

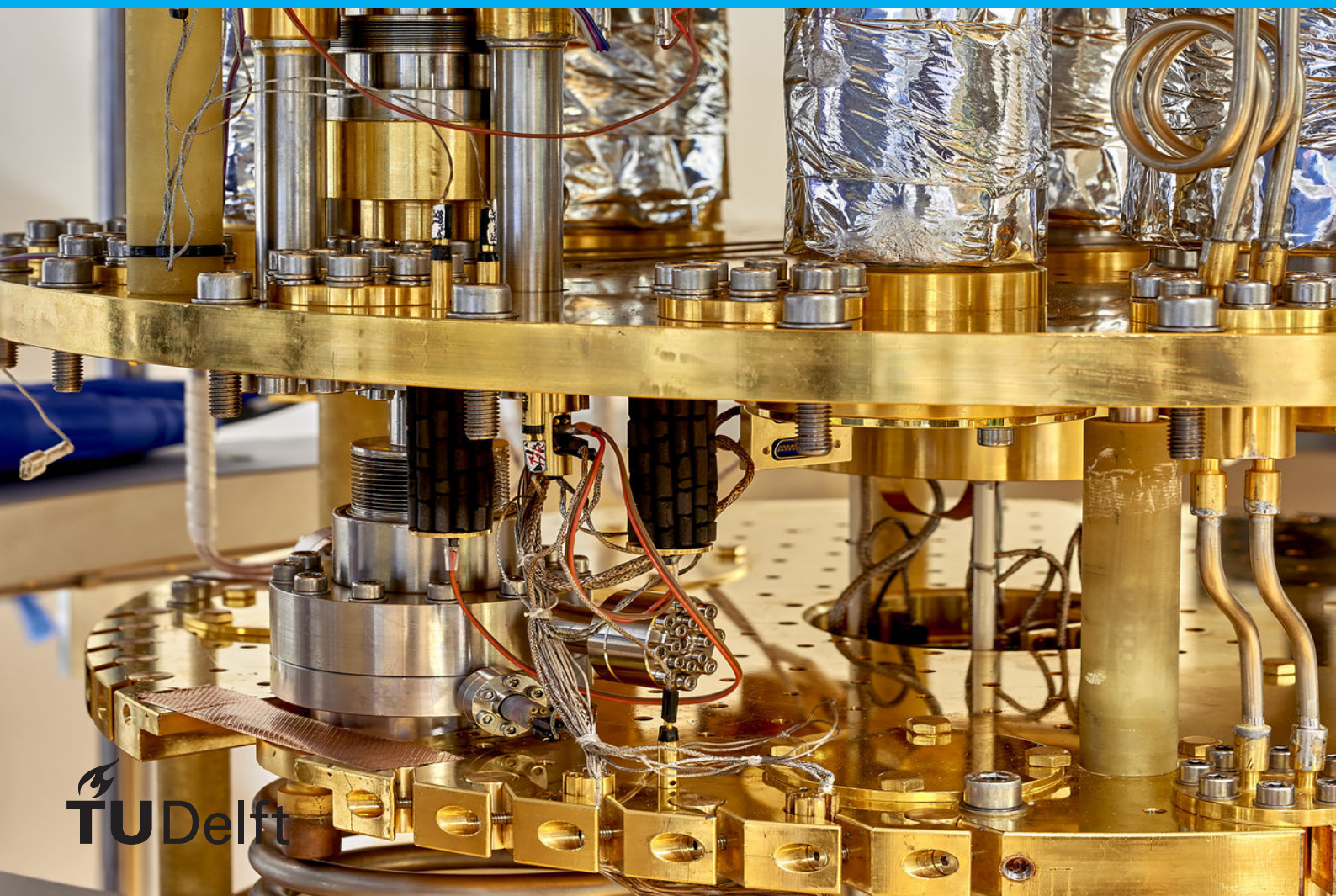
# Unitary Decomposition

Implemented in the OpenQL  
programming language for  
quantum computation

A.M. Krol

**September 13th**

36 Electrical Engineering, Mathematics and Computer  
Sciences LB 01.010 Snijderszaal





# Unitary Decomposition

Implemented in the OpenQL programming language for  
quantum computation

by

A.M. Krol

to obtain the degree of Master of Science  
at the Delft University of Technology,  
to be defended publicly on Friday September 13, 2019 at 15:00.

