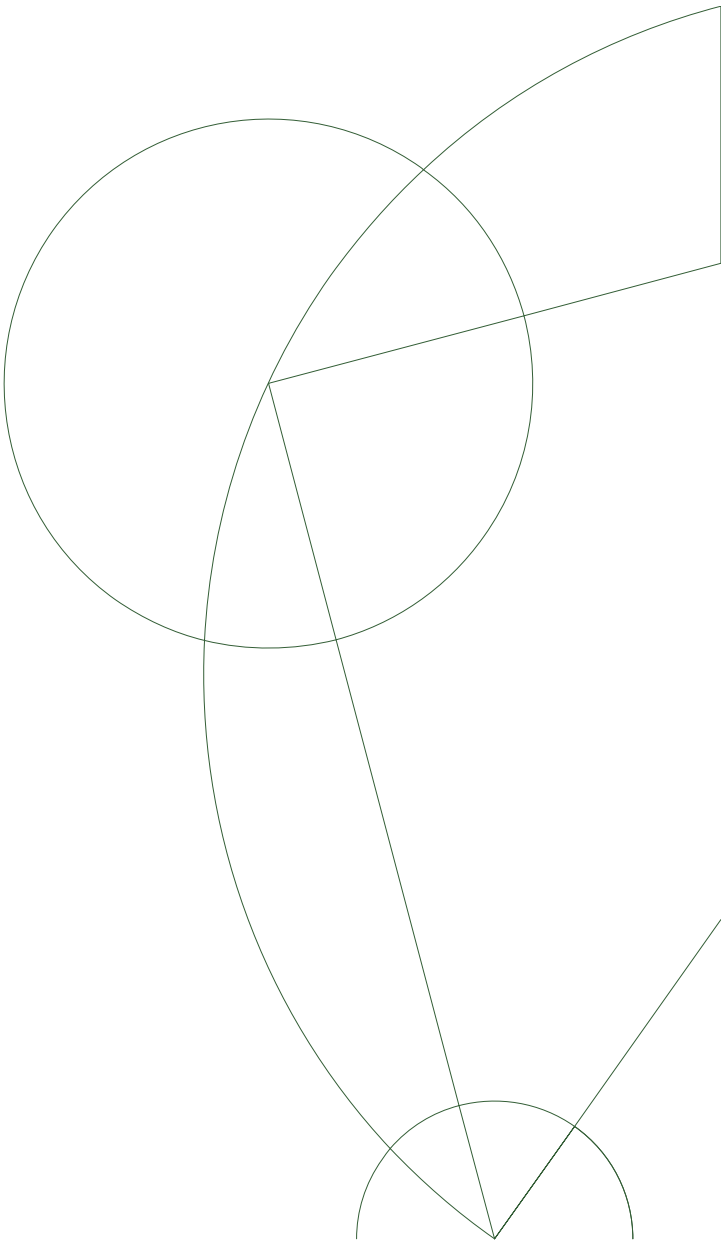
FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

BSc thesis

Petur Andrias Højgaard Jacobsen
Sebastian Vind Scheerer

Quantum Circuit Simulation

Language Design & Efficiency



Academic advisor: Holger Bock Axelsen
Michael Kirkedal Thomsen
Submitted: June 13, 2016

Quantum Circuit Simulation

Language Design & Efficiency

Petur Andrias Højgaard Jacobsen
Sebastian Vind Scheerer
DIKU, Department of Computer Science,
University of Copenhagen, Denmark

June 13, 2016

BSc thesis

Author:	Petur Andrias Højgaard Jacobsen Sebastian Vind Scheerer
Affiliation:	DIKU, Department of Computer Science, University of Copenhagen, Denmark
Title:	Quantum Circuit Simulation / Language Design & Efficiency
Academic advisor:	Holger Bock Axelsen Michael Kirkedal Thomsen
Submitted:	June 13, 2016

Abstract

In this BSc thesis, we investigate if it is possible to implement a quantum circuit simulator with better run time and memory efficiency than a naive solution. We create our own quantum circuit simulation language called Q-LIT. We define a language syntax and specification and implement a compiler for Q-LIT that compiles source files into C code. For this we implement two individual C libraries that perform the same set of quantum simulation operations, but which vary in run time efficiency and memory handling. When we compare these, we find that the improved library foregoes allocating memory for large tensor products where the simple library does. The simple library does not allow input sizes over 12 qubits on average systems because of memory starvation, while the improved library caps out on run time before this happens. The optimized solution also has significant run time improvements for certain classes of quantum circuit structures with a surplus of sparse matrices. We compare our language to the leading quantum circuit simulation language LIQUi|⟩, developed by Microsoft Research, and find that their state of the art simulator still leaves us with a lot of room for improvement.

Dansk Resumé

I dette bachelorprojekt undersøger vi om det er muligt at implementere en simulator for et sprog baseret på kvantekredsløbsmodellen, som har bedre hukommelses-/ tids-forbrug end en naiv løsning. Vi fremstiller vores eget kvantekredsløbsimuleringssprog kaldet Q-LIT. Vi definerer sprogets syntaks og specifikation, og implementerer en oversætter for Q-LIT, der oversætter kildefiler til C-kode. Vi implementerer to individuelle C-biblioteker der udfører det samme sæt af operationer, men som varierer i køretid og hukommelsesforbrug. Når vi sammenligner disse to modeller finder vi frem til at den optimerede løsning undgår at allokere hukommelse til store tensorprodukter, hvor den naive stadig allokerer. Den naive løsning tilladder ikke inddata af flere end 12 qubits på gennemsnitlige systemer på grund af hukommelsesmangel, hvor den optimerede løber løbsk på køretid før dette sker. Den optimerede løsning har også tydelige køretidsforbedringer for specifikke klasser af kvantekredsløbstrukturer med større mængder af sparse matricer. Vi sammenligner vores sprog med det ledende kvantekredsløbssimuleringssprog, LIQUi|⟩, udviklet og udgivet af Microsoft Research, og finder frem til af deres meget veludviklede simulator stadig efterlader os med en del at kunne forbedre.

Contents

Introduction

1

Processors and resistors are becoming smaller and smaller as time progresses in accordance with consumer and technical needs. The way these parts interact changes as the components get smaller, and crosses into the quantum realm of physics. This poses a problem to computing at large. Resistors the size of single atoms can no longer stop electrons passing to represent the bits we want. Electrons simply jump past the resistors, and strange properties to the deterministic world appear. These are the properties of quantum mechanics.

If we are to continue making processors smaller, we might as well exploit these properties, which are a nuisance to classical computing, for speed ups and parallelism. This is quantum computing: an entirely different paradigm in computation.

A quantum computational model supports exploitation of quantum mechanics in the context of computing. Using quantum properties yields dramatic speed-up on certain problems presented in classical computing that can only be solved inefficiently, even when considering the optimal solution in these classical systems.

A mathematical representation of quantum circuits has rules of computation that draw from linear algebra. The circuit consists of a number of wires holding equally many quantum bits and a number of quantum gates. A quantum bit, abbreviated qubit, is the information carrying unit in quantum computers. A quantum gate is represented by a unitary matrix and performs linear transformations. A parallel series of quantum gates represents a Kronecker product between them, producing block matrices, and a sequence of (parallel) gates are represented by multiplication between the resulting block matrices. When we scale up the system, n qubits are represented by a 2^n unit vector with complex entries, and a quantum gate, taking n qubits as input, has $2^n \times 2^n$ entries. The fact that these structures grow exponentially with the input size n is what makes them beneficial, in that they can perform computations linearly with a lot of compacted information.

Even if we do not have access to a quantum computer, it is still possible to simulate these computations on a classical computer using linear algebra and pseudo-randomized numbers. For this project, we want to define and implement a prototype for a simple quantum circuit language which performs such

a simulation.

The implementation consists of a compiler consistent with the quantum circuit model. We use Haskell to compile the programs into C code, making use of our custom made C library operations. There already exists a slew of quantum circuit languages, among them LIQUi|⟩ [2] (pronounced, and henceforth referred to as Liquid), that are well established and have an expressive front-end. Our project will focus on mechanisms which ensue in the back-end and investigate optimization techniques. We will thus not focus on building an equally expressive and helpful front-end.

Because of the linear algebra properties used in *simulating* quantum computing, we theorise in this thesis that is is possible to implement an optimized version of a quantum circuit simulation language, which has better runtime and/or memory consumption than a naive implementation.

Due to complexity analysis theory, any non-trivial optimization is not expected to be asymptotically faster than the naive implementation. Our main focus will be on conserving the use of memory by refraining from allocating matrices that are generated by the repeated use of the Kronecker product.

Theory

2

2.1 Qubits

All theory presented in this chapter is taken from [1] [5]. A qubit is the information carrying unit for quantum computers. It is analogous to the classical bit which can be in two states, 0 or 1. Generally, the state of a bit is simply the value it holds. A state can be represented in vector form:

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \qquad |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{2.1}$$

Here we introduce a traditional way of writing specific vectors in quantum mechanics called Dirac or bra-ket notation. A ket $|x\rangle$ is a column vector and a bra $\langle y|$ is a row vector. When writing a bit in such a vector form, it has exactly one entry with a 1 and zeros in the rest. The 1 tells us which of the possible states are represented by the vector. For n bits it takes 2^n entries to cover all possible combinations of those n bits. Visualised for 2 bits;

$$|a\rangle = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} \begin{matrix} \leftarrow & 00 \\ \leftarrow & 01 \\ \leftarrow & 10 \\ \leftarrow & 11 \end{matrix} \tag{2.2}$$

So $|a\rangle = |10\rangle$. This model is ample for classical computing. In a quantum system, the state space is mathematically modelled by a Hilbert space of wave functions. We will not discuss the concept of the Hilbert space or wave functions further, as they are not crucial to our conclusions. For more information we will refer the reader to the extensive work of *Quantum Computation and Quantum Information* [5]. In this system, the state of one qubit can be seen as a linear combination of two basis vectors, for example $|0\rangle$ and $|1\rangle$ from the representations in equation (2.1), where the scalars are complex numbers. The state of a qubit is thus:

$$|\psi\rangle = \alpha \begin{pmatrix} 1 \\ 0 \end{pmatrix} + \beta \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{2.3}$$

This is normally written as $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ with $\alpha, \beta \in \mathbb{C}^2$. A single qubit with 2 non-zero weights is said to be in a linear superposition. These weights are called probability amplitudes.

It is required that $|\alpha|^2 + |\beta|^2 = 1$, making it form a unit vector in $\mathbb{C}^2$. Knowing the amplitudes is impossible because this would require a measurement which has a probabilistic behaviour. A measurement is performed in some predetermined orthogonal basis $\{v, v^\perp\}$. Relative to the measurement basis, measuring a qubit $|\psi\rangle = \alpha|v\rangle + \beta|v^\perp\rangle$ yields $|v\rangle$ with probability $|\alpha|^2$ and $|v^\perp\rangle$ with probability $|\beta|^2$, and the qubit is then in a measured state. The most common basis is $\{0, 1\}$.

Similarly to the state of one qubit, the state of two qubits is given as a linear combination of four basis vectors. In general, we have a system of n qubits which have a state space of 2^n dimensions, thus requiring a system with dimensions $\mathbb{C}^{2^n}$. Quantum states combine through the Kronecker product, normally denoted as $\otimes$, which will be thoroughly explained in the following section. The state space for two qubits each with basis $\{|0\rangle, |1\rangle\}$, has basis:

$$\begin{aligned} &\{|0\rangle \otimes |0\rangle, |0\rangle \otimes |1\rangle, |1\rangle \otimes |0\rangle, |1\rangle \otimes |1\rangle\} \quad \text{normally written} \\ &\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\} \end{aligned} \tag{2.4}$$

The additional theory is generalizable. A two qubit state can be expressed as $a_1|00\rangle + a_2|01\rangle + a_3|10\rangle + a_4|11\rangle$ where a_1, a_2, a_3 and a_4 are complex numbers such that $|a_1|^2 + |a_2|^2 + |a_3|^2 + |a_4|^2 = 1$. For an n qubit state we have:

$$\begin{aligned} |\psi\rangle &= a_1|0\ldots00\rangle + a_2|0\ldots01\rangle + \cdots + a_{2^n}|1\ldots11\rangle \\ &= \sum_i a_i V_i \end{aligned} \tag{2.5}$$

where $V_i \in V$ is the set of basis vectors. We have that

$$\begin{aligned} &|a_1|^2 + \cdots + |a_{2^{n-1}}|^2 = 1 \\ \Leftrightarrow &\sum_i^{2^n} |a_i|^2 = 1 \end{aligned} \tag{2.6}$$

2.1.1 Kronecker product and bra-ket computations

We saw that kets denote column vectors and bras denote row vectors. Normal rules of linear algebra apply. $\langle x||y\rangle = \langle x|y\rangle$ is the inner product of the vectors, also known as the dot product. $|x\rangle\langle y|$ is the outer product or multiplication between the vectors. The linearity of vectors also follows, so we can write $|0\rangle\langle 1| + |0\rangle\langle 0|$ and it denotes matrix addition. Recall that matrix multiplication requires the components to have certain dimensions. The product of a $n \times m$ matrix and a $m \times p$ matrix is a $n \times p$ matrix. Therefore the product of two vectors where one is transposed creates a matrix with the dimensions of each vector:

$$u \cdot v = uv^T = \begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix} (v_1 v_2 v_3) = \begin{pmatrix} u_1v_1 & u_1v_2 & u_1v_3 \\ u_2v_1 & u_2v_2 & u_2v_3 \\ u_3v_1 & u_3v_2 & u_3v_3 \end{pmatrix} \tag{2.7}$$

The Kronecker product is a special case of the tensor product. If $\mathbf{A}$ is a $m \times n$ matrix and $\mathbf{B}$ is a $p \times q$ matrix, then the Kronecker product $\mathbf{A} \otimes \mathbf{B}$ is a $mp \times nq$ block matrix. A small example with a 2×2 matrix $\mathbf{A}$ looks like

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \otimes \mathbf{B} = \left[\begin{array}{c|c} a_{11}B & a_{12}B \\ \hline a_{21}B & a_{22}B \end{array}\right] \tag{2.8}$$

It can be seen that it creates a replicated block structure and the final matrix contains every $a_{ij}b_{kl}$. With the purpose of displaying the immense growth of these matrices, we can write out the calculation with an example of $\mathbf{B}$ being a 3×4 matrix, and $\mathbf{A}$ being a 2×2 matrix. We get the following result:

$$\mathbf{A} \otimes \mathbf{B} = \left[\begin{array}{cccc|cccc} a_{11}b_{11} & a_{11}b_{12} & a_{11}b_{13} & a_{11}b_{14} & a_{12}b_{11} & a_{12}b_{12} & a_{12}b_{13} & a_{12}b_{14} \\ a_{11}b_{21} & a_{11}b_{22} & a_{11}b_{23} & a_{11}b_{24} & a_{12}b_{21} & a_{12}b_{22} & a_{12}b_{23} & a_{12}b_{24} \\ a_{11}b_{31} & a_{11}b_{32} & a_{11}b_{33} & a_{11}b_{34} & a_{12}b_{31} & a_{12}b_{32} & a_{12}b_{33} & a_{12}b_{34} \\ \hline a_{21}b_{11} & a_{21}b_{12} & a_{21}b_{13} & a_{21}b_{14} & a_{22}b_{11} & a_{22}b_{12} & a_{22}b_{13} & a_{22}b_{14} \\ a_{21}b_{21} & a_{21}b_{22} & a_{21}b_{23} & a_{21}b_{24} & a_{22}b_{21} & a_{22}b_{22} & a_{22}b_{23} & a_{22}b_{24} \\ a_{21}b_{31} & a_{21}b_{32} & a_{21}b_{33} & a_{21}b_{34} & a_{22}b_{31} & a_{22}b_{32} & a_{22}b_{33} & a_{22}b_{34} \end{array}\right] \tag{2.9}$$

As can be seen, even with reasonably small matrices, the growth is staggering and quickly becomes almost unmanageable to write out.

This is what's meant by block properties in quantum gates. In a quantum gate matrix we have that any index is given by either 0 or a number/expression. During a Kronecker product, if we know that some of the matrix $\mathbf{A}$ is 0, we have that a large portion of the final matrix is guaranteed to be 0 in the end. This is regardless of the contents of the $\mathbf{B}$ matrix, since we will scale whatever was in the corresponding block of $\mathbf{B}$ with 0. For example, using the previous equation, if $a_{11} = 0$, then the entire upper-left block of the resulting Kronecker product would be a block of 0, as evident by the multiplications in the written out result.

It is also possible to predict the value of a coordinate within the Kronecker product result without actually having to compute the enormous matrix. As we have access to the block structure and the two source matrices, we can easily find which block the coordinate fits into, then find the specific element in $\mathbf{B}$ the coordinate belongs to and multiply it with the scalar from $\mathbf{A}$ for that specific block. This will come up again later as we try to optimize our approach.

2.2 Quantum gates

Qubits, if not measured, can be transformed, governed by the dynamics of the system. Because the basis vectors are orthonormal, transformations should take qubit states to other states in a way that preserves orthogonality. For a complex vector space, linear transformations which preserve orthogonality are unitary transformations.

For compliance, quantum gates therefore need to be represented by unitary matrices. A matrix U is unitary if its inverse is the conjugate transpose, written $U^\dagger$, that is, $UU^\dagger = I$, where I is the identity matrix. This means that

quantum gate operations must be reversible. Recall that the conjugate of a complex number has the same real part, but with a flipped sign in front of the imaginary part, so that $\overline{(a+bi)} = a - bi$, where overline denotes conjugation.

Quantum gates may operate on multiple bits. Since the matrix is unitary, it is square, and thus its dimensions are $\mathbb{C}^{2^n} \times \mathbb{C}^{2^n}$, where n is the number of qubits the gate acts on. Often we work in the dimensions $\mathbb{C}^2$ and $\mathbb{C}^4$. When a quantum gate acts on a subset of the quantum system, they induce a change in the entire system. For example, a gate $\mathbf{A}$ acting on 2 qubits i and j from a vector of $\mathbb{C}^{2^n}$ is the tensor product of $\mathbf{A}$ acting on i and j, while $n-2$ identity matrices act on the remaining qubits:

$$\mathbb{I}_1 \otimes \cdots \otimes \mathbb{I}_{i-1} \otimes \mathbf{A}_{ij} \otimes \mathbb{I}_i \otimes \cdots \otimes \mathbb{I}_{n-2} \begin{pmatrix} b_1 \\ \vdots \\ b_i \\ b_j \\ \vdots \\ b_{n-2} \end{pmatrix} \tag{2.10}$$

2.2.1 Controlled gates

Controlled gates are a special class of quantum gate acting on 2 or more qubits. They segregate the qubits they act on into a control group and a target group. All the control qubits need to be set for the operation on the target qubits to ensue. An example mapping which contains one control and one target bit with operation $\mathbf{U}$ is:

$$\begin{aligned} |00\rangle &\to |00\rangle \\ |01\rangle &\to |01\rangle \\ |10\rangle &\to |0\rangle\mathbf{U}|0\rangle \\ |11\rangle &\to |0\rangle\mathbf{U}|1\rangle \end{aligned} \tag{2.11}$$

where the first qubit acts as control while the second acts as the target. This notation is generalizable to an arbitrarily large system, where target and control bits are arbitrarily far apart. In the following formula, the control and target bits have been underlined to show their presence.

$$\begin{aligned} |\underline{0}\ldots 00\rangle &\to |0\ldots 0\underline{0}\rangle \\ &\vdots \\ |\underline{0}\ldots 11\rangle &\to |0\ldots 1\underline{1}\rangle \\ &\vdots \\ |\underline{1}\ldots 00\rangle &\to |0\ldots 0\rangle\mathbf{U}|\underline{0}\rangle \\ &\vdots \\ |\underline{1}\ldots 11\rangle &\to |0\ldots 1\rangle\mathbf{U}|\underline{1}\rangle \end{aligned} \tag{2.12}$$

This equation has a coherent and simple representation when identity matrices act on qubits in between the control and target bits. Things get slightly more complicated with alternative gates occupying these slots, however we will not be expanding upon this during the report.

2.2.2 Examples of gates

The simplest gate that can enact an operation is a gate that operates on one qubit and negates it. The quantum equivalent of a classical NOT gate is called a Pauli-X gate and is denoted:

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{2.13}$$

It maps $|0\rangle$ to $|1\rangle$ and $|1\rangle$ to $|0\rangle$. Again, using Dirac-notation allows us to describe it concisely:

$$X = |0\rangle\langle 1| + |1\rangle\langle 0| \tag{2.14}$$

This gate can also enact on a superposition. As we described earlier, a qubit can be written as a vector of the complex numbers acting as our probability amplitudes, meaning the Pauli-X gate simply switches amplitudes, effectively flipping our unknown superposition.

