CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Optimizing quantum computer simulation

Exploring the use of FPGAs and data compression for improved performance

Master's thesis in Computer science and engineering

EDVIN LEIDÖ
MARCUS MATHIASON

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

Master's thesis 2022

Optimizing quantum computer simulation

Exploring the use of FPGAs and data compression for improved performance

EDVIN LEIDÖ
MARCUS MATHIASON

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 simulation
Exploring the use of FPGAs and data compression for improved performance
Edvin Leidö, Marcus Mathiason

Supervisor: Pedro Petersen Moura Trancoso,
Department of Computer Science and Engineering
Examiner: Risat Pathan, Department of Computer Science and Engineering

Master'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

Optimizing quantum computer simulation
Exploring the use of FPGAs and data compression for improved performance
EDVIN LEIDÖ
MARCUS MATHIASON
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Quantum computers are a hot topic in research, with many ongoing projects, both by leading companies in the computing industry such as Google, IBM, and Microsoft, as well as in academia, such as an ongoing project at Chalmers University of Technology. While these research efforts are making strides in increasing the size and complexity of quantum computers, the availability of real quantum computers is still very limited. Thus, using one is infeasible to anyone outside of the research groups working on them. As a result, several simulators of quantum computers have been developed and are publicly available. Quantum simulators allow anyone to simulate small quantum circuits, but they are performance intensive; both time and space complexities are exponential with respect to the number of qubits, which puts a pressure on ensuring that the simulators make efficient use of computer hardware. This work explores how high performance computing techniques can be applied to existing quantum computer simulators, in order to improve both execution time and memory usage. The two main areas of focus are FPGA designs for improved execution time, and data compression for improved memory usage. The *qsim* simulator is used as a basis for the work. The project resulted in a few variants of an FPGA design for accelerating the most demanding part of *qsim*, and both lossless and lossy data compression schemes applied to the memory of the simulator to study their effects. Both of these optimizations show promise for increasing the performance of quantum computer simulation.

Keywords: quantum computer simulation, FPGA, data compression, floating point compression, computer engineering, computer science.

Acknowledgements

We would like to express our gratitude to our supervisor Pedro Petersen Moura Trancoso for providing valuable input and guiding us along the way of the project. We would also like to thank Anton Frisk Kockum at the Department of Microtechnology and Nanoscience for supporting us with his expertise.

Edvin Leidö, Marcus Mathiason, Gothenburg, July 2022

Contents

List of Figures

List of Tables

1

Introduction

1.1 Background

Quantum computing has been a topic of research since the 1980s [1]. The main difference between a classical computer and a quantum computer can be narrowed down to how computations are performed [2]. Whereas classical computers utilize bits to represent one of two possible values of a logical state, quantum computers instead use quantum bits, known as qubits, which can be placed in superpositions between states 0 and 1 [2, 3]. This opens up for possibilities to solve problems with a significantly lower time complexity than what would be possible with normal computers by utilizing certain algorithms [4, 5]. Such algorithms include Grover's algorithm [4], which can perform unstructured search with a time complexity of $O(\sqrt{n})$; a quadratic improvement over classical search algorithms, and Shor's algorithm[1], which is able to factorize integers in polynomial time - almost exponentially faster than the currently most efficient classical algorithm known, the general number field sieve [6]. With such performance one might think that this threatens computer security as a whole as many cryptography algorithms build on the fact that factorization is computationally expensive. While this is something to take into consideration for the future, it is not feasible for just anyone to build a quantum computer as they are very costly and require time, planning and precision to maintain. Real quantum computers are currently limited to research projects, requiring extreme conditions to be upheld, such as cooling the system to near absolute zero, in order for the qubits to remain stable for long enough to operate on them [7]. Even with these conditions upheld, the lifetime of qubits is still limited to less than a millisecond in current state of the art quantum computers [8].

However, if one were to accept the limitation of using only a fraction of a quantum computer's computational power for a certain problem, it is with basic knowledge of linear algebra possible to build a simulator of quantum computer circuits on a classical computer. This is useful for several reasons. Principally, because it allows anyone to reason about quantum circuits and algorithms, and test them with small input sizes, without access to real quantum computers. Even if the size of problems that can be simulated is limited, it is still useful to be able to test them numerically. Moreover, sophisticated quantum computer simulators also provide better accuracy

[1] https://quantum-computing.ibm.com/composer/docs/iqx/guide/shors-algorithm

in order to determine if we have reached so called *quantum advantage* [7], i.e. the limit at which quantum computers would be able to perform tasks that classical computers can not do in any feasible time.

However, the problem of simulating quantum computers is hard. Since quantum bits can exist in superpositions between discrete states, simulating them with a classical computer implies tracking the probability of measuring all possible discrete states, i.e. all possible bit strings of a given length. Several methods for simulating quantum computers exist [9], but the focus of this work is on so called *Schrödinger full state vector* simulators. These simulators keep track of all states with a large vector known as the *state vector* [2], and maintaining it leads to exponential time and space complexities with regard to the number of simulated quantum bits [2]. This complexity is unavoidable, because of the fundamental differences between quantum computers and classical computers; if classical computers could simulate quantum computers easily, there would be no need for real quantum computers. As a result of the computational difficulty of simulation, the size of circuits that can be simulated in feasible time is limited, and simulators are incentivized to utilize the computer hardware effectively to be able to simulate as many quantum bits as possible. Several options exist for accelerating quantum simulators, such as utilizing SIMD units of CPUs, as well as using hardware acceleration with GPU. Another option for hardware acceleration is implementing a hardware design in a *Field-programmable gate array* (FPGA). These are devices that contain reconfigurable hardware, meaning that the hardware design on the chip can be adjusted over time. Furthermore, quantum simulators benefit from reduced memory usage. One method of reducing memory usage is with data compression. These two optimizations, a hardware implementation for FPGAs, and data compression, will be the focus of this work.

1.2 Ethical considerations

Making optimized quantum computer simulations available to anyone with access to an FPGA and an effective data compression algorithm may in the future pose a threat to computer security methods that utilize the fact that integer factorization has an exponential time complexity. Thus, large enough prime numbers become unfeasible for normal computers to compute. For a quantum computer however, with the right algorithm it has been found that integer factorization can be brought down to polynomial time complexity. According to an article written by Tommaso Gagliardoni, the minimum required number of qubits to break a 112, 128 and a 192 bit RSA encryption is 4098, 6146 and 15362 respectively [10]. With the computers of today, such numbers are simply not possible to reach when simulating quantum computers. In an article by University of Chicago a group of researchers used high-end hardware and data compression in order to reduce the memory requirement for Grover's Algorithm. Their results showed that with their method, a 61 qubit Grover's Algorithm's memory requirements could be reduced from 32 exabytes to 768 terabytes [11]. While this is a substantial improvement, it is only possible on supercomputers and it is still a long way from the minimum requirement of 4098

qubits to break 112 bit RSA. In fact, even the top physical quantum computers have some way to go before reaching this requirement. IBM recently reached their goal of a three-digit qubit quantum computer, more specifically 127 qubits, last year [12] with the goal of reaching 433 and then 1121 qubits in the coming years. With this in mind, it is clear that quantum computers are not yet in a state that threatens integer factorization cryptography, and quantum computer simulators even less so.

1.3 Related work

Several quantum computer simulators have been developed by the industry [13–15], as well as in academia [9, 16–19], and there are optimizations with regard to computation for many of them. This includes utilizing SIMD capabilities of modern CPUs [14], and offloading computation to GPUs to exploit parallelism further [14, 16, 17]. Efforts have also been made to improve the performance of the algorithms, by applying gates directly to the state vector [9, 18, 19], and many simulators such as *qsim* work this way [14]. The use of FPGAs for acceleration of quantum computer simulation is a relatively unexplored area, but there have been some research efforts towards that goal. For example, Pilch and Długopolski [20] implemented an FPGA design for simulating quantum computers, using fixed-point number representation. Their design focuses on the capability of performing all operations typically done by simulators in the hardware design, to reduce data transfers. Furthermore, they reduce the complexity of calculations by expanding all quantum gates to the size of the state vector of the whole system, at the cost of area utilization and the number of qubits that can be simulated [20].

This work focuses on implementing optimizations for an existing simulator, such that they could easily be integrated into existing systems. The computational optimizations focus on utilizing an FPGA in combination with the style of operation described in related work [9, 18, 19], and implemented in *qsim*, which provides better space complexity than the method described above, but requires more complex indexing.

1.4 Purpose

The purpose of this work is to discover new potential performance optimizations for simulating quantum computers, with a focus on FPGAs for optimizing execution time, and data compression for optimizing memory usage. Such optimizations could mean that one could, with some given hardware, potentially simulate more complex circuits with more qubits than what is possible without such optimizations. Furthermore, this work serves as a step towards pointing out potential areas of further research on the subject of quantum computer simulation.

1.5 Goal

The goal with this project is to answer the research question *How effective is applying FPGAs and data compression techniques on performance and precision of existing quantum computer simulations, and what are the effects on the results of simulation.* This will be done by examining existing quantum computer simulators in order to identify their bottlenecks, and using that data to develop two optimizations: The first is utilizing an FPGA in order to optimize the performance of said simulators. This will be done by creating a hardware design for performing the most time consuming tasks, that can be implemented in an FPGA. The second optimization is intended to improve memory usage using data compression. By modifying the source code of the simulators to compress the data they write to memory, the overall usage can be lowered potentially allowing for more qubits to be added to circuits. These two optimizations will be performed separately. Finally, the effects of both of these optimizations on simulation results and execution time will be compared with the unoptimized simulators.

1.6 Delimitations

The simulators that will be examined are Schrödinger full state vector simulators. As a result, this project will not take into concern how similar optimization methods could be applied to other kinds of quantum computer simulators. Furthermore, this means that the performance of the optimized simulator can only be reasonably compared to the unoptimized version of the same simulator, or one of the same type. Another limitation is that this work focuses on how the existing algorithms perform for the aforementioned optimization. Thus, the algorithms themselves will not be modified or replaced.

As there were issues encountered with one of the simulators during the initial profiling which lead to difficulties in discovering where in the backend the bottlenecks were located, *qsim* [14][2] was chosen as the simulator which will be examined and on which the optimizations would be based.

[2]https://quantumai.google/qsim

2

Theory

In this chapter we present the fundamental definitions for quantum computing that are needed to understand this work. This includes how qubits and quantum gates are expressed mathematically, and represented in computers. Additionally, we present the method of simulating a quantum computer with a classical computer known as Schrödinger full state vector simulation, which is the method used in the simulators examined in this thesis. Furthermore, we explain the data compression methods that are the basis of our work with memory optimization.

2.1 Quantum bits

A quantum computer is a machine that uses quantum physical effects in order to perform computations. Just as classical computers use bits as fundamental building blocks, quantum computers use the analogous quantum bits, or qubits. In addition to being able to take on the states 0 or 1 like classical bits, qubits can can be in superpositions between these states. These superpositions represent the probability that a certain state will be measured upon observation [2]. This is part of an important fundamental property of qubits and quantum physics in general: measuring a qubit's state will always result in one of the two discrete states 0 or 1, and the superposition of a qubit only exists prior to observation [2]. Real qubits can be constructed in different ways using different real-world quantum effects. However, the abstract definition of a qubit remains the same, and can be reasoned about independently of how actual quantum computers are built.

As mentioned previously, a qubit can be in state 0 or 1, or in a superposition between these two states. Mathematically, this can be expressed as a vector that describes the probability of finding the qubit to be in state 0 or 1 upon observation. The states of 0 and 1 can then be written as two vectors:

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

The states $|0\rangle$ and $|1\rangle$ form an orthonormal basis for describing any qubit in a superposition, expressed as a linear combination of these bases with complex numbers

α and β as weights:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle = \begin{bmatrix}\alpha\\ \beta\end{bmatrix}$$

A qubit is thus a two-dimensional vector in a complex vector space, with $|0\rangle$ and $|1\rangle$ as orthonormal bases [2, Ch. 1.1]. When the state of a qubit is measured, state 0 has probability $|\alpha|^2$, and 1 has probability $|\beta|^2$. We know that the sum of all probabilities are equal to one, so $|\alpha|^2 + |\beta|^2 = 1$. Relating this back to the definition of the qubit, we see that the vector is of unit length.

