

SML Quantum Circuit Optimizer: SQuanto

Therese Lyngby
rsk371

Anna Filippa Biil
ldv927

Jóhann Utne
dv1176

Supervisor: Martin Elsmann

July 1, 2025

Abstract

Experiments of quantum circuits have shown, that big circuits introduce noise, which leads to erroneous results which in turn requires quantum programs to be run a larger number of times. To mitigate this, multiple forms of quantum circuit optimizers have been developed, that use predefined circuit identities. However, for many quantum circuits, optimizations using only predefined circuit identities may be too restrictive. Therefore the concept of an optimizer that automatically generates all possible circuit identities has been explored in Quanto and Quartz. This project tries to replicate these optimizers in the functional language SML and explores the limitations of this approach. We found it possible to successfully implement a prototype. We found through benchmarking that the biggest issue of our implementation was the exponential blowup of operations in the generator. Finally, we identify possibilities for parallelization in the generation, based on our implementation and describe a possible extension, to include handling the generation in a data-parallel language like Futhark.

Contents

1	Introduction	4
2	Background	4
2.1	Quantum Circuit Basics	4
2.1.1	Common Gates	6
2.1.2	Universal Gate Set	6
2.1.3	Measuring Quantum Systems	6
2.2	Optimizing Quantum Circuits	7
2.3	Quanto	8
2.3.1	A Circuit Identity Generator	8
2.3.2	Tiling	9
2.3.3	The Quantum Circuit Optimizer	10
2.4	Quartz	10
3	Method	11
3.1	Generator	11
3.1.1	Representing Circuits	11
3.1.2	Tile Generation	12
3.1.3	Unitary Calculation	12
3.1.4	Generating Fingerprints	13
3.1.5	Database Generation	13
3.2	Optimizer	14
3.2.1	Tiling	14
3.2.2	Optimizing	14
4	Results	15
4.1	Validation	15
4.2	Experimental Results	15
5	Discussion	16
5.1	Implementation	16
5.1.1	Multi-Qubit Gates	16
5.1.2	Gate Set and Tile Size	17
5.1.3	Fingerprints	17
5.1.4	Optimizer	18
5.2	Results	18
6	Further Work	19
6.1	Parallelization of Generation	19
6.2	Parallelization of the Optimizer	20
6.3	Support for Multi-Qubit Gates	20
7	Conclusion	21

8	Appendix	23
8.1	Generator implementation	23
8.2	Optimizer	27
8.3	Unit tests	31
8.4	bashscript	34

1 Introduction

A major challenge in quantum computing is, that as the depth of a quantum circuit increases, noise in the computation of the circuit also increases. Due to the inherent probabilistic nature of quantum circuits, many quantum algorithms require that you run a quantum circuit multiple times to get a result. Consequently, the more noise we have in computation, the more times we need to execute the quantum circuit, increasing runtime.

Another challenge is that some quantum logic gates are more difficult to compute in a quantum circuit than others, and are machine-dependent. The challenges presented are clear motivations for why optimizations of quantum circuits are necessary: to decrease the runtime and error rate of quantum computation.

However, optimization has challenges. One challenge is that there exists an infinite number of logical quantum gates, and different machines use different sub-sets of logical quantum gates for expressing quantum circuits. This is problematic because optimizations have been derived for some combinations of a subset of quantum logical gates, but there are no guarantees that these optimizations are exhaustive.

Solutions to this issue have been addressed by quantum circuit optimizers Quanto [8] and Quartz [11], where the authors propose methods for automatically generating optimizations for circuits on a fixed quantum gate set.

In this project, we implement an optimizer for quantum circuits in the functional language SML, by drawing inspiration from Quanto [8] and Quartz [11].

Due to the project's scope, we only developed a prototype of an optimizer, with limited capabilities compared to Quanto or Quartz.

The project then finishes with addressing the problems Quanto and our implementation and discusses a potential improvement, involving parallelization.

2 Background

2.1 Quantum Circuit Basics

Quantum computers are special because their bits behave differently than the bits of classical computers. A quantum bit (qubit) can be mathematically expressed as a unit vector on the bloch sphere. Specifically, a single qubit state $|v\rangle$ can be expressed as:

$$|v\rangle = \alpha|0\rangle + \beta|1\rangle$$

Where $\alpha, \beta \in \mathbb{C}$ and $\|\alpha\|^2 + \|\beta\|^2 = 1$. We define $|0\rangle$ and $|1\rangle$ as:

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix},$$

A general q -qubit state $|\psi\rangle$ can be described as:

$$|\psi\rangle = \sum_{i=0}^{2^q-1} a_i |i\rangle \quad (1)$$

Where the amplitudes $a_i \in \mathbb{C}$ and $|i\rangle$ is a unit vector of length 2^q with a one in the i 'th entry and zeroes in all others. For all quantum states we require, that the squared amplitudes sum to one, that is:

$$\sum_{i=0}^{2^n-1} \|a_i\|^2 = 1$$

Furthermore, if we have q independent 1-qubit states $|\psi_i\rangle$ for $i \in \{1, 2, \dots, q\}$, we can define their shared q -qubit state $|\Psi\rangle$ as the Kronecker product:

$$|\Psi\rangle = \bigotimes_{i=1}^q |\psi_i\rangle \quad (2)$$

A quantum program can symbolically be represented by a quantum circuit. A quantum circuit consists of a column of qubits representing an initial quantum state to which a sequence of quantum gates are applied to obtain a new state. Each qubit in the circuit is represented by a line upon which quantum gates are drawn in boxes. Similar to quantum states, quantum gates are mathematically represented by unitary matrices. The process of applying a quantum gate X to a quantum state $|\psi_{in}\rangle$ can be expressed by matrix multiplication as:

$$|\psi_{out}\rangle = X |\psi_{in}\rangle$$

Applying 2 quantum gates X and Y to the same qubit $|\psi_{in}\rangle$ in a sequence (first X and then Y) is described by applying a single gate Z to $|\psi_{in}\rangle$ where $Z = Y \cdot X$. The result $|\psi_{out}\rangle$ of the sequence then becomes:

$$|\psi_{out}\rangle = Y \cdot X |\psi_{in}\rangle = Z |\psi_{in}\rangle$$

Figure 1 shows an illustration of an equivalent quantum circuit.

$$\boxed{|\psi\rangle \text{ --- } \boxed{Y} \text{ --- } \boxed{X} \text{ --- } = \text{ --- } \boxed{X \cdot Y} \text{ --- } XY |\psi\rangle}$$

Figure 1: Illustration of a quantum circuit where 2 quantum gates Y and X are applied to the qubit $|\psi\rangle$. From Wikipedia [10]

Similar to how we express quantum states consisting of multiple qubits with Kronecker products, we also express quantum circuits with multiple quantum gates applied to multiple qubits using Kronecker products. Specifically if we

have an input state $|\psi_{in}\rangle \otimes |\phi_{in}\rangle$ and apply quantum gate Y to $|\psi_{in}\rangle$ and quantum gate X to $|\phi_{in}\rangle$, we can express it as:

$$\begin{aligned} |\psi_{out}\rangle \otimes |\phi_{out}\rangle &= (Y \cdot |\psi_{in}\rangle) \otimes (X \cdot |\phi_{in}\rangle) \\ &= (Y \otimes X)(|\psi_{in}\rangle \otimes |\phi_{in}\rangle) \end{aligned}$$

See Figure 2 for an illustration of an equivalent quantum circuit.

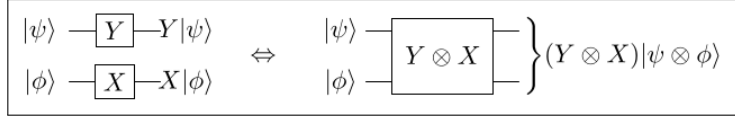


Figure 2: Illustration of a quantum circuit where 2 quantum gates Y and X are applied to the qubits $|\psi\rangle$ and $|\phi\rangle$ respectively. From Wikipedia [10].

Thus, we can represent any quantum circuit C by computing an equivalent unitary matrix $\llbracket C \rrbracket$ by multiplying the Kronecker products of the quantum gates of the circuit. This allows us to simulate running a quantum program by computing $|\phi_{out}\rangle = \llbracket C \rrbracket |\phi_{in}\rangle$.

2.1.1 Common Gates

There exists an infinite amount of quantum gates, but there are some quantum gates that have been named which are often used in the literature. Some notable examples are the Pauli gates (X, Y, Z), the Clifford gates (CX, H , and S), the identity gate I , and the phase gate T . Figure 3 shows the each gate in a quantum circuit along with its unitary matrix. Note, in Figure 3 that the CX gate is applied to 2 qubits, whereas all other gates operate on a single qubit.

