University of Glasgow | School of Computing Science

Honours Individual Project Dissertation

Development and Parallelisation of a Quantum Computing Simulation Library for CPU and GPU architectures

Lakshit Dabas

March 10, 2020

Abstract

Over the past few decades, the field of quantum computing has grown rapidly. With the breakdown of Moore's law, researchers seek to find an alternative to classical computers to continue increasing processing powers. As innovation in developing newer architecture for classical computers slows down, many have proposed using quantum computers as a stepping stone towards gaining further speed-ups for computing systems. In this project I developed a quantum computing simulation library in order to gain a deeper understanding of fundamental quantum computing principles. The library was also optimised for both, CPU and GPU architectures with OpenACC. Furthermore, elementary quantum algorithms essential for creation of scalable physical quantum computers were also developed and implemented. The library was successfully created with minimal deployment complexity, in order to aid users to seemlessly integrate the library into their codebase in order to analyse simulated multi-qubit systems. More importantly, a deep fundamental understanding of quantum computing systems was acquired.

Acknowledgements

I would like to thank my supervisor, Dr. Syed Waqar Nabi for giving me the opportunity to work on this project. I would like to thank my parents for always providing me with the encouragement I need. Lastly, I would like to thank my sister for being my lifelong role model.

Education Use Consent

Contents

1 Introduction

1.1 The Need for Quantum Computing

Upon the successful execution of Shor's algorithm for large numbers by a quantum computer, researchers realised that quantum computers can theoretically deliver incredible speed-ups for a large subsection of problems. This has led to huge strides being made in the field of quantum computing. Currently, researchers are working towards identifying problems that will benefit from execution on quantum computers, and developing algorithms to execute them. The current consensus is that quantum computers will be able to solve problems ranging from optimisation to prediction far more efficiently than their classical architecture counterparts (Engineering 2019).

Moore's Law describes the exponential growth of processing powers of classical computers approximately every two years. This prediction has held true for some decades. However, as the size of a transistor approaches the size of an atom, a quantum phenomenon known as electron tunneling occurs which adds significant noise to the transistors. As innovation in developing newer architecture for classical computers slows down, many have proposed using quantum computers as a stepping stone towards gaining further speed-ups for computing systems.

Where classical computers fail, quantum computers can succeed due to a property unique to quantum systems known as quantum parallelism which allows a quantum register to contain qubits which may exist in a superposition of its fundamental computational bases. Each obtainable value from superposition can represent a possible state of the qubit. Since the number of possible states is 2^n where n is the number of qubits in a quantum register, a quantum computer can obtain solutions in a single operation for problems that require significantly more operations on a classical computer. This can theoretically lead to massive speedups for solving problems. However, the current level of development of quantum computers dictates that highly specific algorithms must be created for isolated problems to demonstrate the use-cases of quantum computing.

Currently qubit systems are being used by researchers across the world for optimisation problems. These systems have also found their way into the private and government sector with implementations in the defense sector (*Quantum computing and defence* n.d.), weather and traffic forecasting (Vela 2019) and even financial modelling (Zapata n.d.).

While present researchers have created some novel Quantum Computers, this quantum architecture must be complemented by an accompanying software architecture to allow users to take full advantage of quantum technologies. A number of researchers in public and private domains have created quantum simulators in order to bridge the gap between physical quantum computers and it's accompanying software. IBM Quantum Experience allows quantum computing enthusiasts to create their own quantum circuits, while QASM (Cross et al. 2017) is a quantum programming language that can be utilised by researchers. Google also allows public access to their Python based quantum computing resources via Cirq. Additionally, simulators in QX Studio and libquantum (Butscher and Weimer n.d.) are based in C/C++ and utilise parallelisation with MPI to deliver an accelerated quantum computing simulator.

1.2 Goals

1.2.1 Problem Statement

In order to complement current and future physical quantum computers, the need for a software architecture to accompany quantum computers exists. The challenge to creating this software architecture is the exponential increase in memory demand and associated measurement errors that follows when the number of qubits in a quantum register is increased. Furthermore, few quantum computing simulators exist that can utilise a GPU architecture and can be embedded within other qubit reliant systems. This creates an obstacle for researchers, limiting them to larger execution times when executing larger problems.

1.2.2 Aims

This project aims to create a quantum computing simulator from the ground up in order to gain a deeper understanding of fundamental quantum computing principles and how they can be implemented efficiently. Furthermore, the aim is to develop a simulator that is optimised for both GPU and CPU architectures, and measure it's viability by executing essential algorithms such as quantum Fourier transforms, inverse quantum Fourier transforms and quantum entanglement. An efficient product implementation will assist researchers in implementing quantum software systems into their existing research frictionlessly.

1.3 Report Outline

The dissertation is structured into 7 chapters noted down blow,

- **Chapter 2** provides background information necessary for quantum computation. Details regarding the acceleration of this product are also mentioned.
- **Chapter 3** provides details regarding the overall design of the simulation library.
- **Chapter 4** details the implementation of the quantum computing simulator. GPU acceleration and additional optimisation are also discussed as part of implementation within the chapter.
- **Chapter 5** discusses the algorithms that have been implemented and exposed to the user.
- **Chapter 6** discusses the method of evaluation for the project.
- **Chapter 7** summarises and concludes the report. Potential for future work is also discussed in this section.

2 | Background

An understanding of key quantum mechanics and linear algebra principles was necessary for the successful design of the delivered product. These fundamental concepts have been referenced using (Yanofsky and Mannucci 2018).

2.1 Quantum Computing Fundamentals

2.1.1 What exactly is a Qubit?

A classical computer, utilised by people everyday, is built using classical gates, which process information in the form of a bit. This is a constant value which can either be 0, or 1. The value of a classical bit is known to us throughout its life-cycle. From the moment the bit enters a classical circuit, to the moment after it leaves, the value of a bit can be predicted. Just as a bit is the fundamental unit of information for a classical computer, a qubit is the fundamental unit of information for a quantum computer. While a bit can possess a value that is either 0, or 1, a qubit can possess values which may be 0, 1, or a superposition of both of these values. A qubit can be expressed as a vector, in a two dimensional domain. This domain is termed as state space.

In quantum mechanics, quantum states are expressed as $|0\rangle$ termed as *ket vectors*. A ket vector denotion describes that the quantum state can be expressed as a complex vector in Hilbert space. For the quantum simulation library, QuantoSim, developed for this project, the quantum states that formed the computational bases of the simulator were chosen to be,

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \; |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{2.1}$$

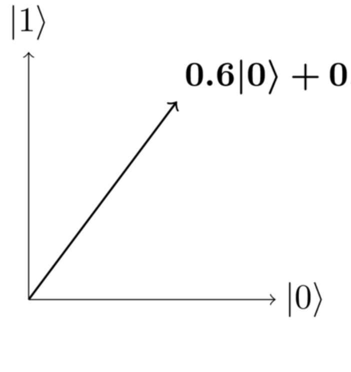


Figure 2.1: *A two-dimensional vector representation of a mixed quantum state* $0.6|0\rangle + 0.8|1\rangle$ *(Nielsen and Chuang 2019)*

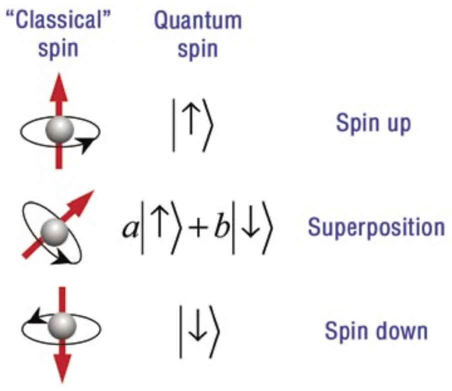


Figure 2.2: *A schematic representation of angular spin and the superposition of states(Bracker and Gammon n.d.)*

In Figure 2.1, it can be observed that the state,

$$0.6|0\rangle + 0.8|1\rangle = 0.6 \begin{bmatrix} 1 \\ 0 \end{bmatrix} + 0.8 \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0.6 \\ 0.8 \end{bmatrix}$$

Many other quantum states can be used to represent qubits. Any two-dimensional ket with the norm of 1 can be used to describe the state of a quantum system. A quantum state is expressed as $[\alpha, \beta\ ]^T$, with $|\alpha|^2 + |\beta|^2 = 1$. Therefore, this grants quantum computing a significant advantage over classical computing by allowing a quantum system with few qubits to represent more outcomes in faster execution times over classical systems with more classical bits.

2.1.2 Quantum Gates

Quantum gates are the analogs of classical logic gates. Since we want to preserve linearity, there is only one class of operators that can serve as quantum gates, namely unitary operators. Unitary operators U leave the norm invariant, so that $|U\Psi| = |\Psi|$ for all Ψ (Blinder and House 2019)

Quantum logic gates are reversible, unlike their classical counterparts. Quantum gates are represented by unitary matrices. Quantum gates largely operate on spaces of one or two qubits. This means that as matrices, quantum gates can be described by 2 x 2 matrices with orthonormal rows (Anonymous 2018).