One of the most useful quantum gates is the Hadamard gate. It operates on one qubit, mapping the classical basis states $|0\rangle$ and $|1\rangle$ to superpositions with equal amplitudes for both states. It is denoted as:

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{2.15}$$

This gate, when used in parallel across multiple qubits, is also known as the Walsh-Hadamard transform, as it can create a superposition on all possible combinations of qubits in a n qubits large system. It is denoted by $H^{\otimes n}$, where n is the number of parallel gates, and the sum can be written as such

$$\frac{1}{\sqrt{2^n}} \sum_x |x\rangle \tag{2.16}$$

This is incredibly useful because it initializes a n bit classical system to a superposition of all 2^n orthogonal states in the $\{|0\rangle, |1\rangle\}$ basis. A 2 qubits example of $|00\rangle$ can be written out as such

$$\left(\frac{|0\rangle + |1\rangle}{\sqrt{2}}\right) \otimes \left(\frac{|0\rangle + |1\rangle}{\sqrt{2}}\right) = \frac{|00\rangle + |01\rangle + |10\rangle + |11\rangle}{2} \tag{2.17}$$

The CNOT gate, short for controlled NOT gate, acts on two qubits. It performs the NOT operation on the second qubit only if the first qubit is $|1\rangle$. It is denoted as:

$$CNOT = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{2.18}$$

The Toffoli gate, also called the CCNOT gate is a quantum gate which operates on 3 qubits. If the first two qubits are in state $|1\rangle$, it applies a Pauli-X operation on the third qubit. It is denoted as:

$$T = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \tag{2.19}$$

These matrices quickly become insurmountable to write out. It is preferable to write it in Dirac notation:

$$T = |0\rangle\langle 0| \otimes \mathbb{I} \otimes \mathbb{I} + |1\rangle\langle 1| \otimes C_{not} \tag{2.20}$$

2.3 Measurement

In quantum computing, as mentioned before, a qubit can be in a superposition between 0 and 1, making it both 0 and 1 at the same time. It is necessary to be able to collapse this nebulous state to a certain 0 or 1. This is done with a measurement gate. A measurement is not a quantum operation like others. It is an operation performed on the qubit rather than a calculation expressible by a unitary matrix[1]. In essence, the operation collapses a qubit to a basis state, but it is a bit more complex than this, as we can have a superposition based on multiple qubits. We can choose to measure a single qubit in this superposition and collapse the collection of outcomes containing the bit of that significance as either 0 or 1, but not both. Like in this example:

$$|\psi\rangle = a_1|00\rangle + a_2|01\rangle + a_3|10\rangle + a_4|11\rangle \tag{2.21}$$

We can collapse around a single qubit in this 2 qubit superposition, so that if we measured on the qubit of least significance and got the result 1, we would have the following collapsed state

[1] There are more general models for dense matrices and superoperators, but we will not get into that in this project

$$|\psi\rangle_2 = b_1|01\rangle + b_2|11\rangle \tag{2.22}$$

It is clear that the probability amplitudes have changed from a to b. This is due to normalization. We must at all times make sure that a superposition abides the rule

$$\sum_{i}^{2^n} |a_i|^2 = 1 \tag{2.23}$$

meaning the sum of all amplitudes' total value squared equals 1. If we simply take away two basis states in a superposition that has been measured, we would end up with a sum of probability amplitudes that we cannot foresee and which cannot be counted on to uphold the rule. In order to re-normalize, the following equation can be used:

$$|\psi\rangle_2 = \frac{a_1|01\rangle + a_2|11\rangle}{\sqrt{|a_1|01\rangle|^2 + |a_2|11\rangle|^2}} \tag{2.24}$$

which results in the probability amplitudes of b_1 and b_2 which again abide by the above rule. This is incidentally also how bell states or entangled states can be produced. By measuring one bit while knowing what we have a superposition of, we can immediately know what the second bit will be.

2.4 Quantum circuit

A quantum circuit is an array of quantum gates placed in a sequence, appropriately called a gate array. It is normally written in diagrammatic form. Quantum algorithms are usually directly constructed by a quantum circuit.

A quantum circuit is a combination of gates that together transforms a set of qubits. It has a fixed number of wires which denote the number of qubits the circuit operates on. The gates combine through the matrix product and the Kronecker product. Let $\mathbf{A}$ be an n-bit gate and $\mathbf{B}$ an m-bit gate. Then $\mathbf{A}$ and $\mathbf{B}$ combine through:

- A matrix product if they are sequential. This requires $n = m$.
- A Kronecker product if they are parallel and we say it is a $m + n$-bit circuit.

An example of a quantum circuit that performs a quantum algorithm is illustrated in Figure 2.1. It implements Deutsch's algorithm [12] to determine if a function is constant or balanced. It can be generalized but this example employs 2 qubits.

The curcuit computes

$$|\psi\rangle = (H \otimes \mathbb{I}) \cdot U_f \cdot (H \otimes H) \cdot |01\rangle \tag{2.25}$$

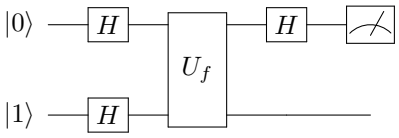


Figure 2.1: Circuit for Deutsch's algorithm

Figure 2.1 shows the diagrammatic form of Deutsch's algorithm. Each wire is a horizontal line with one qubit attached at the left-hand side. For each box, a gate operation at that point in time is executed on those wires' qubits, affecting only them. If two boxes are parallel, it means that they act on their respective sets of quits at the same time. Boxes are labelled with brief identifiers, usually one capital letter, that establishes their operation. Some gates like the CNOT gate require an exact specification of what each qubit signifies for the operation. In general, when a gate employs a different meaning behind individual bits, they have respective symbols for each qubit role, and an additional horizontal line is drawn between the wires to show the connectedness.

Note that the equation belonging to this circuit is written in reverse order with regard to the circuit model. This is to portray that the state vector is multiplied with the final result of the quantum circuit.

Language and translation

3

We have defined a programming language to generate and run quantum algorithms that comply to the theory set forth. The language is domain specific and supports a very targeted, minimal set of operations, merely acting as a blueprint to build and run circuits. In this section, we will present all language features, form a discussion on key aspects and considerations regarding our design choices and walk through the implementation behind the compiler.

3.1 Language introduction

The language supports three unique operations

1. Definition of qubits, circuits and gates.
2. Attachment of qubits to ciruits
3. Insertions of gates onto a circuit

3.1.1 Custom gate definitions

All definitions start with the keyword `def`. What type of definition that is declared is revealed with any of the keywords `qbit`, `qgate` or `qcirc` for respectively a vector of qubits, a quantum gate or a quantum circuit. Quantum gates take a matrix expression as an argument. Matrix expressions are recursive arithmetic expression that support addition, subtraction, multiplication, powers and tensor products. The only valid exponent is a scalar. All statements are terminated with a semi-colon.

The basic components in matrix expressions are scalars, bras, kets and manually created matrices. The syntax for manually created matrices is a list with commas denoting column shift and semi-colons denoting row shifts.

```
def myGate = qgate([1,2;3,4]);
```

It is also possible to use previous matrix definitions in new matrix compositions. It is required to define matrices before use.

```
def myCompGate = qgate(|0> * <0| # identity + |1> * <1| # not);
```

The final type of any gate definition must be a square matrix, which also means that there is no support for binding of scalars to variables. Care must

be taken from the user's side to define matrices that are unitary since the language will not expect errors of this type.

Several quantum gates are shipped with the language by default. Their names might thus not be used to define gates. These are `not`, `hadamard`, `cnot`, `identity` and `measurement`.

3.1.2 Qubit vector definitions

Qubits may solely be defined with bra or ket notation. These two are equivalent with regard to qubit vector definition, but the flexibility makes it more intuitive, and the difference between them will be taken into account when subjected to arithmetic.

```
def myQubit = qubit(|000>);
```

3.1.3 Circuit definitions

Circuit definitions take two integers that represent a slot/wire span. In this context, slots represent succession of gates operating on a qubit vector ranging from slot 0 to $n-1$. At conception, a circuit is void of inserted gates and has no qubit vector attached.

```
def myCircuit = circuit(n,m);
```

It is legal to define an arbitrary amount of circuits. If a program is run with more than one circuit, each result qubit vector will be written to a separate file tagged with the circuit name.

3.1.4 Attachment of qubits to circuits

Attachment of qubit vectors happens by dedicated syntax. The circuit that the qubit vector is attached to is selected with typical object-oriented dot-syntax. The qubit identifier then occurs as a perimeter in the argument list.

```
myCircuit.attach(myQubit);
```

Obviously, both identifiers must be bound. Additionally, we restrict users to only allow one binding of a qubit vector to a circuit to minimize unnecessary commands. It is also required that the qubit vector perfectly maps onto the circuit wires. If the compiler finds a circuit that has no qubit attached at the end of parsing, this is also seen as an error.

3.1.5 Insertion of gates on circuits

Because of the choice to define circuits with a set number of slots as well as wires, it is necessary for the user to specify at what point in time a gate should act on a qubit vector. Insertion uses the same dot-syntax as attachment and takes three arguments: slot number, a list of wires the gate acts on and the gate matrix.

```
myCircuit.add(1, [0], not);
myCircuit.add(1, [1,2,3], toffoli);
```

Certain restrictions are enforced. Both identifiers must be bound. Notice that even when a gate only acts on one qubit, the list notation is mandatory. Additionally, three things are checked for: firstly, any entry which a user attempts to place a gate on must previously have been unoccupied. Secondly, it is not allowed to place a gate out of the bounds of the defined circuit, and thirdly, all wire indices must be adjacent.

A special syntax is allowed for control gates. Because of time restrictions and the less popular demand for control gates with more than one control bit, a control gate is restricted to one control bit and a one bit operation. Also, the control bit must occur before the target bit on the circuit. The syntax for control gates is a two-element list with a semi-colon separator.

```
myCircuit.add(1, [0;2], not);
```

3.1.6 Measurements

Measurements follow the syntax for adding arbitrary gates to circuits, but they are in all ways special. If a slot contains a measurement, no other operations are performed.

```
myCircuit.add(1, [0], measurement);
```

Beware that if these considerations are not taken into account, a silent change is made in the circuit where the first measurement found is the only operation performed in that timeslot.

3.2 Deutsch's algorithm

With all language features thoroughly explained, we demonstrate a classical quantum algorithm written in Q-LIT. The code for Deutsch's algorithm could be

```
-- this is a comment
-- manual markup of the constant version of uf
def uf_constant = qgate([1,0,0,0;0,0,0,1;0,1,0,0;0,0,1,0]);

-- define the qubit
def q = new qbit(|01>);

--define the circuit
def c = new qcirc(4,2);

-- attach q to c
c.attach(q);

-- group 1
c.add(0, [0], hadamard);
c.add(0, [1], hadamard);

-- group 2
c.add(1, [0,1], uf_constant1);

-- group 3
c.add(2, [0], hadamard);
c.add(3, [0], measurement);
```

A rendering of the diagram for Deutsch's algorithm is shown in figure 3.1. The groups are noted in the mockup and labeled on the figure.

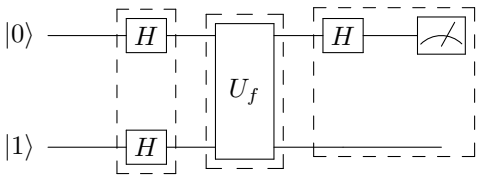


Figure 3.1: Circuit for Deutsch's algorithm

3.3 Usage manual

Here we will show how to use the interface that allows one to compile programs in our language. The compiler is shipped in a folder with the hierarchy presented in figure 3.2.

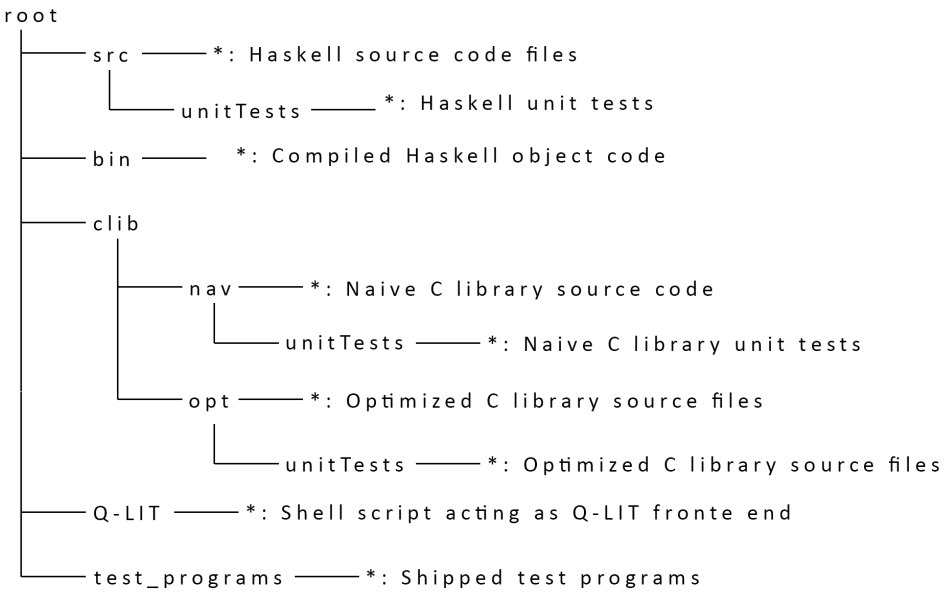


Figure 3.2: Directory tree for Q-LIT program.

The compiler is fully available through the `Q-LIT` shell script. It is designed to carry the user from a source file in Q-LIT to a binary executable. This means that all steps of the compilation process are performed automatically. The user should simply supply the script with a file path as the last terminal argument. If no arguments are supplied, a helpful message is printed conveying the possibilities of usage.

Besides the source file path, the script responds to four individual flags:

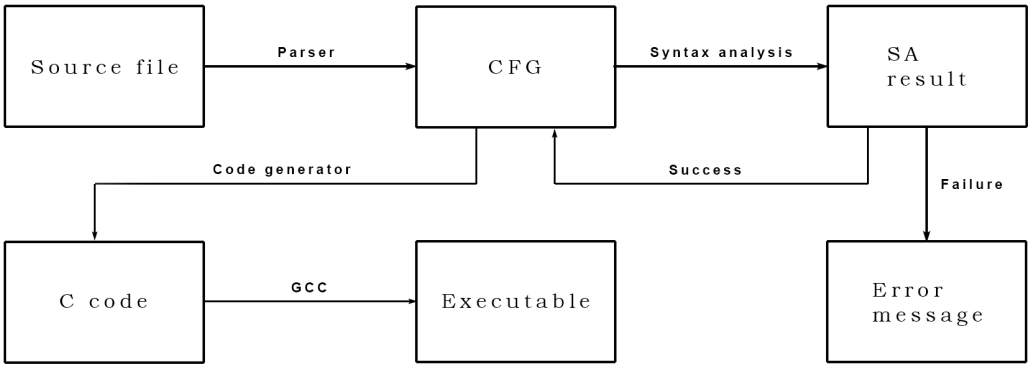


Figure 3.3: Flow graph for compilation of a file.

1. `-n` or `-o`: It is obligatory to supply either flag. `-n` tells the compiler to generate naive C code while `-o` tells the compiler to generate optimized C code using the GCC C-compiler [10].

2. `-s`: Keeps a copy of the C source code that the compiler generates in the folder where the program was compiled from.

3. `-r`: Prints final run-time and ongoing run-time information with regard to every major C function call. Functions that are categorised as such are discussed in chapter 4.

Running the outputted executable generates yet another file called **qubitoutput_{q program name}** where the final result of the circuit operation is stored. If there was an error compiling the Q-LIt source code, the compiler will not attempt to generate C code but will instead print an error message related to the incident. Because of the extensive span of errors caught by our compiler, there should never appear a GCC compilation error.

To compile the Q-LIT compiler written in Haskell, some dependencies are needed. From a fresh installation of Ubuntu 15.02, the dependencies are `ghc`, `cabal-install` and `gcc`. Specifically for Haskell, the dependencies can be installed through the `cabal install` command and consist of `split`, `parsec`, `parsec-numbers`, `mtl`. With these programs and libraries installed, it should be possible to compile the Q-LIT compiler and in turn compile Q-LIT programs. On another note, this report and the full compiler handed in along with it, can be found on the public repository "https://github.com/SausageOfBirth/Q-LIT".

3.4 Design decisions

Language specification

There was a number of considerations that we had to take a stance on regarding the specification of the language.

The first is about settling on the syntax for the circuit structure. An alternative way of representing a circuit could be for the user to create a mathematical expression that conveys the circuit operation, much like in the theory section. Another possibility would be to only specify the set of wires a gate should act on and then let the compiler figure out how the gates stack on top of each other. Instead, the choice to explicitly number and index slots on the wire was to make it easier to gain an overview of a circuit when writing a program. The model also becomes simpler to reason about with regard to compilation - there is no need to create rules concerning how the circuit is to be structured from the order of defined gates.

The second is the allowance of multiple circuit definitions in a single program. Ultimately, support for multiple circuits in one program is limited, as this functionality did not take priority. Incidentally, it is still possible to define them, and they will all be accounted for. We chose to simply write each circuit output to a individual file tagged with the circuit name. The behaviour is relatively ambiguous and the user is advised against it. We still chose to include it during this first incarnation of the language because it might have some niche use, and the role it plays might now may easily be explored further. Another reason is that the language syntax would be counterintuitive if no more than one qubit and circuit could be defined. Why should they have the same syntax as definition of quantum gates if they do not follow the same conventions?

The third is the extent to which we support flexible gate constructions. Most notable is the lack of extensive support for control gates, arbitrarily placed swap gates and less tightly regulated measurements. The control gate is available but in limited form - it is possible to specify a control bit and a one qubit gate, however the control bit wire number (numerated from top to bottom) must be less than that of the gate. These limitations mostly exist because of time restrictions and the emphasis of the project which is program efficiency, not language extensibility.

Standard gates

We believe it would be constructive to define a number of standard gates that ship with any program. This aids the idea that the language could be expanded upon to include library support, and guides the user to focus on an important set of gates. It also bars the user from being required to define the age old gates that are massively frequent.

Extent of errors

We subscribed to the idea that any error handling should be performed on the Haskell side. We still implemented safe guards in the C library to the extent that no illegal mathematical operations can be performed if the functions were to be used in a stand alone context. This is to make the C code robust. The reasoning behind the choice to handle errors in Haskell is the elaborate representation we can gather from the program text when it is parsed. This is coupled with the philosophy that all operations that do not comply with the

quantum circuit model should be prohibited from ever being manifested to C code, since our language is highly domain specific. This turned out to halt us in certain ways - for instance, we were not able to check if a matrix was unitary because this would require a mathematical check. Therefore, this line of reasoning should be abandoned in the long term, or the compiler should be able to do basic mathematics.

3.5 Formal syntax

BNF is short for Backus-Naur form [11]. It is a way of writing context-free grammars to define our syntax. The idea is that we write a set of derivation rules that specify how statements may look like, which conform to exact specifications of languages, defining a language syntax. The specification consists of independent rules with one identifier on the left-hand side and an expression consisting of one or more courses of action on the right-hand side. Each vertical bar | signifies a possible course. The identifier is called a nonterminal and is written inside of < >-brackets. It states that any time we encounter exactly this tag in any expression, we should substitute the tag for an expression for this rule. Symbols that are not enclosed in tags are called terminals and are left alone. We use `::=` to bind expressions to nonterminals.

This particular BNF is relatively informal. It uses `;` to split statements, `\d` to denote a number of any length and `\w` to denote a word. Single quotes denote fixed strings. When vertical bars are inside a tuple, it means that a local choice between any of the symbols may be substituted. Parenthesis that are themselves terminals have been "spaced out" to indicate that they are a symbol in the expression.

```
  <mat-exp> ::= [ (\d+;)+ ] | \d | |\d> | <\d| |
                <mat-exp> (*|#|+|-) <mat-exp> | <mat-exp> ^ \d
<def-state> ::= 'def' <id> = <elem> ;
     <elem> ::= <q-con> ( <mat-exp> ) | 'circuit' ( \d , \d )
    <q-con> ::= 'qubit' | 'qgate'
       <id> ::= \w
 <ins-gate> ::= <id> . ( \d , [ (\d,|;)+ ] , <id> ) ;
```

3.6 Language compilation

3.6.1 Program input analysis

The entry point to the program performs each step of the compilation process sequentially. There are three steps to perform: parsing the program input, syntax analysis and code generation.

The program text is parsed using a parser combinatoric approach, pulling in the Parsec Haskell library. The parser emits a data structure built with algebraic data types that reflect the language. Syntax errors are reported with the built-in error handling system of Parsec, which simply presents an error message wrapped in a custom error type. Only the message is relevant since

we want to outright show it to the end user for him/her to act on.

Testing for program correctness is handled by a separate module. Program correctness is defined as statements that violate the intrinsic rules presented in the previous chapter. The module works by analysing the program data structure one statement at the time, storing information that is important to further analysis for each statement. There are technically two passes during syntax analysis:

- The first pass checks for upfront errors. These are errors that can be expressed at any time using the information gathered up until this point in analysis.
- The second pass checks for all violations concerning superfluous definitions of circuits and qubit vectors. These could not possibly have been caught earlier since it is per language specifications possible to postpone attachment of qubit vectors.

3.6.2 Code Generation

In the code generation section, we explain how the generation of code works from analysis of the raw inputs, to exporting strings of finished generated C code. There are two versions of the code generator that supply code for either version of the C library. We will not make a distinction between the two during this description, as they are almost identical in logic.

Qubits

Qubits should simply be defined with a C function call. The value it holds needs to submitted to the C code somehow - this is done with an accumulator that iterates over i to find 1's. Once a 1 is found it adds 2^i to the final qubit value. A distinction is made when a qubit is defined and when it is attached. When it is attached, A pointer assignment is made to a variable holding the circuit name, signifying that all operations on that qubit belong to that circuit.

Gates

Gates are represented with a Matrix expression datatype in Haskell which is recursively defined. The leaves contain basic mathematical objects like hand crafted matrices, bras, kets and scalars. They are connected by arithmetic operators such as multiplication, addition and so forth. These are generic operators of which the input types bear no impact in the representation. However, the C language is not formally capable of function overloads, which means unique functions have been drafted for every legal combination of input types. Since we cannot approach the tree bottom up where the types are clearly known, this creates a need for operators and return types for intermediary calculations at every step of the matrix definition, when generating the matrix definition in C. We therefore define a new Haskell datatype that encompasses precisely this information, and then pass each matrix expression definition through a

function that translates them into this type. Leaves are the basic mathematical objects while nodes are binary operations. Leaves are now tagged with a type and nodes use the types of leaves and descending nodes to deduce their own type, and to dispatch a C function call. The rest of the process then involves generating strings using the functions and data types deduced from the updated structure.

3.6.3 Circuits

A circuit is generated by first processing the circuit body since the user has no obligation to fill in the empty entrances. This analysis is comprised of finding the qubit vector that has been attached to the circuit, and making the circuit into a list of lists with gate names as elements. When no gate name is present where a slot is, identity gates are automatically filled in. We approach it by first calculating the matrix for an entire slot. We assign the first gate to the result value, and then recursively reassigning it to be the result of a tensor between itself and the following gate on the same slot. This makes sure we do not stand in the end with one too many or one too few gates.

