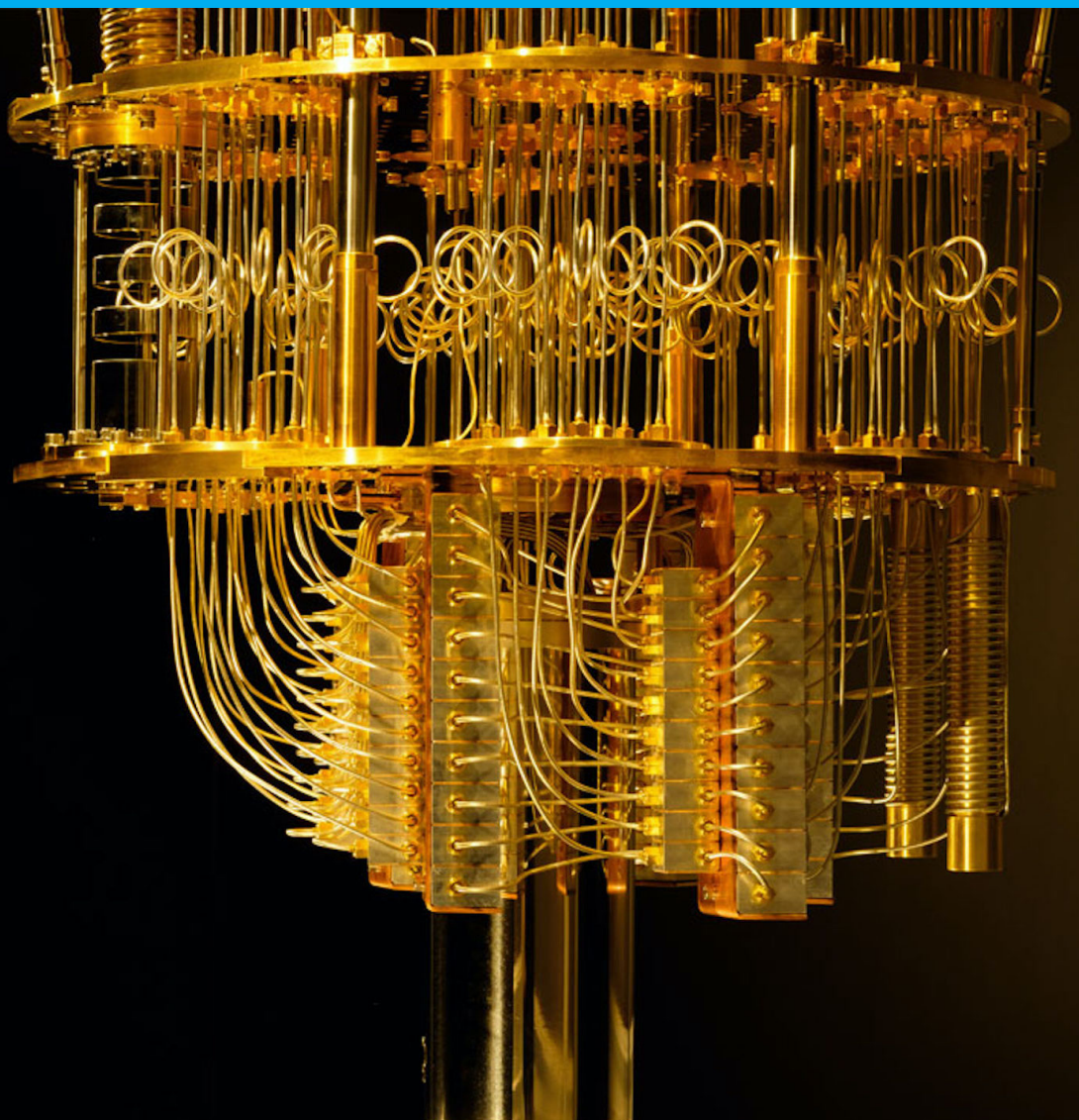


QuantumSim

A memory efficient
quantum computing
simulator

R. Budhrani



QuantumSim

A memory efficient quantum computing simulator

by

R. Budhrani

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Friday October 16, 2020 at 11:00 AM.

Student number: 4861248
Project duration: February 1, 2020 – present
Thesis committee: Prof. dr. K.L.M Bertels, QCA, Q&CE, TU Delft, supervisor
Dr. ir. Z. Al-Ars, ABS, Q&CE, TU Delft
Ir. Aritra. Sarkar, QCA, Q&CE, TU Delft

This thesis is confidential and cannot be made public until December 31, 2020.

An electronic version of this thesis is available at <http://repository.tudelft.nl/>.

Preface

2 years ago I had no idea what quantum computing was. In my final year of undergraduate studies, I started reading up on quantum computing and its potential applications. I knew that I had to study this discipline at a deeper level and decided to pursue a Masters with a specialisation in quantum computing. This thesis has been a culmination of my 2 years at TU Delft. I spent the first year taking as many courses related to quantum computing as I could, so I could find a research area to explore. In my second year I found the Quantum Computing Architectures research group led by **Professor Koen Bertels**. Their goal to speed up the development of quantum algorithms aligned with what I could offer.

Along with Koen, I was also lucky enough to received the help, guidance and feedback of the members of my thesis committee-**Professor Zaid Al-Ars** and **Aritra Sarkar**. I took Zaid's advanced computing systems course and learnt a lot from it. This course gave me a strong foundation in the study of computing systems and was very important in helping me design the quantum simulator that is explained in this thesis and measuring its performance. Aritra had gone through the same life of a Masters student a couple years before I did. His help with both the technical side of quantum computing and the administrative side of the thesis was invaluable. Regular meetings with the committee members and other members in the research group led to discussions that helped me pave my thesis.

Of course none of this would have been possible without my parents, who have provided me a great education and made sure that I have whatever I need to continue learning. Their continued support has helped me to focus on my studies and not worry about anything else. I'd also like to thank my friends for taking the time to listen to me explain my thesis over and over again. Their enthusiasm for my work always pushed me to do more. I spent a lot of time working on my thesis, as it was something I wanted to work on after graduating as well. Hope you enjoy reading this as much as I enjoyed writing it.

R. Budhrani
Delft, October 2020

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Scope of thesis	2
1.3	Organisation	2
2	Background	3
2.1	Primer on quantum computing	3
2.1.1	Qubits	3
2.1.2	Quantum gates	4
2.2	Hardware acceleration	5
2.2.1	Accelerator model	5
2.2.2	Quantum accelerator stack	5
2.2.3	OpenQL	6
2.2.4	cQASM: Common QASM	8
2.3	Conclusion	9
3	Functional design I	11
3.1	Introduction	11
3.2	Grover's Algorithm	11
3.2.1	The Boolean oracle	12
3.2.2	The Phase Oracle	13
3.3	Digitising the logic	14
3.3.1	Memory	15
3.3.2	Gate operations	16
3.3.3	Class Structure	18
3.4	Implementation of Grover's algorithm	18
3.4.1	Results	19
3.5	Conclusion	19
4	Functional design II	21
4.1	Introduction	21
4.2	Digitising the logic	21
4.2.1	Memory	21
4.2.2	Gate operations	22
4.2.3	Class structure	26
4.3	Benchmarking	27
4.3.1	System specifications	27
4.3.2	Memory benchmarking	27
4.3.3	Gate time benchmarking	28
4.3.4	Qubit benchmarking	29
4.3.5	Circuit depth benchmarking	30
4.3.6	Grover's Algorithm	30
4.4	Conclusion	31
5	Optimisations to the Quantum Simulator	33
5.1	Introduction	33
5.2	Parity system	33
5.2.1	Initial method	33
5.2.2	Parity method	34

5.3	Parallelisation	35
5.3.1	One-to-one mapping	36
5.3.2	One-to-two mapping	36
5.3.3	Results	37
5.4	Conclusion and future work	37
A	Background information	39
A.1	Commonly used gates	39
B	OpenQL	43
B.1	OpenQL options	43
	Bibliography	45

1

Introduction

If you think you understand quantum mechanics, you don't understand quantum mechanics.

Richard P. Feynman

As computing needs continue to increase, harnessing the power of quantum mechanics is pivotal in achieving what the industry refers to as "quantum supremacy".

1.1. Motivation

Classical computing has been evolving, to help solve harder problems. Following Moore's Law the miniaturisation of transistors has helped improve performance. However, this has led to a "Power Wall". The clock frequency of processors have not been making the leaps predicted by Moore's law. This is simply because the power dissipation becomes too high after a certain frequency. This, along with a few other factors, led the industry to move to multi-core processors and continue to homogeneous multi-core systems, multiple cores that are identical to each other, and heterogeneous, multiple cores that are not identical to each other. To further improve performance of computing systems, hardware accelerators were introduced. The one we are all familiar with is the GPU (graphics processing unit). As the name suggests it is an accelerator to process graphics. It is used extensively for image processing and is much faster at doing this than a CPU. There are many more types of hardware accelerators that offers high speedups for certain applications. Quantum accelerators are one such example.

There are a certain class of problems that cannot be solved, or rather will take too long to solve on a classical computer, such that it is practically infeasible. One such problem is prime-factorisation problem. In 1994 Peter Shor came up with a prime-factorisation algorithm that could be completed in polynomial time on a quantum computer [7]. For a n -bit number the algorithm would take $\mathcal{O}(n^3)$ time. The fastest classical algorithm is the general number field sieve (GNFS). Its complexity is given by

$$\exp\left(\left(\sqrt[3]{\frac{64}{9}} + o(1)\right)(\log n)^{\frac{1}{3}}(\log \log n)^{\frac{2}{3}}\right) \quad (1.1)$$

The widely used RSA cryptosystem and several other cryptosystem depend on the hardness of the prime-factorisation problem. With Shor's algorithm this is no longer the case.

Another application of quantum computing is the ability to simulation quantum systems itself. These systems cannot be simulated efficiently on a classical computer. Being able to simulate quantum systems are extremely beneficial to fields like drug development. Approaching the limits of classical computing means that new ways of computing are needed. Quantum computing is one such paradigm. By using quantum mechanics, a new approach to computing that offers significant advantages can be researched and created.

1.2. Scope of thesis

This thesis will focus on digital simulation of quantum systems. The main bottleneck of simulating quantum mechanics digitally is memory. The storage requirement for the state of large qubit systems increases exponentially. This thesis will aim to optimise quantum computations by minimising memory allocation and other optimisations. A digital simulator encompassing these optimisations will be designed in C++. Furthermore experiments will be carried out on this simulator to gauge how well certain optimisations perform. These experiments will check the memory requirements and the execution time of the gates. The factors affecting the execution time, will also be investigated. These factors are the number of qubits and the total number of gates in the circuit. The experiments carried out in this thesis, will also serve as a design space exploration for a potential micro-architecture.

1.3. Organisation

The second chapter of this thesis will provide an overview of quantum computing and the fundamental knowledge needed to understand the rest of the thesis. This is crucial as it will prepare the reader with a general understanding of quantum systems, without delving into the physics. It is required to understand the final simulator that will be developed in C++.

The third chapter will then introduce the full stack that is used in the research group that I am working with. This is the stack that the simulator is to be used with. It will provide an understanding of how an algorithm flows from a high-level language down to the hardware.

The fourth chapter will present the first implementation of the simulator, using the most commonly used method to simulated quantum systems digitally. This chapter will highlight the need for a change in the general matrix-based implementation. To make this chapter easier to follow, Grover's algorithm will be used, to explain what the simulator does step-by-step.

The final and fifth chapter will finally introduce the new simulator which used a non-matrix based approach to simulate a quantum system. It will also include several experiments to benchmark this simulator and to highlight why it is a superior choice.

2

Background

You can't build a great building on a weak foundation.

Gordon B. Hinckley

This chapter will provide a background of quantum gates and qubits. It will also provide an overview of the current stack used in the Quantum Computer Architectures Lab at TU Delft.

2.1. Primer on quantum computing

2.1.1. Qubits

In classical computing a bit is used as a means of storing information. The quantum analog of this is a *quantum bit* or *qubit*. A classical bit can only have one value, either 0 or 1. A qubit (or quantum bit) is two-state quantum-mechanical system. An example of a two-state system is the polarisation of a single photon. The 2 states in this system are the horizontal polarisation and the vertical polarisation. A qubit is then given by

$$|\psi\rangle = \alpha |H\rangle + \beta |V\rangle, \quad (2.1)$$

where

$$\alpha, \beta \in \mathbb{C} \quad \text{and} \quad |\alpha|^2 + |\beta|^2 = 1 \quad (2.2)$$

This Dirac or bra-ket notation is used for qubits. A qubit is in a superposition of 2 states. This means that it is an addition of 2 states. It is normally easier to replace the 2 states with $|0\rangle$ and $|1\rangle$. These *kets* are vector representations of the state. Usually the 2 levels are represented as a ground level $|0\rangle$ and an excited level $|1\rangle$. The vector representation of the 2 levels are

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}. \quad (2.3)$$

This is also known as the qubit state vector. In the case of the polarisation of the single photon, the horizontally polarised level $|H\rangle$ can be denoted by $|0\rangle$ and $|V\rangle$ can be denoted by $|1\rangle$. This results in the more commonly seen notation for a qubit,

$$|\psi\rangle = \alpha |0\rangle + \beta |1\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}. \quad (2.4)$$

When a measurement is performed on this state, it collapses to one of the basis states $|0\rangle$ or $|1\rangle$ with probabilities $|\alpha|^2$ and $|\beta|^2$ respectively. From this one measurement the original state $|\psi\rangle$ cannot be reconstructed. Multiple qubits are represented as a tensor product of the individual qubits. For example given 2 qubits $|\psi\rangle_1$ and $|\psi\rangle_2$,

$$|\psi\rangle_1 = \alpha_1 |0\rangle + \beta_1 |1\rangle \quad |\psi\rangle_2 = \alpha_2 |0\rangle + \beta_2 |1\rangle \quad (2.5)$$

the 2 qubit state can be written as a tensor product of the individual qubits. This is shown below,

$$|\psi\rangle_1 \otimes |\psi\rangle_2 = \begin{bmatrix} \alpha_1 \\ \beta_1 \end{bmatrix} \otimes \begin{bmatrix} \alpha_2 \\ \beta_2 \end{bmatrix} = \begin{bmatrix} \alpha_1 \begin{bmatrix} \alpha_2 \\ \beta_2 \end{bmatrix} \\ \beta_1 \begin{bmatrix} \alpha_2 \\ \beta_2 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \alpha_1 \alpha_2 \\ \alpha_1 \beta_2 \\ \beta_1 \alpha_2 \\ \beta_1 \beta_2 \end{bmatrix} \quad (2.6)$$

2.1.2. Quantum gates

In classical computing gates are used to operate on bits. For example the NOT gate flips the bit. In quantum computing the gates are represented as matrices, since the qubit is represented as a vector. An example of a quantum gate is the X gate, which is the quantum analog of the classical NOT gate. The effect of the X gate on the 2 basis states in the computational basis are

$$X|0\rangle = |1\rangle \quad X|1\rangle = |0\rangle. \quad (2.7)$$

Mathematically a quantum gate is represented by a matrix. It transforms the qubit state vector to another vector, which is the output of the gate. The matrix dimensions of the quantum gate follow the qubit state vector dimensions. For an n qubit system, i.e. 2^n states (entries in the qubit state vector) the quantum gate is represent by a matrix of size 2^n by 2^n . Therefore the more qubits there are the larger the size of the matrix that represent the operation on the qubits. The Pauli X gate is given by the matrix

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}. \quad (2.8)$$

Another example is the Hadamard gate, which is represented by the matrix,

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}. \quad (2.9)$$

The effect of the H gate on the 2 basis states in the computational basis are

$$H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \quad H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle). \quad (2.10)$$

A simple quantum circuit is shown in figure 2.1. It operates on a 2 qubit system. The first gate is the

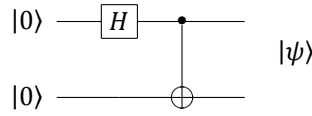


Figure 2.1: Quantum circuit to generate an EPR pair

Hadamard gate on the top qubit, followed by a CNOT gate, with the top qubit as the control and the bottom qubit as the target. The CNOT gate is a controlled Pauli X gate, which means that if the top qubit is $|1\rangle$ then the bottom qubit is flipped. The first set of gates that are applied are the Hadamard gate on the top qubit and the Identity gate on the bottom qubit. When no gate is applied, it is equivalent to applying the Identity gate on the qubit state vector. The resultant gate is given by the Kronecker product of the Hadamard gate, H , and the Identity gate, I ,

$$H \otimes I = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}. \quad (2.11)$$

Given two 2 by 2 matrices g_1 and g_2 ,

$$g_1 = \begin{bmatrix} a_1 & b_1 \\ c_1 & d_1 \end{bmatrix} \quad g_2 = \begin{bmatrix} a_2 & b_2 \\ c_2 & d_2 \end{bmatrix} \quad (2.12)$$

the Kronecker product, u , of these matrices is

$$u = \begin{bmatrix} a_1 \begin{bmatrix} a_2 & b_2 \\ c_2 & d_2 \end{bmatrix} & b_1 \begin{bmatrix} a_2 & b_2 \\ c_2 & d_2 \end{bmatrix} \\ c_1 \begin{bmatrix} a_2 & b_2 \\ c_2 & d_2 \end{bmatrix} & d_1 \begin{bmatrix} a_2 & b_2 \\ c_2 & d_2 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} a_1 a_2 & a_1 b_2 & b_1 a_2 & b_1 b_2 \\ a_1 c_2 & a_1 d_2 & b_1 c_2 & b_1 d_2 \\ c_1 a_2 & c_1 b_2 & d_1 a_2 & d_1 b_2 \\ c_1 c_2 & c_1 d_2 & d_1 c_2 & d_1 d_2 \end{bmatrix} \quad (2.13)$$

Therefore the Kronecker product of H and I is

$$H \otimes I = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}. \quad (2.14)$$

This serves as an example of how the size of the matrix increases with the number of qubits. For a 2 qubit system, there are 2^n states. This means that the qubit state vector has 2^n elements and the quantum gate matrix is a square matrix with 2^{2n} elements.

