

Forside

Eksamensinformation

NDAA93062E - Computer Science Thesis 30 ECTS,
Department of Computer Science - Kontrakt:133069
(Mathias Bohn Rasmussen, Nicholas Christian Langkjær
Ipsen)

Besvarelsen afleveres af

Nicholas Christian Langkjær Ipsen
hts399@alumni.ku.dk

Mathias Bohn Rasmussen
pnd447@alumni.ku.dk

Eksamensadministratorer

DIKU Eksamen
uddannelse@diku.dk

Bedømmere

Troels Henriksen
Eksaminator
athas@di.ku.dk
☎ +4535335718

Patrick Bahr
Censor
paba@itu.dk

Besvarelsesinformationer

Titel: Quthark: En Data-parallel Universel Kvantekredsløbssimulator, Implementeret i Futhark

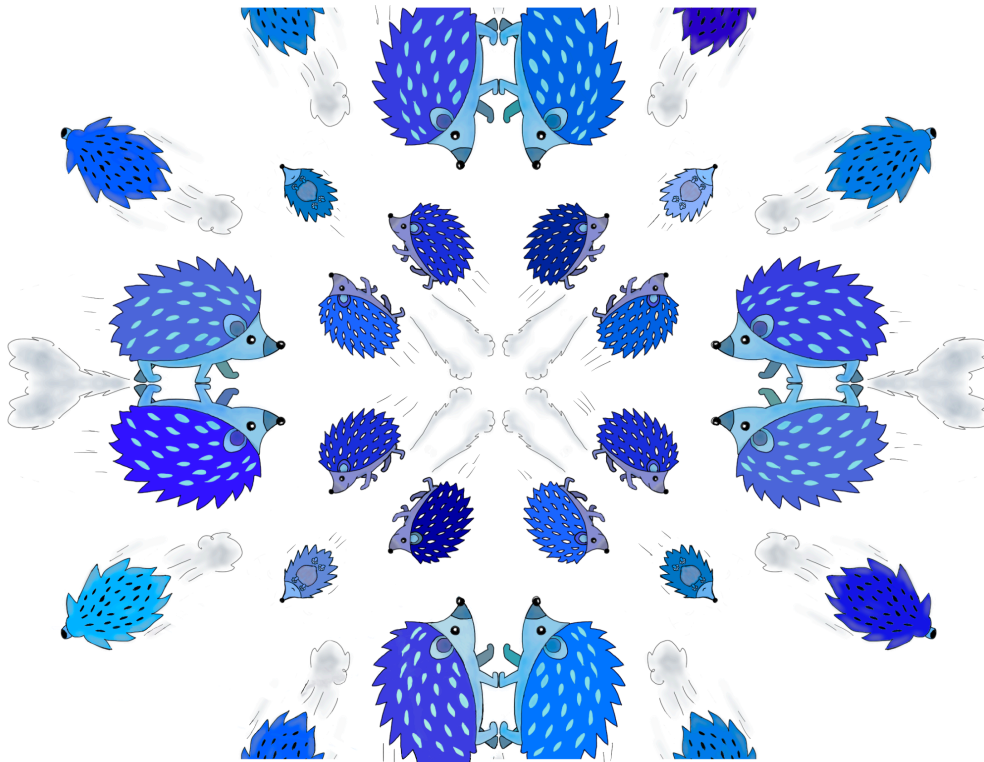
Titel, engelsk: Quthark: A Data-parallel Universal Quantum Circuit Simulator, Implemented in Futhark

Vejleder / eksaminator: Troels Henriksen

Tro og love-erklæring: Ja

Må besvarelsen gøres til genstand for udlån: Ja

Må besvarelsen bruges til undervisning: Ja



Master's Thesis

Mathias B. Rasmussen (pnd447) & Nicholas C. L. Ipsen (hts399)

Quthark

A data-parallel universal quantum circuit simulator, written in Futhark

Advisor: Troels Henriksen
Co-advisor: Matthias Christandl

2022-08-15

Quthark

A data-parallel universal quantum circuit simulator, written in Futhark

Mathias B. Rasmussen (pnd447) & Nicholas C. L. Ipsen (hts399)

Abstract

It is becoming clear that expecting ever greater computing performance at ever lower cost is no longer a given. Writing more efficient, parallel code can be helpful, yet there are some problems which are unlikely to ever be solvable in polynomial time using classical algorithms. Quantum algorithms may help us get there for some of them, but actual quantum hardware is still rare, and the quantum computers accessible to the public are too primitive for most meaningful computation. In order to explore lowering the time-investment needed to participate in the development of performant quantum circuit simulators, we implement a simple quantum circuit simulator in the data parallel functional language Futhark. We examine and document the effects of various optimisation strategies, and end up with a reasonably performant, portable simulator, generated from easily readable, concise code.

Contents

1	Introduction	1
2	Background	3
2.1	Quantum Computation	3
2.1.1	Bra-ket Notation and Other Formalisms	6
2.1.2	The State Vector	7
2.1.3	Quantum Logic Gates	8
2.1.4	Gate Application Example	12
2.1.5	Pure and Mixed States	14
2.1.6	Measurements	14
2.1.7	Identifying Efficiently Simulateable Quantum Circuits	17
2.2	Quantum Fourier Transform (QFT)	18
3	Method	20
3.1	Futhark	20
3.2	Qiskit	21
3.2.1	Integrating with Qiskit	22
3.3	The State Vector Simulator	22
3.4	Implementation	23
3.4.1	Qubit Gates	24
3.4.2	Key Function Definitions	27
3.4.3	Measurements	28
3.4.4	Supported Gates	28
3.4.5	Improving on Theoretical Performance	29
3.5	Benchmarking	29
3.5.1	Circuit Depth vs Size	30
3.5.2	Generating a Test Case	31
3.5.3	Correctness	31
3.5.4	Complex number representation	32
3.6	Results	32
3.6.1	Memory Usage	34
3.6.2	Locality	36
3.6.3	Cache Blocking	37

3.6.4	Applying Custom Gates	39
3.7	Other Experiments	40
3.7.1	A Sparse State Vector	40
3.7.2	Gate fusion	41
3.7.3	Just-in-time	42
3.7.4	Work Optimal Simulator	44
3.8	Future Work	44
3.8.1	Accuracy	45
3.8.2	Better JIT	45
3.8.3	Fusion	45
3.8.4	Control flow	46
3.9	Reproducing Results and Executing Code	46
4	Discussion	48
5	Conclusion	49

List of Figures

2.1	The Bloch sphere	4
2.2	A random quantum circuit containing a wide variety of gates.	9
2.3	A general Quantum Fourier Transform for n qubits.	19
3.1	Two qubit example of applying a single qubit gate to the first and second qubit respectively.	25
3.2	A three qubit quantum circuit with depth three and size eight. The squares with capital letters are gates labelled as in subsection 3.4.4. The two crosses connected by a line is a SWAP-gate.	32
3.3	Execution time of a size 1000 circuit run on differently sized state vectors. . . .	33
3.4	Memory usage per qubit.	35
3.5	Quthark v2 and Aer locality, 29 qubit state vector, 1000 hadamard gates. . . .	36
3.6	Executing a randomly generated unitary gate on a 20 qubit state vector.	39
3.7	Sparsity of state vector after executing random generated circuits. White is sparse, black is dense	40
3.8	Executing a randomly generated circuit of size 1000, on different sized state vectors.	43

Chapter 1

Introduction

High performance and low memory usage are perhaps the two most important determinants of the quality of an algorithm ¹. Yet despite much research, there remains problems not solvable in polynomial time by any known classical algorithm. While not all of them are known to be solvable in polynomial time by quantum computers, some are. Researchers have already developed and verified polynomial-time quantum algorithms which solve some of these problems, some of them even viable on current quantum hardware.

Despite being verified as theoretically solvable in polynomial time, most of the interesting problems need access to more complex quantum computers than currently exist. Even solving small problem instances is often not possible today, leading to obvious problems when researchers need to verify their algorithm on actual problem instances.

Quantum circuit simulators exist primarily to fill this hole between what is possible in theory, and what is possible on reasonably accessible quantum hardware today. Using mathematical models verified on actual quantum hardware, it is possible to simulate quantum computations more complex than what is possible on publicly announced quantum hardware. This is one way in which simulators open up quantum computing to a wider audience of developers.

These simulators can be broadly divided into two camps: Slow simulators written in easily-readable high-level languages such as Python, and fast simulators written in low-level languages such as CUDA and OpenCL. Even though many of the biggest simulators are published as open source, this divide limits wider participation in the development of quantum circuit simulators to simulators which are not actually usable in practice. There are many examples of how to write such a simple simulator, but as soon as one wants to write one which can be used to solve actually interesting problems, it becomes much harder to find code most programmers can read, understand and improve within a reasonable time-frame.

Writing an efficient quantum circuit simulator tends to require many lines of intricate, low-level code in order to efficiently carry out operations simultaneously on devices such as GPUs. Inherently more scalable than CPUs due to their architecture, GPUs are designed to solve many small problems simultaneously. This low-level code is hard to write optimally, and even harder to read.

Futhark is a functional language created in an attempt to address exactly this issue. While

¹Second to correctness.

one can write sequential code in Futhark, it is designed to attempt to make sure that if code is written in a manner which can be executed in parallel, it will be executed in as parallel a way as possible.

Writing in a high-level language does introduce some limitations when designing a program. We do not have the same granular control of how calculations are performed, and thus not the same opportunities for optimization. Likewise, Futhark being a purely functional language entails that side-effects are disallowed by definition, meaning that doing something like in-place updates of a data-structure is only possible when the compiler can be sure that the memory we are writing to is only ever read by the thread writing to it.

What accepting these limitations ends up giving us in return, is a language which provides strong guarantees of performance and correctness. While we cannot customize the details of exactly how the code we write is executed, the compiler generates code that is very performant on average, with the prohibition of side-effects meaning that we end up with much more readable code.

In this thesis we attempt to implement a quantum circuit simulator in Futhark, aiming for fast, portable, easily-readable code. We call this simulator Quthark. In doing so, we hope to provide a basis for experts on classical computing to more easily understand quantum computing, and more easily participate in the development of a practical quantum simulator which can run on a wide range of hardware.

Chapter 2

Background

As computer scientists, we try to abstract ourselves away from thinking of the quantum computational realm in any other way than as pure mathematically representable objects. We acknowledge the limitations of our knowledge regarding the quantum realm and distance ourselves from the complexities of the real world. There lies a simple beauty in only thinking about the purely mathematical representation.

We gladly swallow the blue pill.

2.1 Quantum Computation

The rules of quantum computation are derived from quantum mechanics, which is perhaps one of the most daunting subjects for non-physicists. Despite this, if one chooses to ignore the mysteries of how exactly quantum physics makes sense in the real world, the mathematical models of quantum information theory, which govern all quantum computation, are actually rather simple. All quantum computation can be described by so-called quantum circuits, and the quantum circuit simulation that we will explore in this project can be reduced down to linear algebra and probability theory.

In the real world, to perform quantum computations means taking something very small, such as a photon, putting it in an environment we control to such a degree that it is isolated from almost all noise from the world around it, and manipulating it in a way that makes something about it change in a predictable manner. For a photon, that might be to change its polarization. By starting with a photon with some known polarization we define as the state 0, if we define the opposite of that as 1, we can affect the photon in a controlled manner, and flip it between these two states. We called this quantum computation, and it is, but the discerning reader might wonder why this couldn't just be done in a classical system. After all, what is described so far gives results no different than flipping a regular bit in a classical computer.

Where this equivalence breaks down is when we realize that the polarization of a photon isn't limited to just two possible states. A photon can be polarized in any one of an uncountably infinite number of ways, and by applying the appropriate stimulus, we can achieve any of those polarization states.

This seems like a much better way to store information than classical bits, and one might

wonder why we don't do just that in classical computers. After all, why spend 16 bits to store just a single UTF-16 character when, given the right encoder and decoder, one could potentially encode everything ever written into a single photon, and still have infinite space left.

First of all, encoding accurately becomes an issue. In the real world, we can only manipulate a photon with a certain degree of precision. But even then, it doesn't seem unreasonable to reach for a slightly closer goal, like using just a single photon to encode a single one of those 16-bit characters. That would require being able to reliably put the photon in one of just 2^{16} possible states needed to represent a 16-bit value, and keep it there. Difficult, perhaps, but a lot more reasonable than infinitely many states, for which we would need to define infinitely many decoders.

In the real world, it turns out that particles can only be reliably distinguished in two opposing states.¹

But what does "two opposing states" even mean here? Without going into the physics, which is *certainly* outside the scope of this project, it can be useful to think of the state of a particle as a point on a sphere. Any such point can be represented as a complex number, with the real and imaginary parts the coefficients of a vector pointing from the center to said point on that surface of the sphere. This analogy is in fact so useful that this sphere has been named, and it is commonly referred to as the *Bloch sphere*.

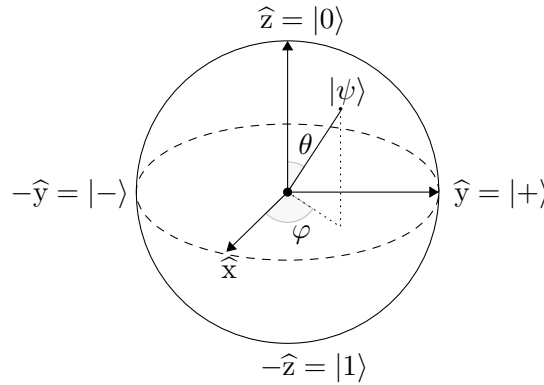


Figure 2.1: The Bloch sphere

Opposing states are any two states that are exactly opposite one another on the Bloch sphere. An example could be $\{0+0i, 1+0i\}$, which is commonly referred to as the computational basis. These states are labelled \hat{z} and $-\hat{z}$ on the Blochs sphere in Figure 2.1.

While those definitions look fancy and complex, at the end of the day, we're back to being able to distinguish between just these two states. So what have we gained over classical computation?

Recall that our main problem was down to building a decoder that can distinguish between more than two states. An encoder, however, is fairly trivial. Now, keeping the particle isolated from the world around it well enough that the state is preserved for long enough to measure it

¹Nielsen and Chuang 2011, p. 280.

with our decoder is tricky. Temperature fluctuations, light, mechanical movement, even cosmic rays can make the particle lose state coherency. In spite of these challenges, it is now very possible to preserve a coherent state through a significant number of operations.²

It turns out that, when the state of the particle is represented by some point *between* the two points on the Bloch sphere, it will still be measured by the decoder as being in either of the two positions it can distinguish between. The probability of it being measured to be in either state corresponds to the distance between the point representing the actual state of the particle on the Bloch sphere, and the point representing the measured state.

