Imperial College London

MENG INDIVIDUAL PROJECT - FINAL REPORT

IMPERIAL COLLEGE LONDON

DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING

Valkyrie and Visual Q - Quantum Computing Simulator and Visual interface

---

Supervisor:
Dr. Zahid Durrani

Author:
Neelesh Ravichandran

Second Marker:
Prof. Stepan Lucyszyn

March 18, 2023

Final Report Plagiarism Statement

I affirm that I have submitted, or will submit, an electronic copy of my final year project report to the provided EEE link. I affirm that I have provided explicit references for all the material in my Final Report that is not authored by me, but is represented as my own work.

Dedication

In memory of my grandparents Dhanalakshmi and Santhanam, who have supported and guided me throughout my life and have helped make me the person I am now.

Abstract

GPU accelerated quantum computer emulation using a C++/CUDA backend and an easy-to-use user interface using the NodeJS Electron UI framework, is presented. This provides an alternative to current, widely available quantum computer simulators written in Python, where computing overheads limit the scale of the quantum circuits and the speed of execution of the simulations. The quantum computer simulator uses a widely adopted quantum assembly language named OpenQASM, which provides a gate level specification language for quantum circuits. The simulator can provide a Javascript Object Notation (JSON) output to maximise compatibility with supporting applications. The implementation is discussed in depth, including the parsing of OpenQASM code, simulating quantum operations and simulating measurement of quantum states into classical registers. Several experiments which test different quantum circuit simulations are also documented, these measure the performance of the presented simulator against its contemporaries. One of the circuits tested is the Deutsch-Jozsa algorithm for measuring balanced functions, which is given for different levels of complexity to provide comprehensive coverage of the use case for the simulator. Execution speed increases range from 20-60%, depending on circuit size and complexity, with results presented and analysed. All code produced is documented in the Appendix and a link is provided to a cloud based repository of the codebase.

Acknowledgements

I would like to thank my project supervisor Dr Zahid Durrani, who has been a constant source of academic and personal support for the entirety of this project. Furthermore, I'd like to thank Shin Morino who helped me understand some of the technical aspects of this project. Finally I'd like to thanks my entire family, especially my parents who have sacrificed much to ensure I could succeed.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Quantum computing is a fast growing new technology, based on physics that we have only understood for the better half of a century. The reader may ask what the difference is between a quantum computer and a classical computer, to this the answer is quite simple; a classical computer stores data and manipulates it using *bits* whereas a quantum computer stores and operates on *qubits* [1].

Naturally, the reader's next question might be what the difference is between a bit and a qubit. This is the where the story of quantum computing starts. A classical bit is defined by the states it can inherit, specifically it can only exist in one of two distinct states often referred to as state "0" and state "1" [2]. On the other hand, a qubit is able to exist in one of two distinct orthogonal states [1], but quantum superposition allows the qubit to also exist in a mixture of these states. Interestingly qubits can exist in a complex mixture of these states, that is to say that a qubit in superposition's state might hold imaginary amounts of the classical states "0" and "1". The reader may find it easier to understand this distinction in the matrix notation presented below, a notation which is well explained by Nakahara and Ohmi [3].

$$\text{Bit states "0": } \begin{bmatrix} 1 \\ 0 \end{bmatrix} \text{ "1": } \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{1.1}$$

$$\text{Qubit state } \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} \text{ ,Where } c_i = a_i + jb_i \text{ and } j = \sqrt{-1} \tag{1.2}$$

It is important to acknowledge that the difference between bit's and qubit's representations leads to an important result. A qubit can be in a mixture of two states at once, an effect called superposition. For example if a qubit is in the state $\begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$ it is effectively half way between classical states "0" and "1". If we extend this to a quantum register, a collection of n qubits, the register can exist in a superposition of 2^n states [4]. Herein lies the paradigm shift that quantum computing provides, if we prepare our qubits in the correct manner and we are able to perform operations on a quantum register of length n we could effectively be doing 2^n calculations in the time it takes to do a single operation. This opens up an exponential acceleration in processing if we are able to write algorithms that can take advantage of quantum computing.

1.1 Quantum Computer - High level overview

It is important to acknowledge that quantum computers thus far have been designed with a classical control computer. This classical computer provides a host of command and control functions for the quantum computer.

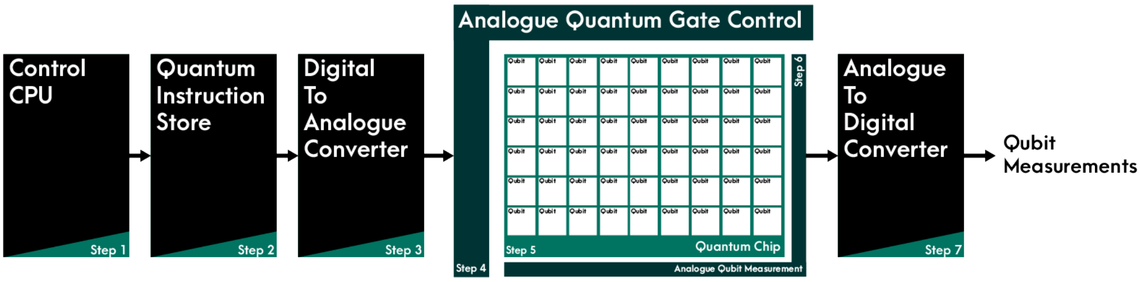


Figure 1.1: High Level diagram of Quantum Computer

Figure 1.1 provides a high level overview of a quantum computing system. For now we cannot use quantum computers as a standalone device, they rely on classical CPU's and storage for command and control. However, it is not impossible to think that in the future we will be able to develop quantum computers with entirely quantum command, control and storage operations.

1.1.1 Operation of a quantum computer

We will now describe and explain the steps presented in Figure 1.1.

- **Step 1**: Classical Control computer is instructed to send a specific set of quantum instructions to the quantum chip. The classical CPU fetches classical representations of the quantum commands from it's memory. This information is sent to a quantum instruction store.

- **Step 2**: Once the classical CPU has sent the required quantum operations to the quantum instruction store, the store now contains the queue of quantum operations required. At every time-step the store will release the next instruction to the Digital to Analogue Converter.

- **Step 3**: Since the qubit control devices in all qubit representations are effectively analogue in nature, we need to convert from our digital control instructions into analogue control signals for quantum gate devices. The Analogue to Digital converter (ADC) performs this task and signals the Analogue Quantum Gate Controls with signals to manipulate the quantum chip.

- **Step 4**: The exact mechanics that govern qubit control vary between different realisations of quantum computers (see Section 3.2). The overall process is very similar, apply control signals to the qubit array on the quantum chip in pre-defined gate patterns. The realisation of these control signals also varies by the realisation of the quantum computer.

- **Step 5**: The quantum chip consists of a grid of qubits. Each qubit is individually addressable by the quantum gate control from Step 4. The quantum chip's qubits hold quantum data which can be manipulated by the gate control until the qubits are measured upon which their quantum information collapses into classical states. Furthermore, as described in Section 3.2, these qubits are extremely susceptible to interference from the environment.

- **Step 6**: As per the original instructions from our classical control CPU, we may want to measure certain Qubits. This measurement collapses the superposed quantum state of the qubit into a classical state.

- **Step 7**: We can read this classical state and using an Analogue to Digital converter (ADC) we are able to acquire a digital representation of the information from the measured qubit. If the instructions loaded into the Classical Control CPU memory was a quantum algorithm, the acquired digital readout will provide the results of the algorithm.

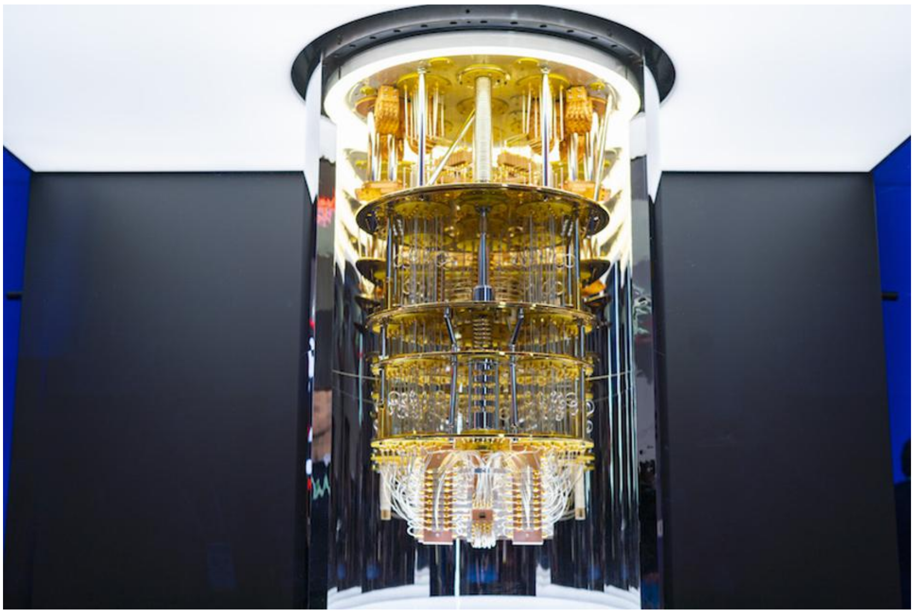


Figure 1.2: Model of an IBM Quantum Computer [5]

1.2 Mathematical formulation of Quantum Computing Operations

Quantum computation can be described by a series of matrix calculations. Furthermore, Bra-Ket notation and Bloch spheres aid us in performing mathematical analysis of quantum operations. Appendix A provides a brief overview of the physics of superposition.

1.2.1 Bra-Ket notation

Bra-Ket notation provides an encapsulated way of representing states. Introduced by Paul Dirac in 1939 [6], this notation is used widely in quantum computing. It provides a mathematical framework for representing quantum states which helps us differentiate between them when doing quantum mathematics.

For example the classical state "0" can be represented in Bra-Ket notation as in Equation 1.3.

$$|0\rangle \tag{1.3}$$

Whereas the classical state "1" can be represented by Equation 1.4.

$$|1\rangle \tag{1.4}$$

Combining equations 1.3, 1.4 and 1.1 we can express the equivalence between Bra-Ket notation and vector representations of bits in Equation 1.5.

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{1.5}$$

This notation allows us to see the effects of quantum superposition more clearly, as in Equation 1.6 a qubit can be represented as a mixture of classical states in bra-ket notation.

$$\text{Qubit state: } \frac{|0\rangle + |1\rangle}{\sqrt{2}} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} \tag{1.6}$$

1.2.2 Bloch Sphere

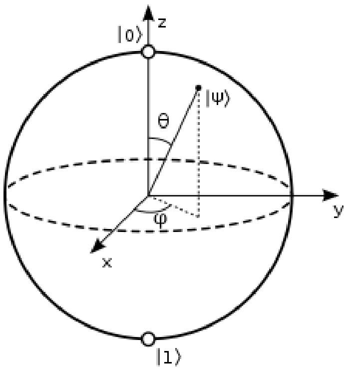


Figure 1.3: Diagram of a Bloch Sphere [7].

The Bloch sphere was introduced by Felix Bloch in 1946 [8]. It provides a uniquely geometric way of representing Qubit states. The Bloch sphere is a unit sphere, and it's center represents the origin of 3-orthogonal axes. Qubit states are represented as vectors that lie on the surface of the Bloch sphere. We can see in Figure 1.3 that the states $|0\rangle$ and $|1\rangle$ lie on opposite poles of the z-axis of the bloch sphere.

To explain the remainder of the Bloch sphere we must reformulate our qubit representation, the explanation presented here is adapted from Noson and Mirco [1]. As discussed in Section 1 qubits can exist in a superposition of these classical states. This is where the two remaining axis of the Bloch sphere play their part. In Figure 1.3 there is a general qubit $|\psi\rangle$ demarked by a line originating at the origin and terminating on the surface of the Bloch sphere . This vector represents a qubit state, and has the associated angles θ and ϕ. To calculate these angles we return to the qubit representation given by Equation 1.7.

$$\begin{bmatrix} c_0 \\ c_1 \end{bmatrix} \tag{1.7}$$

We can manipulate this representation to separate the qubit into components of the $|0\rangle$ and $|1\rangle$ states, as shown in Equation 1.8.

$$\begin{bmatrix} c_0 \\ c_1 \end{bmatrix} = c_0 \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} + c_1 \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{1.8}$$

We can use Equation 1.5 to convert Equation 1.8 into Bra-Ket notation.

$$c_0 \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} + c_1 \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} = c_0 |0\rangle + c_1 |1\rangle \tag{1.9}$$

In Equation 1.9 we have separated out a general qubit state into a superposition of classical Bra-Ket states with complex coefficients. We remind ourselves that complex numbers can be broken down into a polar representation as given by Equation 1.10.

$$c_i = a_i + jb_i = r_i e^{j\phi_i} \text{ where } r_i = \sqrt{a_i^2 + b_i^2}, \phi_i = \tan^{-1}\frac{b_i}{a_i} \tag{1.10}$$

Equation 1.9 can now be rewritten in the form of Equation 1.11.

$$|\psi\rangle = r_0 e^{j\phi_0} |0\rangle + r_1 e^{j\phi_1} |1\rangle \tag{1.11}$$

As noted by Noson and Mirco, this representation contains 4 parameters, which would require 4 dimensional diagram to visualise. To reduce the dimensionality such that we can visualise it we will attempt to move from absolute phases ϕ_0 and ϕ_1 to relative phase $\phi = \phi_1 - \phi_0$ [1].

$$e^{-j\phi_0}\left|\psi\right\rangle = r_0\left|0\right\rangle + r_1 e^{j(\phi_1-\phi_0)}\left|1\right\rangle = r_0\left|0\right\rangle + r_1 e^{j\phi}\left|1\right\rangle \tag{1.12}$$

Equation 1.12 provides a relative phase representation of a qubit state $\left|\psi\right\rangle$. We should note that the absolute phase $e^{-j\phi_0}$ has no physical meaning in terms of the $\left|0\right\rangle$, $\left|1\right\rangle$ basis. So we can drop this phase offset to form an equation with three parameters in Equation 1.13.

$$\left|\psi\right\rangle = r_0\left|0\right\rangle + r_1 e^{j\phi}\left|1\right\rangle \tag{1.13}$$

We observe that since all qubit states $\left|\psi\right\rangle$ have a modulus of 1 then $|c_0|^2 + |c_1|^2 = 1$.

$$1 = |c_0|^2 + |c_1|^2 = |r_0|^2|e^{j\phi_0}|^2 + |r_1|^2|e^{j\phi_1}|^2 \tag{1.14}$$

We further observe that $|e^{j\phi}|^2 = 1\ \forall\phi \in \mathbb{R}$, therefore we have $r_0^2 + r_1^2 = 1$. Therefore we can perform an algebraic trick in Equation 1.15.

$$r_0 = \cos(\theta) \text{ and } r_1 = \sin(\theta) \tag{1.15}$$

We now have a complete formulation of the qubit state $\left|\psi\right\rangle$ in just 2 parameters as given by Equation 1.16.

$$\left|\psi\right\rangle = \cos(\theta)\left|0\right\rangle + e^{j\phi}\sin(\theta)\left|1\right\rangle \tag{1.16}$$

We have arrived at the θ and ϕ in the Bloch sphere diagram presented by Figure 1.3. With this visualisation and derived parameters, we can represent operations on qubits as manipulations of the Bloch sphere. This is most vividly presented in Section 1.2.4 with Pauli gates.

1.2.3 Matrix formulation

Let us assume that we have a classical bit in state $\left|0\right\rangle$ and our classical computer wanted to perform a NOT operation on this bit. We know the expected result of this: $NOT(\left|0\right\rangle) = \left|1\right\rangle$. We can formulate a matrix to perform this NOT operation on a vector representation of classical bits, such as in Equation 1.18.

$$NOT : \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{1.17}$$

Taking the representation of classical bits given in Equation 1.5. We can see the effect of a NOT gate on the $\left|0\right\rangle$ state.

$$NOT(\left|0\right\rangle) : \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{1.18}$$

1.2.4 Common Quantum Computing Gates

Now that we have a formulation of gates as matrix operations on vector representations of bits and qubits, we can explore some crucial quantum computing gate operations. It is important to note here, that all quantum gates, and by extension all quantum computing operations must be reversible [1].

Hadamard Gate

The Hadamard gate serves an important purpose, it allows us to bring a qubit into and out of superposition.

$$\text{Hadamard Gate} \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} \tag{1.19}$$

We can see it's affect on a qubit in state $|0\rangle$ in Equation 1.20. The result of this operation is in an equal superposition of the states $|0\rangle$ and $|1\rangle$.

$$\begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} \tag{1.20}$$

Pauli Gates

Pauli gates appear quite often in quantum computing and quantum mechanics in general [3]. There are three Pauli gates, labelled X, Y and Z, these labels are references to their affects on the orientation of qubits that are in a *Bloch Sphere* as explained by Noson and Mirco [1] and covered in Section 1.2.2.

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \tag{1.21}$$

Note that the Pauli X gate is nothing other than the NOT gate of Equation 1.18. Each gate is named for the axis of the Bloch sphere that the gate rotates the qubit along.

Universal Rotation Gate

The universal rotation gate is featured as the only required single qubit gate for a universal gate set [9]. And provides flexible rotation based on three parameters θ, ϕ and λ. Given a **U** matrix with these parameters, the gate matrix will take the form presented in Equation 1.22.

$$U(\theta, \phi, \lambda) = \begin{bmatrix} cos(\frac{\theta}{2}) & -e^{i\lambda} sin(\frac{\theta}{2}) \\ e^{i\theta} sin(\frac{\theta}{2}) & e^{i\lambda + i\phi} cos(\frac{\theta}{2}) \end{bmatrix} \tag{1.22}$$

Any unitary single qubit gate can be constructed by varying θ, ϕ and λ in this **U** gate.

CNOT Gate

The Controlled-Not gate (CNOT) provides an important function for quantum programmers, it allows us to put two qubits into a state of quantum entanglement. In this state of entanglement, both qubits exist in a superposition of the $|0\rangle$ and $|1\rangle$ as is usual for qubits, however as soon as one of the qubits is measured we can calculate the state which the other qubit will settle in with 100% accuracy. This entanglement has opened the door to quantum cryptographical techniques [10].

The CNOT gate is a two qubit gate, which means we will need to first represent the 2 qubits going into this gate as a single vector. This is done by taking the tensor product of the two qubits. For example if we have a qubit in the state $|\psi\rangle$ and another qubit in the state $|\phi\rangle$ we can take their tensor product as in Equation 1.23.

$$|\psi\rangle \otimes |\phi\rangle = \begin{bmatrix} c_0^\psi \\ c_1^\psi \end{bmatrix} \otimes \begin{bmatrix} c_0^\phi \\ c_1^\phi \end{bmatrix} = \begin{bmatrix} c_0^\psi \cdot \begin{bmatrix} c_0^\phi \\ c_1^\phi \end{bmatrix} \\ c_1^\psi \cdot \begin{bmatrix} c_0^\phi \\ c_1^\phi \end{bmatrix} \end{bmatrix} = \begin{bmatrix} c_0^\psi c_0^\phi \\ c_0^\psi c_1^\phi \\ c_1^\psi c_0^\phi \\ c_1^\psi c_1^\phi \end{bmatrix} \tag{1.23}$$

As shown by equation 1.23, the combined two qubits share a 4 dimensional vector space, such a space will need to be operated on by a 4×4 matrix. It is important to note what states the output vector of this calculation represents as shown in Equation 1.24. These entries represent combined states and so will be important when understanding the results of the CNOT operation.

$$\begin{array}{c} |00\rangle \\ |01\rangle \\ |10\rangle \\ |11\rangle \end{array} \begin{bmatrix} c_0^\psi c_0^\phi \\ c_0^\psi c_1^\phi \\ c_1^\psi c_0^\phi \\ c_1^\psi c_1^\phi \end{bmatrix} \tag{1.24}$$

As implied by the name, the CNOT gate performs a controlled not operation, meaning if the control qubit ($|\psi\rangle$ for us) is in the $|1\rangle$ state it will effectively do a NOT (Pauli-X) operation on the second qubit ($|\phi\rangle$).

Equation 1.25 presents the CNOT gate as a 4×4 matrix which can operate on the combined $|\psi\rangle \otimes |\phi\rangle$ state.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{1.25}$$

For example if we have $|\psi\rangle = |1\rangle$ and $|\phi\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}}$, we would have the matrix multiplication shown in Equation 1.26.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 0 \\ \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} \tag{1.26}$$

We can see clearly in Equation 1.26 that the state of our second qubit has been inversed. However, we can see a more interesting result if we pass two qubits in state $|\psi\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}}$ and $|\phi\rangle = |0\rangle$ into the CNOT gate.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ \frac{1}{\sqrt{2}} \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ \frac{1}{\sqrt{2}} \end{bmatrix} = \frac{|00\rangle + |11\rangle}{\sqrt{2}} \tag{1.27}$$

While the result of Equation 1.27 seems just as mundane as Equation 1.26, on closer inspection we see that we have achieved entanglement. This is because our new 2-qubit state is $\frac{|00\rangle + |11\rangle}{\sqrt{2}}$, this means that if the first qubit is measured to be in state $|0\rangle$ then the second qubit must necessarily collapse into $|0\rangle$ at that very instant, there is no other option for it as the 2-qubit state only contains one entry for the first qubit being in state $|0\rangle$. The same conclusion can be drawn for the first qubit being in state $|1\rangle$ necessitates the second qubit collapsing into state $|1\rangle$. This relationship holds for any distance between the two qubits.

Swap Gate

The final gate I will provide a short overview is the swap gate. This gate allows us to swap the quantum information between two qubits. It is represented by the 4×4 matrix in Equation 1.28.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.28}$$

The swap gate's operation is illustrated clearly by Equation 1.29.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{array}{c} |00\rangle \\ |01\rangle \\ |10\rangle \\ |11\rangle \end{array} \begin{bmatrix} c_0^\psi c_0^\phi \\ c_0^\psi c_1^\phi \\ c_1^\psi c_0^\phi \\ c_1^\psi c_1^\phi \end{bmatrix} = \begin{array}{c} |00\rangle \\ |01\rangle \\ |10\rangle \\ |11\rangle \end{array} \begin{bmatrix} c_0^\psi c_0^\phi \\ c_1^\psi c_0^\phi \\ c_0^\psi c_1^\phi \\ c_1^\psi c_1^\phi \end{bmatrix} \tag{1.29}$$

1.3 Current challenges in quantum computing

Quantum computers have been theorised since the early 1980's, when Paul Benioff proved mathematically that it was possible to build a Turing machine [11] using quantum Hamiltonian operations [12]. Experimental physicists have since been attempting to construct physical implementations of these quantum machines, with the first successful attempt completed by Isaac Chuang and Neil Gershenfeld in 1998 [13].

However, those early attempts confirmed a troublesome theoretical prediction, the phenomenon of *decoherence*. This phenomenon arises from the coupling of a quantum system to it's environment, leading to non-unitary contributions to the system's state that is eventually measured [14]. In essence, a quantum computer's information is in such a fragile state that the environment, any background electromagnetic or nuclear processes, necessarily couples to the quantum information changing it in a irreversible way. This corrupts the data and "destroys the quantum computation" [14].

In the time since those early quantum circuits, we have developed technologies that are more resistant to *decoherence*, a complete review of this can be found in section 3.2. However, we are unable still to create quantum circuits that are stable on the timescales required to unlock the quantum computer's full potential. This phenomenon of decoherence is, by far, the greatest hurdle for the growing community of quantum computing researchers.

1.4 Quantum computer emulation

While the experimental side of the quantum computing cohort grapple with *decoherence*, the theoretical group are developing algorithms designed to run on future fault tolerant quantum hardware. A review of some of the most consequential of these algorithms can be found in section 3.3.

To test the feasibility of these algorithms computer scientists have developed a wide range of software emulation tools for quantum computers. Notable developments such as IBM's Quantum Experience [15] offer free emulation tools for the general public. However, many of these tools have been developed for researchers in quantum computing and are quite difficult to pick up for newcomers to the field. Furthermore, these tools often do not fully utilise the latest developments in computer hardware especially the power of GPU's to parallelise linearly independent tasks.

It will be these shortcomings which I wish to tackle in this final year project, the creating of an easy to use interface for newcomers to quantum computing, with a powerful GPU accelerated backend that pushes the limits of quantum computer emulation on enthusiast grade hardware.

1.5 Quantum Computer programming languages

As we move towards more robust quantum computers and more advanced emulators, we need to consider how to interface with these machines. There are a number of released quantum programming languages such as OpenQASM [9] and Quil [16]. These languages are designed to be counterparts for classical instruction set architectures such as x86 and ARMv8 [17].

1.5.1 OpenQASM

OpenQASM is an important component of this project. OpenQASM is an instruction set architecture for quantum computers. It was developed by a team at International Business Machines (IBM) lead by Andrew Cross [9]. Just as classical processors use NAND gates as their fundamental gate set [18], OpenQASM uses the Hadamard gate and CNOT gate which are mentioned in Section 1.2.4. We are able to combine these two gates together to produce any quantum computing operation that we would like [1].

OpenQASM is an instruction set based on defining registers and commanding gate operations on qubits in those registers. For example:

```
qreg q[3];
h q[0];
```

```
    cx q[0], q[1];
```

This code defines a quantum register called "q", it applies a hadamard gate to qubit "0" of register "q". Then applies a CNOT gate between qubit "0" and "1" of register "q". This simple structure is the core reason why we have chosen the OpenQASM standard as the input standard which our quantum computer simulator will use.

1.6 Summary of report

In this report we discuss our project to build a quantum computer emulator called Valkyrie and a development environment called Visual-Q. We will discuss how our initial implementation of Valkyrie was performant in low complexity circuits but became very slow in high complexity circuits. We then detail the optimisations we made, and how these optimisations made Valkyrie faster than it's contemporaries across all complexities tested.

1.6.1 Objectives

Section 2 summarises the core objectives of this project. This section will how we aim to create a faster quantum computer emulator and a simple integrated development environment for this emulator to be accessed from.

1.6.2 Literature Review

The Literature Review in Section 3 is an indepth look into the research surrounding quantum computing. This section reviews the fundamentals of quantum processes and how they gave arise to the field of quantum computers. We also review the current field of quantum computer emulation and how we can leverage these ideas to build a faster quantum computer emulator.

1.6.3 Design

The Design section (Section 4) gives a high level overview of how we will build a quantum computer emulator and supporting development environment. We also briefly review what core modules we will need to develop to achieve this goal.

1.6.4 Quantum Computation by classical simulation

Section 5 introduces to the reader how we can achieve quantum calculations by using linear algebra to complete matrix calculations. This section also shows how the interpretation of results is just as important as the computation itself.

1.6.5 Implementation

Section 6 covers the implementation of the designs mentioned in Section 5. This implementation section covers in deep detail the algorithms and data-structures required to achieve our objectives.

1.6.6 Evaluation

Section 7 covers the experimentation that we have completed to test and evaluate the performance of our quantum computer emulator. In this section we cover how our emulator competes with other emulators and how performance deficit's that we had in the development phase of this project can be measured and how the changes that we made to address our performance deficits have improved our execution times.

1.6.7 Future Development and Conclusion

Section 8 briefly covers the steps that can be taken in the future to further improve the performance of our quantum computer emulator. Finally, we conclude with a consideration of what we have achieved and how we have achieved it.

1.6.8 Appendices

- Appendix A: A brief rundown on the physics and mathematics of superposition, a core concept for understanding quantum computing.
- Appendix B: The C++ Valkyrie codebase in full, with captions which describe filenames as well as annotated functions. The codebase is also hosted at this repository.
- Appendix C: The javascript codebase for Visual-Q cotnaining operative code. The full codebase can be seen at this repository.
- Appendix D: Evaluation data for all experiments we have run as part of Section 7.

Chapter 2

Objectives

Summary

In this section we cover the core objectives of our project, how we will create a quantum computer emulator called Valkyrie. Furthermore, we will discuss how to build a development environment that can leverage this quantum computer emulator to allow users to easily execute OpenQASM programs.

Newcomers to the quantum computing field have a number of packages to explore, see Section 3.4, however quite often the knowledge needed to use these packages is difficult to find and understand, furthermore software emulators for home use do not utilise the latest advances in consumer hardware to best effect. My project aims to address both of these issues and I have formulated two objectives to formally define the aims of this project.

2.1 GPU Accelerated Quantum Circuit Emulation

While both Qiskit [19] and Google Cirq provide quantum computer emulation neither fully utilise consumer grade hardware to it's full extent. Qiskit's implementation uses a generic python wrapped GPU library that doesn't use some of the the advanced hardware features now available to consumers. We hope to leverage developments in consumer grade GPU hardware such as the Ampere architecture [20] to accelerate quantum computation calculations.

I have decided that my first objective would be to:

- Build a Quantum Computer Software Emulator specifically targetting consumer grade hardware, particularly Nvidia Ampere Architecture GPU's

2.2 Integrated Development Environment for Quantum Programmers

As discussed, while there are packages that make it easy for quantum programmers to build quantum circuits the resources required for learning and visualisation of these circuits are not centrally provided. It can be quite difficult for newcomers especially to understand where and what packages to use together to provide a full quantum programming experience.

Considering this, my second objective will be to:

- Build an integrated development environment allowing users to program the aforementioned software emulator, and visualise their results.

Chapter 3

Literature Review

Summary

In this section we will review the surrounding research on quantum computing. Furthermore, we also review the fundamental quantum concepts that enable us to exploit these quantum computing properties. In additional we also consider some of the algorithms that promise to help revolutionise the field of high intensity computing.

Quantum Computing is a rapidly evolving field with new publications released on an almost daily basis. In this literature review the reader will be given a high level overview of quantum computing and a deeper analysis of crucial developments in the field.

3.1 Quantum Computing

The theorem that one could build a quantum information processor that could perform computation as per the church-turing thesis [11] underpins quantum computing. This theorem was proven by Benioff, in his seminal 1980 paper [12], which concludes that for any Turing machine Q there exists a Hamiltonian H_N^Q and a class of initial states c such that the evolution of the Hamiltonian on an initial state in c can be used to model the computational steps of Q. What Benioff had not considered in this paper is the concept that such a quantum machine could perform calculations that a classical computer could not. His quantum machines were scoped to only model the computations available to Turing machines. The quantum computer would go on to be proven to complete calculations that were infeasible on classical computers. The structure of qubits and the nature of superposition allows quantum computers to expand the set of computable operations beyond that of the church-turing thesis. In the time since Benioff's 1980 paper, many significant contributions to the field of quantum computing have opened our collective imaginations to the possibilities enabled by quantum computing.

Exploration of the ability of quantum information processors to perform calculations inaccessible to classical computers started as a single sentence in Richard Feynman's "Quantum Mechanical Computers" [21]. In this paper Feynman formalises much of the work that was started by Benioff and in the paper's conclusion Feynman postulates that reversible quantum systems may be able to gain speed by "concurrent operation *(on their qubit states)*" . In this single tantalising sentence Feynman blew the starting whistle on the race to discover algorithms that could perform operations on quantum computers that would not be feasible on a classical computer. A review of some of these algorithms can be found in section 3.3. Feynman's key insight does fall slightly short of the concept of *Quantum Supremacy*, a term used often by modern media to describe the anticipated capabilities of quantum computers. On the other hand, David Deutsch in 1985 provided us with an elaboration on the capabilities of quantum computers [22]. In this paper Deutsch describes a novel quantum algorithm, *Deutsch Algorithm* that would utilise Feynman's "concurrent operation" to achieve an efficiency that could not be matched by a classical computer.

3.2 Realisation of Quantum Computers

We will take a break from the theoretical side of quantum computing and now pay some attention to the all important practical implementation of these exotic information processors. To implement a quantum information processor, a system simply needs to be able to hold qubits of information and provide a pathway to manipulate these qubits. Equation 1.2 provides blueprints for what information a physical qubit must be able to encapsulate.

The realisation of our theoretical qubits is a huge experimental challenge, requiring new experimental techniques and important hardware advances to build a quantum computer. The first successful attempt at building such a machine was made by Neil Gershenfeld and Isaac L. Chuang and reportedly used a tabletop arrangement of magnets and fluid [23].

Figure 3.1: Image of Chuang and Gershenfeld's tabletop Quantum Computer (from [23])

One may observe there to be a stark contrast between Figure 3.1 and one of IBM's newest machines in Figure 1.2. These two machines are of different technologies and precision. While Chuang and Gershenfeld's machine was a 2 qubit system relying on nuclear magnetic resonance, IBM's latest efforts include a 53 qubit quantum computer. Google has been developing a quantum computer that holds 72 qubits [24], although it is based on a different technology to IBM's quantum computers.

Before we explore the technologies that are being used to realise quantum computers, let us firstly examine the literature surrounding an important distinguishing feature between these technologies.

Decoherence

Decoherence is a major challenge for quantum computer engineers, recent advances in hardware and quantum error correction have illuminated a path towards a *fault tolerant* quantum computer. Steane in his 1998 review of quantum computing provides an clear and concise explanation of decoherence [14]. Steane models decoherence as the interaction between a quantum system Q and the environment T resulting in $Q^{'}$, since T is in general a non-unitary addition to Q the system $Q^{'}$ is no longer unitary. The loss of unitarity in $Q^{'}$ means any information that it holds has been corrupted as the system cannot be reversed, an essential condition assumed in Benioff's postulation [12]. This explanation of decoherence in Steane's review is a beautifully simple explanation of an extremely complex interaction. On the other hand, Steane's 1998 review falls short of quantifying this effect in a summarised fashion, this is to be expected since at the time of it's writing not many quantum computers had been succesfully built and decoherence wasn't experimentally understood.

For a better understanding of the contributing factors of decoherence the reader may want to review Resch and Karpuzcu's "Quantum Computing: An Overview Across the system stack" [25]. In this more recent review more experimental data has been collected on the various realisations of quantum computers and their decoherence charactersistics. This review in particular goes into great depth in explaining the experimental statistics that have been developed over time to compare and contrast the decoherence characteristics of various physical realisations. It would be a good paper to read for any newcomers to the field of quantum computing.

3.2.1 Nuclear magnetic resonance

Nuclear Magnetic resonance (NMR) is a technology that is used extensively in the field of medicine for the purposes of magnetic resonance imaging (MRI). It has been picked up by the quantum computing community as a possible way to realise a quantum computer, in fact it was the method used to create the first quantum computer [23] which can be seen in Figure 3.1. Gernshenfeld and Chuang go into a good amount of detail on the use of NMR to realise qubits, considering the experimental nature of their article. A more indepth look into NMR for quantum computation is provided by J.A. Jones [26]. In Jones' review the physical implementation is provided in exquisite detail, however the only shortcoming is the age of the text. It doesn't cover the latest experimental techniques and the improved decoherence characteristics that has provided.

We will turn to Resch and Karpuzcu for the latest data on the decoherence characteristics of NMR quantum computers [25]. In this paper, NMR can maintain qubit coherence for the longest time period with coherence times lasting around 16.7 seconds, however NMR also has very slow 'Gate latency', which limits the number of operations that can be performed every second. As a consequence of this we see that current NMR quantum computers can only perform approximately 185 operations on qubits before decoherence causes the qubits to lose their quantum information. This is the least number of feasible operations of the technology space reviewed by Resch and Karpuzcu [25].

3.2.2 Ion Trap

Ion trap's are an interesting implementation of qubits, by holding a string of charged atoms in a linear ion trap [14]. Steane goes on to explain that each ion is addressed by a pair of laser beams (see Figure 12 in [14]) and the same lasers are used to cool the ions in state preparation. One must admire the futurism presented by this realisation of quantum computers. Resch and Karpuzsu are quick to point out that while Ion Traps have long coherence times they suffer from the same gate delay time issues as NMR leading to just 192-196 feasible quantum operations for a full system [25]. However, Resch and Karpuzsu also point out that of all realisations of quantum computers, Ion Traps are the only option where the system can be mobile and who's stability is not overly reliant on the state of motion of the system. This is an important consideration for the future of quantum computing, while quantum laptops are still a while away the mobility of ion traps open up possibilities of reasonable even commercially available quantum computers.

3.2.3 Quantum Dot

The final physical realisation I will review in detail is Quantum Dot based qubit realisations. This technology allows us to return to firmer ground in terms of materials in the form of silicon semiconductors. Daniel Loss provides a detailed account of quantum dot based computation in "Quantum Computation with quantum dots" [27]. Loss provides a strong mathematical analysis of quantum dots and how they can be used to realise qubits, however many might find this paper hard to approach without a strong understanding of quantum dots. One of the strengths of this paper is it's outlining of the implementations of quantum gates from the hardware perspective, the reader might find themselves better understanding some other technologies gate implementations once they've read Loss's descriptions. Returning to Resch and Karpuzsu's 2019 review quantum dot's have very short decoherence times on the order of microseconds, however the fast switching gates (as described by Loss) of Quantum Dot quantum computers mean that of all the technologies in Resch and Karpuzsu's review qauntum dot's are able to achieve the greatest number of operations before decoherence, between 225 and 200 [25].

3.3 Quantum Algorithms

We have now conception of both the promise and difficulty of practical quantum computers. Let us now review the algorithms that have so far been devised for quantum computers. The development of quantum computer algorithms starts with Deutsch in 1985 and had continued ever since, with particularly notable contributions made by Peter Shor in 1994. It must be noted that Noson and Mirco provide a comprehensive explanation of these algorithms [1] and I have found their analysis the most useful tool in understanding these quantum algorithms.

3.3.1 Deutsch's Algorithm

Deutsch presents his algorithm in his seminal 1985 paper [22], it's function seems a little contrived but it provides an important stepping stone to more complex functionality. In essence Deutsch's algorithm concerns itself with calculating whether a function is "balanced" or "constant" as explained by Noson and Mirco [1]. Deutsch's 1985 paper provides a mathematically rigorous approach to explaining the function of this algorithm, for those who might find themselves daunted by the mathematics in that paper Noson and Mirco provide a step by step explanation of the algorithm. In essence, Deutsch's algorithm allows one to analyse a function which takes a single bit input and provides a single bit output.

$$f : \{0, 1\} \longrightarrow \{0, 1\} \tag{3.1}$$

The function is called "balanced" if it's output changes when the input bit changes, and "constant" if it's output doesn't change regardless of input [1]. One might imagine a classical computer having to run this function twice, once on the input of 0 and once on an input of 1 to work this out, however Deutsch's algorithm manages to do this in one step. The reader may feel a little confused, as it seems that this would be impossible, we must remind ourselves that when working with qubits we can operate on a superposition of states, allowing us to consider the results of both 0 and 1 being fed to the function being analysed.

3.3.2 Deutsch - Jozsa Algorithm

Working with Richard Jozsa in 1992, Deutsch extended his previous algorithm into a more general form [28] which could analyse functions of the form:

$$f : \{0, 1\}^n \longrightarrow \{0, 1\} \tag{3.2}$$

Deutsch and Jozsa provide a detailed and mathematically rigorous explanation of their algorithm which is able to calculate whether a function of the type described in Equation 3.2 is "balanced" (exactly half the inputs go to 0 and other half go to 1) or "constant" (all inputs go to 0 or all inputs go to 1) [28]. Noson and Mirco provide a more approachable explanation to the operation of this algorithm, which once again hinges on the principle of superposition to concurrently calculate multiple entries to the function at once. We are finally making good on Feynman's promise of "concurrent operation" of these quantum computers. As noted by Noson and Mirco we observe an exponential speed up from a classical machine which would require $2^{n-1} + 1$ evaluations to calculate the result that a quantum computer running the Deutsch-Jozsa algorithm could provide in a single calculation [1].

3.3.3 Simon's Periodicity Algorithm

Simon's Periodicity algorithm [29] is a powerful algorithm which allows us to analyse functions of the form presented in 3.4 [1].

$$f\{0, 1\}^n \longrightarrow f\{0, 1\}^n \tag{3.3}$$

$$f(x) = f(y) \text{ if and only if } x = y \oplus c \tag{3.4}$$

Simon's periodicity algorithm is used to calculate the period c. Noson and Mirco provide a worked example of Simon's algorithm, which they note provides an exponential efficiency increase over the classical method of finding periodicity [1]. This algorithm is also used in Shor's Factoring Algorithm, and is more intuitive to understand that Deutsch Jozsa.

3.3.4 Shor's Factoring Algorithm

We have now seen an example of how a quantum algorithms can achieve exponential speedup over classical approaches. It would be pertinent to review the most famous quantum algorithm to date, another algorithm which achieves exponential speedup over it's classical contemporaries; Shor's factoring algorithm.

Proposed by Robert Shor in 1994, Shor's factoring algorithm achieves the seemingly impossible, it is able to factor a general integer N in polynomial time [30]. The reader may exclaim that such an algorithm would have the ability to break a large proportion of the cryptography techniques in use today. This includes the widespread RSA schema used for internet security [31]. Shor's algorithm is a complicated algorithm utilising quantum Fourier transforms and large assemblies of quantum gates, Nolson and Mirco provide a step by step breakdown of Shor's algorithm, providing newcomers an approachable path to understanding Shor's revolutionary algorithm.

The fastest classical implementation of large integer factorisation is the general number field sieve [32]. Which has an overall exponential complexity, which for large numbers such as those used by RSA can take hundreds of years to factorise. However, Shor's algorithm is able to factorise such numbers in an hour or two using it's polynomial complexity.

The consequences of Shor's algorithm for the future of quantum computing is profound. Before quantum computers become commercially available we will need to develop security algorithms that are resistant to quantum computer attack and move the internet to these new algorithms. Furthermore, quantum information provides a solution to this in the form of Quantum Cryptography which is not explored in this project but a good source of information on both of these topics can be found in Daniel Bernstein's "Post-Quantum Cryptography" [10].

3.4 Quantum computer emulation

Now that we have gathered a strong understanding of what quantum computers are, the methods of physical realisation we are exploring and the algorithms that can utilise the concurrency provided by qubits, let us now address the topic of quantum computer emulation. Since we have not been able to build large enough quantum computers with appropriate decoherence times to support full execution of the most interesting quantum algorithms, scientists must resort to software emulators of quantum processors.

3.4.1 IBM Quantum Experience

International Business Machines (IBM) are industry leaders in the field of quantum computing and quantum software in particular. IBM have developed a series of software tools to enable the general public to access actual quantum computers for simple calculations and provide an open source software emulator for more complex calculations [15].

OpenQASM

IBM has released a low level assembly-like language named "OpenQASM" which can directly instruct one of their quantum computers [9]. This language is quite flexible and uses a built in set of universal quantum gates in the form of a single qubit 3-axis rotation gate (Equation 3.5) and a 2 qubit controlled not gate (Equation 3.6) for entanglement operations.

$$\text{Single Qubit 3-axis rotation gate: } U(\theta,\phi,\lambda) = \begin{bmatrix} \cos(\frac{\theta}{2}) & -e^{i\lambda}\sin(\frac{\theta}{2}) \\ e^{i\phi}\sin(\frac{\theta}{2}) & e^{i\lambda+i\phi}\cos(\frac{\theta}{2}) \end{bmatrix} \tag{3.5}$$

$$\text{CNOT Gate: } \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{3.6}$$

As Cross et al. present in their 2017 paper, this language allows for simple programming while also allowing for compile-time optimisations to accelerate software emulation. Their analysis of the technical advantages of this very simple universal gate set is quite detailed, however their documentation of the language itself can be a little confusing especially since they often intersperse technical language with more vague terms. Furthermore, OpenQASM provides a simple interface for compiling more complex gate structures from the two basic gates. This allows for compact programming while also providing ample room for optimisation in the compiling of the OpenQASM code.

Qiskit

IBM provide a python language compiler and associated programming API in the form of the QisKit package [19]. This package provides a good scientific programming environment for quantum researchers, furthermore the documentation for this API is excellent, providing clear and concise examples of how to use the package. It would be appropriate to remark on the gap between this simple to use documentation of the QisKit API but difficult to read OpenQASM specification, while one could argue that QisKit is the consumer facing package a counter-argument would be that a full understanding of OpenQASM and quantum gates in general is required to actually use QisKit to it's full extent. This gap forms the basis for my objective in section 2.2, since newcomers are presented with an easy to use API but with little guidance on what the function calls are for and the underlying mechanisms at play.

3.4.2 Google Cirq

Similar to IBM's Quantum Experience, Google have developed a competing package named "Google Cirq" [33]. This package focuses more on the current noisy quantum computers, and has advanced noise emulation for that purpose. Furthermore, google's documentation for Cirq is much more developed and easy to read with helpful tutorials provided. The reader may find Google's approach much more user friendly, in addition Google also allows programs written in Cirq to be uploaded to their quantum processors for experiments to be run. However, Google's Cirq infrastructure is much less open-source and they do not provide a low level assembly-like language. Furthermore, the reader might observe that the Cirq API itself seems a little more *functional* than Qiskit and for those who are unaccustomed to functional programming this might be a little off-putting.

Chapter 4

Design

Summary

In this section we discuss a high level design for our quantum computer emulator. We discuss what modules we will need to develop and how we can implement these. Furthermore, we will illustrate in diagrams how the programs execution will flow. We also discuss how to develop a simple development environment for users to program our quantum computer emulator.

This project will require multiple technical components to work in unison. I believe it would be invaluable to address the design and interlinking of these components in this report. This will aid us in planning the rest of the project as well as allow for a formalisation of the scope of this project. Ultimately, it will be a technical challenge to design, build and test these components, however as mentioned in chapter 7.2 this work can lead to tangible performance improvements in quantum computer emulation.

4.1 *Valkyrie*: GPU accelerate quantum computer emulation

To address objective 2.1 I have developed an architecture for a quantum computer emulator which leverages GPU hardware to accelerate calculations. To ensure the performance of this component I have decided to write the *Valkyrie* emulator in C++. This language has many benefits for this form of work, chief among which is "near-metal" programming features with direct access to pointers and low level concurrency controls. Furthermore, C++ has a strong community and is a mature language with good documentation and a rich suite of packages to help with implementation. Finally, C++ is almost unrivalled in terms of performance and will provide a strong foundation to build a fast quantum computer emulator.

I have decided to build *Valkyrie* in phases as illustrated by Figure 4.1.

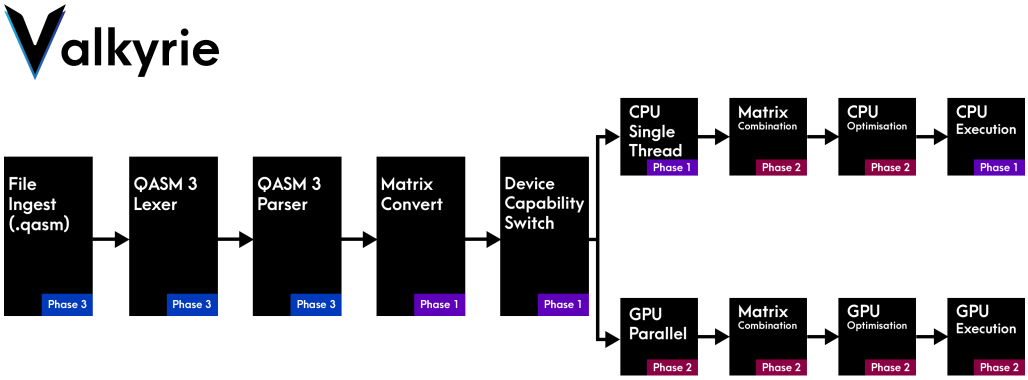


Figure 4.1: Valkyrie architecture

We will now briefly cover each component of *Valkyrie*.

4.1.1 File ingest

Valkyrie will operate by using the QASM 2 language [9] as it's input standard. This decision was made to ensure compatibility with the Qiskit API which I hope to use when running end to end simulations with *Valkyrie*. The file ingest process itself will be very simple and just require inbuilt C++ tools to correctly load the file. There will be some simple validation before the file contents are passed to the Lexer.

4.1.2 QASM 2 Lexer and Parser

I have decided to use the ANTLR [34] parser generator to build a simple QASM 2 parser generator. This package has inbuilt optimisation for parsing which will be useful when pushing the performance of *Valkyrie*. The output of the ANTLR parser will be an abstract syntax tree, which is navigable by pointers to child nodes.

4.1.3 Matrix Convert

The matrix conversion step takes the Abstract Syntax Tree produced by the QASM 2 lexer and converts the tree to a series of matrix calculations, the Abstract syntax tree visitation will be extended to provide some upstream optimisation for matrix calculations. This matrix representation will follow the principles outlined by Noson and Mirco [1].

4.1.4 Device Capability Switch

This module allows the program to discover the capabilities of the device that it is running on. This will allow the program to intelligently switch between CPU and GPU execution modes depending on whether a supported GPU is installed in the device that the program is running on. Using this information we will switch which execution pathway to run the simulation on one out of three options.

- CPU single threaded operation
- GPU Parallelised operation

4.1.5 CPU Single/Multi Thread execution

The execution is reliant on the staging process to produce a matrix representation of the quantum circuit. The execution engine is able to apply these matrices on a set of qubit vectors providing an accurate representation of the quantum states at the end. We can then use a simple measurement function to collapse the quantum information of a qubit into a classical bit.

4.1.6 GPU Execution

The GPU execution pathway will be quite different to the CPU pathway, in the GPU pathway we will attempt to parallelise as much of the calculation as possible, once we have unpacked the matrix calculations into this parallel form, we will send it to the GPU for execution. This approach will be combined with some optimisation steps, particularly to try and leverage a hardware feature called "Tensor Cores" [20] which allow for rapid matrix calculations on an Nvidia Ampere GPU.

4.2 VisualQ Quantum Programming IDE

As per objective 2.2, the second goal of this project is to provide an easy to use development environment for quantum programmers. This section will be significantly less technically challenging but would provide a lot of value to newcomers to quantum programming. I have presented the proposed technology stack for VisualQ in Figure 4.2.

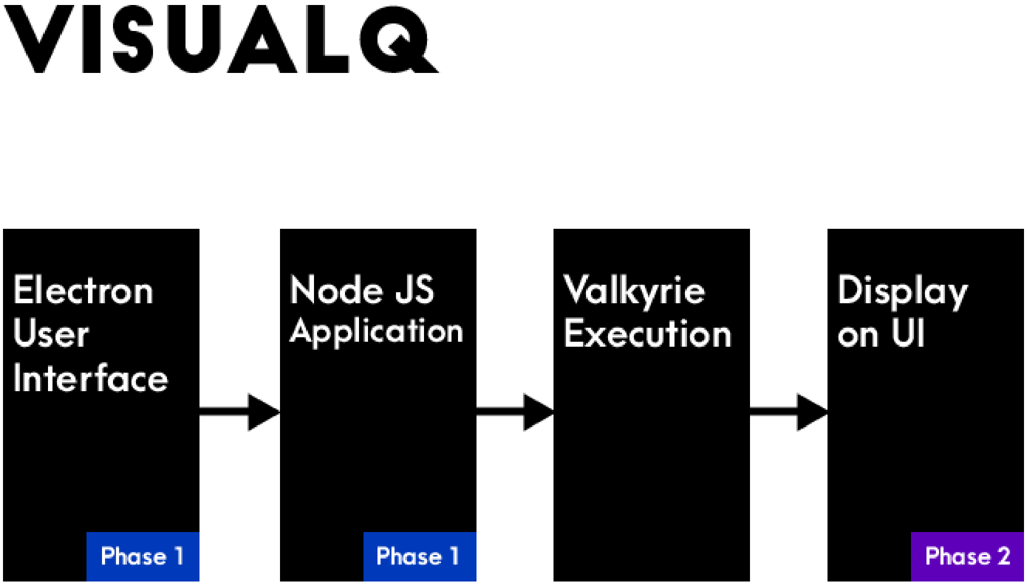


Figure 4.2: VisualQ architecture

The Visual Q stack uses mature and well established technologies which we detail in the following sections.

4.2.1 Electron user interface

The ElectronJS framework allows developers to build cross platform desktop applications using Javascript and it's numerous frameworks [35]. We can integrate electron with ReactJS which is a popular UI framework. This allows us to quickly and competently develop a user interface that is simple to use.

4.2.2 Node JS Application

NodeJS provides a framework to run javascript code in a desktop application. The NodeJS API allows us to access system functions such as the ability to run command line programs is provided by NodeJS. Since Valkyrie will be executed on the command line, the NodeJS framework is crucial to the operation of Visual Q.

Chapter 5

Quantum computation by classical simulation

Summary

In this section we discuss how we can use linear algebra simulate quantum interactions in real quantum systems. We also consider how the complexity of this linear algebra grows exponentially with circuit complexity, which will give us some context for how important optimisation will be to our quantum computer emulation.

We have introduced some of the mathematical structures we will need to build a quantum computer simulator in Section 1.2. In this chapter we will recap these structures and develop the mathematical framework we require to build a software simulator and describe and explain the code that achieves these functions.

5.1 Representing Qubits

As we have discussed, qubits are able to hold a quantum state that exists in a superposition of the classical states "0" and "1". We have seen already that this mixture of states can be represented in the form presented in Equation 5.1 [1].

$$\text{Qubit state } \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} \text{ ,Where } c_i = a_i + jb_i \tag{5.1}$$

This representation works well for one qubit, however when we consider groups of qubits, we run into a problem unique to quantum computing. Quantum circuits are built from multiple qubits, these qubits can be grouped into registers which give a bracket under which we can reference individual qubits. As seen in Equation 1.27, when we perform operations on multiple qubits they can become entangled. In this sense, to ask for the individual state of a single qubit doesn't make much sense. Instead, we must consider the ensemble state of the whole system of qubits [4]. To represent this mathematically we must take the tensor product between each individual qubit state and consider a combined state of all the qubits in our circuit. Equation 5.2 shows how we take the tensor product between two qubits.

$$|\psi\rangle \otimes |\phi\rangle = \begin{bmatrix} c_0^{\psi} \\ c_1^{\psi} \end{bmatrix} \otimes \begin{bmatrix} c_0^{\phi} \\ c_1^{\phi} \end{bmatrix} = \begin{bmatrix} c_0^{\psi} \cdot \begin{bmatrix} c_0^{\phi} \\ c_1^{\phi} \end{bmatrix} \\ c_1^{\psi} \cdot \begin{bmatrix} c_0^{\phi} \\ c_1^{\phi} \end{bmatrix} \end{bmatrix} = \begin{bmatrix} c_0^{\psi} c_0^{\phi} \\ c_0^{\psi} c_1^{\phi} \\ c_1^{\psi} c_0^{\phi} \\ c_1^{\psi} c_1^{\phi} \end{bmatrix} \tag{5.2}$$

We can extend this concept to consider the combined state of 3 qubits as shown in Equation 5.3.

$$|\lambda\rangle \otimes |\psi\rangle \otimes |\phi\rangle = \begin{bmatrix} c_0^\lambda \\ c_1^\lambda \end{bmatrix} \otimes \begin{bmatrix} c_0^\psi c_0^\phi \\ c_0^\psi c_1^\phi \\ c_1^\psi c_0^\phi \\ c_1^\psi c_1^\phi \end{bmatrix} = \begin{bmatrix} c_0^\lambda \cdot & \begin{bmatrix} c_0^\psi c_0^\phi \\ c_0^\psi c_1^\phi \\ c_1^\psi c_0^\phi \\ c_1^\psi c_1^\phi \end{bmatrix} \\ c_1^\lambda \cdot & \begin{bmatrix} c_0^\psi c_0^\phi \\ c_0^\psi c_1^\phi \\ c_1^\psi c_0^\phi \\ c_1^\psi c_1^\phi \end{bmatrix} \end{bmatrix} = \begin{bmatrix} c_0^\lambda c_0^\psi c_0^\phi \\ c_0^\lambda c_0^\psi c_1^\phi \\ c_0^\lambda c_1^\psi c_0^\phi \\ c_0^\lambda c_1^\psi c_1^\phi \\ c_1^\lambda c_0^\psi c_0^\phi \\ c_1^\lambda c_0^\psi c_1^\phi \\ c_1^\lambda c_1^\psi c_0^\phi \\ c_1^\lambda c_1^\psi c_1^\phi \end{bmatrix} \tag{5.3}$$

As the user can imagine this tensor product can be extended to any number of qubits. However, the user may also note that there is an exponential relationship between the number of qubits in a quantum circuit and the size of the combined state we must use to represent this state. That is to say if we have a circuit which involved n qubits, we will need a vector of size 2^n to represent the combined state of the circuit [36]. If we want to faithfully simulate operations on these qubits we must represent their ensemble state in a 2^n sized state vector.

5.2 Representing Quantum Gate Operations

As we have already discussed in Section 1.2.3, we can represent gate operations as matrix operations on the qubit state vectors. A simple example of this can be seen in Equation 5.4.

$$NOT(|0\rangle) : \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{5.4}$$

However, these operations become more complicated with multi-qubit states. For example if we wanted to apply this NOT gate (called the Pauli-X gate in quantum computing) to the λ qubit in the state presented in Equation 5.3, we would need to consider how the 2x2 NOT gate can be applied to the 8x1 statevector. The solution to this problem is to consider what is happening to the other qubits while we apply the NOT gate to λ [37].

While λ is undergoing the NOT operation, the other qubits can be said to have the *Identity* operation applied to them, that is their states are unchanged after the operation. We can define the *Identity* operation in matrix form, as seen in Equation 5.5.

$$ID : \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{5.5}$$

Equation 5.6 formulates the not operation applied to qubit λ.

$$NOT(|\lambda\rangle) : \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \lambda_0 \\ \lambda_1 \end{bmatrix} = \begin{bmatrix} \lambda_1 \\ \lambda_0 \end{bmatrix} \tag{5.6}$$

However to consider the evolution of the multiqubit state we must consider the formulation presented in Equation 5.7. We must also note that we can have any operation on these qubits, such as multiple NOT gates on different qubits.

$$NOT(|\lambda\rangle) \otimes ID(|\psi\rangle) \otimes ID(|\phi\rangle) \tag{5.7}$$

We apply the tensor product between the operation to mirror the tensor product used to resolve the mirror the multi-qubit state [38]. We can then combine these operations to retrieve the multi-qubit state which we already have as in Equation 5.8.

$$NOT(|\lambda\rangle) \otimes ID(|\psi\rangle) \otimes ID(|\phi\rangle) = NOT \otimes ID \otimes ID(|\lambda\psi\phi\rangle) \tag{5.8}$$

We have already calculated the state $|\lambda\psi\phi\rangle$ in Equation 5.3. So if we can compute $NOT \otimes ID \otimes ID$ we will be able to create an 8x8 matrix which when multiplying the 8x1 statevector will produce an updated 8x1 statevector. We will follow a worked example of this given the conditions presented in Equation 5.9.

$$|\lambda\rangle = \begin{bmatrix}0\\1\end{bmatrix}, |\psi\rangle = \begin{bmatrix}1\\0\end{bmatrix}, |\phi\rangle = \begin{bmatrix}1\\0\end{bmatrix}, \tag{5.9}$$

Given the results of Equation 5.3, we can write the resolved statevector as in Equation 5.10.

$$|\lambda\psi\phi\rangle = \begin{bmatrix} c_0^\lambda c_0^\psi c_0^\phi \\ c_0^\lambda c_0^\psi c_1^\phi \\ c_0^\lambda c_1^\psi c_0^\phi \\ c_0^\lambda c_1^\psi c_1^\phi \\ c_1^\lambda c_0^\psi c_0^\phi \\ c_1^\lambda c_0^\psi c_1^\phi \\ c_1^\lambda c_1^\psi c_0^\phi \\ c_1^\lambda c_1^\psi c_1^\phi \end{bmatrix} = \begin{array}{c} |000\rangle \\ |001\rangle \\ |010\rangle \\ |011\rangle \\ |100\rangle \\ |101\rangle \\ |110\rangle \\ |111\rangle \end{array} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{5.10}$$

Given what we know about the NOT gate, we can formulate the first tensor product in $NOT \otimes ID \otimes ID$ in Equation 5.11.

$$NOT(|\lambda\rangle) \otimes ID(|\psi\rangle) = \begin{bmatrix} \mathbf{0} & ID \\ ID & \mathbf{0} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \tag{5.11}$$

The second tensor multiplication is then completed as presented in Equation 5.12.

$$(NOT \otimes ID) \otimes ID = \begin{bmatrix} 0 & 0 & ID & 0 \\ 0 & 0 & 0 & ID \\ ID & 0 & 0 & 0 \\ 0 & ID & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{5.12}$$

Applying the resulting matrix from Equation 5.12 to the resolved statevector in Equation 5.10 gives us the result reached in Equation 5.13.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{array}{c} |000\rangle \\ |001\rangle \\ |010\rangle \\ |011\rangle \\ |100\rangle \\ |101\rangle \\ |110\rangle \\ |111\rangle \end{array} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{5.13}$$

On further consideration this result makes sense, since before multiplication (in Equation 5.10) the state $|\lambda\psi\phi\rangle$ was $|100\rangle$ and after a NOT operation was applied to λ our statevector only has a non-zero entry in position $|000\rangle$ implying that the state $|\lambda\psi\phi\rangle$ has transitioned to $|000\rangle$.

We may now be satisfied that the result of this computation is as we expected, however the reader may be considering why we went to the effort of computing the statevector and the tensor product of our gates, when we could have simply applied the NOT gate in the form presented in Equation 5.14.

$$NOT(|\lambda\rangle) : \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}1\\0\end{bmatrix}, |\psi\rangle = \begin{bmatrix}1\\0\end{bmatrix}, |\phi\rangle = \begin{bmatrix}1\\0\end{bmatrix}, \tag{5.14}$$

This method of calculation has produced the correct state for all three qubits (and the multi qubit state if the tensor product was taken). Valkyrie can use this method of localised matrix calculation when running in *fast* calculation mode (explained further in Section 6.1.5).

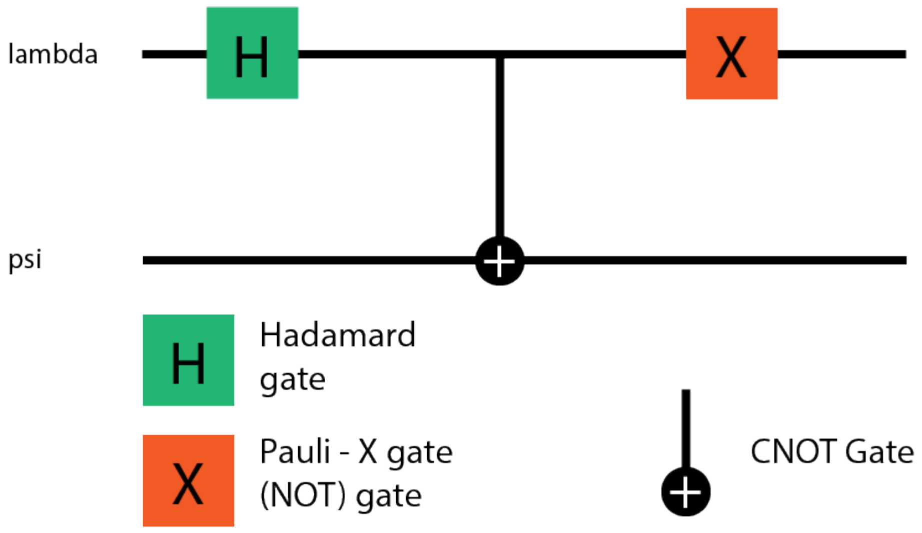


Figure 5.1: Circuit for two qubits

However, this method produces incorrect results when we consider the nature of entangled qubit states. We will follow a worked example of this with a 2-qubit state (for simplicity). Assuming we have two qubits λ and ψ in the states presented in Equation 5.15.

$$|\lambda\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, |\psi\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \tag{5.15}$$

Let us assume that the circuit we want to process on these two qubits is given in Figure 5.1. We will first take the full statevector approach. For which the first step is to resolve the multi-qubit state as presented in Equation 5.16.

$$|\lambda\rangle \otimes |\psi\rangle = \begin{bmatrix} c_0^\psi c_0^\phi \\ c_0^\psi c_1^\phi \\ c_1^\psi c_0^\phi \\ c_1^\psi c_1^\phi \end{bmatrix} = \begin{array}{c c} |00\rangle & \\ |01\rangle & \\ |10\rangle & \\ |11\rangle & \end{array} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{5.16}$$

To perform the first Hadamard gate operation we must take the tensor product between the Hadamard gate (for qubit λ) and the ID gate (for qubit ψ) as presented in Equation 5.17.

$$H \otimes ID = \begin{bmatrix} \frac{1}{\sqrt{2}} \cdot ID & \frac{1}{\sqrt{2}} \cdot ID \\ \frac{1}{\sqrt{2}} \cdot ID & -\frac{1}{\sqrt{2}} \cdot ID \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} & 0 \\ 0 & \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} & 0 \\ 0 & \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} \end{bmatrix} \tag{5.17}$$

We can apply this to the combined state from Equation 5.16, to produce the new state as given by Equation 5.18.

$$\begin{bmatrix} \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} & 0 \\ 0 & \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} & 0 \\ 0 & \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ \frac{1}{\sqrt{2}} \\ 0 \end{bmatrix} \tag{5.18}$$

We can recall the $CNOT$ gate matrix from Equation 1.25, and apply this gate to our state as given by Equation 5.19.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ \frac{1}{\sqrt{2}} \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ \frac{1}{\sqrt{2}} \end{bmatrix} \tag{5.19}$$

We should note that the results of Equation 5.19 implies that our qubits are now in the entangled state $\frac{|00\rangle+|11\rangle}{\sqrt{2}}$. The Pauli-X gate is the same as the NOT gate, so we already have the result of $X \otimes ID$ in Equation 5.11. We can apply this final gate to the combined state as in Equation 5.20.

$$\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ \frac{1}{\sqrt{2}} \end{bmatrix} = \begin{array}{c} |00\rangle \\ |01\rangle \\ |10\rangle \\ |11\rangle \end{array} \begin{bmatrix} 0 \\ \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \\ 0 \end{bmatrix} \tag{5.20}$$

We shall now contrast this result to the naive (but fast) calculation methodology of localised gate applications. Given the starting qubit states in Equation 5.15. We can apply the Hadamard gate to just qubit λ in Equation 5.21.

$$H(|\lambda\rangle) = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} \tag{5.21}$$

For the two qubit $CNOT$ gate we are forced to take the tensor product of $|\lambda\rangle$ and $|\psi\rangle$ which is given in Equation 5.22.

$$|\lambda\rangle \otimes |\psi\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} \\ \frac{1}{\sqrt{2}} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ \frac{1}{\sqrt{2}} \\ 0 \end{bmatrix} \tag{5.22}$$

We should note that at this point this naive approach is giving the same qubit state as in Equation 5.18. We carry out the same operation as in Equation 5.19 to get the two qubit state presented in Equation 5.23.

$$\begin{array}{c} |00\rangle \\ |01\rangle \\ |10\rangle \\ |11\rangle \end{array} \begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ \frac{1}{\sqrt{2}} \end{bmatrix} \tag{5.23}$$

At this point however, we are confronted with a problem. Our state is in the form of the combined state of $|\lambda\psi\rangle$, however the next operation is a single qubit gate operation on λ. We can naively solve this by simply adding together the components of the combined state that relate to the components of the single qubit state for λ. This approach is presented in Equation 5.25.

$$\lambda_0 = |00\rangle + |01\rangle = \frac{1}{\sqrt{2}} + 0 \; \lambda_1 = |10\rangle + |11\rangle = 0 + \frac{1}{\sqrt{2}}. \tag{5.24}$$

$$\psi_0 = |00\rangle + |10\rangle = \frac{1}{\sqrt{2}} + 0 \; \psi_1 = |01\rangle + |11\rangle = 0 + \frac{1}{\sqrt{2}}. \tag{5.25}$$

This gives us new states for λ and ψ as presented in Equation 5.26.

$$|\lambda\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}, |\psi\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} \tag{5.26}$$

We can now proceed to apply the Pauli-X gate (NOT gate) to λ, the results of this is given in Equation 5.27.

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} \tag{5.27}$$

This leaves the individual states of λ and ψ effectively unchanged. We can take the tensor product of these states to have a direct comparison of this naive approach with the full statevector approach as given in Equation 5.28.

$$|\lambda\rangle \otimes |\psi\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} \otimes \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} = \begin{bmatrix} 0.5 \\ 0.5 \\ 0.5 \\ 0.5 \end{bmatrix} \tag{5.28}$$

This final state is very different to the final state we received from the full statevector method as given by Equation 5.20. We must conclude that our naive approach, while requiring fewer calculations, will not lead to an accurate results when entanglement operations are involved in the computation. It can be shown that without entanglement, the naive approach does produce the same results as the full statevector approach.

Chapter 6

Implementation

Summary

In this section we discuss how we have implemented our quantum computer emulation. We go in deep detail as to what algorithms and data-structures we have defined to complete the required steps for quantum emulation. We also make reference to Appendices B and C which contain the entire Valkyrie and Visual-Q codebase.

Valkyrie's construction required multiple technical challenges to be surpassed, I will outline explain the resolution of these challenges in the following section. Furthermore, we will also explore the design of Visual-Q and it's interoperability with Valkyrie. All code that was written by myself is included in the appendix.

6.1 Valkyrie: GPU accelerated quantum computing

To address objective 2.1 I have developed an architecture for a quantum computer emulator which leverages GPU hardware to accelerate calculations. To ensure the performance of this component I have decided to write the *Valkyrie* emulator in C++. This language has many benefits for this form of work, chief among which is "near-metal" programming features with direct access to pointers and low level concurrency controls. Furthermore, C++ has a strong community and is a mature language with good documentation and a rich suite of packages to help with implementation. Finally, C++ is almost unrivalled in terms of performance and will provide a strong foundation to build a fast quantum computer emulator.

The overall system diagram of *Valkyrie* is shown in Figure 6.1 with the codebase included in Appendix B. In this section we will understand the operations that *Valkyrie* performs to allow for quantum computer emulation.

Valkyrie has two fundamental modes of operation, it can run in "fast" operation mode, which provides accurate single qubit gate operations but cannot preserve entanglement relationships between multiple qubits when multi-qubit gates are used. A mathematical explanation of this loss of entanglement can be found in Section 5.2, Valkyrie's fast computation mode effectively follows the "Naive" processing pathway.

For full accuracy and to preserve entanglement relationships Valkyrie can run in "statevector" compute mode (with the command line argument "-sv"), which runs a fully accurate simulation of the quantum system, preserving entanglement relationships through full tensor products of matrices and the statevector as mentioned in Section 5.2.

The speed of the computation in "fast" compute mode will certainly be attractive for users who are doing single qubit simulations, while those researching more complex algorithms requiring accuracy would likely want to use statevectors.

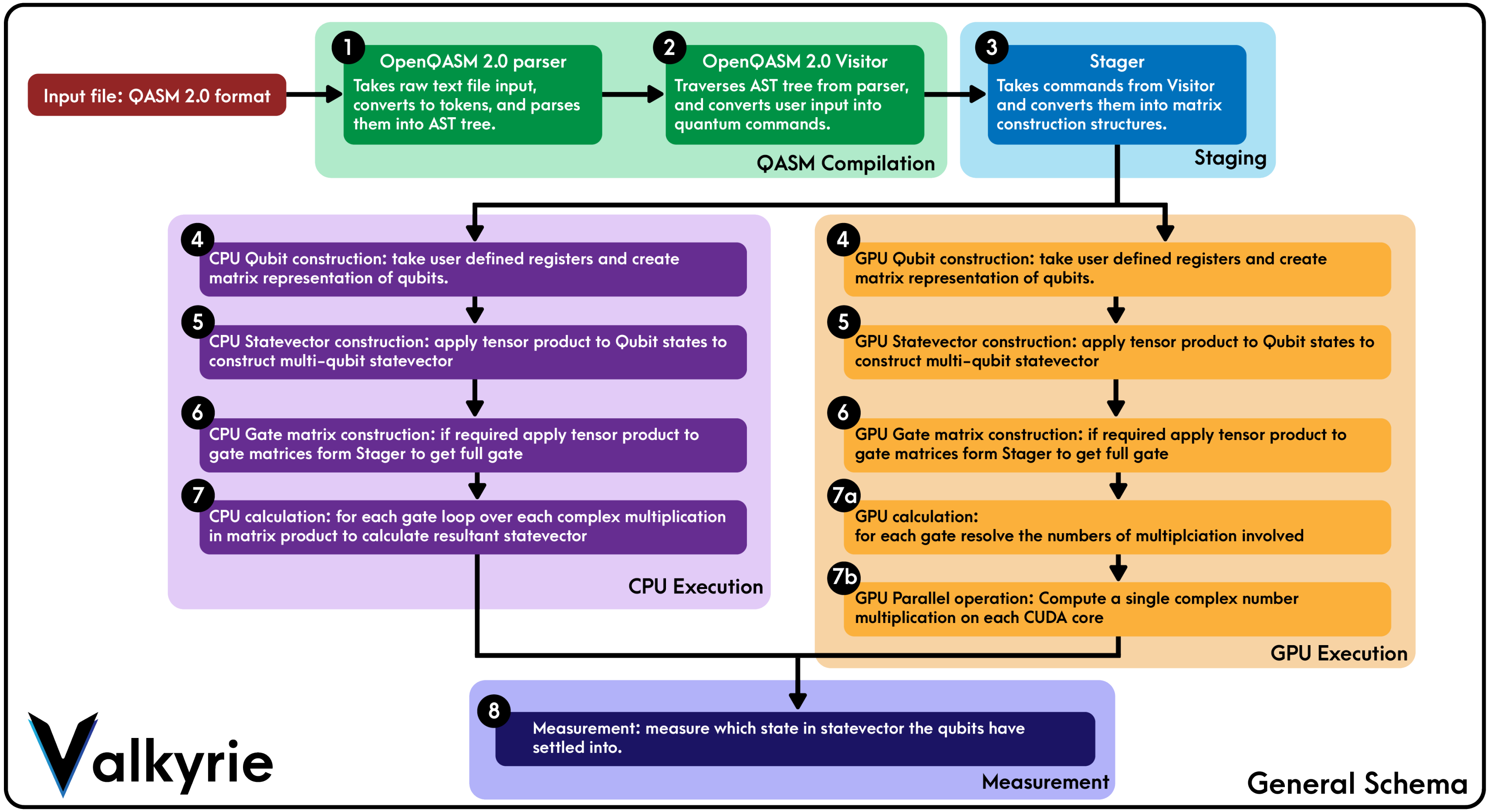


Figure 6.1: Valkyrie overall system diagram detailing the operations and stages required to simulate OpenQASM code

6.1.1 QASM Compilation

To emulate a quantum computer, we must first decide on a language with which to program our simulated quantum computer. We have decided to use the OpenQASM standard released by IBM [9]. This standard provides a flexible, readable and succinct syntax for quantum circuit programming. IBM's Qiskit [19] python package, which is a key competitor for Valkyrie, uses QASM in it's backend operation. This provides us a well recognised and supported quantum assembly language for Valkyrie to simulate on.

When considering how to parse and compile OpenQASM code, we have selected the option of using ANTLR [34] parser generator. The ANTLR tool defines a language definition format, if we provide the tool with a language defined in it's format the tool will produce a parser for this language. This system provides a lot of flexibility and allows for fast parser development. Furthermore, the simplicity of use of the ANTLR tool allows us to abstract away much of the complication of creating a parser while still maintaining access to the full abstract syntax tree.

A file while defines the OpenQASM 2.0 language format for ANTLR, which is defined in Cross's 2017 paper [9] has been made available by Adam Kelly [39]. This file is accepted by the ANTLR parser generator, which produces a lexer, which is capable of identifying OpenQASM tokens and storing them. The tool then produces a parser which generates a traversable Abstract Syntax Tree (AST). Most of the files generated are done so automatically with little input from myself, and so have been omitted from Appendix B since they were not written by me. Each of these files also has a comment stating that it was generated by ANTLR to avoid any mis-attribution of credit. The only ANTLR generated file that I have made a major contribution to and so have included in Appendix B Entry B.2 is the abstract syntax tree visitor which is explained in Section 6.1.1.

At this point, we must traverse and compile the tree into datastructures which the rest of *Valkyrie* can utilise to perform the quantum calculations.

AST Visitor

The Abstract Syntax Tree (AST) produced by the ANTLR parser provides tree like access to the user inputs. This tree has a root node which represents program entry, the tree then splits into branches, each representing OpenQASM commands which must be dealt with in turn. The majority of this AST traversal is performed by the file "qasmBaseVisitor.h" included in Appendix B Entry B.2 which has a Visitor class defined.

The entrypoint to this Visitor is the "visitMainprog" function, this function must be given the root node of the AST tree produced by the ANTLR parser. When this root node is provided the first thing this function does is read the header data of the user input which can be seen on line 102 of Appendix B Entry B.2. Once this header data is resolved and passes checks, the only other children of the root node are the statements which define the program itself. These statements can be broadly grouped in three categories:

- **Declaration**: defines a classical register of bits or quantum register of qubits.
- **Quantum Operation**: defines a quantum operation on one or more qubits.
- **Gate declaration**: allows the user to define custom gates using hardware primitive and default QASM library gates.

Declaration parsing
A typical register declaration will take the form:

```
qreg q[3];
```

This is to be understood as the user requesting a quantum register containing 3 qubits to be instantiated. This register has been given the name "q" by the user. The entrypoint for declaration parsing can be found in the code in Appendix B Entry B.2 Line 145 in the function *visitDecl*. This function accepts the user declaration and calculates whether the user asked for a new classical register or a new quantum register, this can be seen on Line 150. Once that is resolved, the Visitor generates a new "Register" data-structure (defined in Appendix B Entry B.4 Line 173) and pushes this new register into it's register store. Whenever the visitor meets an operation later on in the parsing it will check this register store to calculate which qubits are being operated on.

Quantum operation parsing

In the context of OpenQASM quantum operations define a set of operations on qubits. There are unitary operations on qubits themselves, which are applied by gates, and then there are measure operations where the quantum states of qubits are collapsed into a classical 0 or 1 and stored in a classical register. A typical measurement operation in QASM will take the form:

```
measure q[0] -> c[0];
```

If the user inputs this command, they are requesting that the state of qubit in position 0 of register "q" is measured and then the result passed into position 0 of register "c". These commands are formatted into the "MeasureCommand" data-structure (defined in Appendix B Entry B.4 Line 453) and stored for use by the Measurement module which is detailed in Section 6.1.7.

Unitary operations Typical unitary operation requested by a user:

```
U(0.5,0.4,0.2) q[2];
```

In this command the user is requesting that we apply a Universal rotation gate (see Section 1.2.4) with parameters 0.5, 0.4 and 0.2 to qubit in position 2 in register "q". Unitary operations are the application of gates to qubits in QASM, the reason that this is called "unitary" is that all quantum computing gates are reversible (their gate matrices are invertible) [1]. The entrypoint for parsing these unitary operations can be found in Appendix B Entry B.2 Line 220 which defines the function "visitUop". This function also plays a crucial role in accepting gate declarations, but must behave differently according to whether it is parsing a user defined gate or when a user is requesting to apply a gate to a particular qubit or set of qubits. This is why we have a "gateDeclMode" switch on Line 221.

If we are not in a gate declaration, the function firstly checks whether the unitary operation requested is one of QASM's universal gate set [9]. As seen on Line 222 and Line 234, if the user has requested a U or CX gate the Visitor prepares a "GateRequest" data-structure (defined in Appendix B Entry B.4 Line 267). These "GateRequest" structures don't contain the gate matrix itself, that will be handled in the execution module, instead this data-structure contains all the information required to build a gate-matrix (see Section 5.2) including gate parameters and which qubits are being operated on.

However, there are other predefined gates that the user has access to are included in a standard QASM library file called "qeLib1.inc" (which can be found in this repository). In this file useful gates are defined in QASM's gate declaration format, which can be parsed by Valkyrie. However, since these gates are commonly used we have precompiled them in the file "ParsingGateUtilities.cpp" (included in Appendix B Entry B.22). This allows for very efficient access to these standard library gates which cover most quantum operations. An example of a gate application from the standard library is given below.

```
ccx q[0],q[1],q[2];
```

The "ParsingGateUtilities.h" file (included in Appendix B Entry B.21) defines simple functions on Line 10 and Line 11 which correspond to gates with and without parameters. In the example above a "ccx" gate (controlled-controlled not gate) is applied to qubits from register "q" in positions 0, 1 and 2. The "ccx" string along with these qubits positions is given to "compileCompoundGateRequest" on Line 10 of "ParsingGateUtilities.h", which redirects this to Line 655 of "ParsingGateUtilities.cpp". In this function we enumerate this particular gate on Line 680, and then using a switch statement on Line 717 we return the list of QASM universal set gates required to achieve the "ccx" gate.

The final possibility for unitary operations is for the user to apply a user-defined gate. Valkyrie supports the user defining a custom gate. If the user wants to apply this custom gate they can do so in the same manner as they would any other gate as shown below:

```
customGate(0.1,0.6) q[0];
```

The visitor searches for this gate name which can be seen in the function labelled "visitUop" in the file "qasmBaseVisitor.h" (included in Appendix B Entry B.2 Line 220), specifically Line 253 checks whether the gate name that the user has requested (in this case "customGate") has been defined by the user in the code so far. If not the program then checks whether the gate is from the "qeLib1.inc" standard library precompiled functions in "ParsingGateUtilities.cpp". If Valkyrie cannot find the gate in either location it exits with error.

If a gate is found, whether it be a universal gate set gate, a standard library gate or a user defined gate a "GateRequest" is created and added to the visitors list of Gate requests, this is aided by the "attachGates" utility function detailed on Line 72 of "qasm2BaseVisitor.h".

Gate declaration

Similar to how programming languages allow users to define functions, QASM allows programmers to write custom gates, which apply multiple subgates using the same parameters and qubits. An example of a user defined gate can be seen below.

```
gate customGate(a,b,c) x,y,z {
    U(a,b,c) y;
    cx x, y;
    ccx x, y, z;
}
```

The "customGate" defined above will apply a Universal Rotation gate with parameters "a", "b" and "c" to the qubit represented by "y", which are given by the user when they call the "customGate". It will then perform a controlled-not operation between qubits "x" and "y". Finally this gate will perform a controlled-controlled-not operation between all three qubits in the gate declaration. To invoke this particular gate, the user must provide the three arguments "a", "b" and "c" as well as specify three qubits for "x", "y" and "z". An example of this can be seen below:

```
customGate(0.1,0.2,0.6) q[0], q[1], q[2];
```

The difficulty of parsing these custom gate declaration comes from the fact that we must somehow store the construction of all of the subgates as well as which argument and qubit goes to which subgate. When accepting normal gate applications we would have that immediately, but with custom gates we may have to wait for multiple lines for a gate application if at all.

To achieve this gate declaration parsing we parse the gate declaration and sub-gate operations separately in the "qasmBaseVisitor.h" file in the functions "visitGatedecl" (Line 166) and "visitGoplist" (Line 187). The "visitGatedecl" parses the gate name and stores the names of parameters (such as "a", "b" and "c") and the names of the qubit's that the user has given such as "x", "y" and "z". The function stores all of this information in the "gateDeclaration" data-structure (defined in Appendix B Entry B.4 Line 77).

The "visitGoplist" function has a more difficult task, it must effectively parse "uop" commands (see Section 6.1.1) but without storing them as actual gate operations, instead store them as potential gate operations on abstract parameters and qubits. To achieve this we set the "gateDeclMode" flag (defined in Appendix B Entry B.2 Line 47) and request the "visitUop" function to parse these operations in an abstract sense, and store the name of the gate, the abstract parameters it uses and the abstract qubits it operates on in a "gateOp" data-structure (defined in Appendix B Entry B.4 Line 85).

The results of these two functions is a "gateDeclaration" and list of "gateOp"'s, this information is collected on Line 120 of the function "visitStatement" (defined in Appendix B Entry B.2 Line 112). The data is then sent to the "compileCustomGate" function which is given a declaration on Line 13 of "ParsingGateUtilities.h" (included in Appendix B Entry B.21) and implemented on Line 931 of "ParsingGateUtilities.cpp" (included in Appendix B Entry B.22). This function is able to ready the gate declaration and relate all abstract gate operations to the parameters and qubits given in the gate operation. The function then returns an "std::function" datatype, which is a pointer to an operable function. This function is created on Line 901 of "ParsingGateUtilities.cpp" which takes as input the true parameters and qubits that the user uses to call the "customGate" to create a list of "GateRequest"'s which can be operated on by the rest of Valkyrie. Pseudocode for the algorithm that does this is included in Algorithm 1.

Algorithm 1: Function pointer creation for custom gates

```
Result: Function pointer for constructing GateRequests from custom gate use
input: gateDeclaration, vector<gateOp> operations;
map<paramName,paramLocation> paramMap =
  resolveParameterLocations(gateDeclaration);
map<qubitName,qubitLocation> qubitMap = resolveQubitLocations(gateDeclaration);
functionPointer outputFunc(actualParameters, actualQubits) =
vector<gateRequests> outputRequests;
for gate in operations do
    paramLocations = paramMap[gate.paramNames];
    qubitLocations = paramMap[gate.qubitNames];
    actualParams = actualParameters[paramLocations];
    actualQubs = actualQubits[qubitLocations];
    outputRequests.attachGates(compileGateRequests(gate.name, actualParams,
      actualQubs);
end
return outputFunc
```

The real advantage of storing these custom gates as functions, is the ease of applying the gate once we have a list of the functions. We can see how these function pointers are applied on Line 261 of "qasm2BaseVisitor.h". The list of "GateRequest"s returned by this function are stored like any normal gate application using the "attachGates" function detailed earlier.

Result of Compilation

Now that we have compiled all the user inputted code, we must communicate this information to the *Staging* component of Valkyrie. To do this the "qasm2BaseVisitor.h" class (defined in Appendix B Entry B.2) exposes the functions "getRegisters" (Line 87), "getGates" (Line 91) and "getMeasureCommands" (Line 95). These functions provide the rest of Valkyrie with the information it requires to perform the quantum calculations. An overall view of the QASM Compilation workflow is provided in Algorithm 2.

