# Simulating Quantum Circuits on a Classical Computer

**Mattias Lindberg**

Department of Physics
Stockholm University
Degree 15 HE credits
Spring term 2024
Supervisor: Ingemar Bengtsson

Stockholm University

# Simulating Quantum Circuits on a Classical Computer

**Mattias Lindberg**

## Abstract

This thesis examines how a quantum computer can be simulated using a classical computer. It provides a theoretical background to the basics of qubits and to quantum computing, including a mathematical representation for quantum states and gates using vectors and matrices. This background is then applied through the implementation of a simulator that is able to execute quantum circuits written in the quantum programming language OpenQASM on a classical computer. Finally, the simulator is used to implement Quantum Fourier Transform (QFT) and Shor's algorithm as OpenQASM programs.

**Keywords:** Quantum computing, Simulator, Qubit, Circuit model, Quantum gate, QFT, Shor's algorithm.

# Contents

# 1 Introduction

In 1981 Richard Feynman [5] introduced the idea of using a computer based on concepts from quantum mechanics to simulate how quantum systems behave. He argued that a classical computer is not useful to simulate real world quantum systems because the time and resources required grows exponentially with the size of the problem. However, the resource consumption in a quantum computer only grows linearly with the size of the quantum problem. This makes a computer based on quantum effects much more suitable for solving quantum problems, we call this type of computer a *quantum computer*.

In the beginning there were no quantum computers and even today there are very few of them and they are relatively small. In order to develop the theoretical models required for programming a quantum computer and to validate algorithms developed for use on a quantum computer researchers developed *simulators* for quantum computers that could be executed on a classical computer. These simulators use a mathematical model of quantum mechanics to immitate how a real quantum computer would behave, given a program and a set of input parameters.

The purpose of this thesis is to learn about the techniques used to simulate a quantum computer on a classical computer and to create a program that can simulate the basics of how a quantum computer behaves. At the start of the project there were six objectives: *Theoretical basis for classical simulation of quantum circuits*, *Noise and error correction*, *Develop a quantum circuit simulator for a classical computer*, *Quantum algorithm*, *GPU acceleration* and *Floating point precision*.

Halfway into the project *GPU acceleration* and *Floating point precision* were excluded because they were less relevant for the purpose of the thesis, they focused too much on the performance aspect of the simulator. The objective *Noise and error correction* has been reduced to a theoretical part that is included in the thesis, but it has not been implemented in the simulator.

This chapter introduces concepts that are used throughout the rest of the thesis, it is recommended that the reader become familiar with these concepts before reading the other chapters. Chapter 2 *Simulation of Quantum Circuits* introduces the mathematical model used for simulating quantum circuits and also explores other topics (such as noise and error correction).

After having focused on theory in the first two chapters the following two chapters describe the results from the implementation of the simulator. Chapter 3 *Simulator Design* explains key design decisions made during the implementation of the simulator. To get some real use from a quantum computer (or simulator) we need to use it to solve a problem, chapter 4 *Quantum Algorithms* discusses how the simulator was used to support QFT (Quantum Fourier Transform) and Shor's algorithm.

Finally, chapter 5 summarizes the work performed during this thesis project.

## 1.1 Anatomy of a Quantum Program

There are several different techniques used to physically realize a quantum computer, they include but are not limited to: ion traps, nuclear magnetic resonance and harmonic oscillators. These will not be further discussed in this thesis, they are out-of-scope just like the design of a silicon chip is not included when discussing the design and implementation of a program on classical computers. It is sufficient to say that there exist physical realizations of quantum computers and they allow for quantum programs to be executed on them.

In the context of quantum computing we do not use the term quantum program, instead we use *quantum circuit* to represent the programming logic that is executed on a quantum computer. Figure 1 shows the main concepts that are involved in a quantum circuit. A quantum circuit consists of a sequence of operations (called quantum gates) that each operate on one, two or sometimes three qubits. The input to the circuit is a system state on which the circuit operates, the circuit produces a system state as output. Each of these terms are further explained in the following sections.

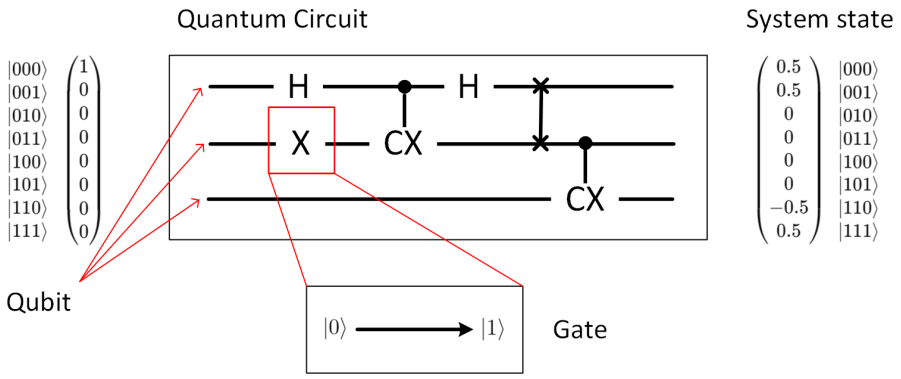



Figure 1: Concepts of a quantum circuit.

Another characteristic of a quantum circuit is that it does not execute on its own. A normal scenario is that a researcher provides input to a program on a classical computer, that program then prepares the system state that is provided as input to the quantum circuit and it also interprets the output from the quantum circuit. Finally, the classical program produces an output to the researcher. This way the classical program can do what it is good at (high-level structured programs) while the quantum circuit provides a quantum advantage in areas where it is more efficient. Figure 2 illustrates this relationship. An example of this structure is how Shor's algorithm is structured in the simulator implemented as part of this thesis, the classical part is described in chapter 4.5 and the quantum part is described in chapter 3.6.1.

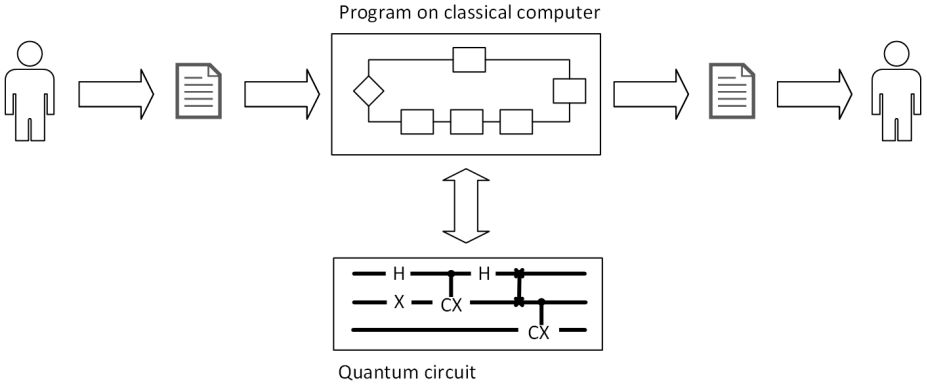



Figure 2: Classical computer using a quantum computer to solve a problem.

### 1.1.1 Qubit

The bearer of information in a quantum system is the *qubit*. To introduce the concept of a qubit we start with a more familiar concept, a *bit* in a classical computer (like the laptop you use to write your homework on). A bit can take two values (0 and 1) and you can combine many bits to represent information, e.g., the character 'Q' can be represented as 01010001 when you save it in a text file. An important characteristic of a classical bit is that it has a specific value, it is either 0 or 1 and never anything in between. When you save the sequence of bits representing Q you don't want them to change their value, because if you read them the sequence would no longer represent a Q.

Using similar language to the above a first attempt at describing a qubit could be as a *quantum bit* that can take the values $|0\rangle$ or $|1\rangle$ [1]. This is an oversimplified description because a qubit is much more complex than a classical bit, but for those unfamiliar with qubits it may serve as a way to get used to the term (which we will use a lot in this thesis).

A more correct characterization is that a qubit is a two-state quantum system that can be in a superposition of $|0\rangle$ and $|1\rangle$. This means that the qubit does not have a specific value, it represents a probability distribution of the two possible states. It is not limited to be either-or, it can also be both.

Equation 1 illustrates that the quantum state $|\Psi\rangle$ has one probability to be in state $|0\rangle$ and another to be in state $|1\rangle$. Note that $\alpha$ and $\beta$ are complex numbers and the probability to measure $|0\rangle$ is $|\alpha|^2$ and the probability for $|1\rangle$ is $|\beta|^2$.

$$|\Psi\rangle = \alpha|0\rangle + \beta|1\rangle \tag{1}$$

When we read the value of the qubit we say that we perform a *measurement*. The measurement forces the qubit to assume one of the possible states, using

[1] The notation $|0\rangle$ or $|1\rangle$ is called the Dirac notation a.k.a. bra-ket notation. It is used to describe the basis states of a quantum system.

quantum mechanical terminology we say that the qubit *collapses* from its indetermined probabilistic state to a specific state. This means that if we perform repeated measurements of the qubit then we will always get the same result, the qubit has assumed a value and it will not change. This is illustrated in Figure 3 below.

$$|\Psi\rangle = \alpha|0\rangle + \beta|1\rangle \xrightarrow{measurement} \begin{cases} |0\rangle & \text{with probability } |\alpha|^2 \\ |1\rangle & \text{with probability } |\beta|^2 \end{cases}$$

Figure 3: Measuring the state of a qubit causes it to collapse to a specific value.

Once again there is more depth to be added to the description. In quantum mechanics you are not restricted to only describe and measure the quantum state using $|0\rangle$ and $|1\rangle$ (a.k.a. the *computational basis*). We can define a new basis where $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and $|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$ and make a measurement using this basis, the quantum state will then collapse to one of $|+\rangle$ or $|-\rangle$. If we repeat the measurement multiple times it will always yield the same result as when we first measured it using the $|+\rangle/|-\rangle$ basis.

Assume that we first measure using the computational basis and get the result $|1\rangle$ and then measure using the $|+\rangle/|-\rangle$ basis and get the result $|+\rangle$. Now, what result will we get if we now perform a measurement in the computational basis? In classical physics you would expect to get $|1\rangle$, the length of something doesn't change when we measure its weight. Also, we have said that the inital measurement in the computational basis collapse the qubit to the state $|1\rangle$, so we would expect the result to be $|1\rangle$ once again.

However, in quantum mechanics when we perform a measurement using one basis we disturb the quantum state with regard to any other basis. The result of this is that if we first measure $|1\rangle$ and then measure $|+\rangle$ and then perform another measurement in the computational basis the result is just as indetermined as it was before the first measurement. The qubit is once again in a superposition of $|0\rangle$ and $|1\rangle$ and which one of these states that we will get from the measurement is not determined until we perform the measurement.

Above we have described measurements as an abstract mechanism that somehow gives us $|0\rangle/|1\rangle$ or $|+\rangle/|-\rangle$. In a physics lab we can make concrete measurements of the physical properties of a particle, e.g. we can measure the polarization of an electron along different axis. If we measure polarization along the z-axis and get the value $+\frac{1}{2}$ we say that the state is $|0\rangle$ and if we get $-\frac{1}{2}$ we say that the state is $|1\rangle$. Similarly, if we measure polarization along the x-axis we interpret the result as $|+\rangle$ or $|-\rangle$.

When you measure using different basis in classical computing you can get results that look different, e.g. 10 (decimal) and $A$ (hexadecimal), but changing basis in classical computing does not alter the actual result and has more to do with visualization of the result. This is a key difference when compared to quantum mechanics where the result is not known when you change basis and perform a measurement.