2.1.3 Superposition

As mentioned previously, a qubit can possess a value of 0, 1, or a superposition of these two values. The numerical coefficient preceding the values in a quantum state represents the *amplitude of the wavefunction* represented by the quantum state. In physics, quantum superposition is used to describe an atom that can be thought to have two position coordinates simultaneously. It can also be visualised by the spin s of a one electron system, with the two states $+\frac{1}{2}$ and $\frac{-1}{2}$ being the two eigenstates of s_z

In a quantum computing simulator, in order to create a superposition of a qubit, the *Hadamard* or H gate is used. The Hadamard gate is given by:

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{2.2}$$

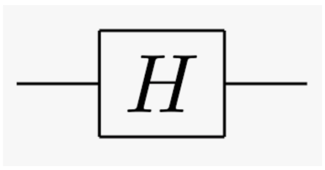


Figure 2.3: *Circuit representation of a Hadamard gate.*

As an example, we observe the application of H gate on $|1\rangle$ vector,

$$H|1\rangle = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \frac{|0\rangle - |1\rangle}{\sqrt{2}} = |-\rangle \tag{2.3}$$

After a superposition has been prepared, a qubit can be in superposition for a fixed time until the superpositon breaks down. This dissolution of superposition is termed as decoherence. The decohorence time is a key value that can be used to ascertain information about the quantum system and determine the error probability that each operation being performed on the qubit during superposition.

2.1.4 Entanglement

Quantum entanglement is a fundamental property of quantum systems that is utilised by quantum computers in essential applications such as quantum teleportation and quantum cryptography. A quantum state ψ is considered entangled if it cannot be expressed as the tensor product $|\psi_1\rangle \bigotimes |\psi_2\rangle$ of two pure quantum states.

$$|\psi\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}} \tag{2.4}$$

2.4 is an entangled quantum state since it clearly cannot be expressed as a tensor product of two pure quantum states.

Entanglement is a valuable property for quantum computers since it allows instantaneous communication between two entangled qubits, regardless of the distance between the two qubits. The quantum entanglement phenomenon was proved at the University of Delft (Hensen et al. 2015) where two photon beams were used to two entangle a pair of photons kept 1.2 kilometres apart from each other. The measurement of the state of $photon_1$ led to the decomposition of the wavefunction of $photon_2$, hence providing the observer the simultaneous measurement of the state of both photons.

In order to implement entanglement into a quantum system, a Hadamard gate must be applied to create a superposition, followed by a Controlled Pauli X Gate. A Controlled X gate is represented by a $4x4$ matrix as shown in 2.5

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{2.5}$$

The method for implementing these gates, along with the process of dealing with controlled qubits has been detailed in Chapter 3. The circuit implementation of quantum entanglement is displayed in Figure 5.1. It should be noted that the CNOT gate is equivalent to a reversible XOR gate in a classical computer. The quantum entanglement algorithm, while straightforward, is critical for the creation of a function quantum computing simulator. Details of the quantum entanglement algorithm, have been discussed in Chapter 5.

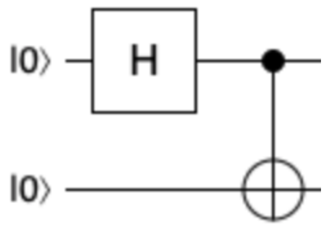


Figure 2.4: *Circuit representation of entangling two qubits gate.*

2.1.5 Measurement

A measurement being conducted on a qubit is equivalent of "peaking" at a qubit. Quantum measurement was popularised by the famous experiment of Schrodinger's cat(Forrester 2017). While Schrodinger's cat is in the box, it is unknown to the observer whether the cat is dead, or alive. The quantum state of the cat is said to be in superposition, it is dead and alive simultaneously. However, upon opening the box and observing the cat, the user measures the quantum state of the cat which leads to the collapse of the wavefunction. Similarly, for a qubit, if the quantum state is observed, or measured, the act of measurement leads to the collapse of the quantum state into either of the classical states $[0, 1]^T$ or $[1, 0]^T$. The probability of obtaining either state is given by the norm of the coefficient accompanying the state, earlier defined as the probability amplitude of the qubit. If a qubit, with a state vector given by $[\alpha, \beta]^T$ is measured, the probability of obtaining the 0 outcome is given by $|\alpha|^2$, similarly the probability of outcome 1 occuring is given by$|\beta|^2$. It must be noted that these probabilities must always sum up to 1 due to the normalisation condition $|\alpha|^2 + |\beta|^2 = 1$.

It should be noted that the act of measurement need not necessarily collapse all quantum states. If the circuit contains no operations and the input qubit is given by $|1\rangle$, then measuring this quantum state will always yield the state $|1\rangle$, leaving the state unchanged. Therefore, it is evident that a quantum computing architecture can be utilised to perform classical operations. However, the ability of storing information in a superposition of both states is what sets quantum computers apart from their classical counterparts(QuantumWriter n.d.).

2.2 Acceleration

Since the aim of the project is to create a simulator that is GPU compliant, it was decided that established GPU parallelisation technologies such as OpenACC and CUDA should be considered. While both were promising options, NVIDIA removed GPU suport for CUDA for MacOS(*CUDA Release Notes* n.d.) in November 2019. Therefore, a decision was made to proceed with GPU acceleration using OpenACC. Furthermore, due to the the use of "pragma" in OpenACC, the application can be accelerated on CPUs by using the flag "-ta=multicore". This was done in order to provide future researchers the opportunity to implement the API within their research and obtain significant speeds while putting forth minimal hurdles in their development.

3 Simulator Design

This chapter describes the overall design of QuantoSim, the quantum computing simulation library. QuantoSim consists of 3 classes,

- QRegister class represents a quantum register.
- VectorString class represents a state vector.
- Simulation class is necessary for displaying and printing values back to the user.

All visual representations of the library are generated by (*Doxygen* n.d.). The dependency graphs of all files in the library are available in Appendix C.

3.1 Quantum Register

The `QRegister` class is responsible for housing all VectorString objects in a quantum circuit. It is a user-facing class which provides users with functions critical to a quantum computer. The collaboration graph for the QRegister class is displayed in Figure 3.1

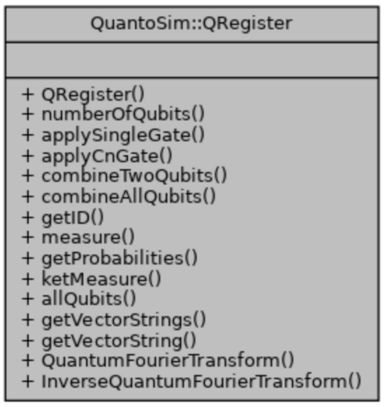


Figure 3.1: *Collaboration Diagram for QRegister.*

3.1.1 Important Functions

measure() function: The **measure** function is used to decompose the superposition of states and obtain a tangible state. The call graph of the function is represented in Figure 3.2

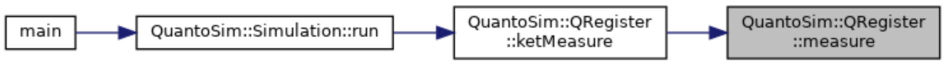

main
QuantoSim::Simulation::run
QuantoSim::QRegister ::ketMeasure
QuantoSim::QRegister ::measure

Figure 3.2: *Call graph for measure function.*

3.2 Gates

The Gates file houses a number of vectors of `complex<double>` datatype. These vectors serve as quantum gates to perform operations on qubits.

3.3 VectorString

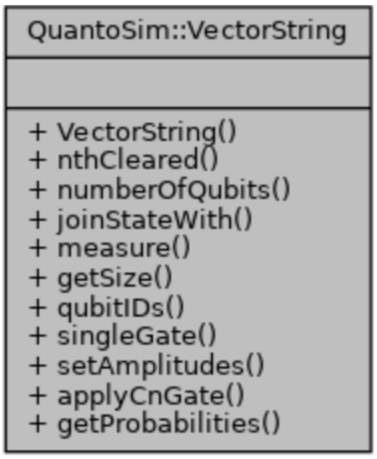


Figure 3.3: *Collaboration Diagram for VectorString.*

The VectorString class is meant to behave like a state vector for a qubit. The VectorString class houses all the background functions necessary for the operation of a quantum computer. Some of the most important functions, along with their associated call graphs can be found in the following subsection. The collaboration graph for the QRegister class is displayed in Figure 3.3

3.3.1 Important Functions

singleGate() function: The **singleGate** function is used to apply Pauli gates to a given qubit. The call graph of the function is represented in Figure 3.4

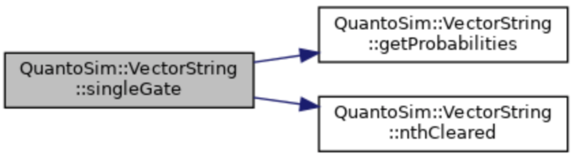

QuantoSim::VectorString ::singleGate
QuantoSim::VectorString ::getProbabilities
QuantoSim::VectorString ::nthCleared

Figure 3.4: *Call graph for singleGate function.*

The algorithm for singleGate function was proposed by (Kelly 2018) and is displayed down below.