After we have finished with the slot, we delegate that the qubits should be multiplied with the final matrix for that slot, giving us the current result value. The reason for this step-wise procedure is that a measurement cannot be multiplied onto a bit like a regular gate operation, or for that matter undergo a tensor product. If we at any point during the calculation of a slot matrix come upon a measurement, we disregard what has been generated up until now (since we only allow one measurement per slot, with no other gates present) and perform the measurement instead. In addition, if only one wire or gate is present, we simply multiply the qubit by the gate and continue.

C library

4

In this section, we will lay out the functionalities of every mathematically significant function in the C library and how datatypes work and are initiated.

We will present two approaches to calculating the result of a qubit vector acting on a quantum circuit. The first approach is designed as simple and naive, meaning that it automatically follows all conventions regarding the computation. The improved library employs a variety of optimization techniques we have drafted with the aim to experience a palpable decrease in runtime and memory consumption.

4.1 Simple library

4.1.1 Types

We defined two types to handle information, `Matrix` and `Qubit`. A matrix structure consists of the number of columns and rows and the two-dimensional array containing the matrix entrances. Qubit vectors contain considerably less information, as only a one dimensional array is necessary, as well as the length of the qubit. All entrances are complex doubles.

Initializing qubits and matrices have their own respective functions. To initialize a qubit we need the total number of qubits that comprise it and the assigned value in base 10. Since we explore all combinations of the binary values in a qubit, going up in index by n simply adds n to the value of the qubit representation in base 10. This enables us to create a new one-dimensional complex array and assign the complex equivalent of 1 on the index of the total base 10 value of the desired qubit. Since we need to fill our array with complex zeros, looping through the array is necessary. This gives us an asymptotic runtime of $O(2^n)$ where n is the number of qubits declared. Memory consumption is similarly equal to $2^n i$ where i is the size of a complex number.

Initializing a matrix creates a new matrix completely filled with zeros of the dimensions specified. It is possible to create a Matrix function where the data is fed directly into the function. That would however prove to be equally consuming in data and time since one would have to declare the two-dimensional complex number array outside calling the function in order to pass it as an argument. It is more straightforward to create a blank new matrix and change the values as necessary by fetching the matrix data component. The asymptotic

runtime for creating a new Matrix is $O(nm)$ since we have to run through the number of columns m for each row n to compensate for all entrances. Memory consumption is equivalent to nmi where i is the size of a complex number.

4.1.2 Matrix Arithmetic

In the arithmetic section, we at all times need to access at least some information of the types that we are passing. For matrices this means getting the dimensions and the data. This is often to see if the two matrices are compatible for the function being applied to them and then utilizing both these dimensions and the data to perform the functions. Some are more restrictive than others, but in these arithmetic sections we rely heavily on the mathematics of linear algebra. All of these functions are of course essential in quantum computing, but when referring to a qubit or a gate, we simply speak of a vector and a matrix respectively. The calculations of these adhere to linear algebra, not specifically to quantum mechanics or quantum computing.

Multiplication

For multiplication the number of columns in the left side matrix must match the number of rows in the right side matrix. It is first necessary to fetch the dimensions of both matrices and check if they are compatible for multiplication. No action is performed if they are incompatible.

The dimensions of the resulting matrix are taken from the number of rows in the left side matrix and the number of columns in the right side matrix. We allocate the space as usual when creating a new matrix, but since we will need to treat every single element in the result matrix during computation, it is unnecessary to instantiate values upfront.

We then iterate over each resulting entrance with a double for-loop. An additional third loop is necessary to compute the accumulation for that entrance in the resulting matrix. Since all these dimensions are equal, we can conclude that the run time thus is $O(n^3)$ where n is the common dimension. The use of space is equivalent to $(n_1m_2 + n_1m_1 + n_2m_2)i$ where i is the size of a complex number.

Plus and Minus

The methods for addition and subtraction are very similar in approach. To add or subtract two matrices, their dimensions must match exactly. If they do not match, no action is performed. Else we allocate a new matrix of the same dimensions without instantiating values upfront. A double for-loop is necessary to run through all row/column pairs. The run time is thus $O(nm)$ where n is the number of rows and m is the number of columns. The memory consumption is equivalent to nmi where i is the size of a complex number.

Scalar

A scalar multiplication is a complex number multiplied with all entrances in a matrix. Since we loop through all entrances, we have a run time equivalent to

the numbers of entrances in our input matrix, $O(nm)$, where n is the number of rows and m is the number of columns. There is no additional memory allocated since it is reasonable to change the input matrix directly.

Kronecker Product

The Kronecker product places no restraints on dimensions for either input matrix. The dimensions for the resulting matrix are the dimensions for the input matrices multiplied, $n_1 n_2 \times m_1 m_2$ where n denotes width and m denoted length. We firstly allocate space for a $n_1 \times m_1$ matrix. The entrances to these will all be matrices as well, so we need to allocate $n_2 \times m_2$ for each entry for our resulting matrix to have the dimensions prescribed.

The computation is then performed with a quadrupole for loop, running through the rows and columns respectively in the first input matrix to cover all entrances. For each entrance, we run through the rows and columns in the second input matrix. To advance to the correct entry in the result matrix, the following formula is used:

$$\begin{aligned} row &= row_2 + (row_{2total} \cdot row_1) \\ column &= column_2 + (column_{2total} \cdot column_1) \end{aligned} \tag{4.1}$$

The entrance value is set by multiplying the entrances in each input gate on each of their current place in the for loop iterators. The asymptotic run time is $O(n_1 m_1 n_2 m_2)$, however, we can assume it adheres to the rules of quantum gates, making n and m equal, and rewrite it as $O(n_1^2 n_2^2)$. We have asymptotically ignored the factor of 2, meaning the run time is in fact double, as we both need to allocate the full matrix, and calculate every entrance of this full matrix. The memory consumption is $n_1 m_1 n_2 m_2$ multiplied with a complex number.

4.1.3 Control gate generation

Generating a control gate takes a 2×2 matrix and outputs a matrix of the size $2^{1+c-t} \times 2^{1+c-t}$ where c is the control bit and t is the target bit. The algorithm that produces the full control gate matrix is straightforward if three assumptions are held: 1) there is only one control bit and it comes before the target bit in the circuit. 2) the operation to be performed if the control bit is set takes one bit. 3) there are only identity gates in between control bit and target bit. If these assumptions are correct, the matrix will take the form in equation (4.2).

The run time of computing this is $O(2^{(1+c-t)2})$ since we need to instantiate all entrances of the resulting matrix. Since we also allocate a new matrix of size $2^{1+c-t} \times 2^{1+c-t}$, the memory consumption is equally $2^{(1+c-t)2}$.

4.1.4 Qubit Arithmetic

Qubit arithmetic is, as with matrix arithmetic, largely based in linear algebra, with little to no special regard for quantum computing. We will therefore refrain from explaining the mathematics of these functions unless explicitly called for.

$$
\begin{array}{c}
\overbrace{}^{2^{1+c-t}/2} \qquad\qquad \overbrace{}^{2^{1+c-t}/2} \\
\left.\begin{array}{c} \\ 2^{1+c-t}/2 \\ \\ \\ \end{array}\right\{
\left.\begin{array}{c} \\ \\ \\ \\ \end{array}\right\}
\left(\begin{array}{cccc|cccc}
1 & 0 & 0 & 0 & & & & \\
0 & 1 & 0 & 0 & & & & \\
0 & 0 & 1 & 0 & & & 0 & \\
0 & 0 & 0 & 1 & & & & \\
\hline
 & & & & u_{11} & u_{12} & 0 & 0 \\
 & & & & u_{21} & u_{22} & 0 & 0 \\
 & & 0 & & 0 & 0 & u_{11} & u_{12} \\
 & & & & 0 & 0 & u_{12} & u_{22} \\
 & & & & & & & \ddots
\end{array}\right)
\end{array}
\tag{4.2}
$$

Plus, minus and multiplication scalar

All three of these are perfectly similar in approach. Input qubit vectors must be of the same dimensions. If they are not, no operation is performed. Otherwise, we run through the data and add, subtract or multiply respectively. We assign the new data and length to a new qubit and return it.

The run time and memory consumption of plus and minus are linear and equal to the length of the input, $O(n)$ where n is the number of indeces in our qubit(s) vector.

Multiply qubit vector by matrix

This function is one of the most used during execution of a circuit as it produces a qubit vector affected by the gate it is run through. Before we start calculating the result, we make sure the length of the qubit vector corresponds to the number of columns in the Matrix. Nothing is done if they are not equal. Every entry in the input qubit vector is then multiplied with each column entry in the input matrix and summed into the result qubit entry.

The run time of this function is $O(nm)$ where n can is the length of the qubits and m is the number of rows in the input matrix. This is easily seen by the fact that we run through the entire matrix, so the run time equals the number of entries the matrix has. The memory consumption of this function is n multiplied by the size of a complex number since we only allocate space for a new qubit vector of length n.

Inner Product

An inner product occurs when multiplying a bra qubit vector with a ket qubit vector. This results in a single entry qubit, so we choose to return a complex number instead of an 'illegal' qubit vector (as qubits are of length 2^n, and a qubit of 0 qubits makes no sense from a computing perspective). We first make sure the lengths match, after which we multiply the corresponding entries in our input vectors and accumulate the results for returning when we are finished. If the lengths do not match, we throw an error. This is due to C's formal

incapability of exception handling and the fact that no safe 'fail' value exists as logically any complex number would be a viable correct result. The run time of this function is $O(n)$ since it takes linear time to multiply pairs of input vector entrances. The memory consumption is a single complex number.

Outer Product

An outer product occurs when multiplying a ket qubit vector with a bra qubit vector. This results in a matrix of dimensions $n_1 \times n_2$, these being the lengths of the first qubit vector and the second qubit vector respectively. No check needs to be done regarding vector lengths as any lengths are applicable. We first allocate memory for our result matrix using the two qubit vector dimensions without instantiating values, as we will iterate over each entry in the matrix. We calculate the resulting Matrix entries by their corresponding row values in the first qubit input vector and column values of the second qubit input vector. We can then directly return the manipulated matrix. The run time is $O(n_1 n_2)$ as every matrix entry is the result of exactly one computation. The memory consumption is $n_1 n_2$ since we create and store a new matrix of the dimensions $n_1 \times n_2$.

4.1.5 Measurement

A measurement is the most quantum computing specific function in our C library. It takes a qubit vector and the bit significance of the desired measurement. First we create a new intermediary array of the same dimension as the input. The purpose of this is to get the actual measures of probability for the outcomes by taking the total value of the complex number squared in each entry.

Now we are ready to select our outcomes. Regardless of the significance chosen we can proceed in the same way; we create a random probabilistic floating point number between 0 and 1, and for each entry in our intermediary array of information of the superposition, we check if our random number is less than that. If not, we subtract that number from our random number and move on to the next index. If it is smaller, we have our pick.

Now we figure out if our pick was either a 0 or a 1, and if so, which bits should now be 0 in outcome probability, and which should be normalized for the new measured qubit. We do this by checking if

$$n \; mod \; 2^{i+1} > \frac{2^i}{2} \tag{4.3}$$

Where n is the index of our pick, and i is our bit significance from left to right. Since we have, when observing how the qubit is indexed with binary equivalence, sets of size 2^{i+1} before the bit of significance i changes from 0 to 1, or 1 to 0. We can check if it is either 0 or 1 through using modulo on our pick to bring it down to an index which is contained in the first set of either 0 or 1 and check against the size of the first set, which we just defined the

calculation of to see if it is indeed a 0 or 1.

After this, we can use a loop check if the current index is contained within the same set as our pick was, meaning checking if the index contains a 0 or a 1 of the significance we picked. Depending on this, we can choose to either equal 0 or normalize through dividing the entry with itself and multiply with

$$\frac{1}{\sqrt{2}}^{log_2(n)} \tag{4.4}$$

where n is the length of the qubit input. Through using logarithm base 2 we get back the original information of how many qubits we are dealing with and can simulate a hadamard gate of the same order. The entries of this hadamard gate would be the result of some Kronecker products, which multiply hadamard entries with themselves the number of times there are qubits, essentially creating a hadamard entry to the power of the number of qubits.

Lastly, we are ready to assign our new measured qubit and return the result. The run time is $O(n)$, asymptotically ignoring the constant tripling of the run time. This is the very worst case and not necessarily the mean run time, as when a probabilistic match has been found the initial for-loop that is searching for the match stops. We then begin assigning values. This loops through the entire qubit in addition to the calculation of the intermediary array, which also loops through the length of the qubit vector. The memory consumption for the function is $2n$ multiplied by the size of a complex number, as we need to allocate the intermediary array and the result array.

4.1.6 Drawbacks

The largest and most consistent issue is that a matrix is produced as a result of each Kronecker operation for every circuit slot. For each slot, this creates a list of exponentially growing matrices based on the number of qubits in the circuit. This creates the need of a two step process during every slot calculation - sequentially building up the resulting slot operation followed by a qubit/matrix multiplication, which might not be the most efficient approach.

Another drawback is the vast amounts of repetitive storage of information. The majority of quantum gates contain matrices with values which are either one or zero, of where a major portion often is comprised of zeros. Because zeros are serve no purpose during multiplication, they are unwanted. Any matrix which has a surplus of zero entrances is deemed sparse. In the naive implementation, the sparse nature of these gates is ignored and an enormous amount of useless, repeated information is stored. All matrix structures in the library share this weakness, but the amplification of matrix sizes during the Kronecker product cements the waste of matrix entries. Any chain of Kronecker products on an empty matrix entrance creates an exponential amount of further zeros. We know that arithmetic functions will not be be used in contexts outside of declarations of gates, which will often have manageable sizes. However, there is a clear purpose behind scaling circuits since different input sizes may be

desirable, making issues with the Kronecker product remain the critical issue.

A programmatic neglect ensures that no memory is freed during chained operation during a gate definition. We make no attempt to free temporary slot results either. Regrading freeing of gates that are no longer used, in many cases, the structure is such that we cannot predict whether a gate will be used again, and cannot foresee if it it safe to delete it from memory.

4.2 Improved library

In this chapter, we will explain our implemented optimization of the language using references to our theory chapter, and pose solutions to it. We will then update the language, illustrate the changes and how they will theoretically improve memory consumption and possibly run-time.

For the purpose of optimization, it is important to analyse the potential of what we already know about quantum circuit simulation:

- Gates are unitary matrices
- Gates are often sparse
- Gates and circuit calculations heavily rely on block structure
- Data and calculations grow exponentially for every qubit

It is clear that the greatest asset we have in optimization is the sparsity of matrices since the calculations hinge on the data, and the intensity of that hinges on how we store said data. The fact that gates are unitary and rely on block structure can aid us in designing our approach. The very foundation of our optimized approach is a more specific data structure which we will go through below.

4.2.1 Data structure

We chose to build our data structure on the fact that the vast majority of matrices we will process have non-zero entrances in only a minor portion of their entries. This means that a large part of the data we are storing in the naive version is repeated a great deal. In essence, we only want to store any value which is not zero.

In the improved matrix structure the data is stored in a single array. Doing this prevents us from being able to simply store the complex number values by themselves as it would be impossible to determine row and column number for any given value that is accessed. So in addition to the complex number, it is necessary to store information regarding the value's coordinate location. For this we made a `Tuple` structure which stores all 3 data points for each non-zero value in a matrix, making up a list that is the matrix data. The data is then saved in the overall structure `Matrix`, which also contains the column size, row size, and the number of non-zero elements that can be expected to be found in the data. The standard for a matrix containing only zeros is a matrix with

but one entry at (0,0), containing the value zero.

However, to reliably calculate with this new type, we need to ensure it is ordered in some specific way. For this purpose we have the rebalance functions. These functions are based on the quicksort algorithm and in our case tuned to sort around either rows or column. This means a row sorted matrix will have data arranged with ascending rows while a column sorted matrix will have data arranged with ascending columns. In either case the culled coordinate is internally sorted, meaning an example of a tuple list sorted by rows would be

```
Before: (1,2),(3,1),(2,3),(3,3),(2,1)
After:  (1,2),(2,1),(2,3),(3,1),(3,3)
```

where the first coordinate denotes rows and the second denotes columns. Since sorting plays an essential role in the correctness of our algorithms, it is important to establish a philosophy of sorts, ours being that the recepient operation is responsible for any input to be sorted, if a need be. This will also come up as a part of the running times our different operations have, as sometimes a sorting will be mandatory. However, due to the fact that the vast majority of our operations will return an already sorted result, with the exception of the tensor operation, we will represent the running time of quicksort for its average case: $O(n \ln n)$. We feel this gives the most realistic running time perspective, however, the exponential alternative, $O(n^2)$, will come forth later, as we examine and discuss our test data.

4.2.2 Operations

In this section we get into the various methods used for performing the arithmetic and quantum computing focused functions. It should be noted that many of the qubit operations are the same as the previous version with only slight changes for optimizing. This is because we decided against making a new datatype for handling qubit vectors, stemming from the fact that we only have one qubit vector in a circuit when not accounting for qubit vectors made in conjunction with arithmetically declared matrices. We also decided against it because of the liberal use of hadamard gates that we can expect. Bringing qubits into superpositions is somewhat in the essence of quantum computing, and doing so prevents the qubits from being sparse, so we very quickly expect a qubit vector with a vast majority of entrances having non-zero values.

Memory allocation

Memory allocation quickly became an obvious problem for the new data type. C dictates that we manually need to allocate the array before we start producing a result matrix, but we cannot foresee the exact amount of values we wish to shore before we have made said calculations. For example, if a plus function is run, we have access to how many values are appointed in each input matrix, but this does not tell us how large an overlap exists between the sets and hence which values are added together into one data point in the result matrix.

This cannot be done in an acceptable run time since we would effectively be running the function twice, initially counting the number of values we need

(a) Addition/subtraction (b) Multiplication.

Figure 4.1: Sets of values we wish to store after each operation.

to allocate memory for. Because of this we start every function with allocating the worst case size of the result. This is necessary since C offers no support for dynamically allocated arrays which would not end up making very expensive calls to allocate memory for each value storage in our functions. This worst case allocation is often times going to be too large, so by the end of the each function, we use our accumulated value counter, which would have counted up the grand total of all values in our result while we operate, to copy exactly the right amount of memory from our worst case array into our final result array using `memcpy`. After this we can free the memory of the worst case array.

Minus/Plus algorithms

In the plus and minus algorithms we first sort our input around rows. The concept behind plus and minus is to loop through our two matrices concurrently and add together the matching results. Because we have sorted inputs, we can see when we might be able to add or subtract a result in the future and when we have passed being able to add or subtract. If, for example, we have an element from the first matrix which has a higher x or y coordinate than the element in the current entrance in the second matrix, we know we will never be able to add or subtract the first matrix entry with another element from the second matrix. We know this because both matrices are sorted, and were we able to add or subtract with an element of the other matrix, we would have already done so. At the point where we encounter this scenario, we add to or subtract from zero instead and increment whatever coordinate is lagging behind. If matching coordinates are found, then the entrance has a non-zero value in both input matrices. In this case, we add or subtract the values. We need to check if the result of the addition or subtraction is 0 since we are not interested in representing zero values.

Once we run out of elements in one or both of the matrices, we need to add to or subtract from zero with the remaining values. For this we have two individual loops, one in case of each input matrix not being exhausted. We simply add or subtract until the remaining entries have been accounted for. Once we are finished, we build a matrix and return it, copying over the temporary worst case array. If no elements were found, we return a standard zero-matrix.

The worst case run-time complexity of this algorithm can be broken down to its logic. At the very worst, we would need to either add all elements in a matrix with all elements in another matrix where all entries are non-zero values. Alternatively, we would need to add together a negative image of one matrix with the other, meaning the one matrix has non-zero values where the other does not and vice versa. This also gives us a run-time of the total matrix size possible. All in all, we have a complexity of $O(nm + n_1 \; ln \; n_1 + n_2 \; ln \; n_2)$ where n is the row for either of the matrices and m is the column for either of the matrices. The latter two parts consisting of sorting both inputs, n_1 being the number of non-zero elements in the first input matrix, and n_2 being the number of non-zero elements in the second input matrix.

The memory usage is difficult to analyse for these functions since we only take up space for whatever values do not result in zero after an addition or subtraction, making it more or less impossible to generalize. However, the worst case memory allocation can reflect the worst case for our function, which is $n + m$, where n is the number of non-zero elements in the first matrix and m is the number of non-zero elements in the second. However, as with the run-time complexity, there is a roof on how much memory this can actually be, which is equal to the dimensions of either matrix multiplied to produce the maximum number of entrances that a matrix with those dimensions can have.

Multiplication algorithm

Multiplications somewhat follow the example set by plus and minus; running through to find matching coordinates of elements, however, multiplication requires a full traversal for each entry calculation. This is where the option to sort around columns comes in handy. To calculate an entry in the result matrix, we need to run through the row of that entry in the first matrix and the column of that entry in the second matrix, multiplying each entry in the two matrices and getting the sum of the multiplications.

This means we need to know where each row and column begins and ends, however, we cannot directly access this information without running through the inputs, but we *can* build this information up while we are calculating. We start by setting wall values and a bottom value for rows. The wall values are the y and x values we are currently calculating for the result matrix, so we can check if the current index we looped to is outside our current x or y value. The bottom value is for quickly returning to the beginning of the row we are currently calculating with. This value is necessary because for each entry in our resulting matrix's row, we loop through the same row in our first matrix.

