CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Quasim: Quantum Circuit Simulation

Design and Implementation of a Rust library enabling construction, inspection and execution of quantum circuits

Bachelor's thesis in Computer science and engineering

Albin Andersson
Folke Ishii
Emil Karlsson
Jonas Mokhtar Lutschke
Sebastian Sandstig
Otto Svalin

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026

Bachelor's thesis 2026

Quasim: Quantum Circuit Simulation

Design and Implementation of a Rust library enabling construction, inspection and execution of quantum circuits

Albin Andersson

Folke Ishii

Emil Karlsson

Jonas Mokhtar Lutschke

Sebastian Sandstig

Otto Svalin

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg

Gothenburg, Sweden 2026

Quasim: Quantum Circuit Simulation
Design and Implementation of a Rust library enabling construction, inspection and execution of quantum circuits
Albin Andersson Folke Ishii Emil Karlsson Jonas Mokhtar Lutschke Sebastian Sandstig Otto Svalin

Supervisor (Handledare): Robin Adams, Department of Computer Science and Engineering
Examiner: Patrik Jansson, Department of Computer Science and Engineering
Graded by teacher (Medrättande lärare): Muhammad Mustafa Hassan, Department of Computer Science and Engineering

Bachelor's Thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2026

Abstract

Quantum computing is becoming more prominent around the world, bringing new approaches to solving computationally difficult problems. The basic computational unit of quantum computers is called *qubit*. Because qubits can exist in a superposition of both 0 and 1 simultaneously, quantum computers have the capacity to let us execute algorithms that are orders of magnitude faster than their classical counterpart.

In this thesis, a quantum circuit simulator is developed as a library for the Rust programming language. This library supports multiple interchangeable simulation backends suited for different quantum circuit computation tasks. The library provides a programming interface for creating quantum circuits, enabling users to construct and simulate quantum algorithms using the provided simulation backends. For backends that support it, a terminal debugging interface lets users inspect qubit states at each stage of an algorithm.

The work focuses on optimization strategies for simulation backends and their comparative performance. The result shows certain simulators can be constructed which excel at simulation of specific circuit configurations, while still subject to the exponential time complexity in the worst case scenario. GPU hardware acceleration proves advantageous for simulating large circuits.

Acknowledgements

The project group wants to thank their supervisor, Robin Adams, who has provided great insight into the field of quantum mechanics and quantum computing to a group to which these subjects were completely foreign before.

Albin Andersson, Folke Ishii, Emil Karlsson, Jonas Mokhtar Lutschke, Sebastian Sandstig, Otto Svalin, Gothenburg, 2026

Contents

List of Figures

1

Background

Quantum computers are being developed around the world, bringing a radical change to how computing can be done. Quantum computers operate on the principle that a so-called "qubit" (quantum bit) can exist in a state representing 0 and 1 simultaneously until being measured, at which point the qubit collapses to a classical result, yielding a definite 0 or 1 [1, p. 13]. This combined state is called a superposition. When applying an operation on a qubit that exists in a superposition, one can think of it as applying the operation to both possible states at once; this enables algorithms that can solve certain problems asymptotically faster than the best-known classical algorithms.

Quantum algorithms have potential applications beyond cryptography, for example in simulating quantum physical systems which is useful in physics and chemistry [1]. To be prepared for this development, students and researchers are learning about and developing quantum algorithms. However, since access to quantum hardware is limited, students and researchers need tools that let them prototype, test and debug algorithms on classical hardware.

2

Theory

Building a quantum computer simulator requires a rudimentary understanding of the mathematical model of a quantum computer. A simulator also lets us inspect the complete system state at intermediate steps, which aids debugging and understanding of algorithms even though such inspection is not possible on real quantum hardware.

2.1 The qubit

In computer science, a bit models any physical system that can be in one of two distinguishable *states*: 0 or 1. However, a quantum bit, also known as a *qubit*, models a physical system that can be in a state $|0\rangle$, $|1\rangle$ or some combination thereof [1, p. 13–14]. In general, the state of a qubit $|\psi\rangle$ can be described by

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle\,, \quad |\alpha|^2 + |\beta|^2 = 1, \quad \alpha, \beta \in \mathbb{C} \tag{2.1}$$

which means that the state of the qubit is in a linear combination of states $|0\rangle$ and $|1\rangle$. This is called a *superposition*. Measuring a qubit causes its state to change to $|0\rangle$ or $|1\rangle$ with probability $|\alpha|^2$ and $|\beta|^2$, respectively. The qubit is said to *collapse* to $|0\rangle$ or $|1\rangle$. Since $|\alpha|^2$ and $|\beta|^2$ represents probabilities, there must be a constraint on α and β, namely $|\alpha|^2 + |\beta|^2 = 1$.

One property of a qubit is the *phase*, which can mean two different things. *Global phase* is a purely mathematical byproduct and does not have any physical significance, that is to say $e^{i\theta}\,|\psi\rangle$ and $|\psi\rangle$ represent the same physical state. A *relative phase* means that the amplitudes of two basis states differ by a complex phase factor, for example $\alpha = e^{i\theta}\beta$ for some θ. Relative phase does not affect the outcome of a measurement, but it can change the result of applying a gate [1, p. 93].

Another way to describe a qubit is by a point on the *Bloch sphere* as shown in figure 2.1. The point (x, y, z) on the Bloch sphere is known as the Bloch vector [1, p. 105]. The angle θ can be thought of as changing the probability of measurement and the angle ϕ as changing the phase.

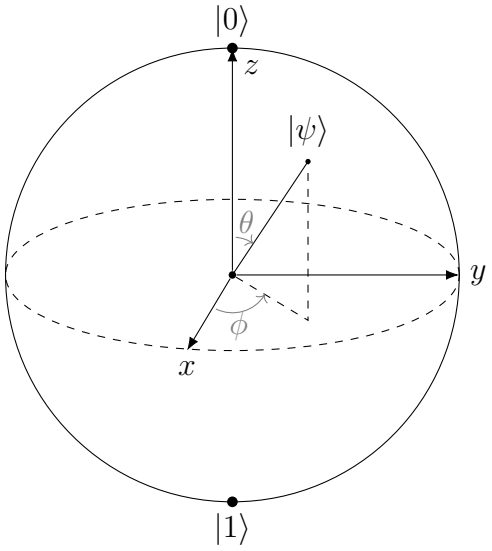


Figure 2.1: Bloch sphere [1, p. 15]

2.2 The state of multiple qubits

As shown in equation 2.1, a system consisting of a single qubit can represented by the linear combination of two basis states $\alpha\,|0\rangle + \beta\,|1\rangle$. It follows that a system consisting of two qubits, as shown in equation 2.2 can be represented by a linear combination of four basis states [1, p. 16]. Equation 2.3 shows that each component of state vector holds the complex coefficient of a unique basis state. In general, for an n qubits system, the state can be expressed as in equation 2.4, a linear combination of 2^n basis states [2, p. 26].

$$|\Psi\rangle = \alpha\,|00\rangle + \beta\,|01\rangle + \gamma\,|10\rangle + \delta\,|11\rangle\,, \quad \alpha,\beta,\gamma,\delta \in \mathbb{C} \tag{2.2}$$

$$\Updownarrow$$

$$|\Psi\rangle = \begin{pmatrix}\alpha\\ \beta\\ \gamma\\ \delta\end{pmatrix} = \alpha\begin{pmatrix}1\\0\\0\\0\end{pmatrix} + \beta\begin{pmatrix}0\\1\\0\\0\end{pmatrix} + \gamma\begin{pmatrix}0\\0\\1\\0\end{pmatrix} + \delta\begin{pmatrix}0\\0\\0\\1\end{pmatrix}, \quad \alpha,\beta,\gamma,\delta \in \mathbb{C} \tag{2.3}$$

$$|\Psi\rangle = \sum_{i=0}^{2^n-1} a_i\,|i\rangle\,, \quad \sum_i |a_i|^2 = 1, \quad a_i \in \mathbb{C} \tag{2.4}$$

The state of a system composed of several independent systems can be constructed using the *tensor product*. Note that this construction applies only when the subsystems are independent (a product or separable state); many joint states are entangled

and cannot be written as a simple tensor product of subsystem states. Let $|\Psi\rangle$ be the state of a system composed of n qubits with known states $|\psi_i\rangle\,,\ i \in [1, n]$, then when the state is separable it can be expressed as a single 2^n component vector as shown in equation 2.5. Due to the associative property of the tensor product, equation 2.5 can be understood as each $|\psi_i\rangle$ being a system of qubits and n as the number of such systems. Equation 2.6 shows an example of the state vector for a two qubit system.

$$|\Psi\rangle = |\psi_1\rangle \otimes |\psi_2\rangle \otimes \cdots \otimes |\psi_n\rangle \tag{2.5}$$

$$|\Psi\rangle = |\psi_1\rangle \otimes |\psi_2\rangle = \begin{pmatrix} \alpha\gamma \\ \alpha\delta \\ \beta\gamma \\ \beta\delta \end{pmatrix}, \quad |\psi_1\rangle = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}, \quad |\psi_2\rangle = \begin{pmatrix} \gamma \\ \delta \end{pmatrix} \tag{2.6}$$

However, most states can not be expressed as a tensor product of other states. For example, the so called *Bell state*, represented by a linear combination in 2.7, can not be represented as a tensor product between the states of two qubits, as shown by the unsolvable system in equation 2.8. Any state that can not be expressed as a tensor product of its components is known as an *entangled state*. [1, p. 93–96]

$$|\Phi^+\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}} \tag{2.7}$$

$$|\Phi^+\rangle = \begin{pmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ \frac{1}{\sqrt{2}} \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \otimes \begin{pmatrix} \gamma \\ \delta \end{pmatrix} \implies \begin{cases} \alpha\gamma = \frac{1}{\sqrt{2}} \\ \alpha\delta = 0 \\ \beta\gamma = 0 \\ \beta\delta = \frac{1}{\sqrt{2}} \end{cases} \tag{2.8}$$

Tensor product: For an $m \times n$ matrix A and a $p \times q$ matrix B, the tensor product $A \otimes B$ can be represented by a $mp \times nq$ matrix of the form

$$A \otimes B \equiv \begin{pmatrix} A_{11}B & A_{12}B & \cdots & A_{1n}B \\ A_{21}B & A_{22}B & \cdots & A_{1n}B \\ \vdots & \vdots & \ddots & \vdots \\ A_{m1}B & A_{m2}B & \cdots & A_{mn}B \end{pmatrix}$$

known as the *Kronecker product* of A and B. [1, p. 73–74]

2.2.1 Density matrix

The density matrix ρ is an alternative way of describing the state of a quantum system. One advantage of using a density matrix over a state vector is the ability

to retrieve the state of an individual qubit in a multi-qubit system, even if it is entangled [1, p. 100], [3, p. 115–116]. For a known state $|\psi\rangle$ the density matrix is defined by $\rho = |\psi\rangle \langle\psi|$.

Given the density matrix ρ_{AB} on a joint system AB, it is possible to take the partial trace over A to get the density matrix of system B [3, p. 141–142]. Since ρ_{AB} is a 4×4 matrix, the equation for ρ_B is

$$\rho_B = \mathrm{tr}_A(\rho_{AB}) = \mathrm{tr}_A \begin{pmatrix} \alpha_{00} & \alpha_{01} & \alpha_{02} & \alpha_{03} \\ \alpha_{10} & \alpha_{11} & \alpha_{12} & \alpha_{13} \\ \alpha_{20} & \alpha_{21} & \alpha_{22} & \alpha_{23} \\ \alpha_{30} & \alpha_{31} & \alpha_{32} & \alpha_{33} \end{pmatrix} = \begin{pmatrix} \alpha_{00} + \alpha_{11} & \alpha_{02} + \alpha_{13} \\ \alpha_{20} + \alpha_{31} & \alpha_{22} + \alpha_{33} \end{pmatrix} \tag{2.9}$$

Another use case for density matrices is the ability to obtain the Bloch vector for a qubit, which can be done with the following equation

$$\rho = \frac{1}{2}I + \frac{1}{2}(x\sigma_x + y\sigma_y + z\sigma_z) = \frac{1}{2}\begin{pmatrix} 1+z & x-iy \\ x+iy & 1-z \end{pmatrix} \tag{2.10}$$

where σ_k are the standard Pauli matrices [4]. With equation 2.9 and equation 2.10, it is possible to get the x, y, z coordinates for the Bloch vector of the second qubit in a 2-qubit system

$$\begin{aligned} 2 \cdot (\alpha_{00} + \alpha_{11}) &= 1 + z \\ 2 \cdot (\alpha_{20} + \alpha_{31}) &= x + iy \end{aligned} \iff \begin{aligned} x &= \Re(2 \cdot (\alpha_{20} + \alpha_{31})) \\ y &= \Im(2 \cdot (\alpha_{20} + \alpha_{31})) \\ z &= 2 \cdot (\alpha_{00} + \alpha_{11}) - 1 \end{aligned} \tag{2.11}$$

Equation 2.11 will be used later in section 6.2.2 to verify that phase kickback works properly.

2.3 Quantum gates

Similarly to classical logic gates, *quantum gates* are used to alter the state of a system of qubits. For example, the quantum counterpart of the classical *NOT* gate would be the X gate, which swaps the complex coefficient between the two basis states [1, p. 17-18].

Logic gate: NOT	**Quantum gate: X**
$0 \longrightarrow 1$	$\begin{pmatrix} \alpha \\ \beta \end{pmatrix} \longrightarrow \begin{pmatrix} \beta \\ \alpha \end{pmatrix}$
$1 \longrightarrow 0$	

The X gate and any other quantum gate that operate on a single qubit can be represented by a 2×2 matrix. Here, operate means multiplying the 2×2 matrix with the 2×1 state vector for a qubit. To preserve the condition $|\alpha^2| + |\beta^2| = 1$ after applying a quantum gate, the matrix representation of any quantum gate must be *unitary*. A unitary matrix U is defined to be any matrix that satisfies $UU^\dagger = I \iff U^{-1} = U^\dagger$. The *adjoint* of U, denoted as $U^\dagger$, is the transpose of the complex conjugate of U. It also follows that any unitary matrix represents a valid quantum gate [1, p. 18]. Any quantum gate U acting on a single qubit can be constructed using three angles as shown in equation 2.12 [1, p. 174–176].

$$\boxed{U} \equiv \begin{bmatrix} \cos\frac{\theta}{2} & -e^{i\lambda}\sin\frac{\theta}{2} \\ e^{i\phi}\sin\frac{\theta}{2} & e^{i(\lambda+\phi)}\cos\frac{\theta}{2} \end{bmatrix}, \quad \theta, \phi, \lambda \in \mathbb{R} \tag{2.12}$$

There are a couple of special cases of U, shown below. These are the the Pauli matrices: X, Y and Z; the Hadamard gate, the S gate, and the rotation gates to rotate the Bloch sphere around an axis θ radians.

$$\boxed{X} \text{ or } \oplus \equiv \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad \boxed{Y} \equiv \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, \quad \boxed{Z} \equiv \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \tag{2.13}$$

$$\boxed{H} \equiv \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \boxed{S} \equiv \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}, \boxed{R_z} \equiv \begin{bmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{bmatrix} \tag{2.14}$$

$$\boxed{R_x} \equiv \begin{bmatrix} \cos\frac{\theta}{2} & -i\sin\frac{\theta}{2} \\ -i\sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{bmatrix}, \boxed{R_y} \equiv \begin{bmatrix} \cos\frac{\theta}{2} & -\sin\frac{\theta}{2} \\ \sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{bmatrix} \tag{2.15}$$

2.3.1 Multi-qubit quantum gates

As shown in section 2.2 the state of a system consisting of n qubits is represented by a 2^n component vector. It follows that any gate acting on n qubits can be represented by a $2^n \times 2^n$ unitary matrix. Note that most n-qubit gates are not tensor products of single-qubit gates; the tensor-product form $U_1 \otimes \cdots \otimes U_n$ applies only to gates that act independently on each subsystem.

Gates that operate on different systems in parallel can be constructed in a similar way to a composite state as done in equation 2.5. The gate U_{tot} that applies U_i to system i of n systems, can be constructed by equation 2.16. U_{tot} distributes over $|\Psi\rangle = |\psi_1\rangle \otimes |\psi_2\rangle \otimes \cdots \otimes |\psi_n\rangle$ as shown in equation 2.17 [1, p. 71–75].

$$U_{tot} = U_1 \otimes U_2 \otimes \cdots \otimes U_n \tag{2.16}$$

$$U_{tot} |\Psi\rangle = U_1 |\psi_1\rangle \otimes U_2 |\psi_2\rangle \otimes \cdots \otimes U_n |\psi_n\rangle \tag{2.17}$$

There is a group of multi-qubit gates that use so called *control qubits*. One of these gates is the *CNOT* gate. It uses one control qubit and one target qubit and can be described as: if the control qubit is $|1\rangle$ then apply the X gate to the target qubit, shown in equation 2.18 with its matrix representation where the first qubit is the control qubit. This gate closely resembles a classical *XOR* gate with an additional pass-through for one of its inputs.

$$CNOT \equiv \quad \equiv |a\rangle |b\rangle \longrightarrow |a\rangle |a \oplus b\rangle \equiv \begin{cases} |00\rangle \longrightarrow |00\rangle \\ |01\rangle \longrightarrow |01\rangle \\ |10\rangle \longrightarrow |11\rangle \\ |11\rangle \longrightarrow |10\rangle \end{cases} \equiv \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{2.18}$$

The *CNOT* gate can be written as a sum of tensor products, as expressed in equation 2.19, where $\langle\psi| \equiv |\psi\rangle^\dagger$ [2, p. 63]. The number of terms in expressions of this form for an arbitrary controlled gate with c control bits will equal 2^c. For example as seen in the expression for a *CNOT* gate with two control bits in equation 2.20 [2, p. 65]. An arbitrary controlled n qubit gate CU with an arbitrary number of control bits and and target bits can be constructed via the formula shown in equation 2.21.

$$CNOT \equiv |0\rangle \langle 0| \otimes I + |1\rangle \langle 1| \otimes X \tag{2.19}$$

$$\begin{aligned} \text{C-}CNOT \equiv |0\rangle \langle 0| \otimes |0\rangle \langle 0| \otimes I + |0\rangle \langle 0| \otimes |1\rangle \langle 1| \otimes I + \\ + |1\rangle \langle 1| \otimes |0\rangle \langle 0| \otimes I + |1\rangle \langle 1| \otimes |1\rangle \langle 1| \otimes X \end{aligned} \tag{2.20}$$

$$CU = \bigotimes_{i=1}^{n} G(i) - \bigotimes_{i=1}^{n} O(i) + \bigotimes_{i=1}^{n} T(i) \tag{2.21}$$

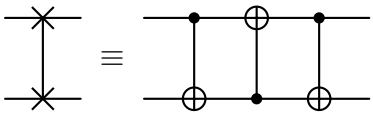
$$G(i) = \begin{cases} |0\rangle\langle 0| + |1\rangle\langle 1|, & \text{If qubit } i \text{ is control} \\ I, & \text{Otherwise} \end{cases}$$

$$O(i) = \begin{cases} |1\rangle\langle 1|, & \text{If qubit } i \text{ is control} \\ I, & \text{Otherwise} \end{cases}$$

$$T(i) = \begin{cases} |1\rangle\langle 1|, & \text{If qubit } i \text{ is control} \\ U_i, & \text{If qubit } i \text{ is target} \\ I, & \text{Otherwise} \end{cases}$$

$$U_i : \text{ Unitary } 2 \times 2 \text{ matrix}$$

Another multi-qubit gate is the *SWAP* gate, which swaps the state of two qubits. The *SWAP* gate is shown in equation 2.22 where it is also shown that the *SWAP* gate can be created via three *CNOT* gates. A derivation of this relation is shown in equation 2.23 [1, p. 23].

$$SWAP \equiv$$  (2.22)

$$\begin{aligned} |a, b\rangle &\rightarrow |a, a \oplus b\rangle \\ &\rightarrow |a \oplus (a \oplus b), a \oplus b\rangle = |b, a \oplus b\rangle \\ &\rightarrow |b, (a \oplus b) \oplus b\rangle = |b, a\rangle \end{aligned} \tag{2.23}$$

2.3.2 Applying quantum gates

As seen in section 2.2 the state of a system with n qubits can be represented by a 2^n component vector, and as seen in section 2.3.1 any gate can be represented by a $2^n \times 2^n$ matrix. The application of an arbitrary gate U on a system with an arbitrary state vector $|\Psi\rangle$ is then represented via the multiplication $U \cdot |\Psi\rangle$ [1, p. 18]. Using the rules of linear algebra this multiplication will produce an altered 2^n component vector which represents the altered system state.

In section 2.3.1 it was shown that some gates can be written as a sum of tensor-product terms built from 2×2 matrices. Equation 2.25 uses this representation and

distribution properties between matrix multiplication and tensor products to show how the three-qubit state vector is altered when applying a single-qubit gate. As can be seen, only the amplitudes corresponding to basis states that differ on the target qubit are updated. Thus, to apply a single-qubit gate one does not need to form the entire $2^n \times 2^n$ matrix; it is enough to update the affected amplitude pairs.

$$\begin{aligned} U \cdot |\Psi\rangle &= (I \otimes I \otimes U_{2\times 2}) \cdot (|\psi_1\rangle \otimes |\psi_2\rangle \otimes |\psi_3\rangle) \\ &= (|\psi_1\rangle \otimes |\psi_2\rangle \otimes U_{2\times 2} |\psi_3\rangle) \end{aligned} \tag{2.24}$$

In equation 2.25 it is shown that even for the arbitrary state vector it is true that the application of a single qubit gate on a system will not require the calculation of the entire $2^n \times 2^n$ matrix representing the gate. As shown in the equation it is enough to apply the single-qubit gate on pairs of basis states where only the target qubit z differs. In the equation an example with three qubits is used but the method will generalize to $n \in \mathbb{N}$ qubits. The method can also be accommodated to respect control qubits by only applying the single-qubit gate when control qubits x, y, etc. all equal 1.

$$\begin{aligned} U \cdot |\Psi\rangle &= (I \otimes I \otimes U_{2\times 2}) |\Psi\rangle = \sum_{x,y,z\in\{0,1\}} (I \otimes I \otimes U_{2\times 2}) \alpha_{xyz} |xyz\rangle \\ &= \sum_{x,y,z\in\{0,1\}} |xy\rangle \otimes U_{2\times 2} \alpha_{xyz} |z\rangle \\ &= \sum_{x,y\in\{0,1\}} |xy\rangle \otimes \Big(U_{2\times 2} \alpha_{xy0} |0\rangle + U_{2\times 2} \alpha_{xy1} |1\rangle \Big) \\ &= \sum_{x,y\in\{0,1\}} \Big(\big(u_{00}\alpha_{xy0} + u_{01}\alpha_{xy1} \big) |xy0\rangle + \big(u_{10}\alpha_{xy0} + u_{11}\alpha_{xy1} \big) |xy1\rangle \Big) \end{aligned} \tag{2.25}$$

The method shown in equation 2.25 can be used to apply the *SWAP* gate via a combination of *CNOT* gates. Another method would be to swap the coefficients of the basis states specified by the *SWAP* gates target bits. If the *SWAP* gates has control bits then the coefficients should only be swapped if all the control bits equal 1.

2.4 Projective measurements

For a state $|\psi\rangle$ and an outcome m, let B be the set of possible basis states given the outcome m occurred, then the state $|\psi'\rangle$ after measurement with outcome m is obtained by equation 2.26. Note that the denominator is equal to the square root of the probability for m to occur. [1, p. 87–90] This division ensures that $|\psi'\rangle$ satisfies the constraint in equation 2.4 which is known as the normalization condition.

$$|\psi'\rangle = \frac{P_m |\psi\rangle}{\sqrt{\langle\psi| P_m |\psi\rangle}} = \frac{P_m |\psi\rangle}{\sqrt{\sum_i |(P_m |\psi\rangle)_i|^2}}, \quad P_m := \sum_{|b\rangle \in B} |b\rangle \langle b| \tag{2.26}$$

What follows is an example of a projective measurement.

$$|\psi\rangle = \frac{1}{\sqrt{2}} |00\rangle + \frac{1}{2} |01\rangle + \frac{1}{2} |11\rangle \qquad m = \text{"First qubit measured to 0"}$$

$$\implies P_m = |00\rangle \langle 00| + |01\rangle \langle 01| \qquad \implies |\psi'\rangle = \sqrt{\frac{2}{3}} |00\rangle + \frac{1}{\sqrt{3}} |01\rangle$$

2.5 Quantum algorithms

Quantum gates can be combined to create quantum circuits. Diagrams of such circuits are executed from left to right, each line represents a qubit. Due to the probabilistic nature of qubits, getting a useful answer from running a quantum circuit requires a well thought out procedure, as is also true for classical algorithms. Over the years, quantum algorithms have been developed that solve certain problems faster than the best-known classical algorithms for those problems.