Algorithm 1: GATE APPLICATION ALGORITHM (v, G, t)

```
Input: An $n$ qubit quantum state represented
       by a column vector $v = (v_1, \dots v_{2^n})^T$ and
       a single qubit gate $G$, represented by a
       $2 \times 2$ matrix, acting on the $t$th qubit.
1 for $i \leftarrow 0$ to $2^{n-1}$ do
2     $a \leftarrow$ the $i$th integer who's $t$th bit is 0;
3     $b \leftarrow$ the $i$th integer who's $t$th bit is 1;
      // The following must be
         simultaneously updated
4     $v_a \leftarrow v_a \cdot G_{0,0} + v_b \cdot G_{0,1}$;
5     $v_b \leftarrow v_b \cdot G_{1,1} + v_a \cdot G_{1,0}$;
```

applyCnGate() function: The **applyCnGate** function is used to apply Pauli gates to a given qubit with the use of a control qubit. The call graph of the function is represented in Figure 3.5

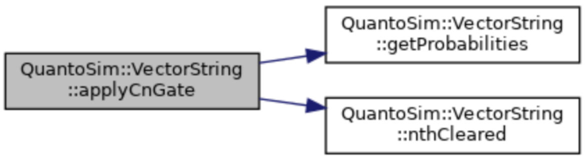

QuantoSim::VectorString ::applyCnGate
QuantoSim::VectorString ::getProbabilities
QuantoSim::VectorString ::nthCleared

Figure 3.5: *Call graph for applyCnGate function.*

The algorithm for applyCnGate function was proposed by (Kelly 2018) and is displayed down below.

Algorithm 2 Matrix-Free Controlled Single-Qubit Gate State Vector Update as per Kelly (2018)

```
for $i \leftarrow 0$ to $2^{n-1}$ do
    $a \leftarrow$ the $i$th integer whose $t$th bit is 0;
    $b \leftarrow$ the $i$th integer whose $t$th bit is 1;
    $c_a \leftarrow ((1 << c_{id}) \& a) > 0$;
    $c_b \leftarrow ((1 << c_{id}) \& b) > 0$;
    if $c_a$ then
        $v_a \leftarrow v_a \cdot G_{0,0} + v_b \cdot G_{0,1}$;
    end if
    if $c_b$ then
        $v_b \leftarrow v_a \cdot G_{1,0} + v_b \cdot G_{1,1}$;
    end if
end for
```

4 Simulator Implementation

QuantoSim, the quantum computing simulation library was implemented in C++. It is executed via `cmake`. The entire simulator has been implemented entirely using `.hpp` files. This is to increase portability and allow users to simply import the **quontosim** header file and utilise all quantum computing functions for themselves.

4.1 Minimising Memory Usage

While, mathematically, operations conducted on qubits are done in the form of matrix multiplication, this is highly inefficient. For the application of an n-qubit gate to a quantum state vector with m qubits, where m > n, the gate needs to be enlarged with continuous tensor products. This can lead to huge vectors with many dimensions, making gate applications incredibly slow. In order to circumvent this issue, Kelly (2018) proposes utilising multiple small 1-D vectors by using algorithms 1 and 2, mentioned in section 3. This quantum computing simulator, implements the algorithms laid out by (Kelly 2018) in order to create a matrix-independent simulator.

4.2 Qubits and State Vectors

The VectorString class was constructed in order to house the data structures necessary to read and iterate through qubits along with helper functions that provide values to QRegister. A VectorString object is a state vector which is initialized using the number of qubits present in the state vector as well as their corresponding addresses. Furthermore, constructing a VectorString initialises a random number generator which is used to obtain the probability amplitudes of all associated quantum states. A list of all combinations for n qubits of the two computational bases, $|1\rangle$ and $|0\rangle$ is also created.

```
VectorString(uint n, const vector<unsigned int> qIDs) : n(n),
     qIDs(qIDs),amps(vector<complex<double> >(pow(2, n)))

{
    amps.at(0) = 1.0;
    pnc = stateTP(n);
    probs = getProbabilities();
    random_device rd;
    gen = mt19937(rd());
```

Listing 4.1: VectorString object implementation.

4.3 Quantum Gates

All Pauli gates used in quantum computing simulation were successfully implemented. The gates were implemented as 1-D vectors of complex doubles. This was done to increase the efficiency of the simulator via efficient memory management. With the use of multi-dimensional arrays, a quantum register with n qubits leads to 2^n states. This may lead to sacrificing significant memory and drastic increases to execution time. The 1-D vectors were applied to qubits to generate similar results as those obtained through matrix multiplication with the aid of algorithms proposed by (Kelly 2018). These algorithms are discussed in greater detail, along with their implementation in 3.3.2. These gates were exposed to the user via the **applySingleGate(qID,Gate)** function, present in the **QRegister.hpp**.

The list of all gates implemented, along with a brief description of their effect on a quantum state are mentioned down below.

4.3.1 Hadamard Gate

The Hadamard gate is used to prepare two equal superpositions of a quantum computational state. It is a one qubit rotation which can be considered a generalised form of Fourier transforms.

```
Gate H = {
      1/sqrt(2), 1/sqrt(2),
      1/sqrt(2), -1/sqrt(2)};
```

Listing 4.2: *Hadamard gate vector implementation.*

4.3.2 X, Y, and Z Gates

X, Y, and Z are Pauli gates that are used to rotate the superposition along the X, Y, and Z axes respectively.

```
Gate X = {
      0,1,
      1,0};
Gate Z = {
      1,0,
      0,-1};
Gate Y = {
      0,-im,
      im,0};
```

Listing 4.3: *Pauli X, Z, and Y gate vector implementation.*

4.3.3 S Gate

The S gate is used to shift the phase of a qubit by $\pi/2$ around the Z axis.

```
Gate S = {
      1,0,
      0,exp(im*pi/2.0)};
```

Listing 4.4: *S gate vector implementation.*

4.3.4 T Gate

The T gate is used to shift the phase of a qubit by $\pi/4$ around the Z axis.

```
Gate T = {
      1,0,
      0,exp(im*pi/4.0)};
```

Listing 4.5: *T gate vector implementation.*

Phase Gate As the name suggests, phase gates are used to rotate the phase of a quantum state by x degrees.

```
Gate Rm(unsigned int x) {
   complex<double> theta = exp(2*M_PI*im/pow(2, x));
   Gate R({
      1,0,
      0,theta});
   return R;}
```

Listing 4.6: *Phase gate vector implementation.*

4.4 Performing Operations

Since state vectors with a larger amount of qubits require an incredible amount of memory, implementing large state vectors for matrix multiplication is highly inefficient. Instead, Kelly (2018) proposed a set of algorithms which allows the implementation of quantum states and quantum gates as 1-D vectors. These algorithms have been described in sections 3.3.1. With the use of these algorithms, quantum gates and states were implemented efficiently as 1-D vectors.

Single Gate application The implementation of Algorithm 1 by (Kelly 2018), present in section 3.3.1, is used to allow Pauli gate application in an efficient manner. Implementation of this algorithm is displayed down below.

CNOT Application (Kelly 2018) also proposes a new algorithm for the implementation of any Pauli gate as a controlled gate. This algorithm is shown section in 3.3.1. The implementation of the algorithm is demonstrated down below.

```
void applyCnGate(unsigned int control, unsigned int qID, Gate &gate) {
        // Refactor
        unsigned int qIndex = find(qIDs.begin(), qIDs.end(), qID)-qIDs.begin();
        for(int i = 0; i < pow(2,n-1); i++) {
           unsigned int cState = nthCleared(i, qIndex);
           unsigned int tState = cState | (1u << qIndex);
           complex<double> cAmp = amps.at(cState);
           complex<double> tAmp = amps.at(tState);
           if (((1u << control) & cState) > 0)
              amps.at(cState) = gate.at(0)*cAmp + gate.at(1)*tAmp;
           if(((1u << control) & tState) > 0)
              amps.at(tState) = gate.at(2)*cAmp + gate.at(3)*tAmp;
        }
```

```
void singleGate(unsigned int qID, Gate &gate){
   unsigned int qIndex = find(qIDs.begin(), qIDs.end(), qID)-qIDs.begin();
   for(unsigned int i = 0; i < pow(2,n-1); i++) {
      unsigned int cState = nthCleared(i, qIndex);
      unsigned int tState = cState | (1u << qIndex);
      complex<double> cAmp = amps.at(cState);
      complex<double> tAmp = amps.at(tState);
      amps.at(cState) = gate.at(0)*cAmp + gate.at(1)*tAmp;
      amps.at(tState) = gate.at(2)*cAmp + gate.at(3)*tAmp;
   }
   probs = getProbabilities(); // Remember to update probs each time amps is
       updated
}
```

Listing 4.7: *Qubit gate application implementation according to (Kelly 2018)*

```
      probs = getProbabilities(); // Remember to update probs each time amps is
          updated
   }
```

4.4.1 Performing Quantum Entanglement

As detailed in Chapter 2, entangling two qubits provides the user with the unique opportunity to instantly change the state of a qubit by changing the state of it's entangled counterpart. Upon measurement, if $qubit_0$ is found to be $|1\rangle$, then we instantly know the state value of $qubit_1$