If you store information using qubits that are in a superposition you don't know what you will get back and this makes qubits a bad choice for storing information. However, this is also the strength of qubits! Instead of focusing on the inability to recover an exact state from a qubit, let us consider that a single qubit can be seen as storing more information when compared to a classical bit. Because the qubit, through superposition, can be both $|0\rangle$ and $|1\rangle$ at the same time.

Using this quantum mechanical property of qubits in clever ways is of part of what makes it possible to create quantum algorithms that perform better than classical algorithms. This will be further discussed in chapter 4.

### 1.1.2 System state

Just like we can construct a sequence of classical bits and say that it holds information, we can also construct a sequence of qubits and say that it holds information. The information carried by a sequence of qubits is called the *system state* or *quantum state* of a quantum system.

Using the computational basis, a two-qubit system has the basis states $|00\rangle$, $|01\rangle$, $|10\rangle$ and $|11\rangle$. To describe a state $|\Psi\rangle$ we write $|\Psi\rangle = \alpha_1|00\rangle + \alpha_2|01\rangle + \alpha_3|10\rangle + \alpha_4|11\rangle$, where $\alpha_1, \alpha_2, \alpha_3$ and $\alpha_4$ are four $(2^2)$ complex parameters. This illustrates that the system state for a two qubit system doesn't have to be in a single basis state (e.g. $|10\rangle$) as a classical state would have to be (e.g. 10). Because each qubit can be in a superposition, the state $|\Psi\rangle$ can be in a superposition of several different basis states. Of course, it is possible that $|\Psi\rangle$ has all but one parameter equal to zero so that it with 100% certainty is in one of the basis states, but that is just one of infinitely many states that $|\Psi\rangle$ can be in.

In a three-qubit system the basis states are $|000\rangle$, $|001\rangle$, $|010\rangle$ $|011\rangle$, $|100\rangle$, $|101\rangle$, $|110\rangle$, $|111\rangle$. To describe a state $|\Psi\rangle$ that consist of three qubits we use eight $(2^3)$ parameter and write $|\Psi\rangle = \alpha_1|000\rangle + \alpha_2|001\rangle + \alpha_3|010\rangle + \alpha_4|011\rangle + \alpha_5|100\rangle + \alpha_6|101\rangle + \alpha_7|110\rangle + \alpha_8|111\rangle$.

We see that we need $2^n$ complex parameters $(\alpha_1, \ldots, \alpha_{2^n})$ to describe a system with $n$ qubits. This exponential growth $(2^n)$ of the number of parameters required to describe a system state quickly becomes a problem when building a simulator for a quantum computer using a classical computer, the available memory (RAM) runs out even for low number of qubits! See chapter 2.1.3 for more details.

It is the state of the quantum system that is manipulated when a program runs in a quantum computer. During execution of the program quantum operations are carried out that change the state. Finally, the output from the program is the system state after all operations have been completed.

### 1.1.3 Pure states and mixed states

To describe the well-defined system states shown above (e.g. $|\Psi\rangle = \alpha_1|00\rangle + \alpha_2|01\rangle + \alpha_3|10\rangle + \alpha_4|11\rangle$) it is sufficient to work with the $2^n$ parameters $\alpha_1 \ldots \alpha_{2^n}$.

We call a state that is completely described by $\alpha_1 \dots \alpha_{2^n}$ a *pure state*.

If a system state represents a statistical mix of different states we call that a *mixed state*. A mixed state is described using a *density matrix*, see chapter 2.1.2. To make this a bit more concrete, let us assume that one system state represents electrons polarized in one direction and other system states represent electrons polarized in a different direction. If our quantum system contains a statistical mix of these two type of particles then that would be a mixed state.

A pure state is a special case of a mixed state and its description can be simplified to be a *state vector*, see chapter 2.1.1. But a pure state can also always be described using a density matrix.

### 1.1.4 Entanglement

Another phenomenon that may affect a quantum state is *entanglement*. When qubits are entangled the state of the qubits are dependent on each other. The dependency does not mean that the qubits must have the same value, but you could say that only certain combinations of their values are available.

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

Consider the system state shown in Equation 2, it is a system state that can only be in one of the states $|00\rangle$ or $|11\rangle$. If you measure the system state then the result may be any of those two states. But also note that if you only measure the first qubit and find that it is $|0\rangle$ then you immediately know that if you measure the second qubit it will also be $|0\rangle$. We say that the qubits are *entangled*.

In addition to a strong correlation between qubits entanglement also has the property of being non-local. If we have a system containing two qubits as described in Equation 2 and separate the individual qubits by any geographical distance, then they are still entangled and measurement of any of the qubits instantaneously determine the state of the other qubit.

When we discussed qubits in section 1.1.1 we said that measurements in one basis disturb measurements in another basis. Therefore it could be expected that entanglement only applies in a single basis. But let us see what happens if we define $|0\rangle = \frac{|+\rangle+|-\rangle}{\sqrt{2}}$ and $|1\rangle = \frac{|+\rangle-|-\rangle}{\sqrt{2}}$ and insert that in Equation 2.

$$\begin{aligned}
|\Psi\rangle &= \frac{|00\rangle + |11\rangle}{\sqrt{2}} \\
&= \frac{\frac{1}{2}\Big((|+\rangle + |-\rangle)(|+\rangle + |-\rangle) + (|+\rangle - |-\rangle)(|+\rangle - |-\rangle)\Big)}{\sqrt{2}} \\
&= \frac{|++\rangle + |--\rangle}{\sqrt{2}}
\end{aligned}$$

We see from the derivation above that the entanglement translate between the computational basis and the $|+\rangle/|-\rangle$ basis.

Entanglement is a complex topic that is an important part of quantum mechanics, but for more on this topic we refer the reader to [17] and [14]. To round off let us just mention that the state in Equation 2 is called a Bell state after John Bell who analyzed properties of quantum mechanics, among them entanglement. The Nobel Prize in physics in 2022 was awarded to Alain Aspect, John Clauser, and Anton Zeilinger "for experiments with entangled photons, establishing the violation of Bell inequalities and pioneering quantum information science" [18].

### 1.1.5 Circuit model of quantum computing

*Quantum gates* or just *gates* are operations that manipulate the system state, a gate typically changes the value of a qubit. But the result of a gate may also depend on an additional qubit to make a logical decision, the additional qubit is called a *control qubit*, see section 2.2.1. A typical gate changes one or two qubits and uses zero or one control qubit.

When many quantum gates are used together we call the assembly a *quantum circuit*. A quantum circuit is the program of a quantum computer.

Potentially you can also view smaller parts of a program as individual quantum circuits, maybe they are well known or reusable circuits. A great example of this is Quantum Fourier Transforms (QFT) which will be discussed in chapter 4.4.

## 1.2 Physical and logical qubits

When building a quantum computer you construct physical qubits, but that is not necessarily what you are interested in when you ask how large a quantum computer is. A physical qubit is very small and very delicate, disturbance from the environment or normal quantum events can easily alter its state.

To compensate for the fragility of a physical qubit we use *error correction*. Error correction uses several physical qubits to create a more reliable qubit, we call these *logical qubits*.

Logical qubits is something we can use in real-world application. The quantum circuits described in previous sections are not built using physical qubits, instead they use logical qubits that are assumed to be stable. When we say that a quantum circuit use three qubits during its execution that is not necessarily three physical qubits, it is three logical qubits which may translate to many more physical qubits.

Today's quantum computers do not provide high-quality qubits and they have a high error rate [19]. To limit the impact of errors we include error correction in quantum circuits, this gives us stable circuits with high confidence in the result. The disadvantage is that error correction logic requires many qubits, so the number of qubits available for the circuit logic decreases. At the time of this writing (September 2024) the largest quantum computer processor

has 1121 qubits [11], but when you implement error correction the number of logical qubits is much lower then 1121. The decrease in number of usable qubits decreases the complexity of the problems the we can solve using the quantum computer.

# 2 Simulation of Quantum Circuits

The previous chapter gave a general introduction to the concept of quantum computing. This chapter discuss those concepts in the context of simulating a quantum computer using a classical computer. The discussion focuses on things that are common to all simulators, specific details about the design of the simulator that was implemented as part of this thesis are described in Chapter 3.

What is the toolbox we have available if we wish to simulate quantum mechanical systems using a classical computer? The general answer is *mathematics* and more specifically *linear algebra*. Maybe you did not think about it, but already in Figure 1 we used mathematics and linear algebra to represent the system state.

In Figure 1 the system state was shown as a vector, but we have also discussed probability and mentioned complex numbers. In addition to the above, this chapter will also make extensive use of matrices (with complex numbers as elements) and multiplication between matrices.

## 2.1 System State Representation

The system state is a complete description of the state for all qubits in the quantum system. The system state does not focus on an individual qubit, only on the state from the combination of all qubits.

Below are two common ways to represent the quantum state during simulation of quantum circuits.

### 2.1.1 State vector

The simplest way to describe the state of a quantum system is as a *state vector* that has $2^n$ elements to describe $n$ qubits, see chapter 1.1.2. Each of the elements are complex numbers

To make it more concrete consider Figure 4 below. It shows a state vector on the left and to the right are the states that each component of the state vector refers to. When using a state vector the probability to measure a state in the computational basis is the squared magnitude of the parameter for that state, in the figure below the probability for the basis state $|001\rangle$ is $|\frac{i}{2}|^2 = \frac{1}{4}$.

$$\begin{pmatrix} 1/2 \\ i/2 \\ 0 \\ 0 \\ \frac{1}{2\sqrt{2}} + \frac{i}{2\sqrt{2}} \\ 0 \\ -\frac{1}{2\sqrt{2}} + \frac{i}{2\sqrt{2}} \\ 0 \end{pmatrix} \begin{matrix} |000\rangle \\ |001\rangle \\ |010\rangle \\ |011\rangle \\ |100\rangle \\ |101\rangle \\ |110\rangle \\ |111\rangle \end{matrix}$$

Figure 4: A state vector representing a quantum system state with three qubits and $2^3 = 8$ possible states.

When a quantum circuit executes the gates operate on the state vector, thereby evolving the system state. Figure 1 illustrate how the input state on the left is evolved to the output state on the right by performing gate operations on the state.

A mathematical property of the state vector is that if the components are represented as $e_i$ then $\sum |e_i|^2 = 1$. Compare to how we defined the probability for a state when we looked at a single qubit in Figure 3.

A limitation of state vectors are that they only can represent pure states, if your state is a mixed state then it is not enough to use a state vector. However, if we avoid non-unitary operators (see section 2.2), quantum noise (see section 2.5) and don't do partial traces (see section 2.1.5) in the simulation it is fine to use state vectors.

### 2.1.2 Density Matrix

A more general way to represent the state of a quantum system is to use a *density matrix*, using a density matrix allows us to represent both pure states and mixed states. The size of a density matrix for $n$ qubits is $2^n \cdot 2^n = 2^{2n}$ elements.

$$2^n \text{ rows} \underbrace{\begin{pmatrix} p_{1,1} & p_{2,1} & \cdots & p_{2^n,1} \\ p_{1,2} & p_{2,2} & \cdots & p_{2^n,2} \\ \cdots & \cdots & \cdots & \cdots \\ p_{1,2^n} & p_{2,2^n} & \cdots & p_{2^n,2^n} \end{pmatrix}}_{2^n \text{ columns}}$$

Figure 5: Density matrix for a quantum state with $n$ qubits.