2.1.2 Universal Gate Set

A set of quantum gates is universal if any other unitary matrix can be expressed as a finite sequence of gates from the set. Thus, a universal quantum gate set should be able to express any possible quantum circuit. There exists many possible universal quantum gate sets, one example is the set of Clifford gates along with the T gate, i.e. the set $\{CX, H, S, T\}$.

2.1.3 Measuring Quantum Systems

As briefly mentioned a q -qubit state can be expressed as a unit vector of length 2^q . When measuring a quantum state $|\psi\rangle$ the state collapses to the state $|i\rangle$ with probability $\|a_i\|^2$. Since the probability of observing state $|i\rangle$ is given by the squared magnitude of its amplitude $\|a_i\|^2$ the global phase of the system does not affect the probability of observing any one state. I.e. $\|a_i\| = \|p \cdot a_i\|$ for $p \in \{1, -1, i, -i\}$ where i is the imaginary unit.









Operator	Gate(s)	Matrix
Pauli-X (X)	 	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$
Pauli-Y (Y)		$\begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$
Pauli-Z (Z)		$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$
Hadamard (H)		$\frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$
Phase (S, P)		$\begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$
$\pi/8$ (T)		$\begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}$
Controlled Not (CNOT, CX)		$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$

Figure 3: Common quantum gates, their symbolic representation in a quantum circuit, and their unitary matrix representation. Image is modified from an image found on Wikipedia [10].

2.2 Optimizing Quantum Circuits

We say that two quantum circuits are equivalent, if $\llbracket C \rrbracket = \llbracket C' \rrbracket$, where $\llbracket \cdot \rrbracket$ denotes the unitary matrix representation of a circuit. Even if two quantum circuits C and C' are equivalent, one circuit C' may be more efficient than executed on a quantum device. The purpose of optimization is to replace circuit C with the equivalent, but more efficient, circuit C' .

For quantum devices deep circuits are problematic as noise in the computation increases as depth increases, possibly leading to errors in the result. Furthermore, some gates are costly to execute on quantum devices, whereas others are costly to execute on quantum simulators. Thus depending on the device the optimal C' differs.

To identify an optimal circuit C' we can use circuit identities. In this paper we define a circuit identity for circuit C as any circuit C' with the same unitary matrix. For example, for any unitary matrix U it is the case that $UU = I$. Thus any circuit that just repeats the same gate twice in a row has the empty circuit as circuit identity. This allows us to make the substitution $UU \mapsto I$, decreasing the depth of the circuit. Many such circuit identities have been identified in the literature for different gates [8]. An example of well-known circuit identities is

the fact that the Pauli matrices anti-commute [5]:

$$XY = iZ \quad YZ = iX \quad ZX = iY \quad XYZ = iI$$

Thus to optimize a quantum circuit it is common to use circuit identities to make substitutions of gates in the original circuits to achieve a new equivalent one that is more optimal.

2.3 Quanto

Quanto [8] claims to be the first quantum optimizer to automatically generate circuit identities for 1- and 2-qubit gate sets. Previous quantum optimization relied on complicated heuristics such as the Verified Optimizer for Quantum Circuits [6], which only include manually discovered circuit identities or automatically generated circuit identities for single-qubit gate sets. This left out optimization opportunities, and meant that the benefit of optimizing was depended on the gate set used to define a circuit.

Automatically generated circuit identities for 2-qubit gate sets contain both previously known and unknown identities, which allow Quanto to optimize circuits better than well known alternatives such as the IBM compiler [8]. Provided the same input circuits, circuits optimized with Quanto both run faster and have smaller error rates compared to circuits optimized by the IBM compiler [8].

A wide variety of circuits can benefit from optimization with Quanto, as its automatic approach allows it to quickly find new circuit identities using new gate sets. This allows Quanto to easily adapt to the gate set native to any quantum computer.

Quanto consists of two main components: A circuit identity generator and a quantum circuit optimizer. The circuit identity generator takes a gate set and a circuit size as input and returns a circuit identity database. The quantum circuit optimizer takes the circuit identity database along with a quantum circuit as input and returns the optimized quantum circuit.

2.3.1 A Circuit Identity Generator

The circuit identity generator uses a grid to represent quantum circuits. In the grid, a row corresponds to a layer in the quantum circuit, and the number of rows n corresponds to the depth d of the circuit. A column corresponds to the gates applied to a single qubit in the circuit, and the number of columns m corresponds to the total amount of qubits (the height) in the circuit. Empty grid cells are filled in by identity gates. As an example the circuit in Figure 4 would correspond to the grid:

$$[[X, Y, I], [H, Z, X], [I, H, Z]]$$

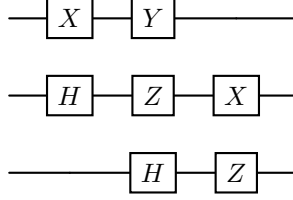


Figure 4: Grid representation as a circuit

In the grid representation, a 2-qubit gate takes up two cells in the same row. The cells do not have to be adjacent, since Quanto does not require the 2-qubit gate to act on two adjacent qubits. A special notation is used to denote which part of the 2-qubit gate is applied to which qubit splitting it into two “subgates”. A CX gate for example is denoted by putting CXC in the column of the control qubit and putting CXT in the column of the target qubit.

Given a gate set G , a height h and a depth d , the generator starts by generating all possible grids of depth d and height h containing only gates from G . For each of the circuits represented by a generated grid, the generator calculates the unitary of the circuit and a fingerprint of the unitary. The fingerprints of the unitaries allow for quick comparisons between different circuits.

The finished circuit identity database consists of two hash tables. The first hash table, fingerprints, maps each of the $d \times h$ grids to their corresponding fingerprint. The second hash table, circuit identities, maps each fingerprint to a list of grids with that fingerprint.

Let g be the number of single-qubit gates in G and t the number of 2-qubit gates in G . Then [8] claims that the number of circuits of depth 1 and height h , that can be made using G is described by equation 3, and the number of circuits of depth d and height h , that be made using G is described by equation 4.

$$S_l = \sum_{r=0}^{\lfloor h/2 \rfloor} \frac{h!}{r!(h-2r)!} g^{h-2r} t^r \quad (3)$$

$$S = S_l^d \quad (4)$$

2.3.2 Tiling

From equations (3) and (4) it is clear that the number of circuits that can be constructed from G explodes as the input circuit size grows. As a result the work of the generator also explodes for larger circuit sizes. At the same time it is necessary to perform expensive multiplications of large matrices to calculate the unitaries of large circuits.

In order to keep the desired circuit size low Quanto uses a tiling method. Larger circuits are split into tiles (subcircuits), and optimizations are performed locally on individual tiles instead of on the whole circuit at once. An example

of a tiled circuit can be found 3.2.1. This way the generator only has to identify circuit identities for circuits the size of a single tile.

2.3.3 The Quantum Circuit Optimizer

Given a quantum circuit C to optimize Quanto starts by splitting C into tiles as described above. Each tile is then checked for *validity*. A valid tile cannot contain any cut-off 2-qubit gates. If a tile contains only one of the qubits acted upon by a 2-qubit gate, it is considered invalid. An invalid tile can be modified into a valid one if the cut-off 2-qubit gate is placed in one of the two outer layers of the tile. The tile is made valid by replacing the cut-off gate with an identity gate.

Quanto modifies all invalid tiles that can be made valid into valid ones. Provided a hash table of fingerprints and a hash table of circuit identities from the generator the optimizer looks up the fingerprint of the valid tiles in the hash table of fingerprints. Next, all circuit identities for each of the valid tiles are found by looking up their fingerprint in the hash table of circuit identities. The optimum circuit identity to replace each tile is picked based on a cost function.

The best circuit identity to replace a certain tile is the one that minimizes the cost function. In this report and in [8] the depth of the circuit is used as the cost function. However, the cost function could also be based on something else like the number of gates or gate errors.

Before each tile can be replaced with the selected circuit identity, the circuit identity must also be checked for validity. Most circuit identities are valid, but if a tile was modified to make it valid, its replacement circuit identity must also be modified in order to be valid. The circuit identity is modified by replacing an identity gate in the appropriate spot with the part of the 2-qubit gate, that was cut-off in the original unmodified tile.

After performing any necessary modifications to the selected circuit identities the tiles are replaced. The entire procedure is repeated a fixed number of times. In [8] 10 times seemed to be sufficient. In the worst case the quantum circuit shrinks by one layer each repetition and the optimizer performs a total of d passes.