When we loop through to find a single entry in our result, we have the same three cases as with plus/minus: *greater than*, *less than* and *equals to*. However, we only need to match the y value for the first matrix entry with the x value of the second instead of both. If we have *greater than* or *less than*, we simply proceed, incrementing the appropriate index of whichever entry between x or y that was smaller. We do not have to add anything to our result because a miss-match here means multiplying with a zero. Once we find a match, we multiply and add it onto our buffer value, holding what will eventually be the final entry result. We can move beyond a wall to finish an entry in two ways; the index can

```
PLUS(m1, m2)
 1  x = 0
 2  y = 0
 3  res = MATRIX
 4  while x < LEN(m1) and y < LEN(m2)
 5        if m1[x] → x > m2[x] → x or m1[y] → y > m2[y] → y
 6              APPEND(res, m2[y])
 7              y = y + 1
 8        elseif m1[x] → x < m2[x] → x or m1[y] → y < m2[y] → y
 9              APPEND(res, m1[x])
10              x = x + 1
11        else
12              if m1[x] + m2[y] = 0
13                    x = x + 1
14                    y = y + 1
15              else tuple = TUPLE(m1[x] → x, m1[x] → y, m1[x] + m2[y])
16                    APPEND(res, tuple)
17                    x = x + 1
18                    y = y + 1
19  while x < LEN(m1)
20        APPEND(res, m1[x])
21        x = x + 1
22  while y < LEN(m2)
23        APPEND(res, m2[y])
24        y = y + 1
25  return res
```

Figure 4.2: Pseudocode for the plus algorithm employed in the optimized library.

find a column value in the second matrix greater than our wall, or the index can find a row value in the first matrix greater than our wall. In either case, we know we are done with the entry, regardless of what is left in the matrix which has not yet hit the wall as we would be multiplying the remains with zero.

If we go beyond the column value in the second matrix, we check if our buffer value is zero. If it is not, we add the index to our results, using the wall values for x and y values. After this, we need to see if this was the last column in our row of results. If it is not, we revert the x value back to where we identified the bottom of the row to be and move y's wall up with one. If it is the last column in that row of our results, we completely reset y, both in index value and wall value. We set x's wall value up by one, looping forward in x until we hit that wall value and have the first element in the next row, setting bottom to this index. Now we are ready to proceed to the next entry.

If we go beyond the row value, the situation is very similar. If we are not at the last column, we reset x to the bottom value and increment the y wall.

However, if we only know x has reached its wall, we need to loop through y to reach the first element in the next column. If we are at our last column, we have no need to loop through x to find the new bottom since in this case we are already there, so we simply do as if had been the column value that had gone beyond the wall, except from that row loop to set the new bottom.

Once the x wall and y wall values respectively are equivalent to the number of rows in the first matrix and columns in the second, we know we have calculated the last entry and can return the matrix.

The run time complexity of this algorithm is $O(n_1 \cdot m_2(m_1 \cdot n_2) + a\ ln\ a + b\ ln\ b)$ where n_1 represents the number of rows in the first matrix, m_2 the number of columns in the second matrix, m_1 the number of columns in the first matrix and n_2 the number of rows in the second matrix. For each entry in the result matrix, we need to go through every entry in the column of the second matrix and every entry in the row of the first matrix. The latter two additions to the run-time signify the mandatory sorting for multiplication, where a is the number of non-zero elements in the first matrix input, and b is the number of non-zero elements of the second input matrix.

The worst case memory consumption is derived from outer product: the most memory consuming kind of multiplication, a column matrix multiplied by a row matrix. This results in a matrix with the number of non-zero elements in the column matrix multiplied with the number of non-zero elements in the row matrix.

Circuit calculation

This function yields by far the largest improvements compared to the naive library. In the naive library, to reach the end of a qubit having been through a slot in the circuit, all matrices are tensored together for that slot, and the result matrix is multiplied with our qubit vector. This creates an enormous matrix. In the improved rendition, no space is allocated. Instead, we defined a tailored function that takes a list of matrices occupying the slot we wish to calculate and the qubit vector we wish the calculation to be performed on.

When multiplying a gate operation with a qubit vector, every resulting entry in the qubit vector is the result of the sum of multiplying all qubit vector entries with their corresponding column number counterparts in the gate operation matrix. The qubit vector entry row is determined by which row in the matrix we perform this summation with. In order to forego instantiating the matrix the full slot operation would yield, we predictively calculate what would theoretically occupy certain entrances of the result matrix.

Because the tensor product creates a block matrix, it is possible to know from any current matrix how many indices are being covered in the theoretical result matrix by the number of later gates and their sizes. For this, we have the value `blocksize`. If, for example, we have two single qubit gates, we would have a result matrix of size 4×4. This would make a horizontal or vertical move in the first matrix a move of two in the result matrix.

```
MULTIPLICATION(m1, m2)
 1  x = y = xwall = ywall = xbottom = 0
 2  res = MATRIX
 3  buff = 0 + 0i
 4  while xwall ≤ m1[LEN(m1)] → x and ywall ≤ m2[LEN(m2)] → y
 5        while x < LEN(m1) and y < LEN(m2) and m1[x] → x == xwall and m2[y] → y == ywall
 6              if m1[x] → y > m2[y] → x
 7                    y = y + 1
 8              elseif m1[x] → y < m2[y] → x
 9                    x = x + 1
10              else buff = buff + m1[x] · m2[y]
11                    x = x + 1
12                    y = y + 1
13        if xwall < m1[x] → x
14              if buff ≠ 0 + 0i
15                    tuple = TUPLE(xwall, ywall, buff)
16                    APPEND(res, tuple)
17                    buff = 0 + 0i
18              if ywall == m2[LEN(m2)] → y
19                    ywall = 0
20                    y = 0
21                    xbottom = x
22                    xwall = xwall + 1
23              else
24                    ywall = ywall + 1
25                    if m2[y] → y ≠ y − wall and y − wall ≤ m2[LEN(m2)] → y
26                          while m2[y] → y < ywall
27                                y = y + 1
28                    x = x − bottom
29        elseif ywall < m2[y] → y
30              if buff ≠ 0 + 0i
31                    tuple = TUPLE(xwall, ywall, buff)
32                    APPEND(res, tuple)
33                    buff = 0 + 0i
34              if ywall == m2[LEN(m2)] → y
35                    ywall = 0
36                    y = 0
37                    xwall = xwall + 1
38                    if m1[x] → x ≠ xwall and xwall ≤ m1[LEN(m1)] → x
39                          while m1[x] → x < xwall
40                                x = x + 1
41                    xbottom = x
42              else
43                    ywall = ywall + 1
44                    x = xbottom
45  return res
```

Figure 4.3: Pseudocode for the multiplication algorithm employed in the optimized library.

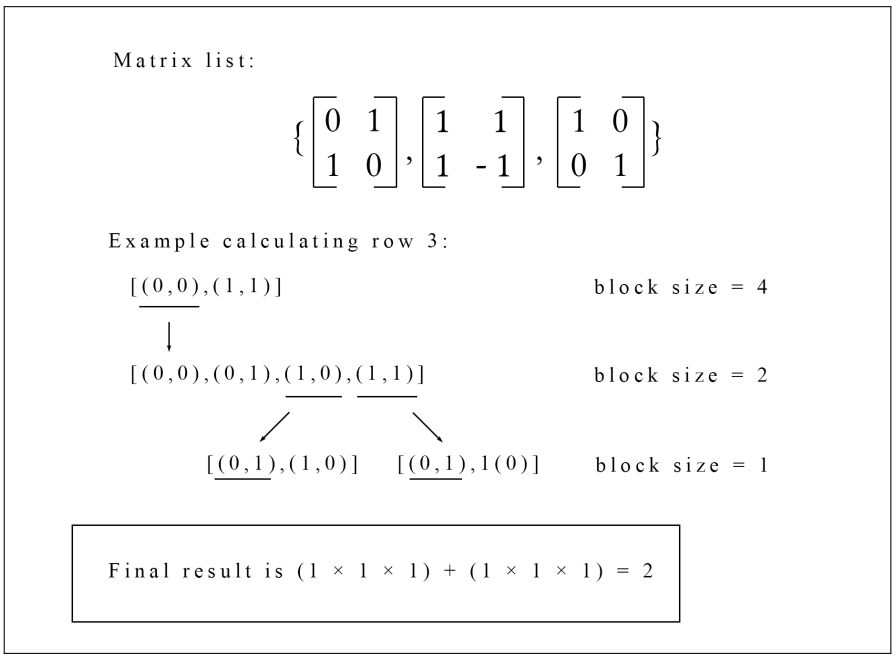


Figure 4.4: Example of row calculation algorithm with three matrices and calculating row 3.

The idea behind the procedure is to traverse the matrix list in search of the set of scalar value from the result matrix that make out the complete result of the qubit vector entrance. If `blocksize > 1`, there still are matrices for the qubit vector entrance to sum over. In this case, we run recursively further into the list of matrices with a revised `blocksize` and destination coordinates for each tuple in the current matrix that has prospective values relevant for the current row. If `blocksize = 1`, we have reached the deepest level and can return a value. Because we sum recursive calls together which individually are the product of all so far encountered values somewhere within that row, we end up with the full calculation for that qubit vector entrance. This recursive function is run within a wrapper that runs for each qubit vector entrance since it is only designed to calculate one row at the time.

Asymptotically, the worst case run-time complexity for this function is $O(2^{n+1}(2^n - 1))$ if every gate in the slot acts on one qubit. We have to calculate for every entry in the qubit vector which sets us off at 2^n entries. In the worst case, the slot matrix has all values instantiated. In this situation every recursive run will spawn a new run twice until we reach the last matrix. This means that across all recursive runs, the last matrix will have been found 2^n times. The second to last will have been found 2^{n-1} times and so forth. The formula then goes

$$\sum_{j=1}^{n} 2^j = 2(2^n - 1) \tag{4.5}$$

Coupled with 2^n, we have our bound. The memory consumption for this function is simple: there is none. We alter every entry in the qubit vector directly and save nothing which does not go on the stack.

```
CalcRow(q, m[], gn, qIndex, relQ, bSize, tRes)
 1  relativeIndex = qIndex / bSize
 2  buff = 0 + 0i
 3  for i = 0 to Len(m[gn])
 4       if m[gn] → x == relativeIndex
 5            if bSize == 1
 6                 buff = buff + (tRes · m[gn][i] · q[relQ + m[gn][i] → y]
 7            else nqi = qIndex % bize
 8                 nrelQ = relQ + (bSize · m[gn][i] → y)
 9                 nbSize = bSize / m[gn + 1] → col
10                 buff = CalcRow(q, m[], gn + 1, nqi, nrelQ,
                                  nbSize, tRes · m[gn][i])
11  return buff;
```

Figure 4.5: Pseudocode for the qubit entry calculation algorithm employed in the optimized library.

Kronecker product

This implementation of the Kronecker product is meant to be used exclusively for gate declaration via an arithmetic expression. The method by which we calculate the result is by the means of two loops. For each non-zero element in the first input we need a copy of all the second matrix elements multiplied with that first input. We then set the x and y value of the result element as we did with the naive Kronecker implementation.

The run-time complexity of this function is $O(n_1 n_2)$ where n_1 is the number of non-zero elements in the first matrix, and n_2 the number of non-zero elements in the second matrix. This is because we need to go through all elements of the second matrix the number of times there are elements in the first. There is no need to sort the input in this operation, so we have no quicksort runtime added onto the runtime. However, this output is almost guaranteed to be unsorted with a slight bias in being 'sorted' more along column lines. This will come up once more in the discussion of our data observasions.

The memory usage of this function can be predicted because we only scale, meaning we only ever multiply non-zero elements with other non-zero elements. This gives us absolutely no risk of having a result value which is zero. The memory usage is $n_1 \cdot n_2$ where n_1 is the number of non-zero elements in the first matrix and n_2 is the number of non-zero elements in the second matrix.

Qubit calculations

For the most part in our qubit calculations, the optimizations are very simple. Instead of allocating a new array for temporary values, we make the calculations as we go (measurement), and instead of saving copies of the arrays that make up the input qubits, we simply take the values directly from the input and use them. During a measurement we also avoid creating a new qubit result qubit. It is now a void function now which directly alters the input qubit. The only significantly altered qubit operation is multiplying a gate with a qubit vector.

Multiplying a qubit vector with a gate is very similar to the plus and minus procedures. It loops through the list of matrix elements, looking for matching indexes between the qubit vector's entrance and the matrix's y value. If we are *greater than* or *less than*, we increment the appropriate value and move to the next element. When we do find a match, we multiply and add it to our buffer, which will be our result for that entrance in the result once we have reached the end of the row in the matrix. When we do reach the end of a row, we apply our buffer to the result array and reset the iterator value. The run time complexity of this function is $O(nm + m\ ln\ m)$ where n is the length of the qubit and m is the number of non-zero rows in the matrix. The last addition being due to the need of a sorted matrix input. The memory usage is entirely predictable as we produce a qubit of the same length as the input, making the memory usage n where n is the length of the qubit.

The rest of the library functions have retained their approach with few changes to optimize memory usage.

4.2.3 Drawbacks

We allow ourselves a greater constant cost of each entrance in the matrix because we take advantage of sparseness. This is bad if all entrances are filled, since in addition to simply holding the complex number, each entry holds two additional entry information values. In certain functions, the run time can be said to have a worse constant cost as well, since we need to traverse the data structure in a far more complex way than we had to in the naive handling of data.

We allocate memory far more repetitively than in the naive implementation, as we need to allocate an instance of the `tuple` structure for every entry we calculate in a matrix and memory accessing is a relatively expensive procedure [6]. In addition to this, we allocate worst case space for every arithmetic function. This may with large enough matrices allocate too much memory for the heap to handle, resulting in a killed process that could have easily been performed had the right amount been allocated from the start.

Finally in this implementation, we have not yet addressed the exponential nature of the problems. This would be very complex and definitely not expected, as mentioned in the introduction. If we are to be stringent we can classify this as a drawback though.

Tests, Discussion & Comparison

5

In this chapter we implement simple functionality tests and programs in our language and run benchmarking programs with diagnostics on both versions of our compiler. We then hypothesize over what our results might be and discuss them afterwards. We will also compare the performance of our language against that of Liquid, a quantum circuit simulator developed by Microsoft. The comparison will be accomplished by running the same algorithms in both languages. We then discuss these results and attempt to explain the difference by means of the way the two simulators differ.

5.1 Functionality tests (unit tests)

A functionality test, or unit test, has the purpose of testing how robust a block of code is and to formally prove that one can count on a correct result under any valid conditions. This is achieved by setting up static tests with fringe, and often times in terms of quantum computing, unthinkable cases which assess the very margins of the procedure. This could for example be using multiplication to essentially make an outer or inner product as these are the extreme cases in terms of how matrix multiplication works.

5.1.1 C Library

For the functionality tests for our C libraries, it was necessary to write two different sets of tests as the algorithms and approaches differ. When these differ, so do their marginal cases which are what we desire to test with these.

Naive

For the naive unit tests, we have a set of tests for each significant function, not including printing functions or functions which initialize matrices or qubits, as they are elementary.

The first unit test is a test of Tensor, which we test successfully using the examples of a row by a column matrix, and the other way around, to create full square matrices. First we run it using the same values in every entry for either matrix, and then run them in reverse order of arguments with the same values. If it was run correctly, they should give the same result. This should

also be the case when the two inputs have distinct and unique entries, however, the values wouldn't be uniform. So an additional test was crafted for this.

The next test is multBitGate, which we test successfully using two different 'Qubits', one with a single entry, and one with three entries. This is to guarantee that both single entry qubits, and multiple entry qubits are calculated correctly.

We then test scalarG by using a single entry matrix, and a row matrix, which are there to test if it traverses and multiplies correctly in the entirety of an unorthodox matrix.

Again, outerProduct is tested using both a single entry qubit, and a three entry qubit, so ensure the outer product is calculated correctly with any number of entries.

Next is plusG tested, which we test by seeing if adding a matrix to itself gives a matrix of the same size, but with entries containing double the value of the original. The second tests attempts to add two matrices which are incompatible for addition, in which we expect a return value `NULL`.

Subtracting gates is tested much the same way as plusG, where we should very well end up with a matrix containing only 0's, and again, an incompatible test resulting in`NULL`.

For multG, we have three tests; all have slightly strange dimensions to keep them from being entirely square and check how robust it is, and the last of them to purposely create a `NULL` return.

For plusQ and minusQ, we have single and multiple entry tests added to themselves, and testing them against a scalar of 2 or 0 (for plusQ and minusQ respectively). The final of the three tests for both of plusQ and minusQ is purposely incompatible, to produce a `NULL` return.

ScalarQ is tested with a single entry, and multiple entry qubits, where a version of each exists both with all 0's, and unique number entries to test that they are scaled properly, as well as traversed properly by the function.

Optimized

First we test multG, we do this by first simulating outer product with a column matrix multiplied with a row matrix, then inner product using the inverse argument order, then a sparse matrix with spread out and few values, yielding a unique result with a single value in the 'centre', and an incompatible multiplication, resulting in `NULL`. This covers the marginal cases for the multiplication.

PlusG is tested by first adding a gate to itself, and see if it's equivalent to a scalar of 2. Then a pair of incompatible matrices to see if we get `NULL`, after which we test adding two different sparse matrices with spread out values to themselves, and see if it's equivalent to a scalar of 2. Lastly, we add a matrix

with a -1 scalar of itself, to see if it results in a completely empty matrix, which is denoted by only a single element of value 0, located at 0,0 in the Matrix.

MinusG is tested by matrices of same structure as in plusG, however, the first two tests are subtracting a scalar -1 of the same matrix, making it a double negative, which should result in a scalar 2 of the original matrix. Otherwise we have the same `NULL` test, and sparse matrices test.

ScalarG simply runs through all elements, so checking if traversal in the matrix is correct isn't necessary. However, we do test with both a scalar of 2 and 0, to see if it multiplies correctly.

We then test tensor using a row matrix on itself, a column matrix on itself, and a row matrix on a column matrix, as well as a single entry matrix on itself. This tests the overall traversal and multiplication at the same time.

We test multBitGate using actual qubits and actual quantum gates, first with two identity gates to see that our value is unchanged, then two not gates to see if our set input max value becomes the minimal value, and then an incompatible run to see that we correctly return `NULL`.

Then we check outerProduct using a singular entry qubit with a multiple entry qubit, and a singular entry qubit with itself, to see that the correct matrix is assembled in the end.

For inner product, we test with both a single entry qubit on itself, and a multiple entry qubit on itself, to see that we calculate the inner product correctly. However, since we cannot set a safe error value, as we do not return a pointer and can therefore not return `NULL`, and any concievable complex number is theoretically a valid return value, we're forced to throw an error on incompatible inputs. Since C doesn't have the try-catch properties, we are unable to test incompatible inputs as part of the unit tests, as it would stop the test program dead. However, we have extensively tested this outside the unit testing document.

In plusQ and minusQ, we do simple tests of single and multiple entry qubits against themselves, resulting in a scalar of 2 and 0 for plusQ and minusQ respectively, but due to their similarity to the naive version, theses tests are mostly a formality. Lastly, both functions have an incompatibility test which returns `NULL`.

ScalarQ is tested exactly the same way scalarG is, with the addition of having the same tests performed on both single and multiple entry qubits.

Then we have tests of rebalance_row and rebalance_col, which are both tested with unsorted[2] single and multiple entry matrices.

[2]Does not apply to a single entry matrix as these can by definition only be sorted

The last test is calcSlot, which we test with a 4 qubit slot. First test is 4 not gates run on 4 qubits at max value $|1111\rangle$, which should very well result in a qubit of $|0000\rangle$ value. Then with a not gate, a matrix consisting of two not gates tensored together, and a last not gate, which should have the same result despite the larger matrix now in the middle between the two single bit gates. The last test is a hadamard, as they are completely filled matrices with no sparsety, and should also work.

5.1.2 Haskell

Haskell has a reminiscent set of unit tests pieced together using HUnit. Testing the Haskell code base is a little more involved since the code is not very modular. During syntax analysis, an array of tests are each performed to test program correctness. These then feed analysis data to dispatch functions that subsequently update state and continue compilation.

A ground base of tests therefore test the atomary functions while a few tests make sure that the overarching functions are correct. As long as the atomary functions behave, the structure should be valid. Notably, functions that stress test the limits of the syntax were kept to a minimum. This is due to the overwhelming span of input that breaks parsing while the set of valid input is comparatively minimal.

We also chose not to perform unit tests for string output during code generation. The reason behind this is thrice-fold: one, GCC will complain during second phase compilation for most aspects of the generated code, creating a frustrating experience if some code generation was wrong, but still not allowing bad programs. Two, extensive compilation of various programs was performed during the course of work on this project which used all logic performable by our language for various reasons, certifying that the C programs that are generated are valid. And three, the complex logic surrounding code generation does not derive from the functions that output C strings but instead from dispatching functions, which are tested.

5.2 Performance tests

During tests of our performance, we wanted to test how our two versions of the program would spar against each other, in addition to testing it against Microsoft's Liquid quantum simulator. The first mentioned is what we dub *internal tests*.

The internal tests are split into two categories: tests exclusively using the hadamard gate and tests exclusively using the not gate. This separation is meant to test for the presence or absence of savings that we might experience from sparsity in the optimized library. The conjecture is that the naive version should compare equally or better when the same amount of elements need to be processed, while the optimized should surpass the naive for any test including sparse matrices. For each gate type, we have three scenarios that we test primarily: growth of gates on a single wire, growth of qubits in a single slot

and growth of both simultaneously.

We also perform a test for gate definition. In it we chain together tensor products on either side of a multiplication. This is meant to cover all types of binary operations that the simulator supports, since they have very similar asymptotic bounds. There is a test each for not gates and hadamard gates.

Our test contra Liquid is based on a procedure constructed as a native part of Liquid to benchmark their simulator's efficiency called *entanglement*. There are two variants: *entanglement1* and *entanglement2*. The first mentioned prints very detailed diagnostics data at the sacrifice of some efficiency, where as the second prints only the essential run time and memory usage data, making it faster. As the data we are interested in is simply the bottom line, total run-time and memory consumption, we elected to test against the faster of the two. This also better represents a fair comparison as Q-LIT has very limited diagnostics support. The entanglement test circuit itself consists of an arbitrary amount of bits, where the first basis bit is brought into a superposition by a hadamard gate initially. Then every following bit is run through a controlled not gate sequentially with the control bit always being the base bit in the superposition. A growth in input size for the circuit therefore includes a growth both in slots and in qubits. We compiled all the data into graphs using R [13]. The collection of all graphs can be found in appendix A-C.