If we have a pure state $|\Psi\rangle$ we can create the density matrix $\rho$ through $\rho = |\Psi\rangle\langle\Psi|$. The density matrix for the vector state given in Figure 4 is shown below, it is much larger so it requires more memory to store and operations on it will take much longer to perform. It is therefore beneficial for a simulator if all system states can be limited to state vectors.

$$\begin{pmatrix}
\frac{1}{4} & -\frac{i}{4} & 0 & 0 & \frac{1}{4\sqrt{2}} - \frac{i}{4\sqrt{2}} & 0 & -\frac{1}{4\sqrt{2}} - \frac{i}{4\sqrt{2}} & 0 \\
\frac{i}{4} & \frac{1}{4} & 0 & 0 & \frac{i}{4\sqrt{2}} + \frac{1}{4\sqrt{2}} & 0 & -\frac{i}{4\sqrt{2}} + \frac{1}{4\sqrt{2}} & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{1}{4\sqrt{2}} + \frac{i}{4\sqrt{2}} & -\frac{i}{4\sqrt{2}} - \frac{1}{4\sqrt{2}} & 0 & 0 & \frac{1}{8} & 0 & -\frac{1}{8} & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-\frac{1}{4\sqrt{2}} + \frac{i}{4\sqrt{2}} & \frac{i}{4\sqrt{2}} - \frac{1}{4\sqrt{2}} & 0 & 0 & -\frac{1}{8} & 0 & \frac{1}{8} & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}$$

Figure 6: A density matrix for a quantum system state with three qubits and $2^{n\,2} = 2^{2n} = 2^{2 \cdot 3} = 2^6 = 64$ matrix elements.

### 2.1.3 Working with multi-qubit systems

Now let us look at how much memory is required to work with these state representations.

The number of parameters required to describe a multi-qubit system as a state vector is $2^n$. If we have 32 qubits we need $2^{32} = 4\ 294\ 967\ 296$ matrix elements to describe the state vector. Each matrix element is a complex floating point number and assuming we use 32-bit (4 bytes) floating point numbers this will require $2^{32} \times 2 \times 4$ bytes = 34 359 738 368 bytes = 32 GB [2] to store the state vector in memory.

Today (2024) a highend personal computer has 32 GB of RAM, so this small system with 32 qubits will allocate all the available memory of the computer just to store the state vector. But to just hold the state is not enough, for the system to be useful we must also perform operations on it and that will require even more memory. When we wish to apply a quantum gate to a state with $2^{32}$ qubits we need a matrix with $2^{32} \times 2^{32} = 2^{64}$ elements, which is a number so large that it cannot possibly be stored in memory on a computer.

The exponential growth of memory required to work on multi-qubit systems limits the size of the quantum systems that can be simulated on a classical computer.

### 2.1.4 Optimizing storage of quantum state

When you build a simulator for a quantum computer you can use smart techniques to optimize memory usage. One such technique is to realize that in many cases the vectors and matrices used during a simulation often have a lot of zeros in them, and then there are optimized ways to work with sparse vectors and matrices.

Example: The version of Shor's Algorithm that is implemented in this thesis project (see chapter 4.5) uses 10 qubits which means that the state vector has

[2] $2^{32}$ matrix elements $\times 2$ floating points per complex number (real and imaginary parts) $\times 4$ bytes per floating point.

1024 elements, but in practise when running the algorithm there are only 16 elements that have a non-zero value and the other 1008 elements are zero. When we apply a quantum gate on that state vector it has a size of $1024 \times 1024 = 1\ 048\ 576$, but only 1024 elements that are non-zero and 1 047 552 are zero.

### 2.1.5 Measurements

When you measure a system state it collapses to one of the possible states of the system, the system is no longer in a superposition and repeated measurements (using the same basis) will give the same answer.

In the physical world you can measure e.g. position, energy or momentum and, within the limitations given by the uncertainty principle, repeated measurements can yield the same result. In the world of quantum computing we measure a qubit using the computational basis, so the result of measurement of a single qubit is either $|0\rangle$ or $|1\rangle$. As discussed in chapter 1.1.1 results may also be interpreted using other basis, e.g. the $|+\rangle/|-\rangle$ basis, when this is required the system state is transformed before a measurement is made in the computational basis. Example: if we have a system state that we wish to measure in the $|+\rangle/|-\rangle$ basis we can apply a Hadamard gate and then perform the measurement in the computational basis, the result can then be interpreted using $|+\rangle/|-\rangle$ basis.

When a state vector is used to describe the system state each vector element gives a probability for the corresponing state, $p_i = |\alpha_i|^2$. When you measure the state during a simulation of a quantum circuit, you can create a random number $[0, 1]$ and check which of the state probability intervals it falls into. Example: If we have $|\Psi\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ then we can define the internal $[0, 0.5)$ to mean $|0\rangle$ and $(0.5, 1]$ to mean $|1\rangle$. If our random number is 0.35 we say that the result of the measurement was $|0\rangle$.

A similar technique, although a bit more complex, can be used if the system state is described by a density matrix.

The above description of measurement assumes that the full state is measured. In many cases you wish to measure only a subset of qubits. This leaves the other qubits untouched, allowing them to either be measured at a later point in time or to be completely ignored. Ignoring the results from some qubits is not as strange as it sound, many quantum algorithms use extra qubits that are necessary for the execeution of the algorithm but they do not carry information about the result of the algorithm.

To measure only a subset of the system state you perform a partial trace that creates a reduced density matrix that only holds information about the qubits you are interested in. Assume the density matrix $\rho_{AB}$ describes a system that consist of two part ($A$ and $B$), when a partial trace is performed over system $B$ the results is a reduced density matrix $\rho_A$ that only contains information about system $A$. We say that we have "traced out" information about system $B$, so only information about system $A$ remains. The reduced density matrix is smaller in size than the original density matrix, because it hold information

about fewer qubits. When a measurement is performed on the reduced density matrix, the result is a measurement of only system $A$.

Qurigo does not implement support for partial trace on density matrices.

## 2.2 Quantum Gates

In section 1.1.5 we described the role of a quantum gate, how it acts on a quantum state to change it from one state to another. We now define a quantum gate in a simulator to be a matrix operating on a state vector or density matrix. To simulate that a quantum gate operates on a state we perform a multiplication between a matrix (the gate) and a vector/matrix (state vector/density matrix).

The matrices used to represent quantum gates are *unitary matrices*, satisfying $UU^\dagger = U^\dagger U = I$ and hence also $U^\dagger = U^{-1}$.

Later in this chapter we will introduce many different types of gates; their purpose and how they are represented as matrices when building a simulator for a quantum computer.

Figure 7 below shows how a CNOT gate (see chapter 2.2.2) transforms $|10\rangle \rightarrow |11\rangle$ and $|11\rangle \rightarrow |10\rangle$, but $|00\rangle$ and $|01\rangle$ remain in their inital state. It only affects $|10\rangle$ and $|11\rangle$, leaving the other states unaffected.

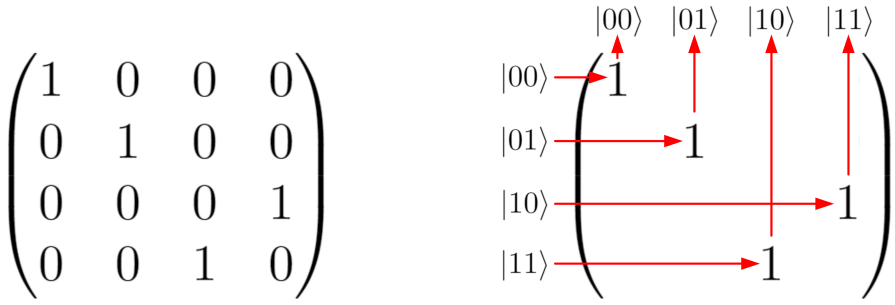



Figure 7: Visualisation of transformation performed by a quantum gate.

The matrix elements of a gate are not limited to integers or even to real numbers, an element can be an arbitrary complex number; $a + b\mathrm{i}$ with $a, b \in \mathbb{R}$.

It is also possible to transform one state to a superposition, this can be done using the Hadamard-gate where the following transformations are performed: $|0\rangle \rightarrow \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and $|1\rangle \rightarrow \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$.

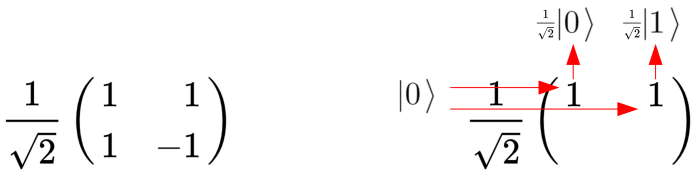



Figure 8: Visualisation of transformation performed by the Hadamard gate.

When building the simulator one challenge is to ensure that systems of arbitrary size (up to constraints given by memory as discussed in section 2.1.3) can be simulated. Even if a gate only operates on a single qubit the matrix that represents the gate must still be a full $2^n \times 2^n$ matrix, but most parts of that matrix will look like an identity matrix. This area is one where an efficient simulator can provide optimizations that keep memory consumption as low as possible.

### 2.2.1 Controlled Gates

A common type of gate is the *controlled gate*, it is when the application of a operation on a qubit is conditioned on the value of another qubit. The qubit whose value is used to determine if the operation should be applied or not is called the *control qubit*. If the control qubit has value $|1\rangle$ the operation is applied, and if the control qubit has value $|0\rangle$ the operation is not applied. If the control qubit is in a superposition of $|0\rangle$ and $|1\rangle$, then that superposition will also be reflected in the resulting system state after applying the operation.

The most well-known example of a controlled gate is the CNOT gate (see chapter 2.2.2) shown in Figure 7, where the prefix C means Controlled and the follows the name of the gate that is controlled (in this case the NOT gate, a.k.a. the X gate).

There are also gates with more than one control qubit. The Toffoli gate is an example of a gate with two control qubits, where both control qubits needs to be in state $|1\rangle$ in order for the operation to be applied.

### 2.2.2 Description of some quantum gates

This section describes selected quantum gates, focus is on gates that have been used to implement Shor's algorithm as described in chapter 4.5.

The matrix representation of the gates are presented in the way most common in litterature and on Wikipedia, this allows the reader to refer to other resources to learn more without causing confusion. However, the representation used in this chapter numbers qubits using the Big Endian notation and the Qurigo simulator implemented as part of this thesis uses Little Endian notation. This means that the matrix representation for two- and three-qubit gates are different between the matrices below and what is used in the source code for Qurigo. See section 3.2 for more details.

**Pauli Gates ($X$, $Y$ and $Z$)** The most well known quantum gates are the Pauli gates a.k.a. Pauli matrices, they are typically introduced to students during their first course in Quantum Mechanics with the names $\sigma_x$, $\sigma_y$ and $\sigma_z$. The $X$, $Y$ and $Z$ gates are single-qubit gates.

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \quad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

**Hadamard Gate ($H$)** The Hadamard gate is a single qubit gate that creates a superposition for a qubit, it can also be seen as a transformation between the computational basis and the $|+\rangle/|-\rangle$ basis. The transformations are $|0\rangle \longmapsto \frac{|0\rangle+|1\rangle}{\sqrt{2}} = |+\rangle$ and $|1\rangle \longmapsto \frac{|0\rangle-|1\rangle}{\sqrt{2}} = |-\rangle$. Creation of superposition is a key component of many quantum algorithms.