2.5.1 Hadamard-CNOT entanglement

The Hadamard-CNOT circuit shown in figure 2.2 is an algorithm to produce the bell state shown in equation 2.7. A particular result of the bell state is what happens when a qubit is measured. If one qubit is measured to be $|0\rangle$, the other qubit must also measure to $|0\rangle$ and the same applies for $|1\rangle$. In other words, we can know the result of measuring the second qubit just by measuring the first. This demonstrates entanglement [1, p. 15–16].

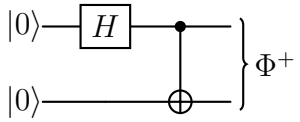


Figure 2.2: Hadamard-CNOT entanglement circuit

2.5.2 Phase kickback

Consider the circuit below, which demonstrates the phenomenon called phase kickback. When a controlled-gate has a target qubit which is an eigenvector of the gate's matrix representation, the control qubit gains relative phase while the target qubit is unaltered [5, p. 350–352]. By picking an eigenvector $|x\rangle$ to a controlled gate U, with

some eigenvalue $\lambda = e^{i\phi}$, the phase will be *kicked back* to the control qubit which means that the control qubit will change but the target qubit will be unaffected. A circuit that exhibits phase kickback is shown in figure 2.3.

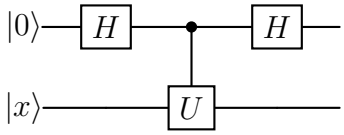


Figure 2.3: Phase kickback circuit

2.5.3 Superdense coding

Superdense coding, shown in figure 2.4, is a protocol that allows for sending two bits of classical information with only one qubit [6]. Alice and Bob share an entangled pair of qubits in the state $|\phi^+\rangle$. When Alice wants to send two classical bits c and d, she will perform a Z gate if $d = 1$, and an X gate if $c = 1$. She then sends the qubit to Bob who performs a $CNOT$ and a H gate. To retrieve the classical bits d and c, Bob measures the received qubit and his own original qubit.

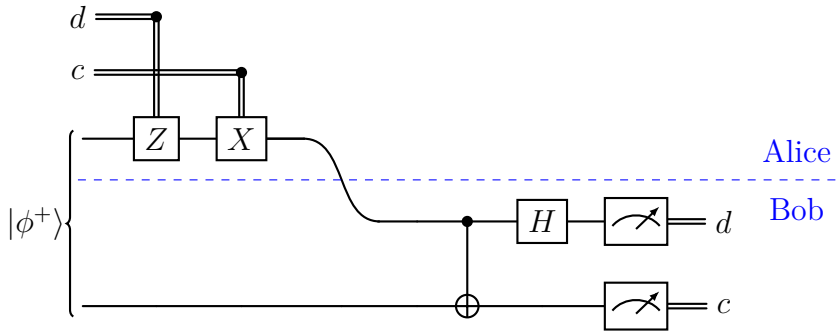


Figure 2.4: Superdense coding protocol [6]

2.5.4 Oracles

Consider any boolean function: $f : B^n \to B$, where $B = \{0, 1\}$. A quantum oracle is a a quantum gate that computes the function for some input. Interestingly, the quantum oracle can take a superposition as input, and return another superposition where f has been applied to all bases in said superposition.

When an implementation is discussed, the usual convention is that the oracle flips a designated target qubit when f evaluates to 1, and otherwise leaves the target qubit unchanged.

Algorithms that incorporate a quantum oracle usually disregard implementation

details about the oracle. However, since other algorithms in this thesis use quantum oracles, an explanation of implementation is justified.

Implementing boolean functions in classical hardware is traditionally done by constructing a *sum of products* or *product of sums* expression from the function's truth table, and then laying out a network of OR-gates and AND-gates that corresponds to the expression. For quantum computers, it is easier to construct a circuit that implements a quantum oracle by using an expression in *algebraic normal form*.

Algebraic normal form is a boolean expression that consists of an XOR-sum of products. This is especially well fitted to a sequence of $CNOT$-gates, since a $CNOT$-gate with control qubits q_1, q_2 and target qubit q_0 is essentially an application of an X-gate conditioned on $q_1 \wedge q_2$, under the assumption that all involved qubits are in a computational basis state. An example with an oracle $f(a, b) = a \wedge b$ is shown in figure 2.5 .

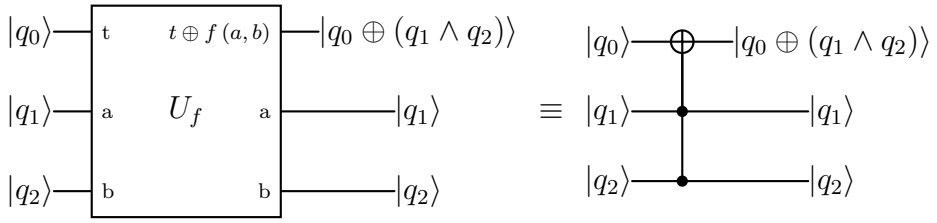


Figure 2.5: A quantum oracle gate implementation for $f(a, b) = a \wedge b$ consists of a single $CNOT$-gate

In the case of any qubit not being in a computational basis state, this implementation still works. Since U_f is a quantum gate, or rather the product of a sequence of quantum gates, it must distribute over superpositions:

$$|q_0 q_1 q_2\rangle = |0++\rangle \implies U_f\,|0++\rangle = U_f\left(\frac{1}{2}\left(|000\rangle + |001\rangle + |010\rangle + |011\rangle\right)\right) \tag{2.27}$$

$$= \frac{1}{2}\left(U_f\,|000\rangle + U_f\,|001\rangle + U_f\,|010\rangle + U_f\,|011\rangle\right) \tag{2.28}$$

$$= \frac{1}{2}\left(|000\rangle + |001\rangle + |010\rangle + |111\rangle\right) \tag{2.29}$$

If the target qubit was also $|+\rangle$ in this example, performing an X-gate on it is pointless since $X\,|+\rangle = |+\rangle$. If f is satisfied by the input, the target qubit will always be transformed to whatever the X-gate transforms it into.

2.5.5 Deutsch's algorithm

Deutsch's algorithm uses a phase oracle U_f to determine whether some single bit boolean function f is constant or balanced. A 1-bit boolean function is considered balanced if $f(0) \neq f(1)$, otherwise it is constant.[5, p. 341] A classical algorithm requires that the function f needs to be evaluated for both $f(0)$ and $f(1)$, whereas Deutsch's algorithm only needs to run once [7].

If $f(x)$ is constant $False$ then $y \oplus f(x) = y$ and U_f may seem like a two-qubit input gate, but in fact is just the identity transformation for y [5, p. 342]. Similarly, if $f(x)$ is constant $True$ then $y \oplus f(x) = \overline{y}$, and instead of identity the X-gate is applied to y.

If $f(x)$ is balanced it can be either the identity function or the *not*-operator. If it is the identity function, then $y \oplus f(x) = \overline{y}$, but only if x is 1, that is the X-gate is applied to y. To apply the X-gate only when x is 1, we let U_f equal the $CNOT$-gate, a controlled X-gate.

If $f(x)$ is the *not*-operator, that is technically the same as the identity function after inverting the control bit x. This may seem obvious, but it reveals the value of U_f. Let U_f be the $CNOT$-gate like the previous example, but after an X-gate is applied to x. Both gates are in this case matrices in $\mathbb{C}^2$, and U_f can be simplified to the matrix product of these gates.

The circuit, shown in figure 2.6, allows inverting the phase of y in the case of $f(x)$ evaluating to $True$.

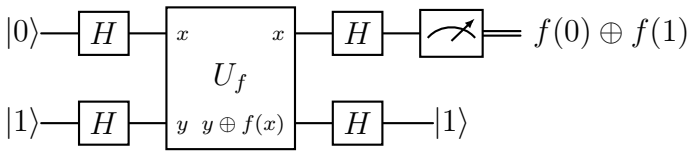


Figure 2.6: The Deutsch circuit [1]

2.5.6 Deutsch-Jozsa circuit

The Deutsch-Jozsa algorithm is an n-bit generalization of Deutsch's algorithm and therefore works in the same way but with n input-bits. In this generalization, f is considered balanced if half of the function's values are 1 and the other 0. The function is constant if $f(x) = 0$ or $f(x) = 1$. The best classical implementation that solves this problem evaluates f at $2^{n-1} + 1$ distinct points and checks if the output evaluates to the same. The worst time complexity for such an algorithm

is exponential. The Deutsch-Jozsa algorithm however, solves the problem in $O(1)$ time [5, p. 343–345]. The circuit for this algorithm is depicted in figure 2.7, which reveals the similarities with Deutsch's algorithm.

The algorithm works by applying Hadamard gates on the first n qubits, as well as adding an extra qubit in the $|0\rangle - |1\rangle = |-\rangle$ state. The state after the first set of Hadamard gates is

$$|\psi_1\rangle = \sum_{x\in\{0,1\}^n} |x\rangle\,|-\rangle \tag{2.30}$$

after U_f the state is

$$|\psi_2\rangle = \sum_{x\in\{0,1\}^n} (-1)^{f(x)}\,|x\rangle\,(|0\rangle - |1\rangle) \tag{2.31}$$

and after the last set of Hadamard gates the state is

$$|\psi_3\rangle = \sum_{x,y\in\{0,1\}^n} (-1)^{f(x)\oplus(x\cdot y)}\,|y\rangle\,|-\rangle \tag{2.32}$$

The amplitude of state $|00\cdots0\rangle$ is

$$\sum_{x\in\{0,1\}^n} \frac{(-1)^{f(x)}}{2^n} \tag{2.33}$$

If $f(x)$ is constant, the amplitude has a 100% probability of being ±1 whereas if $f(x)$ is balanced, it has a 100% probability of being 0. Therefore, only one measurement of the first n qubits is needed to determine whether $f(x)$ is balanced or constant.

The measurement result tells us whether the function is constant or balanced: the all-zero outcome corresponds to a constant function, while any other outcome corresponds to a balanced function.

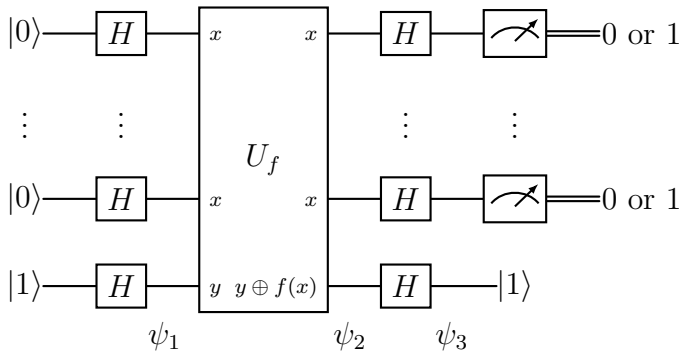


Figure 2.7: The Deutsch-Jozsa circuit

2.5.7 Bernstein-Vazirani algorithm

Given a secret, unknown, string $s = s_0 \cdots s_{n-1}$ and a function $f : \{0,1\}^n \to \{0,1\}$ defined as:

$$f(x) = \sum_{i=0}^{n-1} s_i x_i \mod 2$$

the Bernstein-Vazirani algorithm, shown in figure 2.8, finds the secret string s in constant time, whereas a classical computer needs n queries [8, p. 103–104].

As the similarity between figures 2.7 and 2.8 reveals, the Bernstein-Vazirani algorithm works similarly to Deutsch-Jozsa. To construct the oracle U_f, a $CNOT$ gate is added with q_i as the control qubit and the extra qubit q_a as the target qubit if $s_i = 1$, for all s_i in s. This will result in

$$|\psi_2\rangle = \frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} (-1)^{(s \cdot x)} |x\rangle |-\rangle \tag{2.34}$$

Then, after applying the Hadamard gates, the state is

$$|\psi_3\rangle = H^{\otimes n} \left(\frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} (-1)^{s \cdot x} |x\rangle \right) \otimes (H |-\rangle) \tag{2.35}$$

This can be simplified to

$$|\psi_3\rangle = |s\rangle \otimes |1\rangle \tag{2.36}$$

from which it is possible to retrieve s after measuring the first n qubits [8, p. 103–104].

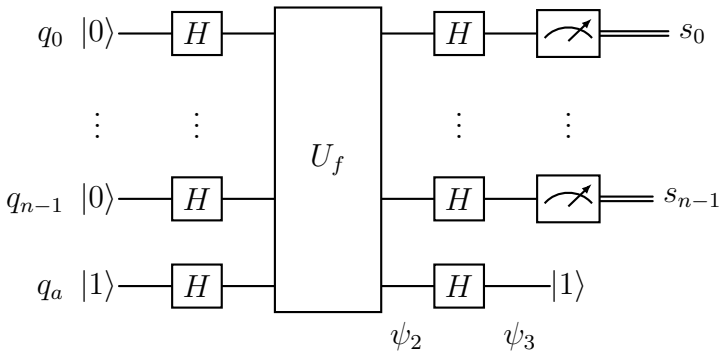


Figure 2.8: Bernstein-Vazirani algorithm [8, p. 103–104]

2.5.8 Grover's algorithm

Grover's algorithm is a quantum search algorithm that finds a (typically unique) input x_0 for which a function f evaluates to a specified value. Classically this search requires $O(N)$ evaluations in the worst case, whereas Grover's algorithm finds the marked item with high probability using $O(\sqrt{N})$ oracle calls [9].

The definition of the algorithm is that, given a function $f : \{0, ..., N-1\} \to \{0, 1\}$ where $N = 2^n$ for some natural number n. There exists an x_0 such that:

$$f(x) = \begin{cases} 1, \text{ if } x = x_0 \\ 0, \text{ otherwise} \end{cases}$$

The goal of the algorithm is to identify x_0. Were this a classical problem, then the most efficient way to find x would be to step through all of the combinations. However, because of quantum parallelism, Grover's algorithm finds x_0 with a probability greater than or equal to $p = 1 - \frac{1}{N}$ in $\frac{\pi}{4}\sqrt{N}$ steps [1, p. 253–260]. This also means that Grover's algorithm is non-deterministic, since it isn't guaranteed that it will return a correct value.

The algorithm takes N and f as inputs and outputs the bit string $x_0 = i_1, ..., i_n$. A Hadamard gate is applied to all qubits to create a uniform superposition before the loop is applied $\frac{\pi}{4}\sqrt{N}$ times [10].

The algorithm's loop utilizes the oracle U_f where:

$$U_f = \sum_x (-1)^{f(x)} |x\rangle \langle x|$$

Written as a circuit diagram, the oracle is implemented according to figure 2.9. By

multiplying the matrices, it can be shown that the Hadamard-CNOT construction can be rewritten as a controlled Z gate.

$$\begin{aligned} HXH &= \frac{1}{2}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}\begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix} \\ &= \frac{1}{2}\begin{pmatrix}1 & 1\\ -1 & 1\end{pmatrix}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix} \\ &= \frac{1}{2}\begin{pmatrix}2 & 0\\ 0 & -2\end{pmatrix} \\ &= Z \end{aligned}$$

The final oracle circuit also inverts the usage of the X gates to instead target qubits whose corresponding bit i has the value 0 as opposed to targeting bits with the value 1 and applying X gates on all qubits.

The oracle u_f marks a correct bit by inverting its sign, for example, given that $x_0 = 11$ and $\psi_0 = \frac{1}{2}\begin{bmatrix}1\\1\\1\\1\end{bmatrix}$ then $\psi_1 = u_f\psi_0 = \frac{1}{2}\begin{bmatrix}1\\1\\1\\-1\end{bmatrix}$

Which as seen, flipped the sign of the state $|11\rangle$, indicating that $|11\rangle$ was the correct state.

After marking correct states, the diffusion operator g is applied, where:

$$g = 2\,|d\rangle\,\langle d| - I_n \tag{2.37}$$

The purpose of which is to amplify the probability of collapsing into a correct state. The circuit for the diffusion operator g seen in figure 2.10 shows a setup using a controlled Z gate instead of a regular Z gate and CNOT setup.

With the oracle U_f and the diffusion operator g, Grover's algorithm is expressed in figure 2.11 as a circuit diagram.

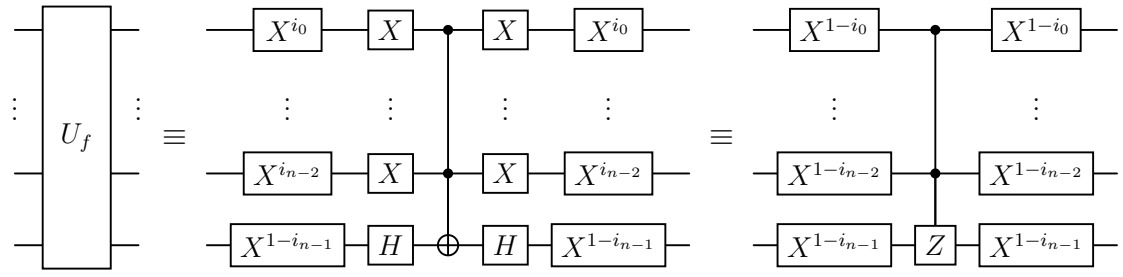


Figure 2.9: The oracle used in Grover's algorithm [11]

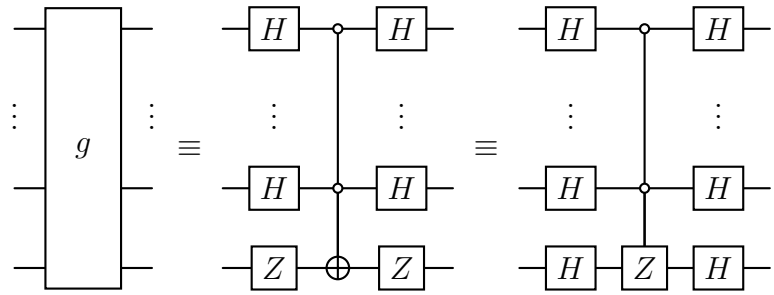


Figure 2.10: The diffusion operator used in Grover's algorithm [11]

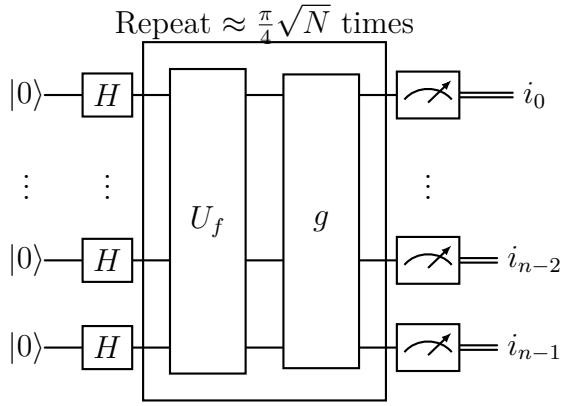


Figure 2.11: Grover's algorithm [11]

2.5.9 Shor's algorithm

Shor's algorithm is a factorization algorithm that takes in an integer and decomposes it into its prime factors. It utilizes quantum parallelism and entanglement to do this in polynomial time, whereas the best known classical counterparts run in exponential time. Given a sufficiently large quantum computer, it will be capable of breaking cryptographic methods that are used in modern society such as RSA and Diffie-Hellman key exchange [11].

The theoretical aspects and implementations used in this section are based on [11] and [12], unless otherwise specified.

Given that a composite natural number N exists, it follows that $N = n_1 n_2$ where n_1 and n_2 are also natural numbers such that $1 < n_1, n_2 < N$. The goal is to determine n_1, which can then be used to determine n_2 by calculating N/n_1.

A uniformly random integer a is picked such that $1 < a < N$.

Before continuing, N and a are checked for trivial solutions.

If:

1. $2 \mid N$, return 2.
2. $N = p^v$ where p is prime and $v \in \mathbb{N}$, return p.
3. $gcd(a, N) > 1$, return $gcd(a, N)$.

The next step is to run the Quantum Phase Estimation (QPE) algorithm with a and N as inputs to estimate an order r of the input a such that $a^r \equiv 1 \bmod N$.

The QPE implementation that is used in the project utilizes mid-circuit measuring to reduce the amount of qubits needed for the algorithm. By using this method, the amount of qubits needed to store the output value was reduced from n qubits to 1 qubit, where n is the minimum amount of bits needed to represent the number N.

The QPE relies heavily on the unitary operator U_a defined by: $U_a |x\rangle \rightarrow |(ax) \bmod N\rangle$.

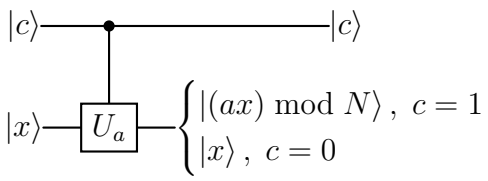


Figure 2.12: Unitary operator U_a used in the QPE [12]

The unitary operator U_a is composed of many underlying gates that will be covered in the order of lowest level to highest level. These gates mainly utilize two registers to compute their respective transformations, the input register $|x\rangle$ and the computational register $|y\rangle$.

The Quantum addition algorithm gate (ADDER) performs the transformation $|\phi(y)\rangle \rightarrow |\phi(y + a)\rangle$. The ADDER uses the conditional phase shift:

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

As such, the ADDER expects y to be a Quantum Fourier Transformed (QFT) register $\phi(y)$.

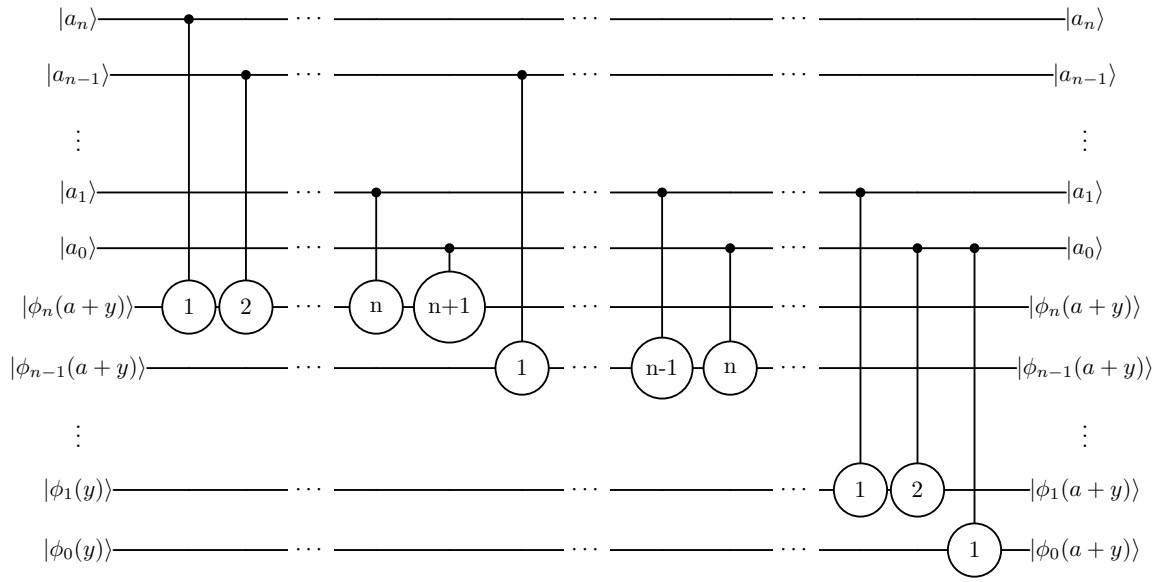


Figure 2.13: Draper's Quantum addition algorithm [13]

The circuit shown in figure 2.13 shows the input register $|\phi(y)\rangle$ and classical register a using $n+1$ qubits. This is due to the most significant qubit $|\phi_0(y)\rangle$ being used as an overflow bit. This will be important in higher level gates.

The ADDER is then incorporated into the Modular Addition Algorithm (MODADDER) to compute the sum modulo an integer N.

The MODADDER is defined by: $|\phi(y)\rangle \rightarrow |\phi((y+a) \bmod N)\rangle$. Like the ADDER, the MODADDER expects an input that has been transformed by the Quantum Fourier Transformation Algorithm.