Algorithm 2: Overall workflow for QASM compilation

```
Result: vector<Register> registers, vector<GateRequest> gates,
        vector<MeasureCommand> commands
input: user input QASM code;
tokens = ANTLRLexer(userInput);
ASTTree = ANTLRParser(tokens);
visitor.visitMainprog(ASTTree);
registers = visitor.getRegisters();
gates = visitor.getGates();
commands = visitor.getCommands();
return registers, gates, commands;
```

6.1.2 Staging

An important process we need to complete before generating gate matrices and running calculations is to work out what gates we can parallelise and which ones we can't. For example if we have a gate arrangement as presented in Figure 6.2. Since these gates affect different qubits we are able to parallelise processing them during "fast" computation mode. In statevector mode, this distinction is less important since we cannot parallelise processing multiple gates at once. However a circuit such as that in Figure 5.1 cannot be parallelised even in "fast" compute mode since it contains a multi-qubit gate.

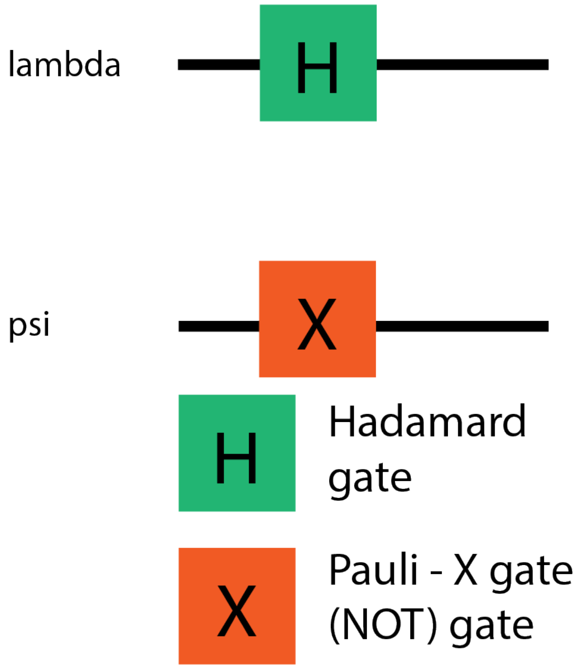


Figure 6.2: Gates which can be parallelised during processing

To convey this sense of parallelism we have implemented a "ConcurrentBlock" data-structure (defined in Appendix B Entry B.4 Line 308). Which stores a list of "GateRequest"'s that can be parallelised. In "fast" compute mode we work on each of the gates in a "ConcurrentBlock" in parallel, while in "statevector" compute mode we work through them in series.

The job of separating out the incoming "GateRequests" into "ConcurrencyBlocks" is performed by the staging block. The "Stager" class contains this functionality and is defined in "staging.h" (included in Appendix B Entry B.23). The function which calculates and separates out the blocks is found on Line 30. The psuedocode for the algorithm it uses can be found in Algorithm 3.

Algorithm 3: Concurrency block resolution algorithm

```
Result: vector<ConcurrencyBlock> blocks
input: vector<GateRequest> gateRequests;
vector<ConcurrencyBlock> blocks;
ConcurrencyBlock tempBlock;
for gate in gateRequests do
    if gate.qubitCount == 1 then
        tempBlock.append(gate);
    else
        blocks.append(tempBlock);
        tempBlock.reset();
    end
end
return blocks;
```

6.1.3 Preparing for quantum calculation

Since Valkyrie supports computation being performed on both the CPU and GPU, we need to define a common interface for the computation flow, which can be implemented in CPU and GPU code respectively. We are impeded in this common execution model due to compilation requirements which say that any code which calls GPU device code must be defined in ".cu" files. This means that we will unfortunately have a lot of code duplication between CPU and GPU execution modes.

The overall workflow is visualised in Figure 6.3. The common interface for both CPU and GPU implementations can be found in the file "AbstractDevice.h" (included in Appendix B Entry B.3). This file defines the core functions which are required to implement the workflow shown in Figure 6.3.

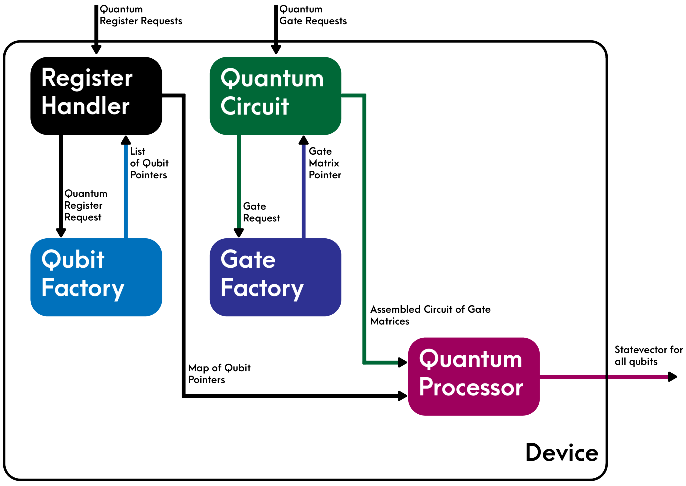


Figure 6.3: Workflow for Quantum Compute device

Qubit Factory

The role of the Qubit Factory is to create new "Qubit" data-structures (defined in Appendix B Entry B.4 Line 332). These "Qubit" data structures are particularly important for the "fast" compute mode. They hold the state of individual qubit's in our circuit. An important role of the Qubit Factory is to allocate heap memory for each qubit, and make sure to de-allocate the heap memory. If this memory wasn't unallocated then we would have a memory leak which could cause other programs to crash. The interface for these Qubit Factories can be found in the virtual class "AbstractQubitFactory" (defined in Appendix B Entry B.3 Line 23). Crucially the function "generateQubit" on Line 27 must be implemented by both CPU and GPU implementations, which returns a "Qubit" pointer allowing other classes to use and manipulate the data stored inside.

Common implementation

Both CPU and GPU implementations of the "AbstractQubitFactory" class share the same code (written twice due to GPU code compilation restriction). The CPU declaration can be found in "CPUQubitFactory" defined in "CPUDevice.h" (Appendix B Entry B.6 Line 21) and implemented in "CPUDevice.cpp" (Appendix B Entry B.7 Line 48). Whereas the GPU declaration can be found in "GPUQubitFactory" defined in "GPUDevice.cuh" (Appendix B Entry B.14 Line 20) and implemented in "GPUDevice.cu" (Appendix B Entry B.15 Line 51).

This code follows the pseudocode given by Algorithm 4.

Algorithm 4: Algorithm for generating qubits

```
Result: Qubit Pointer
input: None;
s1 = allocateMemory(complexNumber);
s2 = allocateMemory(complexNumber);
Qubit* = allocateMemory(Qubit(s1,s2));
track(Qubit*);
return Qubit*;
```

Since the "Qubit" pointer is tracked, it can be deleted in the Qubit Factory destructor. An example of this can be seen in the file "GPUDevice.cu" Line 67 (included in Appendix B Entry B.15).

Gate Factory

The Gate Factory has a similar role to the Qubit Factory. Gate Factories must create new "Gate" data-structures (defined in Appendix B Entry B.4 Line 354). These "Gate" structures store the gate matrix with any parameters integrated into the gate matrix in the appropriate fashion. Once again the Gate Factory allocates, tracks and de-allocates heap memory for these "Gate"'s. The interface for Gate Factories can be found in the virtual class "AbstractGateFactory" (defined in Appendix B Entry B.3 Line 32). In this definition the "generateGate" function (Line 36) contains the important function interface to create a "Gate" pointer which can be operated on by other classes.

Common implementation

Both CPU and GPU implementations of the "AbstractGateFactory" class share the same code (written twice due to GPU code compilation restriction). The CPU declaration can be found in "CPUGateFactory" defined in "CPUDevice.h" (Appendix B Entry B.6 Line 35) and implemented in "CPUDevice.cpp" (Appendix B Entry B.7 Line 75). Whereas the GPU declaration can be found in "GPUGateFactory" defined in "GPUDevice.cuh" (Appendix B Entry B.14 Line 34) and implemented in "GPUDevice.cu" (Appendix B Entry B.15 Line 78).

Both CPU and GPU implementations rely on helper functions to generate the correct gate matrices for each gate. The CPU version of this can be found in "getGateMatrix" in the file "CPUDevice.cpp" on Line 25 while the GPU implementation of this can be found in "getGateMatrixGPU" in the file "GPUDevice.cu" on Line 28. As discussed in Section 5.2 QASM uses a universal gate set of just "U", the universal rotation gate, and "CX" the controlled not gate. Therefore there are only two types of matrices we will need to construct, the "CX" gate matrices are simple to construct and their construction can be seen on Line 43 of "CPUDevice.cpp". The Universal rotation gate has three parameters which must be operated to generate the gate matrix. We covered the construction of this gate in Section 1.2.4, and with the result that can be seen in Equation 6.1.

$$U(\theta,\phi,\lambda)=\begin{bmatrix} cos(\frac{\theta}{2}) & -e^{i\lambda}sin(\frac{\theta}{2}) \\ e^{i\theta}sin(\frac{\theta}{2}) & e^{i\lambda+i\phi}cos(\frac{\theta}{2}) \end{bmatrix} \tag{6.1}$$

Both GPU and CPU devices use utility files to help with the construction of these gates. The CPU implementation can be found on Line 13 of "GateUtilitiesCPU.h" (included Appendix B Entry B.10) and the GPU implementation on Line 13 of "GateUtilities.GPU.cuh" (included Appendix B Entry B.11). These functions perform the mathematics required in Equation 6.1, and returns a gate matrix to their appropriate calling function. Overall the function of the Gate Factory can be summarised in Algorithm 5.

Algorithm 5: Algorithm for generating gates

```
Result: Gate pointer
input: GateRequest;
gateMatrix =
if GateRequest.type == "CX" then
    return CXMatrix;
else
    matrix = use uGateUtilityFunction with GateRequest.parameters;
    return matrix;
end
gate* = allocateMemory(Gate(gateMatrix));
track(gate*);
return gate*;
```

As with the Qubit Factory, we track all Gate's created so that the destructor of the Gate Factory can de-allocate the memory at the end of execution.

Quantum Circuit

We now have the ability to generate "Qubit"s and "Gate"s at will with our factories. We now require a data-structure to arrange these qubits and gates as the user as requested. The "AbstractQuantum-

Circuit" (defined in Appendix B Entry B.3 Line 41) provides us the interface for constructing a class which performs this exact function. The interface defines multiple functions which we must review in detail. The following list describes and explains the function of each of the interface functions, we will also specify where in the codebase each function and implementation are located. Assume that "def" refers to a line in the file "AbstractDevice.h" (Appendix B Entry B.3) while "cpu" refers to a line in the file "CPUDevice.cpp" (Appendix B Entry B.7) and "gpu" refers to a line in the file "GPUDevice.cu" (Appendix B Entry B.15).

Common implementation

- **loadQubitMap** def: 46, cpu: 105, gpu: 108.

  The "loadQubitMap" function accepts the map of register names to "Qubit" pointers. This is crucial to both "fast" and "statevector" compute modes. As can be seen in both CPU and GPU implementations, the qubitMap is used to generate a StateVector.

- **loadBlock** def: 47, cpu: 114, gpu: 117.

  The "loadBlock" function allows us to sequentially load "ConcurrentBlock"s generated by the Stager (see Section 6.1.2). Each block has each of it's gates processed, with the registers affected and "GateRequest" extracted from the block, and then used to generate a "Calculation" data-structure (defined in Appendix B Entry B.4 Line 383). Crucially each "Calculation" can be directly operated on by the "QuantumProcessor", essentially this data-structure has the raw data ready for matrix multiplication.

- **getNextCalculation** def: 48, cpu: 135, gpu: 138.

  The "getNextCalculation" function is called by a "QuantumProcessor", this function allows us to abstract away the order of calculation from the "QuantumProcessor", this is instead tracked by the "QuantumCircuit" itself. Using the "calcCounter" and the size of the "calculations" vector this function can calculate which function to give to the processor.

- **returnResults** def: 49, cpu: 149, gpu: 152.

  The "returnResults" function exposes the "qubitMap", this is important for the "fast" compute mode, since then individual qubit states are stored in this map.

- **getStateVector** def: 50, cpu: 155, gpu: 158.

  The "getStateVector" function exposes the "StateVector" to functions which want to observe the full statevector after computation. This method is useful for both "fast" computation mode where the statevector is used for measurement and for the "statevector" compute mode where this statevector is consistently used for computation and essential for measurement.

- **checkComplete** def: 51, cpu: 160, gpu: 163.

  The "checkComplete" function allows the "QuantumProcessor" to check if there are any calculations left to compute.

The "QuantumCircuit" class is passed into the "QuantumProcessor" class, which uses the circuit to queue up and calculate all the matrix multiplications it needs to perform.

6.1.4 StateVector

The "StateVector" is a crucial data-structure for the operation of Valkyrie. For some mathematical context on the importance and role of the statevector see Section 5.1. The "StateVector" structure defined on Line 475 of "BaseTypes.h" (included in Appendix B Entry B.4) contains a representation of the mathematical statevector we present in Section 5.1 and also contains some important functions for our processing of the statevector. The "StateVector" is instantiated by the "loadQubitMap" function of the "QuantumCircuit".

Tensor Product

The tensor product is an important function to generate the statevector, the mathematical operation is detailed in Section 5.1. This operation is implemented on Line 607 of "BaseTypes.h", this function iterates over all registers contained in the "qubitMap" provided to it and stores each qubit as an "SVPair" (defined on Line 48 of "BaseTypes.h") which stores the name of the register where the qubit was found, and the location of the qubit in that register.

Once the function has all of the SVPair's, it can create the full statevector. To understand how we achieve that, we must return to the mathematical representation of the statevector, given by Equation 6.2.

$$|\lambda\rangle \otimes |\psi\rangle \otimes |\phi\rangle = \begin{bmatrix} c_0^\lambda \\ c_1^\lambda \end{bmatrix} \otimes \begin{bmatrix} c_0^\psi c_0^\phi \\ c_0^\psi c_1^\phi \\ c_1^\psi c_0^\phi \\ c_1^\psi c_1^\phi \end{bmatrix} = \begin{bmatrix} c_0^\lambda \cdot \begin{bmatrix} c_0^\psi c_0^\phi \\ c_0^\psi c_1^\phi \\ c_1^\psi c_0^\phi \\ c_1^\psi c_1^\phi \end{bmatrix} \\ c_1^\lambda \cdot \begin{bmatrix} c_0^\psi c_0^\phi \\ c_0^\psi c_1^\phi \\ c_1^\psi c_0^\phi \\ c_1^\psi c_1^\phi \end{bmatrix} \end{bmatrix} = \begin{bmatrix} c_0^\lambda c_0^\psi c_0^\phi \\ c_0^\lambda c_0^\psi c_1^\phi \\ c_0^\lambda c_1^\psi c_0^\phi \\ c_0^\lambda c_1^\psi c_1^\phi \\ c_1^\lambda c_0^\psi c_0^\phi \\ c_1^\lambda c_0^\psi c_1^\phi \\ c_1^\lambda c_1^\psi c_0^\phi \\ c_1^\lambda c_1^\psi c_1^\phi \end{bmatrix} \tag{6.2}$$

We note that the order in which the three qubits λ, ψ and ϕ were positioned in the tensor product dictated which of their elements (e.g. c_0^λ) appeared in each element of the statevector. For example we can see that since λ is the first qubit state in the tensor product, the entire first half of the resultant statevector have the c_0^λ element while the second half have the c_1^λ element. In fact the element indices for each qubit (such as 0th element or 1st element) form a 3 bit counter in ascending order as we travel down the statevector. That is to say that the indices form the pattern shown in Equation 6.3.

$$\begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} \begin{bmatrix} 000 \\ 001 \\ 010 \\ 011 \\ 100 \\ 101 \\ 110 \\ 111 \end{bmatrix} \tag{6.3}$$

We can conclude that these element indices represent the binary value of the position in the statevector which they inherit. We can further infer that there must be a way of relating the position of a qubit state in the original tensor product (such as λ being first in the tensor product) to the element of that qubit state which is multiplied in that position of the statevector.

We have created a new function called the "inverseTail" function to achieve this, and is described in Algorithm 6 and implemented on Line 494 of "BaseTypes.h". Incidentally the "tail" function implemented on Line 506 of the same file allows us to check whether we are using the 0th or 1st element of a particular qubit state.

Algorithm 6: Algorithm for calculating which element of a qubit is used in the resultant vector of a tensor product

```
Result: integer: either 0 or 1
input: noQubits, indexOfQubitInTensorProduct, locationInStateVector;
j = raise 2 to the power of (noQubits - indexOfQubitInTensorProduct);
if (locationInStateVector % j) < (j / 2) then
    return 0;
else
    return 1;
end
```

We can see how the "inverseTail" function works by running through an example. Say we are trying to apply the tensor product in Equation 6.2, and wanted to work out which element of ψ's qubit state we need to multiply to calculate element 4 of the tensor product. For reference, element 4 of the tensor product is $c_1^{\lambda} c_0^{\psi} c_0^{\phi}$ and therefore has index 0 for qubit state ψ.

The first operation in the "inverseTail" is to calculate j. We know that the number of qubits in total is three and that ψ is in position 1 out of those three qubits in the tensor product.

$$j = 2^{3-1} = 4$$

We then perform the modulo operation between the location in the state vector we are computing for (4) and the value of j.

$$4mod(j) = 4mod(4) = 0$$

Finally we need to do one more operation, the **integer** division of j by 2.

$$j/2 = 2$$

Finally we compare the modulo to the integer division, since $0 < 2$, we satisfy the **if** condition, and return the index 0 as we expect.

Now that we understand how to calculate which index of each qubit state we need to create each element of the final statevector, we simply have to populate the statevector with the product of the "inverseTail" indexed qubit state. This process is completed between Lines 617 and 625 in "BaseTypes.h". The overall tensor product algorithm that Valkyrie uses is detailed in Algorithm 7.

Algorithm 7: Algorithm for calculating the full tensor product between given qubit states

```
Result: vector<complex> stateVector
input: map<registerName, vector<qubit» qubitMap;
for element in qubitMap do
    create SVPair;
    store pair in positions
end
SVSize = raise 2 to the power of positions.size;
for index in StateVector do
    value = 1;
    for qubit in positions do
        stateIndex = inverseTail(positions.size, qubit.position, index);
        value = value * qubitMap[qubit].at(stateIndex);
    end
    StateVector[index] = value;
end
return StateVector;
```

Fast compute mode

When Valkyrie is operating in "fast" compute mode, the statevector only ever acts as storage for the state of the qubit system. The "qubitMap" datastructure (defined in of Appendix B Entry B.6 Line 53 and Entry B.14 Line 52) is used as input to gate matrix calculations and the results are stored back into the map, with the statevector simply updated to attempt to salvage entanglement relationships under certain circumstances.

Single qubit gate As discussed in Section 1.2.4, QASM essentially only has a "U" single qubit gate, which can represent any required single qubit gate by modifying the input parameters to the U gate (see Equation 6.1). In "fast" calculation mode the gate matrix for this single qubit gate, which is stored in a "Calculation" data-structure, is applied to the "Qubit" data-structure for which-ever exact qubit is being operated on. As discussed in section 5.2, this method can fail to accurately maintain the quantum state if multi-qubit gates are requested by the user. Once the "Qubit" data-structure for the qubit in question is updated it is stored in the "qubitMap", the "StateVector" which is held in the quantum circuit is "refreshed". What this means is that the tensor product algorithm defined in Algorithm 7 is run again with the updated "qubitMap" to produce an updated statevector.

Multi-qubit gate As discussed in Section 5.2, "fast" computation mode breaks down when multi-qubit gates are involved. However, this approach to the statevector computation allows us to attempt to preserve entanglement relations. Since the only multi-qubit gate that QASM uses in calculation is the "CX" gate, we know we will need a 4x1 vector to apply the 4x4 "CX" gate matrix. This is acheived by taking the local tensor product of the two qubits which are going through this gate. This process can be seen on Line 258 of "CPUDevice.cpp" (included in Appendix B Entry B.7). Once this is complete, we apply the matrix multiplication and are left with a 4x1 vector which contains the new combined state of the two qubits. This can be seen represented by Equation 6.4.

$$\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&0&1\\0&0&1&0\end{bmatrix} \cdot \begin{array}{c}|00\rangle\\|01\rangle\\|10\rangle\\|11\rangle\end{array}\begin{bmatrix}c_0^{\psi}c_0^{\phi}\\c_0^{\psi}c_1^{\phi}\\c_1^{\psi}c_0^{\phi}\\c_1^{\psi}c_1^{\phi}\end{bmatrix} = \begin{array}{c}|00\rangle\\|01\rangle\\|10\rangle\\|11\rangle\end{array}\begin{bmatrix}c_0^{\psi}c_0^{\phi}\\c_0^{\psi}c_1^{\phi}\\c_1^{\psi}c_1^{\phi}\\c_1^{\psi}c_0^{\phi}\end{bmatrix} \tag{6.4}$$

In the "fast" computation mode we can't faithfully represent this entanglement truly, however we can modify the "StateVector" at to hold the entanglement for at least one more calculation. To perform this we use the "modifyState" function on Line 648 of "BaseTypes.h" (included in Appendix B Entry B.4). This function works out which positions in the tensor product the two qubits that went through the "CX" gate were in. We then consider that in every element of the statevector, the two qubits can only be combined in 4 ways. Namely, $c_0^{\psi}c_0^{\phi}$, $c_0^{\psi}c_1^{\phi}$, $c_1^{\psi}c_0^{\phi}$ and $c_1^{\psi}c_1^{\phi}$. If we can work out for each element of the statevector, which of these combinations are used to make up that element's value we can simply reapply the multiplication for that element with the new $c_i^{\psi}c_j^{\phi}$ and the rest of the qubit state values from the "qubitMap".

This process is completed by the "calculateNewVals" function on Line 525 of "BaseTypes.h". The pseudocode for the operation of this function is given by Algorithm 8.

Algorithm 8: Algorithm for calculating the new values in elements of a tensor product when values are modified

```
Result: vector<complex> stateVector
input: index1, index2, vector<complex> newValues, location1InTensorProduct,
  location2InTensorProduct;
affected = calculateWhichElementsAreAffected(index1, location1InTensorProduct,
  index2, location2InTensorProduct)
for position in affected do
    value = 1;
    for qubit not affected by gate do
        value = value * qubitMap[qubit];
    end
    value = value * newValues[index1*2 + index2];
    StateVector[position] = values;
end
return StateVector;
```

Statevector compute mode

When Valkyrie is in "statevector" compute mode, a full statevector calculation is completed. This involves large matrix products and can leverage the parallel compute ability of the GPU to it's full extent. As described in Section 5.2, for a statevector of size $N \times 1$, all gates that will operate on it will have dimension $N \times N$. This ensures that the dimension of the statevector at the completion of the matrix product is $N \times 1$.

In a sense, the statevector compute mode is less complex from a mathematical point of view. However, from a computational point of view, statevector computation is extremely expensive. As seen in Equations 5.11 and 5.12, just to generate the full $N \times N$ gate we must perform multiple tensor products. For example for a circuit with n qubits, the statevector will be of size 2^n. Consider if we wanted to apply a single qubit gate to qubit i in the circuit, this situation is illustrated in Figure 6.4. To calculate the full matrix which must be applied to the statevector in this case, we must apply the tensor product between identity matrices for every qubit before qubit i and then perform tensor product with the Hadamard gate followed by tensor products with identity matrices for qubits following i up until qubit n. This operation can be illustrated in Equation 6.5.

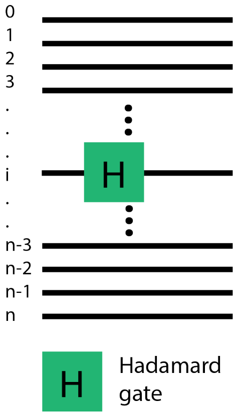


Figure 6.4: A single qubit gate operating on only one qubit, other qubits are unaffected

$$I\left|q_0\right\rangle \otimes I\left|q_1\right\rangle \otimes \cdots \otimes I\left|q_{i-1}\right\rangle \otimes H\left|q_i\right\rangle \otimes I\left|q_{i+1}\right\rangle \otimes \ldots I\left|q_n\right\rangle \tag{6.5}$$

Equation 6.5 can be simplified to Equation 6.6.

$$I \otimes I \otimes \cdots \otimes I \otimes H \otimes I \otimes \cdots \otimes I\left|q_0 q_1 \ldots q_{(i-1)} q_i q_{(i+1)} \ldots q_n\right\rangle \tag{6.6}$$

Once we have calculated the full tensor product implied by Equation 6.6 we can multiply it by the statevector $\left|q_0 q_1 \ldots q_{(i-1)} q_i q_{(i+1)} \ldots q_n\right\rangle$.

Each tensor product is between matrices of dimension 2x2, therefore if we have a matrix M of dimension m by m and matrix N of dimension 2 by 2, and applied the tensor product $M^* = N \otimes M$, the matrix M^* will have dimensions $2m$ by $2m$. Furthermore, to achieve each tensor product we need to complete $4m^2$ calculations.

Given this knowledge, to perform the full tensor product stated in Equation 6.6 we will have to perform more calculations at every tensor product we complete, the number of calculations we will need to complete in total can be expressed as the series given by Equation 6.7.

$$-4 + \sum_{i=1}^{n} 4^n = \frac{4}{3}(4^n - 1) - 4 \tag{6.7}$$

Therefore, for every single qubit gate in a circuit with n qubits we would have to perform $\frac{4}{3}(4^n - 1) - 4$ complex multiplications. While feasible for smaller circuits, as we attempt to emulate larger circuits this number of calculations becomes too computationally expensive. For example, if we wanted to simulate a circuit with 20 qubits, every single qubit gate would require 5.864×10^{12} complex calculations. Each complex calculation would require between 1 to 4 CPU core cycles [40]. Assuming we have a 4GHz CPU core, this calculation would require approximately 98 minutes to complete. We must accept that this amount of time spent carrying out calculation to simply generate the gatematrices is far too long, since we haven't yet reached the actual application of the gatematrices to the statevectors.

Tailed tensor products

We shall take a step back to recap the problem we face here. We recognise that for a circuit with n qubits, we will have a statevector of size 2^n, this is unavoidable. Furthermore, for each gate application we must multiply the statevector by a matrix of dimension 2^n by 2^n, which will entail a total of 2^n by 2^n complex calculations followed by 2^n complex summations. These calculations are unavoidable. However, we are currently in a situation where to form the 2^n by 2^n gate matrix, we will need to perform $\frac{4}{3}(4^n - 1) - 4$ complex multiplications, which is what we are attempting to avoid.

To solve this problem, we must conceive of a method to skip as many of the $\frac{4}{3}(4^n - 1) - 4$ multiplications as possible. We know this is possible, since looking at the results of such tensor products with the identity matrix we can see a large number of the resultant matrix elements are 0 values, this can be seen in Equation 5.12.

The method we have devised we have named the "tailed" tensor product. Consider the following scenario, we have a tensor product between multiple 2x2 matrices, all of these matrices are the identity matrix. This situation is formalised in Equation 6.8.

$$I \otimes I \otimes \cdots \otimes I \tag{6.8}$$

Let us formally compute the first few tensor products, as shown in Equation 6.9.

$$I \otimes I \otimes \cdots \otimes I \tag{6.9}$$

$$I \otimes I = \begin{bmatrix} I & \mathbf{0} \\ \mathbf{0} & I \end{bmatrix}$$

$$I \otimes (I \otimes I) = \begin{bmatrix} (I \otimes I) & \mathbf{0} \\ \mathbf{0} & (I \otimes I) \end{bmatrix} = \begin{bmatrix} I & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & I & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & I & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & I \end{bmatrix}$$

We can see a pattern emerging here, since the Identity matrix only had elements along it's diagonal, if it is on the left of a tensor product we will see zero matrices in the top right and bottom left cornes of the resultant matrix. Furthermore, of the Identity matrix is on both sides of a tensor product we can only expect elements along the leading diagonal. Let us assume that the tensor product we want to compute is $I \otimes I \otimes I \otimes H$ where H is the hadamard gate matrix. We can simply state the result for $I \otimes I \otimes I$ since we know this will simply be a matrix of size 8x8 with the only non zero element's being along the leading diagonal filled with the value 1. Therefore, the final tensor product, $(I \otimes I \otimes I) \otimes H$ is given by Equation 6.10.

$$(I \otimes I \otimes I) \otimes H = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} = \tag{6.10}$$

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

We can clearly see that in this matrix can be very easily constructed, in fact if our tensor is of the form $I \otimes I \otimes \cdots \otimes M$ where M is an arbitrary matrix, we can very easily construct the tensor product result in the form of Equation 6.11.

$$I \otimes I \otimes \cdots \otimes M = \begin{bmatrix} M & 0 & \dots & 0 & 0 \\ 0 & M & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & M & 0 \\ 0 & 0 & \dots & 0 & M \end{bmatrix} \tag{6.11}$$

It should be noted that in Equation 6.11, every 0 entry in the matrix represents a matrix of the same dimensions as M filled with 0's for all entries.

We can conclude from this analysis that if we are able to construct all single qubit gate matrices by performing a tensor product of the form $I \otimes I \otimes \cdots \otimes M$ we no longer have to perform any complex multiplications at all, simply have to construct a matrix datastructure of the appropriate size for the result and simply fill the diagonal elements with M according to Equation 6.11. We must note that this "tail" based tensor product only applies to our purposes if the qubit being operated on is in the last position as shown in Equation 6.12.

$$I \left|q_0\right\rangle \otimes I \left|q_1\right\rangle \otimes \cdots \otimes I \left|q_{n-1}\right\rangle \otimes M \left|q_n\right\rangle \tag{6.12}$$

The tailed tensor product for single qubit gates is implemented on the CPU side in the function "getGenericUResult" (defined on Line 203 in Appendix B Entry B.7) and on the GPU side in the function "getGenericUResult" (defined on Line 206 in Appendix B Entry B.15).

We note that extending this "tail" concept to a two qubit gate is simple, if we have the tensor product in the form presented in Equation 6.13.

$$I \left|q_0\right\rangle \otimes I \left|q_1\right\rangle \otimes \cdots \otimes CX \left|q_{n-1} q_n\right\rangle \tag{6.13}$$

Where the qubit q_{n-1} represents the control qubit and the last qubit q_n represent the qubit that is conditionally inverted.

The implementation of the "CX" gate tailed tensor product is implemented on the CPU side in the function "getCXResult" (defined on Line 170 in Appendix B Entry B.7) and on the GPU side in the function "getCXResult" (defined on Line 173 in Appendix B Entry B.15).

The reader must note that both the single qubit tailed tensor product and "CX" gate tailed tensor product do not perform any multiplications, instead each function simple calculates which indices to insert elements of the matrix M into the result matrix based on the diagonal nature of the Identity matrix product.

Reordering the Statevector

We are now equipped with a method to drastically reduce the number of calculations we need to construct full gate-matrices. However, if we review Equation 6.12 and simplify it to Equation 6.14 we can see that the qubit that we are applying the gate M to must be in the last position of the tensor product used to form the statevector $|q_0q_1 \dots q_{n-1}q_n\rangle$. In essence we must design a way such that every time we want to use the tailed tensor product, our statevector is ordered such that the qubit(s) being operated on by M are in the last position of the statevector tensor product (see Section 6.1.4).

$$I\,|q_0\rangle \otimes I\,|q_1\rangle \otimes \dots \otimes I\,|q_{n-1}\rangle \otimes M\,|q_n\rangle = I \otimes I \otimes \dots \otimes I \otimes M\,|q_0q_1 \dots q_{n-1}q_n\rangle \tag{6.14}$$

We compute the statevector in the function "tensorProduct" on Line 607 of "BaseTypes.h" (included in Appendix B Entry B.4). In this operation we effectively apply the tensor product in an arbitrary order dictated by iterating through the provided qubit map (Line 609). We do store the order which we perform this tensor product in the "positions_" variable (defined on Line 480 of "BaseTypes.h").

To solve the problem of reordering the tensor product to apply the tailed gate matrices, we have written the function "reorder" (Line 700 in Appendix B Entry B.4). This function accepts the order that is desired (dictated by the tailed gate matrix order), and reapplies the modified tensor product (defined on Line 630 in Appendix B Entry B.4) which reorders the current statevector according to the requested order. The algorithm for this is given in Algorithm 9.

Algorithm 9: Algorithm for reordering a statevector for a new tensor product order

```
Result: vector<complex> reorderedStateVector
input: oldStateVector, newOrder, oldOrder;
for position in reorderedStateVector do
    elementOfEachQubit = [];
    for element in newOrder do
        elementOfEachQubit.append(inverseTail(totalNoElements, element, newORder));
    end
    positionInOldState = calculate where elementOfEachQubit was in oldStateVector;
    reorderedStateVector[position] = oldStateVector[positionInOldState];
end
return reorderedStateVector;
```

Using the reordered statevector obtained from the "reorder" function we can apply the tailed gate matrix (detailed in Section 6.1.5).

Once the application is complete, we would like to return the order from the reordered state back to our original order, since this is what the user will be expecting when they see a readout of the qubit state in Visual-Q. To perform this we have defined the function "reconcile" (Line 707 in Appendix B Entry B.4). Which effectively performs Algorithm 9 but mapping the reordered statevector back to the original order.

The Statevector data-structure is now versatile enough to be used in full "statevector" compute mode operations, faithfully representing quantum computing operations.

6.1.5 Quantum Processor

We have seen so far that the "QuantumCircuit" hold all of the calculations that need to be completed and that the "StateVector" and "qubitMap" data-structures hold the state of the qubits in "statevector" and "fast" compute modes respectively. We must now consider how to efficiently and accurately complete the matrix products that will modify the states of qubits in the circuit, this process is completed by the "CPUQuantumProcessor" (defined on Line 78 in Appendix B Entry B.6) for CPU operations and "GPUQuantumProcessor" (defined on Line 77 in Appendix B Entry B.14) for GPU operations.

CPU Processing

The CPU Quantum processor class has a couple of auxilliary functions. The "loadCircuit" function (defined on Line 88 in Appendix B Entry B.6) accepts the completed "QuantumCircuit". It is important to note, that the "QuantumCircuit" has primitive gate matrices, that is to say that they are of dimension 2×2 in the case of a U gate and 4×4 in the case of the CX gate. We have seen in Section 6.1.4, that if Valkyrie is in "statevector" compute mode we will need to perform a tailed tensor product. The functions to perform these tailed tensor product are also defined in the "CPUQuantumProcessor" class in the functions "getCXResult" (Line 82 of "CPUDevice.h") and "getGenericUResult" (Line 83 of "CPUDevice.h").

Fast compute mode

When operating in "fast" compute mode, the "calculate" function of the "CPUQuantumProcessor" class is called, this function is defined on Line 89 of the file "CPUDevice.h" (included in Appendix B Entry B.6) and implemented on Line 240 of the file "CPUDevice.cpp" (included in Appendix B Entry B.7). This function simply iterates through the calculations presented in the "QuantumCircuit" and performs the matrix multiplication with the primitive gate matrix. As discussed in Section 6.1.4 we attempt to preserve entanglement relations using the "StateVector" data-structure from the results of the "fast" computation. The pseudocode for the "calculate" function is given by Algorithm 10.

Algorithm 10: Algorithm for performing gate operations on qubit states in "fast" operation mode

```
Result: modifiedStateVector
input: stateVector, qubitMap, QuantumCircuit;
for calculation in QuantumCircuit do
    if calculation.gate.isCXGate() then
        localTensorProduct = tensorProduct(qubitMap[calculation.qubit1],
          qubitMap[calculation.qubit2]);
        resultantState = calculation.gateMatrix multiply localTensorProduct;
        modify stateVector with resultantState;
        update qubitMap with resultantState;
    else
        resultantState = calculation.gateMatrix multiply qubitMap[calculation.qubit];
        update qubitMap with resultantState;
        refresh stateVector;
    end
end
return stateVector;
```

Statevector compute mode

In "statevector" compute mode, the "CPUQuantumProcessor" uses the "calculateWithStateVector" function, this function is defined on Line 90 of the file "CPUDevice.h" (included in Appendix B Entry B.6) and implemented on Line 290 of the file "CPUDevice.cpp" (included in Appendix B Entry B.7). Since we already have the functions we need to perform the tailed tensor product the calculation function is quite simple. Algorithm 11 outlines this.

Algorithm 11: Algorithm for performing gate operations on qubit states in "statevector" operation mode

```
Result: modifiedStateVector
input: stateVector, QuantumCircuit;
for calculation in QuantumCircuit do
    sv = get(statevector);
    sv.reorder(calculation.newOrder);
    if calculation.gate.isCXGate() then
        gate = getCXResult(sv.numberOfQubits);
    else
        gate = getGenericUResult(sv.numberOfQubits, calculation.primitiveGate);
    end
    sv = gate multiply sv;
    sv.reconcile(oldOrder);
end
return stateVector;
```

GPU Processing

The "GPUQuantumProcessor" is defined on Line 77 of the file "GPUDevice.cuh" (included in Appendix B Entry B.14). Like the "CPUQuantumProcessor" this class defines the tail tensor production functions which work the same way in both GPU and CPU cases. The calculation functions in the "GPUQuantumProcessor" have similar setup to the CPU versions, however the matrix multiplication itself is very different because it is applied on the Graphical Processing Unit.

Fast compute mode The function "calculate" is defined on Line 88 of "GPUDevice.cuh" (included in Appendix B Entry B.14) and implemented on Line 244 of "GPUDevice.cu" (included in Appendix B Entry B.15). The basic algorithm in terms of datastructure unwrapping and storage for the "GPUQuantumProcessor" can be seen in Algorithm 10. The "**multiply**" step of this algorithm is where the graphical processing unit comes into play.

The "calculate" function calls a function named "calculateGPU" defined on Line 85 of "GPUCompute.cuh" (included in Appendix B Entry B.12). The "calculateGPU" function must perform a lot of memory management to ensure that the GPU is being correctly controlled and to prevent any memory leaks. We can see examples of this from Lines 112 to 126 of "GPUCompute.cuh", GPU memory must be carefully handled since it is the same memory that is used to display on your screen. Mishandling of this memory can affect performance of your GPU until restart.

Parallelising the matrix computation for "fast" compute mode is explained in Algorithm 12. Note that in fast compute mode we are only dealing with matrices of size 2×2 and 4×4, this leads to minimal parallelisation.

Algorithm 12: Algorithm for performing GPU gate matrix multiplication in "fast" operation mode

```
Result: resultantQubitState
input: qubitStateBefore, gateMatrix;
if gateMatrix.dim == 2 then
    launch 2 GPUThread(gateMatrix.row multiply qubitStateBefore);
    store result in resultantQubitState;
else
    launch 4 GPUThread(gateMatrix.row multiply qubitStateBefore);
    store result in resultantQubitState;
end
return resultantQubitState;
```

Statevector compute mode The function "calculateWithStateVector" is defined on Line 89 of "GPUDevice.cuh" and implemented on Line 296 of "GPUDevice.cu". As with "fast" compute mode the basic algorithm for data-structure handling and statevector updating is giving by Algorithm

11. However, the "**multiply**" function can now be highly parallelised since we are dealing with matrices of dimension $2^n \times 2^n$ where n is the number of qubits in the circuit.

The "calculateWithStateVector" function calls either the "calculateGPUSV" function (defined on Line 313 of "GPUCompute.cuh") or the "calculateGPULargeSV" (defined on Line 227 of "GPUCompute.cuh"). The reason we have two functions is because there are two ways to parallelise the matrix multiplication.

The first way to parallelise GPU multiplication is to multiply and sum each row of the gate matrix in parallel. The pseudocode for this process is given by Algorithm 13.

Algorithm 13: Algorithm for performing GPU gate matrix multiplication for small matrices in "statevector" compute mode

```
Result: resultantStateVector
input: qubitStateBefore, gateMatrix;
launch gateMatrix.row.dim GPUThread func (row, stateVector)
    (
        oldStateVector = stateVector;
        res = 0;
        for element in row do
            res += element × oldStateVector[element];
        end
        return res;
    );
store results in resultantStateVector;
return resultantStateVector;
```

After testing it was found that this method of parallelisation works well up until a statevector size of 256 (up to 8 qubits in the circuit). When the size of the statevector exceeds 256 elements, Valkyrie switches to the function "calculateGPULargeSV" which uses a second way to parallelise the matrix multiplication.

The second method for parallelisation is to launch a thread for every gate matrix element, which simply multiplies the correct element of the statevector that it needs to. The gate matrix has dimension $2^m \times 2^m$, in this method we launch $2^m \times 2^m$ threads on the GPU, one for each element. Each of threads store's it's result in an element of a $2^m \times 2^m$ result matrix. Finally we need to sum up each row of the result matrix to produce the resultant statevector of the computation. The pseudocode for this algorithm is given in Algorithm 14.

Algorithm 14: Algorithm for performing GPU gate matrix multiplication for large matrices in "statevector" compute mode

```
Result: resultantStateVector
input: qubitStateBefore, gateMatrix;
launch (gateMatrix.row.dim × gateMatrix.col.dim) GPUThread func (element,
  stateVector)
    (
    resultMatrix[element] = gateMatrix[element] × stateVector[element %
      gateMatrix.row.dim];
);
launch gateMatrix.row.dim GPUThread func (row, stateVector)
    (
    resultVector[row] = sum resultMatrix[row];
);
store results in resultantStateVector;
return resultantStateVector;
```

Since we calculate the complex product of each gate matrix element in parallel, we can see that "calculateLargeSV" is much more highly parallelised than "calculateSV". However, we make a tradeoff since we are launching two separate GPU processes in the second method and this comes with overhead. The tradeoff between the more parallel nature of "calculateLargeSV" and "calculateSV" balances out at a circuit size of 8 qubits.

6.1.6 Compute Device

We now have a collection of functions and data-structures which can perform the quantum operations we want to be able to process. To orchestrate these components we have defined the "AbstractDevice" class on Line 67 of "AbstractDevice.h" (included in Appendix B Entry B.3). These devices collect and connect the components we've discussed in the past sections.

The CPU implementation of an "AbstractDevice" is called "CPUDevice" and is defined on Line 96 of "CPUDevice.h" (included in Appendix B Entry B.6). The GPU implementation of an "AbstractDevice" is called "GPUDevice" and is defined on Line 95 of "GPUDevice.cuh" (included in Appendix B Entry B.14).

For both CPU and GPU implementations the entry point function for "fast" compute mode is "run" (Line 117 of "CPUDevice.h" and Line 116 of "GPUDevice.cuh"). While the entry point for "statevector" compute mode is "runSV" (Line 118 of "CPUDevice.h" and Line 117 of "GPUDevice.cuh").

Both "run" and "runSV" have very similar functionality, their pseudocode is given by Algorithm 15. The "CPUDevice" and "GPUDevice" both have the "getStateVector" function (Line 129 in "CPUDevice.h" and Line 128 in "GPUDevice.h"). This function is called by the measurement module to get the statevector.

Algorithm 15: Algorithm for running a user defined quantum circuit

```
Result: stateVector
input: definedRegisters, concurrentBlocks;
for register in definedRegisters do
    create qubits for register using qubitFactory;
end
for block in concurrentBlocks do
    load QuantumCircuit with block;
end
if $mode == "statevector"$ then
    calculateWithStateVector();
else
    calculate();
end
return stateVector;
```

6.1.7 Measurement

Measurement in quantum computers is a complex topic as presented by Jozsa in his "An introduction to measurement based quantum computation" [41]. For measurement in Valkyrie we use a fairly simple system which doesn't take into account decoherence characteristics of quantum circuits.

Statevector measurement

Both "fast" and "statevector" compute modes produce a "StateVector" at the end of the processing. This statevector is used in measurement. The "StateVectorMeasurement" class is defined on Line 47 of "Measurement.h" (included in Appendix B Entry B.19).

Measurement algorithm The "StateVector" provided to the "StateVectorMeasurement" class contains atleast one non-zero element. We firstly sum the square magnitude of all the elements in the vector. The procedure after this is to generate a random number between 0 and this sum of magnitudes. We then iterate through the statevector taking the cumulative sum of the square magnitude of elements. When the cumulative sum surpasses the random value generated we return the state where this surpassing occured as the state of the system upon measurement. This process is completed in the "measure" function implemented on Line 126 of "Measurement.cpp" (included in Appendix B Entry B.20). Pseudocode for this function is provided in Algorithm 16.

Algorithm 16: Measurement of the state vector

```
Result: state
input: statevector;
sum = 0;
for state in statevector do
    sum += state.magnitude()**2;
end
randVal = random(0, sum);
cumulative = 0;
for state in statevector do
    if state == last then
        return state;
    end
    cumulative += state.magnitude()**2;
    if cumulative > randVal then
        return state;
    end
end
```

Measurement commands

As detailed in Section 6.1.1 the user can request certain elements of the measure qubit state to be stored in classical registers. We have already parsed the measurement commands and can pass a list "MeasureCommand" data-structures into the "StateVectorMeasurement" which will tell the class which classical register to store the quantum states into.

This process is completed by the "loadMeasureCommands" function defined on Line 66 of "Measurement.h". This function stores and readies the measurement commands. When the function "passMeasurementsIntoClassicalRegisters" (defined on Line 67 of "Measurement.h") is called then then measured quantum state is passed into the classical registers.

6.1.8 Main function

The entrypoint of Valkyrie is the "kernel.cu" file (included in Appendix B Entry B.1). For this file the function called at the beginning is the "main" function defined on Line 114.

The first thing the main function does is parse the command line arguments, there are a couple of these arguments which provide different functionality and they are listed and explained below.

- **-gpuInfo**: Calls the "DisplayHeader" function, defined on Line 146 of "kernel.cu", displays GPU information for the user to check whether their hardware is configured properly.
- **-test**: Runs Valkyrie's test suite to ensure that any changes made had not affected the accuracy of valkyrie for quantum simulations.
- **-c**: Prepares Valkyrie for a CPU run, when this command line option is specified the main function runs the "CPURun" function (defined on Line 263 of "kernel.cu") and uses the CPU to perform quantum computations.
- **-g**: Prepares Valkyrie for a GPU run, when this command line option is specified the main function runs the "GPURun" function (defined on Line 274 of "kernel.cu") and uses the GPU to accelerate quantum computation.
- **-sv**: Run's Valkyrie in "statevector" compute mode, if this flag isn't specified Valkyrie runs in "fast" compute mode.

- **-json**: Prints out a json parsable string which can be understood by the Visual-Q interface.
- **-o <filename>**: Specifies which QASM file to parse and process.
- **-time <spec>**: Requests Valkyrie complete a timing run of the code given by "filename", if a "spec" is specified Valkyrie will time a specific section of the execution as given by the "spec".

Running Valkyrie

Once all the command line options are parsed, we can set up Valkyrie for the run which is done between Lines 137 and 142. This triggers the parsing stage which processes on the file specified in the command line argument "-o".

6.2 Optimising Valkyrie

Experiment 2, which is discussed in Section 7.2 revealed that Valkyrie at this point in development has very poor performance characteristics when circuit complexity increases. Further investigation in Section 7.2.5 advises on which parts of computation take up the most execution time.

6.2.1 Optimising Statevector reordering

As discussed in Section 6.1.4, to perform the tail algorithm for a particular gate application, we must apply the tailed tensor product to a statevector which is constructed with the qubit(s) in question as the last elements in the tensor product. This can be seen in Equation 6.14.

To achieve this we have defined Algorithm 9 which has a complexity of O(2^{2N}) where N is the number of qubits in the circuit. Since the statevector itself is a datastructure of size 2^N this complexity is reasonable and can be treated as O(M^2) relative to the statevector. However, when considering the results of Experiment 2 in Section 7.2 in particular Figure 7.12, it is clear to us that the reordering process is taking far longer than we expect. Furthermore, considering that each gate application in un-optimised Valkyrie "statevector" compute mode requires a 2^{2N} complex multiplications it seems implausible that the reordering of the tensor product should take double the time that the gate multiplication takes.

After investigation we can find the source of this additional computational complexity on Line 641 of "BaseTypes.h" (included in Appendix B Entry B.4), the "mapToOldScheme" has a complexity of O(2^{2N}) itself. This means the overall complexity of our current implementation of the reordering Algorithm 9 is O($2^N \times 2^{2N}$) which is O(2^{3N}) with respect to the number of qubits in the circuit or O(M^3) relative to the size of the statevector.

We have found a more optimised method, which is included in the Optimised file "BaseTypes.h" featured in Appendix B Entry B.5. The function "getOldSchemeValues" in this optimised file uses a map between the positions of elements in the old order and position of elements in the new order to reduce the complexity of the reordering process back down to O(2^{2N}) relative to the number of qubits. We should note that we used another $O(2^{3N})$ process in the "reconcile" function on Line 707 of Appendix B Entry B.4, fortunately the same fix was applicable to convert this to an $O(2^{2N})$ process and can be seen from Line 728 of Appendix B Entry B.5.

6.2.2 Optimising Valkyrie Execution

While the reordering of the statevector does take up the largest proportion of time in Figure 7.12, the other two components of execution; constructing the gate matrices and the matrix multiplication still take up a large amount of time. The inspiration for the next, and arguably most crucial step of Valkrye's optimisation came from Equation 6.10.

We don't actually need to construct the full gate matrix, or compute the full matrix multiplication. What should have been staring us in the face, was that by adopting the tailed tensor product, we build a gate matrix with the large majoirty of elements being zero. This is clearly seen in Equation 6.10, the only non-zero elements are in the form of the primitive gate matrix (of the U gate (2×2) or CX gate (4×4)) along the leading diagonal. All other elements of the gate matrix are zero.

We must at this point appreciate the utility of documenting our development process, since without the explicit writing out of Equation 6.10 we may have never been aware of this simple truth. Quite often in software development, we can become caught up in the abstractness of the requirements of a codebase without acknowledging the practicalities of it's operation.

Equipped with the knowledge that the tailed tensor product allows us to skip even the construction of the full matrix product. Instead, we can create a "virtualised" representation of the full gate matrix. This is to say, since each element of the output vector of a matrix multiplication only depends on one row of the gate matrix, if we are able to calculate which elements of the primitive gate matrix are in that row and which elements of the initial state-vector are affected by those elements, we can perform the full matrix calculation with far exponentially fewer calculations and without constructing the full gate.

To explain this we will run through the example of the circuit presented in Figure 6.5.

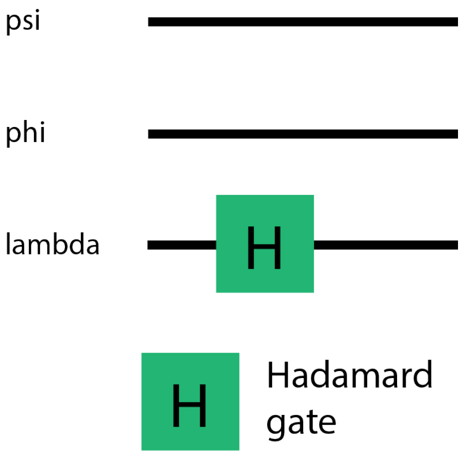


Figure 6.5: Simple circuit to help demonstrate tailed tensor product optimisations

The mathematical representation of this circuit is given in Equation 6.15.

$$(I \otimes I \otimes H)\,|\psi\phi\lambda\rangle = NewState \tag{6.15}$$

The expanded form of this equation is given in 6.16.

$$\begin{array}{c}0\\1\\2\\3\\4\\5\\6\\7\end{array}\begin{bmatrix}
\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 & 0 & 0\\
\frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0\\
0 & 0 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0\\
0 & 0 & 0 & 0 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}}\\
0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}}
\end{bmatrix} \cdot
\begin{array}{c}|000\rangle\\|001\rangle\\|010\rangle\\|011\rangle\\|100\rangle\\|101\rangle\\|110\rangle\\|111\rangle\end{array}
\begin{bmatrix}
c_0^\psi c_0^\phi c_0^\lambda\\
c_0^\psi c_0^\phi c_1^\lambda\\
c_0^\psi c_1^\phi c_0^\lambda\\
c_0^\psi c_1^\phi c_1^\lambda\\
c_1^\psi c_0^\phi c_0^\lambda\\
c_1^\psi c_0^\phi c_1^\lambda\\
c_1^\psi c_1^\phi c_0^\lambda\\
c_1^\psi c_1^\phi c_1^\lambda
\end{bmatrix} =
\begin{bmatrix}
sv_0\\ sv_1\\ sv_2\\ sv_3\\ sv_4\\ sv_5\\ sv_6\\ sv_7
\end{bmatrix} \tag{6.16}$$

We note that to calculate elements sv_i we need to multiply row i of the $I \otimes I \otimes H$ matrix with the old state vector. When we further consider that row i of the matrix only ever has two non-zero elements. We simply need to calculate which elements of the primitive matrix occupy those non-zero elements. If we also calculate which positions of the old state vector are multiplied by these elements we can calculate easily calculate the value of sv_i.

For example, to calculate element sv_2 we need to multiply row 2 of the matrix with the old statevector $|\psi\phi\lambda\rangle$. We can see in Equation 6.16, that this only requires two multiplications with $\frac{1}{\sqrt{2}} \times c_0^{\psi} c_1^{\phi} c_0^{\lambda}$ and $\frac{1}{\sqrt{2}} \times c_0^{\psi} c_1^{\phi} c_1^{\lambda}$.

Notably, we no longer need to store the full gate matrix, we just need to calculate which element of the 2×2 primitive matrix we need to multiply for a given element. The code to selectively compute this can be found in the optimised copy of "CPUDevice.cpp" on Line 247 (included in Appendix B Entry B.8) and for the GPU implementation we perform this optimised compute on Line 54 of the optimised copy of "GPUCompute.cuh" (included in Appendix B Entry B.13).

This implementation of gate matrix multiplication represents a huge performance uplift especially as circuit complexity increases. We can calculate the margin of this performance uplift. Under the old scheme, with a circuit with N qubits we would need to allocate space for a $2^N \times 2^N$ gate matrix. Thanks to the tailed tensor product, we don't have to do much work to calculate each element of this matrix. Therefore this matrix creation process was an O(2^{2N}) process. We then needed to compute the matrix product between this elaborated gate matrix and the old statevector. This process will require $2^N \times 2^N$ operations giving O(2^{2N}). Overall this process is O($2^{2N} + 2^{2N}$) = O(2^{2N+1}) for computational complexity.

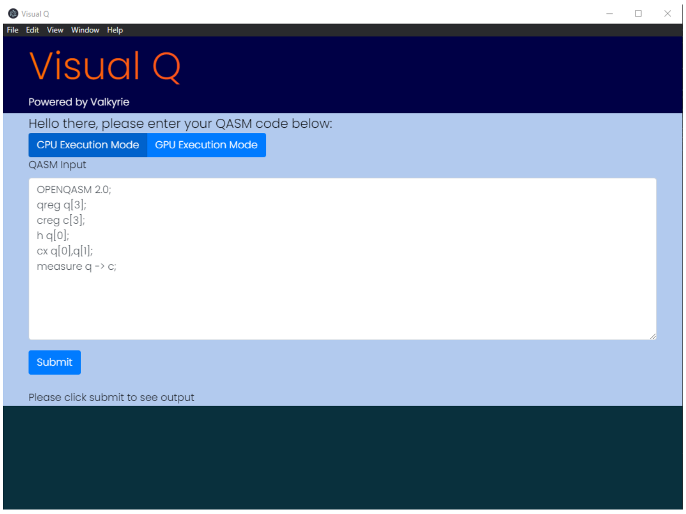
Under the new scheme, we do not need to compute the full gate matrix at all, instead considering we only have two primitive gate types: U and CX with matrix dimensions 2×2 and 4×4 respectively. For each element of the output statevector we need to perform at most 4 calculations. Therefore, the compelxity of the gate matrix multiplication drops to O(4×2^N) which is equivalent to O(2^{N+2}). This is a significant improvement over the O(2^{2N+1}).

6.2.3 Results of optimisation

As detailed further in Section 7.3 Experiment 3 explores the effects of our optimisations. It is detailed that these optimisations have had transformative effects on Valkyrie's performance, for example when running Deutsch Jozsa Algorithm N=10 we saw a 90.3% reduction in execution time.

6.3 Visual Q

Due to personal circumstances, Visual-Q is not as well developed as I might have hoped. However, it provides a simple and accessible interface for Valkyrie, and allows users to dynamically program OpenQASM code.



Hello there, please enter your QASM code below:

```
OPENQASM 2.0;
qreg q[3];
creg c[3];
h q[0];
cx q[0],q[1];
measure q -> c;
```

Please click submit to see output

Figure 6.6: Visual Q landing page, allowing users to directly write OpenQASM code

Figure 6.6 shows the landing page for Visual-Q. Visual-Q automatically loads up the last code that the user had written. This allows for a simple form of persistence between different programming sessions. In the case of Figure 6.6, we can see that the OpenQASM code loaded is:

```
OPENQASM 2.0;
qreg q[3];
creg c[3];
h q[0];
cx q[0],q[1];
measure q -> c;
```

We have seen this circuit in Equation 1.27, and we are aware that it should lead to an entanglement between qubit $q[0]$ and qubit $q[1]$. The easiest way to represent this entanglement is to show to the user the final statevector representing the overall state of the qubits in the circuit.

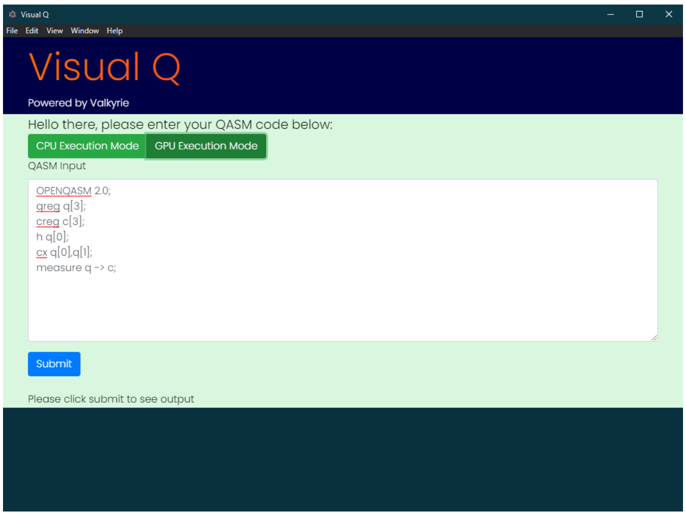
6.3.1 Execution

To execute code on Valkyrie the user simply has to click "Submit", which triggers Visual-Q to send the entered code to Valkyrie. The backend code for this can be found on Line 81 of Appendix C Entry C.1 which starts a subprocess for Valkyrie to start operating.

Figure 6.7 shows the radio-buttons which allow users to switch between CPU and GPU execution modes.

Figure 6.7: Visual Q exeuction mode switch button, which allows users to select which processor to execute their code on

When the user switches to "GPU" execution mode, Visual-Q lights up green to symbolise the switch to GPU execution mode as seen in Figure 6.8. The switch to green is to echo the colour palette that Nvidia uses, since the GPU execution is operating on Nvidia's CUDA architecture [42].



Hello there, please enter your QASM code below:

```
OPENQASM 2.0;
qreg q[3];
creg c[3];
h q[0];
cx q[0],q[1];
measure q -> c;
```

Please click submit to see output

Figure 6.8: Visual Q GPU execution mode colour scheme

Once the user selects their execution mode, they can write the QASM code they'd like to simulate. The user can then select the "Submit" button to execute the code.

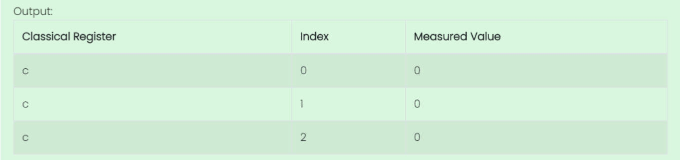
6.3.2 Results

Visual Q presents the results of the calculation in two ways. Firstly we present the results of the statevector itself. It must be noted that on an actual quantum computer we can never measure the quantum state of the circuit, but we can always calculate it. Figure 6.9 shows the statevector presentation for the quantum circuit we defined.

State	Quantum State
000	0.707107 + 0.000000i
001	0.000000 + 0.000000i
010	0.000000 + 0.000000i
011	0.000000 + 0.000000i
100	0.000000 + 0.000000i
101	0.000000 + 0.000000i
110	0.707107 + 0.000000i
111	0.000000 + 0.000000i

Figure 6.9: Visual Q Statevector presented results

The second way is by displaying a "measured" set of results. This represents the partially random measurement of the statevector to give a settled state of the circuit. This measured result can be seen in Figure 6.10.



Output:

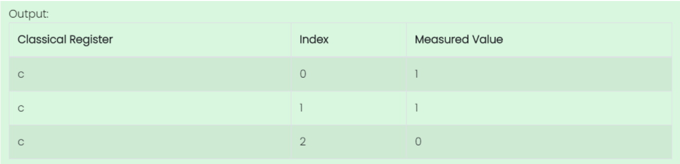
Classical Register	Index	Measured Value
c	0	0
c	1	0
c	2	0

Figure 6.10: Visual Q Measured results presentation

In Figure 6.10 we see what results have been transferred into register "c" from the register "q" which was measured. Register "q" has three qubits, which have been mapped directly into register "c"'s three classical bits. The statevector we have seen so far represents the combined state of of the register "q" now we have measured the statevector we can once again consider the qubits as individual entities who's values have been transferred to register "c".

We can see here that the only non-zero entries of the quantum state were in $|000\rangle$ and $|110\rangle$, this means that the states of the first two qubits are intimately linked and if qubit 0 is in state x then qubit 1 will be measured to be in state x as well. This measurement is confirmed by Figure 6.10 which shows that the qubits 0 and 1 are both in the same state. By coincidence qubit 2 is also in that state.

Since both states $|000\rangle$ and $|110\rangle$ have the same quantum value, we can run the circuit again and we may get a different result as shown by Figure 6.11. Once again in this qubits 0 and 1 are in the same state however qubit 2 is not. This shows that quantum circuits have to be carefully designed to actually give a useful result. Most quantum circuits result in inconclusive results with multiple quantum states holding non-zero values and therefore being candidates for measurement.



Output:

Classical Register	Index	Measured Value
c	0	1
c	1	1
c	2	0

Figure 6.11: Visual Q Measured results presentation showing randomness in the measurement

Chapter 7

Evaluation

Summary

In this section we complete a series of experiments, which compare Valkyrie against it's contemporaries. In these experiments we carefully consider a breakdown of how timings were distributed throughout Valkyrie's operation. Furthermore, we consider how robust and reliable Valkyrie's operation is in comparison to contemporaries. In addition we discover that in it's initial state Valkyrie performs poorly in complex scenarios, and we go on to prove the success of optimisations we have implemented to address the lack of performance at scale.

One of our core objectives is to make a more efficient quantum computer emulator as mentioned in Section 2.1. To evaluate whether we have achieved this we must run simulations on circuits of varying complexity and compare the execution times to those of competing quantum computer simulators. We must also analyse second order statistics on the execution timings to see how robust and reliable the performance of Valkyrie is in comparison to other quantum computer simulators.

For this comparison we have used Valkyrie in CPU and GPU mode with both statevector and fast compute modes, as well as IBM's Qiskit [9] and Google's Cirq [33]. This will give us multiple sources of comparison to compare performance. Our standard format for running tests will be to initially perform 20 runs of the circuit on the relevant simulator to ensure no unusual behaviour, then once we are satisfied with these runs we perform 100 simulations on each simulator to collect statistics. These statistics will be presented in the form of tables, histograms and pie charts. Raw datasets are available in Appendix D.

For reference, the hardware these simulations were completed on is listed below.

```
Motherboard:
    Name:                       Asus ROG Strix x570F
CPU:
    Name:                       AMD Ryzen 5900x
    Physical Core Count:        12
    Thread Count:               24
    Base Clock:                 3.7 GHz
    Max Boost Clock:            4.8 GHz
    Total L2 Cache:             6MB
    Total L3 Cache:             64MB
    CMOS:                       TSMC 7nm FinFET
Memory:
    Name:                       Crucial Ballistix RGB
    Capacity:                   2x16GB Dual Channel
    Clock Speed:                3200 MHz
GPU:
    Name:                       Nvidia RTX 3080
    Base Clock:                 1440 MHz
```

```
        Boost Clock:                1710 Mhz
        VRAM:                       10 GB GRRD6X
        CUDA Cores:                 8704
    Drive running simulations:
        Storage technology:         Solid State
        Interface:                  NVME PCIE 4.0
        Name:                       Sabrent Rocket NVME 4.0
```

The software configuration used to run these simulations is listed below.

```
    Operating system:
        Name:                       Windows 10 Education 64-bit
        Build number:               19042.985
    GPU Driver:
        Name:                       Geforce Game Ready Driver
        Build number:               465.89
    Python:
        Version:                    3.9.4
```

For all experiment theoretical analysis we will assume a CPU clock of 3.7 GHz and GPU clock of 1.44 Ghz.

7.1 Experiment 1: Baseline circuit

The baseline circuit is a very simple circuit and is presented in Figure 7.1.

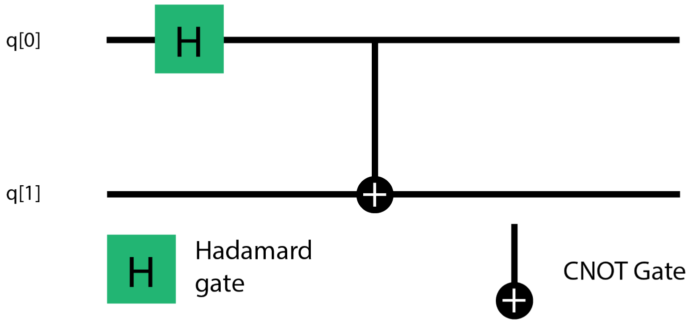


Figure 7.1: Baseline circuit for establishing basic statistics on different Quantum Simulators

The OpenQASM [9] code required to perform this circuit is presented below.

```
OPENQASM 2.0;
qreg q[3];
creg c[3];
h q[0];
cx q[0],q[1];
measure q -> c;
```

7.1.1 Circuit analysis

This circuit involves a very basic set of quantum operations. Therefore we can walk through the entire calculation. Firstly we instantiate three qubits which will be default start in the state 0. This is represented in Equation 7.1.

$$|q[0]\rangle = \begin{bmatrix}1\\0\end{bmatrix}, |q[1]\rangle = \begin{bmatrix}1\\0\end{bmatrix}, |q[2]\rangle = \begin{bmatrix}1\\0\end{bmatrix} \tag{7.1}$$

When considering a full statevector calculation for this process we must form a full statevector, which involves taking the tensor product of all these qubit states as shown by Equation 7.2.

$$|q\rangle = |q[0]q[1]q[2]\rangle = \begin{array}{c} |000\rangle \\ |001\rangle \\ |010\rangle \\ |011\rangle \\ |100\rangle \\ |101\rangle \\ |110\rangle \\ |111\rangle \end{array} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{7.2}$$

We then apply a Hadamard gate to qubit q[0] and Identity gates to the remaining qubits as shown in Equation 7.3.

$$H|q[0]\rangle \otimes I|q[1]\rangle \otimes I|q[2]\rangle = H \otimes I \otimes I|q[0]q[1]q[2]\rangle \tag{7.3}$$

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

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

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

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

$$= \begin{array}{c} |000\rangle \\ |001\rangle \\ |010\rangle \\ |011\rangle \\ |100\rangle \\ |101\rangle \\ |110\rangle \\ |111\rangle \end{array} \begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ 0 \\ \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{7.4}$$

From Equation 7.4 we have the state of the circuit after the first Hadamard gate in Figure 7.1.

The second gate is a "CX" gate between q[0] and q[1] for which the tensor product is given in Equation 7.5.

$$CX|q[0]q[1]\rangle \otimes I|q[2]\rangle = CX \otimes I|q[0]q[1]q[2]\rangle \tag{7.5}$$

Since we are well acquainted with the tensor product we will simply state the resultant matrix of this tensor product in Equation 7.6

$$CX \otimes I\,|q[0]q[1]q[2]\rangle = \begin{bmatrix} 1&0&0&0&0&0&0&0\\ 0&1&0&0&0&0&0&0\\ 0&0&1&0&0&0&0&0\\ 0&0&0&1&0&0&0&0\\ 0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&1\\ 0&0&0&0&1&0&0&0\\ 0&0&0&0&0&1&0&0 \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}}\\ 0\\ 0\\ 0\\ \frac{1}{\sqrt{2}}\\ 0\\ 0\\ 0 \end{bmatrix} \tag{7.6}$$

The result of this last matrix mulitplication is given in Equation 7.7.

$$\begin{array}{c} |000\rangle\\ |001\rangle\\ |010\rangle\\ |011\rangle\\ |100\rangle\\ |101\rangle\\ |110\rangle\\ |111\rangle \end{array} \begin{bmatrix} \frac{1}{\sqrt{2}}\\ 0\\ 0\\ 0\\ 0\\ 0\\ \frac{1}{\sqrt{2}}\\ 0 \end{bmatrix} \tag{7.7}$$

We now have the expected result of this computation. Valkyrie in statevector mode will produce this result, as will Qiskit and Cirq. Valkyrie in fast compute mode will product the result seen in Equation 7.8, to see an explanation for this please consult Section 6.1.4.

$$\begin{array}{c} |000\rangle\\ |001\rangle\\ |010\rangle\\ |011\rangle\\ |100\rangle\\ |101\rangle\\ |110\rangle\\ |111\rangle \end{array} \begin{bmatrix} \frac{1}{2}\\ 0\\ 0\\ 0\\ 0\\ 0\\ \frac{1}{2}\\ 0 \end{bmatrix} \tag{7.8}$$

In statevector compute mode the minimum number of calculations can be broken down into 5 sections as given by the list below.

- **Statevector tensor product**:

  $$8 \textbf{ possible states} \times 2 \textbf{ complex multiplications per state} = 16 \textbf{ complex multiplications}$$

- **Hadamard gate tensor product**: Using the results of Equation 6.7.

  $$\frac{4}{3}(4^n - 1) - 4 \times 1 \textbf{ complex multiplications per element} = 80 \textbf{ complex multiplications}$$

- **Hadamard matrix application**:

  $$8 \textbf{ rows} \times 8 \textbf{ elements in each row} \times 1 \textbf{ complex multiplications per element} =$$
  $$64 \textbf{ complex multiplications}$$

- **CX gate tensor product**: Using the results of Equation 6.7.

  $$\frac{4}{3}(4^n - 1) - 4 \times 1 \textbf{ complex multiplications per element} = 80 \textbf{ complex multiplications}$$

- **CX matrix application**:

$$8 \textbf{ rows} \times 8 \textbf{ elements in each row} \times 1 \textbf{ complex multiplications per element} =$$

$$64 \textbf{ complex multiplications}$$

In total we will be performing at the very least 304 complex multiplications and a complex multiplication taking at most 4 cpu core cycles [40]. We expect to use at the least 1216 CPU core cycles, assuming a CPU core clock of 3.4 GHz the minimum time it would take for execution of this circuit will be **357657** nanoseconds.

7.1.2 Results

The raw result data for the baseline circuit is provided in Appendix D under Table D.1 for the initial 20 iterations and Table D.2 for the full 100 iteration run.

Table 7.1 represents the summary statistics for the results obtained from this experiment.

Simulator	Valkyrie				Qiskit	Cirq
Processor	CPU		GPU		CPU	CPU
Mode	Statevector	Fast	Statevector	Fast	NA	NA
Mean (microseconds)	4005.332	3368.347	5551.452	4526.015	29617.091	6991.965
Variance	2.86E+10	5.58E+10	1.11E+11	7.47E+10	1.1E+13	1.7E+12
Skew	1.809	6.599	1.471	1.516	0.672	-0.196

Table 7.1: Summary statistics for the baseline circuit tests on competing quantum computer simulators

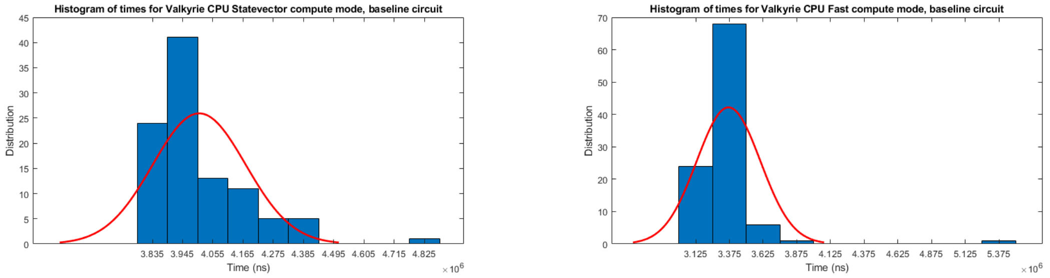
The tabulated results in Table 7.1 provide us with a number of interesting results. Firstly that Valkyrie is in absolute terms faster than Qiskit and Cirq when tasked with simulating the baseline circuit. With Qiskit taking a particlularly long time of 29.6 milliseconds on average to simulate the circuit. Cirq is must closer to Valkyrie performance running the calculation in 6.99 milliseconds. As expected the "fast" compute mode for Valkyrie ran the fastest at 3.37 milliseconds on the CPU and 4.53 milliseconds on the GPU. The more accurate "statevector" compute mode completed the execution in 4.01 milliseconds on the CPU and 5.55 milliseconds on the GPU.

When we consider that Valkyrie on the CPU is faster than Valkyrie on the GPU on the baseline circuit, we must keep in mind that sending operations to the GPU comes with a lot of overhead in terms of transferring memory and waiting for the GPU scheduler to assign compute time. Therefore, for a relatively small calculation as the baseline circuit is, it makes sense that the GPU overhead makes it slower than the CPU on average. We expect, that as the complexity of circuits rise we will see performance on the GPU that surpasses that of the CPU as the overhead becomes a smaller portion of the execution.

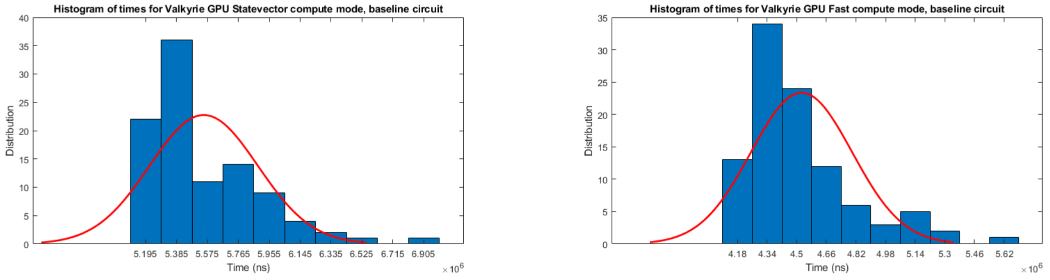
We have generated histograms for the simulations we have run. These histograms are included in Figure 7.2.

Figure 7.2a shows that the execution times have a strong negative skew with one outlier at around 4.8 milliseconds. It is important to consider that these outliers will occur since the computer may be momentarily busy with another task leading to slower than usual execution. This means that there is a strong possibility that the user might encounter some of these outlier times depending on what's running on their computer. As expected Figure 7.2b shows that "fast" compute mode leads to fast average execution times, what is notable is that "fast" execution mode has a tighter distribution implying that it produces a more reliably faster circuit execution. This is expected since it performs fewer operations.

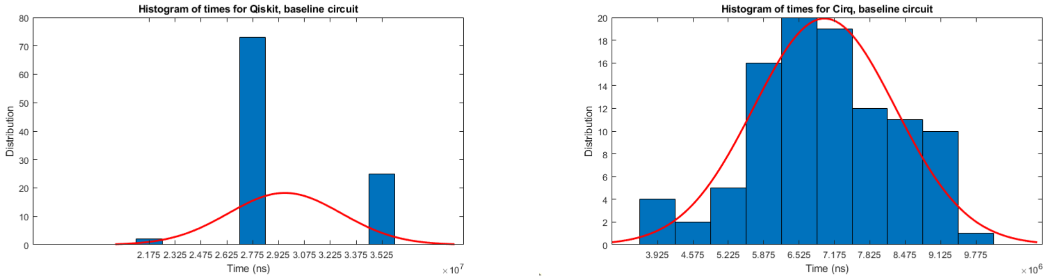
Figure 7.2c shows that much like in CPU mode, Valkyrie GPU mode has fast execution, however the results seem to have a larger variance which is confirmed by Table 7.1. This can be explained by the GPU being physically in a different location to where the program is executing. This physical difference means that there are more steps in the processing pipeline in terms of overhead including software drivers and physical silicon controllers. All of these additional steps in the compute



(a) Histogram for Valkyrie CPU in statevector mode **(b)** Histogram for Valkyrie CPU in fast mode



(c) Histogram for Valkyrie GPU in statevector mode **(d)** Histogram for Valkyrie GPU in fast mode



(e) Histogram for Qiskit **(f)** Histogram for Cirq

Figure 7.2: Histograms for the distribution of execution times for various Quantum simulators

pipeline can lead to more variance for GPU execution. As shown by Figure 7.2d, in fast execution mode Valkyrie is faster since it has fewer multiplications to perform, the distribution for GPU in "fast" mode is once again quite wide lending support to our theory of higher variance due to more steps in the processing pipeline.

Figure 7.2e shows a very unusual distribution. To check if this was correct, this experiment on the qiskit run was repeated several times but produced very similar results each time. We can consider that while the distribution is quite wide, it is quite consistent with a large central pillar in terms of execution times. The large variation might be attributed to Python's interpreter oriented programming model and possibly the CPU being busy in the same way mutliple times during that run. Potentially a subroutine in the python interpreter runs in a particular way under certain circumstances.

Finally, Figure 7.2f shows that the Google Cirq package has quite a spread distribution of results instark contrast to Qiskit's distribution. As seen in the skewness of the distribution in Table 7.1 this distribution is almost unbiased implying this distribution is likely caused by background tasks that the CPU was running interrupting or not interrupting the execution leading to this close to Gaussian distribution.

Table 7.2 shows a direct performance comparison between Valkyrie's various execution modes, Qiskit and Cirq. We can see that we do pay a small penalty for accuracy as "statevector" mode is 15.9% slower than "fast" mode and we expect this gap to grow with circuit complexity. The GPU modes are slower but comparable with Valkyrie CPU and we expect that gap to close with circuit complexity increase. We can see Qiskit is significantly slower than expected, this is likely due to the higher overhead that Qiskit circuits carry since they are not only fully accurate but also come with utility functions such as diagram generation. Cirq is quite close to Valkyrie GPU in terms of

Simulator	Valkyrie				Qiskit	Cirq
Processor	CPU		GPU		CPU	CPU
Mode	Statevector	Fast	Statevector	Fast	NA	NA
Mean (ns)	4005332	3368347	5551452	4526015	29617091	6991965
Additional time (%)	0	-15.9034	38.6015	12.9997	639.4415	74.5664

Table 7.2: Table comparing how much slower other simulators are than Valkyrie in Fast CPU mode

absolute speed, however we expect that gap to widen as circuit complexity increases.

The reader might at this point be wondering whether Valkyrie should be faster, since it is running on fully compiled C++ code and we'd expect a slightly higher performance bump than 74% to Google Cirq, especially since at smaller scales overhead should be a larger factor and we expect python to contain the most overhead. To investigate this we have timed individual sections of Valkyrie's compute stack, the raw results are provided in Table D.3. Figure 7.3 displays a pie chart breakdown of how execution time is distributed across varying stages of the Valkyrie pipeline in CPU mode while Figure 7.4 does the same for GPU mode. Figure 7.3 is particularly telling, it shows that 75% of execution-time when running on the CPU in the baseline circuit is spent parsing the OpenQASM code. This portion is relatively unavoidable since parsing is a computationally expensive process. The percentage is a little lower for GPU mode at only 55%, however this is due to the execution taking longer itself. We must also consider that neither Qiskit or Cirq need to parse any code, since they are accessed directly via a python API. This shows that Valkyrie does make some sacrifices to be a standalone program and that this comes with significant parsing overhead.

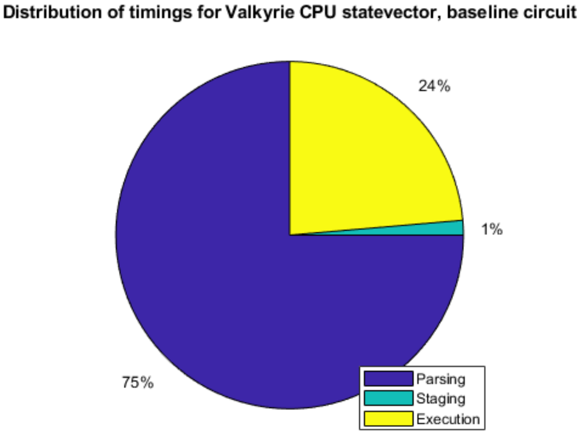


Figure 7.3: Pie chart showing distribution of execution time for Valkyrie CPU running the baseline circuit

On the other hand, we expect this parsing overhead to be relatively constant for more complex circuits, so this shows that Valkyrie is promising when we start considering more complex circuits which take more execution over parsing.

7.1.3 Conclusion

The baseline circuit is quite a simple circuit, it is meant to establish what timings we can expect from each simulator. However, we can already see that Valkyrie is a faster simulator than the two most widely used quantum computer simulators lending credit to the claim that Valkyrie provides faster quantum computer esmulation on consumer grade hardware. We have also seen that Valkyrie does have significant parsing overhead which should become less of a time factor when more complex circuits are considered. Furthermore, we can see that for simple circuits

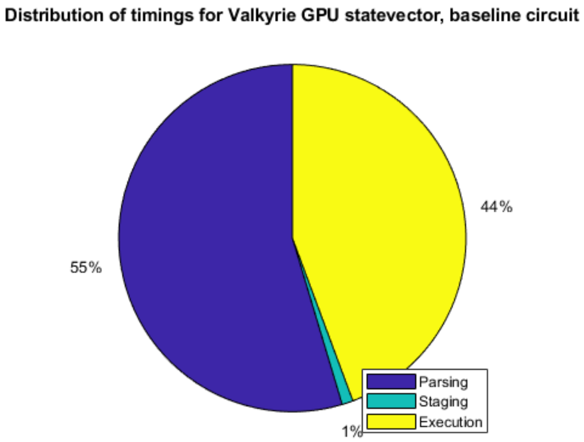


Figure 7.4: Pie chart showing distribution of execution time for Valkyrie GPU running the baseline circuit

Valkyrie is faster in CPU mode and more reliable, however we expect this to change as circuit complexity increases.

Another conclusion we can draw from this experiment is that since Valkyrie has to parse OpenQASM code it is at a disadvantage to the competing simulators. This is because both Qiskit and Cirq have been accessed via API's. Since both Qiskit and Cirq have OpenQASM parsing capability we will use those modes for the rest of the experiments to ensure a level playing field.

7.2 Experiment 2: Deutsch Jozsa Algorithm with an un-optimised Valkyrie

As mentioned in Section 6.1 we can currently run CPU based Quantum Algorithm Simulations. To this effect I have collected a couple of results for the Deutsch-Jozsa algorithm from *Valkyrie*, Qiskit and Cirq across different function complexities. Furthermore, I have included both single threaded and multi-threaded results from *Valkyrie*.

7.2.1 Deutsch-Jozsa Algorithm

As covered briefly in Section 3.3.2, the Deutsch-Jozsa algorithm is a prime example of an algorithm where a quantum approach provides a remarkable speedup over a classical approach. Furthermore, it has some element of scalability with respect to the dimensionality of the equation 3.2.

We are able to simulate the Deutsch-Jozsa algorithm's approach on our set of quantum computer emulators. Before moving onto the results, let us review the time complexity of the Deutsch-Jozsa algorithm when running on a quantum computer simulator. We note that the *Valkyrie* stage 1 stack, as it currently is, uses a universal gate set of $U(\theta, \phi, \lambda)$ single qubit rotation gates and CNOT two-qubit gates. I have implemented the set of Pauli gates [1] using U rotation gates for simplicity.

For translation of the Deutsch-Jozsa algorithm into a quantum circuit supported by the quantum computer emulators we are using, I found IBM's explanation of the Deutsch-Jozsa algorithm in Qiskit very useful [43]. Further to the mathematical explanation, IBM provide an excellent circuit diagram to illustrate Deutsch-Jozsa on a $f\{0,1\}^3 \longrightarrow \{0,1\}$ function, which I have included here to aid with my explanation.

An important point of note is that the diagram illustrates a balanced function being analysed.

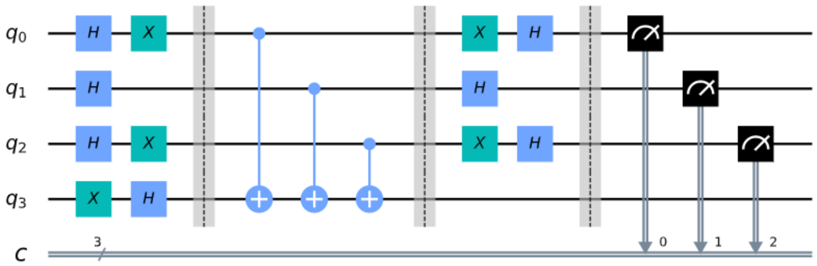


Figure 7.5: Circuit diagram of $N = 3$ Deutsch-jozsa algorithm on a Quantum Computer (from [43])

7.2.2 Complexity Analysis of Deutsch-Jozsa on Quantum Computer Emulator

To analyse the time complexity of running this algorithm on a quantum computer emulator we will firstly remind ourselves of the representation of Qubits, Hadamard Gates, Pauli X gates and CNOT gates.

$$\text{Qubit} \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} \tag{7.9}$$

$$\text{Two Qubits state} \begin{array}{c} |00\rangle \\ |01\rangle \\ |10\rangle \\ |11\rangle \end{array} \begin{bmatrix} c_0^{\psi} c_0^{\phi} \\ c_0^{\psi} c_1^{\phi} \\ c_1^{\psi} c_0^{\phi} \\ c_1^{\psi} c_1^{\phi} \end{bmatrix} \tag{7.10}$$

$$\text{Hadamard Gate} \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} \tag{7.11}$$

$$\text{Pauli X Gate} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{7.12}$$

$$\text{CNOT Gate} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{7.13}$$

Now that we are clear on the mathematical representation of the gates and qubits we are able to calculate the theoretical time complexity of Deutsch-Jozsa on our emulators. Let us look at a more general version of Figure 7.5.