All performance tests were performed on a machine with Ubuntu 15.02 installed, a 3.10 GHz writing speed, 4 logical processors, and 8 GB ram available. In addition, it was run on an SSD.

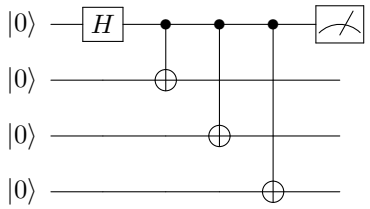


Figure 5.1: An example entanglement circuit consisting of 4 qubits.

5.2.1 Asymptotic tests (runtime)

We will present the results of our run times looking for asymptotic types, meaning we will examine the growth tendencies of both the naive and optimized implementations. After this, will also analyse how they compare to Liquid's complexity.

Internal tests

It is clear from observing the graphs related to growth of qubits that we possess an exponential growth of run-time for both the naive and optimized solutions on all types of input circuits and gate definitions. In some cases there is a large

difference between them regarding thr rate of this exponential growth, which will be further explained upon in the discussion. During circuit calculation, the optimized solution appears to be capable of performing tests with far larger input sizes than the naive - this is related to memory performance though, and we will discuss that in the next section. Unfortunately, this means we cannot fully compare the naive to the optimized in terms of run time. The exponential run time picks up pace after around 12 qubits, which is after the naive solution throttles on memory, meaning the naive is never properly stress tested.

The tests for growth in gates look slightly strange from the graphs. This is due to the fact that the changes in time per gate are extremely small but visually interpreted as a larger difference. The volatile curve can be due to many things, for example other processes syphoning the computational power. We cannot say for sure what causes these outburst, but they should not represent a complexity tendency other than a simply linear growth. This linear growth will be more evident when inspecting the corresponding graph for memory consumption, and we will get into that in the following section.

Liquid comparison

Against Liquid's comparison, both the naive and optimized implementations yield a complexity which is to be expected: exponential growth reflecting the tendencies in the internal benchmark tests. However, Liquid's growth is interesting - it seems to be plateauing. The first eight runs are almost, if not exactly, the same. After this, crossing over to 9, it seems to make an upwards leap to a higher order of circuit. This is also supported by memory consumption, but it would seem the growth is happening in tiers of size following an almost logarithmic shape. Certain thresholds cap the growth off, asymptotically making it far superior to ours in regard to run-time.

5.2.2 Memory behaviour tests

As with the runtime tests we have two parts to this comparison; memory consumption growth examinations regarding our optimized versus our naive, and comparison to the growth of liquid's test case. We used the tool Valgrind [14] the obtain the memory consumption data.

Internal tests

Memory behaviour is to some degree tightly coupled with the diagnostics of run time, however, this is where the naive version begins to diverge considerably. Systems containing an average amount of memory will usually become memory starved when dealing with input sizes beyond 12 qubits. This is mostly due to the enormous tensor results that are produced as intermediary objects in circuit calculation, which the optimized version does not produce at all. It should be brought to attention that there is a clear neglect of deallocation cropping up during code generation for naive programs in the name of simplicity, and it is fair to say that we have the responsibility to free the memory of tensor results after use in the naive version. However this would not cause any significant

improvement. Since continually larger matrices still need to be allocated and instantiated, it is a question of only relatively few extra qubits before it fails again due to the exponential growth of the matrices.

The optimized version excels here. We have yet to encounter a program execution which ran out of memory before the run time became insurmountable. Because the qubit vector data structure remains the same between the two solutions, the optimized library still has an asymptotically exponential memory growth and will thus doubtlessly run out of memory as well, but the difference is between 2^n and $2^n + (2^n \cdot 2^n)$ complex floating point numbers stored. Our experiences are that overwhelmingly long run times halt progress rather than the process being killed for starvation of heap memory.

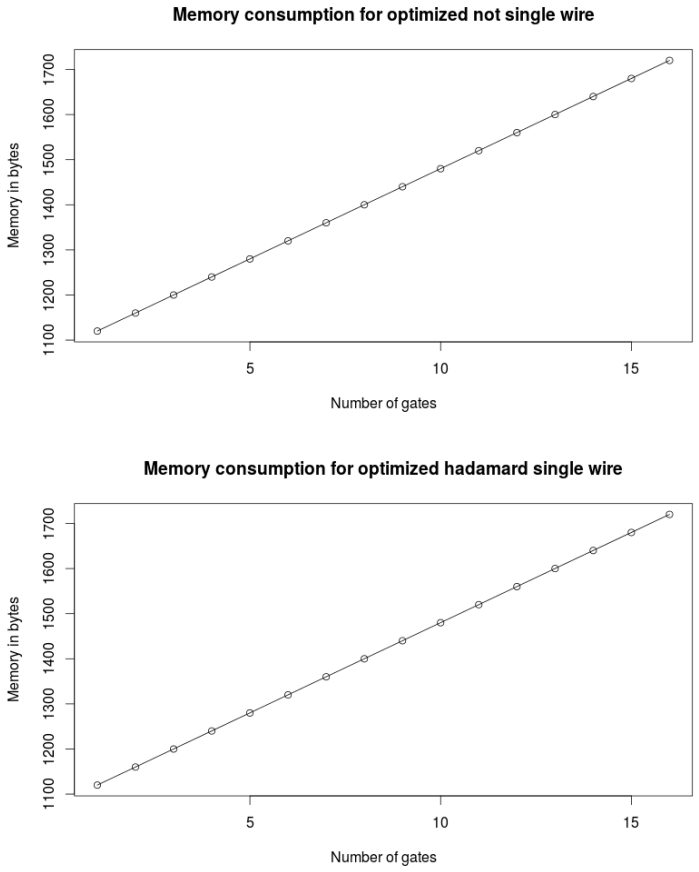


Figure 5.2: A side by side comparison of growth in hadamard and not gates. They are strangely identical.

During a single-wire run, the memory allocation grows linearly for both solutions, which is sound seeing as we only add additional gates to the circuit with exactly the same sizes during these tests, giving us a linear growth by definition. Here the optimized and naive should be, and are, comparable.

However, we would hypothesize that the optimized version using not gates would capture less memory because it would only need to store half the entries in a not gate compared to a hadamard gate - but it does not seem to employ its sparsity in a way which is beneficial to the memory consumption. This can be seen by how in figure 5.2, down to the byte, the allocation for both hadamard (non-sparse) and not (sparse) are exactly the same.

Liquid comparison

Again, Liquid comparison with both optimized and naive follow exactly the same tendency of memory usage as with runtime. Liquid seems to use memory in logarithmic tiers as seen in figure 5.3, while both optimized and naive have an exponential growth in comparison. Although the starting memory usage, and second tier memory usage at 18 qubits, seem to be higher than our optimized memory usage, as time goes on without a doubt our implementation would use more memory than liquid. Again, this is due to liquid being asymptotically superior to our implementations.

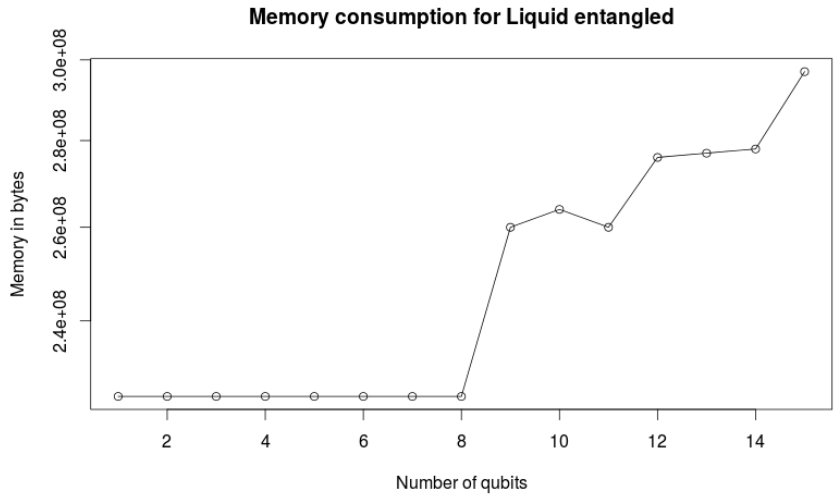


Figure 5.3: Memory consumption for Liquid run on entanglement2

5.3 Discussion

Figure 5.4 shows the comparison between memory consumption of the two solutions. We hypothesized that the exponential increase in memory for the optimized solution stemmed from the instantiation of a qubit vector. We therefore also ran a test where we only instantiated a qubit vector with increasing size alongside the circuit tests. The result is that these two graphs run close to parallel to each other, confirming the theory that the major memory influx in the optimized solution is allocating for the result qubit vector with only a overhead constituting the gates plotting the circuit grid. If the numbers are examined this is doubly clear - we have roughly a doubling of values for each

qubit we add, giving it growth comparable to that of the qubit alone: 2^n. We can thus conclude that the optimized solution does not spend exponential memory to calculate a quantum circuit. The graph also shows us the growth relation between the naive and the optimized solution. It is evident that the slope is much steeper, confirming that it uses a multiple of an exponential more memory.

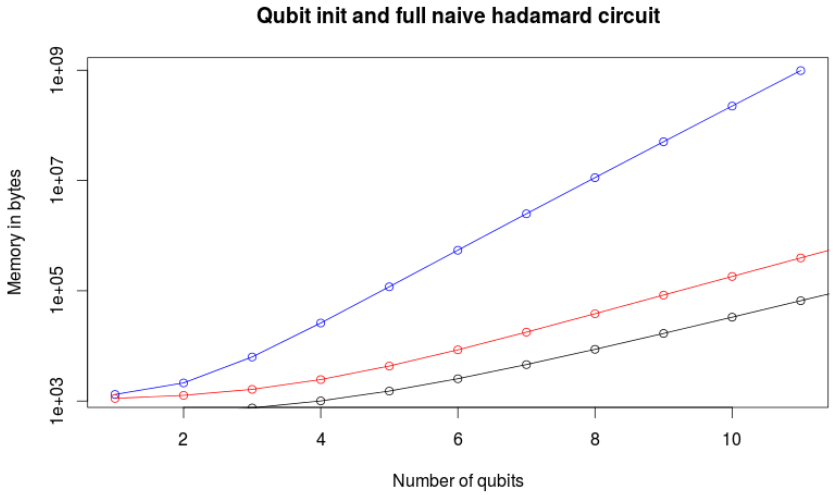


Figure 5.4: Memory consumption for naive version (blue), optimized version (red) and qubit initiation (black) during a hadamard operation.

We are yet to assess the run time results. We expect two wildly different results for two types of circuits: circuits that have a sparse composition and circuits with a non-sparse composition.

- Consider the sparse matrix that arises from a tensor product between n not gates. It has exactly one filled entry for each matrix row. The naive solution does not attempt to realize this during computation and utilizes the run time proposed in chapter 3. n tensor products are performed followed by a qubit/vector multiplication. The final run time is:

$$2^{2n} + \sum_{j=2}^{n} 2^{2(j-1)} 2^{2j}$$

  On the other hand, using our knowledge of the optimized row calculation, we can try to discern its behaviour during the same computation. There is no tensor operation, so we look straight for the slot multiplication. For each entry in the input qubit vector, we need to find only the one non-zero element in the relevant matrix row. The case where every gate acts on precisely one qubit, it necessarily takes n steps to traverse the gate list and find the value. The final run time is therefore $n2^n$.

- This ceases to be the case when all matrix entrances are filled. Then the binary traversal turns into an aggregation of all row elements since the full row vector is relevant for any given row. The traversal cost still applies. Thus the running time becomes $2^{n+1}(2^n - 1)$ as shown in chapter 3.

A multiplicative factor is definitively worse than an addition, making the optimized run time grow faster as $n \to \infty$ in the non-sparse case. On the other hand, 2^{2n} is a greater growth rate than $n2^n$, so the optimized version beats the naive in the sparse case even if the tensor product is discounted. Figure 5.5 shows the run time for the naive and the optimized not circuits and it shows the tendency we expect.

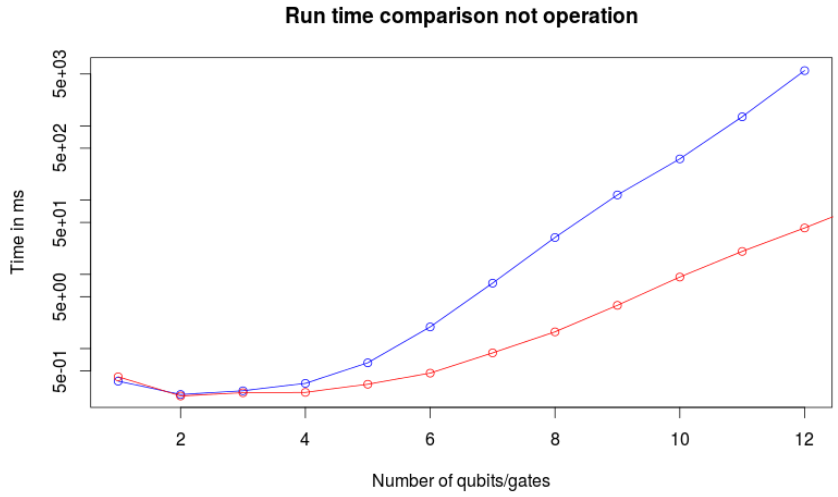


Figure 5.5: Run time comparison between the naive (blue) and optmized (red) solutions during an not operation.

Unfortunately we were not able to test the non-sparse with a sufficient amount of qubits to showcase the optimized version's rampant growth. Growth for qubits up to 12 is shown in figure 5.6. It is a positive sign that the optimized version never performs better than the naive one. For n larger than 3 we can be sure that the optimized version overwhelms the naive run time.

A peculiar discovery was made regarding generated control gates. Examining the output of the optimized solution's delta timing showed an exponential leap in running time of each slot during the entanglement test. A control matrix only has elements in a downwards right diagonal much like an identity gate, meaning all slots only deviate marginally and maintain the same structure. So since every slot has the same amount of qubits, then if one weighed that they should take a uniform execution time because they are equally large and saturated, then the exponential increase in time is a fallacy.

The explanation can be found when dissecting how the control gate is generated. The function takes a range of qubits with target and control bits at

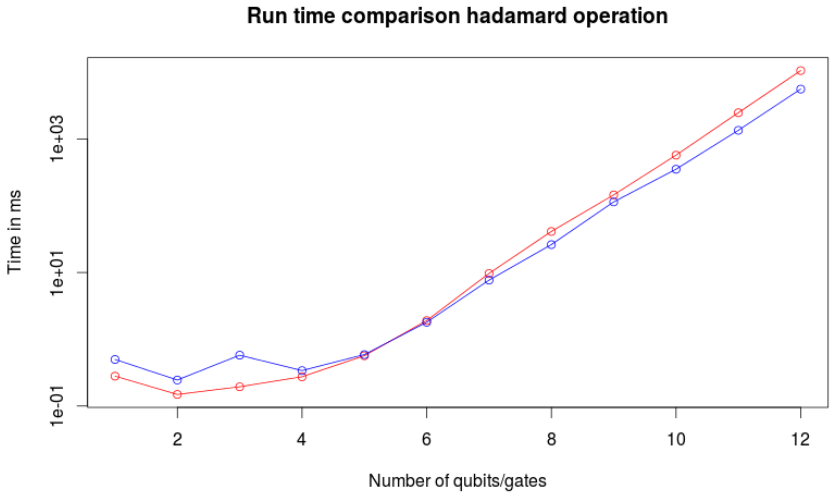


Figure 5.6: Run time comparison between the naive (blue) and optimized (red) solutions during an hadamard operation.

```
delta time for controlGen 0.012000 ms.
delta time for calcSlot 5.352000 ms.
delta time for controlGen 0.023000 ms.
delta time for calcSlot 9.125000 ms.
delta time for controlGen 0.044000 ms.
delta time for calcSlot 17.793000 ms.
delta time for controlGen 0.087000 ms.
delta time for calcSlot 29.141000 ms.
delta time for controlGen 0.178000 ms.
delta time for calcSlot 53.452000 ms.
```

Figure 5.7: Excerpt of delta timing output for entanglement test.

either side, and simply generates a matrix that encompasses this whole span in the circuit. If control and target bits are placed too far apart, this makes the control gate cover 2^{1+c-t} entries in the result matrix. This means the benefit of traversing a gate list to identify indices in the slot matrix that the particular qubit vector entry finds relevant is lost to a simple linear search through the control gate. Obviously this is harmful when the control and target bits are placed on far ends in the circuit as this not only throws us back to something reminiscent of a naive implementation but makes it worse since we also get the added search time that surmounts to $\sum_{j=1}^{2^{1+c-t}} j$ altogether.

It is clear that the definition of gates that act on the whole system or large parts of the system is very inefficient for larger numbers of qubits. It is therefore advisable to not use the simulator for circuits that require this. Examples of gates that fit into this category are what is known as quantum oracles [5]. A quantum oracle is a custom made gate, a 'black box' of sorts, which performs a

desired function f, depending on the system state single, in a single operation. It is in essence a unitary operator U described by its action on a computational basis. Because we made the effort to convert the binary operations that our language supports into optimized versions, we believe it would be beneficial to diagnose the running time for them.

- The naive solution should have identical run time curves. The method is precisely the same regardless of input matrix, as there is no difference in the handling of data. The optimized method for fully instantiated matrices works very much like the naive method, except we need to sort possibly shuffled matrices for certain operations, as described in chapter 3. The running time turns into $2^{3n} + (2^{2n} \log_2 2^{2n}) = 2^{3n} + 2(2^{2n} 2n)$ contra the naive solution's running time of 2^{3n}. Because of the additive notation, we expect a similar growth tendency between them, but with the optimized version having a consistently higher run time.

- A tensored chain of not gates contains exactly one value per row and column. For each entry in the result matrix, only one set of values from each input matrix is thus multiplied. We therefore can ignore the value aggregation that adds another multiple of n to the run time, as well as knock a dimension off the sorting time. The final run time in this case is thus $2^{2n} + 2(2^n n)$.

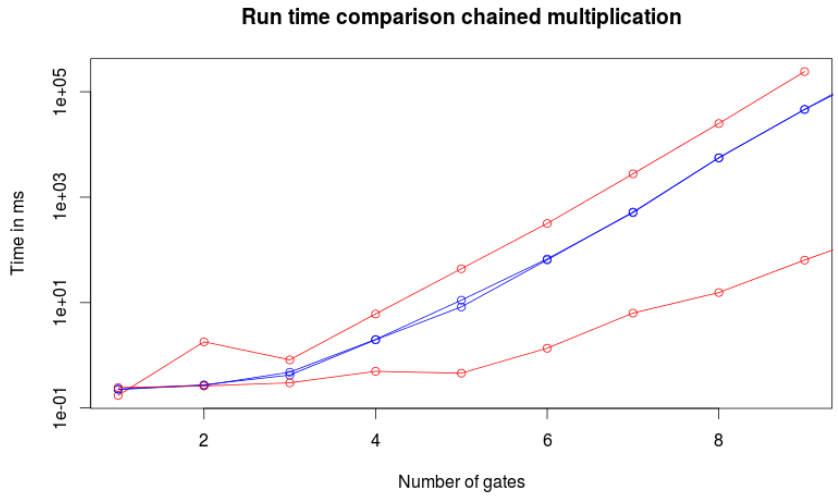


Figure 5.8: Run time comparison of multiplication between chained tensor products. Red lines are the optimized solution and blue lines are the naive solution.

The graph in figure 5.8 shows the run time comparison between not and hadamard gates. The optimized solution's graphs are coloured red while the naive solution's graphs are coloured blue. It can be seen that the optimized hadamard operation follows the naive operations closely, but stays ahead by about one increment of n. We find this flabbergasting as there is no correlation

such that $2^{3n} + 2(2^{2n}2n) = 2^{3(n+1)}$. Instead the difference should diminish as $2(2^{2n}2n)$ grows slower than 2^{3n} as $n \to \infty$. We cannot ably explain this discrepancy. Our qualified guess is that there are not enough data points to represent the tendency or that the sorting time leans more towards the worst case of quicksort. The worst case run time quicksort is $O(n^2)$, making the overall run-time $2^{3n} + 2(2^{2n})^2 = 2^{3n} + 2^{4n+1}$. This should have an even steeper growth curve because of the $4n$ in the exponent, making this infeasible in and of itself to explain the occurring phenomenon.

5.4 Alternative ideas for optimization

Quantum simulation has no silver bullet regarding efficiency of all quantum circuits. If we look to go beyond the asymptotic exponential bound it is necessary to abandon the simple quantum circuit model employed in this chapter and hypothesize classical counter parts. [9] proposes the idea of 'de-quantising' quantum algorithms into classical ones applicable for a probabilistic Turing machine or an equivalent model, when certain attributes are known for the algorithms. These are replications of quantum algorithms observed to be solvable on tailored classical systems - and are thus detached from classical computations we employ in this project. Any improvements strived for henceforth is well known to not trivialize exponential circuit run time but seeks merely to improve constant costs or computation of special quantum states.

Specially optimized hadamard slot

The hadamard gate is one of the most common quantum operations and is often performed initially on all qubits in a system to bring them into an equal superposition. The operation is cumbersome because all entrances in a hadamard matrix have non-zero values. But at the same time the value of each entrance in the qubit vector post operation is known from the amount of bits the system operates on because they are put in an equal superposition. This means that the numeric value that each qubit vector entry will be composed of is known at inception. The difficulty behind the operation would consist of determining the correct signs for each qubit vector entry.

Convert memory work to linked list memory blocks

As of now all optimized algorithms are forced to allocate for a worst case outcome based on the size of the input matrices. This has two downsides: there might be a surplus of memory that is allocated causing a waste, and the memory further needs to be reallocated to a container that fits it subsequent an operation. An alternative approach might be to allocate data in smaller blocks and store these in a linked list, allowing for a lot more modular memory control. This approach is not without faults though. The access time to a specific element becomes dependant on the amount of memory blocks needed to store the matrix data, which might get quite expensive if blocks are too small. A balanced trade off would not be deduced regarding the block size to not recreate the waste issue while simultaneously not create too large of a traversal time to find the correct data point.