Run Time: A single pass of the quantum circuit optimizer on a $d \times h$ circuit considers a total of $O(d \cdot h)$ tiles. The validity of a tile, its fingerprint and replacement circuit identity can all be determined in constant time thanks to the hash tables produced by the generator. As a result the run time of a single optimizer pass is $O(d \cdot h)$.

2.4 Quartz

When the SQuanto project began, we thought Quanto was a very recent quantum optimizer. However, it appears that [8] is a republication of [7]. In between the same authors have published [11], which appear to improve upon some of the concepts from Quanto. Due to how late we discovered this discrepancy, we

have not included the new concepts from Quartz with one exception: We borrow the Quartz method for calculating fingerprints of unitaries. In the Method section of this paper, we will describe the fingerprint method in detail.

3 Method

Our implementation of a quantum circuit optimizer is based on the Quanto optimizer, and developed in the functional language SML. Our optimizer is built using an SML library for working with quantum circuits created by Martin Elsman¹. The library has existing implementations for representing quantum gates, quantum circuits, and complex numbers. Due to the scope of the project, we only implement a "prototype" of the optimizer, for a reduced gate set. Specifically, we don't handle multiple-qubit gates such as control gates, even though Quanto [8] and Quartz [11] handle them. Furthermore, whereas Quanto can optimize any 2-qubit gate set that one can define, our implementation is limited to only work for the 1-qubit gates $\{I, H, X, Y, Z\}$, as these gates are the only ones defined in the SML quantum library defined by Martin Elsman.

Similar to Quanto our implementation is split into 2 parts: A Generator and an Optimizer.

3.1 Generator

The generator operates on a gate set, representing all gates possible in a specific circuit. For example, if a circuit should only consist of H-, X-, and Y- gates, the gate set will be $\{H, X, Y, I\}$. The identity gate I is always included in the gate set, as I is equivalent to doing nothing on a qubit and we use this gate to analyze the depth of a (sub-)circuit, which in our prototype is equivalent to the cost of that (sub-)circuit.

The generator also needs a specific depth and height, which denotes the height and depth of the tiles used for optimizing the circuit.

The generator returns a database, that, given a sub-circuit representation, maps it to the smallest equivalent sub-circuit, without recalculating the unitary matrix. The specifics will be underlined in the following sections

3.1.1 Representing Circuits

For simplicity and efficiency of computation, we use a different representation of quantum circuits as compared to the representation in the SML quantum library.

To work with both representations we implemented functionality for translating our representation to the representation of the existing quantum library and vice versa. This was critical for calculating e.g. fingerprints.

A generated circuit is represented as a tile, where a tile is represented as a list of lists of quantum logic gates. A tile consists of d lists representing the columns

¹<https://github.com/diku-dk/atpl-sml-quantum>

of a circuit/tile, where each column is a list of quantum gates of height h . As an example of our representation, consider the small quantum circuit visualized in figure 5. It will have the following representation in our implementation:

$$[[H, I], [X, Y], [Z, I]]$$

Where $[\cdot]$ denotes a list.

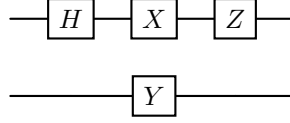


Figure 5: Example of a small quantum circuit.

3.1.2 Tile Generation

To generate all possible tiles, we follow the same approach as in Quanto.

Our generator implementation initially generates all possible tiles of a gate set with height h and depth 1. We implemented a function for this functionality `make_init_columns`, and it works by recursively generating all permutations of list of gates of length h using concatenation.

```
fun make_init_columns (gates, height) : column list =
  case height of
  0 => raise Fail "height must be positive"
  | 1 => map (fn g => [g]) gates
  | n => foldl (fn (gate, res) =>
    (map (fn lst => gate :: lst)
     (make_init_columns (gates, height-1)))
    @ res) [] gates
```

After, if the depth d of the tile should be larger than 1 we recursively concatenate each sub-tile of depth 1, onto each list created. This work grows exponentially on the value of gates as a function of d .

3.1.3 Unitary Calculation

As in Quanto, sub-circuit/tile equivalence is checked by comparing fingerprints of the unitaries of sub-circuits/tiles. In our implementation two sub-circuits/tiles are assumed to be equivalent when their unitaries hash to the same fingerprint.

Computing fingerprints involves computing the unitary of circuits, which is done by computing Kronecker products of the gates in the circuit, which exponentially explodes in the circuit size. However, to generate the fingerprint, we apply the calculated matrix to a state vector. Therefore, we can use the "vec trick" outlined in by Marting Elmsan in an ATPL lecture, to apply the circuit directly to the vector, without calculating the entire huge unitary matrix of

the circuit [3]. This function was implemented in the SML quantum library as the `interp` function. Circumventing generating the huge unitary matrix, might allow bigger tiles to be calculated, and it should decrease the runtime of our generator implementation.

The implementation of the function `interp` only covers a specific number of gates, and therefore we chose only to support the gates $\{I, X, Y, Z, H\}$. However this is straightforward to extend for all 1-qbit gates, but then the `interp` function also needs to be extended.

3.1.4 Generating Fingerprints

To create a fingerprint we use the fingerprint generation method described in [11] by equation (3). It states:

$$\text{FingerPrint}(C) = |\langle \psi_0 | \llbracket C \rrbracket(\vec{p}_0) | \psi_1 \rangle| \quad (5)$$

Where C quantum circuit C over q qubits and m parameters for parameter values $\vec{p} \in \mathbb{R}^m$ and $\llbracket C \rrbracket(\vec{p})$ is a concrete complex matrix of dimension $2^q \times 2^q$ where the gate set uses the parameters specified by \vec{p} . Then $|\psi_0\rangle$, $|\psi_1\rangle$ and \vec{p}_0 are fixed and randomly selected for fingerprint generation. Note, since our implementation only allows the fixed gate set X, Y, H, I, Z , we do not make use of any parameters. Thus $\llbracket C \rrbracket(\vec{p})$ should always evaluate to the same complex matrix no matter the choice of parameters (\vec{p}), therefore generation of fingerprints simplify to:

$$\text{FingerPrint}(C) = |\langle \psi_0 | \llbracket C \rrbracket | \psi_1 \rangle| \quad (6)$$

For randomly generated quantum states $|\psi_0\rangle$, $|\psi_1\rangle$.

3.1.5 Database Generation

The database generation, like in Quanto, is done in 2 parts. First, for any possible tile C we create a hash map that maps

$$C \mapsto \text{FingerPrint}(C)$$

using equation (6). After, we generate a second hash map that maps any possible fingerprint to an equivalent circuit C' that minimizes the cost function:

$$\text{FingerPrint}(C) \mapsto C'$$

where $\text{FingerPrint}(C) = \text{FingerPrint}(C')$. This way, when optimizing, we never need to re-calculate the fingerprints or unitaries of circuits, and can instead look up the fingerprint of circuits immediately to find the optimal circuit identity of a circuit.

In our database, we only store the best circuit identity based on the cost function. The cost function of the circuit is defined as its depth. We calculate the depth as the total depth of the circuit, minus the number of columns with

identity columns. Note, the best circuit is the one that minimizes the cost function.

This contrasts the Quanto implementation, which maintains all circuit identities and then the optimizer chooses the optimal circuit identity during optimization based on cost. We found no advantage in this method and believe storing only the optimal identity saves space and reduces the look-up time in the optimizer.

3.2 Optimizer

As input our optimizer implementation takes a circuit to optimize, a gate set, a tile height, a tile depth, and a number of optimization passes. Initially, the optimizer generates a database followed by performing the specified number of optimization passes. An optimization pass consists of first, partitioning the input circuit into tiles. Then the generated tiles are optimized using the database. The optimized tiles are then reverted back into a circuit, and the depth of the circuit is reduced if possible.

3.2.1 Tiling

When splitting the circuit up into tiles, we choose to only create non-overlapping tiles. To ensure this, we have to "pad" some circuits with identity gates.