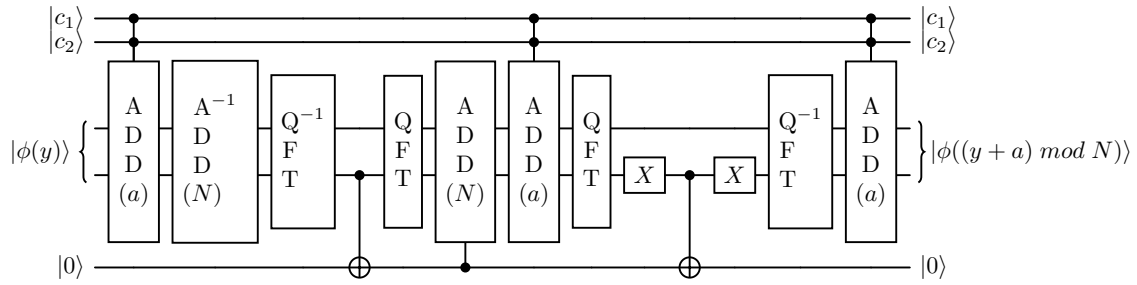


Figure 2.14: The modular addition algorithm [12]

Figure 2.14 shows the implementation for the modular addition algorithm used in the project. The implementation uses the previously described ADDER in combination with the QFT to produce the desired transformation. Gates that are superscripted with "-1" represent the inverse operation of that gate.

The 3rd and 4th wire in the algorithm are part of the same register but are treated individually. The 4th wire represents an overflow qubit which is crucial for the operation of the MODADDER. The 5th wire is a qubit that is only acted upon by the overflow qubit. It's value is always returned to $|0\rangle$ and as such it can be treated as being part of the MODADDER gate.

The transformation on $|\phi(y)\rangle$ is dependent on $c_1 = 1, c_2 = 1$. As such, if either $c_1 = 0$ or $c_2 = 0$ then the output is $|\phi(y)\rangle$.

Using the MODADDER gate, the CMULT gate can be created. The CMULT gate performs the transformation $|x\rangle |y\rangle \rightarrow |x\rangle |(y + ax)\ mod\ N)\rangle$. It achieves this transformation by using the identity:

$$(ax)\ mod\ N = (...((2^0 ax_0)\ mod\ N + 2^1 ax_1)\ mod\ N + ... + 2^{n-1} ax_{n-1})\ mod\ N \text{ [12]}$$

Thus, the implementation for the CMULT gate can be seen implementing this identity in figure 2.15.

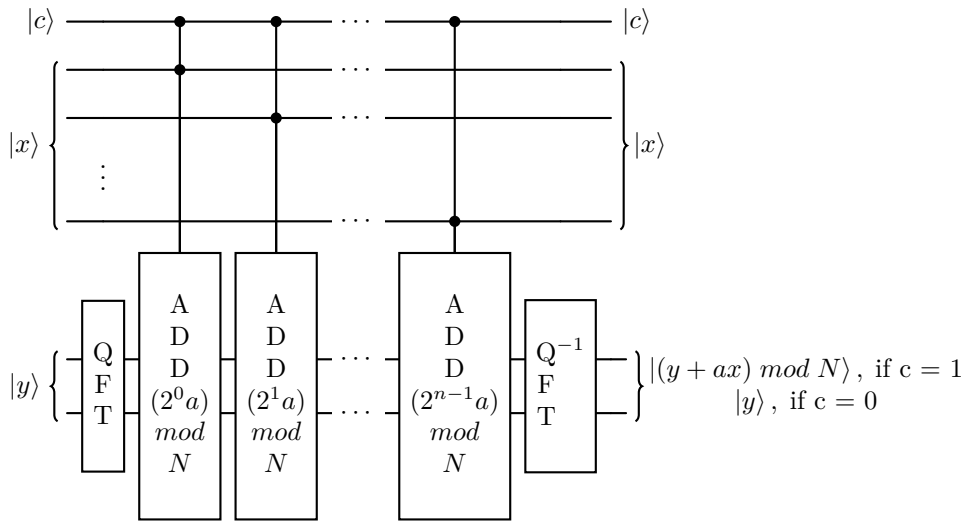


Figure 2.15: The Controlled Multiplication Algorithm [12]

The circuit for the CMULT gate, in addition to using adder gates, applies the QFT on $|y\rangle$ in the beginning and end of the circuit. This is due to the MODADDER, as mentioned earlier, requiring the input to be a Quantum Fourier Transformed register.

With the underlying components explained, the U_a gate can be expressed with the

following implementation:

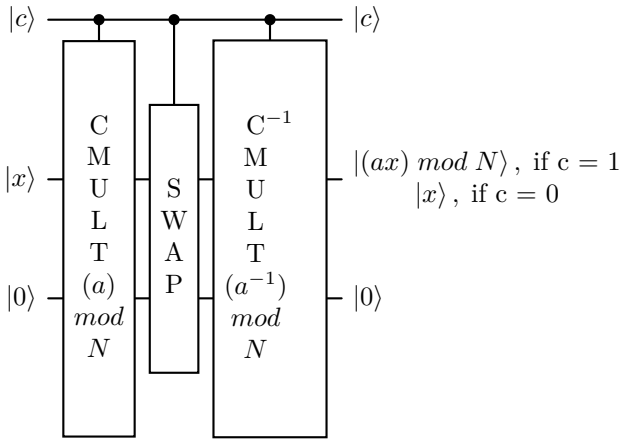


Figure 2.16: The U_a Algorithm [12]

The U_a implementation shown in figure 2.16 uses a SWAP gate in addition to the CMULT gates. This gate simply swaps the top and bottom registers and can be expressed through the transformation $|x\rangle |y\rangle \rightarrow |y\rangle |x\rangle$. Using this and the following inverse CMULT gate, the bottom register is always reset to $|0\rangle$ and can therefore be considered a part of the U_a gate. The inverse CMULT gate also uses a^{-1} as parameter, which is the modular multiplicative inverse of a with respect to N.

With the U_a gate clarified, the final circuit for the QPE can be seen in figure 2.17.

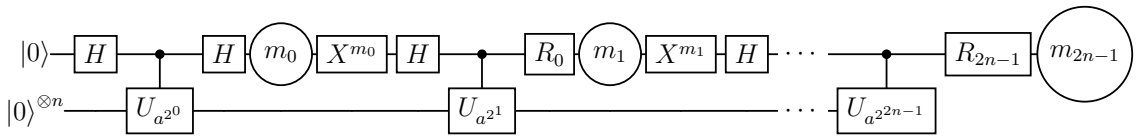


Figure 2.17: The Quantum Phase Estimation Algorithm [12]

The top and bottom registers are initialized with $|0\rangle$ and $|0\rangle^{\otimes n}$ respectively. After measuring a bit, an X gate is applied on the top register if that bit was measured as a 1. The R gates applied are based on the previous measurements and implement the inverse QFT.

The result is then acquired by assembling the measurements as a bit string m.

Run the continued fraction algorithm of m/n and find a denominator r where $1 < r < N$, that satisfies the following conditions:

1. r is even

2. $a^{r/2} \not\equiv -1 \; mod \; N$

Condition 1. relies on the fact that N is constricted to only be composed of natural numbers. Which means the following equation can only produce a solution if r is even.

$$a^r \equiv 1 \; mod \; N$$
$$(a^{r/2})^2 - 1 \equiv 0 \; mod \; N$$
$$(a^{r/2} - 1)(a^{r/2} + 1) \equiv 0 \; mod \; N$$

Condition 2. is a special case where the equation will return trivial factors of N.

By inserting $a^{r/2} \equiv -1 \; mod \; N$ into $(a^{r/2} + 1)(a^{r/2} - 1) \equiv 1$, the factors that would be produced are:

$$(a^{r/2} - 1)(a^{r/2} + 1) \equiv 0 \; mod \; N$$
$$(-2)(0) \equiv 0 \; mod \; N$$
$$\Leftrightarrow$$
$$gcd(-2, N) = \{1, 2\}, \; gcd(0, N) = N$$

If all conditions apply, then $n_1 = gcd(a^{r/2} + 1, N)$ and $n_2 = N/n_1$. Otherwise, rerun the QPE algorithm with a different a.

3

Purpose

The purpose of the project is to build a Rust library for construction, execution and inspection of quantum circuits, and to implement several interchangeable simulators within that library. This library is intended to support experimentation with simulator design choices and to expose debugging and inspection facilities that make quantum algorithms easier to study.

The project further aims to design and evaluate how different simulator implementations influence performance across various quantum circuits.

The main contributions of the project are support for hybrid quantum-classical circuits through classical registers and expression-based control flow, a terminal debugger for stepwise inspection of circuit execution, and several interchangeable simulator backends suited to different circuit characteristics.

4

Problem

The task is to build a quantum computer simulator that can take a description of a circuit as an input and produce an expected result from that circuit. The target users are students aiming to learn about quantum algorithms and researchers with the goal of testing quantum algorithms. Therefore, the interface exposed to users should be intuitive and easy to experiment with, while still providing an extensive toolkit for simulating quantum circuits.

4.1 Circuit description

In order to provide the tools necessary for a user to simulate a quantum algorithm, there needs to be some way for the user to input a description of their quantum circuit. For a user that doesn't have much programming knowledge and is just interested in how a quantum computer works, a graphical interface would be the most simple method for describing an algorithm. For someone with experience in programming of quantum computer, an existing programming language like OpenQASM dedicated to describing circuits in an "assembly-like" style could be preferable. Since Quasim is a quantum computer simulation library for Rust, an intuitive way to describe circuits directly in Rust code is needed.

4.2 Debugging a circuit

A way of running and debugging a circuit is of great value for potential users. The debugging process would work by letting the user step through each operation in the specified quantum circuit. At each step the user would be able to inspect the current quantum state. Since the state space of the system grows exponentially with the number of qubits simulated, it becomes difficult, and in some sense meaningless, to output the whole state vector. Instead, options to view the amplitude of certain states specified by the user or to show a list of the most probable states are reasonable.

4.3 Classical control-flow

Some quantum algorithms rely on being able to perform classical control-flow branching based on previously measured qubits. A common usecase of this is for error correction in quantum circuits. In order for simulators to support this, they require an implementation which integrates both classical and quantum logic.

4.4 Getting the result

Running a simulation of a specified quantum circuit will yield some result. The user may expect to be able to read parts of or the full result of the performed computation. Therefore there should exist tools for reading the collapsed result of the whole circuit or specific qubits. For circuits which contain classical registers, the user should be able to read contents of registers after running the circuit.

5

Scope

This section outlines the boundary of the project and limitations that are intentionally excluded from the implementation of Quasim.

5.1 Simulation model

Quasim focuses on ideal, pure-state quantum circuit simulation. Gates are treated as exact unitary operations and measurements as projective measurements. Mixed states, density-matrix simulation, decoherence, and realistic noise are outside the scope of the project.

5.2 Circuit description

The primary interface for constructing circuits is the Rust circuit builder API. Limited OpenQASM parsing is included, but full OpenQASM support is outside the scope of the project.

The library supports common single-qubit gates and controlled variants available with an arbitrary number of control bits. The only built in multi-qubit target gate is the SWAP gate due to its ubiquity.

5.3 Hybrid quantum-classical simulation

Building a hybrid system was deemed to be feasible, and became a core goal of the project. The main idea behind quantum-classical hybrid computation is using measured qubits as controls in following operations. This means support for mid-circuit measurement and storing of measurements to classical registers, as well as classical control-flow based on previous measurements. General classical computations outside of circuit execution is left to the Rust program using the library.

5.4 Graphical user interface

Building a graphical interface that can further the understanding of quantum algorithms was considered a stretch goal. A prototype was built under a separate repository, and is not covered by the scope of this report. The prototype consists of a backend which manages the simulation state using the Quasim library and exposes a web-facing API for communication with the frontend. The web based frontend was created using generative AI tools and exists as a proof of concept and showcase of how the Quasim library can be used. The prototype is available here [14].

6

Methodology

Building a working simulation for a quantum computer requires a way to construct a quantum circuit and then execute it. The quantum circuit should take an arbitrary amount of qubits, which are then manipulated using quantum gates.

The development of a complex software project requires more than just programming knowledge, it also requires a good workflow to ensure effective contributions from multiple group members.

6.1 Project workflow

Github was used to host and manage the simulator's code, as well as providing a kanban [15] project. In kanban, tickets were created for something that needed to be done and group members would assign themselves to tickets they wanted to work on. As each group member created and worked on tickets related to previously worked on, this naturally led to some components being managed by a single group member.

A group meeting was held each Tuesday where we discussed what each group member has worked on, the overall state of the project, and planning of upcoming work. A lot of important technical decisions were taken during these meetings, as it was decided early on that we would decide the winner by popular vote from all group members.

A weekly meeting with the supervisor was also held where we briefly reported on the progress of the project, and any potential questions about the project or the mathematics behind it were asked. We also held weekly work days, where we all met up on campus and worked on the project. These consistent meetings led to smooth and steady progress on the project.

6.2 Testing

During development, unit tests were added to ensure the correctness of smaller parts of the simulators. Because Quasim is designed around interchangeable simulator backends, a module of common behavioral tests was defined and reused across multiple simulators. This made it possible to verify that different backends implemented the same observable circuit semantics.

To test the simulator as a whole, some widely studied and well known algorithms were implemented. Since some of them, such as Deutsch's algorithm, are deterministic [1, p. 33–34], the result of the simulator can be checked against an expected value.

Due to the nature of qubits, some algorithms are non-deterministic. These algorithms are often designed to maximize the probability of a favorable result, but they still have a small probability of resulting in a non-favorable result. Creating automated tests for theses poses a challenge, but analyzing values of the state vector can assert whether the algorithm has been successfully executed.

To test the performance of the simulator, the benchmarking tool `Divan` was used, which allowed for creating benchmarking modules. Modules were designed to mirror circuits of various character to get an overview of the simulators performance depending on the circuit. The performance of some of the well known algorithms were also using these benchmarking modules.

6.2.1 Hadamard-CNOT entanglement

Verifying that entanglement works correctly can be done by manually inspecting the state vector to confirm that the amplitude of the $|01\rangle$ and $|10\rangle$ states is 0.

6.2.2 Phase kickback

To construct a circuit that exhibits phase kickback, we know from section 2.5.2 that an eigenvector $|x\rangle$ to U with eigenvalue $\lambda = e^{i\theta}$ needs to be picked. A simple case would be the $R_z(\theta)$ gate from equation 2.15. The eigenvalue of this matrix can be calculated with the following formula $\det(\lambda I - R_z(\theta)) = 0$ which gives

$$\begin{vmatrix} \lambda - 1 & 0 \\ 0 & \lambda - e^{i\theta} \end{vmatrix} = 0 \Leftrightarrow (\lambda - 1)(\lambda - e^{i\theta}) = 0 \Rightarrow \lambda_1 = 1, \lambda_2 = e^{i\theta} \tag{6.1}$$

which is a desirable value for $|1\rangle$ since $R_z(\theta)\,|1\rangle = e^{i\theta}\,|1\rangle$. To test that phase kickback works we use the circuit in figure 6.1 and verify that the Bloch vector of the second

qubit is the same in state ψ_1 and ψ_2. Getting the Bloch vector for the second qubit from ψ_1 and ψ_2 can be done with equation 2.11.

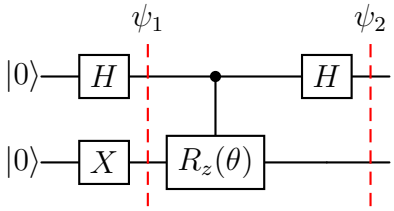


Figure 6.1: Test phase kickback circuit

6.2.3 Superdense coding

Testing the superdense protocol can be done with a simple circuit, shown in figure 6.2. Sending the qubit to another circuit isn't necessary, but one can imagine sending the qubit after the X gate. Verifying that two bits of information has been successfully sent with one qubit means checking that $c_0 = c_1$ and $d_0 = d_1$.

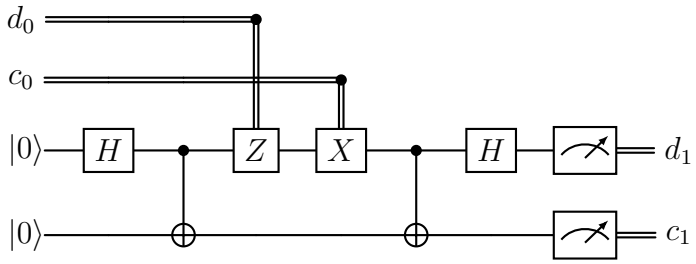


Figure 6.2: Circuit for testing superdense coding protocol

6.2.4 Deutsch's algorithm

To test the Deutsch algorithm, the circuit described in section 2.5.5 is built with an oracle for constant $False$, constant $True$, and balanaced function f. The classical check for which type f falls under is trivial and is used to compare with the quantum version to verify the result of Deutsch's algorithm.

6.2.5 Deutsch-Jozsa circuit

To test the Deutsch-Jozsa algorithm on our simulator, the oracle U_f needs to be constructed. We can use a similar approach from section 6.2.4 to generalize it for n qubits. For a constant function, we apply the same gate and for a balanced function we also apply a $CNOT$-gate with only the first qubit as a control since it will be a one for half of the states. The rest of the test is similar to the one conducted in section 6.2.4.

6.2.6 Bernstein-Vazirani algorithm

Testing the Bernstein-Vazirani algorithm on our simulator works similarly to the testing for Deutsch-Jozsa. To construct U_f we apply a $CNOT$-gate with control q_i and target q_a for all $i, j = 0 \dots n-1, i \neq j$ if $s_i = 1$ and $s_j = 0$. This works because if $s_i = 1$ and is equal to 0 for all other indices, then $f(x) = 1$ which means we apply a $CNOT$-gate as discussed in 2.5.5.

6.2.7 Grover's algorithm

To test Grover's algorithm, the circuit is built in the simulator using the diagrams seen in section 2.5.8. x_0 is given as a parameter for which the number of qubits n and the iteration variable N are calculated.

Because Grover's algorithm is non-deterministic, the algorithm runs 1000 times, each iteration returning either $true$ if x_0 was correctly identified or $false$ if not. Since the algorithm finds x_0 with a probability greater than or equal to $p = 1 - \frac{1}{N}$, the test will check if the amount of iterations that returned $true$ has a higher or equal ratio to p.

6.2.8 Shor's algorithm

Since Shor's algorithm is built from a diverse set of modular components, each of these components are tested to verify that their respective transformations are performed using the implementations covered in section 2.5.9.

The QPE is the main testing area as it is the core quantum component of Shor's algorithm. Due to the algorithm's probabilistic nature, it may not produce a correct result each iteration. Therefore, the test iterates *attempts* times, where *attempts* denotes the maximum number of iterations. If a correct result is produced, the test reports a success.

Since there exist a that does not produce a valid period for a number N, the default test runs using $N = 15$ and $a = 2$.

The complete implementation of Shor's algorithm is tested in similar way. An additional test is conducted using randomly chosen values of a.

6.3 Circuit

`Circuit` is the struct that represents a circuit to be executed. It contains a list of quantum gates, the total number of qubits, labels for use when jumping, break-

points for the debugger, registers for performing measurements, and subcircuits for embedding smaller circuits.

```
let circuit = Circuit::new(2)
    .h(0)
    .cx(&[0], 1);
```

Listing 6.1: Building a simple circuit

To construct a circuit to be used in a quantum algorithm, first the number of qubits to be used must be specified. *Quasim* comes with helper functions on the circuit to add gates. An example of building a simple circuit is shown in listing 6.1.

```
let circuit = Circuit::new(2)
    .new_reg("a", 1)             // Create a register with size 1
   bit
    .h(0).cx(&[0], 1)
    .measure_bit(1, ("a", 0))    // Measure qubit 1 and place result
    in a
    .z(0);                       // Qubit 0 still acts as a qubit
   after measurement
```

Listing 6.2: Measuring a qubit mid-circuit

It is also possible to perform mid-circuit measurements using either `measure_bit` or `measure_bits`. For `measure_bit`, the qubit to be specified, the register to put the measurement into, and the position of the register to place the measurement into must all be specified. For `measure_bits`, only the qubits and register name needs to be specified. It's important that the register has been added with `.new_reg` before measuring. An example of mid-measurements can be seen in listing 6.2.

```
let circuit = Circuit::new(2)
    .new_reg("b", 1)
    .h(0)
    .measure_bit(0, ("b", 0))        // Measure qubit 0 into b
    .jump_if(r("b").eq(0), "end")    // If qubit 0 is measured to be
    0, jump to "end"
    .y(1)                            // Will only be applied if b !=
    0
    .label("end");
```

Listing 6.3: Simple control flow

It is also possible to have simple control flow similar to that of a classical computer. Instructions like `jump_if` and `apply_if` take a *BoolExpr* as an argument. Expressions are explained in more detail in section 6.3.2. A circuit may also contain labels which are equivalent to the classical counterpart. An example of simple control flow using BoolExpr and labels is shown in listing 6.3.

```
let circuit = Circuit::new(2)
    .x(1)
    .new_sub_circuit("qft", Circuit::new_qft(2))  // Create a new
    subcircuit
    .call("qft", 0);                              // Call the
    subcircuit
```

Listing 6.4: Subcircuits

Quasim has support for creating and executing subcircuits in a circuit to make the circuit more readable and easier to work with, just like how we use functions in normal programming. Usage of a subcircuit is shown in listing 6.4 with a helper method to create a quantum fourier transform [16].

Users also have access to the function `fn Circuit::from_qasm_file(file_name)` which allows parsing an OpenQASM source file into the isomorphic `Circuit` instance. However, support for the full OpenQASM specification is out of scope for this project so only regular gate application sequences are interpreted appropriately in the final product.

6.3.1 Instructions

```
enum Instruction {
    Gate(Gate),  // Quantum gate
    MeasureBit(usize, (String, usize)),  // Measurement of a single
    qubit
    MeasureAll(String),  // Measurement of all qubits
    Jump(usize),  // Jump to location
    JumpIf(BoolExpr, usize),  // Conditional jump
    Assign(BitExpr, String),  // Assign a value to a register
    Call(String, usize, QBits),  // Execute a subcircuit
}
```

Listing 6.5: Instruction enum

An instruction is an enum with seven different values, shown in listing 6.5. `Gate` is a quantum gate; the available gates includes all gates from section 2.3, as well as the controlled variants. `MeasureBit`, `MeasureAll`, and `Assign` relates to registers, described in section 6.4.5. `Jump` and `JumpIf` represents jumps, and `Call` is used for subcircuits.

6.3.2 Expression DSL

In order to create classical quantum hybrid circuits which contain classical control flow which can be evaluated on measured qubits stored in registers, we need some

way to express how the control flow is evaluated. This is accomplished with the expression DSL.

```
let example_expr_1: BitExpr = (r("example_register") + rb("
    example_register", 0)) % 2;
let example_expr_2: BoolExpr = (r("example_register") * 5).gt(25);
```

Listing 6.6: Creating expressions

The expressions shown in listing 6.6 can be evaluated using a provided register file, which contains the register values to evaluate the expressions over.

```
let evaluated_expr_1: usize = example_expr_1.eval(&
    example_register_file);
let evaluated_expr_2: bool = example_expr_2.eval(&
    example_register_file);
```

Listing 6.7: Evaluating expressions

The DSL distinguishes between bit expressions, which evaluate to unsigned integer values, and boolean expressions, which evaluate to booleans. Boolean expressions are created from comparisons between bit expressions, for example seen in listing 6.6 where `gt` (greater than) is used. All boolean expression operators can be seen in the appendix table B2, and bit expression operators in appendix table B1. Together these two tables form the full specification of the expression DSL.

6.4 Simulator traits

There were several simulators implemented, each with their own capabilities. As such, Rust traits for specific capabilities were implemented [17, Chapter 10.2]. If a user wants to create their own simulator, the simulator needs to implement these traits.

6.4.1 `Buildable` trait

The `Buildable` trait means that a simulator instance can be created from a `Circuit` instance. When implemented, it will be possible to use the `build()` function which will try to create the simulator instance from a circuit.

6.4.2 `Simulator` trait

As there are capabilities that are assumed to exist for every single simulator, these were collected into a `Simulator` which serves as a base trait that all simulators

should implement. Here, functions to running the simulator, resetting the simulator, and retrieving the current state of the simulator exist.

The structure of the state must be defined by a simulator, which must implement a `QuantumState` trait. This trait requires a way to retrieve the amplitude for a given basis state, and a way to collapse the state. Both the simulator and `QuantumState` need type for the state, this can be anything but the convention in Quasim has been to use `Complex<f32>`.

6.4.3 `Debuggable` trait

As discussed in section 3, one of the main purposes was to make it easy for users to understand quantum algorithms. To achieve this, a `Debuggable` trait was implemented in order for the simulators to make it easy to understand quantum algorithms. The trait gives functions to step over gates one at a time and, if possible, go back. There is also a function to retrieve the current program counter and instruction.

6.4.4 `StoredCircuit` trait

It may be desirable to retrieve the circuit from a simulator, particularly if a program is built to be able to swap out simulators. The `StoredCircuit` trait gives a way to retrieve the circuit, mutate the circuit, the list of instructions, and the total number of qubits.