Student number: 4292391  
Project duration: November 20, 2018 – September 13, 2019  
Thesis committee: Prof. dr. ir. K. Bertels, TU Delft, supervisor  
Dr. I. Ashraf, TU Delft  
Dr. M. Möller, TU Delft  
Dr. Z. Al-Ars, TU Delft

An electronic version of this thesis is available at <http://repository.tudelft.nl/>.



# Preface

In this thesis, I explain my implementation of Unitary Decomposition in OpenQL. Unitary Decomposition is an algorithm for translating a unitary matrix into many small unitary matrices, which correspond to a circuit that can be executed on a quantum computer. It is implemented in the quantum programming framework of the QCA-group at TU Delft: OpenQL, a library for Python and C++. Unitary Decomposition is a necessary part in Quantum Associative Memory, an algorithm used in Quantum Genome Sequencing. The implementation is faster than other known implementations, and generates  $3 * 2^{n-1} * (2^n - 1)$  rotation gates for an n-qubit input gate. This is not the least-known nor the theoretical minimum amount, and there are some optimizations that can still be done to make it closer to these numbers.

I would like to thank my supervisor, Prof. Koen Bertels, for the great supervision and insightful feedback. And my thesis committee member, Dr. Imran Ashraf for the great answers and polite emails as response to my questions about OpenQL and QX. My other committee members, Dr. Matthias Möller and Dr. Zaid Al-Ars, I am also thankful for. I would also like to thank Aritra Sarkar, for supplying this assignment and all the good talks and useful insights and feedback. Finally, my office-mates Amitabh Yadav, for sharing the Master-thesis life and Neil Eelman, for all the great procrastination and interesting talks about everything from the history of cabbage to the weirdness of the English and Dutch languages. And everyone else in the Quantum and Computer Engineering department, who have been very nice, helpful and insightful in the ten months that I have worked here.

*A.M. Krol  
Non-student email: [annerietkrol@gmail.com](mailto:annerietkrol@gmail.com)  
Delft, September 2019*



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Problem statement and research question . . . . .	1
1.2	A "simple" decomposition example . . . . .	1
1.3	The need for unitary decomposition . . . . .	5
1.4	Existing implementations. . . . .	6
1.5	Scope of the thesis . . . . .	6
1.6	Structure of the thesis . . . . .	7
<b>2</b>	<b>Background and related works</b>	<b>9</b>
2.1	Introduction and Notation . . . . .	9
2.1.1	Qubits . . . . .	9
2.1.2	Gates . . . . .	10
2.1.3	Unitary matrices . . . . .	11
2.2	Quantum Accelerator Applications. . . . .	11
2.2.1	Quantum annealing . . . . .	11
2.2.2	Gate-based computation. . . . .	12
2.3	Quantum Annealing: example of a Quantum accelerator . . . . .	12
2.3.1	Fujitsu digital annealer . . . . .	13
2.3.2	D-Wave quantum annealing . . . . .	13
2.4	Gate-based quantum computing. . . . .	14
2.4.1	Rigetti . . . . .	14
2.4.2	IBM and Qiskit . . . . .	14
2.4.3	Intel: superconducting and spin-qubit quantum processor chips. . . . .	15
2.4.4	Alibaba: Cloud Quantum Computer . . . . .	15
2.5	XanaduAI quantum machine learning . . . . .	15
2.6	Quantum Software . . . . .	16
2.6.1	Qubiter . . . . .	16
2.7	Conclusion . . . . .	16
<b>3</b>	<b>How does a compiler work?</b>	<b>21</b>
3.1	What is a compiler? . . . . .	21
3.2	Front End . . . . .	21
3.3	Parsing . . . . .	22
3.3.1	Lexical syntax. . . . .	22
3.3.2	Context-free syntax. . . . .	22
3.4	Type Checking . . . . .	23
3.4.1	Name resolution . . . . .	23
3.4.2	Types . . . . .	25
3.5	Code Generation . . . . .	25
3.5.1	Register Allocation . . . . .	25
3.5.2	Loops . . . . .	26
3.6	Back end . . . . .	26
3.6.1	Memory allocation and garbage collection . . . . .	27
3.6.2	Instruction selection . . . . .	27