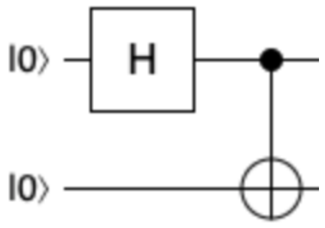


Figure 4.1: *Circuit representation of entangling two qubits gate.*

```
//Entangling function
void EntangleQubits(unsigned int qID1, unsigned int qID2) {
   applySingleGate(qID1, Gates::H);
   applyCnGate(qID1,qID2,Gates::X);
}
```

Listing 4.8: *Quantum entanglement implementation*

4.5 Making Measurements

In order to decompose the wavefunction of a qubit and obtain a value that is not in superposition, the **measure()** function was created. First, a random device was created in the VectorString class in order to generate probability amplitudes. Using the random device created, the probability amplitudes for vectors that needed to be measured were obtained and added to a result string. The result string was then displayed to the user by another function called **ketmeasure()** present in the QRegister class. The code implementation of the measurement functionality is displayed in Listing 4.8

```
\\Random device generation in VectorString Constructor
random_device rd;
gen = mt19937(rd());

\\Randomised string obtained using permution-combination helper function
orString::string measure(){
string m = pnc.at(d(gen));
return m;



ng QRegister::measure() {

string result(n, '-');
for(auto &v: vectors) {
   auto decomposed = v.measure();
   auto qIDs = v.qubitIDs();
   for(int i = 0; i < decomposed.length(); i++) {
      auto qID = qIDs.at(i);
      result.at(qID) = decomposed.at(i);
   }
}
return result;
```

Listing 4.9: Measure function implementation

4.6 Acceleration

The project was parallelised using OpenACC by NVIDIA.

In order to identify regions within the code that would benefit the most from parallelization, the **callgrind** tool was used to perform code profiling. The code profile can be viewed in Appendix A.

Distinct regions of the code which consumed a large amount of resources were identified. How these regions were updated has been discussed in further detail down below.

4.6.1 The VectorString measure() function

```
random_device rd;
gen = mt19937(rd());
pnc = stateTP(n);//Present within constructor
```

```
string measure() {
   string m = pnc.at(d(gen));
   return m;
}

vector<string> stateTP(unsigned int n) {
   vector<string> base = {"0", "1"};
   vector<string> newBase= {"0","1"};
   for(unsigned int i = 0; i < n-1; i++) newBase = product(newBase, base);
   return newBase;
```

Listing 4.10: Measure function optimisaton

Initially, a random number generator was initialized on every call to the **measure()** function. This was changed so that each VectorString Object will initialise a singular random number generator in the constructor.

Additionally, this method, and all its associated calls were parallelised using parallel loops in the **Simulation** header file.

4.6.2 The QRegister measure() function

```
string measure() {
   // TODO: Refactor
   string result(n, '-');
   for(auto &v: vectors) {
      auto decomposed = v.measure();
      auto qIDs = v.qubitIDs();
      for(int i = 0; i < decomposed.length(); i++) {
         auto qID = qIDs.at(i);
         result.at(qID) = decomposed.at(i);
      }
   }
   return result;
}
```

Listing 4.11: Measure function optimisaton in QRegister

The **QRegister::measure()** function increased runtime similarly, and was parallelised by moving its calls to **Simulation.hpp** where parallel for loops were used.

4.6.3 Random numbers and discrete probabilities

```
string measure() {
   // TODO: Refactor
   string result(n, '-');
   for(auto &v: vectors) {
      auto decomposed = v.measure();
      auto qIDs = v.qubitIDs();
      for(int i = 0; i < decomposed.length(); i++) {
         auto qID = qIDs.at(i);
```

```
            result.at(qID) = decomposed.at(i);
        }
    }
    return result;
}
```

Listing 4.12: Random number optimisation

The random number generator was found to consume a significant amount of resources. In order to minimise runtimes, the number of calls to the generator were minimised by attaching a singular number generator to each VectorString object instead of creating a new number generator object for each call to the **measure()** function.

5 Quantum Algorithms

A number of quantum algorithms were created in order to gain a deeper understanding about quantum computing principles. Additionally, these algorithms are essential for the creation of larger, and more complex circuits by the user. The key algorithms implemented were quantum entanglement, quantum Fourier transforms, and inverse quantum Fourier transforms. These algorithms are discussed down below.

5.1 Quantum Entanglement

As discussed in Chapter 2, quantum entanglement is a fundamental property of quantum systems that is utilised by quantum computers in essential applications such as quantum teleportation and quantum cryptography. Therefore, it was essential to create a method which would allow users to simply entangle two qubits in order to develop quantum information teleporters.

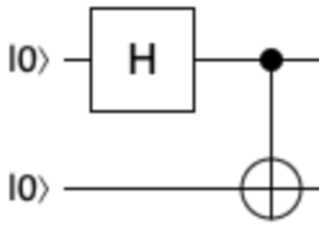


Figure 5.1: *Circuit representation of the entanglement algorithm.*

```
   //Entangling function
void EntangleQubits(unsigned int qID1, unsigned int qID2) {
   applySingleGate(qID1, Gates::H);
   applyCnGate(qID1,qID2,Gates::X);
}
```

Listing 5.1: *Random number optimisation*

5.2 Quantum Fourier Transforms

The reference for this section is Team (2020)

The Fourier transform is used in computers with a classical architecture for a number of applications ranging from data compression, sound systems to signal processing. A quantum Fourier transform attempts to implement the Fourier transform for qubits by modifying the probability amplitudes of a wavefunction. Furthermore, it was essential to implement the algorithm in order to provide users the ability to create Shor's factoring algorithm and quantum phase estimation.

The discrete Fourier transform acts on a vector $(x_0, ..x_{N-1})$ and maps it to the vector $(y_0, ..y_{N-1})$ according to the formula,

$$y_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} x_j w_N^{jk} \tag{5.1}$$

where $w_N^{jk} = e^{2\pi i \frac{jk}{N}}$

Similarly, the quantum Fourier transform acts on a quantum state $\sum_{i=0}^{N-1} x_i |i\rangle$ and maps it to the quantum state $\sum_{i=0}^{N-1} y_i |i\rangle$. The map function can be expressed as a unitary matrix given by the following equation,

$$U_{QFT} = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} \sum_{x=0}^{N-1} w_N^{xy} |y\rangle\langle x| \tag{5.2}$$

where $w_N^{jk} = e^{2\pi i \frac{jk}{N}}$

This method was implemented successfully within the project and is listed in Appendix B.

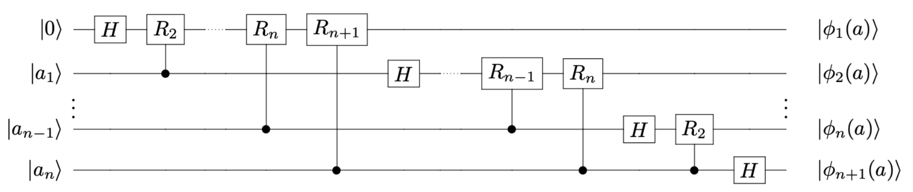


Figure 5.2: *Circuit representation of the quantum Fourier transform algorithm (Ruiz-Perez and Garcia-Escartin 2017)*

5.3 Inverse Quantum Fourier Transform

Additionally, an **inverse** quantum Fourier Transform function was defined such that,

$$IQFT|k\rangle = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} e^{-i\frac{2\pi x l}{N}} |x\rangle \tag{5.3}$$

This algorithm was referenced from (Ruiz-Perez and Garcia-Escartin 2017) and was created in order to allow users to move backwards and forwards between the computational basis and the phase representation. It should be noted that the conversion between phase encoding and computational basis is such that,

$$IQFT|\phi(x)\rangle = QFT^{-1}QFT|x\rangle = |x\rangle \tag{5.4}$$

6 Evaluation

NOTE: Evaluation results could not be obtained due to external circumstances relating to COVID-19. The evaluation methodology has been described below.

In this chapter, the performance and runtimes of the quantum computing simulator, QuantoSim, has been compared to the performances of other pre-existing simulators. The function `time` was used to calculate the runtimes of all simulators.

6.1 3rd Party Simulators

6.1.1 QuEST

QuEST is a performance driven quantum computing simulator written in C/C++(Jones et al. 2019). QuEST is the only open source simulator available that supports both CPU and GPU architectures. This makes QuEST a perfect candidate for a baseline for the developed simulator QuantoSim.

6.1.2 Quantum++

Quantum++ is another performance driven quantum computing simulator written in C/C++ (Gheorghiu 2018). While Quantum++ is easily deployable, it is not accelerated for GPU.

6.2 Evaluation Methodology

For all three simulators, a testing harness was created in order to benchmark the performance of QuantoSim against QuEST and Quantum++. The process of evaluation involved benchmarking elementary algorithms quantum entanglement, quantum Fourier transforms, and quantum inverse Fourier transforms on quantum registers comprising of 5, 10, 20, and 40 qubits. 1 million measurements for each of the algorithms were to be conducted, with the mean of 3 runs of the testing harness being taken as the measured runtime.