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

**$T$ Gate** The $T$ gate is a special version of the phase gate using $\theta = \frac{\pi}{4}$.

$$T = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{4}} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & \frac{1}{\sqrt{2}} + \frac{i}{\sqrt{2}} \end{pmatrix}$$

**$CNOT$ Gate** The $CNOT$ gate (a.k.a. $CX$ gate) is a controlled version of the $X$ ($NOT$) gate, and it acts on two qubits[3] .

$$\text{CNOT} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

**$ECR$ gate** The Echo Cross Response ($ECR$) gate is a two-qubit gate that performs a $CX$ operation and a Hadamard operation in a single gate.

$$ECR = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & -1 & 0 \\ 1 & 0 & 0 & -1 \end{pmatrix}$$

**$CSWAP$ Gate** The $CSWAP$ gate (a.k.a. Fredkin gate) is a three-qubit gate[4] that represents a contolled version of the $SWAP$ gate. It only performs the $SWAP$ operation if the control qubit is $|1\rangle$.

[3] A two-qubit gate means that the matrix will have $2^2 \times 2^2$ elements.
[4] A three-qubit gate means that the matrix will have $2^3 \times 2^3$ elements.

$$\text{CSWAP} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

**Toffoli gate ($CCX$)** The Toffoli gate is a three-qubit gate that has two control qubits and if they both are $|1\rangle$ then an $X$ gate is applied.

$$\text{CCX} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

## 2.3 Universal Quantum Gates

As shown in [14], it is possible to define a set of universal quantum gates that can be used to create all other quantum gates, to any given precision. In [14] they choose to include the H, T and CNOT gates in their universal set, but other universal sets can also be constructed.

In theory it would be enough to just support these three gates in a simulator and when someone wishes to write a program using the simulator he would deconstruct any other gates into these universal gates. However, that would be both cumbersome and inefficient for the researcher because he would have to know all deconstructions and it would also lead to programs that are much longer than necessary.

Instead the simulator implements support for a larger number of different quantum gates, allowing a researcher to create quantum circuits with high-level gates. When the simulator executes it takes on the responsibility of decomposing quantum gates to universal quantum gates. Example, when the simulator executes a circuit with a CZ gate it executes the sequence H-CNOT-H (which are universal gates).

An important note to make here is that the gates supported natively by a real quantum computer is most likely not the H-T-CNOT set of universal gates. So if a simulator is supposed to simulate a specific quantum computer in a relastic way then the set of universal gates will have to be replaced with the gates that the physical qubits support.

### 2.3.1 Gate decomposition vs matrix operation

You may ask yourself why not just implement the CZ gate as a matrix operation and be done with it? The answer is that then the simulator would not be scaling the same way as a real quantum computer would, because the number of gates needed to implement a circuit would decrease significantly.

If the purpose of the simulator is to imitate a real quantum computer by adding noise it can be of great importance that the number of gates used to implement a circuit it close to the number of gates required when executing the same circuit on the real quantum computer.

If on the other hand the purpose is to validate programs on a logical qubit level then all gates could be implemented as direct matrix operations, and skip the decomposition into universal gates.

### 2.3.2 Universal gates vs native gates in a quantum computer

When a real quantum computer is built, i.e. a physical computer, the native gates will have to include gates that together form a set of universal gates. Thus they are in no way limited to just support the gates that form the universal set.

The *IBM Eagle r3* quantum processor has support for these gates: $ECR$, $ID$, $RZ$, $SX$, $X$. To form a universal set it is enough to include $ECR$ and one of $RZ/SX/X$, but by supporting more native gates than is necessary it becomes easier to write programs for the quantum processor.

## 2.4 Quantum Circuits

In chapter 1.1 we introduced the concept of a quantum circuit to be equivalent to a program running on a quantum computer. A better definition would be that a quantum circuit is a sequence of quantum gates that is applied to a quantum state. This is illustrated in Figure 9.

Please note that a conceptual notation is used in this section to describe the logical concept of a quantum circuit and its relation to the matrices of the gates, the purpose is to explain the concept of quantum circuit optimizations. See section 2.4.2 for a mathematical description.

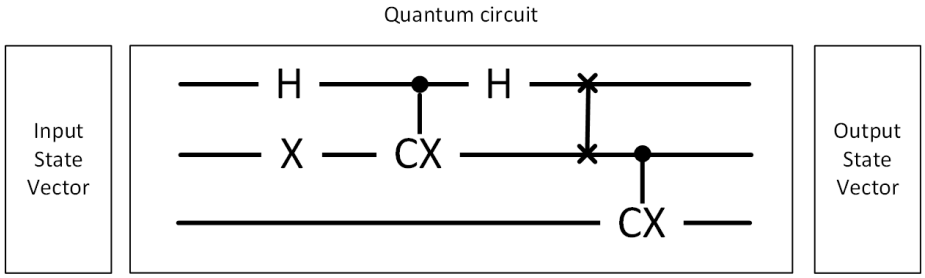



Figure 9: Logical representation of a Quantum Circuit shown as gates.

Now let us combine this definition with the discussion in chapter 2.2 where we have defined a quantum state as a state vector (or density matrix) and a quantum gate as a matrix that transform the state vector. The quantum circuit takes a state vector as input and its output is also a state vector, each gate in the quantum circuit is a matrix that operate on the state vector. This is shown in Figure 10.

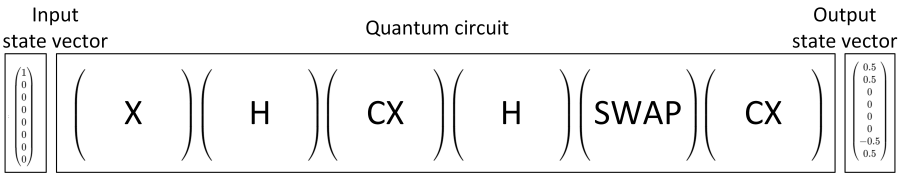



Figure 10: Logical representation of a Quantum Circuit as a sequence of matrices, see section 2.4.2 for a formal mathematical description.

### 2.4.1 Optimizing Execution of a Quantum Circuit

The purpose of a simulator is to be true to the system it simulates, but it should also aim to make an efficient simulation. So let us discuss how we can minimize the number of matrix multiplications that is necessary to execute a quantum circuit.

This is important because the size of each matrix is $2^n \times 2^n$, so for 16 qubits the matrix stores 65 536 $\times$ 65 536 complex numbers which takes up approximately 69 GB of memory on the classical computer used to simulate the quantum circuit.

It can easily be understood that when two gates act in parallel on different qubits the individual matrices can be combined into a single matrix, as shown in Figure 11. Each gate operates independently from the other and the operator on different parts of the state vector, so they can be executed in parallel.

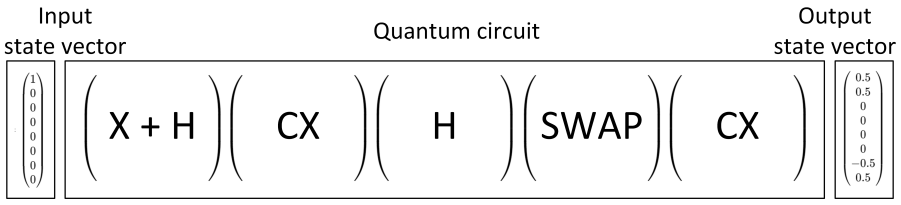



Figure 11: Quantum Circuit with two gates (X + H) combined into one matrix.

Less obvious is that all gates in a quantum circuit can be combined in a single matrix as shown in Figure 12, this is discussed in [16] and is also a technique used by Qiskit. The neat thing is that the size of the combined matrix is the same as each of the individual gate matrices, so just because it has all the information of the individual gate matrices the size of the matrix does not grow. However,

the number of matrix elements that are non-zero will grow so if we consider the techniques discussed in chapter 2.1.4 we will have still have more information to store. But the benefit of not calculating each of the individual matrices each time far exceeds the downside of storing more non-zero elements.

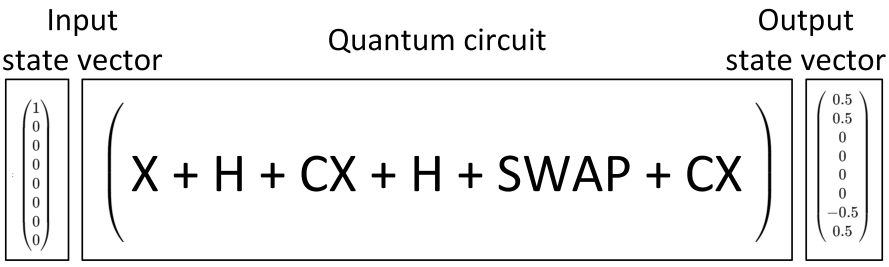



Figure 12: Quantum Circuit with all gates combined into a single matrix.

When implementing a simulator for a quantum computer this is a useful technique to speed up execution when a quantum circuit (or parts of a quantum circuit) is used several times. The work to prepare the combined matrix is the same as applying each gates separately, because the combined matrix is created by multiplying all of the gate matrices, but when the combined matrix has been prepared it can be saved. Using the saved matrix the execution of the quantum circuit becomes a single operation (i.e., a single matrix multiplication).

### 2.4.2 Mathematical description of optimization

If the quantum circuit only has a single $X$ gate the mathematical description of the transformation performed by the quantum circuit would be $|\Psi_{output}\rangle = U_X|\Psi_{input}\rangle$. If it first applied $X$ and then $H$ we would get $|\Psi_{output}\rangle = U_H U_X|\Psi_{input}\rangle$, which could be rewritten as $|\Psi_{output}\rangle = U_{XH}|\Psi_{input}\rangle$ where $U_{XH} = U_H U_X$.

Now, let us assume we have the circuit shown in Figure 9. We can describe as $|\Psi_{output}\rangle = U_{QC}|\Psi_{input}\rangle$, where $U_{QC}$ is a unitary matrix representing the optimized quanatum circuit shown in Figure 12. To construct $U_{QC}$ we use the same technique as in the previous paragraph and the result is show below in Equation 3.

$$|\Psi_{output}\rangle = U_{QC}|\Psi_{input}\rangle = U_{CX}U_{SWAP}U_H U_{CX}U_H U_X|\Psi_{input}\rangle \qquad (3)$$

Note that the first gate ($X$) in the quantum circuit is located to the right next to the input state. The last gate ($CX$) in the circuit is to the left and it is applied to the result of all other gate operations.

Using this information we can prove the claim in section 2.4.1 that the matrices representating the gates in a quantum circuit can be combined into a single matrix. To prove this we use the associative property of matrix multiplication.

$$
\begin{aligned}
& U_n \dots U_3 U_2 U_1 |\Psi> && (4)\\
=& U_n \dots (U_3(U_2(U_1|\Psi>))) && (5)\\
=& U_n \dots (U_3((U_2 U_1)|\Psi>)) && (6)\\
=& U_n \dots ((U_3 U_2 U_1)|\Psi>) && (7)\\
& \dots && (8)\\
=& (U_n \dots U_3 U_2 U_1)|\Psi> && (9)\\
=& U_{QC}|\Psi> && (10)
\end{aligned}
$$

## 2.5 Noise in Simulations

Consider a classical computer where the bits are stored in RAM memory, you do not expect this information to be lost if you leave the computer on for a week. You expect to be able to retrieve the information just as you wrote it to memory. In a quantum computer the state deteriorates much faster due to interaction with the external environment, the time it takes to loose state is called decoherence time. The IBM Eagle r3 processor discussed in section 2.5.1 has a median decoherence time of $119\mu s$. If a simulator wishes to be true to how a real quantum computer behaves, then the simulator has to include this instability in the data model.