Let's do a quick example. We will use the Bloch sphere here, which does have some domain-specific notation, which we will talk more about in subsection 2.1.1. For now, it is enough to understand how the computational basis is represented on the sphere. Now, assume that we have manipulated a large number of particles to be in the state represented by \hat{y} ; located exactly equidistant between the two on the surface of our Bloch sphere. If we apply an appropriate decoder, we find that, as we measure more and more particles, the probabilities of both results gradually approach 50%, exactly as one might expect. Note that, if we measure the particles a second time, they will always be in the same state as the first time they were measured. What this leads us to conclude is that, although we can only reliably distinguish between two opposing states when decoding, repeating the experiment will gradually get us closer to the true state of the particle. This is what allows quantum computers to handle huge and very complicated systems with only a few hundred particles, since infinitely many states can be represented by a single particle, as opposed to the classical limit of just two states. Particles in such a state are said to be in a superposition, as one cannot know the result of measuring them before doing so.

It is important to emphasize that there is nothing special about a particle in a superposition, and that whether a given particle is even in a superposition or not is entirely relative to the basis we choose to measure in. One could instead measure the particle in the basis $\{\hat{y}, -\hat{y}\}$. When measured in this basis, the particle would not be in a superposition, and would always be found to be in the state \hat{y} .

But didn't we just determine that once measured in one way in a certain basis, the particle would continue to be measured in that way? What happens if we prepare the particle in state \hat{y} , measure it to be either $\{\hat{z}, -\hat{z}\}$ basis, then try to measure it in the $\{\hat{y}, -\hat{y}\}$ basis?

If we've measured the particle to be in state $\{\hat{z}, -\hat{z}\}$, will it now behave as though it is in that state, and no longer be consistently in the state \hat{y} ? Or is it actually still in the original state, and just so happens always to measure in a certain way, perhaps thanks to some additional, hidden variable?

The answer turns out to be the former, which may be disconcerting at first. After all, we haven't performed any manipulation of the particle between measurements. The key to understanding what is happening here is to regard even measurements as manipulations; they're just manipulations with the side-effect of telling us something about the state we're manipulating³.

This is why we can only distinguish between particles being in one of two opposing states; it is impossible to know the exact polarization of any given particle by performing a single measurement, since measuring it also changes it. Only by generating the multiple particles in

²Nielsen and Chuang 2011, p. 278.

³Or measurements with the side-effect of modifying the state, which is how they are usually described.

the same state, and measuring them in different bases, can we approximate the true, underlying state. See subsection 2.1.6 for the math behind this.

One last thing we'd be remiss not to mention, and which has given quantum mechanics much of its air of mystery, is the idea of entanglement. An entangled state is simply two or more particles which have had their state made dependant on each other. So if one is measured to be in a certain state, we always know what state its partner will be measured in. More on this in subsection 2.1.2.

One of the most interesting things about these kinds of groups of particles is that they are able to preserve this correlation when moved far apart, and even when used as parts of other quantum systems. This allows for some interesting algorithms such as quantum teleportation, which allows sending a single bit of information, yet actually transferring two, by "using up" an entangled particle, meaning that the bond between two entangled particles is broken.

These are the basic rules of the kind of system we wish to simulate. The state of particles can be uncertain and/or correlated to other particles, and when measured enough times, the normalized sum of the measured states must approach the true state of the system.

So how do we simulate that? There are many ways, each with their own unique advantages and drawbacks. We have chosen to implement perhaps the most intuitive type of simulator, the state vector simulator.

But first, we need to establish a common language for describing quantum systems.

2.1.1 Bra-ket Notation and Other Formalisms

As will become clear, representing quantum states classically can require quite a lot of space. Not just in a computer's memory, but on paper as well. In order to make working with them more tenable, it is beneficial to employ Dirac's bra-ket notation. This makes use of bras and kets, perhaps most intuitively understood as row and column vectors. When we write $\alpha \cdot \langle i|$, this can be interpreted to be a row vector with the value α in position i , while $\alpha \cdot |j\rangle$ can be understood as a column vector with the value α at index j , where α and β are complex numbers. If there is no value α or β , the vector just has the value $1 + 0i$ at the given index. In this thesis, we will sometimes use ψ and ϕ as a shorthand for *some state*, written as:

$$|\psi\rangle = \alpha_0 |0\rangle + \alpha_1 |1\rangle + \dots + \alpha_n |n\rangle \quad (2.1)$$

After this point, the vector $|\psi\rangle$ is taken to be the column-vector with α_0 at index 0, α_1 at index 1, and so on. Having read all of this, one might be tempted to conclude that transposing a bra $\langle\phi|$ would transform it into some ket $|\phi\rangle$. This is dangerously close to being correct; so much so that one can often forget that it is not. Indeed, for many cases, a simple transpose operation *is* sufficient. However, when the imaginary part of $\alpha \neq 0$, this breaks down. The correct way to turn a bra into a ket, and vice-versa, is to take the complex conjugate transpose, denoted as $*$, which amounts to flipping the signs of the imaginary parts of all α s after transposition⁴:

$$\langle\psi| = |\psi\rangle^* \quad (2.2)$$

⁴Nielsen and Chuang 2011, p. 85.

This is due to the definition of the inner product, and what it means for complex numbers. Because we always want the inner product between a bra and its dual ket to be 1 (see subsection 2.1.6), the imaginary part of complex numbers need to have their sign flipped, in order to get rid of them in the result.

Taking the concept a bit further, the inner product of two vectors can be written as $\langle\psi|\phi\rangle$, and the outer product as $|\psi\rangle\langle\phi|$. Finally, we can write the Kronecker product as $|\psi\rangle\otimes|\phi\rangle$, $|\psi\rangle|\phi\rangle$, or simply $|\psi\phi\rangle$.

Both the inner product, outer product, and Kronecker product turn out to be very useful for simulating quantum circuits. In our simulator, however, we do not make use of the outer or inner products. This is because while they are useful for describing the theory of quantum computations, they are not very efficient for actually simulating quantum computations on classical hardware.

2.1.2 The State Vector

State vectors are vectors of complex numbers representing amplitudes, which can be interpreted as the probability of measuring a quantum system to be in a certain state.

To understand why they are structured like they are, we first need to understand the qubit.

The quantum bit, or qubit for short, is, like the bit in classical computing, the smallest building block in quantum computing. In a physical quantum computer, it is the well-isolated particle in a controlled state described in the introduction to this section. Like the classical bit, it also preserves a state of 0 or 1, written as $|0\rangle$ and $|1\rangle$. But instead of being discrete like the classical bit, there is a continuous linear relation between the two outcomes. Such a state Q for a single qubit can be written as:

$$|\psi\rangle = \alpha_0 |0\rangle + \alpha_1 |1\rangle \quad (2.3)$$

Or in the more generally used vector representation:

$$\begin{pmatrix} \alpha_0 \\ \alpha_1 \end{pmatrix} \quad (2.4)$$

Even though the qubit state isn't discrete, upon being measured it will collapse to the value of either 0 or 1 in the measured basis. The squared absolute amplitude $|\alpha_0|^2$ tells us the probability of getting a 0 and $|\alpha_1|^2$ the probability of 1. Thus it makes sense that the sum of the absolute value of the probabilities is equal to 1.

$$|\alpha_0|^2 + |\alpha_1|^2 = 1, \alpha \in \mathbb{C} \quad (2.5)$$

This feature can be generalized to a multi-qubit state as well, such that all values in the state vector, absolute and squared, sum to 1.

$$\sum_{x \in \{0,1\}^n} |\alpha_x|^2 = 1 \quad (2.6)$$

Where $x \in \{0, 1\}^n$ denotes the complete set of binary strings, which consists of n bits being either 0 or 1.

The state vector can be calculated by finding the Kronecker product of the n individual qubits Q we want to represent in our system.

$$|\psi\rangle = Q_0 \otimes Q_1 \otimes \dots \otimes Q_{n-1} \quad (2.7)$$

For our purposes, it makes the most sense to index the state vector in base-2. So instead of writing $\alpha_2 \cdot |2\rangle$ when talking about the value in the third position of the state vector, we write $\alpha_{10} \cdot |10\rangle$. This is the convention we follow for the rest of this thesis.

$$|\psi\rangle = \alpha_{00} |00\rangle + \alpha_{01} |01\rangle + \alpha_{10} |10\rangle + \alpha_{11} |11\rangle \quad (2.8)$$

$$= \begin{pmatrix} \alpha_{00} \\ \alpha_{01} \\ \alpha_{10} \\ \alpha_{11} \end{pmatrix} \quad (2.9)$$

The number of entries in the state vector scales exponentially with the number of qubits that are part of the simulated quantum state.⁵ The value stored the first index in a two-qubit state vector is the probability of measuring the classical state 00 in the computational basis, if this state had been reached on a real quantum computer. The second index holds the same for the classical state 01, and so on. As mentioned earlier, the computational basis is just one of many a state can be measured in, but is perhaps the easiest to understand coming from a computer science background. It is also the basis most commonly used with many quantum circuit simulators, with some having it be the only basis they support performing measurements in.

Some quantum states are more computationally interesting than others. For example, any state which can be described as just $|i\rangle$, where i is some valid index, might as well be a classical state, since the ability of quantum states to take on a range of values is not utilized. In stark contrast to this are states like $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$. This is what is known as a maximally entangled state; the result of measuring one qubit is completely correlated with the result of measuring any other qubit in the state. As opposed to superposition, which is relative to a basis, entanglement is absolute and basis-independent; particles are either entangled or they are not. More about how measurements work, and the consequences of entanglement, in subsection 2.1.6.

This specific maximally entangled state is known as the Bell state, and is used in some of the most interesting results achievable in quantum computing⁶.

2.1.3 Quantum Logic Gates

Quantum circuits consist of a sequence of quantum logic gates, gates for short, which are applied to one or several qubits.

Circuits are typically visualized as wires going through labelled rectangles or some other shape, representing gates. The idea is that a qubit travels through each wire from left to right,

⁵ 2^n , where n is the number of qubits.

⁶Nielsen and Chuang 2011, pp. 16–17.

being affected by the gates it passes through on the way. The shapes representing each gate will be introduced shortly, so don't worry about them for now. The quantum state starts out in the state $|000\rangle$ and ends up in some state $|\phi\rangle \otimes |\psi\rangle$. This circuit has no deeper meaning, but does contain an example of every kind of circuit notation we need to concern ourselves with for this paper.

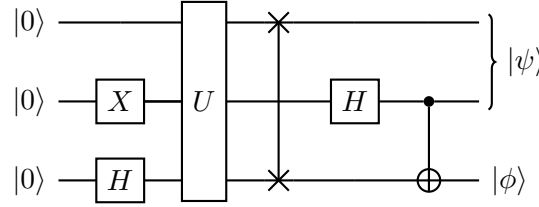


Figure 2.2: A random quantum circuit containing a wide variety of gates.

Quantum gates are norm-preserving, linear operators. They can be represented as unitary matrices, which are square, complex and have the property that their conjugate transpose is their own inverse:

$$U^*U = UU^* = UU^{-1} = I \quad (2.10)$$

This means that to reverse a gate, all one needs to do is find its conjugate transpose, then apply it, guaranteeing reversibility of operations. This is another distinction between classical computing and quantum computing, as classical circuits have many irreversible gates such as NAND and XOR, while all quantum circuits are reversible. As we can produce a quantum gate with the same effect as any classical gate, it is possible to write any classical circuit as a quantum circuit, with the added benefit of it becoming reversible.⁷

Common gates

Some gates are used more frequently than others. One set of very common gates are the rotation gates Pauli-X, Y and Z:

Pauli-X (X)		$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$
Pauli-Y (Y)		$\begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$
Pauli-Z (Z)		$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$

Table 2.1: Circuit- and matrix-representation of the generators of the Pauli-group

⁷Nielsen and Chuang 2011, p. 29.

The Pauli-group When considering the Bloch sphere representation of a qubit, applying one of the Pauli-gates amounts to rotating the state 180 degrees around the X, Y or Z axis. In the computational basis, this makes X the equivalent of a classical NOT-gate.

As $X^2 = Y^2 = Z^2 = I_2$, the two-by-two identity matrix is trivially part of the Pauli-group, along with 12 other gates which can be generated by combining the Pauli-X, Y and Z gates. Because those three gates are sufficient to generate all members of their group, they are called *generators of the Pauli-group*.

There are uncountably infinite possible quantum gates, making it a challenge to introduce all of them in the limited space available. Luckily, we do not need infinite gates to simulate a universal quantum computer.

In fact, there are many combinations of gates which together represent a complete basis for quantum computation. In classical computing terms, we would call a simulator or quantum computer capable of applying all of the gates in such a set to a quantum state Turing complete. One of the most commonly talked about such sets is Clifford+T.

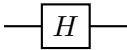
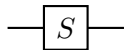
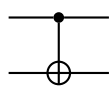
Hadamard (H)		$\frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$
Phase (S)		$\begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$
CNOT (CX)		$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$

Table 2.2: Circuit- and matrix-representation of the generators of the Clifford-group

The Clifford-group Like Pauli-X, Y and Z for the Pauli-group, H, S and CNOT are the generators of the Clifford-group of gates. This group turns out to be especially interesting for classical simulation of quantum circuits. We explore this fact in subsection 2.1.7.

These three gates are all pretty interesting, so we will take a moment to present each of them in turn.

Hadamard is a key gate in quantum computing, because applying this gate to a qubit will bring it into a superposition $\sqrt{2}$.

Phase combined with Hadamard can generate any Pauli-matrix.

CNOT, also known as CX, is the first 2-qubit gate we've introduced, and it happens to be a controlled 2-qubit gate. Controlled gates all follow a very similar pattern, where only a subsection of the matrix representation actually affects the state vector it is applied to. This one is the controlled version of the Pauli-X gate. Controlled gates interact with one or more control-qubits and target-qubits. The state of the former dictates whether or not the state of the latter is actually

modified by the gate, just like classical controlled logic gates⁸. The small dot indicates the control-qubit, which determines whether or not to apply Pauli-X to the target-qubit, indicated by the hollow circle with a cross in the middle. There also exists quantum gates with multiple control-qubits, and multiple targets.

Using the Hadamard-gate to put a qubit into superposition, and then using said qubit as control for a CNOT-gate will entangle the control- and target-qubits, generating an entangled state.

As implied by the name Clifford+T, the Clifford-group on its own is not a complete basis for quantum computation. One more gate is needed.

$$\pi/8 \text{ (T)} \quad \text{---} \boxed{T} \text{---} \quad \begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix}$$

The T-gate Adding T to the Clifford group gets us to the universal gate set Clifford+T, allowing us to perform any quantum computation. It turns out that this is the only gate in that set not efficiently simulateable on classical hardware. As a consequence of this, the time complexity of executing a quantum circuit is often measured in the minimum number of T-gates it requires.⁹

We need to introduce just two more types of gate to more easily be able to understand the quantum circuit in Figure 2.1.3:

$$\text{SWAP} \quad \begin{array}{c} \text{---} \times \text{---} \\ | \\ \text{---} \times \text{---} \end{array} \quad \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The SWAP-gate As its name implies, the SWAP-gate swaps the amplitudes of two qubits in the state vector.

$$\text{Unitary (U)} \quad \begin{array}{c} \text{---} \\ | \\ \boxed{U} \\ | \\ \text{---} \end{array}$$

A unitary gate All quantum gates are unitary gates. So when we want to talk about *some* unspecified gate, we can label it U. This one is a 2-qubit gate, but since they are defined as a generic gate, they can affect any number of qubits.