6.4.5 `StoredRegisters` trait

When a qubit is measured, it collapses to a value. This value needs to be stored and the `StoredRegisters` trait adds functions to retrieve a `RegisterFile` and a `Register`. This should be implemented for simulators that allow for classical operations. The `RegisterFile` allows simulators to evaluate expressions as described in section 6.3.2.

6.4.6 `Sampleable` trait

For some use cases, only a final sample from a circuit desired. To reduce the amount of boilerplate code and provide a generic API interface for sampling different parts of a circuit, the trait `Sampleable` was implemented. The trait exposes functions `sample_once` and `sample` for sampling n times, which takes a `Sampler` that specifies what to sample. For example, `RegisterSampler` specifies some register to sample.

6.5 State vector simulator

State vector simulation is a standard method for classically simulating quantum circuits [18]. Implementations of gate application functions adhere closely to the description found in section 2.3.2. Measurements on the state vector are implemented as shown in section 2.4.

We can use the method shown in equation 2.25 for implementing a function for applying an arbitrary single-qubit gate to a state vector. The method in that section assumes a system with three qubits, but we can generalize to an arbitrary number of qubits. Given an n qubit system, we define $r \in \{0,1\}^{n-1}$ as a bitstring of all qubits with target qubit t excluded. Let $r0_t$ and $r1_t$ mean the n-bit basis states with 0 and 1 respectively inserted at the position of qubit t. Generalizing 2.25, we get:

$$U|\Psi\rangle = \sum_{r\in\{0,1\}^{n-1}} \left(u_{00}\alpha_{r0_t} + u_{01}\alpha_{r1_t}\right)|r0_t\rangle + \left(u_{10}\alpha_{r0_t} + u_{11}\alpha_{r1_t}\right)|r1_t\rangle$$

We can define the function for applying a generic single-qubit gate U to the amplitude pair corresponding to the basis states $|r0_t\rangle$ and $|r1_t\rangle$ of the state vector Ψ as shown below:

1: **procedure** ApplyGateToAmplitudePair(Ψ, r, U)
2: $\quad\alpha_{r0_t} \leftarrow \Psi[r0_t]$
3: $\quad\alpha_{r1_t} \leftarrow \Psi[r1_t]$
4: $\quad\Psi[r0_t] \leftarrow u_{00}\alpha_{r0_t} + u_{01}\alpha_{r1_t}$
5: $\quad\Psi[r1_t] \leftarrow u_{10}\alpha_{r0_t} + u_{11}\alpha_{r1_t}$
6: **end procedure**

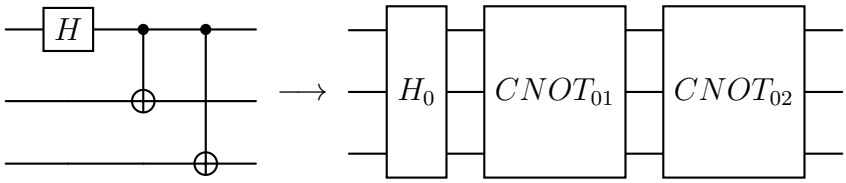
This algorithm only applies the gate to a single amplitude pair and in order to correctly update the entire state vector we need to loop through every amplitude pair. If we assume that the gate is controlled by m qubits $c_0, \ldots, c_m$ we can also add a check for control bits as described in section 2.3.2, in order to make the application algorithm controlled.

1: **procedure** ApplyGate(Ψ, U)
2: $\quad$**for** each $r \in \{0,1\}^{n-1}$ **do**
3: $\quad\quad$**if** bits of $r0_t$ corresponding to $c_0, \ldots, c_m$ are all 1 **then**
4: $\quad\quad\quad$ApplyGateToAmplitudePair(Ψ, r, U)
5: $\quad\quad$**end if**
6: $\quad$**end for**
7: **end procedure**

For the purposes of this simple state vector simulator, we only implement controlled and uncontrolled single-qubit gates. Since arbitrary quantum operations can be decomposed into sequences of single-qubit gates (and their controlled variants) [1], restricting the implementation to the single-qubit case keeps the implementation straightforward.

6.6 Full matrix multiplication simulator

A naive implementation of a simulator is to simply multiply the 2^n dimensional state vector with a $2^n \times 2^n$ unitary matrix at each step. As discussed in section 2.3.1, in a n qubit system, every gate application can be written as applying a gate on all n qubits. The matrix applied to the whole system can be found by padding with identity matrices on qubits that do not change using equation 2.21, resulting in a reinterpretation as shown in figure 6.3.



$$
\begin{aligned}
H_0 &= H \otimes I \otimes I \\
CNOT_{01} &= |0\rangle\langle 0| \otimes I \otimes I + |1\rangle\langle 1| \otimes X \otimes I \\
CNOT_{02} &= |0\rangle\langle 0| \otimes I \otimes I + |1\rangle\langle 1| \otimes I \otimes X
\end{aligned}
$$

Figure 6.3: An equivalent circuit where each gate is expanded to act on the whole system.

6.7 GPU accelerated simulator

Quasim includes a GPU accelerated state vector simulator. In theory this simulator works using the same principle as the state vector simulator on the CPU, but in order to create a simulator utilizing a GPU efficiently we introduce a number of optimization methods.

In order to write code for GPUs we need to create so called "kernels", essentially functions that can be run in parallel across many threads. GPUs follow a hierarchical execution model in which a large number of lightweight threads are grouped into execution units. Threads are organized into groups (e.g., workgroups or blocks) that execute on a single compute unit and can synchronize and share fast local

memory. Multiple such groups are launched as part of a single kernel invocation and are scheduled independently by the hardware. While threads within a group can synchronize, synchronization between groups (cubes) is generally not possible within a single kernel execution [19].

The main difficulty in GPU-based computation lies in the high cost of data transfer and communication between CPU (host) and GPU (device), as well as the need to adapt algorithms to fit the parallel execution model of GPUs. Naturally, one must minimize host-device data transfer and avoid unnecessary kernel launches which introduce synchronization overhead.

The world of GPU computing is fragmented, as different platforms such as CUDA, ROCm, Metal, and Vulkan each require their own dedicated implementations [20]. For this reason, a language or framework that can operate on multiple platforms is preferred to ensure portability. Since the project is implemented in Rust, we selected a Rust-native library called *CubeCL*. CubeCL is a framework for writing portable kernel code that can be executed on WebGPU, CUDA, ROCm, Metal, and Vulkan.

In CubeCL, a *cube* denotes a group of execution units (threads) and corresponds to a CUDA thread block or a GPU workgroup abstraction. Figure 6.4 shows how individual units relate to cubes and hyper-cubes.

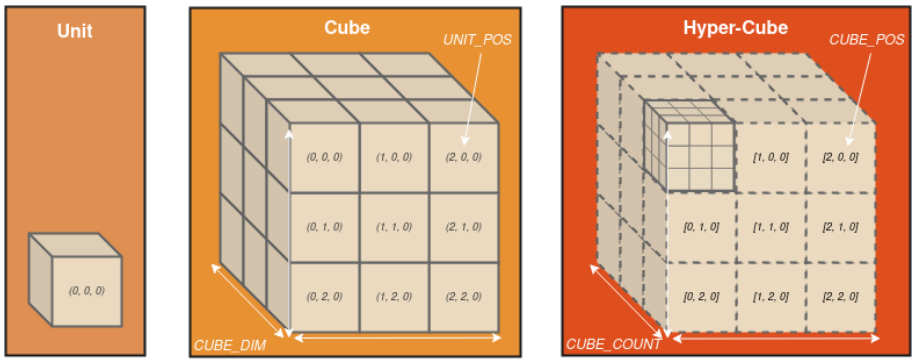


Figure 6.4: The CubeCL topology / execution model is designed around the concept of cubes which map to the hardware architecture of GPUs. Reproduced from [20], licensed under Apache License 2.0.

6.7.1 Batching algorithm

The circuit shown in figure 6.5 consists of 8 gates (SWAP gate decomposed into three gates). A naive execution strategy, in which each gate is applied via a separate GPU call, would therefore require 8 GPU kernel launches. In contrast, the batching algorithm groups gates into larger "batches", reducing the total number of GPU

calls to 2, while ensuring that the number of target qubits involved in any single batch never exceeds 2. The number 2 was chosen for this example but can be chosen arbitrarily with certain tradeoffs described later.

The batching algorithm creates a graph representation of the circuit. Each node in the graph corresponds to a single gate-application operation. Each batch is made up of multiple nodes, such that multiple gate applications can be submitted to the GPU in a single kernel invocation. Each node in the graph belongs to a single batch.

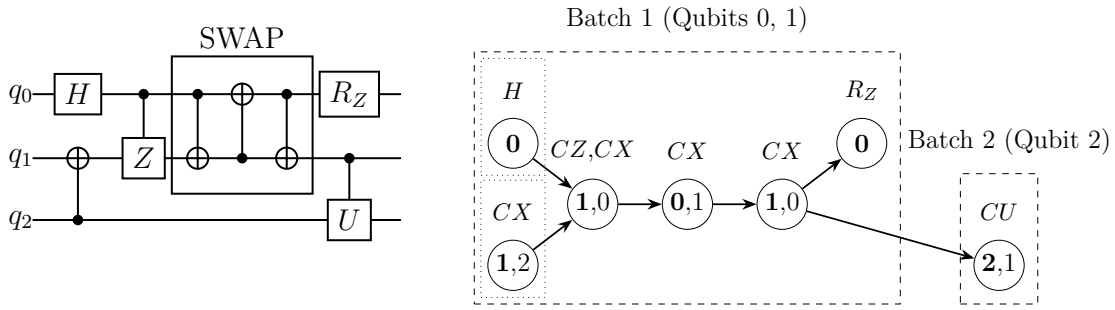


Figure 6.5: Processing a circuit with a max qubit limit of 2 per batch results in the following batch graph. Dotted rectangles shows retired batches that were merged. Observe how the controlled U gate acting on qubits 1 and 2 results in *Batch 2* being created, as it would have exceed the 2 qubit limit if it were added to *Batch 1*.

In the resulting graph, we can see that the controlled Z and controlled X gate were multiplied together into a single node that acts as a single gate. This is possible because their signatures matched exactly (targets qubit 1 and controlled by qubit 0).

We can observe in figure 6.5 that even though the first CX node touches both qubits 1 and 2, *Batch 1* still stays within the 2 qubit limit since its only the combined *targets* of the gate which count towards the limit. This is because control qubits do not increase the dimensionality of the affected subspace, whereas target qubits determine the size of the transformed state vector block.

After creating this graph representation of the circuit, each batch is flattened into a sequential array of 2×2 matrices. In the actual implementation this is done as the batch is built, with each batch containing a vector to which gate matrices (as well as target and control data) are added as they are processed. The resulting array should be ordered so that no node that depends on another node, shown by the directed arrow, comes before it.

In order to actually apply these batches to a state vector residing on the GPU,

the host is required to transfer the data to the device. This is done once as the simulator is created using a specified circuit. All batch data is uploaded in one go and to actually apply gates, the host simply needs to call the kernel shown in figure 6.6 with arguments specifying which slice of the uploaded gate data to apply.

The gate application kernel Once the batch data has been transferred to the GPU, each batch is applied using a single kernel invocation. The execution follows the same underlying principle as the CPU-based state vector simulator, but is adapted to the parallel execution model of the GPU. For a batch acting on k target qubits in an n-qubit system, the state vector can be partitioned into 2^{n-k} independent sub-blocks of size 2^k. Each GPU thread is assigned one such sub-block and sequentially updates all amplitudes within it. Within each thread, all gates in the batch are applied sequentially to the assigned sub-block. Since different sub-blocks are independent, they can be processed in parallel without requiring synchronization between cubes.

```
1: procedure GpuApplyBatch(stateVector, batchData)
2:     blockIndices ← indices of the 2^k amplitudes in this thread's sub-block
3:     for each gate in batchData do
4:         for each index in blockIndices do
5:             GpuApplyGate(stateVector, index, gate)
6:         end for
7:     end for
8: end procedure
```

Figure 6.6: High-level pseudocode of the batched gate application kernel. Each thread processes an independent sub-block of the state vector while iterating over all gates in the batch.

Trade-offs with setting the max number of target qubits per batch limit too high

- Increasing qubits per batch grows the state-vector sub-block size exponentially (subspace size scales as $O(2^k)$, increasing arithmetic operations per kernel.
- Larger batches raise the computational cost per kernel due to more operations on a larger affected subspace.
- More qubits per batch can reduce memory-access contiguity, especially for non-consecutive targets. Reduced contiguity leads to more irregular, strided, or interleaved access patterns over basis states. These irregular patterns can reduce memory coalescing and may lower memory throughput and cache efficiency on GPUs [19, Sec. 4.1.1].

- Larger state-vector blocks mean fewer blocks can be processed concurrently by GPU execution units. Fewer concurrently processed blocks reduce overall parallelism and can underutilize GPU computational resources.

Circuit measurement on GPU A naive implementation of measurement would transfer the entire state vector from the GPU to the CPU and sample from the resulting probability distribution on the host. Since an n-qubit state vector contains 2^n complex amplitudes, this requires $O(2^n)$ data transfer and quickly becomes a significant bottleneck for higher qubit counts.

To avoid this, the GPU simulator performs sampling using a hierarchical reduction (described in [21, Ch. 6]). First, the probabilities $p_i = |\alpha_i|^2$ are computed on the GPU and reduced in parallel into a hierarchy of partial sums. A random threshold is then used to traverse this hierarchy: at each level, the block whose cumulative probability contains the threshold is selected, and the search is refined until a single basis state is identified. Only the final sampled index is transferred back to the CPU.

This avoids transferring the full state vector during measurement. The trade-off is that the reduction and hierarchy traversal require multiple kernel launches, so the overhead can dominate for small state vectors. For larger systems, where the cost of transferring $O(2^n)$ amplitudes is much higher, the approach becomes beneficial and matches the intended use case of the GPU accelerated simulator.

6.8 Product state simulator

As seen in equation 2.5, the state of an n-qubit system can be represented as a tensor product of the states of its subsystems. We call this the *product state representation*. In general, this representation needs fewer amplitudes to fully describe the state of a system compared to the 2^n amplitudes of the general state vector.

For example, if the state of a system can be expressed as n single-qubit states, then the spatial complexity of its product state representation would be $O(n)$ instead of $O(2^n)$. However, as seen in section 2.2, most states are entangled, which can not be expressed as a product state.

In general, if m denotes the largest number of qubits in a subsystem, then the product state representation of a n-qubit has a spatial complexity of $O(n2^m)$. Note that if m is not a function of n, then the spatial complexity is $O(n)$. However, in the case $m = n$, when all qubits are entangled, then the product state representation is just the state vector.

According to equation 2.17, applying a gate on a single subsystem can be done without regard to the other subsystems. However, a gate acting on several subsystems might result in an entangled state that can not be expressed as a product of states. In the case of potential entanglement, gates are applied to the combined system of the involved subsystems, which might cause m to increase as shown in figure 6.7.

Due to convention, the initial state any circuit of n qubits is assumed to be $|0\rangle^{\otimes n}$, thereby $m = 1$ initially. There are ways to prevent m from growing as the state evolves. One way is to identify gates that never cause entanglement, namely single-qubit-gates, as already stated, and regular swap-gates, which can be performed by swapping qubit indices, as shown in figure 6.8.

It is even possible to decrement m after measurements, as shown in figure 6.9. As simulators are expected to randomly choose a pure post-measurement state $|\Psi\rangle$, then $|x\rangle$, where $x \in \{0, 1\}$ is the chosen measured value, is a known factor of $|\Psi\rangle$. If $|\Psi\rangle$ is the state of the single largest system, then factoring $|x\rangle$ from $|\Psi\rangle$ results in two systems: one being a single qubit and another composed of $m - 1$ qubits.

The main idea of the product state simulator was to use the product state representation to store the state. The factors are represented by a list of `SubSystem` structures, which makes up a `ProductState`, each containing the `StateVector` of the subsystem and a list of indices of qubits that are part of the subsystem. Two subsystems can be combined by concatenating their list of qubit indices and evaluating the tensor product of their state vectors. Note that the order of the qubits is not necessarily preserved when combining subsystems, so additional logic was required to ensure correct indexing of amplitudes.

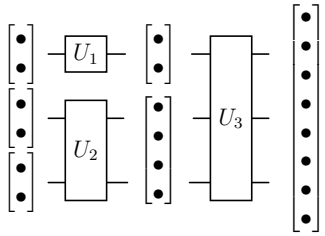


Figure 6.7: A circuit showing that gate application might combine systems.

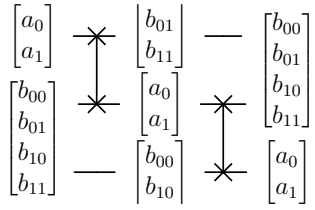


Figure 6.8: A circuit showing swaps performed without combining involved systems.

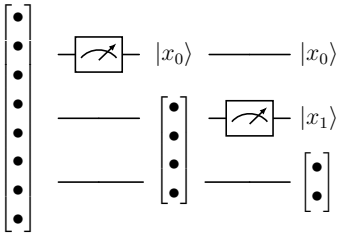


Figure 6.9: A circuit showing that measurement results in a state that can be factored.

6.9 Cube simulator

The possibility of storing the state vector in a linked data structure was also explored. A possible approach would have been a balanced binary tree using the basis state

as a key and the amplitude as a value at each node. Since the time complexity for accessing an element in a binary tree with 2^n elements is $O(n)$, it would have led to the algorithm for applying gates having a time complexity of $O(2^n n)$.

To construct a linked data structure that could apply gates with a time complexity of $O(2^n)$, the fact that gates acting on a single qubit only need to consider two amplitudes at a time, shown in equation 2.25, was used. In this context a basis state i would only share information with another basis state j if and only if the hamming distance between i and j is 1. The graph showing the relations between the basis states of one, two and three qubit systems are shown in figure 6.10. These graphs were referred to as `CubeMaps` due to the fact that for an n qubit system the graph could be thought of as an n dimensional cube.

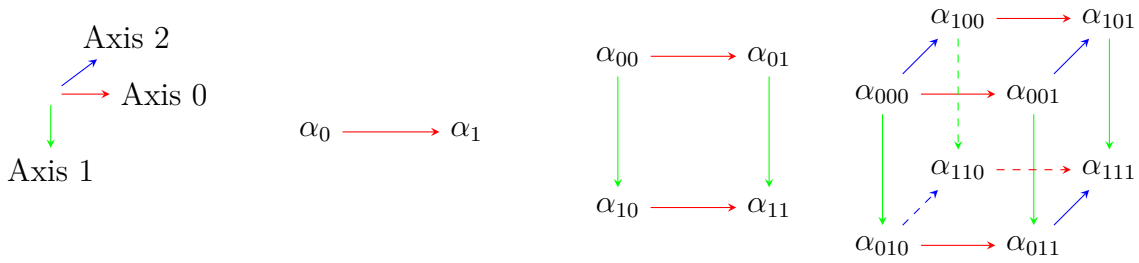


Figure 6.10: Graphs where each vertex represents the complex amplitude of a basis state and edges only exist between vertices if the hamming distance between the states is 1. These edges are all parallel to one of three axes. Graphs are shown for systems of one, two and three qubits.

The n dimensional `CubeMap` consisted of 2^n nodes. Each node could be represented by a unique basis state $k \in [0, 2^n)$; however, these basis states were never stored anywhere in the code and were only used to explain and understand why and how the `CubeMap` works. In figure 6.10 α_k refers to the value stored at the node with basis state k.

Figure 6.10 shows that all edges in the `CubeMap` are directed such that $k_1 < k_2$ for an edge going from a node represented by k_1 to a node represented by k_2 where the hamming distance between k_1 and k_2 is 1. It was then concluded that *to move along an edge, is the same as setting a basis state's bit which was previously* 0 *to* 1. In figure 6.10 it is also shown that the edges that set the same bit q to 1 could be thought of as being parallel to each other. This geometric interpretation of the `CubeMap` was further developed by labeling axes $[0, n)$ where an edge that set bit q to 1 has the same direction as axis q.

Figure 6.10 also shows that the root of the `CubeMap` has a basis state with n bits

all set to 0, n outgoing edges and therefore n relevant axes $[0, n)$. The figure also shows that any node i steps away from the root will have $n - i$ bits set to 0, $n - i$ outgoing edges and therefore $n - i$ relevant axes. It can be concluded that every node in the `CubeMap` is the root of an $(n - i)$ dimensional `CubeMap`. These nodes were referred to as having a dimension of $n - i$ which is always equivalent to the number of outgoing edges from the node.

6.9.1 Structure

An n dimensional `CubeMap` was a linked data structure where every node was defined as shown in listing 6.8. To move along axis `q` from a node the code would access index `q` of the field `edges`. The listing 6.8 also uses a type `SendPtr<CubeMap>` which is a struct that holds a pointer to `CubeMap` and implements the traits necessary to send this pointer between threads. One alternative to `SendPtr<CubeMap>` would have been `Box<Node>` which would not have allowed multi-threading. Another alternative would have been `Arc<RwLock<Node»` which comes with a significant drop in performance. Choosing `SendPtr<Node>` over the other alternatives introduced a performance boost at the cost of writing code that Rust considers unsafe.

```
struct CubeMap {
    amplitude: nalgebra::Complex<f64>
    edges: Vec<SendPtr<CubeMap>>
}
impl CubeMap {
    fn new(n: usize) -> Self {
        if n == 0 {
            Self {
                amplitude: cart!(0), /*0 + 0i*/
                edges: Vec::with_capacity(0),
            }
        } else {
            let mut ret = CubeMap::new(n - 1);
            unsafe {
                ret.link(SendPtr::construct(CubeMap::new(n - 1)));
            };
            ret
        }
    }
    unsafe fn link(&mut self, cube: SendPtr<Self>) {
        for i in 0..self.edges.len() {
            unsafe { self.edges[i].link(cube.edges[i]) };
        }
        self.edges.push(cube);
    }
```

```
}
```

Listing 6.8: Struct defining a node in a *CubeMap* as well as an implementation for it's constructor

```
pub struct CubeVector {
    zero: CubeMap,
}

impl CubeVector {
    pub fn new(n: usize) -> Self {
        let mut zero = CubeMap::new(n);
        zero.amplitude = cart!(1); /*1 + 0i*/
        Self { zero }
    }
}
```

Listing 6.9: Struct defining a `CubeVector` as well as the implementation of it's contructor

The construction of the struct `CubeMap` used the fact that linking two `CubeMaps` with the same dimension n creates a new `CubeMap` with dimension $n + 1$. Here linking would imply creating an edge between all nodes of the first `CubeMap` with their respective node in the second `CubeMap`. The implementation of this construction is shown in listing 6.8. In addition to the struct `CubeMap` there existed another struct `CubeVector` which acted as an interface for `CubeMap`. This struct is shown in listing 6.9. The listing also shows the constructor for the `CubeVector`. It can be seen that a `CubeVector` that can hold states for n qubits creates a `CubeMap` with dimension n and then sets the amplitude of the root node in `CubeMap` (the node representing basis state $|0 \dots 00\rangle$) to $1 + 0i$. A new `CubeVector` will then represent the initial state of a quantum state of n qubits.

6.9.2 Traversal

Figure 6.11 shows an algorithm that traverses an entire `CubeMap` while only visiting any node once. To make sense of this algorithm it is important to note how axes' labels change between nodes. For example figure 6.10 shows that two nodes α_{000} and α_{010} share two axes: 0 and 2. However, these nodes do not share the same labels for those axes. α_{000} labels them 0 and 2 but α_{010} will label them 0 and 1. This could be generalized into two facts that were used throughout the algorithms pertaining to moving from a node `C1` to a child node `C2` along an axis `q`. First, if there exists an axis `p < q` on `C1` then that axis exists on `C2` with the same identifier `p`. Second, if there exists an axis `p > q` on `C1` then that axis exists on `C2` with a new identifier `p-1`. The algorithm shown in figure 6.11 uses the first fact to ensure that all nodes are only visited once. The algorithm uses an argument `start_axis` that tells it

to only further traverse along an axis if it is greater than or equal to `start_axis`. Further when the the algorithm traverses along an axis `q` it sets `start_axis` to `q`. Figure 6.12 shows the traced out traversal when algorithm is applied to a 3 dimensional `CubeMap`. The figure also shows how the first fact is used to make sure axis 0 is only traversed once, axis 1 is only traversed twice and so on.

```
1: procedure TRAVERSAL(root, start_axis)
2:     OUTPUT(root)
3:     n ← dimension of root
4:     for axis in [start_axis, n) do
5:         node ← traverse along axis from root
6:         TRAVERSAL(node, axis)
7:     end for
8: end procedure
```

Figure 6.11: High-level pseudocode for the traversal of a `CubeMap`.