We start by splitting the "columns" of the circuit into the desired height, by recursively taking elements, and then padding if necessary. This is done on the whole circuit and returns all the split columns as a list of circuits. Then we split the circuit lists by depth. An example of tiling is added below

$$\begin{pmatrix} H & X & Y \\ H & X & Y \\ H & X & Y \end{pmatrix} \Rightarrow \begin{pmatrix} \begin{matrix} H & X \\ H & X \end{matrix} & \begin{matrix} Y & \mathbf{I} \\ Y & \mathbf{I} \end{matrix} \\ \begin{matrix} H & X \\ \mathbf{I} & \mathbf{I} \end{matrix} & \begin{matrix} Y & \mathbf{I} \\ \mathbf{I} & \mathbf{I} \end{matrix} \end{pmatrix} \quad (7)$$

Which in our implementation corresponds to the following representation:

$$[[H, H, H], [X, X, X], [Y, Y, Y]] \Rightarrow \quad (8)$$

$$[[[H, H], [X, X]], [[Y, Y], [\mathbf{I}, \mathbf{I}]], [[H, \mathbf{I}], [X, \mathbf{I}]], [[Y, \mathbf{I}], [\mathbf{I}, \mathbf{I}]]] \quad (9)$$

3.2.2 Optimizing

To optimize the tiles, we go through each tile, and find their circuit identity in the database. Since the database only contains the cost-optimal circuit identity for each circuit, we always replace each tile, with the optimal circuit. There cannot be a case, where a tile does not exist in the database, as the generator constructs all possible circuits for a gate set. Therefore in the worst case, the "best tile" to replace a tile is either the tile itself or a tile of equivalent depth.

After optimizing the individual tiles, we assemble them back into a single circuit. This is done by using the original height and depth of the circuit to

flatten the list of tiles, redistributing them into the correct layout of the original circuit. We then “prune” the circuit by pushing all the identity gates, to the far right, to remove all identity columns. This gives a new, possibly reduced circuit, that we optimize again, looping `iterations` number of times.

4 Results

4.1 Validation

During the project, we made several small unit tests, to validate that the implementation behaved as expected.

This was done by creating pretty printers, calling the function on a pre-made input, inspecting the output, and confirming it returns what we would expect.

Thus we have made many small black box tests, and believe this to be sufficient for the extent of the project.

All of the unit tests pass for our optimizer.

The tests can be found in `main.sml` file, in the `optimizer` folder, and can be run by compiling its `.mlb` and then running the executable

```
mlkit main.mlb
./run
```

During testing we also verified we could reduce circuits, for example, for a simple but deep circuit of repeating X, Y, Z gates, with an uneven depth it reduced it to a depth 1 circuit of $[X, Y, Z]$

$$\begin{pmatrix} X & X & X & \dots & X \\ Y & Y & Y & \dots & Y \\ Z & Z & Z & \dots & Z \end{pmatrix} \Rightarrow \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}$$

As we would expect. Similarly we also verified that a deep circuit of repeating gates of even depth reduced to a vector of identity gates.

4.2 Experimental Results

To test the impact of the gate set, tile size and circuit size, we developed a small script for benchmarking out generator.

First, we benchmarked the runtime impact of the size of the gate set and the tile size, with the following results. The results can be viewed in table 1. We did not experiment with tiles larger than (2×3) as our implementation would achieve segfault for most gate sets when tile size exceeded (2×3) .

We also benchmarked the runtime impact of increasing the size of the circuit optimized, for a fixed gate set and tile size. The results can be seen in figure 2. Note, the run times listed includes database generation.

Gate Set	Tile Size		
	(1, 2)	(2, 2)	(2, 3)
$[I, X]$.0271 s	.0263 s	.0363 s
$[I, X, H]$.0307 s	.0273 s	.0520 s
$[I, X, H, Y, Z]$.0290 s	.0328 s	.6463 s

Table 1: Time for generating database in our implementation. All units are in seconds.

Circuit size	3 x 3	5 x 7	10 x 20
Runtime	.0383 s	.0458 s	.0346 s

Table 2: Runtime for optimizing 3 different circuits of sizes $((3 \times 3), (5 \times 7), (10 \times 20))$ where circuits were arbitrarily generated from the gate set $[I, X, H]$ and the tile size (2×2) . All units are in seconds, and optimization always uses 3 passes for comparability.

5 Discussion

5.1 Implementation

5.1.1 Multi-Qubit Gates

Due to the limited time for the project, our SML implementation of Quanto only handles a small set of gates. These gates deliberately do not include any 2-qubit gates, and simplifies the optimizer. Without 2-qubit gates, it is possible to guarantee that all tiles are valid regardless of the circuit, its size and the tile size. This allows the optimizer to skip checking for valid tiles.

The simplification severely impacts the kinds of circuits SQuanto can optimize. Without support for 2-qubit gates, SQuanto cannot optimize on any known universal gate set for quantum computing [10].

Given more time to work on SQuanto one of the first steps should be to extend the current implementation to handle at least one 2-qubit gate. If SQuanto had support for one 2-qubit gate, it should be simple to extend to other 2-qubit gates the same way it is currently easy to add support for new single-qubit gates. Going beyond Quanto extending SQuanto to support 3-qubit gate sets would be interesting as the 3-qubit Tofolli gate is universal for classical computing [10]. Further extension to 4-qubit gates would probably follow easily from the 2- and 3-qubit approaches, but be uninteresting due to the rarity of 4-qubit gates.

To handle 2- and 3-qubit gates SQuanto needs a way to represent them that is compatible with the existing, `gate`, `column` and `tile` types. As the `tile` type currently only allows gates to take up one cell, the most straightforward solution would be to break multi-qubit gates into parts the way Quanto does. For example, the CX gate becomes the “subgates” CXC and CXT. This approach could be extended to 3-qubit gates by, for example, breaking the Tofolli gate into the “subgates” CTC1, CTC2 and CTX.

This solution poses several challenges, which would have to be solved: Since we cannot apply only parts of a multi-qubit gate how do we ensure when generating a tile, that its columns contain either none or all parts of a multi-qubit gate? If a column contains multiple multi-qubit gates of the same type how do we denote which set of subgates belong to which multi-qubit gate? Furthermore tiling in the optimizer has the potential to cut off multi-gates creating invalid tiles. The optimizer structure would need to be updated with a way to detect invalid tiles, and potentially modify them into valid tiles, as discussed in section 2.3.1.

5.1.2 Gate Set and Tile Size

When generating our identity circuits, we get an exponential explosion of operations. Just to generate all possible circuits, before calculating their fingerprints, we would get

$$|G|^{H \cdot D}$$

operations. This is due to each gate being a potential candidate for each of the $H \cdot D$ spots in the circuit. This scales exponentially in the number of q-bits in a tile (H), the number of gates in a tile (D) on the number of gates in the gate set.

This leads to our generator seg-faulting for relatively small tiles, on bigger to medium gate sets. If we have a meaningful number of gates, tile sizes of (4×4) segfaults. We suspect this is due in part to the recursive structure of our circuit generation, which will run out of stack space, and/or in the computation of the fingerprints

To mitigate the size of the calculation, we use the `interp` function provided by Elsmann, to avoid calculating the tensor product, but we still seg-fault quite early.

To mitigate the recursion depth, one could try to implement tail recursion, and would be interesting to test, in a future project.

Another approach could be to take inspiration from the Quartz project [11], which limits the number of generated, by using a notion of *equivalent circuit classes* (ECCs), as sets of functional equivalent circuits. To avoid generating all possible circuits, they generate the ECCs from smaller to larger circuits, and for each ECC selects a representative circuit, to expand the ECC.

But overall the generator can generate the database for non-trivial tiles, such as 3×3 tiles, on a realistic gate set.

5.1.3 Fingerprints

As mentioned earlier, we used the same method as Quartz to generate fingerprints, see equation (6) [11]. During testing, we noted that this approach may be problematic for the following reasons.

The fingerprint generation method described in equation (6) ensures that a circuit C will have the same fingerprint as any other circuit C' that is equivalent

to C apart from a global phase. As mentioned previously, for a quantum circuit C , the global phase of C does not affect the measurement probability of the output state. However, the fingerprint generation described by Quartz determines that sub-circuits, that are equivalent apart from a phase, get the same fingerprint because the fingerprint is computed as the magnitude of a complex number. In our implementation, we assume that sub-circuits are equivalent based only on the fingerprint. Thus we will substitute sub-circuits that are equivalent apart from their global phase.

While it is not problematic to change the phase of an entire circuit, we cannot ensure that changing the phase of sub-circuits during optimization is also not problematic. Thus we could have improved fingerprint generation to ensure sub-circuits, which vary by a global phase, do not hash to the same fingerprint. A possible method for achieving this would be to hash the real and imaginary part in $\langle \psi_0 | \llbracket C \rrbracket | \psi_1 \rangle$ and then combine these 2 numbers into 1 hash.