We observe that for a function of the form $f\{0,1\}^N \longrightarrow \{0,1\}$ we need $N + 1$ qubits, this is because we need qM for our lower branch comparison.

Deutsch-Jozsa Phase 1 Complexity Analysis

In phase 1 of Deutsch-Jozsa, each qubit is treated with either a Hadamard gate and a Pauli X gate or just a Hadamard gate. We will assume a worst case scenario for complexity and assume that both gates are applied to all qubits. Since both the Hadamard gate and Pauli X gates are both single qubit gates, applying them to a qubit is in essence a simple matrix multiplication. An example is provided in equation 7.14.

$$\begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} \tag{7.14}$$

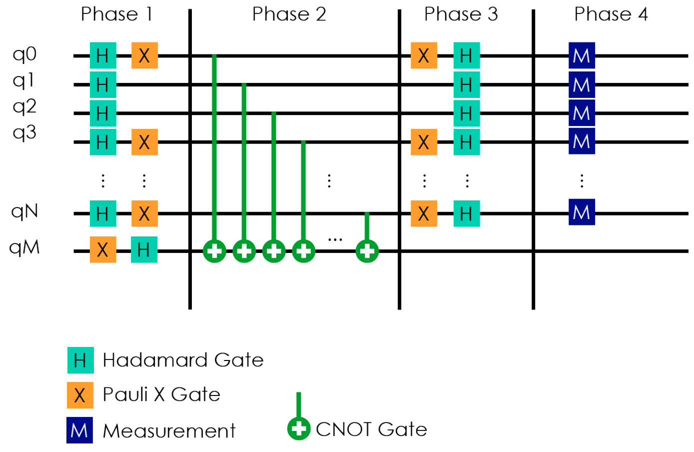


Figure 7.6: Circuit diagram of general Deutsch-jozsa algorithm on a Quantum Computer

However, as discussed in Section 5.2, we need a tensor product of multiple gate matrices to produce an appropriately sized gate matrix which can multiply the statevector. Hence, we must take into account this tensor product. We have already demonstrated how many calculations we need to calculate the tensor product in Equation 6.7. We have at maximum $2(N+1)$ of these gates in Phase 1.

In addition to this, we have to complete a matrix multiplication of the final matrix and the complete statevector which will require $(N+1)^2$ complex multiplications. Leading to the complexity given by Equation 7.15.

$$2(N+1) \times (\frac{4}{3}(4^{(N+1)}-1)-4+(N+1)^2) = 2(N+1)((\frac{4}{3}(4^{(N+1)}-1)-4)+(N+1)^2) \quad (7.15)$$

Deutsch-Jozsa Phase 2 Complexity Analysis

In phase 2 we have a more complex matrix calculation since the CNOT gate is a two qubit gate. We must combine the qubits in the way illustrated by Equation 7.10. We can still use Equation 6.7 and can conclude each gate will require $\frac{4}{3}(4^{(N+1)}-1)-4$ complex multiplications to acquire the tensor product. We perform the CX gate N times between every qubit q_i and q_m.

This is followed by the matrix product for each gate which required $(N+1)^2$ complex multiplications.

$$N(\frac{4}{3}(4^{(N+1)}-1)-4+(N+1)^2) \quad (7.16)$$

Deutsch-Jozsa Phase 3 Complexity Analysis

We notice the near symmetry between Phase 1 and 3, the only complexity difference is that the final qubit qM is not operated on in this phase. Hence we arrive at the complexity of:

$$2N \times (\frac{4}{3}(4^{(N+1)}-1)-4+(N+1)^2) = 2(N)((\frac{4}{3}(4^{(N+1)}-1)-4)+(N+1)^2) \quad (7.17)$$

Total complexity of Deutsch-Jozsa on a Quantum Computer simulator

We notice that phase 4 effectively has no calculations and so we will not consider it in our analysis of time complexity. Given the 3 complexities we have, we simply need to add them together to arrive at a total time complexity of:

$$(5N+2)((\frac{4}{3}(4^{(N+1)}-1)-4)+(N+1)^2) \quad (7.18)$$

Of course in (Big-O) notation this reduced to $O(4^N)$ which is quite computationally expensive but necessary for quantum computation. With the knowledge of this time complexity we can predict the theoretical minimal execution time on a fixed speed CPU for executing the Deutsch Jozsa algorithm on a quantum computer emulator. We will assume a CPU speed of 4 GHz

$$t_{dj} = \frac{(5N+2)((\frac{4}{3}(4^{(N+1)}-1)-4)+(N+1)^2)}{4\times 10^9}\text{ s} \tag{7.19}$$

We will use nanoseconds for mathematical convenience.

$$t_{dj} = \frac{(5N+2)((\frac{4}{3}(4^{(N+1)}-1)-4)+(N+1)^2)}{4}\text{ ns} \tag{7.20}$$

We will keep this equation in mind as a rough guide for the expected execution times of the algorithm. We of course must acknowledge that it would be almost impossible to achieve this time since the CPU of a computer has other tasks to do as a minimum.

Furthermore, our programs themselves have separate data-structure complexity and subroutines which will add to this time. We should note that if we were to run this algorithm on a quantum computer, it would be completed in a single time step.

7.2.3 Results

We will run simulations on Deutsch Jozsa algorithm for $N = 4$ to $N = 10$ which has given us a large dataset of varying circuit complexity to compare and constrast Valkyrie with it's contemporaries.

Deutsch Jozsa Algorithm with N = 4

The raw result data for this experiment can be found in Appendix D Section D.0.2. Please consult Table D.4 for the initial 20 iterations and Table D.5 for the full 100 iteration run. Table 7.3 represents the summary statistics for the results in this experiment.

Simulator	Valkyrie				Qiskit	Cirq
Processor	CPU		GPU		CPU	CPU
Mode	Statevector	Fast	Statevector	Fast	NA	NA
Mean (ms)	25.04	20.71	33.25	28.38	43.39	60.56
Variance (ms^2)	0.410	0.403	1.752	2.80	462	11.8
Skew	2.29	1.68	2.03	3.23	2.30	0.66

Table 7.3: Table summarising the performance metrics of running Deutsch Jozsa Algorithm with N=4 on Valkyrie, Qiskit and Cirq

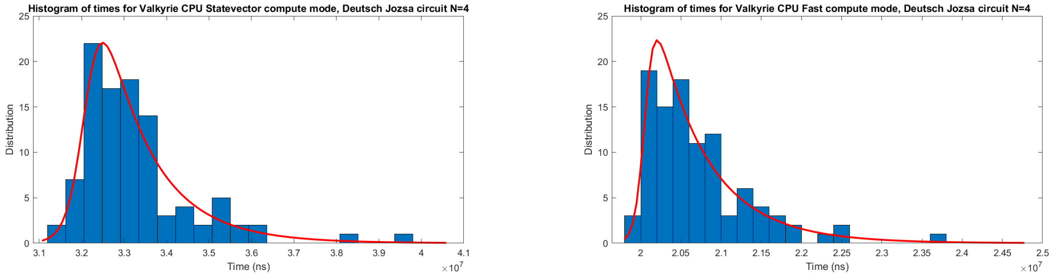
The results presented in Table 7.3 are promising, it shows that Valkyrie in all computation modes is faster than both Qiskit and Cirq. The margin of this performance improvement is better seen in Table 7.4.

Simulator	Valkyrie				Qiskit	Cirq
Processor	CPU		GPU		CPU	CPU
Mode	Statevector	Fast	Statevector	Fast	NA	NA
Mean (ms)	25.04	20.71	33.25	28.38	43.39	60.56
Relative mean (%)	0	-17.28	32.83	13.35	73.33	141.91

Table 7.4: Table summarising the relative performance of running Deutsch Jozsa Algorithm with N=4 on Valkyrie, Qiskit and Cirq

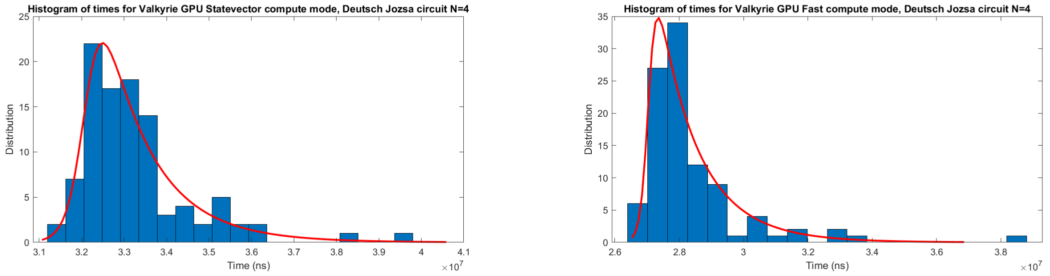
Table 7.4 shows that Valkyrie's performance margin has decreased significantly from Table 7.2. We will monitor the decrease in performance margin as we go upward in circuit complexity to attempt to discover the source of it.

Table 7.3 shows that Valkyrie is not only the fastest simulator but also has the most reliable performance, showing much lower variance than both Qiskit and Cirq. However, we can see that Valkyrie's results are quite heavily skewed especially in the case of GPU operation suing "fast" compute mode. We can inspect the histograms and distribution of this data for more detail in Figure 7.7.



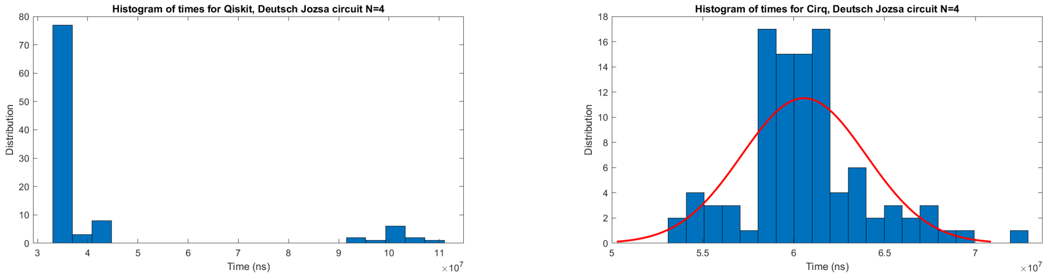
(a) Histogram for Valkyrie CPU in statevector mode

(b) Histogram for Valkyrie CPU in fast mode



(c) Histogram for Valkyrie GPU in statevector mode

(d) Histogram for Valkyrie GPU in fast mode



(e) Histogram for Qiskit

(f) Histogram for Cirq

Figure 7.7: Histograms for the distribution of execution times for various Quantum simulators with N = 4

Figure 7.7 shows a very similar distribution of results to Figure 7.2. That is to say positive skew on all datasets with particularly pronounced distributions for Valkyrie in CPU mode. This reinforces our view that Valkyrie produced more reliable performance results with a strong peak shown in the distributions of Figure 7.7a and Figure 7.7c.

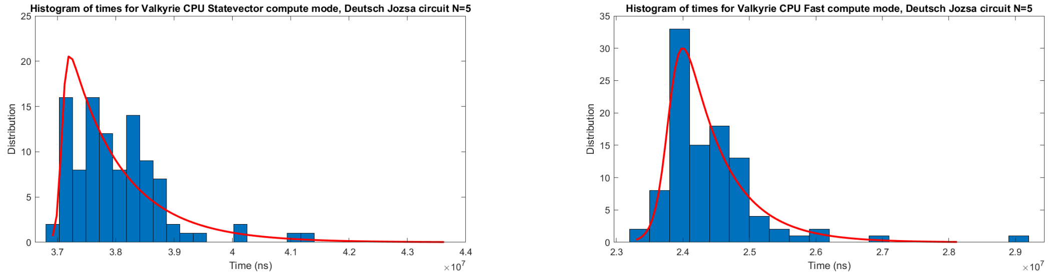
We once again observe very unusual split results for Qiskit in Figure 7.7e. We can hypothesize that Qiskit could be using some runtime optimisation that may not be available on occasion which leads to the extreme execution times. This split in results causes Qiskit's distribution curve to be very spread out leading to the large variance we observed in Table 7.3.

Using Equation 7.20 and $N = 4$ we obtain the minimum theoretical execution speed as given in Equation 7.21.

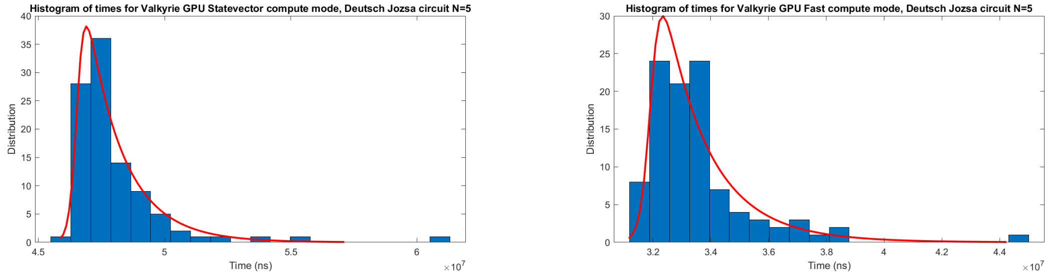
$$t_{dj} = \frac{(5N+2)((\frac{4}{3}(4^{(N+1)}-1)-4)+(N+1)^2)}{4} = 7617.5 \text{ ns} \tag{7.21}$$

We can clearly see that even our best efforts are a significant amount slower than this theoretical minimum.

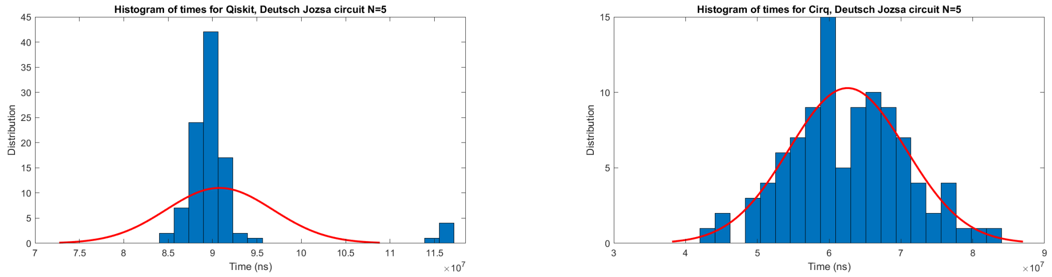
Deutsch Jozsa Algorithm with N = 5



(a) Histogram for Valkyrie CPU in statevector mode **(b)** Histogram for Valkyrie CPU in fast mode



(c) Histogram for Valkyrie GPU in statevector mode **(d)** Histogram for Valkyrie GPU in fast mode



(e) Histogram for Qiskit **(f)** Histogram for Cirq

Figure 7.8: Histograms for the distribution of execution times for various Quantum simulators with Deutsch Jozsa N = 5 circuit

The raw result data for this experiment can be found in Appendix D Section D.0.3. Please consult Table D.7 for the initial 20 iterations and Table D.8 for the full 100 iteration run. Table 7.5 represents the summary statistics for the results in this experiment.

Simulator	Valkyrie				Qiskit	Cirq
Processor	CPU		GPU		CPU	CPU
Mode	Statevector	Fast	Statevector	Fast	NA	NA
Mean (ms)	38.00	24.40	48.03	33.58	90.78	62.56
Variance (ms)	0.61	0.57	3.94	3.57	36.22	66.33
Skew	1.65	3.16	3.83	2.81	3.69	0.20

Table 7.5: Table summarising the performance metrics of running Deutsch Jozsa Algorithm with N=5 on Valkyrie, Qiskit and Cirq

The results of Deutsch Jozsa with $N = 5$ are quite similar to $N = 4$ from a Valkyrie perspective, CPU mode is still for now faster than GPU mode. However, some interesting results have been attained for Qiskit and Cirq. Qiskit sees far less variance in this circuit and Cirq sees significantly more variance. This might be indicative of different built in optimisations stepping in at different circuit complexities. We see very similar skew results as last time, with Cirq showing the most balanced results.

Table 7.6 shows the relative performance of Valkyrie compared with the other simulators for

Deutsch Jozsa's algorithm with N=5.

Simulator	Valkyrie				Qiskit	Cirq
Processor	CPU		GPU		CPU	CPU
Mode	Statevector	Fast	Statevector	Fast	NA	NA
Mean (ms)	38.00	24.40	48.03	33.58	90.78	62.56
Relative mean (%)	0	-35.79	26.37	-11.64	138.87	64.60

Table 7.6: Table summarising the relative performance of running Deutsch Jozsa Algorithm with N=5 on Valkyrie, Qiskit and Cirq

Interestingly Table 7.6 shows that Cirq's performance has surpassed Qiskit's. The performance gap between Cirq and Valkyrie is quickly closing and we will need consider the reasons for this.

We can see more clearly the distribution of results in Figure 7.8. An interesting result is that Valkyrie CPU in "statevector" mode has a wider spread in Figure 7.8a than in Figure 7.7a possibly indicating that there was some background processes the CPU was performing which caused these results to be more spread out than we expected.

Qiskit is once again displaying very unusual performance characteristics with a wide distribution with a small distribution of results at a slower execution time. Using Equation 7.20 we can calculate the theoretical fastest time to complete this circuit. The results of this can be see in Equation 7.22. And we are, once again quite far adrift from this minimum time.

$$t_{dj} = \frac{(5N+2)((\frac{4}{3}(4^{(N+1)}-1)-4)+(N+1)^2)}{4} = 37071.5 \text{ ns} \tag{7.22}$$

Deutsch Jozsa Algorithm with N = 6

The raw result data for this experiment can be found in Appendix D Section D.0.4. Please consult Table D.10 for the initial 20 iterations and Table D.11 for the full 100 iteration run. Table 7.7 represents the summary statistics for the results in this experiment. Table 7.7 shows a worrying result, the performance gap between Valkyrie and it's contemporaries is rapidly diminishing as we go up circuit complexity. We must consider the reasons for this, since we expect that Valkyrie should become faster as we increase scale. Another important conclusion we can draw from these results is that Valkyrie in CPU mode is consistently faster than GPU mode. We expected that by now the parallelism granted by GPU computation would have allowed this mode to surpass CPU computation however this hasn't happened either.

Simulator	Valkyrie				Qiskit	Cirq
Processor	CPU		GPU		CPU	CPU
Mode	Statevector	Fast	Statevector	Fast	NA	NA
Mean (ms)	88.44	35.17	100.45	47.08	106.25	109.06
Variance (ms)	22.15	0.82	4.83	4.39	33.70	139.00
Skew	-0.51	1.64	4.64	3.23	4.43	0.18

Table 7.7: Table summarising the performance metrics of running Deutsch Jozsa Algorithm with N=6 on Valkyrie, Qiskit and Cirq

We have contrasted the performance directly in Table 7.8. We can see that Valkyrie's performance margin between Valkyrie CPU and Qiskit has fallen from 138% with $N = 5$ (see Table 7.6) to only 20%. The performance gap to Cirq has fallen by a similar amount.

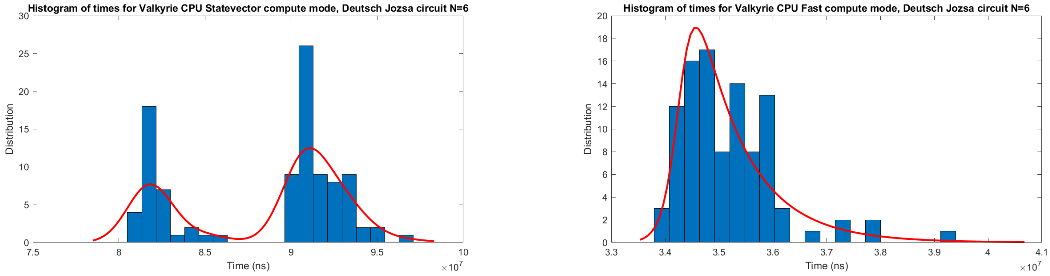
We can observe the distributions in Figure 7.9 to try and help discern a reason for this sudden fall in relative performance for Valkyrie. From Figure 7.9a we can see a sudden split in the results data with two time points showing very strong peaks and then a clear gap between them. Further to this, we can see that Valkyrie running in GPU mode now shows similar behaviour to Qiskit, with a small spike on the extreme right of the distribution.

A potential explanation of this unusual behaviour for Valkyrie in GPU mode is that now we are executing more complex calculations, the GPU scheduler might be having difficulty sometimes

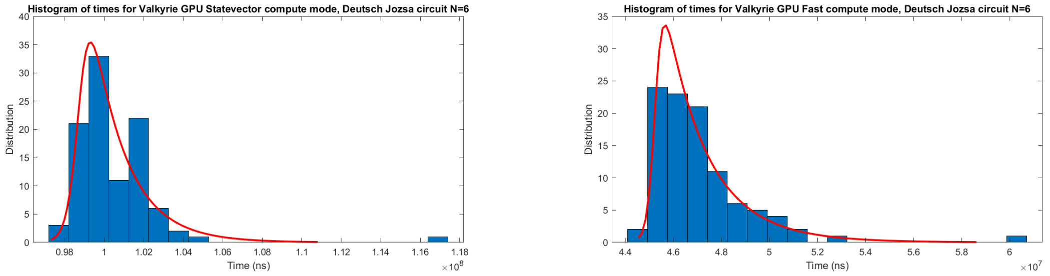
Simulator	Valkyrie				Qiskit	Cirq
Processor	CPU		GPU		CPU	CPU
Mode	Statevector	Fast	Statevector	Fast	NA	NA
Mean (ms)	88.44	35.17	100.45	47.08	106.25	109.06
Relative mean (%)	0	-60.23	13.57	-46.77	20.12	23.30

Table 7.8: Table summarising the relative performance of running Deutsch Jozsa Algorithm with N=6 on Valkyrie, Qiskit and Cirq

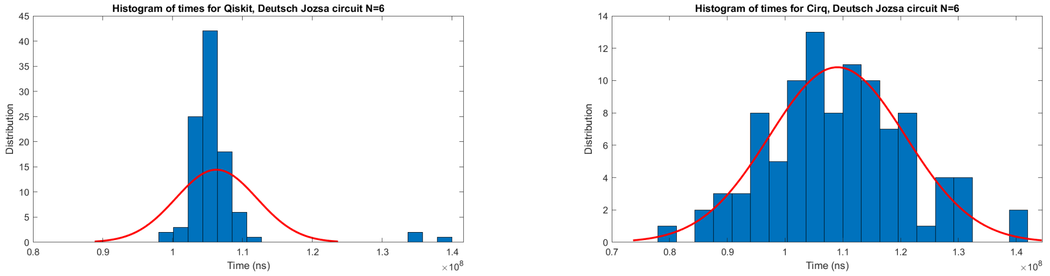
between handling our simulation and handling display tasks. The GPU is constantly updating the users display, for smaller calculations the GPU scheduler would likely have no issue finding computation slots between screen refreshes to carry out our simulation. However, as shown by Table 7.7 each statevector calculation on the GPU is now taking upward of 100 milliseconds the scheduler is likely having to delay our computation to carry out priority display tasks more often. This leads tot the late spike in GPU execution times.



(a) Histogram for Valkyrie CPU in statevector mode **(b)** Histogram for Valkyrie CPU in fast mode



(c) Histogram for Valkyrie GPU in statevector mode **(d)** Histogram for Valkyrie GPU in fast mode



(e) Histogram for Qiskit **(f)** Histogram for Cirq

Figure 7.9: Histograms for the distribution of execution times for various Quantum simulators with Deutsch Jozsa N = 6 circuit

From a theoretical standpoint using Equation 7.20 we can compute the minimum compute times required for this circuit to be calculated is as given by Equation 7.23.

$$t_{dj} = \frac{(5N+2)((\frac{4}{3}(4^{(N+1)}-1)-4)+(N+1)^2)}{4} = 175112 \text{ ns} \tag{7.23}$$

This is equivalent to 0.175 ms. Overall, this experiment is showing that Valkyrie may need some further optimisations since other simulators are catching up to Valkyrie in terms of performance. A trend which is continued in the next experiment.

Deutsch Jozsa Algorithm with N = 7

The raw result data for this experiment can be found in Appendix D Section D.0.5. Please consult Table D.13 for the initial 20 iterations and Table D.14 for the full 100 iteration run. Table 7.9 represents the summary statistics for the results in this experiment.

Simulator	Valkyrie				Qiskit	Cirq
Processor	CPU		GPU		CPU	CPU
Mode	Statevector	Fast	Statevector	Fast	NA	NA
Mean (ms)	171.83	41.04	199.05	52.14	132.37	132.22
Variance (ms)	9.87	0.58	13.96	44.10	120.12	5.12
Skew	0.88	1.05	2.32	-2.18	2.90	0.63

Table 7.9: Table summarising the performance metrics of running Deutsch Jozsa Algorithm with N=7 on Valkyrie, Qiskit and Cirq

The Deutsch Jozsa Algorithm with $N = 7$ confirms our worst fears, both Qiskit and Cirq operate faster than Valkyrie in the accurate Statevector compute mode, whether on CPU or GPU. This is confirmation of the trend we've seen from $N = 4$ to $N = 6$ and we have no reason not to expect it to continue. From this we can conclude that Valkyrie will need some more optimisations before we can consider it complete. The particular optimisations that we have done to rectify this are detailed in Section 7.2.4.

We can see how much performance we need to make up in Table 7.10. We can see that we will need to make up significantly more that 20% performance to close the gap to Qiskit and Cirq.

Simulator	Valkyrie				Qiskit	Cirq
Processor	CPU		GPU		CPU	CPU
Mode	Statevector	Fast	Statevector	Fast	NA	NA
Mean (ms)	171.8	41.0	199.0	52.1	132.3	132.2
Relative mean (%)	0	-76.11	15.83	-69.65	-22.96	-23.05

Table 7.10: Table summarising the relative performance of running Deutsch Jozsa Algorithm with N=7 on Valkyrie, Qiskit and Cirq

The distributions for $N = 7$ are very similar to those for $N = 6$ and are included in Appendix D Section D.0.5 under Figure D.1.

We expect Experiments with $N = 8, 9, 10$ to show a further erosion of Valkyrie's relative performance to Qiskit and Cirq. Using Equation 7.20 we can calculate the minimum computation time of this circuit to be 0.81ms. Which shows that theoretically it is possible to simulate this circuit much faster than we are doing, which motivates us further to find the optimisations to improve Valkyrie's performance.

Deutsch Jozsa Algorithm with N = 8

The raw result data for this experiment can be found in Appendix D Section D.0.6. Please consult Table D.16 for the initial 20 iterations and Table D.17 for the full 100 iteration run. Table 7.11 represents the summary statistics for the results in this experiment.

We can see from Table 7.11 further performance loss from Valkyrie, and the execution times seem to be scaling exponentially with circuit complexity whereas other simulators are managing a polynomial execution time scaling. We investigate this further in Section 7.2.4. Relative performance can be seen in Table 7.12.

As Table 7.12 shows we have seen three times worse relative performance than we saw with $N = 7$ as seen in Table 7.10. We can conclude that the relative performance loss is therefore not linear, and is at the very least polynomial if not exponential in N.

However, a silver lining from the $N = 8$ experiment is that for the first time GPU computation

Simulator	Valkyrie				Qiskit	Cirq
Processor	CPU		GPU		CPU	CPU
Mode	Statevector	Fast	Statevector	Fast	NA	NA
Mean (ms)	515.63	54.54	434.45	71.029	182.11	170.34
Variance (ms)	33.49	0.95	18.87	5.37	80.75	7.66
Skew	1.96	1.25	1.94	3.05	3.11	0.32

Table 7.11: Table summarising the performance metrics of running Deutsch Jozsa Algorithm with N=8 on Valkyrie, Qiskit and Cirq

Simulator	Valkyrie				Qiskit	Cirq
Processor	CPU		GPU		CPU	CPU
Mode	Statevector	Fast	Statevector	Fast	NA	NA
Mean (ms)	515.63	54.54	434.45	71.02	182.11	170.34
Relative mean (%)	0	-89.42	-15.74	-86.22	-64.68	-66.96

Table 7.12: Table summarising the relative performance of running Deutsch Jozsa Algorithm with N=8 on Valkyrie, Qiskit and Cirq

was completed faster than GPU execution. This implies that parallelism does provide performance benefits to Quantum circuit emulation at large enough scales. Since this is one of the core conjectures of this project we should note this result. On the other hand we have a growing performance deficit we will need to optimise to solve. The graphs of the distributions for $N = 8$ can be found in Appendix D Section D.0.6 under Figure D.2.

Deutsch Jozsa Algorithm with N = 9

The raw result data for this experiment can be found in Appendix D Section D.0.7. Please consult Table D.19 for the initial 20 iterations and Table D.20 for the full 100 iteration run. Table 7.13 represents the summary statistics for the results in this experiment.

Simulator	Valkyrie				Qiskit	Cirq
Processor	CPU		GPU		CPU	CPU
Mode	Statevector	Fast	Statevector	Fast	NA	NA
Mean (ms)	1389.30	75.70	1254.77	94.58	215.50	187.49
Variance (ms)	1742.54	2.16	278.44	4.99	923.34	12.56
Skew	-0.39	1.19	-0.64	1.87	1.71	-0.29

Table 7.13: Table summarising the performance metrics of running Deutsch Jozsa Algorithm with N=9 on Valkyrie, Qiskit and Cirq

This experiment shows conclusively that Valkyrie's performance without optimisation isn't acceptable for the project goals, the optimisations completed are discussed in Section 7.2.5 with the results presented in Section 7.2.4. The relative performance of the different quantum computer simulators is presented in Table 7.14.

Deutsch Jozsa Algorithm with N = 10

The raw result data for this experiment can be found in Appendix D Section D.0.8. Please consult Table D.22 for the initial 20 iterations and Table D.23 for the full 100 iteration run. Table 7.15 represents the summary statistics for the results in this experiment.

Simulator	Valkyrie				Qiskit	Cirq
Processor	CPU		GPU		CPU	CPU
Mode	Statevector	Fast	Statevector	Fast	NA	NA
Mean (ms)	1389.30	75.70	1254.77	94.58	215.50	187.49
Relative mean (%)	0	-94.55	-9.68	-93.19	-84.494	-86.50

Table 7.14: Table summarising the relative performance of running Deutsch Jozsa Algorithm with N=9 on Valkyrie, Qiskit and Cirq

Simulator	Valkyrie				Qiskit	Cirq
Processor	CPU		GPU		CPU	CPU
Mode	Statevector	Fast	Statevector	Fast	NA	NA
Mean (ms)	3729.24	131.04	3111.81	157.09	497.00	423.81
Variance (ms)	224.07	5.66	187.46	10.34	5933.37	41.96
Skew	0.174	2.80	0.79	1.33	1.19	-0.82

Table 7.15: Table summarising the performance metrics of running Deutsch Jozsa Algorithm with N=10 on Valkyrie, Qiskit and Cirq

Simulator	Valkyrie				Qiskit	Cirq
Processor	CPU		GPU		CPU	CPU
Mode	Statevector	Fast	Statevector	Fast	NA	NA
Mean (ms)	3729.24	131.04	3111.81	157.09	497.00	423.81
Relative mean (%)	0	-96.48	-16.55	-95.78	-86.67	-88.63

Table 7.16: Table summarising the relative performance of running Deutsch Jozsa Algorithm with N=10 on Valkyrie, Qiskit and Cirq

As we have seen as complexity has increased, Valkyrie's performance scales very poorly with performance. The reasons for this are discussed in Section 7.2.4 below.

7.2.4 Analysis of Performance degradation

As we scaled up complexity with Valkyrie we found a surprising result, which was that Valkyrie's performance scales very poorly with complexity. Since we have Equation 7.20 we can graph the performance of our quantum computer simulators against the complexity of the circuit. This can be seen in Figure 7.10.

We can infer a number of conclusions from this figure. Firstly, when Valkyrie is operating in "fast" execution mode whether on the CPU or GPU, we get very good performance scaling, as is evident in Figure 7.10 we see a linear relationship between increase in minimum compute time and actual "fast" mode execution time. This is a good result, fast computation mode is clearly gaining huge performance benefits over "statevector" mode even though it sacrifices accuracy in results.

The second conclusion we can draw us that Qiskit and Cirq must have some inbuilt optimisations over the basic approach that Valkyrie takes, since we can clearly see that both have an almost linear relationship between the log of their execution time and the log of the minimum execution time.

The third conclusion we can draw is that "statevector" compute mode scales very poorly with the minimum execution time. We can see a polynomial relationship between both Valkyrie CPU and GPU in "statevector" modes and the minimum compute time.

In Figure 7.11 we can see how the distribution of execution time changes as we progress through circuits of increasing complexity. It is evident from this that the "Execution" stage itself is where we can find most of the execution time being taken up as we move higher in circuit complexity.

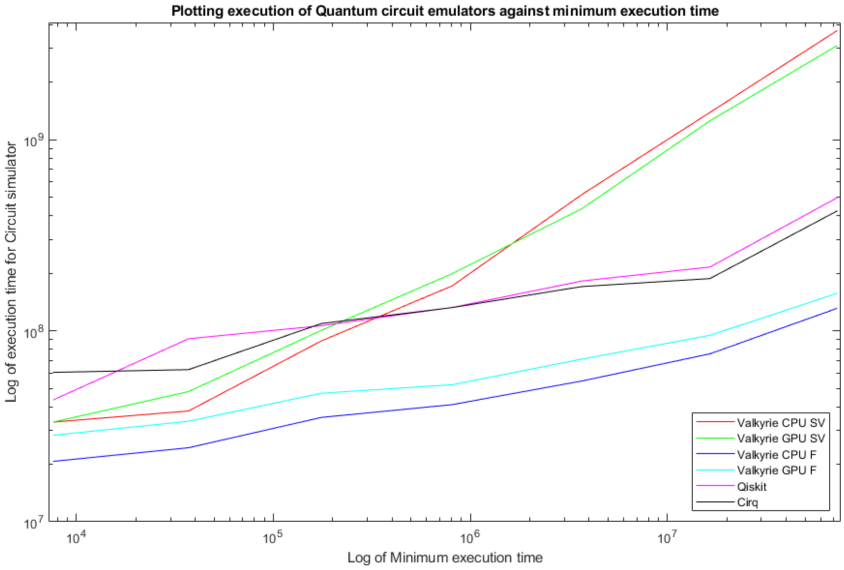
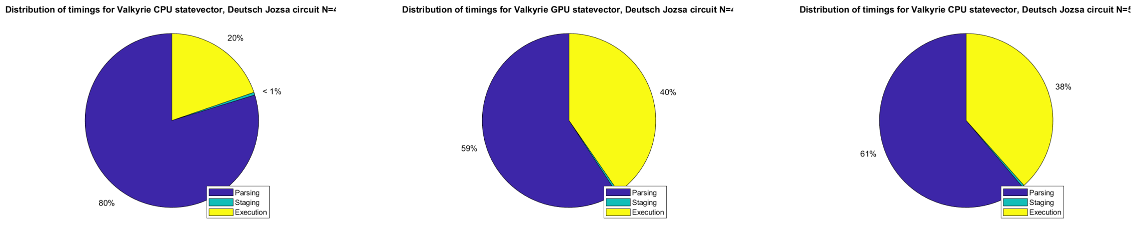
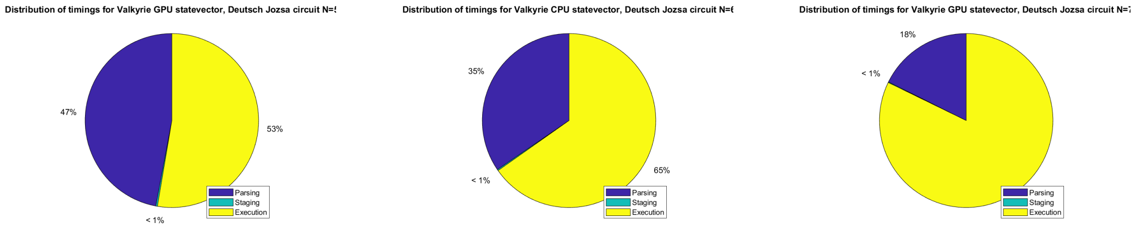


Figure 7.10: Comparison of execution times of various quantum circuit emulators as a function increasing circuit complexity plotted on a Log Log graph

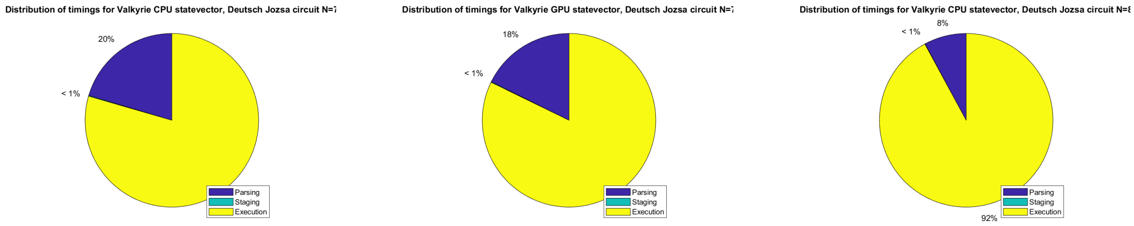
This at the very least means we know where to focus our efforts to optimise Valkyrie, we discuss exactly what areas for optimisation have been discovered in Section 7.2.5.



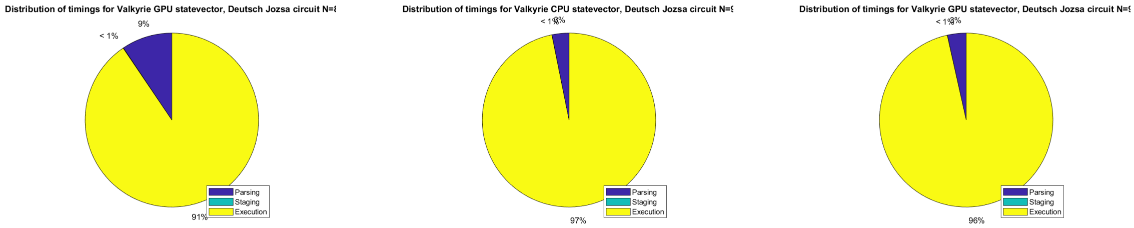
(a) CPU Mode N = 4 **(b)** GPU Mode N = 4 **(c)** CPU Mode N = 5



(d) GPU Mode N = 5 **(e)** CPU Mode N = 6 **(f)** GPU Mode N = 6



(g) CPU Mode N = 7 **(h)** GPU Mode N = 7 **(i)** CPU Mode N = 8



(j) GPU Mode N = 8 **(k)** CPU Mode N = 9 **(l)** GPU Mode N = 9

(m) CPU Mode N = 10 **(n)** GPU Mode N = 10

Figure 7.11: Pie charts showing how the distribution of compute time varies as circuit complexity increases

7.2.5 Searching for high computational complexity

Figure 7.10 makes it clear that we must heavily optimise Valkyrie especially in the case of larger circuits, and Figure 7.11 shows we will find the largest number of opportunities to do this in the execution block of Valkyrie itself.

In Section 6.1.4 we outlined how Valkyrie operates in statevector compute mode, from this section we can conclude that to carry out a single gate operation Valkyrie performs three essential functions as listed below:

- **Gate construction**: Valkyrie takes the primitive U gate (2×2) or CX (4×4) gate which constitutes all operations, and converts them into a gate correctly sized to operate on the statevector. Since the statevector has dimension $2^N \times 1$, we need to construct a gate of size $2^N \times 2^N$.
- **Re-ordering the statevector**: Valkyrie follows an algorithm we called the "tail" algorithm to reduce the number of calculations required to perform a tensor product. This algorithm requires us to re-order the statevector so that the qubit(s) being operated on are in the last position of the tensor product (see Section 6.1.4). This requires us to calculate this reordering.
- **Matrix multiplication**: We must multiply our "tailed" $2^N \times 2^N$ gate matrix by out $2^N \times 1$ statevector. This requires a 2^{2N} calculations. The exact method depends on whether Valkyrie is running in CPU or GPU mode.

We can time the time taken to complete each of these steps for Valkyrie CPU "statevector" running Deutsch Jozsa with N = 10 for which the raw data can be found in Appendix D Section D.0.9 in Table D.25. Figure 7.12 shows the results as a pie chart.

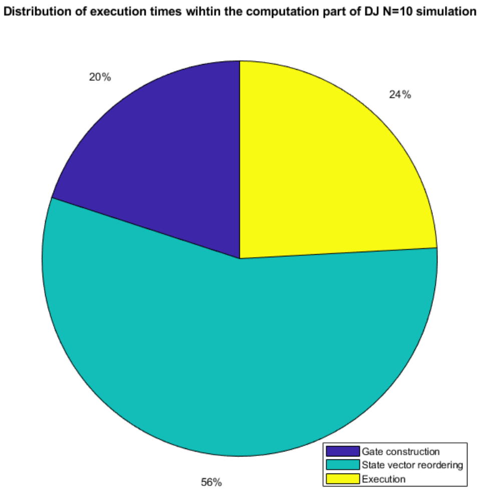


Figure 7.12: Distribution of execution times of the three stages of computation for Deutsch Jozsa Algorithm with N=10

Evidently the statevector re-ordering process must be optimised, the tail algorithm is meant to reduce computational complexity but if re-ordering the statevector takes this much time we may want to consider alternative options. We can also see that despite the simplicity of a tailed tensor product the construction time for gates still constitutes a significant percentage (20%) of the overall compute time for Valkyrie. We will want to optimise this as well. It is possible with optimisations to the matrix multiplication section will come more easily after the other sections are optimised.

7.2.6 Conclusion

The Deutsch Jozsa algorithm with varying N has revealed a lot of shortfalls of Valkyrie for circuits of higher complexity in terms of performance. During development, only simple circuits were used to compare and contrast performance leading to a false sense of security that Valkyrie was highly performant. This can be seen in Figure 7.10 where Valkyrie performs well for simplistic circuits.

On the other hand, Figure 7.10 also show that the promise of GPU accelerated quantum computation can be delivered. As even with an un-optimised compute process Valkyrie in GPU mode does eventually surpass the efficiency of CPU mode. Furthermore, Valkyrie consistently achieves more reliable timings with righter peaks that Qiskit the most comparable competitor. Google's Cirq package does a particularly outstanding job in consistency and a future goal for Valkyrie is to find a way to compete with Cirq on this metric.

Finally, we have also been able to analyse what steps in the computation in Valkyrie take up the most time. These steps are the target for the optimisations mentioned in Section 6.2 and we have endeavoured to repeat our Deutsch Jozsa circuit runs in Section 7.3. Which see's huge performance improvements.

It is clear to us that comprehensive experimentation is an important part of the development process for any software product. This vivid illustration of how important evaluation is, won't be easily forgotten and as proven in Section 7.3 this experiment in particular and it's conclusions have helped make Valkyrie faster.

7.3 Experiment 3: Deutsch Jozsa Algorithm with an optimised Valkyrie

In Experiment 2 covered in Section 7.2 we saw that Valkyrie's performance scaled very poorly as circuit complexity increased. In section 7.2.5 we identified sources of high computational complexity and we addressed these efficiency issues in Section 6.2.

We then ran simulations of the Deutsch Jozsa Circuit with $N = 4$ to $N = 10$ with Valkyrie optimised, this has lead to excellent performance results which are featured in full in Appendix D Section D.0.10. A summary of these results is presented in Table 7.17.

Simulator	Valkyrie unoptimised				Valkyrie optimised		Qiskit	Cirq
Mode	statevector		fast		statevector		NA	NA
Processor	CPU	GPU	CPU	GPU	CPU	GPU	CPU	CPU
N = 4	25.04	33.25	20.71	28.38	21.61	29.6	43.39	60.56
N = 5	38	48.03	24.4	33.58	26.88	36.17	90.78	62.56
N = 6	88.44	100.45	35.17	47.08	41.43	53.91	106.25	109.06
N = 7	171.83	199.05	41.04	52.14	76	71.55	132.37	132.22
N = 8	515.63	434.45	54.54	71.03	109.36	88.71	182.11	170.34
N = 9	1389.3	1254.77	75.7	94.58	177.98	150.48	215.5	187.49
N = 10	3729.24	3111.81	131.04	157.09	362.27	320.58	497	423.81

Table 7.17: Comparison of the mean execution times of different Quantum computer simulators for the Deutsch Jozsa algorithm with varying values of N

We can see that the simple optimisations covered in Section 6.2 has lead to substantial performance benefits and now see's Valkyrie even in "statevector" compute mode as faster than it's contemporary simulators at any circuit scale. Figure 7.13 compares the mean execution time of optimised Valkyrie against contemporary quantum circuit simulators.

We can see from Figure 7.13 shows Valkyrie outperforming Qiskit and Cirq even in "statevector" mode. Furthermore, Valkyrie in GPU is running faster than CPU mode showing that we are able to leverage the parallelisation that GPU's provide. Figure 7.14 compares the mean execution times of Valkyrie in "statevector" compute mode before and after optimisation.

An interesting observation we can make from Figure 7.14 is that the crossover point between

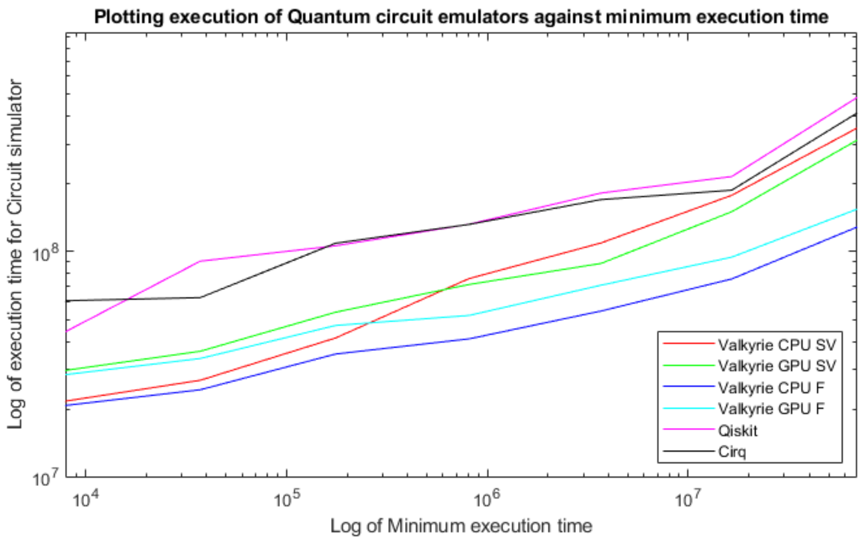


Figure 7.13: Comparing Execution times of optimised Valkyrie with other simulators

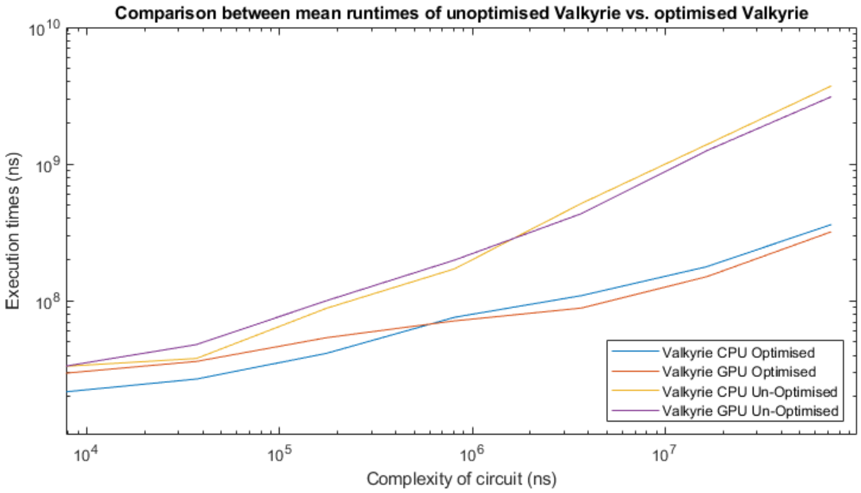


Figure 7.14: Comparing mean execution times of optimised Valkyrie with those of un-optimised Valkyrie

when Valkyrie running on the GPU's performance and the CPU performance has moved lower down in complexity as a result of our optimisations. This result could be explained by the fact that we no longer have to allocate GPU memory for a full gate matrix which itself will take some time. Furthermore, the fact that each parallel thread only has to perform 2 or 4 complex multiplications (see Section 6.2) significantly lightens the workload on the less complex CUDA cores [20].

Another obvious conclusion here, is that Qiskit and Cirq's implementations of quantum computer emulation scale very well, even with optimisation Valkyrie still sees a higher acceleration in execution times than Qiskit and Cirq. Overall the trajectory of these results suggest Valkyrie to still be the most performant even in "statevector" compute mode.

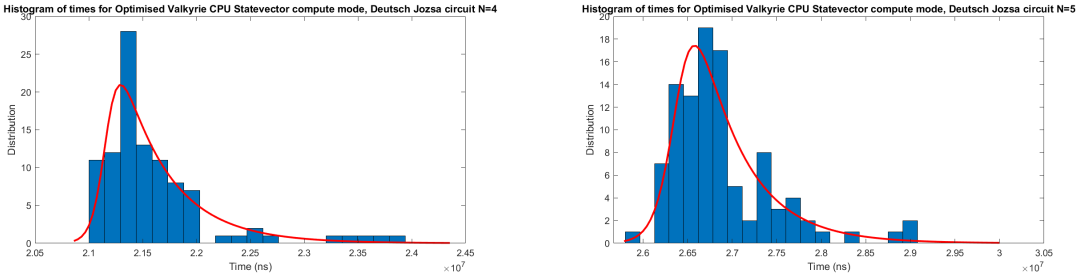
Figure 7.15 shows the distribution of execution times for our optimised Valkyrie running on the CPU. Throughout the figures we see an unusual progression of execution distributions. In Figure 7.15a we can see quite a well behaved distribution, with an slight negative skew. However, as we progress to Figures 7.15d and 7.15f we can see extremely polarised distributions with two strong peaks like we've seen from Qiskit distributions.

We can speculate that the cause of these polarised distributions is likely to be the CPU being busy with periodic tasks which affect a proportion of the result but not others. Figure 7.16 shows that the GPU execution times follow a more regular pattern showing negatively skewed Gaussian like distributions. We could suggest that since the GPU is not plagued with as many periodic tasks as the CPU it is able to produce more reliable execution times.

7.3.1 Conclusion

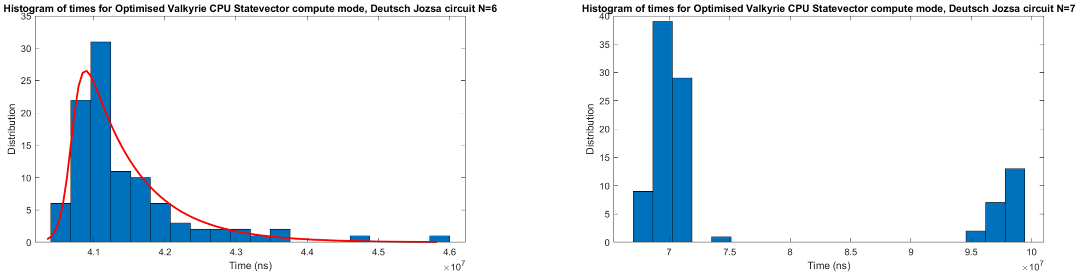
The conclusion of Experiment 3 is in stark contrast to that of Experiment 2. By implementing very simple optimisations to Valkyrie's execution stage, we have a quantum circuit simulator which performs well at all scales of circuit within the test. Furthermore, on the whole Valkyrie's execution times are quite reliable with the exception of CPU mode executing Deutsch Jozsa with N=7.

While we have not tested every scenario we are confident that Valkyrie is more performant than it's competitors. We anticipate that further optimisations can be performed and some of these are discussed in Section 8.



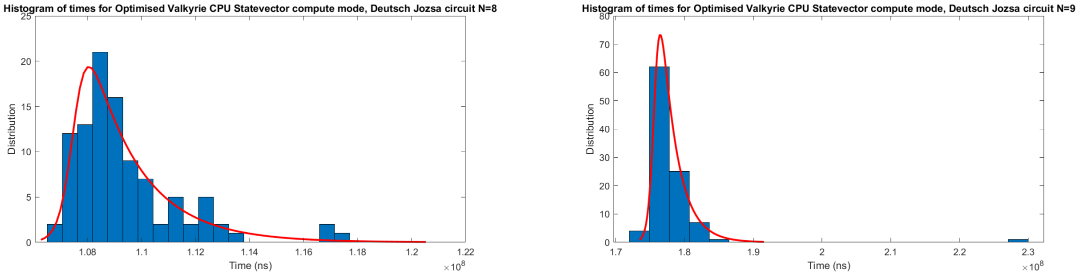
(a) Histogram for Optimised Valkyrie running on the CPU with N=4

(b) Histogram for Optimised Valkyrie running on the CPU with N=5



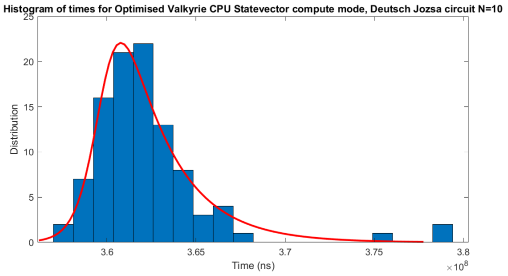
(c) Histogram for Optimised Valkyrie running on the CPU with N=6

(d) Histogram for Optimised Valkyrie running on the CPU with N=7



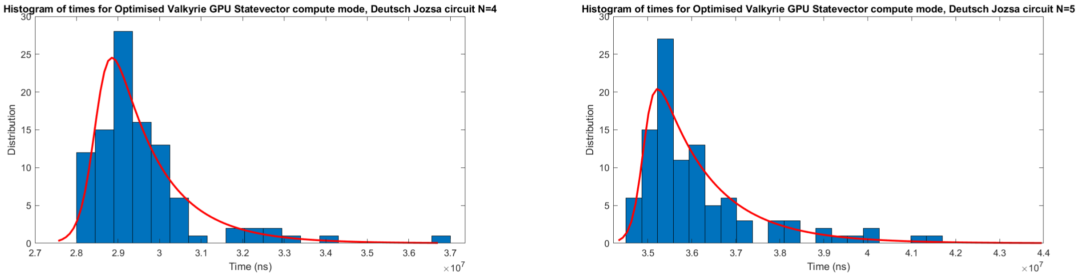
(e) Histogram for Optimised Valkyrie running on the CPU with N=8

(f) Histogram for Optimised Valkyrie running on the CPU with N=9



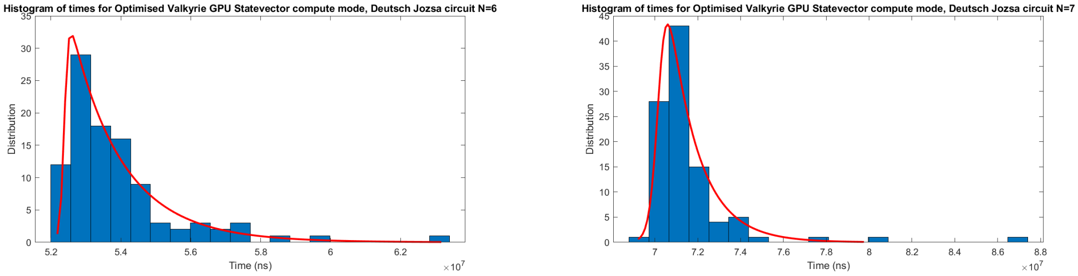
(g) Histogram for Optimised Valkyrie running on the CPU with N=10

Figure 7.15: Histograms for the distribution of execution times for optimised Valkyrie CPU with Deutsch Jozsa's Algorithm



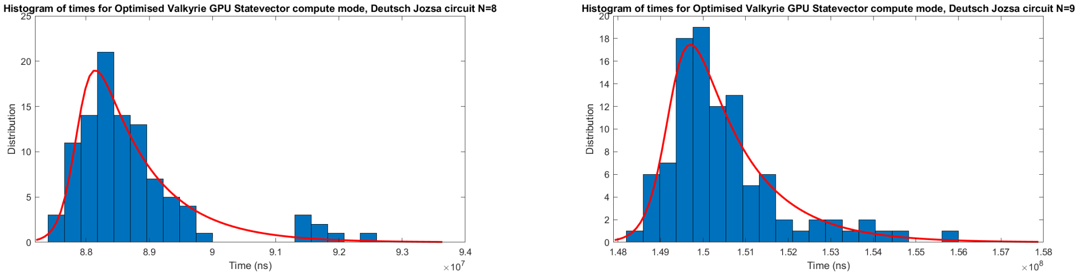
(a) Histogram for Optimised Valkyrie running on the GPU with N=4

(b) Histogram for Optimised Valkyrie running on the GPU with N=5



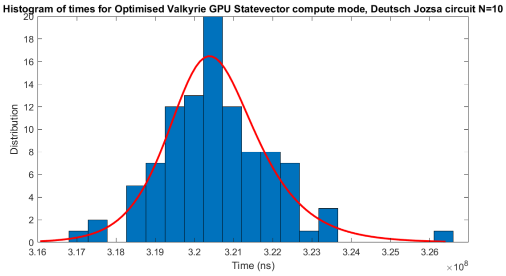
(c) Histogram for Optimised Valkyrie running on the GPU with N=6

(d) Histogram for Optimised Valkyrie running on the GPU with N=7



(e) Histogram for Optimised Valkyrie running on the GPU with N=8

(f) Histogram for Optimised Valkyrie running on the GPU with N=9



(g) Histogram for Optimised Valkyrie running on the GPU with N=10

Figure 7.16: Histograms for the distribution of execution times for optimised Valkyrie GPU with Deutsch Jozsa's Algorithm

Chapter 8

Future Development and Conclusion

Summary

In this section we review what we have achieved in this project, and recount how we have changed and adapted Valkyrie throughout it's development. Furthermore, we also consider what future development can be applied to Valkyrie to add features and improve performance.

8.1 Future Development

We have seen that Valkyrie and Visual-Q both do achieve the objectives laid out in Section 2. However, we can also see that both programs can be further developed. I currently plan to keep the Valkyrie/Visual-Q codebase open source.

8.1.1 Valkyrie future developments

- **Additional Performance**: As we have discussed in Section 6.2 Valkyrie has had some optimisations. However, undoubtedly we can do more. CPU instruction sets have become more complicated over time, with special instructions which can allow for exceptionally efficient execution of certain algorithms implemented in hardware. Valkyrie could certainly leverage these special instruction sets with appropriate research and development work. This effort could further improve the performance of Valkyrie.

- **Accurate measurement**: Valkyrie uses quite a naive measurement methodology. We simply scan through the statevector and and randomly generate a location which to report as the state of the system. However, every practical implementation of a quantum computer has slight biases as to which state that a quantum system will settle into. Some quantum computer emulators may show slight preference towards states with more 0's than 1's. We could imagine a module which we could add to Valkyrie which uses a statistics based approach to biasing the measurement of a quantum computer state.

- **OpenCL GPU acceleration**: Valkyrie uses the Nvidia CUDA architecture to provide it's GPU acceleration. However, many users may not have an Nvidia GPU. The OpenCL library can be accelerated by all graphics processors, even integrated GPU's. This would expand the universality of Valkyrie's acceleration allowing for better performance for all users.

8.1.2 Visual-Q future developments

As mentioned in Section 6.3 we were unable to develop Visual-Q as much as we would have liked. Therefore we are able to list more development options of Visual-Q.

- **Display circuit diagram**: Circuit diagrams can be easily generated from certain datastructures inside Valkyrie's codebase. If Visual-Q could hook into that datastructure, it can generate a simple circuit diagram to show the user what circuit they had specified.
- **Bloch Sphere**: The Bloch sphere allows a user to see a visual representation of the quantum state of a qubit, even modify it if they wish. We detail the construction of the Bloch sphere in Section 1.2.2, and the calculations required to display one should be quite achievable.
- **Step by step execution**: Since OpenQASM is defined in sequential steps we can conceive that the user may want to observe the quantum statevector at multiple steps in the execution, therefore the ability to step through the OpenQASM code would be invaluable to users.
- **File handling**: While we already have a form of persistence in Valkyrie, where it loads the last OpenQASM code that it executed. However, it would be better to have a formal file explorer which allows users to load up previous files that they had executed.

8.2 Conclusion

The final year project represents the culmination of a lifetime of fascination with engineering. Quantum computing, in my eyes, represents the future of scientific computing. Academic papers have been published showing that operational quantum computing can revolutionise artificial intelligence and cryptography. Valkyrie is designed to help bring high performance quantum computer simulations to average consumers.

The implementation of Valkyrie started with understanding the OpenQASM language, we started by building a lexer and parser for OpenQASM, using the ANTLR tool [34] to build the lexing side and then adding custom parsing and Abstract Syntax tree visitation. we then progressed to building the quantum circuits into data-structures within the code and implementing algorithms to provide fast execution of these quantum circuits. This implementation proved to be inefficient as we went into performance testing.

Valkyrie's performance achievements are a result of a journey of success and failure. After Experiment 2 in Section 7.2, we realised that we had a significant performance deficit we had to overcome. In some tests this performance deficit stood at 89% between Valkyrie and Cirq. We performed further performance testing on Valkyrie to investigate where we were losing time. The investigations primary conclusion was that Valkyrie had implementations of algorithms that were far less efficient than theoretically possible.

We went on to implement optimisations in Valkyrie including the discovery of a method to dramatically reduce the number of calculations required as a side benefit to the "tail" method we have developed. These optimisations had a transformative effect on our execution times with performance improvements of up to 90.3% over the unoptimised version of Valkyrie. We also consider that execution times for the Deutsch-Jozsa circuit with $N = 10$ on Valkyrie with optimisations progressed from 3 seconds to 300 milliseconds. From a user perspective this represents a significant difference in the experience of using the simulator.

During performance testing, we also considered how reliable the execution times for each competing simulator were. The result of this testing showed Valkyrie had very comparable variances to Cirq and provided better variance values than Qiskit. In consideration of the distribution of results, Valkyrie had generally negatively skewed result while Cirq provided a very balanced distribution. Furthermore, Qiskit showed very unusual results, with a bimodal distribution potentially caused by CPU scheduling conflicts with other programs. Since we are testing in the expected use environment we must consider this as a feature of Qiskit's execution times rather than a set of anomalous results.

Visual-Q provides a simple to use interface for users to program a quantum circuit in OpenQASM. It is less developed that we might have hoped, but it delivers on the core goals of allowing users to access the performance and accuracy of Valkyrie without the difficulty of using a command line interface.

Throughout this project we have confronted many problems unique to quantum computing, primarily the difficulty of accurately simulating the behaviour of qubits. Einstein once said "God does not play dice with the universe" [44], after the difficulties we have confronted in tackling the enigmatic nature of quantum calculations we might wonder if our task would have been easier if our universe was as deterministic as Einstein would have preferred it. On the other hand, overcoming the challenges we faced was a rewarding experience and has raised the quality of the delivered software.

In conclusion, Valkyrie and Visual-Q do provide researchers using consumer grade hardware, access to a fast and accurate quantum computer emulator which gives them an additional tool to explore the fascinating world of quantum computing.

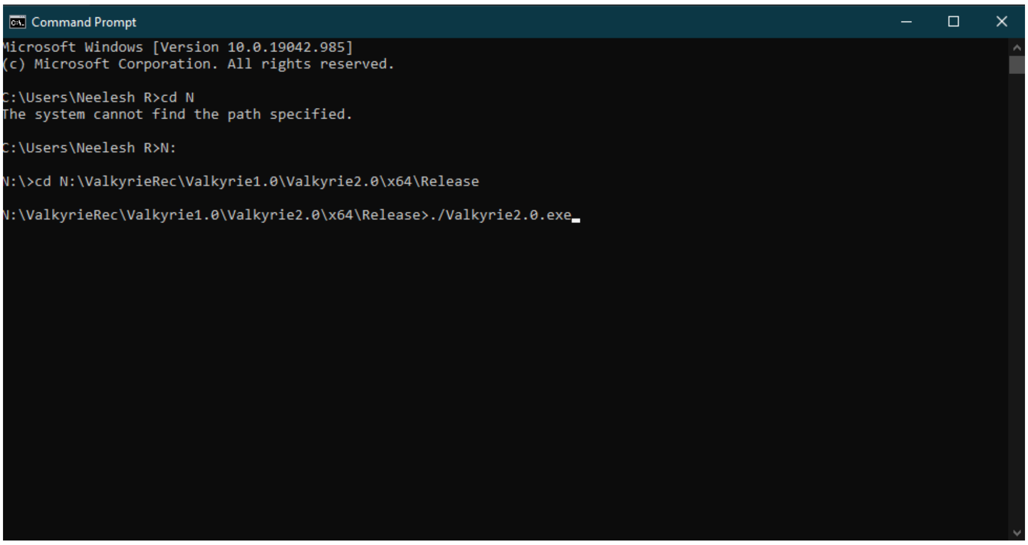
Chapter 9

User manual

9.1 Valkyrie

9.1.1 Setup for normal use

Valkyrie uses a simple compiled ".exe" file to run it's code. It is easiest to use this from the windows command line. We can see this from Figure 9.1.



```
Microsoft Windows [Version 10.0.19042.985]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Neelesh R>cd N
The system cannot find the path specified.

C:\Users\Neelesh R>N:

N:\>cd N:\ValkyrieRec\Valkyrie1.0\Valkyrie2.0\x64\Release

N:\ValkyrieRec\Valkyrie1.0\Valkyrie2.0\x64\Release>./Valkyrie2.0.exe
```

Figure 9.1: Command line interface to run Valkyrie at a basic level

The user simply needs to copy the "Release" folder from the github repo (https://github.com/Neelesh99/Valkyrie1.0 they must also ensure that they have the most updated drivers for their Nvidia GPU as well as the CUDA compute toolkit [45]. Once these are installed the user can execute Valkyrie from the command line.

9.1.2 Running Valkyrie

We have covered the details of the command line options in Section 6.1.8. We will see how these are used in the following section. Before the user can run Valkyrie the user must prepare an OpenQASM file [9]. An example of this is shown below:

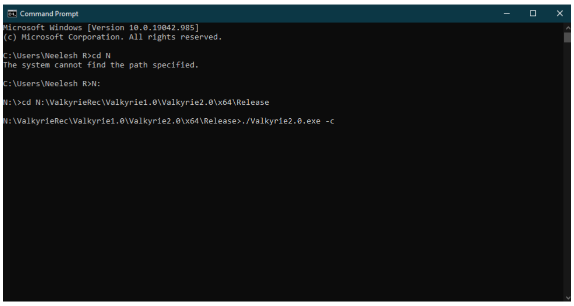
```
OPENQASM 2.0;
qreg q[3];
creg c[3];
```

```
h q[0];
cx q[0],q[1];
measure q -> c;
```

Once the user has prepared their OpenQASM file, they must then select which processor and execution mode they'd like to run Valkyrie. We must remember that Valkyrie can not only both run on the CPU and the GPU, it also has two execution modes. The "fast" execution mode allows for very quick operation at the cost of accuracy, while "statevector" compute mode is fully accurate but can be much slower for larger circuits.

Running Valkyrie on the CPU

The first command line option that Valkyrie has is which processor to execute on. To select a CPU based run, the user's first flag should be "-c". This is shown in Figure 9.2.

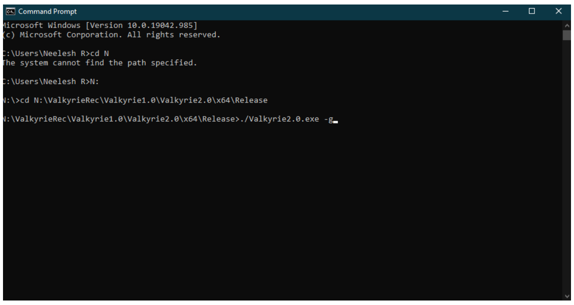


```
Microsoft Windows [Version 10.0.19042.985]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Neelesh R>cd N
The system cannot find the path specified.

C:\Users\Neelesh R>N:

N:\>cd N:\ValkyrieRec\Valkyrie1.0\Valkyrie2.0\x64\Release

N:\ValkyrieRec\Valkyrie1.0\Valkyrie2.0\x64\Release>./Valkyrie2.0.exe -c
```

Figure 9.2: Command line interface to run Valkyrie in CPU processing mode

Running Valkyrie on the GPU

If the user wants to run Valkyrie on the GPU, the first flag should be "-g". This is shown in Figure 9.3.



```
icrosoft Windows [Version 10.0.19042.985]
c) Microsoft Corporation. All rights reserved.

:\Users\Neelesh R>cd N
he system cannot find the path specified.

:\Users\Neelesh R>N:

:\>cd N:\ValkyrieRec\Valkyrie1.0\Valkyrie2.0\x64\Release

:\ValkyrieRec\Valkyrie1.0\Valkyrie2.0\x64\Release>./Valkyrie2.0.exe -g
```

Figure 9.3: Command line interface to run Valkyrie in GPU processing mode

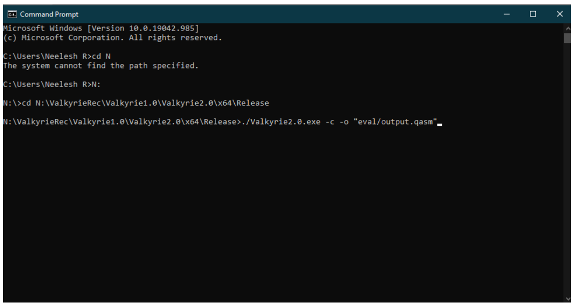
If the user specifies both CPU and GPU flags, the CPU flag has preference.

Specifying the file to execute

We noted before that before running Valkyrie the user must prepare an OpenQASM file. To point Valkyrie to the correct file the user must specify "-o" followed by the filename. The user can specify files in subfolders, such as presented in Figure 9.4.

Specifying compute mode

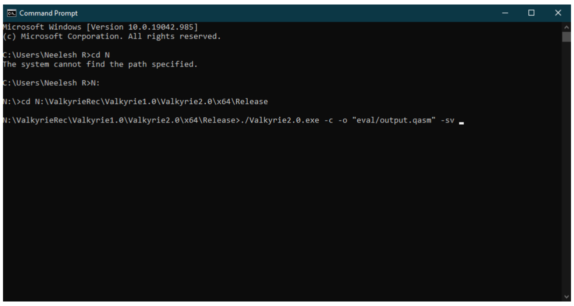
The user has the options of "statevector" and "fast" compute modes. If the user would like to select "fast" compute mode they should specify the "-fast" flag. If the user would like to use "statevector"



```
Microsoft Windows [Version 10.0.19042.985]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Neelesh R>cd N
The system cannot find the path specified.

C:\Users\Neelesh R>N:

N:\>cd N:\ValkyrieRec\Valkyrie1.0\Valkyrie2.0\x64\Release

N:\ValkyrieRec\Valkyrie1.0\Valkyrie2.0\x64\Release>./Valkyrie2.0.exe -c -o "eval/output.qasm"
```

Figure 9.4: Command line interface to point Valkyrie to the file the user would like to run.

compute mode, the user must specify the "-sv" flag. If the user does not specify a flag, Valkyrie will revert to "fast" compute mode.

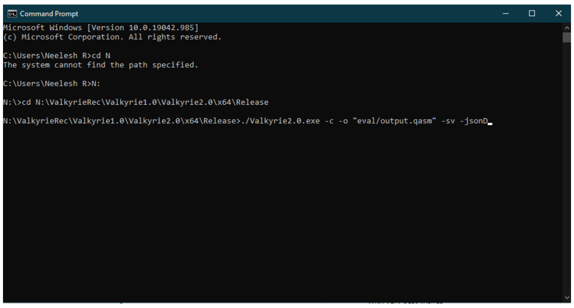


```
Microsoft Windows [Version 10.0.19042.985]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Neelesh R>cd N
The system cannot find the path specified.

C:\Users\Neelesh R>N:

N:\>cd N:\ValkyrieRec\Valkyrie1.0\Valkyrie2.0\x64\Release

N:\ValkyrieRec\Valkyrie1.0\Valkyrie2.0\x64\Release>./Valkyrie2.0.exe -c -o "eval/output.qasm" -sv
```

Figure 9.5: Command line interface to instruct Valkyrie as to which compute mode to use.

Output specification

Valkyrie can output it's results in two ways, the command line method requires no additional arguments. However, if the user would like a JSON styled output, the user can specify the "-json" flag which prints a parsable JSON format. The use of this flag is referred to in Figure 9.6.

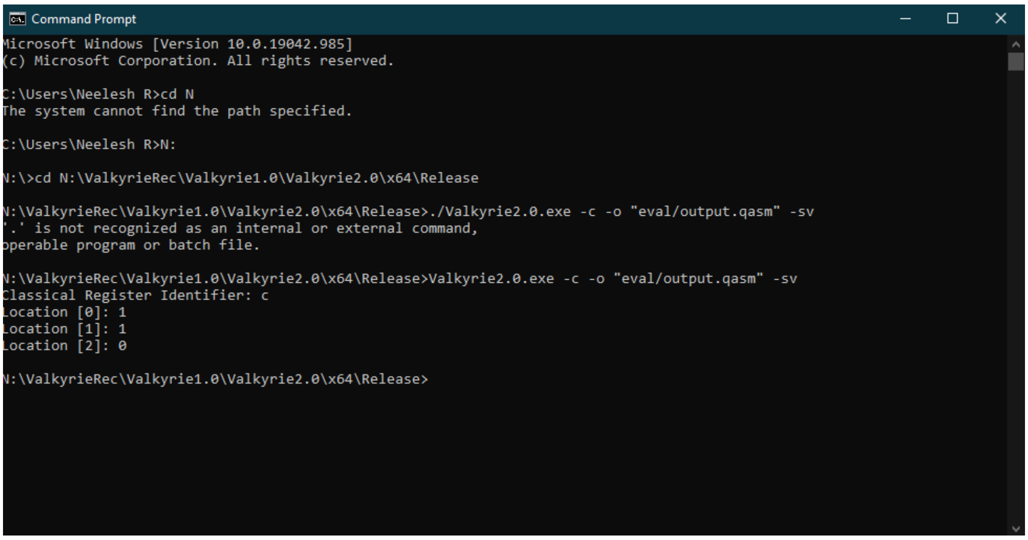


```
Microsoft Windows [Version 10.0.19042.985]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Neelesh R>cd N
The system cannot find the path specified.

C:\Users\Neelesh R>N:

N:\>cd N:\ValkyrieRec\Valkyrie1.0\Valkyrie2.0\x64\Release

N:\ValkyrieRec\Valkyrie1.0\Valkyrie2.0\x64\Release>./Valkyrie2.0.exe -c -o "eval/output.qasm" -sv -jsonD
```

Figure 9.6: Command line interface to instruct Valkyrie to print a json parsable output.

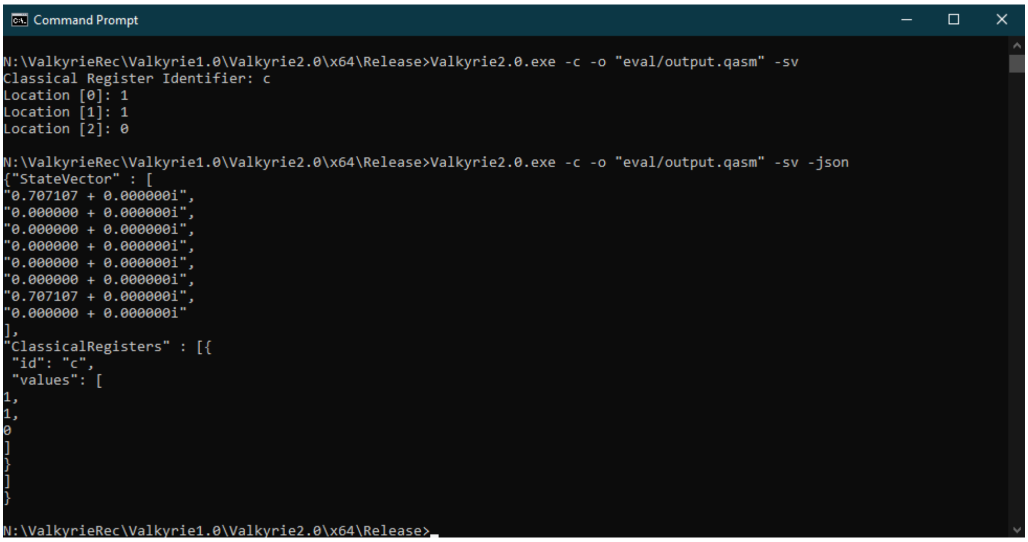
Results

If the user has specified no "-json" flag, they will receive an output similar to Figure 9.7. If the user has specified the "-json" flag, they will receive an output as seen in Figure 9.8.



```
Microsoft Windows [Version 10.0.19042.985]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Neelesh R>cd N
The system cannot find the path specified.

C:\Users\Neelesh R>N:

N:\>cd N:\ValkyrieRec\Valkyrie1.0\Valkyrie2.0\x64\Release

N:\ValkyrieRec\Valkyrie1.0\Valkyrie2.0\x64\Release>./Valkyrie2.0.exe -c -o "eval/output.qasm" -sv
'.' is not recognized as an internal or external command,
operable program or batch file.

N:\ValkyrieRec\Valkyrie1.0\Valkyrie2.0\x64\Release>Valkyrie2.0.exe -c -o "eval/output.qasm" -sv
Classical Register Identifier: c
Location [0]: 1
Location [1]: 1
Location [2]: 0

N:\ValkyrieRec\Valkyrie1.0\Valkyrie2.0\x64\Release>
```

Figure 9.7: Standard print output for Valkyrie.



```
N:\ValkyrieRec\Valkyrie1.0\Valkyrie2.0\x64\Release>Valkyrie2.0.exe -c -o "eval/output.qasm" -sv
Classical Register Identifier: c
Location [0]: 1
Location [1]: 1
Location [2]: 0

N:\ValkyrieRec\Valkyrie1.0\Valkyrie2.0\x64\Release>Valkyrie2.0.exe -c -o "eval/output.qasm" -sv -json
{"StateVector" : [
"0.707107 + 0.000000i",
"0.000000 + 0.000000i",
"0.000000 + 0.000000i",
"0.000000 + 0.000000i",
"0.000000 + 0.000000i",
"0.000000 + 0.000000i",
"0.707107 + 0.000000i",
"0.000000 + 0.000000i"
],
"ClassicalRegisters" : [{
 "id": "c",
 "values": [
1,
1,
0
]
}
]
}

N:\ValkyrieRec\Valkyrie1.0\Valkyrie2.0\x64\Release>_
```

Figure 9.8: JSON print output for Valkyrie.

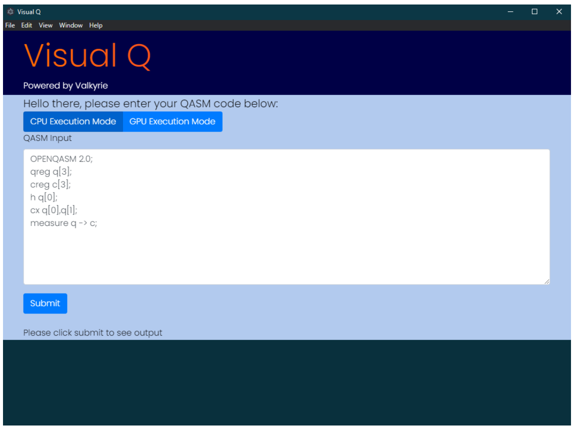
9.2 Visual Q

9.2.1 Running Visual Q

Visual Q is already packaged for the windows platform. That means that the user must simply download the program folder from github (https://github.com/Neelesh99/Valkyrie1.0/tree/master/visual-q/visual-q). In this folder the user can simply double click the "Visual-Q" application. Which launches the window shown in Figure 9.9.

9.2.2 Writing QASM code

The full specification for OpenQASM code can be found in IBM's whitepaper for OpenQASM [9]. In Visual-Q the user can quickly enter QASM code in the code window as shown in Figure 9.10.

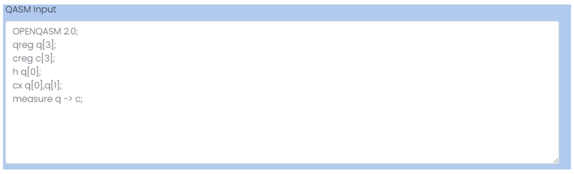


Hello there, please enter your QASM code below:

```
OPENQASM 2.0;
qreg q[3];
creg c[3];
h q[0];
cx q[0],q[1];
measure q -> c;
```

Please click submit to see output

Figure 9.9: Visual Q landing page as soon as the user launches the application.



```
OPENQASM 2.0;
qreg q[3];
creg c[3];
h q[0];
cx q[0],q[1];
measure q -> c;
```

Figure 9.10: QASM input window.

9.2.3 Execution mode

The user can select which processor the user would like Valkyrie to execute on. The processor switch is shown in Figure 9.11.

Figure 9.11: Visual Q processor switch button.

9.2.4 Executing code

The user can execute their OpenQASM code by clicking the "Submit" button in the Visual-Q window, which can be seen in Figure 9.9.

9.2.5 Viewing Results

The results are presented in the Output section of Visual-Q, which shows both the "measured" value of the state of the circuit as well as the quantum statevector. This can be seen in Figure 9.12.

This section of Visual-Q is directly under the "Submit" button.

Visual Q

Powered by Valkyrie

Output:

Classical Register	Index	Measured Value
c	0	0
c	1	0
c	2	0

State	Quantum State
000	0.707107 + 0.000000i
001	0.000000 + 0.000000i
010	0.000000 + 0.000000i
011	0.000000 + 0.000000i
100	0.000000 + 0.000000i
101	0.000000 + 0.000000i
110	0.707107 + 0.000000i
111	0.000000 + 0.000000i

Figure 9.12: Visual Q output section, showing both measured output and quantum statevector output

Bibliography

[1] Y. Noson and M. Mirco, *Quantum Computing for Computer Scientists*. Cambridge University Press, 2008.

[2] S. Taylor, “Bits and bytes,” [Online]. Available: https://web.stanford.edu/class/cs101/bits-bytes.html (visited on 01/09/2021).

[3] M. Nakahara and T. Ohmi, *Quantum Computing, From Linear Algebra to Physical Realizations*. CRC Press, Taylor Francis group, 2008.

[4] R. Brylinski and G. Chen, *Mathematics of Quantum Computation*. CRC Press, Chapman and Hall, 2002.

[5] M. Hunter, “The quantum computing era is here. why it matters—and how it may change our world.,” 2020. [Online]. Available: https://www.forbes.com/sites/ibm/2020/01/16/the-quantum-computing-era-is-here-why-it-mattersand-how-it-may-change-our-world/#4f9cf795c2b1.

[6] P. A. M. Dirac, “A new notation for quantum mechanics,” *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 35, no. 3, pp. 416–418, 1939. DOI: 10.1017/S0305004100021162.

[7] Smite-Meister, 2009. [Online]. Available: https://en.wikipedia.org/wiki/Bloch_sphere#/media/File:Bloch_sphere.svg.

[8] F. Bloch, “Nuclear induction,” *Phys. Rev.*, vol. 70, pp. 460–474, 7-8 Oct. 1946. DOI: 10.1103/PhysRev.70.460. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRev.70.460.

[9] A. W. Cross, L. S. Bishop, J. A. Smolin, and J. M. Gambetta, *Open quantum assembly language*, 2017. arXiv: 1707.03429 [quant-ph].

[10] *Post-quantum cryptography*, eng, 2009.

[11] A. M. Turing, “On computable numbers, with an application to the entscheidungsproblem,” *Proceedings of the London Mathematical Society*, vol. s2-42, no. 1, pp. 230–265, 1937. DOI: https://doi.org/10.1112/plms/s2-42.1.230. eprint: https://londmathsoc.onlinelibrary.wiley.com/doi/pdf/10.1112/plms/s2-42.1.230. [Online]. Available: https://londmathsoc.onlinelibrary.wiley.com/doi/abs/10.1112/plms/s2-42.1.230.

[12] P. Benioff, “The computer as a physical system: A microscopic quantum mechanical hamiltonian model of computers as represented by turing machines,” *Journal of Statistical Physics*, vol. 22, pp. 563–591, 1980. DOI: https://doi.org/10.1007/BF01011339. [Online]. Available: https://link.springer.com/article/10.1007%5C%2FBF01011339.

[13] W. C. Holton, *Quantum computer*, 2020.

[14] A. Steane, “Quantum computing,” *Reports on Progress in Physics*, vol. 61, p. 117, 1998. [Online]. Available: http://iopscience.iop.org/0034-4885/61/2/002).

[15] IBM, *Ibm quantum*, 2020. [Online]. Available: https://www.ibm.com/quantum-computing/.

[16] R. S. Smith, M. J. Curtis, and W. J. Zeng, *A practical quantum instruction set architecture*, 2017. arXiv: 1608.03355 [quant-ph].

[17] E. Blem, J. Menon, and K. Sankaralingam, “A detailed analysis of contemporary arm and x86 architectures,” Jan. 2013.

[18] M. M. Mano and C. R. Kime, *Logic and computer design fundamentals, third edition.* 2004.

[19] IBM, *Qiskit*, 2020. [Online]. Available: https://qiskit.org/.