A simple way to introduce noise would be to add noise each time a gate is applied to a state. It could be a fixed probability vector that is applied every time regardless of which gate is being executed, it can also be a more complex model where the probability of introducing an error depends on the type of gate being executed. Using this model immediately makes it important if a gate is native or if it is derived from other gates. Executing a native gate (e.g. H, T or CNOT) will have a low probability of introducing an error, but if you execute the Toffoli (CCX) gate which is created using 21 H/T/CNOT gates the risk of an error is $21\times$ larger than for a native gate.

Assume that the probability of success for a gate is $p$ and you execute $n$ gates, then the success rate of is $p^n$. If we apply this to the Toffoli gate mentioned above in the context of the an IBM Eagle r3 processor that has a median error rate of $6.8 \cdot 10^{-3}$ and $p = (1 - 6.8 \cdot 10^{-3}) = 0.9932$. For 21 native gates the overall success rate is $p_{\text{Toffoli}} = p^{21} = 0.9932^{21} = 0.8665$. So the risk of a failure when executing a Toffoli gate is over 13%!

A more advanced model for noise would be to look at the decay time for each individual qubit you simulate, e.g. by taking information from a real quantum computer and study how the stability varies between individual qubits. It is also possible that the noise is different depending on if the state is $|0\rangle$ or $|1\rangle$. Another possibility is that noise is correlated between qubits, i.e. the noise for one qubit is linked to the noise of another qubit.

If you also study the time required to execute an individual quantum gate you can calculate how long it takes to execute a quantum circuit, and then you can introduce appropriate noise during each gate operation.

The simulator built as part of this thesis does not include noise in its simulation.

### 2.5.1 Noise in IBM quantum computers

IBM make some of their real quantum computers available as cloud resources, i.e. a researcher can create programs and execute them remotely on one of IBM's quantum processors. Currently (September 2024) the most common quantum processor in IBM's cloud offering is the *Eagle r3*, which has 127 physical qubits, see `https://quantum.ibm.com/services/resources?system=ibm_kyiv` for more details. IBM Eagle was launched at the end of 2021 and has since then been succeeded by new generations called Osprey, Heron and Condor. Of these newer generations only Heron is part of IBM's cloud offering.

IBM publishes error rates for their public quantum computers, see Figure 13 for the error rates of the IBM Eagle and its 127 qubits when they execute the CZ gate. The qubit with the highest error rates has 0.1807 but the median over all qubits is 0.0068.

Another interesting thing to note in Figure 13 is that at the top they show that this quantum computer supports *OpenQASM 3*. This is the same programming language that I have chosen for my simulator, see chapter 3.6.

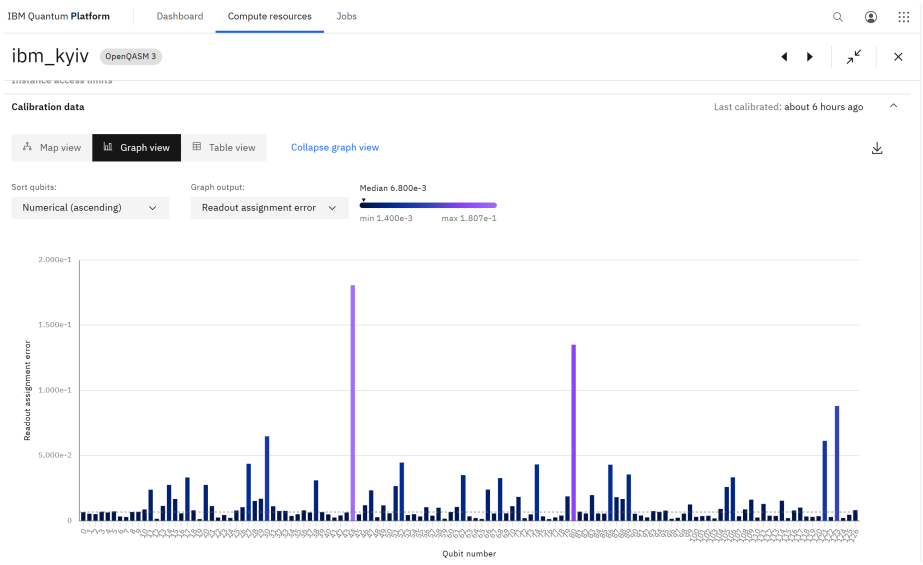



Figure 13: Readout errors for IBM physical qubits.

### 2.5.2 Error correction

When there is noise in program you want to perform steps to minimize the effect of the noise. In chapter 1.2 we introduced a *logical qubit* to represent a qubit

free of errors, built from *physical qubits* which have errors. Creation of a logical qubit relies on *error correction*.

To detect errors in classical computing we can use a parity bit, it can be used to detect if any single one of the involved bits have changed value. But it cannot detect if two bits change value, and it cannot recover the original value.

Using three-bit coding in classical computing it is possible to also recover from errors. Reference [14] discusses in detail how the three-bit coding can be generalized to quantum computing.

# 3 Simulator Design

Much of the work during this thesis has been put into building a quantum computer simulator that runs on a classical computer. The simulator has been named Qurigo and it will be referred to that henceforth. The purpose of creating a simulator is to put the theory presented in the previous chapters into practical work and see how hard it is to create a simulator. Qurigo has been published as open source on GitHub [15].

## 3.1 Modular design

A primary goal when designing Qurigo was to make it modular to allow easily swapping core building blocks without affecting other building blocks. This makes Qurigo a generic simulator that can be configured to behave differently or to run different simulations. The building blocks that can be swapped are:

- State representation: State vector or Density matrix, see section 3.1.1.
- Native instructions: H-T-CNOT or IBM Eagle r3, see section 3.1.2.

In addition to the above the initial plans included to support noise and compare results in noisy and noiseless simulations, but that feature was excluded due to time constraints.

To support easy swapping of building blocks the software design technique called dependency injection [6] was used.

### 3.1.1 State representation

Qurigo support both state vector and density matrix to represent the state of a quantum system. During the work to implement quantum algorithms (see chapter 4) focus was on using state vectors, but the density matrix representation was validated in unit tests where identical tests were performed using both state vector and densit matrix.

### 3.1.2 Instruction Sets and Native Gates

The Qurigo simulator defines an instruction set that consists of a number of gates that can be used to build a circuit. The supported gates are: $X$, $Y$, $Z$, $H$, $T$, $CNOT$ ($CX$), $SWAP$, $CSWAP$, Toffoli ($CCX$), $SX$, $RZ$, $S$, $ECR$, and $CR_k$.

The idea is that Qurigo always implements the same instruction set but uses different native gates to do it, in one instruction set $CNOT$ can be a native gates but in another instruction set $CNOT$ can be a derived gate. This allow for evaluation of the same quantum circuit using different instruction sets and compare the result in resource consumption and noise. However, as the work progressed the focus on comparison of resource consumption, performance and noise between different implementations was removed to better match the scope of a Bachelor thesis in Physics.

Qurigo defines two implementations of the instruction set:

- *H-T-CNOT*: Use $H$, $T$ and $CNOT$ as native gates and derive all other gates from these, an exception was made for $X$, $Y$ and $Z$ which also are considered native. This set of native gates is based on the discussion on universal quantum gates in chapter 2.3 .
- *IBM Eagle r3*: During the thesis work some quantum programs have been executed on real quantum computers from IBM [1]. It uses the $ECR$, $ID$, $RZ$, $SX$ and $X$ gates as native gates. In order to execute the same quantum circuit both on the real quantum computer and in the simulator an instructions set using these gates as native gates was developed.

## 3.2 Qubit ordering

The first decision in the design of Qurigo was in which order qubits would be numbered. At first glance there it may seem like a small question with perhaps an obvious answer, but it affects the core of the implementation and is not so easy to take.

Assume we have the state $|100\rangle$ and also assume that we reference qubits using a zero-based numbering, then we have 3 qubits and their indexes are 0, 1 and 2. But which qubit does the 1 represent? Is it qubit number zero or qubit number two?

When using Little Endian encoding we read the qubits from right to left, which means that in the example above qubit number two has the value 1. Big Endian encoding read from left to right, making qubit zero in the example above to have the value 1.

It may seem obvious to choose Little Endian encoding due to its alignment with binary numbers, but books such as [17] and [14] as well as Wikipedia pages discussing quantum computing typically assume Big Endian encoding. However, the quantum computing simulators from IBM (Qiskit) and Microsoft (Q#) both use Little Endian.

I have chosen to use Little Endian for my simulator. My reasons are that it makes the interpretation of a state aligned with the normal interpretation of binary numbers and also that it allows me to easily compare my results to those from Qiskit.

Using Little Endian the state $|100\rangle$ would be interpreted as $|100\rangle = \left[1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = 8\right] = |8\rangle$ .

## 3.3 Native and Derived Gates

A *native gate* is a gate that is defined as a direct mathematical operation, i.e. it is defined as a matrix multiplication on a state vector. A *derived gate* is defined by executing a sequence of two or more native gates to get the desired result. For a simulator the difference is that for a native gate it only has to perform one matrix multiplication, but for a derived gate there will be at least two matrix multiplications to execute the gate. As we will see below there are typically

much more than two matrix multiplications involved in a derived gate, so using native gates is very much preferred.

Let us look at two concrete examples. The SWAP gate swaps the value of two qubits and we can easily define this as a matrix multiplication, but if we want to simulate the behaviour of a quantum computer that cannot perform SWAP as a native operation in the hardware then this is not an accurate simulation. Let's instead assume that we have CNOT as a native gate, then we can define the SWAP gate as a sequence of three CNOT gates.

The Toffoli gate (a.k.a. CCX) can also be defined as a single matrix multiplication, but when using the *H-T-CNOT* instruction set we need to use a sequence of 15 gates to define the Toffoli gate. The end result is the same, but the amount of resources require to run the program will increase if 15 matrix multiplications are performed compared to a single matrix multiplication.

If the only reason for implementing derived gates as a sequence of native gates was that we want to consume more CPU time and more memory, then that would be a poor design choice because in a quantum computer the resource consumption does not scale the same way it does in a simulator. The actual value we get from implementing derived gates the way we do is:

- *Quantum circuit complexity*: If the simulator implements a gate as a single operation where a real quantum computer needs e.g. 15 gates to implement the same gate then the simulator would avoid the complexity of the quantum computer, which defeats the purpose of the simulator. However, if the only purpose is to validate a quantum algorithm then it makes sense to focus on the sequence of gates and not on how the are implemented.

- *Execution time*: Avoiding the complexity of gates will also make the simulator look more efficient than the quantum computer it intends to simulate. This would also limit the usefulness of the simulator to estimate the depth of the circuit and the execution time required on a real quantum computer (remember that one purpose of a simulator is to make a realistic simulation of a quantum computer in order to learn how to best use them).

- *Noise and error propagation*: As discussed in chapter 2.5 noise and errors occur naturally in a quantum computer. If a simulator aims at realistically simulate noise then it cannot take 15 gates and replace them with one gate, that would defeat the purpose of a simulator where noise can occur.