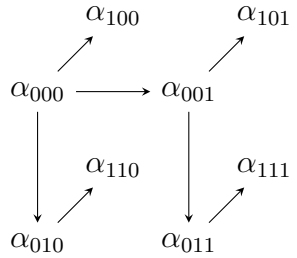


Figure 6.12: Visualized traversal of a 3 dimensional `CubeMap` following the algorithm shown in figure 6.11.

6.9.3 Applying gates

The algorithms for applying gates is similar to the algorithm used to traverse the `CubeMap`, shown in figure 6.11. However, one third fact would be needed. The third fact was that applying a gate along a set of target axes would imply that there would be no need to also traverse any of these axes. The algorithm for applying single qubit gates and the algorithm for applying the SWAP gate, shown in figure 6.13 and 6.14 is similar to the algorithm for traversing the `CubeMap` except the function `Output` is specified to be the application of theories discussed in section 2.3.2. Additionally the second fact is applied to the target axes if they are larger than the traversed axis and the third fact is applied by never traversing along the target axes.

The algorithms shown in figure 6.13 and 6.14 achieves a time complexity of $O(2^n)$ since they only visit nodes once. The algorithms were also further optimized via multi-threading. As can be seen in figure 6.12 the traversal can be multi-threaded

by spawning a new thread every time an edge is traversed. This method of spawning threads would also be free of data races since the nodes visited would be mutually exclusive between threads. However, if a thread were to be spawned every time an edge was traversed then the number of threads would increase exponentially. This would result in a multi-threaded `CubeMap` performing worse than a single-threaded `CubeMap`. To avoid spawning to many threads, the `CubeMap` was implemented to only spawn new threads if it's dimension was larger than 16.

1: **procedure** APPLY2X2(root, start_axis, target_axis, $U_{2\times2}$)
2: target ← traverse along target_axis from root
3: α_{root} ← Complex altitude stored at root
4: α_{target} ← Complex altitude stored at target
5: $\begin{bmatrix} \alpha_{root} \\ \alpha_{target} \end{bmatrix} = U_{2\times2} \begin{bmatrix} \alpha_{root} \\ \alpha_{target} \end{bmatrix}$
6: n ← dimension of root
7: **for** axis in [start_axis, n) **do**
8: node ← traverse along axis from root
9: **if** axis < target_axis **then**
10: APPLY2X2(node, axis, target_axis, $U_{2\times2}$)
11: **else if** axis == target_axis **then**
12: Skip axis
13: **else**
14: APPLY2X2(node, axis, target_axis-1, $U_{2\times2}$)
15: **end if**
16: **end for**
17: **end procedure**

Figure 6.13: High-level pseudocode for the application of single-qubit gates on a *CubeMap*

```
1: procedure ApplySwap(root, start_axis, target_axis1, target_axis2)
2:     target1 ← traverse along target_axis1 from root
3:     target2 ← traverse along target_axis2 from root
4:     α1 ← Complex altitude stored at target1
5:     α2 ← Complex altitude stored at target2
6:     Swap(α1, α2)

7:     n ← dimension of root
8:     for axis in [start_axis, n) do
9:         node ← traverse along axis from root
10:        if axis < target_axis1 then
11:            ApplySwap(node, axis, target_axis1, target_axis2)
12:        else if axis == target_axis1 then
13:            Skip axis
14:        else if axis < target_axis2 then
15:            ApplySwap(node, axis, target_axis1-1, apply_axis2)
16:        else if axis == target_axis2 then
17:            Skip axis
18:        else
19:            ApplySwap(node, axis, target_axis1-1, apply_axis2-1)
20:        end if
21:    end for
22: end procedure
```

Figure 6.14: High-level pseudocode for the application of the SWAP gate on a *CubeMap* where `target_axis1` $<$ `target_axis2`

6.9.4 Applying controlled gates

When applying controlled gates a fourth fact would need to be recognized. The fourth fact is that applying a gate with c control bits on an n dimensional `CubeMap` will only affect one of the $n - c$ dimensional child `CubeMaps`. For example, it can be seen in figure 6.10 that applying a gate with controlled bits identified by the bitmask 100 will only affect the nodes in the 2 dimensional `CubeMap` that has α_{100} as it's root. In figure 6.15 an algorithm that applies controlled qubits is shown. The algorithm uses the second and fourth fact to traverse to an $n - c$ dimensional `CubeMap` before applying a specified gate.

```
1: procedure ApplyControlledGate(node, control_axes, target_axes)
2:     if control_axes is empty then
3:         ApplyGate(node, target_axes)
4:     end if
5:     control ← Min(control_axes)
6:     Remove control from control_axes
7:     Decrement all axes in control_axes
8:     Decrement all axes larger than control in target_axes
9:     ApplyControlledGate(node, control_axes, target_axes)
10: end procedure
```

Figure 6.15: High-level pseudocode for the application of a controlled gate on a `CubeMap` where `control_axes` $\cap$ `target_axes` $= \emptyset$

6.10 Syntax simulator

When learning about quantum computation, it is common to express circuits as multiplication of matrices, across a sum of scaled basis states in the form of an algebraic expression.

A matrix H in this case, is a Hadamard gate. The matrices all have known values and multiplying a matrix with a basis state is a well defined operation. However, humans usually use shortcuts to avoid having to do matrix multiplication on paper.

By memorizing the effect of gates on the six most common points of the bloch sphere, $|0\rangle, |1\rangle, |+\rangle, |-\rangle, |i\rangle$, and $|-i\rangle$, gate application need not matrix multiplication to be performed at all. For example, applying a Hadamard gate to $|0\rangle$ is widely known to transform it into $|+\rangle$. A syntax simulator knows the effect of representable gates by precomputed translation. For example, the implementation of the translation that a Hadamard gate performs is shown in listing 6.10

```
fn h(self: &ExtendedQubitBasis) -> ExtendedQubitBasis {
    use ExtendedQubitBasis::*;
    match self {
        Zero => Plus,
        One => Minus,
        Plus => Zero,
        Minus => One,
        I => MinusI,
        MinusI => I,
    }
}
```

Listing 6.10: The syntax simulator implementation for the translation of qubit states via the Hadamard gate

The syntax simulator stores not a state vector, but a sum of scaled states. From linear algebra, it follows that matrix multiplication distributes over sums, and multiplying a scaled expression is isomorphic to scaling after multiplying the expression. For the syntax simulator, performing a gate is a matter of mapping the state in each term of the sum to the state where one of the qubits has been transformed by the gate.

There are two extra challenges to the aforementioned approach that need to be taken into account:

1. Controls need a value of $|0\rangle$ or $|1\rangle$ to be evaluated.

2. Applying gates can cause global phase change.

To evaluate whether a control condition of a gate is satisfied or not, the control qubits in all terms need to be expanded into true binary basis states. In the best case, the control qubit is already $|0\rangle$ or $|1\rangle$, and evaluating the control is trivial. In the worst case, the control qubit is in a superposition in all terms. Then, all terms will be expanded into two terms, one with the control qubit now $|0\rangle$, and another where it is $|1\rangle$. That effectively doubles the computation time of all subsequent gates, now having twice as many terms to transform. Also, one gate can more than double the number of terms since gates can have more than one control qubit.

It is not obvious why the syntax simulator needs to take global phase into account. A proof showing that disregarding global phase change from gate application will produce incorrect results follows. Consider the following onset of an evaluation:

$$X_{0,1}\left|+-\right\rangle = X_{0,1}\frac{1}{\sqrt{2}}(\left|0-\right\rangle + \left|1-\right\rangle) = \frac{1}{\sqrt{2}}(X_{0,1}\left|0-\right\rangle + X_{0,1}\left|1-\right\rangle) = \frac{1}{\sqrt{2}}(\left|0-\right\rangle + X_1\left|1-\right\rangle) \tag{6.2}$$

where $X_{0,1}$ is a $CNOT$-gate with the left qubit as control and the right qubit as target, and X_1 is a simple X-gate with the rightmost qubit as target. Now, know that the effect of applying an X-gate to the $\left|-\right\rangle$ state is isomorphic to applying a global phase change of π radians, that is multiplying by -1. Observe the following continuation where global phase is correctly taken into account:

$$\frac{1}{\sqrt{2}}(\left|0-\right\rangle + X_1\left|1-\right\rangle) = \frac{1}{\sqrt{2}}(\left|0-\right\rangle - \left|1-\right\rangle) = \left|--\right\rangle \tag{6.3}$$

Compare the previous continuation to simple symbolic translation without respecting global phase change:

$$\frac{1}{\sqrt{2}}(\left|0-\right\rangle + X_1\left|1-\right\rangle) = \frac{1}{\sqrt{2}}(\left|0-\right\rangle + \left|1-\right\rangle) = \left|+-\right\rangle \tag{6.4}$$

If one where to apply a Hadamard gate targeted at the left qubit and measure, different results would be attained depending on if global phase was taken into account. The syntax simulator therefore applies an appropriate global phase in the case the input is an eigenvector of the gate's matrix.

6.11 Debug terminal

The debug terminal is a terminal-based application whose purpose is to debug a quantum circuit. It is inspired by GDB [22], a popular terminal-based debugger. GDB was referenced during development of the debug terminal with many GDB commands being mirrored, for example, "step" and "break". The full list of implemented commands and a short description can be found in table 6.1.

Command	Description
continue (c|run)	Continue execution until a breakpoint is hit or end of circuit is reached. Optionally specify to skip a number of breakpoints or ignore breakpoints entirely.
step (s)	Step forward one instruction. Optionally specify a number of instructions to step forward.
next (n)	Step forward and over one instruction. Optionally specify a number of instructions to step forward.
previous (p|prev)	Step back one instruction. Optionally specify a number of instructions to step back.
break	Insert a breakpoint at the specified gate index. Optionally specify to only enable an already existing breakpoint.
delete	Delete the breakpoint at the specified gate index.
disable	Disable the breakpoint at the specified gate index.
enable	Enable the breakpoint at the specified gate index. Only works for already existing breakpoints.
state	Show the current state. Optionally specify to show only a specific part of the state.
collapse (cl)	Collapse the current state into a single value and show the count of each value. Optionally specify to collapse multiple times for a more even distribution of collapsed values.
circuit	Show information about the circuit diagram.
show	Show the contents of classical registers.
help (h)	Show this help message. Optionally specify a command to get more specific help.
quit (q)	Exit the debugger.

Table 6.1: List of available commands and description

6.12 Benchmarking

The benchmarking was done using the Rust library `divan` on a number of circuits testing various capabilities of the simulators. The parameters that were used when benchmarking was not exactly the same for every simulator. However, there were always common parameters between the simulators to allow comparison between them. The commands used to execute the benchmarking is shown in listing 6.11.

The tests were performed using the following hardware:

Processor: AMD Ryzen 5 5500 (12) @ 4.268 GHz

Total Physical Memory: 15872 MiB

GPU: NVIDIA GeForce RTX 2060 SUPER

```
export DIVAN_SAMPLE_COUNT=10
export DIVAN_SAMPLE_SIZE=10
cargo bench --bench mid-measure-all --features="cuda","wgpu"
cargo bench --bench mid-measurements  --features="cuda","wgpu"
cargo bench --bench num-gates  --features="cuda","wgpu"
cargo bench --bench qft  --features="cuda","wgpu"
cargo bench --bench system-size  --features="cuda","wgpu"
cargo bench --bench system-size-entanglement  --features="cuda","
    wgpu"
cargo bench --bench bernstein-vazirani-benchmark  --features="cuda"
    ,"wgpu"
cargo bench --bench deutsch-jozsa-benchmark  --features="cuda","
    wgpu"
cargo bench --bench grovers-benchmark  --features="cuda","wgpu"
cargo bench --bench shors-benchmark  --features="cuda","wgpu"
```

Listing 6.11: Command used to benchmark

7

Results

In this section, the supported features of the library and the different simulators performance will be shown.

7.1 Codebase

The codebase was written in Rust due to it being a systems level programming language that allows for strong typing and great control over how memory operations are performed. The general structure of the input of a circuit was based on OpenQASM. OpenQASM was chosen because of its aligned purpose of representing quantum circuits, independent of the application where the circuits may be used. Other established quantum computing tools often compile into/from OpenQASM as an intermediate representation [23]. Thus, using OpenQASM for this project may allow for easier interoperability with other tools in the future. The codebase was kept in a public git repository hosted on GitHub to allow anyone interested to view, review, or further develop the code.

7.1.1 General architecture

Quasim is a Rust library, meaning it is suited more towards use in other applications rather than as a standalone program. There are several simulators provided, discussed in Sections 6.5–6.10, which can be used to run a quantum algorithm. Quasim also comes with the capabilities for a user to construct their own simulators, as long as they implement the required traits.

7.1.2 Debug terminal

The debug terminal is a program in which the user can enter a command. The prompt contains a number at the start, which is the current step of the quantum algorithm. Stepping through the algorithm causes this number to increase. Usage of debug terminal is shown in figure A.1 and a full list of supported commands can be found in table 6.1.

7.1.3 Getting the result

There are multiple ways to read the result of a computation. Which method the user chooses depends on how much control the user wants over the sampling process.

To make sure that every simulator could be used in similar ways, they will all have to implement the trait `Simulator`. This trait will require all the simulators to implement the functions: `fn run(&mut self)` which runs the simulation and updates the internal state, `fn state(&self) -> &Self::State` which can be used to access the internal state such as basis amplitudes, and collapsing of the state for sampling. One case where this will be useful is for benchmarking using a Rust library like `divan` that will be able to test a provided list of simulators.

The trait `Sampleable` exposes a generic interface for sampling of circuits. This allows for interchangeable usage of different types of simulators for execution and sampling of quantum circuits. This is useful for both testing purposes of the library, as well as consistency across multiple simulator backends which increases the ease of use of the library.

7.1.4 Integrating with a classical computer

Since a simulator's `run` method can be executed and gives the collapsed and measured qubits inside regular Rust code, there would be no restrictions on using the simulated quantum results inside a classical algorithm. Implementation of classical-quantum hybrid simulation would therefore not necessarily need extra time, if ample care is taken to properly modularize the simulator. Modularization will be achieved by having a well-designed interface between all steps in the simulation process. Rust allows for this by letting definition of usage come before implementation, in the form of traits.

7.2 Simulator performance

All of the results from the benchmarks are listed in appendix D. The benchmark that took the most time for all simulators was the benchmark on Shor's algorithm. The results for every simulator when $N = 255$ and the total number of qubits was 19 is shown in table 7.1. Note that the full matrix multiplication simulator and the syntax simulator is not shown in this table. This is due to them not being able to perform well enough on Shor's algorithm when the total number of qubits get that high. The full matrix multiplication simulator got an average time of 54.75 seconds when benched on Shor's algorithm when $N = 7$ and the total number of qubits was 9. The syntax simulator got an average time of 18.03 milliseconds when benched on

Shor's algorithm when $N = 1$ and the total number of qubits was 5.

	Fastest	Slowest	Median	Mean	Samples	Iterations
Cube Simulator	3.358 m	3.775 m	3.765 m	3.633 m	10	100
GPU accelerated simulator (CUDA)	468.3 ms	473.3 ms	471.5 ms	470.9 ms	10	100
GPU accelerated simulator (WGPU)	683.8 ms	691.3 ms	688.2 ms	688 ms	10	100
Product State Simulator	35.11 s	35.45 s	35.29 s	35.29 s	10	100
State Vector Simulator	27.73 s	27.86 s	27.76 s	27.77 s	10	100

Table 7.1: Benchmark of the Simulators on Shor's algorithm when the total number of qubits was 19.

8

Discussion

This project succeeded in its main goal of building a classical computer program that, when provided a quantum circuit, returns the result a quantum computer would return when provided with that circuit. Some of notable achievements, shortcomings, and possible future extensions, among other interesting points, are discussed further in the rest of this section.

8.1 Use case

Even though Quasim is a rust library, it is perfectly able to be used on its own. It comes with different simulators tailored for different use cases behind feature flags. For example, a *DebuggableSimulator* is perfect for trying to debug and understand an algorithm whereas *GpuStateVectorSimulator* is useful if one is trying to execute a large algorithm for lots of qubits and only the result is interesting.

Quasim is also suitable as a library. The provided functions are easy to use, which enables creation of additional simulators should the provided ones not suffice. It can also be used to perform various operations on the matrices should one want to use the library but not build a simulator.

8.2 Limitations

The prospect of building a quantum computer simulator may seem like a purely algorithmic challenge. However, much work proved to be needed around the building of a quantum circuit description framework and mechanisms for enabling users to run simulations through this framework. Of course, user friendliness, among others, still had value, but priorities had to be made due to time constraints. A graphical user interface is friendly to use, but time did not allow for creation of a complex application.

8.3 Quantum circuit description

OpenQASM parsing was, at the beginning of the project, expected to be the main circuit description inputting mechanism. With time, it became apparent that, as long as writing any rust code was required to run the simulator, it was also easier to simply use the builder methods of the `Circuit` struct to describe what to run.

Most features of OpenQASM, for example classical bit operations after measurements, are available to users of the product, although not necessarily through `fn Circuit::from_qasm_file(file_name)`. All implemented simulator capabilities are available through the rust library interface.

Exposing the full feature set of the final product through all available interfaces would certainly be better than one of the interfaces lacking some features. As the project progressed, efforts focused more on extending simulator capabilities and efficiency, and less time was spent on extending the interfaces.

Hindsight would suggest that establishing priorities of where resources should have been spent could have been more prevalent in the project plan. Possibly, limiting the number of interface in total to only one, could have been a reasonable resource saving. As stated earlier, however, OpenQASM was believed to become the main circuit inputting mechanism, which is not what actually resulted in reality.

If this project resulted in a binary executable which only took a circuit description as input, and provided the result that quantum hardware would have provided as output, users would not be expected to write any code other than the actual circuit description. In such a case it would be unreasonable to expect users to use the rust library interface, and describing a quantum circuit is exactly the purpose of the OpenQASM specification [24]. OpenQASM would have been the better option in this case if only one interface was to be implemented. This also suggests that the project plan could have been more explicit in how the final product should be used.

8.4 Full matrix multiplication simulator

This simulator shows the tradeoff when opting for simplicity over performance. In particular, its spatial complexity quickly becomes an issue as it tries to allocate a $2^n \times 2^n$ matrix to perform gate application. For example, at 20 qubits, such a matrix would require 8 TB of memory (each element is 8 bytes), and for each additional qubit the memory usage would quadruple.

8.5 GPU accelerated simulator

As shown in the results in table 7.1, the GPU accelerated simulator offers a large speedup over the single-threaded state vector simulator. On smaller circuits however, the overhead of synchronization between host and device becomes apparent. This overhead can be clearly observed in the benchmark on the QFT algorithm in appendix section D.3. This benchmark shows that it only becomes more efficient to run the algorithm on the GPU accelerated state vector simulator over the single-threaded state vector simulator at around a circuit size of 12 qubits.

The speedup offered by offloading work to the GPU is also heavily dependent on the hardware and runtime on which the program is executed. The performance difference between the CUDA runtime and WGPU is shown to be very large for circuits with few qubits (section D.3), showing the significant difference in launch overhead between the two runtimes. In the QFT benchmark we can see the performance difference between the two runtimes becoming smaller as circuit size increases.

One drawback of the GPU accelerated simulator is the requirement of allocating the entire state vector in VRAM. Therefore the simulation is mainly limited by how much VRAM is available, which is usually significantly more expensive compared to ordinary RAM. Another drawback with the current implementation is that the required size of the state vector also needs to be allocated in RAM on the host. The reason for this is allowing the user to sync the state vector for access on the CPU, which is a requirement in our specification for the simulator traits.

A hard 32-qubit limit is imposed in the current implementation due to a technical constraint: the kernel code is limited to 32-bit integers. As a result, bitmasks and other related operations would overflow for inputs exceeding 32 qubits.

8.6 Product state simulator

As seen in tables D19, D47 and D54, in large circuits when the state is completely entangled the product state simulator does not outperform the state vector simulator, which was expected in the discussion in section 6.8. In fact, the product state simulator is generally slower in these cases. This is most likely due to the additional overhead logic needed to maintain the product state representation.

Another expectation discussed in section 6.8 was the potential reduction in spatial complexity with smaller subsystems. As expected, there is a linear relation between the number of qubits and execution time in table D12. Furthermore, table D33 shows that the size of the largest subsystem, m, plays a greater role in determining

the performance than the number of qubits as suggested by the spatial complexity of the product state representation.

There are also some other notable observations. For instance, decrementing m by factoring the state after measurements do seem to improve performance in circuits of a greater number of qubits but worsen it in those of fewer, as shown in table 8.1. This shows that the overhead caused by factoring the state after measurement is too significant in smaller circuits to be beneficial. Another underestimated phenomenon was the performance gain for smaller circuits that still entangles all qubits, as in the cases shown in tables D40 and D26. This is could be explained by the fact that full entanglement only happens at the very end of the circuit, meaning that most gates are applied to smaller subsystems. Again, this seem to be only beneficial for circuits of a larger number of qubits.

Circuit configuration	Fastest	Slowest	Median	Mean
8 qubits and 2 measurements	0.795	0.615	0.589	0.639
8 qubits and 4 measurements	0.829	0.688	0.809	0.783
8 qubits and 6 measurements	0.837	0.614	0.815	0.786
8 qubits and 8 measurements	0.837	0.786	0.825	0.818
16 qubits and 2 measurements	1.109	1.021	1.129	1.115
16 qubits and 4 measurements	1.106	1.198	1.133	1.135
16 qubits and 6 measurements	1.138	1.157	1.153	1.152
16 qubits and 8 measurements	1.167	1.201	1.192	1.190
16 qubits and 10 measurements	1.230	1.217	1.211	1.219
16 qubits and 12 measurements	1.230	1.230	1.231	1.232
16 qubits and 14 measurements	1.260	1.259	1.259	1.258
16 qubits and 16 measurements	1.295	1.394	1.280	1.305

Table 8.1: Speedup of the Product State Simulator relative to the State Vector Simulator from benchmarks in section D.1.

8.7 Cube simulator

As seen in benchmarks compiled in appendix D the Cube Simulator performed worse than the state vector simulator. This is probably inevitable since they have the same time complexity when it comes to applying gates except the algorithms used for the cube simulator are more complex leading to worse performance.

Another reason for the cube simulator performing worse as the number of qubits grow is probably due to too many threads existing at once. This problem was already considered in the methodology (see section 6.9) where it was decided to only implement multi-threading for `CubeMaps` with a dimension larger than 16. However,

the number of threads will still grow at an exponential rate albeit slower than if that restriction did not exist. To further optimize the use of multi-threading the algorithms would need to make sure that they do not spawn too many threads.

8.8 Syntax simulator

Scott Aronson and Daniel Gottesman proved efficient simulation of *stabilizer circuits*, with an implementation they called *CHP* [25]. A full explanation and discussion of the stabilizer formalism is outside the scope of this report, but one conclusion that could be drawn from the previously mentioned work is that if the set of possible resulting state vectors after simulation of a circuit is finite and known before simulation, then subexponential complexity simulation of that circuit is possible.

The syntax simulator stores possible resulting states from gate application in its precomputed translation tables to simulate some circuits surprisingly efficiently. In that regard, it is similar to *CHP*, even if the implementations differ substantially. Also, *CHP* is always efficient, but limited to simulating circuits consisting solely of *CNOT*, Hadamard, and Phase-gates. The syntax simulator can simulate any circuit that other simulators of this thesis can but, as benchmarks would suggest, generally slower than most other simulators.

8.9 Future work

As the target users were students trying to learn about quantum algorithms, it is desirable to make the debugger as accessible as possible [26]. A graphical user interface to both run and debug algorithms would make it easier for more students to learn about quantum algorithms, rather than having to spend time learning about Quasim's terminal app.

OpenQASM is an established programming language for quantum programs and improved support for OpenQASM in Quasim would mean better compatibility with existing resources. If OpenQASM support were to be improved, users with existing OpenQASM programs would not need to spend time porting it over to a Quasim circuit.

As simulated measurements are assumed to be observed, so the only way to get a probability distribution of results for hybrid circuits is by running the simulation several times. A way for a user to get the actual probability distribution might be of use for a deeper understanding of hybrid circuits.

In order to more accurately model the inner workings and reality of quantum computing, quantum noise and decoherence would be a logical next step to include in the simulation.

From the results it is clear that parallelization of state vector simulation on the GPU can provide a significant performance boost. Parallelization on the CPU however, was not thoroughly investigated but could possibly prove to also be a fruitful method. The reason for this being the fact that system RAM can be shared across multiple CPU:s and threads. This would enable simulation of larger state vectors since RAM is usually significantly cheaper than VRAM of equivalent capacity.