[20] NVidia, *Nvidia ampere whitepaper*, 2020. [Online]. Available: https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf.

[21] R. Feynman, "Quantum mechanical computers," *Foundations of Physics*, vol. 16, p. 507, 1986.

[22] D. Deutsch and R. Penrose, "Quantum theory, the church–turing principle and the universal quantum computer," *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences*, vol. 400, no. 1818, pp. 97–117, 1985. DOI: 10.1098/rspa.1985.0070. eprint: https://royalsocietypublishing.org/doi/pdf/10.1098/rspa.1985.0070. [Online]. Available: https://royalsocietypublishing.org/doi/abs/10.1098/rspa.1985.0070.

[23] N. Gershenfeld and I. L. Chuang, "Quantum computing with molecules," *Scientific American*, vol. 278, no. 6, pp. 66–71, 1998, ISSN: 00368733, 19467087. [Online]. Available: http://www.jstor.org/stable/26057857.

[24] L. Crane and C. Whyte, *Google's 72-qubit chip is the largest yet*, 2018. [Online]. Available: https://www.newscientist.com/article/2162894-googles-72-qubit-chip-is-the-largest-yet.

[25] S. Resch and U. R. Karpuzcu, *Quantum computing: An overview across the system stack*, 2019. arXiv: 1905.07240 [quant-ph].

[26] J. Jones, "Course 10 - nuclear magnetic resonance quantum computation," in *Quantum Entanglement and Information Processing*, ser. Les Houches, D. Estève, J.-M. Raimond, and J. Dalibard, Eds., vol. 79, Elsevier, 2004, pp. 357–400. DOI: https://doi.org/10.1016/S0924-8099(03)80034-3. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0924809903800343.

[27] D. Loss and D. P. DiVincenzo, "Quantum computation with quantum dots," *Physical Review A*, vol. 57, p. 120, 1998.

[28] D. Deutsch and R. Jozsa, "Rapid solution of problems by quantum computation," *Proceedings of the Royal Society A*, vol. 439, 1982. DOI: https://doi.org/10.1098/rspa.1992.0167. [Online]. Available: https://royalsocietypublishing.org/doi/10.1098/rspa.1992.0167.

[29] D. Simon, "On the power of quantum computation," *Proceedings of the 35th Annual IEEE Symposium on Foundations of Computer Science*, vol. 26, May 1997. DOI: 10.1137/S0097539796298637.

[30] P. W. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," pp. 124–134, 1994. DOI: 10.1109/SFCS.1994.365700.

[31] Xin Zhou and Xiaofei Tang, "Research and implementation of rsa algorithm for encryption and decryption," vol. 2, pp. 1118–1121, 2011. DOI: 10.1109/IFOST.2011.6021216.

[32] J. P. Buhler, H. W. Lenstra, and C. Pomerance, "Factoring integers with the number field sieve," in *The development of the number field sieve*, A. K. Lenstra and H. W. Lenstra, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 50–94, ISBN: 978-3-540-47892-8.

[33] Google, *Google cirq*, 2020. [Online]. Available: https://quantumai.google/cirq.

[34] ANTLR, *Antlr parser generator*. [Online]. Available: https://www.antlr.org/.

[35] *Electron js frameowrk*, 2021. [Online]. Available: https://www.electronjs.org/.

[36] I. D. K. Brown, S. Stepney, A. Sudbery, and S. L. Braunstein, "Searching for highly entangled multi-qubit states," *Journal of Physics A: Mathematical and General*, vol. 38, no. 5, pp. 1119–1131, Jan. 2005. DOI: 10.1088/0305-4470/38/5/013. [Online]. Available: https://doi.org/10.1088/0305-4470/38/5/013.

[37] R. Portugal, "Quantum walks and search algorithms," *Quantum Science and Technology*, 2013. DOI: 10.1007/978-1-4614-6336-8. [Online]. Available: https://cds.cern.ch/record/1522001/files/978-1-4614-6336-8_BookBackMatter.pdf.

[38] K. W. R. Richard J. Lipton, *QUANTUM ALGORITHMS VIA LINEAR ALGEBRA, A Primer*. The MIT Press, 2014.

[39] A. Kelly, *Openqasm 2.0 grammar*, 2018. [Online]. Available: https://github.com/libtangle/QASM-Grammar/blob/master/QASM.g4.

[40] T. Instruments, *Optimizing compiler v7.4*, 2012. [Online]. Available: https://www.ti.com/lit/ug/spru187u/spru187u.pdf.

[41] R. Jozsa, *An introduction to measurement based quantum computation*, 2005. arXiv: quant-ph/0508124 [quant-ph].

[42] *Nvidia product brand identity and usage guideline*, 2005. [Online]. Available: https://www.nvidia.com/content/imagekit/guidelines/NVIDIA_nForce_logo_guidelines.pdf.

[43] IBM, *Ibm deutsch-jozsa algorithm*, 2019. [Online]. Available: https://qiskit.org/textbook/ch-algorithms/deutsch-jozsa.html#1.-Introduction-.

[44] A. Einstein, *The Born-Einstein Letters*. Walker and Company, 1971.

[45] *Nvidia compute toolkit download and documentation*, 2021. [Online]. Available: https://developer.nvidia.com/cuda-toolkit.

[46] H. Hertz, "Ueber einen einfluss des ultravioletten lichtes auf die electrische entladung," *Annalen der Physik*, vol. 267, no. 8, pp. 983–1000, 1887. DOI: https://doi.org/10.1002/andp.18872670827. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/andp.18872670827. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/andp.18872670827.

[47] A. Einstein, "ber einen die erzeugung und verwandlung des lichtes betreffenden heuristischen gesichtspunkt," *Annalen der Physik*, vol. 322, no. 6, pp. 132–148, 1905. DOI: https://doi.org/10.1002/andp.19053220607. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/andp.19053220607. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/andp.19053220607.

[48] L. D. Broglie, "Waves and quanta," *Nature*, vol. 112, p. 540, 1923. DOI: https://doi.org/10.1038/112540a0. [Online]. Available: https://www.nature.com/articles/112540a0#citeas.

Appendix A

Quantum Physics

Even though the topic of this final year project is quite abstracted from the underlying quantum mechanics. It would benefit us to briefly review the fundamentals of quantum physics and gain an understanding of the mechanics that allow the complex interactions which power quantum computers.

A.1 Duality

The story of quantum physics begins with a series of groundbreaking experiments which presented a startlingly inconsistent set of results. At the turn of the 19^{th} century physicists had a puzzling question, was light a particle or a wave. Fortunately, in 1801 Thomas Young provided, what was thought to be, definitive proof that it was a wave through his famous "Double slit experiment". His results seemed indisputable at the time, light must be a wave, and many physicists of the time could be forgiven for thinking that the matter was settled for good.

However, in 1887 Heinrich Hertz, who famously experimentally proved the existence of Maxwell's predicted electromagnetic waves, observed that ultraviolet light causes charged objects to lose their charge more quickly than visible light of any intensity [46]. Hertz had unearthed one of the founding experimental observations of quantum physics, the **photoelectric effect**. The photoelectric effect was fully explained later by Albert Einstein [47], in his explanation Einstein proved that light behaved as an explicitly particle with quantised absorption and emission.

This is the incompatibility which lead to physicists in the early 20^{th} century to abandon the classical notions of separate particles and separate waves, to a new framework named quantum mechanics. In this new field of physics not only could particles be waves and waves be particles, matter itself existed as both particles and waves as per the experiments of De Broglie [48]. The detail of this theory is left out of this appendix entry since it is beyond the scope of our analysis.

A.2 Superposition

A direct result of wave particle duality is the principle of superposition, which is ubiquitous in quantum computing. Noson and Mirco provide an indepth and complete coverage of this concept and our explanation here is adapted from their text [1].

We will assume that we have a single particle confined to a line in 1-dimensional space. Instead of allowing the particle to take any position on that line for both simplicity and relevance to quantum computing, we will only allow the particle to take a discrete set of positions x_i with $x_{i+1} = x_i + \delta$. We will use this set of discrete positions with the understanding that we can take $\lim_{\delta \to 0}$ to move from our discrete position to the continuous set of positions. This discrete set of positions is shown in Figure A.1.

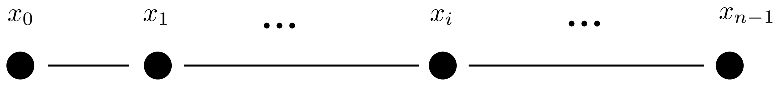


Figure A.1: Possible positions of a particle on a line

Furthermore, let us create a formal way of representing each position x_i as a state of the particle. For this we will use the Bra-Ket notation introduced in Section 1.2.1. We will represent the state where the particle is in position x_i ($|x_i\rangle$) as a vector of length n filled with 0's and a single 1 in position i, as outlined below.

$$\begin{aligned} |x_0\rangle &= [1, 0, ..., 0]^T \\ |x_1\rangle &= [0, 1, 0..., 0]^T \\ &\vdots \\ |x_{n-1}\rangle &= [0, 0, ..., 1]^T \end{aligned}$$

In classical physics where particles are just that, particles, our particle will exist in one of the states $|x_i\rangle$. However, as per the principles of wave-particle duality our particle is also a wave. As noted be Noson and Mirco [1], we must make a "bold" leap. Since our particle can be a wave, we must assume that like any wave, our wave can be defined over more than one of our positions x_i at once as seen in Figure A.2.

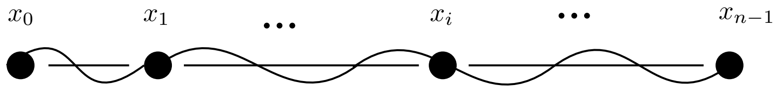


Figure A.2: Wave position on a line

In fact we postulate that since our particle is also a wave which is unbounded our particle's state must have components from all $|x_i\rangle$.

In Equation A.1 we state the wave equation in one dimension.

$$\frac{\partial^2 f(x,t)}{\partial t^2} = c^2 \frac{\partial^2 f(x,t)}{\partial x^2} \tag{A.1}$$

The eigenmode general solution for $f(x,t)$ separates out the spacial and time relationship of the wave into the form $f(x,t) = f(x)e^{-iwt}$. The reader might note the similarity of this general solution to our polar form of a complex number in Equation 1.10. We can therefore resolve this equation for a general position x_i to a complex number c_i.

We will use complex coefficients to multiply each $|x_i\rangle$ to form the full state of the particle ($|\psi\rangle$) as in Equation A.2.

$$|\psi\rangle = c_0 |x_o\rangle + c_1 |x_1\rangle + \cdots + c_{n-1} |x_{n-1}\rangle \tag{A.2}$$

Since we know that all $|x_i\rangle$ are represented by a vector with a single non-zero entry, we can simplify Equation A.2 to Equation A.3.

$$|\psi\rangle = [c_0, c_1, \ldots, c_{n-1}]^T \tag{A.3}$$

We then observe finally that our complex components c_i must relate to "how much" of the particle exists in position x_i. Furthermore, if we were to try an directly observe our particle we must find a particle not a wave or a complex vector. Hence these coefficients c_i must relate to the probability of finding the particle at a position x_i. The exact relationship is given by Equation A.4.

$$p(x_i) = \frac{|c_i|^2}{\sum_{j=0}^{n-1} |c_j|^2} \tag{A.4}$$

We have now understood how a quantum system (in our case a particle) can exist in a state of superposition, it is the wave-particle duality which gives rise to the superposed state $|\psi\rangle$. By exploiting this superposition over just two degrees of freedom we can create a representation of a qubit. This also explains why we have so many competing technologies as covered in Section 3.2. All we need for a qubit is a wave-particle duality with two degrees of freedom. As De Broglie postulated, any matter can satisfy this requirement if we look at a small enough wavelength. This task is made much easier when particles are of sufficiently low mass. Finally, we need not just be constricted to matter for our qubits since light itself exists both as photons and waves (as indeed the rest of the standard model) and so we have an extremely wide scope of qubit technologies to explore.

Appendix B

Valkyrie Codebase

```


#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include "antlr4-runtime.h"
#include "libs/qasm2Lexer.h"
#include "libs/qasm2Parser.h"
#include "libs/qasm2Visitor.h"
#include "libs/qasm2BaseVisitor.h"
#include "libs/staging.h"
#include "libs/CPUDevice.h"
#include "libs/GPUDevice.cuh"
#include "libs/Measurement.h"
#include "libs/JSONify.h"
#include <Windows.h>
#include <string>
#include <fstream>
#include <iostream>
#include <chrono>

#include <stdio.h>

#include "test/ValkyrieTests.h"

using namespace antlr4;

// getexepath allows vakyrie to resolve the directory it is operating
    ↪ in
std::string getexepath()
{
    char result[MAX_PATH];
    return std::string(result, GetModuleFileName(NULL, result, MAX_PATH
        ↪ ));
}
// DisplayHeader is used during info print command to display GPU
    ↪ devices connected
void DisplayHeader();
// printHelp will print help if the user enters an invalid set of
    ↪ command line options
void printHelp();
// resolveDeviceType resolves what type of device the user wants to
    ↪ rpocess on
```

```
DeviceType resolveDeviceType(std::vector<std::string> arguments);
// fetchFileName finds the file name specified by the user
std::string fetchFileName(std::vector<std::string> arguments);
// CPURun performs CPU execution of the target QASM code
void CPURun(std::string filename, bool SV, bool jsonMode);
// GPURun performs GPU execution of the target QASM code
void GPURun(std::string filename, bool SV, bool jsonMode);
// handleInfoRequest prints the info requested by user in command line
    ↪ options
void handleInfoRequest(std::vector<std::string> arguments);
// resolveComputeMode resolves whether the user wants fast or
    ↪ statevector compute modes
bool resolveComputeMode(std::vector<std::string> arguments);
// resolveJSONPrint resolves whether this is a VisualQ call which
    ↪ requires json output
bool resolveJsonPrint(std::vector<std::string> arguments);

enum timingPoint
{
    NONE_,
    FULL,
    PARSE,
    STAGE,
    EXECUTION
};

// resolveTimingRequest
timingPoint resolveTimingRequest(std::vector<std::string> arguments);

// timeCPUExecution is used for experimentation and metric gathering
void timeCPUExecution(std::string filename, bool SV, bool jsonMode,
    ↪ timingPoint point) {
    // Start clock
    std::chrono::steady_clock::time_point begin;
    std::chrono::steady_clock::time_point end;
    if (point == FULL || point == PARSE) {
        begin = std::chrono::high_resolution_clock::now();
    }
    std::ifstream stream;
    stream.open(filename);          // Open File requested
    if (!stream.is_open()) {
        std::cout << "Couldn't find file specified" << std::endl;
        printHelp();
        return;
    }
    ANTLRInputStream input(stream);                  // Convert
        ↪ filestream to ANTLR stream
    qasm2Lexer lexer(&input);                        // Lex file
    CommonTokenStream tokens(&lexer);                // get the tokens
    qasm2Parser parser(&tokens);                     // send to antlr
        ↪ parser
    qasm2Parser::MainprogContext* tree = parser.mainprog();
        ↪             // Fetch AST tree
    qasm2BaseVisitor visitor;
    visitor.visitMainprog(tree);
        ↪                                         // Use custom visitor
        ↪  to process information
```

```
    std::vector<Register> registers = visitor.getRegisters();
        ↪           // Get registers defined by user
    std::vector<GateRequest> gateRequests = visitor.getGates();
        ↪         // Get gates defined by user
    if (point == PARSE) {
        end = std::chrono::high_resolution_clock::now();
    }
    if (point == STAGE) {
        begin = std::chrono::high_resolution_clock::now();
    }
    Stager stage = Stager();
    std::vector<ConcurrentBlock> blocks = stage.stageInformation(
        ↪ registers, gateRequests);          // User stager to convert
        ↪  parsed information into calculation commands
    if (point == STAGE) {
        end = std::chrono::high_resolution_clock::now();
    }
    if (point == EXECUTION) {
        begin = std::chrono::high_resolution_clock::now();
    }
    CPUDevice device = CPUDevice();
    if (!SV) {
        ↪                                                           //
        ↪ If we are in statevector compute mode, run in statevector
        ↪ mode
        device.run(stage.getRegisters(), blocks);
    }
    else {
        device.runSV(stage.getRegisters(), blocks);
    }
    if (point == EXECUTION) {
        end = std::chrono::high_resolution_clock::now();
    }
    StateVectorMeasurement measure = StateVectorMeasurement(device.
        ↪ getStateVector(), registers);        // Initialise
        ↪ statevector measurement
    measure.measure();
    if (point == FULL) {
        end = std::chrono::high_resolution_clock::now();
    }
    std::cout << std::chrono::duration_cast<std::chrono::nanoseconds>(
        ↪ end - begin).count() << std::endl;
}
// timeGPUExecution is used for experimentation and metric gathering
void timeGPUExecution(std::string filename, bool SV, bool jsonMode,
    ↪ timingPoint point) {
    // Start clock
    std::chrono::steady_clock::time_point begin;
    std::chrono::steady_clock::time_point end;
    if (point == FULL || point == PARSE) {
        begin = std::chrono::high_resolution_clock::now();
    }
    std::ifstream stream;
    stream.open(filename);          // Open File requested
    if (!stream.is_open()) {
        std::cout << "Couldn't find file specified" << std::endl;
        printHelp();
        return;
```

```
    }
    ANTLRInputStream input(stream);                    // Convert filestream to ANTLR stream
    qasm2Lexer lexer(&input);                          // Lex file
    CommonTokenStream tokens(&lexer);                  // get the tokens
    qasm2Parser parser(&tokens);                       // send to antlr parser
    qasm2Parser::MainprogContext* tree = parser.mainprog();            // Fetch AST tree
    qasm2BaseVisitor visitor;
    visitor.visitMainprog(tree);                                        // Use custom visitor to process information
    std::vector<Register> registers = visitor.getRegisters();          // Get registers defined by user
    std::vector<GateRequest> gateRequests = visitor.getGates();        // Get gates defined by user
    if (point == PARSE) {
        end = std::chrono::high_resolution_clock::now();
    }
    if (point == STAGE) {
        begin = std::chrono::high_resolution_clock::now();
    }
    Stager stage = Stager();
    std::vector<ConcurrentBlock> blocks = stage.stageInformation(registers, gateRequests);          // User stager to convert parsed information into calculation commands
    if (point == STAGE) {
        end = std::chrono::high_resolution_clock::now();
    }
    if (point == EXECUTION) {
        begin = std::chrono::high_resolution_clock::now();
    }
    GPUDevice device = GPUDevice();
    if (!SV) {                                                                    // If we are in statevector compute mode, run in statevector mode
        device.run(stage.getRegisters(), blocks);
    }
    else {
        device.runSV(stage.getRegisters(), blocks);
    }
    if (point == EXECUTION) {
        end = std::chrono::high_resolution_clock::now();
    }
    StateVectorMeasurement measure = StateVectorMeasurement(device.getStateVector(), registers);        // Initialise statevector measurement
    measure.measure();
    if (point == FULL) {
        end = std::chrono::high_resolution_clock::now();
    }
    std::cout << std::chrono::duration_cast<std::chrono::nanoseconds>(end - begin).count() << std::endl;
}


```

```
// main is the entrypoint of the program
int main(int argc, char *argv[])
{
    std::vector<std::string> arguments;
    for (int i = 1; i < argc; i++) {
        arguments.push_back(argv[i]);           // collect command line
            ↪  arguments
    }
    handleInfoRequest(arguments);               // check if information
        ↪ was requested by user and print

    DeviceType type = resolveDeviceType(arguments);         //
        ↪ calculate whether the user wants to use the CPU or GPU
    if (type == INVALID) {
        std::cout << "Invalid or No execution mode provided, specify -c
            ↪  or -g" << std::endl;
        printHelp();
        return 1;
    }

    std::string fileName = fetchFileName(arguments);        // resolve
        ↪ the qasm file the user wants to process
    if (fileName == "INVALID") {
        std::cout << "File not specified, please use -o <filename> to
            ↪ indicate which file Valkyrie should process" << std::
            ↪ endl;
        printHelp();
        return 1;
    }
    bool svMode = resolveComputeMode(arguments);            // resolve
        ↪ whether the user wanted to user statevector or fast compute
        ↪ mode
    bool jsonMode = resolveJsonPrint(arguments);            // reolve
        ↪ whether the user wants a JSON print at the end or normal
        ↪ print
    timingPoint timing = resolveTimingRequest(arguments);
    if (timing != NONE_) {
        if (type == CPU_) {                                     //
            ↪ depending on requested devicetype run on CPU or GPU
            for (int i = 0; i < 121; i++) {
                timeCPUExecution(fileName, svMode, jsonMode, timing);
            }
        }
        else {
            for (int i = 0; i < 121; i++) {
                timeGPUExecution(fileName, svMode, jsonMode, timing);
            }
        }
        return 0;
    }
    if (type == CPU_) {                                     //
        ↪ depending on requested devicetype run on CPU or GPU
        CPURun(fileName, svMode, jsonMode);
    }
    else {
        GPURun(fileName, svMode, jsonMode);
    }
    return 0;
```

```
}

void DisplayHeader()
{
    const int kb = 1024;
    const int mb = kb * kb;
    std::cout << "NBody.GPU" << std::endl << "=========" << std::endl << std::endl;

    std::cout << "CUDA version:   v" << CUDART_VERSION << std::endl;

    int devCount;
    cudaGetDeviceCount(&devCount);
    std::cout << "CUDA Devices: " << std::endl << std::endl;

    for (int i = 0; i < devCount; ++i)
    {
        cudaDeviceProp props;
        cudaGetDeviceProperties(&props, i);
        std::cout << i << ": " << props.name << ": " << props.major << "." << props.minor << std::endl;
        std::cout << "  Global memory:   " << props.totalGlobalMem / mb << "mb" << std::endl;
        std::cout << "  Shared memory:   " << props.sharedMemPerBlock / kb << "kb" << std::endl;
        std::cout << "  Constant memory: " << props.totalConstMem / kb << "kb" << std::endl;
        std::cout << "  Block registers: " << props.regsPerBlock << std::endl << std::endl;

        std::cout << "  Warp size:         " << props.warpSize << std::endl;
        std::cout << "  Threads per block: " << props.maxThreadsPerBlock << std::endl;
        std::cout << "  Max block dimensions: [ " << props.maxThreadsDim[0] << ", " << props.maxThreadsDim[1] << ", " << props.maxThreadsDim[2] << " ]" << std::endl;
        std::cout << "  Max grid dimensions:  [ " << props.maxGridSize[0] << ", " << props.maxGridSize[1] << ", " << props.maxGridSize[2] << " ]" << std::endl;
        std::cout << std::endl;
    }
}

void printHelp() {
    std::cout << "Welcome to Valkyrie Help" << std::endl;
    std::cout << "Command line options" << std::endl;
    std::cout << "CPU execution mode: \t \t \t -c" << std::endl;
    std::cout << "GPU execution mode: \t \t \t -g" << std::endl;
    std::cout << "Path to file: \t \t \t \t -o <filepath>" << std::endl;
    std::cout << "State vector computation: -sv" << std::endl;
}

DeviceType resolveDeviceType(std::vector<std::string> arguments) {
    DeviceType val = INVALID;
    for (std::string argument : arguments) {
        if (argument == "-g") {
```

```
            val = GPU_;
            break;
        }
        if (argument == "-c") {
            val = CPU_;
            break;
        }
    }
    return val;
}

bool resolveComputeMode(std::vector<std::string> arguments) {
    for (std::string argument : arguments) {
        if (argument == "-sv") {
            return true;
        }
    }
    return false;
}

bool resolveJsonPrint(std::vector<std::string> arguments) {
    for (std::string argument : arguments) {
        if (argument == "-json") {
            return true;
        }
    }
    return false;
}

timingPoint resolveTimingRequest(std::vector<std::string> arguments) {
    for (int i = 0; i < arguments.size(); i++) {
        if (arguments[i] == "-time") {
            if (i != arguments.size() - 1) {
                if (arguments[i + 1] == "parse") {
                    return PARSE;
                }
                if (arguments[i + 1] == "staging") {
                    return STAGE;
                }
                if (arguments[i + 1] == "execution") {
                    return EXECUTION;
                }
            }
            return FULL;
        }
    }
    return NONE_;
}

std::string fetchFileName(std::vector<std::string> arguments) {
    std::string returnVal = "INVALID";
    if (arguments.size() == 0) {
        return returnVal;
    }
    for (int i = 0; i < arguments.size()-1; i++) {
        if (arguments[i] == "-o") {
            return arguments[i + 1];
        }
```

```
    }
    return returnVal;
}

void CPURun(std::string filename, bool SV, bool jsonMode) {
    std::ifstream stream;
    stream.open(filename);          // Open File requested
    if (!stream.is_open()) {
        std::cout << "Couldn't find file specified" << std::endl;
        printHelp();
        return;
    }
    ANTLRInputStream input(stream);                  // Convert filestream
      ↪ to ANTLR stream

    qasm2Lexer lexer(&input);                        // Lex file
    CommonTokenStream tokens(&lexer);                // get the tokens
    qasm2Parser parser(&tokens);                     // send to antlr parser

    qasm2Parser::MainprogContext* tree = parser.mainprog();
      ↪              // Fetch AST tree

    qasm2BaseVisitor visitor;
    visitor.visitMainprog(tree);
      ↪                                              // Use custom visitor
      ↪  to process information
    std::vector<Register> registers = visitor.getRegisters();
      ↪             // Get registers defined by user
    std::vector<GateRequest> gateRequests = visitor.getGates();
      ↪           // Get gates defined by user
    Stager stage = Stager();
    std::vector<ConcurrentBlock> blocks = stage.stageInformation(
      ↪ registers, gateRequests);          // User stager to convert
      ↪  parsed information into calculation commands
    CPUDevice device = CPUDevice();
    if (!SV) {
      ↪                                                                   //
      ↪ If we are in statevector compute mode, run in statevector
      ↪ mode
        device.run(stage.getRegisters(), blocks);
    }
    else {
        device.runSV(stage.getRegisters(), blocks);
    }
    StateVectorMeasurement measure = StateVectorMeasurement(device.
      ↪ getStateVector(), registers);         // Initialise
      ↪ statevector measurement
    measure.measure();
    std::vector<MeasureCommand> commands = visitor.getMeasureCommands()
      ↪ ;
    measure.loadMeasureCommands(commands);
    measure.passMeasurementsIntoClassicalRegisters();
    if (!jsonMode) {
        measure.printClassicalRegisters();
    }
    else {
        JSONify json = JSONify(measure.getAllRegisters(), device.
          ↪ getStateVector());                            // If requested
```

```
            ↪ print in JSON format
        json.printJSON();
    }
}

void GPURun(std::string filename, bool SV, bool jsonMode) {
    std::ifstream stream;
    stream.open(filename);          // Open File requested
    if (!stream.is_open()) {
        std::cout << "Couldn't find file specified" << std::endl;
        printHelp();
        return;
    }
    ANTLRInputStream input(stream);                 // Convert
      ↪ filestream to ANTLR stream

    qasm2Lexer lexer(&input);                       // Lex file
    CommonTokenStream tokens(&lexer);               // get the tokens
    qasm2Parser parser(&tokens);                    // send to antlr
      ↪ parser

    qasm2Parser::MainprogContext* tree = parser.mainprog();
      ↪             // Fetch AST tree

    qasm2BaseVisitor visitor;
    visitor.visitMainprog(tree);
      ↪                                     // Use custom visitor
      ↪  to process information
    std::vector<Register> registers = visitor.getRegisters();
      ↪           // Get registers defined by user
    std::vector<GateRequest> gateRequests = visitor.getGates();
      ↪         // Get gates defined by user
    Stager stage = Stager();
    std::vector<ConcurrentBlock> blocks = stage.stageInformation(
      ↪ registers, gateRequests);          // User stager to convert
      ↪  parsed information into calculation commands
    GPUDevice device = GPUDevice();
    if (!SV) {
      ↪                                                     //
      ↪ If we are in statevector compute mode, run in statevector
      ↪ mode
        device.run(stage.getRegisters(), blocks);
    }
    else {
        device.runSV(stage.getRegisters(), blocks);
    }
    StateVectorMeasurement measure = StateVectorMeasurement(device.
      ↪ getStateVector(), registers);        // Initialise
      ↪ statevector measurement
    measure.measure();
    std::vector<MeasureCommand> commands = visitor.getMeasureCommands()
      ↪ ;
    measure.loadMeasureCommands(commands);
    measure.passMeasurementsIntoClassicalRegisters();
    if (!jsonMode) {
        measure.printClassicalRegisters();
    }
    else {
```

```
        JSONify json = JSONify(measure.getAllRegisters(), device.getStateVector());                   // If requested print in JSON format
        json.printJSON();
    }
}

void handleInfoRequest(std::vector<std::string> arguments)
{
    for (auto argument : arguments) {
        if (argument == "-gpuInfo") {
            DisplayHeader();
        }
        if (argument == "-test") {
            ValkyrieTests tester = ValkyrieTests();
            tester.runTests();
            std::cout << "Number of Tests passed: " << tester.noPassed() << std::endl;
            std::cout << "Test pass percentage: " << tester.getPercentagePassed() << std::endl;
            if (tester.getPercentagePassed() != 100.0) {
                for (auto fail : tester.testsFailed()) {
                    std::cout << "Test Failed: " << fail << std::endl;
                }
            }
        }
    }
}
```

Listing B.1: kernel.cu: Main compilation file, accepts input and calls functions to run Valkyrie

```

// Generated from qasm2.g4 by ANTLR 4.9.2
// Completed by Neelesh Ravichandran

#pragma once


#include "antlr4-runtime.h"
#include "qasm2Visitor.h"
#include "BaseTypes.h"
#include <cmath>
#include "ParsingGateUtilities.h"
#include <map>

/*
    qasm2BaseVisitor.h
    Description: File provides implementation for QASM2 visitation

*/


const double PI = 3.1415926535;

enum unaryOp {
    SIN_,
    COS_,
    TAN_,
    EXP_,
```

```
    LN_,
    SQRT_
};




/**
 * This class provides an empty implementation of qasm2Visitor, which
    ↪ can be
 * extended to create a visitor which only needs to handle a subset of
    ↪ the available methods.
 */
class  qasm2BaseVisitor : public qasm2Visitor {
private:
    int debugLevel = 1;
    std::vector<Register> registers_;
    std::vector<GateRequest> gates_;
    std::vector<MeasureCommand> commands_;
    std::map<std::string, std::function <std::vector<GateRequest>(std::
        ↪ vector<double> params, idLocationPairs idLoc)>> customGates_
        ↪ ;
    bool gateDeclMode = false;

    int findRegWidth(std::string identifier) {
        for (auto register_ : registers_) {
            if (register_.getName() == identifier) {
                if (register_.isQuantum()) {
                    return register_.getQuantumRegister().getWidth();
                }
                else {
                    return register_.getClassicalRegister().getWidth();
                }
            }
        }
        return -1;
    }

    int findReg(std::string identifier) {
        for (int i = 0; i < registers_.size(); i++) {
            if (registers_[i].getName() == identifier) {
                return i;
            }
        }
        return -1;
    }

    void attachGates(std::vector<GateRequest> gates) {
        for (int i = 0; i < gates.size(); i++) {
            gates_.push_back(gates[i]);
        }
    }

    idLocationPairs makePair(idLocationPairs p1, int i1) {
        idLocationPairs newPair;
        newPair.identifiers.push_back(p1.identifiers[i1]);
        newPair.locations.push_back(p1.locations[i1]);
        return newPair;
```

```
    }

public:

    std::vector<Register> getRegisters() {
        return registers_;
    }

    std::vector<GateRequest> getGates() {
        return gates_;
    }

    std::vector<MeasureCommand> getMeasureCommands() {
        return commands_;
    }

  // visitMainprog provides parsing logic for program as a whole
  virtual antlrcpp::Any visitMainprog(qasm2Parser::MainprogContext *ctx
      ↪ ) override {
      if (ctx->version()) {
          HeaderData headerD = visitVersion(ctx->version()).as<
              ↪ HeaderData>();
          std::vector<qasm2Parser::StatementContext*> statements = ctx
              ↪ ->statement();
          for (auto statement : statements) {
              visitStatement(statement);
          }
      }
      return 1;
  }

  // visitStatement provides parsing logic for visiting a single
      ↪ statement as a whole
  virtual antlrcpp::Any visitStatement(qasm2Parser::StatementContext *
      ↪ ctx) override {
      if (ctx->decl()) {
          /*Register newRegister = visitDecl(ctx->decl()).as<Register
              ↪ >();
          registers_.push_back(newRegister);*/
      }
      else if (ctx->qop()) {

      }
      else if (ctx->gatedecl() && ctx->goplist()) {
          gateDeclMode = true;
          gateDeclaration gDecl = visitGatedecl(ctx->gatedecl()).as<
              ↪ gateDeclaration>();
          std::vector<gateOp> gateOps = visitGoplist(ctx->goplist()).as
              ↪ <std::vector<gateOp>>();
          gateDeclMode = false;
          customGates_[gDecl.gateName] = compileCustomGate(gDecl,
              ↪ gateOps);
          return 1;
      }
      else {
          return 1;
      }
    return visitChildren(ctx);
```

```
    }

    // visitVersion provides parsing logic for the qasm version
    virtual antlrcpp::Any visitVersion(qasm2Parser::VersionContext *ctx)
       ↪ override {       // Complete
        if (ctx->REAL()) {
            std::vector<std::string> includes;
            HeaderData header(ctx->REAL()->toString(), includes);
            return header;
        }
      return visitChildren(ctx);
    }

    // visitVersion provides parsing logic for a single declaration such
       ↪ as qreg or creg
    virtual antlrcpp::Any visitDecl(qasm2Parser::DeclContext *ctx)
       ↪ override {     // Complete
          antlr4::tree::TerminalNode* id = ctx->ID();
          std::string identifier = id->getText();
          antlr4::tree::TerminalNode* intVal = ctx->INT();
          std::string widthString = intVal->getText();
        if (ctx->getStart()->getText() == "qreg") {
            QuantumRegister qReg = QuantumRegister(identifier, std::stoi(
                ↪ widthString));
            Register reg(quantum_, qReg);
            registers_.push_back(reg);
            return 0;
        }
        else {
            ClassicalRegister cReg = ClassicalRegister(identifier, std::
                ↪ stoi(widthString));
            Register reg(classical_, cReg);
            registers_.push_back(reg);
            return 0;
        }
        return 1;
    }

    // visitVersion provides parsing logic for a custom gate declaration
    virtual antlrcpp::Any visitGatedecl(qasm2Parser::GatedeclContext *ctx
       ↪ ) override {
        if (ctx->idlist().size() == 1) {
            std::vector<std::string> idLocNames = visitIdlist(ctx->idlist
                ↪ ()[0]);
            gateDeclaration gDecl;
            gDecl.gateName = ctx->ID()->getText();
            gDecl.idLocList = idLocNames;
            return gDecl;
        }
        else {
            std::vector<std::string> idLocNames = visitIdlist(ctx->idlist
                ↪ ()[1]);
            std::vector<std::string> paramNames = visitIdlist(ctx->idlist
                ↪ ()[0]);
            gateDeclaration gDecl;
            gDecl.gateName = ctx->ID()->getText();
            gDecl.idLocList = idLocNames;
            gDecl.paramList = paramNames;
```

```
        return gDecl;
    }
  return visitChildren(ctx);
}

// visitVersion provides parsing logic for a custom gate operation
   ↪ declaration
virtual antlrcpp::Any visitGoplist(qasm2Parser::GoplistContext *ctx)
   ↪ override {
    if (ctx->uop().size() > 0) {
        std::vector<gateOp> gateOperations;
        for (auto uop : ctx->uop()) {
            gateOp gop = visitUop(uop).as<gateOp>();
            gateOperations.push_back(gop);
        }
        return gateOperations;
    }
    return 1;
}

// visitQop provides parsing logic for a quantum operation
virtual antlrcpp::Any visitQop(qasm2Parser::QopContext *ctx) override
   ↪ {
  // Incomplete -measure
  if (ctx->getStart()->getText() == "measure") {
      if (ctx->argument().size() == 2) {
          idLocationPairs pairs1 = visitArgument(ctx->argument()[0]);
          idLocationPairs pairs2 = visitArgument(ctx->argument()[1]);
          if (pairs1.getSize() == pairs2.getSize()) {
              for (int i = 0; i < pairs1.getSize(); i++) {
                  idLocationPairs p1 = makePair(pairs1, i);
                  idLocationPairs p2 = makePair(pairs2, i);
                  MeasureCommand command = MeasureCommand(p1, p2);
                  commands_.push_back(command);
              }
          }
      }
  }
  return visitChildren(ctx);
}

// visitUop provides parsing logic for a unitary gate operation
virtual antlrcpp::Any visitUop(qasm2Parser::UopContext *ctx) override
   ↪ {
    if (!gateDeclMode) {
        if (ctx->getStart()->getText() == "U") {
            if (ctx->explist()) {
                std::vector<double> gateArguments = visitExplist(ctx
                   ↪ ->explist()).as<std::vector<double>>();
                if (gateArguments.size() == 3) {
                    idLocationPairs pairs = visitArgument(ctx->
                       ↪ argument()[0]);
                    if (pairs.identifiers.size() == 1) {
                        GateRequest gate = compileGateRequest("U",
                           ↪ gateArguments, pairs);
                        gates_.push_back(gate);
                    }
                }
```

```
            }
        }
        if (ctx->getStart()->getText() == "CX") {
            idLocationPairs pairs1 = visitArgument(ctx->argument()
                ↪ [0]).as<idLocationPairs>();
            idLocationPairs pairs2 = visitArgument(ctx->argument()
                ↪ [1]).as<idLocationPairs>();
            idLocationPairs combinedPairs;
            for (int i = 0; i < pairs1.identifiers.size(); i++) {
                combinedPairs.identifiers.push_back(pairs1.
                    ↪ identifiers[i]);
                combinedPairs.locations.push_back(pairs1.locations[i
                    ↪ ]);
            }
            for (int i = 0; i < pairs2.identifiers.size(); i++) {
                combinedPairs.identifiers.push_back(pairs2.
                    ↪ identifiers[i]);
                combinedPairs.locations.push_back(pairs2.locations[i
                    ↪ ]);
            }
            if (combinedPairs.identifiers.size() == 2) {
                GateRequest gate = compileGateRequest("CX",
                    ↪ combinedPairs);
                gates_.push_back(gate);
            }
        }
        if (ctx->ID()) {
            std::string uopGate = ctx->ID()->getText();
            bool customGate = customGates_.find(uopGate) !=
                ↪ customGates_.end();
            if (ctx->explist()) {
                std::vector<double> gateArguments = visitExplist(ctx
                    ↪ ->explist()).as<std::vector<double>>();
                if (ctx->anylist()) {
                    if (ctx->anylist()->mixedlist()) {
                        idLocationPairs idLoc = visitMixedlist(ctx->
                            ↪ anylist()->mixedlist()).as<
                            ↪ idLocationPairs>();
                        std::vector<GateRequest> gates;
                        if (customGate) {
                            gates = customGates_[uopGate](
                                ↪ gateArguments, idLoc);
                        }
                        else {
                            gates = compileCompoundGateRequest(
                                ↪ uopGate, gateArguments, idLoc);
                        }
                        attachGates(gates);
                    }
                    else {
                        std::vector<std::string> identifiers =
                            ↪ visitIdlist(ctx->anylist()->idlist()).
                            ↪ as<std::vector<std::string>>();
                        for (auto identifier : identifiers) {
                            int width = findRegWidth(identifier);
                            idLocationPairs pairs;
                            for (int i = 0; i < width; i++) {
                                pairs.identifiers.push_back(
```

```
                       ↪ identifier);
                    pairs.locations.push_back(i);
                }
                std::vector<GateRequest> gates;
                if (customGate) {
                    gates = customGates_[uopGate](
                       ↪ gateArguments, pairs);
                }
                else {
                    gates = compileCompoundGateRequest(
                       ↪ uopGate, gateArguments, pairs)
                       ↪ ;
                }
                attachGates(gates);
            }
        }
    }
}
else {
    bool customGate = customGates_.find(uopGate) !=
       ↪ customGates_.end();
    if (ctx->anylist()) {
        if (ctx->anylist()->mixedlist()) {
            idLocationPairs idLoc = visitMixedlist(ctx->
               ↪ anylist()->mixedlist()).as<
               ↪ idLocationPairs>();
            std::vector<GateRequest> gates;
            if (customGate) {
                gates = customGates_[uopGate](std::vector
                   ↪ <double>(), idLoc);
            }
            else {
                gates = compileCompoundGateRequest(
                   ↪ uopGate, idLoc);
            }
            attachGates(gates);
        }
    }
    else {
        std::vector<std::string> identifiers =
           ↪ visitIdlist(ctx->anylist()->idlist()).as<
           ↪ std::vector<std::string>>();
        for (auto identifier : identifiers) {
            int width = findRegWidth(identifier);
            idLocationPairs pairs;
            for (int i = 0; i < width; i++) {
                pairs.identifiers.push_back(identifier);
                pairs.locations.push_back(i);
            }
            std::vector<GateRequest> gates;
            if (customGate) {
                gates = customGates_[uopGate](std::vector
                   ↪ <double>(), pairs);
            }
            else {
                gates = compileCompoundGateRequest(
                   ↪ uopGate, pairs);
            }
```

```
                            attachGates(gates);
                        }
                    }
                }
            }
        }
        else {
            gateOp gOP;
            if (ctx->getStart()->getText() == "U") {
                gOP.gateName = "U";
            }
            else if (ctx->getStart()->getText() == "CX") {
                gOP.gateName = "CX";
            }
            else {
                gOP.gateName = ctx->ID()->getText();
            }
            std::vector<expEval> paramList;
            if (ctx->explist()) {
                paramList = visitExplist(ctx->explist()).as<std::vector<expEval>>();
            }
            gOP.params = paramList;
            if (ctx->argument().size() > 0) {
                idLocationPairs pairs = visitArgument(ctx->argument()[0]);
                if (ctx->argument().size() == 2) {
                    idLocationPairs pairs2 = visitArgument(ctx->argument()[1]);
                    for (int i = 0; i < pairs2.identifiers.size(); i++) {
                        pairs.identifiers.push_back(pairs2.identifiers[i]);
                    }
                }
                gOP.idLocs = pairs.identifiers;
            }
            else {
                if (ctx->anylist()->idlist()) {
                    gOP.idLocs = visitIdlist(ctx->anylist()->idlist()).as<std::vector<std::string>>();
                }
            }
            return gOP;
        }

      return 0;
    }

    // visitAnyList provides parsing logic for a AnyList parsing
    virtual antlrcpp::Any visitAnylist(qasm2Parser::AnylistContext *ctx) override {             // Complete
        return visitChildren(ctx);
    }

    // visitIdList provides parsing logic for an IdList
    virtual antlrcpp::Any visitIdlist(qasm2Parser::IdlistContext *ctx) override {               // Complete
        std::vector<antlr4::tree::TerminalNode*> ids = ctx->ID();
```

```
        std::vector<std::string> idStrings;
        for (auto id : ids) {
            idStrings.push_back(id->getText());
        }
        return idStrings;
    }

    // visitMixedList provides parsing logic for a MixedList
    virtual antlrcpp::Any visitMixedlist(qasm2Parser::MixedlistContext *
      ↪ ctx) override {             // Complete
        int countID = ctx->ID().size();
        int countINT = ctx->INT().size();
        if (countID == countINT) {
            std::vector<std::string> identifiers;
            for (auto id : ctx->ID()) {
                identifiers.push_back(id->getText());
            }
            std::vector<int> locations;
            for (auto val : ctx->INT()) {
                locations.push_back(std::stoi(val->getText()));
            }
            idLocationPairs idLoc;
            idLoc.identifiers = identifiers;
            idLoc.locations = locations;
            return idLoc;
        }
        else {
            std::string decider = ctx->getToken(sizeof(antlr4::Token), 1)
                ↪ ->getText();
            if (decider == "[") {
                std::vector<std::string> identifiers;
                for (int i = 0; i < countINT; i++) {
                    identifiers.push_back(ctx->ID()[i]->getText());
                }
                std::string finalID = ctx->ID()[countINT]->getText();
                std::vector<int> locations;
                for (auto val : ctx->INT()) {
                    locations.push_back(std::stoi(val->getText()));
                }
                int width = findRegWidth(finalID);
                identifiers.push_back(finalID);
                for (int i = 0; i < width; i++) {
                    locations.push_back(i);
                }
                idLocationPairs idLoc;
                idLoc.identifiers = identifiers;
                idLoc.locations = locations;
                return idLoc;
            }
            if (decider == ",") {
                std::vector<std::string> identifiers;
                std::vector<int> locations;
                for (int i = 0; i < countID - 1; i++) {
                    for (int j = 0; j < findRegWidth(ctx->ID()[j]->
                        ↪ getText()); j++) {
                        identifiers.push_back(ctx->ID()[i]->getText());
                        locations.push_back(j);
                    }
```

```
                }
                std::string finalID = ctx->ID()[countID - 1]->getText();
                identifiers.push_back(finalID);
                locations.push_back(std::stoi(ctx->INT()[0]->getText()));
                idLocationPairs idLoc;
                idLoc.identifiers = identifiers;
                idLoc.locations = locations;
                return idLoc;
            }
        }
      return visitChildren(ctx);
    }

    // visitArgument provides parsing logic for an argument for a gate
    virtual antlrcpp::Any visitArgument(qasm2Parser::ArgumentContext *ctx
       ↪ ) override {             // Complete
        idLocationPairs pairs;
        if (ctx->INT()) {
            pairs.identifiers.push_back(ctx->ID()->getText());
            pairs.locations.push_back(std::stoi(ctx->INT()->getText()));
        }
        else {
            if (!gateDeclMode) {
                std::string register_ = ctx->ID()->getText();
                int width = findRegWidth(register_);
                for (int i = 0; i < width; i++) {
                    pairs.identifiers.push_back(register_);
                    pairs.locations.push_back(i);
                }
            }
            else {
                pairs.identifiers.push_back(ctx->ID()->getText());
            }
        }
        return pairs;
    }

    // visitExpList provides parsing logic for an expList for a gate
    virtual antlrcpp::Any visitExplist(qasm2Parser::ExplistContext *ctx)
       ↪ override {             // Complete
        if (!gateDeclMode) {
            std::vector<double> values;
            for (auto exp : ctx->exp()) {
                double value = visitExp(exp).as<double>();
                values.push_back(value);
            }
            return values;
        }
        else {
            std::vector<expEval> values;
            for (auto exp : ctx->exp()) {
                expEval value = visitExp(exp).as<expEval>();
                values.push_back(value);
            }
            return values;
        }
    }

```

```
    // visitExp provides parsing logic for any general expression
    virtual antlrcpp::Any visitExp(qasm2Parser::ExpContext *ctx) override
       ↪ {
        if (!ctx->unaryop()) {
            if (!ctx->ID()) {
                std::vector<qasm2Parser::ExpContext*> subexpressions =
                    ↪ ctx->exp();
                if (subexpressions.size() == 0) {
                    if (ctx->getStart()->getText() == "pi") {
                        if (gateDeclMode) {
                            expEval exp;
                            exp.identNotVal = false;
                            exp.value = PI;
                            return exp;
                        }
                        return PI;
                    }
                    double value = 0;
                    if (ctx->REAL()) {
                        std::string unparsed = ctx->REAL()->getText();
                        value = std::stod(unparsed);
                    }
                    else if (ctx->INT()) {
                        std::string unparsed = ctx->INT()->getText();
                        value = std::stod(unparsed);
                    }
                    if (gateDeclMode) {
                        expEval exp;
                        exp.identNotVal = false;
                        exp.value = value;
                        return exp;
                    }
                    return value;
                }
                if (subexpressions.size() == 1) {
                    if (ctx->getStart()->getText() == "-") {
                        return -1 * visitExp(subexpressions[0]).as<double
                            ↪ >();
                    }
                    else {
                        return visitExp(subexpressions[0]);
                    }
                }
                if (subexpressions.size() == 2) {
                    std::string operator_ = ctx->getToken(sizeof(antlr4::
                        ↪ Token), 1)->getText();
                    if (operator_ == "+") {
                        return visitExp(subexpressions[0]).as<double>() +
                            ↪ visitExp(subexpressions[1]).as<double>();
                    }
                    if (operator_ == "-") {
                        return visitExp(subexpressions[0]).as<double>() -
                            ↪ visitExp(subexpressions[1]).as<double>();
                    }
                    if (operator_ == "*") {
                        return visitExp(subexpressions[0]).as<double>() *
                            ↪ visitExp(subexpressions[1]).as<double>();
                    }
```

```
                if (operator_ == "/") {
                    return visitExp(subexpressions[0]).as<double>() /
                       ↪  visitExp(subexpressions[1]).as<double>();
                }
            }
        }
    }
    if (ctx->ID()) {
        expEval val;
        val.identNotVal = true;
        val.ident = ctx->ID()->getText();
        return val;
    }
    if (ctx->unaryop()) {
        double expressionVal = visitExp(ctx->exp()[0]);
        unaryOp operation_ = visitUnaryop(ctx->unaryop()).as<unaryOp
           ↪ >();
        switch (operation_) {
        case SIN_:
            return std::sin(expressionVal);
            break;
        case COS_:
            return std::cos(expressionVal);
            break;
        case TAN_:
            return std::tan(expressionVal);
            break;
        case EXP_:
            return std::exp(expressionVal);
            break;
        case LN_:
            return std::log(expressionVal);
            break;
        case SQRT_:
            return std::pow(expressionVal, 0.5);
            break;
        }
    }
  return visitChildren(ctx);
}

// visitUnaryOp provides parsing logic for generic unary operation
virtual antlrcpp::Any visitUnaryop(qasm2Parser::UnaryopContext *ctx)
   ↪ override {                    // Complete
    std::string operation_ = ctx->getText();
    if (operation_ == "sin") {
        return SIN_;
    }
    if (operation_ == "cos") {
        return COS_;
    }
    if (operation_ == "tan") {
        return TAN_;
    }
    if (operation_ == "exp") {
        return EXP_;
    }
    if (operation_ == "ln") {
```

```
          return LN_;
      }
      if (operation_ == "sqrt") {
          return SQRT_;
      }
      return SIN_;
  }
};
```

Listing B.2: qasmBaseVisitor.h: Visits all nodes for generated AST tree and processes information into registers and gates

```
#pragma once
#include "BaseTypes.h"
#include <map>
#include <string>
#include <iostream>

/*
 AbstractDevice.h
 Description: Defines interface for quantum processing devices to be
    ↪ implemented either on CPU or GPU.
 The common interface allows us to simplify function calling from
    ↪ higher order files.

 Defined Classes:
 (Abstract) AbstractQubitFactory
 (Abstract) AbstractGateFactory
 (Abstract) AbstractQuantumCircuit
 (Abstract) AbstractQuantumProcessor
 (Abstract) AbstractDevice

*/

// AbstractQubitFactory defines interface for a qubit factory. This
    ↪ class when implemented, will allocate heap memory and
// create instances of the Qubit class (defined in BaseTypes.h) to hold
    ↪  the individual states and names of qubits.
class AbstractQubitFactory {
private:
 DeviceType type_;
public:
 virtual Qubit* generateQubit() = 0;
};

// AbstractGateFactory defines interfact for a gate factory. This class
    ↪  when implemented, will allocate heap memory and
// create instances of the Gate class (defined in BaseTypes.h) to hold
    ↪ primitive versions of gates.
class AbstractGateFactory {
private:
 DeviceType type_;
public:
 virtual Gate* generateGate(GateRequest request) = 0;
};

// AbstractQuantumCircuit defines the interface for quantum circuits.
    ↪ These are data-structures which will collect the
// calculations requested by the user, and convert them into operable
```

```
matrix calculations.
class AbstractQuantumCircuit {
private:
 DeviceType type_;
 bool done_;
public:
 virtual void loadQubitMap(std::map<std::string, std::vector<Qubit*>> qubitMap) = 0;
 virtual void loadBlock(ConcurrentBlock block) = 0;
 virtual std::vector<Calculation> getNextCalculation() = 0;
 virtual std::map<std::string, std::vector<Qubit*>> returnResults() = 0;
 virtual StateVector* getStateVector() = 0;
 virtual bool checkComplete() = 0;
};

// AbstractQuantumProcessor defines the interface for the actual calculation part of the quantum simulation
class AbstractQuantumProcessor {
private:
 DeviceType type_;
 AbstractQuantumCircuit* circuit_;
public:
 virtual void loadCircuit(AbstractQuantumCircuit* circuit) = 0;
 virtual void calculate() = 0;
 virtual std::map<std::string, std::vector<Qubit*>> qubitMapfetchQubitValues() = 0;
};

// AbstractDevice provides an interface for the computation device as a whole. This structure will be used to
// collect the classes defined above for a calling function to easily have access to quantum calculation.
class AbstractDevice {
private:
 DeviceType type_;
public:
 virtual void loadRegister(Register registerx) = 0;
 virtual void transferQubitMap() = 0;
 virtual void loadConcurrentBlock(ConcurrentBlock block) = 0;
 virtual void runSimulation() = 0;
 virtual void run(std::vector<Register> registers, std::vector<ConcurrentBlock> blocks) = 0;
 virtual std::map<std::string, std::vector<Qubit*>> revealQuantumState() = 0;
};
```

Listing B.3: AbstractDevice.h: Defines interface for quantum processing devices to be implemented either on CPU or GPU. The common interface allows us to simplify function calling from higher order files.

```
#pragma once
#include <vector>
#include <string>
#include <complex>
#include <map>
#include <iostream>

/*
```

```
  BaseTypes.h
  Description: Defines all common datatypes used throughout the codebase

  Defined Classes:
  idLocationPairs
  SVPair
  expEval
  doubleOrArg
  gateDeclaration
  gateOp
  DeviceType
  HeaderData
  RegisterType
  QuantumRegister
  ClassicalRegister
  Register
  GateRequestType
  GateRequest
  ConcurrentBlock
  Qubit
  Gate
  Calculation
  MeasureCommand
  StateVector
*/


struct idLocationPairs {
 // idLocationPairs is a datastructure used extensively in the parsing stage of Valkyrie, used to relate
 // which exact qubit(s) a particular gate is supposed to be operating on.
 std::vector<std::string> identifiers;
 std::vector<int> locations;
 int getSize() {
  return identifiers.size();
 }
};

// SVPair similar ot idLocationPairs but to be used exclusively in statevector manipulation. The SVPair
// datastructure is used as a key when navigating the various combination of states in the Statevector.
struct SVPair {
 std::string name_;
 int location_;
 SVPair(std::string name, int location) {
  name_ = name;
  location_ = location;
 }
 bool areEqual(SVPair comp) {
  return comp.location_ == location_ && comp.name_ == name_;
 }
};

// expEval used in the parsing of custom gate definitions, allows the program to distinguish between a
// variable or constant being used as input in a gate.
```

```
struct expEval {
 std::string ident;
 double value;
 bool identNotVal;
};

// doubleOrArg conversion of an expEval into an expression of which position in a variable list to get
// a parameter or the constant value of the parameter provided in gate definition.
struct doubleOrArg {
 bool doubleNotArg;
 double valD;
 int position;
};

// gateDeclaration datastructure to be used to carry custom gate declaration header after parsing.
struct gateDeclaration {
 std::string gateName;
 std::vector<std::string> idLocList;
 std::vector<std::string> paramList;
};

// gateOp datastructure to hold representation of a gate operation in a custom gate, uses the
// expVal datastructrure to differentiate between constant parameters and required parameters
struct gateOp {
 std::string gateName;
 std::vector<expEval> params;
 std::vector<std::string> idLocs;
};

// DeviceType enumeration of what kind of device we are using to run quantum simulations.
enum DeviceType {
 CPU_,
 GPU_,
 INVALID
};

// HeaderData datastructure to hold header data provided in the OPENQASM standard.
class HeaderData {
private:
 double openQASMStandard_ = 0.0;
 std::vector<std::string> includeFiles_;
public:
 HeaderData(std::string value, std::vector<std::string> includes) {
  openQASMStandard_ = std::stod(value);
  includeFiles_ = includes;
 }
};

// RegisterType enumeration for what kind of register is being instantiated.
enum RegisterType {
```

```
 quantum_,
 classical_,
 invalid_
};

// QuantumRegister datastructure to hold a parsed representation of a quantum register.
// This datastructure holds no qubits, but will be used by the staging module to
// generate qubit construction instructions.
class QuantumRegister {
private:
 std::string identifier_;
 int width_;
public:
 QuantumRegister(std::string identifier, int width) {
  identifier_ = identifier;
  width_ = width;
 }
 std::string getIdentifier() {
  return identifier_;
 }
 int getWidth() {
  return width_;
 }
 bool isQubit() {
  return width_ == 1;
 }
 QuantumRegister() = default;
};

// QuantumRegister datastructure to hold a parsed representation of a classical register.
// This datastructure does hold classical bits via the int representation.
class ClassicalRegister {
private:
 std::string identifier_;
 int width_;
 std::vector<int> values_;
public:
 ClassicalRegister(std::string identifier, int width) {
  identifier_ = identifier;
  width_ = width;
  for (int i = 0; i < width; i++) {
   values_.push_back(0);
  }
 }
 std::string getIdentifier() {
  return identifier_;
 }
 int getWidth() {
  return width_;
 }
 void setValue(int i, int val) {
  values_[i] = val;
 }
 int getValue(int i) {
```

```
  return values_[i];
 }
 ClassicalRegister() = default;
};

// Register is a wrapper for Quantum and Classical registers, allowing the staging module
// to differentiate between the two.
class Register {
private:
 RegisterType regType_;
 QuantumRegister qReg_;
 ClassicalRegister cReg_;
public:
 Register(RegisterType type, QuantumRegister qreg) {
  regType_ = type;
  qReg_ = qreg;
 }
 Register(RegisterType type, ClassicalRegister creg) {
  regType_ = type;
  cReg_ = creg;
 }

 QuantumRegister getQuantumRegister() {
  return qReg_;
 }

 ClassicalRegister getClassicalRegister() {
  return cReg_;
 }

 void setClassicalRegister(ClassicalRegister cReg) {
  cReg_ = cReg;
 }

 std::string getName() {
  if (regType_ == quantum_) {
   return qReg_.getIdentifier();
  }
  else {
   return cReg_.getIdentifier();
  }
 }

 bool isQuantum() {
  return regType_ == quantum_;
 }
};

// GateRequestType defines an enumeration for the primitive gate types U and CX as well
// as all qeLib1 gates. Allowing for efficient compilation
enum GateRequestType {
 I,
 U,
 CX,
 h,
 cx,
```

```
 u3,
 u2,
 u1,
 id,
 u0,
 u,
 p,
 x,
 y,
 z,
 s,
 sdg,
 t,
 tdg,
 rx,
 ry,
 rz,
 sx,
 sxdg,
 cz,
 cy,
 swap,
 ch,
 ccx,
 cswap,
 crx,
 cry,
 crz,
 cu1,
 cp,
 cu3,
 csx,
 cu,
 rxx,
 rzz,
 rccx,
 rc3x,
 c3x,
 c3sqrtx,
 c4x,
 CUSTOM
};

// GateRequest is a datastructure to represent a user commanded gate operation, will be used by
// computation device to generate the gate matrices itself
class GateRequest {
private:
 GateRequestType gateType_;
 std::vector<std::string> registerIdentifiers_;
 std::vector<int> locations_;
 std::vector<double> parameters_;

public:
 GateRequest(){}
 GateRequest(GateRequestType type) {
  gateType_ = type;
 }
```

```
 void addressQubit(std::string registerID, int location) {
  registerIdentifiers_.push_back(registerID);
  locations_.push_back(location);
 }
 void setParameters(std::vector<double> params) {
  parameters_ = params;
 }
 void addParameter(double value) {
  parameters_.push_back(value);
 }
 int getGateDim() {
  return registerIdentifiers_.size();
 }
 std::vector<std::string> getRegisters() {
  return registerIdentifiers_;
 }
 std::vector<int> getLocations() {
  return locations_;
 }
 GateRequestType getGateType() {
  return gateType_;
 }
 std::vector<double> getParameters() {
  return parameters_;
 }
};

// ConcurrentBlock represent a block of gates which cna be processed in parallel without affecting
// accuracy of the computation. Used by staging module to send gates to Device
class ConcurrentBlock {
private:
 int count_ = 0;
 std::vector<GateRequest> gates_;
public:
 ConcurrentBlock(int count) {
 }
 void addGate(GateRequest newGate) {
  gates_.push_back(newGate);
  count_++;
 }
 void setCount(int count) {
  count_ = count;
 }
 int getCount() {
  return count_;
 }
 std::vector<GateRequest> getGates() {
  return gates_;
 }
};

// Qubit is the basic Qubit representation which is used to initially store qubit values. If
// Valkyrie is in fast computation mode then Qubit's are used exclusively to store the idividual states
class Qubit {
```

```cpp
private:
 std::complex<double>* s_0;
 std::complex<double>* s_1;
public:
 Qubit(std::complex<double>* s0, std::complex<double>* s1) {
  s_0 = s0;
  s_1 = s1;
 }

 std::complex<double>* fetch(int i) {
  if (i == 0) {
   return s_0;
  }
  else {
   return s_1;
  }
 }
};

// Gate provides a basic gate representation for the computation device to perform matrix operations.
// If Valkyrie is in fast computation mode then the Gate matrix is directly used for computation.
class Gate {
private:
 std::vector<std::vector<std::complex<double>>> gateArray_;
 int m_; // dimensions
 int n_;
public:
 Gate(int m, int n, std::vector<std::vector<std::complex<double>>> gateArray) {
  m_ = m;
  n_ = n;
  gateArray_ = gateArray;
 }
 std::complex<double> fetchValue(int x, int y) {
  return gateArray_[x][y];
 }
 std::vector<std::vector<std::complex<double>>> getArray() {
  return gateArray_;
 }
 int getM() {
  return m_;
 }
 int getN() {
  return n_;
 }

};

// Calculation is a class which is used by both fast and statevector computation modes
// It holds the primitive gate and qubit values or state vector locations that are used
// by the matrix processing modules.
class Calculation {
private:
 Gate* gate_;
```

```
 std::vector<Qubit*> qubitValues_;
 std::vector<SVPair> locations_;

 // getNewOrder1 under the tensor product reordering procedure, this
    ↪ function
 // is able to shuffle the qubit that this Calculation is concerning
    ↪ right to the end
 // of the tensor product stack
 std::vector<SVPair> getNewOrder1(std::vector<SVPair> oldOrder) {
  std::vector<SVPair> newOrder;
  for (int i = 0; i < oldOrder.size(); i++) {
   if (!locations_[0].areEqual(oldOrder[i])) {
    newOrder.push_back(oldOrder[i]);
   }
  }
  newOrder.push_back(locations_[0]);
  return newOrder;
 }

 // getNewOrder2 under the tensor product reordering procedure, this
    ↪ function
 // is able to shuffle the two qubits that this Calculation is
    ↪ concerning right to the end
 // of the tensor product stack
 std::vector<SVPair> getNewOrder2(std::vector<SVPair> oldOrder) {
  std::vector<SVPair> newOrder;
  for (int i = 0; i < oldOrder.size(); i++) {
   if (!locations_[0].areEqual(oldOrder[i]) && !locations_[1].areEqual(
      ↪ oldOrder[i])) {
    newOrder.push_back(oldOrder[i]);
   }
  }
  newOrder.push_back(locations_[0]);
  newOrder.push_back(locations_[1]);
  return newOrder;
 }
public:
 Calculation(Gate* gate, std::vector<Qubit*> qubitVals, std::vector<
    ↪ SVPair> locations) {
  gate_ = gate;
  qubitValues_ = qubitVals;
  locations_ = locations;
 }
 Gate* getGate() {
  return gate_;
 }
 Qubit* getQubit(int i) {
  return qubitValues_[i];
 }

 std::vector<SVPair> getLocations() {
  return locations_;
 }
 // getNewOrder is important for the statevector computation mode. When
    ↪  performing the
 // matrix multiplication we are using a specialised tensor product (
    ↪ for efficiency) which
 // relies on the two concerned qubits to be pushed to the back of the
```

```
    ↪ tensor product stack.
 std::vector<SVPair> getNewOrder(std::vector<SVPair> oldOrder) {
  if (locations_.size() != 2 && locations_.size() != 1) {
   return oldOrder;
  }
  if (locations_.size() == 2) {
   return getNewOrder2(oldOrder);
  }
  return getNewOrder1(oldOrder);
 }

 std::vector<Qubit*> getQubits() {
  return qubitValues_;
 }
};

// MeasureCommand provides a simple datastructure to track measurement
    ↪ commands from
// the user during parsing.
class MeasureCommand {
private:
 idLocationPairs from_;
 idLocationPairs to_;
public:
 MeasureCommand(idLocationPairs from, idLocationPairs to) {
  from_ = from;
  to_ = to;
 }

 idLocationPairs getFrom() {
  return from_;
 }

 idLocationPairs getTo() {
  return to_;
 }
};

// StateVector is a core component of the quantum computation stack.
// In Fast compute mode, Statevector is used to store the overall
    ↪ results of the computation
// In Statevector computer mode, the Statevector is used both in the
    ↪ input and output of the computation
class StateVector {
private:
 // positions_ stores the current locations of different Quantum
    ↪ register and position pairs (each defining a qubit)
 // these locations are relevant to the order in which these qubits
    ↪ were multiplied in the tensor product used
 // to generate the StateVector
 std::vector<SVPair> positions_;
 // state_ is the statevector in full tensorproduct form, for a system
    ↪ which uses n qubits the state_variable
 // will be 2^n long
 std::vector<std::complex<double>> state_;
 // For Fast computation mode, the qubitMap_ provides access to the
    ↪ actual qubit values stored in the Qubit datastructure
 std::map<std::string, std::vector<Qubit*>>* qubitMap_;
```

```
// reordered_ is a temporary state vector used during computation to
    ↪ represent the temporary reordering
// of the state vector for the tail computation
StateVector* reordered_;
bool isReorder = false;

// inverseTail provides the inverse of the tail function, this allows
    ↪ us to calculate (given the position of
// the qubit in the tensor product and the location in the statevector
    ↪ ) which component of the qubit state (0th or 1th component)
// we need to process on.
int inverseTail(int nTotal, int indexInPositions, int
    ↪ locationInStateVec) {
 int j = std::pow(2, (nTotal − indexInPositions));
 if ((locationInStateVec % j) < (j / 2)) {
  return 0;
 }
 else {
  return 1;
 }
}

// tail provides a function (using inverseTail) to calculate whether
    ↪ we need the 0th or 1th component
// of a particular qubit.
bool tail(int nTotal, int indexInPositions, int locationInStateVec,
    ↪ int index) {
 return inverseTail(nTotal, indexInPositions, locationInStateVec) ==
     ↪ index;
}

// used in Fast computation mode to calculate which values in a
    ↪ statevector is affected by a particular
// calculation result.
std::vector<int> affectedValues(int loc1, int index1, int loc2, int
    ↪ index2) {
 std::vector<int> affected;
 int n = positions_.size();
 for (int i = 0; i < state_.size(); i++) {
  if (tail(n, loc1, i, index1) && tail(n, loc2, i, index2)) {
   affected.push_back(i);
  }
 }
 return affected;
}

// calculateNewVals is used in fast computation mode for keeping track
    ↪  of qubit values changing compensating
// in the state vector.
void calculateNewVals(int pos1, int pos2, std::vector<std::complex<
    ↪ double>> newValues, int loc1Index, int loc2Index) {
 int pos = pos1 * 2 + pos2;
 std::complex<double> newVal = newValues[pos];
 std::vector<int> affected = affectedValues(loc1Index, pos1, loc2Index
     ↪ , pos2);
 int n = positions_.size();
 for (int position : affected) {
  std::complex < double> val = newVal;
```

```
   for (int i = 0; i < positions_.size(); i++) {
    if (i != loc1Index && i != loc2Index) {
     int tailedPos = inverseTail(n, i, position);
     std::complex<double> value = *(qubitMap_->find(positions_[i].name_
        ↪ )->second[positions_[i].location_]->fetch(tailedPos));
     val = val * value;
    }
   }
   state_[position] = val;
  }
 }

 int searchIndex(SVPair val) {
  for (int i = 0; i < positions_.size(); i++) {
   SVPair res = positions_[i];
   if (res.name_ == val.name_ && res.location_ == val.location_) {
    return i;
   }
  }
  return -1;
 }

 int searchIndex(SVPair val, std::vector<SVPair> positions) {
  for (int i = 0; i < positions.size(); i++) {
   SVPair res = positions[i];
   if (res.name_ == val.name_ && res.location_ == val.location_) {
    return i;
   }
  }
  return -1;
 }

 std::vector<int> mapToOldScheme(std::vector<int> values, std::vector<
    ↪ SVPair> newScheme, std::vector<SVPair> oldScheme) {
  std::vector<int> oldValues;
  int n = values.size();
  oldValues.resize(n);
  for (int i = 0; i < n; i++) {
   oldValues[i] = values[searchIndex(oldScheme[i], newScheme)];
  }
  return oldValues;
 }

 // resolvePosition calculates which position in the statevector is
    ↪ addressed by the values given
 int resolvePosition(std::vector<int> values) {
  int n = values.size();
  int position = 0;
  for (int i = 0; i < n; i++) {
   int j = n - i;
   int val = values[i] * std::pow(2, j - 1);
   position += val;
  }
  return position;
 }

public:
 StateVector() {};
```

```
StateVector(std::map<std::string, std::vector<Qubit*>>* linkToQubits) {
 qubitMap_ = linkToQubits;
 initialiseReorder();
}

std::vector<std::complex<double>> getState() {
 return state_;
}

void initialiseReorder() {
 reordered_ = new StateVector();
 reordered_->setReorder(true);
}

void setReorder(bool reorder) {
 isReorder = reorder;
}

// Used in initialisation of StateVector, tensorProduct produces the default statevector which is populated by
// calculation and returned at the end.
void tensorProduct() {
 for (std::map<std::string, std::vector<Qubit*> >::iterator it = qubitMap_->begin(); it != qubitMap_->end(); ++it) {
  for (int i = 0; i < it->second.size(); i++) {
   SVPair pair(it->first, i);
   positions_.push_back(pair);
  }
 }
 int n = positions_.size();
 int dimStateVec = std::pow(2, n);
 state_.resize(dimStateVec);
 for (int i = 0; i < dimStateVec; i++) {
  std::complex<double> start = 1;
  for (int j = 0; j < n; j++) {
   int element = inverseTail(n, j, i);
   SVPair resolvedPair = positions_[j];
   Qubit* qubit = qubitMap_->find(resolvedPair.name_)->second[resolvedPair.location_];
   start = start * *(qubit->fetch(element));
  }
  state_[i] = start;
 }
}

// Will only be called during reordering, provides same function as standed tensorProduct function
void tensorProduct(std::vector<SVPair> newOrder, std::vector<SVPair> oldOrder, std::vector<std::complex<double>> oldState) {
 positions_ = newOrder;
 int n = positions_.size();
 int dimStateVec = std::pow(2, n);
 state_.resize(dimStateVec);

 for (int i = 0; i < dimStateVec; i++) {
  std::vector<int> newSchemeVals;
  for (int j = 0; j < n; j++) {
```

```
   newSchemeVals.push_back(inverseTail(n, j, i));
  }
  std::vector<int> oldSchemeVals = mapToOldScheme(newSchemeVals, newOrder, oldOrder);
  state_[i] = oldState[resolvePosition(oldSchemeVals)];
 }

}

// modifyState used in fast computation mode to modify the statevector to try and preseve entanglement
void modifyState(std::vector<std::complex<double>> newValues, SVPair loc1, SVPair loc2) {
 int loc1Index = searchIndex(loc1);
 int loc2Index = searchIndex(loc2);
 if (loc1Index == -1 || loc2Index == -1) {
  return;
 }
 if (newValues.size() != 4) {
  return;
 }
 calculateNewVals(0, 0, newValues, loc1Index, loc2Index);
 calculateNewVals(0, 1, newValues, loc1Index, loc2Index);
 calculateNewVals(1, 0, newValues, loc1Index, loc2Index);
 calculateNewVals(1, 1, newValues, loc1Index, loc2Index);
}

// directModify allows statevector computation mode to modify the entire statevector
void directModify(int index, std::complex<double> value) {
 if (index >= state_.size()) {
  return;
 }
 state_[index] = value;
}
void directModify(std::vector<std::complex<double>> values) {
 if (values.size() != state_.size()) {
  return;
 }
 state_ = values;
}

// quickRefresh used in fast computation mode to recalculate the statevector values.
void quickRefresh() {
 int n = positions_.size();
 int dimStateVec = std::pow(2, n);
 for (int i = 0; i < dimStateVec; i++) {
  std::complex<double> start = 1;
  for (int j = 0; j < n; j++) {
   int element = inverseTail(n, j, i);
   SVPair resolvedPair = positions_[j];
   Qubit* qubit = qubitMap_->find(resolvedPair.name_)->second[resolvedPair.location_];
   start = start * *(qubit->fetch(element));
  }
  state_[i] = start;
 }
```

```
  }

  int getVal(int positionInStateVector, SVPair pair) {
    int position = searchIndex(pair);
    return inverseTail(positions_.size(), position, positionInStateVector);
  }

  // reorder allows us to reorder the statevector and returns this temporary vector. This
  // vector is used in compuitation and the reconciled with original tensor product later on.
  StateVector* reorder(std::vector<SVPair> newOrder) {
    reordered_->tensorProduct(newOrder, positions_, state_);
    return reordered_;
  }

  // reconcile accepts the temporary statevector and reorders it into the orginal order
  // and modifies the appropriate order.
  void reconcile(StateVector* reordered) {
    int n = positions_.size();
    int dimStateVec = std::pow(2, n);
    state_.resize(dimStateVec);

    for (int i = 0; i < dimStateVec; i++) {
      std::vector<int> newSchemeVals;
      for (int j = 0; j < n; j++) {
        newSchemeVals.push_back(inverseTail(n, j, i));
      }
      std::vector<int> oldSchemeVals = mapToOldScheme(newSchemeVals, positions_, reordered->getOrder());
      state_[i] = reordered->getState()[resolvePosition(oldSchemeVals)];
    }
  }

  int getN() {
    return positions_.size();
  }

  std::vector<SVPair> getOrder() {
    return positions_;
  }

  std::complex<double> getSVValue(int i) {
    return state_[i];
  }

 ~StateVector() {
    if (!isReorder) {
      delete reordered_;
    }
  }
};
```

Listing B.4: BaseTypes.h: Defines all common datatypes used throughout the codebase

```
    #pragma once
#include <vector>
```

```
#include <string>
#include <complex>
#include <map>
#include <iostream>

/*
 BaseTypes.h
 Description: Defines all common datatypes used throughout the codebase

 Defined Classes:
 idLocationPairs
 SVPair
 expEval
 doubleOrArg
 gateDeclaration
 gateOp
 DeviceType
 HeaderData
 RegisterType
 QuantumRegister
 ClassicalRegister
 Register
 GateRequestType
 GateRequest
 ConcurrentBlock
 Qubit
 Gate
 Calculation
 MeasureCommand
 StateVector
*/


struct idLocationPairs {
 // idLocationPairs is a datastructure used extensively in the parsing stage of Valkyrie, used to relate
 // which exact qubit(s) a particular gate is supposed to be operating on.
 std::vector<std::string> identifiers;
 std::vector<int> locations;
 int getSize() {
  return identifiers.size();
 }
};

// SVPair similar ot idLocationPairs but to be used exclusively in statevector manipulation. The SVPair
// datastructure is used as a key when navigating the various combination of states in the Statevector.
struct SVPair {
 std::string name_;
 int location_;
 SVPair(std::string name, int location) {
  name_ = name;
  location_ = location;
 }
 bool areEqual(SVPair comp) {
  return comp.location_ == location_ && comp.name_ == name_;
```

```
 }
};

// expEval used in the parsing of custom gate definitions, allows the program to distinguish between a
// variable or constant being used as input in a gate.
struct expEval {
 std::string ident;
 double value;
 bool identNotVal;
};

// doubleOrArg conversion of an expEval into an expression of which position in a variable list to get
// a parameter or the constant value of the parameter provided in gate definition.
struct doubleOrArg {
 bool doubleNotArg;
 double valD;
 int position;
};

// gateDeclaration datastructure to be used to carry custom gate declaration header after parsing.
struct gateDeclaration {
 std::string gateName;
 std::vector<std::string> idLocList;
 std::vector<std::string> paramList;
};

// gateOp datastructure to hold representation of a gate operation in a custom gate, uses the
// expVal datastructrure to differentiate between constant parameters and required parameters
struct gateOp {
 std::string gateName;
 std::vector<expEval> params;
 std::vector<std::string> idLocs;
};

// DeviceType enumeration of what kind of device we are using to run quantum simulations.
enum DeviceType {
 CPU_,
 GPU_,
 INVALID
};

// HeaderData datastructure to hold header data provided in the OPENQASM standard.
class HeaderData {
private:
 double openQASMStandard_ = 0.0;
 std::vector<std::string> includeFiles_;
public:
 HeaderData(std::string value, std::vector<std::string> includes) {
  openQASMStandard_ = std::stod(value);
  includeFiles_ = includes;
```

```
 }
};

// RegisterType enumeration for what kind of register is being instantiated.
enum RegisterType {
 quantum_,
 classical_,
 invalid_
};

// QuantumRegister datastructure to hold a parsed representation of a quantum register.
// This datastructure holds no qubits, but will be used by the staging module to
// generate qubit construction instructions.
class QuantumRegister {
private:
 std::string identifier_;
 int width_;
public:
 QuantumRegister(std::string identifier, int width) {
  identifier_ = identifier;
  width_ = width;
 }
 std::string getIdentifier() {
  return identifier_;
 }
 int getWidth() {
  return width_;
 }
 bool isQubit() {
  return width_ == 1;
 }
 QuantumRegister() = default;
};

// QuantumRegister datastructure to hold a parsed representation of a classical register.
// This datastructure does hold classical bits via the int representation.
class ClassicalRegister {
private:
 std::string identifier_;
 int width_;
 std::vector<int> values_;
public:
 ClassicalRegister(std::string identifier, int width) {
  identifier_ = identifier;
  width_ = width;
  for (int i = 0; i < width; i++) {
   values_.push_back(0);
  }
 }
 std::string getIdentifier() {
  return identifier_;
 }
 int getWidth() {
```

```
  return width_;
 }
 void setValue(int i, int val) {
  values_[i] = val;
 }
 int getValue(int i) {
  return values_[i];
 }
 ClassicalRegister() = default;
};

// Register is a wrapper for Quantum and Classical registers, allowing the staging module
// to differentiate between the two.
class Register {
private:
 RegisterType regType_;
 QuantumRegister qReg_;
 ClassicalRegister cReg_;
public:
 Register(RegisterType type, QuantumRegister qreg) {
  regType_ = type;
  qReg_ = qreg;
 }
 Register(RegisterType type, ClassicalRegister creg) {
  regType_ = type;
  cReg_ = creg;
 }

 QuantumRegister getQuantumRegister() {
  return qReg_;
 }

 ClassicalRegister getClassicalRegister() {
  return cReg_;
 }

 void setClassicalRegister(ClassicalRegister cReg) {
  cReg_ = cReg;
 }

 std::string getName() {
  if (regType_ == quantum_) {
   return qReg_.getIdentifier();
  }
  else {
   return cReg_.getIdentifier();
  }
 }

 bool isQuantum() {
  return regType_ == quantum_;
 }
};

// GateRequestType defines an enumeration for the primitive gate types U and CX as well
// as all qeLib1 gates. Allowing for efficient compilation
```

```cpp
enum GateRequestType {
 I,
 U,
 CX,
 h,
 cx,
 u3,
 u2,
 u1,
 id,
 u0,
 u,
 p,
 x,
 y,
 z,
 s,
 sdg,
 t,
 tdg,
 rx,
 ry,
 rz,
 sx,
 sxdg,
 cz,
 cy,
 swap,
 ch,
 ccx,
 cswap,
 crx,
 cry,
 crz,
 cu1,
 cp,
 cu3,
 csx,
 cu,
 rxx,
 rzz,
 rccx,
 rc3x,
 c3x,
 c3sqrtx,
 c4x,
 CUSTOM
};

// GateRequest is a datastructure to represent a user commanded gate operation, will be used by
// computation device to generate the gate matrices itself
class GateRequest {
private:
 GateRequestType gateType_;
 std::vector<std::string> registerIdentifiers_;
 std::vector<int> locations_;
 std::vector<double> parameters_;
```

```

public:
 GateRequest(){}
 GateRequest(GateRequestType type) {
  gateType_ = type;
 }
 void addressQubit(std::string registerID, int location) {
  registerIdentifiers_.push_back(registerID);
  locations_.push_back(location);
 }
 void setParameters(std::vector<double> params) {
  parameters_ = params;
 }
 void addParameter(double value) {
  parameters_.push_back(value);
 }
 int getGateDim() {
  return registerIdentifiers_.size();
 }
 std::vector<std::string> getRegisters() {
  return registerIdentifiers_;
 }
 std::vector<int> getLocations() {
  return locations_;
 }
 GateRequestType getGateType() {
  return gateType_;
 }
 std::vector<double> getParameters() {
  return parameters_;
 }
};

// ConcurrentBlock represent a block of gates which cna be processed in parallel without affecting
// accuracy of the computation. Used by staging module to send gates to Device
class ConcurrentBlock {
private:
 int count_ = 0;
 std::vector<GateRequest> gates_;
public:
 ConcurrentBlock(int count) {
 }
 void addGate(GateRequest newGate) {
  gates_.push_back(newGate);
  count_++;
 }
 void setCount(int count) {
  count_ = count;
 }
 int getCount() {
  return count_;
 }
 std::vector<GateRequest> getGates() {
  return gates_;
 }
};
```

```

// Qubit is the basic Qubit representation which is used to initially store qubit values. If
// Valkyrie is in fast computation mode then Qubit's are used exclusively to store the idividual states
class Qubit {
private:
 std::complex<double>* s_0;
 std::complex<double>* s_1;
public:
 Qubit(std::complex<double>* s0, std::complex<double>* s1) {
  s_0 = s0;
  s_1 = s1;
 }

 std::complex<double>* fetch(int i) {
  if (i == 0) {
   return s_0;
  }
  else {
   return s_1;
  }
 }
};

// Gate provides a basic gate representation for the computation device to perform matrix operations.
// If Valkyrie is in fast computation mode then the Gate matrix is directly used for computation.
class Gate {
private:
 std::vector<std::vector<std::complex<double>>> gateArray_;
 int m_; // dimensions
 int n_;
public:
 Gate(int m, int n, std::vector<std::vector<std::complex<double>>> gateArray) {
  m_ = m;
  n_ = n;
  gateArray_ = gateArray;
 }
 std::complex<double> fetchValue(int x, int y) {
  return gateArray_[x][y];
 }
 std::vector<std::vector<std::complex<double>>> getArray() {
  return gateArray_;
 }
 int getM() {
  return m_;
 }
 int getN() {
  return n_;
 }

};

// Calculation is a class which is used by both fast and statevector computation modes
```

```
// It holds the primitive gate and qubit values or state vector locations that are used
// by the matrix processing modules.
class Calculation {
private:
 Gate* gate_;
 std::vector<Qubit*> qubitValues_;
 std::vector<SVPair> locations_;

 // getNewOrder1 under the tensor product reordering procedure, this function
 // is able to shuffle the qubit that this Calculation is concerning right to the end
 // of the tensor product stack
 std::vector<SVPair> getNewOrder1(std::vector<SVPair> oldOrder) {
  std::vector<SVPair> newOrder;
  for (int i = 0; i < oldOrder.size(); i++) {
   if (!locations_[0].areEqual(oldOrder[i])) {
    newOrder.push_back(oldOrder[i]);
   }
  }
  newOrder.push_back(locations_[0]);
  return newOrder;
 }

 // getNewOrder2 under the tensor product reordering procedure, this function
 // is able to shuffle the two qubits that this Calculation is concerning right to the end
 // of the tensor product stack
 std::vector<SVPair> getNewOrder2(std::vector<SVPair> oldOrder) {
  std::vector<SVPair> newOrder;
  for (int i = 0; i < oldOrder.size(); i++) {
   if (!locations_[0].areEqual(oldOrder[i]) && !locations_[1].areEqual(oldOrder[i])) {
    newOrder.push_back(oldOrder[i]);
   }
  }
  newOrder.push_back(locations_[0]);
  newOrder.push_back(locations_[1]);
  return newOrder;
 }
public:
 Calculation(Gate* gate, std::vector<Qubit*> qubitVals, std::vector<SVPair> locations) {
  gate_ = gate;
  qubitValues_ = qubitVals;
  locations_ = locations;
 }
 Gate* getGate() {
  return gate_;
 }
 Qubit* getQubit(int i) {
  return qubitValues_[i];
 }

 std::vector<SVPair> getLocations() {
  return locations_;
```

```
 }
 // getNewOrder is important for the statevector computation mode. When
    ↪  performing the
 // matrix multiplication we are using a specialised tensor product (
    ↪ for efficiency) which
 // relies on the two concerned qubits to be pushed to the back of the
    ↪ tensor product stack.
 std::vector<SVPair> getNewOrder(std::vector<SVPair> oldOrder) {
  if (locations_.size() != 2 && locations_.size() != 1) {
   return oldOrder;
  }
  if (locations_.size() == 2) {
   return getNewOrder2(oldOrder);
  }
  return getNewOrder1(oldOrder);
 }

 std::vector<Qubit*> getQubits() {
  return qubitValues_;
 }
};

// MeasureCommand provides a simple datastructure to track measurement
   ↪ commands from
// the user during parsing.
class MeasureCommand {
private:
 idLocationPairs from_;
 idLocationPairs to_;
public:
 MeasureCommand(idLocationPairs from, idLocationPairs to) {
  from_ = from;
  to_ = to;
 }

 idLocationPairs getFrom() {
  return from_;
 }

 idLocationPairs getTo() {
  return to_;
 }
};

// StateVector is a core component of the quantum computation stack.
// In Fast compute mode, Statevector is used to store the overall
   ↪ results of the computation
// In Statevector computer mode, the Statevector is used both in the
   ↪ input and output of the computation
class StateVector {
private:
 // positions_ stores the current locations of different Quantum
    ↪ register and position pairs (each defining a qubit)
 // these locations are relevant to the order in which these qubits
    ↪ were multiplied in the tensor product used
 // to generate the StateVector
 std::vector<SVPair> positions_;
 // state_ is the statevector in full tensorproduct form, for a system
```

```
which uses n qubits the state_variable
// will be 2^n long
std::vector<std::complex<double>> state_;
// For Fast computation mode, the qubitMap_ provides access to the actual qubit values stored in the Qubit datastructure
std::map<std::string, std::vector<Qubit*>>* qubitMap_;
// reordered_ is a temporary state vector used during computation to represent the temporary reordering
// of the state vector for the tail computation
StateVector* reordered_;
bool isReorder = false;

// inverseTail provides the inverse of the tail function, this allows us to calculate (given the position of
// the qubit in the tensor product and the location in the statevector) which component of the qubit state (0th or 1th component)
// we need to process on.
int inverseTail(int nTotal, int indexInPositions, int locationInStateVec) {
 int j = std::pow(2, (nTotal - indexInPositions));
 if ((locationInStateVec % j) < (j / 2)) {
  return 0;
 }
 else {
  return 1;
 }
}

// tail provides a function (using inverseTail) to calculate whether we need the 0th or 1th component
// of a particular qubit.
bool tail(int nTotal, int indexInPositions, int locationInStateVec, int index) {
 return inverseTail(nTotal, indexInPositions, locationInStateVec) == index;
}

// used in Fast computation mode to calculate which values in a statevector is affected by a particular
// calculation result.
std::vector<int> affectedValues(int loc1, int index1, int loc2, int index2) {
 std::vector<int> affected;
 int n = positions_.size();
 for (int i = 0; i < state_.size(); i++) {
  if (tail(n, loc1, i, index1) && tail(n, loc2, i, index2)) {
   affected.push_back(i);
  }
 }
 return affected;
}

// calculateNewVals is used in fast computation mode for keeping track of qubit values changing compensating
// in the state vector.
void calculateNewVals(int pos1, int pos2, std::vector<std::complex<double>> newValues, int loc1Index, int loc2Index) {
 int pos = pos1 * 2 + pos2;
```

```
 std::complex<double> newVal = newValues[pos];
 std::vector<int> affected = affectedValues(loc1Index, pos1, loc2Index
    ↪ , pos2);
 int n = positions_.size();
 for (int position : affected) {
  std::complex < double> val = newVal;
  for (int i = 0; i < positions_.size(); i++) {
   if (i != loc1Index && i != loc2Index) {
    int tailedPos = inverseTail(n, i, position);
    std::complex<double> value = *(qubitMap_->find(positions_[i].name_
       ↪ )->second[positions_[i].location_]->fetch(tailedPos));
    val = val * value;
   }
  }
  state_[position] = val;
 }
}

int searchIndex(SVPair val) {
 for (int i = 0; i < positions_.size(); i++) {
  SVPair res = positions_[i];
  if (res.name_ == val.name_ && res.location_ == val.location_) {
   return i;
  }
 }
 return -1;
}

int searchIndex(SVPair val, std::vector<SVPair> positions) {
 for (int i = 0; i < positions.size(); i++) {
  SVPair res = positions[i];
  if (res.name_ == val.name_ && res.location_ == val.location_) {
   return i;
  }
 }
 return -1;
}

std::vector<int> mapToOldScheme(std::vector<int> values, std::vector<
   ↪ SVPair> newScheme, std::vector<SVPair> oldScheme) {
 std::vector<int> oldValues;
 int n = values.size();
 oldValues.resize(n);
 for (int i = 0; i < n; i++) {
  oldValues[i] = values[searchIndex(oldScheme[i], newScheme)];
 }
 return oldValues;
}

// resolvePosition calculates which position in the statevector is
   ↪ addressed by the values given
int resolvePosition(std::vector<int> values) {
 int n = values.size();
 int position = 0;
 for (int i = 0; i < n; i++) {
  int j = n - i;
  int val = values[i] * std::pow(2, j - 1);
  position += val;
```

```
  }
  return position;
 }

 std::vector<int> buildMap(std::vector<SVPair> newOrder, std::vector<SVPair> oldOrder) {
  std::vector<int> mapper;
  mapper.resize(newOrder.size());
  for (int i = 0; i < newOrder.size(); i++) {
   mapper[i] = searchIndex(newOrder[i], oldOrder);
  }
  return mapper;
 }

 std::vector<int> getOldSchemeValues(std::vector<int> mapper, std::vector<int> newSchemeVals) {
  std::vector<int> oldVals;
  int n = newSchemeVals.size();
  oldVals.resize(n);
  for (int i = 0; i < n; i++) {
   oldVals[mapper[i]] = newSchemeVals[i];
  }
  return oldVals;
 }

public:
 StateVector() {};
 StateVector(std::map<std::string, std::vector<Qubit*>>* linkToQubits) {
  qubitMap_ = linkToQubits;
  initialiseReorder();
 }

 std::vector<std::complex<double>> getState() {
  return state_;
 }

 void initialiseReorder() {
  reordered_ = new StateVector();
  reordered_->setReorder(true);
 }

 void setReorder(bool reorder) {
  isReorder = reorder;
 }

 // Used in initialisation of StateVector, tensorProduct produces the default statevector which is populated by
 // calculation and returned at the end.
 void tensorProduct() {
  for (std::map<std::string, std::vector<Qubit*> >::iterator it = qubitMap_->begin(); it != qubitMap_->end(); ++it) {
   for (int i = 0; i < it->second.size(); i++) {
    SVPair pair(it->first, i);
    positions_.push_back(pair);
   }
  }
  int n = positions_.size();
```

```
 int dimStateVec = std::pow(2, n);
 state_.resize(dimStateVec);
 for (int i = 0; i < dimStateVec; i++) {
  std::complex<double> start = 1;
  for (int j = 0; j < n; j++) {
   int element = inverseTail(n, j, i);
   SVPair resolvedPair = positions_[j];
   Qubit* qubit = qubitMap_->find(resolvedPair.name_)->second[resolvedPair.location_];
   start = start * *(qubit->fetch(element));
  }
  state_[i] = start;
 }
}

// Will only be called during reordering, provides same function as standed tensorProduct function
void tensorProduct(std::vector<SVPair> newOrder, std::vector<SVPair> oldOrder, std::vector<std::complex<double>> oldState) {
 positions_ = newOrder;
 int n = positions_.size();
 int dimStateVec = std::pow(2, n);
 state_.resize(dimStateVec);

 std::vector<int> mapper = buildMap(newOrder, oldOrder);

 for (int i = 0; i < dimStateVec; i++) {
  std::vector<int> newSchemeVals;
  for (int j = 0; j < n; j++) {
   newSchemeVals.push_back(inverseTail(n, j, i));
  }
  std::vector<int> oldSchemeVals = getOldSchemeValues(mapper, newSchemeVals);
  state_[i] = oldState[resolvePosition(oldSchemeVals)];
 }

}

// modifyState used in fast computation mode to modify the statevector to try and preseve entanglement
void modifyState(std::vector<std::complex<double>> newValues, SVPair loc1, SVPair loc2) {
 int loc1Index = searchIndex(loc1);
 int loc2Index = searchIndex(loc2);
 if (loc1Index == -1 || loc2Index == -1) {
  return;
 }
 if (newValues.size() != 4) {
  return;
 }
 calculateNewVals(0, 0, newValues, loc1Index, loc2Index);
 calculateNewVals(0, 1, newValues, loc1Index, loc2Index);
 calculateNewVals(1, 0, newValues, loc1Index, loc2Index);
 calculateNewVals(1, 1, newValues, loc1Index, loc2Index);
}

// directModify allows statevector computation mode to modify the entire statevector
```

```
void directModify(int index, std::complex<double> value) {
 if (index >= state_.size()) {
  return;
 }
 state_[index] = value;
}
void directModify(std::vector<std::complex<double>> values) {
 if (values.size() != state_.size()) {
  return;
 }
 state_ = values;
}

// quickRefresh used in fast computation mode to recalculate the
   ↪ statevector values.
void quickRefresh() {
 int n = positions_.size();
 int dimStateVec = std::pow(2, n);
 for (int i = 0; i < dimStateVec; i++) {
  std::complex<double> start = 1;
  for (int j = 0; j < n; j++) {
   int element = inverseTail(n, j, i);
   SVPair resolvedPair = positions_[j];
   Qubit* qubit = qubitMap_->find(resolvedPair.name_)->second[
      ↪ resolvedPair.location_];
   start = start * *(qubit->fetch(element));
  }
  state_[i] = start;
 }
}

int getVal(int positionInStateVector, SVPair pair) {
 int position = searchIndex(pair);
 return inverseTail(positions_.size(), position, positionInStateVector
    ↪ );
}

// reorder allows us to reorder the statevector and returns this
   ↪ temporary vector. This
// vector is used in compuitation and the reconciled with original
   ↪ tensor product later on.
StateVector* reorder(std::vector<SVPair> newOrder) {
 reordered_->tensorProduct(newOrder, positions_, state_);
 return reordered_;
}

// reconcile accepts the temporary statevector and reorders it into
   ↪ the orginal order
// and modifies the appropriate order.
void reconcile(StateVector* reordered) {
 int n = positions_.size();
 int dimStateVec = std::pow(2, n);
 state_.resize(dimStateVec);

 std::vector<int> mapper = buildMap(positions_, reordered->getOrder())
    ↪ ;

 for (int i = 0; i < dimStateVec; i++) {
```

```
   std::vector<int> newSchemeVals;
   for (int j = 0; j < n; j++) {
    newSchemeVals.push_back(inverseTail(n, j, i));
   }
   std::vector<int> oldSchemeVals = getOldSchemeValues(mapper,
      ↪ newSchemeVals);
   state_[i] = reordered->getState()[resolvePosition(oldSchemeVals)];
  }
 }

 int getN() {
  return positions_.size();
 }

 std::vector<SVPair> getOrder() {
  return positions_;
 }

 std::complex<double> getSVValue(int i) {
  return state_[i];
 }

 ~StateVector() {
  if (!isReorder) {
   delete reordered_;
  }
 }
};
```