6.3 Sidenote

While the testing harness was developed for GPU benchmarking and could not be utilised for obtaining results due to external circumstances relating to COVID 19 which prevented access to university GPU infrastructure. It was however observed that, upon executing QuantoSim on a CPU after implementing parallelisation and conducting optimisations mentioned in section 4.6, a 2.92x acceleration was observed over the serialised implementation of QuantoSim. These primary results can be found in Appendix A.

7 Conclusion

7.1 Project Summary

The goals of this project and their degree of completion have been summarised down below.

7.1.1 Learning quantum computation fundamentals

The primary goal of this project was to learn with great detail the fundamental concepts necessary to develop quantum computers. This was achieved by:

- Successful creation of a quantum computing simulator which can create multiple quantum registers that are capable of containing multi-qubit state vectors.
- Successful implementation of Pauli and controlled gates were implemented into the project in order to gain a deeper understanding of quantum gate operations.
- Successful comprehension of quantum algorithms essential for developing scalable quantum computers and their subsequent implementation.

7.1.2 Development of a user-friendly and optimised simulator

The secondary goal of this project was to develop a simulator which can be used with ease by new users and researchers and ensure that it was optimised. This was achieved by,

- Successful creation of the simulator as a library containing `.hpp` files which can simply be imported to an existing project.
- Successful compatibility with CPU and GPU architectures.
- Successful implementation of parallelisation to optimise the simulator. On a CPU infrastructure, the simulator had a 2.92x increase in performance after parallelisation.

Although the product could not be evaluated against other pre-existing simulators, the primary and secondary goals were still achieved.

7.2 Future Work

7.2.1 Support for convoluted controlled gates

While the simulator is able to create controlled gates, these gates currently accept a single qubit as a control. Future implementation will involve allowing the user to pass in a vector containing multiple qubit addresses as controls.

7.2.2 Development of a custom random number generator

Due to OS X incompatibility with GPU acceleration offered by CUDA, openACC was implemented. However, if I have access to a windows machine, I intend to implement cuRand by CUDA for random number generation.

7.2.3 Additional support for QFT adders

Although the simulator supports QFT and inverse QFT, support for QFT adders will be implemented in future.

7.2.4 Addition of Eigen library to replace std:vector

Eigen (*Eigen* n.d.) is a highly optimised linear algebra library for C++. It could not be implemented in the current project due to the lack of support for GPUs. In future, data structures will be converted to Eigen in order to further optimise the simulator after modifying the Eigen library to be GPU compatible.

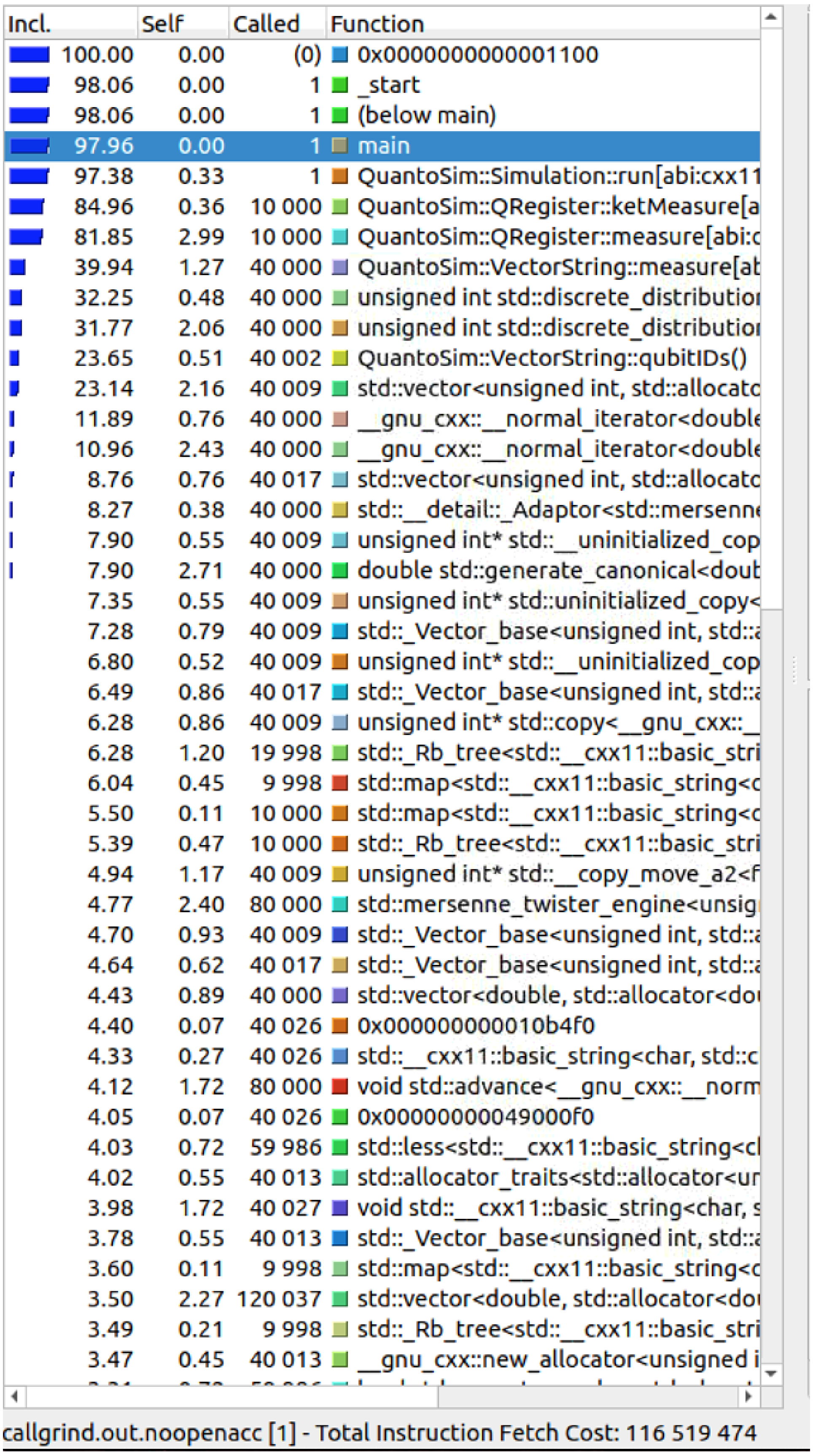
7.3 Personal Reflection

I really enjoyed learning more about quantum computing fundamentals by designing the simulation library from the bottom up. While I have been programming for 4 years, I never paid much attention to the elements that make up a computer. During this project, I learned with great detail, the fundamental concepts behind the development of both, classical and quantum architectures. Furthermore, as a Joint Honours student in Computing Science and Physics, I had a great deal of interest in learning about the process of implementation of quantum physics concepts into a practical product. Following this project, I intend to pursue higher education in the quantum technology domain.

A Code Profiles

The code profiles of the project pre-parallelisation and post parallelisation are added in the Appendix. It can be observed that the functions with the largest memory demands are the measure functions as well as the random number generator.

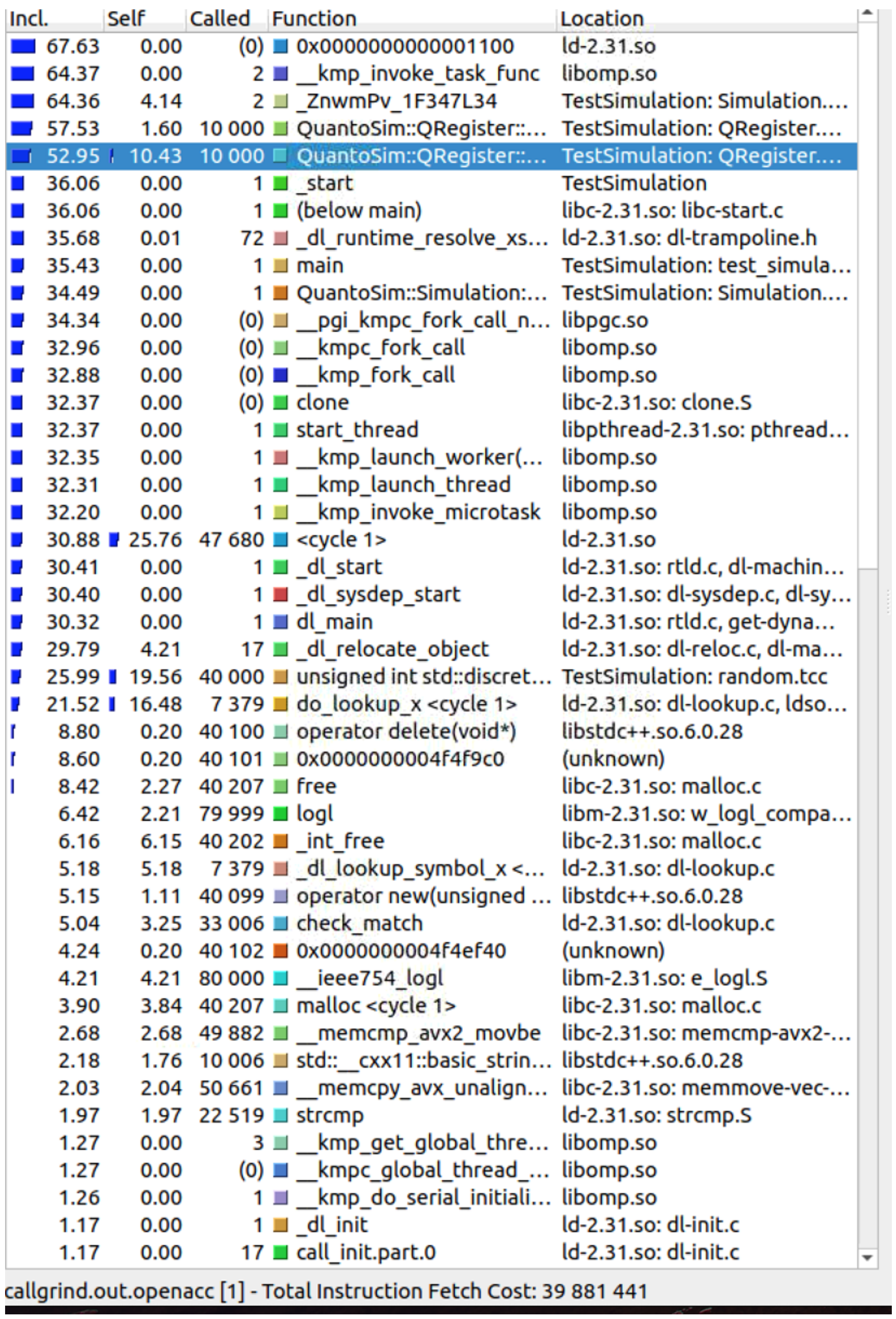
Furthermore, execution times of the simulator pre- and post-parallelisation are also added in this appendix. Code profiling was conducted with the aid of callgrind tool.