Multiple qubits are expressed as vectors of larger dimensions, comprising all possible states of that number of qubits. Thus, the state vector representing n qubits must be 2^n-dimensional in order to encapsulate all possible states of n qubits. For example, two qubits can be in four different states: 00, 01, 10, and 11, or superpositions between them. The bases for two qubits are

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

Therefore, a two-qubit state vector $|\psi\rangle$ is a four-dimensional vector, with $|00\rangle$, $|01\rangle$, $|10\rangle$, $|11\rangle$ as bases, and complex numbers a_{00}, a_{01}, a_{10}, and a_{11} as weights associated with each base:

$$|\psi\rangle = a_{00}\,|00\rangle + a_{01}\,|01\rangle\,a_{10}\,|10\rangle + a_{11}\,|11\rangle = \begin{bmatrix}a_{00}\\a_{01}\\a_{10}\\a_{11}\end{bmatrix}$$

2.2 Quantum gates

Just as qubits are the quantum computing analogue of classical bits, quantum gates are the analogue of classical logic gates. Mathematically, quantum gates are defined as square matrices of order 2^n for n input qubits. The application of a gate to some input qubits is defined as multiplying the matrix with the state vector describing the qubits. For example, we can apply the analogue of the classical NOT gate, known as the X gate, to one input qubit. The X gate is expressed as the matrix $X = \left(\begin{smallmatrix}0 & 1\\ 1 & 0\end{smallmatrix}\right)$, so we get

$$X\begin{bmatrix}\alpha\\ \beta\end{bmatrix} = \begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}\begin{bmatrix}\alpha\\ \beta\end{bmatrix} = \begin{bmatrix}\beta\\ \alpha\end{bmatrix}$$

In the classical case, there are only four conceivable single-bit gates: the NOT/Inverter gate shown above, the gate that sets to 1, the gate that sets to 0, and an identity gate that has no effect. In contrast, there are many singe-qubit quantum gates that can be conceived, because they can affect the superposition of the input

qubits, which means the components of the qubit state vectors can be non-integer values. Several such gates are useful, such as the Hadamard (H) gate,

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

Which places both the base states $|0\rangle$ and $|1\rangle$ into superpositions halfway between the bases. This means that the superposition will be such that the measured state will be uniformly distributed, i.e. 50% probability of measuring either a 0 or a 1:

$$\frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ -1 \end{bmatrix} = \frac{|0\rangle - |1\rangle}{\sqrt{2}}$$

Gates with multiple input qubits are expressed as larger matrices, similarly to how multiple qubit states are expressed as larger vectors. Below are the matrices of the two-qubit controlled-NOT ($CNOT$), and controlled-Z ($C(Z)$) gates:

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \quad C(Z) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$

The $CNOT$ gate is universal, which means that it can be used to construct any other gate, similarly to the NAND gate in classical computers [2]. Universality is an important topic because it puts a limit to how many gates are needed to represent any circuit. There are several sets of gates that have been proven to be universal, with varying degrees of feasibility for use in practice [21]. One example of such a set, used for the benchmarking circuits in this work (see section 3.1), is $\{H, T, C(Z), X^{1/2}, Y^{1/2}\}$. The H and $C(Z)$ gates are mentioned above, and the matrix representations of the other gates [22] are shown below:

$$T = \begin{bmatrix} 1 & 0 \\ 0 & e^{\frac{i\pi}{4}} \end{bmatrix}$$

$$X^{1/2} = \sqrt{X} = \frac{1}{\sqrt{2i}} \begin{bmatrix} i & 1 \\ 1 & i \end{bmatrix}$$

$$Y^{1/2} = \sqrt{Y} = \frac{1}{\sqrt{2i}} \begin{bmatrix} i & -i \\ i & i \end{bmatrix}$$

Why this set is universal, and how the effects of the gates can be interpreted will not be explained here, as it does not pertain to understanding the rest of this thesis.

2.3 Quantum circuits

Using qubits and quantum gates, one can construct quantum circuits. These can then be executed in a quantum computer, or a simulator running on a classical computer. A quantum circuit is typically defined as a set of gates, each with specified input qubits and a time step at which it is to execute. The number of time steps in the circuit is known as the circuit depth.

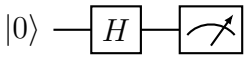


Figure 2.1: A simple, single-qubit quantum circuit that places a qubit into a superposition using a Hadamard gate, and then measures its state. The measured state has equal probability of being 0 and 1.

The example in figure 2.1 is very simple, but with larger quantum circuits, whole quantum algorithms can be expressed, allowing them to be executed by a quantum computer or simulator [2]. As an example, we describe how to express Grover's search algorithm as a circuit. This algorithm can perform unstructured search with a time complexity of $O(\sqrt{n})$, which is better than any classical algorithms [23]. The algorithm works as follows [4]: the items in the list are mapped to qubit states. So for three qubits, each of the qubit states $|000\rangle, |001\rangle, |010\rangle, ..., |111\rangle$ corresponds to an item in the list. Then, a matrix known as the *oracle matrix* is constructed, which adds a negative phase to the state that represents the item that is being searched for. This is the same as an identity matrix modified with -1 in the right position:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

The initial state - before knowing where the goal item is - can be expressed with a uniform superposition. This is a state which, upon measurement, will collapse into any of the possible discrete states with equal probability. Using quantum gates, this is the same as applying the Hadamard gate to all qubits initially, as shown in figure 2.1.

The algorithm works by applying the oracle matrix to the initial state. This swaps the sign of the "goal" qubit's amplitude, which can be interpreted geometrically as a reflection around the x axis of the vector space. Then, we can reflect the new state vector around the previous state vector, which will effectively double the amplitude of the goal qubit, compared to it's amplitude in the previous step. By repeating this process, the amplitude of the goal qubit grows until it is sufficiently likely that

that state will be measured. This turns out to have a time complexity of $O(\sqrt{n})$ [23]. Thus, one can represent Grover's algorithm using Hadamard gates, the oracle matrix (because any matrix can be a gate), and rotation gates. The circuit for two qubits when searching for $|00\rangle$ is shown in figure 2.2.

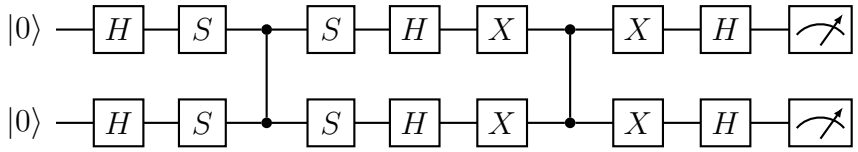


Figure 2.2: This circuit describes Grover's algorithm for two qubits, which could be used to search a list of at most four (2^2) items. Each gate is defined as a matrix. This circuit has depth 11, because it has 11 time steps in total. The gates spanning both qubits in the third and seventh time steps are two qubit *C(Z)* gates.

2.4 Quantum computer simulation

Several ways of simulating quantum computers have been researched. These include special case simulators that are designed for a more narrow subset of applications or circuits, and achieve higher performance by focusing only on that subset [24, 25], as well as general case simulators that should be able to simulate any quantum circuit [14, 26, 27]. This last class is where the simulators examined in this paper belong, known as Schrödinger full state vector simulation [11]. This type of simulator maintains the state vector for all qubits, containing amplitudes for each state. This means that the simulator will require 2^n amplitudes for a circuit of n qubits [2]. However, it scales polynomially for the circuit depth (number of time steps) and the number of gates [11].

We can now describe how a full state vector simulation of a quantum circuit can be performed. The input to a simulator is a quantum circuit of n qubits and depth (number of time steps) d, which contains a number of gates defined by matrices. Furthermore, the state is described by a state vector of 2^n dimensions (which will typically be zero initialized), denoted $|\Phi\rangle = a_{00...0}\,|00..0\rangle + a_{00...1}\,|00..1\rangle + ... + a_{1...1}\,|11...1\rangle$. To simulate the application of a single gate to its input qubits, the gate matrix can be multiplied by the state vector of the input qubits as seen in section 2.3. Thus, simulating the whole circuit becomes a process of multiplying each gate with the state vector of its respective input qubits.

However, since gates often only take a subset of all qubits as input, not all qubits in the system, there has to be some transformation before this multiplication can happen. For example, a two-input gate matrix can not be multiplied directly with $|\Phi\rangle$ if $n > 2$, because a matrix of order 2 can not be multiplied with a vector of more than two dimensions. A straight-forward way of doing this transformation is to order the gate matrices from left to right, pad all matrices with identity matrices using the Kronecker product (see appendix B.1 for the definition), which expands the matrices to $(2^n \times 2^n)$-matrices, and then multiply all of them together into one

matrix representing the whole circuit, which can then be multiplied with the state vector $|\Phi\rangle$ [18, 19]. However, this requires a large amount of matrix multiplication, and will result in very large and sparse gate matrices. An example of this can be seen in figure 2.3, which demonstrates the sparseness of the resulting matrix after performing this expansion.

As a result of the inefficiency of expanding the gate matrices to the order of the state vector, methods where the state vector of the relevant qubits is modified directly are commonly used instead [9, 19]. Applying a k-input gate directly an n-qubit state vector works by multiplying the gate matrix to 2^{n-k} vectors of selected elements from the state vector. These elements are selected based on their indices. For example, if a single-qubit gate is applied to the kth qubit, the vectors to multiply with are all the pairs of elements which have the same index bits in all but the kth position [18, 19]:

$$v = \begin{bmatrix} a_{*\ldots*0*\ldots*} \\ a_{*\ldots*1*\ldots*} \end{bmatrix}$$

For example, in a three qubit state vector, these would be the vectors $\begin{pmatrix} a_{0\mathbf{0}0} \\ a_{0\mathbf{1}0} \end{pmatrix}$, $\begin{pmatrix} a_{0\mathbf{0}1} \\ a_{0\mathbf{1}1} \end{pmatrix}$, $\begin{pmatrix} a_{1\mathbf{0}0} \\ a_{1\mathbf{1}0} \end{pmatrix}$, and $\begin{pmatrix} a_{1\mathbf{0}1} \\ a_{1\mathbf{1}1} \end{pmatrix}$.

For gates with more than one input qubit, the vectors are constructed similarly; the elements have differing bits only in the positions corresponding to the input qubits' indices in the state vector.

The whole process of applying a gate then becomes to extract, or gather, the relevant elements from the state vector, multiplying the gate matrix with those elements, and then writing the result back to the state vector of the system. This is the method used by the simulators used as a basis in this thesis work [14].

$$H_e |\Phi\rangle = I \otimes H \otimes I \times |\Phi\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} a_{000} \\ a_{001} \\ a_{010} \\ a_{011} \\ a_{100} \\ a_{101} \\ a_{110} \\ a_{111} \end{bmatrix}$$

Figure 2.3: In this example, a single-qubit gate, the Hadamard gate, is applied to the second qubit of a three qubit state vector $|\Phi\rangle$. Expanding the gate matrix H to a matrix H_e of order 2^n that can be multiplied with $|\Phi\rangle$.

2.5 Lossy and lossless data compression

To reduce the amount of bits needed to represent some form of content one can use different methods of data compression. Doing so could potentially mean significant improvement in performance where the content is being processed in some way. There are two different classes of data compression; lossy and lossless data compression. Lossy compression means that some information regarding the bits will be lost during the compression process. As a consequence, when decompression is done, the bit information will not be sufficient to restore the reduced bits to their exact initial values resulting in an altered version of the original state. Depending on the compression rate the resulting decompressed version may look considerably different. This form of compression is commonly used when streaming video and audio as the end result, while noticeably of worse quality, it is still possible to make out what the compressed version stems from. This is especially advantageous for slower network speeds as the data can be communicated faster at the expense of degraded quality. In applications that fundamentally relies on the data being compressed one needs to be careful when tuning the rate of which compression is done - referred to in this paper as *compression rate*[1]. With too high of a rate, we will likely end up with very low memory usage, but the original state will be corrupted giving us faulty outputs for the application. With the compression rate set too low, our application will work as intended, but memory usage reduction will be negligible. If we still want to be able to fully reconstruct the initial data after compression, then lossless data compression methods are to be considered.

Lossless compression means that after the data has been compressed, it can then be perfectly reconstructed using the data that is kept. This method of compression is especially beneficial for scenarios where precise data is used along the way of solving some problem, like SGD (stochastic gradient descent). Each step taken to update the nodes of the SGD instance makes a lot of approximations and thus brings with it lots of noise. In general, the noisier the results get the longer the convergence time so it is easy to see that optimal circumstances can only be reached when approximations are brought down to a minimum.

Much like SGD, quantum computers save the states of previous iterations to be used later on in computing other states. This means that each approximation or error will affect the system as a whole. The earlier the approximation is done, the error will escalate for each iteration giving us an even greater error in the final output. This error, often described as distance to the correct state vector output, is typically measured using fidelity which is defined as such

$$F(\rho, \sigma) = (tr\sqrt{\sqrt{\rho}\sigma\sqrt{\rho}})^2$$

where ρ and σ are the density matrices examined. Typically we will have ideal conditions when simulating quantum computers and small errors will occur in the physical quantum computer as it is fundamentally analog. Fidelity helps us measure the probability that one state can be identified as another despite the small errors.

[1] https://zfp.readthedocs.io/en/release0.5.4/modes.html#fixed-rate-mode

This is important as we cannot always expect the physical quantum computer to behave the same way as the simulator.

Three different compressors are used in this project; Gzip, 7-zip and ZFP. Gzip is a lossless data compression utility aimed at single-file compression[28]. Although only being able to compress one file at a time, gzip is relatively efficient in doing so and is very easy to use. Listing 2.1 shows the syntax for compressing a .bin file to .bin.gz. By simply adding the -d flag to the same line but for the newly created foo.bin.gz, the file is decompressed to its initial state. Similarily, 7-zip is also a free to use compression software with the addition of being open source and offering multi-file archiving as well as a file format with extra high compression rate called 7z[29]. ZFP is an open source library developed at Lawrence Livermore National Laboratory focused on compressed numerical arrays[30]. In addition, ZFP also offers a file compressor capable of performing lossy compression. Listing 2.2 demonstrates how one would compress a file foo.bin containing a vector of floating point values of size 131072 with an absolute error tolerance of $1e-3$.

```
1    gzip foo.bin
```

Listing 2.1: File compression using gzip

```
1    ./zfp -i foo.bin -f -1 131072 -a 1e-3
```

Listing 2.2: Lossy file compression using zfp

2.6 Hardware acceleration

CPUs are generalist tools for executing instructions from a predefined set in an arbitrary order. This makes them very flexible and usable for a variety of different workloads. However, this flexibility also limits them in many ways: Because the specific instructions that will run depend entirely on the software being executed, no optimizations can be done for any specific application.

If the goal is to optimize a certain program or process, CPUs do not offer much help, because they are limited by the requirement of being general purpose devices. On the other hand, hardware could be built specifically for the purpose of executing a specific set of operations. This is known as hardware acceleration, and there are multiple examples of devices that do that, such as GPUs, and ASICs (*application-specific integrated circuits*).