Listing B.5: BaseTypes.h: Defines an Optimised implementation of all common datatypes used throughout the codebase

```
#pragma once

#include "AbstractDevice.h"

/*
 CPUDevice.h
 Description: This header file defines the CPU implementation of an
    ↪ Abstract Device as
 presented in AbstractDevice.h.

 Defined Classes:
 CPUQubitFactory
 CPUGateFactory
 CPUQuantumCircuit
 CPUQuantumProcessor
 CPUDevice

*/

// CPUQubitFactory implements the interface for AbstractQubitFactory
// Allocates, tracks and de-allocates memory for QubitStates
class CPUQubitFactory : public AbstractQubitFactory {
private:
 DeviceType type_;
 std::vector<Qubit*> qubits_;
public:
 CPUQubitFactory(){
```

```
  type_ = CPU_;
 }
 Qubit* generateQubit();
 ~CPUQubitFactory();
};

// CPUGateFactory implements the interface for AbstractGateFactory
// Allocates, tracks and de-allocates memory for Gate values
class CPUGateFactory : public AbstractGateFactory {
private:
 DeviceType type_;
 std::vector<Gate*> gates_;
public:
 CPUGateFactory() {
  type_ = CPU_;
 }
 Gate* generateGate(GateRequest request);
 ~CPUGateFactory();
};

// CPUQuantumCircuit implements the interface for AbstractQuantumCircuit
// Compiles calculation commands into actual matrices ready for computation
class CPUQuantumCircuit : public AbstractQuantumCircuit {
private:
 DeviceType type_;
 bool done_;
 std::map<std::string, std::vector<Qubit*>> qubitMap_;
 std::vector<std::vector<Calculation>> calculations_;
 CPUGateFactory* gateFactory_;
 int calcCounter = 0;
 std::vector<SVPair> zipSVPairs(std::vector<std::string> names, std::vector<int> locs);
 StateVector* sv_;
public:
 CPUQuantumCircuit(CPUGateFactory* gateFactory) {
  gateFactory_ = gateFactory;
  type_ = CPU_;
  done_ = false;
 }
 void loadQubitMap(std::map<std::string, std::vector<Qubit*>> qubitMap);
 void loadBlock(ConcurrentBlock block);
 std::vector<Calculation> getNextCalculation();
 std::map<std::string, std::vector<Qubit*>> returnResults();
 StateVector* getStateVector();
 bool checkComplete();
 ~CPUQuantumCircuit() {
  delete sv_;
 }
};

// CPUQuantumProcessor implements the interface for AbstractQuantumProcessor
// performs matrix calculations using the loaded quantum circuit to fetch calculations
class CPUQuantumProcessor : public AbstractQuantumProcessor {
```

```
private:
 DeviceType type_;
 AbstractQuantumCircuit* circuit_;
 std::vector<std::vector<std::complex<double>>> getCXResult(int n);
 std::vector<std::vector<std::complex<double>>> getGenericUResult(Gate*
    ↪  gate, int n);
public:
 CPUQuantumProcessor() {
  type_ = CPU_;
 }
 void loadCircuit(AbstractQuantumCircuit* circuit);
 void calculate();
 void calculateWithStateVector();
 std::map<std::string, std::vector<Qubit*>> qubitMapfetchQubitValues();
};

// CPUDevice implements the Abstract device interface
// Collects all components required for CPU execution
class CPUDevice : public AbstractDevice {
private:
 DeviceType type_;
 std::map<std::string, std::vector<Qubit*>> registerMap;
 CPUQubitFactory* qubitFactory;
 CPUGateFactory* gateFactory;
 CPUQuantumCircuit* quantumCircuit;
 CPUQuantumProcessor* quantumProcessor;
public:
 CPUDevice() {
  type_ = CPU_;
  qubitFactory = new CPUQubitFactory();
  gateFactory = new CPUGateFactory();
  quantumCircuit = new CPUQuantumCircuit(gateFactory);
  quantumProcessor = new CPUQuantumProcessor();
 }
 void loadRegister(Register registerx);
 void transferQubitMap();
 void loadConcurrentBlock(ConcurrentBlock block);
 void runSimulation();
 void runSimulationSV();
 void run(std::vector<Register> registers, std::vector<ConcurrentBlock>
    ↪  blocks);
 void runSV(std::vector<Register> registers, std::vector<
    ↪ ConcurrentBlock> blocks);
 std::map<std::string, std::vector<Qubit*>> revealQuantumState();
 void prettyPrintQubitStates(std::map<std::string, std::vector<Qubit*>>
    ↪  qubits) {
  for (std::map<std::string, std::vector<Qubit*>>::iterator it = qubits
     ↪ .begin(); it != qubits.end(); ++it) {
   std::cout << "Register: " << it->first << std::endl;
   std::vector<Qubit*> regQubits = it->second;
   for (int i = 0; i < regQubits.size(); i++) {
    std::cout << "Location [" << i << "]: " << regQubits[i]->fetch(0)->
       ↪ real() << "+" << regQubits[i]->fetch(0)->imag() << "i" << "
       ↪ ||| " << regQubits[i]->fetch(1)->real() << "+" << regQubits[
       ↪ i]->fetch(1)->imag() << "i" << std::endl;
   }
  }
 }
```

```
 StateVector* getStateVector() {
  return quantumCircuit->getStateVector();
 }
 ~CPUDevice() {
  delete qubitFactory;
  delete gateFactory;
  delete quantumCircuit;
  delete quantumProcessor;
 }
};
```

Listing B.6: CPUDevice.h: This header file defines the CPU implementation of an Abstract Device as presented in AbstractDevice.h.

```
#pragma once
#include "CPUDevice.h"
#include <cmath>
#include "GateUtilitiesCPU.h"

using namespace std::complex_literals;
const double ROOT2INV = 1.0 / std::pow(2, 0.5);

/*
 CPUDevice.cpp
 Description: This file defines the implementation of the functions
    ↪ defined
 in CPUDevice.h

 Defined Classes:
 CPUQubitFactory
 CPUGateFactory
 CPUQuantumCircuit
 CPUQuantumProcessor
 CPUDevice

*/

// getGateMatrix gneerates basic primitive gates (U, CX)
// uses buildU3GateCPU to construct the parameterised U gate.
std::vector<std::vector<std::complex<double>>> getGateMatrix(
   ↪ GateRequest gate) {
 GateRequestType gateType = gate.getGateType();
 switch (gateType) {
 case I:
  return std::vector<std::vector<std::complex<double>>> { {1, 0}, {0,
     ↪ 1} };
  break;
 case h:
  return std::vector<std::vector<std::complex<double>>> { {ROOT2INV,
     ↪ ROOT2INV}, {ROOT2INV, -1.0 * ROOT2INV} };
  break;
 case cx:
  return std::vector<std::vector<std::complex<double>>> { {1, 0, 0, 0},
     ↪ { 0, 1, 0, 0 }, { 0, 0, 0, 1 }, { 0, 0, 1, 0 } };
  break;
 case U:
  return buildU3GateCPU(gate);
  break;
 case CX:
```

```
    return std::vector<std::vector<std::complex<double>>> { {1, 0, 0, 0},
      ↪ { 0, 1, 0, 0 }, { 0, 0, 0, 1 }, { 0, 0, 1, 0 } };
    break;
  }
}

// generateQubit allocates heap memory for complex number and loads it
    ↪ into
// a heap memory allocated Qubit and tracks the generated qubits
Qubit* CPUQubitFactory::generateQubit()
{
  // Allocate heap memory for Qubit values
  std::complex<double>* s0 = new std::complex<double>;
  std::complex<double>* s1 = new std::complex<double>;
  *s0 = 1.0;
  *s1 = 0.0;
  // Allocate heap memory for Qubit and store values
  Qubit* generatedQubit = new Qubit(s0, s1);
  // Push into qubit tracker for deletion
  qubits_.push_back(generatedQubit);

  return generatedQubit;
}

// deconstructor cleans up any heap memory allocation
CPUQubitFactory::~CPUQubitFactory()
{
  for (auto qubit : qubits_) {
    delete qubit->fetch(0);
    delete qubit->fetch(1);
    delete qubit;
  }
}

// generateQubit allocates heap memory for complex numbers and loads it
    ↪ into
// a heap memory allocated Gate and tracks the generated gates
Gate* CPUGateFactory::generateGate(GateRequest request)
{
  std::vector<std::vector<std::complex<double>>> gateMatrix =
      ↪ getGateMatrix(request);
  int gateM = gateMatrix.size();
  int gateN = gateMatrix[0].size();

  Gate* generatedGate = new Gate(gateM, gateN, gateMatrix);
  gates_.push_back(generatedGate);
  return generatedGate;
}

// deconstructor cleans up any heap memory allocation
CPUGateFactory::~CPUGateFactory()
{
  for (auto gate : gates_) {
    delete gate;
  }
}

// zipSVPairs zips together identifiers and locations to generate
```

```
   ↪ SVPairs which can be used in
// statevector lookup
std::vector<SVPair> CPUQuantumCircuit::zipSVPairs(std::vector<std::
   ↪ string> names, std::vector<int> locs)
{
  std::vector<SVPair> values;
  for (int i = 0; i < names.size(); i++) {
    values.push_back(SVPair(names[i], locs[i]));
  }
  return values;
}

void CPUQuantumCircuit::loadQubitMap(std::map<std::string, std::vector<
   ↪ Qubit*>> qubitMap)
{
  qubitMap_ = qubitMap;
  sv_ = new StateVector(&qubitMap_);
  sv_->tensorProduct();
}

// loadBlock takes a concurretn block from the Staging module and
   ↪ converts it into
// a series if operable Calculation datatypes
void CPUQuantumCircuit::loadBlock(ConcurrentBlock block)
{
  std::vector<GateRequest> gates = block.getGates();
  std::vector<Calculation> calcs;
  for (auto gate : gates) {
    std::vector<std::string> registers = gate.getRegisters();
    std::vector<int> locations = gate.getLocations();
    std::vector<Qubit*> qubitValues;
    for (int i = 0; i < registers.size(); i++) {
      qubitValues.push_back(qubitMap_[registers[i]][locations[i]]);
    }
    Gate* gateTrue = gateFactory_->generateGate(gate);
    std::vector<SVPair> svPairs = zipSVPairs(registers, locations);
    Calculation calc = Calculation(gateTrue, qubitValues, svPairs);
    calcs.push_back(calc);
  }
  calculations_.push_back(calcs);
}

// getNextCalculation is used during the processing, to queue up
   ↪ calculations and
// raises the done_ flag if computation is complete
std::vector<Calculation> CPUQuantumCircuit::getNextCalculation()
{
  if (calcCounter == calculations_.size() - 1) {
    done_ = true;
    return calculations_[calcCounter];
  }
  else {
    std::vector<Calculation> val = calculations_[calcCounter];
    calcCounter++;
    return val;
  }
}

```

```
// For fast computation
std::map<std::string, std::vector<Qubit*>> CPUQuantumCircuit::
    ↪ returnResults()
{
 return qubitMap_;
}

// For Statevector computation
StateVector* CPUQuantumCircuit::getStateVector()
{
 return sv_;
}

bool CPUQuantumCircuit::checkComplete()
{
 if (calculations_.size() == 0) {
  return true;
 }
 return done_;
}

// getCXResults generates an 2^n by 2^n matrix from the tensor product
    ↪ of I gates and a final CX gate
// returns this matrix for computation
std::vector<std::vector<std::complex<double>>> CPUQuantumProcessor::
    ↪ getCXResult(int n)
{
 // n is the number of qubits, we have to have n-2 I gates and then a
    ↪ CX gate at the end
 if (n < 2) {
  return std::vector<std::vector<std::complex<double>>>();
 }
 std::vector<std::vector<std::complex<double>>> output;
 // overall sidelength of resultant gate
 int dimOverall = std::pow(2, n);
 // number of I multiplications required
 int leftOver = n - 2;
 if (leftOver == 0) {
  output = { {1, 0, 0, 0}, {0, 1, 0, 0}, {0, 0, 0, 1}, {0, 0, 1, 0} };
  return output;
 }
 output.resize(dimOverall);
 for (int i = 0; i < dimOverall; i++) {
  std::vector<std::complex<double>> subVec;
  subVec.resize(dimOverall);
  output[i] = subVec;
 }
 // skinny calculation due to the CX being the last matrix in a series
    ↪ of I tensor products
 // using tail methodology
 for (int i = 0; i < std::pow(2, leftOver); i++) {
  output[4 * i][4 * i] = 1;
  output[4 * i + 1][4 * i + 1] = 1;
  output[4 * i + 2][4 * i + 3] = 1;
  output[4 * i + 3][4 * i + 2] = 1;
 }
 return output;
}
```

```

// getGenericUResult return tensor product of a series of I gates and
    ↪ finally the U gate we are applying
std::vector<std::vector<std::complex<double>>> CPUQuantumProcessor::
    ↪ getGenericUResult(Gate* gate, int n)
{
 // n is the number of qubits, we have to have n-2 I gates and then a
    ↪ CX gate at the end
 if (n < 1) {
  return std::vector<std::vector<std::complex<double>>>();
 }
 std::vector<std::vector<std::complex<double>>> output;
 // overall sidelength of resultant gate
 int dimOverall = std::pow(2, n);
 // number of I multiplications required
 int leftOver = n - 1;
 if (leftOver == 0) {
  output = gate->getArray();
  return output;
 }
 output.resize(dimOverall);
 for (int i = 0; i < dimOverall; i++) {
  std::vector<std::complex<double>> subVec;
  subVec.resize(dimOverall);
  output[i] = subVec;
 }
 // skinny calculation due to the CX being the last matrix in a series
    ↪ of I tensor products
 // using tail methodology
 for (int i = 0; i < std::pow(2, leftOver); i++) {
  output[2 * i][2 * i] = gate->fetchValue(0,0);
  output[2 * i][2 * i + 1] = gate->fetchValue(0, 1);
  output[2 * i + 1][2 * i] = gate->fetchValue(1, 0);
  output[2 * i + 1][2 * i + 1] = gate->fetchValue(1, 1);
 }
 return output;
}

void CPUQuantumProcessor::loadCircuit(AbstractQuantumCircuit* circuit)
{
 circuit_ = circuit;
}
// calculate method for isolated fast computation
void CPUQuantumProcessor::calculate()
{
 while (!circuit_->checkComplete()) { // check if there still
    ↪ calculations to complete
  std::vector<Calculation> calcBlock = circuit_->getNextCalculation();
     ↪ // fetch next calculation
  for (auto calc : calcBlock) {
   Gate* gate = calc.getGate();
   int m = gate->getM();  // resolve gate dimensions
   int n = gate->getN();
   int qubitN = m / 2;
   std::vector<std::complex<double>> qubitValsBefore_;
   std::vector<std::complex<double>> qubitValsAfter_;
   std::vector<Qubit*> qubits = calc.getQubits();
   if (m == 2) {
```

```
    qubitValsBefore_.push_back(*qubits[0]->fetch(0));
    qubitValsBefore_.push_back(*qubits[0]->fetch(1));
   }
   else {
    // Perform local tensor product
    qubitValsBefore_.push_back(*qubits[0]->fetch(0) * *qubits[1]->fetch(0));
    qubitValsBefore_.push_back(*qubits[0]->fetch(0) * *qubits[1]->fetch(1));
    qubitValsBefore_.push_back(*qubits[0]->fetch(1) * *qubits[1]->fetch(0));
    qubitValsBefore_.push_back(*qubits[0]->fetch(1) * *qubits[1]->fetch(1));
   }
   for (int i = 0; i < m; i++) {
    std::complex<double> val = 0;
    for (int j = 0; j < n; j++) {
     val += gate->fetchValue(i, j) * qubitValsBefore_[j];  // matrix multiplication
    }
    qubitValsAfter_.push_back(val);
   }
   if (m == 2) {
    Qubit* qubit = qubits[0];
    *qubit->fetch(0) = qubitValsAfter_[0];
    *qubit->fetch(1) = qubitValsAfter_[1];
    circuit_->getStateVector()->quickRefresh();      // Since qubit vals are already update, we can just quickly refresh the statevector
   }
   else {
    Qubit* qubit1 = qubits[0];
    Qubit* qubit2 = qubits[1];
    *qubit1->fetch(0) = qubitValsAfter_[0] + qubitValsAfter_[1];
    *qubit1->fetch(1) = qubitValsAfter_[2] + qubitValsAfter_[3];
    *qubit2->fetch(0) = qubitValsAfter_[0] + qubitValsAfter_[2];
    *qubit2->fetch(1) = qubitValsAfter_[1] + qubitValsAfter_[3];
    circuit_->getStateVector()->modifyState(qubitValsAfter_, calc.getLocations()[0], calc.getLocations()[1]); // For entanglement relations we have to use the modifyState method
   }
  }
 }
}

// calculateWithStateVector for accurate Quantum Computer emulation, uses statevector in it's entirety
void CPUQuantumProcessor::calculateWithStateVector()
{
 while (!circuit_->checkComplete()) { // check if there are still calculations to consume
  std::vector<Calculation> calcBlock = circuit_->getNextCalculation(); // fetch calculation
  for (auto calc : calcBlock) {
   Gate* gate = calc.getGate();
   int m = gate->getM();
   int n = gate->getN();
   int qubitN = m / 2;
```

```
   StateVector* sv = circuit_->getStateVector();      // get current
      ↪ state vector
   std::vector<SVPair> newOrder = calc.getNewOrder(sv->getOrder()); //
      ↪ use the calculation function to work out the new order of the
      ↪  state vector for tail procedure
   StateVector* reordered = sv->reorder(newOrder);      // fetch
      ↪ temporary statevector using reordered tensor product
   std::vector<std::vector<std::complex<double>>> gateValues;
   if (m == 2) {
    gateValues = getGenericUResult(gate, sv->getN());     // Generate
       ↪ full gate matrix
   }
   if (m == 4) {
    gateValues = getCXResult(sv->getN());        // Generate full gate
       ↪ matrix
   }
   if (gateValues.size() == 0) {
    return;
   }
   std::vector<std::complex<double>> newValues;
   for (int i = 0; i < gateValues.size(); i++) {
    std::complex<double> acc = 0;
    for (int j = 0; j < gateValues.size(); j++) {
     acc += gateValues[i][j] * reordered->getSVValue(j);    // full
        ↪ state vector calculation
    }
    newValues.push_back(acc);
   }
   reordered->directModify(newValues);          // set newValues of
      ↪ reordered state vector
   sv->reconcile(reordered);           // reconcile temporary order for
      ↪  statevector for the original order
  }
 }
}

std::map<std::string, std::vector<Qubit*>> CPUQuantumProcessor::
   ↪ qubitMapfetchQubitValues()
{
 return circuit_->returnResults();
}

void CPUDevice::loadRegister(Register registerx)
{
 if (registerx.isQuantum()) {
  QuantumRegister qReg = registerx.getQuantumRegister();
  std::string regName = qReg.getIdentifier();
  int width = qReg.getWidth();
  std::vector<Qubit*> registerQubits;
  for (int i = 0; i < width; i++) {
   registerQubits.push_back(qubitFactory->generateQubit());
  }
  registerMap.insert(std::pair<std::string, std::vector<Qubit*>>(
     ↪ regName, registerQubits));
 }
}

void CPUDevice::transferQubitMap()
```

```
{
 quantumCircuit->loadQubitMap(registerMap);
}

void CPUDevice::loadConcurrentBlock(ConcurrentBlock block)
{
 quantumCircuit->loadBlock(block);
}

void CPUDevice::runSimulation()
{
 quantumProcessor->loadCircuit(quantumCircuit);
 quantumProcessor->calculate();
}

void CPUDevice::runSimulationSV()
{
 quantumProcessor->loadCircuit(quantumCircuit);
 quantumProcessor->calculateWithStateVector();
}

void CPUDevice::run(std::vector<Register> registers, std::vector<
    ↪ ConcurrentBlock> blocks)
{
 for (auto reg : registers) {
  loadRegister(reg);
 }
 transferQubitMap();
 for (auto block : blocks) {
  loadConcurrentBlock(block);
 }
 runSimulation();
}

void CPUDevice::runSV(std::vector<Register> registers, std::vector<
    ↪ ConcurrentBlock> blocks)
{
 for (auto reg : registers) {
  loadRegister(reg);
 }
 transferQubitMap();
 for (auto block : blocks) {
  loadConcurrentBlock(block);
 }
 runSimulationSV();
}

std::map<std::string, std::vector<Qubit*>> CPUDevice::
    ↪ revealQuantumState()
{
 return quantumProcessor->qubitMapfetchQubitValues();
}
```

Listing B.7: CPUDevice.cpp: This file defines the implementation of the functions defined in CPUDevice.h

```
#pragma once
#include "CPUDevice.h"
#include <cmath>
```

```
#include "GateUtilitiesCPU.h"
#include <chrono>

using namespace std::complex_literals;
const double ROOT2INV = 1.0 / std::pow(2, 0.5);

/*
 CPUDevice.cpp
 Description: This file defines the implementation of the functions
    ↪ defined
 in CPUDevice.h

 Defined Classes:
 CPUQubitFactory
 CPUGateFactory
 CPUQuantumCircuit
 CPUQuantumProcessor
 CPUDevice

*/

// getGateMatrix gneerates basic primitive gates (U, CX)
// uses buildU3GateCPU to construct the parameterised U gate.
std::vector<std::vector<std::complex<double>>> getGateMatrix(
    ↪ GateRequest gate) {
 GateRequestType gateType = gate.getGateType();
 switch (gateType) {
 case I:
  return std::vector<std::vector<std::complex<double>>> { {1, 0}, {0,
    ↪ 1} };
  break;
 case h:
  return std::vector<std::vector<std::complex<double>>> { {ROOT2INV,
    ↪ ROOT2INV}, {ROOT2INV, -1.0 * ROOT2INV} };
  break;
 case cx:
  return std::vector<std::vector<std::complex<double>>> { {1, 0, 0, 0},
    ↪ { 0, 1, 0, 0 }, { 0, 0, 0, 1 }, { 0, 0, 1, 0 } };
  break;
 case U:
  return buildU3GateCPU(gate);
  break;
 case CX:
  return std::vector<std::vector<std::complex<double>>> { {1, 0, 0, 0},
    ↪ { 0, 1, 0, 0 }, { 0, 0, 0, 1 }, { 0, 0, 1, 0 } };
  break;
 }
}

// generateQubit allocates heap memory for complex number and loads it
    ↪ into
// a heap memory allocated Qubit and tracks the generated qubits
Qubit* CPUQubitFactory::generateQubit()
{
 // Allocate heap memory for Qubit values
 std::complex<double>* s0 = new std::complex<double>;
 std::complex<double>* s1 = new std::complex<double>;
 *s0 = 1.0;
```

```
 *s1 = 0.0;
 // Allocate heap memory for Qubit and store values
 Qubit* generatedQubit = new Qubit(s0, s1);
 // Push into qubit tracker for deletion
 qubits_.push_back(generatedQubit);

 return generatedQubit;
}

// deconstructor cleans up any heap memory allocation
CPUQubitFactory::~CPUQubitFactory()
{
 for (auto qubit : qubits_) {
  delete qubit->fetch(0);
  delete qubit->fetch(1);
  delete qubit;
 }
}

// generateQubit allocates heap memory for complex numbers and loads it
    ↪ into
// a heap memory allocated Gate and tracks the generated gates
Gate* CPUGateFactory::generateGate(GateRequest request)
{
 std::vector<std::vector<std::complex<double>>> gateMatrix =
     ↪ getGateMatrix(request);
 int gateM = gateMatrix.size();
 int gateN = gateMatrix[0].size();

 Gate* generatedGate = new Gate(gateM, gateN, gateMatrix);
 gates_.push_back(generatedGate);
 return generatedGate;
}

// deconstructor cleans up any heap memory allocation
CPUGateFactory::~CPUGateFactory()
{
 for (auto gate : gates_) {
  delete gate;
 }
}

// zipSVPairs zips together identifiers and locations to generate
    ↪ SVPairs which can be used in
// statevector lookup
std::vector<SVPair> CPUQuantumCircuit::zipSVPairs(std::vector<std::
    ↪ string> names, std::vector<int> locs)
{
 std::vector<SVPair> values;
 for (int i = 0; i < names.size(); i++) {
  values.push_back(SVPair(names[i], locs[i]));
 }
 return values;
}

void CPUQuantumCircuit::loadQubitMap(std::map<std::string, std::vector<
    ↪ Qubit*>> qubitMap)
{
```

```
 qubitMap_ = qubitMap;
 sv_ = new StateVector(&qubitMap_);
 sv_->tensorProduct();
}

// loadBlock takes a concurretn block from the Staging module and
   ↪ converts it into
// a series if operable Calculation datatypes
void CPUQuantumCircuit::loadBlock(ConcurrentBlock block)
{
 std::vector<GateRequest> gates = block.getGates();
 std::vector<Calculation> calcs;
 for (auto gate : gates) {
  std::vector<std::string> registers = gate.getRegisters();
  std::vector<int> locations = gate.getLocations();
  std::vector<Qubit*> qubitValues;
  for (int i = 0; i < registers.size(); i++) {
   qubitValues.push_back(qubitMap_[registers[i]][locations[i]]);
  }
  Gate* gateTrue = gateFactory_->generateGate(gate);
  std::vector<SVPair> svPairs = zipSVPairs(registers, locations);
  Calculation calc = Calculation(gateTrue, qubitValues, svPairs);
  calcs.push_back(calc);
 }
 calculations_.push_back(calcs);
}

// getNextCalculation is used during the processing, to queue up
   ↪ calculations and
// raises the done_ flag if computation is complete
std::vector<Calculation> CPUQuantumCircuit::getNextCalculation()
{
 if (calcCounter == calculations_.size() - 1) {
  done_ = true;
  return calculations_[calcCounter];
 }
 else {
  std::vector<Calculation> val = calculations_[calcCounter];
  calcCounter++;
  return val;
 }
}

// For fast computation
std::map<std::string, std::vector<Qubit*>> CPUQuantumCircuit::
   ↪ returnResults()
{
 return qubitMap_;
}

// For Statevector computation
StateVector* CPUQuantumCircuit::getStateVector()
{
 return sv_;
}

bool CPUQuantumCircuit::checkComplete()
{
```

```
 if (calculations_.size() == 0) {
  return true;
 }
 return done_;
}


void CPUQuantumProcessor::loadCircuit(AbstractQuantumCircuit* circuit)
{
 circuit_ = circuit;
}

// calculate method for isolated fast computation
void CPUQuantumProcessor::calculate()
{
 while (!circuit_->checkComplete()) { // check if there still
    ↪ calculations to complete
  std::vector<Calculation> calcBlock = circuit_->getNextCalculation();
     ↪ // fetch next calculation
  for (auto calc : calcBlock) {
   Gate* gate = calc.getGate();
   int m = gate->getM();  // resolve gate dimensions
   int n = gate->getN();
   int qubitN = m / 2;
   std::vector<std::complex<double>> qubitValsBefore_;
   std::vector<std::complex<double>> qubitValsAfter_;
   std::vector<Qubit*> qubits = calc.getQubits();
   if (m == 2) {
    qubitValsBefore_.push_back(*qubits[0]->fetch(0));
    qubitValsBefore_.push_back(*qubits[0]->fetch(1));
   }
   else {
    // Perform local tensor product
    qubitValsBefore_.push_back(*qubits[0]->fetch(0) * *qubits[1]->fetch
       ↪ (0));
    qubitValsBefore_.push_back(*qubits[0]->fetch(0) * *qubits[1]->fetch
       ↪ (1));
    qubitValsBefore_.push_back(*qubits[0]->fetch(1) * *qubits[1]->fetch
       ↪ (0));
    qubitValsBefore_.push_back(*qubits[0]->fetch(1) * *qubits[1]->fetch
       ↪ (1));
   }
   for (int i = 0; i < m; i++) {
    std::complex<double> val = 0;
    for (int j = 0; j < n; j++) {
     val += gate->fetchValue(i, j) * qubitValsBefore_[j];  // matrix
        ↪ multiplication
    }
    qubitValsAfter_.push_back(val);
   }
   if (m == 2) {
    Qubit* qubit = qubits[0];
    *qubit->fetch(0) = qubitValsAfter_[0];
    *qubit->fetch(1) = qubitValsAfter_[1];
    circuit_->getStateVector()->quickRefresh();      // Since qubit
       ↪ vals are already update, we can just quickly refresh the
       ↪ statevector
   }
```

```
      else {
        Qubit* qubit1 = qubits[0];
        Qubit* qubit2 = qubits[1];
        *qubit1->fetch(0) = qubitValsAfter_[0] + qubitValsAfter_[1];
        *qubit1->fetch(1) = qubitValsAfter_[2] + qubitValsAfter_[3];
        *qubit2->fetch(0) = qubitValsAfter_[0] + qubitValsAfter_[2];
        *qubit2->fetch(1) = qubitValsAfter_[1] + qubitValsAfter_[3];
        circuit_->getStateVector()->modifyState(qubitValsAfter_, calc.getLocations()[0], calc.getLocations()[1]); // For entanglement relations we have to use the modifyState method
      }
    }
  }
}

// calculateWithStateVector for accurate Quantum Computer emulation, uses statevector in it's entirety
void CPUQuantumProcessor::calculateWithStateVector()
{
  long long counter = 0;
  while (!circuit_->checkComplete()) { // check if there are still calculations to consume
    std::vector<Calculation> calcBlock = circuit_->getNextCalculation(); // fetch calculation
    for (auto calc : calcBlock) {
      Gate* gate = calc.getGate();
      int m = gate->getM();
      int n = gate->getN();
      int qubitN = m / 2;
      StateVector* sv = circuit_->getStateVector();       // get current state vector
      std::vector<SVPair> newOrder = calc.getNewOrder(sv->getOrder()); // use the calculation function to work out the new order of the  state vector for tail procedure

      StateVector* reordered = sv->reorder(newOrder);       // fetch temporary statevector using reordered tensor product
      std::vector<std::vector<std::complex<double>>> gateValues = gate->getArray();
      int svLength = reordered->getState().size();
      std::vector<std::complex<double>> newValues;
      for (int i = 0; i < svLength; i++) {       // Only compute what is required
        int startIndex = m * (i / m);
        std::complex<double> acc = 0;
        for (int j = 0; j < m; j++) {
          acc += gateValues[(i % m)][j] * reordered->getSVValue(startIndex +  j);
        }
        newValues.push_back(acc);
      }

      reordered->directModify(newValues);          // set newValues of reordered state vector
      sv->reconcile(reordered);            // reconcile temporary order for  statevector for the original order
    }
  }
```

```cpp
}

std::map<std::string, std::vector<Qubit*>> CPUQuantumProcessor::qubitMapfetchQubitValues()
{
  return circuit_->returnResults();
}

void CPUDevice::loadRegister(Register registerx)
{
  if (registerx.isQuantum()) {
    QuantumRegister qReg = registerx.getQuantumRegister();
    std::string regName = qReg.getIdentifier();
    int width = qReg.getWidth();
    std::vector<Qubit*> registerQubits;
    for (int i = 0; i < width; i++) {
      registerQubits.push_back(qubitFactory->generateQubit());
    }
    registerMap.insert(std::pair<std::string, std::vector<Qubit*>>(regName, registerQubits));
  }
}

void CPUDevice::transferQubitMap()
{
  quantumCircuit->loadQubitMap(registerMap);
}

void CPUDevice::loadConcurrentBlock(ConcurrentBlock block)
{
  quantumCircuit->loadBlock(block);
}

void CPUDevice::runSimulation()
{
  quantumProcessor->loadCircuit(quantumCircuit);
  quantumProcessor->calculate();
}

void CPUDevice::runSimulationSV()
{
  quantumProcessor->loadCircuit(quantumCircuit);
  quantumProcessor->calculateWithStateVector();
}

void CPUDevice::run(std::vector<Register> registers, std::vector<ConcurrentBlock> blocks)
{
  for (auto reg : registers) {
    loadRegister(reg);
  }
  transferQubitMap();
  for (auto block : blocks) {
    loadConcurrentBlock(block);
  }
  runSimulation();
}

```

```cpp
void CPUDevice::runSV(std::vector<Register> registers, std::vector<ConcurrentBlock> blocks)
{
  for (auto reg : registers) {
    loadRegister(reg);
  }
  transferQubitMap();
  for (auto block : blocks) {
    loadConcurrentBlock(block);
  }
  runSimulationSV();
}

std::map<std::string, std::vector<Qubit*>> CPUDevice::revealQuantumState()
{
  return quantumProcessor->qubitMapfetchQubitValues();
}
```

Listing B.8: CPUDevice.cpp: This file defines the Optimised implementation of the functions defined in CPUDevice.h

```cpp
#pragma once

#include <iostream>
#include <exception>

/*
  Exceptions.h
  Description: Defines custom exceptions.

*/

// VersionExcpetion is thrown if the OpenQASM version isn't 2.0.
struct VersionException : public std::exception {
  const char* what() const throw () {
    return "OpenQASM version is invalid";
  }
};
```

Listing B.9: Exceptions.h: Defines custom exceptions.

```cpp
#pragma once
#include "BaseTypes.h"

/*
  GateUtilitiesCPU.h
  Description: defines utilities for gate matrix generation

*/

const std::complex<double> IMAGCPU(0,1);

// buildU3GateCPU uses trigonometric functions to generate U gate
std::vector<std::vector<std::complex<double>>> buildU3GateCPU(GateRequest gate) {
  double theta = gate.getParameters()[0];
  double phi = gate.getParameters()[1];
  double lambda_ = gate.getParameters()[2];
```

```
    double cosHalfTheta = std::cos(theta / 2);
    double sinHalfTheta = std::sin(theta / 2);
    std::complex<double> elem1 = cosHalfTheta;
    std::complex<double> elem2 = sinHalfTheta * -1 * std::exp(lambda_ *
        ↪ IMAGCPU);
    std::complex<double> elem3 = sinHalfTheta * std::exp(phi * IMAGCPU);
    std::complex<double> elem4 = cosHalfTheta * std::exp(lambda_ * IMAGCPU
        ↪ + phi * IMAGCPU);
    return std::vector<std::vector<std::complex<double>>> { {elem1, elem2
        ↪ }, { elem3, elem4 }};
}
```

Listing B.10: GateUtilitiesCPU.h: defines utilities for gate matrix generation.

```
#pragma once
#include "BaseTypes.h"

/*
 GateUtilitiesGPU.h
 Description: defines utilities for gate matrix generation

*/

const std::complex<double> IMAGGPU(0, 1);

// buildU3GateGPU uses trigonometric functions to generate U gate
std::vector<std::vector<std::complex<double>>> buildU3GateGPU(
    ↪ GateRequest gate) {
    double theta = gate.getParameters()[0];
    double phi = gate.getParameters()[1];
    double lambda_ = gate.getParameters()[2];
    double cosHalfTheta = std::cos(theta / 2);
    double sinHalfTheta = std::sin(theta / 2);
    std::complex<double> elem1 = cosHalfTheta;
    std::complex<double> elem2 = sinHalfTheta * -1 * std::exp(lambda_ *
        ↪ IMAGGPU);
    std::complex<double> elem3 = sinHalfTheta * std::exp(phi * IMAGGPU);
    std::complex<double> elem4 = cosHalfTheta * std::exp(lambda_ * IMAGGPU
        ↪ + phi * IMAGGPU);
    return std::vector<std::vector<std::complex<double>>> { {elem1, elem2
        ↪ }, { elem3, elem4 }};
}
```

Listing B.11: GateUtilitiesGPU.cuh: defines utilities for gate matrix generation.

```
#include "cuda_runtime.h"
#include <stdio.h>
#include "cuComplex.h"
#include "BaseTypes.h"

/*
 GPUCompute.cuh
 Description: Library of GPU specific functions needed to parallelise
    ↪ gate calculations

 Functions defined:
 matrixMul
 svMatrixMul
 svMatrixUltraMul
```

```
 svAddLargeScale
 convertQubitComplex
 convertComplexQubit
 tensorProduct
 calculateGPU2x2
 calculateGPU4x4
 calculateGPU
 calculateGPU2x2
 calculateGPU4x4
 calculateGPULargeSV
 calculateGPUSV
*/

namespace ValkGPULib {

 // matrixMul is GPU code for fast compute mode parallel row computation
 __global__ void matrixMul(cuDoubleComplex* output, const cuDoubleComplex* input, const cuDoubleComplex* gate, const int m) {
  int loc = threadIdx.x;
  output[loc] = make_cuDoubleComplex(0, 0);
  for (int i = 0; i < m; i++) {
   output[loc] = cuCadd(cuCmul(input[i], gate[m * loc + i]), output[loc]);
  }
 }

 // svMatrixMul is GPU code for statevector compute mode parallel row computation
 __global__ void svMatrixMul(cuDoubleComplex* output, const cuDoubleComplex* input, const cuDoubleComplex* gate, int m){
  int loc = blockIdx.x * blockDim.x + threadIdx.x;
  output[loc] = make_cuDoubleComplex(0, 0);
  for (int i = 0; i < m; i++) {
   output[loc] = cuCadd(cuCmul(input[i], gate[m * loc + i]), output[loc]);
  }
 }

 // svMatrixUltraMul is GPU code for massively parallel large scale computation
 __global__ void svMatrixUltraMul(cuDoubleComplex* output, const cuDoubleComplex* input, const cuDoubleComplex* gate, int m) {
  int loc = blockIdx.x * blockDim.x + threadIdx.x;
  output[loc] = cuCmul(input[loc % m], gate[loc]);
 }

 // svAddLargeScale provides the summation of the temporary calculations provided by the svMatrixUltraMul
 __global__ void svAddLargeScale(cuDoubleComplex* output, cuDoubleComplex* input, int m) {
  int loc = threadIdx.x;
  output[loc] = make_cuDoubleComplex(0, 0);
  for (int i = 0; i < m; i++) {
   output[loc] = cuCadd(output[loc], input[m * loc + i]);
  }
 }
```

```

// convertQubitComplex converts  representation C++ stlib complex
   ↪ number into CUDA Complex number representation
cuDoubleComplex convertQubitComplex(std::complex<double> input) {
 return make_cuDoubleComplex(input.real(), input.imag());
}

// convertQubitComplex converts CUDA Complex number representation
   ↪ into C++ stlib complex number representation
std::complex<double> convertComplexQubit(cuDoubleComplex input) {
 return std::complex<double>(input.x, input.y);
}

// tensorProduct calculates tensor product values for fast compute
   ↪ mode
cuDoubleComplex tensorProduct(std::vector<Qubit*> inputQubits, int i)
   ↪ {
 Qubit* qubit1 = inputQubits[0];
 Qubit* qubit2 = inputQubits[1];
 std::complex<double> result = *qubit1->fetch(i / 2) * *qubit2->fetch(
    ↪ i % 2);
 return make_cuDoubleComplex(result.real(), result.imag());
}

std::vector<std::complex<double>> calculateGPU2x2(cuDoubleComplex*
   ↪ beforeGate, cuDoubleComplex* gateValues, cuDoubleComplex*
   ↪ afterGate, Gate* gate, std::vector<Qubit*> qubits, int m, int n
   ↪ );

std::vector<std::complex<double>> calculateGPU4x4(cuDoubleComplex*
   ↪ beforeGate, cuDoubleComplex* gateValues, cuDoubleComplex*
   ↪ afterGate, Gate* gate, std::vector<Qubit*> qubits, int m, int n
   ↪ );

// calculateGPU is the overall calling function for fast compute mode
std::vector<std::complex<double>> calculateGPU(cuDoubleComplex*
   ↪ beforeGate, cuDoubleComplex* gateValues, cuDoubleComplex*
   ↪ afterGate, Gate* gate, std::vector<Qubit*> qubits) {
 int m = gate->getM();
 int n = gate->getN();
 int qubitN = m / 2;

 cudaError_t cudaStatus;

 // Generate Host side arrays for qubit values
 std::vector<std::complex<double>> results;
 if (m == 2) {
  results = calculateGPU2x2(beforeGate, gateValues, afterGate, gate,
     ↪ qubits, m, n);
 }
 else if (m == 4) {
  results = calculateGPU4x4(beforeGate, gateValues, afterGate, gate,
     ↪ qubits, m, n);
 }
 return results;
}

// calculateGPU2x2 allows for single qubit gate parallelisation for
```

```
   ↪ fast compute mode
std::vector<std::complex<double>> calculateGPU2x2(cuDoubleComplex*
   ↪ beforeGate, cuDoubleComplex* gateValues, cuDoubleComplex*
   ↪ afterGate, Gate* gate, std::vector<Qubit*> qubits, int m, int n
   ↪ ){
 const int arraySize = 2;
 const cuDoubleComplex before[arraySize] = { convertQubitComplex(*(
    ↪ qubits[0]->fetch(0))), convertQubitComplex(*qubits[0]->fetch
    ↪ (1)) };
 const cuDoubleComplex gateVal[4] = { convertQubitComplex(gate->
    ↪ fetchValue(0,0)), convertQubitComplex(gate->fetchValue(0,1)),
    ↪ convertQubitComplex(gate->fetchValue(1,0)),
    ↪ convertQubitComplex(gate->fetchValue(1,1)) };
 cuDoubleComplex after[arraySize] = { 0 };
 cudaError_t cudaStatus;
 std::vector<std::complex<double>> forStateVector;
 // Copy input vectors into CUDA memory
 cudaStatus = cudaMemcpy(beforeGate, before, m * sizeof(
    ↪ cuDoubleComplex), cudaMemcpyHostToDevice);
 if (cudaStatus != cudaSuccess) {
  fprintf(stderr, "cudaMemcpy failed!");
  goto Error;
 }
 cudaStatus = cudaMemcpy(gateValues, gateVal, (m * n) * sizeof(
    ↪ cuDoubleComplex), cudaMemcpyHostToDevice);
 if (cudaStatus != cudaSuccess) {
  fprintf(stderr, "cudaMemcpy failed!");
  goto Error;
 }
 cudaStatus = cudaMemcpy(afterGate, after, m * sizeof(cuDoubleComplex)
    ↪ , cudaMemcpyHostToDevice);
 if (cudaStatus != cudaSuccess) {
  fprintf(stderr, "cudaMemcpy failed!");
  goto Error;
 }

 // Run matrix calc kernel
 ValkGPULib::matrixMul << <1, 2 >> > (afterGate, beforeGate,
    ↪ gateValues, 2);

 // Check for any errors launching the kernel
 cudaStatus = cudaGetLastError();
 if (cudaStatus != cudaSuccess) {
  fprintf(stderr, "addKernel launch failed: %s\n", cudaGetErrorString(
     ↪ cudaStatus));
  goto Error;
 }

 // cudaDeviceSynchronize waits for the kernel to finish, and returns
 // any errors encountered during the launch.
 cudaStatus = cudaDeviceSynchronize();
 if (cudaStatus != cudaSuccess) {
  fprintf(stderr, "cudaDeviceSynchronize returned error code %d after
     ↪ launching addKernel!\n", cudaStatus);
  goto Error;
 }

 // Copy output vector from GPU buffer to host memory.
```

```
    cudaStatus = cudaMemcpy(after, afterGate, m * sizeof(cuDoubleComplex), cudaMemcpyDeviceToHost);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMemcpy failed!");
        goto Error;
    }
    Qubit* qubit = qubits[0];
    *qubit->fetch(0) = convertComplexQubit(after[0]);
    *qubit->fetch(1) = convertComplexQubit(after[1]);
    forStateVector = { convertComplexQubit(after[0]), convertComplexQubit(after[1]) };
    return forStateVector;
Error:
    return std::vector<std::complex<double>>();
}

// calculateGPU4x4 allows for double qubit gate parallelisation for fast compute mode
std::vector<std::complex<double>> calculateGPU4x4(cuDoubleComplex* beforeGate, cuDoubleComplex* gateValues, cuDoubleComplex* afterGate, Gate* gate, std::vector<Qubit*> qubits, int m, int n) {
    const int arraySize = 4;
    const cuDoubleComplex before[arraySize] = { tensorProduct(qubits, 0),  tensorProduct(qubits, 1) , tensorProduct(qubits, 2), tensorProduct(qubits, 3) };
    const cuDoubleComplex gateVal[16] = {
        convertQubitComplex(gate->fetchValue(0,0)), convertQubitComplex(gate->fetchValue(0,1)), convertQubitComplex(gate->fetchValue(0,2)), convertQubitComplex(gate->fetchValue(0,3)),
        convertQubitComplex(gate->fetchValue(1,0)), convertQubitComplex(gate->fetchValue(1,1)), convertQubitComplex(gate->fetchValue(1,2)), convertQubitComplex(gate->fetchValue(1,3)),
        convertQubitComplex(gate->fetchValue(2,0)), convertQubitComplex(gate->fetchValue(2,1)), convertQubitComplex(gate->fetchValue(2,2)), convertQubitComplex(gate->fetchValue(2,3)),
        convertQubitComplex(gate->fetchValue(3,0)), convertQubitComplex(gate->fetchValue(3,1)), convertQubitComplex(gate->fetchValue(3,2)), convertQubitComplex(gate->fetchValue(3,3)),
    };
    cuDoubleComplex after[arraySize] = { 0 };
    cudaError_t cudaStatus;
    // Copy input vectors into CUDA memory
    cudaStatus = cudaMemcpy(beforeGate, before, m * sizeof(cuDoubleComplex), cudaMemcpyHostToDevice);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMemcpy failed!");
        return std::vector<std::complex<double>>();
    }
    cudaStatus = cudaMemcpy(gateValues, gateVal, (m * n) * sizeof(cuDoubleComplex), cudaMemcpyHostToDevice);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMemcpy failed!");
        return std::vector<std::complex<double>>();
    }
    cudaStatus = cudaMemcpy(afterGate, after, m * sizeof(cuDoubleComplex), cudaMemcpyHostToDevice);
    if (cudaStatus != cudaSuccess) {
```

```
   fprintf(stderr, "cudaMemcpy failed!");
   return std::vector<std::complex<double>>();
  }

  // Run matrix calc kernel
  ValkGPULib::matrixMul << <1, 4 >> > (afterGate, beforeGate,
      ↪ gateValues, 4);

  // Check for any errors launching the kernel
  cudaStatus = cudaGetLastError();
  if (cudaStatus != cudaSuccess) {
   fprintf(stderr, "addKernel launch failed: %s\n", cudaGetErrorString(
       ↪ cudaStatus));
   return std::vector<std::complex<double>>();
  }

  // cudaDeviceSynchronize waits for the kernel to finish, and returns
  // any errors encountered during the launch.
  cudaStatus = cudaDeviceSynchronize();
  if (cudaStatus != cudaSuccess) {
   fprintf(stderr, "cudaDeviceSynchronize returned error code %d after
       ↪ launching addKernel!\n", cudaStatus);
   return std::vector<std::complex<double>>();
  }

  // Copy output vector from GPU buffer to host memory.
  cudaStatus = cudaMemcpy(after, afterGate, m * sizeof(cuDoubleComplex)
      ↪ , cudaMemcpyDeviceToHost);
  if (cudaStatus != cudaSuccess) {
   fprintf(stderr, "cudaMemcpy failed!");
   return std::vector<std::complex<double>>();
  }
  Qubit* qubit = qubits[0];
  *qubit->fetch(0) = convertComplexQubit(after[0]) +
      ↪ convertComplexQubit(after[1]);
  *qubit->fetch(1) = convertComplexQubit(after[2]) +
      ↪ convertComplexQubit(after[3]);
  qubit = qubits[1];
  *qubit->fetch(0) = convertComplexQubit(after[0]) +
      ↪ convertComplexQubit(after[2]);
  *qubit->fetch(1) = convertComplexQubit(after[1]) +
      ↪ convertComplexQubit(after[3]);
  std::vector<std::complex<double>> forStateVector = {
      ↪ convertComplexQubit(after[0]), convertComplexQubit(after[1]),
      ↪ convertComplexQubit(after[2]), convertComplexQubit(after[3])
      ↪ };
  return forStateVector;
 }

 /// State Vector compute mode ///

 // calculateGPULargeSV allows for large size statevector -- gate
     ↪ multiplication to be entirely parallelised
 std::vector<std::complex<double>> calculateGPULargeSV(cuDoubleComplex*
     ↪  beforeGate, cuDoubleComplex* gateValues, cuDoubleComplex*
     ↪ afterGate, StateVector* reordered, std::vector<std::vector<std
     ↪ ::complex<double>>> gateValuesV) {
  int arraySize = gateValuesV.size();
```

```
cuDoubleComplex* before = new cuDoubleComplex[arraySize];
cuDoubleComplex* gateVal = new cuDoubleComplex[arraySize * arraySize
    ↪ ];
cuDoubleComplex* after = new cuDoubleComplex[arraySize];

std::vector<std::complex<double>> currentState = reordered->getState
    ↪ ();
for (int i = 0; i < currentState.size(); i++) {
  before[i] = convertQubitComplex(currentState[i]);
}
for (int i = 0; i < arraySize; i++) {
  for (int j = 0; j < arraySize; j++) {
    gateVal[i * arraySize + j] = convertQubitComplex(gateValuesV[i][j])
        ↪ ;
  }
}
// Copy input vectors into CUDA memory
cudaError_t cudaStatus;
cudaStatus = cudaMemcpy(beforeGate, before, arraySize * sizeof(
    ↪ cuDoubleComplex), cudaMemcpyHostToDevice);
if (cudaStatus != cudaSuccess) {
  fprintf(stderr, "cudaMemcpy failed!");
  return std::vector<std::complex<double>>();
}
cudaStatus = cudaMemcpy(gateValues, gateVal, (arraySize * arraySize)
    ↪ * sizeof(cuDoubleComplex), cudaMemcpyHostToDevice);
if (cudaStatus != cudaSuccess) {
  fprintf(stderr, "cudaMemcpy failed!");
  return std::vector<std::complex<double>>();
}
cudaStatus = cudaMemcpy(afterGate, after, arraySize * sizeof(
    ↪ cuDoubleComplex), cudaMemcpyHostToDevice);
if (cudaStatus != cudaSuccess) {
  fprintf(stderr, "cudaMemcpy failed!");
  return std::vector<std::complex<double>>();
}
cuDoubleComplex* tempOutput;
cudaStatus = cudaMalloc((void**)&tempOutput, arraySize*arraySize*
    ↪ sizeof(cuDoubleComplex));
if (cudaStatus != cudaSuccess) {
  fprintf(stderr, "cudaMalloc failed!");
  return std::vector<std::complex<double>>();
}
int blockSize = 256;
int numBlocks = (arraySize*arraySize + blockSize - 1) / blockSize;
ValkGPULib::svMatrixUltraMul << <numBlocks, blockSize >> > (
    ↪ tempOutput, beforeGate, gateValues, arraySize);
// Check for any errors launching the kernel
cudaStatus = cudaGetLastError();
if (cudaStatus != cudaSuccess) {
  fprintf(stderr, "addKernel launch failed: %s\n", cudaGetErrorString(
      ↪ cudaStatus));
  return std::vector<std::complex<double>>();
}

// cudaDeviceSynchronize waits for the kernel to finish, and returns
// any errors encountered during the launch.
cudaStatus = cudaDeviceSynchronize();
```

```
 if (cudaStatus != cudaSuccess) {
  fprintf(stderr, "cudaDeviceSynchronize returned error code %d after
     ↪ launching addKernel!\n", cudaStatus);
  return std::vector<std::complex<double>>();
 }
 ValkGPULib::svAddLargeScale << <1, arraySize >> > (afterGate,
    ↪ tempOutput, arraySize);
 // Check for any errors launching the kernel
 cudaStatus = cudaGetLastError();
 if (cudaStatus != cudaSuccess) {
  fprintf(stderr, "addKernel launch failed: %s\n", cudaGetErrorString(
     ↪ cudaStatus));
  return std::vector<std::complex<double>>();
 }

 // cudaDeviceSynchronize waits for the kernel to finish, and returns
 // any errors encountered during the launch.
 cudaStatus = cudaDeviceSynchronize();
 if (cudaStatus != cudaSuccess) {
  fprintf(stderr, "cudaDeviceSynchronize returned error code %d after
     ↪ launching addKernel!\n", cudaStatus);
  return std::vector<std::complex<double>>();
 }

 // Copy output vector from GPU buffer to host memory.
 cudaStatus = cudaMemcpy(after, afterGate, arraySize * sizeof(
    ↪ cuDoubleComplex), cudaMemcpyDeviceToHost);
 if (cudaStatus != cudaSuccess) {
  fprintf(stderr, "cudaMemcpy failed!");
  return std::vector<std::complex<double>>();
 }
 cudaFree(tempOutput);
 std::vector<std::complex<double>> output;
 for (int i = 0; i < arraySize; i++) {
  output.push_back(convertComplexQubit(after[i]));
 }
 return output;
}

// calculateGPUSV allows for small to medium statevectors to be easily
   ↪ parallelised
std::vector<std::complex<double>> calculateGPUSV(cuDoubleComplex*
   ↪ beforeGate, cuDoubleComplex* gateValues, cuDoubleComplex*
   ↪ afterGate, StateVector* reordered, std::vector<std::vector<std
   ↪ ::complex<double>>> gateValuesV) {
 int arraySize = gateValuesV.size();
 cuDoubleComplex* before = new cuDoubleComplex[arraySize];
 cuDoubleComplex* gateVal = new cuDoubleComplex[arraySize * arraySize
    ↪ ];
 cuDoubleComplex* after = new cuDoubleComplex[arraySize];

 std::vector<std::complex<double>> currentState = reordered->getState
    ↪ ();
 for (int i = 0; i < currentState.size(); i++) {
  before[i] = convertQubitComplex(currentState[i]);
 }
 for (int i = 0; i < arraySize; i++) {
  for (int j = 0; j < arraySize; j++) {
```

```
      gateVal[i * arraySize + j] = convertQubitComplex(gateValuesV[i][j]);
    }
  }
  // Copy input vectors into CUDA memory
  cudaError_t cudaStatus;
  cudaStatus = cudaMemcpy(beforeGate, before, arraySize * sizeof(cuDoubleComplex), cudaMemcpyHostToDevice);
  if (cudaStatus != cudaSuccess) {
   fprintf(stderr, "cudaMemcpy failed!");
   return std::vector<std::complex<double>>();
  }
  cudaStatus = cudaMemcpy(gateValues, gateVal, (arraySize * arraySize) * sizeof(cuDoubleComplex), cudaMemcpyHostToDevice);
  if (cudaStatus != cudaSuccess) {
   fprintf(stderr, "cudaMemcpy failed!");
   return std::vector<std::complex<double>>();
  }
  cudaStatus = cudaMemcpy(afterGate, after, arraySize * sizeof(cuDoubleComplex), cudaMemcpyHostToDevice);
  if (cudaStatus != cudaSuccess) {
   fprintf(stderr, "cudaMemcpy failed!");
   return std::vector<std::complex<double>>();
  }
  if (arraySize > 256) {
   int blockSize = 256;
   int numBlocks = (arraySize + blockSize - 1) / blockSize;
   ValkGPULib::svMatrixMul << <numBlocks, blockSize >> > (afterGate, beforeGate, gateValues, arraySize);
  }
  else {
   ValkGPULib::svMatrixMul << <1, arraySize >> > (afterGate, beforeGate, gateValues, arraySize);
  }
  // Check for any errors launching the kernel
  cudaStatus = cudaGetLastError();
  if (cudaStatus != cudaSuccess) {
   fprintf(stderr, "addKernel launch failed: %s\n", cudaGetErrorString(cudaStatus));
   return std::vector<std::complex<double>>();
  }

  // cudaDeviceSynchronize waits for the kernel to finish, and returns
  // any errors encountered during the launch agrim.
  cudaStatus = cudaDeviceSynchronize();
  if (cudaStatus != cudaSuccess) {
   fprintf(stderr, "cudaDeviceSynchronize returned error code %d after launching addKernel!\n", cudaStatus);
   return std::vector<std::complex<double>>();
  }

  // Copy output vector from GPU buffer to host memory.
  cudaStatus = cudaMemcpy(after, afterGate, arraySize * sizeof(cuDoubleComplex), cudaMemcpyDeviceToHost);
  if (cudaStatus != cudaSuccess) {
   fprintf(stderr, "cudaMemcpy failed!");
   return std::vector<std::complex<double>>();
  }
```

```
  std::vector<std::complex<double>> output;
  for (int i = 0; i < arraySize; i++) {
   output.push_back(convertComplexQubit(after[i]));
  }
  return output;
 }

}
```

Listing B.12: GPUCompute.cuh: Library of GPU specific functions needed to parallelise gate calculations.

```
#include "cuda_runtime.h"
#include <stdio.h>
#include "cuComplex.h"
#include "BaseTypes.h"

/*
 GPUCompute.cuh
 Description: Library of GPU specific functions needed to parallelise gate calculations

 Functions defined:
 matrixMul
 svMatrixMul
 svMatrixUltraMul
 svAddLargeScale
 convertQubitComplex
 convertComplexQubit
 tensorProduct
 calculateGPU2x2
 calculateGPU4x4
 calculateGPU
 calculateGPU2x2
 calculateGPU4x4
 calculateGPUSVPrime
 DEPRECATED:
  calculateGPULargeSV
  calculateGPUSV
*/

namespace ValkGPULib {

 // matrixMul is GPU code for fast compute mode parallel row computation
 __global__ void matrixMul(cuDoubleComplex* output, const cuDoubleComplex* input, const cuDoubleComplex* gate, const int m) {
  int loc = threadIdx.x;
  output[loc] = make_cuDoubleComplex(0, 0);
  for (int i = 0; i < m; i++) {
   output[loc] = cuCadd(cuCmul(input[i], gate[m * loc + i]), output[loc]);
  }
 }

 // svMatrixMul is GPU code for statevector compute mode parallel row computation
 __global__ void svMatrixMul(cuDoubleComplex* output, const
```

```
    ↪ cuDoubleComplex* input, const cuDoubleComplex* gate, int m){
 int loc = blockIdx.x * blockDim.x + threadIdx.x;
 output[loc] = make_cuDoubleComplex(0, 0);
 for (int i = 0; i < m; i++) {
  output[loc] = cuCadd(cuCmul(input[i], gate[m * loc + i]), output[loc
    ↪ ]);
 }
}

__global__ void svMatrixMulPrime(cuDoubleComplex* output, const
    ↪ cuDoubleComplex* input, const cuDoubleComplex* gate, int m, int
    ↪  m_prim) {
 int loc = blockIdx.x * blockDim.x + threadIdx.x;
 output[loc] = make_cuDoubleComplex(0, 0);
 int startIndex = m_prim * (loc / m_prim);
 for (int i = 0; i < m_prim; i++) {
  output[loc] = cuCadd(cuCmul(input[startIndex + i], gate[(loc %
    ↪ m_prim) * m_prim  + i]), output[loc]);
 }
}

// svMatrixUltraMul is GPU code for massively parallel large scale
    ↪ computation
__global__ void svMatrixUltraMul(cuDoubleComplex* output, const
    ↪ cuDoubleComplex* input, const cuDoubleComplex* gate, int m) {
 int loc = blockIdx.x * blockDim.x + threadIdx.x;
 output[loc] = cuCmul(input[loc % m], gate[loc]);
}

// svAddLargeScale provides the summation of the temporary
    ↪ calculations provided by the svMatrixUltraMul
__global__ void svAddLargeScale(cuDoubleComplex* output,
    ↪ cuDoubleComplex* input, int m) {
 int loc = threadIdx.x;
 output[loc] = make_cuDoubleComplex(0, 0);
 for (int i = 0; i < m; i++) {
  output[loc] = cuCadd(output[loc], input[m * loc + i]);
 }
}

// convertQubitComplex converts  representation C++ stlib complex
    ↪ number into CUDA Complex number representation
cuDoubleComplex convertQubitComplex(std::complex<double> input) {
 return make_cuDoubleComplex(input.real(), input.imag());
}

// convertQubitComplex converts CUDA Complex number representation
    ↪ into C++ stlib complex number representation
std::complex<double> convertComplexQubit(cuDoubleComplex input) {
 return std::complex<double>(input.x, input.y);
}

// tensorProduct calculates tensor product values for fast compute
    ↪ mode
cuDoubleComplex tensorProduct(std::vector<Qubit*> inputQubits, int i)
    ↪ {
 Qubit* qubit1 = inputQubits[0];
 Qubit* qubit2 = inputQubits[1];
```

```
  std::complex<double> result = *qubit1->fetch(i / 2) * *qubit2->fetch(i % 2);
  return make_cuDoubleComplex(result.real(), result.imag());
}

std::vector<std::complex<double>> calculateGPU2x2(cuDoubleComplex* beforeGate, cuDoubleComplex* gateValues, cuDoubleComplex* afterGate, Gate* gate, std::vector<Qubit*> qubits, int m, int n);

std::vector<std::complex<double>> calculateGPU4x4(cuDoubleComplex* beforeGate, cuDoubleComplex* gateValues, cuDoubleComplex* afterGate, Gate* gate, std::vector<Qubit*> qubits, int m, int n);

// calculateGPU is the overall calling function for fast compute mode
std::vector<std::complex<double>> calculateGPU(cuDoubleComplex* beforeGate, cuDoubleComplex* gateValues, cuDoubleComplex* afterGate, Gate* gate, std::vector<Qubit*> qubits) {
  int m = gate->getM();
  int n = gate->getN();
  int qubitN = m / 2;

  cudaError_t cudaStatus;

  // Generate Host side arrays for qubit values
  std::vector<std::complex<double>> results;
  if (m == 2) {
    results = calculateGPU2x2(beforeGate, gateValues, afterGate, gate, qubits, m, n);
  }
  else if (m == 4) {
    results = calculateGPU4x4(beforeGate, gateValues, afterGate, gate, qubits, m, n);
  }
  return results;
}

// calculateGPU2x2 allows for single qubit gate parallelisation for fast compute mode
std::vector<std::complex<double>> calculateGPU2x2(cuDoubleComplex* beforeGate, cuDoubleComplex* gateValues, cuDoubleComplex* afterGate, Gate* gate, std::vector<Qubit*> qubits, int m, int n){
  const int arraySize = 2;
  const cuDoubleComplex before[arraySize] = { convertQubitComplex(*(qubits[0]->fetch(0))), convertQubitComplex(*qubits[0]->fetch(1)) };
  const cuDoubleComplex gateVal[4] = { convertQubitComplex(gate->fetchValue(0,0)), convertQubitComplex(gate->fetchValue(0,1)), convertQubitComplex(gate->fetchValue(1,0)), convertQubitComplex(gate->fetchValue(1,1)) };
  cuDoubleComplex after[arraySize] = { 0 };
  cudaError_t cudaStatus;
  std::vector<std::complex<double>> forStateVector;
  // Copy input vectors into CUDA memory
  cudaStatus = cudaMemcpy(beforeGate, before, m * sizeof(cuDoubleComplex), cudaMemcpyHostToDevice);
```

```
 if (cudaStatus != cudaSuccess) {
  fprintf(stderr, "cudaMemcpy failed!");
  goto Error;
 }
 cudaStatus = cudaMemcpy(gateValues, gateVal, (m * n) * sizeof(
    ↪ cuDoubleComplex), cudaMemcpyHostToDevice);
 if (cudaStatus != cudaSuccess) {
  fprintf(stderr, "cudaMemcpy failed!");
  goto Error;
 }
 cudaStatus = cudaMemcpy(afterGate, after, m * sizeof(cuDoubleComplex)
    ↪ , cudaMemcpyHostToDevice);
 if (cudaStatus != cudaSuccess) {
  fprintf(stderr, "cudaMemcpy failed!");
  goto Error;
 }

 // Run matrix calc kernel
 ValkGPULib::matrixMul << <1, 2 >> > (afterGate, beforeGate,
    ↪ gateValues, 2);

 // Check for any errors launching the kernel
 cudaStatus = cudaGetLastError();
 if (cudaStatus != cudaSuccess) {
  fprintf(stderr, "addKernel launch failed: %s\n", cudaGetErrorString(
     ↪ cudaStatus));
  goto Error;
 }

 // cudaDeviceSynchronize waits for the kernel to finish, and returns
 // any errors encountered during the launch.
 cudaStatus = cudaDeviceSynchronize();
 if (cudaStatus != cudaSuccess) {
  fprintf(stderr, "cudaDeviceSynchronize returned error code %d after
     ↪ launching addKernel!\n", cudaStatus);
  goto Error;
 }

 // Copy output vector from GPU buffer to host memory.
 cudaStatus = cudaMemcpy(after, afterGate, m * sizeof(cuDoubleComplex)
    ↪ , cudaMemcpyDeviceToHost);
 if (cudaStatus != cudaSuccess) {
  fprintf(stderr, "cudaMemcpy failed!");
  goto Error;
 }
 Qubit* qubit = qubits[0];
 *qubit->fetch(0) = convertComplexQubit(after[0]);
 *qubit->fetch(1) = convertComplexQubit(after[1]);
 forStateVector = { convertComplexQubit(after[0]), convertComplexQubit
    ↪ (after[1]) };
 return forStateVector;
Error:
 return std::vector<std::complex<double>>();
}

// calculateGPU4x4 allows for double qubit gate parallelisation for
   ↪ fast compute mode
std::vector<std::complex<double>> calculateGPU4x4(cuDoubleComplex*
```

```
    ↪ beforeGate, cuDoubleComplex* gateValues, cuDoubleComplex*
    ↪ afterGate, Gate* gate, std::vector<Qubit*> qubits, int m, int n
    ↪ ) {
  const int arraySize = 4;
  const cuDoubleComplex before[arraySize] = { tensorProduct(qubits, 0),
     ↪  tensorProduct(qubits, 1) , tensorProduct(qubits, 2),
     ↪ tensorProduct(qubits, 3) };
  const cuDoubleComplex gateVal[16] = {
   convertQubitComplex(gate->fetchValue(0,0)), convertQubitComplex(gate
      ↪ ->fetchValue(0,1)), convertQubitComplex(gate->fetchValue(0,2)
      ↪ ), convertQubitComplex(gate->fetchValue(0,3)),
   convertQubitComplex(gate->fetchValue(1,0)), convertQubitComplex(gate
      ↪ ->fetchValue(1,1)), convertQubitComplex(gate->fetchValue(1,2)
      ↪ ), convertQubitComplex(gate->fetchValue(1,3)),
   convertQubitComplex(gate->fetchValue(2,0)), convertQubitComplex(gate
      ↪ ->fetchValue(2,1)), convertQubitComplex(gate->fetchValue(2,2)
      ↪ ), convertQubitComplex(gate->fetchValue(2,3)),
   convertQubitComplex(gate->fetchValue(3,0)), convertQubitComplex(gate
      ↪ ->fetchValue(3,1)), convertQubitComplex(gate->fetchValue(3,2)
      ↪ ), convertQubitComplex(gate->fetchValue(3,3)),
  };
  cuDoubleComplex after[arraySize] = { 0 };
  cudaError_t cudaStatus;
  // Copy input vectors into CUDA memory
  cudaStatus = cudaMemcpy(beforeGate, before, m * sizeof(
     ↪ cuDoubleComplex), cudaMemcpyHostToDevice);
  if (cudaStatus != cudaSuccess) {
   fprintf(stderr, "cudaMemcpy failed!");
   return std::vector<std::complex<double>>();
  }
  cudaStatus = cudaMemcpy(gateValues, gateVal, (m * n) * sizeof(
     ↪ cuDoubleComplex), cudaMemcpyHostToDevice);
  if (cudaStatus != cudaSuccess) {
   fprintf(stderr, "cudaMemcpy failed!");
   return std::vector<std::complex<double>>();
  }
  cudaStatus = cudaMemcpy(afterGate, after, m * sizeof(cuDoubleComplex)
     ↪ , cudaMemcpyHostToDevice);
  if (cudaStatus != cudaSuccess) {
   fprintf(stderr, "cudaMemcpy failed!");
   return std::vector<std::complex<double>>();
  }

  // Run matrix calc kernel
  ValkGPULib::matrixMul << <1, 4 >> > (afterGate, beforeGate,
     ↪ gateValues, 4);

  // Check for any errors launching the kernel
  cudaStatus = cudaGetLastError();
  if (cudaStatus != cudaSuccess) {
   fprintf(stderr, "addKernel launch failed: %s\n", cudaGetErrorString(
      ↪ cudaStatus));
   return std::vector<std::complex<double>>();
  }

  // cudaDeviceSynchronize waits for the kernel to finish, and returns
  // any errors encountered during the launch.
  cudaStatus = cudaDeviceSynchronize();
```

```
  if (cudaStatus != cudaSuccess) {
   fprintf(stderr, "cudaDeviceSynchronize returned error code %d after
      ↪ launching addKernel!\n", cudaStatus);
   return std::vector<std::complex<double>>();
  }

  // Copy output vector from GPU buffer to host memory.
  cudaStatus = cudaMemcpy(after, afterGate, m * sizeof(cuDoubleComplex)
     ↪ , cudaMemcpyDeviceToHost);
  if (cudaStatus != cudaSuccess) {
   fprintf(stderr, "cudaMemcpy failed!");
   return std::vector<std::complex<double>>();
  }
  Qubit* qubit = qubits[0];
  *qubit->fetch(0) = convertComplexQubit(after[0]) +
     ↪ convertComplexQubit(after[1]);
  *qubit->fetch(1) = convertComplexQubit(after[2]) +
     ↪ convertComplexQubit(after[3]);
  qubit = qubits[1];
  *qubit->fetch(0) = convertComplexQubit(after[0]) +
     ↪ convertComplexQubit(after[2]);
  *qubit->fetch(1) = convertComplexQubit(after[1]) +
     ↪ convertComplexQubit(after[3]);
  std::vector<std::complex<double>> forStateVector = {
     ↪ convertComplexQubit(after[0]), convertComplexQubit(after[1]),
     ↪ convertComplexQubit(after[2]), convertComplexQubit(after[3])
     ↪ };
  return forStateVector;
 }

 /// State Vector compute mode ///

 // calculateGPULargeSV allows for large size statevector -- gate
    ↪ multiplication to be entirely parallelised
 std::vector<std::complex<double>> calculateGPULargeSV(cuDoubleComplex*
    ↪  beforeGate, cuDoubleComplex* gateValues, cuDoubleComplex*
    ↪ afterGate, StateVector* reordered, std::vector<std::vector<std
    ↪ ::complex<double>>> gateValuesV) {
  int arraySize = gateValuesV.size();
  cuDoubleComplex* before = new cuDoubleComplex[arraySize];
  cuDoubleComplex* gateVal = new cuDoubleComplex[arraySize * arraySize
     ↪ ];
  cuDoubleComplex* after = new cuDoubleComplex[arraySize];

  std::vector<std::complex<double>> currentState = reordered->getState
     ↪ ();
  for (int i = 0; i < currentState.size(); i++) {
   before[i] = convertQubitComplex(currentState[i]);
  }
  for (int i = 0; i < arraySize; i++) {
   for (int j = 0; j < arraySize; j++) {
    gateVal[i * arraySize + j] = convertQubitComplex(gateValuesV[i][j])
       ↪ ;
   }
  }
  // Copy input vectors into CUDA memory
  cudaError_t cudaStatus;
  cudaStatus = cudaMemcpy(beforeGate, before, arraySize * sizeof(
```

```
        cuDoubleComplex), cudaMemcpyHostToDevice);
    if (cudaStatus != cudaSuccess) {
     fprintf(stderr, "cudaMemcpy failed!");
     delete before;
     delete gateVal;
     delete after;
     return std::vector<std::complex<double>>();
    }
    cudaStatus = cudaMemcpy(gateValues, gateVal, (arraySize * arraySize) * sizeof(cuDoubleComplex), cudaMemcpyHostToDevice);
    if (cudaStatus != cudaSuccess) {
     fprintf(stderr, "cudaMemcpy failed!");
     delete before;
     delete gateVal;
     delete after;
     return std::vector<std::complex<double>>();
    }
    cudaStatus = cudaMemcpy(afterGate, after, arraySize * sizeof(cuDoubleComplex), cudaMemcpyHostToDevice);
    if (cudaStatus != cudaSuccess) {
     fprintf(stderr, "cudaMemcpy failed!");
     delete before;
     delete gateVal;
     delete after;
     return std::vector<std::complex<double>>();
    }
    cuDoubleComplex* tempOutput;
    cudaStatus = cudaMalloc((void**)&tempOutput, arraySize*arraySize*sizeof(cuDoubleComplex));
    if (cudaStatus != cudaSuccess) {
     fprintf(stderr, "cudaMalloc failed!");
     delete before;
     delete gateVal;
     delete after;
     return std::vector<std::complex<double>>();
    }
    int blockSize = 256;
    int numBlocks = (arraySize*arraySize + blockSize - 1) / blockSize;
    ValkGPULib::svMatrixUltraMul << <numBlocks, blockSize >> > (tempOutput, beforeGate, gateValues, arraySize);
    // Check for any errors launching the kernel
    cudaStatus = cudaGetLastError();
    if (cudaStatus != cudaSuccess) {
     fprintf(stderr, "addKernel launch failed: %s\n", cudaGetErrorString(cudaStatus));
     delete before;
     delete gateVal;
     delete after;
     return std::vector<std::complex<double>>();
    }

    // cudaDeviceSynchronize waits for the kernel to finish, and returns
    // any errors encountered during the launch.
    cudaStatus = cudaDeviceSynchronize();
    if (cudaStatus != cudaSuccess) {
     fprintf(stderr, "cudaDeviceSynchronize returned error code %d after launching addKernel!\n", cudaStatus);
     delete before;
```

```
  delete gateVal;
  delete after;
  return std::vector<std::complex<double>>();
 }
 ValkGPULib::svAddLargeScale << <1, arraySize >> > (afterGate,
   ↪ tempOutput, arraySize);
 // Check for any errors launching the kernel
 cudaStatus = cudaGetLastError();
 if (cudaStatus != cudaSuccess) {
  fprintf(stderr, "addKernel launch failed: %s\n", cudaGetErrorString(
    ↪ cudaStatus));
  delete before;
  delete gateVal;
  delete after;
  return std::vector<std::complex<double>>();
 }

 // cudaDeviceSynchronize waits for the kernel to finish, and returns
 // any errors encountered during the launch.
 cudaStatus = cudaDeviceSynchronize();
 if (cudaStatus != cudaSuccess) {
  fprintf(stderr, "cudaDeviceSynchronize returned error code %d after
    ↪ launching addKernel!\n", cudaStatus);
  delete before;
  delete gateVal;
  delete after;
  return std::vector<std::complex<double>>();
 }

 // Copy output vector from GPU buffer to host memory.
 cudaStatus = cudaMemcpy(after, afterGate, arraySize * sizeof(
   ↪ cuDoubleComplex), cudaMemcpyDeviceToHost);
 if (cudaStatus != cudaSuccess) {
  fprintf(stderr, "cudaMemcpy failed!");
  delete before;
  delete gateVal;
  delete after;
  return std::vector<std::complex<double>>();
 }
 cudaFree(tempOutput);
 std::vector<std::complex<double>> output;
 for (int i = 0; i < arraySize; i++) {
  output.push_back(convertComplexQubit(after[i]));
 }
 delete before;
 delete gateVal;
 delete after;
 return output;
}

// calculateGPUSV allows for small to medium statevectors to be easily
   ↪ parallelised
std::vector<std::complex<double>> calculateGPUSV(cuDoubleComplex*
   ↪ beforeGate, cuDoubleComplex* gateValues, cuDoubleComplex*
   ↪ afterGate, StateVector* reordered, std::vector<std::vector<std
   ↪ ::complex<double>>> gateValuesV) {
 int arraySize = gateValuesV.size();
 cuDoubleComplex* before = new cuDoubleComplex[arraySize];
```

```
 cuDoubleComplex* gateVal = new cuDoubleComplex[arraySize * arraySize];
 cuDoubleComplex* after = new cuDoubleComplex[arraySize];

 std::vector<std::complex<double>> currentState = reordered->getState();
 for (int i = 0; i < currentState.size(); i++) {
  before[i] = convertQubitComplex(currentState[i]);
 }
 for (int i = 0; i < arraySize; i++) {
  for (int j = 0; j < arraySize; j++) {
   gateVal[i * arraySize + j] = convertQubitComplex(gateValuesV[i][j]);
  }
 }
 // Copy input vectors into CUDA memory
 cudaError_t cudaStatus;
 cudaStatus = cudaMemcpy(beforeGate, before, arraySize * sizeof(cuDoubleComplex), cudaMemcpyHostToDevice);
 if (cudaStatus != cudaSuccess) {
  fprintf(stderr, "cudaMemcpy failed!");
  delete before;
  delete gateVal;
  delete after;
  return std::vector<std::complex<double>>();
 }
 cudaStatus = cudaMemcpy(gateValues, gateVal, (arraySize * arraySize) * sizeof(cuDoubleComplex), cudaMemcpyHostToDevice);
 if (cudaStatus != cudaSuccess) {
  fprintf(stderr, "cudaMemcpy failed!");
  delete before;
  delete gateVal;
  delete after;
  return std::vector<std::complex<double>>();
 }
 cudaStatus = cudaMemcpy(afterGate, after, arraySize * sizeof(cuDoubleComplex), cudaMemcpyHostToDevice);
 if (cudaStatus != cudaSuccess) {
  fprintf(stderr, "cudaMemcpy failed!");
  delete before;
  delete gateVal;
  delete after;
  return std::vector<std::complex<double>>();
 }
 if (arraySize > 256) {
  int blockSize = 256;
  int numBlocks = (arraySize + blockSize - 1) / blockSize;
  ValkGPULib::svMatrixMul <<<numBlocks, blockSize >>> (afterGate, beforeGate, gateValues, arraySize);
 }
 else {
  ValkGPULib::svMatrixMul <<<1, arraySize >>> (afterGate, beforeGate, gateValues, arraySize);
 }
 // Check for any errors launching the kernel
 cudaStatus = cudaGetLastError();
 if (cudaStatus != cudaSuccess) {
  fprintf(stderr, "addKernel launch failed: %s\n", cudaGetErrorString(
```

```
      ↪ cudaStatus));
  delete before;
  delete gateVal;
  delete after;
  return std::vector<std::complex<double>>();
 }

 // cudaDeviceSynchronize waits for the kernel to finish, and returns
 // any errors encountered during the launch agrim.
 cudaStatus = cudaDeviceSynchronize();
 if (cudaStatus != cudaSuccess) {
  fprintf(stderr, "cudaDeviceSynchronize returned error code %d after
      ↪ launching addKernel!\n", cudaStatus);
  delete before;
  delete gateVal;
  delete after;
  return std::vector<std::complex<double>>();
 }

 // Copy output vector from GPU buffer to host memory.
 cudaStatus = cudaMemcpy(after, afterGate, arraySize * sizeof(
     ↪ cuDoubleComplex), cudaMemcpyDeviceToHost);
 if (cudaStatus != cudaSuccess) {
  fprintf(stderr, "cudaMemcpy failed!");
  delete before;
  delete gateVal;
  delete after;
  return std::vector<std::complex<double>>();
 }
 std::vector<std::complex<double>> output;
 for (int i = 0; i < arraySize; i++) {
  output.push_back(convertComplexQubit(after[i]));
 }
 delete before;
 delete gateVal;
 delete after;
 return output;
}


// calculateGPUSV allows for small to medium statevectors to be easily
    ↪  parallelised
std::vector<std::complex<double>> calculateGPUSVPrime(cuDoubleComplex*
    ↪  beforeGate, cuDoubleComplex* gateValues, cuDoubleComplex*
    ↪ afterGate, StateVector* reordered, std::vector<std::vector<std
    ↪ ::complex<double>>> gateValuesV, int m_prim) {
 int arraySize = reordered->getState().size();
 cuDoubleComplex* before = new cuDoubleComplex[arraySize];
 cuDoubleComplex* gateVal = new cuDoubleComplex[m_prim * m_prim];
 cuDoubleComplex* after = new cuDoubleComplex[arraySize];

 std::vector<std::complex<double>> currentState = reordered->getState
     ↪ ();
 for (int i = 0; i < currentState.size(); i++) {
  before[i] = convertQubitComplex(currentState[i]);
 }
 for (int i = 0; i < m_prim; i++) {
  for (int j = 0; j < m_prim; j++) {
```

```
    gateVal[i * m_prim + j] = convertQubitComplex(gateValuesV[i][j]);
   }
  }
  // Copy input vectors into CUDA memory
  cudaError_t cudaStatus;
  cudaStatus = cudaMemcpy(beforeGate, before, arraySize * sizeof(
     ↪ cuDoubleComplex), cudaMemcpyHostToDevice);
  if (cudaStatus != cudaSuccess) {
   fprintf(stderr, "cudaMemcpy failed!");
   delete before;
   delete gateVal;
   delete after;
   return std::vector<std::complex<double>>();
  }
  cudaStatus = cudaMemcpy(gateValues, gateVal, (m_prim * m_prim) *
     ↪ sizeof(cuDoubleComplex), cudaMemcpyHostToDevice);
  if (cudaStatus != cudaSuccess) {
   fprintf(stderr, "cudaMemcpy failed!");
   delete before;
   delete gateVal;
   delete after;
   return std::vector<std::complex<double>>();
  }
  cudaStatus = cudaMemcpy(afterGate, after, arraySize * sizeof(
     ↪ cuDoubleComplex), cudaMemcpyHostToDevice);
  if (cudaStatus != cudaSuccess) {
   fprintf(stderr, "cudaMemcpy failed!");
   delete before;
   delete gateVal;
   delete after;
   return std::vector<std::complex<double>>();
  }
  if (arraySize > 256) {
   int blockSize = 256;
   int numBlocks = (arraySize + blockSize - 1) / blockSize;
   ValkGPULib::svMatrixMulPrime << <numBlocks, blockSize >> > (
      ↪ afterGate, beforeGate, gateValues, arraySize, m_prim);
  }
  else {
   ValkGPULib::svMatrixMulPrime << <1, arraySize >> > (afterGate,
      ↪ beforeGate, gateValues, arraySize, m_prim);
  }
  // Check for any errors launching the kernel
  cudaStatus = cudaGetLastError();
  if (cudaStatus != cudaSuccess) {
   fprintf(stderr, "addKernel launch failed: %s\n", cudaGetErrorString(
      ↪ cudaStatus));
   delete before;
   delete gateVal;
   delete after;
   return std::vector<std::complex<double>>();
  }

  // cudaDeviceSynchronize waits for the kernel to finish, and returns
  // any errors encountered during the launch agrim.
  cudaStatus = cudaDeviceSynchronize();
  if (cudaStatus != cudaSuccess) {
   fprintf(stderr, "cudaDeviceSynchronize returned error code %d after
```

```
        ↪ launching addKernel!\n", cudaStatus);
      delete before;
      delete gateVal;
      delete after;
      return std::vector<std::complex<double>>();
    }

    // Copy output vector from GPU buffer to host memory.
    cudaStatus = cudaMemcpy(after, afterGate, arraySize * sizeof(
        ↪ cuDoubleComplex), cudaMemcpyDeviceToHost);
    if (cudaStatus != cudaSuccess) {
      fprintf(stderr, "cudaMemcpy failed!");
      delete before;
      delete gateVal;
      delete after;
      return std::vector<std::complex<double>>();
    }
    std::vector<std::complex<double>> output;
    for (int i = 0; i < arraySize; i++) {
      output.push_back(convertComplexQubit(after[i]));
    }
    delete before;
    delete gateVal;
    delete after;
    return output;
  }
}
```