3.7	Optimization . . . . .	27
3.7.1	Loop optimizations . . . . .	28
3.7.2	Reusing repeating code . . . . .	28
3.7.3	Dead code elimination . . . . .	28
3.7.4	Loop unrolling. . . . .	28
3.7.5	Inlining . . . . .	29
3.7.6	Combining and removing instructions . . . . .	29
3.8	Different kinds of compilers . . . . .	29
3.8.1	Source-to-source compilers . . . . .	29
3.8.2	Machine code compilers . . . . .	29
3.8.3	Just-In-Time compilers . . . . .	29
3.8.4	Hardware compilers . . . . .	29
<b>4</b>	<b>OpenQL</b>	<b>31</b>
4.1	Quantum computers as accelerators . . . . .	31
4.2	An OpenQL program . . . . .	32
4.3	Components of an OpenQL application . . . . .	33
4.4	Compilation steps . . . . .	34
4.4.1	Decomposition . . . . .	34
4.4.2	Optimization . . . . .	35
4.4.3	Scheduling . . . . .	37
4.4.4	Mapping. . . . .	37
4.4.5	QASM Compilation . . . . .	37
4.4.6	Compilation to specific back-ends . . . . .	37
4.5	QASM . . . . .	37
4.6	Compiler structure . . . . .	38
4.6.1	openql/openql.h. . . . .	39
4.6.2	The definition of a gate . . . . .	40
4.6.3	Compilation . . . . .	40
<b>5</b>	<b>Quantum Genome Sequencing</b>	<b>41</b>
5.1	Genome Sequencing. . . . .	41
5.2	Quantum algorithms for genome sequencing . . . . .	43
5.2.1	Grover's search. . . . .	43
5.2.2	Conditional Oracle call . . . . .	43
5.2.3	Quantum phone directory . . . . .	44
5.2.4	Quantum associative memory . . . . .	44
<b>6</b>	<b>Unitary Decomposition Implementation</b>	<b>45</b>
6.1	Advantages and disadvantages of unitary decomposition . . . . .	45
6.2	Unitary gates . . . . .	46
6.3	Quantum Logic Gates . . . . .	47
6.4	Special gates . . . . .	48
6.5	Comparison of different algorithms for unitary decomposition . . . . .	49
6.5.1	QR decomposition: Efficient decomposition of Quantum Gates . . . . .	49
6.5.2	QR decomposition (Theorem 9 from [39]). . . . .	49
6.5.3	CSD: Quantum circuits for general multi-qubit gates . . . . .	49
6.5.4	CSD and NQ: Decomposition of general quantum gates. . . . .	49
6.5.5	Theoretical lower bounds . . . . .	49
6.6	One qubit gate: ZYZ decomposition. . . . .	50
6.7	Minimum universal quantum circuit for a two qubit gate . . . . .	51
6.8	Complete Quantum Shannon Decomposition algorithm . . . . .	51
6.8.1	Quantum Shannon decomposition. . . . .	51
6.8.2	Cosine Sine decomposition . . . . .	51
6.8.3	Singular Value Decomposition . . . . .	52
6.8.4	Decomposing a uniformly controlled arbitrary gate . . . . .	53
6.8.5	Decomposing multi-controlled rotation gates . . . . .	53



