16 qubits (a) complete block range and (b) last 6 block ranges

Figure 4.13 (a) shows how execution time is impacted for ZFP in fixed-rate mode. The graph describes the trend as the number of qubits increases. The conclusion drawn from Figure 4.12, is that the largest block size results in the fastest execution time. Figure 4.13 (a), depicts the qubits using the largest block size. From the provided speed-up of using GPU acceleration, shown in Figure 4.13 (b), one can see that execution time has decreased significantly. For mode *zfp -r 32 -d* using 16 qubits, execution time has decreased from 170000 to 4500 milliseconds, an improvement of about 38 times. The same rate of improvement can be seen from the other modes run on the GPU. Figure 4.13 (a) also shows that execution time has decreased compared to the original implementation of QuEST. While execution time was a big dilemma after integrating ZFP and FPZIP, these results indicate that by using GPU acceleration the compression algorithms could provide a feasible solution with a reasonable execution time.



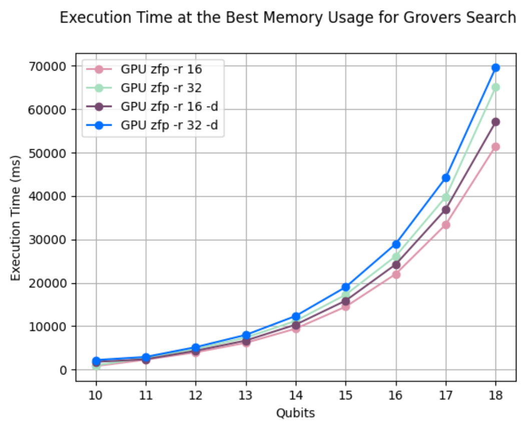
(a)



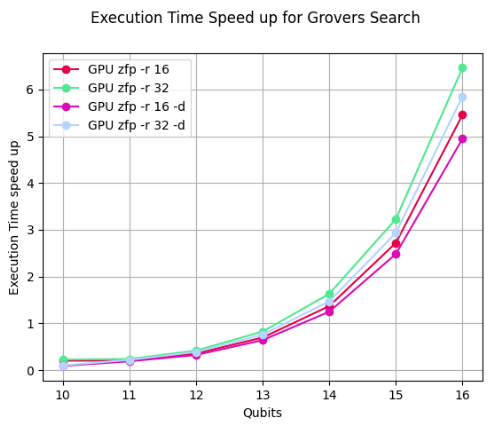
(b)

Figure 4.13: Best execution time (a) and speed up for best execution time (b) on the GPU. Fixed rate (-r), and dynamic allocation (-d)

However, drawing a conclusion purely based on execution time and Figure 4.13 (a) creates a skewed view of the results. The figure shows the impact at the largest possible block size (which, as previously stated, significantly improves execution time), however, this will have an adverse effect on memory usage. In fact, the memory usage will be greater than ZFP without GPU acceleration, as seen in Figure 4.4. To gather a complete picture of the results, Figure 4.14 illustrates how execution time and speed-up are impacted when the block size with the lowest possible memory usage is used. Here one can see that instead of a reduction in execution time by around 38 times for 16 qubits, it is decreased to around 5.5 times, which is a much lower improvement.



(a)



(b)

Figure 4.14: Execution time at the best memory usage (a) and speed up of the execution time (b) on the GPU. Fixed rate (-r), and dynamic allocation (-d)

4.5.1 Limitations of GPU acceleration

Executing the simulator on the GPU has the potential to improve the execution time but could also limit the possible memory usage. In the current implementation of the GPU accelerated simulator, the computations and memory operations are restricted by the size of the GPU's VRAM. However, GPUs tend to have much smaller amounts of VRAM when compared to the RAM available in most modern computers. For example, a modern computer may have a RAM size of 16 GBs, while the VRAM on the graphics card is around 6 GBs. As such, the current implementation of GPU accelerated code is limited to only running circuits that are small enough to fit inside the available VRAM of the GPU. A possible solution would be to utilize unified memory during the execution of the simulator, where memory is shared between VRAM and RAM. This would utilize the high level of parallelism in the GPU while simultaneously avoiding the GPUs memory limit.

4.6 Determining the feasibility of compression

From the results, one can conclude whether data compression is a feasible alternative for running larger quantum simulations using the same hardware. One must consider how the memory usage will scale with the number of qubits. Figure 4.10 (a) reveals that all implementations except *zfp -a 1e-6* are able to decrease memory usage. The best alternative among the implementations are *zfp -p 16 -d*, *zfp -a 1e-4 -d* and *zfp -a 1e-6 -d* which are able to achieve a compression ratio above 4× for 16 qubits. If this ratio were to be maintained for higher qubits, it would theoretically allow for the simulation of circuits with 2 more qubits using the same hardware configuration. The results of ZFP and FPZIP also show that ZFP, on average, can achieve a lower compression size than FPZIP. Thus, it can be concluded that lossy compression is the best alternative to achieve a lower compression size.