Listing B.13: GPUCompute.cuh: Optimised Library of GPU specific functions needed to parallelise gate calculations.

```
#include "AbstractDevice.h"
#include "cuda_runtime.h"

/*
 GPUDevice.cuh
 Description: This header file defines the GPU implementation of an
     ↪ Abstract Device as
 presented in AbstractDevice.h.

 Defined Classes:
 GPUQubitFactory
 GPUGateFactory
 GPUQuantumCircuit
 GPUQuantumProcessor
 GPUDevice

*/

// GPUQubitFactory implements the interface for AbstractQubitFactory
// Allocates, tracks and de-allocates memory for QubitStates
class GPUQubitFactory : public AbstractQubitFactory {
private:
 DeviceType type_;
 std::vector<Qubit*> qubits_;
public:
 GPUQubitFactory() {
  type_ = CPU_;
 }
```

```
 Qubit* generateQubit();
 ~GPUQubitFactory();
};

// GPUGateFactory implements the interface for AbstractGateFactory
// Allocates, tracks and de-allocates memory for Gate values
class GPUGateFactory : public AbstractGateFactory {
private:
 DeviceType type_;
 std::vector<Gate*> gates_;
public:
 GPUGateFactory() {
  type_ = CPU_;
 }
 Gate* generateGate(GateRequest request);
 ~GPUGateFactory();
};

// GPUQuantumCircuit implements the interface for AbstractQuantumCircuit
// Compiles calculation commands into actual matrices ready for computation
class GPUQuantumCircuit : public AbstractQuantumCircuit {
private:
 DeviceType type_;
 bool done_;
 std::map<std::string, std::vector<Qubit*>> qubitMap_;
 std::vector<std::vector<Calculation>> calculations_;
 GPUGateFactory* gateFactory_;
 int calcCounter = 0;
 std::vector<SVPair> zipSVPairs(std::vector<std::string> names, std::vector<int> locs);
 StateVector* sv_;
public:
 GPUQuantumCircuit(GPUGateFactory* gateFactory) {
  gateFactory_ = gateFactory;
  type_ = CPU_;
  done_ = false;
 }
 void loadQubitMap(std::map<std::string, std::vector<Qubit*>> qubitMap);
 void loadBlock(ConcurrentBlock block);
 std::vector<Calculation> getNextCalculation();
 std::map<std::string, std::vector<Qubit*>> returnResults();
 StateVector* getStateVector();
 bool checkComplete();
 ~GPUQuantumCircuit() {
  delete sv_;
 }
};

// GPUQuantumProcessor implements the interface for AbstractQuantumProcessor
// performs matrix calculations using the loaded quantum circuit to fetch calculations
class GPUQuantumProcessor : public AbstractQuantumProcessor {
private:
 DeviceType type_;
```

```
  AbstractQuantumCircuit* circuit_;
  std::vector<std::vector<std::complex<double>>> getCXResult(int n);
  std::vector<std::vector<std::complex<double>>> getGenericUResult(Gate* gate, int n);
 public:
  GPUQuantumProcessor() {
   type_ = CPU_;
  }
  void loadCircuit(AbstractQuantumCircuit* circuit);
  void calculate();
  void calculateWithStateVector();
  std::map<std::string, std::vector<Qubit*>> qubitMapfetchQubitValues();
 };

// GPUDevice implements the Abstract device interface
// Collects all components required for GPU execution
 class GPUDevice : public AbstractDevice {
 private:
  DeviceType type_;
  std::map<std::string, std::vector<Qubit*>> registerMap;
  GPUQubitFactory* qubitFactory;
  GPUGateFactory* gateFactory;
  GPUQuantumCircuit* quantumCircuit;
  GPUQuantumProcessor* quantumProcessor;
 public:
  GPUDevice() {
   type_ = CPU_;
   qubitFactory = new GPUQubitFactory();
   gateFactory = new GPUGateFactory();
   quantumCircuit = new GPUQuantumCircuit(gateFactory);
   quantumProcessor = new GPUQuantumProcessor();
  }
  void loadRegister(Register registerx);
  void transferQubitMap();
  void loadConcurrentBlock(ConcurrentBlock block);
  void runSimulation();
  void runSimulationSV();
  void run(std::vector<Register> registers, std::vector<ConcurrentBlock> blocks);
  void runSV(std::vector<Register> registers, std::vector<ConcurrentBlock> blocks);
  std::map<std::string, std::vector<Qubit*>> revealQuantumState();
  void prettyPrintQubitStates(std::map<std::string, std::vector<Qubit*>> qubits) {
   for (std::map<std::string, std::vector<Qubit*> >::iterator it = qubits.begin(); it != qubits.end(); ++it) {
    std::cout << "Register: " << it->first << std::endl;
    std::vector<Qubit*> regQubits = it->second;
    for (int i = 0; i < regQubits.size(); i++) {
     std::cout << "Location [" << i << "]: " << regQubits[i]->fetch(0)->real() << "+" << regQubits[i]->fetch(0)->imag() << "i" << " ||| " << regQubits[i]->fetch(1)->real() << "+" << regQubits[i]->fetch(1)->imag() << "i" << std::endl;
    }
   }
  }
  StateVector* getStateVector() {
   return quantumCircuit->getStateVector();
```

```
 }
 ~GPUDevice() {
  delete qubitFactory;
  delete gateFactory;
  delete quantumCircuit;
  delete quantumProcessor;
 }
};
```

Listing B.14: GPUDevice.cuh:This header file defines the GPU implementation of an Abstract Device as presented in AbstractDevice.h.

```
#pragma once
#include "GPUDevice.cuh"
#include "cuComplex.h"
#include <cmath>
#include <stdio.h>
#include "GPUCompute.cuh"
#include "GateUtilitiesGPU.cuh"

using namespace std::complex_literals;
const double ROOT2INV = 1.0 / std::pow(2, 0.5);

/*
 GPUDevice.cu
 Description: This file defines the implementation of the functions defined
 in GPUDevice.cuh

 Defined Classes:
 GPUQubitFactory
 GPUGateFactory
 GPUQuantumCircuit
 GPUQuantumProcessor
 GPUDevice

*/

// getGateMatrix gneerates basic primitive gates (U, CX)
// uses buildU3GateGPU to construct the parameterised U gate.
std::vector<std::vector<std::complex<double>>> getGateMatrixGPU(GateRequest gate) {
 GateRequestType gateType = gate.getGateType();
 switch (gateType) {
 case I:
  return std::vector<std::vector<std::complex<double>>> { {1, 0}, { 0, 1 } };
  break;
 case h:
  return std::vector<std::vector<std::complex<double>>> { {ROOT2INV, ROOT2INV}, { ROOT2INV, -1.0 * ROOT2INV } };
  break;
 case cx:
  return std::vector<std::vector<std::complex<double>>> { {1, 0, 0, 0}, { 0, 1, 0, 0 }, { 0, 0, 0, 1 }, { 0, 0, 1, 0 } };
  break;
 case U:
  return buildU3GateGPU(gate);
  break;
```

```
  case CX:
   return std::vector<std::vector<std::complex<double>>> { {1, 0, 0, 0},
     ↪ { 0, 1, 0, 0 }, { 0, 0, 0, 1 }, { 0, 0, 1, 0 } };
   break;
  }
 }

// generateQubit allocates heap memory for complex number and loads it
   ↪ into
// a heap memory allocated Qubit and tracks the generated qubits
Qubit* GPUQubitFactory::generateQubit()
{
 // Allocate heap memory for Qubit values
 std::complex<double>* s0 = new std::complex<double>;
 std::complex<double>* s1 = new std::complex<double>;
 *s0 = 1.0;
 *s1 = 0.0;
 // Allocate heap memory for Qubit and store values
 Qubit* generatedQubit = new Qubit(s0, s1);
 // Push into qubit tracker for deletion
 qubits_.push_back(generatedQubit);

 return generatedQubit;
}

// deconstructor cleans up any heap memory allocation
GPUQubitFactory::~GPUQubitFactory()
{
 for (auto qubit : qubits_) {
  delete qubit->fetch(0);
  delete qubit->fetch(1);
  delete qubit;
 }
}

// generateQubit allocates heap memory for complex numbers and loads it
   ↪ into
// a heap memory allocated Gate and tracks the generated gates
Gate* GPUGateFactory::generateGate(GateRequest request)
{
 std::vector<std::vector<std::complex<double>>> gateMatrix =
    ↪ getGateMatrixGPU(request);
 int gateM = gateMatrix.size();
 int gateN = gateMatrix[0].size();

 Gate* generatedGate = new Gate(gateM, gateN, gateMatrix);
 gates_.push_back(generatedGate);
 return generatedGate;
}

// deconstructor cleans up any heap memory allocation
GPUGateFactory::~GPUGateFactory()
{
 for (auto gate : gates_) {
  delete gate;
 }
}

```

```
// zipSVPairs zips together identifiers and locations to generate
   ↪ SVPairs which can be used in
// statevector lookup
std::vector<SVPair> GPUQuantumCircuit::zipSVPairs(std::vector<std::
   ↪ string> names, std::vector<int> locs)
{
 std::vector<SVPair> values;
 for (int i = 0; i < names.size(); i++) {
  values.push_back(SVPair(names[i], locs[i]));
 }
 return values;
}

void GPUQuantumCircuit::loadQubitMap(std::map<std::string, std::vector<
   ↪ Qubit*>> qubitMap)
{
 qubitMap_ = qubitMap;
 sv_ = new StateVector(&qubitMap_);
 sv_->tensorProduct();
}

// loadBlock takes a concurrent block from the Staging module and
   ↪ converts it into
// a series if operable Calculation datatypes
void GPUQuantumCircuit::loadBlock(ConcurrentBlock block)
{
 std::vector<GateRequest> gates = block.getGates();
 std::vector<Calculation> calcs;
 for (auto gate : gates) {
  std::vector<std::string> registers = gate.getRegisters();
  std::vector<int> locations = gate.getLocations();
  std::vector<Qubit*> qubitValues;
  for (int i = 0; i < registers.size(); i++) {
   qubitValues.push_back(qubitMap_[registers[i]][locations[i]]);
  }
  Gate* gateTrue = gateFactory_->generateGate(gate);
  std::vector<SVPair> svPairs = zipSVPairs(registers, locations);
  Calculation calc = Calculation(gateTrue, qubitValues, svPairs);
  calcs.push_back(calc);
 }
 calculations_.push_back(calcs);
}

// getNextCalculation is used during the processing, to queue up
   ↪ calculations and
// raises the done_ flag if computation is complete
std::vector<Calculation> GPUQuantumCircuit::getNextCalculation()
{
 if (calcCounter == calculations_.size() - 1) {
  done_ = true;
  return calculations_[calcCounter];
 }
 else {
  std::vector<Calculation> val = calculations_[calcCounter];
  calcCounter++;
  return val;
 }
}
```

```

// For fast computation
std::map<std::string, std::vector<Qubit*>> GPUQuantumCircuit::returnResults()
{
 return qubitMap_;
}

// For Statevector computation
StateVector* GPUQuantumCircuit::getStateVector()
{
 return sv_;
}

bool GPUQuantumCircuit::checkComplete()
{
 if (calculations_.size() == 0) {
  return true;
 }
 return done_;
}

// getCXResults generates an 2^n by 2^n matrix from the tensor product of I gates and a final CX gate
// returns this matrix for computation
std::vector<std::vector<std::complex<double>>> GPUQuantumProcessor::getCXResult(int n)
{
 // n is the number of qubits, we have to have n-2 I gates and then a CX gate at the end
 if (n < 2) {
  return std::vector<std::vector<std::complex<double>>>();
 }
 std::vector<std::vector<std::complex<double>>> output;
 // overall sidelength of resultant gate
 int dimOverall = std::pow(2, n);
 // number of I multiplications required
 int leftOver = n - 2;
 if (leftOver == 0) {
  output = { {1, 0, 0, 0}, {0, 1, 0, 0}, {0, 0, 0, 1}, {0, 0, 1, 0} };
  return output;
 }
 output.resize(dimOverall);
 for (int i = 0; i < dimOverall; i++) {
  std::vector<std::complex<double>> subVec;
  subVec.resize(dimOverall);
  output[i] = subVec;
 }
 // skinny calculation due to the CX being the last matrix in a series of I tensor products
 // using tail methodology
 for (int i = 0; i < std::pow(2, leftOver); i++) {
  output[4 * i][4 * i] = 1;
  output[4 * i + 1][4 * i + 1] = 1;
  output[4 * i + 2][4 * i + 3] = 1;
  output[4 * i + 3][4 * i + 2] = 1;
 }
 return output;
```

```
}

// getGenericUResult return tensor product of a series of I gates and finally the U gate we are applying
std::vector<std::vector<std::complex<double>>> GPUQuantumProcessor::getGenericUResult(Gate* gate, int n)
{
 // n is the number of qubits, we have to have n-2 I gates and then a CX gate at the end
 if (n < 1) {
  return std::vector<std::vector<std::complex<double>>>();
 }
 std::vector<std::vector<std::complex<double>>> output;
 // overall sidelength of resultant gate
 int dimOverall = std::pow(2, n);
 // number of I multiplications required
 int leftOver = n - 1;
 if (leftOver == 0) {
  output = gate->getArray();
  return output;
 }
 output.resize(dimOverall);
 for (int i = 0; i < dimOverall; i++) {
  std::vector<std::complex<double>> subVec;
  subVec.resize(dimOverall);
  output[i] = subVec;
 }
 // skinny calculation due to the CX being the last matrix in a series of I tensor products
 // using tail methodology
 for (int i = 0; i < std::pow(2, leftOver); i++) {
  output[2 * i][2 * i] = gate->fetchValue(0, 0);
  output[2 * i][2 * i + 1] = gate->fetchValue(0, 1);
  output[2 * i + 1][2 * i] = gate->fetchValue(1, 0);
  output[2 * i + 1][2 * i + 1] = gate->fetchValue(1, 1);
 }
 return output;
}

void GPUQuantumProcessor::loadCircuit(AbstractQuantumCircuit* circuit)
{
 circuit_ = circuit;
}

// calculate method for isolated fast computation
void GPUQuantumProcessor::calculate()
{
 // Generate initial arrays
 //cuDoubleComplex* initialValues;
 cuDoubleComplex* beforeGate;
 cuDoubleComplex* gateValues;
 cuDoubleComplex* afterGate;
 while (!circuit_->checkComplete()) {
  std::vector<Calculation> calcBlock = circuit_->getNextCalculation();
  for (auto calc : calcBlock) { // parallelisation next iteration
   Gate* gate = calc.getGate();
   int m = gate->getM();
   int n = gate->getN();
```

```
      int qubitN = m / 2;
      cudaError_t cudaStatus;
      // Allocate shared space
      cudaStatus = cudaMalloc((void**)&beforeGate, m * sizeof(
         ↪ cuDoubleComplex));    // Allocate GPU memory for gate arrays
      if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMalloc failed!");
        goto Error;
      }
      cudaStatus = cudaMalloc((void**)&afterGate, m * sizeof(
         ↪ cuDoubleComplex));
      if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMalloc failed!");
        goto Error;
      }
      cudaStatus = cudaMalloc((void**)&gateValues, (m*n) * sizeof(
         ↪ cuDoubleComplex));
      if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMalloc failed!");
        goto Error;
      }
      // Uses GPUCompute.cuh functions to perform calculation
      std::vector<std::complex<double>> res = ValkGPULib::calculateGPU(
         ↪ beforeGate, gateValues, afterGate, calc.getGate(), calc.
         ↪ getQubits());
      if (res.size() == 2) {
        circuit_->getStateVector()->quickRefresh();
      }
      if (res.size() == 4) {
        circuit_->getStateVector()->modifyState(res, calc.getLocations()
           ↪ [0], calc.getLocations()[1]);
      }
      cudaFree(beforeGate);
      cudaFree(afterGate);
      cudaFree(gateValues);
    }
  }
  return;
Error:
  cudaFree(beforeGate);
  cudaFree(afterGate);
  cudaFree(gateValues);
}

// calculateWithStateVector for accurate Quantum Computer emulation,
   ↪ uses statevector in it's entirety
void GPUQuantumProcessor::calculateWithStateVector()
{
  // Generate initial arrays
  //cuDoubleComplex* initialValues;
  cuDoubleComplex* beforeGate;
  cuDoubleComplex* gateValues;
  cuDoubleComplex* afterGate;
  while (!circuit_->checkComplete()) { // check if there are still
     ↪ calculations to consume
    std::vector<Calculation> calcBlock = circuit_->getNextCalculation();
       ↪ // fetch calculation
    for (auto calc : calcBlock) {
```

```
    Gate* gate = calc.getGate();
    int m = gate->getM();
    int n = gate->getN();
    int qubitN = m / 2;
    StateVector* sv = circuit_->getStateVector();      // get current
       ↪ state vector
    int gateDim = sv->getState().size();
    std::vector<SVPair> newOrder = calc.getNewOrder(sv->getOrder()); //
       ↪ use the calculation function to work out the new order of the
       ↪  state vector for tail procedure
    StateVector* reordered = sv->reorder(newOrder);      // fetch
       ↪ temporary statevector using reordered tensor product
    std::vector<std::vector<std::complex<double>>> gateValuesV;
    if (m == 2) {
     gateValuesV = getGenericUResult(gate, sv->getN());    // Generate
        ↪ full gate matrix
    }
    if (m == 4) {
     gateValuesV = getCXResult(sv->getN());       // Generate full gate
        ↪ matrix
    }
    if (gateValuesV.size() == 0) {
     return;
    }
    cudaError_t cudaStatus;
    // Allocate shared space
    cudaStatus = cudaMalloc((void**)&beforeGate, gateDim * sizeof(
       ↪ cuDoubleComplex)); // Allocate GPU memory for gate arrays
    if (cudaStatus != cudaSuccess) {
     fprintf(stderr, "cudaMalloc failed!");
     goto Error;
    }
    cudaStatus = cudaMalloc((void**)&afterGate, gateDim * sizeof(
       ↪ cuDoubleComplex));
    if (cudaStatus != cudaSuccess) {
     fprintf(stderr, "cudaMalloc failed!");
     goto Error;
    }
    cudaStatus = cudaMalloc((void**)&gateValues, (gateDim * gateDim) *
       ↪ sizeof(cuDoubleComplex));
    if (cudaStatus != cudaSuccess) {
     fprintf(stderr, "cudaMalloc failed!");
     goto Error;
    }
    std::vector<std::complex<double>> res;
    if (gateDim < 256) {
     res = ValkGPULib::calculateGPUSV(beforeGate, gateValues, afterGate,
        ↪  reordered, gateValuesV); // For smaller scale gates we
        ↪ used partially parallel processing
    }
    else {
     // Ultra parallel
     res = ValkGPULib::calculateGPULargeSV(beforeGate, gateValues,
        ↪ afterGate, reordered, gateValuesV); // Larger gate require
        ↪ fully parallel processing
    }
    reordered->directModify(res);        // set newValues of reordered
       ↪ state vector
```

```
   sv->reconcile(reordered);          // reconcile temporary order for
      ↪ statevector for the original order
   cudaFree(beforeGate);
   cudaFree(afterGate);
   cudaFree(gateValues);
  }
 }
 return;
Error:
 cudaFree(beforeGate);
 cudaFree(afterGate);
 cudaFree(gateValues);
}

std::map<std::string, std::vector<Qubit*>> GPUQuantumProcessor::
   ↪ qubitMapfetchQubitValues()
{
 return circuit_->returnResults();
}

void GPUDevice::loadRegister(Register registerx)
{
 if (registerx.isQuantum()) {
  QuantumRegister qReg = registerx.getQuantumRegister();
  std::string regName = qReg.getIdentifier();
  int width = qReg.getWidth();
  std::vector<Qubit*> registerQubits;
  for (int i = 0; i < width; i++) {
   registerQubits.push_back(qubitFactory->generateQubit());
  }
  registerMap.insert(std::pair<std::string, std::vector<Qubit*>>(
     ↪ regName, registerQubits));
 }
}

void GPUDevice::transferQubitMap()
{
 quantumCircuit->loadQubitMap(registerMap);
}

void GPUDevice::loadConcurrentBlock(ConcurrentBlock block)
{
 quantumCircuit->loadBlock(block);
}

void GPUDevice::runSimulation()
{
 quantumProcessor->loadCircuit(quantumCircuit);
 quantumProcessor->calculate();
}

void GPUDevice::runSimulationSV()
{
 quantumProcessor->loadCircuit(quantumCircuit);
 quantumProcessor->calculateWithStateVector();
}

void GPUDevice::run(std::vector<Register> registers, std::vector<
```

```
    ↪ ConcurrentBlock> blocks)
{
 for (auto reg : registers) {
  loadRegister(reg);
 }
 transferQubitMap();
 for (auto block : blocks) {
  loadConcurrentBlock(block);
 }
 runSimulation();
}

void GPUDevice::runSV(std::vector<Register> registers, std::vector<
    ↪ ConcurrentBlock> blocks)
{
 for (auto reg : registers) {
  loadRegister(reg);
 }
 transferQubitMap();
 for (auto block : blocks) {
  loadConcurrentBlock(block);
 }
 runSimulationSV();
}

std::map<std::string, std::vector<Qubit*>> GPUDevice::
    ↪ revealQuantumState()
{
 return quantumProcessor->qubitMapfetchQubitValues();
}
```

Listing B.15: GPUDevice.cu: This file defines the implementation of the functions defined in GPUDevice.cuh

```
#pragma once
#include "GPUDevice.cuh"
#include "cuComplex.h"
#include <cmath>
#include <stdio.h>
#include "GPUCompute.cuh"
#include "GateUtilitiesGPU.cuh"
#include <chrono>

using namespace std::complex_literals;
const double ROOT2INV = 1.0 / std::pow(2, 0.5);

/*
 GPUDevice.cu
 Description: This file defines the implementation of the functions
     ↪ defined
 in GPUDevice.cuh

 Defined Classes:
 GPUQubitFactory
 GPUGateFactory
 GPUQuantumCircuit
 GPUQuantumProcessor
 GPUDevice

```

```
*/

// getGateMatrix gneerates basic primitive gates (U, CX)
// uses buildU3GateGPU to construct the parameterised U gate.
std::vector<std::vector<std::complex<double>>> getGateMatrixGPU(GateRequest gate) {
 GateRequestType gateType = gate.getGateType();
 switch (gateType) {
 case I:
  return std::vector<std::vector<std::complex<double>>> { {1, 0}, { 0, 1 } };
  break;
 case h:
  return std::vector<std::vector<std::complex<double>>> { {ROOT2INV, ROOT2INV}, { ROOT2INV, -1.0 * ROOT2INV } };
  break;
 case cx:
  return std::vector<std::vector<std::complex<double>>> { {1, 0, 0, 0}, { 0, 1, 0, 0 }, { 0, 0, 0, 1 }, { 0, 0, 1, 0 } };
  break;
 case U:
  return buildU3GateGPU(gate);
  break;
 case CX:
  return std::vector<std::vector<std::complex<double>>> { {1, 0, 0, 0}, { 0, 1, 0, 0 }, { 0, 0, 0, 1 }, { 0, 0, 1, 0 } };
  break;
 }
}

// generateQubit allocates heap memory for complex number and loads it into
// a heap memory allocated Qubit and tracks the generated qubits
Qubit* GPUQubitFactory::generateQubit()
{
 // Allocate heap memory for Qubit values
 std::complex<double>* s0 = new std::complex<double>;
 std::complex<double>* s1 = new std::complex<double>;
 *s0 = 1.0;
 *s1 = 0.0;
 // Allocate heap memory for Qubit and store values
 Qubit* generatedQubit = new Qubit(s0, s1);
 // Push into qubit tracker for deletion
 qubits_.push_back(generatedQubit);

 return generatedQubit;
}

// deconstructor cleans up any heap memory allocation
GPUQubitFactory::~GPUQubitFactory()
{
 for (auto qubit : qubits_) {
  delete qubit->fetch(0);
  delete qubit->fetch(1);
  delete qubit;
 }
}

```

```
// generateQubit allocates heap memory for complex numbers and loads it into
// a heap memory allocated Gate and tracks the generated gates
Gate* GPUGateFactory::generateGate(GateRequest request)
{
 std::vector<std::vector<std::complex<double>>> gateMatrix = getGateMatrixGPU(request);
 int gateM = gateMatrix.size();
 int gateN = gateMatrix[0].size();

 Gate* generatedGate = new Gate(gateM, gateN, gateMatrix);
 gates_.push_back(generatedGate);
 return generatedGate;
}

// deconstructor cleans up any heap memory allocation
GPUGateFactory::~GPUGateFactory()
{
 for (auto gate : gates_) {
  delete gate;
 }
}

// zipSVPairs zips together identifiers and locations to generate SVPairs which can be used in
// statevector lookup
std::vector<SVPair> GPUQuantumCircuit::zipSVPairs(std::vector<std::string> names, std::vector<int> locs)
{
 std::vector<SVPair> values;
 for (int i = 0; i < names.size(); i++) {
  values.push_back(SVPair(names[i], locs[i]));
 }
 return values;
}

void GPUQuantumCircuit::loadQubitMap(std::map<std::string, std::vector<Qubit*>> qubitMap)
{
 qubitMap_ = qubitMap;
 sv_ = new StateVector(&qubitMap_);
 sv_->tensorProduct();
}

// loadBlock takes a concurrent block from the Staging module and converts it into
// a series if operable Calculation datatypes
void GPUQuantumCircuit::loadBlock(ConcurrentBlock block)
{
 std::vector<GateRequest> gates = block.getGates();
 std::vector<Calculation> calcs;
 for (auto gate : gates) {
  std::vector<std::string> registers = gate.getRegisters();
  std::vector<int> locations = gate.getLocations();
  std::vector<Qubit*> qubitValues;
  for (int i = 0; i < registers.size(); i++) {
   qubitValues.push_back(qubitMap_[registers[i]][locations[i]]);
  }
```

```
    Gate* gateTrue = gateFactory_->generateGate(gate);
    std::vector<SVPair> svPairs = zipSVPairs(registers, locations);
    Calculation calc = Calculation(gateTrue, qubitValues, svPairs);
    calcs.push_back(calc);
  }
  calculations_.push_back(calcs);
}

// getNextCalculation is used during the processing, to queue up calculations and
// raises the done_ flag if computation is complete
std::vector<Calculation> GPUQuantumCircuit::getNextCalculation()
{
  if (calcCounter == calculations_.size() - 1) {
    done_ = true;
    return calculations_[calcCounter];
  }
  else {
    std::vector<Calculation> val = calculations_[calcCounter];
    calcCounter++;
    return val;
  }
}

// For fast computation
std::map<std::string, std::vector<Qubit*>> GPUQuantumCircuit::returnResults()
{
  return qubitMap_;
}

// For Statevector computation
StateVector* GPUQuantumCircuit::getStateVector()
{
  return sv_;
}

bool GPUQuantumCircuit::checkComplete()
{
  if (calculations_.size() == 0) {
    return true;
  }
  return done_;
}


void GPUQuantumProcessor::loadCircuit(AbstractQuantumCircuit* circuit)
{
  circuit_ = circuit;
}

// calculate method for isolated fast computation
void GPUQuantumProcessor::calculate()
{
  // Generate initial arrays
  //cuDoubleComplex* initialValues;
  cuDoubleComplex* beforeGate;
  cuDoubleComplex* gateValues;
```

```
 cuDoubleComplex* afterGate;
 while (!circuit_->checkComplete()) {
  std::vector<Calculation> calcBlock = circuit_->getNextCalculation();
  for (auto calc : calcBlock) { // parallelisation next iteration
   Gate* gate = calc.getGate();
   int m = gate->getM();
   int n = gate->getN();
   int qubitN = m / 2;
   cudaError_t cudaStatus;
   // Allocate shared space
   cudaStatus = cudaMalloc((void**)&beforeGate, m * sizeof(cuDoubleComplex));    // Allocate GPU memory for gate arrays
   if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMalloc failed!");
    goto Error;
   }
   cudaStatus = cudaMalloc((void**)&afterGate, m * sizeof(cuDoubleComplex));
   if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMalloc failed!");
    goto Error;
   }
   cudaStatus = cudaMalloc((void**)&gateValues, (m*n) * sizeof(cuDoubleComplex));
   if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMalloc failed!");
    goto Error;
   }
   // Uses GPUCompute.cuh functions to perform calculation
   std::vector<std::complex<double>> res = ValkGPULib::calculateGPU(beforeGate, gateValues, afterGate, calc.getGate(), calc.getQubits());
   if (res.size() == 2) {
    circuit_->getStateVector()->quickRefresh();
   }
   if (res.size() == 4) {
    circuit_->getStateVector()->modifyState(res, calc.getLocations()[0], calc.getLocations()[1]);
   }
   cudaFree(beforeGate);
   cudaFree(afterGate);
   cudaFree(gateValues);
  }
 }
 return;
Error:
 cudaFree(beforeGate);
 cudaFree(afterGate);
 cudaFree(gateValues);
}

// calculateWithStateVector for accurate Quantum Computer emulation, uses statevector in it's entirety
void GPUQuantumProcessor::calculateWithStateVector()
{
 // Generate initial arrays
 //cuDoubleComplex* initialValues;
 cuDoubleComplex* beforeGate;
```

```
 cuDoubleComplex* gateValues;
 cuDoubleComplex* afterGate;
 long long counter = 0;
 while (!circuit_->checkComplete()) { // check if there are still
    ↪ calculations to consume
  std::vector<Calculation> calcBlock = circuit_->getNextCalculation();
     ↪ // fetch calculation
  for (auto calc : calcBlock) {
   Gate* gate = calc.getGate();
   int m = gate->getM();
   int n = gate->getN();
   int qubitN = m / 2;
   StateVector* sv = circuit_->getStateVector();      // get current
      ↪ state vector
   int gateDim = sv->getState().size();
   std::vector<SVPair> newOrder = calc.getNewOrder(sv->getOrder()); //
      ↪ use the calculation function to work out the new order of the
      ↪  state vector for tail procedure
   StateVector* reordered = sv->reorder(newOrder);      // fetch
      ↪ temporary statevector using reordered tensor product
   std::vector<std::vector<std::complex<double>>> gateValuesV = gate->
      ↪ getArray();
   cudaError_t cudaStatus;
   // Allocate shared space
   cudaStatus = cudaMalloc((void**)&beforeGate, gateDim * sizeof(
      ↪ cuDoubleComplex)); // Allocate GPU memory for gate arrays
   if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMalloc failed!");
    goto Error;
   }
   cudaStatus = cudaMalloc((void**)&afterGate, gateDim * sizeof(
      ↪ cuDoubleComplex));
   if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMalloc failed!");
    goto Error;
   }
   cudaStatus = cudaMalloc((void**)&gateValues, (m * m) * sizeof(
      ↪ cuDoubleComplex));
   if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMalloc failed!");
    goto Error;
   }
   std::vector<std::complex<double>> res;
   res = ValkGPULib::calculateGPUSVPrime(beforeGate, gateValues,
      ↪ afterGate, reordered, gateValuesV, m);
   reordered->directModify(res);         // set newValues of reordered
      ↪ state vector
   sv->reconcile(reordered);          // reconcile temporary order for
      ↪ statevector for the original order
   cudaFree(beforeGate);
   cudaFree(afterGate);
   cudaFree(gateValues);
  }
 }
 return;
Error:
 cudaFree(beforeGate);
 cudaFree(afterGate);
```

```
 cudaFree(gateValues);
}

std::map<std::string, std::vector<Qubit*>> GPUQuantumProcessor::qubitMapfetchQubitValues()
{
 return circuit_->returnResults();
}

void GPUDevice::loadRegister(Register registerx)
{
 if (registerx.isQuantum()) {
  QuantumRegister qReg = registerx.getQuantumRegister();
  std::string regName = qReg.getIdentifier();
  int width = qReg.getWidth();
  std::vector<Qubit*> registerQubits;
  for (int i = 0; i < width; i++) {
   registerQubits.push_back(qubitFactory->generateQubit());
  }
  registerMap.insert(std::pair<std::string, std::vector<Qubit*>>(regName, registerQubits));
 }
}

void GPUDevice::transferQubitMap()
{
 quantumCircuit->loadQubitMap(registerMap);
}

void GPUDevice::loadConcurrentBlock(ConcurrentBlock block)
{
 quantumCircuit->loadBlock(block);
}

void GPUDevice::runSimulation()
{
 quantumProcessor->loadCircuit(quantumCircuit);
 quantumProcessor->calculate();
}

void GPUDevice::runSimulationSV()
{
 quantumProcessor->loadCircuit(quantumCircuit);
 quantumProcessor->calculateWithStateVector();
}

void GPUDevice::run(std::vector<Register> registers, std::vector<ConcurrentBlock> blocks)
{
 for (auto reg : registers) {
  loadRegister(reg);
 }
 transferQubitMap();
 for (auto block : blocks) {
  loadConcurrentBlock(block);
 }
 runSimulation();
}
```

```

void GPUDevice::runSV(std::vector<Register> registers, std::vector<
    ↪ ConcurrentBlock> blocks)
{
 for (auto reg : registers) {
  loadRegister(reg);
 }
 transferQubitMap();
 for (auto block : blocks) {
  loadConcurrentBlock(block);
 }
 runSimulationSV();
}

std::map<std::string, std::vector<Qubit*>> GPUDevice::
    ↪ revealQuantumState()
{
 return quantumProcessor->qubitMapfetchQubitValues();
}
```

Listing B.16: GPUDevice.cu: This file defines the Optimised implementation of the functions defined in GPUDevice.cuh

```
#pragma once
#include "BaseTypes.h"

/*
 JSONify.h
 Description: File provides interface for JSON printing

*/

// JSONify provides methods to convert results of computation into
    ↪ easily parsable JSON format
class JSONify {
private:
 std::vector<Register> registers_;
 StateVector* sv_;
public:
 JSONify(std::vector<Register> registers, StateVector* sv) {
  registers_ = registers;
  sv_ = sv;
 }
 void printJSON();
};
```

Listing B.17: JSONify.h: File provides interface for JSON printing

```
#include "JSONify.h"

/*
 JSONify.cpp
 Description: File provides implementation of JSON printing

*/

// printJSON converts results of computation into JSOn format readable
    ↪ by VisualQ
void JSONify::printJSON()
```

```
{
 std::string starter = "{";
 // Tell it about the state vector
 starter = starter + "\"StateVector\" : [ \n";
 for (int i = 0; i < sv_->getState().size(); i++) {
  std::complex<double> svVal = sv_->getState()[i];
  starter = starter + "\"" + std::to_string(svVal.real()) + " + " + std
    ↪ ::to_string(svVal.imag()) + "i\",\n";
 }
 starter = starter.substr(0, starter.size() - 2);
 starter = starter + "\n],\n";
 starter = starter + "\"ClassicalRegisters\" : [";
 for (int i = 0; i < registers_.size(); i++) {
  if (!registers_[i].isQuantum()) {
   ClassicalRegister cr = registers_[i].getClassicalRegister();
   starter = starter + "{ \n \"id\": \"" + cr.getIdentifier() + "\", \n
     ↪ \"values\": [ \n";
   for (int j = 0; j < cr.getWidth(); j++) {
    starter = starter + std::to_string(cr.getValue(j)) + ",\n";
   }
   starter = starter.substr(0, starter.size() - 2);
   starter = starter + "\n] \n},";
  }
 }
 starter = starter.substr(0, starter.size() - 1);
 starter = starter + "\n]";
 starter = starter + "\n}";
 std::cout << starter << std::endl;
}
```

Listing B.18: JSONify.cpp: File provides implementation of JSON printing

```
#pragma once

#include <map>
#include "BaseTypes.h"
#include <iostream>

/*
 Measurement.h
 Description: File provides interface for Quantum state measurement

*/

// MeasurementCalculator provides methods for measurement of the
   ↪ quantum state
// into classical registers for fast compute mode
class MeasurementCalculator {
private:
 std::map<std::string, std::vector<Qubit*>> registerMap_;
 std::map<std::string, std::vector<int>> measuredMap_;
 bool selectState0(Qubit* val);
 std::vector<MeasureCommand> commands_;
 std::vector<Register> allRegisters_;
 int findReg(std::string identifier) {
  for (int i = 0; i < allRegisters_.size(); i++) {
   if (allRegisters_[i].getName() == identifier) {
    return i;
   }
```

```
  }
  return -1;
 }
public:
 MeasurementCalculator(std::vector<Register> allRegisters);
 void registerHandover(std::map<std::string, std::vector<Qubit*>> registerMap);
 int measureSingle(std::string registerName, int location);
 void measureAll();
 std::map<std::string, std::vector<int>> returnMeasurementMap();
 void loadMeasureCommands(std::vector<MeasureCommand> commands);
 void passMeasurementsIntoClassicalRegisters();
 Register fetchRegister(std::string name);
 std::vector<Register> getAllRegisters() {
  return allRegisters_;
 }
 void printClassicalRegisters();
};

// StateVectorMeasurement provides methods for measurement of the entire
// statevector in statevector compute mode
class StateVectorMeasurement {
private:
 StateVector* sv_;
 double getMagnitude(std::complex<double> value);
 double getTotalMagnitude();
 int state_;
 std::vector<MeasureCommand> commands_;
 std::vector<Register> allRegisters_;
 int findReg(std::string identifier) {
  for (int i = 0; i < allRegisters_.size(); i++) {
   if (allRegisters_[i].getName() == identifier) {
    return i;
   }
  }
  return -1;
 }
public:
 StateVectorMeasurement(StateVector* sv, std::vector<Register> allRegister);
 void measure();
 void loadMeasureCommands(std::vector<MeasureCommand> commands);
 void passMeasurementsIntoClassicalRegisters();
 void printClassicalRegisters();
 std::vector<Register> getAllRegisters();
 StateVector* getStateVector();
};
```

Listing B.19: Measurement.h: File provides interface for Quantum state measurement

```
#include "Measurement.h"
#include <random>
#include <ctime>

/*
 Measurement.cpp
 Description: File provides implementation for Quantum state measurement
```

```

*/

// selectState0 calculates whether the qubit is in state 0 or not
// for single qubit measurement
bool MeasurementCalculator::selectState0(Qubit* val)
{
 std::complex<double>* s_0 = val->fetch(0);
 std::complex<double>* s_1 = val->fetch(1);

 double val_0 = std::pow(s_0->real(), 2) + std::pow(s_0->imag(), 2);
 double val_1 = std::pow(s_1->real(), 2) + std::pow(s_1->imag(), 2);

 double randomVal = ((double)std::rand() / (RAND_MAX)) * (val_0 + val_1);
 return randomVal <= val_0;
}

MeasurementCalculator::MeasurementCalculator(std::vector<Register> allRegister)
{
 allRegisters_ = allRegister;
}

void MeasurementCalculator::registerHandover(std::map<std::string, std::vector<Qubit*>> registerMap)
{
 registerMap_ = registerMap;
}

// Measures single qubits in fast compute mode
int MeasurementCalculator::measureSingle(std::string registerName, int location)
{
 Qubit* quantumValue = registerMap_[registerName][location];
 return selectState0(quantumValue) ? 0 : 1;
}

void MeasurementCalculator::measureAll(){
 for (std::map<std::string, std::vector<Qubit*>>::iterator it = registerMap_.begin(); it != registerMap_.end(); ++it) {
  for (int i = 0; i < it->second.size(); i++) {
   measuredMap_[it->first].push_back(measureSingle(it->first, i));
  }
 }
}

std::map<std::string, std::vector<int>> MeasurementCalculator::returnMeasurementMap()
{
 return measuredMap_;
}

void MeasurementCalculator::loadMeasureCommands(std::vector<MeasureCommand> commands)
{
 commands_ = commands;
}
```

```

// Uses user inputted measure commands to pass resolved qubit states
    ↪ into the classical registers
// that the measurement was requested into
void MeasurementCalculator::passMeasurementsIntoClassicalRegisters()
{
 for (auto command : commands_) {
  idLocationPairs qReg = command.getFrom();
  idLocationPairs cReg = command.getTo();

  int cRegLoc = findReg(cReg.identifiers[0]);
  if (cRegLoc != -1) {
   ClassicalRegister cRegVal = allRegisters_[cRegLoc].
       ↪ getClassicalRegister();
   cRegVal.setValue(cReg.locations[0], measuredMap_[qReg.identifiers
       ↪ [0]][qReg.locations[0]]);
   Register cRegFin = allRegisters_[cRegLoc];
   cRegFin.setClassicalRegister(cRegVal);
   allRegisters_[cRegLoc] = cRegFin;
  }
 }
}

Register MeasurementCalculator::fetchRegister(std::string name)
{
 int loc = findReg(name);
 if (loc == -1) {
  loc = 0;
 }
 return allRegisters_[loc];
}

void MeasurementCalculator::printClassicalRegisters()
{
 for (auto reg : allRegisters_) {
  if (!reg.isQuantum()) {
   ClassicalRegister cReg = reg.getClassicalRegister();
   std::cout << "Classical Register Identifier: " << cReg.getIdentifier
       ↪ () << std::endl;
   for (int i = 0; i < cReg.getWidth(); i++) {
    std::cout << "Location [" << i << "]: " << cReg.getValue(i) << std
        ↪ ::endl;
   }
  }
 }
}

// getMagnitude calculates magnitude of single complex number in a
    ↪ euclidean sense
double StateVectorMeasurement::getMagnitude(std::complex<double> value)
{
 return std::pow(value.real(), 2) + std::pow(value.imag(), 2);
}

// getTotalMagnitude calculates magnitude of the entire statevector in
    ↪ terms of L2 Norm
double StateVectorMeasurement::getTotalMagnitude()
{
```

```
 double total = 0;
 std::vector<std::complex<double>> state = sv_->getState();
 for (int i = 0; i < state.size(); i++) {
  total += getMagnitude(state[i]);
 }
 return total;
}

StateVectorMeasurement::StateVectorMeasurement(StateVector* sv, std::vector<Register> allRegister)
{
 sv_ = sv;
 allRegisters_ = allRegister;
}

// measure calculates which state in the statevector the qubits have collpased to using a random number
// generator and the probabilities expressed in the statevector
void StateVectorMeasurement::measure()
{
 double totalMag =  getTotalMagnitude();       // Find total magnitude of the statevector
 std::srand(std::time(nullptr));          // set seed fopr random number  generator to use the current time
 int randomVal = std::rand() % 100;        // generate number between 1  and 100
 double measurement = ((double)randomVal / 100) * (totalMag); // normalise this number to total magnitude of the statevector
 int state = 0;
 if (sv_->getState().size() == 0) {
  return;
 }
 std::vector<std::complex<double>> values = sv_->getState();
 double soFar = getMagnitude(values[state]);      // Keep iterating through the possible states until the accumulated magnitude
 while (measurement > soFar) {          // goes above the "measured" value
  state++;
  soFar += getMagnitude(values[state]);
 }
 state_ = state;             // return selected state
}

void StateVectorMeasurement::loadMeasureCommands(std::vector<MeasureCommand> commands)
{
 commands_ = commands;
}

// Uses user inputted measure commands to pass resolved qubit states into the classical registers
// that the measurement was requested into
void StateVectorMeasurement::passMeasurementsIntoClassicalRegisters()
{
 for (auto command : commands_) {
  idLocationPairs qReg = command.getFrom();
  idLocationPairs cReg = command.getTo();

```

```
  int cRegLoc = findReg(cReg.identifiers[0]);
  if (cRegLoc != -1) {
   ClassicalRegister cRegVal = allRegisters_[cRegLoc].getClassicalRegister();
   cRegVal.setValue(cReg.locations[0], sv_->getVal(state_, SVPair(qReg.identifiers[0], qReg.locations[0])));
   Register cRegFin = allRegisters_[cRegLoc];
   cRegFin.setClassicalRegister(cRegVal);
   allRegisters_[cRegLoc] = cRegFin;
  }
 }
}

void StateVectorMeasurement::printClassicalRegisters()
{
 for (auto reg : allRegisters_) {
  if (!reg.isQuantum()) {
   ClassicalRegister cReg = reg.getClassicalRegister();
   std::cout << "Classical Register Identifier: " << cReg.getIdentifier() << std::endl;
   for (int i = 0; i < cReg.getWidth(); i++) {
    std::cout << "Location [" << i << "]: " << cReg.getValue(i) << std::endl;
   }
  }
 }
}

std::vector<Register> StateVectorMeasurement::getAllRegisters()
{
 return allRegisters_;
}

StateVector* StateVectorMeasurement::getStateVector()
{
 return sv_;
}
```

Listing B.20: Measurement.cpp: File provides implementation for Quantum state measurement

```
#pragma once
#include "BaseTypes.h"
#include <map>
#include <functional>

GateRequestType getGateTypeS(std::string gateType);
GateRequestType getGateTypeM(std::string gateType);
GateRequest compileGateRequest(std::string gateType, idLocationPairs idLoc);
GateRequest compileGateRequest(std::string gateType, std::vector<double> params, idLocationPairs idLoc);
std::vector<GateRequest> compileCompoundGateRequest(std::string gateType, idLocationPairs idLoc);
std::vector<GateRequest> compileCompoundGateRequest(std::string gateType, std::vector<double> params, idLocationPairs idLoc);
std::function <std::vector<GateRequest>(std::vector<double> params, idLocationPairs idLoc)> compileCustomGateInternal(std::vector<std::string> gates, std::vector<std::vector<doubleOrArg>> paramsForGate, std::vector<std::vector<int>> locationsPerGate);
```

```
std::function <std::vector<GateRequest>(std::vector<double> params, idLocationPairs idLoc)> compileCustomGate(gateDeclaration decl, std::vector<gateOp> gateOperations);
```

Listing B.21: ParsingGateUtilities.h: File provides interface for functions needed for parsing compound and custom gates.

```
#include "ParsingGateUtilities.h"
#include <functional>

const double PI = 3.1415926535;

/*
    ParsingGateUtilities.h
    Description: File provides interface for functions needed for parsing compound and
    custom gates.

*/

// attachGateRequests is a utility function which appends a GateRequest vector to another GateRequest vector.
std::vector<GateRequest> attachGateRequests(std::vector<GateRequest> initial, std::vector<GateRequest> addition) {
    for (int i = 0; i < addition.size(); i++) {
        initial.push_back(addition[i]);
    }
    return initial;
}

// fetchIDLoc selects a specific elements of a compound idLocationPairs
idLocationPairs fetchIDLoc(idLocationPairs input, int i) {
    if (i >= input.getSize()) {
        return input;
    }
    idLocationPairs x;
    x.identifiers.push_back(input.identifiers[i]);
    x.locations.push_back(input.locations[i]);
    return x;
}

// zipIDLoc can combine the contents of two sets of idLocationPairs
idLocationPairs zipIDLoc(idLocationPairs inp1, idLocationPairs inp2) {
    idLocationPairs x;
    x.identifiers = inp1.identifiers;
    x.locations = inp1.locations;

    for (int i = 0; i < inp2.getSize(); i++) {
        x.identifiers.push_back(inp2.identifiers[i]);
        x.locations.push_back(inp2.locations[i]);
    }
    return x;
}

// getGateTypeS returns the primitive gate type for a given gate
GateRequestType getGateTypeS(std::string gateType) {
    GateRequestType gtType;
    if (gateType == "U") {
        gtType = U;
```

```
    }
    else if (gateType == "CX") {
        gtType = CX;
    }
    else if (gateType == "h") {
        gtType = h;
    }
    else if (gateType == "cx") {
        gtType = cx;
    }
    else {
        gtType = CUSTOM;
    }
    return gtType;
}

std::map<std::string, GateRequestType> mapOfGateRequests = {
    {"U", U},
    {"CX", CX},
    {"h", h},
    {"cx", cx},
    {"u3", u3},
    {"u2", u2},
    {"u1", u1},
    {"id", id},
    {"u0", u0},
    {"u", u},
    {"p", p},
    {"x", x},
    {"y", y},
    {"z", z},
    {"s", s},
    {"sdg", sdg},
    {"t", t},
    {"tdg", tdg},
    {"rx", rx},
    {"ry", ry},
    {"rz", rz},
    {"sx", sx},
    {"sxdg", sxdg},
    {"cz", cz},
    {"cy", cy},
    {"swap", swap},
    {"ch", ch},
    {"ccx", ccx},
    {"cswap", cswap},
    {"crx", crx},
    {"cry", cry},
    {"crz", crz},
    {"cu1", cu1},
    {"cp", cp},
    {"cu3", cu3},
    {"csx", csx},
    {"cu", cu},
    {"rxx", rxx},
    {"rzz", rzz},
    {"rccx", rccx},
    {"rc3x", rc3x},
```

```
    {"c3x", c3x},
    {"c3sqrtx", c3sqrtx},
    {"c4x", c4x}
};

// gateGateTypeM returns the compound gate type for any given gate
GateRequestType getGateTypeM(std::string gateType) {
    std::map<std::string, GateRequestType>::iterator it = mapOfGateRequests.find(gateType);
    if (it != mapOfGateRequests.end()) {
        return it->second;
    }
    return CUSTOM;
}

// attachIDLocPairs attaches pairs onto a gate request
GateRequest attachIDLocPairs(GateRequest input, idLocationPairs pairs) {
    for (int i = 0; i < pairs.identifiers.size(); i++) {
        input.addressQubit(pairs.identifiers[i], pairs.locations[i]);
    }
    return input;
}

// Hardware Primitive gates //

// compileU3Gate compiles a parameterised U gate with 3 parameters
std::vector<GateRequest> compileU3Gate(double theta, double phi, double lambda, idLocationPairs idLoc) {
    GateRequest gate(U);
    gate.addParameter(theta);
    gate.addParameter(phi);
    gate.addParameter(lambda);
    if (idLoc.getSize() != 1) {
        return std::vector<GateRequest>();
    }
    gate = attachIDLocPairs(gate, idLoc);
    std::vector<GateRequest> output {gate};
    return output;
}

// compileU2Gate compiles a parameterised U gate with 2 parameters
std::vector<GateRequest> compileU2Gate(double phi, double lambda, idLocationPairs idLoc) {
    return compileU3Gate(PI / 2.0, phi, lambda, idLoc);
}

// compileU1Gate compiles a parameterised U gate with 1 parameter
std::vector<GateRequest> compileU1Gate(double lambda, idLocationPairs idLoc) {
    return compileU3Gate(0, 0, lambda, idLoc);
}

// compileCXGate compiles a primitive CX gate
std::vector<GateRequest> compileCXGate(idLocationPairs idLoc) {
    GateRequest gate(CX);
    if (idLoc.getSize() != 2) {
        return std::vector<GateRequest>();
```

```
    }
    gate = attachIDLocPairs(gate, idLoc);
    std::vector<GateRequest> output{ gate };
    return output;
}

// compileIDGate compiles Identity gate
std::vector<GateRequest> compileIDGate(idLocationPairs idLoc) {
    return compileU3Gate(0, 0, 0, idLoc);
}

// compileU0Gate compiles Identity gate
std::vector<GateRequest> compileU0Gate(double gamma, idLocationPairs idLoc) {
    return compileU3Gate(0, 0, 0, idLoc);
}

// Standard Gates //

std::vector<GateRequest> compileUGate(double theta, double phi, double lambda, idLocationPairs idLoc) {
    return compileU3Gate(theta, phi, lambda, idLoc);
}

std::vector<GateRequest> compilePGate(double lambda, idLocationPairs idLoc) {
    return compileU3Gate(0, 0, lambda, idLoc);
}

std::vector<GateRequest> compileXGate(idLocationPairs idLoc) {
    return compileU3Gate(PI, 0, PI, idLoc);
}

std::vector<GateRequest> compileYGate(idLocationPairs idLoc) {
    return compileU3Gate(PI, PI / 2, PI / 2, idLoc);
}

std::vector<GateRequest> compileZGate(idLocationPairs idLoc) {
    return compileU1Gate(PI, idLoc);
}

std::vector<GateRequest> compileHGate(idLocationPairs idLoc) {
    return compileU2Gate(0, PI, idLoc);
}

std::vector<GateRequest> compileSGate(idLocationPairs idLoc) {
    return compileU1Gate(PI / 2, idLoc);
}

std::vector<GateRequest> compileSDGGate(idLocationPairs idLoc) {
    return compileU1Gate(-PI / 2, idLoc);
}

std::vector<GateRequest> compileTGate(idLocationPairs idLoc) {
    return compileU1Gate(PI / 4, idLoc);
}

std::vector<GateRequest> compileTDGGate(idLocationPairs idLoc) {
```

```cpp
    return compileU1Gate(-PI / 4, idLoc);
}

// Standard Rotations //
std::vector<GateRequest> compileRXGate(double theta, idLocationPairs
    ↪ idLoc) {
    return compileU3Gate(theta, -PI / 2, PI / 2, idLoc);
}

std::vector<GateRequest> compileRYGate(double theta, idLocationPairs
    ↪ idLoc) {
    return compileU3Gate(theta, 0, 0, idLoc);
}

std::vector<GateRequest> compileRZGate(double phi, idLocationPairs
    ↪ idLoc) {
    return compileU1Gate(-PI / 4, idLoc);
}

// Standard User Defined Gates //
std::vector<GateRequest> compileSXGate(idLocationPairs idLoc) {
    std::vector<GateRequest> req = compileSDGGate(idLoc);
    req = attachGateRequests(req, compileHGate(idLoc));
    req = attachGateRequests(req, compileSDGGate(idLoc));
    return req;
}

std::vector<GateRequest> compileSXDGGate(idLocationPairs idLoc) {
    std::vector<GateRequest> req = compileSGate(idLoc);
    req = attachGateRequests(req, compileHGate(idLoc));
    req = attachGateRequests(req, compileSGate(idLoc));
    return req;
}

std::vector<GateRequest> compileCZGate(idLocationPairs idLoc) {
    if (idLoc.getSize() != 2) {
        return std::vector<GateRequest>();
    }
    idLocationPairs pairB;
    pairB.identifiers.push_back(idLoc.identifiers[1]);
    pairB.locations.push_back(idLoc.locations[1]);
    std::vector<GateRequest> req = compileHGate(pairB);
    req = attachGateRequests(req, compileCXGate(idLoc));
    req = attachGateRequests(req, compileHGate(pairB));
    return req;
}

std::vector<GateRequest> compileCYGate(idLocationPairs idLoc) {
    if (idLoc.getSize() != 2) {
        return std::vector<GateRequest>();
    }
    idLocationPairs pairB;
    pairB.identifiers.push_back(idLoc.identifiers[1]);
    pairB.locations.push_back(idLoc.locations[1]);
    std::vector<GateRequest> req = compileSDGGate(pairB);
    req = attachGateRequests(req, compileCXGate(idLoc));
    req = attachGateRequests(req, compileSGate(pairB));
    return req;
```

```
}

std::vector<GateRequest> compileSwapGate(idLocationPairs idLoc) {
    if (idLoc.getSize() != 2) {
        return std::vector<GateRequest>();
    }
    idLocationPairs swapped;
    swapped.identifiers.push_back(idLoc.identifiers[1]);
    swapped.identifiers.push_back(idLoc.identifiers[0]);
    swapped.locations.push_back(idLoc.locations[1]);
    swapped.locations.push_back(idLoc.locations[0]);

    std::vector<GateRequest> req = compileCXGate(idLoc);
    req = attachGateRequests(req, compileCXGate(swapped));
    req = attachGateRequests(req, compileCXGate(idLoc));
    return req;
}

std::vector<GateRequest> compileCHGate(idLocationPairs idLoc) {
    if (idLoc.getSize() != 2) {
        return std::vector<GateRequest>();
    }
    idLocationPairs pairA;
    pairA.identifiers.push_back(idLoc.identifiers[0]);
    pairA.locations.push_back(idLoc.locations[0]);
    idLocationPairs pairB;
    pairB.identifiers.push_back(idLoc.identifiers[1]);
    pairB.locations.push_back(idLoc.locations[1]);

    std::vector<GateRequest> req = compileHGate(pairB);
    req = attachGateRequests(req, compileSDGGate(pairB));
    req = attachGateRequests(req, compileCXGate(idLoc));
    req = attachGateRequests(req, compileHGate(pairB));
    req = attachGateRequests(req, compileTGate(pairB));
    req = attachGateRequests(req, compileCXGate(idLoc));
    req = attachGateRequests(req, compileTGate(pairB));
    req = attachGateRequests(req, compileHGate(pairB));
    req = attachGateRequests(req, compileSGate(pairB));
    req = attachGateRequests(req, compileXGate(pairB));
    req = attachGateRequests(req, compileSGate(pairA));
}

std::vector<GateRequest> compileCCXGate(idLocationPairs idLoc) {
    if (idLoc.getSize() != 3) {
        return std::vector<GateRequest>();
    }

    idLocationPairs pairA = fetchIDLoc(idLoc, 0);
    idLocationPairs pairB = fetchIDLoc(idLoc, 1);
    idLocationPairs pairC = fetchIDLoc(idLoc, 2);

    std::vector<GateRequest> req = compileHGate(pairC);
    req = attachGateRequests(req, compileCXGate(zipIDLoc(pairB, pairC))
        ↪ );
    req = attachGateRequests(req, compileTDGGate(pairC));
    req = attachGateRequests(req, compileCXGate(zipIDLoc(pairA, pairC))
        ↪ );
    req = attachGateRequests(req, compileTGate(pairC));
```

```
    req = attachGateRequests(req, compileCXGate(zipIDLoc(pairB, pairC))
        ↪ );
    req = attachGateRequests(req, compileTDGGate(pairC));
    req = attachGateRequests(req, compileCXGate(zipIDLoc(pairA, pairC))
        ↪ );
    req = attachGateRequests(req, compileTGate(pairB));
    req = attachGateRequests(req, compileTGate(pairC));
    req = attachGateRequests(req, compileHGate(pairC));
    req = attachGateRequests(req, compileCXGate(zipIDLoc(pairA, pairB))
        ↪ );
    req = attachGateRequests(req, compileTGate(pairA));
    req = attachGateRequests(req, compileTDGGate(pairB));
    req = attachGateRequests(req, compileCXGate(zipIDLoc(pairA, pairB))
        ↪ );
}

std::vector<GateRequest> compileCSwapGate(idLocationPairs idLoc) {
    if (idLoc.getSize() != 3) {
        return std::vector<GateRequest>();
    }
    idLocationPairs pairC = fetchIDLoc(idLoc, 2);
    idLocationPairs pairB = fetchIDLoc(idLoc, 1);
    std::vector<GateRequest> req = compileCXGate(zipIDLoc(pairC, pairB)
        ↪ );
    req = attachGateRequests(req, compileCCXGate(idLoc));
    req = attachGateRequests(req, compileCXGate(zipIDLoc(pairC, pairB))
        ↪ );
    return req;
}

std::vector<GateRequest> compileCRXGate(double lambda, idLocationPairs
    ↪ idLoc) {
    if (idLoc.getSize() != 2) {
        return std::vector<GateRequest>();
    }
    idLocationPairs pairA = fetchIDLoc(idLoc, 0);
    idLocationPairs pairB = fetchIDLoc(idLoc, 1);

    std::vector<GateRequest> req = compileU1Gate(PI / 2, pairB);
    req = attachGateRequests(req, compileCXGate(idLoc));
    req = attachGateRequests(req, compileU3Gate(-lambda / 2, 0, 0,
        ↪ pairB));
    req = attachGateRequests(req, compileCXGate(idLoc));
    req = attachGateRequests(req, compileU3Gate(lambda / 2, -PI / 2, 0,
        ↪  pairB));
    return req;
}

std::vector<GateRequest> compileCRYGate(double lambda, idLocationPairs
    ↪ idLoc) {
    if (idLoc.getSize() != 2) {
        return std::vector<GateRequest>();
    }
    idLocationPairs pairA = fetchIDLoc(idLoc, 0);
    idLocationPairs pairB = fetchIDLoc(idLoc, 1);

    std::vector<GateRequest> req = compileRYGate(lambda / 2, pairB);
    req = attachGateRequests(req, compileCXGate(idLoc));
```

```
    req = attachGateRequests(req, compileRYGate(-lambda / 2, pairB));
    req = attachGateRequests(req, compileCXGate(idLoc));
    return req;
}

std::vector<GateRequest> compileCRZGate(double lambda, idLocationPairs idLoc) {
    if (idLoc.getSize() != 2) {
        return std::vector<GateRequest>();
    }
    idLocationPairs pairA = fetchIDLoc(idLoc, 0);
    idLocationPairs pairB = fetchIDLoc(idLoc, 1);
    std::vector<GateRequest> req = compileRZGate(lambda / 2, pairB);
    req = attachGateRequests(req, compileCXGate(idLoc));
    req = attachGateRequests(req, compileRZGate(-lambda / 2, pairB));
    req = attachGateRequests(req, compileCXGate(idLoc));
    return req;
}

std::vector<GateRequest> compileCU1Gate(double lambda, idLocationPairs idLoc) {
    if (idLoc.getSize() != 2) {
        return std::vector<GateRequest>();
    }
    idLocationPairs pairA = fetchIDLoc(idLoc, 0);
    idLocationPairs pairB = fetchIDLoc(idLoc, 1);

    std::vector<GateRequest> req = compileU1Gate(lambda / 2, pairA);
    req = attachGateRequests(req, compileCXGate(idLoc));
    req = attachGateRequests(req, compileU1Gate(-lambda / 2, pairB));
    req = attachGateRequests(req, compileCXGate(idLoc));
    req = attachGateRequests(req, compileU1Gate(lambda / 2, pairB));

    return req;
}

std::vector<GateRequest> compileCPGate(double lambda, idLocationPairs idLoc) {
    if (idLoc.getSize() != 2) {
        return std::vector<GateRequest>();
    }
    idLocationPairs pairA = fetchIDLoc(idLoc, 0);
    idLocationPairs pairB = fetchIDLoc(idLoc, 1);

    std::vector<GateRequest> req = compilePGate(lambda / 2, pairA);
    req = attachGateRequests(req, compileCXGate(idLoc));
    req = attachGateRequests(req, compilePGate(-lambda / 2, pairB));
    req = attachGateRequests(req, compileCXGate(idLoc));
    req = attachGateRequests(req, compilePGate(lambda / 2, pairB));
    return req;
}

std::vector<GateRequest> compileCU3Gate(double theta, double phi, double lambda, idLocationPairs idLoc) {
    if (idLoc.getSize() != 2) {
        return std::vector<GateRequest>();
    }
    idLocationPairs pairA = fetchIDLoc(idLoc, 0);
```

```
    idLocationPairs pairB = fetchIDLoc(idLoc, 1);

    std::vector<GateRequest> req = compileU1Gate((lambda + phi) / 2,
       ↪ pairA);
    req = attachGateRequests(req, compileU1Gate((lambda - phi) / 2,
       ↪ pairB));
    req = attachGateRequests(req, compileCXGate(idLoc));
    req = attachGateRequests(req, compileU3Gate(-theta / 2, 0, -(phi +
       ↪ lambda) / 2, pairB));
    req = attachGateRequests(req, compileCXGate(idLoc));
    req = attachGateRequests(req, compileU3Gate(theta / 2, phi, 0,
       ↪ pairB));
    return req;
}

std::vector<GateRequest> compileCSXGate(idLocationPairs idLoc) {
    if (idLoc.getSize() != 2) {
        return std::vector<GateRequest>();
    }
    idLocationPairs pairA = fetchIDLoc(idLoc, 0);
    idLocationPairs pairB = fetchIDLoc(idLoc, 1);

    std::vector<GateRequest> req = compileHGate(pairB);
    req = attachGateRequests(req, compileCU1Gate(PI / 2, idLoc));
    req = attachGateRequests(req, compileHGate(pairB));
    return req;
}

std::vector<GateRequest> compileCUGate(double theta, double phi, double
   ↪ lambda, double gamma, idLocationPairs idLoc) {
    if (idLoc.getSize() != 2) {
        return std::vector<GateRequest>();
    }
    idLocationPairs pairA = fetchIDLoc(idLoc, 0);
    idLocationPairs pairB = fetchIDLoc(idLoc, 1);
    std::vector<GateRequest> req = compilePGate(gamma, pairA);
    req = attachGateRequests(req, compilePGate((lambda + phi) / 2,
       ↪ pairA));
    req = attachGateRequests(req, compilePGate((lambda - phi) / 2,
       ↪ pairB));
    req = attachGateRequests(req, compileCXGate(idLoc));
    req = attachGateRequests(req, compileUGate(-theta / 2, 0, -(phi +
       ↪ lambda) / 2, pairB));
    req = attachGateRequests(req, compileCXGate(idLoc));
    req = attachGateRequests(req, compileUGate(theta / 2, phi, 0, pairB
       ↪ ));
    return req;
}

std::vector<GateRequest> compileRXXGate(double theta, idLocationPairs
   ↪ idLoc) {
    if (idLoc.getSize() != 2) {
        return std::vector<GateRequest>();
    }
    idLocationPairs pairA = fetchIDLoc(idLoc, 0);
    idLocationPairs pairB = fetchIDLoc(idLoc, 1);

    std::vector<GateRequest> req = compileU3Gate(PI / 2, theta, 0,
```

```
        ↪ pairA);
    req = attachGateRequests(req, compileHGate(pairB));
    req = attachGateRequests(req, compileCXGate(idLoc));
    req = attachGateRequests(req, compileU1Gate(-theta, pairB));
    req = attachGateRequests(req, compileCXGate(idLoc));
    req = attachGateRequests(req, compileHGate(pairB));
    req = attachGateRequests(req, compileU2Gate(-PI,PI - theta, pairA))
        ↪ ;
    return req;
}

std::vector<GateRequest> compileRZZGate(double theta, idLocationPairs
    ↪ idLoc) {
    if (idLoc.getSize() != 2) {
        return std::vector<GateRequest>();
    }
    idLocationPairs pairA = fetchIDLoc(idLoc, 0);
    idLocationPairs pairB = fetchIDLoc(idLoc, 1);

    std::vector<GateRequest> req = compileCXGate(idLoc);
    req = attachGateRequests(req, compileU1Gate(theta, pairB));
    req = attachGateRequests(req, compileCXGate(idLoc));
    return req;
}

std::vector<GateRequest> compileRCCXGate(idLocationPairs idLoc) {
    if (idLoc.getSize() != 3) {
        return std::vector<GateRequest>();
    }
    idLocationPairs pairA = fetchIDLoc(idLoc, 0);
    idLocationPairs pairB = fetchIDLoc(idLoc, 1);
    idLocationPairs pairC = fetchIDLoc(idLoc, 2);

    std::vector<GateRequest> req = compileU2Gate(0, PI, pairC);
    req = attachGateRequests(req, compileU1Gate(PI / 4, pairC));
    req = attachGateRequests(req, compileCXGate(zipIDLoc(pairB, pairC))
        ↪ );
    req = attachGateRequests(req, compileU1Gate(-PI / 4, pairC));
    req = attachGateRequests(req, compileCXGate(zipIDLoc(pairA, pairC))
        ↪ );
    req = attachGateRequests(req, compileU1Gate(PI / 4, pairC));
    req = attachGateRequests(req, compileCXGate(zipIDLoc(pairB, pairC))
        ↪ );
    req = attachGateRequests(req, compileU1Gate(-PI / 4, pairC));
    req = attachGateRequests(req, compileU2Gate(0, PI, pairC));
    return req;
}

std::vector<GateRequest> compileRC3XGate(idLocationPairs idLoc) {
    if (idLoc.getSize() != 4) {
        return std::vector<GateRequest>();
    }
    idLocationPairs pairA = fetchIDLoc(idLoc, 0);
    idLocationPairs pairB = fetchIDLoc(idLoc, 1);
    idLocationPairs pairC = fetchIDLoc(idLoc, 2);
    idLocationPairs pairD = fetchIDLoc(idLoc, 3);

    std::vector<GateRequest> req = compileU2Gate(0, PI, pairD);
```

```
    req = attachGateRequests(req, compileU1Gate(PI / 4, pairD));
    req = attachGateRequests(req, compileCXGate(zipIDLoc(pairC, pairD))
        ↪ );
    req = attachGateRequests(req, compileU1Gate(-PI / 4, pairD));
    req = attachGateRequests(req, compileU2Gate(0, PI, pairD));
    req = attachGateRequests(req, compileCXGate(zipIDLoc(pairA, pairD))
        ↪ );
    req = attachGateRequests(req, compileU1Gate(PI / 4, pairD));
    req = attachGateRequests(req, compileCXGate(zipIDLoc(pairB, pairD))
        ↪ );
    req = attachGateRequests(req, compileU1Gate(-PI / 4, pairD));
    req = attachGateRequests(req, compileCXGate(zipIDLoc(pairA, pairD))
        ↪ );
    req = attachGateRequests(req, compileU1Gate(PI / 4, pairD));
    req = attachGateRequests(req, compileCXGate(zipIDLoc(pairB, pairD))
        ↪ );
    req = attachGateRequests(req, compileU1Gate(-PI / 4, pairD));
    req = attachGateRequests(req, compileU2Gate(0, PI, pairD));
    req = attachGateRequests(req, compileU1Gate(PI / 4, pairD));
    req = attachGateRequests(req, compileCXGate(zipIDLoc(pairC, pairD))
        ↪ );
    req = attachGateRequests(req, compileU1Gate(-PI / 4, pairD));
    req = attachGateRequests(req, compileU2Gate(0, PI, pairD));
    return req;
}

std::vector<GateRequest> compileC3XGate(idLocationPairs idLoc) {
    if (idLoc.getSize() != 4) {
        return std::vector<GateRequest>();
    }
    idLocationPairs pairA = fetchIDLoc(idLoc, 0);
    idLocationPairs pairB = fetchIDLoc(idLoc, 1);
    idLocationPairs pairC = fetchIDLoc(idLoc, 2);
    idLocationPairs pairD = fetchIDLoc(idLoc, 3);

    std::vector<GateRequest> req = compileHGate(pairD);
    req = attachGateRequests(req, compilePGate(PI / 8, pairA));
    req = attachGateRequests(req, compilePGate(PI / 8, pairB));
    req = attachGateRequests(req, compilePGate(PI / 8, pairC));
    req = attachGateRequests(req, compilePGate(PI / 8, pairD));
    req = attachGateRequests(req, compileCXGate(zipIDLoc(pairA, pairB))
        ↪ );
    req = attachGateRequests(req, compilePGate(-PI / 8, pairB));
    req = attachGateRequests(req, compileCXGate(zipIDLoc(pairA, pairB))
        ↪ );
    req = attachGateRequests(req, compileCXGate(zipIDLoc(pairB, pairC))
        ↪ );
    req = attachGateRequests(req, compilePGate(-PI / 8, pairC));
    req = attachGateRequests(req, compileCXGate(zipIDLoc(pairA, pairC))
        ↪ );
    req = attachGateRequests(req, compilePGate(PI / 8, pairC));
    req = attachGateRequests(req, compileCXGate(zipIDLoc(pairB, pairC))
        ↪ );
    req = attachGateRequests(req, compilePGate(-PI / 8, pairC));
    req = attachGateRequests(req, compileCXGate(zipIDLoc(pairA, pairC))
        ↪ );
    req = attachGateRequests(req, compileCXGate(zipIDLoc(pairC, pairD))
        ↪ );
```

```
    req = attachGateRequests(req, compilePGate(−PI / 8, pairD));
    req = attachGateRequests(req, compileCXGate(zipIDLoc(pairB, pairD))
        ↪ );
    req = attachGateRequests(req, compilePGate(PI / 8, pairD));
    req = attachGateRequests(req, compileCXGate(zipIDLoc(pairC, pairD))
        ↪ );
    req = attachGateRequests(req, compilePGate(−PI / 8, pairD));
    req = attachGateRequests(req, compileCXGate(zipIDLoc(pairA, pairD))
        ↪ );
    req = attachGateRequests(req, compilePGate(PI / 8, pairD));
    req = attachGateRequests(req, compileCXGate(zipIDLoc(pairC, pairD))
        ↪ );
    req = attachGateRequests(req, compilePGate(−PI / 8, pairD));
    req = attachGateRequests(req, compileCXGate(zipIDLoc(pairB, pairD))
        ↪ );
    req = attachGateRequests(req, compilePGate(PI / 8, pairD));
    req = attachGateRequests(req, compileCXGate(zipIDLoc(pairC, pairD))
        ↪ );
    req = attachGateRequests(req, compilePGate(−PI / 8, pairD));
    req = attachGateRequests(req, compileCXGate(zipIDLoc(pairA, pairD))
        ↪ );
    req = attachGateRequests(req, compileHGate(pairD));
    return req;
}

std::vector<GateRequest> compileC3SQRTGate(idLocationPairs idLoc) {
    if (idLoc.getSize() != 4) {
        return std::vector<GateRequest>();
    }
    idLocationPairs pairA = fetchIDLoc(idLoc, 0);
    idLocationPairs pairB = fetchIDLoc(idLoc, 1);
    idLocationPairs pairC = fetchIDLoc(idLoc, 2);
    idLocationPairs pairD = fetchIDLoc(idLoc, 3);

    std::vector<GateRequest> req = compileHGate(pairD);
    req = attachGateRequests(req, compileCU1Gate(−PI / 8, zipIDLoc(
        ↪ pairA, pairD)));
    req = attachGateRequests(req, compileHGate(pairD));
    req = attachGateRequests(req, compileCXGate(zipIDLoc(pairA, pairB))
        ↪ );
    req = attachGateRequests(req, compileHGate(pairD));
    req = attachGateRequests(req, compileCU1Gate(PI / 8, zipIDLoc(pairB
        ↪ , pairD)));
    req = attachGateRequests(req, compileHGate(pairD));
    req = attachGateRequests(req, compileCXGate(zipIDLoc(pairA, pairB))
        ↪ );
    req = attachGateRequests(req, compileHGate(pairD));
    req = attachGateRequests(req, compileCU1Gate(−PI / 8, zipIDLoc(
        ↪ pairB, pairD)));
    req = attachGateRequests(req, compileHGate(pairD));
    req = attachGateRequests(req, compileCXGate(zipIDLoc(pairB, pairC))
        ↪ );
    req = attachGateRequests(req, compileHGate(pairD));
    req = attachGateRequests(req, compileCU1Gate(PI / 8, zipIDLoc(pairC
        ↪ , pairD)));
    req = attachGateRequests(req, compileHGate(pairD));
    req = attachGateRequests(req, compileCXGate(zipIDLoc(pairA, pairC))
        ↪ );
```

```
    req = attachGateRequests(req, compileHGate(pairD));
    req = attachGateRequests(req, compileCU1Gate(-PI / 8, zipIDLoc(pairC, pairD)));
    req = attachGateRequests(req, compileHGate(pairD));
    req = attachGateRequests(req, compileCXGate(zipIDLoc(pairB, pairC)));
    req = attachGateRequests(req, compileHGate(pairD));
    req = attachGateRequests(req, compileCU1Gate(PI / 8, zipIDLoc(pairC, pairD)));
    req = attachGateRequests(req, compileHGate(pairD));
    req = attachGateRequests(req, compileCXGate(zipIDLoc(pairA, pairC)));
    req = attachGateRequests(req, compileHGate(pairD));
    req = attachGateRequests(req, compileCU1Gate(-PI / 8, zipIDLoc(pairC, pairD)));
    req = attachGateRequests(req, compileHGate(pairD));
    return req;
}

std::vector<GateRequest> compileC4XGate(idLocationPairs idLoc) {
    if (idLoc.getSize() != 5) {
        return std::vector<GateRequest>();
    }
    idLocationPairs pairA = fetchIDLoc(idLoc, 0);
    idLocationPairs pairB = fetchIDLoc(idLoc, 1);
    idLocationPairs pairC = fetchIDLoc(idLoc, 2);
    idLocationPairs pairD = fetchIDLoc(idLoc, 3);
    idLocationPairs pairE = fetchIDLoc(idLoc, 4);

    std::vector<GateRequest> req = compileHGate(pairE);
    req = attachGateRequests(req, compileCU1Gate(-PI / 2, zipIDLoc(pairD, pairE)));
    req = attachGateRequests(req, compileHGate(pairE));
    req = attachGateRequests(req, compileC3XGate(zipIDLoc(zipIDLoc(pairA, pairB), zipIDLoc(pairC, pairD))));
    req = attachGateRequests(req, compileHGate(pairE));
    req = attachGateRequests(req, compileCU1Gate(PI / 2, zipIDLoc(pairD, pairE)));
    req = attachGateRequests(req, compileHGate(pairE));
    req = attachGateRequests(req, compileC3XGate(zipIDLoc(zipIDLoc(pairA, pairB), zipIDLoc(pairC, pairD))));
    req = attachGateRequests(req, compileC3SQRTGate(zipIDLoc(zipIDLoc(pairA, pairB), zipIDLoc(pairC, pairE))));
    return req;

}

// compileGateRequest compiles a primitive gate request, as per user input
GateRequest compileGateRequest(std::string gateType, idLocationPairs idLoc) {
    //GateRequestType gtType = getGateTypeS(gateType);
    GateRequestType gtType = getGateTypeM(gateType);
    GateRequest gate(gtType);
    for (int i = 0; i < idLoc.identifiers.size(); i++) {
        gate.addressQubit(idLoc.identifiers[i], idLoc.locations[i]);
    }
    return gate;
```

```
}

// compileGateRequest compiles a primitive gate request, as per user
    ↪ input
GateRequest compileGateRequest(std::string gateType, std::vector<double
    ↪ > params, idLocationPairs idLoc) {
    //GateRequestType gtType = getGateTypeS(gateType);
    GateRequestType gtType = getGateTypeM(gateType);
    GateRequest gate(gtType);
    for (int i = 0; i < idLoc.identifiers.size(); i++) {
        gate.addressQubit(idLoc.identifiers[i], idLoc.locations[i]);
    }
    gate.setParameters(params);
    return gate;
}

// compileCompoundGateRequest compiles a qelib1 gate, as per user input
std::vector<GateRequest> compileCompoundGateRequest(std::string
    ↪ gateType, idLocationPairs idLoc)
{
    GateRequestType gtType = getGateTypeM(gateType);
    switch (gtType) {
    case CX:
        return compileCXGate(idLoc);
    case cx:
        return compileCXGate(idLoc);
    case id:
        return compileIDGate(idLoc);
    case x:
        return compileXGate(idLoc);
    case y:
        return compileYGate(idLoc);
    case z:
        return compileZGate(idLoc);
    case h:
        return compileHGate(idLoc);
    case s:
        return compileSGate(idLoc);
    case sdg:
        return compileSDGGate(idLoc);
    case t:
        return compileTGate(idLoc);
    case tdg:
        return compileTDGGate(idLoc);
    case sx:
        return compileSXGate(idLoc);
    case sxdg:
        return compileSXDGGate(idLoc);
    case cz:
        return compileCZGate(idLoc);
    case cy:
        return compileCYGate(idLoc);
    case swap:
        return compileSwapGate(idLoc);
    case ch:
        return compileCHGate(idLoc);
    case ccx:
        return compileCCXGate(idLoc);
```

```
    case cswap:
        return compileCSwapGate(idLoc);
    case csx:
        return compileCSXGate(idLoc);
    case rccx:
        return compileRCCXGate(idLoc);
    case rc3x:
        return compileRC3XGate(idLoc);
    case c3x:
        return compileC3XGate(idLoc);
    case c3sqrtx:
        return compileC3SQRTGate(idLoc);
    case c4x:
        return compileC4XGate(idLoc);
    default:
        return std::vector<GateRequest>();
    }
    return std::vector<GateRequest>();
}
std::vector<GateRequest> compileCompoundGateRequest(std::string
    ↪ gateType, std::vector<double> params, idLocationPairs idLoc)
{
    GateRequestType gtType = getGateTypeM(gateType);
    switch (gtType) {
    case U:
        if (params.size() == 3) {
            return compileU3Gate(params[0], params[1], params[2], idLoc
                ↪ );
        }
        break;
    case u3:
        if (params.size() == 3) {
            return compileU3Gate(params[0], params[1], params[2], idLoc
                ↪ );
        }
        break;
    case u2:
        if (params.size() == 2) {
            return compileU2Gate(params[0], params[1], idLoc);
        }
        break;
    case u1:
        if (params.size() == 1) {
            return compileU1Gate(params[0], idLoc);
        }
        break;
    case u0:
        if (params.size() == 1) {
            return compileU0Gate(params[0], idLoc);
        }
        break;
    case u:
        if (params.size() == 3) {
            return compileUGate(params[0], params[1], params[2], idLoc)
                ↪ ;
        }
        break;
    case p:
```

```
            if (params.size() == 1) {
                return compilePGate(params[0], idLoc);
            }
            break;
        case rx:
            if (params.size() == 1) {
                return compileRXGate(params[0], idLoc);
            }
            break;
        case ry:
            if (params.size() == 1) {
                return compileRYGate(params[0], idLoc);
            }
            break;
        case rz:
            if (params.size() == 1) {
                return compileRZGate(params[0], idLoc);
            }
            break;
        case crx:
            if (params.size() == 1) {
                return compileCRXGate(params[0], idLoc);
            }
            break;
        case cry:
            if (params.size() == 1) {
                return compileCRYGate(params[0], idLoc);
            }
            break;
        case crz:
            if (params.size() == 1) {
                return compileCRZGate(params[0], idLoc);
            }
            break;
        case cu1:
            if (params.size() == 1) {
                return compileCU1Gate(params[0], idLoc);
            }
            break;
        case cp:
            if (params.size() == 1) {
                return compileCPGate(params[0], idLoc);
            }
            break;
        case cu3:
            if (params.size() == 3) {
                return compileCU3Gate(params[0], params[1], params[2],
                    ↪ idLoc);
            }
            break;
        case cu:
            if (params.size() == 4) {
                return compileCUGate(params[0], params[1], params[2],
                    ↪ params[3], idLoc);
            }
            break;
        case rxx:
            if (params.size() == 1) {
```

```
            return compileRXXGate(params[0], idLoc);
        }
        break;
    case rzz:
        if (params.size() == 1) {
            return compileRZZGate(params[0], idLoc);
        }
        break;
    default:
        return std::vector<GateRequest>();
    }
    return std::vector<GateRequest>();
}

int findMin(std::vector<int> values) {
    int minSoFar = INT_MAX;
    int loc = 0;
    for (int i = 0; i < values.size(); i++) {
        if (values[i] < minSoFar) {
            minSoFar = values[i];
            loc = i;
        }
    }
    return loc;
}

int findMax(std::vector<int> values) {
    int maxSoFar = 0;
    int loc = 0;
    for (int i = 0; i < values.size(); i++) {
        if (values[i] > maxSoFar) {
            maxSoFar = values[i];
            loc = i;
        }
    }
    return loc;
}

std::vector<int> findMinMax(std::vector<std::vector<int>> paramsForGate, std::vector<std::vector<int>> locationsPerGate) {
    std::vector<int> minForParams, maxForParams, minForLocations, maxForLocations, result;
    for (int i = 0; i < paramsForGate.size(); i++) {
        minForParams.push_back(paramsForGate[i][findMin(paramsForGate[i])]);
        maxForParams.push_back(paramsForGate[i][findMax(paramsForGate[i])]);
    }
    for (int i = 0; i < locationsPerGate.size(); i++) {
        minForLocations.push_back(locationsPerGate[i][findMin(locationsPerGate[i])]);
        maxForLocations.push_back(locationsPerGate[i][findMax(locationsPerGate[i])]);
    }
    if (paramsForGate.size() == 0) {
        minForParams.push_back(0);
        maxForParams.push_back(0);
    }
```

```
    result = { minForParams[findMin(minForParams)], maxForParams[
       ↪ findMax(maxForParams)], minForLocations[findMin(
       ↪ minForLocations)], maxForLocations[findMax(maxForLocations)
       ↪ ]};
    return result;
}

// gateCoupling used during custom gate compilation to hold information
   ↪ for full gate compilation when required
struct gateCoupling {
    std::string gateName;
    std::vector<doubleOrArg> paramLocations;
    std::vector<int> idLocations;
    gateCoupling(std::string name, std::vector<doubleOrArg> param, std
       ↪ ::vector<int> idLoc) {
        gateName = name;
        paramLocations = param;
        idLocations = idLoc;
    }
};

std::function <std::vector<GateRequest>(std::vector<double> params,
   ↪ idLocationPairs idLoc)> compileCustomGateInternal(std::vector<
   ↪ std::string> gates, std::vector<std::vector<doubleOrArg>>
   ↪ paramsForGate, std::vector<std::vector<int>> locationsPerGate) {
    std::vector<gateCoupling> couplings;
    for (int i = 0; i < gates.size(); i++) {
        couplings.push_back(gateCoupling(gates[i], paramsForGate[i],
           ↪ locationsPerGate[i]));
    }
    std::function <std::vector<GateRequest>(std::vector<double> params,
       ↪  idLocationPairs idLoc)> deltaFunc = [couplings](std::vector
       ↪ <double> params, idLocationPairs idLoc) {
        std::vector<GateRequest> requests;
        for (auto coupling : couplings) {
            std::vector<double> localParams;
            for (int i = 0; i < coupling.paramLocations.size(); i++) {
                doubleOrArg da = coupling.paramLocations[i];
                if (da.doubleNotArg) {
                    localParams.push_back(da.valD);
                }
                else {
                    localParams.push_back(params[da.position]);
                }
            }
            idLocationPairs localPairs;
            for (int i = 0; i < coupling.idLocations.size(); i++) {
                localPairs = zipIDLoc(localPairs, fetchIDLoc(idLoc,
                   ↪ coupling.idLocations[i]));
            }
            if (localParams.size() > 0) {
                requests = attachGateRequests(requests,
                   ↪ compileCompoundGateRequest(coupling.gateName,
                   ↪ localParams, localPairs));
            }
            else {
                requests = attachGateRequests(requests,
                   ↪ compileCompoundGateRequest(coupling.gateName,
```

```
                    ↪ localPairs));
            }
        }
        return requests;
    };
    return deltaFunc;
}

// compileCustomGate generates a template function which can generate a
    ↪ full set of GateRequests for a custom defined gate
std::function<std::vector<GateRequest>(std::vector<double>params,
    ↪ idLocationPairs idLoc)> compileCustomGate(gateDeclaration decl,
    ↪ std::vector<gateOp> gateOperations)
{
    std::map<std::string, int> paramToLocation, idLocToLocation;
    std::vector<std::string> params = decl.paramList;
    for (int i = 0; i < params.size(); i++) {
        paramToLocation[params[i]] = i;
    }
    std::vector<std::string> idLocs = decl.idLocList;
    for (int i = 0; i < idLocs.size(); i++) {
        idLocToLocation[idLocs[i]] = i;
    }
    std::vector<std::string> gates;
    std::vector<std::vector<doubleOrArg>> paramLocs;
    std::vector<std::vector<int>> idLocsI;
    for (auto gop : gateOperations) {
        gates.push_back(gop.gateName);
        std::vector<doubleOrArg> parmsLocal;
        for (int i = 0; i < gop.params.size(); i++) {
            if (gop.params[i].identNotVal) {
                doubleOrArg arg;
                arg.doubleNotArg = false;
                arg.position = paramToLocation[gop.params[i].ident];
                parmsLocal.push_back(arg);
            } else{
                doubleOrArg doub;
                doub.doubleNotArg = true;
                doub.valD = gop.params[i].value;
                parmsLocal.push_back(doub);
            }
        }
        std::vector<int> idLocal;
        for (int i = 0; i < gop.idLocs.size(); i++) {
            idLocal.push_back(idLocToLocation[gop.idLocs[i]]);
        }
        paramLocs.push_back(parmsLocal);
        idLocsI.push_back(idLocal);
    }
    return compileCustomGateInternal(gates, paramLocs, idLocsI);
}
```