ASICs, as the name implies, are designed for specific applications. As a result, they can utilize optimizations for the operations and flow of data in their specific application, such as pipelining, which can increase throughput. However, they are of course limited to the application they were designed for. Furthermore, if a design needs to be updated, new ASIC chips need to be manufactured. This increases time to market, as significant time has to be spent on making sure the correctness of a design. One way of improving this is using reconfigurable hardware, such as FPGAs (*field-programmable gate arrays*). These are chips which contain reprogrammable hardware, and can be used either as prototyping devices when developing ASICs,

or used on their own, with the added flexibility allowing reconfiguring the hardware when needed [31].

FPGAs are based on LUTs (*look-up tables*), which take some number of bits as an input, and can be configured to give certain output for certain inputs. This way, logic gates can be represented, and with them larger blocks of logic can be built [31]. This is often done using a hardware description language which can represent objects at the level of logic gates, or with a high level synthesis tool which can transform code written in a software programming language, such as C, into a hardware description language.

3

Methods

This chapter describes the process of the work done for this thesis. Initially, two simulators were examined and profiled: IBM's *Qiskit*, and Google's *qsim*. These both worked similarly, and provided similar results from profiling. Thus, only one of them, *qsim*, was chosen as the basis of the optimizations. Using the profiling data, the most time consuming part of *qsim* could be discerned, which became the focus for the FPGA implementation. Similarly, the part of *qsim* which required the most memory was revealed, and was the primary focus for the data compression implementation.

3.1 Profiling the simulators

Two quantum computer simulators were chosen for performance profiling: IBM's *Qiskit*, and Google's *qsim*. The goal was to assess the behaviour of the simulators, both in execution time and memory usage, as the input size increased. Furthermore, the goal included an assessment of existing optimizations such as parallel simulation on multiple CPU cores, and GPU simulation.

There are two main dimensions in which the input size can be increased. These are the number of qubits in the circuit, and the circuit depth (or the number of timesteps). In order to test the simulators with inputs of different sizes, a set of circuits for benchmarking was required. These circuits needed to be representative of typical quantum circuits, and maintain a similar structure across input sizes. Thus, a method of automatically generating circuits would be the most convenient. One such method is described by Boixo *et al.* [32], and was used to generate the testing circuits for the benchmarks in this work. This algorithm generates pseudorandom circuits containing gates from the universal set $\{H, T, C(Z), X^{1/2}, Y^{1/2}\}$, and is described below:

```
// initialization
for each gate at depth 0:
    gate := H
end

apply C(Z) gates in a diagonal pattern (see figure)
```

```
for each gate following a CZ gate:
    if no T gate exists on this qubit:
        gate := T
    end
    else
        gate := random_gate(T, X^1/2, Y^1/2)
    end
end
```

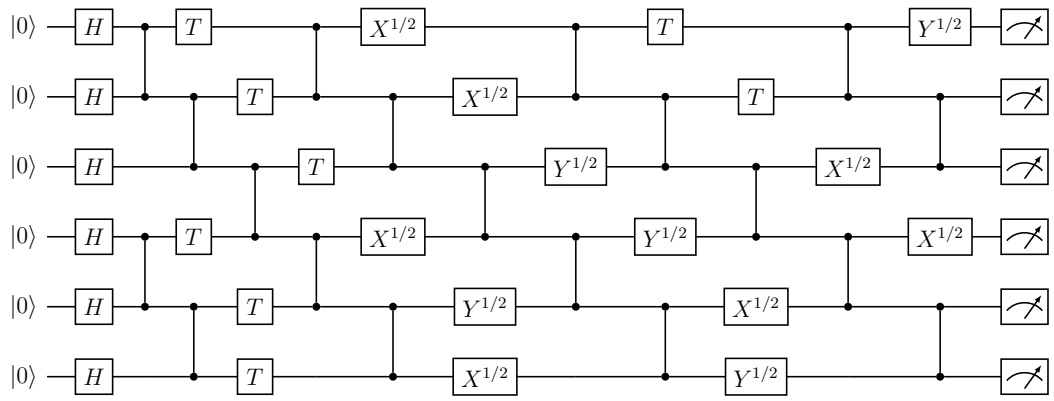


Figure 3.1: An example of a randomly generated circuit of the kind used to benchmark the simulators.

Using this algorithm, circuits were generated at four different depths: 25, 50, 75, and 100, for each qubit count in the range 16 to 32. Figure 3.1 is an example of a circuit with six qubits and 12 timesteps generated using this algorithm.

To profile the execution time of the simulators, each circuit from this set was simulated, and the reported simulation time was recorded. The memory allocations made by the simulators were profiled using *Valgrind* with the *massif* tool [33], which provides detailed information about the heap allocations done during runtime.

When the data for execution time and memory allocations was recorded, the simulators were investigated to find the subroutines in which they spend most of the execution time. This was done in order to find which parts would be most useful to optimize by offloading to an FPGA. The subroutines of *qsim* were found using *Valgrind* with the *callgrind* tool [34], along with the visualization tool *KCachegrind* [35].

These profiling results (found in section 4.1) showed how the input size relates to execution time and the amount of allocated memory. Additionally, they revealed what part of the code the simulators spent most of the execution time in, and what part of the code that allocated the most memory. This provided good information for what to focus on when implementing the two optimizations that were primarily considered; offloading to an FPGA accelerator to improve execution time, and compressing memory to increase the number of qubits that could be simulated.

3.2 FPGA implementation

Because an FPGA design was the primary goal from the onset of this project, and because other optimizations such as using SIMD units via Advanced Vector Extension (AVX) instructions, and utilizing GPUs via Cuda had already been implemented in *qsim*, the focus of the optimizations for execution time was implementing an FPGA design.

Upon examination of the profiling results, it was found that the matrix-vector multiplication as a result of applying a gate to the state vector was where the simulator spent over 90% of the execution time. As a result, the FPGA design was made to solely work on matrix-vector multiplication, to which the input parameters of the functions could be fed and the result be extracted for the simulator to continue working with. As such, an FPGA would execute only part of the full simulation: each gate being applied to the state vector. This had the advantage of conforming to the existing design of *qsim*, which already contains several different versions of this part of the code base that utilize SIMD and GPU acceleration.

The initial decision to be made before we started working on the FPGA design was what language we were going to use. The two main options we considered were Vitis HLS (High Level Synthesis), and VHDL (VHSIC Hardware Description Language). Vitis HLS allows for code to be written in a high level language, for example C++, with additional pragma directives to instruct the high level synthesis tool, which transforms the high level language code into a hardware description language such as VHDL. Vitis HLS is inherently more intuitive to use, and requires fewer lines of code, but lacks in overall control compared to lower level hardware description languages. Since we both are relatively new to working with FPGAs, it was decided that the code would be written with Vitis HLS and C++, and potential improvements that would be possible by using VHDL were left as future work.

After this, work began on the FPGA design. A few different variants were implemented, in order to compare the effects of different design choices. Firstly, a version similar to the basic *qsim* simulator, using floating point numbers without any SIMD instructions. Secondly, a variant using fixed point number representation. Lastly, a variant using SIMD-like operation by working on wider data. This variant utilizes the *hlslib* library which contains customizable wide data types, allowing for SIMD-style code [36]. These designs are fully functional hardware designs that can apply any gate with up to six input qubits to a state vector, using the same data that *qsim* uses. However, because of time constraints and lack of access to computers with FPGA chips in them, the design is not fully incorporated into the rest of *qsim*. This means that it is not possible to run *qsim* with an FPGA connected to the computer and using it for offloading with our design. This is left for future work.

The design is described in more depth in section 4.2 and visualized in figure 4.5, however a brief description is as follows: First initialization data is read from onboard memory, and the full state vector is assumed to exist in the onboard memory as well. The initial data includes two relatively small arrays: the masks and strides used

for indexing the state vector, as well as the gate matrix array, and some smaller supporting data that depend on the gate size, such as the number of iterations to perform. Then four tasks are executed with task-level pipelining for the given number of iterations: Firstly, calculate the start index from which to read from the state vector using the masks. Secondly, gather a vector of data from the state vector by reading from it using the strides. Thirdly, multiply the gate matrix with this vector. Because the simulator uses complex numbers, these are multiplications of complex numbers. Lastly, scatter the result vector to the state vector using the same strides as when gathering.

To test the correctness of the different variants, input data and a correct reference result were needed. Since the code of the *qsim* simulator was publicly available, it could be modified to write the needed data to files while running the simulation. The input data was the state vector prior to applying a given gate, the gate's matrix, as well as some other smaller pieces of data such as the start index and strides needed to extract data. The reference result was the resulting state vector after the gate had been applied, which could be compared to the output of our implementation. Using this data, the implementations were simulated in two ways. Firstly in C simulation, which uses the high level code directly, using the Vitis HLS libraries and a standard C++ compiler. Secondly, in C/RTL Cosimulation, which happens after the high level synthesis step, using the resulting VHDL code.

When the designs worked correctly in simulation, they were also tested in real hardware. The board used was the Xilinx PYNQ Z-2, which contains a Zynq FPGA chip. This board is somewhat limited in size, so the designs synthesised for it were reduced to support up to three qubit gates, instead of the full six that *qsim* supports. However, this does not fundamentally change the design, but only reduces the matrix and array sizes, which also reduces the area used by the matrix multiplication hardware as a result. The files for the FPGA were built using Vivado, and the python-based interface was used to load them onto the board. The same testing data that was used in simulation could be used to verify that the output was correct, by loading the data into the onboard memory and comparing with the simulation output. Other than the PYNQ Z-2, the floating point design with support for six gate qubits was also synthesized for a larger board, namely the Virtex Ultrascale+ HBM. The other designs were not, due to time constraints.

3.3 Data compression implementation

Profiling the execution of a qsim simulation showed that most time was spent in the functions *ApplyGateH* as well as *ApplyGateL* which primarily perform matrix multiplications. Once correctly built and compiled, ZFP's compressed arrays[1] proved to be easy to implement. The first simple attempt involved converting the two fixed-length vectors used in matrix multiplications to compressed arrays. Compared to a standard C++ array, ZFP's compressed array classes' storage size is set by the user using a compression rate parameter as input[37] meaning that we can choose

[1] https://zfp.readthedocs.io/en/release0.5.4/arrays.html

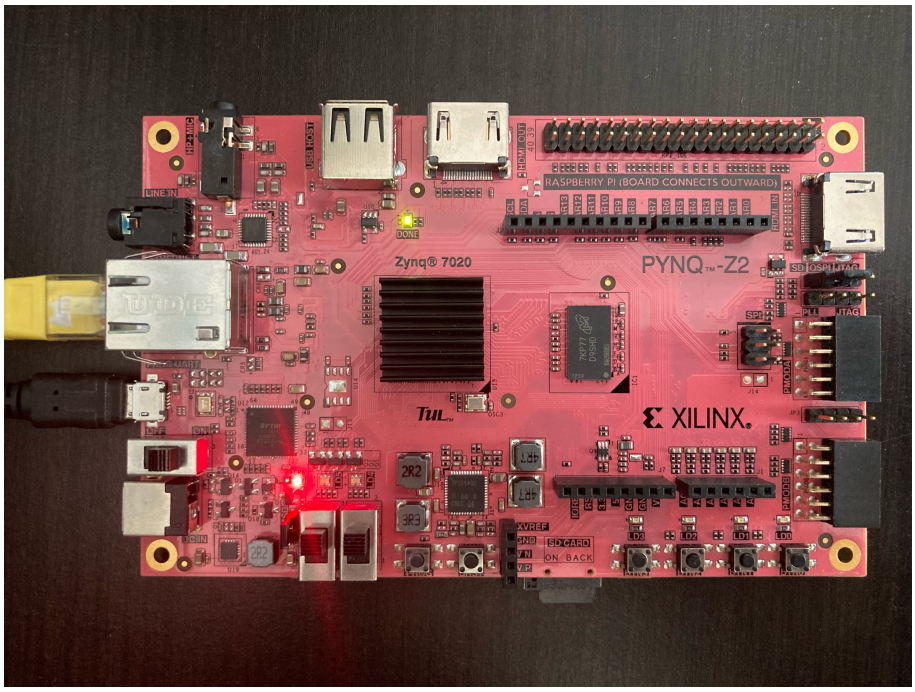


Figure 3.2: The PYNQ Z-2 board in the process of running the FPGA design.

to perform either lossy of lossless compression depending on our needs. As these vectors are used to build the foundation of the matrix multiplications, the system relies heavily on these being correct throughout the iterations and as such, we set the compression rate to 0 in this case. As can be seen in Table 3.1 the results of this simple addition seem promising at a low amount of qubits but quickly fall off as this number is increased. It was speculated that the doubling in memory consumption per qubit added was the cause of the state vector's size and so work was put into figuring out how to compress the state vector while still being able to extract necessary values when needed. Due to a lack of documentation of the inner workings of the simulator and hard to understand variable names, understanding how the simulator worked proved more difficult than what was originally anticipated. Upon looking into the implementations more thoroughly, it was discovered that the state vector could not be referenced as a whole as it was represented by a non-standard Vector class. The class definition of this vector can be seen in Listing 3.1. As can be seen, the only way to extract the state vector's values is to start at index 0 using the *get* function, and to then increment one by one up to the maximum size of the state vector which will always be equal to $2 * 2^{num\_qubits}$. This proved detrimental as ZFP expects an array as input in both the high level as well as low level compression methods[2]. As the simulator is built around this custom vector class, to perform online compression a huge part of the simulator code would have to be refactored and built from the ground up. Because of this, the aim was shifted to instead make a dump of the state vector at certain time steps to a file and compress it offline. The general flow of the instructions can be visualized in Figure 3.3. As the state vector starts out with a lot of zero-valued elements in the first simulation iterations