Although it is theoretically possible to simulate larger quantum circuits, in practice it might not be feasible given the potential for high execution time. Figure 4.10 (b), shows the growth of execution time for the different compression modes. Comparing these results with the execution time of the unmodified QuEST, shown in Figure 4.5 (b), one can see that the increase is significant. Even the best performing implementation, *zfp -a 1e-4 -d*, still has an execution time for 16 qubits that is more than 50 times higher than the original execution time. For larger qubits, this indicates that the execution time may prove infeasible and thereby make it impractical to utilize data compression.

Due to the impractical nature of running the simulations on the CPU, utilizing the GPU could improve the performance. Figure 4.13 (a) illustrates the execution time for the GPU accelerated implementation. The execution time remains higher than the original implementation, but the difference is significantly less than the simulation using compression without GPU acceleration. For example, at 16 qubits the GPU accelerated implementation of *zfp -r 16 -d* is only 1.5 times slower than the original. The decrease in execution time shows promise in improving the feasibility of using compression with the help of GPU acceleration. The GPU works most efficiently when there is a higher level of parallelism, which means it works best when the block size is large. However, the large block size also leads to an increase in the compression size. Figure 4.14 (a) shows instead execution time when the best memory usage is the main priority. Here, the execution time for *zfp -r 16 -d* has increased at 16 qubits from 1.5 times to around 10 times slower than the original. This is still an improvement over the CPU implementation and could potentially allow for the execution of larger quantum circuits with the help of compression and GPU acceleration.

Finally, when determining the feasibility of using compression, one must consider fidelity. The fidelity for each of the implementations is presented in Figure 4.11. The two graphs show that most of the algorithms maintain a fidelity near 1.0. However, *zfp -r 16*, *zfp -a 1e-4*, and *zfp -p 16* diverge from the fidelity of 1.0. The mode *zfp -r 16* even rapidly decreases in fidelity after just 12 qubits. This rapid decrease in fidelity gives a strong indication that *zfp -r 16* is not a feasible alternative given the potential for inconsistent results at higher qubits. There are also fluctuations in *zfp -a 1e-4* and *zfp -p 16*, which indicates that the implementations might deviate from 1.0 and provide a much lower fidelity at higher qubits. This lower level of fidelity increases the risk that the underlying state vector representation becomes inconsistent, thereby leading to unreliable results.

In conclusion, *zfp -a 1e-6 -d* is the best solution for data compression. It can provide a memory usage level 4 times smaller than the original memory size at 16 qubits and lead to a better execution time than almost all other ZFP modes. It also maintains fidelity at around 1.0. By potentially creating a GPU accelerated implementation of *zfp -a 1e-6 -d*, it may become a practical solution for using data compression in quantum computer simulators.

4.7 Ethics

The possibility of powerful and accessible quantum computers is certainly a highly contentious issue from an ethical standpoint. The benefits of the technology are obvious, solving tasks that would be much more difficult for a classical computer such as optimizing machine learning models [34] and simulating protein-folding [35], providing further insight into vaccine development and Alzheimer's disease. However, what could be the results if quantum computers were to be used malevolently? Today, most personal online data such as messages, passwords, and bank records, are protected by encryption methods. Most of these algorithms depend on not being able to do extremely large-scale computations [4]. If quantum computers became available, the users of these machines would potentially be able to access all of this previously secure data, clearly an immense safety concern.

No external individuals were directly affected throughout this project. One could argue, however, that a likely outcome of the project is to accelerate the science of quantum computers and whether or not the development of quantum computers is ethical due to their destructive potential. The potential impacts it could have on society are indeed almost limitless [34] however, the development of these systems will happen regardless of the outcomes of this study. Primarily, the goal of optimizing quantum simulators is to make it more accessible for the everyday programmer to learn about quantum computing. The goal is not to accelerate the development of state-of-the-art quantum computers at companies like IBM or Google. With this in mind, this study did not have any significant impact on society or significant ethical ramifications.

5 Conclusion

This chapter presents some final conclusions regarding the compression and GPU acceleration. Also presented are some final thoughts regarding error sources and potential for improvement.

5.1 Project Conclusion

This report presented the results of combining QuEST with the ZFP and FPZIP compression algorithms to reduce total memory consumption when simulating Grover's search algorithm. Furthermore, the use of GPU acceleration was analyzed to determine if it would lead to better execution time. From the data, it is possible to conclude that using lossy compression with ZFP may be more favorable than using lossless compression with FPZIP. Among the different ZFP modes, using fixed-accuracy mode with 6 decimal accuracy (*zfp -a 1e-6 -d*) has shown the most promising results in regards to memory usage, execution time, and a high level of fidelity.