A key difference between gate in quantum computing and gates in classical computing, is that quantum gates are reversible. This means that the operators that correspond to the gates are unitary. A unitary matrix, U , is a matrix that obeys the following

$$U^\dagger U = U U^\dagger = \mathbb{I}. \quad (2.15)$$

The product of the unitary matrix, U , and its Hermitian conjugate, U^\dagger , gives the identity matrix. This product is equivalent to doing an operation on a qubit and then undoing it, thus showing the reversible nature of quantum gates. Commonly used quantum gates are shown in appendix A.1.

2.2. Hardware acceleration

A quantum computer will not be able to replace many functions of a classical computer. Instead it will be used as an accelerator, similar to GPUs and FPGAs. In this subsection the accelerator model will be introduced along with the current full stack implementation of the Quantum Accelerator developed by the Quantum Computer Architectures Lab at TU Delft. The idea of having a full stack for the quantum application is similar to what would be done if an application were to be designed specifically for another accelerator like a GPU. It would be written in CUDA and kernels would be used to execute the application on the GPU.

2.2.1. Accelerator model

An accelerator is a special piece of hardware that accelerates a specific task. For example, a Digital Signal Processor (DSP) is a specialised hardware optimised for digital signal processing. Accelerators as co-processors are shown in figure 2.2. A generic accelerator is viewed as a separate I/O device connected to the host CPU, whereas a co-processor is an extension of the host CPU's architecture. QPU is the *quantum processing unit*.

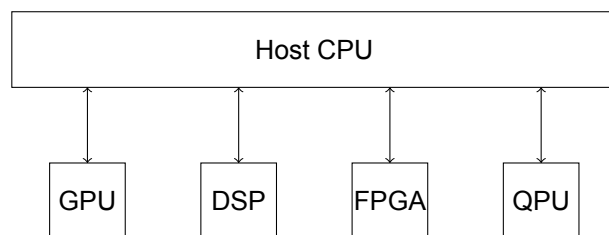


Figure 2.2: Accelerator as a co-processor

Quantum computing will not replace classical computing completely. It is merely a type of accelerator that speeds up certain types of tasks. The CPU will allocate a certain task to it, which the QPU will execute and send the results back to the host CPU.

2.2.2. Quantum accelerator stack

The Quantum & Computer Engineering department at TU Delft came up with an accelerator model shown in figure 2.3. The scope of this thesis will be limited to the stack with the perfect qubits, figure 2.3b. Perfect qubits are defined as qubits that experience no decoherence and no noise. The purpose of this stack is to further the field of quantum algorithms. A simulator that simulates perfect qubits, can be used as a testing ground for quantum algorithm development. The quantum simulator, QX simulator,

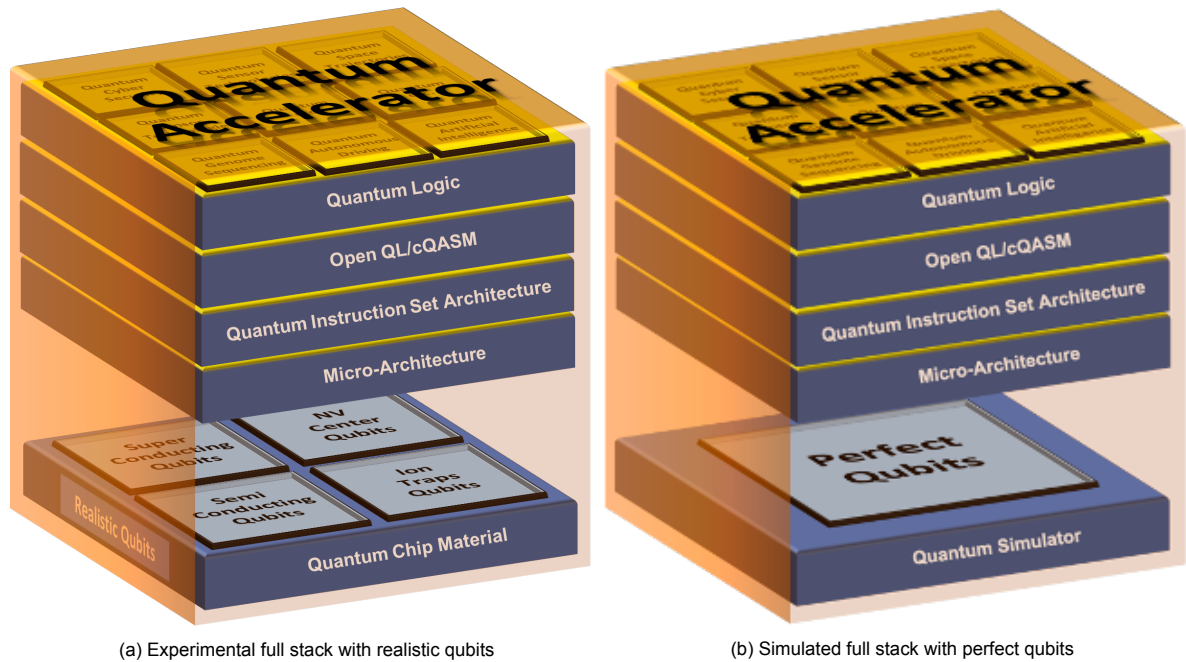


Figure 2.3: Quantum accelerator stacks

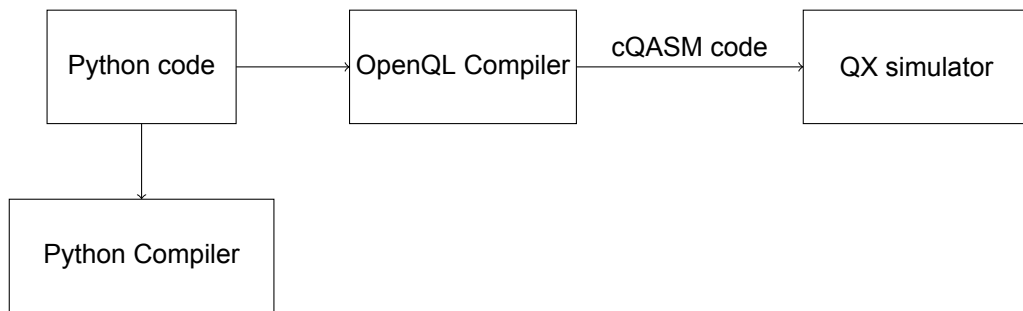


Figure 2.4: Python to cQASM to QX simulator

is used as a co-processor as explained in section 2.2.1. The flow of the program from high-level to low-level is shown in figure 2.4. This thesis will propose another simulator that utilises less memory and the entire "stack" is written in C++. Hereinafter, this simulator will be known as QuantumSim. In the following chapters this new simulator will be introduced. The following subsections will explain the current stack.

2.2.3. OpenQL

OpenQL is a framework that is used to program the quantum logic in the stack using C++ or Python. It provides a compiler that compiles and optimises the quantum code to produce the low-level cQASM (Quantum Assembly Language) instructions which are sent to the quantum simulator (QX Simulator). Figure 2.4 shows how the Python code written for a particular quantum circuit is compiled by both the OpenQL compiler and python compiler. The part compiled by the OpenQL compiler is translated to cQASM code, which can then be simulated on the QX simulator. A simple example can be used to show the syntax of OpenQL. The Python code for circuit in figure 2.1 is shown in Listing 2.1 and it generates the EPR pair,

$$|\psi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle). \quad (2.16)$$

```

1 #the OS module has functions that help with interfacing with the OS that python is
  running on
  
```

```

2 import os
3 #the openql package obviously
4 from openql import openql as ql
5
6 #get the current directory that you are working in
7 curdir = os.path.dirname(__file__)
8 #create a folder to store the QASM instructions in the current directory
9 output_dir = os.path.join(curdir, 'qasm')
10 #let OpenQL know where the output directory to store the QASM instructions is
11 ql.set_option('output_dir', output_dir)
12 #let OpenQL know that you would like to store the QASM files
13 ql.set_option('write_qasm_files', 'yes')
14 #add the configuration file to your program
15 config_fn = os.path.join(curdir, 'config_qx.json')
16
17 #create the platform object using the json configuration file
18 platform = ql.Platform('platform_none', config_fn)
19 #define the number of qubits that will be used
20 num_qubits = 2
21 #create a Program object that can contain one or more kernels
22 p = ql.Program('genEPR', platform, num_qubits)
23
24 #create the kernel object
25 k = ql.Kernel('min_kernel', platform, num_qubits)
26 #initialise both qubits to the state 0
27 k.prepz(0)
28 k.prepz(1)
29 #apply a Hadamard gate to qubit 0
30 k.gate('h', [0])
31 #apply a CNOT gate to qubit 0 (control) and 1 (target)
32 k.gate('cnot', [0, 1])
33
34 #add the kernel to the program object
35 p.add_kernel(k)
36 #compile the program
37 p.compile()
38 #optionally it is possible to extract the QASM instructions from the program
   object and print it to console
39 qasm = p.qasm()
40 print(qasm)

```

Listing 2.1: Python code for an EPR pair generation circuit using OpenQL

In line 4 the OpenQL framework is imported as `ql`. Then in lines 11 and 13, the relevant options are set. The `set_option` method is defined as

```
set_option(option_name, option_value)
```

The method requires 2 parameters, both of which are strings. The first parameter is the option name and the second parameter is the value that the option takes. The options for Listings 2.1 are given in table 2.1. The full set of options is shown in table B.1 in appendix B.1.

Option	Value	Description
<code>output_dir</code>	<PATH>	The path to where the QASM code will be stored
<code>write_qasm_files</code>	yes	Boolean value to determine if QASM code is to be written

Table 2.1: OpenQL options for Listing 2.1

In line 15 the configuration file is defined. This file specifies the platform that the QASM code is created for and the hardware configuration in a `json` file. The constructor for this class takes 2 parameters as defined below

```
ql.Platform(platform_name, <path_to_json_config_file>),
```

where `ql` is the alias for `openql`. The `platform_name` is defined by the user. The `json` file is used for the compiler passes. One of the most important sections in the configuration file, is the set of instructions that are allowed. An example of one such instruction is the Hadamard gate. This gate was used in line 30 in Listing 2.1. Inside the `instructions` section of the `json` file, the details of this gate operation are defined. A snippet of the configuration file is shown in Listing 2.2.

```

1  {
2    "eqasm_compiler" : "qx",
3
4    "hardware_settings":
5    {
6      ...
7    },
8
9    "topology":
10   {
11     ...
12   },
13
14   "resources":
15   {
16     ...
17   },
18
19   "instructions":
20   {
21     "h" : {
22       "duration": 40,
23       "matrix" : [ [0.7071068,0.0], [-0.7071068,0.0],
24                   [0.7071068,0.0], [ 0.7071068,0.0] ],
25       "disable_optimization": true
26     },
27     ...
28   },
29
30   "gate_decomposition":
31   {
32     ...
33   }
34 }
```

Listing 2.2: Instruction definition for the Hadamard gate in json

The instruction for the Hadamard gate is the string `h`. For the case of compiling for the QX simulator there is only one option that is needed for the instruction. This is the unitary matrix for the gate.

2.2.4. cQASM: Common QASM

There are 2 variations of QASM-cQASM and eQASM. cQASM [6] stands for *common QASM*, which abstracts away qubit technology details. eQASM stands for executable QASM and is the instruction set that executes on hardware. eQASM allows for the simulation of particular qubit technologies, such as superconducting qubits, spin qubits etc. It allows the specification of gate time, measurement time, cycle time and other parameters that are important for a physical chip. As this thesis focuses on simulation of perfect qubits, cQASM will be the chosen instruction set. Listings 2.3 shows the cQASM code for the circuit in figure 2.1.

```
1 version 1.0
```



```
2 qubits 2
3
4 .hadamardGate
5   h q[0]
6
7 .cnotGate
8   cnot q[0],q[1]
9   display
```

Listing 2.3: cQASM code to generate an EPR pair

The first 2 lines define the cQASM version which at the time of writing is 1.0 and the number of qubits, which is 2 for the circuit. The next set of instructions are `kernels`. Each kernel starts with a period. The first kernel, for example, is `.hadamardGate`. This kernel contains the first set of gates and for this circuit it is a hadamard gate on the first qubit. The first qubit is addressed as you would address an item in a list/array. The next kernel, contains the cnot gate, with `q[0]` as the control and `q[1]` as the target. The last command `display` is present to show the results on the console, when running this circuit on the QX simulator.

2.3. Conclusion

In this chapter, the basics concepts of quantum computing have been introduced. Qubits, which are the quantum analog of classical bits, have been explained. Quantum gates operating on qubits and performing quantum operations have also been explained. The essence of quantum operations stems down to linear algebra, as mathematically, the qubits and gates are represented as vector and matrices. This chapter also introduced the quantum stack that is being used by the QCA (Quantum Architectures Lab) at TU Delft. OpenQL and cQASM are explained. To reiterate, the focus of the QCA is perfect qubits, as the current state of physical qubits is not mature enough to execute an algorithm requiring many qubits. The largest one, in terms of physical qubits, at the time of writing this thesis is Bristlecone [5] by Google, which has 72 qubits. QCA wants to move to the application layer of the stack and focus on quantum algorithms, in various fields, such as genomics, finance and security. In order to achieve this, a quantum simulator to execute and test these algorithms is required. The proposed quantum simulator will be proposed in the following chapters.