5.1.4 Optimizer

Our optimizer differs from the Quanto implementation [8], since we do not have overlapping tiles, and do not update a tile in place. As mentioned previously this is a consequence of the functional nature of SML. We could have created a monad for updating tiles in place but decided to pad the circuit instead when the circuit size was not a multiple of the tile size, allowing us to always have non-overlapping tiles. This has the benefit that we can evaluate each tile separately. Furthermore, this presents some unique opportunities discussed in the future work section.

The optimizer seems to apply substitutions correctly and reduces the circuit between optimizations. This is based on the series of unit tests of our implementation created in `main.sml` all passing. Notably, we optimized a small set of small circuits using our implementation in `main.sml` and verified manually, that the optimized circuit was correct.

5.2 Results

Observing the results in table 1 we note the results generally follow the trend we expected. We observe the largest runtime of 0.643 seconds for the combination of the largest gate set $[I, X, H, Y, Z]$ and the largest tile size $(2, 3)$. The smallest runtime achieved is 0.263 seconds for the second smallest tile $(2, 2)$ and smallest gate set $[I, X]$. But the runtime for the smallest gate set and smallest tile only differ from the smallest result by 0.0008 seconds, so this difference is negligible.

Consulting the table of results 1 it is also clear that the runtime increases exponentially as expected in both the gate set size and tile size. This is evident from the runtime for the largest gate set $[I, X, H, Y, Z]$ and largest tile size $(2, 3)$ in table 1 is 0.6463 seconds which is more than ten times larger than the second largest runtime of 0.0520 seconds for the second largest gate set $[I, X, H]$ and largest tile size $(2, 3)$. This is notable because increasing either tile size or gate set in the table 1, only gives small increases in run time, less than doubling

it. Then for the big tile size and gate set, the run time explodes to 10X the runtime. Thus it is clear that the runtime we observe is a consequence of the exponential blow-up in the generation phase of our database.

The runtimes listed in table 2 also follow our expectations. When using a fixed tile size and gate set for optimizing 3 different-sized circuits run-time does not seem to depend on circuit size. Indeed, the lowest runtime is surprisingly achieved for the largest circuit size. This is likely caused by the generation of the database dominating the actual runtime, with the runtime of the optimization passes being negligible in comparison. However, the circuits chosen for benchmarking might also have been too small to observe the impact of optimization passes on runtime. However, increasing circuit size does not drastically increase runtime as quickly as tile size and gate set size do.

6 Further Work

The work in Quanto [8] and Quartz [11] did not consider parallelism for performance increase. However in our estimation both the generation of tiles and optimization of circuits, have opportunity for parallelization.

6.1 Parallelization of Generation

In the generation phase, we discovered that all fingerprints can be generated independently. Since a fingerprint needs to be generated either by the "vec-trick" outlined in the methods section, or by computing the unitary matrix using Kronecker products, this step is very computation-heavy and is a prime candidate for parallelization.

We believe that the data parallel language Futhark [1], would be ideal for this parallelization since Futhark has shown great results in generating GPU-optimized code for example CUDA. However Futhark has a restriction of only handling regular parallelism, so there could be difficulties in handling different-sized gates and or circuits. However, due to our tiling, we ensure that all circuits that need to be calculated are guaranteed to have the same size and are therefore regular. In our prototype, we only handle a subset of 1-qubit gates, so this would not change the size of tiles, and even 2-bit gates, can be "transformed" to 1-qubit gates as mentioned previously and shown in [8]. This ensures that all circuits become regular, which is ideal for Futhark to generate efficient GPU code. Furthermore, it could also be possible to generate the Futhark code, from the SML implementation, using a monad and pretty printer, as outlined by Martin Elsman in slide 11 in "Synthesizing Futhark Quantum Circuit Simulation" [2].

To estimate the speed up potential we note that the generator needs to calculate the following number of unitaries:

$$|G|^{h \cdot d}$$

where $|G|$ is the size of the gate set given to the optimizer, h is the height or number of qubits in a tile, and d is the depth of a tile. This implies we could utilize a big part of GPU for relatively small circuits, and potentially achieve a huge speedup, in the order of the number of processors or threads. This would then generate all the needed fingerprints, which we could use to create the database in the generator.

The circuit generation however seems to be a bit more tricky, due to the recursive structure of our implementation, which is not inherently appropriate for parallelization. To handle this we note that the resulting circuits correspond to an n -ary cartesian product [9]. So the tiles correspond to

$$\mathbf{G}^n = \underbrace{\mathbf{G} \times \mathbf{G} \times \dots \times \mathbf{G}}_n = \{(g_1, \dots, g_n) \mid g_i \in \mathbf{G} \text{ for every } i \in \{1, \dots, n\}\}$$

where \mathbf{G} is the gate-set and $n = h \cdot d$, with $h = \text{height}$ and $d = \text{depth}$.

Each tuple is a unique ordering, that can be converted to a circuit. This might give possibilities for parallelization. It is however not strictly associative, so a simple map-reduce in Futhark, would not suffice. But it might be possible by modifying the operation, such that only one tuple is maintained.

There is other work, that suggests that an n -ary cartesian product is possible to parallelize, such as the work by Hu Chen et.al [4] that parallelizes the product for microprocessors, and shows a version for the GPU.

We believe that parallelizing the generation would be worthwhile to research in a future project, with a small prototype in Futhark. Due to the scope of the project, we did not have the time to implement it ourselves.

6.2 Parallelization of the Optimizer

We believe there would also be opportunities for parallelization in the optimizer since we tile each circuit, and then look up and substitute each tile independently. Since the tiles in our implementation are non-overlapping, unlike in Quanto, the substitution is inherently parallel. This might give a small speed-up, depending on the number of tiles in our circuit.

The biggest limitation is whether the conversion of the circuit to tiles and vice-versa can be parallelized, as right now they rely on a recursive structure. However, tiling is a common technique for matrix multiplication ??, therefore tiling in parallel should be achievable.

In the optimizer, we would only be able to parallelize the optimization pass, and the loop of optimization passes would need to run sequentially, due to the dependency of the output.

6.3 Support for Multi-Qubit Gates

As mentioned in the discussion an obvious improvement for our prototype would be to implement support for 2-bit gates such as C-NOT. This would mean extending the generator to split the gates into target and control gates, being

able to handle cut-off control gates, and discarding invalid tiles. All this is a functionality of Quanto [8], but was unfortunately not realized in our prototype.

7 Conclusion

In this project, we have analyzed the Quanto optimizer described in the article "Quanto: optimizing quantum circuits with automatic generation of circuit identities" by Jessica Pointing et.al [8], and implemented a prototype in the functional language SML, based on the quantum library created by Martin Elsmann, which we have named SQuanto. Our prototype only handles a subset of gates, and only 1-bit gates. Our prototype can successfully generate and discover circuit identities for a limited gate set and small tile sizes, and store them in a database. We can then use the database to optimize quantum circuits. Through benchmarking, we discovered that the runtime of our generator increased exponentially in the size of the tile and gate set used in the generation phase, while the size of the circuits to optimize, has a negligible effect. During the analysis of Quanto and the implementation of our prototype, we found opportunities for performance increase in the generation and optimization part, not mentioned in Quanto. However, time constraints did not allow us to examine this in detail.

We believe we can parallelize the generation phase since the creation of identity circuits is independent from each other, and that we can also parallelize the optimization phase since our implementation optimizes circuits using non-overlapping tiles.

Further work on this project would include handling more gates, especially 2-qubit gates, implementing part of the generation in a data-parallel language like Futhark, and examining the impact on runtime.

References

- [1] Why futhark? <https://futhark-lang.org/>. [Online; accessed 19-01-2025].
- [2] 2024. slide 11.
- [3] December 3, 2024. slide 17-20.
- [4] Hu Chen, Fei Teng, and Yu Wang. Parallel computation for n-ary cartesian product on modern microprocessors. In *2019 IEEE 8th Joint International Information Technology and Artificial Intelligence Conference (ITAIC)*, pages 750–753, 2019.
- [5] Gavin E. Crooks. Quantum gates. "https://threeplusone.com/pubs/on_gates.pdf", 2024. [Online; accessed 18-01-2025].
- [6] Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. A verified optimizer for quantum circuits. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–29, January 2021.