⁸XOR is in fact the classical equivalent of CNOT.

⁹Gottesman 1998.

Multi-qubit states

The dimensions of the matrix representations of quantum gates depend entirely on how many qubits they target. A one-qubit gate targeting a single qubit can be represented as a two-by-two matrix, a two-qubit gate a four-by-four matrix, a three-qubit gate eight-by-eight, and so on. This can be generalized as $2^k \cdot 2^k = 2^{2k}$, where k is the number of qubits the gate can interact with.

A one-qubit gate U can be applied to a qubit Q by calculating the dot product between the two.

$$U = \begin{pmatrix} u_{00} & u_{01} \\ u_{10} & u_{11} \end{pmatrix} \quad (2.11)$$

$$Q \cdot U \quad (2.12)$$

Much like in classical computing, working with a single qubit is usually not that interesting.

However, in moving to a multi-qubit state, a new problem arises: Since a single gates are defined by how they apply on a certain number of qubits, we can't apply a gate of differing dimensionality to the state as easily as we can when they match.

A simple solution to this problem is to scale the gate to the size of the state vector. We can do this by finding the Kronecker product of the gate U and $n - 1$ two-by-two identity matrices I_2 . By placing U as the i 'th part of the product, the corresponding i 'th qubit will be affected by the gate.

In the following equation I subscript denotes the position and not the size of the identity matrix, as is else the convention in this report.

$$U = I_1 \otimes I_2 \otimes \cdots \otimes I_{i-1} \otimes U_i \otimes I_{i+1} \otimes \cdots \otimes I_n \quad (2.13)$$

Now we can once again apply our gates to specific qubits in a multi-qubit state vector.

$$|\psi\rangle U \quad (2.14)$$

Another challenge is multi-qubit gates. As we have already established, all two qubit gates contain 2^{2n} values, and both application and scaling work exactly the same as for single qubit gates; so long as both the targeted qubits are next to each other.

If there are qubits which the gate is not supposed to affect in-between the two targets however, one of two things need to happen. Either an appropriate gate must be used to transform the state vector and move the targeted qubits next to each other, and back again, or the gate must somehow be split in two, yet still retain its effect.

2.1.4 Gate Application Example

First, let's introduce CNOT in more detail, as an example of a two-qubit gate:

$$\begin{pmatrix} \alpha_{00} \\ \alpha_{01} \\ \alpha_{10} \\ \alpha_{11} \end{pmatrix} \cdot \text{CNOT} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \quad (2.15)$$

Recall that the state vector can be seen as a collection of probabilities of measuring the quantum bits to be in a certain state. By observing the result of applying CNOT to the state vector, it should be clear that it performs as expected.

Practically speaking, applying the CNOT-gate is just switching the value of two indices of the state vector, while leaving everything else alone. Making this gate apply to qubits at arbitrary neighboring indices works just like it did for single qubit gates, but having qubits in between the targets becomes an issue. We have to scale what is essentially the identity gate and the x-gate separately, then add them together somehow. And that is exactly the solution. Calculating the Kronecker product of scaling one gate, then scaling the other and adding the results is a bit too simple though. We need to mask out the parts which are being done by the matrix we are adding, or the matrix will no longer be unitary, and we will be performing an illegal (and wrong) operation on our state vector! For the CNOT-gate at least, this amounts to masking out the lower-right or upper-left square. The resulting gate U can then be scaled using Kronecker products and identity matrices just like any other gate. Below are the calculations for obtaining a CNOT-gate with control-bit 0 and target-bit 2, and the reverse of that.

$$\left(\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \otimes I_2 \otimes I_2 \right) \left(\begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \otimes I_2 \otimes \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \right) = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix} \quad (2.16)$$

$$I_2 \otimes I_2 \otimes \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \otimes I_2 \otimes \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \quad (2.17)$$

Although this is mathematically sound, we have introduced a new problem. As the above results illustrate, the size of the gate grows very quickly as the number of qubits it must be scaled to increases. In fact, the state vector has dimensions 1×2^n , and the corresponding U matrix must have the dimensions $2^n \times 2^n = 2^{2n}$. Holding this huge matrix in memory would be very challenging, and would limit even the best supercomputers to simulating less than 24 qubits¹⁰. In practice, we will see that we can work directly on the state vector by identifying which entries are actually affected by the very sparse U matrix, without actually scaling the gate.

¹⁰As of Spring 2022, the fastest supercomputer in the world is the Japanese 富岳 (Fugaku), which has 5.1 PB of work memory. Assuming that we are representing a complex number as two 64-bit floats, and disregarding the state vector, $2^{25 \cdot 2} \text{B} \approx 18 \text{PB}$ of memory would be required for 25 qubit gates, vs. 4.5 PB for 24 qubit gates.

2.1.5 Pure and Mixed States

A state vector simulator can handle all quantum states which are representable as a state vector. These states are known as pure states. Some quantum systems cannot be represented as simple 2^n state vectors, and instead require a 2^{2n} state *matrix*, known as a density matrix. Such states are known as mixed states. Whereas pure states can only represent points on the surface of the Bloch sphere (See Figure 2.1), mixed states can also be *inside* the sphere. An equivalent density matrix representation of any given pure state $|\psi\rangle$ can be found by calculating the outer product $|\psi\rangle\langle\psi|$.

Fortunately for our aspirations to create a universal quantum circuit simulator based on the state vector representation, it is proven that any mixed state can be purified. Performing an appropriate purification results in an equivalent pure state representation of any mixed state. We won't go into it here, but a proof can be found in Nielsen and Chuang 2011, pp. 110–111.

2.1.6 Measurements

While we do not need to perform formal measurements in order to know the state of our simulated quantum system, we do want to be able to model post-measurement states, in order to accurately model quantum systems. On real quantum hardware, the state is unknowable until measured. While Futhark and other functional programming languages go to great lengths to avoid side effects from an operation, measurements are an example of an operation in the real world where side effects are unavoidable. We can simulate the effects of measurement on a real quantum state by applying a series of measurement operators M_m to a state $|\psi\rangle$. Measurement operators are just possible configurations of the state vector, represented as density matrices instead of state vectors. The small m indicates the outcome represented by the measurement operator. In order to perform a complete measurement, meaning that we are guaranteed to find out what state the system is in, these operators should form an orthonormal basis for the state. As an example, measuring a two-qubit state in the computational basis requires applying four measurement operators; one to represent each possible state. Each comparison returns the probability of that state being measured. So

$$p_{01} = \langle\psi| M_{01}^* M_{01} |\psi\rangle \quad (2.18)$$

is the probability of $|\psi\rangle = |01\rangle$. When measuring a single-qubit system in the computational basis, the set M_m looks like this:

$$M_m = \left\{ M_0 = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}, M_1 = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \right\} \quad (2.19)$$

Let's do a simple example, and measure the maximally entangled state $|\psi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ in the computational basis. This is a 2-qubit state, so we need $2^2 = 4$ measurement operators:

$$M_{00} = |00\rangle \langle 00| \quad (2.20)$$

$$M_{01} = |01\rangle \langle 01| \quad (2.21)$$

$$M_{10} = |10\rangle \langle 10| \quad (2.22)$$

$$M_{11} = |11\rangle \langle 11| \quad (2.23)$$

We start by applying just the first operator, showing everything as standard matrices and vectors to make it clear what is going on:

$$p_{00} = \langle \psi | M_{00}^* M_{00} | \psi \rangle \quad (2.24)$$

$$= \left(\frac{1}{\sqrt{2}} |00\rangle + \frac{1}{\sqrt{2}} |11\rangle \right)^* (|00\rangle \langle 00|)^* (|00\rangle \langle 00|) \left(\frac{1}{\sqrt{2}} |00\rangle + \frac{1}{\sqrt{2}} |11\rangle \right) \quad (2.25)$$

$$= \begin{pmatrix} \frac{1}{\sqrt{2}} & 0 & 0 & \frac{1}{\sqrt{2}} \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ \frac{1}{\sqrt{2}} \end{pmatrix} \quad (2.26)$$

$$= \frac{1}{2} \quad (2.27)$$

$$(2.28)$$

With this explicit example out of the way, we calculate the rest of the probabilities in a more compact manner:

$$p_{00} = \frac{1}{2} \quad (2.29)$$

$$p_{01} = \langle \psi | M_{01}^* M_{01} | \psi \rangle = 0 \quad (2.30)$$

$$p_{10} = \langle \psi | M_{10}^* M_{10} | \psi \rangle = 0 \quad (2.31)$$

$$p_{11} = \langle \psi | M_{11}^* M_{11} | \psi \rangle = \frac{1}{2} \quad (2.32)$$

$$(2.33)$$

In practice, $\langle \psi | M_m^* M_m | \psi \rangle = |\alpha_m|^2$, where α is the probability stored at index m of the state vector. So by just doing 2^n lookups and $O(1)$ calculations per lookup, we can determine the probability of each possible state. Quite a bit faster than applying two 2^{2n} measurement operators to two 2^n vectors!

As alluded to earlier, measuring the system has unavoidable side-effects on the post-measurement state, which can be calculated using this formula¹¹:

$$\frac{M_m | \psi \rangle}{\sqrt{\langle \psi | M_m^* M_m | \psi \rangle}} \quad (2.34)$$

¹¹Nielsen and Chuang 2011, p. 85.

Here, M_m has to be chosen from the set of applied measurement operators. To simulate a real measurement correctly, this should be a weighted choice, based on the previously calculated probabilities of the state measured by a given operator in the set M_m . In this case $\{M_{00}, M_{01}, M_{10}, M_{11}\}$. We note that the contents of the expression inside the square root in the denominator of this fraction is what we just calculated in Equation 2.29. Let $|\phi\rangle$ be the post-measurement state, and let us assume that through an appropriately weighted choice, we have picked the first measurement operator M_{00} :

$$|\phi\rangle = \frac{M_{00} |\psi\rangle}{\sqrt{\langle\psi| M_{00}^* M_{00} |\psi\rangle}} \quad (2.35)$$

$$= \frac{(|00\rangle \langle 00|) (\frac{1}{\sqrt{2}} |00\rangle + \frac{1}{\sqrt{2}} |11\rangle)}{\sqrt{\frac{1}{2}}} \quad (2.36)$$

$$= \frac{\sqrt{\frac{1}{2}} \cdot |00\rangle}{\sqrt{\frac{1}{2}}} = |00\rangle \quad (2.37)$$

To see why this is correct, it may be helpful to take a step back from theory, and consider the situation we are trying to simulate. In the real world, we do not have 2^n physical entities representing the possible states of our n quantum bits; we just have the n physical bits themselves. Exactly like a classical bit, when we actually measure a physical quantum bit, it can only be either on(1) or off(0). We cannot find the probability of measuring a state without recreating and measuring the same state many times. It can also be demonstrated that just like classical bits, barring any further manipulation after measuring, subsequent measurements in the same basis will always yield the same result. With this in mind, it must be the case that, at least from the moment we measure a qubit in a quantum system, the probability of this qubit being in the measured state cannot be anything but 100% or 0%. What this means for our state vector representation is that, following a complete measurement of the system, there can only be a single index with a non-zero value. As the absolute squared sum of the state vector must always be equal to 1, this non-zero value must in fact be 1. So given a complete set of orthonormal basis vectors for the state vector, one of them must always be identical to the collapsed state vector.¹²

It is also possible to take partial measurements. To demonstrate how partial measurements affect the state vector, consider the state:

$$|\psi\rangle = \frac{1}{\sqrt{4}}(|00\rangle + |01\rangle + |10\rangle + |11\rangle) = \frac{1}{\sqrt{4}} \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \quad (2.38)$$

Say that we only perform a measurement on the first qubit in this system. This is equivalent to applying one of the measurement operators $\{M_{00}, M_{01}\}$ to $|\psi\rangle$, and has two equally likely¹³

¹²Nielsen and Chuang 2011.

¹³This doesn't have to be the case. A system $(|\psi\rangle = \frac{1}{\sqrt{8}} |00\rangle + \frac{1}{\sqrt{8}} |01\rangle + \frac{3}{\sqrt{8}} |10\rangle + \frac{3}{\sqrt{8}} |11\rangle)$ would be more likely to end up in the second state after the partial measurement.

possible outcomes for the state of the example system post-measurement:

$$|\phi\rangle = \frac{M_m |\psi\rangle}{\sqrt{\langle\psi| M_m^* M_m |\psi\rangle}} \Rightarrow \begin{cases} \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} & \text{iff } m = 00 \\ \frac{1}{\sqrt{2}} \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} & \text{iff } m = 01 \end{cases} \quad (2.39)$$

If this seems odd, remember that the values of a the state vector representing a multi-qubit state refer not to the probability of a single qubit being on, but the probability of a combination of qubits being either on or off. As we haven't measured the second qubit, in the real world we cannot know what state it is in. However, based on the results of measuring the first qubit, we definitely know what state *that* is in, which affects the probabilities of every measurement represented by the state vector.

There are optimizations which become possible to perform when we don't have to care about the entire state vector, because we know from the beginning that we are only interested in measuring the probability of a certain state, or in knowing whether a specific qubit is on or not. Closer examinations of these go beyond the scope of this project.

What we aim to do is simply simulate the evolution of a state vector over time, subject to the same changes a real quantum system can go through and return a correctly modified state to the user in the end.

2.1.7 Identifying Efficiently Simulateable Quantum Circuits

The Gottesman-Knill theorem¹⁴ states that any quantum circuit which is representable as a combination of Phase(A generalized form of the Pauli-gates), Hadamard and CNOT-gates¹⁵, and measurements in the computational basis, can be simulated efficiently on a classical computer. These gates are collectively referred to as the Clifford-group. Many interesting quantum circuits can be written to only use these three gates, combined with measurements in the computational basis. Even one of the poster children for what makes quantum computing so unique, the maximally entangled state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$, is reachable utilizing only this gate-set, and meaningful results can be obtained by measuring states in the computational basis.

Fortunately for the builders of actual quantum computers, it turns out that this gate set is simply not sufficient for simulating all quantum computations. If combined with the T-gate(Table 2.1.3), the set *does* become a basis for universal quantum computing, but it will no longer be efficient to simulate on classical hardware.

¹⁴Gottesman 1998.

¹⁵DiVincenzo 1995.

This doesn't mean that our state vector simulator will operate more efficiently faced with the right circuit. Instead, what is required is a specialized simulator, made only to run circuits consisting of these gates.

This type of simulator tracks not the actual qubit states, but instead the state of at most $2n$ operators, so two for each qubits, which are modified depending on the gate being applied in each step.¹⁶

As our goal was to write a universal simulator, and these operations do not form a complete basis for quantum computations, we chose not to go down this path.