3

Functional design I

3.1. Introduction

As explained in the previous chapter, the need to have a simulator to test quantum algorithms is required. To reiterate, perfect qubits will be used for the simulator, as physical qubits are hard to scale and suffer from noise and decoherence. Perfect qubits are not affected by noise or decoherence. It can be argued as to why one would use perfect qubits at all. The end goal is to use real quantum computers to run quantum algorithms, but by having a simulator the study and execution of quantum algorithms for many qubits is possible right now, while the hardware is still maturing. In this chapter, the first implementation of a digital simulator in C++ will be realised. Grover's algorithm on 2 qubits will serve as a small example to explain the design. The digitising of quantum logic will be investigated, by exploring how classical memory can be used to store qubit systems. Most importantly this chapter will serve to highlight the large memory requirement to simulate quantum systems, using matrix vector multiplications. For the remainder of this thesis, the ISO 80000 standard will be used for the prefixes of multiples of bytes. Table 3.1 shows the prefixes that will be used throughout this thesis and their corresponding conversion to bytes. The rest of this chapter is organised as follows-first Grover's algorithm

Unit	KiB	MiB	GiB	TiB	PiB
Bytes	2^{10}	2^{20}	2^{30}	2^{40}	2^{50}

Table 3.1: Prefixes for multiples of Bytes

is introduced, as it is the algorithm that will be used to benchmark the simulator. Next, a description of how the simulator works is presented. The conversion of quantum gates and qubits into classical hardware is explained. Finally the memory required and execution time of the simulator is measured and presented.

3.2. Grover's Algorithm

Grover's algorithm [4] is an example of a quantum algorithm that offers a quadratic speedup over its classical counterpart. There are 2 ways to implement the oracle that tags the solution(s) [3]. One uses a Boolean oracle to tag the solution(s) and the other uses a phase oracle. The Boolean oracle uses an ancilla qubit and implementing this oracle classically would flip this ancilla qubit if the input to the circuit was the solution. This is exactly what happens with the quantum version as well. The ancilla qubit is flipped for the solution state. In quantum computing, it is possible to initialise the state to the system with a superposition of all possible states and therefore the ancilla qubit is no longer required. The point of using the Boolean oracle is to compare it to the classical analog of Grover's search algorithm, which uses a state marking scheme for performing a classical search. The state marking scheme marks the state with the solution with a phase. The Boolean and Phase oracles are equivalent, and the Boolean oracle helps show the superiority of the quantum version of classical algorithms.

3.2.1. The Boolean oracle

An example of Grover's algorithm on 3 qubits (2 data qubits, defining the search space and 1 ancilla qubit) using the Boolean oracle is shown in figure 3.1. The 2 qubits are labelled q_0 and q_1 and the

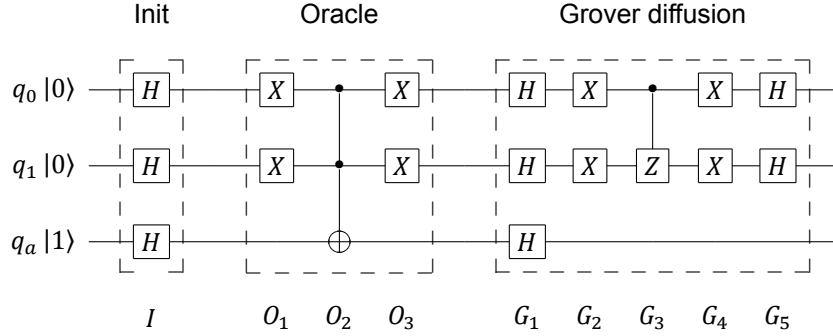


Figure 3.1: Grover's algorithm using a Boolean oracle for a 2 qubit search space with solution $|00\rangle$

ancilla qubit is q_a . The oracle is solution specific. This particular oracle tags the state $|00\rangle$ as the solution. After the *Init* stage the quantum state of the circuit is given by the state $|\psi\rangle_I$ which contains 3 qubits,

$$|\psi\rangle_I = |++-\rangle. \quad (3.1)$$

The subscript of I determines which stage the qubit $|\psi\rangle$ is at. I stands for *Init*, O for *Oracle* and G for *Grover*. The number after the subscript value determines which step of gates the state is at. After the first set of X gates in the *Oracle* the state is

$$|\psi\rangle_{O_1} = |++-\rangle = \frac{1}{2\sqrt{2}}(|000\rangle - |001\rangle + |010\rangle - |011\rangle + |100\rangle - |101\rangle + |110\rangle - |111\rangle). \quad (3.2)$$

This is because the X gate does nothing to the state $|+\rangle$ or $|-\rangle$,

$$X|+\rangle = \frac{1}{\sqrt{2}}(X|0\rangle + X|1\rangle) = \frac{1}{\sqrt{2}}(|1\rangle + |0\rangle) = |+\rangle \quad (3.3)$$

$$X|-\rangle = \frac{1}{\sqrt{2}}(X|0\rangle - X|1\rangle) = \frac{1}{\sqrt{2}}(|1\rangle - |0\rangle) = -\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) = -|-\rangle. \quad (3.4)$$

The negative sign before $|-\rangle$ is a global phase and has no effect on the system. The next gate is the Toffoli gate which changes the state to

$$|\psi\rangle_{O_2} = \frac{1}{2\sqrt{2}}(|000\rangle - |001\rangle + |010\rangle - |011\rangle + |100\rangle - |101\rangle + |111\rangle - |110\rangle). \quad (3.5)$$

Reordering the previous equation gives,

$$\begin{aligned} |\psi\rangle_{O_2} &= \frac{1}{2\sqrt{2}}(|000\rangle - |001\rangle + |010\rangle - |011\rangle + |100\rangle - |101\rangle - (-|111\rangle + |110\rangle)) \\ |\psi\rangle_{O_2} &= \frac{1}{2}(|00\rangle \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) + |01\rangle \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) + |10\rangle \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) - (-|11\rangle \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle))) \\ |\psi\rangle_{O_2} &= \frac{1}{2}(|00-\rangle + |01-\rangle + |10-\rangle - |11-\rangle) \end{aligned} \quad (3.6)$$

The last qubit can be separated to give

$$|\psi\rangle_{O_2} = \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle - |11\rangle)|-\rangle. \quad (3.7)$$

The last set of gates in the *Oracle* are X gates on all the qubits except the ancilla qubit, which results in the state,

$$|\psi\rangle_{O_3} = \frac{1}{2}(|11\rangle + |10\rangle + |01\rangle - |00\rangle)|-\rangle. \quad (3.8)$$

After the *Oracle* the solution $|00\rangle$ has been tagged with a phase, indicated by the negative amplitude. Since the solution is now tagged with a negative amplitude, an inversion about the mean can be performed. Recall that the Hadamard gate has the following effect on the basis states of the standard basis,

$$H|0\rangle = |+\rangle \quad H|1\rangle = |-\rangle. \quad (3.9)$$

After the first set of Hadamard gates, i.e. G_1 , in *Grover diffusion* stage the quantum state is transformed from $|\psi\rangle_{O3}$ to

$$|\psi\rangle_{G1} = \frac{1}{2}(|--\rangle + |-\rangle + |+-\rangle - |++\rangle) |1\rangle. \quad (3.10)$$

The next set of gates, i.e. G_2 are the X gates on all qubits except the ancilla qubit. The X gates change the quantum state to

$$|\psi\rangle_{G2} = \frac{1}{2}(|--\rangle - |-\rangle - |+-\rangle - |++\rangle) |1\rangle. \quad (3.11)$$

After expanding out the $|+\rangle$ and $|-\rangle$ states in the standard basis, the equation above is equal to the following in the standard basis

$$|\psi\rangle_{G2} = -\frac{1}{2}(|01\rangle - |11\rangle + |00\rangle + |10\rangle) |1\rangle. \quad (3.12)$$

After the controlled Z gate, i.e. G_3 , the state of the quantum system is

$$|\psi\rangle_{G3} = -\frac{1}{2}(|01\rangle + |11\rangle + |00\rangle + |10\rangle) |1\rangle. \quad (3.13)$$

The X gates in G_4 transform the state to

$$|\psi\rangle_{G4} = -\frac{1}{2}(|10\rangle + |00\rangle + |11\rangle + |01\rangle) |1\rangle. \quad (3.14)$$

In the Hadamard basis this can be written as

$$|\psi\rangle_{G4} = -\frac{1}{2}((|1\rangle + |0\rangle)|0\rangle + (|1\rangle + |0\rangle)|1\rangle) |1\rangle = -|+\rangle|+\rangle |1\rangle. \quad (3.15)$$

After the final set of Hadamard gates the state is

$$|\psi\rangle_{G5} = -|001\rangle. \quad (3.16)$$

The solution is the first 2 qubits as the third one is the ancilla qubit,

$$|\psi\rangle_{G5} = -|00\rangle_{q_0 q_1} \otimes |1\rangle_{q_a}. \quad (3.17)$$

This is the solution that the oracle tagged. Therefore the solution has been found, along with a global phase (the negative sign before the state $|00\rangle$) that can be ignored.

3.2.2. The Phase Oracle

The Boolean oracle requires more resources and an extra qubit. The Phase oracle does not. The same 2 qubit example is shown figure 3.2.

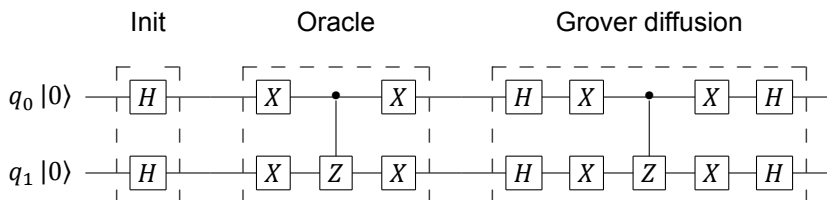


Figure 3.2: Grover's algorithm using a Phase oracle for a 2 qubit search space with solution $|00\rangle$

The state after the *Init* stage is

$$|\psi\rangle_I = |++\rangle. \quad (3.18)$$

The first set of X gates in the *Oracle* does nothing to the state

$$|\psi\rangle_{o1} = |++\rangle = \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle). \quad (3.19)$$

The controlled Z changes the quantum state to

$$|\psi\rangle_{o2} = \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle - |11\rangle). \quad (3.20)$$

The X gates then flips all the qubits,

$$|\psi\rangle_{o3} = \frac{1}{2}(|11\rangle + |10\rangle + |01\rangle - |00\rangle). \quad (3.21)$$

The solution is now tagged with a phase. The rest of the circuit is completely identical to the one in figure 3.1, as the ancilla qubit is separable.

Grover's algorithm is probabilistic, as is all of quantum computing. Therefore getting the correct solution is not always possible with perfect certainty. The success probability can be improved, by performing part of the circuit multiple times. The oracle and the Grover diffusion operation are performed $\lfloor \frac{\pi\sqrt{N}}{4m} \rfloor$ times [1], where $N = 2^n$ is the number of states and m is the number of solutions. In a classical simulator the entire qubit state vector is available and therefore measurement is not destructive. Pending further investigation it is possible that for a classical simulator, the number of repetitions can be reduced, as the entire state vector is always available.

3.3. Digitising the logic

Consider the circuit in figure 3.2 again. The entire circuit can be decomposed as matrices and vectors. The gates are unitary matrices and the qubits are normalised vectors. The first set of gates are the Hadamard gates and they can be written as

$$H \otimes H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \otimes \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix}. \quad (3.22)$$

The input state of the qubits is

$$|00\rangle = |0\rangle \otimes |0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}. \quad (3.23)$$

Applying this state to the Hadamard gates is done using linear algebra

$$H \otimes H |00\rangle = \frac{1}{2} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}. \quad (3.24)$$

In *ket* notation this is

$$\frac{1}{2} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle). \quad (3.25)$$

This is a superposition of all the possible states with 2 qubits, which is what is expected with the array of Hadamard gates in the beginning.

3.3.1. Memory

The previous example of Hadamard gates showed that the quantum circuit can be solved on a digital computer using matrices and vectors. In this subsection, the classical memory requirements will be explained. Gates in quantum mechanics can also contain a complex part like the T gate for example,

$$T = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{4}} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & \frac{1}{\sqrt{2}}(1 + i) \end{bmatrix}. \quad (3.26)$$

The 4th entry in the matrix has both a real and a complex part. For this reason, every element in a unitary matrix (i.e. a gate) and in the qubit vector is stored as a `2 tuple`. A `tuple` is a data structure that holds elements of different types. To store a complex number a `2 tuple` is sufficient. The first entry holds the real entry of the complex number and the second part holds the imaginary part. In C++ this is done by using a template class for complex numbers. A complex number is constructed using `std::complex<T>`. The constructor constructs a complex number, where `T` is the type and is one of the following-`float`, `double` and `long double`. The type determines the size of each complex number and is determined according to the precision requirement of the application. The size for each of these types is shown in table 3.2. The size of a complex number of type `double`, for example, is

Type	float	double	long double
Size (Bytes)	4	8	16

Table 3.2: Sizes of different types

therefore 16 bytes (8 bytes for the real part and 8 for the imaginary). Using this information the memory required for performing quantum gates on digital logic can be obtained. There are 2 data structures that need to be considered-the qubit state vector and the unitary gate matrix. The way elements are stored in the qubit state vector is shown in equation 3.27.

$$q = \begin{bmatrix} (r_0, i_0) \\ (r_1, i_1) \\ \cdot \\ \cdot \\ \cdot \\ (r_{N-1}, i_{N-1}) \end{bmatrix} \quad (3.27)$$

For an n qubit state there are a possible $N = 2^n$ states that can be represented, hence the number of `2 tuples` is equal to N . The total size of a qubit state vector is then given by

$$s_v = 2 \cdot s_T \cdot N = 2 \cdot s_T \cdot 2^n, \quad (3.28)$$

where s_T is the size of the type as listed in table 3.2 and the scalar factor of 2 is a consequence of the `2 tuple`. Each entry in the `tuple` requires s_T bytes of storage.

The way elements are stored in a unitary matrix, i.e. a quantum gate, is shown in equation 3.29.

$$U = \begin{bmatrix} (r_{0,0}, i_{0,0}) & (r_{0,1}, i_{0,1}) & \cdot & \cdot & \cdot & (r_{0,N-1}, i_{0,N-1}) \\ (r_{1,0}, i_{1,0}) & & & & \cdot & (r_{1,N-1}, i_{1,N-1}) \\ \cdot & & & & \cdot & \cdot \\ \cdot & & & & \cdot & \cdot \\ \cdot & & & & \cdot & \cdot \\ (r_{N-1,0}, i_{N-1,0}) & (r_{N-1,1}, i_{N-1,1}) & \cdot & \cdot & \cdot & (r_{N-1,N-1}, i_{N-1,N-1}) \end{bmatrix} \quad (3.29)$$

The size occupied by the unitary matrix is simply the size occupied by the qubit vector multiplied by $N - 1$,

$$s_m = 2 \cdot s_T \cdot N^2 = 2 \cdot s_T \cdot 2^{2n}. \quad (3.30)$$

The space requirement of both the state vector and the unitary matrix is exponential, assuming a constant s_T . For the qubit state vector it is of $\mathcal{O}(2^n)$ and for the unitary matrix it is of $\mathcal{O}(2^{2n})$. The memory requirement for the unitary matrix/quantum gate grows much quicker than that of the qubit state vector.

The state vectors representing qubits are stored as 1-dimensional arrays. The unitary matrices are 2-dimensional, however they are also stored as 1-dimensional arrays as shown in figure 3.3. A 1-dimensional array offers better memory locality and lower allocation and deallocation overhead. In a 2-dimensional array, every row is a separately allocated dynamic 1-dimensional array. This is what contributes to the higher allocation and deallocation overhead. This also means that the 2-dimensional array is discontinuous, thus the poor memory locality. Traversing a 2-dimensional array is therefore slower. Each entry $u_{i,j}$ in figure 3.3 is equal to $(r_{i,j}, i_{i,j})$ in equation 3.29. Each row contains N elements,

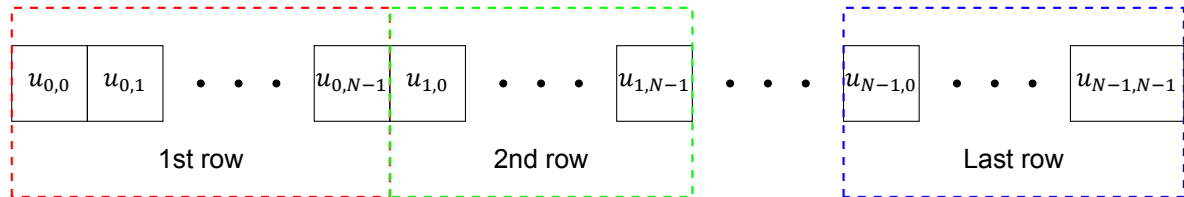


Figure 3.3: Unitary gate in memory

or N^2 tuples. The array is allocated in sequence, i.e. contiguous memory. By knowing the size of the array and the dimensions of the unitary matrix, U , each element of the matrix can be accessed using this 1D array. Understanding the structure of how a unitary matrix is stored is important as it is important in implementing the Kronecker product of 2 matrices.