- [7] Jessica Pointing, Oded Padon, Zhihao Jia, Henry Ma, Auguste Hirth, Jens Palsberg, and Alex Aiken. Quanto: Optimizing quantum circuits with automatic generation of circuit identities, 2021.
- [8] Jessica Pointing, Oded Padon, Zhihao Jia, Henry Ma, Auguste Hirth, Jens Palsberg, and Alex Aiken. Quanto: optimizing quantum circuits with automatic generation of circuit identities. *Quantum Science and Technology*, 9(4):045009, jul 2024.
- [9] Wikipedia contributors. Cartesian product. [Online; accessed 18-01-2025].
- [10] Wikipedia contributors. Quantum logic gate. [Online; accessed 18-01-2025].
- [11] Mingkuan Xu, Zikun Li, Oded Padon, Sina Lin, Jessica Pointing, Auguste Hirth, Henry Ma, Jens Palsberg, Alex Aiken, Umut A. Acar, and Zhihao Jia. Quartz: Superoptimization of quantum circuits (extended version), 2022.

8 Appendix

8.1 Generator implementation

```
structure QGenerator : QGENERATOR = struct
  datatype gate = I | X | Y | Z | H (*For now only these 1-qubit gates are handled*)

  type ('key, 'val) hash_map = ('key, 'val) Table.table

  type column = gate list
  type tile = column list
  type fingerprint = real
  type depth = int
  type height = int

  type circuit_identities = (tile, fingerprint) hash_map

  type fingerprints = (fingerprint, (tile * depth)) hash_map (* should also store the cost of the circuit *)
  type database = circuit_identities * fingerprints

  fun pow2 (n : int) : int =
    case n of
      0 => 1
    | _ => 2 * (pow2 (n-1))

  fun hash file = raise Fail "not implemented"

  fun gate_to_circuit_gate I = Circuit.I
    | gate_to_circuit_gate X = Circuit.X
    | gate_to_circuit_gate Y = Circuit.Y
    | gate_to_circuit_gate Z = Circuit.Z
    | gate_to_circuit_gate H = Circuit.H

  fun circuit_gate_to_gate (t : Circuit.t) =
    case t of
      Circuit.I => I
    | Circuit.X => X
    | Circuit.Y => Y
    | Circuit.Z => Z
    | Circuit.H => H
    | _ => raise Fail "Gate not supported"

  fun make_init_columns (gates, height) : column list =
    case height of
      0 => raise Fail "height must be positive"
```

```

    | 1 => map (fn g => [g]) gates
    | n => foldl (fn (gate, res) => (map (fn list => gate :: list) (make_init_columns (gate, height)))) (make_init_columns (gate, height))

(* wait we need to keep all depths up to depth *)
fun make_tiles (gates, height, depth) : tile list =
  let fun make_tiles1 (d : depth, init_col : column list) : tile list =
    case d of
    0 => raise Fail "depth must be positive"
    | 1 => map (fn t => [t]) init_col
    | n => let val prev_step = make_tiles1 (d-1, init_col)
          in (foldl (fn (col, acc) => (map (fn tile => col :: tile) prev_step) @ acc)) prev_step
          end
    in make_tiles1 (depth, make_init_columns (gates, height))
  end

fun cost (tile : tile) : depth =
  foldl (fn (col, acc) => if (List.all (fn g => g = I) col) then acc else acc + 1) 0 tile

fun column_to_circuit (col : column) : Circuit.t =
  case col of
  [] => raise Fail "Empty Columns not allowed\n"
  | g::gs => foldl (fn (g0, g_acc) => Circuit.Tensor(g_acc, (gate_to_circuit_gate g0))) (Circuit.I) gs

fun tile_to_circuit (tile : tile) : Circuit.t =
  case tile of
  [] => raise Fail "Empty Tiles not allowed\n"
  | [[]] => raise Fail "Empty Tiles not allowed\n"
  | col::cols => foldl (fn (c0, c_acc) => Circuit.Seq(c_acc, (column_to_circuit c0))) (Circuit.I) cols

fun tile_to_matrix (tile : tile) : Semantics.mat =
  Semantics.sem (tile_to_circuit tile)

(* Code for Fingerprinting starts here *)

(* Generate random complex numbers *)
fun gen_random_complex (rand_gen : Random.generator) : Complex.complex =
  let val a = Random.random rand_gen
    val b = Random.random rand_gen
  in Complex.mk (a, b) end

(*For generating psi_1, psi_0 for fingerprints*)
(* n=2^(num qubits in tile) *)
fun gen_random_complex_list (rand_gen : Random.generator, n : int) =
  let val alist = Random.randomlist (n, rand_gen)
    val blist = Random.randomlist (n, rand_gen)
  in (alist, blist) end

```



```

        in ListPair.map Complex.mk (alist, blist) end

(* Normalize vector, s.t. complex vector is a complex state *)
fun normalize_complex_list (comp_list : Complex.complex list) : Complex.complex list =
  let val norm = Math.sqrt (foldl (fn (elm, acc) => Complex.re (Complex.* (elm, Complex.fromRe norm))) 0 comp_list)
  in map (fn c_val => Complex./ (c_val, Complex.fromRe norm)) comp_list end

(* *)
fun gen_fingerprint (tile : tile, psi_0 : Complex.complex list, psi_1 : Complex.complex list) : fingerprint =
  let val psi_1_state = Vector.fromList psi_1
      val res_1 = Semantics.interp (tile_to_circuit tile) psi_1_state
      fun complex_dot_product (avec : Complex.complex list, bvec : Complex.complex list) : Complex.complex =
        case (avec, bvec) of
          ([], []) => Complex.fromInt 0
        | ([], _) => raise Fail "Vectors not same length"
        | (_, []) => raise Fail "Vectors not same length"
        | (a::atail, b::btail) =>
            Complex.+ (Complex.* (a, b), complex_dot_product(atail, btail))
      in Complex.mag (complex_dot_product (psi_0, Vector.foldl(fn (elm, acc) => elm :: acc) [] psi_1_state))
      end

fun gate_to_num (gate : gate) : string =
  case gate of
    I => "1"
  | X => "2"
  | Y => "3"
  | Z => "4"
  | H => "5"

fun tile_to_word (tile : tile) : word =
  let val str = (
      Word.fromString (
        foldr (fn (col, acc) => (
          foldr (fn (gate, acc2) => gate_to_num (gate) ^ acc2) "" col) ^ acc)
        "" tile))
  in
    case str of
      NONE => raise Fail "Could not generate string"
    | SOME(s) => s
  end

fun fingerprint_to_word (fp : fingerprint) : word =
  Word.fromInt (Real.floor (fp / 0.0000000000000002))

```

```

fun fingerprint_equality (f0 : fingerprint, f1 : fingerprint) : bool =
  Real.abs (f0-f1) < 0.000000000000001 (* 10^-15 as in Quartz paper *)

(* Make databases *)
fun make_identities (tlst : tile list, psi0 : Complex.complex list, psi1 : Complex.complex list) : identities_db =
  let val identities_db = Table.new {hash= tile_to_word, eq= fn (t0, t1) => t0 = t1}
      val _ = map (fn tile => Table.add (tile, gen_fingerprint(tile, psi0, psi1), identities_db)) tlst
  in
    identities_db
  end

fun update_fingerprints_db (fp_db : fingerprints, cur_tile : tile, cur_fp : fingerprint) : fingerprints =
  let val cost_cur_tile = cost cur_tile
  in case Table.lookup fp_db cur_tile of
      NONE => Table.add (cur_tile, (cur_fp, cost_cur_tile), fp_db)
    | SOME (old_tile, old_cost) =>
        if cost_cur_tile < old_cost then
          Table.add (cur_tile, (cur_fp, cost_cur_tile), fp_db)
        else
          ()
      end
  end

fun make_fingerprints(circuit_identities : circuit_identities) : fingerprints =
  let val fingerprints_db = Table.new {hash= fingerprint_to_word, eq= fingerprint_equality}
      val _ = Table.Map (fn (tile, fp) => update_fingerprints_db(fingerprints_db, tile, fp)) circuit_identities
  in
    fingerprints_db
  end

fun generator (gates : gate list, height : height, depth : depth) : database =
  let val tiles = make_tiles (gates, height, depth)
      val rand_gen = Random.newgen ()
      val n = pow2 height
      val psi0 = normalize_complex_list (gen_random_complex_list (rand_gen, n))
      val psi1 = normalize_complex_list (gen_random_complex_list (rand_gen, n))
      val circuit_identities = make_identities(tiles, psi0, psi1)
      val fingerprints = make_fingerprints(circuit_identities)
  in
    (circuit_identities, fingerprints)
  end

(* REALM OF THE PP *)
fun pp_list (list : 'a list, pp_a : 'a -> string) =
  let fun pp_list1 (lst : 'a list) =
        case lst of