Nevertheless, for circuits which *can* be described with this limited set of operations, significant performance gains can be achieved. According to Gottesman 1998, storing an n -qubit state using this systems takes only $2n + 1$ bits, and actually executing the circuit can be done in polynomial time.

2.2 Quantum Fourier Transform (QFT)

What problems might then be interesting to solve with the might of quantum computing? Instead of looking too much into the forefront of this field, we instead briefly explore an algorithm that is a part of many quantum algorithms, the Quantum Fourier Transform (QFT).

The quantum fourier transform is equivalent to the classical fourier transform, but much faster. It is used in algorithms such as Shor's factoring algorithm and quantum phase estimation.

The QFT transforms a vector from the computational basis $|0\rangle$ and $|1\rangle$, to the Fourier basis, which corresponds to $|+\rangle$ and $|-\rangle$.

This is an alternate encoding of numbers, which turns out to be useful for solving many important numerical problems faster. Instead of flipping the qubits using CNOT like we would like using XOR and classical bits, values in the Fourier-basis are encoded using parameterized controlled rotation-gates, which move the state around the axis in-between the computational basis states. The angle by which the state is rotated to encode a number is different for each qubit in the system. For the most basic case of just one qubit, the state is rotated 180 degrees around the axis for each time the number stored is incremented, which is really no different from the computational basis. Things change when encoding numbers using more than one qubit. The second qubit in such a system is rotated 90 degrees per incrementation, the third 45, the fourth 22.5, and so on. To encode numbers up to 3, two qubits are needed. Three are needed to encode up to 7, four to encode up to 15, and so on, just like when encoding in the computational basis. The result of applying this function on an orthonormal basis $|0\rangle$ is the complex vector $|k\rangle$ ¹⁷. Formally, this transformation can be described by $|j\rangle \longrightarrow \frac{1}{\sqrt{n}} \sum_{k=0}^{N-1} e^{2\pi i j k / N} |k\rangle$, where $|j\rangle$ is a vector encoding the number we want to transform in the computational basis.

This looks very complicated, but it is fairly straight-forward to represent as a circuit-diagram:

¹⁶Gottesman 1998, p. 17.

¹⁷Nielsen and Chuang 2011, p. 217.

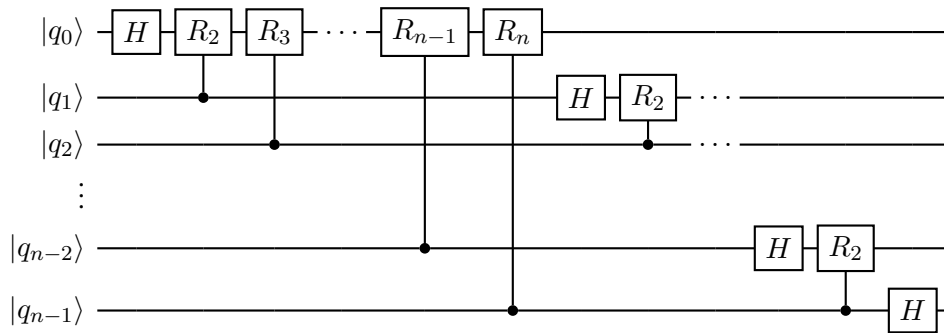


Figure 2.3: A general Quantum Fourier Transform for n qubits.

The code to execute a QFT-transformation in `src/kuthark/test/test_job.py` looks a bit different to this diagram. This is because Qiskit uses a different ordering of qubit significance. See section 3.4.4 for details.

Chapter 3

Method

In this chapter we discuss the implementation and general strategy for our quantum circuit simulator called Quthark, a fusion of the words quantum and Futhark, our programming language of choice.

There are many types of quantum circuit simulators, but we have chosen to implement what is known as a state vector simulator. While other types of simulators can simulate states consisting of significantly more qubits much faster than state vector simulators, they all have their own issues we won't go into here. We chose to implement a this type of simulator because we found it to strike a good balance between being able to execute any quantum computation, while still supporting a reasonably sized state.

3.1 Futhark

We have written our implementation in the data-parallel functional programming language Futhark. Futhark aims to make it trivial to write code that compiles to a form that can be efficiently executed by processors with a high level of parallelism, such as GPUs. The Futhark language does not allow the programmer to perform the same low-level optimizations as would be possible in a language such as CUDA or OpenCL, but instead aims to generate CUDA and OpenCL code which performs very well on average. It can be considered a high-level language specifically designed for writing massively multi-threaded code, and is intended to be used to write libraries that speed up the most intensive operations performed by an application. Code for which exploiting parallelism is less interesting is usually written in some other, high-level general-purpose language, such as Python.

This makes Futhark a great choice for writing a state vector simulator, as the most labor-intensive part of execution is applying the gates to the state vector, which is embarrassingly parallel.

Code written in Futhark can be compiled to run on all widely-available GPUs.

Futhark being a relatively young language does mean that some things which one has come to expect to just work sometimes required unexpected effort. In our case, the standard library for complex numbers was missing significant operations, such as a function for calculating the absolute value of a complex number, or taking it to a power. We implemented these and submitted

a pull-request, which was accepted.

All of our results are from using Futhark version 0.21.14. This version has known compiler-bugs, one of which did cause us some issues when trying to implement gate-fusion, but later versions had even worse problems with leaking memory, so we stuck to what worked most of the time.

As a domain-specific language, Futhark is intended to be used solely for the most intensely parallel operations. This is why it has been written to easily integrate into code written in other, more general-purpose languages. One of those is Python, which is what the front-end of the Qiskit Software Development Kit is written in.

3.2 Qiskit

Qiskit is a free and open source quantum circuit and computing library developed by IBM. It allows for interfacing between real quantum computers and simulators, seamlessly interchanging the two when building and running circuits. Circuits can be defined using a number of interfaces. These interfaces range from a visual interface accessible through a web browser, to the Python interface we have chosen to work with. The interface always presents the user with the exact same experience, regardless of the machine they execute the circuit on. Whether it is a quantum circuit simulator written in Futhark running on a GPU, or an actual quantum computer in a lab in Frankfurt - the experience is mostly the same to the end user.

The first public version of Qiskit was released in 2017. Since then, it has been under active development, and today supports a number of advanced quality-of-life features. Most importantly for us is the ability to easily create circuits as seen in Listing 1, and then being able to run them on different backends.

```

1 qc = QuantumCircuit(3)
2 qc.y(0); qc.swap(1,2)
3 qc.t(0); qc.z(1)
4 qc.t(2); qc.y(0)
5 qc.s(1); qc.x(2)

```

Listing 1: Three qubit quantum circuit, built with the Qiskit API, generating the circuit seen in Figure 3.2

Qiskit provides a multitude of different backends, each with their own benefits and drawbacks. But as we are building a state vector simulator, we only focus on comparing with the equivalent Qiskit backend; the Aer state vector simulator.

The Aer state vector simulator Qiskit Aer is a state-of-the-art quantum circuit simulator library which we chose to use for validation and performance comparison, as it is one of the fastest publicly available state vector simulators¹. Its state vector simulator consists of more

¹Suzuki *et al.* 2021.

than 10.000 lines of code, and it is written to take advantage of both CPUs and GPUs, allowing us to make meaningful comparisons with Quthark.

To keep things simple, when we refer to Aer after this point, we are referring to the Aer state vector simulator specifically.

For Aer's performance to be comparable to Quthark's, it needs specifically an Nvidia GPU. This is because Aer's GPU-backend is written using the Nvidia-specific CUDA-framework. Currently, Nvidia GPUs are among the best on the market, but it is still a limitation of Aer that means exploiting theoretical better-performing GPUs from another manufacturer in the future would require a complete rewrite of the GPU-specific functions.

3.2.1 Integrating with Qiskit

In order to actually use the Aer simulator and be able to make use of the well-developed, free and open source front end for our simulator, we had to integrate Quthark with Qiskit's Python API as a backend.

Qiskit's own simulators are implemented in C++, Python and the C++-compatible API CUDA. The Qiskit frontend generates an object, which it passes on to a specified backend. To use the values contained within this object, we first need to manually extract them and put them into arrays, which Futhark can actually work with. Parsing this structure to isolate the values we need was not trivial, and this is just one example of extra work needed because we chose to make Quthark compatible with Qiskit. Understanding and conforming to the API, and creating proper classes for adequate integration was a time consuming task, but valued to be worth it. Ultimately, it amounted to just over 500 lines of Python code in its final form.

3.3 The State Vector Simulator

A state vector simulator has some major advantages compared to a real quantum computer. First of all, it is precise. Though limited by the number of values representable using two 64-bit floats, this is still far better than how precisely we can determine the state of a real quantum computer. Environmental noise, and the need to execute circuits a huge number of times to accurately estimate the actual state, are going to be obstacles for actual quantum computers for the foreseeable future.

It is also not affected by environmental noise in any real sense. A real quantum computer is limited to a certain number of operations before the state of the qubit becomes incoherent². This can be alleviated by implementing quantum error correction³, but these corrections are computationally expensive, and have their limits, at least with current hardware.

Another major advantage is that the coherence of the system is not affected by the number of qubits it consists of. In the real world, the more physical qubits you have as part of your quantum computer, the shorter the span of time you can expect your qubits to remain in a coherent state.⁴

²Nielsen and Chuang 2011, p. 278.

³Nielsen and Chuang 2011, p. 453.

⁴Tannu and Qureshi 2019.

Faced with all of these advantages, it might seem that simulating a quantum circuit on classical hardware is superior to executing it on an actual quantum device, but this is not the case. First of all, quantum computers are faster. Despite needing to repeat experiments many times, good-enough results can be reached much faster in most cases. This is in part because a quantum computer can actually apply separate gates to each qubit simultaneously, and in part because it doesn't actually need to keep track of a state consisting of 2^n values.

This is the primary advantage of actual quantum hardware. Whereas even the best classical supercomputers can barely handle accurately simulating a 50 qubit state today, we already have actual quantum computers which can handle > 100 qubit states. They are expensive, imprecise, and extremely limited in number, but both availability and precision is only expected to improve.

There are two main reasons that we still find it meaningful to build quantum circuit simulators. First of all, actual quantum hardware powerful enough to handle qubit states larger than those we can simulate today are not accessible to the broader public. While this situation is expected to improve over time, quantum computers are in many ways black boxes. Since we cannot inspect the state during computation without affecting it, debugging quantum algorithms can be challenging, and verifying that a quantum computer gives correct results would be difficult to do by hand. State vector simulators allow users to inspect the state of a quantum system at any time, making for much easier development, and trivializing theoretical verification of results obtained on actual quantum hardware. So long as a meaningful version of the circuit can be simulated classically in practice.

3.4 Implementation

In this section, we introduce the process that lead to the final version of Quthark. During development, our simulator underwent three major transformations.

Quthark v0 was implemented in a completely naive manner, blindly applying the theoretically operations as presented in the literature. Just as described in subsection 2.1.2, gates were scaled using the Kronecker product, and applied to the state vector using matrix multiplication. As we expected the memory usage made the simulator infeasible and we quickly discontinued development, to instead focus on implementing more viable options. We don't discuss this implementation further, as it can be completely deduced from the theory section.

Quthark v1 moved beyond scaling gates, addressing the memory issues of v0, instead carefully applying the gate to the specific indices of the state vector. This approach saved significant memory, and improved performance, as described in subsection 3.4.1. This is also the most generic simulator, and application of custom unitary gates is performed in the same way in v2 and v3.

Quthark v2 got rid of the gate matrix representations entirely for our set of supported standard gates. Instead, each gate was transformed to a unique, optimized function, reducing the number of operations performed when applying a gates to the minimum number possible. This also

had the aim of giving the compiler more information about each gate, theoretically allowing it to better optimize the compiled code. V2 is our fastest simulator, but not the most memory efficient.

Quthark v3 was developed in an attempt to reduce memory usage while improving performance, by utilizing cache-blocking to dynamically optimize cache-utilization based on the target qubit(s) of each gate. This did reduce Quthark's memory usage to approximately the same level as comparable simulators, but at the cost of significant performance. V3 is our most memory efficient simulator, but only slightly faster than v1.

Other optimizations were considered, such as a sparse representation of the state vector, compiling circuit-specific Futhark code just-in-time, and fusing gates to reduce the number of times the state vector needs to be accessed. Some were implemented and found to result in nothing but worse performance, some were found to be theoretically uninteresting, and some we just did not have the time to implement.

Our attempts at optimization, successful or not, are documented in the following sections, where we try to introduce the core concepts of the different simulators one by one.

3.4.1 Qubit Gates

First of we need to get an idea of how we applied a single qubit gate for v1 of Quthark. As described, naively implementing the textbook calculations becomes too expensive in memory, and is not as effective as could be. Scaling a gate with the Kronecker product and identity matrix just copies a lot of values, and it is easily avoidable if we instead *carefully* perform each multiplication.

We found that when applying a gate U to the k 'th qubit, not all amplitudes from $|\psi\rangle$ are needed in order to update another amplitude in the vector. In fact only a pair of two amplitudes are needed for each amplitude update, or in general a group of 2^m amplitudes, where m is the size of the gate. Intuitively this is also the case for the gate that is being applied; only one row of values is needed. So in order to apply a single qubit gate to the state vector, all we need to do is:

1. Go through each entry
2. Find its pair
3. Multiply each part of the pair with its corresponding gate row entry
4. Sum the two products
5. Update the entry with the newly calculated sum

Finding the pairwise amplitudes is best visualized in base 2, as the paired value for each entry is determined by the target bit k . It is then a matter of flipping the k 'th bit, so if $k = 3$, then

$17 = 10001_2$ and $21 = 10101_2$ would be a pair. In figure 3.1 the procedure for applying a gate on both the first and second qubit respectively is visualized, and we can now reason that each pair must be 2^k apart from each other.

$$|\psi\rangle = \begin{pmatrix} \alpha_{00} \\ \alpha_{01} \\ \alpha_{10} \\ \alpha_{11} \end{pmatrix}$$

$\alpha'_{00} = u_{00} \cdot \alpha_{00} + u_{01} \cdot \alpha_{10}$ $\alpha'_{10} = u_{10} \cdot \alpha_{00} + u_{11} \cdot \alpha_{10}$	$\alpha'_{00} = u_{00} \cdot \alpha_{00} + u_{01} \cdot \alpha_{01}$ $\alpha'_{01} = u_{10} \cdot \alpha_{00} + u_{11} \cdot \alpha_{01}$
$\alpha'_{01} = u_{00} \cdot \alpha_{01} + u_{01} \cdot \alpha_{11}$ $\alpha'_{11} = u_{10} \cdot \alpha_{01} + u_{11} \cdot \alpha_{11}$	$\alpha'_{10} = u_{00} \cdot \alpha_{10} + u_{01} \cdot \alpha_{11}$ $\alpha'_{11} = u_{10} \cdot \alpha_{10} + u_{11} \cdot \alpha_{11}$

Figure 3.1: Two qubit example of applying a single qubit gate to the first and second qubit respectively.