To conclude, the results show that the limiting factor of data compression is the high execution time. The execution time of QuEST using compression was more than 50 times greater than the execution time of the original QuEST implementation. Using GPU acceleration proved to decrease this difference and could potentially be an alternative for running the QuEST simulator with data compression. Assuming that feasible execution time can be achieved and that the compression ratio is maintained, hypothetically it would be possible to run quantum circuits with 2 more qubits for the same hardware configuration. This is under the assumption that *zfp -a 1e-6 -d* is used with a compression ratio of 4.

5.2 Possible Sources of Error

One factor that could affect the testing is the nature of floating-point numbers on digital computers. Some precise floating-point numbers have decimals that stretch further than the range that double-precision floating-point numbers do. This means that the number has to be rounded or cut off, which results in a slight loss of information. Ultimately, this could perhaps affect the fidelity and error rate tests. This could be the case in section 4.4 where the graphs show a fidelity that fluctuates for *zfp -p 16* and *zfp -a 1e-4*. Another source of error is an insufficient number of tests. Several tests were performed to determine how compression and GPU acceleration affect memory usage and execution time. While the goal was to test as much as possible, one source of error could be that the tests were not exhaustive enough. For example, running more tests for a higher number of qubits may have allowed for a more accurate understanding of the growth rate of execution time and memory usage.

5.3 Future Work

Based on the results, some key aspects to explore in future studies is how the compression algorithm's performance is affected by an increasing number of qubits. For example, one could perform tests of up to 25 qubits or more, which would allow for the creation of projection graphs and determine if using data compression is a feasible solution for larger qubits. Another aspect to explore further is how utilizing unified memory between RAM and VRAM might impact the ability to decrease execution time whilst providing a lower memory utilization. For the current implementation, only using VRAM limits the size of the simulated circuit to the size of the VRAM itself. By utilizing unified memory it is possible to avoid this limitation. Finally, one might also explore how data compression and GPU acceleration impact memory usage and execution time for quantum algorithms other than Grover's search algorithm. One might discover that data compression is better suited for specific algorithms and use cases, with more feasible execution times. It may also be possible to determine which types of algorithms would work well with data compression and thereby allow dedicated memory optimizations through compression techniques.

Bibliography

[1] C. Oh, C. Lee, L. Banchi, and et al, "Optimal measurements for quantum fidelity between gaussian states," 01 2019.

[2] M. Dyakonov, "When will useful quantum computers be constructed? not in the foreseeable future, this physicist argues. here's why: The case against: Quantum computing," *IEEE Spectrum*, vol. 56, no. 3, pp. 24–29, 2019.

[3] R. P. Feynman, "Simulating physics with computers," *International Journal of Theoretical Physics*, vol. 21, no. 6-7, p. 467–488, 1982.

[4] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM Journal on Computing*, vol. 26, no. 5, p. 1484–1509, Oct 1997. [Online]. Available: http://dx.doi.org/10.1137/S0097539795293172

[5] P. Ball, "First quantum computer to pack 100 qubits enters crowded race," *Nature*, vol. 599, no. 7886, p. 542–542, 2021.

[6] T. H. Johnson, S. R. Clark, and D. Jaksch, "What is a quantum simulator?" *EPJ Quantum Technology*, vol. 1, no. 1, pp. 1–12, 2014.

[7] T. Jones, A. Brown, I. Bush, and et al, "Quest and high performance simulation of quantum computers," *Scientific Reports*, vol. 9, no. 1, 2019.

[8] Y. Zhou, E. M. Stoudenmire, and X. Waintal, "What limits the simulation of quantum computers?" *Phys. Rev. X*, vol. 10, p. 041038, Nov 2020. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevX.10.041038

[9] K. Sayood, *Introduction to data compression.* Morgan Kaufmann, 2017.

[10] J. H. Pujar and L. M. Kadlaskar, "A new lossless method of image compression and decompression using huffman coding techniques," *https://www.jatit.org/*, vol. 15, no. 1, 2005. [Online]. Available: https://www.jatit.org/volumes/research-papers/Vol15No1/3Vol15No1.pdf

[11] P. Lindstrom, "Fixed-rate compressed floating-point arrays," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 12, pp. 2674–2683, 2014.