3.3.2. Gate operations

There are 2 main gate operations in this implementation. The first one is computing the Kronecker product of the quantum gates that are to be applied in parallel. The Kronecker product is an operation on 2 matrices, that computes a larger matrix from 2 smaller ones. An example of this was shown in section 3.3. In quantum computing, Kronecker products and tensor products are analogous to each other, since quantum gates are linear transformations and thus the Kronecker product of 2 gates represents the tensor product of the 2 transformations. The second gate operation is applying this larger gate computed using the Kronecker product to the qubit state vector. The qubit state vector is a vector containing the amplitudes of all the quantum states. For both these operations, knowing how to index the 1D array stored in memory that represents the gate (which is a 2-dimensional array), is very important. The way to iterate over this array in the correct order is shown in Listing 3.1. It prints out the elements of the matrix as a 2-dimensional array.

```

1 for (int row=0; row<size; row++){
2     for (int column=0; column<size; column++)
3         std::cout<<matrix[row*size+column]<<" ";
4     std::cout<<"\n";
5 }

```

Listing 3.1: Indexing 2D array stored as 1D array

The variable `size` is the number of rows/columns of the matrix. As the gate is a square matrix, the number of rows and columns are equal. The number of elements in the matrix is therefore `size*size`. To access the correct element, by keeping the 2-dimensional nature of matrix intact, the matrix is indexed over 2 loops, even though it is simply one row. It is the offset `row*size` that provides the ability to access the next row.

The Kronecker product of 2 unitary gates is performed by iterating over both gates and computing the new entries in the larger unitary matrix. Snippet 3.2 shows how this is done. It is important to not that this snippet highlights the main part of the function that performs the Kronecker product, but is not the complete function. Variable initialisation, memory allocation and object-referencing is done to obtain all the necessary data.

```

1 for(int r1=0; r1<s1; r1++)

```



```

2     for(int c1=0; c1<s1; c1++)
3         for(int r2=0; r2<s2; r2++)
4             for(int c2=0; c2<s2; c2++)
5                 newGate[r1*sN*2 + c1*s2 + r2*sN + c2] = gate1[r1*s1+c1
                    ]*gate2[r2*s2+c2];

```

Listing 3.2: Kronecker product of 2 matrices

The variables in Listings 3.2 are:

- $r1, c1$: number of rows and columns in the first gate (both are equal to $s1$)
- $s1$: size of first gate (in any direction, since the gate is a square matrix)
- $r2, c2$: number of rows and columns in the second gate (both are equal to $s2$)
- $s2$: size of second gate (in any direction, since the gate is a square matrix)
- sN : size of new gate (in any direction, since the gate is a square matrix). This is simply the product of $s1$ and $s2$
- $gate1, gate2$: the arrays that hold all the values of the first and second gate respectively
 $newGate$: the array that contains the Kronecker product of the first and second gates

To understand the code and the calculation of the indices at which to store elements, an simple example can be considered. Two 2 by 2 matrices are given in equation 3.31.

$$g_1 = \begin{bmatrix} a_1 & b_1 \\ c_1 & d_1 \end{bmatrix} \quad g_2 = \begin{bmatrix} a_2 & b_2 \\ c_2 & d_2 \end{bmatrix} \quad (3.31)$$

The Kronecker product of these 2 matrices is given in equation 3.32.

$$u = \begin{bmatrix} a_1 \begin{bmatrix} a_2 & b_2 \\ c_2 & d_2 \end{bmatrix} & b_1 \begin{bmatrix} a_2 & b_2 \\ c_2 & d_2 \end{bmatrix} \\ c_1 \begin{bmatrix} a_2 & b_2 \\ c_2 & d_2 \end{bmatrix} & d_1 \begin{bmatrix} a_2 & b_2 \\ c_2 & d_2 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} a_1 a_2 & a_1 b_2 & b_1 a_2 & b_1 b_2 \\ a_1 c_2 & a_1 d_2 & b_1 c_2 & b_1 d_2 \\ c_1 a_2 & c_1 b_2 & d_1 a_2 & d_1 b_2 \\ c_1 c_2 & c_1 d_2 & d_1 c_2 & d_1 d_2 \end{bmatrix} \quad (3.32)$$

Building from figure 3.3, the larger matrix u is stored as shown in figure 3.4.

$a_1 a_2$	$a_1 b_2$	$b_1 a_2$	$b_1 b_2$	$a_1 c_2$	$a_1 d_2$	$b_1 c_2$	$b_1 d_2$	$c_1 a_2$	$c_1 b_2$	$d_1 a_2$	$d_1 b_2$	$c_1 c_2$	$c_1 d_2$	$d_1 c_2$	$d_1 d_2$
1st row				2nd row				3rd row				4th row			

Figure 3.4: Unitary gate, u , in memory

As shown in Listings 3.2 the index for each element of the new gate produced after performing the Kronecker product is given by

$$r1 \cdot sN \cdot 2 + c1 \cdot s2 + r2 \cdot sN + c2. \quad (3.33)$$

$c2$ is the index of the column of the second gate, which is always incremented, and never stays the same between 2 successive blocks as shown in figure 3.4. $r2$ is the index of the second matrix and is increased by a factor of the size of the new matrix, sN , because the terms in the second row of g_2 are always stored in a new row. The term $c1 \cdot s2$ adds an offset by $s2$ whenever $c1$ is incremented. This is because the next entry when switching columns in the first matrix is $s2$ elements away. The last term to look at applies an offset of $2 \cdot sN$ whenever the next row of the first matrix is used. This is because the entries multiplied by the next row are always $2 \cdot sN$ elements away from each other.

3.3.3. Class Structure

The essence of the implementation is the operations on the quantum/unitary gate. Everything else is an input and output to it. The whole structure is implemented as a class, called `Unitary`, which contains the relevant data—the input qubit state vector, the unitary matrix to be applied and the output qubit state vector. It also contains functions to manipulate this data and produce the output qubit state vector. The implementation is shown in figure 3.5. *in* is the input qubit state vector, *unitary* is the unitary matrix/gate that is to be applied and *out* is the output qubit state vector. The *unitary* block, computes

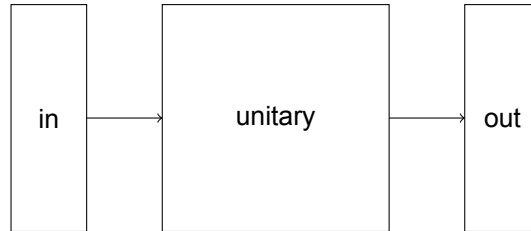


Figure 3.5: Class structure for `Unitary`

the kronecker product of the quantum gates that are to be applied.

This is where a relationship between this structure and the input cQASM instructions is formed. Each set of parallel gates is enclosed in a kernel in cQASM. This concept is explained in section 2.2.4. A kernel is the first input to the `Unitary` class, which first converts each gate to an object of the class `Unitary`. This way the member functions of `Unitary` can be used. Next it performs a Kronecker product on the gates given in the kernel to obtain the final *unitary*. This *unitary* takes as input the n qubit state and outputs the result of the unitary. This large *unitary* and the 2 qubit state vectors *in* and *out* are stored in one data structure defined by a class. The class has its own member functions for data manipulation. After the first kernel is executed, the *unitary* block is populated with the next set of gates. *out* becomes *in* and for the next gate.

3.4. Implementation of Grover's algorithm

The circuit in figure 3.2 is implemented in C++ to help design the micro-architecture. The 2 important memory blocks required are defined in section 3.3.1—one for the qubit state vector and one for the unitary. As an example the circuit in figure 3.2 can be considered again. In figure 3.6 the circuit has

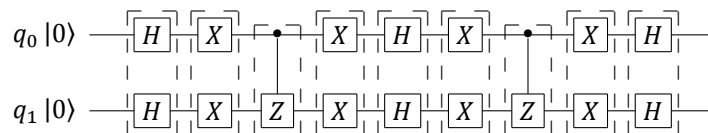


Figure 3.6: Grover's algorithm using a Phase oracle for a 2 qubit search space with solution $|00\rangle$

been split up into its gate operations. The first set of gates are two Hadamard gates. The effect of these gates was shown in section 3.3. The Hadamard gate is stored in memory as a 1D array, which is shown in figure 3.7. The 1st row of the unitary matrix representing the Hadamard gate is stored in the first 2 blocks (denoted by the dotted red line) and the 2nd row is stored in the next 2 blocks (denoted by the dotted green line). Each entry is a 2 tuple. The Hadamard gate has no imaginary component

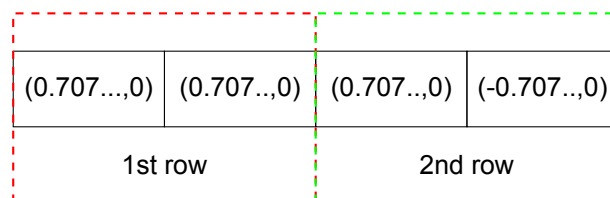


Figure 3.7: Hadamard gate in memory

and therefore the second entry in the tuple is 0. Each element is $2 \cdot s_T$ bytes long, where s_T is the

size of the data type being used. Common data types and their memory sizes are given in table 3.2. The number of decimal places that the coefficient $\frac{1}{\sqrt{2}}$ can have is determined by the type of data that is being used (i.e. `float`, `double` etc.). The first step to take when calculating the output of the 2 Hadamard gates, is to compute the Kronecker product of the gates. With an understanding of how the Hadamard gate is stored in memory, the process of indexing the two Hadamard gates and computing the Kronecker product and storing the elements in the right order can be done.

After the large *unitary* is calculated for the kernel, the input qubit state, *in*, is applied to it and the output qubit state, *out*, is obtained. This output now becomes the input to the next kernel, and the process is repeated for each kernel.

3.4.1. Results

A proof of concept simulator to evaluate the memory requirement of simulating quantum circuits using vector matrix multiplications is implemented. This simulator runs Grover's algorithm on 2 qubits. A 2 qubit state vector is a column vector with $2^2 = 4$ rows. To represent the complex amplitude in each row a 2 `tuple` is used as described earlier. If the data type `double` is used to store the real and imaginary parts of the amplitude then the memory needed to store a state vector is $4 \cdot 2 \cdot 8$. The $2 \cdot 8$ comes from the 2 `tuple` storing the complex number and the 4 comes from the number of states. The memory needed to store the unitary gate matrix is equal to the memory needed to store the state vector multiplied by the number of columns in the unitary matrix (which is equal to the rows of both the matrix and the state vector). Therefore the memory required to store the entire unitary matrix is $4 \cdot 4 \cdot 2 \cdot 8$, where the extra factor of 4 comes from the number of columns of the matrix. The total memory needed is **384 bytes** and the breakdown is:

- Input state vector: 64 bytes
- Output state vector: 64 bytes
- Unitary gate matrix: 256 bytes

This is confirmed by using `heap` on the Mac OS system, when running the simulator. On average running Grover's algorithm on 2 qubits using matrix vector multiplications as described in this chapter takes **25.6 μ s**. The memory requirement of the quantum circuit can be summarised as follows,

$$m = 2 \cdot s_v + s_m, \quad (3.34)$$

where s_v and s_m comes from equations 3.28 and 3.30 respectively. The factor of 2 before s_v comes from the need to store both the input qubit state vector and output qubit state vector. Figure 3.8 shows the memory needed to store the unitary gate matrix (green), the qubit state vectors (blue) and the sum of the 2 (red). Note that the y -axis is plotted on a log scale. It is clear that the unitary matrix requires much more memory than the qubit state vector. This is expected as the matrix is larger. When simulating many qubits, the memory required to store the gate matrix is responsible for most of the memory requirements for the entire circuit. For 29 qubits, the unitary gate matrix requires **4194304 TiB**¹ of memory (99.99999963% of the total memory), whereas the qubit state vector requires **0.015625 TiB** of memory. This highlights the benefits that can be obtained if a more memory efficient way of simulating quantum circuits was developed. The memory requirements for larger qubit systems are staggering. Using the equation 3.34 and the data type of `double` the memory requirement for 150 qubits, for example, in bytes is

$$m = 2 \cdot 2 \cdot 8 \cdot 2^{150} + 2 \cdot 8 \cdot 2^{300} = 2^{155} + 2^{304}. \quad (3.35)$$

Even the the most powerful supercomputer according TOP500 [2] does not have enough memory to simulate a system with 150 qubits. The memory requirement and execution time are very important as they will be used to justify the newer method of simulating quantum circuits in the following chapter.

3.5. Conclusion

In this chapter, executing quantum circuits using vector matrix multiplications was explored. This was done by storing both the qubit amplitudes and the quantum gate matrix in memory. This chapter explained how vectors and matrices are stored in memory as 1D arrays and how the complex amplitudes

¹1 TiB = 2^{40} bytes

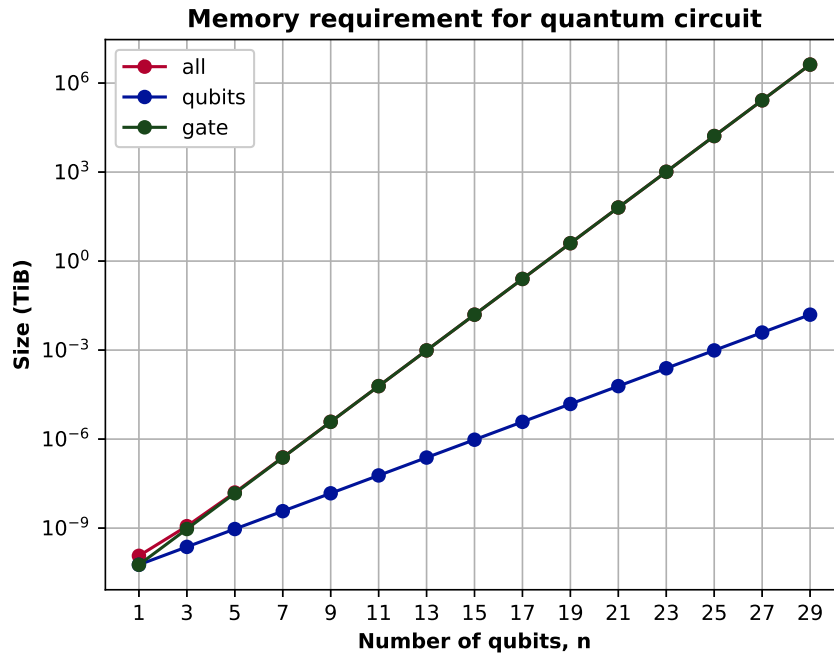


Figure 3.8: Memory required to simulate a quantum circuit when using `double`

of the qubit state vector and the complex entries in the quantum gate matrix are stored as 2 `tuples`. To simulate a quantum gate, the qubit state vector is multiplied by the gate matrix and the result is then stored in another vector. The example used to compute the results of simulating a quantum circuit this way was Grover's algorithm. This method of using vector matrix multiplications is perhaps the most common way of simulating a quantum circuit. Most quantum simulators that are being developed right now, use vector matrix multiplications. In the previous subsection, the results of this simulation was shown. For 29 qubits, the unitary gate matrix required 4194304 TiB of memory for storage.