As of right now the simulation is limited by the available memory. For a more scalable approach beyond what is feasible to run on a single piece of hardware, research on implementations of distributed computing would be required. This could be an exciting area of research in conjunction with hardware accelerated computing.

8.10 Ethical and societal issues

Quantum computers can cause ethical problems in part due to their capability of breaking encryption algorithms [27]. However, since the project is a simulator of a quantum computer on classical hardware, it is only as capable as the classical hardware its running on, lacking the computational advantage provided by real quantum computers. The simulator can therefore not be thought of to share the societal and ethical aspects of a quantum computer.

However, as the simulator has the aim to be used as a tool for learning and testing of new circuits in research, there is a responsibility of providing truthful results to users. If the simulator does not, confusion may arise among students and the field of research may be impacted. The limitations of the simulator aimed to be made clear to the user.

Bibliography

[1] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information*. Cambridge University Press, 2010.

[2] G. Jaeger, *Quantum Information: An Overview*, 1st ed. New York, NY: Springer New York, 2007, ISBN: 978-0-387-36944-0. DOI: 10.1007/978-0-387-36944-0.

[3] W.-H. Steeb and Y. Hardy, *Problems and Solutions in Quantum Computing and Quantum Information*, 4th. WORLD SCIENTIFIC, 2018. DOI: 10.1142/10943. eprint: https://www.worldscientific.com/doi/pdf/10.1142/10943. [Online]. Available: https://www.worldscientific.com/doi/abs/10.1142/10943.

[4] L. Jakóbczyk and M. Siennicki, "Geometry of Bloch vectors in two-qubit system," *Physics Letters A*, vol. 286, no. 6, p. 388, 2001, ISSN: 0375-9601. DOI: 10.1016/S0375-9601(01)00455-8.

[5] R. Cleve, A. Ekert, C. Macchiavello, and M. Mosca, "Quantum Algorithms Revisited," *Proceedings: Mathematical, Physical and Engineering Sciences*, vol. 454, no. 1969, 1998, ISSN: 13645021, 14712946. Accessed: May 7, 2026. [Online]. Available: http://www.jstor.org/stable/53169.

[6] C. H. Bennett and S. J. Wiesner, "Communication via one- and two-particle operators on Einstein-Podolsky-Rosen states," *Physical Review Letters*, vol. 69, no. 20, pp. 2881–2884, 1992. DOI: 10.1103/PhysRevLett.69.2881.

[7] D. Deutsch, "Quantum theory, the Church-Turing principle and the universal quantum computer," *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences*, vol. 400, no. 1818, pp. 97–117, 1985.

[8] M. Nakahara and T. Ohmi, *Quantum Computing: From Linear Algebra to Physical Realizations*. CRC Press, 2008, ISBN: 978-0-7503-0983-7. DOI: 10.1201/9781420012293.

[9] L. K. Grover, "Quantum Mechanics Helps in Searching for a Needle in a Haystack," *Physical Review Letters*, vol. 79, no. 2, pp. 325–328, 1997. DOI: 10.1103/PhysRevLett.79.325.

[10] G. G. Moreno, "Grover's Algorithm." Accessed: Mar. 5, 2026. [Online]. Available: https://www.ucm.es/data/cont/docs/1461-2018-02-27-Grover.pdf.

[11] F. de Lima Marquezino, R. Portugal, and C. Lavor, *A Primer on Quantum Computing*. Springer, 2019. DOI: 10.1007/978-3-030-19066-8.

[12] S. Beauregard, "Circuit for Shor's algorithm using 2n+3 qubits," *Quantum Information and Computation*, vol. 3, no. 2, pp. 175–185, 2003. DOI: 10.26421/QIC3.2-8.

[13] T. G. Draper, *Addition on a Quantum Computer*, 2000. arXiv: quant-ph/0008033 [quant-ph]. [Online]. Available: https://arxiv.org/abs/quant-ph/0008033.

[14] A. Andersson, *quasim-cli*. Accessed: Apr. 26, 2026. [Online]. Available: https://github.com/Virtuo1/quasim-cli.

[15] Github, *Changing the layout of a view*. Accessed: Apr. 7, 2026. [Online]. Available: https://docs.github.com/en/issues/planning-and-tracking-with-projects/customizing-views-in-your-project/changing-the-layout-of-a-view#about-the-board-layout.

[16] L. Hales and S. Hallgren, "An improved quantum Fourier transform algorithm and applications," in *Proceedings 41st Annual Symposium on Foundations of Computer Science*, 2000, pp. 515–525. DOI: 10.1109/SFCS.2000.892139.

[17] The Rust Project Developers, *The Rust Programming Language*. 2026, https://doc.rust-lang.org/book.

[18] I. L. Markov and Y. Shi, "Simulating Quantum Computation by Contracting Tensor Networks," *SIAM Journal on Computing*, vol. 38, no. 3, pp. 963–981, 2008. DOI: 10.1137/050644756.

[19] T. M. Aamodt, W. W. L. Fung, T. G. Rogers, and M. Martonosi, *General-purpose graphics processor architectures*. Springer, 2018. DOI: 10.1007/978-3-031-01759-9.

[20] Tracel AI, *The CubeCL Book*. Accessed: Apr. 23, 2026. [Online]. Available: https://burn.dev/books/cubecl.

[21] S. Cook, *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes, 2012. DOI: 10.1016/C2011-0-00029-7.

[22] GNU Project, *GDB: The GNU Project Debugger*. Accessed: Apr. 17, 2026. [Online]. Available: https://www.sourceware.org/gdb/.

[23] IBM, *Introduction to OpenQASM*. Accessed: Feb. 12, 2026. [Online]. Available: https://quantum.cloud.ibm.com/docs/en/guides/introduction-to-qasm.

[24] A. W. Cross, L. S. Bishop, J. A. Smolin, and J. M. Gambetta, "Open quantum assembly language," *arXiv preprint arXiv:1707.03429*, 2017.

[25] S. Aaronson and D. Gottesman, "Improved simulation of stabilizer circuits," *Phys. Rev. A*, vol. 70, p. 052 328, 5 Nov. 2004. DOI: 10.1103/PhysRevA.70.052328. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevA.70.052328.

[26] A. Miniukovich, S. Sulpizio, and A. De Angeli, "Visual complexity of graphical user interfaces," in *Proceedings of the 2018 International Conference on Advanced Visual Interfaces*, ser. AVI '18, Castiglione della Pescaia, Grosseto, Italy: Association for Computing Machinery, 2018, ISBN: 9781450356169. DOI: 10.1145/3206505.3206549.

[27] L. Possati, "Ethics of Quantum Computing: an Outline," *Philosophy & Technology*, vol. 36, Jul. 2023. DOI: 10.1007/s13347-023-00651-6.

A

Debug terminal

The debug terminal has a set of commands shown in table 6.1. Figure A.1 shows an example of how the debug terminal can be used.

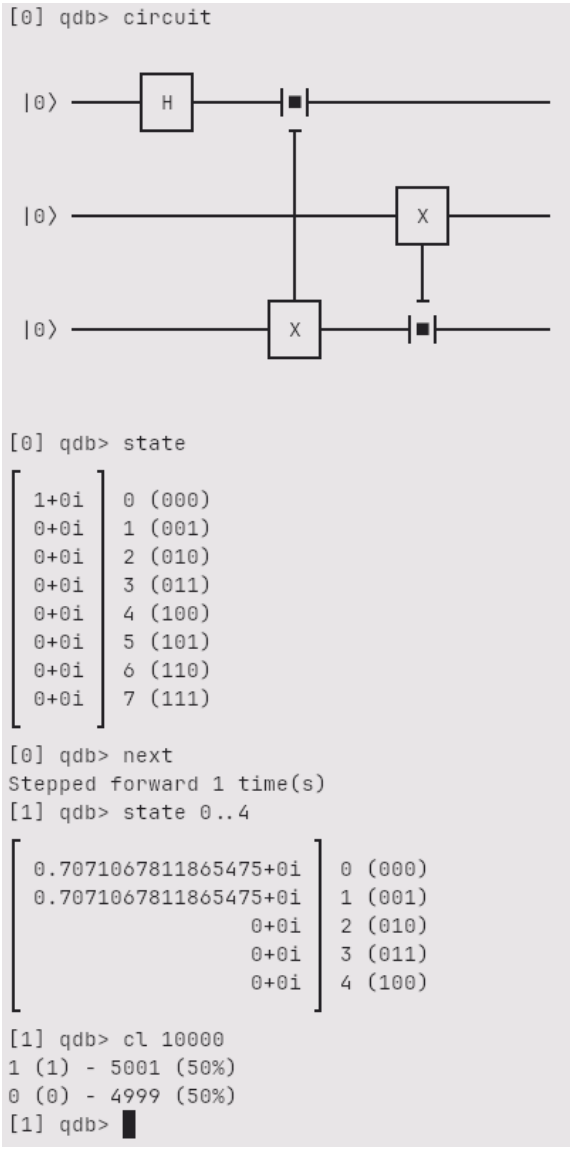


Figure A.1: Usage of debug terminal

B

Expression DSL

There are two types of domain specific languages for expressions, one which always resolves to a numeric value and one that always resolves to a boolean value. These are called *bit* and *boolean* expressions and the set of supported operations are shown in table B1 and table B2 respectively.

Expression	Rust syntax	Notes
Constant	`4`	Literal unsigned integer (4 is an example)
Register	`r("m")`	Read a full register (m is an example)
Register bit	`rb("m", 4)`	Read a bit in a register (m & 4 are examples)
Bitwise NOT	`!a`	
Bitwise AND	`a & b`	
Bitwise OR	`a | b`	
Bitwise XOR	`a ^ b`	
Integer addition	`a + b`	
Integer subtraction	`a - b`	
Integer multiplication	`a * b`	
Integer division	`a / b`	
Integer remainder	`a % b`	

Table B1: Supported bit expressions. $a, b \in$ BitExpr

Expression	Rust syntax
Equality	`a.eq(b)`
Less than	`a.lt(b)`
Less or equal	`a.lte(b)`
Greater than	`a.gt(b)`
Greater or equal	`a.gte(b)`
Boolean NOT	`!p`
Boolean AND	`p & q`
Boolean OR	`p | q`
Boolean XOR	`p ^ q`

Table B2: Supported boolean expressions. $a, b \in$ BitExpr $\quad p, q \in$ BoolExpr

C

Supported Circuit Operations

There were two kinds of circuits: Pure and Hybrid. There were therefore pure and hybrid circuit operations, which are shown in table C1 and C2.

Category	Supported operations
Single-qubit gates	X, Y, Z, H, S
Parameterized single-qubit gates	$U(\theta, \phi, \lambda)$, $R_x(\theta)$, $R_y(\theta)$, $R_z(\theta)$
Controlled gates	Controlled variants of all single-qubit gates
Swap gates	SWAP and controlled SWAP
Composite circuit operations	QFT, generated oracles, subcircuit calls
Debugging markers	Breakpoints

Table C1: Supported pure circuit operations. Pure circuits contain only quantum operations and debugging metadata.

Category	Rust syntax	Notes
Register	`new_reg(name, size)`	Create classical register
Measure bit	`measure_bit(t, (reg, bit))`	Store one measured qubit into one bit in `reg`
Measure bits	`measure_bits(targets, reg)`	Store multiple measured qubits consecutively in `reg`
Measure all	`measure(reg)`	Store full measurement into a register
Assignment	`assign(reg, expr)`	Store result of evaluated expression into register
Label	`label(name)`	Define jump target
Jump	`jump(label)`	Unconditional branch
Conditional jump	`jump_if(expr, label)`	Branch on boolean expression
Conditional apply	`apply_if(expr)`	Guard following instruction
Reset	`reset(t)`	Reset qubit to $\|0\rangle$

Table C2: Supported hybrid circuit operations. Classical conditions are represented using `BoolExpr`, and register-valued expressions are represented using `BitExpr`. Hybrid circuits are an extension to pure circuits, and therefore support all operations listed in table C1.

D

Benchmarks

Benchmarks were done using the rust library `divan`. The results from these tests are compiled in sections D.1, D.2, D.3, D.4, D.5, D.6, D.7 and D.8.

D.1 Measure bits individually mid circuit

Benchmarks done on a circuit scaled by number of qubits and number of measurements on individual bits mid circuit are shown in tables D1, D2, D3, D4, D5, D6 and D7.

Cube Simulator	Fastest	Slowest	Median	Mean	Samples	Iterations
8 qubits and 2 measurements	40.29 µs	47.54 µs	41.41 µs	42.07 µs	10	100
8 qubits and 4 measurements	42.99 µs	53.46 µs	43.44 µs	44.47 µs	10	100
8 qubits and 6 measurements	45.3 µs	49.93 µs	46.37 µs	46.84 µs	10	100
8 qubits and 8 measurements	47.26 µs	53.66 µs	48.3 µs	48.79 µs	10	100
16 qubits and 2 measurements	154.2 ms	174.8 ms	170.6 ms	169.7 ms	10	100
16 qubits and 4 measurements	175 ms	187.4 ms	180.3 ms	180.9 ms	10	100
16 qubits and 6 measurements	173.1 ms	187.4 ms	176.9 ms	176.9 ms	10	100
16 qubits and 8 measurements	152.4 ms	170.5 ms	154.1 ms	157 ms	10	100
16 qubits and 10 measurements	171.2 ms	196 ms	184.3 ms	183.9 ms	10	100
16 qubits and 12 measurements	175.3 ms	194.7 ms	184.9 ms	184.2 ms	10	100
16 qubits and 14 measurements	159.3 ms	178 ms	161.9 ms	164.7 ms	10	100
16 qubits and 16 measurements	188.2 ms	216.5 ms	192.9 ms	195.5 ms	10	100

Table D1: Benchmark of Cube Simulator on a circuit scaled by number of qubits and number of measurements on individual bits mid circuit.

Full Matrix Multiplication Simulator	Fastest	Slowest	Median	Mean	Samples	Iterations
4 qubits and 2 measurements	53.18 µs	56.51 µs	54.08 µs	54.15 µs	10	100
4 qubits and 4 measurements	53.63 µs	54.72 µs	54.16 µs	54.16 µs	10	100
8 qubits and 2 measurements	149.7 ms	151.5 ms	150.1 ms	150.2 ms	10	100
8 qubits and 4 measurements	149.5 ms	151 ms	150 ms	150.2 ms	10	100
8 qubits and 6 measurements	149.5 ms	150.7 ms	150.1 ms	150.1 ms	10	100
8 qubits and 8 measurements	149.6 ms	151.4 ms	150 ms	150.2 ms	10	100

Table D2: Benchmark of Full Matrix Multiplication Simulator on a circuit scaled by number of qubits and number of measurements on individual bits mid circuit.

GPU accelerated simulator (CUDA)	Fastest	Slowest	Median	Mean	Samples	Iterations
8 qubits and 2 measurements	340.1 µs	13.53 ms	383.6 µs	1.692 ms	10	100
8 qubits and 4 measurements	485.4 µs	559.4 µs	549 µs	526.6 µs	10	100
8 qubits and 6 measurements	584.3 µs	718.6 µs	692 µs	668.8 µs	10	100
8 qubits and 8 measurements	729 µs	765.4 µs	757.5 µs	751.8 µs	10	100
16 qubits and 2 measurements	849 µs	896.3 µs	861.6 µs	866.6 µs	10	100
16 qubits and 4 measurements	940.8 µs	1.044 ms	956 µs	972.4 µs	10	100
16 qubits and 6 measurements	1.028 ms	1.186 ms	1.087 ms	1.094 ms	10	100
16 qubits and 8 measurements	1.172 ms	1.259 ms	1.208 ms	1.21 ms	10	100
16 qubits and 10 measurements	1.322 ms	1.408 ms	1.332 ms	1.345 ms	10	100
16 qubits and 12 measurements	1.473 ms	1.579 ms	1.488 ms	1.511 ms	10	100
16 qubits and 14 measurements	1.611 ms	1.714 ms	1.622 ms	1.638 ms	10	100
16 qubits and 16 measurements	1.76 ms	1.917 ms	1.774 ms	1.803 ms	10	100

Table D3: Benchmark of GPU accelerated simulator (CUDA) on a circuit scaled by number of qubits and number of measurements on individual bits mid circuit.

GPU accelerated simulator (WGPU)	Fastest	Slowest	Median	Mean	Samples	Iterations
8 qubits and 2 measurements	1.019 ms	7.008 ms	1.052 ms	1.677 ms	10	100
8 qubits and 4 measurements	1.443 ms	1.821 ms	1.466 ms	1.544 ms	10	100
8 qubits and 6 measurements	1.868 ms	2.186 ms	1.94 ms	1.996 ms	10	100
8 qubits and 8 measurements	2.321 ms	2.674 ms	2.524 ms	2.49 ms	10	100
16 qubits and 2 measurements	1.785 ms	2.028 ms	1.822 ms	1.87 ms	10	100
16 qubits and 4 measurements	2.282 ms	2.623 ms	2.38 ms	2.4 ms	10	100
16 qubits and 6 measurements	2.767 ms	3.071 ms	2.94 ms	2.922 ms	10	100
16 qubits and 8 measurements	3.256 ms	3.573 ms	3.475 ms	3.428 ms	10	100
16 qubits and 10 measurements	3.685 ms	4.107 ms	3.969 ms	3.964 ms	10	100
16 qubits and 12 measurements	4.294 ms	4.948 ms	4.459 ms	4.5 ms	10	100
16 qubits and 14 measurements	4.827 ms	5.161 ms	4.968 ms	4.982 ms	10	100
16 qubits and 16 measurements	5.35 ms	5.626 ms	5.481 ms	5.475 ms	10	100

Table D4: Benchmark of GPU accelerated simulator (WGPU) on a circuit scaled by number of qubits and number of measurements on individual bits mid circuit.

Product State Simulator	Fastest	Slowest	Median	Mean	Samples	Iterations
8 qubits and 2 measurements	53.54 µs	76.79 µs	72.98 µs	67.85 µs	10	100
8 qubits and 4 measurements	54.34 µs	67 µs	56.69 µs	58.37 µs	10	100
8 qubits and 6 measurements	56.43 µs	79.26 µs	58.69 µs	60.93 µs	10	100
8 qubits and 8 measurements	59.72 µs	65.54 µs	61.12 µs	61.75 µs	10	100
16 qubits and 2 measurements	22.63 ms	26.1 ms	22.94 ms	23.22 ms	10	100
16 qubits and 4 measurements	22.94 ms	23.65 ms	23.27 ms	23.29 ms	10	100
16 qubits and 6 measurements	23.11 ms	23.56 ms	23.31 ms	23.31 ms	10	100
16 qubits and 8 measurements	22.94 ms	23.52 ms	23.17 ms	23.17 ms	10	100
16 qubits and 10 measurements	22.84 ms	23.54 ms	23.35 ms	23.24 ms	10	100
16 qubits and 12 measurements	23.04 ms	23.67 ms	23.48 ms	23.41 ms	10	100
16 qubits and 14 measurements	22.98 ms	23.66 ms	23.45 ms	23.42 ms	10	100
16 qubits and 16 measurements	22.85 ms	24.03 ms	23.55 ms	23.44 ms	10	100

Table D5: Benchmark of Product State Simulator on a circuit scaled by number of qubits and number of measurements on individual bits mid circuit.

State Vector Simulator	Fastest	Slowest	Median	Mean	Samples	Iterations
8 qubits and 2 measurements	42.54 µs	47.19 µs	42.96 µs	43.39 µs	10	100
8 qubits and 4 measurements	45.03 µs	46.08 µs	45.89 µs	45.68 µs	10	100
8 qubits and 6 measurements	47.24 µs	48.65 µs	47.86 µs	47.89 µs	10	100
8 qubits and 8 measurements	49.98 µs	51.49 µs	50.44 µs	50.5 µs	10	100
16 qubits and 2 measurements	25.1 ms	26.66 ms	25.9 ms	25.9 ms	10	100
16 qubits and 4 measurements	25.37 ms	28.33 ms	26.37 ms	26.43 ms	10	100
16 qubits and 6 measurements	26.3 ms	27.27 ms	26.88 ms	26.86 ms	10	100
16 qubits and 8 measurements	26.76 ms	28.24 ms	27.62 ms	27.57 ms	10	100
16 qubits and 10 measurements	28.09 ms	28.65 ms	28.28 ms	28.34 ms	10	100
16 qubits and 12 measurements	28.35 ms	29.12 ms	28.91 ms	28.84 ms	10	100
16 qubits and 14 measurements	28.96 ms	29.78 ms	29.53 ms	29.47 ms	10	100
16 qubits and 16 measurements	29.58 ms	33.49 ms	30.15 ms	30.59 ms	10	100

Table D6: Benchmark of State Vector Simulator on a circuit scaled by number of qubits and number of measurements on individual bits mid circuit.

Syntax Simulator	Fastest	Slowest	Median	Mean	Samples	Iterations
8 qubits and 2 measurements	722.1 µs	1.139 ms	776.9 µs	860 µs	10	100
8 qubits and 4 measurements	682.5 µs	1.007 ms	841.1 µs	834.2 µs	10	100
8 qubits and 6 measurements	714.6 µs	1.012 ms	840.4 µs	833.6 µs	10	100
8 qubits and 8 measurements	636.4 µs	769.1 µs	695.1 µs	704 µs	10	100
16 qubits and 2 measurements	288.9 ms	309.7 ms	297.2 ms	298.2 ms	10	100
16 qubits and 4 measurements	273.5 ms	314.4 ms	294.1 ms	295.2 ms	10	100
16 qubits and 6 measurements	280.3 ms	315.4 ms	296.3 ms	297.2 ms	10	100
16 qubits and 8 measurements	275.4 ms	312.5 ms	293.6 ms	295 ms	10	100
16 qubits and 10 measurements	276.3 ms	304.1 ms	298.9 ms	295.6 ms	10	100
16 qubits and 12 measurements	270.5 ms	305.8 ms	299.9 ms	292.6 ms	10	100
16 qubits and 14 measurements	279.3 ms	314.7 ms	305.5 ms	301.7 ms	10	100
16 qubits and 16 measurements	283.9 ms	315 ms	300.7 ms	297.8 ms	10	100

Table D7: Benchmark of Syntax Simulator on a circuit scaled by number of qubits and number of measurements on individual bits mid circuit.

D.2 Circuit with a large number of gates

Benchmarks done on a circuit scaled by number of qubits and number of single qubit gates are shown in tables D8, D9, D10, D11, D12, D13 and D14.

Cube Simulator	Fastest	Slowest	Median	Mean	Samples	Iterations
8 qubits and 250 gates	181.3 µs	185.1 µs	182.3 µs	182.6 µs	10	100
8 qubits and 500 gates	375 µs	377.3 µs	376.2 µs	376.2 µs	10	100
8 qubits and 1000 gates	737.3 µs	741 µs	738.2 µs	738.6 µs	10	100
8 qubits and 2000 gates	1.483 ms	1.488 ms	1.484 ms	1.485 ms	10	100
16 qubits and 250 gates	318 ms	352.8 ms	336.2 ms	338.3 ms	10	100
16 qubits and 500 gates	548.6 ms	614.7 ms	557.8 ms	562.1 ms	10	100
16 qubits and 1000 gates	1.103 s	1.165 s	1.116 s	1.118 s	10	100
16 qubits and 2000 gates	1.977 s	2.255 s	1.995 s	2.045 s	10	100

Table D8: Benchmark of Cube Simulator on a circuit scaled by number of qubits and number of single qubit gates.

Full Matrix Multiplication Simulator	Fastest	Slowest	Median	Mean	Samples	Iterations
4 qubits and 250 gates	203.3 µs	206.7 µs	204.4 µs	204.6 µs	10	100
4 qubits and 500 gates	406.8 µs	409.3 µs	408.3 µs	408 µs	10	100
4 qubits and 1000 gates	818.2 µs	822.4 µs	820 µs	820.2 µs	10	100
4 qubits and 2000 gates	1.634 ms	1.642 ms	1.637 ms	1.637 ms	10	100
8 qubits and 250 gates	34.51 ms	39.91 ms	37.52 ms	37.47 ms	10	100
8 qubits and 500 gates	73.02 ms	83.49 ms	80.87 ms	79.82 ms	10	100
8 qubits and 1000 gates	140.9 ms	162.7 ms	154 ms	153.3 ms	10	100
8 qubits and 2000 gates	287.4 ms	310.2 ms	297.9 ms	298.1 ms	10	100

Table D9: Benchmark of Full Matrix Multiplication Simulator on a circuit scaled by number of qubits and number of single qubit gates.