[12] L. A. Baumgardner, A. K. Shanmugam, H. Lam, and et al, "Fast parallel tandem mass spectral library searching using gpu hardware acceleration," *Journal of Proteome Research*, vol. 10, no. 6, pp. 2882–2888, 2011, pMID: 21545112. [Online]. Available: https://doi.org/10.1021/pr200074h

[13] G. Aleksandrowicz, T. Alexander, P. Barkoutsos, and et al, "Qiskit: An open-source framework for quantum computing," jan 2019. [Online]. Available: https://doi.org/10.5281/zenodo.2562111

[14] C. Developers, “Cirq,” Aug. 2021, See full list of authors on Github: https://github .com/quantumlib/Cirq/graphs/contributors. [Online]. Available: https://doi.org/10.5281/zenodo.5182845

[15] Y. Suzuki, Y. Kawase, Y. Masumura, and et al, “Qulacs: a fast and versatile quantum circuit simulator for research purpose,” *Quantum*, vol. 5, p. 559, Oct 2021. [Online]. Available: http://dx.doi.org/10.22331/q-2021-10-06-559

[16] X.-C. Wu, S. Di, E. M. Dasgupta, and et al, “Full-state quantum circuit simulation by using data compression,” *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2019. [Online]. Available: http://dx.doi.org/10.1145/3295500.3356155

[17] P. G. Kwiat, J. R. Mitchell, P. D. D. Schwindt, and et al, “Grover's search algorithm: An optical approach,” *Journal of Modern Optics*, vol. 47, no. 2-3, pp. 257–266, feb 2000. [Online]. Available: https://doi.org/10.1080%2F09500340008244040

[18] P. A. M. Dirac, “A new notation for quantum mechanics,” *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 35, no. 3, p. 416–418, 1939.

[19] R. LaRose, “Distributed memory techniques for classical simulation of quantum circuits,” 2018. [Online]. Available: https://arxiv.org/abs/1801.01037

[20] F. A. Bais and J. D. Farmer, “The physics of information,” in *Philosophy of Information*, ser. Handbook of the Philosophy of Science, P. Adriaans and J. van Benthem, Eds. Amsterdam: North-Holland, 2008, pp. 670–671. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780444517265500200

[21] I. Djordjevic, “Chapter 3 - quantum circuits and quantum information processing fundamentals,” in *Quantum Information Processing and Quantum Error Correction*, I. Djordjevic, Ed. Oxford: Academic Press, 2012, p. 92. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780123854919000034

[22] C. Zalka, “Grover’s quantum searching algorithm is optimal,” *Phys. Rev. A*, vol. 60, pp. 2746–2751, Oct 1999. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevA.60.2746

[23] Cglosser, “Cglosser/quest: Quantum electromagnetics simulation toolkit (quest).” [Online]. Available: https://github.com/cglosser/QuEST

[24] O. A. Mahdi, M. A. Mohammed, and et al, “Implementing a novel approach an convert audio compression to text coding via hybrid technique,” *International Journal of Computer Science Issues*, vol. 9, no. 6, 2012.

[25] “Floating point compression: Lossless and lossy solutions.” [Online]. Available: https://computing.llnl.gov/projects/floating-point-compression

[26] P. Lindstrom, M. Salasoo, M. Larsen, and et al, “Zfp 0.5.5 documentation,” 2019. [Online]. Available: https://zfp.readthedocs.io/en/release0.5.5/index.html

[27] J. Diffenderfer, A. L. Fox, J. A. Hittinger, and et al, “Error analysis of zfp compression for floating-point data,” *SIAM Journal on Scientific Computing*, vol. 41, no. 3, 2019.

[28] B. Furht, Ed., *Discrete Cosine Transform (DCT).* Boston, MA: Springer US, 2008, pp. 186–188. [Online]. Available: https://doi.org/10.1007/978-0-387-78414-4_304

[29] P. Lindstrom and M. Isenburg, "Fast and efficient compression of floating-point data," *IEEE transactions on visualization and computer graphics*, vol. 12, pp. 1245–50, 09 2006.

[30] H. AI. (2022) What is hardware acceleration? [Online]. Available: https://www.heavy.ai/technical-glossary/hardware-acceleration

[31] A. Amariutei and S. Caraiman, "Parallel quantum computer simulation on the gpu," in *15th International Conference on System Theory, Control and Computing*, 2011, pp. 1–6.

[32] "Cuda c++ programming guide." [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#introduction

[33] D. A. Patterson and J. L. Hennessy, *Computer Organisation and Design, The Hardware/Software Interface.* Morgan Kaufmann, 2013.

[34] R. de Wolf, "The potential impact of quantum computers on society." *Ethics Inf Techno*, vol. 19, 12 2017. [Online]. Available: https://link.springer.com/article/10.1007/s10676-017-9439-z

[35] R. A. Barkoutsos, P. W. S., and et al., "Resource-efficient quantum algorithm for protein folding," *npj Quantum Inf*, vol. 7, no. 38, 2021. [Online]. Available: https://doi.org/10.1038/s41534-021-00368-4