Listing B.22: ParsingGateUtilities.cpp: File provides implementation for functions needed for parsing compound and custom gates.

```
#pragma once
#include <iostream>
#include <vector>
#include "BaseTypes.h"
```

```

/*
 Stager.h
 Description: File provides interface for functions needed for staging functionality

*/

class Stager {
private:
 std::vector<Register> registers_;
 std::vector<GateRequest> gates_;
 std::vector<ConcurrentBlock> blocks_;

 bool loadRegisters(std::vector<Register> registers) {
  registers_ = registers;
  return true;
 }

 bool loadGates(std::vector<GateRequest> requests) {
  gates_ = requests;
  return true;
 }

 // concurrencyCalc can calculate whether we need a break for concurrency
 //  then loads the appropriate gates into each block
 bool concurrencyCalc() {
  ConcurrentBlock block(0);
  for (auto gateR : gates_) {
   if (gateR.getGateDim() == 1) {
    block.addGate(gateR);
   }
   else {
    if (block.getCount() > 0) {
     blocks_.push_back(block);
    }
    ConcurrentBlock tempBlock(0);
    tempBlock.addGate(gateR);
    blocks_.push_back(tempBlock);
    block = ConcurrentBlock(0);
   }
  }
  if (block.getCount() > 0) {
   blocks_.push_back(block);
  }
  return true;
 }

public:
 Stager() {

 }
 Stager(std::vector<Register> registers, std::vector<GateRequest> requests) {
  loadRegisters(registers);
  loadGates(requests);
  concurrencyCalc();
```

```
  }
  std::vector<ConcurrentBlock> stageInformation(std::vector<Register> registers, std::vector<GateRequest> requests) {
    loadRegisters(registers);
    loadGates(requests);
    concurrencyCalc();
    return blocks_;
  }
  std::vector<ConcurrentBlock> getConcurrencyBlocks() {
    return blocks_;
  }

  std::vector<Register> getRegisters() {
    return registers_;
  }

};
```

Listing B.23: staging.h: File provides interface for functions needed for staging functionality.

```
#pragma once
#include "../libs/BaseTypes.h"

class ValkyrieTests {
private:
  int total_;
  int passed_;
  std::vector<std::string> failedTests;
  std::vector<std::string> passedTests;
  void runParserTests();
  void runStagingTests();
  void runCPUDeviceTests();
  void runGPUDeviceTests();
  void runMeasurementTests();
  void runStateVectorTests();
public:
  ValkyrieTests();
  void runTests();
  void handleTestResult(bool res, std::string testDescription);
  double getPercentagePassed();
  int noPassed() { return passed_; }
  std::vector<std::string> testsFailed();
};
```

Listing B.24: ValkyrieTests.h: File provides interface for Valkyrie test functions.

```
#include "ValkyrieTests.h"
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include "antlr4-runtime.h"
#include "../libs/qasm2Lexer.h"
#include "../libs/qasm2Parser.h"
#include "../libs/qasm2Visitor.h"
#include "../libs/qasm2BaseVisitor.h"
#include "../libs/staging.h"
#include "../libs/CPUDevice.h"
#include "../libs/GPUDevice.cuh"
#include "../libs/Measurement.h"
#include "../libs/ParsingGateUtilities.h"
```

```
#include <Windows.h>
#include <string>
#include <fstream>
#include <iostream>
#include <chrono>

using namespace antlr4;

// Parser Tests
bool parserTest1() {
 std::ifstream stream;
    stream.open("test/parserTest1.qasm");
    if (!stream.is_open()) {
        std::cout << "Couldn't find file specified" << std::endl;
        return false;
    }
    ANTLRInputStream input(stream);

    qasm2Lexer lexer(&input);
    CommonTokenStream tokens(&lexer);
    qasm2Parser parser(&tokens);

    qasm2Parser::MainprogContext* tree = parser.mainprog();

    qasm2BaseVisitor visitor;
    visitor.visitMainprog(tree);
    std::vector<Register> registers = visitor.getRegisters();
    std::vector<GateRequest> gateRequests = visitor.getGates();
    std::vector<MeasureCommand> commands = visitor.getMeasureCommands()
        ↪ ;

    // Testing registers
    if (registers.size() != 2) {
        return false;
    }
    Register qReg = registers[0];
    Register cReg = registers[1];

    // Checking Quantum register parameters
    if (!(qReg.getName() == "q") || !(qReg.getQuantumRegister().
        ↪ getWidth() == 3)) {
        return false;
    }

    // Checking Classical register parameters
    if (!(cReg.getName() == "c") || !(cReg.getClassicalRegister().
        ↪ getWidth() == 3)) {
        return false;
    }
    return true;
}

bool parserTest2() {
    std::ifstream stream;
    stream.open("test/parserTest2.qasm");
    if (!stream.is_open()) {
        std::cout << "Couldn't find file specified" << std::endl;
        return false;
```

```
    }
    ANTLRInputStream input(stream);

    qasm2Lexer lexer(&input);
    CommonTokenStream tokens(&lexer);
    qasm2Parser parser(&tokens);

    qasm2Parser::MainprogContext* tree = parser.mainprog();

    qasm2BaseVisitor visitor;
    visitor.visitMainprog(tree);
    std::vector<Register> registers = visitor.getRegisters();
    std::vector<GateRequest> gateRequests = visitor.getGates();
    std::vector<MeasureCommand> commands = visitor.getMeasureCommands()
        ↪ ;

    // Testing registers
    if (registers.size() != 2) {
        return false;
    }
    Register qReg = registers[0];
    Register cReg = registers[1];

    // Checking Quantum register parameters
    if (!(qReg.getName() == "q") || !(qReg.getQuantumRegister().
        ↪ getWidth() == 3)) {
        return false;
    }

    // Checking Classical register parameters
    if (!(cReg.getName() == "c") || !(cReg.getClassicalRegister().
        ↪ getWidth() == 3)) {
        return false;
    }

    // Checking Hadamard gate operation
    if (!(gateRequests.size() == 1)) {
        return false;
    }
    GateRequest gate = gateRequests[0];
    if (!(gate.getGateType() == U)) {
        return false;
    }
    std::vector<double> params = gate.getParameters();
    if (params.size() != 3) {
        return false;
    }
    const double PIc = 3.1415926535;
    if (!(params[0] == PIc / 2) || !(params[1] == 0) || !(params[2] ==
        ↪ PIc)) {
        return false;
    }
    return true;
}

bool parserTest3() {
    std::ifstream stream;
    stream.open("test/parserTest3.qasm");
```

```cpp
    if (!stream.is_open()) {
        std::cout << "Couldn't find file specified" << std::endl;
        return false;
    }
    ANTLRInputStream input(stream);

    qasm2Lexer lexer(&input);
    CommonTokenStream tokens(&lexer);
    qasm2Parser parser(&tokens);

    qasm2Parser::MainprogContext* tree = parser.mainprog();

    qasm2BaseVisitor visitor;
    visitor.visitMainprog(tree);
    std::vector<Register> registers = visitor.getRegisters();
    std::vector<GateRequest> gateRequests = visitor.getGates();
    std::vector<MeasureCommand> commands = visitor.getMeasureCommands();

    // Testing registers
    if (registers.size() != 2) {
        return false;
    }
    Register qReg = registers[0];
    Register cReg = registers[1];

    // Checking Hadamard gate operation
    if (!(gateRequests.size() == 2)) {
        return false;
    }
    GateRequest gate = gateRequests[1];
    if (!(gate.getGateType() == CX)) {
        return false;
    }
    std::vector<std::string> identifiers = gate.getRegisters();
    std::vector<int> locations = gate.getLocations();
    const double PIc = 3.1415926535;
    if (!(identifiers[0] == "q") || !(identifiers[1] == "q")) {
        return false;
    }
    if (!(locations[0] == 0) || !(locations[1] == 1)) {
        return false;
    }
    return true;
}

bool parserTest4() {
    std::ifstream stream;
    stream.open("test/parserTest4.qasm");
    if (!stream.is_open()) {
        std::cout << "Couldn't find file specified" << std::endl;
        return false;
    }
    ANTLRInputStream input(stream);

    qasm2Lexer lexer(&input);
    CommonTokenStream tokens(&lexer);
    qasm2Parser parser(&tokens);
```

```

    qasm2Parser::MainprogContext* tree = parser.mainprog();

    qasm2BaseVisitor visitor;
    visitor.visitMainprog(tree);
    std::vector<Register> registers = visitor.getRegisters();
    std::vector<GateRequest> gateRequests = visitor.getGates();
    std::vector<MeasureCommand> commands = visitor.getMeasureCommands()
        ↪ ;

    // Testing registers
    if (registers.size() != 2) {
        return false;
    }
    Register qReg = registers[0];
    Register cReg = registers[1];

    // Checking Hadamard gate operation
    if (commands.size() != 1) {
        return false;
    }
    MeasureCommand command = commands[0];
    if (!(command.getFrom().identifiers[0] == "q") || !(command.getTo()
        ↪ .identifiers[0] == "c")) {
        return false;
    }
    if (!(command.getFrom().locations[0] == 0) || !(command.getTo().
        ↪ locations[0] == 0)) {
        return false;
    }
    return true;
}

bool parserTest5() {
    std::ifstream stream;
    stream.open("test/parserTest5.qasm");
    if (!stream.is_open()) {
        std::cout << "Couldn't find file specified" << std::endl;
        return false;
    }
    ANTLRInputStream input(stream);

    qasm2Lexer lexer(&input);
    CommonTokenStream tokens(&lexer);
    qasm2Parser parser(&tokens);

    qasm2Parser::MainprogContext* tree = parser.mainprog();

    qasm2BaseVisitor visitor;
    visitor.visitMainprog(tree);
    std::vector<Register> registers = visitor.getRegisters();
    std::vector<GateRequest> gateRequests = visitor.getGates();
    std::vector<MeasureCommand> commands = visitor.getMeasureCommands()
        ↪ ;

    // Testing gate registrations
    if (!(gateRequests.size() == 3)) {
        return false;
```

```
    }

    for (auto gate : gateRequests) {
        if (!(gate.getGateType() == U)) {
            return false;
        }
    }

    GateRequest sdg1 = gateRequests[0];
    GateRequest h1 = gateRequests[1];
    GateRequest sdg2 = gateRequests[2];
    const double PIc = 3.1415926535;
    if (!(sdg1.getParameters()[2] == -PIc / 2) || !(sdg2.getParameters
        ↪ ()[2] == -PIc / 2)) {
        return false;
    }
    return true;
}

// Staging Tests
bool stagingTest1(std::vector<Register> mockReg1, std::vector<
    ↪ GateRequest> mockGateRequest1) {
    Stager stager = Stager(mockReg1, mockGateRequest1);
    std::vector<ConcurrentBlock> blocks = stager.getConcurrencyBlocks()
        ↪ ;
    if (blocks.size() != 2) {
        return false;
    }
    std::vector<GateRequest> req1 = blocks[0].getGates();
    std::vector<GateRequest> req2 = blocks[1].getGates();
    if (!(req1.size() == 1) || !(req2.size() == 1)) {
        return false;
    }
    if (!(req1[0].getGateType() == CX) || !(req2[0].getGateType() == U)
        ↪ ) {
        return false;
    }
    return true;
}

bool stagingTest2(std::vector<Register> mockReg1, std::vector<
    ↪ GateRequest> mockGateRequest1) {
    Stager stager = Stager(mockReg1, mockGateRequest1);
    std::vector<ConcurrentBlock> blocks = stager.getConcurrencyBlocks()
        ↪ ;
    if (blocks.size() != 2) {
        return false;
    }
    std::vector<GateRequest> req1 = blocks[0].getGates();
    std::vector<GateRequest> req2 = blocks[1].getGates();
    if (!(req1.size() == 1) || !(req2.size() == 2)) {
        return false;
    }
    if (!(req1[0].getGateType() == CX) || !(req2[0].getGateType() == U)
        ↪ || !(req2[1].getGateType() == U)) {
        return false;
    }
    return true;
```

```
}

bool stagingTest3(std::vector<Register> mockReg1, std::vector<
  ↪ GateRequest> mockGateRequest1) {
    Stager stager = Stager(mockReg1, mockGateRequest1);
    std::vector<ConcurrentBlock> blocks = stager.getConcurrencyBlocks()
        ↪ ;
    if (blocks.size() != 3) {
        return false;
    }
    std::vector<GateRequest> req1 = blocks[0].getGates();
    std::vector<GateRequest> req2 = blocks[1].getGates();
    std::vector<GateRequest> req3 = blocks[2].getGates();
    if (!(req1.size() == 1) || !(req2.size() == 2) || !(req3.size() ==
        ↪ 1)) {
        return false;
    }
    if (!(req1[0].getGateType() == CX) || !(req2[0].getGateType() == U)
        ↪  || !(req2[1].getGateType() == U)) {
        return false;
    }
    if (!(req3[0].getGateType() == CX)) {
        return false;
    }
    return true;
}

// CPU Device Tests
bool cpuQubitFactoryTest() {
    CPUQubitFactory cpuQubitFactory = CPUQubitFactory();
    Qubit* newQubit = cpuQubitFactory.generateQubit();
    if (!newQubit) {
        return false;
    }
    return true;
}

bool cpuGateFactoryTest() {
    CPUGateFactory gateFactory = CPUGateFactory();
    idLocationPairs pair;
    pair.identifiers.push_back("q");
    pair.locations.push_back(0);
    GateRequest hadamardGate = compileCompoundGateRequest("h", pair)
        ↪ [0];
    Gate* gate = gateFactory.generateGate(hadamardGate);
    if (!gate) {
        return false;
    }
    double oneOverSQRT2 = (1 / std::pow(2, 0.5));
    double diff1 = gate->fetchValue(0, 0).real() - oneOverSQRT2;
    double diff2 = gate->fetchValue(0, 1).real() - oneOverSQRT2;
    double diff3 = gate->fetchValue(1, 0).real() - oneOverSQRT2;
    double diff4 = gate->fetchValue(1, 1).real() + oneOverSQRT2;
    if (!(diff1 + diff2 + diff3 + diff4 < std::pow(10, -9))) {
        ↪          // Some numerical differences expexcted since we
        ↪ have fixed precisiona nd PI to 10 dp
        return false;
    }
```

```
    return true;
}

bool cpuQuantumCircuitTest() {

    CPUQubitFactory cpuQubitFactory = CPUQubitFactory();
    Qubit* newQubit = cpuQubitFactory.generateQubit();
    Qubit* newQubit2 = cpuQubitFactory.generateQubit();

    // Set up required gate factory
    CPUGateFactory* gateFactory = &CPUGateFactory();
    CPUQuantumCircuit circuit = CPUQuantumCircuit(gateFactory);

    // Set up required qubit map
    std::map<std::string, std::vector<Qubit*>> qubitMap{
        {"q", {newQubit, newQubit2}}
    };

    // Set up required concurrency blocks
    std::vector<Register> mockReg1;
    std::vector<GateRequest> mockGateRequest1;
    idLocationPairs mockPair;
    mockPair.identifiers.push_back("q");
    mockPair.locations.push_back(0);
    idLocationPairs mockPair2 = mockPair;
    mockPair.identifiers.push_back("q");
    mockPair.locations.push_back(1);

    mockGateRequest1 = compileCompoundGateRequest("cx", mockPair);
    mockGateRequest1.push_back(compileCompoundGateRequest("h",
        ↪ mockPair2)[0]);
    Stager stager = Stager(mockReg1, mockGateRequest1);
    std::vector<ConcurrentBlock> blocks = stager.getConcurrencyBlocks()
        ↪ ;

    // Load qubitMap
    circuit.loadQubitMap(qubitMap);
    // Load first concurrency block
    circuit.loadBlock(blocks[0]);
    std::vector<Calculation> calculations = circuit.getNextCalculation
        ↪ ();
    if (!(calculations.size() == 1)) {
        return false;
    }
    Calculation firstCalc = calculations[0];
    if (!(firstCalc.getQubit(0) == newQubit) || !(firstCalc.getQubit(1)
        ↪ == newQubit2)) {
        return false;
    }
    return true;
}

bool cpuDeviceAllUpTest() {
    // Setting up all required info
    std::vector<Register> mockReg1;
    QuantumRegister qReg = QuantumRegister("q", 3);
    ClassicalRegister cReg = ClassicalRegister("c", 3);
    Register qRegWrapped = Register(quantum_, qReg);
```

```
    Register cRegWrapped = Register(classical_, cReg);
    mockReg1.push_back(qRegWrapped);
    mockReg1.push_back(cRegWrapped);
    std::vector<GateRequest> mockGateRequest1;
    idLocationPairs mockPair;
    mockPair.identifiers.push_back("q");
    mockPair.locations.push_back(0);
    idLocationPairs mockPair2 = mockPair;
    mockPair.identifiers.push_back("q");
    mockPair.locations.push_back(1);

    mockGateRequest1 = compileCompoundGateRequest("cx", mockPair);
    mockGateRequest1.push_back(compileCompoundGateRequest("h",
        ↪ mockPair2)[0]);
    mockGateRequest1.push_back(compileCompoundGateRequest("h",
        ↪ mockPair2)[0]);
    mockGateRequest1.push_back(compileCompoundGateRequest("cx",
        ↪ mockPair)[0]);
    Stager stager = Stager(mockReg1, mockGateRequest1);

    std::vector<ConcurrentBlock> blocks = stager.getConcurrencyBlocks()
        ↪ ;

    CPUDevice device = CPUDevice();
    device.run(mockReg1, blocks);
    std::map<std::string, std::vector<Qubit*>> results = device.
        ↪ revealQuantumState();
    if (!(results["q"][0]->fetch(0)->real() == 1) || !(results["q"
        ↪ ][1]->fetch(0)->real() == 1)) {
        return false;
    }
    return true;
}

bool cpuStateVectorRunTest() {
    // Setting up all required info
    std::vector<Register> mockReg1;
    QuantumRegister qReg = QuantumRegister("q", 3);
    ClassicalRegister cReg = ClassicalRegister("c", 3);
    Register qRegWrapped = Register(quantum_, qReg);
    Register cRegWrapped = Register(classical_, cReg);
    mockReg1.push_back(qRegWrapped);
    mockReg1.push_back(cRegWrapped);
    std::vector<GateRequest> mockGateRequest1;
    idLocationPairs mockPair;
    mockPair.identifiers.push_back("q");
    mockPair.locations.push_back(0);
    idLocationPairs mockPair2 = mockPair;
    mockPair.identifiers.push_back("q");
    mockPair.locations.push_back(1);

    mockGateRequest1 = compileCompoundGateRequest("h", mockPair2);
    mockGateRequest1.push_back(compileCompoundGateRequest("cx",
        ↪ mockPair)[0]);
    Stager stager = Stager(mockReg1, mockGateRequest1);

    std::vector<ConcurrentBlock> blocks = stager.getConcurrencyBlocks()
        ↪ ;
```

```

    CPUDevice device = CPUDevice();
    device.runSV(mockReg1, blocks);
    std::map<std::string, std::vector<Qubit*>> results = device.
       ↪ revealQuantumState();
    if (!(results["q"][0]->fetch(0)->real() == 1) || !(results["q"
       ↪ ][1]->fetch(0)->real() == 1)) {
        return false;
    }
    return true;
}

// GPU Device Tests
bool gpuQubitFactoryTest() {
    GPUQubitFactory cpuQubitFactory = GPUQubitFactory();
    Qubit* newQubit = cpuQubitFactory.generateQubit();
    if (!newQubit) {
        return false;
    }
    return true;
}

bool gpuGateFactoryTest() {
    GPUGateFactory gateFactory = GPUGateFactory();
    idLocationPairs pair;
    pair.identifiers.push_back("q");
    pair.locations.push_back(0);
    GateRequest hadamardGate = compileCompoundGateRequest("h", pair)
       ↪ [0];
    Gate* gate = gateFactory.generateGate(hadamardGate);
    if (!gate) {
        return false;
    }
    double oneOverSQRT2 = (1 / std::pow(2, 0.5));
    double diff1 = gate->fetchValue(0, 0).real() - oneOverSQRT2;
    double diff2 = gate->fetchValue(0, 1).real() - oneOverSQRT2;
    double diff3 = gate->fetchValue(1, 0).real() - oneOverSQRT2;
    double diff4 = gate->fetchValue(1, 1).real() + oneOverSQRT2;
    if (!(diff1 + diff2 + diff3 + diff4 < std::pow(10, -9))) {
       ↪           // Some numerical differences expexcted since we
       ↪ have fixed precisiona nd PI to 10 dp
        return false;
    }
    return true;
}

bool gpuQuantumCircuitTest() {

    GPUQubitFactory cpuQubitFactory = GPUQubitFactory();
    Qubit* newQubit = cpuQubitFactory.generateQubit();
    Qubit* newQubit2 = cpuQubitFactory.generateQubit();

    // Set up required gate factory
    GPUGateFactory* gateFactory = &GPUGateFactory();
    GPUQuantumCircuit circuit = GPUQuantumCircuit(gateFactory);

    // Set up required qubit map
    std::map<std::string, std::vector<Qubit*>> qubitMap{
```

```
        {"q", {newQubit, newQubit2}}
    };

    // Set up required concurrency blocks
    std::vector<Register> mockReg1;
    std::vector<GateRequest> mockGateRequest1;
    idLocationPairs mockPair;
    mockPair.identifiers.push_back("q");
    mockPair.locations.push_back(0);
    idLocationPairs mockPair2 = mockPair;
    mockPair.identifiers.push_back("q");
    mockPair.locations.push_back(1);

    mockGateRequest1 = compileCompoundGateRequest("cx", mockPair);
    mockGateRequest1.push_back(compileCompoundGateRequest("h",
        ↪ mockPair2)[0]);
    Stager stager = Stager(mockReg1, mockGateRequest1);
    std::vector<ConcurrentBlock> blocks = stager.getConcurrencyBlocks()
        ↪ ;

    // Load qubitMap
    circuit.loadQubitMap(qubitMap);
    // Load first concurrency block
    circuit.loadBlock(blocks[0]);
    std::vector<Calculation> calculations = circuit.getNextCalculation
        ↪ ();
    if (!(calculations.size() == 1)) {
        return false;
    }
    Calculation firstCalc = calculations[0];
    if (!(firstCalc.getQubit(0) == newQubit) || !(firstCalc.getQubit(1)
        ↪ == newQubit2)) {
        return false;
    }
    return true;
}

bool gpuDeviceAllUpTest() {
    // Setting up all required info
    std::vector<Register> mockReg1;
    QuantumRegister qReg = QuantumRegister("q", 3);
    ClassicalRegister cReg = ClassicalRegister("c", 3);
    Register qRegWrapped = Register(quantum_, qReg);
    Register cRegWrapped = Register(classical_ , cReg);
    mockReg1.push_back(qRegWrapped);
    mockReg1.push_back(cRegWrapped);
    std::vector<GateRequest> mockGateRequest1;
    idLocationPairs mockPair;
    mockPair.identifiers.push_back("q");
    mockPair.locations.push_back(0);
    idLocationPairs mockPair2 = mockPair;
    mockPair.identifiers.push_back("q");
    mockPair.locations.push_back(1);

    mockGateRequest1 = compileCompoundGateRequest("cx", mockPair);
    mockGateRequest1.push_back(compileCompoundGateRequest("h",
        ↪ mockPair2)[0]);
    mockGateRequest1.push_back(compileCompoundGateRequest("h",
```

```
        ↪ mockPair2)[0]);
    mockGateRequest1.push_back(compileCompoundGateRequest("cx",
        ↪ mockPair)[0]);
    Stager stager = Stager(mockReg1, mockGateRequest1);

    std::vector<ConcurrentBlock> blocks = stager.getConcurrencyBlocks()
        ↪ ;

    GPUDevice device = GPUDevice();
    device.run(mockReg1, blocks);
    std::map<std::string, std::vector<Qubit*>> results = device.
        ↪ revealQuantumState();
    if (!(results["q"][0]->fetch(0)->real() == 1) || !(results["q"
        ↪ ][1]->fetch(0)->real() == 1)) {
         return false;
    }
    return true;
}

bool gpuStateVectorRunTest() {
    // Setting up all required info
    std::vector<Register> mockReg1;
    QuantumRegister qReg = QuantumRegister("q", 3);
    ClassicalRegister cReg = ClassicalRegister("c", 3);
    Register qRegWrapped = Register(quantum_, qReg);
    Register cRegWrapped = Register(classical_, cReg);
    mockReg1.push_back(qRegWrapped);
    mockReg1.push_back(cRegWrapped);
    std::vector<GateRequest> mockGateRequest1;
    idLocationPairs mockPair;
    mockPair.identifiers.push_back("q");
    mockPair.locations.push_back(0);
    idLocationPairs mockPair2 = mockPair;
    mockPair.identifiers.push_back("q");
    mockPair.locations.push_back(1);

    mockGateRequest1 = compileCompoundGateRequest("h", mockPair2);
    mockGateRequest1.push_back(compileCompoundGateRequest("cx",
        ↪ mockPair)[0]);
    Stager stager = Stager(mockReg1, mockGateRequest1);

    std::vector<ConcurrentBlock> blocks = stager.getConcurrencyBlocks()
        ↪ ;

    GPUDevice device = GPUDevice();
    device.runSV(mockReg1, blocks);
    std::map<std::string, std::vector<Qubit*>> results = device.
        ↪ revealQuantumState();
    if (!(results["q"][0]->fetch(0)->real() == 1) || !(results["q"
        ↪ ][1]->fetch(0)->real() == 1)) {
         return false;
    }
    return true;
}

// Measurement Test

bool runSimpleMeasurementTest() {
```

```
    std::ifstream stream;
    stream.open("test/measureTest1.qasm");
    if (!stream.is_open()) {
        std::cout << "Couldn't find file specified" << std::endl;
        return false;
    }
    ANTLRInputStream input(stream);

    qasm2Lexer lexer(&input);
    CommonTokenStream tokens(&lexer);
    qasm2Parser parser(&tokens);

    qasm2Parser::MainprogContext* tree = parser.mainprog();

    qasm2BaseVisitor visitor;
    visitor.visitMainprog(tree);
    std::vector<Register> registers = visitor.getRegisters();
    std::vector<GateRequest> gateRequests = visitor.getGates();
    std::vector<MeasureCommand> commands = visitor.getMeasureCommands()
        ↪ ;
    Stager stager = Stager(registers, gateRequests);
    CPUDevice device = CPUDevice();
    device.run(registers, stager.getConcurrencyBlocks());
    MeasurementCalculator calc = MeasurementCalculator(registers);
    calc.loadMeasureCommands(commands);
    calc.registerHandover(device.revealQuantumState());
    calc.measureAll();
    calc.passMeasurementsIntoClassicalRegisters();
    Register cReg = calc.fetchRegister("c");
    if (cReg.getClassicalRegister().getWidth() != 3) {
        return false;
    }
    if (cReg.getClassicalRegister().getValue(0) != 0) {
        return false;
    }
    return true;
}

// StateVector Tests
bool stateVectorSimpleReorderTest() {
    std::vector<Register> mockReg1;
    QuantumRegister qReg = QuantumRegister("q", 3);
    ClassicalRegister cReg = ClassicalRegister("c", 3);
    Register qRegWrapped = Register(quantum_, qReg);
    Register cRegWrapped = Register(classical_, cReg);
    mockReg1.push_back(qRegWrapped);
    mockReg1.push_back(cRegWrapped);
    std::vector<GateRequest> mockGateRequest1;
    idLocationPairs mockPair;
    mockPair.identifiers.push_back("q");
    mockPair.locations.push_back(0);
    idLocationPairs mockPair2 = mockPair;
    mockPair.identifiers.push_back("q");
    mockPair.locations.push_back(1);

    mockGateRequest1 = compileCompoundGateRequest("h", mockPair2);
    mockGateRequest1.push_back(compileCompoundGateRequest("cx",
        ↪ mockPair)[0]);
```

```
    Stager stager = Stager(mockReg1, mockGateRequest1);

    std::vector<ConcurrentBlock> blocks = stager.getConcurrencyBlocks()
        ↪ ;

    CPUDevice device = CPUDevice();
    device.run(mockReg1, blocks);
    StateVector* sv = device.getStateVector();
    std::vector<std::complex<double>> oldState = sv->getState();
    std::vector<SVPair> newOrder;
    SVPair elem1("q", 0);
    SVPair elem2("q", 2);
    SVPair elem3("q", 1);
    newOrder = { elem1, elem2, elem3 };
    StateVector* reordered = sv->reorder(newOrder);
    std::vector<std::complex<double>> newState = reordered->getState();
    if (!(newState[0] == oldState[0]) || !(newState[5] == oldState[6]))
        ↪ {
        return false;
    }
    reordered->directModify(2, 5);
    newState = reordered->getState();
    sv->reconcile(reordered);
    oldState = sv->getState();
    if (!(newState[2] == oldState[1])) {
        return false;
    }
    return true;
}

// Test banks

void ValkyrieTests::runParserTests() {

    // Basic Register set-up test
    handleTestResult(parserTest1(), "Parser Test: Basic Register set up
        ↪ ");
    // Register setup and simple gate applications
    handleTestResult(parserTest2(), "Parser Test: Simple Gate
        ↪ application");
    // Register setup and CX multi qubit gate application
    handleTestResult(parserTest3(), "Parser Test: CX multi-qubit gate
        ↪ application");
    // Measure command setup check
    handleTestResult(parserTest4(), "Parser Test: Checking Measure
        ↪ command operation");
    // Compound gate setup check
    handleTestResult(parserTest5(), "Parser Test: Checking compound
        ↪ gate setup is working");
}

void ValkyrieTests::runStagingTests()
{
    std::vector<Register> mockReg1;
    std::vector<GateRequest> mockGateRequest1;
    idLocationPairs mockPair;
    mockPair.identifiers.push_back("q");
    mockPair.locations.push_back(0);
```

```
    idLocationPairs mockPair2 = mockPair;
    mockPair.identifiers.push_back("c");
    mockPair.locations.push_back(1);

    mockGateRequest1 = compileCompoundGateRequest("cx", mockPair);
    mockGateRequest1.push_back(compileCompoundGateRequest("h",
        ↪ mockPair2)[0]);

    handleTestResult(stagingTest1(mockReg1, mockGateRequest1), "Staging
        ↪  test: Checking for correct concurrency resolution simple
        ↪ case");

    mockGateRequest1.push_back(compileCompoundGateRequest("h",
        ↪ mockPair2)[0]);
    handleTestResult(stagingTest2(mockReg1, mockGateRequest1), "Staging
        ↪  test: Checking for correct concurrency resolution in
        ↪ intermediate case");
    mockGateRequest1.push_back(compileCompoundGateRequest("cx",
        ↪ mockPair)[0]);
    handleTestResult(stagingTest3(mockReg1, mockGateRequest1), "Staging
        ↪  test: Checking for correct concurrency resolution in
        ↪ complex case");
}

void ValkyrieTests::runCPUDeviceTests()
{
    // Test CPU Qubit Factory
    handleTestResult(cpuQubitFactoryTest(), "CPU Device Test: Checking
        ↪ whether CPU Qubit factory is able to emit Qubits");
    // Test CPU Gate Factory
    handleTestResult(cpuGateFactoryTest(), "CPU Device Test: Checking
        ↪ whether CPU Gate factory is able to resolve correct gates");
    // Test CPU Quantum Circuit
    handleTestResult(cpuQuantumCircuitTest(), "CPU Device Test:
        ↪ Checking whether CPU Quantum Circuit is able to compile
        ↪ calculations");
    // Test CPU device all up
    handleTestResult(cpuDeviceAllUpTest(), "CPU Device Test: Full run
        ↪ all up test");

    handleTestResult(cpuStateVectorRunTest(), "CPU Device Test:
        ↪ Checking State vector operation");
}

void ValkyrieTests::runGPUDeviceTests()
{
    // Test GPU Qubit Factory
    handleTestResult(gpuQubitFactoryTest(), "GPU Device Test: Checking
        ↪ whether GPU Qubit factory is able to emit Qubits");
    // Test GPU Gate Factory
    handleTestResult(gpuGateFactoryTest(), "GPU Device Test: Checking
        ↪ whether GPU Gate factory is able to resolve correct gates");
    // Test GPU Quantum Circuit
    handleTestResult(gpuQuantumCircuitTest(), "GPU Device Test:
        ↪ Checking whether GPU Quantum Circuit is able to compile
        ↪ calculations");
    // Test GPU device all up
    handleTestResult(gpuDeviceAllUpTest(), "GPU Device Test: Full run
```

```
       ↪ all up test");
    // Test GPU using statevector
    handleTestResult(gpuStateVectorRunTest(), "GPU Device Test:
       ↪ Statevector simulation for GPU");
}

void ValkyrieTests::runMeasurementTests()
{
    handleTestResult(runSimpleMeasurementTest(), "Measurement test:
       ↪ Checking whether simple emasure case is handled");
}

void ValkyrieTests::runStateVectorTests()
{
    handleTestResult(stateVectorSimpleReorderTest(), "StateVector test:
       ↪  checking reordering works");
}



ValkyrieTests::ValkyrieTests()
{
    // Module initialisation passed ;)
 total_ = 1;
 passed_ = 1;
}

void ValkyrieTests::runTests()
{
 // Valkyrie Test Suite
    // Parser Tests
    runParserTests();
    // Staging Tests
    runStagingTests();
    // CPU Device Tests
    runCPUDeviceTests();
    // GPU Device Tests
    runGPUDeviceTests();
    // Measurement Test
    runMeasurementTests();
    // StateVector tests
    runStateVectorTests();
}

void ValkyrieTests::handleTestResult(bool res, std::string
   ↪ testDescription)
{
    total_++;
    if (res) {
        passed_++;
        passedTests.push_back(testDescription);
    }
    else {
        failedTests.push_back(testDescription);
    }
}

double ValkyrieTests::getPercentagePassed()
```

```
{
 return 100 * (double) passed_ / (double) total_;
}

std::vector<std::string> ValkyrieTests::testsFailed()
{
 return failedTests;
}
```

Listing B.25: ValkyrieTests.cpp: File provides implementation for Valkyrie test functions.

Appendix C

VisualQ Codebase

```
const path = require("path");

const { app, BrowserWindow, ipcMain } = require("electron");
const isDev = require("electron-is-dev");
const fs = require('fs');
const util = require('util');
const exec = util.promisify(require('child_process').exec);

let installExtension, REACT_DEVELOPER_TOOLS; // NEW!

if (isDev) {
  const devTools = require("electron-devtools-installer");
  installExtension = devTools.default;
  REACT_DEVELOPER_TOOLS = devTools.REACT_DEVELOPER_TOOLS;
} // NEW!

// Handle creating/removing shortcuts on Windows when installing/uninstalling
if (require("electron-squirrel-startup")) {
    app.quit();
  } // NEW!

function createWindow() {
  // Create the browser window.
  const win = new BrowserWindow({
    width: 1080,
    height: 800,
    webPreferences: {
      nodeIntegration: true
    }
  });

  // and load the index.html of the app.
  // win.loadFile("index.html");
  win.loadURL(
    isDev
      ? "http://localhost:3000"
      : `file://${path.join(__dirname, "../build/index.html")}`
  );

  // Open the DevTools.
  if (isDev) {
```

```
    win.webContents.openDevTools({ mode: "detach" });
  }
}

// This method will be called when Electron has finished
// initialization and is ready to create browser windows.
// Some APIs can only be used after this event occurs.
app.whenReady().then(() => {
    createWindow();

    if (isDev) {
      installExtension(REACT_DEVELOPER_TOOLS)
        .then(name => console.log(`Added Extension:  ${name}`))
        .catch(error => console.log(`An error occurred: , ${error}`));
    }
  }); // UPDATED!

// Quit when all windows are closed, except on macOS. There, it's
   ↪ common
// for applications and their menu bar to stay active until the user
   ↪ quits
// explicitly with Cmd + Q.
app.on("window-all-closed", () => {
  if (process.platform !== "darwin") {
    app.quit();
  }
});

app.on("activate", () => {
  // On macOS it's common to re-create a window in the app when the
  // dock icon is clicked and there are no other windows open.
  if (BrowserWindow.getAllWindows().length === 0) {
    createWindow();
  }
});

// In this file you can include the rest of your app's specific main
   ↪ process
// code. You can also put them in separate files and require them here.



async function runValkyrie(gpuMode){
    if(gpuMode){
        try {
            const { stdout, stderr } = await exec('Debug\\Valkyrie2.0.
               ↪ exe -g -o "output.qasm" -sv -json');
            return stdout;
          } catch (e) {
            console.error(e); // should contain code (exit code) and
               ↪ signal (that caused the termination).
          }
    } else{
        try {
            const { stdout, stderr } = await exec('Debug\\Valkyrie2.0.
               ↪ exe -c -o "output.qasm" -sv -json');
            return stdout;
          } catch (e) {
```

```
            console.error(e); // should contain code (exit code) and signal (that caused the termination).
          }
    }

}

ipcMain.on("sendFile", async (event, arg) => {
    fs.writeFile(arg[0], arg[1], function (err) {
        if (err) return console.log(err);
    });
    try {
        var result = await runValkyrie(arg[2] === "g");
        event.returnValue = result;
    } catch (e){
        console.error("Valkyrie run failed");
        event.returnValue = "Valkyrie run failed";
    }
})

ipcMain.on("fetchLast", async (event, arg) => {
  const returnVal = fs.readFileSync("output.qasm", 'utf-8');
  event.returnValue = returnVal;
})
```

Listing C.1: electron.js: Electron function file for VisualQ handle file IO to Valkyrie

```
import React from "react"
import "./mainContainer.css"
import Form from 'react-bootstrap/Form'
import Button from 'react-bootstrap/Button'
import Table from 'react-bootstrap/Table'
import ToggleButtonGroup from 'react-bootstrap/ToggleButtonGroup'
import ToggleButton from 'react-bootstrap/ToggleButton'

const electron = window.require('electron');
const ipcRenderer = electron.ipcRenderer;



async function sendFile(val){
    const res = await ipcRenderer.sendSync('sendFile', val);
    console.log(res);
    return res;
}

async function getLastFile(){
    const res = await ipcRenderer.sendSync('fetchLast', 0);
    return res;
}

class MainContainer extends React.Component{

    constructor(props){
        super(props);
        this.state = {value: '', result: '', execMode: 'c'};
        this.handleSubmit = this.handleSubmit.bind(this);
        this.handleChange = this.handleChange.bind(this);
        this.renderOutput = this.renderOutput.bind(this);
```

```
        this.handleExecutionModeSwitch = this.handleExecutionModeSwitch.bind(this);
        this.getButtonVariant = this.getButtonVariant.bind(this);
        this.getBackGround = this.getBackGround.bind(this);
        this.handleFirstMount = this.handleFirstMount.bind(this);
    }

    componentDidMount(){
        this.handleFirstMount();
    }

    async handleFirstMount(){
        try {
            var res = await getLastFile();
            console.log(res);
            this.setState({value: res, result: this.state.result, execMode: this.state.execMode});
        } catch(e){
            console.log("Error opening last saved output.qasm");
        }
    }

    async handleSubmit(event){
        const fileName = "output.qasm";
        console.log(this.state.execMode);
        const val = [fileName, this.state.value, this.state.execMode];
        try{
            var res = await sendFile(val);
            this.setState({value: this.state.value, result: res, execMode: this.state.execMode});
        }
        catch(e){
            console.log("Error during valkyrie run")
        }
        return false;
    }

    handleChange(event){
        this.setState({value: event.target.value, result: this.state.result, execMode: this.state.execMode});
    }

    handleExecutionModeSwitch(event){
        if(event === 1){
            this.setState({value: this.state.value, result: this.state.result, execMode: 'c'});
        } else {
            this.setState({value: this.state.value, result: this.state.result, execMode: 'g'});
        }
    }

    getButtonVariant(){
        if(this.state.execMode === 'g'){
            return "success"
        } else{
            return "primary"
        }
```

```
    }

    getBackGround(){
        if(this.state.execMode === 'g'){
            return "mainContent-gpu"
        } else{
            return "mainContent"
        }
    }

    render(){
        return (<div className={this.getBackGround()}>
            <div className="write-code">
                Hello there, please enter your QASM code below:
            </div>
            <div>
            <ToggleButtonGroup type="radio" name="options" defaultValue={1} onChange={this.handleExecutionModeSwitch}>
                <ToggleButton variant={this.getButtonVariant()} value={1}>CPU Execution Mode</ToggleButton>
                <ToggleButton variant={this.getButtonVariant()} value={2}>GPU Execution Mode</ToggleButton>
            </ToggleButtonGroup>
            </div>
            <div>
                <Form>
                    <Form.Group controlId="qasmInput">
                        <Form.Label>QASM Input</Form.Label>
                        <Form.Control
                            as="textarea"
                            rows={10}
                            placeholder="OPENQASM 2.0;"
                            defaultValue={this.state.value}
                            onChange={this.handleChange}
                        />
                    </Form.Group>
                    <Button variant="primary" type="button" onClick={this.handleSubmit}>
                        Submit
                    </Button>
                </Form>
                <br/>
                {this.renderOutput()}
            </div>
        </div>)
    }

    evaluateRow(cState){
        var name = cState["id"];
        var values = cState["values"];
        return (values.map((val, index) => (
            <tr>
                <td>
                    {name}
                </td>
                <td>
                    {index}
                </td>
```

```
                <td>
                    {val}
                </td>
            </tr>
        )))
    }

    handleIndex(index, svLength){
        var str = index.toString(2);
        const noElem = Math.log2(svLength);
        console.log(noElem - str.length);
        const len = str.length;
        for(var i = 0; i < noElem - len; i++){
            str = "0" + str;
        }
        return str;
    }

    renderOutput(){
        if(this.state.result === ''){
            return <div>Please click submit to see output</div>
        } else {
            var obj = JSON.parse(this.state.result);
            var StateVector = obj["StateVector"];
            var svLength = StateVector.length;
            var ClassicalRegisters = obj["ClassicalRegisters"];
            console.log(StateVector);
            console.log(ClassicalRegisters);
            return <div>
                Output:
                <br/>
                <Table striped bordered hover>
                        <thead>
                            <tr>
                                <th>Classical Register</th>
                                <th>Index</th>
                                <th>Measured Value</th>
                            </tr>
                        </thead>
                        <tbody>
                        {ClassicalRegisters.map((cState, index) => (
                            this.evaluateRow(cState)
                        ))}
                        </tbody>
                    </Table>
                    <Table striped bordered hover>
                        <thead>
                            <tr>
                                <th>State</th>
                                <th>Quantum State</th>
                            </tr>
                        </thead>
                        <tbody>
                        {StateVector.map((qState, index) => (
                            <tr>
                                <td>
                                    {this.handleIndex(index, svLength)}
                                </td>
```

```
                                <td>
                                    {qState}
                                </td>
                            </tr>
                        ))}
                        </tbody>
                    </Table>
                </div>
        }
    }
}

export default MainContainer;
```

Listing C.2: mainContainer.jsx: Main component of VisualQ provides simple user interface

Appendix D

Evaluation Data

D.0.1 Baseline test results

Simulator	Valkyrie				Qiskit	Cirq
Processor	CPU		GPU		CPU	CPU
Mode	Statevector	Fast	Statevector	Fast	NA	NA
	Time(ns)					
Run 1	4086100	3288900	5750700	4086100	35008400	6999300
Run 2	4130700	3528800	5555700	4130700	28004900	7002800
Run 3	4005700	3496400	5292400	4005700	28007000	7002100
Run 4	3887500	4302100	5511600	3887500	35009100	7001400
Run 5	3870200	3363600	5299800	3870200	28004900	7000700
Run 6	3967800	3264700	5255400	3967800	28008400	7001400
Run 7	3872000	3205000	5712000	3872000	28005600	7541200
Run 8	3873000	3518700	5460700	3873000	35007000	7002800
Run 9	3920100	3245600	5200100	3920100	28009100	7001400
Run 10	4069600	3233200	5296900	4069600	28004200	5451500
Run 11	3876000	3218800	5127600	3876000	28007700	7000700
Run 12	3993000	3360000	5224100	3993000	28007000	7007000
Run 13	3865500	3236800	6052100	3865500	35004900	6998600
Run 14	3858100	3220500	5174300	3858100	28007700	6445140
Run 15	3882400	3211800	5602500	3882400	28007000	7000700
Run 16	3866900	3206100	5309100	3866900	28004200	7002100
Run 17	3851800	3337100	5132800	3851800	35009100	6999300
Run 18	3840500	3205700	5323700	3840500	28004900	7005900
Run 19	3863900	3315300	5233400	3863900	28010500	7003500
Run 20	4050900	3248300	5061700	4050900	28002800	7003500

Table D.1: Execution times for Valkyrie, Qiskit and Cirq for baseline circuit using 20 iterations as initial test

Simulator	Valkyrie				Qiskit	Cirq
Processor	CPU		GPU		CPU	CPU
Mode	Statevector	Fast	Statevector	Fast	NA	NA
	Time(ns)					
Run 1	4220600	3270800	5585600	4534600	28000700	7327300
Run 2	3916400	3238300	5375900	4254700	28009800	4885600
Run 3	3964200	3467900	5232900	4344200	35010500	6559300
Run 4	3869400	3336300	5308800	4856500	28004900	8430000
Run 5	3877300	3209400	6003800	4685700	28003500	4115800
Run 6	3879900	3321500	6300900	4411500	28007700	8691600

Run 7	3964900	3301000	5759600	4257100	28003500	7178900
Run 8	3896100	3227000	5241100	4405200	28047600	6160100
Run 9	4046000	3316000	5330100	4258600	27984600	8817200
Run 10	3968100	3258800	5283900	4323200	35006300	7507700
Run 11	3932900	3464000	5213100	4578500	28004900	6137900
Run 12	3889800	3236100	5357400	4544600	28009800	5767200
Run 13	4193500	3242800	5221700	4307100	28004200	6364100
Run 14	3938200	3345400	5695600	4678500	35009800	7747700
Run 15	4101700	3346600	5584100	4382700	28007700	6837500
Run 16	3884500	3293400	5254800	4223300	35006300	5662100
Run 17	3874900	3228200	5286900	4349600	28004200	10014300
Run 18	3871500	3256900	5319100	4664100	28007700	6313500
Run 19	3908900	3214000	5441100	4475900	35007000	5708600
Run 20	3872900	3335400	5406300	4437500	28006300	8886300
Run 21	3870800	3409700	5756700	4362000	28006300	6113800
Run 22	3902700	3242200	5794100	4405000	35007700	7100600
Run 23	3894600	3211300	5328900	4247000	28009800	7097200
Run 24	4137200	3200500	5474000	4591700	28001400	6031500
Run 25	3884300	3239400	5381400	4516900	28009100	7086600
Run 26	3887900	3374700	6185500	4351800	35006300	6068900
Run 27	3888500	3284300	5571000	4414800	35011900	6388400
Run 28	3873000	3375900	5360500	4233900	28003500	8527600
Run 29	3890600	3173000	5930800	4335200	35007000	5251000
Run 30	4043200	3212600	5323800	4440000	28004900	6511400
Run 31	3927800	3216000	6524100	4408800	28183400	8799200
Run 32	4214200	3419100	5430400	4302700	27843200	5729100
Run 33	4316500	3218100	5708300	4405700	35009100	8506600
Run 34	4212300	3414200	5854000	4601100	28003500	6780800
Run 35	4061100	3382600	5355400	4373900	28009100	6837200
Run 36	3885300	3325600	5618800	4388900	28004200	7157500
Run 37	3996400	3213500	5781700	4358200	35009100	6798900
Run 38	3945700	3404700	5416100	4773200	28009100	6132400
Run 39	4072700	3306300	5755900	4323800	28000000	6787000
Run 40	3893400	3466700	5693900	4504800	35010500	7147200
Run 41	3956700	5409400	5289800	5226600	28007000	9106900
Run 42	4114800	3313600	5368300	4231000	28008400	7758400
Run 43	3971200	3302600	5239600	4901300	35006300	5208600
Run 44	3917200	3423500	5162500	4229100	28009800	5550800
Run 45	4019300	3342200	5426400	4394300	28005600	8843700
Run 46	3912400	3399600	5399200	4317500	28003500	7432300
Run 47	3881500	3218000	5856200	4501900	35007700	8846900
Run 48	3926800	3300200	5254100	4850000	28008400	4070200
Run 49	4283500	3486400	5751500	4195500	28002800	6913600
Run 50	3906100	3577800	5584300	4414000	35010500	6758700
Run 51	4112900	3384200	5149000	4238500	28006300	7257300
Run 52	3901000	3248600	5408000	4394300	28008400	7111700
Run 53	3961600	3421900	5422000	5061500	28004200	7349300
Run 54	3886800	3480000	5277300	4759400	35011200	6171100
Run 55	4124200	3489300	6066000	4325500	28016100	7063800
Run 56	3969000	3538000	5883400	4618200	28006300	5905700
Run 57	3857000	3383800	5855700	4704500	35016100	6274400
Run 58	3991400	3703800	5357700	4241600	27997200	8593000
Run 59	3911700	3315000	5257800	4656000	28012600	8821200
Run 60	4357500	3315200	6035600	5243800	28001400	3994500
Run 61	3899300	3470200	5349200	4271300	35007000	6811400

Run 62	3924700	3356900	5265200	5208800	28007000	6886000
Run 63	4056000	3306900	5500300	5119600	28005600	8210600
Run 64	3892000	3303900	5953000	4453800	35011200	8016100
Run 65	3955900	3229400	5169300	4559600	28004900	4970700
Run 66	4389400	3330500	6940900	4236200	35010500	8618200
Run 67	3861600	3390600	5315000	4373600	28002800	6077600
Run 68	3827100	3473800	6200500	4367200	28009800	9200300
Run 69	3859700	3513800	5397500	4501500	35007700	3711400
Run 70	4385000	3523800	5673300	5614500	28005600	6902000
Run 71	4429500	3203800	5362800	4324200	27997200	6927300
Run 72	4290100	3288400	5307400	4515700	28015400	8267900
Run 73	4191200	3483100	5954800	4441900	28007700	6697400
Run 74	3895200	3480000	5495900	4566600	28006300	6100000
Run 75	3867900	3288500	6400500	5061400	35004900	8353500
Run 76	4398200	3354700	5410700	4405700	21004200	9202700
Run 77	4007800	3213300	5350100	4565500	21004200	7540900
Run 78	4052100	3266800	5502900	4499900	28006300	6863300
Run 79	3860800	3225000	5226900	4386200	28010500	7881700
Run 80	3900700	3156500	5544000	4249600	28002800	5285400
Run 81	4299000	3253900	6123600	4937700	28007700	8070400
Run 82	3903500	3287400	5871100	4586600	35006300	8888500
Run 83	3944200	3311100	5360000	4482800	28006300	7282100
Run 84	4083000	3363100	5560800	4493600	28006300	8056600
Run 85	3860100	3459000	5391100	4448800	28009800	7833300
Run 86	3866900	3480200	5288000	4512400	28002800	6620400
Run 87	3899300	3298500	5249600	5213000	28007000	4410800
Run 88	4093300	3231200	5463500	4712400	28006300	5025100
Run 89	3917000	3331700	5951300	4646400	28006300	8824300
Run 90	3898300	3520200	5335400	4836900	28004900	6288500
Run 91	3898600	3348100	5466000	4447900	28004200	7622200
Run 92	4053100	3391300	5663500	4873600	28010500	7199600
Run 93	3898400	3485400	5245700	4598800	28004900	7533300
Run 94	4126000	3458100	5430200	4431900	28005600	8080600
Run 95	4171400	3270100	5278300	4282200	35011200	6425800
Run 96	3906400	3370200	5284900	4359900	28004200	8174700
Run 97	4795300	3480200	5982900	4925600	28009800	6271900
Run 98	3902300	3833500	5312000	4566300	35004900	6597300
Run 99	4023200	3421100	5670600	4393800	28007000	6343600
Run 100	4217200	3327200	5404100	4311100	28005600	6125400

Table D.2: Execution times for Valkyrie, Qiskit and Cirq for baseline circuit using 100 iterations

Simulator	Valkyrie					
Processor	CPU			GPU		
Mode	Parsing	Staging	Execution	Parsing	Staging	Execution
Run 1	2945400	53800	940000	3196700	55200	2438400
Run 2	3067000	55800	939600	2937600	52800	2647800
Run 3	2921300	52800	1019600	2997700	53000	2283100
Run 4	2966300	52600	929100	2925400	52900	2324000
Run 5	2913800	52600	923400	2899100	67800	2348500
Run 6	2903600	52800	961500	2885800	54100	2316000
Run 7	2896300	55500	920700	2925200	54900	2369400
Run 8	2890700	53100	926600	2914400	57700	2439500
Run 9	2961000	58700	920500	2909000	53200	2496300
Run 10	2974100	53600	933300	2904100	55600	2172200
Run 11	2909600	53700	924600	3132100	55300	2933200
Run 12	2918400	53700	928400	2962600	55200	2552400
Run 13	3052900	53500	922800	2901200	52700	2140500
Run 14	3002400	53200	919700	2930100	56500	2361000
Run 15	3026100	53600	918700	2841300	53100	2445000
Run 16	2894700	55800	923300	2882500	53000	2109400
Run 17	3126000	53800	918900	2951400	54400	2391100
Run 18	2889100	54300	922700	3040300	65700	2394500
Run 19	2884500	53300	918900	3138500	60300	2331500
Run 20	2972500	55000	917600	2899700	52900	2526500

Table D.3: Breakdown of execution time for baseline circuit running on Valkyrie

D.0.2 Deutsch Jozsa with N=4 test results

Simulator	Valkyrie				Qiskit	Cirq
Processor	CPU		GPU		CPU	CPU
Mode	Statevector	Fast	Statevector	Fast	NA	NA
Run 1	25776000	21111600	35178400	27684900	48011900	58018700
Run 2	25188700	22717400	32515700	29334500	43009700	55586600
Run 3	25522000	20866000	33057800	27534600	37008500	61556300
Run 4	25184600	20692200	33644600	28148300	93020500	61738300
Run 5	24984400	21151700	32454300	26918300	36008500	59874100
Run 6	25934000	20764000	33476700	28000000	36008200	60362900
Run 7	25276400	20490400	33198300	27276900	36007800	59285500
Run 8	24901500	20786700	33402200	28704400	36008400	54227700
Run 9	24794600	20367700	32364200	29392600	36008000	61702400
Run 10	25316200	20595100	31468200	28817200	42010100	62221600
Run 11	25248000	20869300	33227800	29019200	37008100	56951600
Run 12	27230300	22055500	32585000	29222000	42633300	64682900
Run 13	25300500	21160900	32030800	27548800	1.15E+08	61486300
Run 14	25073300	20625000	32137700	27925000	41009400	63625000
Run 15	24602100	20844800	32874100	27937000	37008600	59163400
Run 16	24834000	20743800	32218100	28067500	42013600	57562400
Run 17	25257000	20463900	34249400	27912700	37008800	66039400
Run 18	25177400	20362300	33361000	26986400	41995800	67910600
Run 19	25629200	19954700	33552000	27317000	42010000	60402700
Run 20	25078800	20138700	33323100	27314300	38008600	57983900

Table D.4: Execution times for Valkyrie, Qiskit and Cirq for Deutsch Jozsa N=4 circuit using 20 iterations as initial test (Valkyrie not optimised)

Simulator	Valkyrie				Qiskit	Cirq
Processor	CPU		GPU		CPU	CPU
Mode	Statevector	Fast	Statevector	Fast	NA	NA
Run 1	24969800	21612200	39725000	28947300	37008400	55507100
Run 2	25170000	21532000	34530200	28697800	35008400	59103000
Run 3	26478600	20234700	32603000	28258100	35008300	59396800
Run 4	25935800	20814200	33070900	28045000	94021600	60084600
Run 5	25101600	20678800	33777700	28230000	35007600	60108900
Run 6	25109500	19860500	35105500	27680200	41009300	65279300
Run 7	25394500	20861700	33446900	27418400	36008400	54206900
Run 8	24919600	20985500	34227600	31559700	43008700	58052500
Run 9	25254400	20991800	33378100	27723500	43009900	60939500
Run 10	25024900	21279500	33646100	27740000	44010900	59218400
Run 11	25522100	20827100	34712000	28634000	37007700	60072700
Run 12	25336900	21861500	33194600	27344400	35008100	61360400
Run 13	24895400	20399500	32246300	38793100	1.04E+08	61046600
Run 14	24833500	20771100	36140200	28464100	35007900	61306900
Run 15	25401100	20521700	33980600	32737700	41009200	61502200
Run 16	25366200	20474600	33816600	27316600	36008100	58519100
Run 17	24976000	20719400	33256500	27285800	37008400	63803400
Run 18	24968100	21437200	32753900	27043500	43214700	59196400
Run 19	25220800	20784000	32906900	28038300	36008200	59034100
Run 20	25294200	20799800	33273800	27755500	44010100	60841200
Run 21	25109300	21055000	33401800	31407000	1E+08	59821300
Run 22	25400000	20748100	33328500	30437900	36008300	59306000
Run 23	25328500	21473600	33215000	30358000	34007700	59838400
Run 24	26036100	20903900	32470400	28879800	35008000	65095900
Run 25	28432500	20841000	33407200	30181700	35008000	56092000
Run 26	25236800	20861900	32392800	30030300	35007800	61441500
Run 27	25940700	21228100	32562900	31151200	44009900	61490300
Run 28	24974600	21358700	32705700	29257600	36008300	66489700
Run 29	25643400	21243000	33397700	29165200	1.01E+08	59113000
Run 30	25675700	21285400	33603000	29471100	35008000	58727000
Run 31	25325100	23600600	32560700	29169000	35007300	63590900
Run 32	25519400	20668900	33557300	27838300	35008800	56092200
Run 33	25508200	20349100	35163700	27793600	35007800	61273400
Run 34	25323600	20273900	33002800	27204200	35007300	67939300
Run 35	25780200	20353600	32933100	27661900	35008500	58071200
Run 36	25279500	20043500	34747500	26982000	35008400	62596600
Run 37	27637300	20603900	34631100	27870400	35007600	59373200
Run 38	26244500	20260200	34118100	27748200	1E+08	64997000
Run 39	25688700	20589500	35140300	26941000	35007500	59711900
Run 40	24769600	20197600	33051200	27076200	34008400	58685500
Run 41	24814600	19979200	33157300	27063500	34008000	63518600
Run 42	24425000	20079500	33360900	27668600	35007300	63426900
Run 43	24688100	20143100	32821800	27764600	35008000	60688900
Run 44	25090900	20277300	32604700	27935100	34007600	58214100
Run 45	24984000	20559200	32399200	27240300	35008200	62255400
Run 46	25052600	21605200	32211500	27472900	1.05E+08	60473300
Run 47	24347900	21672900	36289000	28206300	35008400	61858600
Run 48	24671500	20672700	33227800	28111600	34007300	58141400
Run 49	24531600	20722900	33215900	28593500	34007800	60393900
Run 50	24557300	20913600	32058600	29230300	35007900	60296400
Run 51	24379300	22465800	31512300	27749400	34007700	61600900
Run 52	24859300	22516300	32389900	28820400	35007900	64901600

Run 53	25024500	21437000	32129600	32852000	34007800	61683700
Run 54	24863200	21166100	32158700	27868500	35007900	55767500
Run 55	24663300	21332500	31705100	28702900	1.01E+08	69768200
Run 56	24871400	21935900	33528000	28135300	34007700	58403500
Run 57	25055300	22380100	35593000	27455100	34007500	58522300
Run 58	24796800	20966600	32513700	27431800	35008000	61878200
Run 59	25379200	20419100	32198100	27351500	35008300	53287900
Run 60	24926700	20551800	33053900	27125000	34007400	60476700
Run 61	25227400	20760200	32651400	27059300	35007900	60870500
Run 62	25736900	20512700	32569200	27257200	35007800	63822700
Run 63	24734900	20273800	35791800	27476100	98022700	60346900
Run 64	25149500	20127200	33186200	33354000	34007400	54895700
Run 65	25032500	20877300	33564400	27944300	34007700	57734200
Run 66	24582300	20823200	32213600	28486700	34007500	66418100
Run 67	25118900	20475200	32774900	27842400	34007900	72834000
Run 68	24595900	20137500	33172200	29278600	35007700	61172700
Run 69	24499100	20539400	31699300	28971200	36008400	58664700
Run 70	24472100	20243700	32102100	28504900	35008100	59615600
Run 71	24436800	19994100	33712100	29363500	1.09E+08	60433000
Run 72	24464700	20167600	35219400	30495600	34007200	58693000
Run 73	24436700	20176600	31830800	28395400	35007900	60125200
Run 74	24820600	20415900	32171800	27360800	34007900	59876100
Run 75	24954700	20597800	32502400	28638600	34007700	61104000
Run 76	24459300	20370900	32145700	27529900	34006900	58307400
Run 77	24415800	20145400	32539300	28097700	34008000	61177300
Run 78	24861200	20433000	32484400	27987000	35008200	56255900
Run 79	24589500	21055700	32444300	27678300	1E+08	60284100
Run 80	25532000	20416500	31973100	28191100	35008000	59090400
Run 81	24797000	20135700	32419600	28045900	34007700	61288800
Run 82	25391900	20070800	32434800	27666000	35007900	62834000
Run 83	24462200	20063200	32467400	28204000	34007500	54154900
Run 84	25001100	20537000	32753300	26977200	35008100	54990700
Run 85	24849400	20170700	32815900	27490300	35008400	62174200
Run 86	24388700	20574000	31984600	27868500	34007100	55598500
Run 87	24321400	20347400	31478800	27538800	95021900	53586100
Run 88	24607800	20577700	32422100	27972400	35007600	61656500
Run 89	24748600	20524900	32142200	26832500	34007600	67372500
Run 90	24475800	20560300	38094300	27596800	34008100	64324100
Run 91	24320600	20110700	33112900	27073000	34007500	67173900
Run 92	24235300	20000500	32458100	27252200	35007900	58772400
Run 93	24315800	20176800	35446500	27690800	35008000	63117800
Run 94	24578000	20314300	33128000	27516200	36429500	59401000
Run 95	24175400	20379300	33210800	27770500	34007900	58705700
Run 96	24239000	20262700	34397500	28433500	1.03E+08	58873200
Run 97	24820200	20222900	33678600	26760700	35007700	61263100
Run 98	24701400	20165000	32341100	27630400	35008300	59624000
Run 99	24399500	20004000	31772200	27010300	34007400	58063400
Run 100	24669200	20096200	32879500	27485200	35008300	68083200

Table D.5: Execution times for Valkyrie, Qiskit and Cirq for Deutsch Jozsa N=4 circuit using 100 iterations (Valkyrie not optimised)

Simulator	Valkyrie					
Processor	CPU			GPU		
Mode	Parsing	Staging	Execution	Parsing	Staging	Execution
Run 1	19918600	122400	5266100	20011900	131100	12842400
Run 2	19875200	129600	4851000	20132500	129900	13221400
Run 3	20202700	124700	4919400	20209300	229700	13432300
Run 4	20076500	134000	5811600	20303700	122400	13071700
Run 5	19398200	127000	4769000	19955100	122400	12951800
Run 6	20320700	130600	4857200	19819600	127100	12986000
Run 7	19854000	126000	4767000	19981700	130900	13200200
Run 8	20277100	127800	4911400	20102400	127100	12705000
Run 9	19852700	127500	4921700	20158300	127700	13476300
Run 10	19583400	125100	4763500	20779900	129100	13678200
Run 11	20916700	127800	4869800	20351600	129300	13452300
Run 12	19446700	129200	5063200	20248900	128800	13498000
Run 13	19674200	130000	4944200	20239200	130100	17704600
Run 14	20211900	129000	4903700	20134300	128300	12688100
Run 15	20053400	128000	4775400	20297700	125600	12295800
Run 16	19817200	126000	4817500	19617800	120400	12389800
Run 17	19447900	127200	4765700	19735300	121600	12249900
Run 18	20007100	121800	4893900	19560600	127000	13336100
Run 19	19932700	128500	4967000	19525300	120200	19988800
Run 20	20068000	129100	4944700	19537300	119600	13613800

Table D.6: Breakdown of execution time for Deutsch Jozsa N=4 circuit running on Valkyrie (Valkyrie not optimised)

D.0.3 Deutsch Jozsa with N=5 test results

Simulator	Valkyrie				Qiskit	Cirq
Processor	CPU		GPU		CPU	CPU
Mode	Statevector	Fast	Statevector	Fast	NA	NA
Run 1	37629000	24661700	49379000	36518700	90307500	70406100
Run 2	38694600	24729000	48951800	34443000	87794500	58936800
Run 3	38243700	24571800	48887400	32848600	1.16E+08	72795400
Run 4	39519900	24795700	49014500	32502500	87183900	72393400
Run 5	38684400	24917900	48645200	34108600	89270300	61484900
Run 6	43503200	25009400	48017700	34103000	88916400	53915300
Run 7	38367400	26545100	49111100	33941500	90643400	62745700
Run 8	39174800	24748800	49118900	34092400	90128700	71372000
Run 9	38447500	24455300	47458800	33763200	88633200	54491100
Run 10	38827000	24178100	47847800	39224100	88977100	60517700
Run 11	39035100	24710200	47440800	33116300	92788500	66738100
Run 12	38702600	23820800	46906600	33655900	89736200	66416300
Run 13	38404200	24334900	47728400	33859100	1.15E+08	59133500
Run 14	38613700	24525500	47476500	33261100	85974800	70676400
Run 15	38440900	23917900	50948500	32713400	91637000	67981000
Run 16	37690100	24982700	48127600	33354300	91040500	56386500
Run 17	37729200	24564400	49086100	34142600	89871600	58628600
Run 18	38564400	24832500	49675700	34725800	90130200	63760400
Run 19	39071500	25011700	47872900	43193400	90995200	69195500
Run 20	41684700	24960000	48383600	34935300	89406800	75329500

Table D.7: Execution times for Valkyrie, Qiskit and Cirq for Deutsch Jozsa N=5 circuit using 20 iterations as initial test (Valkyrie not optimised)

Simulator	Valkyrie				Qiskit	Cirq
Processor	CPU		GPU		CPU	CPU
Mode	Statevector	Fast	Statevector	Fast	NA	NA
Run 1	38694200	24066100	47865600	38530200	84054600	60336900
Run 2	38037800	24395500	47434900	36281000	90606800	42916000
Run 3	38652400	24358800	49587400	32245300	89307300	82340600
Run 4	38364400	24877500	48035500	36858400	89797400	56768200
Run 5	37606800	25020100	46991800	34068900	89336200	54048100
Run 6	38412800	25858500	49060800	33526800	89481200	76284400
Run 7	41343900	24683500	48807100	33746300	89746000	59626900
Run 8	38737000	24402200	48507600	33571800	88658400	66049200
Run 9	38589000	24671200	48939900	35135100	88583100	51762600
Run 10	38751500	24718700	46641700	34304200	90521800	53795800
Run 11	38393600	24663100	46897900	37938400	88513600	67452800
Run 12	38220300	24126000	47598500	33708600	86981100	64518200
Run 13	38870300	24120100	47329000	34297000	88921900	60777900
Run 14	38759200	24144100	47993100	33246500	90183000	65044600
Run 15	37770200	24187100	46134900	32756700	94612600	66271800
Run 16	38190100	24728900	46856100	33557100	87793000	62014400
Run 17	38079200	23941900	47277300	33888200	88131700	59781400
Run 18	37171400	24517500	48375900	33929300	89911200	65404100
Run 19	37842600	24721200	48236700	34244000	91176400	66181900
Run 20	37884400	29175100	48616500	34621900	89485300	57661400
Run 21	38131100	24414800	49001200	33862300	89335400	75964900
Run 22	38728300	24696500	48969600	33675400	90066500	68704700
Run 23	39372900	24617200	52366300	33833600	91550600	67588300
Run 24	38383000	24211400	50457300	33620200	88353400	60341500
Run 25	38273900	24275900	49148200	32571500	90381600	64340800
Run 26	39106600	24006800	54024700	33273200	88694900	48736800
Run 27	38472300	23718900	49775600	33349900	88813400	50776900
Run 28	38558700	25534500	48692800	32481500	90664000	59709200
Run 29	38472700	25059000	47394200	31974500	90842700	70659500
Run 30	38835400	24813700	61162800	32101400	88302600	80676000
Run 31	40994300	24604400	46685300	33204600	88998600	48590500
Run 32	40239600	24038100	49031100	33460000	90170600	55967300
Run 33	37427900	23770900	47890000	34599400	88172300	51033100
Run 34	37713300	23835500	48278300	36668600	1.17E+08	54687600
Run 35	38537700	23950400	47429100	35377100	91170700	73005900
Run 36	37439500	24010100	47785300	35276600	92177900	58514200
Run 37	37589700	23990700	47257100	34557100	88993700	66380300
Run 38	38012900	23995700	46999900	38300500	92110000	72114700
Run 39	37522000	23884200	46709100	33459600	89756300	73060500
Run 40	37508400	23903200	47241700	32985500	89123100	70086300
Run 41	37740300	24122300	47774000	32985700	88186600	54897800
Run 42	38968000	24365100	46521900	32599800	90718400	68928600
Run 43	37132800	24028600	49945300	37368700	89205100	67017000
Run 44	37912800	23885100	47566400	32549000	88258400	54430000
Run 45	37682600	23791600	51497100	32954200	89805600	63465100
Run 46	37753900	23998600	47522900	32881100	91610000	57364500
Run 47	37661300	24111300	47706600	31865400	90084400	60930100
Run 48	37636900	24485600	47242200	32181500	88111400	74531300
Run 49	37213500	24400900	47766900	33108200	90005400	59904200
Run 50	37095800	23973400	47488200	32284400	88271900	60508900
Run 51	37137500	24674200	47464900	32520400	92456900	64744100
Run 52	37271200	24626900	47470800	32617800	92733200	54586800

Run 53	37374900	24716400	48331200	32221300	88299500	58914300
Run 54	37312000	25171800	47039800	31852000	88772500	54750600
Run 55	37216100	24924800	47691300	31716500	90064200	53834300
Run 56	37122100	25324700	47786800	31665400	86407000	68053100
Run 57	37207100	24491100	47421700	32425800	86872300	52555400
Run 58	37712700	24719500	47440400	32557600	90483200	73566300
Run 59	38198100	24466100	47208200	33347800	91787200	59166900
Run 60	37453100	25955500	48152700	33857000	88952200	60670500
Run 61	37742800	24902000	46753300	33201500	1.14E+08	79275400
Run 62	37493700	25076100	50423300	32829800	90700900	60054200
Run 63	38465100	24774700	47474800	32488300	86960800	67389200
Run 64	37234100	24851800	47163400	32557300	89076200	64830600
Run 65	38195200	25989000	46400400	32962400	89209600	66563500
Run 66	37172700	26941200	48962400	32825900	91299900	44130700
Run 67	38048700	23891400	47494700	32092800	90150700	64381800
Run 68	37647400	24750100	48077200	44859500	89483400	73436300
Run 69	37164800	23953600	47147700	32878600	89048600	57163200
Run 70	37696700	24006500	46637100	32726100	89733200	69841900
Run 71	37603300	24312200	49544000	32135100	91014900	66988300
Run 72	37928200	23949000	47706200	31963400	88605600	63953900
Run 73	37168500	23947700	46894900	31880100	91252200	62048900
Run 74	37085600	23349400	46888200	32150600	88890300	69585300
Run 75	37860600	23923600	47534900	32229000	89372000	57327200
Run 76	38473800	23945800	47758200	33347100	89845700	56911000
Run 77	38287600	23737600	47694400	32180600	90817200	44888300
Run 78	38506100	24071000	46890800	31767400	87337500	55091100
Run 79	38298200	23551300	47257300	32352600	1.17E+08	69711800
Run 80	38209100	24528100	46880800	32104500	91219900	67174700
Run 81	38267800	23966000	46538800	31808500	87315700	61662400
Run 82	38114900	23860000	46760500	32405000	90533200	69760200
Run 83	37889400	23710900	47024300	33598500	89591300	77100500
Run 84	37994100	23927900	47883200	32793200	1.16E+08	68919500
Run 85	38317800	24187800	47561500	32189400	89429000	56151700
Run 86	38306400	23781000	46328600	31458200	90304800	68278700
Run 87	40056100	24341100	46311500	35731700	90812600	60466500
Run 88	37116200	24049900	48403700	32844000	89664000	62745100
Run 89	37018900	24173300	49524800	34918600	87150100	59587000
Run 90	37888000	23858800	47232600	37390100	87275600	76241300
Run 91	38010500	23683200	46390900	33433200	88703300	54986000
Run 92	37925400	23975100	55218700	33544600	89448100	66830800
Run 93	37658700	23883100	46947500	32991800	89221700	58737500
Run 94	37158300	23958200	46895300	34918200	87661100	63687900
Run 95	37411200	24501200	46559300	33587700	88811200	49938300
Run 96	37598900	24875300	48185900	33361800	89808200	67812600
Run 97	37311000	24621900	47140200	32816900	84910700	70134700
Run 98	37644500	24099700	46633700	33650200	1.16E+08	57475500
Run 99	37113300	23392600	46495800	35508000	89018700	52115800
Run 100	36948800	23988300	46751000	33001800	90331200	59753100

Table D.8: Execution times for Valkyrie, Qiskit and Cirq for Deutsch Jozsa N=5 circuit using 100 iterations (Valkyrie not optimised)

Simulator	Valkyrie					
Processor	CPU			GPU		
Mode	Parsing	Staging	Execution	Parsing	Staging	Execution
Run 1	23633500	151400	14464300	23715100	150700	28419200
Run 2	25202800	142800	14434700	23499800	144600	27148300
Run 3	23146200	142600	14214300	23624800	142800	34612200
Run 4	23414500	149700	14287400	23670200	148700	24885600
Run 5	23099200	144100	17229600	22935200	150500	27000500
Run 6	23374800	146000	14750400	23491600	165500	26013600
Run 7	23912100	141100	14683900	23857400	151100	27519600
Run 8	23400500	139500	14438500	26586000	145400	26244400
Run 9	23195400	140400	14537100	23474700	146100	26703400
Run 10	23325900	147600	14271600	24074800	142000	25902200
Run 11	23235000	147300	14543400	29800800	148800	25510000
Run 12	23288600	147000	14668600	23834700	141200	26861100
Run 13	23573900	146500	14655200	24397100	257600	24657400
Run 14	23456500	144100	14487100	23387300	150800	25902900
Run 15	23795600	142300	14767500	24455000	146400	25940600
Run 16	22643300	143500	14540900	23762900	151700	25595100
Run 17	22968800	142000	14502300	23001100	152900	29667400
Run 18	22573000	152400	14775800	23794000	151800	26256700
Run 19	23058200	143900	14741700	23131000	144200	28371200
Run 20	23754600	204600	14725000	22997400	147500	26038200

Table D.9: Breakdown of execution time for Deutsch Jozsa N=5 circuit running on Valkyrie (Valkyrie not optimised)

D.0.4 Deutsch Jozsa with N=6 test results

Simulator	Valkyrie				Qiskit	Cirq
Processor	CPU		GPU		CPU	CPU
Mode	Statevector	Fast	Statevector	Fast	NA	NA
Run 1	92547100	34445200	102712500	49120500	102947900	120964100
Run 2	95196900	34331900	101243400	47569200	102165100	96609600
Run 3	95377300	34716400	103770900	47821000	104467700	111437300
Run 4	93207800	35400500	103398700	45682900	104530900	107415400
Run 5	94639800	38054900	100198900	46045500	104185400	118320100
Run 6	94310400	34850300	101826500	47598600	135081900	109449500
Run 7	94105900	34259900	103290600	48590300	106006800	122488000
Run 8	94157200	33691100	101985500	49403600	105269900	107353700
Run 9	96668500	34478100	104779000	48442900	105195200	109083900
Run 10	98374600	34873000	105032000	47330200	106176600	108766500
Run 11	93932100	35172500	101698700	48224400	103615800	112554000
Run 12	92431000	37166000	102166200	48261400	104887700	92854200
Run 13	92757500	35367700	101858300	48259300	106008600	96950900
Run 14	91734000	35197500	117120200	48604100	107788100	109926600
Run 15	92631900	36045700	99746500	47751600	106689500	108385800
Run 16	91964000	34541100	105201300	47584400	103754900	110753300
Run 17	92726400	35119300	100256300	47996900	106599000	122784300
Run 18	91897900	34275500	100737300	47443700	106040100	93516100
Run 19	91406900	34726100	101795500	48999900	104750700	110057200
Run 20	91843700	35107600	100573300	49895500	105940400	101583500