Parallelism

Usage of parallelism with matrix operations is not new [7] [8]. The theory is very well suited because of the massive amount of reads that are necessary while the data remains static. Consider calculating a qubit vector entry during a slot calculation in the optimized compiler. Because all qubit vector entries are handled in isolation with variable sets that would be generated for each running thread per instantiation of the algorithm, we need only worry about read-write conflicts regarding the result matrix. But the matrix data is never overwritten, ergo parallelism would run undisturbed.

Optimization of badly designed circuits

This one is a little left field and has lower precedence than some of the other suggestions. Basically, it is possible that some operations are redundant or may be rewritten to a more efficient representation of the user has not been careful in how he wrote the program.

Avoid constantly allocating memory

The optimized solution has the downside that memory is allocated frequently for `tuple` structs. The struct is necessary because the data structure is more advanced than a simple two-dimensional array, meaning each entry must contains more information. But memory allocation is a relatively expensive operation. It is theoretically possible to maintain more exotic data representations which can cut down on these.

Avoid superfluous usage of double arithmetic

Only a subset of quantum gate matrices contain fractional or complex numbers, but for simplicity, we represent all values as double complex numbers. Floating point arithmetic is magnitudes more expensive than natural counter parts. Some scheme that could differentiate between what value `tuples` hold could mean calculating with the strictly least expensive C value type.

Sparse qubit representation

There is no reason why qubit vectors should not also be represented with a datatype that takes reflect a sparse structure. We forfeited the idea because qubit vectors typically are saturated early in a circuit calculation, counteracting the aim. But it would be a neat little optimization.

Alternative method for generating control gates

The optimized slot calculation is reliant on the use of many small matrices to take advantage of sparseness. We saw how the entanglement test spiralled in run-time when this was replaced with creating increasingly larger control matrices. This is of course because the search time goes up to find relevant indices in each matrix. The search time is linear turning the worst time into $O(2^{1+t-c})$ as well as increasing memory usage. Because we already supply several restrictions to control gate usage there are definitely more effective ways to compute them, like checking the qubit vector directly and performing

a simple 1-bit operation on the target bit should the qubit vector entrance be non-zero.

Freeing of old gates

Our compiler could theoretically be rewritten to analyse matrix liveness. We would account for when gate(s) would no longer be used and free its/their memory. This is however, due to the somewhat small gates which are conventionally used in quantum computation, a minor memory conservation. Gates working on a greater number of qubits are of course a exponential expense on memory. This improvement does not in any way improve memory usage asymptotically.

Conclusions

6

We wished to review if it was possible to implement a library that simulated a quantum circuit calculation with better performance metrics than a naive solution. For this purpose we developed our own quantum circuit simulation language. We specified a syntax and wrote a compiler in Haskell, generating C code that depends on two separate C libraries that contain tools to perform quantum circuit simulations. Our naive solution is defined as a C library that performs any part of the problem precisely as defined in the mathematical theory, while a optimized solution might exploit attributes in the problem, namely the mathematics of linear algebra and sparsity of many quantum gates, that allow it to decrease necessary resources while still reaching the same conclusion. For our language we defined basic matrix arithmetic in C such as addition, subtraction, multiplication, Kronecker product and scaling. Certain other operations were also developed specifically for the optimized version, such as a function made specifically to calculate the result of a slot in the circuit, and a function to sort our new matrix datatype using quicksort. The optimized algorithms were drafted and engineered by ourselves for this purpose.

When collecting data, we quickly realized that the limiting factor for whether we would run out of memory or time was the number of qubits. During benchmarking of the naive solution we actually ran out of memory around the 12 qubit mark when calculating circuits while we never experienced memory starvation during optimized runs before the run time became insurmountable. From observing our data it is clear the optimized version is observably superior to the naive solution regarding allocating memory when running a circuit. However, as it is, it is not asymptotically superior to the naive implementation, because we currently need to allocate the full qubit vector.

We never intended to aim for a growth rate difference in running time, but as can be gathered from our discussion, we achieve significant improvements during circuits calculations that contain a majority of sparse matrices. Through examining the benchmarking graphs and runtime analysis of the various circuit types, we were able to realize the fact that the exponential growth was largely tied to qubit vectors not being sparse. Thus, if a different rate of growth is hoped for, simultaneous sparsity of qubit vectors and gates matrices will be essential to that. In contrast to the validity of this scenario, is the establishing of our reasoning behind not making qubits sparse - non-zero qubits are unlikely to exist in the majority of quantum circuits. Superpositions are essential to quantum computing, meaning we will be hard pressed to . It is

incidentally also a factor which may revert the growth of our optimized version to an exponential tendency.

Despite reasoning that large gates are inefficient for our optimized slot calculation, we wanted to collected data for our optimized arithmetic. The results are what one might call polarized. When the matrices are sparse there is little data to process and sort through, making it better than the naive approach. However, if the matrices are dense, such as the hadamard gate, our optimized solution is actually worse than the naive one. This could be due to the fact that our instance of quicksort is operating in the worst case run-time or that we are prevented from collecting sufficient data by running out of heap memory, to show that the naive has a worse growth rate. It is so far unknown what the cause is, but the naive implementation shows a tendency we expected; very similar, if not completely identical, growth rate for both sparse and dense gates, since we don't take this factor into account.

The comparison to Liquid was as expected - Liquid was superior in terms of growth of both time and memory. It uses techniques which are not disclosed through sheer benchmarking, but given a input qubit number large enough, the exponential tendency is revealed. It is clear that we have other venues we could pursue looking forward in terms of optimization, but we should not count on a overarching non-exponential growth when Liquid does not escape the asymptotic bound.

Quantum computing is based on the exploit of physical properties yielding exponential speed-ups and parallelism. Being unable to use a quantum computer, reason would serve our computational expense in simulating it (without ground-breaking research supporting our methods) would also be exponential. This proves to be true, as evident by Liquid, which still falls victim exponential growth.

The language still has certain short-comings and features we would like to implement. However, everything needed to perform a circuit simulation is present. We designed a suite of tools for quantum circuit simulation in C, and designed our own language to go with them in Haskell. We successfully exploited the properties of simulating quantum computing on a classical system, to optimize our naive version with documentable speed-ups in runtime, and savings in memory. All in all, this project fulfilled its initial purpose, and should be considered a success.

Bibliography

[1] Eleanor Rieffel and Wolfgang Polak, *An Introduction to Quantum Computing for Non-Physicists*, 1998, quant-ph/9809016m ACM Comput.Surveys 32:300-335,2000

[2] Dave Wecker *LIQUi|⟩ Users Manual* 2015,2016, Microsoft Corporation, in press

[3] Stephen Diehl, *Write You a Haskell - Building a modern functional compiler from first principles*, 2015, in press

[4] Miran Lipovača, *Learn You a Haskell for Great Good! - A Beginner's Guide*, 2011, in press

[5] Michael A. Nielsen and Isaac L. Chuang, *Quantum Computing and Quantum Information - 10th edition*, 2010, in press

[6] Berger, Emery D., Benjamin G. Zorn, and Kathryn S. McKinley, *OOPSLA 2002: Reconsidering custom memory allocation*, ACM SIGPLAN Notices 48.4S (2013): 46-57.

[7] Choi, Jaeyoung, David W. Walker, and Jack J. Dongarra, *PUMMA: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers*, Concurrency: Practice and Experience 6.7 (1994): 543-570.

[8] Fatahalian, Kayvon, Jeremy Sugerman, and Pat Hanrahan, *Understanding the efficiency of GPU algorithms for matrix-matrix multiplication*, Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware. ACM, 2004.

[9] Abbott, Alastair A., and Cristian S. Calude, *Understanding the quantum computational speed-up via de-quantisation*, arXiv preprint arXiv:1006.1419 (2010).

[10] GCC developers: multiple contributors, GCC 6.1, *https://gcc.gnu.org/*, 1987-2016

[11] McCracken, Daniel D., and Edwin D. Reilly, *Backus-naur form (bnf)*, (2003): 129-131.

[12] Cleve, Richard, et al, *Quantum algorithms revisited*, Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences. Vol. 454. No. 1969. The Royal Society, 1998.

[13] R developers: multiple contributors, R version 3.3.0, *https://www.r-project.org/*, 1993-2016

[14] Valgrind developers: multiple contributors, Valgrind-3.11.0, *http://valgrind.org/*, 2000-2015

Appendix

A

These are all of the graphs produced in gathering benchmark data. The runtime data is collected through delta time in the C programs, and the memory data is collected through Valgrind. For naive and optimized solutions, we have 12 graphs each, 6 unique tests for both memory and runtime. These 6 tests are subdivided into 3 forms of benchmarking tests performed with hadamard and not gates; growth in qubits, with a single gate being performed on them throughout, growth in gates with a single qubit being performed on throughout, and growth for both qubits, and how many gates are run on them.

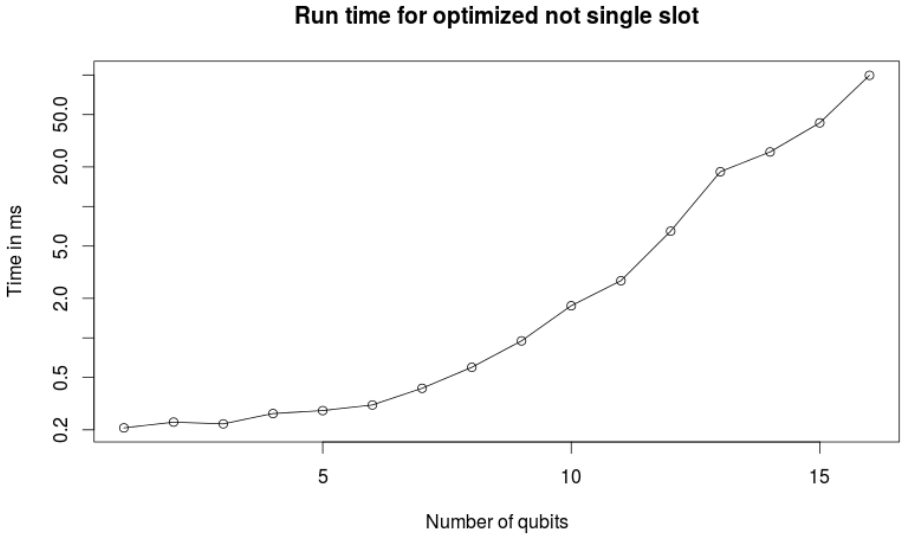


Figure A.1: Running time for optimized not gate increasing in qubits.

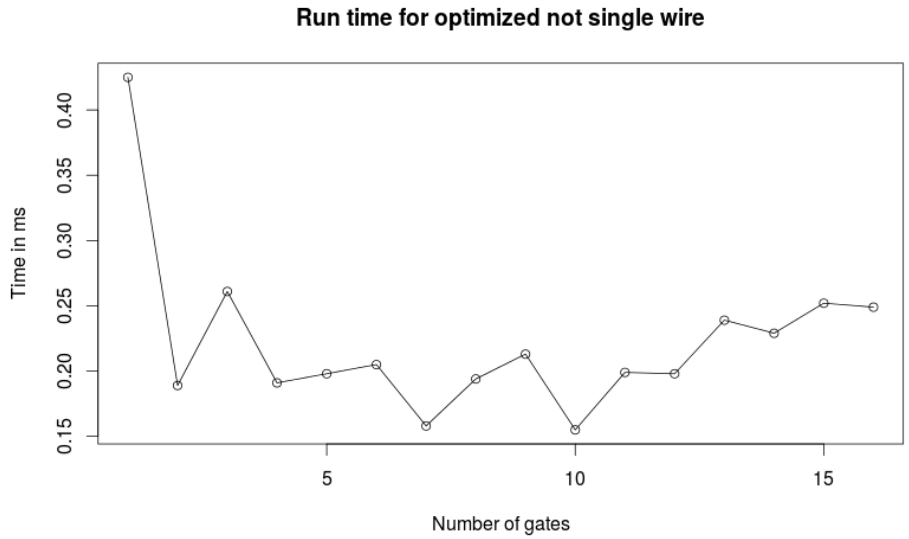


Figure A.2: Memory consumption for optimized not gate increasing in gates.

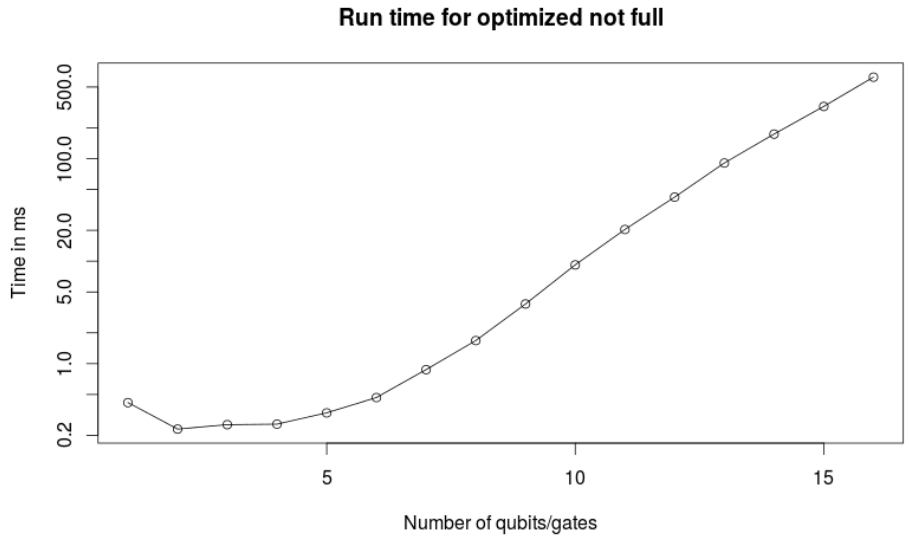


Figure A.3: Running time for optimized version increasing in qubits and gates.

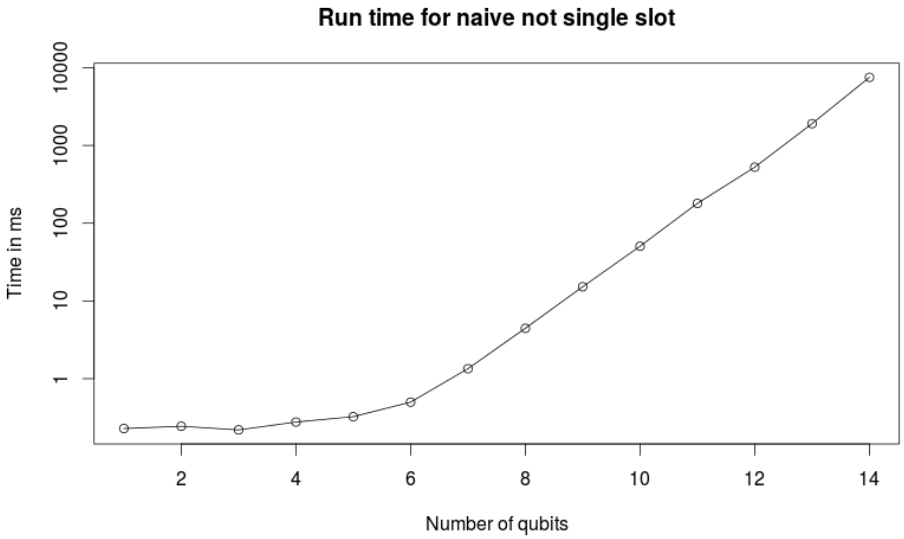


Figure A.4: Running time for naive version increasing in qubits.

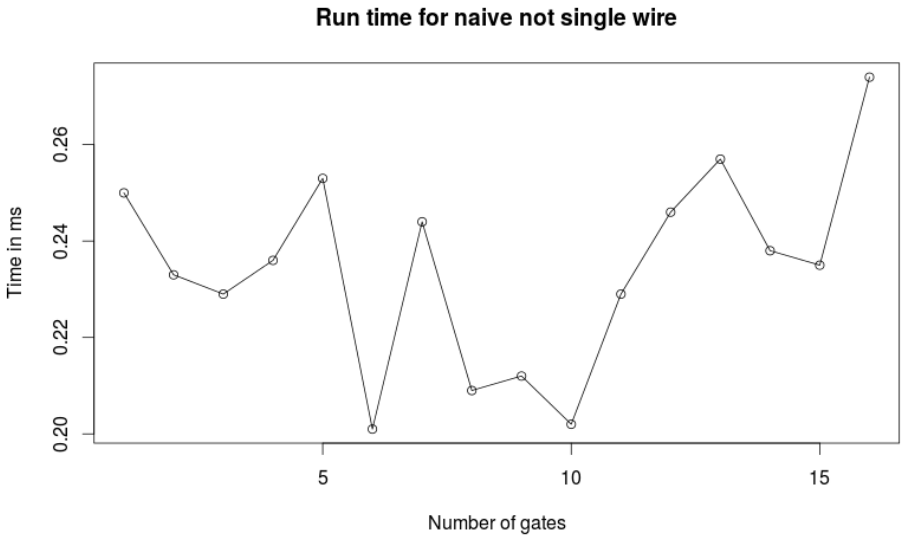


Figure A.5: Memory consumption for naive version increasing in qubits.

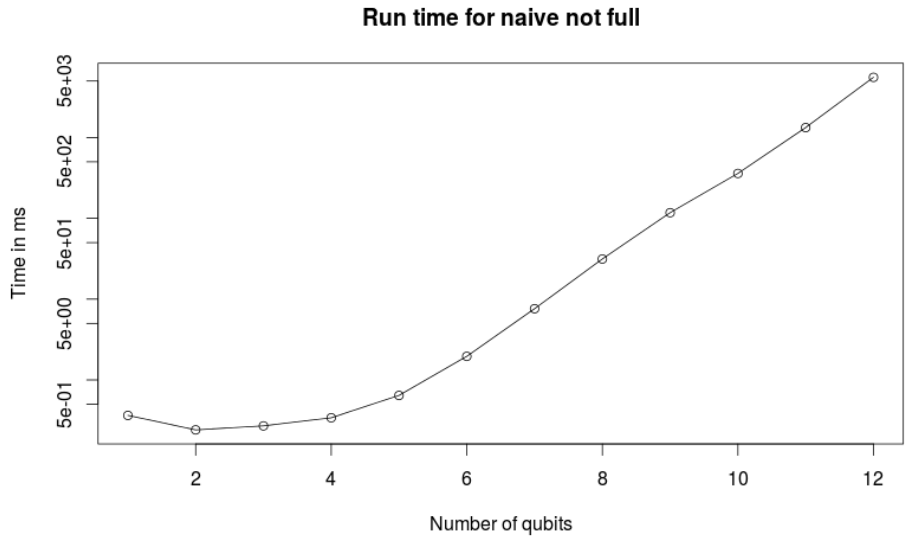


Figure A.6: Running time for naive version increasing in qubits and gates.

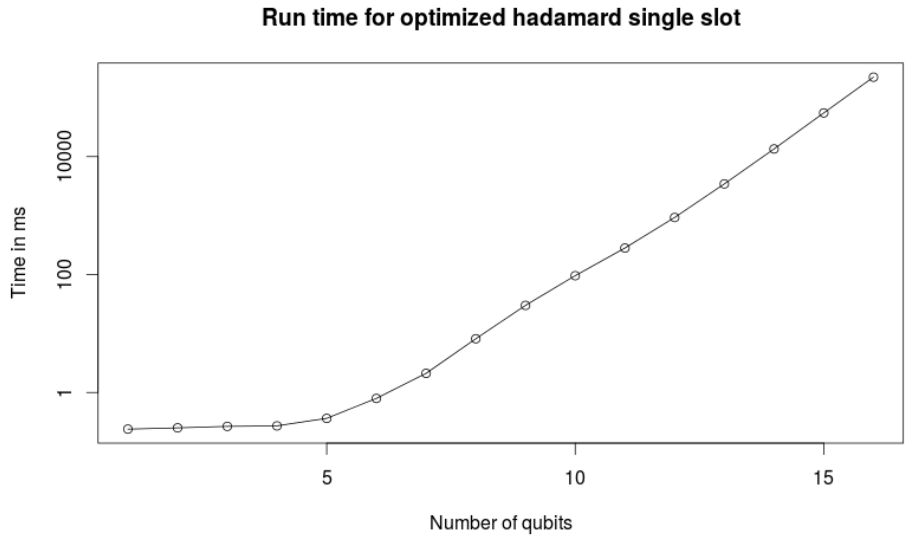


Figure A.7: Running time for optimal version increasing in qubits.

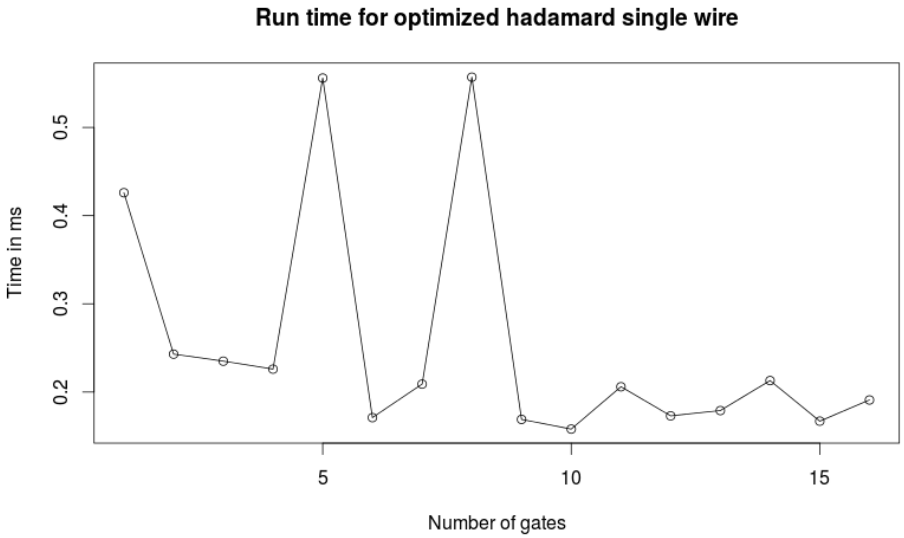


Figure A.8: Memory consumption for naive version increasing in qubits.