## 3.4 Gate matrices for arbitrary size of system state

A challenge during the implementation of Qurigo was to create gates that can be applied to system states of arbitrary size.

Let us assume that we have a system state that consists of eight qubits represented as state vector and that we want to apply the $X$ gate to qubit 3, we can illustrate this as $X_3|\Psi\rangle$. In chapter 2 we learned that a gate operation is multiplication between the state vector and the gate matrix. In this case our state vector has size $2^8$ so the gate matrix needs to have size $2^8 \times 2^8$, and the

$128 \times 128$ matrix $X_3$ is very different from the $2 \times 2$ matrix for the $X$ gate shown in chapter 2.2.2. The mathematical definition of the matrix we need to create in this case is $\mathbb{I} \otimes \mathbb{I} \otimes X \otimes \mathbb{I} \otimes \mathbb{I} \otimes \mathbb{I} \otimes \mathbb{I} \otimes \mathbb{I}$ where $\otimes$ is the tensor product applied between the eight $2 \times 2$ matrices that represent identity ($\mathbb{I}$) and $X$ gate.

From a mathematical standpoint it may not have been so complex to build these matrices, but it was challenging to control the loops and qubit numbering when they should support any number of qubits.

### 3.4.1 Matrices for controlled gates

Creating the gate matrices for controlled gates was another challenge. The mathematical definition of a controlled gate is shown in Equation 11. Here we have the system state $|\Psi\rangle = |\Phi_c\rangle|\Phi\rangle$ with $|\Phi_c\rangle$ representing the control qubit, and $U$ is the $2^n \times 2^n$ matrix that apply the gate.

$$|0\rangle|\Phi\rangle + |1\rangle U|\Phi\rangle \tag{11}$$

This definition assumes that the control qubit ($\Phi_c$) is the first qubit in the system state, but that is of course not true in the general case which makes the creation more complex.

An additional complexity is when we have two control qubits as in the Toffoli ($CCX$) and $CR_k$ gates. The Toffoli gate was solved by implementing it using 15 native $H/T/CNOT$ gates. The $CR_k$ gate was implemented as a native gate which required support for building the transformation matrix. To generalize this to support all cases became too complex, and therefore Qurigo only support $CR_k$ in the special case that was required for implementing Shor's algorithm.

## 3.5 Validation of the Simulator

When developing the simulator it is sometimes hard to know if your program is correct. One way of ensuring that the program is correct is by using unit testing, in Qurigo I have tested basic operations on single quantum gates which can be mathematically verified using a pen and paper. However, when the number of qubits grow above two and we apply multiple gate in the test quickly becomes hard to verify manually.

If the expected result is hard to calculate by hand then the alternative is to compare results with an authorative source, and in this case the authorative source is three different tools from IBM.

- **IBM Composer:** A browser based visual designer for quantum circuits, it also supports writing and testing OpenQASM 2.0 code. The limitations are that it only supports states with up to six qubits and that OpenQASM 2.0 has a more restrictive syntax. The tool available for free at `https://quantum.ibm.com/composer`.

- **IBM Eagle r3:** IBM makes some of their real quantum computers available on the Internet and using the Open Plan option this allow for up

to 10 minutes of free execution time per month; `https://docs.quantum.ibm.com/guides/instances#open-plan`, see also section 2.5.1 for details about IBM Eagle. I have taken advantage of this program and executed programs on their hardware and then compared the resulting state vectors with those produced by the Qurigo simulator. Execution was done using Qiskit, as describe below.

- **Qiskit:** Qiskit is a quantum programming software package developed by IBM, targeting researchers who want to use IBM's real quantum computers. It includes options to either run quantum circuits in a local simulator or to send them to an IBM quantum computer.

Circuits developed using Qiskit can be executed on both a real quantum computer and in Qiskits simulator running locally on your computer. In this capacity Qiskit has been very useful to validate the behaviour of Qurigo, it was used both for verification of individual gates for multi-qubit system states and for comparing the result of the QFT and phase estimation algorithms.

Qiskit was used both to extend the unit tests and to write small programs (representing quantum circuits) whose output was compared to the results in Qurigo.

When making validations using the IBM Eagle quantum processor the program was written using the Qiskit language which was then transpiled to OpenQASM using the Qiskit toolkit. To perform the validation of Qurigo I copied the OpenQASM that was executed on the IBM Eagle quantum processor and then ran the same code in Qurigo. Minor adjustmensts were necessary due to that IBM Eagle use OpenQASM 2.0 and Qurigo use OpenQASM 3.0.

## 3.6 Programming Language

The programming language chosen to represent quantum circuits in Qurigo is OpenQASM 3 (Open Quantum ASseMbly language) [2] [4], it was developed by IBM and they provide a reference implementation in their quantum computing platform called Qiskit.

OpenQASM was chosen due to its simple structure which makes it easy to implement a language interpreter. In general OpenQASM uses the structure `command paramater1, parameter2;` which makes it straight-forward to parse from left to right. In addition to OpenQASM I evaluated using the Python-based syntax of Qiskit (from IBM) and Q# (from Microsoft), but they were rejected due to their more complex structure.

Only a small subset of the capabilities in the full OpenQASM specification has been implemented, but these capabilities are enough to implement a quantum algorithm such as phase estimation as part of Shor's algorithm. Supported features are shown below.

| Feature | Example |
|---|---|
| Declare qubits | `qubit[10] q;` to define a state of 10 qubits, numbered 0...9. |
| Quantum gates | `H, T, CNOT, X, Y, Z, SWAP,` $CR_k$`, CSWAP, ...` |
| Programming logic | `if, while` |
| Declare and manipulate variables | `int count = 0; count = count + 1` |
| Simple expressions | `parameter1 >= 8` |
| Declare and call functions | `def qft6i(int input)` to define and `qft6i(6);` to call. |
| Comments | `# this is a comment` |

### 3.6.1 Example: Phase Estimation for Shor's Algorithm

Let us look at an example of what a program written in OpenQASM might look like, the program below in Listing 1 is used in the implementation of Shor's algorithm (described in section 4.5) and it performs a phase estimation.

Row 1 define that this as an OpenQASM program.

Row 5 declare a quantum state `q` with 8 qubits, numbered 0 to 7.

There are two subroutines in this program, they are reusable components that can be called multiple times within the scope of the program. The first subroutine is `qft4i` which is defined on rows 7-24 and is called once on row 52. The second subroutine is `cswapHelper` (defined on rows 26-36) and it is called four times on rows 46-49. There are no parameters for `qft4i`, but `cswapHelper` declares two parameters (`control` and `repeat`) that are used to control the programming logic of the `cswapHelper` subroutine.

Looking at the programming logic that define the `qft4i` subroutine we first find the Hadamard gate on row 8. `h q[3]` means that we apply the Hadamard gate to the $4^{\text{th}}$ qubit, remember from section 3.2 that we use zero-based numbering for the qubit so `q[3]` becomes the $4^{\text{th}}$ qubit. The Hadamard gate is a single-qubit gate, so it only require one parameter.

Next we have three controlled rotation gates (`crk`) on row 9-11. When applying this gate we need to provide three parameters; the control qubits, the qubit on which to apply to rotation and the $k$-value that determines the angle of the rotation (see section 4.3.1). `crk q[0], q[3], 4;` means apply the rotation $\theta = \frac{2\pi}{2^4}$ to the $4^{\text{th}}$ qubit (`q[3]`) if the $1^{\text{st}}$ qubit (`q[0]`) is $|1\rangle$.

The last gate used in the `qft4i` subroutine is the $SWAP$ gate. It switch the values of two qubits with eachother, so we require two parameters when using it.

The `cswapHelper` subroutine takes two parameters. The first parameter, `control`, is used on lines 30-32 where it dynamically choose which qubit that is used as control qubit. If `control` has the value 1 then the statement `cswap q[control], q[4], q[5]` means that the $5^{\text{th}}$ and $6^{\text{th}}$ qubits are swapped if the $2^{\text{nd}}$ qubit is $|1\rangle$.

There is also a variable defined in `cswapHelper` on row 27, `index`. It gets an initial value of 0 and it is then used in the condition for the `while` statement on row 29. On row 34 the `index` variable is increased by one.

Finally, the actual program that is executed starts at row 39 and ends on row 52. It apply an $X$ gate and four $H$ gates before it calls the `cswapHelper`

subroutine and then the `qft4i` subroutine.

```
 1:  OPENQASM 3;
 2:
 3:  # 0, 1, 2, 3 : control registers
 4:  # 4, 5, 6, 7 : target registers
 5:  qubit[8] q;
 6:
 7:  def qft4i() {
 8:      h q[3];
 9:      crk q[3], q[0], 4;
10:      crk q[3], q[1], 3;
11:      crk q[3], q[2], 2;
12:
13:      h q[2];
14:      crk q[2], q[0], 3;
15:      crk q[2], q[1], 2;
16:
17:      h q[1];
18:      crk q[1], q[0], 2;
19:
20:      h q[0];
21:
22:      swap q[0], q[3];
23:      swap q[1], q[2];
24:  }
25:
26:  def cswapHelper(int control, int repeat) {
27:      int index = 0;
28:
29:      while(index < repeat) {
30:          cswap q[control], q[4], q[5];
31:          cswap q[control], q[5], q[6];
32:          cswap q[control], q[6], q[7];
33:
34:          index = index + 1;
35:      }
36:  })
37:
38:  # Psi_prep
39:  x q[4];
40:
41:  h q[0];
42:  h q[1];
43:  h q[2];
44:  h q[3];
45:
46:  cswapHelper(0, 1);
47:  cswapHelper(1, 2);
48:  cswapHelper(2, 4);
49:  cswapHelper(3, 8);
50:
51:  # Inverse QFT
52:  qft4i();
```

Listing 1: Phase estimation written in OpenQASM 3.0.

## 3.7 OpenQASM Interpreter

To support writing generic programs in OpenQASM an interpreter was implemented in Qurigo. An interpreter is a computer program that takes the source code of another program as input and execute the programming logic of that source code to produce an output. Popular languages that use an interpreter to execute programs are Python and JavaScript.

The interpreter in Qurigo has two phases. The first phase makes a syntaxtical analysis and verification of the program and build an in-memory model of the logical steps of the program. The in-memory model is called the *parse tree* in this thesis, sometimes it can also be called an *abstract syntax tree (AST)*. When the parse tree has been created an execution engine take that as input and run the program from the first step to the last step, at which point the execution of the program has finished.

The syntaxtical analysis and verification is performed by a *parser*. The parser compares the source code to the formal definition of the language, in this case it is a subset of OpenQASM 3. The parser ensures that there are no unknown syntaxtical elements in the source code, it also performs other validation tasks such as checking datatypes and that there are correct number of parameters for the quantum gates. If we assume that the source code contains the incorrect line `swap q[3];`, then the parser will detect that `swap` is a two-qubit gate but there is only one qubit specified as a parameter. The parser will then stop parsing the source code and report an error to the programmer.

Most of the programming work for Qurigo was spent building the interpreter for OpenQASM. The execution engine is the component that does the actual quantum computer simulation, and the previous sections in this chapter describe challenges related to the exection engine. Naturally the execution engine was challenging in many ways, but the OpenQASM parser also took a long time to implement.