It was made apparent that the gate matrix takes up an exorbitant amount of memory space, and increases exponentially as the number of qubits grows. The purpose of this chapter was to show how high the memory requirements of simulating a large quantum circuit can get and this highlights why it is important to reduce the memory requirements to simulate quantum circuits. It is a huge bottleneck and usually restricts the number of qubits that can be simulated.

4

Functional design II

4.1. Introduction

In this chapter, another functional implementation of a digital quantum simulator will be realised. From the previous chapter, the memory requirements for simulating a quantum system were shown to scale exponentially with the number of qubits. The primary contributor of this large memory requirement was the unitary gate matrix. For 29 qubits, the unitary gate matrix was responsible for 99.9999963% of the memory consumption. The main challenge when simulating a quantum system is the memory consumption. In this implementation, the unitary gate matrix will not be used to evolve the quantum state. Rather a classical approach will be taken to replace quantum gates as classical functions. The chapter begins by explaining how the simulator implements quantum gates and qubits in classical hardware. Examples are provided to show the storage of the qubits in memory. Furthermore, flow charts are provided to show the effect of gate operations on the qubits that are stored in the memory. To wrap up the design of the simulator, the class structure for it is provided to show all the gates and functions that can be used with QuantumSim. At the end of the chapter, the simulator is used to execute Grover's algorithm so it can be benchmarked. Both memory and execution time benchmarking are performed. These benchmarks will show the impact of QuantumSim on the memory requirement of simulating a quantum system and how it solves the memory bottleneck faced by quantum simulators.

4.2. Digitising the logic

The quantum gates will be encoded as classical functions. This is done by using the `bitset` template class for the quantum states and complex numbers for the amplitudes of these states. Compared to the previous chapter, the memory requirement is lower, due to the absence of the unitary gate matrix. The first subsection will explain the storage of qubits in memory and the following subsection will explain the gate operations in QuantumSim.

4.2.1. Memory

The qubit state vectors are stored in a 1D array. The input and output qubit state vectors are stored in the same array. The entries for each state are interleaved. This is done so that the input and amplitudes of the same state are next to each other and this reduces memory copying time as most gates either scale the amplitude or map to a quantum state that is close to it. The memory structure for an n qubit system is shown in figure 4.1. Since there are n qubits there are $N = 2^n$ possible states. The entries

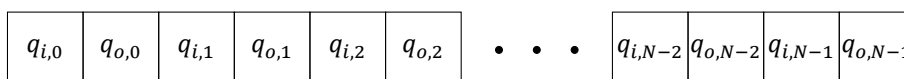


Figure 4.1: Qubit state vector in memory

$q_{i,x}$ are the amplitudes of the input qubit state and $q_{o,x}$ are the amplitudes of the output qubit state. This makes the array length equal to $2 \cdot N$. Each amplitude is a complex number, as explained in section

3.3.1. In the same section the memory requirement to store a qubit state vector was determined in equation 3.28. The memory requirement to store this new vector is simply double that amount,

$$s_v = 2 \cdot 2 \cdot s_T \cdot 2^{2n} = s_T \cdot 2^{n+2}. \quad (4.1)$$

s_T is the size of the type that is being used, as listed in table 3.2.

Recall that a qubit is represented as a vector. The entries are complex numbers. They can be rewritten as a `tuple` and using equation 3.27, the qubit state vector, $|0\rangle$, can be written as

$$|0\rangle = \begin{bmatrix} 1 + 0i \\ 0 + 0i \end{bmatrix} = \begin{bmatrix} (1, 0) \\ (0, 0) \end{bmatrix}. \quad (4.2)$$

In classical memory this vector is stored as shown in figure 4.2. Instead of the 2 elements, there are 4. The input qubit state vector amplitudes are stored in memory along with the output qubit state vector amplitudes (i.e. the values obtained after a gate operation). The output qubit state vector amplitudes are always initialised to the `tuple` $(0, 0)$. The amplitudes are interleaved. As described in section

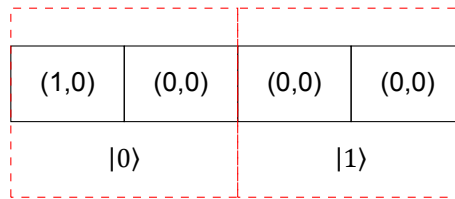


Figure 4.2: 1 qubit state vector in memory

3.3.1, each entry is stored as a `tuple`. The even elements correspond to the amplitudes of the input state vector and the odd elements correspond to the amplitudes of the output state vector. The odd elements will store the values of the amplitudes after a gate operates on the qubit. The reason for this will be explained in the next subsection, which describes the operations of the gates. As another example, figure 4.3 shows how a 2-qubit state vector $|\psi\rangle = \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle)$ is stored. There

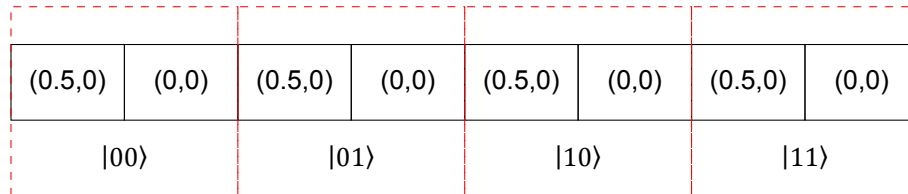


Figure 4.3: 2 qubit state vector in memory

are 8 elements in this array - 4 for the input and 4 for the output. The vector representation of the qubit state vector is written as

$$|\psi\rangle = \begin{bmatrix} (0.5, 0) \\ (0.5, 0) \\ (0.5, 0) \\ (0.5, 0) \end{bmatrix}. \quad (4.3)$$

4.2.2. Gate operations

The gate operations are where this design improves over the previous one. Each of the quantum gates is implemented as a classical function, that moves around amplitudes and performs arithmetic operations on them if needed. These classical functions are essentially mappings between the states in the computational basis. They are also able to map between states that are in superposition. This implementation of the quantum gate removes the need to store a matrix in memory, thereby reducing the memory requirements of QuantumSim.

Each gate iterates through the 2^N states and performs its operations on each of these states. Therefore each gate is called 2^N times every time it is applied. Listing 4.1 shows the implementation of the Pauli X gate as a classical function.

```

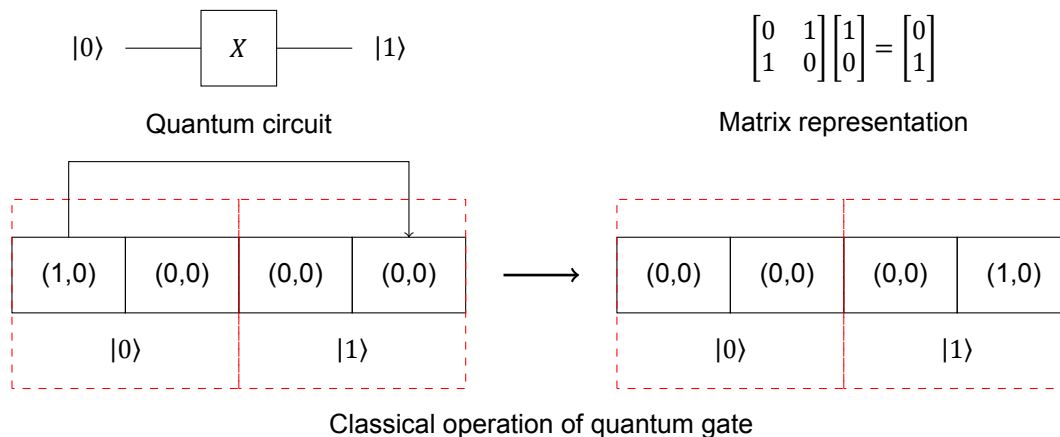
1 void QubitLayer::pauliX(int target){
2     for (int i = 0; i < numStates; i++){
3         if (checkZeroState(i)){
4             std::bitset<numQubits> state = i;
5             state.flip(target);
6             qL_[2*state.to_ulong()+1] = qL_[2*i];
7         }
8     }
9     updateLayer();
10 }

```

Listing 4.1: Pauli X function

The function takes as input one parameter, which is the address of the target qubit. `qL_` is the variable that points to the 1D array storing all the amplitudes of the states. It iterates through all the states in the qubit state vector, by using the indices of the array. To save some computation time, it only performs the X gate if that state has a non-zero amplitude. The `checkZeroState()` function returns `true`, if the state has a non-zero amplitude. It checks if either the real or imaginary component of the amplitude is 0. With the interleaved memory, the first index, $i = 0$, corresponds to the state $|00..00\rangle$, the third index, $i = 2$, corresponds to the state $|00..01\rangle$, the fifth index, $i = 4$, corresponds to the state $|00..10\rangle$, and so on. The address of the new state is obtained, by flipping the target qubit. Then the amplitude of the input state vector is set to the output state vector. The function `to_ulong()`, converts the binary state to an decimal index. This is needed as the flipping of the target qubit is done in binary. `state.to_ulong()` is the address of the state where the input amplitude is to be transferred. The address for the input state amplitude is $2 \cdot i$ and its corresponding output amplitude is $2 \cdot i + 1$.

The operation of the X gate on the 1D array is shown in figure 4.4. The corresponding quantum circuit and matrix representation are given along with the classical operation performed on the 1D array. This 1D array, described in figure 4.1, contains blocks/elements that are memory addresses, which hold the tuples representing the amplitudes. Each `tuple` represents a complex number and is comprised of 2 numbers, the first being the real part and the complex part. The classical operation of the quantum gate is moving around elements in the 1D array. For the X gate the first element (input state amplitude of $|0\rangle$) is moved to the last element (output state amplitude of $|1\rangle$).

Figure 4.4: Pauli X gate on $|0\rangle$

After iterating through all the states, and performing the X gate on the qubit state vector, the `updateLayer()` function is called. This is another member function, that simply maps the odd elements to the even elements in the 1D array, so that the next gate can be applied. Every gate always writes to an odd element and never an even element. This prevents overwriting any states in the array that have not had the gate act on it yet. Recall that contrary to a quantum gate, the classical version of the gate acts on each state sequentially and not in parallel.

The operation of the Y gate on the 1D array is shown in figure 4.5. It is similar to the X gate with an additional scalar multiplication. For the $|0\rangle$ state the scalar multiple is i and for $|1\rangle$ is $-i$.

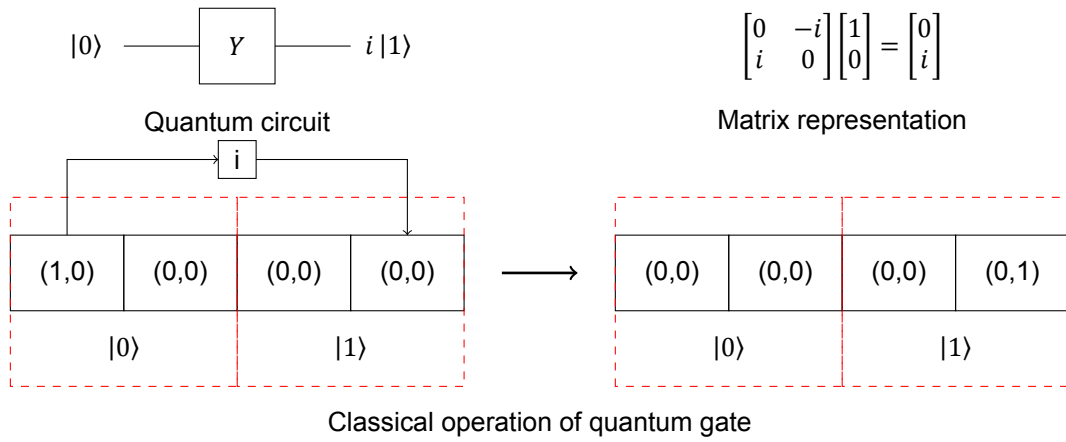


Figure 4.5: Pauli Y gate on $|0\rangle$

The operation of the Z gate on the 1D array is shown in figure 4.6. It adds a phase of -1 if the state is $|1\rangle$, otherwise it does nothing to the state as shown in figure 4.6.

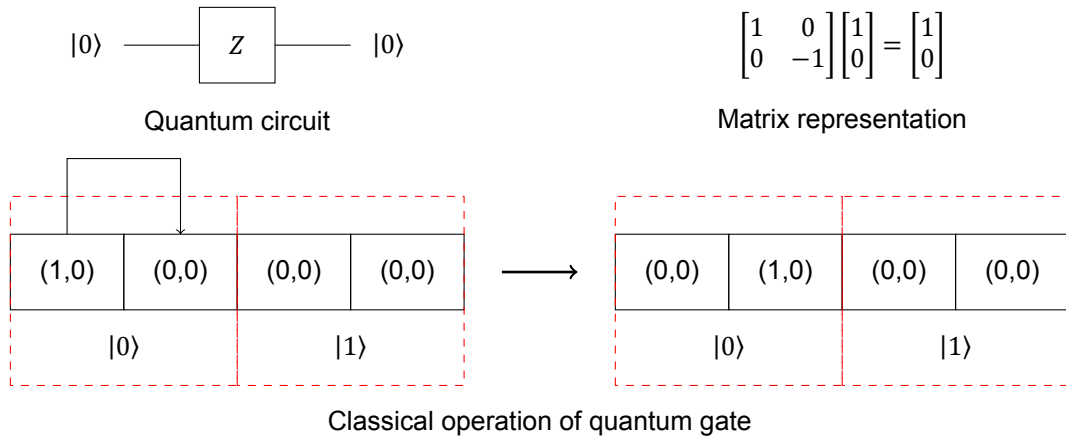


Figure 4.6: Pauli Z gate on $|0\rangle$

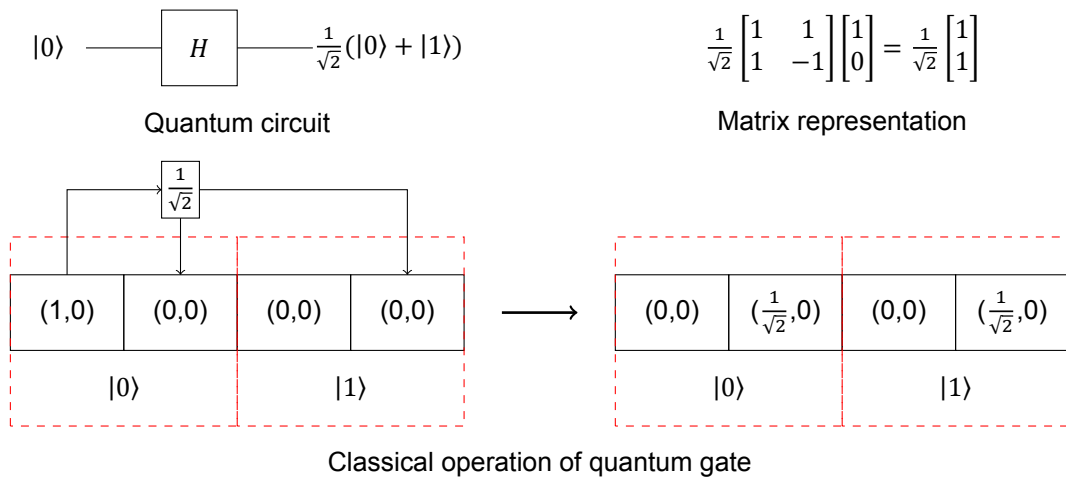


Figure 4.7: Hadamard gate on $|0\rangle$

The operation of the Hadamard, H , gate on the 1D array is shown in figure 4.7. It is important to note that for visual purposes the exact constant of $\frac{1}{\sqrt{2}}$ is used, but in digital memory, the precision of this value is determined by the datatype used. The more bytes that are used to represent this constant, the more precise it will be. This highlights the trade off between memory and accuracy. When H acts on $|1\rangle$, the constant $\frac{1}{\sqrt{2}}$ is multiplied by -1 , before the amplitude is stored in the array element for $|1\rangle$. For $|0\rangle$, this factor of $\frac{1}{\sqrt{2}}$ is not needed as the gate applies the same constant to both $|0\rangle$ and $|1\rangle$. The circuit representation of H shows that the Hadamard gate creates a superposition of 2 states. In the previous examples of the Pauli gates, superposition is not present.