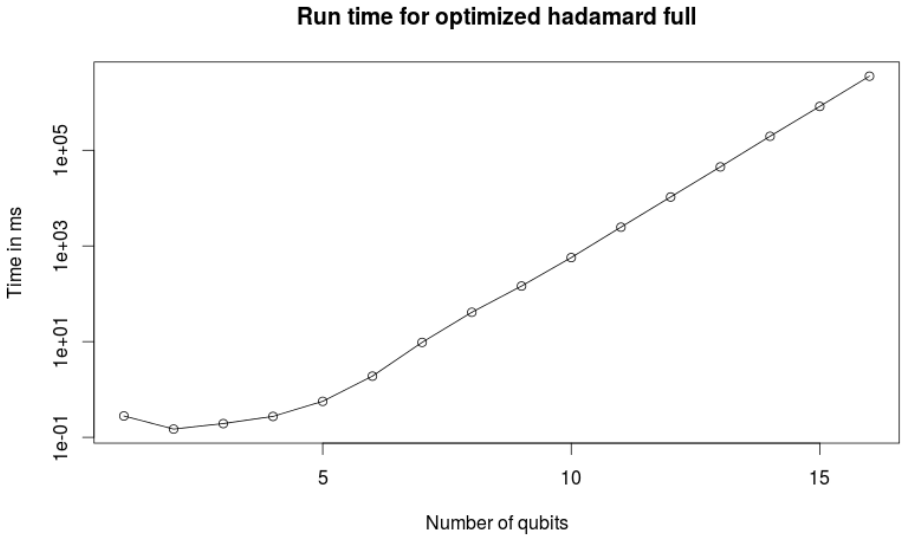


Figure A.9: Running time for optimal version increasing in qubits and gates.

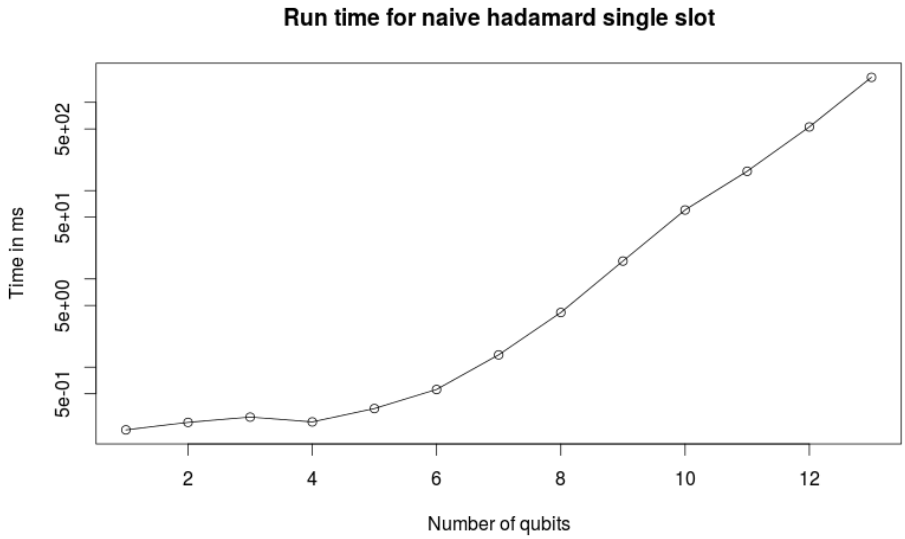


Figure A.10: Running time for naive version increasing in qubits.

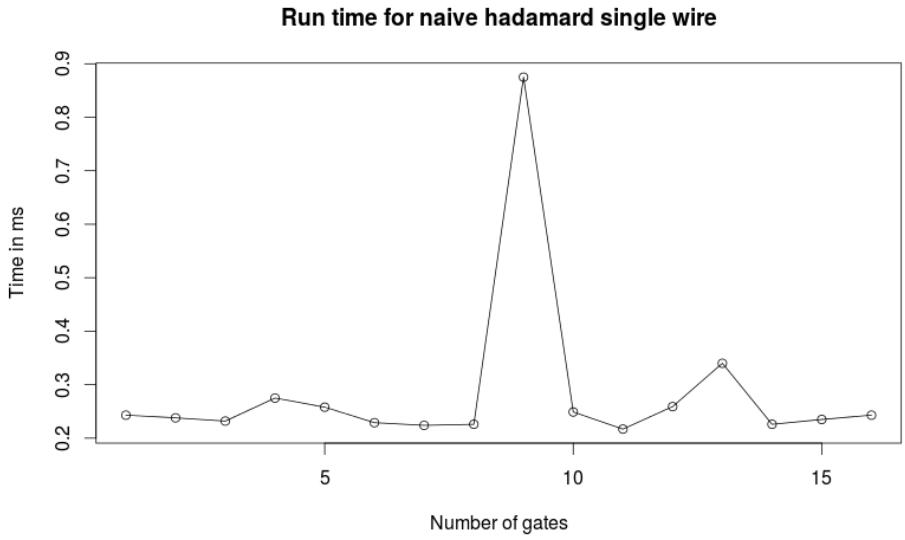


Figure A.11: Memory consumption for naive version increasing in qubits.

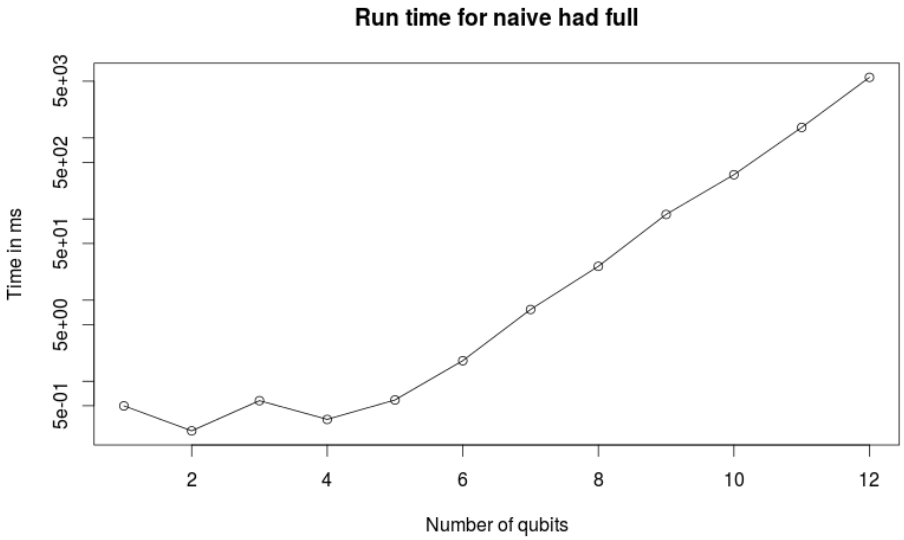


Figure A.12: Running time for naive version increasing in qubits and gates.

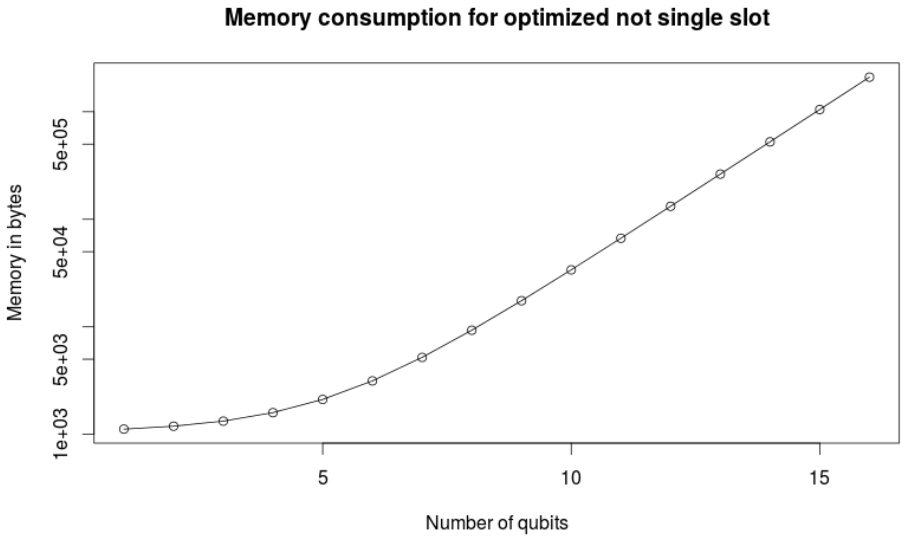


Figure A.13: Memory consumption for optimal version increasing in qubits.

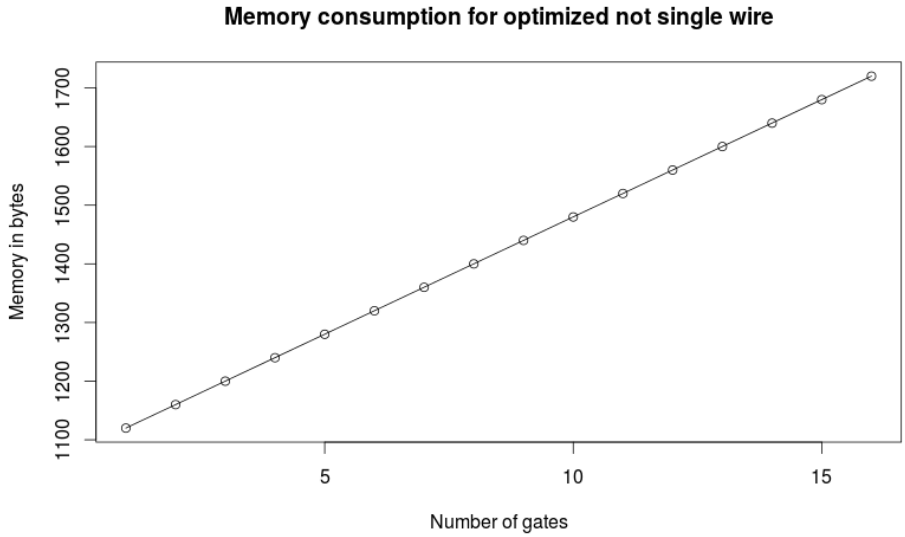


Figure A.14: Memory consumption for naive version increasing in qubits.

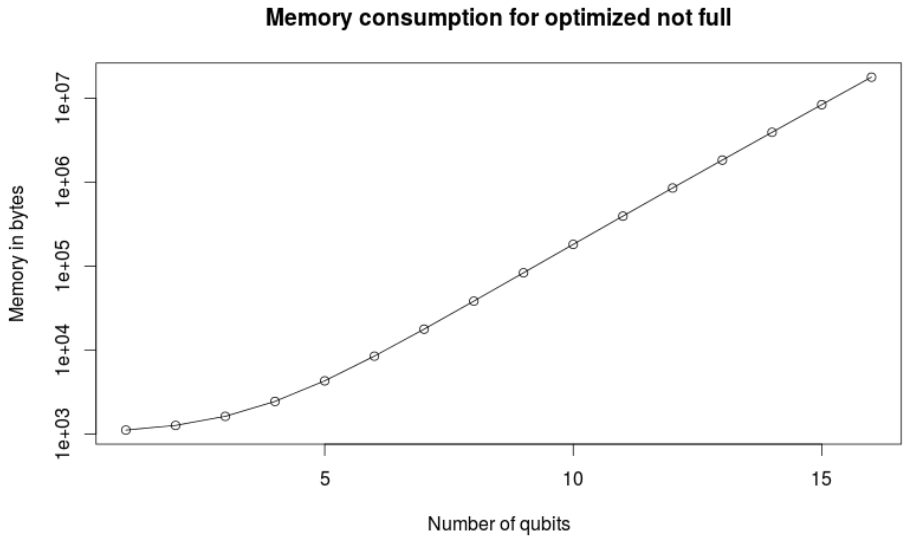


Figure A.15: Memory consumption for optimal version increasing in qubits and gates.

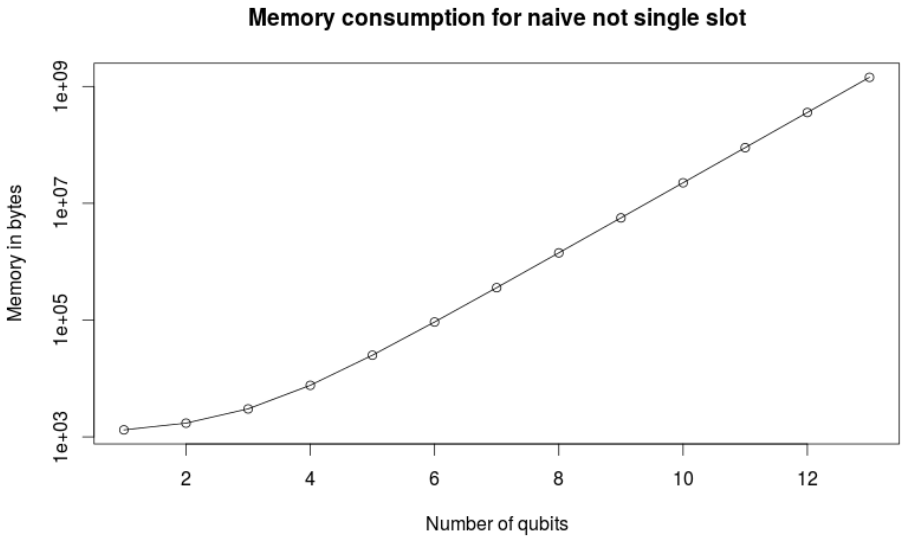


Figure A.16: Memory consumption for naive version increasing in qubits.

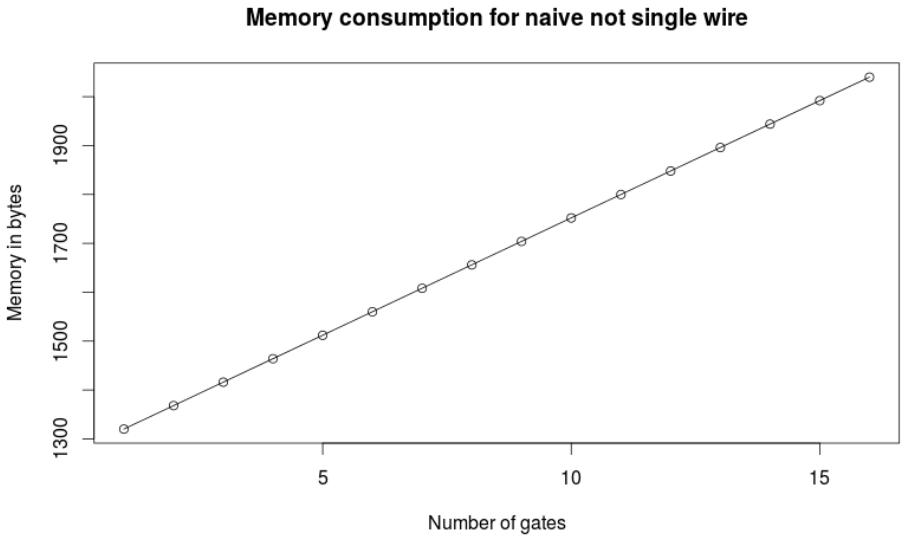


Figure A.17: Memory consumption for naive version increasing in qubits.

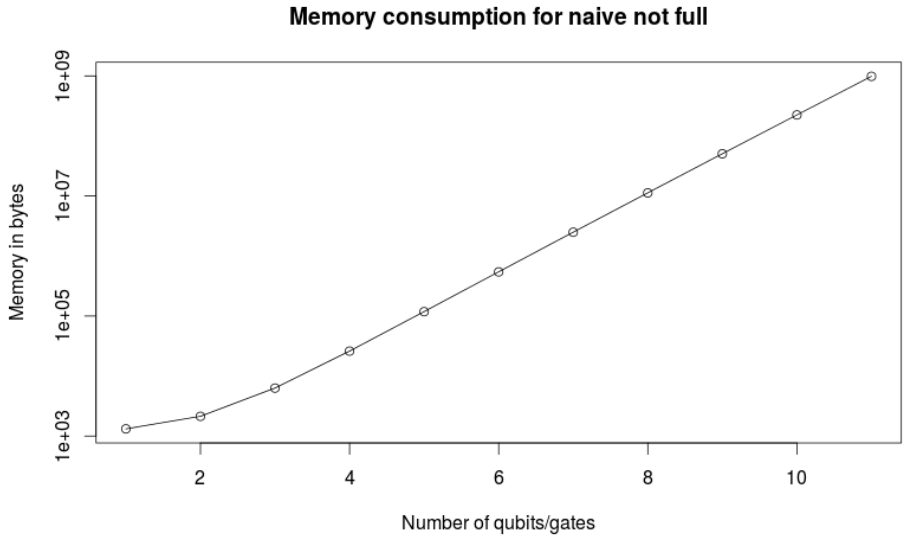


Figure A.18: Memory consumption for naive version increasing in qubits and gates.

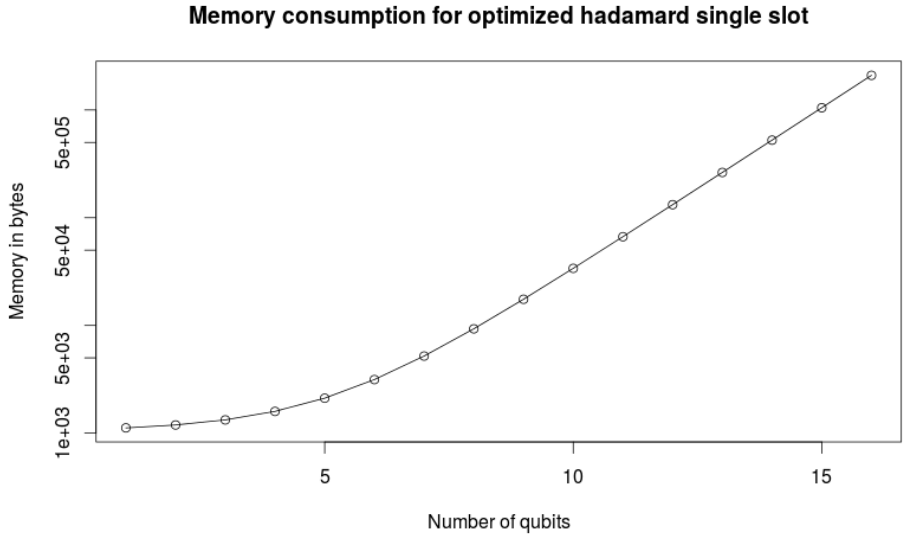


Figure A.19: Memory consumption for optimal version increasing in qubits.

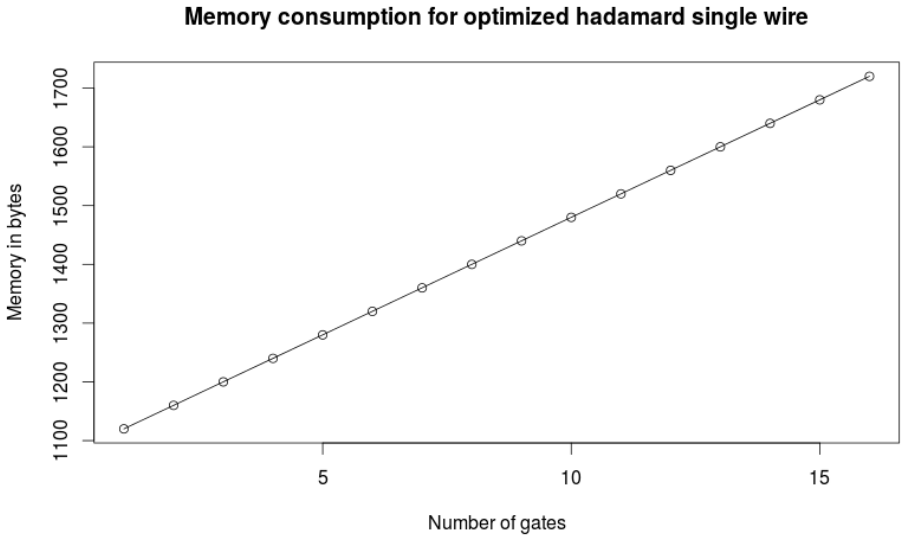


Figure A.20: Memory consumption for naive version increasing in qubits.

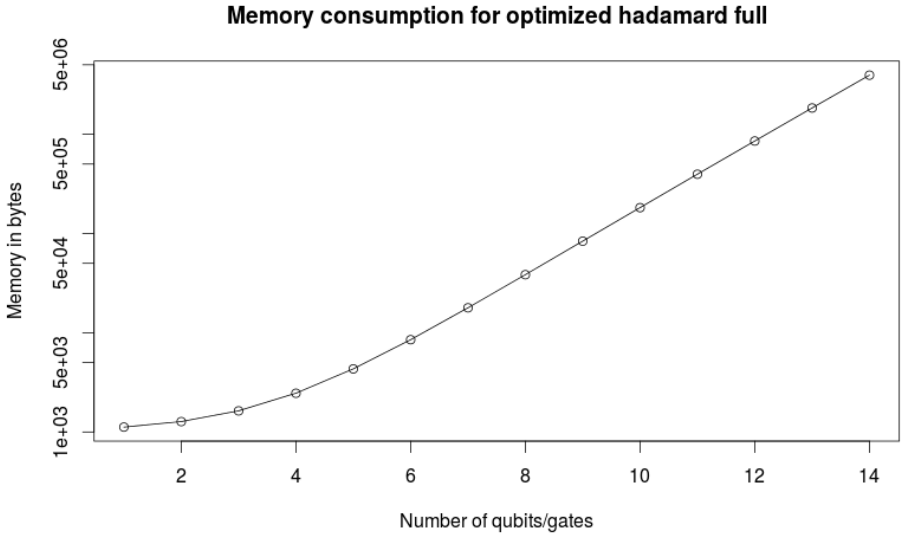


Figure A.21: Memory consumption for optimal version increasing in qubits and gates.