### 3.7.1 Relationship between source code and parse tree

To illustrate the relationship between the source code and the parse tree consider the OpenQASM source code in Listing 2 and then crossreference that with Figure 14 to see how the source code is parsed into an parse tree. Note that when executing a subroutine it uses a shared definition of the subroutine, the nodes are not duplicated in the parse tree.

```
qubit[8] q;

def cswapHelper(int control, int repeat) {
    int index = 0;

    while(index < repeat) {
        cswap q[control], q[4], q[5];
        cswap q[control], q[5], q[6];

        index = index + 1;
    }
}

x q[4];
h q[3];

cswapHelper(2, 4);
cswapHelper(3, 8);
```

Listing 2: Sample OpenQASM program.

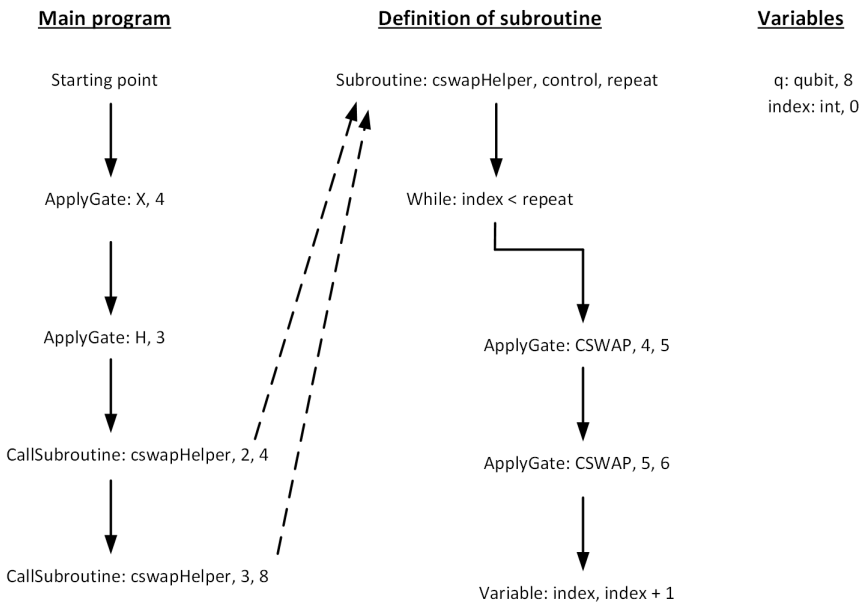



Figure 14: A sample parse tree created by Qurigo.

### 3.7.2 Language that are not interpreted

The opposite to using an interpreter is to perform syntaxtical analysis and verification using a *compiler*. A compiler translates the source code to machine code that a computer can execute natively. This is generally seen as better approach if performance is the goal because many optimizations can be made by the compiler. The classical examples of compiled languages are C and C++, but the Qiskit programming model also uses a compiler. Qiskit use the term *transpiler* to descibe the transformation of the Qiskit source code to the native instructions of a quantum computer.

# 4 Quantum Algorithms

## 4.1 Introduction

One of the objectives defined in chapter 1 was to use the simulator to implement and execute a quantum algorithm. This chapter describes how QFT (Quantum Fourier Transform) and Shor's algorithm are supported in Qurigo.

As discussed in chapter 1.1 a program that solves a problem using a quantum computer is not just a quantum computer running a program. There is always a classical program that prepares a system state for the quantum computer, interprets the result and perhaps makes further calculations before presenting a result to the user. I believe that the main reason for this is that it is hard to program a quantum computer and it is expensive to execute programs on a quantum computer, therefore you want to limit the usage of a quantum computer to only those parts of an algorithm where the quantum computer gives an advantage.

It is important to understand that while some quantum algorithms gives a definite answer to a problem (e.g. Grover's algorithm), other algorithms give a result which is correct only with a certain probability (e.g. QFT). The impact is that the quantum algorithm may give an incorrect result, so you have to test to see if the result is correct or not. This is of course an inconvenience for most types of problems, but if you have a problem that is unsolvable using a classical computer then getting a correct answer with some probability is so much worth it. You just rerun the quantum program a few times until you get the correct answer. This requires that there is an effective way to test if an answer is correct or not (i.e., that correctness can be validated in polynomial time). For Shor's algorithm the probabilty of a correct answer is approximately 40%, so you would expected to run the quantum program of the algorithm a few times to get a correct answer. To test if the answer is valid or not you can just make a simple division and see if the answer is a factor or not, this is a test that can be efficiently executed on a classical computer.

In the case of Shor's algorithm the quantum advantage is in the execution of the QFT. The result from the quantum part of Shor's algorithm is processed in the classical part and among other things a GCD (Greatest Common Divider) is executed. [5] The GCD can be effectively (polynomial time) implemented using a classical program. Running the GCD on a quantum computer just makes the algorithm more complex and more expensive to execute.

An interesting conclusion from this is that if the secret in Shor's algorithm is that the QFT gives a quantum advantage when applied to an entagled quantum state, then other algorithms that apply QFT on an entangled state may also be able to gain the same advantage.

But the opposite is also true. If an algorithm does not contain a step that

[5] Taking a step back it is worth reflecting on that in the implementation of Shor's algorithm, an innovative modern algorithm from the 20th century, we make use of GCD, which was described by Euclid's as ealy as 300 B.C. No work in science stands alone, it is always built on the work of previous generations.

has a quantum advantage, then there is no advantage at all in trying to run it on a quantum computer.

Unfortunately there are not that many known algorithms that give a quantum advantage so the number of problems that can be sped up with a quantum computer is quite limited. Much of the research on quantum algorithms focus on optimizating the implementation of the known algorithms that gives a quantum advantage, the goal is to make them useful on the small and noisy quantum computers that are available today.

## 4.2 Computational complexity: P and NP

The discussion in the previous section can also be formulated in the terminology of *computational complexity theory*.

There are many complexity classes but we limit our discussion to the classes P and NP. The P (Polynomial) complexity class contains problems that can be solved in polynomial time. When the size of the problem (e.g. the number of digits in a number to be factorized) grows the time and resources require to solve the problem grows as some polynomial. These are problems that typically are solvable using a classical computer.

The NP (Non-deterministic Polynomial) complexity class contains problems for which there may not be any known solution using polynomial time. But if a potential solution is found, then the validity of the solution can be verified in polynomial time (using a P-class algorithm).

Using this terminology we can say that prime number factorization is a NP-problem, there is not any known classical algorithm executing in polynomial time. Using a quantum computer and Shor's algorithm we can get a potential solution to the factorization problem, but we don't know for sure if that is the solution or not. However, we can validate the potential solution using the GCD-algorithm, which is a P-class algorithm.

More on complexity theory can be found in [14] and [17].

## 4.3 Additional Quantum Gates

When the work started on implementing QFT and Shor's algorithm there was a need to add two more gates to the instruction set of Qurigo. They were not part of the inital set of gates supported by the instruction set, but as they are key components of the algorithms it made sense adding them to the supported gates. The alternative would have been to define them as subroutines each time they were needed, and this would both make the quantum program more complex and a larger subset of the OpenQASM specification would have had to be supported.

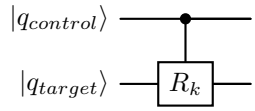
### 4.3.1 Controlled Rotation $CR_k$

The controlled rotation gate ($CR_k$) is a two-qubit controlled gate that rotates the target qubit if the control qubit is $|1\rangle$. The rotation is determined by the

value of $k$ and the rotation angle $\theta = \frac{2\pi}{2^k}$.

$$R_k = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{\theta i} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{\frac{2\pi i}{2^k}} \end{pmatrix}$$

When used in a quantum circuit the $CR_k$ gate is drawn as below.

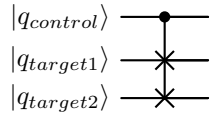



### 4.3.2 CSWAP

The $CSWAP$ gate is a three-qubit controlled gate that swap the value of two qubits if the control qubit is $|1\rangle$. Being a three-qubit gate it is represented by a $2^3 \times 2^3 = 8 \times 8$ matrix.

$$\text{CSWAP} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

When used in a quantum circuit the $CSWAP$ gate is drawn as below.




## 4.4 Quantum Fourier Transforms (QFT)

Fast Fourier Transform (FFT) is a technique that is widely used in science, the quantum version of FFT is called a Quantum Fourier Transform (QFT). The algorithm described in this chapter is the Inverse QFT, because that is the version of QFT that is used by Shor's Algorithm.

The general implementation of the Inverse QFT for an arbitrary number of qubits is fairly simple and follows a repetitive pattern, as shown in Figure 15. This algorithm requires $n$ Hadamard gates and $\frac{n(n-1)}{2}$ $CR_k$ gates when applied to a system with $n$ qubits.

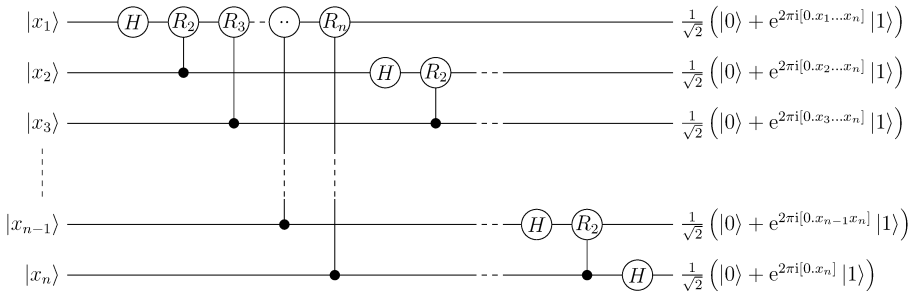


Figure 15: Quantum circuit for QFT with $n$ qubits. [By Trenar3 - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=66422968]

Below is a circuit that implement QFT for four qubits. These circuits also include the $SWAP$ operations that are required in order to get the qubit-ordering correct after performing the QFT.

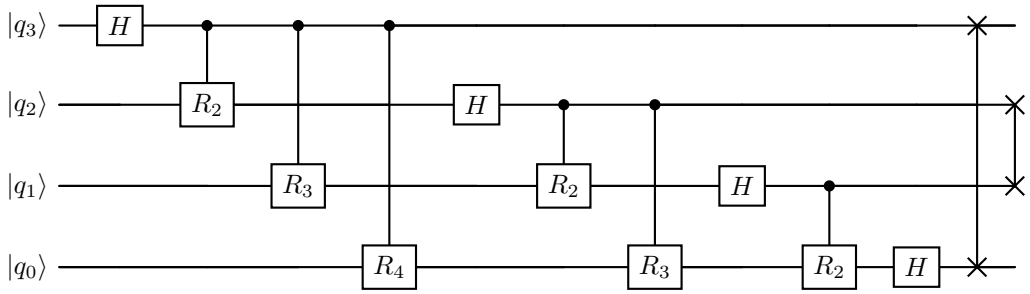


Figure 16: QFT for four qubits.

### 4.4.1 Implementation in Qurigo

Below is the implementation of the QFT circuit using OpenQASM in Qurigo, it is the implementation of the 4-qubit QFT shown in Figure 16. It defines a reusable subroutine called `qft4i` that applies a sequence of Hadamard gates (`h`), Controlled Rotation ($CR_k$) gates and `swap` gates to the qubits in the system state.