[2]https://zfp.readthedocs.io/en/release0.5.5/tutorial.html

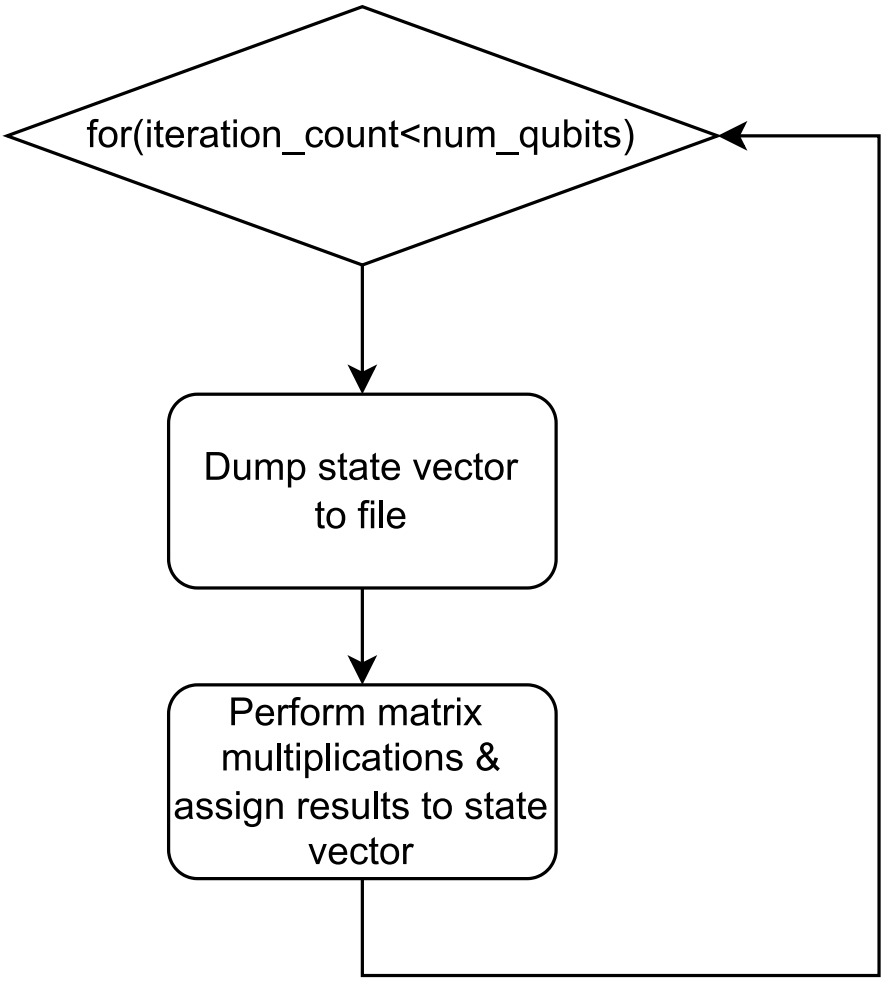


Figure 3.3: The general flow of instructions inside the function ApplyGateH

to then be further populated as the simulation gets further, it is expected that the compression ratio will decrease over time. It is important to note that this method is just used to discover the potential of utilizing data compression. For any real integration of data compression one would ideally be performing compression online, during simulations and potentially dump the state vector to memory rather than a separate file.

For the lossless compression, three different algorithms were used; gzip, 7zip's .bin to .zip compression as well as 7zip's .bin to .7z compression. One key difference between gzip's and 7zip's compression methods is that with 7zip it is possible to compress multiple files into one compressed archive while with gzip, it is only possible to compress a singular file at a time.

Qubits	Memory allocation before (Bytes)	Memory allocation after (Bytes)	Improvement(%)
16	994,795	804,424	19.14%
17	1,545,747	1,334,552	13.66%
18	2,601,203	2,388,688	8.17%
19	4,719,235	4,491,016	4.84%
20	8,937,619	8,691,352	2.76%
21	17,350,395	17,085,576	1.53%
22	34,150,939	33,868,056	0.83%
23	67,732,115	67,428,232	0.45%
24	134,867,739	134,542,696	0.24%
25	269,111,243	268,765,720	0.13%

Table 3.1: Memory allocation for 16 up to 25 qubits before and after utilizing ZFP's compressed arrays classes.

```
    public:
  class Vector {
   public:
    Vector() = delete;

    Vector(Pointer&& ptr, unsigned num_qubits)
        : ptr_(std::move(ptr)), num_qubits_(num_qubits) {}

    fp_type* get() {
      return ptr_.get();
    }

    const fp_type* get() const {
      return ptr_.get();
    }

    fp_type* release() {
      num_qubits_ = 0;
      return ptr_.release();
    }

    unsigned num_qubits() const {
      return num_qubits_;
    }

    bool requires_copy_to_host() const {
      return false;
    }

   private:
    Pointer ptr_;
    unsigned num_qubits_;
  };
```

Listing 3.1: Custom Vector class definition utilized by Qsim

4

Results

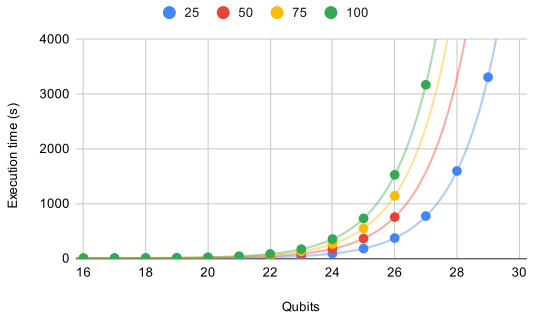
In this chapter we present the results of the initial profiling done on the simulators *qsim* and *Qiskit*. Furthermore, the results of both the FPGA implementation and the data compression are presented and compared with the unoptimized simulator.

4.1 Initial profiling

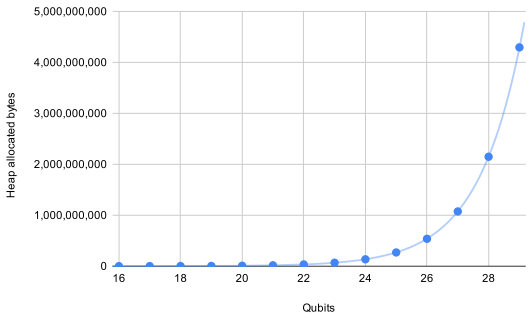
To profile the simulators, we used pseudorandomly generated circuits as described by Boixo *et al.* [32]. As explained in section 2.3, quantum circuits are usually defined by the number of qubits they contain, the number of time steps they are simulated for, known as the circuit *depth*, and a set of gates, each one with a defined time step and input qubits. The testing circuits were generated with 16 to 32 qubits, and depths of 25, 50, 75, and 100. Then they were all executed, and the execution time as well as memory heap allocations were profiled. The computer which these tests were run on was a laptop running Linux, with an Intel i5 4300M CPU at 2.6 GHz and 8 GB of physical memory.

The resulting data for *qsim* in figure 4.1 shows that both memory allocations and execution time increase exponentially with the number of qubits. Furthermore, figure 4.2 shows that these values appear to increase linearly with the circuit depth. This is expected of this type of simulator, as it is a Schrödinger full state vector simulator [14], and such simulators exhibit those complexities [9].

Both execution time and heap allocations increase exponentially with the number of qubits. By performing regression analysis using the data for depth 25 in figure 4.1a to fit an exponential function, we get $T_{exec}(q) \approx 2.02 \cdot 10^{-6} \cdot 2^{1.056q}$s, where q is the number of qubits. Of course the exact numbers depend on the CPU used. Similarly, regression analysis of the data in figure 4.1b gives the function $Memory(q) \approx 8.63 \cdot 2^{0.995q}$B. The memory requirements impose a hard upper limit on the number of qubits that can be simulated on any given computer, because the simulators fail to allocate enough memory when trying to simulate a circuit that requires significantly more memory than is available. The testing machine could simulate no more than 30 qubits, which required most of the available memory: about 8GB. The memory requirements of 31 qubits are $8.63 \cdot 2^{31}$B $\approx$ 17GB. This is roughly twice that of the available RAM on the testing computer. At this point, doubling the amount of memory in the system would allow us to simulate at most one more qubit, since the

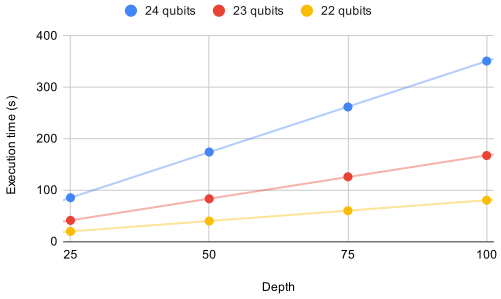


(a) Execution time.

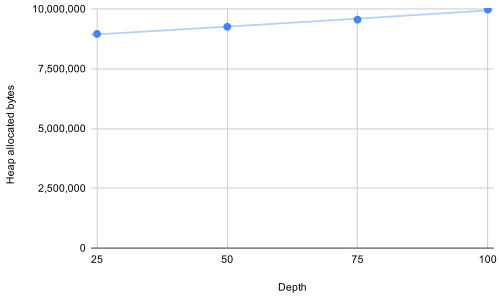


(b) Heap allocations with depth 25.

Figure 4.1: The execution time and heap allocations increase exponentially with the number of qubits in *qsim*. The lines are the regression lines when the data is fit to an exponential function.



(a) Execution time for some different qubit counts.



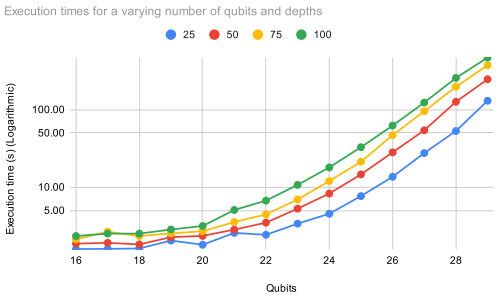
(b) Heap allocations for 20 qubits.

Figure 4.2: The execution time and heap allocations increase linearly with the circuit depth in *qsim*. The lines are the regression lines when the data is fit to a linear function.

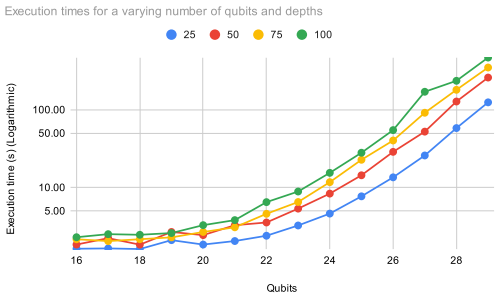
number of bytes that need to be allocated increases exponentially with the qubit count.

Qiskit allows the user to define what type of method the simulator should use for the simulation. If no method is defined, the simulator will pick the method best suited for that setup of qubits and depth. Exploring the performance of the different methods gave the results that can be seen in figure 4.3. While the state vector method seems to have roughly the same performance regardless of cores utilized, matrix product state seems superior at lower depths. While execution times are kept under 8 minutes at depths 25, 50 and 75 - they increase significantly for each qubit added at around the 24 qubits mark, reaching upwards of 9+ hours for simulating a circuit of 28 qubits. In Qiskit's own documentation, the following can be read; "In the worst case, the tensors may grow exponentially. However, the size of the overall structure remains 'small' for circuits that do not have 'many' two-qubit gates. This allows much more efficient operations in circuits with relatively 'low' entanglement."[38] which explains the extremely high execution times for higher

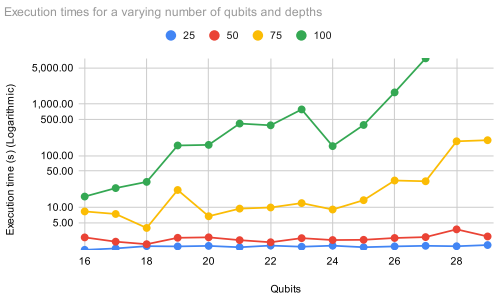
depths. When to use matrix product state over other methods remains a subject of current research. A summary of the execution times for the different methods compared to each other can be seen in figure 4.4.



(a) State vector (Non-parallel)



(b) State vector (parallel)



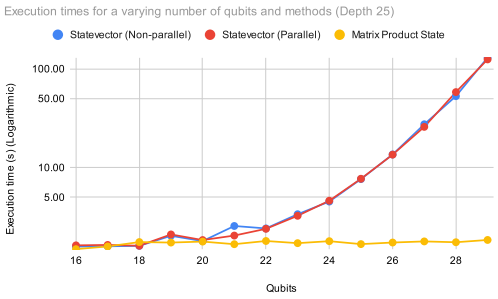
(c) Matrix product state

Figure 4.3: Logarithmic execution times for non-parallel state vector, parallel state vector and matrix product state methods at depths 25, 50, 75 and 100 respectively using Qiskit.

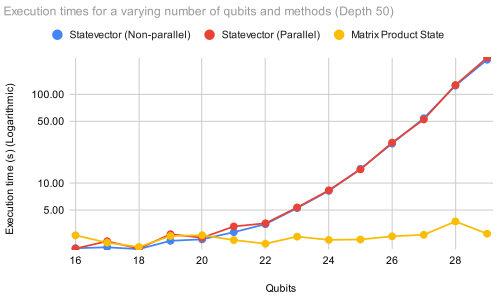
4.2 FPGA implementation

An implementation of the most demanding parts of the simulator was written using Vitis HLS for Vivado flow. This part was the logic for applying gates to qubits, which include the following steps performed for several iterations: gathering data using some strides from the state vector into a contiguous array, multiplying the gate matrix with this array, and then scattering the result back to the state vector. After this implementation worked correctly for floating point numbers, two additional variants were developed: one using fixed point numbers, and one using wider SIMD-like data. The design choices had to be based what is most effective in hardware, but still conform to how *qsim* handles its data. As a result, the final design is based on the structure of *qsim* in order to maintain the same space complexity, while also utilizing techniques to attempt at keeping the code effective for hardware.