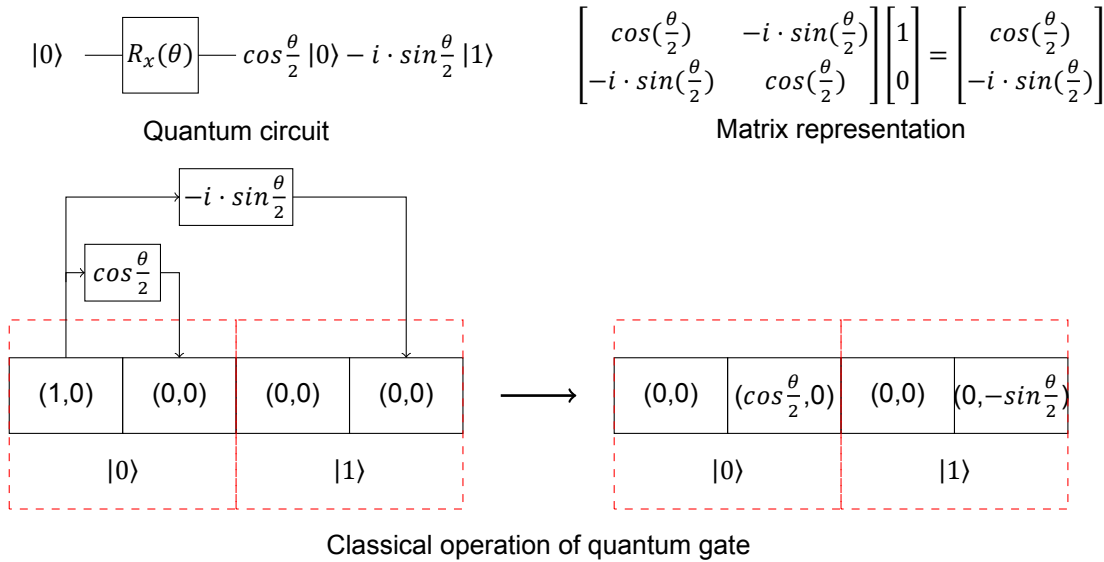


Figure 4.8: R_x gate on $|0\rangle$

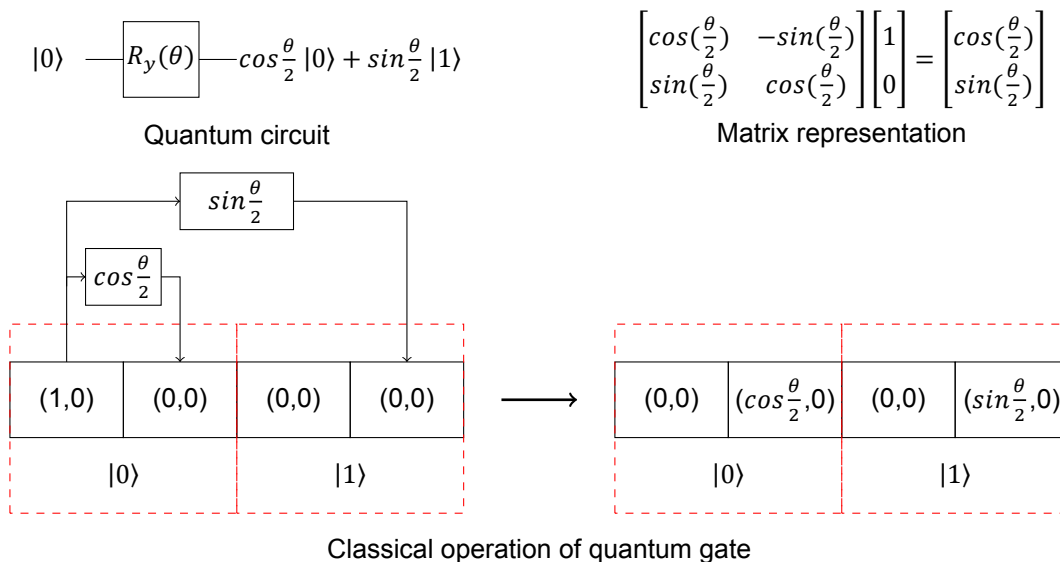


Figure 4.9: R_y gate on $|0\rangle$

The rotation gates deal with scalar constants that are sines and cosines of the rotation angle, θ . The operation of the rotation gate, R_x , on the 1D array is shown in figure 4.8. The constants to be multiplied when the gate acts on the state $|0\rangle$ are $\cos(\theta)$ and $-i \cdot \sin(\theta)$. The same figure also shows the corresponding quantum circuit and matrix representation of R_x . Similar to the Hadamard gate, R_x ,

creates a superposition of 2 states, but with different amplitudes.

The operation of the rotation gate, R_y , on the 1D array is shown in figure 4.9. Similar to R_x , R_y also creates a superposition of 2 states. The constants are different depending on whether the gate is acting on $|0\rangle$ or $|1\rangle$.

The operation of the rotation gate, R_z , on the 1D array is shown in figure 4.10. The constants in the R_z can be expanded using Euler's identity,

$$e^{-i\frac{\theta}{2}} = \cos\frac{\theta}{2} - i \cdot \sin\frac{\theta}{2}. \quad (4.4)$$

For visual purposes $e^{-i\frac{\theta}{2}} = \cos\theta - i \cdot \sin\theta = a + i \cdot b$. Therefore $a = \cos\theta$ and $b = -\sin\theta$.

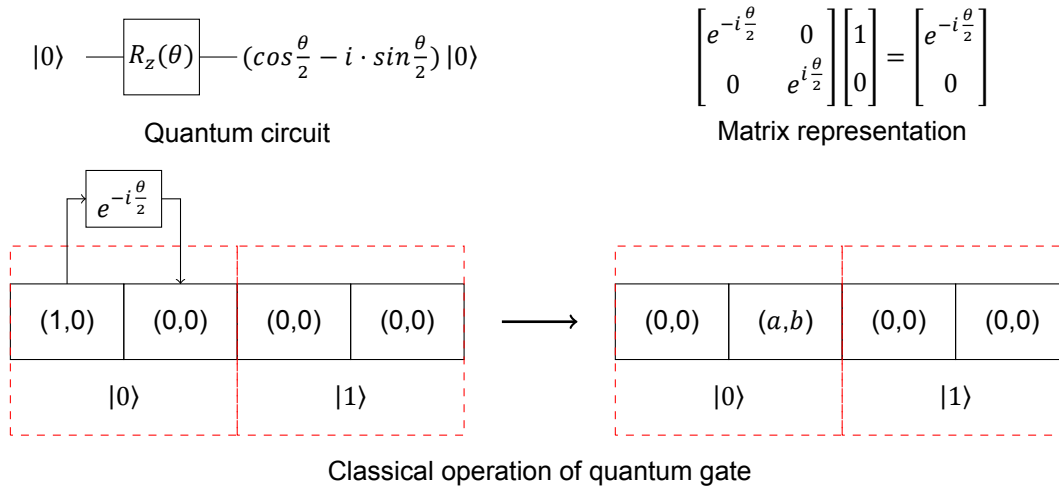


Figure 4.10: R_z gate on $|0\rangle$

The 2 qubit and 3 qubit gates have the same data flow, with the exception of performing simply binary checks on the target qubit, to determine if any data movement/arithmetic operation is required. The CNOT gate, is the X with a control qubit. If the control qubit is set, then an X gate is performed on the target. The same goes for the CPHASE and Toffoli gates.

4.2.3. Class structure

The main class used in this implementation is the `QubitLayer` class. This class stores the input qubit state vector and output qubit state vector as a 1D array. It also contains member functions that replace the unitary matrices needed to perform gate operations. Member functions manipulate data to compute the output qubit state vector. The class is implemented in C++ as shown in Listings 4.2.

```

1 class QubitLayer{
2     public:
3         QubitLayer(qubitLayer *qL = nullptr);
4         ~QubitLayer();
5         void pauliX(int target);
6         void pauliY(int target);
7         void pauliZ(int target);
8         void hadamard(int target);
9         void rx(int target, precision theta);
10        void ry(int target, precision theta);
11        void rz(int target, precision theta);
12        bool checkControls(int *controls, int numControls, std::bitset<
            numQubits> state);
13        void cnot(int control, int target);
14        void toffoli(int controll1, int controll2, int target);

```

```

15     void mcnot(int *controls, int numControls, int target);
16     void cphase(int control, int target);
17     void mcphase(int *controls, int numControls, int target);
18     bool checkZeroState(int qubit);
19     qProb getMaxAmplitude();
20     void updateLayer();
21     void printMeasurement();
22     void printQubits();
23 private:
24     qubitLayer *qL_;
25 };

```

Listing 4.2: QubitLayer class

Most of the functions are the quantum gates. The first 2 are the constructor and destructor, which perform the memory allocation and deallocation. This ensures that there are no memory leaks, which could be drastic for large qubit systems. There are some user defined typedefs. The first one is `precision`, which is simply one of the data types in table 3.2. By using a typedef the entire precision of the simulator can be changed by changing one value. `qProb` is a user defined struct. It contains 2 entries-the qubit state and the the corresponding probability of the state. `qubitLayer` is a compact version of `std::complex<precision>`.

4.3. Benchmarking

To evaluate this design, experiments are performed to measure the execution time and memory consumption of quantum algorithms. For these benchmarks, all the variables will be of type `double`, i.e. each value is 8 bytes, and therefore each complex number is 16 bytes. Therefore each amplitude is represented using 16 bytes. The data type defines the precision of the amplitudes.

4.3.1. System specifications

The system specifications that these experiments are performed on are given in table 4.1.

Processor speed	2.6 GHz
Number of cores	4
L1 cache	32 KB ¹
L2 cache/core	256 KB
L3 cache	6 MB
RAM	16GB @ 1600 MHz

Table 4.1: System specifications

¹same for instruction cache and data cache

4.3.2. Memory benchmarking

The memory benchmark is very predictable, since it is solely determined by the number of qubits being initialised and not the quantum algorithm being used. Grover's algorithm is performed on a range of qubits with the solution at the same location. The profiling tool being used is the `heap` command line tool. The C++ program is run without any code, to establish the base allocations for a C++ program. Table 4.2 shows the amount of memory allocated as the number of qubits is increased. It can be seen that the memory requirement doubles for every additional qubit and it is trivial to calculate the memory needed to simulate larger qubit systems. The memory allocation is in line with the discussion in section 3.3.1.

Number of qubits (n)	2	3	4	5	6	7	8	9	10	11	12	13
Memory (KiB)	0.125	0.25	0.5	1	2	4	8	16	32	64	128	256
Number of qubits (n)	14	15	16	17	18	19	20	21	22	23	24	25
Memory (MiB)	0.5	1	2	4	8	16	32	64	128	256	512	1024

Table 4.2: Memory requirement for different number of qubits

4.3.3. Gate time benchmarking

The number of qubits, the circuit depth and the exact gates being used are the 3 main factors that determine the execution time of a complete circuit. For a classical simulator, the circuit depth will refer to the total number of gates, since the entire program runs sequentially. The first item to benchmark is the execution times of each gate. Each gate is implemented classically and to measure the execution time for each gate, 1 qubit is used for single qubit gates, 2 qubits for two qubit gates and 3 qubits for 3-qubit gates. This is done to highlight the difference between the gate times and more importantly the reasons for these differences. `steady_clock` is used to measure the execution times. This is a class in the `chrono` library. This particular clock is used as it is monotonic and therefore perfect for measuring time intervals. Table 4.3 shows the gate times for the basic gates implemented in the quantum simulator.

Gate	X	Y	Z	H	R_x	R_y	R_z	CNOT	CPHASE	Toffoli
T (ns)	298.4	422.2	308.0	464.6	490.4	448.0	521.0	363.2	690.6	503.2

Table 4.3: Single qubit gate times

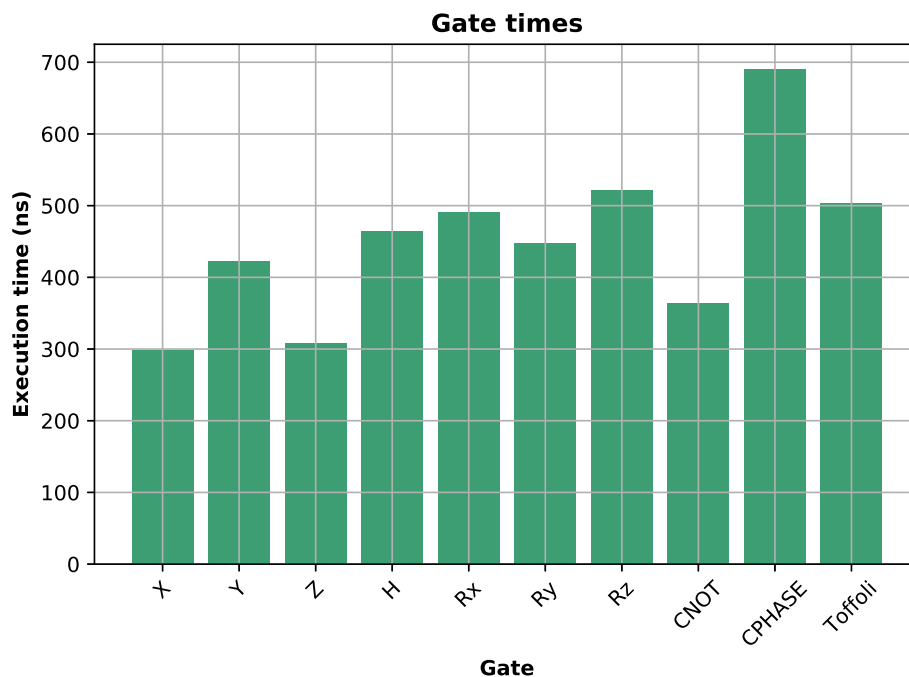


Figure 4.11: Gate times

The results are also shown in figure 4.11. The y-axis shows the execution time in nanoseconds. The execution time for each gate is affected by 3 factors:

1. The state of the qubit

2. The number of operations performed by each gate
3. The number of qubits

Each of the gates are only performed on states that have a non-zero amplitude. Therefore the execution time when performing an X gate on the state $|0\rangle$ is lower than if it were performed on the state $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$. The input state for the single qubit gates is $|0\rangle$. The fastest gate is the X gate. This is because in the computational basis, it simply moves a value from one address in the 1D array stored in memory to another address. The Y and Z gates have to check the qubit state before applying their operations. This makes them slower than the X gate. Furthermore both of them also need to multiply a scalar constant to the amplitude. The Z gate need only apply the phase -1 , but the Y gate applies a phase of $\pm i$ depending on the state of the qubit. This makes Y slower than Z . The Hadamard gate is not only applying a scalar multiple of $\frac{1}{\sqrt{2}}$ to the current state, but it is also "creating" another state based on the input state. Recall that the action of the Hadamard gate is $|0\rangle \rightarrow \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$. It is multiplying a constant of $\frac{1}{\sqrt{2}}$ to $|0\rangle$ and "creating" a new state $\frac{1}{\sqrt{2}}|1\rangle$. The rotations gates taken longer as they all need to compute sines and cosines of the rotation angle. R_z takes the longest as it has to multiply both the sine and cosine of the rotation angle to the states. R_x takes longer than R_y as it also has a complex term in the scalar multiple for the states. The CNOT gate operates on 2 qubits, and does exactly what the X gate does except with an additional checked on the control qubit. The CPHASE gate has to perform a scalar multiplication by -1 based on the target qubit and that makes it slower than the CNOT gate. The Toffoli gate acts on 3 qubits but is considerably faster than both the R_z gate and the CPHASE gate as it is simply moving data and not performing and computations.