Incl.	Self	Called	Function
100.00	0.00	(0)	0x0000000000001100
98.06	0.00	1	_start
98.06	0.00	1	(below main)
97.96	0.00	1	main
97.38	0.33	1	QuantoSim::Simulation::run[abi:cxx11
84.96	0.36	10 000	QuantoSim::QRegister::ketMeasure[a
81.85	2.99	10 000	QuantoSim::QRegister::measure[abi:c
39.94	1.27	40 000	QuantoSim::VectorString::measure[ab
32.25	0.48	40 000	unsigned int std::discrete_distributio
31.77	2.06	40 000	unsigned int std::discrete_distributio
23.65	0.51	40 002	QuantoSim::VectorString::qubitIDs()
23.14	2.16	40 009	std::vector<unsigned int, std::allocato
11.89	0.76	40 000	__gnu_cxx::__normal_iterator<double
10.96	2.43	40 000	__gnu_cxx::__normal_iterator<double
8.76	0.76	40 017	std::vector<unsigned int, std::allocato
8.27	0.38	40 000	std::__detail::_Adaptor<std::mersenne
7.90	0.55	40 009	unsigned int* std::__uninitialized_cop
7.90	2.71	40 000	double std::generate_canonical<doub
7.35	0.55	40 009	unsigned int* std::uninitialized_copy<
7.28	0.79	40 009	std::_Vector_base<unsigned int, std::a
6.80	0.52	40 009	unsigned int* std::__uninitialized_cop
6.49	0.86	40 017	std::_Vector_base<unsigned int, std::a
6.28	0.86	40 009	unsigned int* std::copy<__gnu_cxx::_
6.28	1.20	19 998	std::_Rb_tree<std::__cxx11::basic_stri
6.04	0.45	9 998	std::map<std::__cxx11::basic_string<c
5.50	0.11	10 000	std::map<std::__cxx11::basic_string<c
5.39	0.47	10 000	std::_Rb_tree<std::__cxx11::basic_stri
4.94	1.17	40 009	unsigned int* std::__copy_move_a2<f
4.77	2.40	80 000	std::mersenne_twister_engine<unsig
4.70	0.93	40 009	std::_Vector_base<unsigned int, std::a
4.64	0.62	40 017	std::_Vector_base<unsigned int, std::a
4.43	0.89	40 000	std::vector<double, std::allocator<do
4.40	0.07	40 026	0x000000000010b4f0
4.33	0.27	40 026	std::__cxx11::basic_string<char, std::c
4.12	1.72	80 000	void std::advance<__gnu_cxx::__norm
4.05	0.07	40 026	0x00000000049000f0
4.03	0.72	59 986	std::less<std::__cxx11::basic_string<c
4.02	0.55	40 013	std::allocator_traits<std::allocator<un
3.98	1.72	40 027	void std::__cxx11::basic_string<char, s
3.78	0.55	40 013	std::_Vector_base<unsigned int, std::a
3.60	0.11	9 998	std::map<std::__cxx11::basic_string<c
3.50	2.27	120 037	std::vector<double, std::allocator<do
3.49	0.21	9 998	std::_Rb_tree<std::__cxx11::basic_stri
3.47	0.45	40 013	__gnu_cxx::new_allocator<unsigned i

callgrind.out.noopenacc [1] - Total Instruction Fetch Cost: 116 519 474

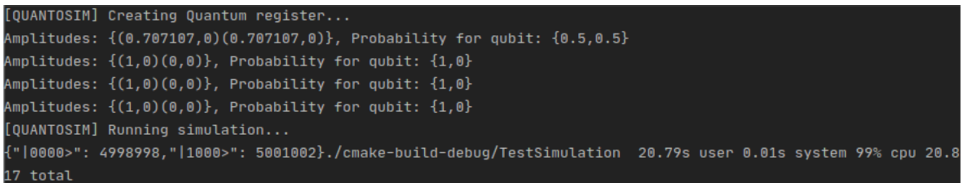
Figure A.1: Code profile of unoptimised QuantoSim



Incl.	Self	Called	Function	Location
67.63	0.00	(0)	0x0000000000001100	ld-2.31.so
64.37	0.00	2	__kmp_invoke_task_func	libomp.so
64.36	4.14	2	_ZnwmPv_1F347L34	TestSimulation: Simulation.…
57.53	1.60	10 000	QuantoSim::QRegister::…	TestSimulation: QRegister.…
52.95	10.43	10 000	QuantoSim::QRegister::…	TestSimulation: QRegister.…
36.06	0.00	1	_start	TestSimulation
36.06	0.00	1	(below main)	libc-2.31.so: libc-start.c
35.68	0.01	72	_dl_runtime_resolve_xs…	ld-2.31.so: dl-trampoline.h
35.43	0.00	1	main	TestSimulation: test_simula…
34.49	0.00	1	QuantoSim::Simulation::…	TestSimulation: Simulation.…
34.34	0.00	(0)	__pgi_kmpc_fork_call_n…	libpgc.so
32.96	0.00	(0)	__kmpc_fork_call	libomp.so
32.88	0.00	(0)	__kmp_fork_call	libomp.so
32.37	0.00	(0)	clone	libc-2.31.so: clone.S
32.37	0.00	1	start_thread	libpthread-2.31.so: pthread…
32.35	0.00	1	__kmp_launch_worker(…	libomp.so
32.31	0.00	1	__kmp_launch_thread	libomp.so
32.20	0.00	1	__kmp_invoke_microtask	libomp.so
30.88	25.76	47 680	<cycle 1>	ld-2.31.so
30.41	0.00	1	_dl_start	ld-2.31.so: rtld.c, dl-machin…
30.40	0.00	1	_dl_sysdep_start	ld-2.31.so: dl-sysdep.c, dl-sy…
30.32	0.00	1	dl_main	ld-2.31.so: rtld.c, get-dyna…
29.79	4.21	17	_dl_relocate_object	ld-2.31.so: dl-reloc.c, dl-ma…
25.99	19.56	40 000	unsigned int std::discret…	TestSimulation: random.tcc
21.52	16.48	7 379	do_lookup_x <cycle 1>	ld-2.31.so: dl-lookup.c, ldso…
8.80	0.20	40 100	operator delete(void*)	libstdc++.so.6.0.28
8.60	0.20	40 101	0x0000000004f4f9c0	(unknown)
8.42	2.27	40 207	free	libc-2.31.so: malloc.c
6.42	2.21	79 999	logl	libm-2.31.so: w_logl_compa…
6.16	6.15	40 202	_int_free	libc-2.31.so: malloc.c
5.18	5.18	7 379	_dl_lookup_symbol_x <…	ld-2.31.so: dl-lookup.c
5.15	1.11	40 099	operator new(unsigned …	libstdc++.so.6.0.28
5.04	3.25	33 006	check_match	ld-2.31.so: dl-lookup.c
4.24	0.20	40 102	0x0000000004f4ef40	(unknown)
4.21	4.21	80 000	__ieee754_logl	libm-2.31.so: e_logl.S
3.90	3.84	40 207	malloc <cycle 1>	libc-2.31.so: malloc.c
2.68	2.68	49 882	__memcmp_avx2_movbe	libc-2.31.so: memcmp-avx2-…
2.18	1.76	10 006	std::__cxx11::basic_strin…	libstdc++.so.6.0.28
2.03	2.04	50 661	__memcpy_avx_unalign…	libc-2.31.so: memmove-vec-…
1.97	1.97	22 519	strcmp	ld-2.31.so: strcmp.S
1.27	0.00	3	__kmp_get_global_thre…	libomp.so
1.27	0.00	(0)	__kmpc_global_thread_…	libomp.so
1.26	0.00	1	__kmp_do_serial_initiali…	libomp.so
1.17	0.00	1	_dl_init	ld-2.31.so: dl-init.c
1.17	0.00	17	call_init.part.0	ld-2.31.so: dl-init.c