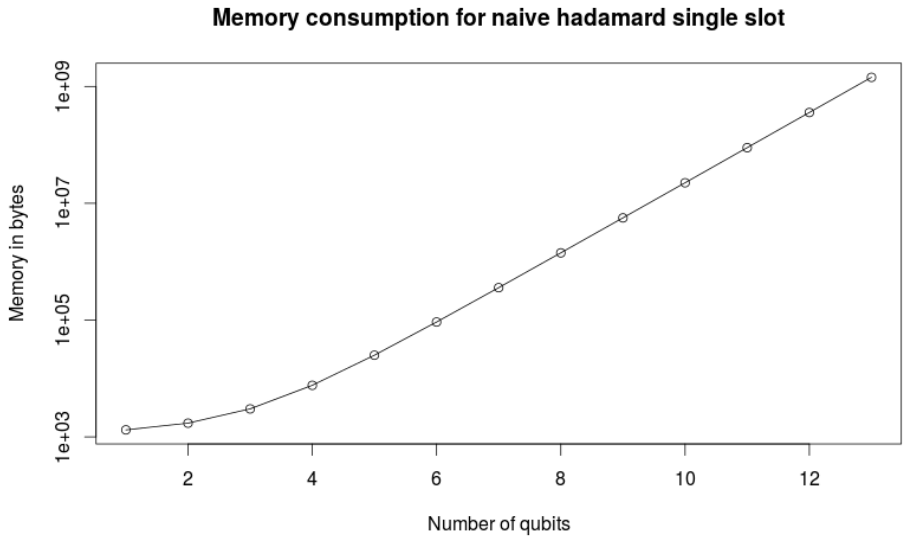


Figure A.22: Memory consumption for naive version increasing in qubits.

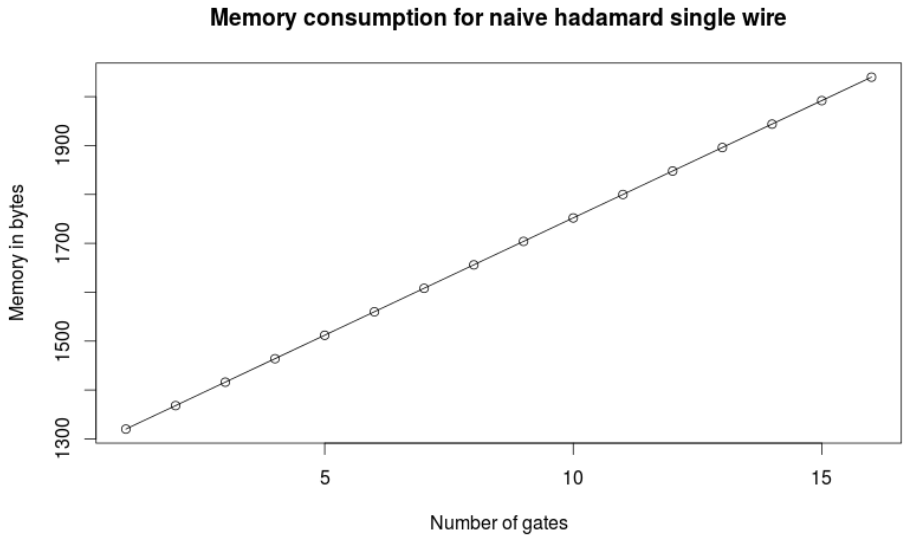


Figure A.23: Memory consumption for naive version increasing in qubits.

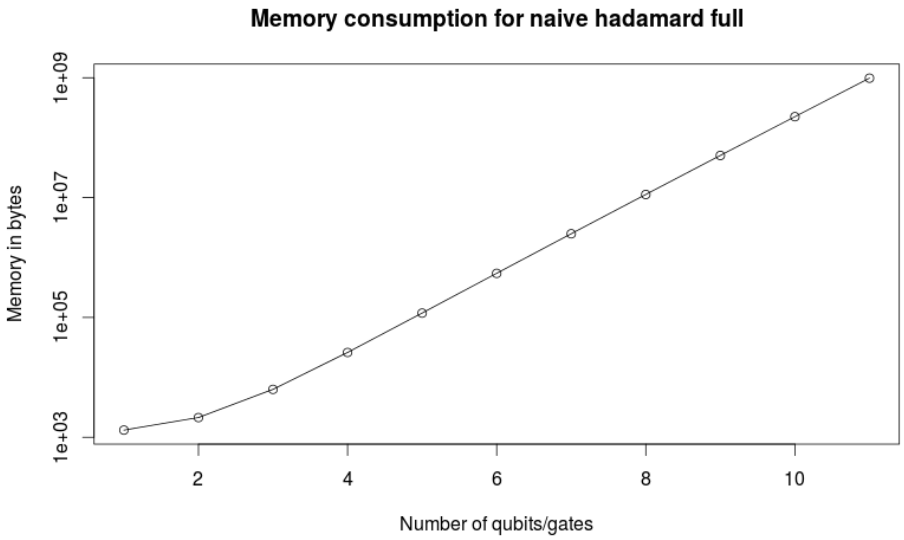


Figure A.24: Memory consumption for naive version increasing in qubits and gates.

Appendix

B

These are the graphs for comparison with Liquid, all of the tests increase in the number of qubits which are brought into an entangled state. Both memory consumption and runtime graphs are included for both versions and Liquid.

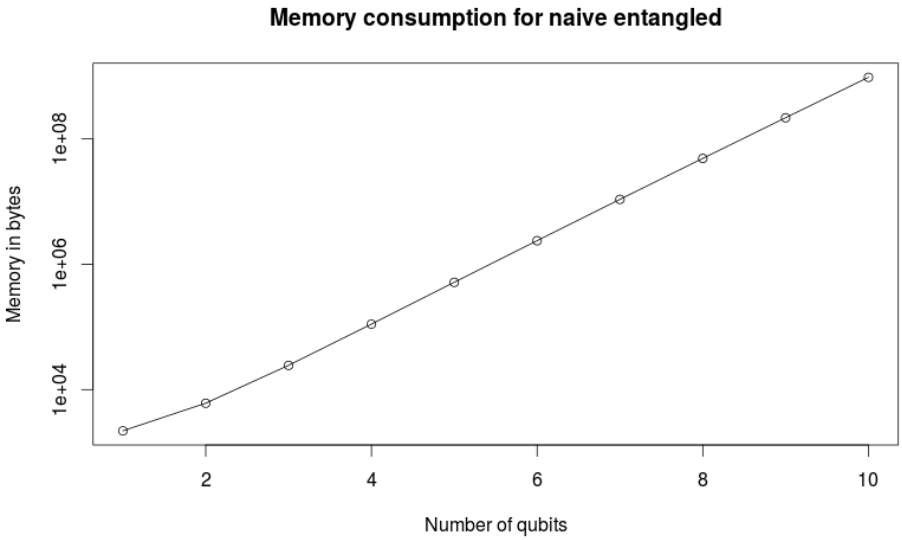


Figure B.1: Memory consumption for naive version increasing in qubits.

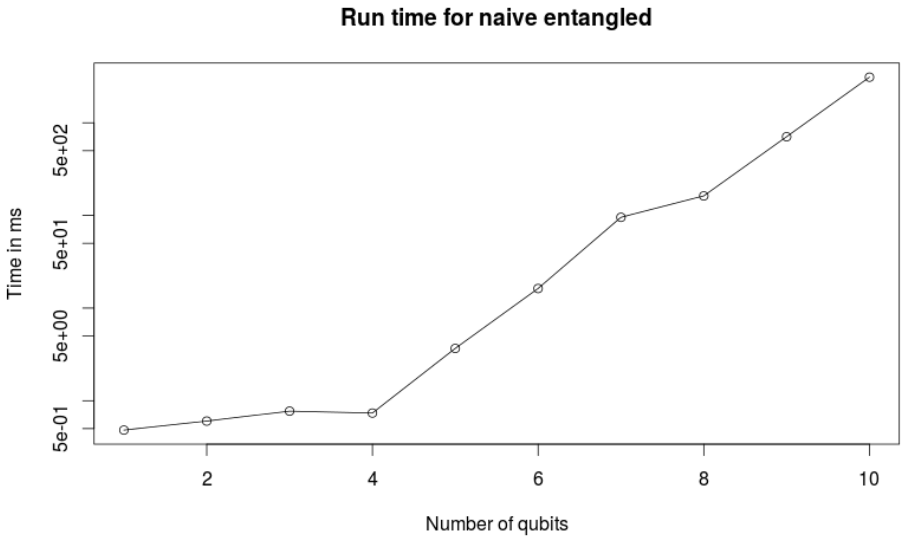


Figure B.2: Running time for naive version increasing in qubits.

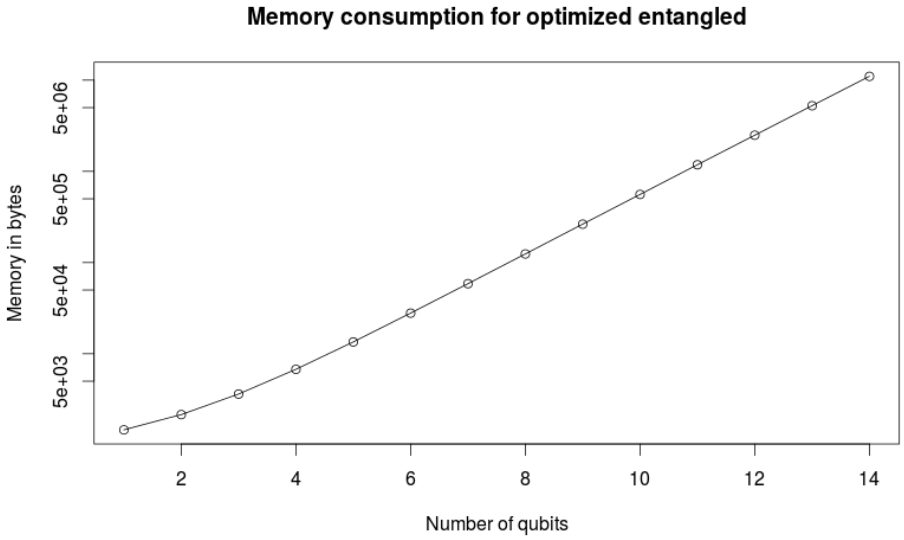


Figure B.3: Memory consumption for optimal version increasing in qubits.

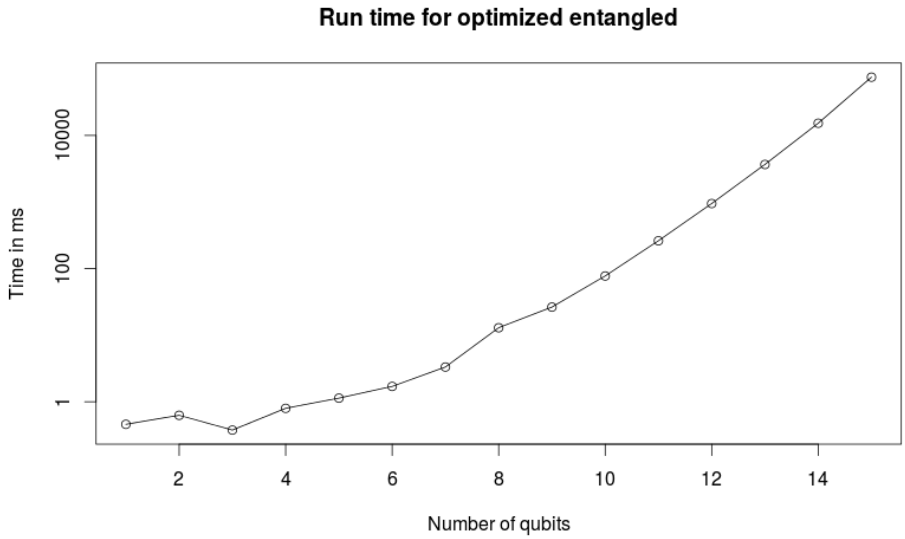


Figure B.4: Running time for optimal version increasing in qubits.

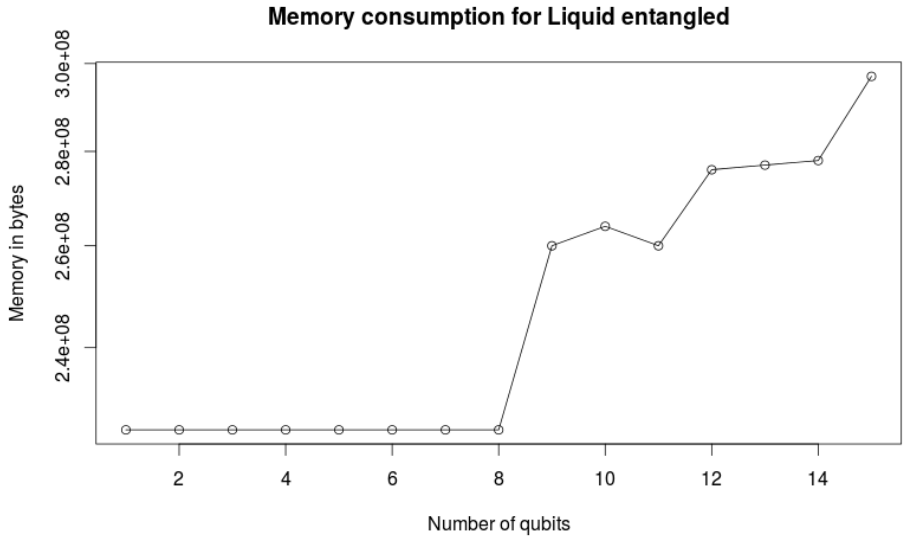


Figure B.5: Memory consumption for liquid increasing in qubits.

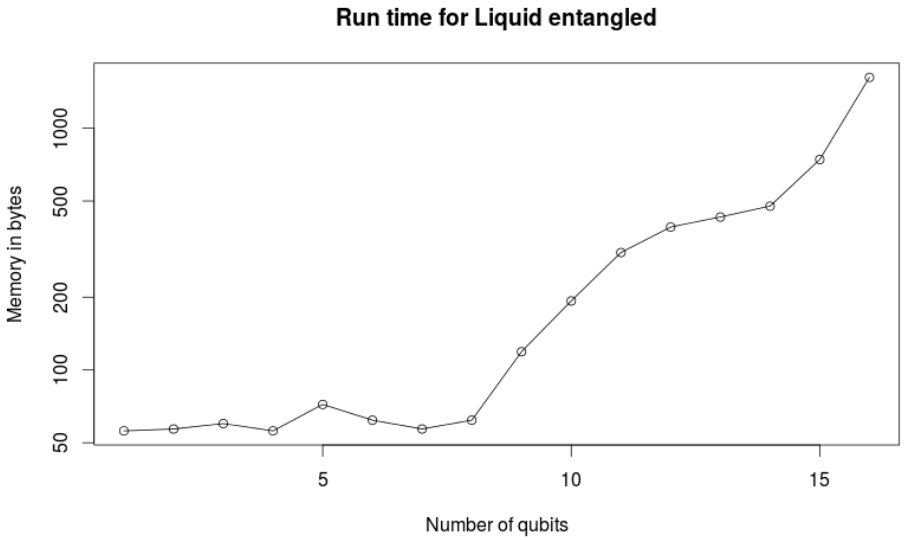


Figure B.6: Running time for liquid increasing in qubits.

Appendix

C

These are the graphs for growing chains of Kronecker products multiplied together. All of the tests increase in the number of gates on either side of the binary operation. We only tested for multiplication run time.

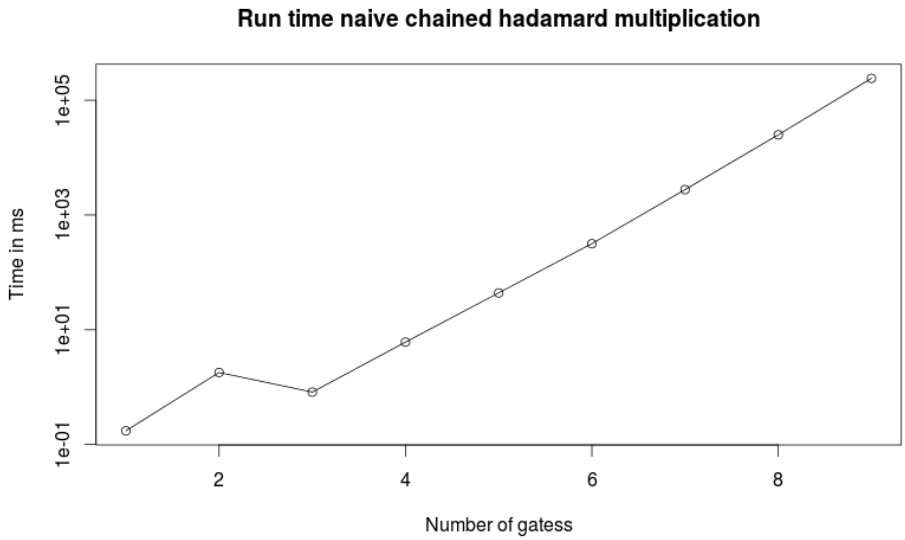


Figure C.1: Running time for optimized hadamard multiplication.

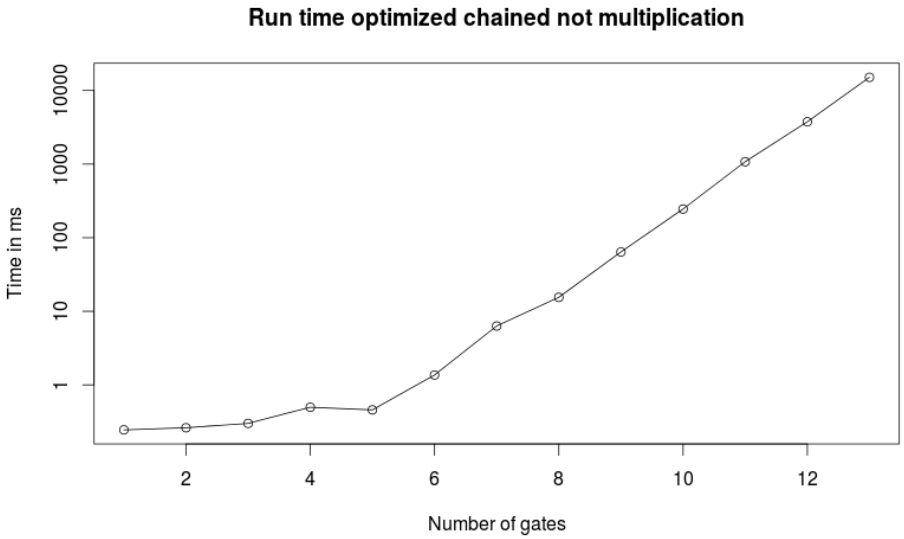


Figure C.2: Running time for optimized not multiplication.

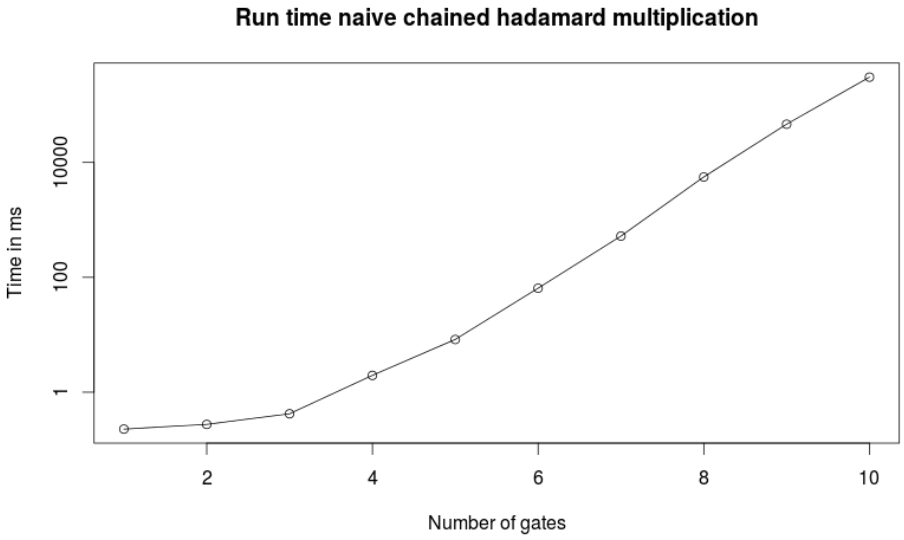


Figure C.3: Running time for naive hadamard multiplication.

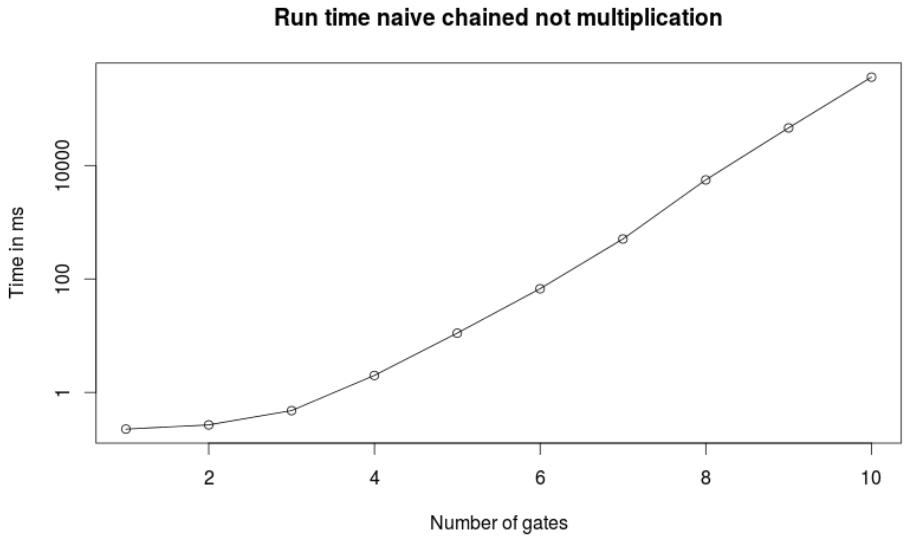


Figure C.4: Running time for naive not multiplication.