GPU accelerated simulator (CUDA)	Fastest	Slowest	Median	Mean	Samples	Iterations
8 qubits and 250 gates	99.69 µs	8.104 ms	102.1 µs	912.5 µs	10	100
8 qubits and 500 gates	99.92 µs	134.4 µs	102.1 µs	108.2 µs	10	100
8 qubits and 1000 gates	100.3 µs	133.7 µs	101.7 µs	105.8 µs	10	100
8 qubits and 2000 gates	99.88 µs	133.9 µs	101.2 µs	104.3 µs	10	100
16 qubits and 250 gates	200.9 µs	235.3 µs	203.6 µs	212.8 µs	10	100
16 qubits and 500 gates	200.3 µs	227.2 µs	202.3 µs	205.6 µs	10	100
16 qubits and 1000 gates	200.4 µs	233.2 µs	203 µs	211 µs	10	100
16 qubits and 2000 gates	198.9 µs	236.3 µs	203.7 µs	210.1 µs	10	100

Table D10: Benchmark of GPU accelerated simulator (CUDA) on a circuit scaled by number of qubits and number of single qubit gates.

GPU accelerated simulator (WGPU)	Fastest	Slowest	Median	Mean	Samples	Iterations
8 qubits and 250 gates	379 µs	2.346 ms	506.1 µs	676 µs	10	100
8 qubits and 500 gates	380.4 µs	712 µs	509.6 µs	502.7 µs	10	100
8 qubits and 1000 gates	374 µs	703.5 µs	384.5 µs	442.9 µs	10	100
8 qubits and 2000 gates	380 µs	630.7 µs	392.4 µs	420.3 µs	10	100
16 qubits and 250 gates	662.2 µs	840.4 µs	700.9 µs	711.2 µs	10	100
16 qubits and 500 gates	624.2 µs	927.8 µs	677 µs	694.6 µs	10	100
16 qubits and 1000 gates	620.6 µs	891 µs	695.5 µs	708 µs	10	100
16 qubits and 2000 gates	615.8 µs	916.2 µs	693.2 µs	700.7 µs	10	100

Table D11: Benchmark of GPU accelerated simulator (WGPU) on a circuit scaled by number of qubits and number of single qubit gates.

Product State Simulator	Fastest	Slowest	Median	Mean	Samples	Iterations
8 qubits and 250 gates	31.24 µs	36.1 µs	32.17 µs	32.72 µs	10	100
8 qubits and 500 gates	63.59 µs	66.01 µs	64.09 µs	64.31 µs	10	100
8 qubits and 1000 gates	126.7 µs	131.1 µs	128 µs	128.6 µs	10	100
8 qubits and 2000 gates	252 µs	256.8 µs	254.2 µs	254.6 µs	10	100
16 qubits and 250 gates	33.42 µs	34.63 µs	33.76 µs	33.93 µs	10	100
16 qubits and 500 gates	65.95 µs	67.19 µs	66.62 µs	66.48 µs	10	100
16 qubits and 1000 gates	128.5 µs	133.9 µs	131.3 µs	131.3 µs	10	100
16 qubits and 2000 gates	255.2 µs	261.9 µs	257.2 µs	257.9 µs	10	100

Table D12: Benchmark of Product State Simulator on a circuit scaled by number of qubits and number of single qubit gates.

State Vector Simulator	Fastest	Slowest	Median	Mean	Samples	Iterations
8 qubits and 250 gates	62.06 µs	66.23 µs	62.84 µs	63.16 µs	10	100
8 qubits and 500 gates	116.2 µs	118.9 µs	116.5 µs	116.9 µs	10	100
8 qubits and 1000 gates	231.3 µs	234.1 µs	232 µs	232.2 µs	10	100
8 qubits and 2000 gates	463.5 µs	475.5 µs	464.4 µs	465.7 µs	10	100
16 qubits and 250 gates	12.15 ms	12.48 ms	12.19 ms	12.25 ms	10	100
16 qubits and 500 gates	24.12 ms	24.53 ms	24.28 ms	24.28 ms	10	100
16 qubits and 1000 gates	47.83 ms	50.07 ms	49.06 ms	48.84 ms	10	100
16 qubits and 2000 gates	95.56 ms	100.1 ms	96.45 ms	96.91 ms	10	100

Table D13: Benchmark of State Vector Simulator on a circuit scaled by number of qubits and number of single qubit gates.

Syntax Simulator	Fastest	Slowest	Median	Mean	Samples	Iterations
8 qubits and 250 gates	33.66 µs	53.02 µs	35.06 µs	39.63 µs	10	100
8 qubits and 500 gates	40.25 µs	64.67 µs	41.17 µs	45.64 µs	10	100
8 qubits and 1000 gates	117.2 µs	162.7 µs	131.9 µs	136.3 µs	10	100
8 qubits and 2000 gates	150.5 µs	217.1 µs	173.8 µs	173.3 µs	10	100
16 qubits and 250 gates	59.93 µs	96.6 µs	73.76 µs	71.77 µs	10	100
16 qubits and 500 gates	1.629 ms	1.967 ms	1.81 ms	1.794 ms	10	100
16 qubits and 1000 gates	138.6 µs	183.3 µs	140.9 µs	148 µs	10	100
16 qubits and 2000 gates	16.14 ms	16.27 ms	16.2 ms	16.2 ms	10	100

Table D14: Benchmark of Syntax Simulator on a circuit scaled by number of qubits and number of single qubit gates.

D.3 QFT

Benchmarks done on a circuit implementing QFT scaled by number of qubits are shown in tables D15, D16, D17, D18, D19, D20 and D21.

Cube Simulator	Fastest	Slowest	Median	Mean	Samples	Iterations
2 qubits	293.1 ns	2.36 µs	334.9 ns	533.3 ns	10	100
4 qubits	900.6 ns	1.347 µs	1.096 µs	1.122 µs	10	100
6 qubits	3.582 µs	3.68 µs	3.638 µs	3.633 µs	10	100
8 qubits	18.5 µs	20.56 µs	18.64 µs	18.94 µs	10	100
10 qubits	147 µs	150.4 µs	148.6 µs	148.5 µs	10	100
12 qubits	982.1 µs	1.114 ms	987.7 µs	1.011 ms	10	100
14 qubits	10.27 ms	10.4 ms	10.35 ms	10.34 ms	10	100
16 qubits	67.1 ms	76.04 ms	69.01 ms	69.81 ms	10	100
18 qubits	695.6 ms	720.4 ms	709.3 ms	709.4 ms	10	100
20 qubits	1.56 s	1.595 s	1.569 s	1.573 s	10	100

Table D15: Benchmark of Cube Simulator on a circuit implementing QFT scaled by number of qubits.

Full Matrix Multiplication Simulator	Fastest	Slowest	Median	Mean	Samples	Iterations
2 qubits	1.822 µs	3.198 µs	1.906 µs	2.02 µs	10	100
4 qubits	27.53 µs	29.15 µs	27.91 µs	28.05 µs	10	100
6 qubits	1.116 ms	1.124 ms	1.118 ms	1.119 ms	10	100
8 qubits	74.68 ms	75.33 ms	74.78 ms	74.88 ms	10	100

Table D16: Benchmark of Full Matrix Multiplication Simulator on a circuit implementing QFT scaled by number of qubits.

GPU accelerated simulator (CUDA)	Fastest	Slowest	Median	Mean	Samples	Iterations
2 qubits	65.47 µs	8.189 ms	68.2 µs	883.3 µs	10	100
4 qubits	81.56 µs	6.576 ms	84.31 µs	737 µs	10	100
6 qubits	100.4 µs	5.156 ms	101.9 µs	607.6 µs	10	100
8 qubits	138.2 µs	5.215 ms	140.5 µs	648 µs	10	100
10 qubits	170.8 µs	212 µs	174.7 µs	180.8 µs	10	100
12 qubits	205.9 µs	238.3 µs	209 µs	216.5 µs	10	100
14 qubits	245.4 µs	279.7 µs	249.2 µs	252 µs	10	100
16 qubits	423.7 µs	464.1 µs	430.4 µs	436.7 µs	10	100
18 qubits	1.054 ms	1.335 ms	1.28 ms	1.26 ms	10	100
20 qubits	4.211 ms	5.836 ms	4.227 ms	4.391 ms	10	100

Table D17: Benchmark of GPU accelerated simulator (CUDA) on a circuit implementing QFT scaled by number of qubits.

GPU accelerated simulator (WGPU)	Fastest	Slowest	Median	Mean	Samples	Iterations
2 qubits	264 µs	2.118 ms	292 µs	473.2 µs	10	100
4 qubits	329.9 µs	1.048 ms	333.6 µs	405.7 µs	10	100
6 qubits	371.4 µs	908.3 µs	375.3 µs	428.2 µs	10	100
8 qubits	450.4 µs	983.9 µs	453.3 µs	506.5 µs	10	100
10 qubits	483.1 µs	737.9 µs	504.2 µs	525.7 µs	10	100
12 qubits	534.8 µs	753.4 µs	560.5 µs	576.2 µs	10	100
14 qubits	614.4 µs	820 µs	639.6 µs	660.2 µs	10	100
16 qubits	823.5 µs	1.028 ms	831.8 µs	857.1 µs	10	100
18 qubits	1.686 ms	1.951 ms	1.717 ms	1.752 ms	10	100
20 qubits	5.454 ms	6.042 ms	5.5 ms	5.544 ms	10	100

Table D18: Benchmark of GPU accelerated simulator (WGPU) on a circuit implementing QFT scaled by number of qubits.

Product State Simulator	Fastest	Slowest	Median	Mean	Samples	Iterations
2 qubits	1.152 µs	2.109 µs	1.235 µs	1.316 µs	10	100
4 qubits	3.023 µs	4.239 µs	3.722 µs	3.711 µs	10	100
6 qubits	7.06 µs	8.339 µs	7.249 µs	7.422 µs	10	100
8 qubits	20.52 µs	22.3 µs	20.98 µs	21.09 µs	10	100
10 qubits	74.66 µs	88.35 µs	75.48 µs	77.09 µs	10	100
12 qubits	367.3 µs	428.1 µs	370.3 µs	379.6 µs	10	100
14 qubits	1.858 ms	1.958 ms	1.881 ms	1.885 ms	10	100
16 qubits	10.18 ms	10.44 ms	10.23 ms	10.24 ms	10	100
18 qubits	49.48 ms	51.29 ms	50.05 ms	50.1 ms	10	100
20 qubits	328.2 ms	331.3 ms	329.9 ms	330 ms	10	100

Table D19: Benchmark of Product State Simulator on a circuit implementing QFT scaled by number of qubits.

State Vector Simulator	Fastest	Slowest	Median	Mean	Samples	Iterations
2 qubits	279.1 ns	565.4 ns	355.9 ns	360.8 ns	10	100
4 qubits	907.7 ns	1.103 µs	994.9 ns	998.4 ns	10	100
6 qubits	3.778 µs	4.665 µs	3.83 µs	3.925 µs	10	100
8 qubits	18.5 µs	19.87 µs	18.79 µs	18.85 µs	10	100
10 qubits	95.78 µs	96.32 µs	95.88 µs	95.95 µs	10	100
12 qubits	492.9 µs	499.7 µs	494.5 µs	495.1 µs	10	100
14 qubits	2.422 ms	2.54 ms	2.437 ms	2.468 ms	10	100
16 qubits	12.1 ms	12.22 ms	12.14 ms	12.14 ms	10	100
18 qubits	60.03 ms	61.62 ms	60.76 ms	60.78 ms	10	100
20 qubits	290 ms	297.2 ms	293.4 ms	293.3 ms	10	100

Table D20: Benchmark of State Vector Simulator on a circuit implementing QFT scaled by number of qubits.

Syntax Simulator	Fastest	Slowest	Median	Mean	Samples	Iterations
2 qubits	2.101 µs	3.442 µs	2.185 µs	2.301 µs	10	100
4 qubits	10.6 µs	12.32 µs	10.67 µs	10.97 µs	10	100
6 qubits	46.74 µs	50.52 µs	48.2 µs	48.21 µs	10	100
8 qubits	207.3 µs	212.9 µs	210.2 µs	210.2 µs	10	100
10 qubits	957.8 µs	966.6 µs	961.4 µs	962 µs	10	100
12 qubits	4.377 ms	4.434 ms	4.398 ms	4.399 ms	10	100
14 qubits	19.68 ms	19.93 ms	19.83 ms	19.82 ms	10	100
16 qubits	88.54 ms	91.08 ms	89.21 ms	89.38 ms	10	100
18 qubits	398.1 ms	403.9 ms	400.4 ms	400.6 ms	10	100
20 qubits	1.822 s	1.879 s	1.856 s	1.854 s	10	100

Table D21: Benchmark of Syntax Simulator on a circuit implementing QFT scaled by number of qubits.

D.4 Circuit with a single large system

Benchmarks done on a circuit that extends the Hadamard-CNOT to several qubits and is scaled by number of qubits are shown in tables D22, D23, D24, D25, D26, D27 and D28.

Cube Simulator	Fastest	Slowest	Median	Mean	Samples	Iterations
2 qubits	230.2 ns	1.962 µs	240.7 ns	425.7 ns	10	100
4 qubits	390.8 ns	621.3 ns	474.7 ns	501.2 ns	10	100
6 qubits	1.089 µs	1.361 µs	1.155 µs	1.186 µs	10	100
8 qubits	4.253 µs	4.378 µs	4.326 µs	4.327 µs	10	100
10 qubits	24.22 µs	25.9 µs	25.13 µs	25.17 µs	10	100
12 qubits	123.2 µs	130.7 µs	124.5 µs	125.5 µs	10	100
14 qubits	1.288 ms	1.317 ms	1.305 ms	1.304 ms	10	100
16 qubits	8.366 ms	9.356 ms	8.85 ms	8.87 ms	10	100
18 qubits	54.29 ms	61.28 ms	56.62 ms	56.92 ms	10	100
20 qubits	134.9 ms	147.1 ms	135.8 ms	137.2 ms	10	100

Table D22: Benchmark of Cube Simulator on a circuit that extends the Hadamard-CNOT to several qubits and is scaled by number of qubits.

Full Matrix Multiplication Simulator	Fastest	Slowest	Median	Mean	Samples	Iterations
2 qubits	628.2 ns	2.563 µs	715.5 ns	893 ns	10	100
4 qubits	5.391 µs	5.866 µs	5.44 µs	5.483 µs	10	100
6 qubits	83.74 µs	84.68 µs	83.91 µs	83.99 µs	10	100
8 qubits	1.8 ms	1.82 ms	1.811 ms	1.81 ms	10	100
10 qubits	48.58 ms	51.16 ms	49.3 ms	49.7 ms	10	100

Table D23: Benchmark of Full Matrix Multiplication Simulator on a circuit that extends the Hadamard-CNOT to several qubits and is scaled by number of qubits.

GPU accelerated simulator (CUDA)	Fastest	Slowest	Median	Mean	Samples	Iterations
2 qubits	60.16 µs	8.281 ms	61.54 µs	885.9 µs	10	100
4 qubits	68.68 µs	5.138 ms	75.71 µs	589.5 µs	10	100
6 qubits	68.75 µs	2.301 ms	70.65 µs	293.6 µs	10	100
8 qubits	88.14 µs	5.305 ms	89.46 µs	611.9 µs	10	100
10 qubits	100.7 µs	1.597 ms	103.2 µs	253.1 µs	10	100
12 qubits	110 µs	161.2 µs	111.5 µs	122.9 µs	10	100
14 qubits	121.2 µs	153.9 µs	124 µs	126.7 µs	10	100
16 qubits	191.3 µs	222.3 µs	195.1 µs	202.8 µs	10	100
18 qubits	436.4 µs	483.8 µs	451.5 µs	457 µs	10	100
20 qubits	1.554 ms	3.511 ms	1.573 ms	1.768 ms	10	100

Table D24: Benchmark of GPU accelerated simulator (CUDA) on a circuit that extends the Hadamard-CNOT to several qubits and is scaled by number of qubits.

GPU accelerated simulator (WGPU)	Fastest	Slowest	Median	Mean	Samples	Iterations
2 qubits	260.1 µs	2.897 ms	401.5 µs	613.1 µs	10	100
4 qubits	394.1 µs	950.3 µs	419.7 µs	470.4 µs	10	100
6 qubits	402.9 µs	701.8 µs	414.6 µs	446.4 µs	10	100
8 qubits	324 µs	979.7 µs	394.5 µs	442.8 µs	10	100
10 qubits	336.6 µs	732.1 µs	398.6 µs	420.5 µs	10	100
12 qubits	349.5 µs	594 µs	488.6 µs	467.7 µs	10	100
14 qubits	478.7 µs	602.2 µs	502.8 µs	508.5 µs	10	100
16 qubits	568 µs	774.8 µs	571.9 µs	595.5 µs	10	100
18 qubits	713.5 µs	1.067 ms	719.7 µs	773.4 µs	10	100
20 qubits	1.811 ms	2.409 ms	1.885 ms	1.933 ms	10	100

Table D25: Benchmark of GPU accelerated simulator (WGPU) on a circuit that extends the Hadamard-CNOT to several qubits and is scaled by number of qubits.

Product State Simulator	Fastest	Slowest	Median	Mean	Samples	Iterations
2 qubits	956.6 ns	1.738 µs	1.043 µs	1.108 µs	10	100
4 qubits	2.276 µs	2.535 µs	2.294 µs	2.331 µs	10	100
6 qubits	4.078 µs	4.965 µs	4.134 µs	4.278 µs	10	100
8 qubits	6.676 µs	7.528 µs	6.868 µs	6.909 µs	10	100
10 qubits	13.53 µs	15.47 µs	13.81 µs	14.01 µs	10	100
12 qubits	32.4 µs	34.11 µs	33.03 µs	32.98 µs	10	100
14 qubits	83.17 µs	93.63 µs	90.28 µs	89.48 µs	10	100
16 qubits	307.3 µs	319 µs	309.3 µs	310.4 µs	10	100
18 qubits	1.189 ms	1.265 ms	1.21 ms	1.212 ms	10	100
20 qubits	6.691 ms	6.812 ms	6.746 ms	6.754 ms	10	100

Table D26: Benchmark of Product State Simulator on a circuit that extends the Hadamard-CNOT to several qubits and is scaled by number of qubits.

State Vector Simulator	Fastest	Slowest	Median	Mean	Samples	Iterations
2 qubits	188.3 ns	390.8 ns	261.6 ns	270.6 ns	10	100
4 qubits	341.9 ns	614.4 ns	530.5 ns	519.3 ns	10	100
6 qubits	760.9 ns	942.6 ns	848.3 ns	842.7 ns	10	100
8 qubits	2.667 µs	3.764 µs	2.688 µs	2.799 µs	10	100
10 qubits	10.16 µs	11 µs	10.27 µs	10.34 µs	10	100
12 qubits	43.17 µs	46.02 µs	45.14 µs	44.81 µs	10	100
14 qubits	186.5 µs	192.4 µs	187.6 µs	188.2 µs	10	100
16 qubits	816.8 µs	823.3 µs	817.9 µs	818.5 µs	10	100
18 qubits	3.555 ms	3.58 ms	3.572 ms	3.569 ms	10	100
20 qubits	15.48 ms	15.62 ms	15.53 ms	15.55 ms	10	100

Table D27: Benchmark of State Vector Simulator on a circuit that extends the Hadamard-CNOT to several qubits and is scaled by number of qubits.

Syntax Simulator	Fastest	Slowest	Median	Mean	Samples	Iterations
2 qubits	705.1 ns	1.682 µs	767.9 ns	844.8 ns	10	100
4 qubits	1.249 µs	1.382 µs	1.333 µs	1.322 µs	10	100
6 qubits	1.759 µs	1.892 µs	1.829 µs	1.835 µs	10	100
8 qubits	2.325 µs	2.381 µs	2.363 µs	2.358 µs	10	100
10 qubits	2.863 µs	3.051 µs	2.901 µs	2.912 µs	10	100
12 qubits	3.387 µs	4.176 µs	3.439 µs	3.517 µs	10	100
14 qubits	4.085 µs	5.908 µs	4.155 µs	4.324 µs	10	100
16 qubits	4.811 µs	7.151 µs	4.916 µs	5.152 µs	10	100
18 qubits	5.461 µs	5.719 µs	5.576 µs	5.589 µs	10	100
20 qubits	6.097 µs	6.914 µs	6.152 µs	6.265 µs	10	100

Table D28: Benchmark of Syntax Simulator on a circuit that extends the Hadamard-CNOT to several qubits and is scaled by number of qubits.

D.5 Circuit with separate entangled systems

Benchmarks done on a circuit scaled by number of total qubits and number of qubits per entangled system are shown in tables D29, D30, D31, D32, D33, D34 and D35.

Cube Simulator	Fastest	Slowest	Median	Mean	Samples	Iterations
8 qubits and 2 qubits per system	10.62 µs	12.79 µs	10.74 µs	11.02 µs	10	100
8 qubits and 4 qubits per system	13.84 µs	18.24 µs	13.99 µs	14.52 µs	10	100
8 qubits and 6 qubits per system	12.92 µs	23.04 µs	13.18 µs	14.56 µs	10	100
8 qubits and 8 qubits per system	20.17 µs	22.64 µs	20.26 µs	20.63 µs	10	100
16 qubits and 2 qubits per system	21.64 ms	28.2 ms	23.32 ms	24.22 ms	10	100
16 qubits and 4 qubits per system	32.81 ms	41.05 ms	37.58 ms	37.42 ms	10	100
16 qubits and 6 qubits per system	27.34 ms	38.9 ms	32.16 ms	32.85 ms	10	100
16 qubits and 8 qubits per system	50.81 ms	61.33 ms	52.88 ms	54.73 ms	10	100
16 qubits and 10 qubits per system	34.44 ms	44.66 ms	38.49 ms	38.96 ms	10	100
16 qubits and 12 qubits per system	42.25 ms	52.2 ms	44.2 ms	44.97 ms	10	100
16 qubits and 14 qubits per system	56.49 ms	62.91 ms	58.85 ms	59.02 ms	10	100
16 qubits and 16 qubits per system	70.46 ms	87.71 ms	76.51 ms	76.67 ms	10	100

Table D29: Benchmark of Cube Simulator on a circuit scaled by number of total qubits and number of qubits per entangled system.

Full Matrix Multiplication Simulator	Fastest	Slowest	Median	Mean	Samples	Iterations
4 qubits and 2 qubits per system	21.1 µs	23.19 µs	21.25 µs	21.6 µs	10	100
4 qubits and 4 qubits per system	26.75 µs	27.97 µs	27.11 µs	27.24 µs	10	100
8 qubits and 2 qubits per system	70.15 ms	70.44 ms	70.34 ms	70.33 ms	10	100
8 qubits and 4 qubits per system	72.22 ms	73.5 ms	72.28 ms	72.41 ms	10	100
8 qubits and 6 qubits per system	55.65 ms	55.93 ms	55.72 ms	55.74 ms	10	100
8 qubits and 8 qubits per system	76.04 ms	76.24 ms	76.09 ms	76.12 ms	10	100

Table D30: Benchmark of Full Matrix Multiplication Simulator on a circuit scaled by number of total qubits and number of qubits per entangled system.

GPU accelerated simulator (CUDA)	Fastest	Slowest	Median	Mean	Samples	Iterations
8 qubits and 2 qubits per system	98.28 µs	8.133 ms	100.6 µs	907.3 µs	10	100
8 qubits and 4 qubits per system	115.5 µs	3.04 ms	117.5 µs	409.6 µs	10	100
8 qubits and 6 qubits per system	114.2 µs	149.4 µs	116.4 µs	119.4 µs	10	100
8 qubits and 8 qubits per system	137.3 µs	174.8 µs	140.3 µs	146.8 µs	10	100
16 qubits and 2 qubits per system	223.1 µs	259 µs	229.3 µs	237.4 µs	10	100
16 qubits and 4 qubits per system	294.5 µs	325.3 µs	299.5 µs	304.2 µs	10	100
16 qubits and 6 qubits per system	275.9 µs	332.8 µs	278.3 µs	287.3 µs	10	100
16 qubits and 8 qubits per system	324.8 µs	351.4 µs	330.5 µs	336 µs	10	100
16 qubits and 10 qubits per system	285.7 µs	315.7 µs	290.2 µs	295 µs	10	100
16 qubits and 12 qubits per system	321.1 µs	365.3 µs	327.1 µs	334.2 µs	10	100
16 qubits and 14 qubits per system	360.3 µs	423.3 µs	368.9 µs	378.2 µs	10	100
16 qubits and 16 qubits per system	363.4 µs	486 µs	380.3 µs	389.2 µs	10	100

Table D31: Benchmark of GPU accelerated simulator (CUDA) on a circuit scaled by number of total qubits and number of qubits per entangled system.