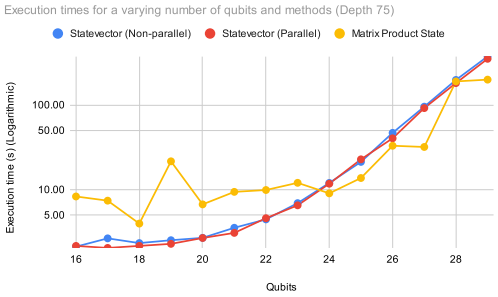
The *qsim* simulator follows a similar algorithm as is described in section 2.4. The simulator maintains the state vector of the whole system in memory. This state vector is $O(2^n)$ space complexity for n qubits. The numbers are represented with 4-byte floating point, and each number is complex, with a real part and an imaginary



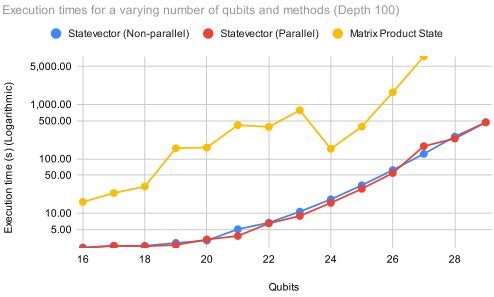
(a) Depth: 25



(b) Depth: 50



(c) Depth: 75



(d) Depth: 100

Figure 4.4: Logarithmic execution times for the different methods examined at depths 25, 50, 75 and 100 respectively using Qiskit

part. Thus, the memory that the state vector requires is $4 \times 2 \times 2^n$ bytes for n qubits. Each gate is applied to this state vector by performing the steps described below, which we identified and separated from the source code of *qsim*. The steps are executed in a loop for a number of iterations, and operate on a certain set of input data. This input data includes the 2^n elements long state vector, the number of input qubits, g, to the gate, the $2^g \times 2^g$ elements large gate matrix, as well as the masks and strides used to index into the state vector. The number of iterations of the loop is calculated as 2^{n-g}. Thus, the time complexity of this loop is $O(2^n)$. The steps (described in the list below and in figure 4.5) operate on complex numbers, so each number consists of two parts, one real, and one imaginary.

1. **Compute start index:** Use the current iteration to find the starting index of the state vector in which to read and write. These calculations use an input array of bitmasks, one for each qubit, and calculates the starting index based on this array. This step has a very small impact on the latency and area usage compared to the others.

2. **Gather:** read from the state vector into local, contiguous state vectors for the affected qubits. This is done by starting from the starting index, and stepping forwards using a certain stride, reading each value from the state vector. The variable g below is the number of input qubits to the gate, which is no larger than six.

```
for i in [0, 2^g] {
    reals[i]       = state_vector[start_index + strides[i]]
    imaginaries[i] = state_vector[start_index + strides[i] + 1]
}
```

3. **Compute:** multiply the gate matrix by the local state vectors we have extracted and store the result locally. This was implemented as a systolic array-style matrix-vector multiplication, with a two-dimensional loop, the innermost of which is unrolled.

4. **Scatter:** write the resulting values back to the state vector, in the same way as when we gathered:

```
for i in [0, 2^g] {
    state_vector[start_index + strides[i]]     = reals[i]
    state_vector[start_index + strides[i] + 1] = imaginaries[i]
}
```

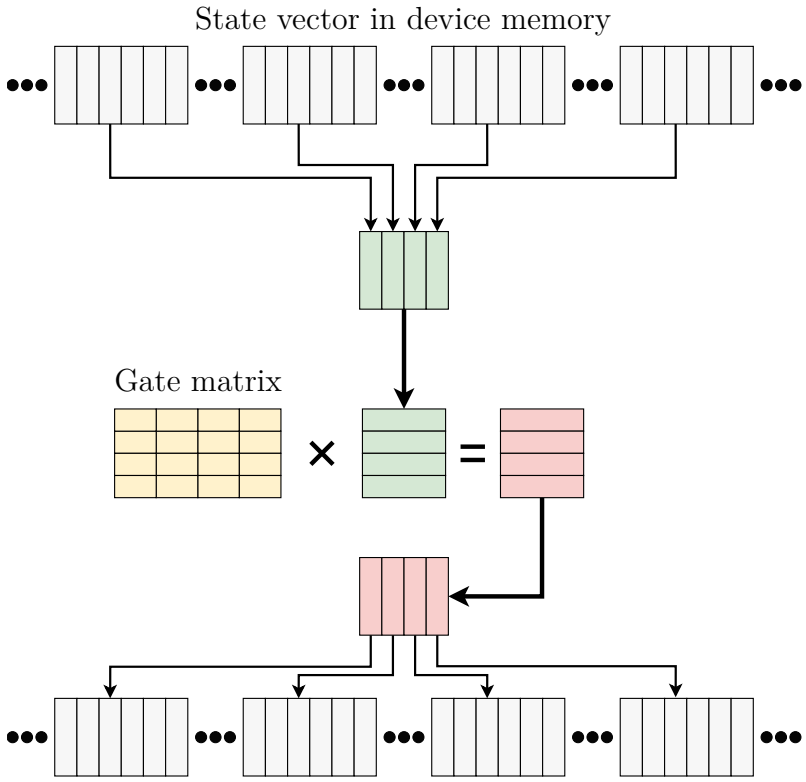


Figure 4.5: A conceptual visualization of what the FPGA design does in each iteration. It gathers complex numbers into a contiguous array, multiplies it with the gate matrix, and writes the result back to the state vector, or a separate output state vector. This example has matrix and array dimensions for a two-qubit gate. Each additional gate qubit requires a doubling of these dimensions.

This structure does not change for any of the three implemented versions: using

4-byte floating point, fixed point, or SIMD. Only minor modifications had to be done, thanks to the use of the built in support from Vitis HLS for both floating point and fixed point, as well as the *hlslib* library which contains customizable wide data types, allowing for SIMD-style programming [36]. The modifications that were needed included rearranging the state vector depending on the SIMD width, as is demonstrated in figure 4.6. This is necessary because the real and imaginary parts need to be independently accessible in the multiplication of complex numbers.

The steps were pipelined using task-level pipelining, allowing each step to run concurrently as long as there is some output data available from the previous step. Out of these steps, the *compute* step takes up the most area on the board, although it only sometimes has the highest latency, as can be seen in the data presented in section 4.2.2. In fact, the *gather* and *scatter* steps also have significant latencies, as they involve reading data from memory, which limits their ability to be parallelized or pipelined due to memory bus contention.

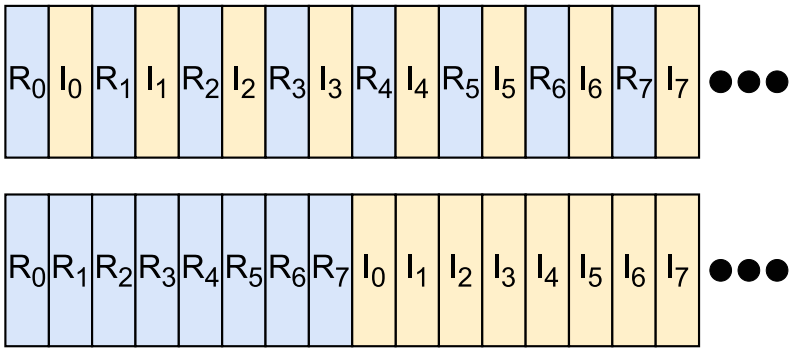


Figure 4.6: Comparison of the ordering of the real and imaginary parts of the state vector at different data widths. At the top is the ordering with single-width data, and at the bottom the ordering with eight-wide SIMD.

Because the gate matrix is variably sized depending on the number of input qubits, the loop bounds in all the steps are variable too. This is not suitable for a hardware design, as a variably bounded loop can not be pipelined. As a result, the hardware design is made to always use loop bounds based on six input qubits; the largest amount supported by *qsim*. This has an impact on the performance of the design, because the gate matrix is now always treated as a (64×64)-matrix. If it were acceptable to only support a subset of all gates that *qsim* supports, the hardware design could be built for a lower amount of input qubits. Note that, as mentioned in section 2.1, all quantum gates can be constructed using the universal two-qubit CNOT gate [2]. This means that all gates are still technically supported by a hardware design that supports a maximum of two qubit inputs, but would require a circuit in which all larger gates are composed of CNOT gates.

The reason that the first, second, and fourth steps are needed, is that quantum gates usually only affect a subset of all the qubits in the circuit, but the stored state vector describes all the qubits in the circuit. As a result, some translation has to be performed in order to make sure the gate matrix is applied only to the state of its

input qubits. As described in section 2.4, the most basic such translation is to use the Kronecker product, padding the gate matrix up to order 2^n for n qubits. This will give it the same width and height as the length of the circuit's state vector, which means the multiplication can just be done as a regular matrix-vector multiplication. While this method avoids the more complex gather-scatter indexing that is required by the method described above, it comes with the drawback of significantly higher memory usage. After being padded, a gate matrix is a square matrix of order 2^n, for an n-qubit circuit. As a result, the matrix has $2^n \cdot 2^n = 2^{2n}$ elements, which would need to be stored along with the 2^n elements of the state vector. On the other hand, using the extraction method described by the four steps above, the order 2^n matrix can be avoided, and only the state vector would need to be stored. If we use 4-byte data types to represent our numbers, each element will take up 8 bytes, as it is a complex number. Thus, for n qubits, the state vector takes $8 \cdot 2^n$ B, and a padded matrix takes $8 \cdot 2^{2n}$ B. In other words, the matrix will take up 2^n times the memory that the state vector does. As a result, a computer system that has enough memory to store the state vector for n qubits would only be able to store the padded matrix for $n/2$ qubits.

The gather-scatter design results in a memory access pattern that is not sequential, but strided at intervals that differ depending on the gate being applied and which qubits are affected. This has some implications on how the data can be handled by the FPGA, since it is difficult to implement a streaming model where the state vector is just traversed from start to finish, with subsections being worked on at any given time. This means that it would be difficult to have the host maintain the whole state vector, while only transferring one section at a time to the FPGA device memory. For a similar reason, scalability is limited. Since the data has to be gathered from various, non-predictable parts of the state vector for each computation, it would be more difficult to subdivide the work between multiple FPGAs, such that each FPGA only maintains a section of the full state vector. Potentially this could be possible if one delimits each FPGA to only a subsection of qubits, since the starting indices and strides are computed based on which qubits are the inputs to the gate. However, this would introduce more complexities when the input qubits to a gate come from different sections. If one accepts that each FPGA would need access to the full state vector, the work could be split along loop iterations instead; the iterations are independent, so they allow for parallelization by dividing the iterations between workers.

Since the whole state vector of the system is required by the FPGA, it would not need to read it from the host more than once, as visualised in figure 4.7. The system could be setup so that the host only transfers the state vector to the FPGA during initialization, and then several gates could be applied, where every gate application reads the gate matrix, the number of input qubits, as well as the start indices and strides that are used for indexing into the state vector. The exception to this is measuring gates, which are not applied the same way as other gates. In our implementation the host would need to apply measuring gates. This reduces the complexity of the FPGA hardware, which seems like a reasonable trade-off since most gates in a circuit are not measuring gates. However, this design means our

hardware design provides significantly less speedup in a case of using a circuit in which a significant proportion of the gates are measuring gates.

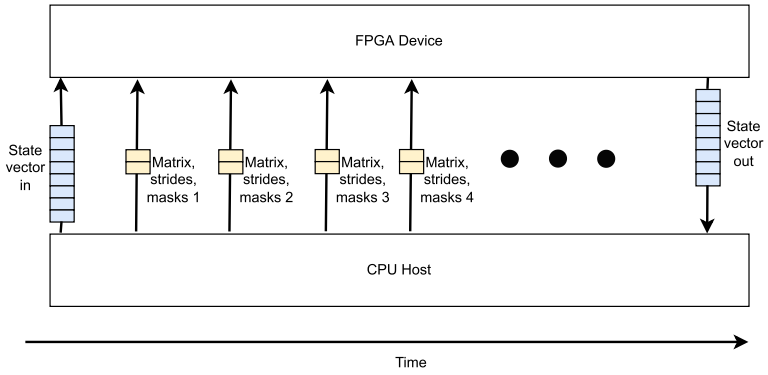


Figure 4.7: How data would flow between a host CPU and the FPGA if our implementation was integrated with *qsim* fully. For every gate that would be applied, the matrix, strides, and masks would be transferred. The state vector would only need to be transferred to the device at initialization, and back to the host when all gates have been applied, or when a measuring gate is to be applied. This is noteworthy since the state vector is usually much larger than the rest of the data. While our design uses data in a way that would be compatible with this mode of operation in principle, it has not been integrated with *qsim* fully, and so it is not possible to actually run it in this manner without integrating it with *qsim*.

Because the FPGA implementation uses task-level pipelining, it is more efficient to maintain a feed-forward flow of data by writing the output to a separate state vector, as visualized in figure 4.5, rather than writing back to the same state vector. This is done by making an initial copy of the input state vector, allowing the *scatter* step to write output to this copy of the input state vector. Of course this requires twice the amount of memory, as we now need to maintain two state vectors. Given the exponential nature of memory usage, this means one less qubit can be simulated with the same amount of memory, since $2^{n+1} = 2 \cdot 2^n$. An obvious attempt at improving this memory usage, at the cost of more calculations, would be to not copy the whole state vector, but only the sections that are affected by the *compute* step. The FPGA could write its output straight to an output array, which would then be iterated over and extracted to the state vector in an additional pass, using the same calculations as in the *compute start index* step. If the FPGA is doing this additional step as well, it could maintain the property of not being dependent on the host to receive a new state vector for every gate.