Table D.10: Execution times for Valkyrie, Qiskit and Cirq for Deutsch Jozsa N=6 circuit using 20 iterations as initial test (Valkyrie not optimised)

Simulator	Valkyrie				Qiskit	Cirq
Processor	CPU		GPU		CPU	CPU
Mode	Statevector	Fast	Statevector	Fast	NA	NA
Run 1	93172500	35039900	101327500	53156700	104998900	110434600
Run 2	93440500	35090500	102180100	49443600	108397300	110305700
Run 3	95321400	34658400	99870600	51177500	104678900	114923600
Run 4	93219300	35590800	101309100	48485400	108078600	103842900
Run 5	92188200	39284900	102351300	47030600	109133500	90299500
Run 6	93476600	34975700	101770200	47019700	105258100	112872200
Run 7	91650400	35118600	102035300	47466100	101465600	94338400
Run 8	92919700	34736000	101278300	51528800	135792000	104966200
Run 9	93176200	34444600	103869700	45633400	106190400	95695600
Run 10	94448400	35448000	104624500	45403400	110515000	119938700
Run 11	94669000	37263700	101324000	45302000	103029300	111497700
Run 12	96936800	35938500	102365300	60644900	104727000	111334100
Run 13	92468400	36298400	100247300	49347100	103613300	117588100
Run 14	90637800	37398100	103039100	49197300	103769300	102125400
Run 15	91660900	35870600	99518000	47529900	106078600	105280900
Run 16	91700300	35243600	99264400	48332400	107053900	90525900
Run 17	92389200	35501800	100978300	47154900	99824600	118965100
Run 18	90310500	35805700	99032400	48370500	107820900	106264700
Run 19	90620400	34741200	98152300	47173200	106692800	101870600
Run 20	90972300	35497300	99490900	47571100	104680700	104799000
Run 21	90800000	35526400	99506500	47263100	104084800	93354000
Run 22	91571900	34679000	100632900	46310200	104343400	128002600
Run 23	91512400	35428700	101401300	46779000	104865000	103786200
Run 24	90552100	35683100	99131900	48438800	134491300	122977500
Run 25	90751400	35892200	99614600	47506200	106498900	130571100
Run 26	91243600	35947100	100072400	49740700	105706600	110524600
Run 27	91073000	36040500	98365900	49418100	102483600	105086900
Run 28	92576700	36157600	103283500	50310200	105397800	109646300
Run 29	90529000	35696400	101334700	50109600	106003600	116266100
Run 30	90755700	36784100	99581200	46395700	105282300	106537600
Run 31	91076200	35952100	101784200	46222100	102330100	102911200
Run 32	90331600	35934000	100131900	50038400	100356900	108998700
Run 33	90528100	37766400	101393000	46013100	108097100	109314500
Run 34	93184500	35668000	100444500	47535600	105307600	140681700
Run 35	93661300	37945000	102383100	46588200	107480300	120632200
Run 36	92955500	35392900	99007700	46955700	139988300	114255500
Run 37	93303900	35205000	101396200	45660500	106054800	108334600
Run 38	92077300	34874400	99981600	46335500	105394600	116705400
Run 39	92239200	34458300	101961500	46467700	102880300	121723400
Run 40	90887200	34807700	101181000	46290500	103846800	115784200
Run 41	90594700	35950800	100160000	45536800	105179700	118280200
Run 42	90755800	34790900	99390400	45804100	103924500	112017700
Run 43	92216200	34939500	98609000	46714000	103348200	113400000
Run 44	91087300	34668700	99084500	45039100	101107700	116109100
Run 45	91802300	34460000	99500200	45517800	110915600	121495900
Run 46	90831300	34056700	99897700	45526400	105399000	126015300
Run 47	90672800	34213700	99174300	45437400	103637800	116197200
Run 48	90865600	34329700	98519400	45335800	106453500	99634500
Run 49	90991400	34898000	99974200	45243300	104492300	101437900
Run 50	90975200	34600000	102426600	45694800	104629900	121055500
Run 51	81590200	34265700	99688500	45490800	102888300	100816400
Run 52	93933300	34094900	98692700	46603200	106937400	107340600

Run 53	82053900	34261600	101938800	45247900	105519600	93684900
Run 54	81787500	35369900	99629700	45615700	104515800	120121500
Run 55	82719200	34489100	98635100	46712600	104235700	121017300
Run 56	82164100	34625400	100062300	47180700	106353700	129313200
Run 57	81794400	34424900	100085500	46055400	105000200	129885000
Run 58	89951000	34658200	99317600	46696600	103671100	100578100
Run 59	85645200	34092900	101285300	47591600	99322800	99448400
Run 60	81869100	34840500	101603000	45481900	104598100	85056000
Run 61	90587100	34493200	101452400	46752000	106156800	80170800
Run 62	82275800	34903500	98788500	46924500	103647200	107530300
Run 63	81371200	34528200	99044000	45450500	105207600	106635200
Run 64	81235400	35120100	98306600	50062000	105175200	114583800
Run 65	81704900	34527300	99311700	48580300	104248900	117350700
Run 66	90674500	34422500	99289600	48654400	103955900	99365400
Run 67	92367600	34603500	98544100	48016300	103739500	113766200
Run 68	91523500	34349300	99320400	47198600	103398600	129276800
Run 69	90689900	35867800	97940700	47337600	105433200	102017100
Run 70	89985100	35033000	99352700	46125700	104341700	103650700
Run 71	90088500	34774200	98600800	47932300	107102800	94892900
Run 72	89735200	34166900	99094200	45841300	105232400	96417100
Run 73	91389400	34342800	99324900	48180700	105092100	114793600
Run 74	81185600	34778500	100101100	46244400	104778300	106385200
Run 75	81202100	34361500	98937300	46072700	109377800	86601600
Run 76	81728500	34701200	100029700	46281900	106119700	97160000
Run 77	81367800	34994900	98750200	46779200	108909100	101369300
Run 78	81555600	34893500	101379400	47596400	107954400	102452500
Run 79	82695200	33976700	99867700	45919200	105518500	93674300
Run 80	90171600	34092300	102089100	46661000	105758300	121781800
Run 81	81601600	34087400	100380500	47060300	108180400	106906800
Run 82	82200800	34081300	98680300	47092200	104197300	94143400
Run 83	90514300	34410900	99797200	46148900	106008300	94570200
Run 84	85349000	34464500	102059700	45083100	109356900	105112200
Run 85	82766600	34606300	117307700	44787800	106659700	139841900
Run 86	81654300	34005400	99570800	46081600	105725400	88272600
Run 87	81555700	35943800	101370800	46028600	103633900	127296700
Run 88	83853800	35278400	100241300	45299300	105961300	98086900
Run 89	82535400	35832700	98760900	46436600	107937000	112289200
Run 90	82140300	35250300	100462300	46302000	103973200	107451500
Run 91	90524900	35913700	99172400	45687500	109814300	104126800
Run 92	89964000	35316600	97990600	45688200	106534100	100507200
Run 93	81407900	35212300	99355200	44875300	106729600	111530900
Run 94	81355200	35968600	100218800	45494400	103601700	110389400
Run 95	81063500	35458000	100923200	46195300	105502200	117883000
Run 96	84640800	35280800	101118300	45439600	107819200	118169200
Run 97	83028300	35407000	99476800	45588800	104870800	96820800
Run 98	81850200	35639100	102738200	47545300	103306400	110444800
Run 99	90137200	34790600	101310400	45825300	103518400	98947000
Run 100	81670800	35230600	100479700	46857800	105631700	126051900

Table D.11: Execution times for Valkyrie, Qiskit and Cirq for Deutsch Jozsa N=6 circuit using 100 iterations (Valkyrie not optimised)

Simulator	Valkyrie					
Processor	CPU			GPU		
Mode	Parsing	Staging	Execution	Parsing	Staging	Execution
Run 1	33137700	186600	60755300	36101500	206800	72348000
Run 2	32467200	191100	60035900	33047000	206600	69298300
Run 3	31641900	216900	60900200	32853200	204500	70640000
Run 4	31488000	194700	62298900	31449900	206500	68143300
Run 5	31208000	199600	60654000	33213200	193100	70500600
Run 6	31590700	197300	60964200	32751100	206900	72049600
Run 7	32629600	205100	60762500	32712500	197700	72115700
Run 8	31890400	214200	64870800	32690300	210200	68213700
Run 9	31437000	198800	59327800	33308800	204400	68741600
Run 10	32613900	204600	59031200	32073200	210800	68082800
Run 11	32460900	206100	60970200	32406300	205700	69806900
Run 12	34531500	200400	59105400	32258900	203800	68194100
Run 13	32063800	189000	59272500	35154200	203600	69647200
Run 14	31693100	188100	58931000	32099100	197900	73040000
Run 15	31533000	198200	61246100	32092900	200500	68150400
Run 16	31103400	193600	59000600	31635200	202100	68410900
Run 17	30979200	199200	58920300	32608200	196300	67276300
Run 18	30938300	192200	60325200	31827000	196900	68236300
Run 19	31144000	188900	59005300	31354500	207000	67555600
Run 20	31481900	189900	59070300	31737000	196500	67615800

Table D.12: Breakdown of execution time for Deutsch Jozsa N=6 circuit running on Valkyrie (Valkyrie not optimised)

D.0.5 Deutsch Jozsa with N=7 test results

Simulator	Valkyrie				Qiskit	Cirq
Processor	CPU		GPU		CPU	CPU
Mode	Statevector	Fast	Statevector	Fast	NA	NA
Run 1	182433100	40798100	222049700	57528500	126826300	131450900
Run 2	178791500	40549100	197782200	55809000	129656200	132991600
Run 3	176782800	41377600	202951200	56344100	130361700	135750700
Run 4	179407100	42919600	202835900	56515900	128702400	125567100
Run 5	179681600	41757200	212384000	56214200	127186200	136728100
Run 6	182653800	40885600	206921000	55953200	128730300	129810500
Run 7	175132800	41062100	202952300	54762100	127340300	134005000
Run 8	174961400	40529700	209226700	54481300	127114400	135014100
Run 9	175006400	41513300	200503900	60399600	127589400	128775100
Run 10	173837600	42364700	200317400	56210400	129640900	134570400
Run 11	173532800	42574200	199791100	55648900	130746000	127506000
Run 12	174876100	41854200	200173100	56508400	128236700	132336400
Run 13	173878300	43565300	196975800	54800300	127233200	135026000
Run 14	180275900	41790200	198700700	53886200	167630500	134067600
Run 15	174170600	41472900	199886400	53181900	127880900	132295400
Run 16	173070200	45863700	198568700	52749000	125947500	128185400
Run 17	174122300	41636500	199718700	53150800	129334600	130572900
Run 18	178467800	43952400	200660600	54014600	128648600	133681900
Run 19	175834300	42049500	196400200	54518200	128833800	132167300
Run 20	91843700	41831600	204174000	56756300	130021200	132061100

Table D.13: Execution times for Valkyrie, Qiskit and Cirq for Deutsch Jozsa N=7 circuit using 20 iterations as initial test (Valkyrie not optimised)

Simulator	Valkyrie				Qiskit	Cirq
Processor	CPU		GPU		CPU	CPU
Mode	Statevector	Fast	Statevector	Fast	NA	NA
Run 1	174992700	41042700	200084200	55610000	128107300	135480800
Run 2	175188900	41652100	201052300	54419200	130259800	130509700
Run 3	176527400	42029600	206633800	56371000	128388200	133752900
Run 4	177601500	42156200	209199100	56656900	130313400	137581200
Run 5	175844400	41309000	205183400	58530700	129442300	131533700
Run 6	172576600	41348100	201449700	56105200	127234400	132542800
Run 7	178142500	40801500	201978200	56699400	126035200	130225900
Run 8	178242600	42576500	203708300	54594800	130374800	133046600
Run 9	177687000	42049200	204150100	55126400	166043500	131369400
Run 10	178460100	40611700	201956500	53987500	128929600	129375600
Run 11	170291500	41409300	202131500	54282900	129954100	133641600
Run 12	180050600	41592400	201811500	54443300	126056900	133958700
Run 13	176203600	41677300	199901200	56396500	129564900	131036500
Run 14	176469000	41735100	198609500	53446000	127531500	133178800
Run 15	171638400	41935500	199155300	53425000	167726400	135375600
Run 16	175873100	41970600	205415500	53539000	135326200	128873800
Run 17	171266400	41708500	206746400	54750900	128075900	131160200
Run 18	174604100	42179900	199105700	58929600	130875000	136520200
Run 19	174713600	41025500	197410500	53552900	124936700	129424500
Run 20	173396600	40874300	197893800	52561800	128220800	128431100
Run 21	171389600	41521300	200355400	52490800	130582400	132296400
Run 22	171201400	43164100	196623300	52757700	128410300	135286100
Run 23	171777600	42079100	197901200	53742900	128587500	128799400
Run 24	171700500	43614700	199599400	61117900	132100500	135339500
Run 25	171384400	41867400	199531100	53735800	128854600	133807400
Run 26	170251300	41307800	197737800	53793500	130669700	133432600
Run 27	170454600	42587400	199626900	52310500	130659300	130038200
Run 28	168299000	40805200	198924400	52506100	132757100	132980500
Run 29	167148900	41122800	197495400	52538800	170359400	134671100
Run 30	167653800	40648000	196477000	53411800	125953500	127605900
Run 31	168844100	41454800	199057400	54983000	129532100	134183400
Run 32	167653000	40525100	197569600	52918600	125231500	136141800
Run 33	169269700	40785500	197677300	53385800	127620200	129782200
Run 34	167447700	40792800	203295500	53709500	130547600	132509200
Run 35	170147300	40025800	198743000	52807500	125775900	132248700
Run 36	168413100	42190600	196888700	52304000	131573600	130156000
Run 37	170995400	40242400	199348600	53175300	129824300	133474600
Run 38	172055300	41391800	198546000	54790100	171735100	133729300
Run 39	176263700	40770100	198391800	52451300	127774500	130149200
Run 40	172094800	40472700	198958100	52599900	132776900	129418100
Run 41	172737100	40384600	196134600	53376700	131663900	130314200
Run 42	168859200	41208700	202336600	53104900	129481800	130677700
Run 43	170763300	40788800	198131200	58778100	128889000	130355400
Run 44	168657700	40833700	197598900	54469200	130057000	135300700
Run 45	170614200	41199500	194061100	54586700	129073400	140537100
Run 46	172352800	40451500	197736300	54525900	165937900	131489500
Run 47	171128600	40589100	196248600	55609300	129832500	132543800
Run 48	168312700	40693700	194759000	55134200	130558300	133673300
Run 49	178030000	40598400	196912000	54337500	129046400	130935600
Run 50	171814100	40344300	195977200	53893600	129095200	136285700
Run 51	172980400	40784800	216724300	53903600	125454200	129535000
Run 52	170908800	40586900	201166500	55322600	129627500	133535500

Run 53	170822200	40218600	199040100	54597800	126093500	128603400
Run 54	170184400	40271900	197019400	55253900	129764100	131583800
Run 55	170245900	39997200	198669200	52792100	130558600	130499700
Run 56	172318900	40712400	215209300	52947000	128107200	133125500
Run 57	180007400	40294600	205580000	53330500	126410400	135402600
Run 58	170892100	40048800	196317800	53727000	129674800	131638900
Run 59	171603100	40439800	197203800	53009400	131943800	131704000
Run 60	168951800	41354500	196165300	53005800	129581700	130850800
Run 61	168309600	40193800	198528700	52962600	131920500	132382200
Run 62	171705200	40580500	200186400	53426000	124188100	133755500
Run 63	170958500	41001000	195118200	59216100	126338800	131856100
Run 64	168441500	41554500	199469000	56191400	129838500	131469400
Run 65	170632800	40719200	196104200	53099400	127855000	133397500
Run 66	171561200	40604900	197920200	54029400	126097900	133495300
Run 67	173655600	40668700	197207800	54345500	128361900	132613100
Run 68	172772500	40910100	196164300	54421200	129364000	131362400
Run 69	170404000	40128300	197166900	52457400	128829100	133504600
Run 70	170487400	40112100	198577200	55133100	128761500	130228800
Run 71	171975800	40637100	196194600	54534900	131968300	130108000
Run 72	170875900	41020100	201948500	56393500	126555700	134012200
Run 73	177090000	40354500	197748300	54601000	131584600	132936800
Run 74	177951500	40551400	197767100	53241200	133176500	129462700
Run 75	169938700	43550600	197750600	52653300	130513300	130240600
Run 76	171552500	41759400	197878800	54040900	129176100	130889800
Run 77	172690400	41316100	197466000	52778500	132410500	133080900
Run 78	169651700	41572800	199242800	53977100	132935900	136650900
Run 79	169122300	41986700	198518600	54205200	129223000	129570000
Run 80	171431900	41113500	196116300	52802200	168485200	131539100
Run 81	175795500	41430100	197934700	54329700	133625000	134072700
Run 82	170364000	41446400	198186000	52867400	130612800	129762300
Run 83	170347900	40147500	197150300	57872300	128075400	132662000
Run 84	170753900	41868500	197433000	52517400	128306000	133827100
Run 85	168437400	40588700	196260400	52693600	127877900	131569800
Run 86	170083300	40849200	196042900	55233800	133047500	132278800
Run 87	168340600	40594700	197781600	54539100	126805700	130908800
Run 88	171513400	40562000	195555600	56698000	127326400	132056400
Run 89	169959200	40906600	195475500	58799800	127796700	132252800
Run 90	169427000	40108400	194808400	37390100	169638000	132980000
Run 91	168463800	40918400	200288300	33433200	128217000	131233200
Run 92	168260700	40440800	196688600	33544600	130096900	131934800
Run 93	168596400	40470100	196881000	32991800	131196700	131915500
Run 94	169725500	40012000	195250600	34918200	131603000	131094800
Run 95	167809200	39908900	195736600	33587700	131360700	135794300
Run 96	169172900	40511600	196033100	33361800	129061900	129024400
Run 97	170993600	40411200	196969900	32816900	169127700	130198000
Run 98	177591900	40411200	196894700	33650200	126832900	133581400
Run 99	169468600	41389800	197469200	35508000	126892200	129151000
Run 100	171090800	40489700	200042800	33001800	127231600	131716800

Table D.14: Execution times for Valkyrie, Qiskit and Cirq for Deutsch Jozsa N=7 circuit using 100 iterations (Valkyrie not optimised)

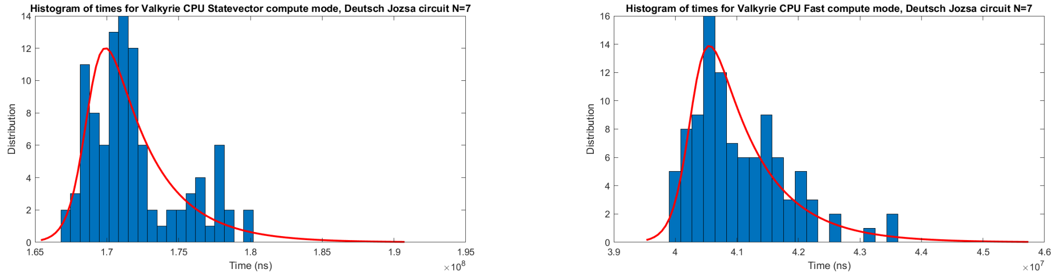
Simulator	Valkyrie					
Processor	CPU			GPU		
Mode	Parsing	Staging	Execution	Parsing	Staging	Execution
Run 1	42058200	225100	140002600	36099800	220700	172053500
Run 2	36182600	226800	140080100	36592900	224400	166707900
Run 3	36068200	222400	140437600	35974800	233500	166839200
Run 4	36354800	216000	142437900	35376800	233900	173976500
Run 5	36740800	228700	147485200	35227400	219900	168700700
Run 6	35174500	205400	137626000	38207700	230800	170606800
Run 7	36573100	219500	137934800	36527000	216200	165803700
Run 8	35376600	236900	138408600	35595300	222000	162733200
Run 9	34825900	238800	137193500	35329700	228900	162568400
Run 10	34819200	222800	138457500	35455500	209200	165331300
Run 11	35020200	225400	140108200	35406300	213900	164285600
Run 12	34907800	209100	138385900	35069900	219200	164550700
Run 13	34792600	228700	138355700	35054600	227100	163086000
Run 14	34589600	214800	137169900	34993900	216400	165396400
Run 15	34380000	222600	138620800	37718100	232200	166711400
Run 16	35523700	204200	139420100	36064300	227300	168833800
Run 17	35075700	218600	140552200	35821000	229400	164814500
Run 18	35559200	225600	139276500	35014100	220400	167451000
Run 19	36042200	230100	137310000	34975200	224400	163276900
Run 20	34730000	222500	136862700	35404700	218800	164665500

Table D.15: Breakdown of execution time for Deutsch Jozsa N=7 circuit running on Valkyrie (Valkyrie not optimised)

D.0.6 Deutsch Jozsa with N=8 test results

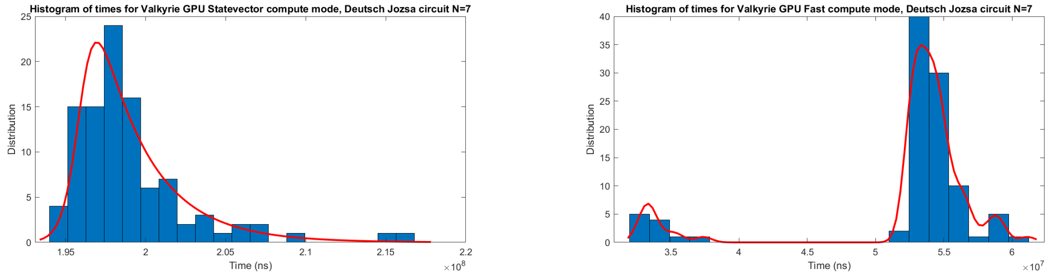
Simulator	Valkyrie				Qiskit	Cirq
Processor	CPU		GPU		CPU	CPU
Mode	Statevector	Fast	Statevector	Fast	NA	NA
Run 1	533673400	54623600	451356800	70992600	181724500	171414300
Run 2	522858200	55890900	460246700	72174900	181690300	168345100
Run 3	516063100	57557800	440939400	74240900	174632200	168266800
Run 4	514300600	56485000	446556200	70875000	176587600	169273700
Run 5	517151200	56387200	442897400	73114500	179049300	171760300
Run 6	522423100	55828000	437897600	72467600	187254800	170607000
Run 7	515182600	56319900	448936000	73289900	176550800	171193900
Run 8	529854900	56324800	443482900	72043100	177153600	166449900
Run 9	514342900	56315100	442403300	71746200	175879700	171397200
Run 10	521394200	55933700	442710400	74248600	178230600	171474800
Run 11	514186400	56722000	435210000	75045100	174774600	171222700
Run 12	512510100	57053900	438497900	74912400	178383400	177197200
Run 13	521854300	57094200	433201100	78715300	183098900	170778800
Run 14	515288800	56754800	433516400	77014400	181222900	167088300
Run 15	517927000	58480500	439658900	71359400	184207500	170587000
Run 16	515166300	58472300	431151700	71957200	185418600	169971100
Run 17	517487900	58202900	434158700	71682500	177857900	173379200
Run 18	516961500	55186200	431766900	70471100	180594000	162553600
Run 19	515644500	55387800	431441800	69464000	183202200	171482900
Run 20	520345900	54368100	429678900	71781900	185950100	168652300

Table D.16: Execution times for Valkyrie, Qiskit and Cirq for Deutsch Jozsa N=8 circuit using 20 iterations as initial test (Valkyrie not optimised)



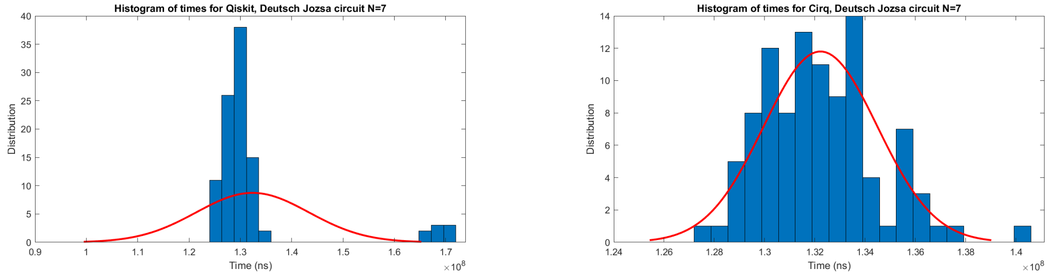
(a) Histogram for Valkyrie CPU in statevector mode

(b) Histogram for Valkyrie CPU in fast mode



(c) Histogram for Valkyrie GPU in statevector mode

(d) Histogram for Valkyrie GPU in fast mode



(e) Histogram for Qiskit

(f) Histogram for Cirq

Figure D.1: Histograms for the distribution of execution times for various Quantum simulators with Deutsch Jozsa N = 7 circuit

Simulator	Valkyrie				Qiskit	Cirq
Processor	CPU		GPU		CPU	CPU
Mode	Statevector	Fast	Statevector	Fast	NA	NA
Run 1	530169800	54717400	448073500	74014200	175874100	172013300
Run 2	525867100	54372600	450618900	71309800	177120100	167738800
Run 3	526137000	55875400	449956100	72127700	182185700	168065900
Run 4	520460500	55433900	438653300	71645200	186901000	171559400
Run 5	525712600	54188500	445140200	72407200	176395200	170770500
Run 6	525597000	56111200	435431500	72717200	180288700	174044400
Run 7	530606900	55500500	450492800	71947800	188881400	169971800
Run 8	525332700	54163800	443930900	74792100	176363100	171785700
Run 9	532168400	53992000	440441000	73187600	181044800	166836900
Run 10	516582500	55933000	440321100	74582000	184999300	169669100
Run 11	529683100	53863800	435544200	70403400	220433100	170440900
Run 12	519558800	54403900	438695800	70607400	178396400	172475300
Run 13	517546400	55738800	435428800	70297500	178386800	168914400
Run 14	517059400	55436900	439689800	72128400	187865100	171729400
Run 15	514249100	56245600	433876900	72539000	173533600	172214500
Run 16	514785000	55812400	435337500	76284800	215995700	172368600
Run 17	514526700	55288700	434152500	76793300	178429900	166851100
Run 18	517661300	53940000	431493000	85435400	183062700	171323600

Run 19	515290400	53651500	436286000	70031100	181565400	172419000
Run 20	512808500	53822700	433196000	71094900	182624200	172111600
Run 21	513759400	54206100	432314000	71141300	175334600	172703000
Run 22	511889600	54412400	434482000	70906300	178359700	169202200
Run 23	520321500	55549100	439275700	70279100	179487600	167929600
Run 24	514365200	57030300	433971900	72513100	183754700	166549900
Run 25	513272800	57977200	433306600	77088300	173344600	173706500
Run 26	511204900	55521600	434231700	70849100	176698500	167122000
Run 27	510623000	57282900	433807500	70232200	185291000	177432700
Run 28	514724800	54269300	434706300	72179400	187969000	174009100
Run 29	509907400	54656500	431823700	71378800	179046300	165748200
Run 30	515502300	54596100	434438900	70249000	185210900	167048000
Run 31	541017600	54714300	433886600	70338100	182248400	177690300
Run 32	520374300	53787900	438496900	70084200	175427000	168757000
Run 33	519625200	54146200	432058800	69487100	184367300	173981500
Run 34	520890400	53508900	431511400	70485800	181952700	174762100
Run 35	515623000	53900700	435368400	69401200	174649800	170505300
Run 36	512297700	53994800	430743700	69903300	179398900	168145500
Run 37	532463200	53694000	434197200	70040700	184056500	172548400
Run 38	512300200	54579100	433389700	69599200	179545700	170977900
Run 39	518242500	54398300	431778800	68718100	181083500	169895000
Run 40	513964300	53679200	433043400	73507000	181533100	170654400
Run 41	511445400	54663500	437103600	71158100	178664100	171609100
Run 42	513098700	55566900	432097300	76105900	175301900	163214000
Run 43	512324800	55734100	434660200	71802900	182591500	173855500
Run 44	512209500	53747300	435513800	72426700	184750000	174558600
Run 45	513519700	53914400	431736000	71317500	180599500	171707100
Run 46	509861300	53744500	434664100	70440700	180241300	170787300
Run 47	513806600	53971700	429738800	70611600	179950600	169236300
Run 48	515710300	54095100	438272400	70834600	172535500	167906300
Run 49	510243200	54230600	434560700	69408700	184160200	169888300
Run 50	511548200	53397300	433505900	70294400	180741500	171637400
Run 51	515468400	54820800	432970600	68970300	183530500	175236300
Run 52	512651600	53680300	433410300	70029700	178943400	168805900
Run 53	512029400	57723000	434507300	69802200	175201500	171946800
Run 54	512263500	53970600	430995200	69919100	178360100	168108200
Run 55	515978800	54177800	434690700	71299200	176218900	170935400
Run 56	511702100	53040800	431192800	70292700	182064200	170435600
Run 57	514105900	53753700	431500100	69988800	174321400	170371400
Run 58	510183300	56031400	434957700	71203100	176980300	168618800
Run 59	507836200	55344900	426889100	69620900	183100100	171360200
Run 60	513650400	55865300	434030800	72930300	178637900	172990400
Run 61	513075600	55655700	429031000	69612500	184214300	170313600
Run 62	513485200	55119500	429887100	69840100	181577600	167716400
Run 63	515442000	55732800	432068200	69881500	183716600	167926500
Run 64	514527400	54562200	430619500	70665600	179183800	168498200
Run 65	510679800	55158700	433284700	68586500	220204800	168761000
Run 66	510024200	53964300	430080000	70031300	179133200	169940400
Run 67	513798000	53850300	434882900	69044300	178932800	162601200
Run 68	513362000	53756800	429877900	70434000	175447900	166976400
Run 69	513558500	54399600	434584600	69831200	181726600	171022900
Run 70	510923600	53623900	438617100	71375900	178979200	174829700
Run 71	515312300	53269500	430966200	70021300	175271100	166484600
Run 72	512835700	54139300	436952200	70159100	178055200	171173400
Run 73	512273300	54194500	430217500	69032600	184794900	171104400

Run 74	512130200	53989100	436867400	74878100	182780600	170264500
Run 75	513464700	53493200	430162000	71993900	216400500	171826700
Run 76	513954700	55260300	433156000	70257000	216038900	166451600
Run 77	513390700	54704300	432582600	69664800	175984000	171833000
Run 78	515869500	54256600	431683300	70551700	180098700	167566100
Run 79	512111200	54637300	434499800	68514800	173438700	173235200
Run 80	511119300	53825700	431881300	73431200	182490000	168574800
Run 81	512890900	54844400	433096600	70604800	180093200	174606300
Run 82	511110400	53726300	429652400	69877100	178405000	168774800
Run 83	517451000	53720600	433970200	68914500	179125500	170803800
Run 84	512173800	55048900	433927500	69975100	178103500	169898400
Run 85	522427800	54069500	432544900	69026600	185347700	174692900
Run 86	514210600	54010300	434457400	70586700	180843400	170095500
Run 87	511679700	54338800	429498400	69087400	178187500	169375400
Run 88	516362400	54782900	433153100	69725100	184547300	171916800
Run 89	511370700	54007400	431013400	69869600	184851000	167113100
Run 90	517148600	54633000	433803300	69263000	179918400	172477900
Run 91	512505800	54413500	432505700	69639300	178002800	167446600
Run 92	515348300	53811100	435027400	68970300	185863300	167672000
Run 93	511355800	53163600	433619400	70389100	183714300	172119800
Run 94	514774400	54116200	430320900	69003900	185304700	168741500
Run 95	515956900	53918700	434634700	70604200	178815300	169585700
Run 96	510779200	53657300	432832900	68985800	179416400	174087100
Run 97	515438800	53595000	430858200	69013800	183226300	166518800
Run 98	513460700	53230600	432464300	69847100	178676200	167136000
Run 99	512180800	53691000	433544400	69910400	179297400	167318700
Run 100	510069300	54215500	432255400	69793200	175422600	170725000

Table D.17: Execution times for Valkyrie, Qiskit and Cirq for Deutsch Jozsa N=8 circuit using 100 iterations (Valkyrie not optimised)

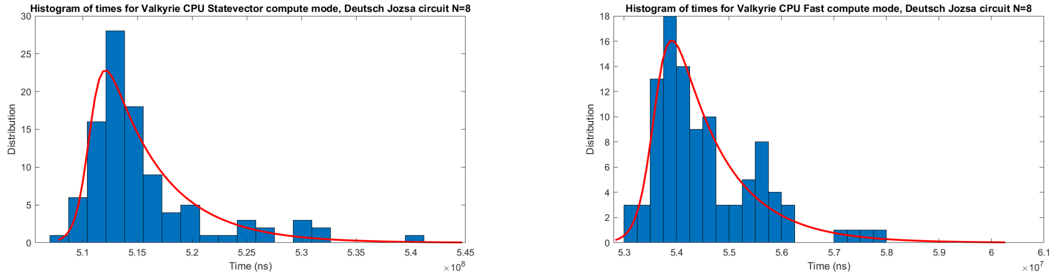
Simulator	Valkyrie					
Processor	CPU			GPU		
Mode	Parsing	Staging	Execution	Parsing	Staging	Execution
Run 1	42898800	290000	499647200	43000400	263400	406780800
Run 2	41837500	262400	494286100	42240700	269500	411767400
Run 3	41089400	273700	490478800	41361400	241500	398844700
Run 4	42113300	279500	489144400	43771000	252400	400960800
Run 5	41165800	266600	493516900	42369300	244600	397540300
Run 6	42601600	279200	487367600	40632400	262800	400025500
Run 7	41426100	265700	491146400	43304900	245100	408612800
Run 8	42499600	267800	490874800	40492300	257900	400155000
Run 9	40985400	381100	487691600	40844800	248200	399790900
Run 10	44333600	264700	487010600	41782000	263900	396104100
Run 11	41279400	261800	484953400	41422500	250600	394207200
Run 12	40901300	259600	484476200	40701300	239300	391535900
Run 13	40470600	273200	474169100	40546900	248200	393745300
Run 14	40597500	269600	474974900	41012600	244100	394210400
Run 15	40700600	278600	479048500	40571900	236300	393793300
Run 16	41557600	253600	490065800	40605000	237900	388871000
Run 17	40829200	263300	481846000	41280500	234800	395460700
Run 18	41510600	260000	480820300	40890000	233600	392145400
Run 19	41149800	320200	486857000	40794100	237300	386862800
Run 20	40854000	259700	480148800	40698800	231100	389800900

Table D.18: Breakdown of execution time for Deutsch Jozsa N=8 circuit running on Valkyrie (Valkyrie not optimised)

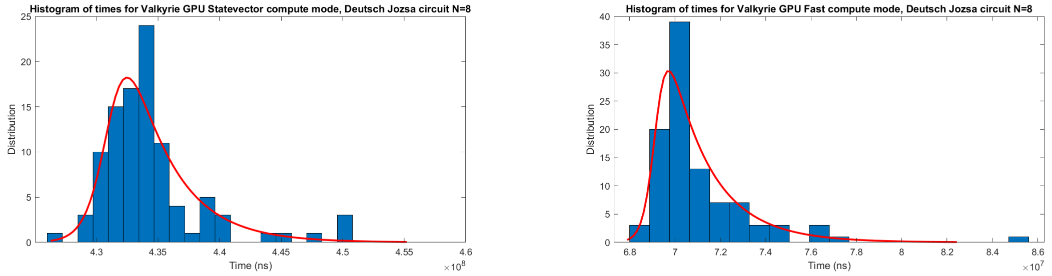
D.0.7 Deutsch Jozsa with N=9 test results

Simulator	Valkyrie				Qiskit	Cirq
Processor	CPU		GPU		CPU	CPU
Mode	Statevector	Fast	Statevector	Fast	NA	NA
Run 1	1390866400	75836000	1227966200	98457500	274006900	194716200
Run 2	1382223500	77934800	1214628100	93162600	204011400	187578800
Run 3	1370427200	77420900	1293413700	96202600	197208200	180088300
Run 4	1378695400	76789900	1288667800	1E+08	202844000	187499800
Run 5	1379947700	77332200	1288884200	96189500	194027100	180390200
Run 6	1376882600	76605100	1286993100	97950600	196454200	184194100
Run 7	1378387400	79585700	1279764600	95443100	204321100	182853000
Run 8	1392238100	75869800	1292851100	98583200	279575500	187835400
Run 9	1378691100	77454600	1287768800	1E+08	277338400	190959800
Run 10	1376753700	78246800	1283096900	99839300	190559400	194653800
Run 11	1384834900	76280000	1273948200	96213900	200697000	184428300
Run 12	1375100100	78750000	1276896500	95886000	197034600	192241800
Run 13	1376875200	77869100	1261351200	93484500	204041800	189339100
Run 14	1376351700	80144100	1256279900	94826500	200817700	191523500
Run 15	1412943800	78448800	1143707500	93515500	200737600	184152900
Run 16	1386101800	75229900	1262206000	94688200	207155200	168763600
Run 17	1387200900	75154700	1249134300	93347500	204798200	183430100
Run 18	1379526100	75083700	1246168200	94049200	209106900	194588300
Run 19	1374413500	75169100	1247266200	1.04E+08	201752300	192261600
Run 20	1374723100	75559000	1177674000	93915900	198720000	188965800

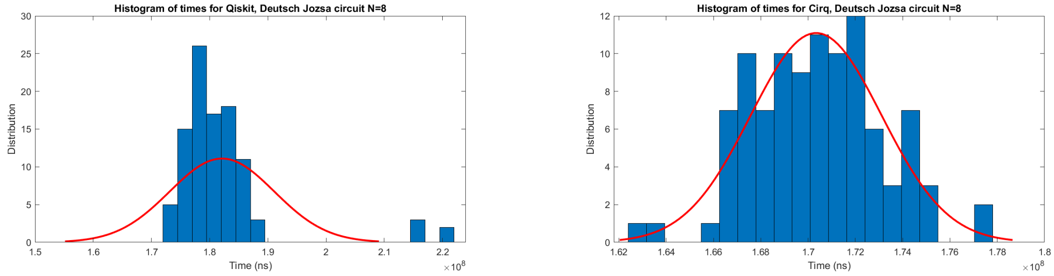
Table D.19: Execution times for Valkyrie, Qiskit and Cirq for Deutsch Jozsa N=9 circuit using 20 iterations as initial test (Valkyrie not optimised)



(a) Histogram for Valkyrie CPU in statevector mode **(b)** Histogram for Valkyrie CPU in fast mode



(c) Histogram for Valkyrie GPU in statevector mode **(d)** Histogram for Valkyrie GPU in fast mode



(e) Histogram for Qiskit **(f)** Histogram for Cirq

Figure D.2: Histograms for the distribution of execution times for various Quantum simulators with Deutsch Jozsa N = 8 circuit

Simulator	Valkyrie				Qiskit	Cirq
Processor	CPU		GPU		CPU	CPU
Mode	Statevector	Fast	Statevector	Fast	NA	NA
Run 1	1501214400	75270800	1302389500	94342100	200116100	186761500
Run 2	1485863700	76465500	1240763300	93008400	289143600	189838000
Run 3	1402015500	75693800	1292757200	95492900	202847100	186432400
Run 4	1359221100	74539600	1289526500	94246700	205998200	187429500
Run 5	1371834400	75881200	1292217600	94110600	198539500	185423500
Run 6	1419408100	78741200	1290031400	97082000	201297200	184864700
Run 7	1483164400	76891900	1286263400	98119700	206091900	184904000
Run 8	1454919300	76048500	1285511100	98446500	277500300	191361400
Run 9	1455846100	79272800	1281947500	97490800	201456500	186135100
Run 10	1408270700	77030800	1194045200	95841600	203918900	191566800
Run 11	1411604100	76950800	1261417200	97653400	204518900	187671400
Run 12	1398286200	77785200	1272826800	94687100	203353800	184259800
Run 13	1406405800	77579900	1252968600	92547900	200749000	189061300
Run 14	1405845600	77164500	1268392400	92746800	202460100	183252200
Run 15	1410784700	77776100	1251047200	92377400	203660100	188276700
Run 16	1359856800	77125700	1261940600	95395200	196465700	188582100
Run 17	1329570000	79549700	1258032400	1.01E+08	197802400	183888900
Run 18	1312693100	75937200	1256217000	96600900	201196700	186504800

Run 19	1315735700	79492300	1251815200	97534200	205515100	192751900
Run 20	1414108000	76526000	1253315100	94098500	278734200	187782600
Run 21	1415624300	75550300	1247014600	93281200	279641300	177180200
Run 22	1415181200	75581900	1250456500	93444100	206794600	187014600
Run 23	1400396000	74317700	1273894300	92493400	194573400	194172600
Run 24	1423653900	75094100	1257554100	92292300	205481000	190706300
Run 25	1400547000	75808800	1262707900	94877900	277124600	187714500
Run 26	1403455700	74002400	1256475200	92890500	204535700	186759400
Run 27	1406208300	74440600	1246645400	94470900	199417200	182905400
Run 28	1407575500	74649500	1246784100	93496100	206205500	185332900
Run 29	1400391100	74242400	1256011000	94508900	204593000	189778200
Run 30	1321093700	73963400	1268167100	99189800	202351300	185611100
Run 31	1320169600	74142300	1254324900	96998100	282519600	195207200
Run 32	1311775700	76239700	1253204400	93767500	204502200	182726900
Run 33	1365649500	74397100	1252077000	97535000	201368800	183477200
Run 34	1415809900	74708800	1254246500	95342800	202423300	190500500
Run 35	1422097100	75085500	1253486700	97142200	205486900	184622500
Run 36	1403409100	75385200	1267396100	94150300	284286600	189794000
Run 37	1404768400	74869500	1253396300	93896900	198926100	194692200
Run 38	1390917500	75124500	1256554500	94420300	204278700	186966700
Run 39	1396755400	74096600	1248218000	95215300	195450700	189090900
Run 40	1393273700	74397300	1250454000	96156800	206360300	185781100
Run 41	1411129300	74774400	1259424600	96485200	202221300	189539000
Run 42	1411735300	77345000	1254277700	93947100	200830100	194540200
Run 43	1375845500	77086900	1250406600	94736400	205974700	187182500
Run 44	1311603300	77745400	1255209900	1.01E+08	206640900	178027600
Run 45	1313495100	77585800	1252017400	97600100	207792800	188335600
Run 46	1327998500	76701000	1252857600	93734200	201286100	185600300
Run 47	1412131700	75431800	1252604400	94303900	198812000	181965800
Run 48	1421223000	75804400	1248557900	1.06E+08	281293500	185969400
Run 49	1415990000	75704900	1200624800	95168500	199494100	186666900
Run 50	1400060900	77068700	1274685600	92707200	204059800	189629800
Run 51	1402488100	77424700	1247956500	93229200	202423200	183967700
Run 52	1401640700	75000500	1249598700	93533400	279318300	189803200
Run 53	1402942900	77048800	1255556700	93858100	202900300	187540900
Run 54	1406940900	76661800	1251483300	92922000	206002900	191840700
Run 55	1399915100	77086800	1251064000	95557700	282403300	188147700
Run 56	1406936200	76470100	1251670500	97483800	197971500	189648600
Run 57	1322835600	76969400	1246501900	95112100	203321300	182470800
Run 58	1312416400	77802200	1247633800	92982100	284564400	182953000
Run 59	1311448700	75576400	1243887100	93131700	206818800	187228900
Run 60	1357828500	76333500	1251277400	93917200	198138300	189130000
Run 61	1414486400	74975500	1254231200	94943200	194173600	184819600
Run 62	1406150800	81471200	1259107900	93080900	284846000	188037700
Run 63	1413137300	74918500	1213962900	93109800	201358200	178538900
Run 64	1387445700	74680000	1247342500	92482100	199073300	191043200
Run 65	1405882300	74874400	1242449600	92861400	276103500	185471400
Run 66	1406757100	74755300	1254088100	92871100	281128500	189493900
Run 67	1412183800	75440100	1252785400	95213900	200540500	189654800
Run 68	1414753300	75328000	1248348200	93997700	199480300	186483400
Run 69	1412163400	73716100	1254480600	93259800	204711200	190344400
Run 70	1366176300	76089700	1264589100	92994500	201242400	190737300
Run 71	1314417800	74474100	1191485800	94109200	206859500	184801500
Run 72	1335718400	75416600	1249524600	93037800	200008100	188814700
Run 73	1331448800	75036700	1246696100	92336600	283962000	189791900

Run 74	1411532400	74571600	1251440900	92367300	202812000	187430800
Run 75	1416742600	74325200	1247735200	93118200	196958300	188234400
Run 76	1421835100	74973400	1259190900	93323100	198244500	179859200
Run 77	1404062600	74085100	1245353500	92753900	201425400	188395600
Run 78	1412207900	74323800	1247277200	93000100	200001500	186068500
Run 79	1408542400	74597600	1249400600	93452800	196530700	185756100
Run 80	1409314200	74292700	1252814600	96867800	206662800	188648300
Run 81	1408241800	76287500	1259486500	94503700	201028100	193708100
Run 82	1414606300	75197100	1254889300	94116200	194313700	188486400
Run 83	1409469800	74435700	1251964800	94626500	287074500	190252300
Run 84	1316804800	75879500	1247879000	93100100	201462300	188700500
Run 85	1324929800	74522000	1256463600	93932700	193159300	193615800
Run 86	1326675500	74616000	1275403400	92456600	281914700	190171600
Run 87	1356313800	74474100	1269413000	92098200	206526300	191145400
Run 88	1419962300	75117300	1259600800	92737800	203847800	187555000
Run 89	1405627500	74926400	1253044300	93661900	203408600	187073200
Run 90	1414082800	74886900	1267710900	96023000	207520900	185069300
Run 91	1395325500	74431000	1266684000	92199400	206806500	182775700
Run 92	1401284400	74790000	1245200100	94588900	199132300	188087400
Run 93	1404415900	73674900	1244345600	97919500	201433700	188758200
Run 94	1420994900	73907700	1247896300	96706900	203410600	181671300
Run 95	1408179200	74672000	1254284500	92237200	203930300	187804900
Run 96	1407617200	75596200	1247163000	93029400	202614600	187720700
Run 97	1364866800	74566400	1250695000	92403700	196203700	195070500
Run 98	1326084600	74463500	1245986800	92707300	202856700	186574500
Run 99	1323761400	74184500	1251460100	92280200	194803800	190463200
Run 100	1324145100	77713700	1245282700	93472400	203469700	184630400

Table D.20: Execution times for Valkyrie, Qiskit and Cirq for Deutsch Jozsa N=9 circuit using 100 iterations (Valkyrie not optimised)

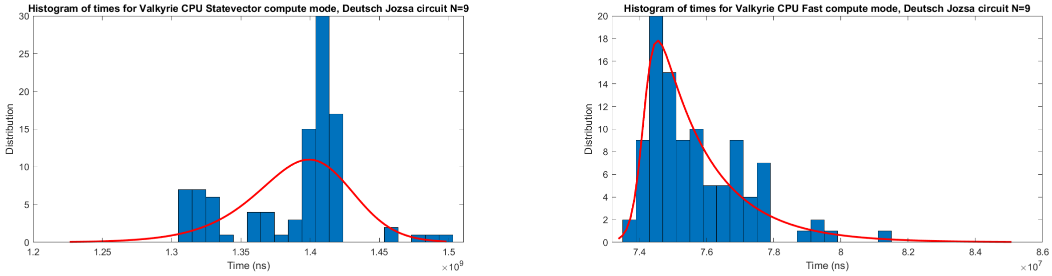
Simulator	Valkyrie					
Processor	CPU			GPU		
Mode	Parsing	Staging	Execution	Parsing	Staging	Execution
Run 1	45393300	308600	1416773300	47221800	258300	1247914800
Run 2	47165300	294900	1416667500	44070500	274700	1260586200
Run 3	45802400	293100	1409587100	44702400	285100	1252508300
Run 4	45025700	304600	1422780600	43971600	284600	1245025200
Run 5	45775600	278300	1421595000	43858300	260000	1237400800
Run 6	44223700	289000	1414980900	44231900	271400	1240229500
Run 7	46317000	309600	1422182200	44075100	278100	1245006400
Run 8	45142400	290000	1419141000	43793000	294300	1237024200
Run 9	45784700	279900	1397714900	44133700	282700	1228768300
Run 10	44691100	283300	1376477300	43600200	261000	1236226900
Run 11	44420700	287000	1363622800	43979100	280000	1132988800
Run 12	44886300	279200	1354605200	44020700	260600	1220079600
Run 13	44320100	280900	1369265100	44130800	297900	1210700700
Run 14	46032000	280700	1354953900	45324400	296100	1214831900
Run 15	44720900	300000	1352886700	43778100	288300	1198880400
Run 16	44269600	290200	1357682800	43711800	269300	1201538900
Run 17	44663900	291800	1354912900	44284500	297200	1195771400
Run 18	44344900	293100	1354940600	43837600	290500	1209256200
Run 19	44658200	302000	1350211200	43349800	279700	1202160000
Run 20	45397300	294600	1348852500	43536700	261300	1202760300

Table D.21: Breakdown of execution time for Deutsch Jozsa N=9 circuit running on Valkyrie (Valkyrie not optimised)

D.0.8 Deutsch Jozsa with N=10 test results

Simulator	Valkyrie				Qiskit	Cirq
Processor	CPU		GPU		CPU	CPU
Mode	Statevector	Fast	Statevector	Fast	NA	NA
Run 1	3669937600	131647300	3118585800	153402600	470319500	421349800
Run 2	3697939100	130365900	3131372200	150727000	455799300	409774700
Run 3	3719026100	129467100	3124812500	156879400	475932400	429124400
Run 4	3713795400	127646800	3119881200	151880900	468191200	428958600
Run 5	3787826800	132208900	3133364900	151719100	468147600	435909500
Run 6	3748126300	131607100	3141012500	156904200	465473000	404785800
Run 7	3780638400	131024500	3140491100	153694600	462301700	429545500
Run 8	3777475500	135044700	3169351200	156498500	461504000	421922800
Run 9	3767019900	127893000	3170626800	154922800	443346600	418754500
Run 10	3766581900	127496600	3140677300	151811400	458834000	422849100
Run 11	3775523100	126797900	3124305500	150368500	469581900	426171800
Run 12	3801673500	126712100	3114777100	151772500	463419200	426667000
Run 13	3817113000	127586200	3124653600	153221400	467984300	425907100
Run 14	3785732700	126435400	3116461300	150048500	462508900	432033300
Run 15	3809032900	126037800	3116035700	152811200	474580600	432543200
Run 16	3780637900	126026500	3105877200	150386900	458217000	419118400
Run 17	3808319900	127184700	3111213900	149795300	466941300	418101700
Run 18	3819424300	126624100	3114709500	149769000	461176100	433816900
Run 19	3821579200	127365300	3124895900	150411300	463896400	424960800
Run 20	3565529000	128033500	3099315900	153817400	478453700	426017500

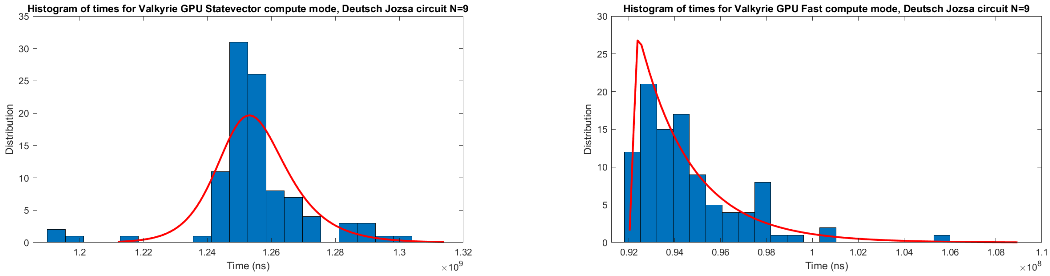
Table D.22: Execution times for Valkyrie, Qiskit and Cirq for Deutsch Jozsa N=10 circuit using 20 iterations as initial test (Valkyrie not optimised)


Histogram of times for Valkyrie CPU Statevector compute mode, Deutsch Jozsa circuit N=9

Histogram of times for Valkyrie CPU Fast compute mode, Deutsch Jozsa circuit N=9

(a) Histogram for Valkyrie CPU in statevector mode

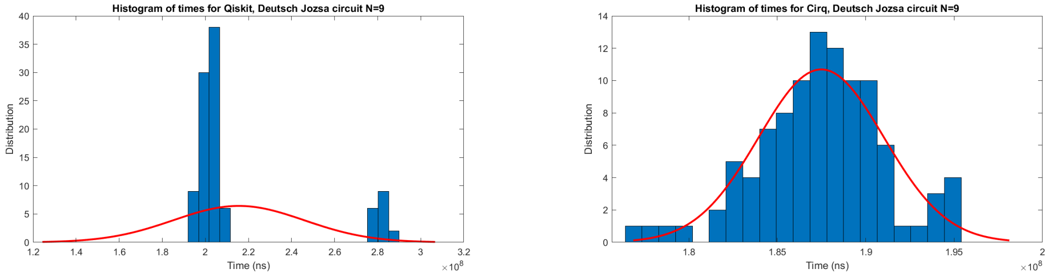
(b) Histogram for Valkyrie CPU in fast mode


Histogram of times for Valkyrie GPU Statevector compute mode, Deutsch Jozsa circuit N=9

Histogram of times for Valkyrie GPU Fast compute mode, Deutsch Jozsa circuit N=9

(c) Histogram for Valkyrie GPU in statevector mode

(d) Histogram for Valkyrie GPU in fast mode


Histogram of times for Qiskit, Deutsch Jozsa circuit N=9

Histogram of times for Cirq, Deutsch Jozsa circuit N=9

(e) Histogram for Qiskit

(f) Histogram for Cirq

Figure D.3: Histograms for the distribution of execution times for various Quantum simulators with Deutsch Jozsa N = 9 circuit

Simulator	Valkyrie				Qiskit	Cirq
Processor	CPU		GPU		CPU	CPU
Mode	Statevector	Fast	Statevector	Fast	NA	NA
Run 1	3716835600	134981300	3151506200	156309900	447387700	420760400
Run 2	3722104500	134443900	3122761100	161291500	511728000	417362300
Run 3	3735814200	131679200	3140941400	165357000	472494800	412905300
Run 4	3720262200	137856900	3116830000	156826000	466713200	422493300
Run 5	3712528200	136249400	3095102400	154919800	439560300	427366600
Run 6	3701084500	134453500	3117045400	160804300	433365500	421098900
Run 7	3719438300	130922100	3112976400	164471300	652211200	429647300
Run 8	3740864700	130426200	3115555800	155725600	637716000	433230100
Run 9	3730272800	132388100	3122681000	154864500	485283500	422113800
Run 10	3731279200	138335300	3107905900	160901500	486908500	427228200
Run 11	3705977700	133563800	3107589000	155352100	469052300	427330500
Run 12	3736638100	131037100	3111592200	153052700	663363200	430202100
Run 13	3781258500	132858600	3124170800	160554300	463885800	423143700
Run 14	3739991300	130337200	3095096400	155103900	527767700	420444100
Run 15	3732803100	130834800	3126861200	153689500	454053000	438025200
Run 16	3705233900	131223100	3101779600	155051800	488284400	435764700
Run 17	3732697400	132149100	3097918800	155039100	475401900	426976600
Run 18	3741112300	130391400	3119670500	158743800	657661800	431127200

Run 19	3729293500	130066200	3120721500	165978400	377051700	416784800
Run 20	3739555100	131573400	3102579400	159531700	517380600	424934400
Run 21	3712061800	131590000	3116326600	158408000	448016500	418230200
Run 22	3721978600	129400100	3117466600	157077000	482469200	426499300
Run 23	3755274100	130074500	3113999100	160094800	445885500	421187200
Run 24	3741557600	145103700	3110122200	159857400	490947000	419982700
Run 25	3734419000	134052900	3131359200	168363400	458998800	422746600
Run 26	3709201100	131986600	3093502400	159812100	469711600	425858300
Run 27	3731138800	131012200	3123820800	157476600	656212700	422890200
Run 28	3751173300	129115200	3105887400	155198500	470638400	425903200
Run 29	3738346900	129342400	3089078500	157394600	467048400	423312000
Run 30	3724596100	129539300	3116240900	155557900	470896400	421016800
Run 31	3720778800	131020500	3112787100	155672900	418933100	423076700
Run 32	3721268500	131605700	3090811600	168014700	527041700	425021600
Run 33	3730485100	130304100	3111305500	154132600	438398400	421636000
Run 34	3733693600	133784500	3110547200	154937900	482336900	439778200
Run 35	3725916100	129284400	3104347800	154755100	455403300	421280300
Run 36	3702179600	129442000	3120983200	158879900	498706300	425590800
Run 37	3729826300	129684700	3115792500	158573400	466336100	418596200
Run 38	3741704100	129775000	3112293100	155416400	439091500	420631000
Run 39	3733741400	129434800	3121210900	157361900	495141700	434192600
Run 40	3722334200	130008500	3107277800	159672400	395053100	426994100
Run 41	3713149900	131126600	3125764100	162865800	527568100	430755800
Run 42	3716004300	129513600	3107694900	155092900	471315200	421795300
Run 43	3741893000	129804000	3109975200	154667600	464118400	425668700
Run 44	3733864900	129733400	3106877500	160978500	426853000	427160900
Run 45	3736693600	129775000	3115184300	156075700	370038700	419159900
Run 46	3723887700	129530300	3104936800	155793500	664197700	425112900
Run 47	3718478500	128420300	3099199000	156811400	651961000	407073200
Run 48	3737668100	129778900	3107047800	157613100	651468300	431640700
Run 49	3748456700	128753000	3097411500	155140600	429667600	416714700
Run 50	3740889900	131015100	3127134500	155516300	461389200	427293800
Run 51	3705051100	131337500	3144939600	157680800	393698700	424295400
Run 52	3718628800	133417100	3117178400	156445600	445634000	418020800
Run 53	3749735800	132846000	3137181600	155337900	443455700	431551900
Run 54	3751308800	132831300	3118374700	154741500	647051100	430366500
Run 55	3727705800	129477000	3099684300	158179800	473789000	423896400
Run 56	3720221000	130176300	3087892600	153910600	649644700	423618000
Run 57	3718558600	130719600	3106006100	156326700	471950700	420952900
Run 58	3731269100	128999700	3111970900	154305500	466778200	395524500
Run 59	3745414900	131270000	3091554400	154139000	645323300	426014800
Run 60	3732504000	129379300	3108761400	153923300	451581800	421002300
Run 61	3699417100	128260500	3096623200	153652800	457477400	431658000
Run 62	3732628800	129757800	3098981500	155560800	472477600	421882800
Run 63	3741778300	129390900	3110012400	154780900	656921700	415908500
Run 64	3739468500	128168900	3097220900	159595500	475595700	421641000
Run 65	3733684700	130974500	3112410800	155075500	473260700	425476700
Run 66	3707739200	129348600	3113933300	155989400	435390600	422775000
Run 67	3720415400	129791400	3103479400	159355000	641649900	420706700
Run 68	3731993400	131081200	3093897800	158250300	480577600	423443200
Run 69	3725811300	129843600	3122512800	158605400	483807400	425090700
Run 70	3723300000	130057100	3115123000	158247100	428713500	413931900
Run 71	3703565900	129013400	3104465400	155684700	465452700	419165900
Run 72	3725477700	129595200	3108389600	157467400	453421300	417719700
Run 73	3747877900	130078100	3108085300	153504900	478169300	428862900

Run 74	3734708500	129885000	3102242100	156875000	466294200	428252200
Run 75	3747072900	129639800	3106533500	160153100	444555100	423464000
Run 76	3716735200	130046900	3120906300	160850700	473239200	431046900
Run 77	3722200000	129831700	3161437200	160401100	422452600	428300200
Run 78	3739696200	131514700	3125427000	161256500	476881300	414001700
Run 79	3725607200	130316100	3114259400	154525500	448170500	425720200
Run 80	3730727700	129362000	3099516400	154104700	560690200	429515500
Run 81	3713088400	129907700	3105813300	154929300	469025500	419253200
Run 82	3705899700	130740900	3144143000	153484400	502164000	430413500
Run 83	3750488800	129415300	3108197200	154314300	525262500	422744800
Run 84	3719269100	130087500	3115622900	156683800	651642200	425859700
Run 85	3721487600	130663300	3106455100	153867400	457455500	420289300
Run 86	3696549600	134533100	3098820700	155664900	464540300	425566700
Run 87	3719725000	130490900	3106754200	153640400	661369500	417761000
Run 88	3746100400	130637000	3119576500	160928100	654404400	420331000
Run 89	3745683500	131648800	3108384200	155548300	477027600	416496700
Run 90	3724416900	131724900	3130828600	159383300	660641200	431235200
Run 91	3709500200	130249900	3090266200	159846700	457965000	429628300
Run 92	3719139300	129596500	3107728200	155221100	430592100	422272700
Run 93	3742339400	130235700	3107767400	156605900	447385300	424824300
Run 94	3751061500	128334400	3120698500	154681300	456149500	426842500
Run 95	3742801600	131306500	3081814000	154055200	443558100	426916800
Run 96	3728758600	134051800	3119791700	154361600	510181400	422309500
Run 97	3717895400	133523900	3103009400	153009600	462801500	417371700
Run 98	3751160000	132069500	3092234800	155178800	526196700	406904400
Run 99	3747106500	129335500	3127258300	153270400	475330600	434664600
Run 100	3753917500	130751500	3109548100	154648100	459394400	430466000

Table D.23: Execution times for Valkyrie, Qiskit and Cirq for Deutsch Jozsa N=10 circuit using 100 iterations (Valkyrie not optimised)

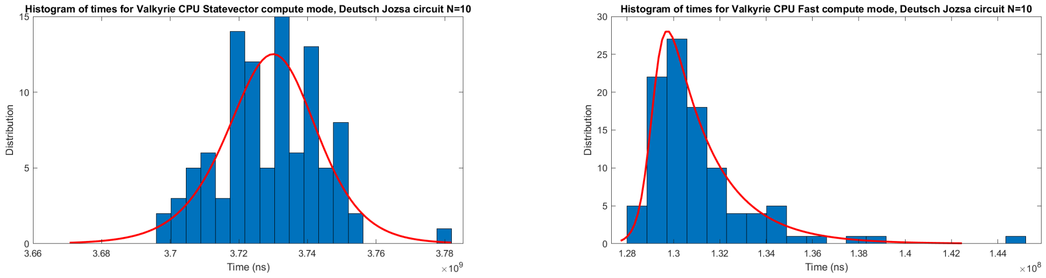
Simulator	Valkyrie					
Processor	CPU			GPU		
Mode	Parsing	Staging	Execution	Parsing	Staging	Execution
Run 1	50889100	318500	3699754200	50021100	334400	3122888600
Run 2	50491200	303500	3680096800	49678300	320700	3115789300
Run 3	50734600	324100	3704901400	52685300	343100	3132204300
Run 4	50874000	324900	3665513500	50098500	354300	3105550200
Run 5	50607700	316100	3695465900	50479300	322900	3128832400
Run 6	50471300	312300	3648281300	50148600	332500	3159568000
Run 7	54102600	334000	3687160200	50442400	320100	3118350300
Run 8	51302400	321800	3701778100	50121400	321200	3100210700
Run 9	50633400	318300	3672974300	49956500	325000	3110923200
Run 10	50114300	315500	3651369600	51070800	331500	3102512900
Run 11	50234700	310400	3660686000	50081800	458300	3103362600
Run 12	50087200	306700	3691529500	50652400	321300	3108989200
Run 13	51524500	297900	3707580400	50953200	325200	3157838500
Run 14	50147800	320900	3673648500	50262600	321600	3099950900
Run 15	50582000	324700	3680022600	50670200	323500	3106410600
Run 16	50876100	325700	3652256100	49737600	326100	3104568200
Run 17	50168100	306300	3731633700	49738700	333300	3089007400
Run 18	52171500	308100	3788554500	49282500	333600	3080890200
Run 19	50658900	315800	3781563300	50133300	328600	3078651200
Run 20	50143100	308700	3701137100	49524900	322700	3083257600

Table D.24: Breakdown of execution time for Deutsch Jozsa N=10 circuit running on Valkyrie (Valkyrie not optimised)

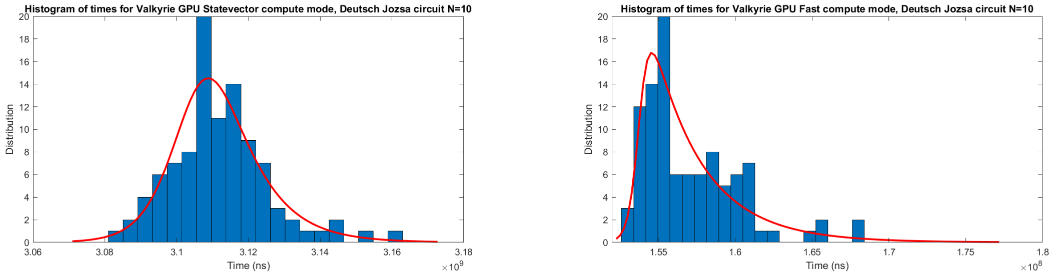
D.0.9 Deutsch Jozsa unoptimised Valkyrie analysis

	Gate Construction	Statevector Reorder	Execution
Run 1	285890400	781425000	342897200
Run 2	284338700	804746100	343278700
Run 3	293351000	779961600	340294100
Run 4	287789500	792484500	338217600
Run 5	286302100	794240200	340438500
Run 6	285600900	783744100	340407000
Run 7	292228400	793863800	334524300
Run 8	278307500	785490200	338349300
Run 9	282751500	815261300	340713900
Run 10	280284300	818412100	341612400
Run 11	278367700	798570600	343592700
Run 12	294823900	803253200	344788800
Run 13	276751200	791480300	340632400
Run 14	282216500	792933100	356311500
Run 15	281805900	797274100	348862500
Run 16	287020000	737563000	340773100
Run 17	278808400	793426400	345766400
Run 18	280386300	800828100	340614300
Run 19	278779400	784827100	341376600
Run 20	277557300	799997900	342972500

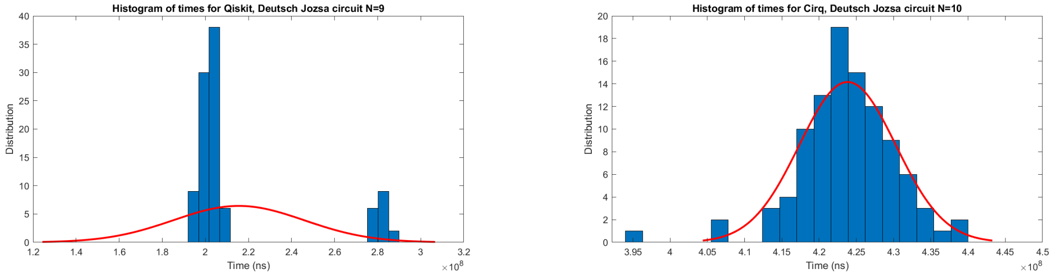
Table D.25: Table comparing the time taken to complete the individual stages of execution for Deutsch Jozsa N=10 with Valkyrie running in "statevector" compute mode on the CPU



(a) Histogram for Valkyrie CPU in statevector mode **(b)** Histogram for Valkyrie CPU in fast mode



(c) Histogram for Valkyrie GPU in statevector mode **(d)** Histogram for Valkyrie GPU in fast mode



(e) Histogram for Qiskit **(f)** Histogram for Cirq

Figure D.4: Histograms for the distribution of execution times for various Quantum simulators with Deutsch Jozsa N = 10 circuit

D.0.10 Optimised Valkyrie Results

N	4		5		6	
Processor	CPU	GPU	CPU	GPU	CPU	GPU
Run 1	21566600	32671800	27955900	36150000	43633900	58517000
Run 2	21771200	29803500	27716700	35820700	44818700	56611800
Run 3	21435600	29526900	27823600	38127700	42820200	57128100
Run 4	21260100	29996500	27349700	38956600	43696300	54639800
Run 5	21262100	29380700	27541500	36053800	45818500	54372000
Run 6	21413000	29685200	27553100	39959200	41492600	53078200
Run 7	21168700	29138500	27517800	36586900	41468400	54481800
Run 8	21400400	29046800	27760400	35991800	41400500	54454300
Run 9	21465900	29509600	29074800	35669000	40953900	63295800
Run 10	21492600	29101400	27302200	36202800	41187100	52865800
Run 11	22231400	29899100	28979900	36202900	41089100	52939700
Run 12	21801500	29703900	27775000	41657200	41134100	56515400
Run 13	21725000	29580200	28801900	35213000	41629900	53159800
Run 14	21587200	29233000	26854400	36189400	41955700	53401600
Run 15	21779200	29220500	26867800	37268400	41859400	52724000
Run 16	21929000	28760000	26737100	37979400	41117600	53138700
Run 17	21359600	28518600	26910100	38910100	40995500	52727500
Run 18	21324400	28416400	27296600	38418500	40837300	54113300

Run 19	21034700	28508600	26672200	36396900	40890500	52947900
Run 20	21609400	29963900	26707800	36191300	41163500	53564700
Run 21	21145800	29004100	27297900	38062800	40496700	53058500
Run 22	21659800	29288400	26619500	36837200	41527900	56311800
Run 23	22017300	28736200	26752800	38298500	41196700	52372800
Run 24	21935900	29576100	26485300	35342300	40806700	53576800
Run 25	22539300	29314400	26775800	35843600	41129400	52371500
Run 26	22591100	29929100	26897000	37753800	41052300	53275500
Run 27	23658200	29959100	26483000	36130100	40813700	53901600
Run 28	22001900	30283300	26796900	36938500	41019600	53990300
Run 29	21973200	31930200	26571200	35254300	41448700	52905600
Run 30	23932000	30613600	26600600	35105900	41062300	53950300
Run 31	22667400	30653900	26215000	35576100	41165200	52671100
Run 32	23311100	30370500	26249500	35392900	40532200	52508100
Run 33	22369700	29982100	26635200	35301000	40988300	53498200
Run 34	23374900	33362400	26922200	35435200	41165300	56350000
Run 35	21462700	30303900	26453500	35384000	40980500	55333200
Run 36	21632100	32109200	26344800	34829000	40581300	53817500
Run 37	21433100	32030600	26469400	35408000	40916800	55456900
Run 38	21419400	29478500	26394300	35061500	41055800	55692600
Run 39	21549100	28971600	26188500	34554300	41447700	54607400
Run 40	21635600	29968500	26963900	34672300	40849800	54335500
Run 41	21599800	30063100	26844700	35410400	40849500	55216400
Run 42	21297700	29264400	26790400	35675500	40759600	57294000
Run 43	21314800	29785200	26662100	35115700	40885200	53187200
Run 44	21424900	29467600	27367100	34904700	41545900	54082200
Run 45	21440800	30970000	26851100	36394700	41149500	54290200
Run 46	21539000	30522300	26731100	35173500	40997600	53130700
Run 47	23531500	36956100	26743200	35767600	41079300	52406400
Run 48	21370000	29777900	26674600	35422800	42383600	52687600
Run 49	21505600	29794300	27168300	35001200	41140400	53028900
Run 50	21429800	29244000	26702800	35894100	40946400	53059900
Run 51	21799300	29871700	25896100	35448500	41671300	53714800
Run 52	21407200	29951500	27357300	35155400	41095500	53553900
Run 53	21371200	29366600	26183500	35745000	40684400	53249600
Run 54	21525600	29086200	26448700	35316300	41034300	52718800
Run 55	21827200	29756000	26834000	35809600	42261600	53644300
Run 56	21305100	29189700	26674200	36508200	42503600	54158000
Run 57	21857600	29413900	27235300	35790600	43026500	53766900
Run 58	21393600	28878600	26647900	37281700	42892000	53132200
Run 59	21344800	28357100	26867800	35589100	43413500	53740500
Run 60	21041400	29178700	26427400	40216000	42319200	53577100
Run 61	21373400	28490700	26772400	35547100	41580100	52693000
Run 62	21359200	29222900	26550200	39342500	41669300	52802900
Run 63	21255800	30246300	26571400	35564700	41962600	52534500
Run 64	21367600	28330900	26903300	36894600	42008400	57585500
Run 65	21691300	28328900	26421800	34953100	42015300	52406700
Run 66	21128200	28917000	26611400	35336900	41304700	55226200
Run 67	21271300	28573600	26698800	35494700	40612600	53303400
Run 68	21142300	29826800	26777300	34806000	41036600	53042900
Run 69	21509000	28734200	26335300	35148100	40922100	52524300
Run 70	21342800	28457300	26875500	35426400	41183100	52450400
Run 71	21616400	28159900	26754100	36376400	40729400	53639700
Run 72	21286200	29281100	26167300	35462800	41233700	52582600
Run 73	21529600	32116000	26220000	35276000	41001200	54002400

Run 74	21512800	28352800	26770100	35082400	41371000	52610100
Run 75	21137200	29276600	26537700	36025300	42152900	53932400
Run 76	21491000	28424200	26258700	35367300	40661300	53150500
Run 77	21396500	28801500	26430500	35012000	41491900	52497500
Run 78	21975100	29341900	26595500	34878000	41141200	53142000
Run 79	21265200	28682800	27024800	35079600	41626800	54175900
Run 80	21385900	28537700	26595400	35816700	41141800	52512200
Run 81	21256300	28594000	26717900	35460600	41555300	52592400
Run 82	21072400	28284300	26869700	35360000	41621100	54496000
Run 83	21368900	28870600	26347800	35121500	41888200	53231400
Run 84	21159500	28124100	26377300	34759300	41523800	52986800
Run 85	21350800	28094300	27671000	35521000	41459400	52999800
Run 86	21848600	29336300	28318100	35536800	41204100	53768000
Run 87	21104700	28923100	26444800	35253000	40867900	53118800
Run 88	21290600	28953500	26326600	35249000	40801200	52525300
Run 89	21621000	28351100	26417000	34689300	40889700	53600700
Run 90	21193200	29178100	26309000	39763800	40696200	54821400
Run 91	21214400	28387900	26829200	35383700	40967500	53427100
Run 92	21735800	28875900	26518100	41059200	40762100	52229500
Run 93	21118600	32906800	26838900	36842400	43055800	52786000
Run 94	21336900	29581000	27694700	36051800	40633900	53906400
Run 95	21014600	33850900	27374500	36103900	40765200	52814600
Run 96	21129600	28992000	26951300	36768900	41469800	53129000
Run 97	21906400	29260500	27379200	37375300	40982100	57353200
Run 98	21655700	29223100	26805800	36682700	40872900	54092900
Run 99	21603200	29003600	27062400	36023800	41320600	59580300
Run 100	21295200	29226000	27059700	36025900	40714200	52620700

Table D.26: Raw timing data for Optimised Valkyrie running Deutsch Jozsa Algorithms for N=4,5,6,7

N	7		8	
Processor	CPU	GPU	CPU	GPU
Run 1	71057800	72992600	109507200	89551700
Run 2	71115600	71902000	107716200	88454100
Run 3	98600000	87280700	108589700	88271100
Run 4	68837800	70488900	108789500	88761200
Run 5	70394100	70620800	109349400	88329100
Run 6	69523800	70735700	107991800	87707000
Run 7	71276400	71126200	109688100	88890800
Run 8	96775000	69935000	112325900	89265700
Run 9	97976600	70134000	113662400	88713300
Run 10	69750900	70629400	111184300	87749600
Run 11	96999000	71447200	111943400	87553200
Run 12	68893800	70440000	111782800	88210100
Run 13	69989100	70825900	108686200	89467100
Run 14	69827700	72222600	109206300	91412800
Run 15	69789000	71557300	107519300	91654000
Run 16	98068800	70959100	113146800	89995500
Run 17	68958600	74179800	110336100	88188900
Run 18	70821400	74809300	111525300	88589500
Run 19	69139400	70260100	108052300	88268700
Run 20	71698700	70344200	108051700	88139200
Run 21	73854500	71716000	107487400	88440100
Run 22	70302300	71336600	109370200	88009200

Run 23	70057100	72919900	110634300	88177900
Run 24	69655500	70659000	107263600	88392100
Run 25	97094500	71090400	109219100	88910700
Run 26	70815600	72146900	109983100	88877700
Run 27	95483200	69879300	113102200	88994600
Run 28	96366400	70075900	117622100	89087400
Run 29	67986500	72069200	109381700	88644900
Run 30	69132900	70364800	108249400	88813000
Run 31	69221800	71167900	112128400	89280700
Run 32	68800700	71027000	107477900	89625100
Run 33	68529900	70958400	108790500	88109700
Run 34	69851000	70312800	107452600	87858700
Run 35	70352700	71488600	109267200	88533900
Run 36	67423600	71549000	107791500	91636900
Run 37	99043500	72453100	108131100	88235900
Run 38	69608200	73888000	108276200	88222500
Run 39	69702300	71103000	107153300	87532900
Run 40	98039500	74289800	109884600	88738400
Run 41	70364400	70278800	110999500	88994500
Run 42	68744600	71482000	108269100	89362200
Run 43	68812700	70603400	108389300	88711200
Run 44	67284800	70109500	108908300	88428500
Run 45	70713500	70539200	108252000	87747400
Run 46	69290300	71694300	107701700	87839400
Run 47	69563300	70041300	108366600	88069800
Run 48	70542800	70889700	108229300	88394900
Run 49	69136700	71548300	108356100	89397600
Run 50	68290400	69860800	110291500	88653500
Run 51	70405300	71058600	109610700	89126500
Run 52	97056300	73717300	109020300	88502000
Run 53	69328100	71565700	111325300	89643900
Run 54	70066700	70698200	108219400	89004400
Run 55	69607300	71842100	109901400	88088800
Run 56	97511200	70864900	108205400	88070300
Run 57	70192900	70961500	108635300	88048300
Run 58	70481500	70972900	106752400	92530900
Run 59	69394300	71785000	108214900	88205300
Run 60	69525200	71046100	109940100	88563400
Run 61	70676200	70458200	109095000	87892000
Run 62	69674700	70172100	108173500	88214500
Run 63	70285800	71765000	112225900	88595200
Run 64	71310500	70603500	117067600	87826600
Run 65	98374500	70709100	107170400	91928200
Run 66	70356500	70617000	108756500	87820500
Run 67	70071900	71708800	109205900	88349200
Run 68	95870000	71511800	108456000	88347600
Run 69	70693200	71096700	107520200	88889900
Run 70	98392000	69902500	110685300	87971200
Run 71	96244600	72036200	109447600	88357900
Run 72	69626300	72783800	108207300	88598400
Run 73	69895700	71395100	108911600	88090400
Run 74	98825000	70903200	109101500	88313300
Run 75	98914100	70749600	109094500	88480500
Run 76	71056400	70494200	109968100	88299100
Run 77	70945700	70709500	109161300	88850800

Run 78	70162700	71421700	107449800	88299100
Run 79	99390900	70021600	109589900	88919200
Run 80	71275200	70853600	107453300	88589800
Run 81	68741700	71766600	107966100	88942000
Run 82	68421800	77286400	109228000	91389900
Run 83	69504100	70974600	112150600	88997400
Run 84	97927100	71431200	107051900	88010000
Run 85	71122500	70757700	107993600	88437700
Run 86	70436800	80071300	107759600	88679800
Run 87	71701100	71468600	108734400	88120400
Run 88	97798300	70572400	108675300	88714700
Run 89	68093600	72835000	108148400	87881900
Run 90	70250200	69393500	107248900	88397300
Run 91	69068500	71176700	116951600	87646900
Run 92	68504200	70763500	111122200	89590800
Run 93	71455100	70282000	108141100	91385200
Run 94	70885200	71092200	107379000	87703300
Run 95	68550500	71859800	112110100	87808700
Run 96	98987400	70755500	108860100	88411900
Run 97	71052600	74108100	108296800	88964100
Run 98	69625400	71403500	109320900	88073800
Run 99	69631800	70955700	108619500	88010600
Run 100	69325400	71604400	108591800	88641900

Table D.27: Raw timing data for Optimised Valkyrie running Deutsch Jozsa Algorithms for N=7,8

N	9		10	
Processor	CPU	GPU	CPU	GPU
Run 1	179377500	149885200	359645300	323371400
Run 2	177674400	150179300	363756300	320995400
Run 3	174900800	149069600	361367100	320001600
Run 4	176657900	150085900	363682300	320615200
Run 5	177248900	150432200	362010200	323213900
Run 6	176857600	149914600	360373100	320503700
Run 7	182424800	150089900	359281000	321197700
Run 8	181291600	150745000	364027300	322247800
Run 9	178240100	150587300	362947000	320445400
Run 10	176245100	149151300	361289100	323361100
Run 11	178626100	149680800	361140000	318831800
Run 12	175784200	150360600	361584900	320282300
Run 13	178299600	150889200	359460600	319128800
Run 14	181506700	150010400	360543300	320556100
Run 15	176877800	149379600	361783200	319947000
Run 16	176317600	149421000	366181400	319519300
Run 17	176215400	150181400	357354600	322257200
Run 18	176578000	149473300	363220200	319781000
Run 19	177036800	149811000	361720800	317621500
Run 20	176447400	148805000	359511200	320897500
Run 21	178315700	150608000	360206200	318688000
Run 22	180242500	150818800	362033800	318298300
Run 23	177556100	149769600	359187500	319939200
Run 24	176632000	150715200	375061700	323155300
Run 25	177666300	150129000	359678800	319501300
Run 26	174832600	149512400	358500400	320442500
Run 27	174338600	149858900	362446800	321718300

Run 28	178785100	150287800	362518200	320531100
Run 29	179007600	149270500	359381100	321218700
Run 30	176231500	150604900	362939600	322475800
Run 31	176575100	149541100	362349200	319337800
Run 32	175176100	149490300	379104200	321172900
Run 33	177866800	150899000	359833700	321964200
Run 34	177973600	149487500	360985700	318277900
Run 35	176268300	150431600	362847300	320412000
Run 36	177041100	150101600	360329300	321640500
Run 37	177709800	150096900	363550400	320671800
Run 38	176266700	149587200	365544600	320275000
Run 39	178963800	148875100	361300400	320126200
Run 40	181006600	149226100	361075700	321866400
Run 41	176417700	153115500	366546100	319147600
Run 42	178423600	152014900	358370400	320885200
Run 43	176399200	153384700	360665100	321161100
Run 44	177523800	150232800	379324400	319660500
Run 45	177094300	150721500	361478700	320638500
Run 46	179561500	149444300	361996800	321132400
Run 47	179714200	150329000	360938600	318702700
Run 48	179867800	150840600	361722800	320455100
Run 49	176378700	149233300	361370900	320957200
Run 50	177512800	151242000	359330900	319688200
Run 51	177238500	150775000	359871300	320714800
Run 52	176564600	151654400	362142700	322454500
Run 53	177864300	150793600	359194900	321464900
Run 54	228110800	149473100	367705100	319410900
Run 55	175978700	151281900	363222600	320295700
Run 56	177460600	153145500	359513400	318589200
Run 57	185396200	153980100	364655900	321831000
Run 58	176015500	152069300	362009400	320318900
Run 59	176300700	151547400	358070900	322058500
Run 60	181345000	152691600	362579300	319090600
Run 61	178573000	154648200	360742300	321574700
Run 62	177245600	154132800	362739300	321230500
Run 63	177137900	150282900	362368600	317057200
Run 64	178549000	152521900	364432200	319562400
Run 65	177711200	148673600	360561600	321695300
Run 66	177058200	149526800	360278500	318845200
Run 67	176878800	150084600	360536500	319604500
Run 68	181217100	150329900	364111500	320587300
Run 69	176952600	149957900	359980400	319629000
Run 70	176965900	150980800	362145400	319545600
Run 71	174693400	153976800	366830800	318831500
Run 72	176671300	151528000	362359100	320121300
Run 73	178408000	149861100	361331500	320775800
Run 74	182795500	149542900	363288500	321867100
Run 75	176150000	149668100	360006000	321154600
Run 76	176129300	150069000	363165800	326417400
Run 77	175133700	151512000	366051700	321691000
Run 78	176193900	149475200	359130900	320925600
Run 79	176373000	151130800	360373800	320311700
Run 80	175689500	148949500	363383500	320038000
Run 81	178407300	149892800	358256500	322437400
Run 82	179525000	148261600	362490700	320040100

Run 83	176641900	150013100	361527300	322429200
Run 84	178107200	152108700	362009200	320801900
Run 85	176633300	149740600	361465000	317777000
Run 86	174916200	148597700	361682300	320368600
Run 87	175917000	149101400	363776000	321434600
Run 88	178382200	148975900	363825300	319733200
Run 89	176343400	149107900	360735600	319967200
Run 90	176799500	155812800	360255600	320531200
Run 91	176174500	150310000	365380800	322334500
Run 92	176353400	149863000	360985000	321770000
Run 93	176280200	149433900	363381100	322149400
Run 94	178861500	150954100	364395000	319626000
Run 95	175962300	150566600	363055700	319194200
Run 96	177211200	149781500	360507800	319998800
Run 97	176031500	151656100	362259000	319964600
Run 98	173571800	149455800	358978500	320113000
Run 99	176815100	151558400	365288000	320718900
Run 100	178501800	150285600	362246200	319947000

Table D.28: Raw timing data for Optimised Valkyrie running Deutsch Jozsa Algorithms for N=9,10