```

```

        [] => ""
        | [elm] => pp_a elm
        | elm::tail => (pp_a elm) ^ ", " ^ (pp_list1 tail)
    in "[" ^ (pp_list1 list) ^ "]"
end

fun pp_gate gate = Circuit.pp (gate_to_circuit_gate gate)

fun pp_column (col : column) =
    pp_list (col, pp_gate)

(* Note, column list = tile *)
fun pp_column_list (columns : column list) = pp_list (columns, pp_column)

fun pp_tile (t : tile) = pp_column_list t

fun pp_tile_list (tiles : tile list) = pp_list (tiles, pp_tile)

fun pp_tuple ((a, b) : 'a * 'b, pp_a : 'a -> string, pp_b : 'b -> string) : string =
    "(" ^ pp_a a ^ ", " ^ pp_b b ^ ")"

fun pp_database ((circuit_identities, fps) : database) : string =
    let val circuit_id_str = pp_list (Table.list circuit_identities,
                                      fn tp => pp_tuple (tp, pp_tile,
                                                           Real.toString))
        val fp_str = pp_list (Table.list fps,
                              fn tp => pp_tuple (tp, Real.toString,
                                                  fn t =>
                                                    pp_tuple (t,
                                                                pp_tile,
                                                                Int.toString)))
    in "Circuit identities:\n" ^ circuit_id_str ^ "\n\nfingerprints:\n" ^ fp_str
    end
end

```

8.2 Optimizer

```

structure Optimizer : OPTIMIZER = struct

    structure QG = QGenerator

    fun get_fingerprint (tile : QG.tile, c_ids : QG.circuit_identities) : QG.fingerprint =
        case Table.lookup c_ids tile of
            NONE => raise Fail "Missing Tile in Circuit Identities"
        | SOME fp => fp

```

```

fun get_best_tile (fp : QG.fingerprint, fps : QG.fingerprints) : QG.tile =
  case Table.lookup fps fp of
    NONE => raise Fail "Missing Fingerprint in Fingerprints"
  | SOME (tile, _) => tile

fun best_tile (tile : QG.tile, (circuit_ids : QG.circuit_identities,
                                fps : QG.fingerprints) : QG.database) : QG.tile =
  let val fp = get_fingerprint (tile, circuit_ids)
  in get_best_tile (fp, fps) end

fun rep_identity_gate (height : QG.height) : QG.gate list =
  case height of
    0 => []
  | n => QG.I :: rep_identity_gate (n - 1)

fun split_column (col : QG.column,
                  og_height : QG.height,
                  grid_height : QG.height) : QG.column list =
  if og_height < grid_height then
    if og_height < 1 then
      []
    else
      [col @ rep_identity_gate (grid_height - og_height)]
  else
    List.take (col, grid_height) :: split_column (List.drop (col, grid_height),
                                                    og_height - grid_height,
                                                    grid_height)

fun rep_col (col : QG.column, depth : QG.depth) : QG.column list =
  case depth of
    0 => []
  | n => col :: rep_col (col, depth - 1)

fun split_tile_depth (tile : QG.tile, (* Entire circuit, represented as a big tile *)
                      height : QG.height, (* Tile height *)
                      og_depth : QG.depth, (* Depth of entire circuit *)
                      grid_depth : QG.depth) : QG.tile list = (* Tile depth *)
  if og_depth < grid_depth then
    if og_depth < 1 then
      []
    else
      [tile @ rep_col (rep_identity_gate (height), grid_depth - og_depth)]
  else
    List.take (tile, grid_depth) :: split_tile_depth (List.drop (tile, grid_depth),
                                                         height,
                                                         grid_depth)

```

```

og_depth - grid_depth,
grid_depth)

fun split_all_cols (tile : QG.tile,
                    height: QG.height) : QG.column list list =
  List.map (fn col => (split_column (col, List.length col, height))) tile

fun create_depth_lists (split_columns : QG.tile list) : QG.tile list =
  let val transposed_cols = Matrix.listlist (Matrix.transpose (Matrix.fromListList split_
  in transposed_cols end

fun split_all_depths (depth_lists : QG.tile list, grid_depth : QG.depth) : QG.tile list =
  let val og_depth = List.length (List.hd depth_lists)
      val height = List.length (List.hd (List.hd depth_lists))
  in
    List.foldl (fn (tile, acc) => acc @ split_tile_depth (tile, height, og_depth, grid_
  end

fun find_optimal_tile (tile : QG.tile, db : QG.database) : QG.tile =
  let val (ci_db, fp_db) = db
      val tile_cost = QG.cost tile
  in
    case Table.lookup ci_db tile of
      NONE => raise Fail "Incorrect generation of circuit identities"
    | SOME fp =>
        case Table.lookup fp_db fp of
          NONE => raise Fail "Incorrect fingerprint database generation"
        | SOME (t_new, depth) =>
            t_new
    end

fun optimize_tile_partition (c_tiles : QG.tile list, db : QG.database) : QG.tile list =
  List.map (fn tile => find_optimal_tile (tile, db)) c_tiles

(* Create list of non-overlapping tiles of dimension tile_height x tile_depth in circuit
fun circuit_to_tile_partition (circuit : QG.tile, tile_height : QG.height, tile_depth : QG
  let val column_split = split_all_cols (circuit, tile_height)
      val row_split = create_depth_lists column_split
  in split_all_depths (row_split, tile_depth)
  end

fun move_Is_right (column : QG.column) : QG.column =
  List.foldr (fn (gate, acc) =>
    case gate of

```

```

        QG.I => QG.I :: acc
      | t => acc @ [t]
    ) [] column

  (* Gather I's at end of circuit *)
fun move_Is_in_circuit (circuit : QG.tile) : QG.tile =
  let val circuit_transpose = Matrix.listlist (Matrix.transpose (Matrix.fromListList circuit))
      val circuit_Is_moved_right = List.map (fn col => move_Is_right col) circuit_transpose
  in Matrix.listlist (Matrix.transpose (Matrix.fromListList circuit_Is_moved_right))
  end

  (* Idea: to improve further, move all gates as far left as possible, and all I's to the far right *)
fun remove_I_columns (circuit : QG.tile) : QG.tile =
  let val circuit_Is_moved_right = move_Is_in_circuit circuit
      val column_height = List.length (List.hd circuit_Is_moved_right)
      val circuit_Is_removed =
        List.foldl (fn (col, acc) =>
          if List.all (fn gate => gate = QG.I) col then
            acc
          else
            col :: acc
        ) [] circuit_Is_moved_right
  in
    case circuit_Is_removed of
    [] => [List.tabulate (column_height, fn i => QG.I)]
    | circuit => circuit
  end

  (* Gather all columns in column cur_column from tile partitioning *)
fun helper (columns : QG.column list, num_columns : int, cur_column : int) : QG.column list =
  case columns of
  [] => []
  | _ =>
    if num_columns <= List.length columns then
      let val row = List.take (columns, num_columns)
      in
        if cur_column < List.length (row) then
          if num_columns <= List.length columns then
            (List.nth (row, cur_column)) :: (helper (List.drop (columns, num_columns), num_columns, cur_column + 1))
          else
            raise Fail "Cannot drop more columns than number of existing columns."
          else raise Fail "Try to take an entry from a list that does not exist."
        end
      end
    else
      raise Fail ("Length of columns= " ^ (Int.toString (List.length columns)) ^ ", num_columns= " ^ (Int.toString num_columns))

```

```

fun tile_partition_to_circuit (tiles : QG.tile list, original_height : QG.height, original_depth : QG.depth)
  let val num_horizontal_tiles = original_height div (List.length (List.hd tiles)) (* Show this *)
  val flatten_tiles = List.concat tiles
  in
    List.tabulate (original_depth,
      fn col_idx =>
        List.concat(helper(flatten_tiles, original_depth, col_idx))
      )
  end

(* A single optimization pass of a circuit*)

fun optimization_pass (circuit : QG.tile, tile_db : QG.database, tile_height : QG.height, tile_depth : QG.depth)
  let
    val og_height = List.length (List.hd circuit)
    val og_depth = List.length circuit
    val circuit_tile_partition = circuit_to_tile_partition (circuit, tile_height, tile_depth)
    val optimized_tile_partition = optimize_tile_partition (circuit_tile_partition, tile_height, tile_depth)
    val optimized_circuit =
      tile_partition_to_circuit (
        optimized_tile_partition
        , og_height + ((tile_height - (og_height mod tile_height)) mod tile_height)
        , og_depth + ((tile_depth - ((og_depth mod tile_depth))) mod tile_depth))
    val optimized_circuit_column_padding_removed =
      List.map (fn col => List.take (col, og_height)) optimized_circuit
    val optimized_circuit_I_cols_removed = remove_I_columns (optimized_circuit_column_padding_removed)
  in optimized_circuit_I_cols_removed
  end

fun loop (circuit : QG.tile, db : QG.database, tile_height : QG.height, tile_depth : QG.depth, iterations : int)
  case iterations of
    0 => circuit
  | _ =>
    loop(optimization_pass(circuit, db, tile_height, tile_depth), db, tile_height, tile_depth, iterations - 1)

fun optimize_circuit (circuit : QG.tile, gate_set : QG.gate list, iterations : int, tile_height : QG.height, tile_depth : QG.depth)
  let val db = QG.generator(gate_set, tile_height, tile_depth)
  in loop (circuit, db, tile_height, tile_depth, iterations) end

end