4.2.1 Number representation

Real numbers have to be represented approximately by computers. Several ways of doing this exist, and floating point is commonly used in modern CPUs and GPUs, with dedicated hardware for that task. An alternative to floating point is fixed point, which requires less from the hardware, as calculations can be computed in the same way as integer calculations. The simulator software uses 32-bit (single

precision) floating point numbers, in order to strike a balance between performance and accuracy [14]. This makes sense as qsim runs on CPUs and GPUs which support floating point by default. In contrast, FPGAs only support the hardware design that is currently implemented on them, and floating point hardware is typically more complex than fixed point hardware, resulting in higher latencies and more area usage. Thus, using floating point becomes a trade for higher precision at the expense of higher latency and increased area usage.

We can reason about the properties of the two ways of representing numbers within the context of quantum computer simulators. Fixed point has fixed precision: the number of bits used to represent the integer part affects the maximum magnitude of the number, and the number of bits for the fractional part affects the precision. Thus, we can express the smallest representable value as 2^{-f}, where f is the number of fractional bits. In contrast, floating point has variable precision depending on the magnitude of the number. This is because of how numbers are represented: using a mantissa and an exponent, as $mantissa \times 2^{exponent}$. Thus, the mantissa bits partition the range $[2^{exponent}, 2^{exponent+1}]$, which varies in size depending on the exponent. Let us denote the number of bits in the mantissa as m. Then we can express the smallest representable value in the range $[2^n, 2^{n+1}]$ as $\frac{2^{n+1}-2^n}{2^m} = \frac{2^n}{2^m}$.

As explained in section 2.1, state vectors are of unit length. Thus the magnitude of any element α is limited to $\alpha \in [-1, 1]$. For a fixed point representation, this means that we need one sign bit and one bit to represent the integer part of α. The remaining 30 bits can be used for representing the fractional part. Let us compare the precision of 32-bit floating point and 32-bit fixed point. Fixed point will have 30 bits for the fractional part, so the smallest representable value will be $2^{-30} \approx 9.3 \cdot 10^{-10}$. On the other hand, floating point will have the lowest precision for values in the range $[2^{-1}, 2^0] = [0.5, 1]$. 32-bit floating point has 23 mantissa bits, so using the expression above, we get $\frac{2^{0.5}}{2^{23}} = 2^{-22.5} \approx 1.7 \cdot 10^{-7}$ between each increment of the mantissa. Clearly, the precision in this range is lower in floating point than in fixed point representation. Let us find the range $[2^x, 2^{x+1}]$, at which the two systems have the same precision:

$$\frac{2^x}{2^{23}} = 2^{-30}$$

$$x = -7$$

In other words: all numbers within the interval $[2^{-7}, 2^{-6}] = [0.0078125, 0.015625]$ have the same precision in both systems. All numbers greater than 2^{-6} have better precision in fixed point with 30 fractional bits, and all positive numbers less than 2^{-7} have better precision in 32-bit floating point. Since floating point precision is symmetric around 0, the same principle holds for negative numbers around the range $[-2^{-6}, -2^{-7}]$. Notably, the irrational number $\frac{1}{\sqrt{2}}$, which is used frequently in quantum gates, can be approximated with greater precision using fixed point, since $\frac{1}{\sqrt{2}} > 2^{-6}$. However, for very small values, floating point provides much higher precision. 32-bit floating point has an 8-bit exponent, which can represent integers between $[-126, 127]$. The highest precision range is thus $[2^{-126}, 2^{-125}]$. This provides significantly greater precision very close to zero, as fixed point with 30 fractional bits

is unable to represent values smaller than 2^{-30}. In order for fixed point to achieve the same precision as 32-bit floating point in this range, it would require around 5 times the bits of floating point.

4.2.2 Area and latency results

As mentioned previously, three variants were implemented: one using floating point, one using fixed point, and one using floating point in a SIMD style. All of these designs were developed in Vitis HLS, and imported into Vivado to be synthesised. All designs were run in simulation, and the PYNQ Z-2 board seen in figure 3.2 was used for testing. The results in the tables below include data about these different metrics:

1. Latency, expressed in cycles, where each cycle is 10 nanoseconds.
2. Interval, the initiation interval between loop iterations. This is the number of cycles until another iteration can begin. Because of the task-level pipelining, it is the same as the latency of the slowest of the tasks.
3. BRAM, block RAM tiles. This is memory inside of the programmable logic of the FPGA.
4. FFs, flip flops. These are often used as storage in the FPGA, for things like FIFO buffers that pass data between logic segments.
5. DSPs, digital signal processors. These are more complex blocks, that are used to implement operations like multiplication.
6. LUTs, look-up tables, the building blocks of the FPGA which can implement logic gates by defining their truth tables.

Table 4.1: The estimated results from Vivado synthesis, without any restriction on the use of hardware. These results may not be the same as those after the place and route step for a specific board. However, because our testing board PYNQ Z-2 is limited to 220 DSPs and 53000 LUTs, these designs can not be synthesised for that board. However, they could be synthesised for a larger board, such as the Alveo U280 or the Virtex Ultrascale+ series.

Variant	Slowest step	Latency (cycles)	Interval	BRAM	FFs	DSPs	LUTs
single width floating point	*compute*	2317	970	16	349738	1024	280651
8-wide SIMD floating point	*gather*	4109	1602	16	933546	8192	1294475
Fixed point	*gather*	1357	578	16	328599	768	172699

Table 4.1 show the estimations after synthesis of the three variants in Vivado. These designs do not use any limitations in Vitis HLS in order to reduce the area utilization, so they can be seen as having the highest area utilization and the lowest latency,

while still supporting up to six input qubits like *qsim*. In all of these designs, the *compute* step is the most significant contributor to area utilization. This is because it contains the matrix-vector multiplication of complex numbers, which for six input qubits is a two-dimensional loop of 128×64 iterations (complex numbers mean one dimension has $2 \cdot 64$ elements), with the inner loop fully unrolled to increase paralellization. This unrolling can be reduced to decrease area utilization at the cost of latency. Additionally, statements could be introduced via Vitis HLS to limit the number of a certain operation, or reduce array partitioning, both of which would also reduce the area at the cost of increased latency. Another alternative is reducing the number of maximum input qubits for the gate, which can be done by changing a single parameter. Experimentation showed that this value affects area usage and latency significantly, because the gate matrix size doubles in both dimensions for every additional input qubit, which also means the matrix-vector multiplication becomes more expensive.

As can be seen in table 4.1, the fixed point variant has the lowest amount in all of the metrics. This is as expected, since fixed point can be implemented using integer operations, which are cheaper than floating point. The 8-wide SIMD variant has exactly eight times as many DSPs as the single-width floating point, which is because each iteration of the loop will have eight as many floating point operations. Notably, the number of LUTs and flip-flops are not eight times as large as for the single-width design, indicating that the design scales fairly well for wider data types.

The designs were modified to support up to three input qubits, and run in the PYNQ Z-2 board. The results from Vivado of the place and route step of these designs are listed in table 4.2. As can be seen from these results, the latency of the *compute* step in the SIMD version (130 cycles) is similar to that of the single width floating point version (122 cycles). The small increase is most likely because the amount of DSPs had to be limited in order to fit the PYNQ Z-2 board. On the other hand, the *gather* and *scatter* steps have higher latencies in the SIMD version, presumably due to the larger amount of memory that is being read/written.

In addition to the PYNQ Z-2, the floating point design was also synthesised for the Virtex Ultrascale+ HBM, which is a significantly larger board which can fit the full design for six gate qubits. The results presented in table 4.3 show the area utilization there. When comparing the synthesis results for floating point in table 4.1 with these results, we notice that the actual usage is lower than the estimates from synthesis. The same conservative synthesis estimates were found for the limited PYNQ designs, when comparing to the post-place and route results.

While the *compute* step is clearly the one that impacts area utilization the most, it can also be parallelized and pipelined fairly effectively to reduce latency. On the other hand, the *gather* and *scatter* steps are quite memory bound, and thus much more difficult to parallelize because of the limitations of the data bus. As a result, these steps are often of comparable or higher latency than the *compute* step.

Table 4.2: These are the area results from implementing the designs in the PYNQ testing board, as well as the latencies for each step reported by Vitis HLS (the *compute start index* step has been omitted as it is very small in comparison). These designs are set to a maximum of three qubits, in order to fit in the hardware of this board. The SIMD also had to be reduced to be 4-wide for the same reason, and the amount of floating point multiplication operations had to be limited to reduce the DSP usage.

(a) Single width floating point.

	FFs	DSPs	LUTs	BRAM tiles	Latency
ApplyGate IP	25017	128	13960	1.5	294
Gather	1313	0	410	0	89
Compute	12151	128	6827	0	122
Scatter	340	0	376	0	81
Percentage of PYNQ max	25.16%	58.18%	28.75%	1.07%	-

(b) 4-wide SIMD floating point.

	FFs	DSPs	LUTs	BRAM tiles	Latency
ApplyGate IP	48514	206	27598	1.5	398
Gather	3047	0	414	0	137
Compute	27576	206	17127	0	130
Scatter	540	0	654	0	129
Percentage of PYNQ max	47.24%	93.64%	54.39%	1.07%	-

(c) Fixed point.

	FFs	DSPs	LUTs	BRAM tiles	Latency
ApplyGate IP	21916	128	13664	1.5	174
Gather	2355	0	1382	0	73
Compute	7731	128	3797	0	34
Scatter	690	0	1531	0	65
Percentage of PYNQ max	22.24%	58.18%	28.19%	1.07%	-

4.3 Full integration with *qsim*

As stated previously, our FPGA implementation is not fully integrated with *qsim*. In order to do this, code would need to be added that interfaces with the FPGA. This code would set up the FPGA before each gate application, by passing the same data that we did in the Python interface when testing with the PYNQ Z-2 board. This could be done using a library such as OpenCL, or another tool with

Table 4.3: The full floating point design for the Virtex Ultrascale+ HBM. Note that the latency of the IP is not indicative of the throughput. Due to the task-level pipelining, the interval for each iteration of these steps is the same as the maximum latency of all the steps. In this case this is *gather* with latency of 641 cycles.

	FFs	DSPs	LUTs	BRAM tiles	Latency
ApplyGate IP	380859	1024	168461	11.0	1741
Gather	16124	0	5779	0	641
Compute	301584	1024	12518	0	521
Scatter	332	0	2012	0	577
Percentage of Board max	14.61%	11.35%	12.92%	0.55%	-

similar capabilities. Since we lack a full integration with *qsim*, we can not make a true side by side comparison between *qsim* using an FPGA, and the regular CPU version. Even if we could do that, the specific speedup would of course be quite different depending on the particular models of CPU and FPGA used, as well as how the data is transferred to the FPGA (some FPGAs such as the PYNQ Z-2 use ethernet, while others use the PCIe bus). Nevertheless, we can still calculate an estimate of how our FPGA version compares to the CPU version, by measuring the execution time on the CPU of the parts that we implemented for an FPGA, and comparing that to the estimated latencies of the single-width floating point FPGA version synthesised for the Virtex Ultrascale+ HBM, found in table 4.3. The initiation interval of this version is 641 cycles, which we can use as an estimate of the latency of each iteration of the FPGA version. This latency does not depend on the number of input qubits, because the FPGA always works with the largest possible gate matrix, for reasons described in section 4.2. In comparison, the number of cycles of the CPU version depends on the number of gate input qubits. These cycle counts, measured with the *rdtsc* instruction, are shown in table 4.4. These numbers constitute the average number of cycles for one iteration of the steps described in section 4.2. For six input qubits, the CPU version requires significantly more cycles than the FPGA: 8131 (with *O3* optimization level) compared to the FPGA's 641. However, the latency of a cycle depends on the clock frequency, and CPUs generally have higher clock frequencies than FPGAs. The CPU used to extract the numbers in table 4.4 was an Intel i7-8550U, which has a base frequency of 1.8 GHz and dynamic frequency scaling up to 4 GHz [39]. The Virtex Ultrascale+ HBM version targeted 300 MHz, which is a sixth of the CPU's base frequency. If we divide the CPU cycles to compensate we get $\frac{8131}{6} \approx 1334$, which is roughly twice the latency of the FPGA version. However, if the CPU can run at its purported maximum of 4 GHz for all the gates, it could reach a similar latency per gate as the FPGA could at 300 MHz. Other FPGAs might be able to run at a higher frequency, however that is not something that has been investigated in this work. From table 4.2 we see that increasing the data width does not affect the latency of the *compute* step significantly, so the comparison of the CPU SIMD version and the FPGA SIMD version should be similar to the above.

Table 4.4: The average number of cycles for one iteration of the steps described in section 4.2, for one to six gate input qubits, with different levels of compiler optimizations.

gate qubits	-O3	-O2	-O1	-O0
1	15	16	17	61
2	32	40	43	186
3	124	145	165	608
4	485	532	598	2212
5	1924	1952	2309	8248
6	8131	8343	8918	32604

4.4 Data compression integration

As the definition of the state vector used in Qsim does not allow for referencing of the full vector, a methodology of dumping the state vector to a file at certain time steps and performing data compression on it offline was used. During simulation, the state vector starts out with mostly zeroes, to then be populated more for each iteration of the functions responsible for applying gates to qubits. Theoretically, this means that compression ratios will be very large at the first iterations and then decrease for each time the state vector is further populated. This claim is supported by the data seen in Table 4.5. Compression ratio starts at a very high value, as expected and then settles at around 3. As can be seen in Appendix A.1, A.2 and A.3 this is in fact true for all of our lossless data compression experiments.