---

6.9	OpenQL implementation . . . . .	54
6.9.1	Unitary . . . . .	54
6.9.2	Kernel . . . . .	55
6.9.3	Compilation . . . . .	55
6.10	Gate count Optimization . . . . .	55
6.10.1	Detection of multiplexors . . . . .	55
6.10.2	Unaffected qubits . . . . .	56
6.11	Execution time optimizations. . . . .	56
6.11.1	Adding the gate to the kernel . . . . .	57
6.11.2	The decompose() function . . . . .	58
6.12	Number of gates . . . . .	60
6.13	Conclusion . . . . .	61
<b>7</b>	<b>Verification and unit tests</b>	<b>63</b>
7.1	The different types of unit tests . . . . .	63
7.2	Output file comparison . . . . .	63
7.3	Exceptions and errors . . . . .	64
7.4	Verifying the result . . . . .	65
7.5	Corner cases . . . . .	66
<b>8</b>	<b>Performance</b>	<b>67</b>
8.1	Metrics . . . . .	67
8.2	Metrics that are not used. . . . .	68
8.3	Number of generated gates . . . . .	68
8.4	Execution time . . . . .	69
8.5	Memory allocation . . . . .	71
8.6	Execution time compared to Qubiter . . . . .	72
8.7	Expected scaling for bigger gates . . . . .	74
8.8	Limitations . . . . .	75
8.8.1	Size of the circuit . . . . .	75
8.8.2	Execution time and memory use. . . . .	76
<b>9</b>	<b>Conclusion and future work</b>	<b>77</b>
9.1	Conclusion . . . . .	77
9.2	Future work . . . . .	78
<b>A</b>	<b>Full Code</b>	<b>81</b>
<b>B</b>	<b>Code from A.Sarkar</b>	<b>95</b>
<b>C</b>	<b>Compilation examples</b>	<b>99</b>
<b>D</b>	<b>Tests</b>	<b>103</b>
	<b>Bibliography</b>	<b>121</b>



# Introduction

The goal of this thesis is to provide language support for the Quantum programming language, OpenQL, which is used to implement and test Quantum Genome Sequencing (QGS) algorithms. Currently, some features that are required by QGS require manual writing of Quantum Assemble Language (QASM) or are not possible at all. This thesis aims to provide the support for QGS so that genome sequencing algorithms can be written in the high-level OpenQL Python/C++ library. Specific examples used in this thesis come from the QGS domain, and refer to fictional DNA-profiles that are investigated. However, this compiler extension has many other potential application cases, and aims to be as complete as possible to facilitate all possible use-cases.

## 1.1. Problem statement and research question

One of the algorithms that requires manual handling of assembly code is Quantum Associative Memory, which requires decomposition of unitary matrices into quantum gates [38]. The workflow for this algorithm is shown in Figure 1.1. When unitary decomposition is implemented, this algorithm can be programmed completely in OpenQL. This lead to the following research question:

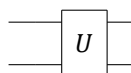
**How can unitary decomposition be implemented in an efficient way in the OpenQL framework, both in terms of number of generated gates and compilation time?**

## 1.2. A "simple" decomposition example

To illustrate what unitary decomposition entails, I will use the example of a general two-qubit gate. Using the program in Listing B.2, a general unitary matrix is generated. Our example matrix is called "U", for "Unitary", and is shown below\*:

```
1 U = [ 0.077+0.669j, -0.234+0.315j,  0.138+0.428j, -0.39 -0.196j,
2 -0.202+0.208j,  0.757+0.308j, -0.451+0.143j,  0.032+0.153j,
3 -0.533+0.091j, -0.094+0.05j ,  0.451+0.192j,  0.192+0.648j,
4 0.412+0.029j, -0.393+0.111j, -0.494+0.299j,  0.407+0.404j]
```

The circuit representation of this matrix would be:



## Cosine Sine Decomposition

The matrix is decomposed in several steps. The first is the Cosine Sine Decomposition. This decomposition calculates three matrices, and the product of these matrices is the original matrix  $U$ .

\*"j" is the python representation for the imaginary number  $i = \sqrt{-1}$

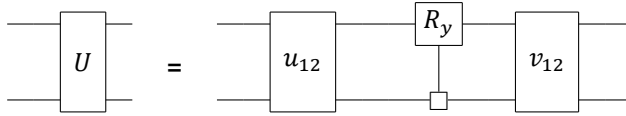
$$U = u_{12} * Mux(R_y) * v_{12}$$

$$u_{12} = \begin{pmatrix} 0.517 + 0.658j & -0.131 + 0.532j & 0 + 0j & 0 + 0j \\ -0.504 + 0.214j & 0.731 + 0.407j & 0 + 0j & 0 + 0j \\ 0 + 0j & 0 + 0j & -0.581 + 0.340j & -0.644 + 0.364j \\ 0 + 0j & 0 + 0j & 0.780 + -0.208j & -0.650 + 0.180j \end{pmatrix}$$