`h q[3];` means apply the Hadamard gate on the $4^{\text{th}}$ qubit, remember from section 3.2 that we use zero-based numbering for the qubit. When applying the controlled rotation gate it takes three parameters; the control qubits, the qubit on which to apply to rotation and the $k$-value that determines the angle of the rotation (see section 4.3.1). `crk q[0], q[3], 4;` means apply the rotation $\theta = \frac{2\pi}{2^4}$ to the $4^{\text{th}}$ qubit if the $1^{\text{st}}$ qubit is $|1\rangle$.

```
def qft4i() {
    h q[3];
    crk q[2], q[3], 2;
    crk q[1], q[3], 3;
    crk q[0], q[3], 4;

    h q[2];
    crk q[1], q[2], 2;
    crk q[0], q[2], 3;

    h q[1];
    crk q[0], q[1], 2;

    h q[0];

    swap q[0], q[3];
    swap q[1], q[2];
}
```

Listing 3: OpenQASM subroutine performing a QFT on four qubits.

## 4.5 Shor's Algorithm

Perhaps the best known quantum algorithm is Shor's algorithm for integer factorization, it was developed in 1994 by Peter Shor [21]. To factorize an integer means that we find all prime factors of that integer. Calculating prime factors is complex and time consuming on a classical computer when the numbers get large, the computational resources required grow exponentially ($2^n$, where $n$ is the number of digits in the number).

When using Shor's algorithm the computational resources required to factor integers grow polynomially. There are many different implementations of Shor's algorithm, but in [7] they describe an algorithm that require $3n + 0.002n \log n$ logical qubits and $0.3n^3 + 0.0005n^3 \log n$ Toffoli gates to factor a $n$-bit RSA integer. This is significantly better than the best know classical algorithm which is $2^{n^{1/3}}$.

Shor's algorithm is a concrete example of an algorithm where a quantum computer can solve a problem that has real world applications. When the times comes that quantum computers are large enough and stable enough to efficiently execute Shor's algorithm they will pose a severe threat to the integrity of cryptographic methods based on large numbers with large prime factors, specifically the RSA [6] crypto would become insecure.

But we are still a long way from being able to practicaly apply Shor's algorithm. The estimation by [7] show that we need 20 million noisy qubits to be able to solve the RSA crypto with 2048 bits.

[6] RSA, named after its inventors Rivest-Shamir-Adleman, is a cryptographic algorithm whose security relies on that it is impossible to determine the prime factors of very large numbers (when using 2048 bits it compares to a decimal number with 617 digits). If quantum computing can factor numbers of this order then a message encrypted using RSA will no longer be secure.

### 4.5.1 High-Level Description

As already mentioned, Shor's algorithm is a mix of classical computation and quantum computation. The classical part can be summarized as below, a simplified implementation is shown in section 4.5.2.

1. Test N for being even or a prime power.
2. Generate a random number $1 < a < N$.
3. Check if $a$ is a factor to N.
4. Use quantum algorithm to find the order $r$ or $a$, using a phase estimation algorithm based on QFT.
5. Use $r$ to construct a candidate factor and check if it is valid using GCD.
6. If it is not a factor, go back to step 2.

The quantum part of the algorithm is a phase estimation using the quantum circuit shown below in figure 17. It creates an entangled state using Hadamard gates and then apply powers of the controlled unitary $U$, finally an inverse QFT is applied to the state and a mesaurement is performed. The unitary $U$ multiply by $a$ (modulo $N$). An implementation of phase estimate using OpenQASM is detailed in chapter 3.6.1.

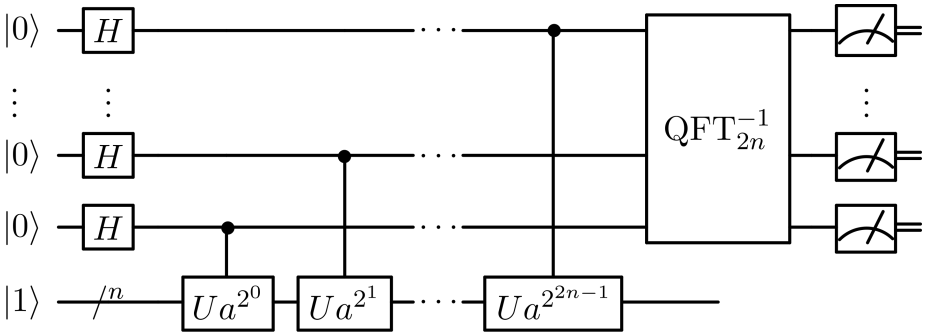



Figure 17: Quantum circuit for phase estimation with $n$ qubits. [By Bender2k14 - Own work. Created in LaTeX using Q-circuit. Source code below., CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=34319883]

### 4.5.2 Implementation of Shor's Algorithm

The program below is a simplified version of the implementation of the classical computer part of Shor's algorithm in Qurigo. It is written using Microsoft .NET and C#, but it could have been written in any programming language (e.g. Java, Python or C++). In this case C# was chosen because Qurigo is implemented in C# and it makes it simple to call Qurigo without having to spend time implementing a cross-language function call mechanism.

The functionality covered by this program are steps 3, 4 and 5 from the description of Shor's algorithm in section 4.5.

The quantum algorithm part of this program is the execution of a OpenQASM program using `qurigo.Run`. The logic of the quantum program that is executed (`phase-estimation-6.qasm`) is analyzed in section 3.6.1.

```
public static void Factor(int N, IServiceProvider serviceProvider)
{
    // Step 1 and 2 are not included.

    // Step 3: Use quantum algorithm to find the order r or a,
    // using a phase estimation algorithm based on QFT.
    QurigoSimulator app = new QurigoSimulator(serviceProvider.
        GetService<IExecutionContext>());
    double phase = qurigo.Run("phase-estimation-6.qasm");

    if (phase == 0 || phase == 1)
    {
        // Trivial factor, try again
        Factor(N, serviceProvider);
        return;
    }

    // Step 4: Use r to construct a candidate factor and check
    // if it is valid using GCD
    Fraction frac = Fraction.LimitDenominator(phase, 64);
    int r = frac.Denominator;
    int factor = GCD((int)Math.Pow(8, r / 2) - 1, N);

    // Step 5: If it is not a factor, go back to step 2.
    if(factor == 1 || factor == N)
    {
        // Trivial factor, try again
        Factor(N, serviceProvider);
        return;
    }
    if(N % factor == 0)
    {
        Console.WriteLine($"Non-trivial factor found: {factor}");
        return;
    }

    // Not a real factor, try again
    Factor(N, serviceProvider);
}
```

Listing 4: Classical part of Shor's Algorithm in Qurigo.

# 5 Results and Discussion

The deliverables from this thesis project is this report and Qurigo [15], a quantum circuit simulator running on a classical computer. During the development of Qurigo I have applied the theoretical background of quantum circuit simulation, as described in chapters 1 and 2, and the design of Qurigo is described in chapter 3.

After the initial research phase the development of Qurigo was started and the focus was on developing functionality directly related to quantum circuit simulation. This was interesting and it was rewarding to see new results every day. However, when the base functionality was completed and validated I started working on support for Shor's algorithm. This changed focus to implementing support for OpenQASM by writing an OpenQASM interpreter, which consists of a syntax parser and an execution engine. I choose to write my own parser for OpenQASM after looking at open source alternatives, in hindsight it might have been more productive to use an open source parser as a base.

The implementation of Shor's algorithm using Qurigo was translated and adapted from a sample in Qiskit. This required extending the subset of OpenQASM that Qurigo supported and also required new gates to be implemented in the instruction set.

The proper function of Qurigo has been validated using both manual mathematical validation using small quantum circuits and comparing with IBM's Qiskit simulator. The validation using Qiskit was done by comparing the state vectors before measurements, using trivial gates, QFT and phase estimation as input. It was very valuable to use Qiskit to validate the functionality of Qurigo, because it would have been hard to fully trust results if I both developed Qurigo and also performed the validation matrix multiplications (the largest matrices used were $1024 \times 1024$). I recommend Qiskit to anyone who wants to get started with quantum computation.

Since work started on supporting OpenQASM most of the executions of quantum circuits has been performed using the $H/T/CNOT$ instruction set. But early in the project I also implemented the instruction set used by the IBM Eagle r3 quantum computer. I used that to compare and validate the results from my simulator to that of the state vector from IBM's real quantum computer. Executing a quantum circuit on a real quantum computer was a very inspiring experience.

Overall I think I have reached the main goal of the thesis project, to show that it is possible to implement a quantum circuit simulator in this limited time scope and that it behaves as expected. The one regret is not having been able to include quantum noise and error correction in Qurigo.

# References


[1] IBM Quantum Platform. https://quantum.ibm.com/, 2024.

[2] OpenQASM. https://github.com/openqasm/openqasm, 2024.

[3] S. Aaronson and D. Gottesman. Improved simulation of stabilizer circuits. *Physical Review A*, 70(5), Nov. 2004.

[4] A. Cross, A. Javadi-Abhari, T. Alexander, N. De Beaudrap, L. S. Bishop, S. Heidel, C. A. Ryan, P. Sivarajah, J. Smolin, J. M. Gambetta, and B. R. Johnson. Openqasm 3: A broader and deeper quantum assembly language. *ACM Transactions on Quantum Computing*, 3(3):1–50, Sept. 2022.

[5] R. P. Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21(6/7), 1982.

[6] M. Fowler. Inversion of control containers and the dependency injection pattern. https://martinfowler.com/articles/injection.html, 2004.

[7] C. Gidney and M. Ekerå. How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits. *Quantum*, 5:433, Apr. 2021.

[8] S. S. Gill. Quantum computing: A taxonomy, systematic review and future directions, 2021.

[9] D. Gottesman. The Heisenberg Representation of Quantum Computers, 1998.

[10] B. Heim, M. Soeken, S. Marshall, C. Granade, M. Roetteler, A. Geller, M. Troyer, and K. Svore. Quantum programming languages. *Nature Reviews Physics*, 2(December 2020), 2020.

[11] IBM Quantum Research Blog. The hardware and software for the era of quantum utility is here. https://www.ibm.com/quantum/blog/quantum-roadmap-2033, 2023.

[12] T. Jones, A. Brown, I. Bush, and S. C. Benjamin. Quest and high performance simulation of quantum computers. *Sci Rep*, 9(10736), 2019.

[13] S. Jordan. Quantum algorithm zoo. https://quantumalgorithmzoo.org, 2022.

[14] P. Kaye, R. Laflamme, and M. Mosca. *An Introduction to Quantum Computing*. Oxford University Press, 2007.

[15] M. Lindberg. Qurigo - a quantum circuit simulator. https://github.com/MattiasLindberg/Qurigo, 2024.

[16] S. Maity, A. Pal, T. Roy, S. B. Mandal, and A. Chakrabarti. Design of an efficient quantum circuit simulator. *2010 Internaltional Symposium on Electronic System Design*, (2010), 2010.

[17] M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2016.

[18] NobelPrize.org. The nobel prize in physics 2022. `https://www.nobelprize.org/prizes/physics/2022/summary`, 2022.

[19] J. Preskill. Quantum computing in the nisq era and beyond. *Quantum*, 2:79, Aug. 2018.

[20] E. Rieffel and W. Polak. An introduction to quantum computing for non-physicists. *ACM Computing Surveys*, 32(3), 2000.

[21] P. W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, 1994.

[22] B. Villalonga, S. Boixo, B. Nelson, C. Henze, R. Biswas, and S. Mandra. A flexible high-performance simulator for verifying and benchmarking quantum circuits implemented on real hardware. *npj Quantum Inf*, 5(86), 2019.

# Index