A more general formula for updating pairs of amplitudes can be written for an arbitrary number of qubits like so:

$$\begin{aligned} \alpha'_{*...*0_k*...*} &= u_{00} \cdot \alpha_{*...*0_k*...*} + u_{01} \cdot \alpha_{*...*1_k*...*} \\ \alpha'_{*...*1_k*...*} &= u_{10} \cdot \alpha_{*...*0_k*...*} + u_{11} \cdot \alpha_{*...*1_k*...*} \end{aligned} \quad (3.1)$$

It is important to note that each amplitude must be updated with its pair *immediately*, as the original value is needed for both calculations.

This new approach significantly reduces the amount of memory needed, as we apply the gate directly to the state vector, without wasting memory on scaling it first.

With this algorithm, the aforementioned supercomputer would now be able to handle up to 48 qubits⁵. Perhaps more importantly, this theoretically takes a typical modern laptop with 16 GB of RAM from being able to handle a 14 qubit state, to being capable of processing up to a 29 qubit state, allowing for the simulation of significantly more complex algorithms.⁶

Implementing Equation 3.1 in an imperative language is not that difficult, as we can split the equation into an inner and outer loop. The outer loop moves in strides of 2^{k+1} for 2^{n-k-1}

⁵ $2^{48}\text{B} \approx 4.5\text{PB}$, which is just the size of the state vector. The gate is now insignificant, as it is not scaled.

⁶29 qubits would require $2^{29} \cdot 2 \cdot 64 \approx 8.6\text{ GB}$

iterations, while the inner loop iterates over 2^k pairs. The pseudocode for this can be seen in Algorithm 1.

Algorithm 1 Imperative single qubit gate operation pseudocode.

```

for  $i \leftarrow 0; i < 2^n; i \leftarrow i + 2^{k+1}$  do
  for  $j \leftarrow i; j < i + 2^k; j \leftarrow j + 1$  do
     $\alpha'_j \leftarrow u_{00} \cdot \alpha_j + u_{01} \cdot \alpha_{j+2^k}$ 
     $\alpha'_{j+2^k} \leftarrow u_{10} \cdot \alpha_j + u_{11} \cdot \alpha_{j+2^k}$ 
  end for
end for

```

While it is somewhat more straightforward to understand the imperative version, we have (as described in section 3.1) chosen to write our implementation in Futhark. A direct translation of the algorithm is not an option as it employs side effects which can't be implemented in a pure functional language. Instead we flatten the two loops, so that we can map over the entirety of the state vector. This way we also utilize Futhark's built in array bulk operators, called Second-Order Array Combinators (SOACs), that allow us to semantically write sequential code but allow for parallel execution.

Doing so does come with the drawback of having to do more work, as we now map over all 2^n amplitudes instead of half of them, and have to find the pair for *each* entry. In the imperative version we only had to find the pair once, as we updated the vector in place. But we have to do this twice, each for their own, and it becomes even worse when we apply multi qubit gates. We suddenly go from amplitude pairs, to groups of 2^m . The imperative version only had to loop through $2^n \cdot \frac{1}{2^m}$ values, but we continue to map over everything. The amount of work for the imperative implementation is of course the same, but we scale worse with m .

Luckily most gates are single qubit gates, and rarely exceed three qubits, but the time difference between the two proved to be significant, as we measured missed cache lookups to be one of the most time consuming workloads within our map function. Unfortunately it is not trivial to implement a pattern in Futhark that combats this problem. We explore an attempt to do so in subsection 3.7.4, but end up coming to the conclusion that the compiler can't parallelize and fuse the SOACs that we use efficiently the way we have written it.

We also explore another solution in subsection 3.6.3, where we make sure that the cache is loaded with the appropriate pairs attempting to minimize the repeated workload, but end up with a similar result as in subsection 3.7.4.

Apart from the flattening of the for-loop the algorithm is the same, and can easily be generalized to support multi qubit gates.

Gates as functions

When inspecting the gates introduced in the theory section, it is clear that a lot of the multiplications that are happening between the amplitude and the entrances in the gates are pointless. The gates contain mostly 1s and 0s, and so only very specific places in the state vector are actually being affected. So instead of wasting resources on executing these meaningless multiplications, we can represent the gate as a function instead of a matrix. These can be much more efficient

to apply than generic matrices, which are required to be of some regular shape. An example of the CNOT gate written as a function can be seen in Listing 2. Recall that it is very sparse, and ultimately just has to flip the value of two amplitudes.

```

91 def cnot_gate_f [nn] (state: [nn]complex) (startidx: i64) (gate_row: i64)
    ↪ (bitperms: []i64) (idx: i64) : complex =
92     match gate_row
93     case 1 -> state[startidx ^ bitperms[3]]
94     case 3 -> state[startidx ^ bitperms[1]]
95     case _ -> state[idx]
```

Listing 2: CNOT gate written as a function.

3.4.2 Key Function Definitions

The file `src/qiskit_kuthark/backend.py` contains the code which actually parses the data structure we get from the Qiskit API. We then pass the extracted values from Python to the function `main` in `src/kuthark/backend.fut`. It reorganizes the data received from Python into something more manageable, and pass it on to `python_entry_point` in `kuthark.fut`:

```

148 def python_entry_point [d][o][p]
149 (n: i64) (circuit: [d] action)
150 (ts: [p]i64) (cgates: [o]complex): ([f64], [f64]) =
151     let initial_state = create_state n
152     let state = loop state = initial_state for c in circuit do
153         apply_action state c ts cgates
154     in unzip state
```

Listing 3: Function `python_entry_point` in `kuthark.fut`

We start with a brief description of our input-variables:

`n` Describes the number of qubits in the quantum state.

`circuit` Is an array of records, with most importantly the gate id's and their parameters.

`ts` The qubit-targets for each gate.

`cgates` All custom unitary gates as a 1D-array.

Each entry in the circuit is then evaluated in the sequential for-loop together with the shared data-structures, each time updating the state vector and passing it on to the next action in the circuit.

3.4.3 Measurements

Measurements are implemented much like single-qubit gates. In fact, the same code is re-used for most of the process. The major difference is that we need to go through the state vector twice, once to calculate each side of the divisor in Equation 2.34. We then combine the results following that equation, which gives us the expected state. It is possible to perform both partial and complete measurements of a state.

We chose not to implement actually returning the probability of measuring a given state, since it wasn't interesting to parallelize. While it isn't quite as user-friendly as writing something custom, one can determine each probability simply by calculating the absolute squared value of the complex number representing the result in the state vector one is interested in.

We only implemented measurements in the computational basis, but adding more is about as easy as adding any other gate to Quthark. See section 3.4.4 for details.

While implementing measurements in any given basis is fairly trivial in Quthark, Qiskit only allows measurements in the computational basis⁷. Adding support for measuring in other bases to Qiskit would require getting familiar with a codebase consisting of tens of thousands of lines of C++ alone, and falls outside the scope of this project.

3.4.4 Supported Gates

We chose to support the full set of gates implemented exposed by the Qiskit API⁸. This allows us to run any circuit generated by the built-in random circuit generator without a transpiling step, but does mean that optimizing gate-application becomes a lot more challenging. This also means supporting custom unitary gates defined by the user. This provided some unique challenges when working in Futhark, but ultimately turned out to be very performant.

Gate format

In most textbooks and also in our report, controlled gates are presented in matrix form with the most significant qubits as the control, Qiskit uses the least significant bit as the control. Thus there is a discrepancy between the matrices that we present in the report and the one we implemented for v0/v1 and optimized for v2. As we use Qiskit Aer for property based testing, we of course conform with their implementation, even though we find the gates easier to understand and explain in the "standard" representation.

Custom gates

We wanted to support not just parameterized standard gates, but also completely user-defined custom unitary gates. This meant getting Futhark to accept a data structure containing a number of matrices of varying shape and size as input.

⁷As of Qiskit version 0.37.1

⁸ANIS *et al.* 2021a.

This could not be a tuple, as that would require us to always send in a specific number of matrices of a certain size. It also could not be an array of 2D-arrays, since that would be considered an irregular 3D-array by Futhark, and irregular arrays are disallowed.

The solution we came up with was to flatten the custom gates and store them in a single 1D-array. Combined with information about where each custom gate starts, as well as how many qubits it should apply to, we can then isolate and reshape the gate to its original shape.

3.4.5 Improving on Theoretical Performance

In theory, applying a two-qubit gate requires $2^{2 \cdot 2} = 16$ operations when applied to a 2-qubit state, as each entry in the four-by-four matrix can potentially affect the result. However, that is not always the case. At least in the case of the gate set defined by the Qiskit API, most gates are extremely sparse, and predictably so. For instance, we know that a controlled gate always has a part that does absolutely nothing! Consider CNOT⁹ from our earlier example. One way of viewing it is that it only acts when the control-qubit is set to something other than 0. But if you view it with your more traditional linear algebra glasses on, it is clear that all it does is swap two indices, and that half of it is just the identity matrix with some extra zeroes!

In other words, when applying CNOT to a 2-qubit state, it suffices to carry out a simple index-swap. But we can go even further than that. If the indices being swapped have never been written to before, we know that they must both be zero, and we can avoid applying the gate altogether, saving us a full trip through the state vector.

While CNOT might appear to be an extreme example, controlled gates are commonplace in quantum computing, and optimizing for them can lead to meaningful speedups for both randomized circuits, *and* popular algorithms.

As the state vector grows, the number of calculations needed to apply a gate grows with it. So if a superficial generalized optimization of controlled gates saves 8 operations when applying a 2-qubit controlled gate to a 2-qubit state, it will save 16 operations when applying to a 3-qubit state, and a substantial 536 million operations when applying it to a 30-qubit state!

More generally, as gates are usually extremely sparse, a simple short-circuit of the gate application, skipping looking up the values in the state vector and doing the calculation of new values when they're going to be 0 regardless, saves several seconds when working with large state vectors.

3.5 Benchmarking

Hardware

In order to perform experiments on larger state vectors in a repeatable and practical manner, we were given access to "The Futhark cluster". This is a cluster of servers dedicated to Futhark-projects, with modern GPUs such as the Nvidia A100 40GB; our GPU of choice for all benchmarks throughout this project.

⁹ Equation 2.1.4

Methodology

We chose to primarily compare performance between simulators executing random circuits, based primarily on two major factors:

1. We did not find it reasonable to think that we would have time to be able to implement algorithm-specific optimizations, while also writing a fast generic simulator, within the scope of this project.
2. As quantum computing is still in its early stages, we cannot reasonably guess at what kind of algorithms will be interesting to run on these machines. There *are* a few core algorithms which are interesting now, and we did do benchmarks for the widely applicable Quantum Fourier Transform.

We based decisions such as not moving to a sparse state vector representation on the result of benchmarking on random circuits. It may be that we would have come to a different conclusion if we had instead written a test-suite based on popular quantum algorithms.

We do not benchmark circuits with measurements. As described in subsection 3.4.3, measurement application is basically just two gate applications. Additionally, no example circuit we could find performed a meaningful number of measurements compared to normal gate applications, with most measuring each qubit in the state at most once. On this basis, we concluded that the performance of our measurement implementation would be completely insignificant at the circuit lengths we were testing on.

Execution time

When comparing Quthark with Aer, we found that circuits with a size of 1000 were long enough enough to show the trends in the performance of each simulator. Larger circuits simply added a corresponding factor to the execution time, i.e. size 2000 took twice as long, while going lower than 1000 put more emphasis on the general overhead and added more fluctuations. As we did not have exclusive access to the GPU that we utilized, we also did multiple runs for each experiment. We found that removing outliers and averaging over ten runs gave us repeatable results. The exact configurations can be seen in section 3.9.

One problem we did run into when running our code, was the fact that we used a shared GPU. As previously established, getting good performance is quite dependant on the whole cache being available to us. We experienced massive spikes in our run time when other clients performed their experiments, which we did try to work around, but some experiments were simply too long-winded for us to wait for.

3.5.1 Circuit Depth vs Size

As described in subsection 2.1.3, a circuit consists of a number of gates. The more gates, the more calculations needs to be done, as each gate needs to be applied to the state. But when working with an actual quantum computer the *depth* is often the word used to describe the complexity of a circuit, as some gates can be applied simultaneously on real quantum hardware.

So if two gates are implemented, and applied on different qubits, those two gates can be applied at the same time, at no extra cost. This can make two circuits with different number of gates of equal depth, as they would take the same amount of time to run on quantum hardware. However this is not the case when simulating, not because the gates can not be applied in parallel (although also the case), but because there is more work to be done. We assume that we use up all the hardware available to us for applying just one gate, and thus trying to apply two gates at once vs one at a time would effectively take the same amount of time. As a result, when simulating a quantum circuit on classical hardware, each gate adds another level to what we call the *size* of the circuit, so running a circuit of size n is just another way of saying n gates were applied to the state vector. In Figure 3.2 the concept of size and depth is illustrated with a random generated three qubit circuit with size eight and depth three.

3.5.2 Generating a Test Case

The field of quantum computation is far from fixed. New, more efficient algorithms are constantly emerging, and this process is only likely to accelerate in the coming years, as actual quantum hardware becomes increasingly available to end-users.

With this in mind, we decided not to focus on optimising some subset of gates which are useful for some algorithm(s) in use today, but instead on optimising a wide range of gates, and using a random circuit to compare performance.

We utilize Qiskits random circuit generator¹⁰ for producing a circuit with a *depth* of a given number, we subsequently reduce this to a given *size*, which is then applied to an n qubit vector with the initial state of $|0\rangle^{\otimes n}$. Aer does however not support the full set of gates that a random circuit is generated from, so when doing comparison with Aer, we first had the Qiskit transpiler convert the circuit to a form consisting only of their support base gate set.

3.5.3 Correctness

When developing the Quthark quantum circuit simulator, we aimed for speed. However, speed is obviously an easy feat if we don't care about correctness. For good benchmarks to be meaningful, we had to thoroughly verify that our implementation returned the expected deterministic result. This is not easily done, as applying a gate doesn't necessarily change anything, if the state, gate, and target are a particularly boring combination. As an example: applying a Pauli z-gate on qubit 0 with state $|0\rangle$, would yield the same state $|0\rangle$. For simple gates, like the Pauli-gates, we did create some specific states and targets which would yield more interesting results, which we could calculate by hand and run unit tests on. But as we have implemented more than 30 different gates, where some are parameterized, and also support user-defined unitary gates, we found that it was an extremely time consuming task to write tests that would provide coverage that we found reliable and trustworthy. Instead of attempting this, we put our trust in the correctness of the implementation of Aer, and tested our implementation against theirs, employing

¹⁰ANIS *et al.* 2021b.

a property-based testing methodology¹¹. By executing deep¹² randomly generated circuits, we found that the state vectors were sufficiently 'interesting' for testing equality between Quthark and Aer.