GPU accelerated simulator (WGPU)	Fastest	Slowest	Median	Mean	Samples	Iterations
8 qubits and 2 qubits per system	345.5 µs	4.529 ms	360 µs	778.1 µs	10	100
8 qubits and 4 qubits per system	384.5 µs	3.414 ms	390 µs	692.2 µs	10	100
8 qubits and 6 qubits per system	371.5 µs	624 µs	379.2 µs	407.9 µs	10	100
8 qubits and 8 qubits per system	412.3 µs	637.1 µs	418.1 µs	455.9 µs	10	100
16 qubits and 2 qubits per system	545.2 µs	757.4 µs	622.8 µs	620.4 µs	10	100
16 qubits and 4 qubits per system	623.8 µs	838.3 µs	633.4 µs	659.1 µs	10	100
16 qubits and 6 qubits per system	613.6 µs	792.5 µs	620.3 µs	640 µs	10	100
16 qubits and 8 qubits per system	700.7 µs	904 µs	706.7 µs	729.1 µs	10	100
16 qubits and 10 qubits per system	651 µs	808.6 µs	711.3 µs	715.8 µs	10	100
16 qubits and 12 qubits per system	674.8 µs	916.8 µs	682.4 µs	712.9 µs	10	100
16 qubits and 14 qubits per system	751.9 µs	954 µs	760.2 µs	782.8 µs	10	100
16 qubits and 16 qubits per system	806.8 µs	971.8 µs	810.3 µs	837.7 µs	10	100

Table D32: Benchmark of GPU accelerated simulator (WGPU) on a circuit scaled by number of total qubits and number of qubits per entangled system.

Product State Simulator	Fastest	Slowest	Median	Mean	Samples	Iterations
8 qubits and 2 qubits per system	5 µs	6.39 µs	5.087 µs	5.346 µs	10	100
8 qubits and 4 qubits per system	7.228 µs	8.918 µs	7.493 µs	7.704 µs	10	100
8 qubits and 6 qubits per system	8.541 µs	10.67 µs	8.695 µs	9.119 µs	10	100
8 qubits and 8 qubits per system	21.83 µs	24.6 µs	22.23 µs	22.85 µs	10	100
16 qubits and 2 qubits per system	12.44 µs	14.36 µs	12.72 µs	13.02 µs	10	100
16 qubits and 4 qubits per system	18.64 µs	22.37 µs	18.94 µs	19.52 µs	10	100
16 qubits and 6 qubits per system	21.22 µs	32.91 µs	21.57 µs	22.76 µs	10	100
16 qubits and 8 qubits per system	49.52 µs	54.7 µs	52.03 µs	51.75 µs	10	100
16 qubits and 10 qubits per system	82.77 µs	99.6 µs	84.75 µs	87.73 µs	10	100
16 qubits and 12 qubits per system	354.7 µs	389.3 µs	359.5 µs	363.8 µs	10	100
16 qubits and 14 qubits per system	1.753 ms	1.869 ms	1.756 ms	1.771 ms	10	100
16 qubits and 16 qubits per system	9.267 ms	10.86 ms	9.335 ms	9.511 ms	10	100

Table D33: Benchmark of Product State Simulator on a circuit scaled by number of total qubits and number of qubits per entangled system.

State Vector Simulator	Fastest	Slowest	Median	Mean	Samples	Iterations
8 qubits and 2 qubits per system	12.69 µs	14.02 µs	13.12 µs	13.23 µs	10	100
8 qubits and 4 qubits per system	15.29 µs	16.6 µs	15.48 µs	15.71 µs	10	100
8 qubits and 6 qubits per system	12.81 µs	13.87 µs	12.87 µs	13.05 µs	10	100
8 qubits and 8 qubits per system	19.56 µs	23.18 µs	20.41 µs	20.68 µs	10	100
16 qubits and 2 qubits per system	5.284 ms	5.857 ms	5.509 ms	5.552 ms	10	100
16 qubits and 4 qubits per system	6.216 ms	6.875 ms	6.545 ms	6.537 ms	10	100
16 qubits and 6 qubits per system	5.678 ms	6.252 ms	5.994 ms	5.978 ms	10	100
16 qubits and 8 qubits per system	8.358 ms	9.038 ms	8.569 ms	8.608 ms	10	100
16 qubits and 10 qubits per system	5.758 ms	6.597 ms	6.171 ms	6.154 ms	10	100
16 qubits and 12 qubits per system	7.692 ms	8.196 ms	7.877 ms	7.884 ms	10	100
16 qubits and 14 qubits per system	9.726 ms	10.18 ms	9.95 ms	9.946 ms	10	100
16 qubits and 16 qubits per system	11.98 ms	12.38 ms	12.08 ms	12.11 ms	10	100

Table D34: Benchmark of State Vector Simulator on a circuit scaled by number of total qubits and number of qubits per entangled system.

Syntax Simulator	Fastest	Slowest	Median	Mean	Samples	Iterations
8 qubits and 2 qubits per system	222 µs	227 µs	223.3 µs	223.6 µs	10	100
8 qubits and 4 qubits per system	214.6 µs	216.9 µs	216 µs	215.8 µs	10	100
8 qubits and 6 qubits per system	47.73 µs	50.26 µs	48.18 µs	48.4 µs	10	100
8 qubits and 8 qubits per system	209.3 µs	212.3 µs	210.1 µs	210.4 µs	10	100
16 qubits and 2 qubits per system	85.12 ms	90.63 ms	87.72 ms	87.54 ms	10	100
16 qubits and 4 qubits per system	81.84 ms	91.33 ms	86.72 ms	87.2 ms	10	100
16 qubits and 6 qubits per system	4.237 ms	4.443 ms	4.409 ms	4.38 ms	10	100
16 qubits and 8 qubits per system	85.25 ms	89.93 ms	88.26 ms	87.68 ms	10	100
16 qubits and 10 qubits per system	959.5 µs	967.9 µs	964.5 µs	964.2 µs	10	100
16 qubits and 12 qubits per system	4.357 ms	4.431 ms	4.387 ms	4.392 ms	10	100
16 qubits and 14 qubits per system	19.04 ms	19.79 ms	19.61 ms	19.48 ms	10	100
16 qubits and 16 qubits per system	82.6 ms	89.13 ms	85.34 ms	85.6 ms	10	100

Table D35: Benchmark of Syntax Simulator on a circuit scaled by number of total qubits and number of qubits per entangled system.

D.6 Bernstein Vazirani's algorithm

Benchmarks done on Bernstein Vazirani's algorithm scaled by number of qubits are shown in tables D36, D37, D38, D39, D40, D41 and D42.

Cube Simulator	Fastest	Slowest	Median	Mean	Samples	Iterations
2 qubits	607.4 ns	2.185 µs	659.7 ns	879.7 ns	10	100
4 qubits	1.717 µs	1.829 µs	1.77 µs	1.77 µs	10	100
6 qubits	6.816 µs	6.949 µs	6.84 µs	6.852 µs	10	100
8 qubits	37.87 µs	39.67 µs	38.03 µs	38.32 µs	10	100
10 qubits	234 µs	250.8 µs	235.9 µs	239.4 µs	10	100
12 qubits	2.38 ms	2.453 ms	2.419 ms	2.417 ms	10	100
14 qubits	13.01 ms	13.22 ms	13.1 ms	13.11 ms	10	100
16 qubits	122.5 ms	144.7 ms	131.3 ms	130.8 ms	10	100
18 qubits	312.2 ms	323.2 ms	315.2 ms	315.6 ms	10	100
20 qubits	1.224 s	1.233 s	1.228 s	1.228 s	10	100

Table D36: Benchmark of Cube Simulator on Bernstein Vazirani's algorithm scaled by number of qubits.

Full Matrix Multiplication Simulator	Fastest	Slowest	Median	Mean	Samples	Iterations
2 qubits	3.289 µs	4.665 µs	3.387 µs	3.513 µs	10	100
4 qubits	40.57 µs	42.17 µs	40.86 µs	41.11 µs	10	100
6 qubits	830.2 µs	840.9 µs	833.9 µs	834.7 µs	10	100
8 qubits	17.21 ms	20.44 ms	17.65 ms	18.03 ms	10	100
10 qubits	565.3 ms	571.4 ms	568.1 ms	568.1 ms	10	100

Table D37: Benchmark of Full Matrix Multiplication Simulator on Bernstein Vazirani's algorithm scaled by number of qubits.

GPU accelerated simulator (CUDA)	Fastest	Slowest	Median	Mean	Samples	Iterations
2 qubits	66.68 µs	21.33 ms	67.71 µs	2.197 ms	10	100
4 qubits	80.66 µs	5.071 ms	82.33 µs	581.8 µs	10	100
6 qubits	92.9 µs	5.085 ms	95.68 µs	598 µs	10	100
8 qubits	117.6 µs	1.682 ms	121.1 µs	280.6 µs	10	100
10 qubits	133.9 µs	146.9 µs	136.1 µs	138.2 µs	10	100
12 qubits	148.5 µs	184.7 µs	150.4 µs	159.7 µs	10	100
14 qubits	192.4 µs	227.2 µs	198.3 µs	205.1 µs	10	100
16 qubits	386.4 µs	431.6 µs	402.7 µs	408 µs	10	100
18 qubits	1.26 ms	2.984 ms	1.278 ms	1.446 ms	10	100
20 qubits	4.529 ms	5.285 ms	4.541 ms	4.619 ms	10	100

Table D38: Benchmark of GPU accelerated simulator (CUDA) on Bernstein Vazirani's algorithm scaled by number of qubits.

GPU accelerated simulator (WGPU)	Fastest	Slowest	Median	Mean	Samples	Iterations
2 qubits	266.5 µs	2.944 ms	280.3 µs	547.2 µs	10	100
4 qubits	305.3 µs	862.6 µs	324.4 µs	375.3 µs	10	100
6 qubits	334.4 µs	891.6 µs	434.9 µs	458.1 µs	10	100
8 qubits	383.3 µs	785.2 µs	395.4 µs	435.2 µs	10	100
10 qubits	500.4 µs	730.1 µs	523.5 µs	545.7 µs	10	100
12 qubits	441.4 µs	806.8 µs	460.7 µs	519.4 µs	10	100
14 qubits	547.3 µs	771 µs	558.1 µs	582.4 µs	10	100
16 qubits	755.5 µs	942.2 µs	762.5 µs	795.6 µs	10	100
18 qubits	1.631 ms	1.919 ms	1.659 ms	1.698 ms	10	100
20 qubits	5.366 ms	6.637 ms	5.39 ms	5.507 ms	10	100

Table D39: Benchmark of GPU accelerated simulator (WGPU) on Bernstein Vazirani's algorithm scaled by number of qubits.

Product State Simulator	Fastest	Slowest	Median	Mean	Samples	Iterations
2 qubits	2.109 µs	4.092 µs	2.199 µs	2.541 µs	10	100
4 qubits	4.469 µs	5.768 µs	4.536 µs	4.677 µs	10	100
6 qubits	6.432 µs	9.351 µs	6.743 µs	7.345 µs	10	100
8 qubits	14.38 µs	15.95 µs	14.6 µs	14.82 µs	10	100
10 qubits	41.03 µs	46.19 µs	42.51 µs	42.89 µs	10	100
12 qubits	151.1 µs	175.4 µs	155.9 µs	159.6 µs	10	100
14 qubits	642.8 µs	734.2 µs	646.9 µs	660.3 µs	10	100
16 qubits	2.752 ms	2.916 ms	2.774 ms	2.795 ms	10	100
18 qubits	12.38 ms	12.91 ms	12.59 ms	12.61 ms	10	100
20 qubits	87.13 ms	90.92 ms	87.91 ms	88.11 ms	10	100

Table D40: Benchmark of Product State Simulator on Bernstein Vazirani's algorithm scaled by number of qubits.

State Vector Simulator	Fastest	Slowest	Median	Mean	Samples	Iterations
2 qubits	474.6 ns	698.1 ns	534 ns	545.8 ns	10	100
4 qubits	1.145 µs	1.717 µs	1.215 µs	1.282 µs	10	100
6 qubits	3.917 µs	4.874 µs	3.956 µs	4.106 µs	10	100
8 qubits	16.18 µs	16.99 µs	16.23 µs	16.35 µs	10	100
10 qubits	72.67 µs	73.55 µs	73.41 µs	73.27 µs	10	100
12 qubits	333.6 µs	335 µs	333.9 µs	334.2 µs	10	100
14 qubits	1.519 ms	1.524 ms	1.52 ms	1.52 ms	10	100
16 qubits	5.914 ms	6.905 ms	5.925 ms	6.158 ms	10	100
18 qubits	21.67 ms	30.61 ms	26.13 ms	27.05 ms	10	100
20 qubits	108.2 ms	134.4 ms	110.8 ms	113.5 ms	10	100

Table D41: Benchmark of State Vector Simulator on Bernstein Vazirani's algorithm scaled by number of qubits.

Syntax Simulator	Fastest	Slowest	Median	Mean	Samples	Iterations
2 qubits	4.616 µs	7.878 µs	4.717 µs	5.025 µs	10	100
4 qubits	65.26 µs	68.88 µs	65.95 µs	66.14 µs	10	100
6 qubits	1.126 ms	1.169 ms	1.131 ms	1.136 ms	10	100
8 qubits	19.84 ms	19.91 ms	19.88 ms	19.88 ms	10	100
10 qubits	356.6 ms	359 ms	357.5 ms	357.7 ms	10	100

Table D42: Benchmark of Syntax Simulator on Bernstein Vazirani's algorithm scaled by number of qubits.

D.7 Grover's algorithm

Benchmarks done on Grover's algorithm scaled by number of qubits are shown in tables D43, D44, D45, D46, D47, D48 and D49.

Cube Simulator	Fastest	Slowest	Median	Mean	Samples	Iterations
2 qubits	1.312 µs	3.289 µs	1.337 µs	1.536 µs	10	100
4 qubits	8.164 µs	8.688 µs	8.199 µs	8.251 µs	10	100
6 qubits	49.84 µs	52.7 µs	50.34 µs	50.5 µs	10	100
8 qubits	437.9 µs	453.9 µs	439.6 µs	441.5 µs	10	100
10 qubits	5.093 ms	5.102 ms	5.095 ms	5.096 ms	10	100
12 qubits	80.25 ms	80.69 ms	80.51 ms	80.51 ms	10	100
14 qubits	1.333 s	1.346 s	1.342 s	1.341 s	10	100

Table D43: Benchmark of Cube Simulator on Grover's algorithm scaled by number of qubits.

Full Matrix Multiplication Simulator	Fastest	Slowest	Median	Mean	Samples	Iterations
2 qubits	3.959 µs	5.587 µs	4.043 µs	4.188 µs	10	100
4 qubits	94.39 µs	95.92 µs	94.8 µs	94.87 µs	10	100
6 qubits	4.274 ms	4.281 ms	4.277 ms	4.277 ms	10	100
8 qubits	410.8 ms	448.5 ms	430.5 ms	429.1 ms	10	100

Table D44: Benchmark of Full Matrix Multiplication Simulator on Grover's algorithm scaled by number of qubits.

GPU accelerated simulator (CUDA)	Fastest	Slowest	Median	Mean	Samples	Iterations
2 qubits	102.5 µs	8.979 ms	107.9 µs	997.5 µs	10	100
4 qubits	174.2 µs	5.332 ms	176.9 µs	695.7 µs	10	100
6 qubits	300.2 µs	3.954 ms	303.3 µs	674.2 µs	10	100
8 qubits	609.8 µs	4.241 ms	614.3 µs	987.3 µs	10	100
10 qubits	1.378 ms	1.429 ms	1.406 ms	1.406 ms	10	100
12 qubits	3.241 ms	3.275 ms	3.254 ms	3.255 ms	10	100
14 qubits	7.507 ms	7.621 ms	7.575 ms	7.566 ms	10	100
16 qubits	20.9 ms	21.09 ms	21.03 ms	21.02 ms	10	100
18 qubits	129.3 ms	130.5 ms	130.3 ms	130.2 ms	10	100
20 qubits	1.504 s	1.512 s	1.509 s	1.509 s	10	100

Table D45: Benchmark of GPU accelerated simulator (CUDA) on Grover's algorithm scaled by number of qubits.

GPU accelerated simulator (WGPU)	Fastest	Slowest	Median	Mean	Samples	Iterations
2 qubits	480.1 µs	2.384 ms	497.2 µs	704.7 µs	10	100
4 qubits	617 µs	1.003 ms	644.1 µs	714.4 µs	10	100
6 qubits	907.1 µs	1.143 ms	929.2 µs	965.2 µs	10	100
8 qubits	1.607 ms	2.165 ms	1.695 ms	1.764 ms	10	100
10 qubits	3.342 ms	3.971 ms	3.462 ms	3.5 ms	10	100
12 qubits	7.383 ms	7.704 ms	7.581 ms	7.561 ms	10	100
14 qubits	17.3 ms	18.71 ms	18.1 ms	18.07 ms	10	100
16 qubits	39.46 ms	42.82 ms	40.93 ms	40.96 ms	10	100
18 qubits	207.1 ms	209.4 ms	207.4 ms	207.9 ms	10	100
20 qubits	1.692 s	1.698 s	1.697 s	1.696 s	10	100

Table D46: Benchmark of GPU accelerated simulator (WGPU) on Grover's algorithm scaled by number of qubits.

Product State Simulator	Fastest	Slowest	Median	Mean	Samples	Iterations
2 qubits	4.183 µs	8.024 µs	5.017 µs	5.342 µs	10	100
4 qubits	14.52 µs	18.83 µs	14.91 µs	15.56 µs	10	100
6 qubits	50.74 µs	60.94 µs	52.91 µs	53.63 µs	10	100
8 qubits	249.6 µs	305.7 µs	270.5 µs	270.6 µs	10	100
10 qubits	2.014 ms	2.445 ms	2.038 ms	2.082 ms	10	100
12 qubits	13.89 ms	17.57 ms	15.26 ms	15.48 ms	10	100
14 qubits	132.9 ms	158.1 ms	153.3 ms	148.8 ms	10	100
16 qubits	1.338 s	1.444 s	1.375 s	1.387 s	10	100

Table D47: Benchmark of Product State Simulator on Grover's algorithm scaled by number of qubits.

State Vector Simulator	Fastest	Slowest	Median	Mean	Samples	Iterations
2 qubits	1.075 µs	1.501 µs	1.166 µs	1.186 µs	10	100
4 qubits	6.097 µs	6.97 µs	6.152 µs	6.252 µs	10	100
6 qubits	27.6 µs	28.6 µs	28.02 µs	28.09 µs	10	100
8 qubits	169.6 µs	173.4 µs	170.4 µs	170.6 µs	10	100
10 qubits	1.428 ms	1.476 ms	1.434 ms	1.437 ms	10	100
12 qubits	12.89 ms	14.78 ms	14.66 ms	14.16 ms	10	100
14 qubits	110.8 ms	132.6 ms	125.1 ms	123.3 ms	10	100
16 qubits	1.012 s	1.191 s	1.076 s	1.088 s	10	100

Table D48: Benchmark of State Vector Simulator on Grover's algorithm scaled by number of qubits.

Syntax Simulator	Fastest	Slowest	Median	Mean	Samples	Iterations
2 qubits	4.504 µs	6.048 µs	4.525 µs	4.765 µs	10	100
4 qubits	633.8 ms	639 ms	634.7 ms	635.3 ms	10	100

Table D49: Benchmark of Syntax Simulator on Grover's algorithm scaled by number of qubits.

D.8 Shor's algorithm

Benchmarks done on Shor's algorithm scaled by number of n bits in N. The number of total qubits is $2n + 3$ are shown in tables D50, D51, D52, D53, D54, D55 and D56.

Cube Simulator	Fastest	Slowest	Median	Mean	Samples	Iterations
7 qubits	199.8 µs	202.2 µs	200.4 µs	200.4 µs	10	100
9 qubits	1.326 ms	1.333 ms	1.328 ms	1.329 ms	10	100
11 qubits	11.65 ms	11.73 ms	11.66 ms	11.67 ms	10	100
13 qubits	125.3 ms	126.3 ms	125.9 ms	125.9 ms	10	100
15 qubits	1.77 s	1.791 s	1.784 s	1.782 s	10	100
17 qubits	14.49 s	14.54 s	14.51 s	14.52 s	10	100
19 qubits	3.358 m	3.775 m	3.765 m	3.633 m	10	100

Table D50: Benchmark of Cube Simulator on Shor's algorithm scaled by number of n bits in N. The number of total qubits is $2n + 3$.

Full Matrix Multiplication Simulator	Fastest	Slowest	Median	Mean	Samples	Iterations
7 qubits	323.3 ms	325.2 ms	323.9 ms	324 ms	10	100
9 qubits	54.71 s	54.78 s	54.74 s	54.75 s	10	100

Table D51: Benchmark of Full Matrix Multiplication Simulator on Shor's algorithm scaled by number of n bits in N. The number of total qubits is $2n + 3$.

GPU accelerated simulator (CUDA)	Fastest	Slowest	Median	Mean	Samples	Iterations
7 qubits	1.335 ms	27.85 ms	1.356 ms	4.007 ms	10	100
9 qubits	3.704 ms	6.188 ms	3.762 ms	3.995 ms	10	100
11 qubits	8.504 ms	8.594 ms	8.513 ms	8.522 ms	10	100
13 qubits	16.24 ms	16.36 ms	16.26 ms	16.28 ms	10	100
15 qubits	29.78 ms	30.59 ms	29.83 ms	30.01 ms	10	100
17 qubits	101.2 ms	103.1 ms	101.2 ms	101.6 ms	10	100
19 qubits	468.3 ms	473.3 ms	471.5 ms	470.9 ms	10	100

Table D52: Benchmark of GPU accelerated simulator (CUDA) on Shor's algorithm scaled by number of n bits in N. The number of total qubits is $2n + 3$.

GPU accelerated simulator (WGPU)	Fastest	Slowest	Median	Mean	Samples	Iterations
7 qubits	2.897 ms	3.905 ms	2.93 ms	3.068 ms	10	100
9 qubits	9.069 ms	9.661 ms	9.28 ms	9.312 ms	10	100
11 qubits	18.83 ms	19.18 ms	18.97 ms	18.99 ms	10	100
13 qubits	34.38 ms	35.15 ms	34.73 ms	34.74 ms	10	100
15 qubits	60.67 ms	62.34 ms	60.97 ms	61.11 ms	10	100
17 qubits	158 ms	160.2 ms	159.1 ms	159.1 ms	10	100
19 qubits	683.8 ms	691.3 ms	688.2 ms	688 ms	10	100

Table D53: Benchmark of GPU accelerated simulator (WGPU) on Shor's algorithm scaled by number of n bits in N. The number of total qubits is $2n + 3$.

Product State Simulator	Fastest	Slowest	Median	Mean	Samples	Iterations
7 qubits	463.6 µs	503.1 µs	465 µs	468.7 µs	10	100
9 qubits	2.918 ms	2.945 ms	2.936 ms	2.932 ms	10	100
11 qubits	16.55 ms	16.82 ms	16.67 ms	16.67 ms	10	100
13 qubits	119.6 ms	121.2 ms	120 ms	120.2 ms	10	100
15 qubits	816.1 ms	832.9 ms	819.4 ms	821.3 ms	10	100
17 qubits	5.599 s	5.636 s	5.619 s	5.618 s	10	100
19 qubits	35.11 s	35.45 s	35.29 s	35.29 s	10	100

Table D54: Benchmark of Product State Simulator on Shor's algorithm scaled by number of n bits in N. The number of total qubits is $2n + 3$.

State Vector Simulator	Fastest	Slowest	Median	Mean	Samples	Iterations
7 qubits	251.4 µs	256 µs	252.8 µs	253.1 µs	10	100
9 qubits	1.8 ms	1.826 ms	1.805 ms	1.809 ms	10	100
11 qubits	12.45 ms	12.63 ms	12.51 ms	12.52 ms	10	100
13 qubits	96.68 ms	97.53 ms	97.02 ms	97.05 ms	10	100
15 qubits	651.8 ms	667 ms	659.6 ms	660.1 ms	10	100
17 qubits	4.513 s	4.541 s	4.52 s	4.523 s	10	100
19 qubits	27.73 s	27.86 s	27.76 s	27.77 s	10	100

Table D55: Benchmark of State Vector Simulator on Shor's algorithm scaled by number of n bits in N. The number of total qubits is $2n + 3$.

Syntax Simulator	Fastest	Slowest	Median	Mean	Samples	Iterations
5 qubits	18.03 ms	18.23 ms	18.08 ms	18.1 ms	10	100

Table D56: Benchmark of Syntax Simulator on Shor's algorithm scaled by number of n bits in N. The number of total qubits is $2n + 3$.