Table 4.5: Compression ratio evolution over 100 iterations for a 16 qubit setup using Gzip

Qubits: 16			
Iteration count	**Memory allocated before compression (Bytes)**	**Memory allocated after compression (Bytes)**	**Compression ratio**
0	393216	434	906
10	859936	12121	70
100	1685714	543020	3
119	1671384	538781	3

4.4.1 Data compression results

Utilizing data compression in the simulator indeed seems promising. The function which serves as the major bottleneck for the simulator was worked on such that it now utilizes ZFP's compressed array classes as well as making a dump of the state vector at certain time steps to be compressed outside simulation. By compressing the state vector offline, it was possible to find potential in memory allocation reduction. The data from these compression experiments can be seen in Figure 4.6. In the appendix the complete test results can be found. As can be observed, 7zip starts

out with the greatest compression rate when compressing to the 7z format, but at the higher time steps all compression methods end up at around a 3:1 compression rate.

Table 4.6: Compression ratios for offline lossless compression of the state vector for circuits consisting of 16 up to 25 qubits

No. qubits in circuit	Gzip	7zip(Bin to zip)	7zip(Bin to 7z)
16	3.10	3.49	3.64
17	3.11	3.52	3.68
18	3.12	3.54	3.71
19	3.11	3.53	3.71
20	2.97	3.37	3.61
21	3.05	3.46	3.71
22	3.04	3.46	3.71
23	3.05	3.47	3.73
24	3.04	3.46	3.74
25	3.10	3.55	3.81

As can be read in section 3.3, the initial experiments showed that for each qubit added, the memory allocation would increase exponentially in such a way that the allocated memory would double for every qubit added. With this given, it is clear that to find any improvements in terms of memory allocation, we would need to at least halve it which would lead to us now being able to add one additional qubit to our circuit. While the compression rate of 3 that was reached does not give us any further improvement over 2, we have still reached the goal of increasing the amount of possible qubits to simulate from the base simulator design.

As expected, when utilizing ZFP's lossy file compressor, the compression ratios increased significantly compared to lossless. These results can be seen in Figure 4.7. At 16 qubits the compression rate already exceeds all of our lossless compression tests, and for each qubit added, this rate increases up to 18.

Table 4.7: Compression ratios for offline lossy compression of the state vector for circuits consisting of 16 up to 25 qubits

No. qubits in circuit	ZFP
16	4.58
17	4.72
18	5.27
19	6.03
20	7.11
21	8.55
22	9.75
23	10.9
24	11.2
25	18.3

5

Conclusion

5.1 Discussion

The results of this work show promise for utilizing both an FPGA design or data compression for quantum computer simulation. The time-consuming task of applying gates to the state vector is a task well-suited for FPGAs. However, the non-sequential memory access pattern in our design is more problematic since it limits the parallelizability. One way to circumvent this issue is to avoid the indexing altogether, by expanding all gate matrices to apply to the whole state vector, as described in section 2.4, and implemented previously by Pilch and Długopolski [20]. However, this halves the number of qubits that can be simulated with a given amount of memory, as demonstrated in section 4.2. In our view, the trade-off of more complex indexing for doubling the number of possible qubits is worth it, but this will of course vary depending on any user's specific use-case.

Another comparison to make is whether it is more useful to widen the SIMD units, or to increase parallelism with loop unrolling in the matrix-vector multiplication (in the case where the FPGA area does not allow both). Because of the task-level pipelining, the matrix-vector multiplication should be able to start before all of its input data has been read from memory, so wider SIMD would require fewer iterations, and may balance the latency of the *gather*, *compute*, and *scatter* steps better.

In addition, our results show that by utilizing lossless compression it is possible to reduce memory allocated per qubit. The reduced memory allocation allows for at least one more qubit to be added to quantum computing circuits while still staying inside the available memory. By utilizing lossy compression it was found that compression ratios of up to $\sim$18 was possible for the circuits that were experimented on which in theory would allow us to add 4 additional qubits. However, for each qubit added to a quantum computer circuit, additional error correction has to be done in order to achieve accurate results. Briefly mentioned in section 2.5, Fidelity is often used to determine the probability that a quantum state will pass a test to identify as another - meaning that some margin of error is often allowed if we don't need 100% accuracy. However, the higher our compression rate is with lossy compression, the more data will be lost and thus more errors will be introduced. It is important to note that the work conducted in this project did not touch on how

closely to the uncompressed vector resembles the original one. Thus we cannot say for sure that these compression ratios are attainable when we want accurate results. The main takeaway from our findings is that by utilizing lossless compression, it is fully possible to increase the number of qubits in a circuit by one. As we do not corrupt any data with lossless compression, we can expect the accuracy of the system to not degrade from this addition. Our lossy compression experiments show that it is possible to add 4 additional qubits - potenitally more, while staying within the limits of the available memory of our system.

Neither of our optimizations are fully integrated with the *qsim* simulator. The main reasons that this engineering work was not done as part of this thesis were time constraints, and a larger focus on investigating the effects of the optimizations rather than integrating them fully with *qsim*. As a result, more work would be needed in order to be able to run *qsim* with these optimizations in the same way as the already existing AVX and GPU optimizations. As a result, assessing the speedup of utilizing FPGAs compared to the other pre-existing optimizations is somewhat difficult. Although, even if the FPGA implementations were fully integrated with *qsim*, the speedup would vary significantly depending on the specific combinations of CPU, GPU, and FPGA models that were used in the comparison. Data compression in this project was done in a rather unconventional way where the data to be compressed is dumped to a file which was then compressed after the simulation had finished. This method shows us the compression rates that we can reach with the given data, but to what extent the data is still usable for simulations remains unanswered. It is expected that this will not be an issue when utilizing lossless compression, but lossy compression will definitely be more problematic in the matter. In a sophisticated integration of data compression one would ideally compress and decompress the data during simulations and dump data to memory rather than a separate file.

5.2 Future work

Because this work was an initial exploration of how quantum computer simulations can be optimized, the results present several avenues for future work. A natural continuation of this work would be to combine the two methods of optimization implemented here, by both using memory compression and an FPGA to accelerate the computation at the same time. There are several challenges with that, such as deciding whether compression should be performed by the FPGA, or only by the host, and investigating how the type of compression affects the FPGA design.

Additionally, the FPGA design could be improved in a few ways on its own. Firstly, it could be implemented using a hardware description language such as VHDL or Verilog, instead of the current high-level synthesis, to investigate if that would result in noteworthy gains in latency, power, or area usage. Secondly, one could investigate the effects of letting the host perform the steps that gather and scatter data to memory, which would simplify the FPGA design. Another possibility is to investigate ways of improving these steps in the FPGA design, as they are the bottlenecks in

terms of latency in many of the variants shown in our results. Perhaps this could be done with some sort of prefetching system which could utilize the strides and start indices given as input to the design. Thirdly, an area of interest might be processing in memory, especially considering the non-sequential access patterns exhibited by the aforementioned gather and scatter steps. If an in-memory processing system could take the relevant parts of the gate matrix and apply them to each section of memory, it might be an improvement over the current design. The reason for this is that less data would need to be read, because the gate matrix has at maximum $2^6 * 2^6 = 2^{12}$ elements, so state vectors for more than 13 qubits will be larger than the gate matrices. Finally, the design requires significantly less area when a lower maximum number of gate input qubits are supported. For example, having a maximum of three instead of six would shrink the gate matrix from $128 \cdot 64 = 8192$ numbers, to only $16 \cdot 8 = 128$ numbers. The problem with this is of course that the design will not support as large gates as the simulator it is meant to interact with. However, given that CNOT gates are universal, as described in section 2.2, one could investigate a scheme of transforming all larger gates in the circuit to being composed of CNOT gates. Because the circuit is given as an input to the simulator and does not change dynamically, this would only have to happen once during initialization.

Compression methods in this project only take into account to which grade the state vector can be compressed. While this presents the potential performance improvements, future work that aims to continue working on these improvements also needs to look into the accuracy to which the state vector can be reconstructed after lossy compression. In general, there is nothing hindering from adding qubits to a quantum computer. However, for each qubit added, additional error correction must be performed in order to keep the system stable and reach the correct output.

5.3 Conclusion

This work has examined how quantum computer simulators can be optimized, with a focus on an FPGA design and data compression. The research question was: *How effective is applying FPGAs and data compression techniques on performance and precision of existing quantum computer simulations, and what are the effects on the results of simulation*, and the work went more or less according to plan to answer this question. It began with profiling the simulators *qsim* and *Qiskit*. Because of time limitations and issues working with *Qiskit*, only *qsim* was used as the basis for our implementations. Using the profiling, the relevant parts to optimize were identified: applying gates to the state vector. These were the focus when developing the optimizations: three variants of an FPGA design, implemented and tested on a PYNQ Z-2 board, as well as data compression algorithms tested on the memory representing the state vector.

The results of these optimizations show promise for improving quantum computer simulations, and are an indication that further research can be done in this area. Several potential ways of improving the results of this work have been presented, and by spending more time on the FPGA design it could surely be improved with

regards to area utilization and execution time. Most of the goals we had initially with this thesis were achieved, although more work could be done to connect the optimizations more tightly to *qsim*, so that they could be used in a similar fashion to the already existing GPU and AVX optimized versions available there. While the optimizations in this work are based on *qsim*, they are not exclusive to it, and should be applicable to other simulators that work in similar ways.

The results presented in regard to data compression show that there is indeed potential for improvement in this area as well. While lossy data compression methods expectedly showed significant improvement to that of lossless data compression, fidelity was not measured and thus we cannot guarantee that the resulting output state vector will pass as the same as the state vector produced when no - or lossless data compression is used. Nevertheless, the goal of reducing memory allocation of qubits to add more to circuits was reached when utilizing several different algorithms for offline data compression. It is definitely possible that by instead performing online data compression and refactoring some boilerplate code that compression ratios could be further increased and as such this is an area worth looking further into in future work.

Bibliography

[1] R. P. Feynman, "Simulating physics with computers," *International Journal of Theoretical Physics*, vol. 21, pp. 467–488, 1982.

[2] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information*, 10th ed. The Edinburgh Building, Cambridge CB2 8RU, UK: Cambridge University Press, 2010.

[3] C. Bennett, L. Bishop, S. Bravyi, A. Cross, J. Gambetta, P. Nation, J. Smolin, and K. Temme, "Learn quantum computing: a field guide." [Online]. Available: https://quantum-computing.ibm.com/composer/docs/iqx/guide/

[4] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, ser. STOC '96. New York, NY, USA: Association for Computing Machinery, 1996, p. 212–219. [Online]. Available: https://doi.org/10.1145/237814.237866

[5] P. Shor, "Algorithms for quantum computation: discrete logarithms and factoring," in *Proceedings 35th Annual Symposium on Foundations of Computer Science*, 1994, pp. 124–134.

[6] L. T. Yang, L. Xu, S.-S. Yeo, and S. Hussain, "An integrated parallel gnfs algorithm for integer factorization based on linbox montgomery block lanczos method over gf(2)," *Computers & Mathematics with Applications*, vol. 60, no. 2, pp. 338–346, 2010, advances in Cryptography, Security and Applications for Future Computer Science. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0898122110000428

[7] Wallenberg Centre for Quantum Technology, "Quantum computing," 2020. [Online]. Available: https://www.chalmers.se/en/centres/wacqt/discover/Pages/Quantum-computing.aspx

[8] C. Wang, X. Li, H. Xu, Z. Li, J. Wang, Z. Yang, Z. Mi, X. Liang, T. Su, C. Yang, and et al., "Towards practical quantum computers: Transmon qubit with a lifetime approaching 0.5 milliseconds," *npj Quantum Information*, vol. 8, no. 1, 2022.

[9] A. Fatima and I. L. Markov, "Faster Schrödinger-style simulation of quantum

circuits," *HPCA 2021*, 2020. [Online]. Available: https://doi.org/10.48550/arXiv.2008.00216

[10] T. Gagliardoni, "Quantum attack resource estimate: Using shor's algorithm to break rsa vs dh/dsa vs ecc," Aug 2021. [Online]. Available: https://research.kudelskisecurity.com/2021/08/24/quantum-attack-resource-estimate-using-shors-algorithm-to-break-rsa-vs-dh-dsa-vs-ecc/

[11] X.-C. Wu, S. Di, E. M. Dasgupta, F. Cappello, H. Finkel, Y. Alexeev, and F. T. Chong, "Full-state quantum circuit simulation by using data compression," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, nov 2019. [Online]. Available: https://doi.org/10.1145%2F3295500.3356155

[12] P. Ball, "First quantum computer to pack 100 qubits enters crowded race," *Nature*, Nov 2021. [Online]. Available: https://www.nature.com/articles/d41586-021-03476-5

[13] "IBM Quantum simulators." [Online]. Available: https://quantum-computing.ibm.com/lab/docs/iql/manage/simulator/

[14] Q. A. team and collaborators, "qsim," Sep. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.4023103

[15] Microsoft, "Q# and the quantum development kit." [Online]. Available: https://azure.microsoft.com/en-us/resources/development-kit/quantum-computing/#features

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

[17] P. Zhang, J. Yuan, and X. Lu, "Quantum computer simulation on multi-gpu incorporating data locality," in *Algorithms and Architectures for Parallel Processing*, G. Wang, A. Zomaya, G. Martinez, and K. Li, Eds. Cham: Springer International Publishing, 2015, pp. 241–256.