3.5.4 Complex number representation

Futhark represents complex numbers slightly differently from Numpy, the Python library we use to handle matrices and vectors. This means that we have to spend a not-insignificant amount of time¹³ converting to the appropriate representation before presenting the result to the user. This overhead is removed from our benchmarks, as fixing it should be possible by altering Futhark's complex number representation to match Numpy's. *Our automated tests can be executed with the instruction `make test`.*

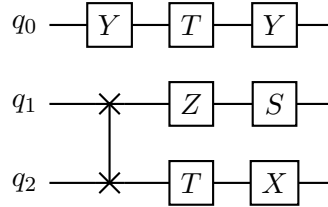


Figure 3.2: A three qubit quantum circuit with depth three and size eight. The squares with capital letters are gates labelled as in subsection 3.4.4. The two crosses connected by a line is a SWAP-gate.

3.6 Results

In this section we present the final results of executing our state vector simulator.

When looking at the plot in Figure 3.3 we can clearly see all three simulators flatlining until around 20 qubits. One would expect circuit-executions to complete exponentially slower, and thus going in a straight line on the logarithmic scale of the plot, which is not the case. This is a result of the GPU being under-utilized. Each gate needs to be applied in a sequential order, and the number of threads needed to update the small states we are applying each gate to simply cannot saturate the number of GPU-threads we have available. We also see that Aer, is quite significantly faster, especially compared to Quthark v1, being twice as fast at 29 qubits, Aer running in 9.9 seconds, vs Quthark v1 in 20.4 seconds. Even when the GPU is not saturated,

¹¹A large number of correct, random inputs are fed to a function. Done right, this will reveal any bugs serious enough to cause a crash. If one is able to compare the results to a known-good implementation, property-based testing can also be used to verify results. This is a black box testing methodology, where we do not concern ourselves with what is actually happening inside the function, just that it returns the correct result.

¹²Somewhere between 100 and 200 random gates applied to a 10-qubit state. The exact number of gates varies due to how Qiskit generates circuits, but it is always ≥ 100 .

¹³About 3 seconds for a 29-qubit state, half of that for a 28-qubit state, and so on.

Aer maintains a significant lead. This seems to indicate that there must be some fixed overhead we incur that Aer does not. Either it is a limitation of Futhark, or something we do that is unexpectedly slow. We investigated, but were unable to find anything in our own code, leading us to believe that it must be outside our control. While Aer consistently outperforms even our fastest simulator, the graph does showcase the immense improvements we gained from optimizing each our gate set when converting them to functions, as that was the only major difference Quthark v1 and v2.

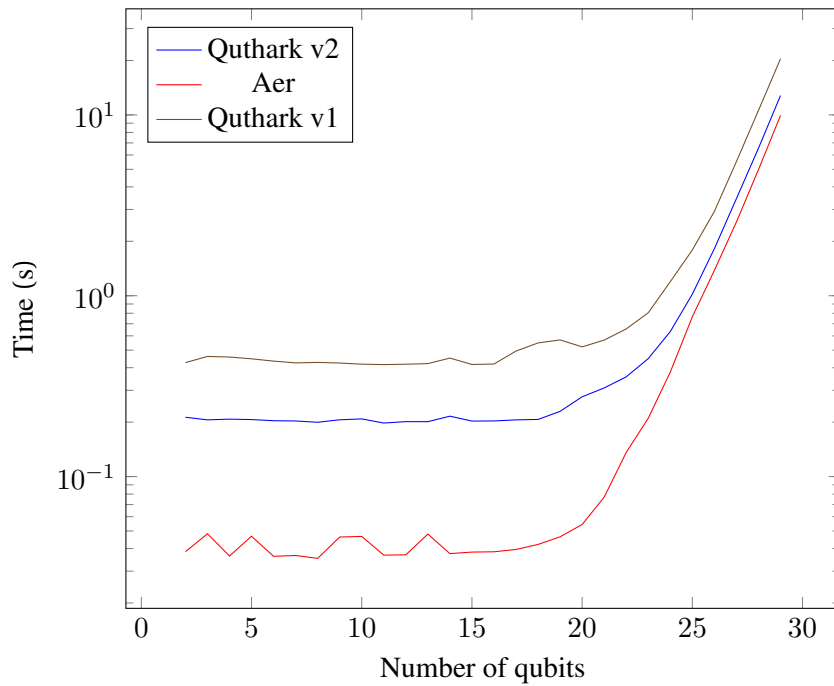


Figure 3.3: Execution time of a size 1000 circuit run on differently sized state vectors.

QUBITS	QUTHARK v1	QUTHARK v2	AER
17	$4.94 \cdot 10^{-1}$	$2.06 \cdot 10^{-1}$	$3.95 \cdot 10^{-2}$
18	$5.48 \cdot 10^{-1}$	$2.07 \cdot 10^{-1}$	$4.21 \cdot 10^{-2}$
19	$5.7 \cdot 10^{-1}$	$2.29 \cdot 10^{-1}$	$4.64 \cdot 10^{-2}$
20	$5.21 \cdot 10^{-1}$	$2.76 \cdot 10^{-1}$	$5.43 \cdot 10^{-2}$
21	$5.68 \cdot 10^{-1}$	$3.09 \cdot 10^{-1}$	$7.7 \cdot 10^{-2}$
22	$6.55 \cdot 10^{-1}$	$3.56 \cdot 10^{-1}$	$1.36 \cdot 10^{-1}$
23	$8.05 \cdot 10^{-1}$	$4.49 \cdot 10^{-1}$	$2.1 \cdot 10^{-1}$
24	$1.2 \cdot 10^0$	$6.34 \cdot 10^{-1}$	$3.78 \cdot 10^{-1}$
25	$1.79 \cdot 10^0$	$1.02 \cdot 10^0$	$7.65 \cdot 10^{-1}$
26	$2.93 \cdot 10^0$	$1.82 \cdot 10^0$	$1.38 \cdot 10^0$
27	$5.52 \cdot 10^0$	$3.44 \cdot 10^0$	$2.56 \cdot 10^0$
28	$1.06 \cdot 10^1$	$6.54 \cdot 10^0$	$4.98 \cdot 10^0$
29	$2.04 \cdot 10^1$	$1.28 \cdot 10^1$	$9.92 \cdot 10^0$

Table 3.1: Execution time (s) of Aer, Quthark v1 and v2 for a randomized circuit of size 1000, limited to the range 17 to 29 qubits for brevity.

3.6.1 Memory Usage

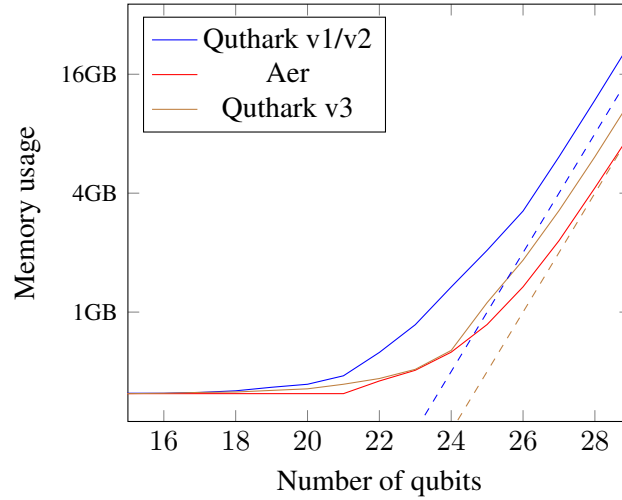
Futhark, as a functional programming language, does not allow for side-effects. What this means is that any single line of code will return some output and that this output is the only effect this line of code has. In other words, if we execute a line of code that takes as input some number and returns a different number, the original number still exists as it was before execution; multiplying a by two does not erase the original value of a . This sounds like it would be an awful waste of memory, but usually, it isn't a problem. The Futhark compiler is smart enough to get rid of data when it is no longer needed. When executing a map to modify some array, such as our state vector, it is even possible for the compiler to detect operations that are isolated to the index which they read from, letting it avoid creating an output array of the same size, and instead modifying the array in-place. When it does become a problem is when the compiler cannot determine that this is the case, which is what happens in our code when updating the state vector. In reality, we know that it is possible to modify the structure in-place. However, because this is not detectable by the compiler, and we have no way to relay our insight to it, we have to allocate space for two instances of the state vector at all times. We got around this somewhat by implementing cache-blocking, but it would be nice to just be able to promise the compiler that a specific SOAC will never write to something before it is read for the last time.

As previously described in section subsection 2.1.2, the theoretical memory usage of storing the state vector on classical hardware is exponential in the number of qubits represented by it. As we aim at achieving the same precision as Aer, we use 2×64 bit floating numbers for our complex amplitudes. So disregarding the general overhead, an in-place update of the state vector would require $2^n \cdot 2 \cdot 64$ bits. However, Futhark does not allow in-place updates when using its SOACs, as we have no way of manually assuring the compiler that we will not be overwriting indices which could be read from again at a later point. The compiler is supposed to be able to

figure this out itself in a soon-to-be-released version.

Q	v1	v3	AER
16	417	417	415
17	421	421	415
18	429	423	415
19	447	431	415
20	463	439	415
21	511	463	415
22	671	495	481
23	927	551	545
24	1,439	687	673
25	2,207	1,199	929
26	3,487	1,967	1,441
27	6,559	3,503	2,467
28	12,703	6,567	4,571
29	24,991	12,719	8,617

(a) Memory usage in megabytes (MB) for Aer and different versions of Quthark. (Q is short for Qubits)



(b) Plot of table from Figure 3.4a with theoretical minimum usage for copying the whole state vector and only copying enough to fill up the GPU cache indicated with dashed lines.

Figure 3.4: Memory usage per qubit.

In Quthark v1 and v2, Futhark keeps the original state vector for reference, and creates a copy with the new values, which is then in the end used as the state for the next gate to be applied to. This doubles our memory usage to $2^n \cdot 2 \cdot 64 \cdot 2$ bits, which sounds bad. In practice it doesn't matter much. Since our memory usage is exponential, all it means is that we can represent one less qubit in our state vector. Based on these numbers, a 29 qubit state should take up 16 GB, and a 30 qubit state 32 GB. In practice, we found that Futhark actually uses 24 GB when processing a 29-qubit state. After some experimentation, we found the reason for this to be that an intermediate calculation of the state vector was not cleared fast enough. Fixing this in the Futhark compiler is outside the scope of this project. In Quthark v3, which duplicates only a small part of the state vector at a time, theoretical memory usage should be almost halved. This is close to what we see in our results, but Futhark still uses about one third extra on top of that which it should not need. Once this is fixed in the compiler, Quthark v3 should use approximately the same amount of memory per qubit as Aer. In figure Figure 3.4b, the actual memory usage of Quthark can be seen, alongside the in-place update memory usage of Aer, with the theoretical usage as a dashed line. When reading this figure it is important to note that, when the memory allocator bug is fixed in some future version of Futhark, Quthark's memory usage should drop by $\frac{1}{3}$, bringing us to the expected result of using twice the amount of memory per qubit compared to Aer.

3.6.2 Locality

The locality of the state vector is based on the target of the gate, as the pair that is needed to update the amplitude lies 2^k away from each other. So for a single qubit gate with a low target, e.g. 0, the spatial locality is very good as the pair is right next to it $2^0 = 1$. But becomes worse if we target higher qubits, for target $k = 19$ the pair is $2^{19} = 524,288$ away. This is quite interesting as we now start to get L2 cache misses, this can clearly be seen in Figure 3.5, where the blocks that are loaded into the 40.96MB cache of the NVIDIA A100 GPU can't hold all the pairs. At $k = 20$ and above we see that not a single pair is in the cache and our performance takes a major hit with a 6.5 second increase from best to worst. We can reason for this as the cache (theoretically) can hold $40.96MB / (64b \cdot 2) = 2,560,000$ values from the state vector, and *probably* holds both the original values and the newly calculated values, leaving us with $2,560,000/2 = 1,280,000$ values for the lookup. This is obviously a very rough estimate as the cache also has to hold many other things, such as the gate that is being applied, but it gives us a nice, though equally rough, idea of when we can expect misses. For $k = 19$ the *worst* pair of the block is positioned at $2 \cdot (19 + 1) - 1 = 1,048,575$, very close to our estimate of what can be stored in the L2 cache, and judging from what we see in Figure 3.5 we must conclude outside. At $k = 20$ the *worst* pair is at $2 \cdot (20 + 1) - 1 = 2,097,151$ which is completely outside the cache, confirming the plot.

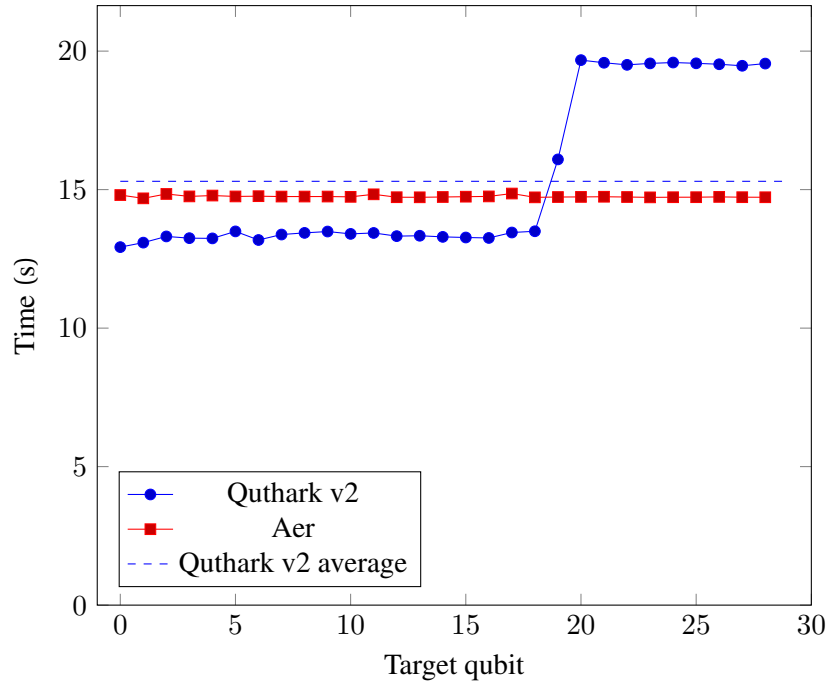


Figure 3.5: Quthark v2 and Aer locality, 29 qubit state vector, 1000 hadamard gates.

What is also interesting is the fact that Aer is actually running slower than our implementation for lower qubit targets, but always runs at the same pace. To us, this indicates some kind of

overhead from clever cache loading, making sure that each pair is always in the cache, avoiding unnecessary flushes. This was the primary motivation for our next experiment of implementing cache blocking.

3.6.3 Cache Blocking

After inspecting Figure 3.5 we saw a big potential for manipulating with the order of execution for gates with target 19 or above, and this way gain the same speed that we saw for gates that targeted qubit 18 and lower. This would require some overhead but could be justified up to a number of $15.3 - 13 = 2.3$ seconds, as our average case would then still have improved. As we can see on the plot, Aer has the same execution speed regardless of what qubit is targeted. While it is not as fast as our implementation when targeting qubit 18 or lower, the overhead of implementing such a feature seemed like it could be worth it.