4.3.4. Qubit benchmarking

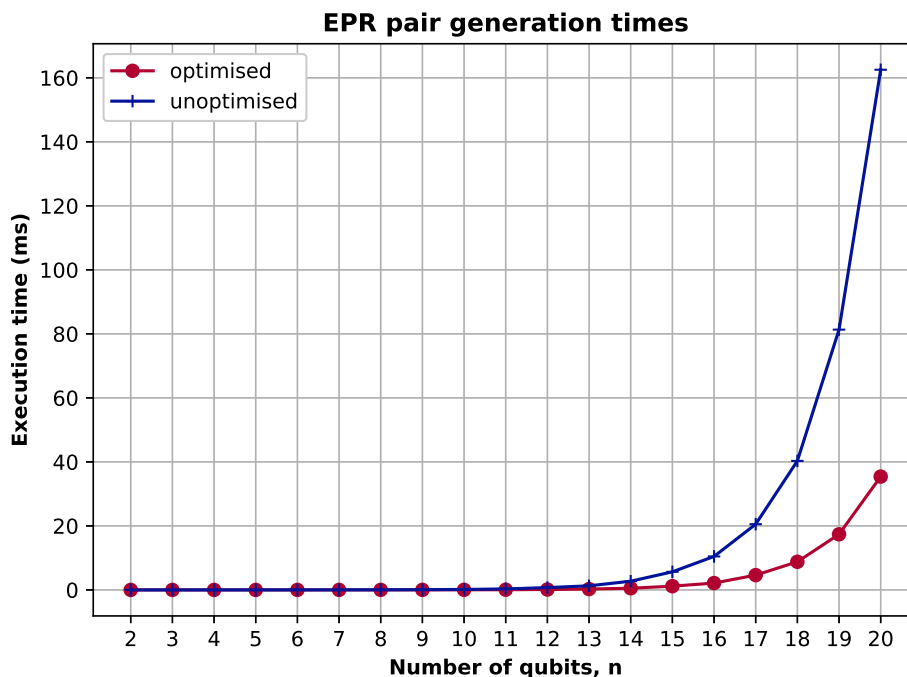


Figure 4.12: Generating EPR pairs as the number of qubits scale

In this subsection, the effect of increasing the number of qubits will be explored. The first experiment is carried out by using the same number of gates and while increasing the number of qubits. The goal of this experiment is to investigate whether gates acting on larger arrays result in longer execution times. This experiment uses a simple EPR pair generation circuit as shown earlier in figure 2.1. It uses an unoptimised version of the simulator and an optimised version. The optimisation is the skipping of

states that have 0 amplitude, hereinafter referred to as zero-state skipping. Every gate skips states that have no amplitude, as there is no reason to waste computation time on these states. The results of this experiment are shown in figure 4.12. The optimisation is clearly effective, especially as it has almost no overhead. Another result to note is that the execution time of the circuit increases with the number of qubits. This is because there are more states to iterate over. The amplitude checking optimisation will skip 0 amplitude states, but the gate will still have to iterate over all 2^n states and perform this check for each of those states.

4.3.5. Circuit depth benchmarking

The last factor that determines the execution time of a circuit is the depth of the circuit, i.e. how many gates are being used. Intuitively it is expected that the more gates a circuit uses, the longer its execution time. The experiment is carried out by adding an extra gate every time and measuring the execution

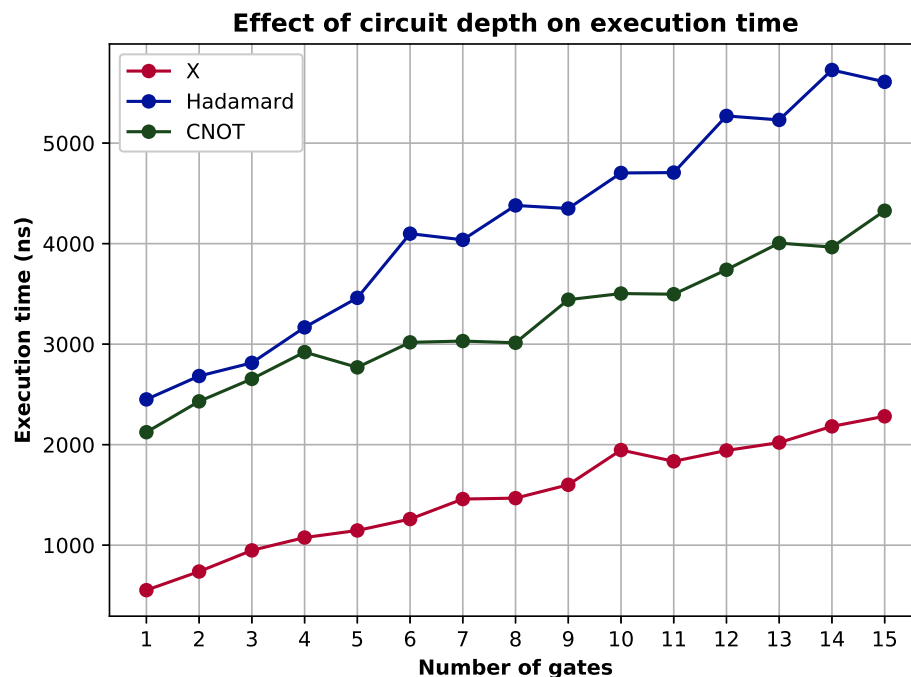


Figure 4.13: Effect of number of gates on the execution time

time of the circuit. As can be seen in figure 4.13, the execution time increases as the number of gates applied does. This relationship however, is linear, compared to the previous experiment where the effect of the number of qubits on the execution time was explored and the trend was exponential. This experiment is carried out on 2 qubits (to test two qubit gates as well) and the timing is measured after memory allocation has been performed, so only the the execution time of the gates is taken into account. It is carried out on 3 different gates to show that the relationship is indeed linear.

4.3.6. Grover's Algorithm

The simulator is tested on Grover's algorithm as a way to evaluate it on a real application. Grover's algorithm uses the Pauli X gate, the Hadamard gate and the multiple controlled Z gate, as can be seen in figure 3.6. The results are shown figure 4.14. The y -axis is plotted on a log scale. This experiment encompasses all the factors that affect the execution time of a quantum circuit. As the number of qubits increase so does the circuit depth. The number of iterations of applying Grover's algorithm increases with the number of qubits, which further increases the circuit depth. Recall that for a single solution the oracle and the Grover diffusion operation are performed $\lfloor \frac{\pi\sqrt{N}}{4} \rfloor$ times [1], where $N = 2^n$ and n is the number of qubits. The execution time scales exponentially with the number of qubits. The main reason for this is shown in the benchmark performed in section 4.3.4.

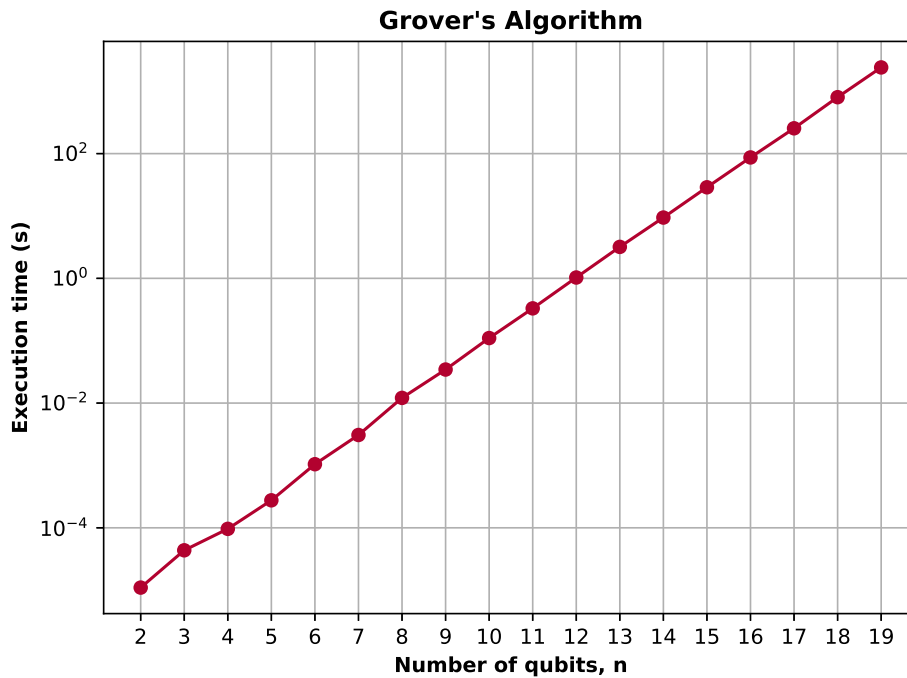


Figure 4.14: Grover's Algorithm

4.4. Conclusion

In this chapter, a novel way of performing quantum operations was introduced. This method does not need to store unitary matrices for the quantum gates. Instead it uses the defined mappings that are obtained by applying any quantum gate to the 2 basis states in the standard basis, $|0\rangle$ and $|1\rangle$. This results in a lower memory requirement, as the only part of the circuit that needs to be stored, is the state vector of the qubits. The state vector of the qubits is the vector containing the amplitudes of each state of the qubit. It grows exponentially with the number of qubits. The execution of this simulator has an exponential dependence on the number of qubits. This is due to the way that each quantum gate is applied. This simulator iterates over all the possible states, which scales exponentially with the number of qubits. As the number of applications of the quantum gate depend on the number of states, which depends exponentially on the number of qubits, the execution time is therefore exponentially dependant on the number of qubits as well. The simulator designed in chapter 3 had to store the unitary matrix. The simulator designed in this chapter, QuantumSim, does not. Using equation 3.34 the memory requirement for simulating 150 qubits with the simulator in chapter 3 was

$$m_{old} = 2 \cdot s_v + s_m = 2^{155} + 2^{304}. \quad (4.5)$$

s_v is the amount of memory needed to store the qubit state vector and s_m is the amount of memory needed to store the unitary matrix. The memory requirement for the simulator designed in this chapter was

$$m_{new} = 2^{155}. \quad (4.6)$$

The second term 2^{304} is not needed in the new simulator, as it corresponds to the memory needed to store the unitary matrix.

5

Optimisations to the Quantum Simulator

Art is never finished, only abandoned.

Leonardo da Vinci

5.1. Introduction

After investigating the initial design of the quantum simulator, there are several potential avenues for improvement, when it comes to reducing the execution time. The zero-state skipping optimisation was done in the previous chapter. This optimisation skips over states, which have no amplitude (i.e. an amplitude of zero). The main purpose of this thesis was to design a memory efficient Quantum simulator. With that goal achieved, the next step would be to improve the execution time of the simulator. In this chapter, 2 other optimisations will be implemented. First, a technique will be implemented to reduce the copying of data. This will be referred to as the parity system as it splits the input and output qubit state vectors into odd and even arrays. The second optimisation is a parallel implementation using OpenMP. This naive parallel implementation is there to show that QuantumSim is parallelisable. As part of the future development of QuantumSim a more customised parallel implementation can be explored. These 2 optimisations show that QuantumSim can still be optimised and there is much to do in the development of this simulator.

5.2. Parity system

The parity system reduces the time needed to prepare the qubit state vectors for the next gate. The next subsection explains the current method of preparing the array for the next gate and the subsection following it introduces the parity system.

5.2.1. Initial method

The initial design of the quantum simulator, requires the copying of the output amplitudes to the input amplitudes, so that the next gate can be applied. This is done using the `updateLayer()` function provided in the `QubitLayer` class. This function is shown in Listings 5.1

```
1 void QubitLayer::updateLayer() {
2     for (int i = 0; i < numStates; i++) {
3         qL_[2*i] = qL_[2*i+1];
4         qL_[2*i+1] = {0,0};
5     }
6 }
```

Listing 5.1: `updateLayer()` function

As with the other gate functions, `updateLayer()` iterates through all the states. First in line 4 it copies the odd entry to the even entry in the qubit state vector array that stores the amplitudes. The structure of

this array was shown in section 4.2.1 in figure 4.1. Then in line 4, it resets the value of the odd element to 0. The `tuple {0,0}` sets the real and complex values to 0. This ensures that the odd elements are ready to accept the output of the next gate.

5.2.2. Parity method

The parity system splits the 1D array that was storing the amplitudes of the qubit state vector into 2 separate arrays, namely an even array and an odd array. This is done to reduce the copying of data after every application of the gates. The first problem to tackle is keeping tracking of the number of gates that have been applied. This is done by setting up a simple boolean variable that toggles whenever a gate is applied, i.e. toggles between **true** and **false** (or "odd" and "even"). This makes line 3 in Listings 5.1 redundant, thereby improving the performance of the `updateLayer()` function as it does not need to copy data anymore and therefore is much faster. An example of a gate application (X gate) is shown in figure 5.1. The red blocks correspond to the array that has *even* parity and the blue blocks correspond to the array that has *odd* parity. After the gate is applied the *even* array is reset to

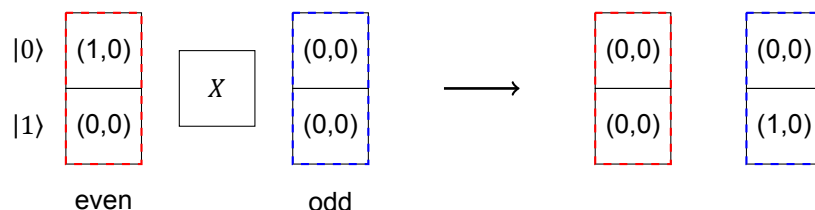


Figure 5.1: Applying gate 1

all 0s. In the parity system, the `updateLayer()` function zeros out the input array ("even" for the first gate) and does not copy anything between the 2 arrays, unlike the design in Chapter 4. It also toggles the parity which is a member variable of the `QubitLayer` class. Applying the next gate, a Z gate for example, simply switches the order of the *odd* and *even* arrays as shown in figure 5.2. The input is the

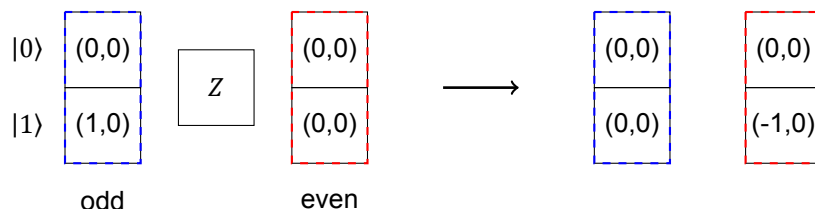


Figure 5.2: Applying gate 2

odd array this time, which was the output of the previous gate, X . The output is the *even* array which was reset to 0 by the `updateLayer()` function called by the X gate. If a third gate were to be applied it would look like figure 5.1, when the input would be the *even* array and the output the *odd* array. The X gate is shown in Listings 5.2.

```

1 void QubitLayer::pauliX(int target){
2     for (int i = 0; i < numStates; i++)
3         if (checkZeroState(i)){
4             std::bitset<numQubits> state = i;
5             state.flip(target);
6             parity ? qOdd_[state.to_ulong()] = qEven_[i] : qEven_[state.
              to_ulong()] = qOdd_[i];
7         }
8     updateLayer();
9 }

```

Listing 5.2: Pauli X function

The ternary `if` statement decides whether the *even* or *odd* array is the input. The new `updateLayer()` function using the parity system is shown in Listings 5.3.

```

1 void QubitLayer::updateLayer(){
2     parity ? std::fill(qEven_, qEven_ + numStates, zeroComplex) : std::
3         fill(qOdd_, qOdd_ + numStates, zeroComplex);
4     toggleParity();
5 }

```

Listing 5.3: new `updateLayer()` function

The `std::fill()` function is used to reset the *odd* or *even* array depending on the parity to *zeroComplex* which is simply the tuple `(0,0)`. The last operation performed by the `updateLayer()` function is to toggle the parity of the quantum circuit, so the next gate has the right input and output arrays.