[18] M. Smelyanskiy, N. P. D. Sawaya, and A. Aspuru-Guzik, "qhipster: The quantum high performance software testing environment," 2016. [Online]. Available: https://arxiv.org/abs/1601.07195

[19] H. D. Raedt, F. Jin, D. Willsch, M. Willsch, N. Yoshioka, N. Ito, S. Yuan, and K. Michielsen, "Massively parallel quantum computer simulator, eleven years later," *Computer Physics Communications*, vol. 237, pp. 47–61, apr 2019. [Online]. Available: https://doi.org/10.48550/arXiv.1805.04708

[20] J. Pilch and J. Długopolski, "An fpga-based real quantum computer emulator," *Journal of Computational Electronics*, 2019. [Online]. Available: https://doi.org/10.1007/s10825-018-1287-5

[21] C. P. Williams, *Quantum Gates*. London: Springer London, 2011, pp. 51–122. [Online]. Available: https://doi.org/10.1007/978-1-84628-887-6_2

[22] R. Muradian and D. Frias, "Generators and roots of quantum logic gates," 2005. [Online]. Available: https://arxiv.org/abs/quant-ph/0511250

[23] IBM, "Grover's algorithm." [Online]. Available: https://quantum-computing.ibm.com/composer/docs/iqx/guide/grovers-algorithm

[24] Z.-Y. Chen, Q. Zhou, C. Xue, X. Yang, G.-C. Guo, and G.-P. Guo, "64-qubit quantum circuit simulation," *Science Bulletin*, vol. 63, no. 15, pp. 964–971, aug 2018. [Online]. Available: https://doi.org/10.48550/arXiv.1802.06952

[25] E. Pednault, J. Gunnels, G. Nannicini, L. Horesh, T. Magerlein, E. Solomonik, and R. Wisnieff, "Breaking the 49-qubit barrier in the simulation of quantum circuits," 10 2017.

[26] J. Niwa, K. Matsumoto, and H. Imai, "General-purpose parallel simulator for quantum computing," *Physical Review A*, vol. 66, no. 6, dec 2002. [Online]. Available: https://doi.org/10.48550/arXiv.1704.01127

[27] T. Häner and D. S. Steiger, "0.5 petabyte simulation of a 45-qubit quantum circuit," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, nov 2017. [Online]. Available: https://doi.org/10.48550/arXiv.1704.01127

[28] J. loup Gailly and M. Adler. Gzip. [Online]. Available: https://www.gzip.org/

[29] 7zip. [Online]. Available: https://www.7-zip.org/

[30] P. Lindstrom, M. Salasoo, M. Larsen, S. Herbein, and M. Miller. Zfp. [Online]. Available: https://zfp.readthedocs.io/en/release0.5.5/introduction.html

[31] S. Hauck and A. DeHon, *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. 30 Corporate Drive, Suite 400, Burlington, MA: Elsevier, 2008.

[32] S. Boixo, S. V. Isakov, V. N. Smelyanskiy, R. Babbush, N. Ding, Z. Jiang, M. J. Bremner, J. M. Martinis, and H. Neven, "Characterizing quantum supremacy in near-term devices," *Nature Physics*, vol. 14, no. 6, p. 595–600, Apr 2018. [Online]. Available: http://dx.doi.org/10.1038/s41567-018-0124-x

[33] Valgrind, "Massif: a heap profiler," 2022. [Online]. Available: https://valgrind.org/docs/manual/ms-manual.html

[34] ——, "Callgrind: a call-graph generating cache and branch prediction profiler," 2022. [Online]. Available: https://valgrind.org/docs/manual/cl-manual.html

[35] J. Weidendorfer, "KCachegrind: Call Graph Viewer," 2013. [Online]. Available: https://kcachegrind.github.io/html/Home.html

[36] J. de Fine Licht and T. Hoefler, "hlslib: Software engineering for hardware design," *arXiv:1910.04436*, 2019.

[37] P. Lindstrom, M. Salasoo, M. Larsen, S. Herbein, and M. Miller. Compressed arrays. [Online]. Available: https://zfp.readthedocs.io/en/release0.5.4/arrays.html#array-classes

[38] Q. D. Team. Matrix product state simulation method. [Online]. Available: https://qiskit.org/documentation/tutorials/simulators/7_matrix_product_state_method.html

[39] I. Corporation, "Intel® core™ i7-8550u processor." [Online]. Available: https://www.intel.com/content/www/us/en/products/sku/122589/intel-core-i78550u-processor-8m-cache-up-to-4-00-ghz/specifications.html

[40] E. W. Weisstein, "Kronecker product." [Online]. Available: https://mathworld.wolfram.com/KroneckerProduct.html

A

Appendix 1

A.1 Gzip compression results

Qubits: 16			
Iteration count	**Memory allocated before compression (Bytes)**	**Memory allocated after compression (Bytes)**	**Compression ratio**
0	393216	434	906
10	859936	12121	70
100	1685714	543020	3
119	1671384	538781	3
Qubits: 17			
Iteration count	**Memory allocated before compression (Bytes)**	**Memory allocated after compression (Bytes)**	**Compression ratio**
0	768432	815	942
10	1719872	24092	71
100	3403140	1089989	3
127	3379267	1086121	3
Qubits: 18			
Iteration count	**Memory allocated before compression (Bytes)**	**Memory allocated after compression (Bytes)**	**Compression ratio**
0	1572864	1577	997
10	2506304	25655	97
100	6885945	2244642	3
135	6837858	2188881	3
Qubits: 19			
Iteration count	**Size before compression (Bytes)**	**Size after compression (Bytes)**	**Compression ratio**
0	3145728	3101	1014
10	4079168	27345	149
100	13851945	4478214	3

Table A.1 continued from previous page

143	13804883	4436111	3
Qubits: 20			
Iteration count	**Size before compression (Bytes)**	**Size after compression (Bytes)**	**Compression ratio**
0	6291456	6149	1023
10	8158528	55614	146
100	27799091	8918720	3
151	27894600	9380414	2
Qubits: 21			
Iteration count	**Size before compression (Bytes)**	**Size after compression (Bytes)**	**Compression ratio**
0	12582912	12258	1026
10	14450048	62212	232
100	55932476	17702059	3
159	55936578	18315277	3
Qubits: 22			
Iteration count	**Size before compression (Bytes)**	**Size after compression (Bytes)**	**Compression ratio**
0	25165824	24464	1028
10	27032960	74418	363
100	112111010	35310069	3
167	112109578	36818920	3
Qubits: 23			
Iteration count	**Size before compression (Bytes)**	**Size after compression (Bytes)**	**Compression ratio**
0	50331648	48889	1029
10	54066304	152935	353
100	224449662	58361353	3
175	224518195	73445104	3
Qubits: 24			
Iteration count	**Size before compression (Bytes)**	**Size after compression (Bytes)**	**Compression ratio**
0	100663296	97739	1029
10	104398080	202085	516
100	449097025	137733323	3
183	449184941	147630405	3
Qubits: 25			

Table A.1 continued from previous page

Iteration count	Size before compression (Bytes)	Size after compression (Bytes)	Compression ratio
0	201326592	195439	1030
10	205061376	299771	684
100	898315864	283504507	3
175	898492935	290732450	3
191	898493099	289163740	3

A.2 7zip compression results (.bin to .zip)

Qubits: 16			
Iteration count	**Memory allocated before compression (Bytes)**	**Memory allocated after compression (Bytes)**	**Compression ratio**
0	393216	639	615
10	859936	19952	43
100	1685714	483060	3
119	1671384	478715	3
Qubits: 17			
Iteration count	**Memory allocated before compression (Bytes)**	**Memory allocated after compression (Bytes)**	**Compression ratio**
0	768432	1098	699
10	1719872	40155	42
100	3403140	960436	3
127	3379267	958691	3
Qubits: 18			
Iteration count	**Memory allocated before compression (Bytes)**	**Memory allocated after compression (Bytes)**	**Compression ratio**
0	1572864	2031	774
10	2506304	41121	60
100	6885945	1981105	3
135	6837858	1927852	3
Qubits: 19			
Iteration count	**Size before compression (Bytes)**	**Size after compression (Bytes)**	**Compression ratio**
0	3145728	3896	807
10	4079168	42973	94
100	13851945	3948051	3
143	13804883	3909582	3

Table A.2 continued from previous page

Qubits: 20			
Iteration count	**Size before compression (Bytes)**	**Size after compression (Bytes)**	**Compression ratio**
0	6291456	7614	826
10	8158528	90219	90
100	27799091	7859686	3
151	27894600	8264812	3
Qubits: 21			
Iteration count	**Size before compression (Bytes)**	**Size after compression (Bytes)**	**Compression ratio**
0	12582912	15061	835
10	14450048	97565	148
100	55932476	15577265	3
159	55936578	16136246	3
Qubits: 22			
Iteration count	**Size before compression (Bytes)**	**Size after compression (Bytes)**	**Compression ratio**
0	25165824	29958	840
10	27032960	112461	240
100	112111010	31000693	3
167	112109578	32359716	3
Qubits: 23			
Iteration count	**Size before compression (Bytes)**	**Size after compression (Bytes)**	**Compression ratio**
0	50331648	59751	842
10	54066304	232931	232
100	224449662	50638454	4
175	224518195	64623491	3
Qubits: 24			
Iteration count	**Size before compression (Bytes)**	**Size after compression (Bytes)**	**Compression ratio**
0	100663296	119324	843
10	104398080	292336	357
100	449097025	121023874	3
183	449184941	129569670	3
Qubits: 25			
Iteration count	**Size before compression (Bytes)**	**Size after compression (Bytes)**	**Compression ratio**

Table A.2 continued from previous page

0	201326592	238470	844
10	205061376	411495	498
100	898315864	247719722	3
175	898492935	254688798	3
191	898493099	253041016	3

A.3 7zip compression results (.bin to .7z)

Qubits: 16			
Iteration count	**Memory allocated before compression (Bytes)**	**Memory allocated after compression (Bytes)**	**Compression ratio**
0	393216	268	1467
10	859936	17671	48
100	1685714	455499	3
119	1671384	458166	3
Qubits: 17			
Iteration count	**Memory allocated before compression (Bytes)**	**Memory allocated after compression (Bytes)**	**Compression ratio**
0	768432	323	2379
10	1719872	26317	65
100	3403140	827273	4
127	3379267	918276	3
Qubits: 18			
Iteration count	**Memory allocated before compression (Bytes)**	**Memory allocated after compression (Bytes)**	**Compression ratio**
0	1572864	434	3624
10	2506304	35126	71
100	6885945	1868934	3
135	6837858	1842547	3
Qubits: 19			
Iteration count	**Size before compression (Bytes)**	**Size after compression (Bytes)**	**Compression ratio**
0	3145728	665	4730
10	4079168	35348	115
100	13851945	3557438	3
143	13804883	3712728	3
Qubits: 20			

Table A.3 continued from previous page

Iteration count	Size before compression (Bytes)	Size after compression (Bytes)	Compression ratio
0	6291456	1128	5577
10	8158528	68041	119
100	27799091	7032781	3
151	27894600	7708936	3
Qubits: 21			
Iteration count	**Size before compression (Bytes)**	**Size after compression (Bytes)**	**Compression ratio**
0	12582912	2044	6156
10	14450048	67788	213
100	55932476	12792374	4
159	55936578	15055607	3
Qubits: 22			
Iteration count	**Size before compression (Bytes)**	**Size after compression (Bytes)**	**Compression ratio**
0	25165824	3876	6492
10	27032960	69620	388
100	112111010	23165245	4
167	112109578	30194480	3
Qubits: 23			
Iteration count	**Size before compression (Bytes)**	**Size after compression (Bytes)**	**Compression ratio**
0	50331648	7539	6676
10	54066304	151505	356
100	224449662	39733658	5
175	224518195	60062185	3
Qubits: 24			
Iteration count	**Size before compression (Bytes)**	**Size after compression (Bytes)**	**Compression ratio**
0	100663296	14944	6736
10	104398080	159085	656
100	449097025	99280103	4
183	449184941	120047255	3
Qubits: 25			
Iteration count	**Size before compression (Bytes)**	**Size after compression (Bytes)**	**Compression ratio**
0	201326592	29680	6783

Table A.3 continued from previous page

10	205061376	173900	1179
100	898315864	191452620	4
175	898492935	236780906	3
191	898493099	235478002	3

B

Appendix 2

B.1 Kronecker product

The Kronecker product is an operator on matrices, usually denoted $\otimes$. Given an $m \times n$-matrix A and a $p \times q$-matrix B, the Kronecker product $A \otimes B$ is of dimensions $pm \times qn$ [40], and defined as:

$$A \otimes B = \begin{bmatrix} a_{00}B & \dots & a_{0n}B \\ \vdots & \ddots & \vdots \\ a_{m0}B & \dots & a_{mn}B \end{bmatrix}$$

For example, the Kronecker product of the matrices $\left(\begin{smallmatrix} 1 & 2 \\ 3 & 4 \end{smallmatrix}\right)$ and $\left(\begin{smallmatrix} 2 & 2 \\ 9 & 8 \end{smallmatrix}\right)$ is shown below:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \otimes \begin{bmatrix} 2 & 2 \\ 9 & 8 \end{bmatrix} = \begin{bmatrix} 1\begin{bmatrix} 2 & 2 \\ 9 & 8 \end{bmatrix} & 2\begin{bmatrix} 2 & 2 \\ 9 & 8 \end{bmatrix} \\ 3\begin{bmatrix} 2 & 2 \\ 9 & 8 \end{bmatrix} & 4\begin{bmatrix} 2 & 2 \\ 9 & 8 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} 2 & 2 & 4 & 4 \\ 9 & 8 & 18 & 16 \\ 6 & 6 & 8 & 8 \\ 27 & 24 & 36 & 32 \end{bmatrix}$$