The idea was that instead of mapping over the whole state vector, we could divide it into chunks which could fit into the level 2 (L2) cache of the GPU, map over those chunks instead, and then scatter them to the original state vector. This would of course require us to introduce another sequential loop into our program, as the work on each chunk would have to be performed sequentially. Still, we hoped that the improved temporal and spatial locality would give us a significant speedup that would make the general overhead added insignificant on average. As described in subsection 3.6.2 the GPU we test on has an L2 cache with room for approximately 2^{18} values - confirmed by Figure 3.5. So we attempted to divide our state vector into 2^{n-18} chunks, where each chunk could then encapsulate all pairs.

This should also avoid us needing to keep two copies of the state vector in memory during calculations, theoretically letting us use the same amount of memory as an imperative solution, plus the size of the cache. This will roughly cut our memory usage in half, allowing us to simulate circuits with one extra qubit.

Actually filling the cache with the appropriate pairs is a matter of bit swapping. Previously we would map over the indices for the state vector. We can imagine that if we kept doing this for the blocks we have established, we would place the following set into the first block $[0, 1, 2, \dots, \text{cache_size} - 1]$. Then we have to bit swap each target bit into an appropriate place for every index, creating the desired cache of indices.

Bit swapping

We have to be careful when swapping bit indexes for our cache block. If we swap the 0'th bit with the target bit k , we do get the pairs of indices which need each other to be update to be located next to each other, as we can see in the example in Table 3.2. However, we also get fragmentation in our indices, as $[1, 3, 5, \dots]$ and their corresponding pairs are positioned in a block 2^k away, thus ending up outside the size of our cache block. They will eventually be loaded in by themselves in their own block and also have good locality, but we end up wasting a lot of potential as when the cache is loaded it takes a chunk of the array and puts in to the cache, expecting spatial locality, and thus loading every consecutive number $[0, 1, 2, 3, 4, 5, \dots]$, even though we only need half.

In order to avoid this fragmented locality, we need to swap with a higher bit index. If we instead swap with the highest bit that still showcases good locality, our fragmentation won't be bad, and we can attain both good temporal and spatial locality. We know we have great locality from bit 18 and down, so this should be the target to swap with¹⁴. For multi qubit gates, we simply swap with the second highest bit, and so on.

0_2	0	→		0_2	0
1_2	1	→	10000000000000000000 ₂	524288	
10_2	2	→		10_2	2
11_2	3	→	100000000000000000010 ₂	524290	
100_2	4	→		100_2	4
101_2	5	→	1000000000000000000100 ₂	524292	

Table 3.2: Swapping target bit 19 with bit 0. Creating fragmented locality.

As we can see in Table 3.3 the difference when swapping with a higher bit is immense. Swapping with the right bit loads the cache efficiently and we see that the targets becomes irrelevant for performance. Instead of spiking as we saw in Figure 3.5, we get the same straight line as Aer does. However we also see that running a circuit with 1000 Hadamard gates has become significantly slower. Without cache blocking we had a best case of 13 seconds and worst case of 19.5, giving us an average execution time of 15.3 seconds for a circuit which targets every qubit with the same frequency. With cache blocking enabled, we finish the calculation in 22 seconds, even slower than the worst case. This is obviously not the kind of result we were hoping to achieve. We had aimed to at least shave significant time off the worst case scenario, while only slightly worsening our best case. What we were looking for was a number below our previous average of 15.3 seconds, but with a deficit of $15.3 - 22 = -6.7$ s, we ended up with significantly worse performance.

	Target bit 0	Target bit 28
Swap bit 28 and 0	22.50s	46.10s
Swap bit 28 and 10	21.89s	22.13s

Table 3.3: 1000 Hadamard gates applied to target bit.

The reason behind this 6.7 second increase is hard to determine, which is one of the drawbacks of using Futhark. The high level nature of the language prevents us from looking into what might have caused it specifically, and ascribing the whole thing to just "overhead" is not very meaningful. Instead we drew on the expertise from one of the language creators to gain an understanding of what might have happened. After doing so, we come to the conclusion that the actual culprit is the compiler's ability to fuse our SOACs, as the rewritten gate applicator is not as transparent to the compiler as the simpler version without cache-blocking.

¹⁴On this particular piece of hardware with a 40 MB cache.

3.6.4 Applying Custom Gates

Even though we did not succeed in beating Aer when executing randomly generated circuits, we saw a trend that went in our favour when applying larger unitary gates. For gates up to sizes of nine qubits Aer is faster, but when moving past 10-qubit gates, the run time explodes in more than exponential time, where Quthark stays on the expected exponential growth. This can clearly be seen in Figure 3.6.

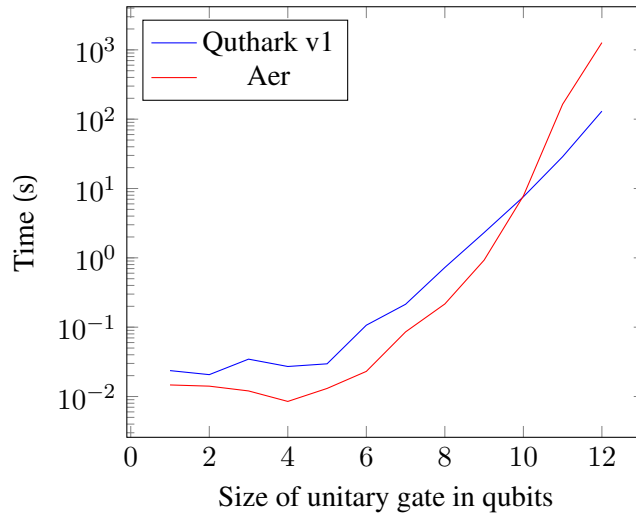


Figure 3.6: Executing a randomly generated unitary gate on a 20 qubit state vector.

In theory, this could be useful when distributing very complex circuits between users. As the size of a custom gate scales solely with the number of qubits represented by the state it is applied to, a circuit consisting of a sufficient number of standard gates relative to the size of the state would be cheaper to distribute, and run faster, as a custom gate.

These would have to be extremely long circuits however, and in reality it is perhaps most interesting to observe that Aer is clearly doing something clever for smaller custom gates, which then stops paying off once the gate size crosses a certain point. The performance benefits we see for Quthark are most likely just the result of us applying gates in a simpler way, getting rid of the optimization overhead, but being worse for the gate sizes we usually operate with.

It could also be that Aer is using a different data-structure entirely to represent its state vector and gates, which makes application more efficient for smaller gate-sizes. Either way, the fact that Aer's performance doesn't scale linearly once the GPU is fully utilized indicates that something interesting must be going on.

3.7 Other Experiments

3.7.1 A Sparse State Vector

Another potential saving we explored comes from realizing that not only are our gates generally extremely sparse, but so is our state vector. By iterating over a list of indices that have already been written to, instead of iterating over the entire state vector, we hoped to both save on memory, and improve performance performance, since any non-zero output must necessarily involve some non-zero value in both the gate and the state vector. We would then update the state vector using a scatter operation, since the result of the calculations may need to also be written to an index that previously contained zero. Keeping track of non-zero entries clearly isn't free, but by using a sparse representation of the state vector, we hoped to still keep memory consumption to a reasonable level, while gaining a significant performance uplift for most cases. It turned out that, at least with random circuits, the sparsity of a state vector decreased very fast, and it would not make sense even for large states past a few hundred gate-applications. This fact is clearly showcased in Figure 3.7, but what might actually be the case of the sparsity for an actual quantum algorithm has not been evaluated.

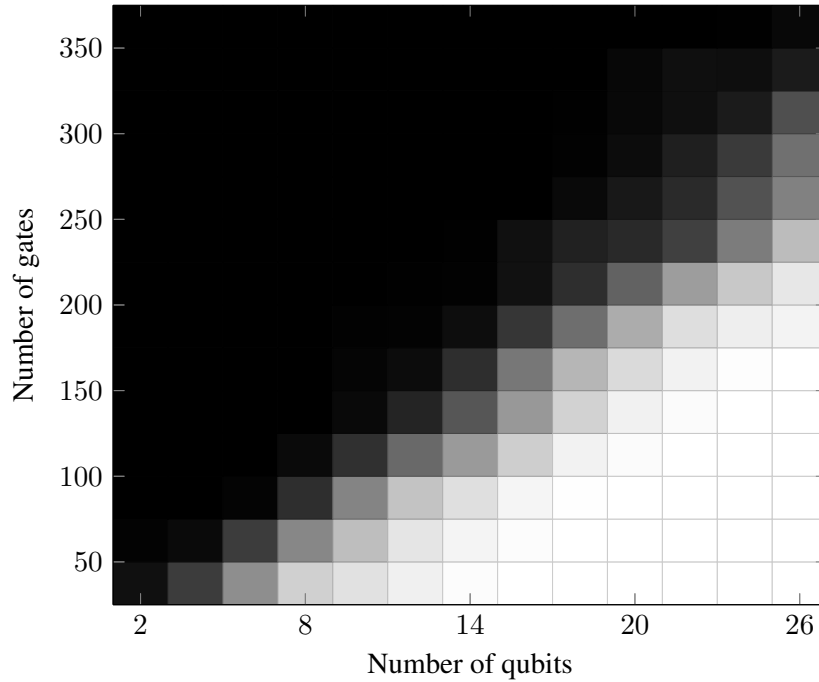


Figure 3.7: Sparsity of state vector after executing random generated circuits. White is sparse, black is dense

As a consequence of this result, we decided not to pursue a sparse implementation any further. While it would likely make Quthark faster than Aer for smaller circuits, we aimed to be fast on average, not just for some relatively small subset of scenarios.

3.7.2 Gate fusion

The idea of gate fusion is based on the fact that matrix-multiplication, and by extension matrix-vector multiplication, is associative. More formally, for matrices A , B and vector x :

$$x(AB) = xA(B) \quad (3.2)$$

By associativity then, to fuse some quantum gates we want applied to a state, all we need to do is multiply them with each other. The resulting fused gate will then give us the same result as applying each the non-fused gate in the same order would.

This does present us with some complications. As we don't want to have to work with a 2^{2n} state, which would be necessary to combine all of the gates being applied to an n -qubit state regardless of their target, we have to restrict ourselves to combining gates which apply to the same qubit. Let's see how this approach performs on an example circuit:

Additionally, as matrix-multiplication is not commutative¹⁵, we cannot do anything which upsets the chronological order of how the same parts of the state are affected.

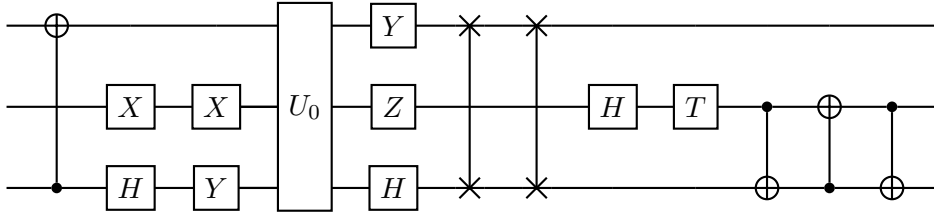


Table 3.4: A random quantum circuit excerpt pre-fusion.

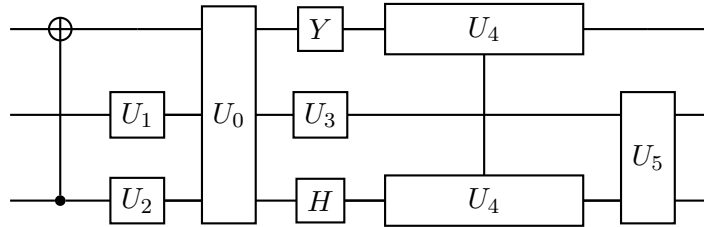


Table 3.5: The random quantum circuit excerpt post-fusion.

We see that nothing has happened to the CNOT-gate at the beginning of the example. It could be fused with the two SWAP-gates, but there are gates with different dimensionality in-between. X fuses with X to become U_1 , which must be the identity matrix I_2 by definition of Pauli- X . H and Y fuse to become U_2 , and U_0 is unchanged, but prevents any further fusion from taking place. U_3 is formed from the Z , H and T -gates. Even though there are qubits with different dimensionality in-between them, they do not share any targets, so merging is possible. Finally,

¹⁵Commutative operations have the property that the order of operations does not matter. So $AB = BA$. With non-commutative operations like matrix-multiplication, $AB \neq BA$, for $B \neq A$.

the two SWAP-gates fuse to form U_4 , which must be equivalent to I_4 , and the CNOT-gates form to become what amounts to the generic matrix-representation of a SWAP-gate. Considering that we just stated that matrix-vector multiplication is not commutative, it may be surprising to see that we were able to fuse gate Z with gate H and T , when there are two other gates being applied in-between them chronologically.

The key to this result is in the definition of the state vector. Recall that a state vector consisting of multiple qubits is constructed using the Kronecker-product Equation 2.1.2. By definition of the Kronecker product, $(A \otimes B)(|\psi\rangle \otimes |\phi\rangle) = A|\psi\rangle \otimes B|\phi\rangle$, where A and B are linear operators, which is another way of say matrices¹⁶. In other words, we can consider the order of operations separately for each qubit in the state. So as long as there is no partial overlap of targets between some gate chronologically in-between the gates we're trying to merge, and the targets of the mergeable gates, we can always merge.

This results in fewer runs through the state vector, but would convert our highly-efficient functional gates back to generic matrices. Combined with getting constant internal compiler errors if we didn't force the code to be much less parallel than it should be able to be, we decided to move on and focus on other possible improvements.

If we had more time, a more intelligent version of the fusion-algorithm could have been written to minimize the number of generic gates, and intelligently remove gates which cancel each other out. Such an idealized function should be capable of producing something like the circuit below.

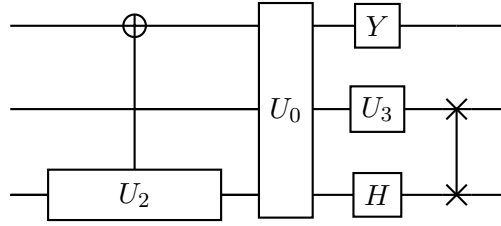


Table 3.6: The random quantum circuit excerpt after an idealized functional reduction.

The version we currently have only combines single-qubit gates, while leaving all other gates as they were. It is fairly trivial to extend to gates of higher dimensionality, as the structure of the function is already set-up for it. All that is needed is some testing and minor adjustments of the logic, but for it to make sense performance-wise, we would have to get the code running more in parallel.