$$Mux(r_y) = \begin{pmatrix} C & S \\ -S & C \end{pmatrix} = \begin{pmatrix} 0.706 + 0j & 0 + 0j & -0.708 + 0j & 0 + 0j \\ 0 + 0j & 0.927 + 0j & -0 + -0j & -0.374 + -0j \\ 0.708 + 0j & 0 + 0j & 0.706277 + 0j & 0 + 0j \\ 0 + 0j & 0.374 + 0j & 0 + 0j & 0.927 + 0j \end{pmatrix}$$

$$v_{12} = \begin{pmatrix} 0.886 + 0.330j & -0.325 + -0j & 0 + 0j & 0 + 0j \\ 0.305 + 0.114j & 0.946 + -0j & 0 + 0j & 0 + 0j \\ 0 + 0j & 0 + 0j & -0.863 + -0.219j & 0.444 + -0.100j \\ 0 + 0j & 0 + 0j & 0.166 + -0.424j & -0.086 + -0.886j \end{pmatrix}$$

Which looks like this in a quantum circuit:



The  $u_{12}$  and  $v_{12}$  matrices represent two-qubit gates, where either the upper left matrix, or the lower right matrix gets applied to the lower qubit, depending on the value of the top qubit. This is called a multiplexor. The  $Mux(R_y)$  gate is a multiplexed rotation-y gate. This gate rotates the qubit around the Y-axis with a different angle for each possible value of the controlling qubit(s).

## Singular Value Decomposition

The general multiplexors  $u_{12}$  and  $v_{12}$  get decomposed using Singular Value Decomposition. This generates again three matrices per multiplexor, which result in the original matrix when they are multiplied:

$$u_{12} = (I \otimes V_u) \cdot Mux(R_{xu})_u \cdot (I \otimes W_u)$$

$$I \otimes V_u = \begin{pmatrix} 0.0636 + 0.078j & -0.995 + 0j & 0 + 0j & 0 + 0j \\ 0.897 + 0.430j & 0.091 + -0.043j & 0 + 0j & 0 + 0j \\ 0 + 0j & 0 + 0j & 0.064 + 0.078j & -0.995 + 0j \\ 0 + 0j & 0 + 0j & 0.897 + 0.430j & 0.091 + -0.043j \end{pmatrix}$$

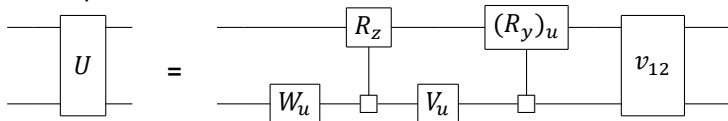
$$V_u = \begin{pmatrix} 0.0636 + 0.078j & -0.995 + 0j \\ 0.897 + 0.430j & 0.0911 + -0.043j \end{pmatrix}$$

$$Mux(R_z)_u = \begin{pmatrix} D & 0 \\ 0 & D^\dagger \end{pmatrix} = \begin{pmatrix} 0.682 + 0.731j & 0 + 0j & 0 + 0j & 0 + 0j \\ 0 + 0j & 0.067 + -0.998j & 0 + 0j & 0 + 0j \\ 0 + 0j & 0 + 0j & 0.682 - 0.731j & 0 + 0j \\ 0 + 0j & 0 + 0j & 0 + 0j & 0.067 + 0.998j \end{pmatrix}$$

$$I \otimes W_u = \begin{pmatrix} 0.219 + -0.349j & 0.285 + -0.865j & 0 + 0j & 0 + 0j \\ 0.247 + -0.877j & -0.013 + 0.411j & 0 + 0j & 0 + 0j \\ 0 + 0j & 0 + 0j & 0.219 + -0.349j & 0.285 + -0.865j \\ 0 + 0j & 0 + 0j & 0.247 + -0.877j & -0.019 + 0.411j \end{pmatrix}$$

$$W_u = \begin{pmatrix} 0.219 + -0.349j & 0.285 + -0.865j \\ 0.247 + -0.877j & -0.013 + 0.411j \end{pmatrix}$$

As a quantum circuit, it now looks like this:



As you can see, the 2-by-2 matrices  $V_u$  and  $W_u$  are single qubit gates on the lower qubit, while the multiplexed  $R_z$  gate is applied to the first qubit, just like the multiplexed  $R_y$  gate.