This new method removes the need to copy data between the 2 arrays and therefore improves the performance of the simulator. The parity system "switches" which array is used as the input by tracking the number of gates that have been performed. To test this parity system, an experiment is carried out on up to 19 qubits. QuantumSim runs Grover's algorithm to evaluate the improvement provided by the parity system. The improvement can be seen in figure 5.3. While the figure may not show a lot

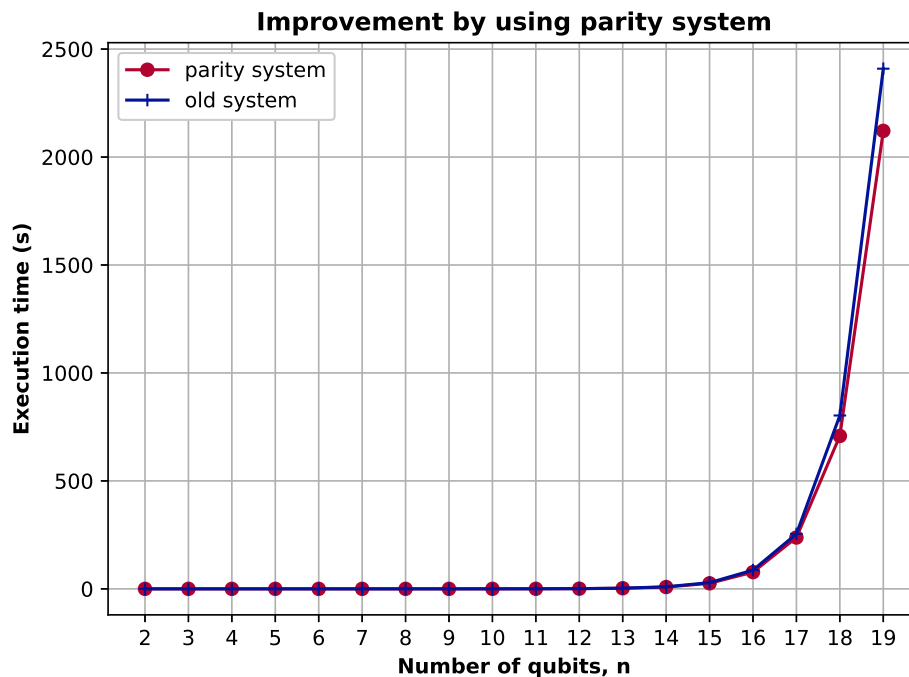


Figure 5.3: Improvement using the parity system

of improvement in the lower range due to the scale of the y axis, the improvement does exist. A linear scale to plot this was chosen to show the improvement at the higher end of the x axis. On average the speedup by using the parity system is about 1.105, which is **> 10%**.

5.3. Parallelisation

Parallelisation is accomplished in the quantum simulator by using OpenMP. It is an API that makes multiprocessing programming for shared-memory architectures easier. Every gate applied to an n -qubit state vector is applied 2^n times, i.e. applied to each of the states. This is done sequentially. To realise parallelisation, the quantum gates can be broken up into 2 types of gates:

1. **One-to-one mapping**: one state mapped to another
2. **One-to-two mapping**: one state mapped to 2 others

5.3.1. One-to-one mapping

An example of a gate that has one to one mapping is the X gate, as it maps $|0\rangle$ to $|1\rangle$ and vice versa. Since the mappings are unique, there will be not contention for the shared memory. Threads can be assigned to the gate, so that the gate can be applied to multiple states at the same time. To parallelise a loop a `#pragma` directive can be used. On the gates with one-to-one mapping the following directive will parallelise the gate

```
#pragma omp parallel for shared(qOdd_, qEven_)
```

It is placed right before the for loop that iterates over the states. The directive also specifies the shared variables so that the threads have access to the same data. The shared variables are `qOdd_` and `qEven_`. As an example consider the 2 qubit state $|\psi\rangle$,

$$|\psi\rangle = \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle). \quad (5.1)$$

Applying the X gate to the first qubit flips it. The simulator iterates over the $2^2 = 4$ states and performs this X gate on the first qubit of every state,

$$X|\psi\rangle = \frac{1}{2}(|10\rangle + |11\rangle + |00\rangle + |01\rangle). \quad (5.2)$$

A one-to-one mapping was performed in the example:

- $\frac{1}{2} |00\rangle \rightarrow \frac{1}{2} |10\rangle$
- $\frac{1}{2} |01\rangle \rightarrow \frac{1}{2} |11\rangle$
- $\frac{1}{2} |10\rangle \rightarrow \frac{1}{2} |00\rangle$
- $\frac{1}{2} |11\rangle \rightarrow \frac{1}{2} |01\rangle$

The mapping determines where to move the amplitude of the state. This mapping is highly parallelisable. A thread can be allocated for the gate operation on each state. In this example, 4 threads would do the job in parallel.

5.3.2. One-to-two mapping

An example of a gate that maps one state to 2 is the Hadamard gate as shown in equation 5.3.

$$|0\rangle \rightarrow \frac{1}{\sqrt{2}} |0\rangle + \frac{1}{\sqrt{2}} |1\rangle \quad (5.3)$$

The state $|0\rangle$ is mapped to both $|0\rangle$ and $|1\rangle$ with scalar constants multiplied for normalisation. Gates that have a one-to-one mapping are easy to parallelise, as shown in the previous subsection, but a gate with one-to-two mapping is harder as there could be race conditions when writing to the same block of memory, since the mapping is not unique. The way to parallelise this is to split the gate into 2 sub gates, and each of them performs a one-to-one mapping. The Hadamard gate is split into two subgates. The first sub gate performs the following mapping based on the input

$$|0\rangle \rightarrow \frac{1}{\sqrt{2}} |0\rangle \quad |1\rangle \rightarrow -\frac{1}{\sqrt{2}} |1\rangle. \quad (5.4)$$

The next sub gate flips the input state and then applies the scalar coefficient,

$$|0\rangle \rightarrow \frac{1}{\sqrt{2}} |1\rangle \quad |1\rangle \rightarrow \frac{1}{\sqrt{2}} |0\rangle. \quad (5.5)$$

Both these sub gates realise the Hadamard gate. The parallel version of the Hadamard gate is shown in Listing 5.4.

```

1 void QubitLayer::hadamard(int target){
2     //map |1> to -hadamardCoef*|1> and |0> to hadamardCoef*|0>
3     #pragma omp parallel for shared(qOdd_, qEven_)
4     for (int i = 0; i < numStates; i++)
5         if (checkZeroState(i)){
6             std::bitset<numQubits> state = i;
7             if (state.test(target))
8                 parity ? qOdd_[i] -= hadamardCoef*qEven_[i] : qEven_[i] -=
9                     hadamardCoef*qOdd_[i];
10            else
11                parity ? qOdd_[i] += hadamardCoef*qEven_[i] : qEven_[i] +=
12                    hadamardCoef*qOdd_[i];
13        }
14    #pragma omp barrier
15    //map |0> to hadamardCoef*|1> and |1> to hadamardCoef*|0>
16    #pragma omp parallel for shared(qOdd_, qEven_)
17    for (int i = 0; i < numStates; i++)
18        if (checkZeroState(i)){
19            std::bitset<numQubits> state = i;
20            state.flip(target);
21            parity ? qOdd_[state.to_ulong()] += hadamardCoef*qEven_[i] :
22                qEven_[state.to_ulong()] += hadamardCoef*qOdd_[i];
23        }
24    updateLayer();
25 }

```

Listing 5.4: Hadamard function

The downside of splitting the gate into 2 sub gates, is that the gate now performs 2 sweeps over the 2^n states. The directive `#pragma omp barrier` makes sure that the previous threads are all completed before the next sub gate can start. This prevents any memory contention issues.

5.3.3. Results

Figure 5.4 shows a log plot of the execution time of the sequential version of the simulator and the parallel version. For a low number of qubits (< 9 for this example), the parallel version provides no speedup. In fact, with a relatively low number of qubits, the parallelisation slows down the algorithm. This is attributed to the overhead that is a result of OpenMP creating, scheduling and managing the threads. After 9 qubits, the overhead is no longer an issue and the speedup provided by the multiple threads compensates for it. On the local machine that is being used to run the simulator **4 cores** are available. Therefore a maximum theoretical speedup of 4 can be achieved assuming that the entire simulator is parallelisable, which is indeed the case. As explained earlier, there is some overhead that is a result of the thread management provided by OpenMP, so this theoretical speedup is never possible. Secondly, Grover's algorithm uses the Hadamard gate, which is not perfectly parallelisable as explained earlier in this text, due to the one-to-two mapping of the gate. The average speedup (for more than 10 qubits) for Grover's algorithm is approximately **2.31**. This value is highly dependent on the algorithm and in turn the type of gates that are used. For algorithms that use gates that create superpositions (i.e. one-to-two mappings) the speedup will be lower.

5.4. Conclusion and future work

In this chapter, two optimisations were implemented in QuantumSim. Both of them resulted in a speedup. The parity system resulted in a speedup of more than 10%, due to a reduction in the copying of data. The experiment carried out to calculate this speedup was done on up to 19 qubits using Grover's algorithm. The mapping method used by QuantumSim is not completely parallelisable, due to certain quantum gates that have one-to-two mappings, which requires deconstructing these gates to 2 gates with one-to-one mappings. The speedup achieved when parallelising QuantumSim using OpenMP is approximately 2.31. Running this simulator on one node, limits the memory that can be used. If QuantumSim could be run on a distributed system, then a larger number of qubits could be

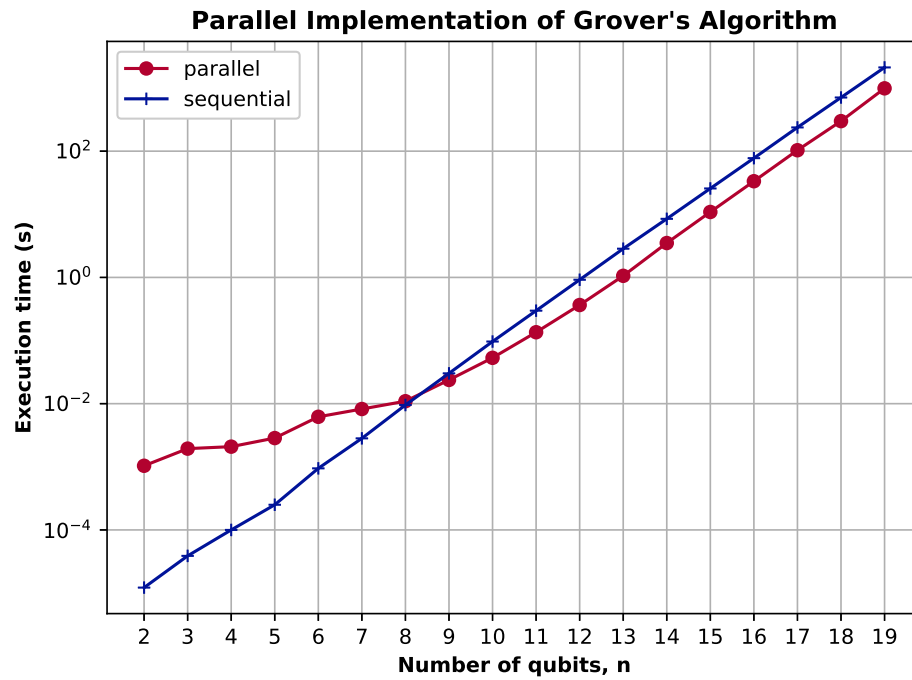
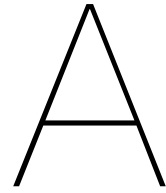


Figure 5.4: Parallel implementation of simulator

simulated. MPI (message passing interface) is a standard that can be used to parallelise QuantumSim over multiple nodes and thereby allow the study and execution of larger quantum systems.



Background information

A.1. Commonly used gates

Table A.1 shows the general rotation gates and the Pauli gates.

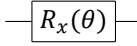
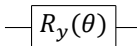
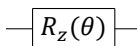



Name	Unitary	Circuit Symbol
Rotation-X	$R_x(\theta) = \begin{bmatrix} \cos\frac{\theta}{2} & -i\sin\frac{\theta}{2} \\ -i\sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{bmatrix}$	
Rotation-Y	$R_y(\theta) = \begin{bmatrix} \cos\frac{\theta}{2} & -\sin\frac{\theta}{2} \\ \sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{bmatrix}$	
Rotation-Z	$R_z(\theta) = \begin{bmatrix} e^{-i\frac{\theta}{2}} & 0 \\ 0 & e^{i\frac{\theta}{2}} \end{bmatrix}$	
Pauli-X	$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$	
Pauli-Y	$Y = \begin{bmatrix} 0 & i \\ -i & 0 \end{bmatrix}$	
Pauli-Z	$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$	

Table A.1: Rotation gates and Pauli gates

The first 3 gates are the rotation gates, with which every other single qubit gate can be created. These

3 gates cover the 3 axes that a qubit can be rotated around. The next 3 are the Pauli gates, which are based on the Pauli matrices that occur in the Pauli equation. There are other single qubit gates that can be constructed from the rotation operators. These are shown in table A.2. Along with single qubit





Name	Unitary	Circuit Symbol
Identity	$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$	
Hadamard	$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix}$	
Phase	$S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$	
T	$T = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{4}} \end{bmatrix}$	

Table A.2: Other single qubit gates

gates, there are multi-qubit gates which operate on 2 or more qubits. Commonly used 2 qubit gates are shown in table A.3.

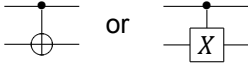
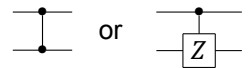

Name	Unitary	Circuit Symbol
Controlled Not	$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$	
Controlled Z	$CZ = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$	
Swap	$SWAP = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	

Table A.3: Commonly used 2 qubit gates

Commonly used 3 qubit gates are shown in table A.4.



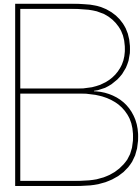
Name	Unitary	Circuit Symbol
Toffoli	$CCNOT = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$	
Fredkin	$CSWAP = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$	

Table A.4: Commonly used 3 qubit gates



OpenQL

B.1. OpenQL options

Table B.1 shows the full list of options that the compiler accepts.

Option	Value	Description
<code>scheduler</code>	ASAP or ALAP	Type of scheduling for the circuit. The options are As Soon As Possible (ASAP) or As Late As Possible (ALAP)
<code>output_dir</code>	<PATH>	The path to where the QASM code will be stored
<code>write_qasm_files</code>	yes or no	Boolean value to determine if QASM code is to be written
<code>scheduler_uniform</code>	yes or no	Boolean value to determine if QASM code is to be written
<code>scheduler_commute</code>	yes or no	Commutation rules for CNOT and CZ gates are used if set to <code>yes</code> . This freedom reduces circuit latency
<code>print_dot_graphs</code>	yes or no	Prints dependence graph of circuit using dot notation

Table B.1: OpenQL options

Bibliography

- [1] Apr 2020. URL https://en.wikipedia.org/wiki/Grover's_algorithm.
- [2] Jun 2020. URL <https://www.top500.org/news/japan-captures-top500-crown-arm-powered-supercon>
- [3] C. Figgatt, D. Maslov, K. A. Landsman, N. M. Linke, S. Debnath, and C. Monroe. Complete 3-qubit grover search on a programmable quantum computer. *Nature Communications*, 8(1), Dec 2017. ISSN 2041-1723. doi: 10.1038/s41467-017-01904-7. URL <http://dx.doi.org/10.1038/s41467-017-01904-7>.
- [4] Lov K. Grover. A fast quantum mechanical algorithm for database search, 1996.
- [5] Julian Kelly. A preview of bristlecone, google's new quantum processor, Mar 2018. URL <https://ai.googleblog.com/2018/03/a-preview-of-bristlecone-googles-new.html>.
- [6] N. Khammassi, G. G. Guerreschi, I. Ashraf, J. W. Hogaboam, C. G. Almudever, and K. Bertels. cqasm v1.0: Towards a common quantum assembly language, 2018.
- [7] Peter W. Shor. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM Review*, 41(2):303–332, January 1999. doi: 10.1137/S0036144598347011.