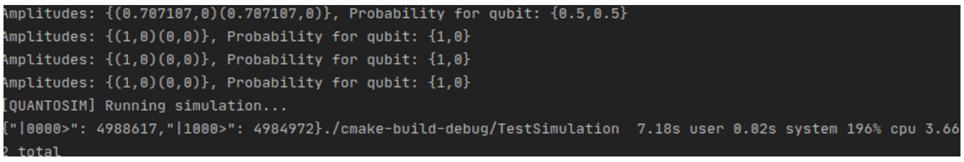
callgrind.out.openacc [1] - Total Instruction Fetch Cost: 39 881 441

Figure A.2: Code profile of optimised QuantoSim



```
[QUANTOSIM] Creating Quantum register...
Amplitudes: {(0.707107,0)(0.707107,0)}, Probability for qubit: {0.5,0.5}
Amplitudes: {(1,0)(0,0)}, Probability for qubit: {1,0}
Amplitudes: {(1,0)(0,0)}, Probability for qubit: {1,0}
Amplitudes: {(1,0)(0,0)}, Probability for qubit: {1,0}
[QUANTOSIM] Running simulation...
{"|0000>": 4998998,"|1000>": 5001002}./cmake-build-debug/TestSimulation  20.79s user 0.01s system 99% cpu 20.8
17 total
```

Figure A.3: *Execution times of optimised QuantoSim*



```
mplitudes: {(0.707107,0)(0.707107,0)}, Probability for qubit: {0.5,0.5}
mplitudes: {(1,0)(0,0)}, Probability for qubit: {1,0}
mplitudes: {(1,0)(0,0)}, Probability for qubit: {1,0}
mplitudes: {(1,0)(0,0)}, Probability for qubit: {1,0}
[QUANTOSIM] Running simulation...
{"|0000>": 4988617,"|1000>": 4984972}./cmake-build-debug/TestSimulation  7.18s user 0.02s system 196% cpu 3.66
2 total
```

Figure A.4: *Execution times of optimised QuantoSim*

B | Algorithms

The implementations of the quantum entanglement algorithm, quantum Fourier transform, inverse quantum Fourier transform, are provided in this section.

```
void EntangleQubits(unsigned int qID1, unsigned int qID2) {
   applySingleGate(qID1, Gates::H);
   applyCnGate(qID1,qID2,Gates::X);
```

Listing B.1: *Quantum entanglement implementation*

```
void QuantumFourierTransform(vector<unsigned int> qIDs) {
   for(int qID = 0; qID < qIDs.size(); qID++) {
      vectors[getID(qID)].singleGate(qID, Gates::H);
      for(int otherqID = qID+1; otherqID < qIDs.size(); otherqID++) {
         applyCnGate(getID(otherqID), getID(qID), Gates::Rm(otherqID-qID+1));
      }
   }
}
```

Listing B.2: *Quantum Fourier transform implementation*

```
void InverseQuantumFourierTransform(vector<unsigned int> qIDs) {
   for(int qID = qIDs.size() - 1; qID >= 0; qID--) {
      for(int otherqID = qIDs.size() - 1; otherqID > qID; otherqID--) {
         applyCnGate(getID(otherqID), getID(qID), Gates::Rm(-(otherqID-qID+1)));
      }
      applySingleGate(getID(qID), Gates::H);
   }
}
```

Listing B.3: *Inverse quantum Fourier transform implementation*

C Dependency Graphs

The dependency graphs for all major classes is added in this section.