3.7.3 Just-in-time

Just-in-time (JIT) compilation, is a way of executing code that is being compiled at run time. Like many other compilers, the Futhark compiler works better the more information it has about the code it has to run. As an experiment, we tried to write a Python program to generate Futhark code, which is then subsequently compiled and executed - just-in-time. This eliminates a lot of

¹⁶Nielsen and Chuang 2011, pp. 71–74.

uncertainties for the compiler, and we did see some performance gains. However, the overhead of first sequentially compiling turned out to dominate the execution time to such a degree that scaling the size of our state to the limit of what our simulator can handle was not sufficient. The other thing we can talk about performance scaling with is the number of gates. Unfortunately, it turned out that the additional time needed to compile each added gate outweighed the performance gained by the additional optimizations performed by the compiler.

Any repeat executions do not need to worry about compilation-time, but generally it is not interesting to repeatedly execute circuits on a deterministic quantum circuit simulator such as Quthark. The result will always be the same, as the only variable one could possibly change, the number of qubits, is very much tied to the qubits targeted by the gates in the circuit. Executing a circuit which targets three separate qubits on a two-qubit state simply will not work, and moving to a larger state will give the same results spread over a larger state vector, achieving nothing new but wasting memory.

The performance improvements gained from this did inspire us to try and give the compiler more information about what the code it is processing is actually doing. We moved away from defining our gates as generic matrices, and instead wrote each out as an explicit function. This also allowed us to ignore the parts of gates which don't actually do anything, such as most of a CNOT-gate, entirely.

In Figure 3.8 we can see the run, compile and total time of the JIT implementation. The plot was produced by running a randomly generated circuit of size 1000 on different number qubit states. But ran under a massive handicap, as the GPU was being utilized by another student while the experiment was conducted. Unfortunately for us, it was a very long experiment spanning days, and so we had no opportunity to get comparable results. We did get to run a few uninterrupted experiments beforehand and so we can say that the trend remains the same.

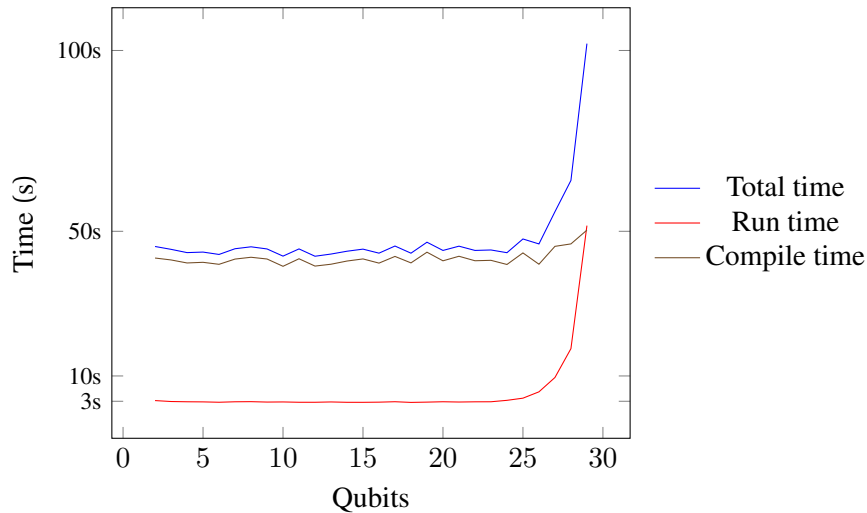


Figure 3.8: Executing a randomly generated circuit of size 1000, on different sized state vectors.

3.7.4 Work Optimal Simulator

As described in subsection 3.4.1 the workload of applying a unitary gate is greater than that of the imperative example. So it was natural to attempt to cut down the work to match it. Recall that the group of amplitudes needed to update one amplitude is 2^m where m is the size of the qubit gate, and that we then correspondingly only need to go through $size = \frac{2^n}{2^m}$ values. Once again we can take advantage of the binary properties of the state vector, by tabulating over $size$ while swapping the most significant bit with the target, we get the first indices of all pairs. Subsequently we can then first map over each of these indices and return their group. Then map over the group of indices calculating the new values for the state vector, lastly flattening and scattering them to the state vector. A snippet of the code described can be seen in Listing 4.

```

74 let indices: [size]i64 = tabulate size swap
75 let groups: [size][mm]i64 = map group indices
76 let values: [size][mm]complex = map vals groups
77 let flat_groups = flatten groups :> [nn]i64
78 let flat_values = flatten values :> [nn]complex
79 in scatter state flat_groups flat_values

```

Listing 4: Snippe

The results from this approach were disappointing to say the least, as can be seen in Table 3.7, running 100 randomly generated unitary gates of size m , didn't beat the approach implemented in Quthark v1.

m	1	2	3	4
Quthark v1	2.9s	3.0s	4.8s	7.7s
Work Optimal	8.8s	9.8s	11.8s	18.0s

Table 3.7: Running 100 random unitary gates of 4 different sizes on a 29 qubit state vector.

It is once again hard to reason for why this is, as it theoretically should be faster. And we can only once again suspect that the compiler can't optimize this solution as easily. There are many SOACs that needs to be fused, and the structure is in general more complicated.

If we had seen better results on this approach, we could also have implemented a specifically better approach for controlled gates. By looking at the control bit, we can deduct that we only have to apply the gate to one half of the state vector. Even $\frac{1}{4}$ for double controlled gates, and so on. But as it is with time restrained research, we had to prioritize our efforts else were.

3.8 Future Work

We believe we have written a fairly concise quantum circuit simulator, which can be expanded upon without the need for reading more than 150 lines of code. The foundation that lies behind

it makes up quite a lot more, around 600 lines of Futhark code, but can be ignored by most. So building upon, improving and adding features is definitely possible. Speaking of adding features, we did choose not to implement some features of the Qiskit API, such as returning the result of a measurement, rather than just the post-measurement state, and specifying a custom state to run a circuit on. Still, considering that the Qiskit Aer state vector simulator on its own consists of almost 20 times more lines of code than Quthark, while implementing more gates and most of the feature-set of Qiskit, we consider it unlikely that Quthark will ever balloon to anything close to that size. The fact that it supports one of the most used quantum APIs also might make it more interesting for others to experiment with, as they can run their own benchmarks against our implementation.

For the rest of this section, we explore opportunities for improvement which we decided against pursuing, didn't come to a final conclusion on, or simply did not have the time to implement.

3.8.1 Accuracy

As mentioned, the accuracy of our calculations is limited by the number of values representable by two 64-bit floats. But in our current implementation, as the amplitudes stored in the state vector are fairly small, accuracy is much lower in practice. This could be remedied by instead allowing the values to fluctuate between the maximum and minimum representable value.

This would theoretically allow us to work with even larger, denser state vectors, where distinguishing very small values from zero could otherwise start to become a problem. In practice, we did not encounter any problems related to this issue with the size of states we worked with.

3.8.2 Better JIT

The JIT-implementation we wrote was very superficial. It basically just generated the input-variables, such as the size of our state vector and which gates were to be applied, and compiled the code with these already set, instead of feeding them dynamically to Quthark from Python at execution-time.

This made sense then, as our gate-application was entirely generic. With our gates no longer being generic, it would be interesting to instead generate more barebones Futhark-code, which would simply explicitly apply one gate after another, ridding the generated code entirely of unused gates and generic functions.

We did do some initial experimentation, and found that the increase in compilation-time as we added more gates did significantly outweigh any possible performance benefits, but we did not spend significant time on trimming down the code, or experimenting with different strategies for the generated code.

3.8.3 Fusion

While we did implement gate-fusion for our generic gate simulator(Quthark v1), implementing something similar for Quthark v2 was not feasible within the time we had for this project, as

it would require a significant amount of code for each type of gate. It would likely lead to a measurable performance uplift, as we do see the it gives better results when enabled for Qiskit Aer, and similar projects to ours seem to have needed to make heavy use of circuit optimization though fusion to match or exceed Aer’s performance¹⁷. If we were to go down this route, we would likely choose to reduce our supported gate-set to make fusion more manageable.

3.8.4 Control flow

Qiskit supports control flow statements conditioned on measured values. Quthark does not support this feature at present. It could be emulated trivially by returning the state and passing it back to Quthark after doing a manual measurement, and using that as input to a standard Python if-else statement. We currently do not support setting a custom state vector, but this omission is not due to any real obstacles standing in the way of allowing for that. It would be trivial to add. We did not spend time to look into what it would take to add control flow statements, and no such circuits are generated by Qiskit for our benchmarks or tests. We estimate the overhead added from implementing such a feature to be trivial. Each check would require a maximum of $O(n^2)$ operations in the number of qubits. The actual time required depends on whether or not we need to simulate the effect an actual measurement would have on the state, but it would at most be the same overhead as adding a single additional gate to a circuit.

3.9 Reproducing Results and Executing Code

The Qiskit API is of course the preferred way to interact with the Quthark backend, but in order to execute and time our code in an easy and repeatable manner, we wrote a Python script for running and comparing execution of random quantum circuits between simulators. In the following section we explain the API and the commands used to reach the data from section 3.6

benchmark.py API The basic instruction for running Quthark with a random circuit, is specifying the number of qubits and the size of the circuit.

```
$ python benchmark.py <qubits> <size>
```

Quthark is regarded as the default simulator for running our benchmarks, so if we want Qiskit Aer to run the circuit, we have to specify it explicitly. We can choose which simulator we want the circuit to run on with the `[--simulator -s quthark,qiskit,all]` flag, where `-s all` runs both simulators.

The `[--runs -r RUNS]` flag denotes the number of times the experiment should be repeated. Paired with the `[--avg]` flag, the average time of the runs for a given experiment is reported.

As we described in subsection 3.4.4 Aer needs a transpiled circuit for execution, this can be achieved with the `[--transpile -t]` flag.

We ran the following command for achieving the data displayed in Figure 3.3

```
$ python benchmark.py 2-29 1000 -s all -t -r 10 --avg
```

¹⁷Suzuki *et al.* 2021.

A full explanation for all flags can be achieved with the help command.

```
$ python benchmark.py --help
```

Chapter 4

Discussion

Supporting many gates means that we can execute the shortest possible version of any circuit supported by Qiskit, which is good for performance. However, supporting many gates makes it harder to optimize our code. Gate fusion especially becomes very challenging, as we have to consider how every single kind of gate we support will be transformed when it fuses with another kind of supported gate.

We could always just fall back to representing a gate generically when we can't figure it out, but that will not result in the best possible performance, and we would have to start considering how many gates we should fuse as a minimum if the gains are to outweigh the losses.

While Futhark is capable of generating very efficient parallel code, as can be seen from the numerous benchmarks on its website, it doesn't always perform as well as one might naively expect. As functions become more complicated, increasingly intricate knowledge of the compiler is required to achieve good performance. One example from this project is cache blocking, which we dismissed several times based on poor performance when benchmarking a basic proof of concept implementation. Through consulting with the head Futhark developer ¹, it became clear that we were leaving a lot of performance on the table by not writing the code exactly right. We didn't manage to optimize cache blocking to the extent that having it was more performant than not having it before running out of time, but we did see clear improvements with each suggestion we implemented, leading us to believe that it may have been possible with more work, and a better understanding of the Futhark compiler. This need to be able to predict how the compiler will interpret what we write was frustrating. We would write code which theoretically seemed like it did minimal work in a parallel way, only to find that the compiler could not understand it well enough to generate code which lived up to that theoretical performance. Compared to low-level languages such as OpenCL and CUDA where what happens is more clear to the developer, we found this to be a disadvantage of developing in Futhark.

¹Troels Henriksen, project advisor

Chapter 5

Conclusion

We have in this report researched and discussed the benefits of developing a quantum circuit simulator in the data parallel functional programming language Futhark.

Futhark is very portable. While Qiskit can only utilize the compute capabilities of CUDA-compatible hardware, Futhark code can be compiled to run on any modern GPU. Most of the time, we appreciated that Futhark allowed us to avoid writing low level GPU code, making the project of creating the simulator fairly pain free, but with the drawback of making it harder to optimize for performance. Instead of writing low level optimisations, we spent a lot of time fighting the compiler. While we did have measurably worse performance overall in the most meaningful benchmarks, the performance of our simulator is still comparable to that of Qiskit Aer on average, while having a code base less than a tenth of the size of the the Aer state vector simulator alone. While we did choose not to implement some features, we do not estimate that any of them are of a scope large enough to expand our code base to anywhere near that level.

Unfortunately, choosing Futhark did also come with some drawbacks. We ran in to a number of bugs over different versions of Futhark, resulting in problems ranging from excessive memory usage to correct code just not compiling. We also found writing code which allows the compiler to do its best work to be a somewhat opaque process. Ideally, one would want two versions of code which theoretically performs the same number of operations with the same degree of parallelism to yield the same performance, but this was not always the case.

Futhark still being relatively new also meant that we had to implement features ourselves which would be part of the standard library of most programming languages, such as the work we did to expand the complex number library.

Working in a high-level functional language also made it difficult for us to gain an advantage from theoretically meaningful optimisations, such as cache-blocking, without consulting with expert users like our primary project advisor. We continuously ran into problems regarding the compilers ability to make are code performant and when trying to debug it quickly became a black box. We did also have fantastic moments, where Futhark managed to gives us great performance from simple code, and when that works it felt amazing, but sometime it came down to a coin flip. In contrast, working with the Qiskit API was reasonably unproblematic. Documentation is generally good, though it can be difficult to find the right documentation due to the size of the project. The complexity of Qiskit also makes it hard to add new features for an

outsider. For instance, Quthark is capable of modelling the effect of measurements in any valid basis, but Qiskit only allows measurements in the computational basis. Adding measurements for new bases to Quthark takes only a few dozen lines of fairly trivial code. These kinds of modifications being possible without a major time investment is a benefit of the more compact code-base achievable with a high-level language such as Futhark, and Quthark's fairly straightforward architecture.

In conclusion, we do not think that the choice to develop a state vector simulator in Futhark was a mistake. While there were unexpected challenges, and we did not end up with performance quite matching Qiskit Aer, we have come very close in a relatively short amount of time. Looking at projects written in other languages with similar goals to ours, the amount of work needed to achieve performance better than the Aer state vector simulator seems immense.

Bibliography

1. M. S. ANIS *et al.*, *Qiskit Circuit library*, July 2021, (https://qiskit.org/documentation/stable/0.37/apidoc/circuit_library.html).
2. M. S. ANIS *et al.*, *Qiskit: An Open-source Framework for Quantum Computing*, 2021.
3. D. P. DiVincenzo, *Physical Review A* **51**, 1015–1022, (<https://doi.org/10.1103/PhysRevA.51.1015>) (1995).
4. D. Gottesman, (<https://arxiv.org/abs/quant-ph/9807006>) (1998).
5. M. A. Nielsen, I. L. Chuang, *Quantum Computation and Quantum Information: 10th Anniversary Edition* (Cambridge University Press, USA, 10th, 2011), ISBN: 1107002176.
6. Y. Suzuki *et al.*, *Quantum* **5**, 559, (<https://doi.org/10.22331/q-2021-10-06-559>) (Oct. 2021).
7. S. S. Tannu, M. K. Qureshi, presented at the Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 987–999, ISBN: 9781450362405, (<https://doi.org/10.1145/3297858.3304007>).