```

8.3 Unit tests

```

open QGenerator Random Optimizer

fun run a =

```

```

(print "Run all tests: \n";
print "test 1: make_init_columns ([X, Y], 2)\n";
print ("Expected: " ^ (pp_column_list [[X, X], [Y, Y], [X, Y], [Y, X]]) ^ "\n");
print ("Result: " ^ (pp_column_list (make_init_columns ([X, Y], 2))) ^ "\n\n");
print "test 2: pp_tile_lst [[[X, X], [I, I]], [[I, X], [X, I]]] \n";
print "Expected: [[[X, X], [I, I]], [[I, X], [X, I]]] \n";
print ("Result: " ^ (pp_tile_list [[[X, X], [I, I]], [[I, X], [X, I]]]) ^ "\n\n");
print "test 3: make_tiles ([I, X], 2, 2)\n";
print "Expected: [[[I I], [I I]], [[I I], [I X]], [[I I], [X I]], [[I I], [X X]], [[I X], [X I]]] \n";
print ("Result: " ^ (pp_tile_list (make_tiles ([I, X], 2, 2))) ^ "\n\n");
print "test 4: print cost of tile\n";
print ("Result: " ^ Int.toString (cost [[X, I], [I, I], [I, X]]) ^ "\n\n");
print "test 5: convert tile to unitary matrix\n";
print ("Semantics of circiut:\n" ^ Semantics.pp_mat(tile_to_matrix [[X, I], [I, I], [X, I]]));
print "test 6: Column to circuit works\n";
print ("Column to circuit\n" ^ (Circuit.draw (column_to_circuit [X, Y, Z])) ^ "\n");
print "test 7: Test tile to circuit works\n";
print ("Tile to circuit\n" ^ (Circuit.draw (tile_to_circuit [[X, Y, Z], [I, X, Z]])) ^ "\n");
print "test 8 generate random complex \n";
print (Complex.toString (gen_random_complex (Random.newgen ()) ) ^ "\n\n");
print "test 9: generate list of random complexes \n" ;
print (pp_list (gen_random_complex_list (Random.newgen (), 10), Complex.toString) ^ "\n");
print "test 10: normalize list of complex numbers \n";
print (pp_list (normalize_complex_list (gen_random_complex_list (Random.newgen (), 10))) ^ "\n");
print "test 11: normalize list and check sums to 1 \n";
print (Real.toString ( Math.sqrt (foldl (fn (elm, acc) => Complex.re (Complex.* (elm, Complex.conj elm)), 0, gen_random_complex_list (Random.newgen (), 10)))) ^ "\n");
print "test 12: fingerprint vibe check\n";
print (Real.toString (gen_fingerprint ([[X, H], [I, I]], ListPair.map Complex.mk ([1.0, 1.0], [1.0, 1.0])))) ^ "\n";
print "test 13: fingerprint vibe check\n";
print (Real.toString (gen_fingerprint ([[I, H], [I, I]], ListPair.map Complex.mk ([1.0, 1.0], [1.0, 1.0])))) ^ "\n";
print "test 14: fingerprint vibe check\n";
print (Real.toString (gen_fingerprint ([[X, H], [X, I]], ListPair.map Complex.mk ([1.0, 1.0], [1.0, 1.0])))) ^ "\n";
print "test 15: tile_to_word [[I, X, Y], [Z, H, I]] = 123451\n";
print (Word.toString (tile_to_word [[I, X, Y], [Z, H, I]])) ^ "\n\n";
print "test 16: tile equality vibe check \n";
print (Bool.toString ([[X, Y, Z], [I, H, I]] = [[X, Y, Z], [I, H, I]])) ^ "\n\n";
print "test 17: Fingerprint equality vibe check \n";
let val psi0 = normalize_complex_list (ListPair.map Complex.mk ([1.0, 1.0, 7.0, 2.0], [1.0, 1.0, 7.0, 2.0]), gen_random_complex_list (Random.newgen (), 10))
    val psi1 = normalize_complex_list (ListPair.map Complex.mk ([1.0, 1.0, 0.5, 0.1], [1.0, 1.0, 0.5, 0.1]), gen_random_complex_list (Random.newgen (), 10))
    val t = [[X, H], [X, I]]
    val t2 = [[X, X], [X, X]]
    val f0 = gen_fingerprint (t, psi0, psi1)
    val f1 = gen_fingerprint (t2, psi0, psi1)
in
print((Bool.toString (fingerprint_equality(f0, f0+0.0000000000000001))) ^ "\n\n")
    ^ "test 18: Fingerprint not equal\n"

```



```

        ^ (Bool.toString (fingerprint_equality(f0, f1)) ^ "\n\n"))
end;
print "test 19: Basic generator\n";
print "Gate set: [I, X], height=2, depth = 3\n";
print (pp_database (generator([I, X], 2, 2)) ^ "\n\n");
(*print "test 20: Make columns for sliding window tiles:\n";
print "Input column=[X, Y, Z, I, H]\nOutput:\n";
print ((pp_column_list (split_column ([X, Y, Z, I, H], 5, 3))) ^ "\n");*)
print "test 21: Make non-overlapping tiles from circuit:\n";
print "Input circuit=[X, X, Z], [H, H, Z], [I, I, Z]], tile dimension= 2 x 2\nOutput:\n";
print ((pp_tile_list (circuit_to_tile_partition ([X, X, Z], [H, H, Z], [I, I, Z]), 2,
let val circuit = [[X, X, Z], [H, H, Z], [I, I, Z]]
    val height = 2
    val depth = 2
    val database = generator([I, X, Z, H], height, depth)
    val circuit_tile_part = circuit_to_tile_partition (circuit, height, depth)
    val circuit_tiles_optimized_once = optimize_tile_partition (circuit_tile_part, data
in
    print ("Test 22: Optimize Circuit tiles once\nBefore optimization:\n" ^
        (pp_tile_list circuit_tile_part) ^ "\nAfter optimization:\n" ^
        (pp_tile_list circuit_tiles_optimized_once) ^ "\n\n"
    ) end;
print "test 23: Test circuit can remove superflous I's\n";
print (pp_column_list (remove_I_columns [[I, X, X], [Y, I, Y], [Z, Z, I]]) ^ "\n\n");
print "test 24: Test we can go from tile partitioning to circuit\n";
let val circuit = [[X, X, Z, Z], [X, X, Z, Z], [I, I, Z, Z]]
    val height = 3
    val depth = 3
    val database = generator([I, X, Z], height, depth)
    val circuit_tile_part = circuit_to_tile_partition (circuit, height, depth)
    val part_to_circuit = tile_partition_to_circuit (circuit_tile_part, 6, 3)
in
print ("Original circuit:\n" ^ (pp_column_list circuit)
    ^ "\nCircuit tile partitioning:\n" ^ (pp_tile_list circuit_tile_part)
    ^ "\nTile partitioning to circuit:" ^ (pp_column_list part_to_circuit)
    ^ "\n\n")
end;
print ("test 25: Test we can make 1 optimization pass :} \n");
let val circuit = [[X, Z, Y], [X, Y, Z], [I, Z, X]]
    val height = 2
    val depth = 3
    val database = generator([I, X, Y, Z], height, depth)
    val optimized_circuit = optimization_pass (circuit, database, height, depth)
in print ("Original circuit:\n" ^ pp_tile circuit ^ "\n");
    print ("Optimized circuit: \n" ^ pp_tile optimized_circuit ^ "\n\n")
end;

```



```
# Time the execution of the compiled program
echo "Running the compiled program..."
START=$(date +%s.%N)
./run
END=$(date +%s.%N)

# Calculate elapsed time
DURATION=$(echo "$END - $START" | bc)
echo "Execution time for $file: $DURATION seconds"
echo "-----"

# Clean up the generated 'run' file
echo "Cleaning up..."
rm -f ./run
done
```