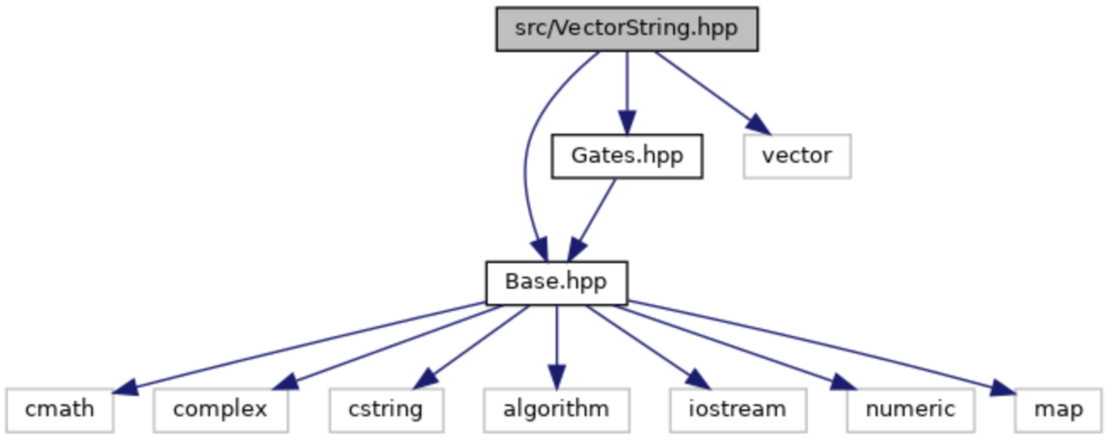


Figure C.1: Dependency graph of `VectorString.hpp`

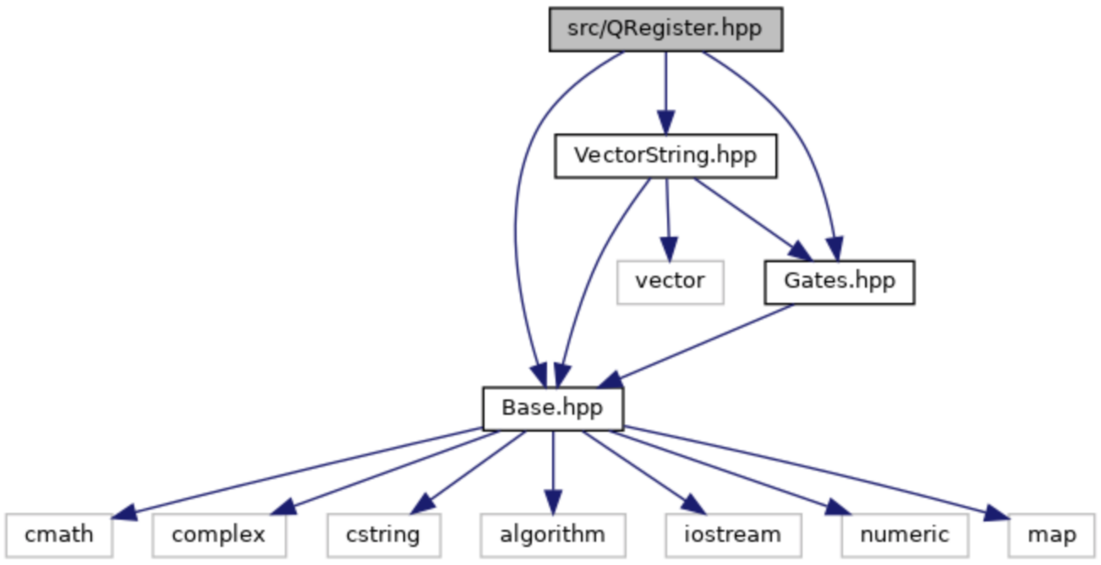


Figure C.2: Dependency graph of `QRegister.hpp`

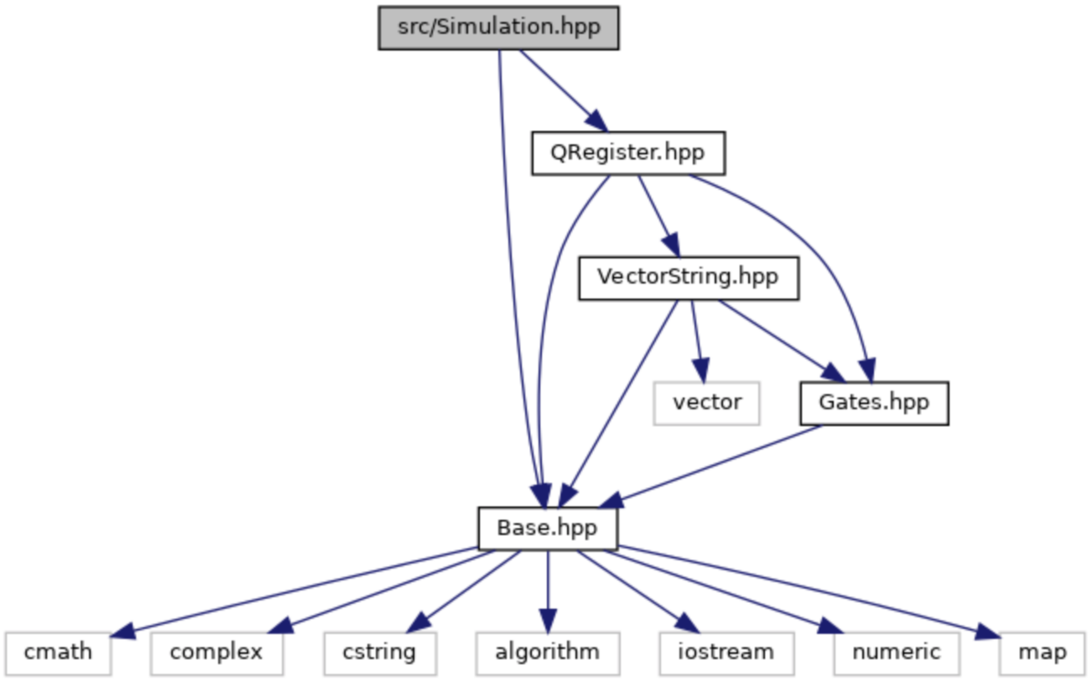


Figure C.3: Dependency graph of `Simulation.hpp`

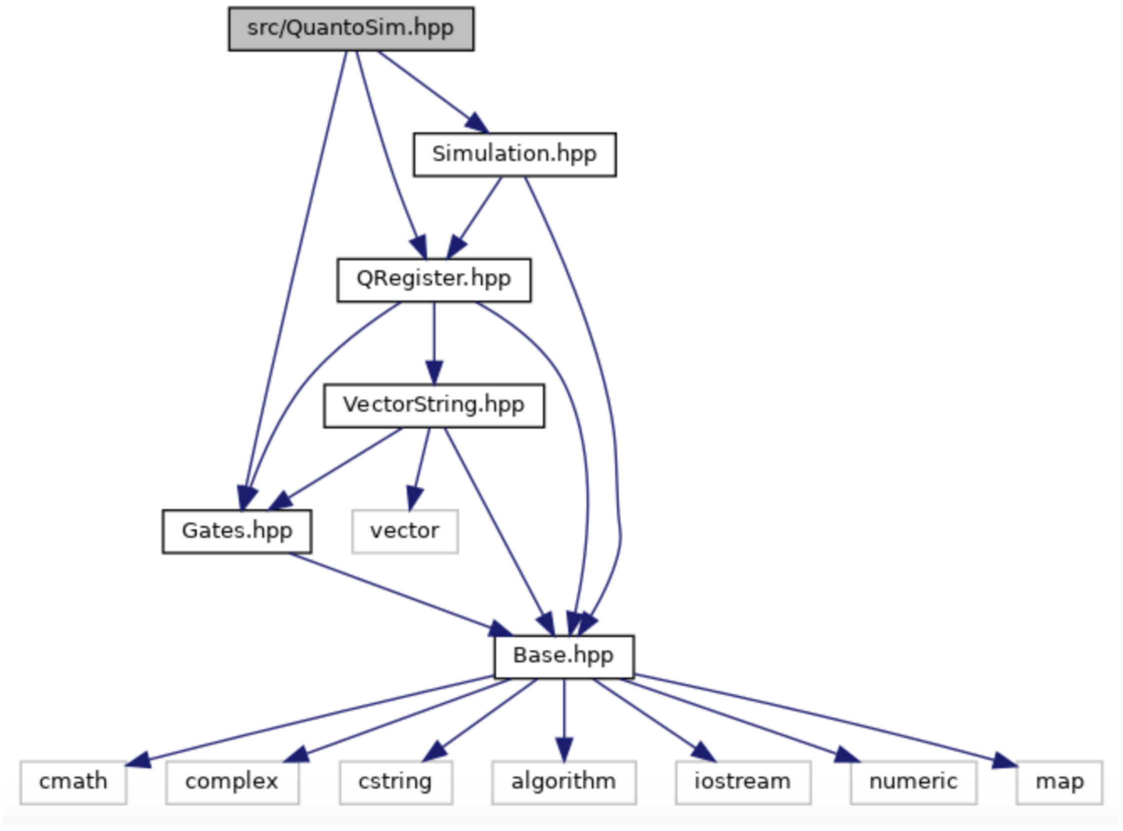


Figure C.4: Dependency graph of `QuontoSim.hpp`

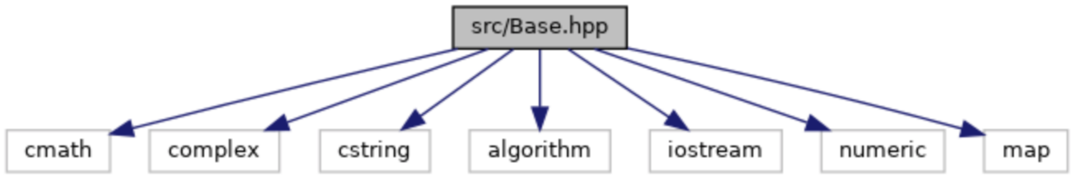


Figure C.5: Dependency graph of `Base.hpp`

D Deployment

The library uses CMAKE to compile and execute. Some changes may be required in `CMakeLists.txt` at the discretion of the user in order to build the project in their environment. In order to enable GPU acceleration, the PGI compiler must be toggled on in `CMakeLists.txt`. Please note that, by default, the GPU acceleration is set to off.

Bibliography

Anonymous (2018), 'Quantum gates'.
URL: *https://www.quantiki.org/wiki/quantum-gates*

Blinder, S. M. and House, J. E. (2019), *Mathematical physics in theoretical chemistry*, Elsevier.

Bracker, A. S. and Gammon, D. (n.d.), 'Using light to prepare and probe an electron spin in a quantum dot', *Electronics Science and Technology Division* .
URL: *https://www.nrl.navy.mil/content$_i$mages/05FA3.pdf*

Butscher, B. and Weimer, H. (n.d.), 'libquantum - simulation of quantum mechanics'.
URL: *http://libquantum.de/*

Cross, A. W., Bishop, L. S., Smolin, J. A. and Gambetta, J. M. (2017), 'Open quantum assembly language'.

CUDA Release Notes (n.d.).
URL: *https://docs.nvidia.com/cuda/cuda-toolkit-release-notes/index.htmltitle-new-features*

Doxygen (n.d.).
URL: *http://www.doxygen.nl/*

Eigen (n.d.).
URL: *https://eigen.tuxfamily.org/dox/*

Engineering, N. A. o. (2019), *Frontiers of Engineering: Reports on Leading-Edge Engineering from the 2018 Symposium*, National Academies Press.

Forrester, R. (2017), 'The quantum measurement problem: Collapse of the wave function explained', *SSRN Electronic Journal* .
URL: *https://papers.ssrn.com/sol3/papers.cfm?abstract$_i$d* = 2901820

Gheorghiu, V. (2018), 'Quantum++: A modern c++ quantum computing library', *PLOS ONE* **13**(12), e0208073.
URL: *http://dx.doi.org/10.1371/journal.pone.0208073*

Hensen, B., Bernien, H., Dréau, A., Reiserer, A., Kalb, N., Blok, M., Ruitenberg, J., Vermeulen, R., Schouten, R., Abellan, C., Amaya, W., Pruneri, V., Mitchell, M., Markham, M., Twitchen, D., Elkouss, D., Wehner, S., Taminiau, T. and Hanson, R. (2015), 'Loophole-free bell inequality violation using electron spins separated by 1.3 kilometres', *Nature* **526**.

Jones, T., Brown, A., Bush, I. and Benjamin, S. C. (2019), 'Quest and high performance simulation of quantum computers', *Scientific Reports* **9**(1).

Kelly, A. (2018), 'Simulating quantum computers using opencl'.

Nielsen, M. A. and Chuang, I. L. (2019), *Quantum computation and quantum information*, Cambridge University Press.

Quantum computing and defence (n.d.).
URL: *https://www.iiss.org/publications/the-military-balance/the-military-balance-2019/quantum-computing-and-defence*

QuantumWriter (n.d.), 'The qubit in quantum computing - microsoft quantum'.
URL: *https://docs.microsoft.com/en-us/quantum/concepts/the-qubitmeasuring-a-qubit*

Ruiz-Perez, L. and Garcia-Escartin, J. C. (2017), 'Quantum arithmetic with the quantum fourier transform', *Quantum Information Processing* **16**(6).

Team, Q. (2020), 'Quantum fourier transform'.
URL: *https://qiskit.org/textbook/ch-algorithms/quantum-fourier-transform.html*

Vela, H. (2019), 'How quantum computers are transforming travel'.
URL: *https://eandt.theiet.org/content/articles/2019/04/how-quantum-computers-are-transforming-travel/*

Yanofsky, N. S. and Mannucci, M. A. (2018), *Quantum computing for computer scientists*, Cambridge University Press.

Zapata (n.d.), 'Zapata'.
URL: *https://www.zapatacomputing.com/orquestra/*