The same is then done for matrix  $v_{12}$ :

$$v_{12} = (I \otimes V_v) \cdot \text{Mux}(R_z)_v \cdot (I \otimes W_v)$$

$$I \otimes V_u = \begin{pmatrix} 0.938 + 0j & 0.310 + -0.156j & 0 + 0j & 0 + 0j \\ 0.183 + -0.295j & -0.083 + 0.934j & 0 + 0j & 0 + 0j \\ 0 + 0j & 0 + 0j & 0.938 + 0j & 0.310 + -0.156j \\ 0 + 0j & 0 + 0j & 0.183 + -0.295j & -0.083 + 0.934j \end{pmatrix}$$

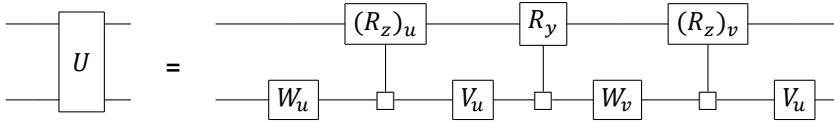
$$V_u = \begin{pmatrix} 0.938 + 0j & 0.310 + -0.156j \\ 0.183 + -0.295j & -0.083 + 0.934j \end{pmatrix}$$

$$\text{Mux}(R_z)_v = \begin{pmatrix} D & 0 \\ 0 & D^\dagger \end{pmatrix} = \begin{pmatrix} 0.825 + -0.566j & 0 + 0j & 0 + 0j & 0 + 0j \\ 0 + 0j & 0.135 + -0.991j & 0 + 0j & 0 + 0j \\ 0 + 0j & 0 + 0j & 0.825 + 0.566j & 0 + 0j \\ 0 + 0j & 0 + 0j & 0 + 0j & 0.135 + 0.991j \end{pmatrix}$$

$$I \otimes W_u = \begin{pmatrix} -0.015 + 0.605j & -0.536 + 0.589j & 0 + 0j & 0 + 0j \\ -0.691 + 0.397j & 0.593 + 0.116j & 0 + 0j & 0 + 0j \\ 0 + 0j & 0 + 0j & -0.015 + 0.605j & -0.536 + 0.589j \\ 0 + 0j & 0 + 0j & -0.691 + 0.397j & 0.593 + 0.116j \end{pmatrix}$$

$$W_u = \begin{pmatrix} -0.015 + 0.605j & -0.536 + 0.589j \\ -0.691 + 0.397j & 0.593 + 0.116j \end{pmatrix}$$

So the complete circuit is now:



## Demultiplexing and ZYZ decomposition

Before this circuit can be executed on a real quantum computer, two more steps need to happen. The first is to translate the multiplexed rotation gates into regular rotation gates and CNOTs. For a two-qubit gate, this means two CNOT gates and two rotation gates per multiplexed gate.

$$\text{Mux}(R_y) = \text{CNOT} \cdot (I \otimes R_y(\theta_0)) \cdot \text{CNOT} \cdot (I \otimes R_y(\theta_1))$$

$$\text{Mux}(R_z) = \text{CNOT} \cdot (I \otimes R_z(\theta_0)) \cdot \text{CNOT} \cdot (I \otimes R_z(\theta_1))$$

The angles for these rotation gates are found by solving the following equation, which follows from the set order of CNOTs:

$$\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} \theta_0 \\ \theta_1 \end{pmatrix} = \begin{pmatrix} \alpha_0 \\ \alpha_1 \end{pmatrix}$$

$$\begin{pmatrix} \alpha_0 \\ \alpha_1 \end{pmatrix}_{R_y} = 2 \cdot \arcsin(\text{diag}(S)) = 2 \cdot \arcsin \begin{pmatrix} -0.708 \\ -0.374 \end{pmatrix}$$

$$\begin{pmatrix} \alpha_0 \\ \alpha_1 \end{pmatrix}_{R_z} = -2j \cdot \log(\text{diag}(D))$$

$$\begin{pmatrix} \alpha_0 \\ \alpha_1 \end{pmatrix}_{R_z u} = -2j \cdot \log \begin{pmatrix} 0.682 + 0.731j \\ 0.067 - 0.998j \end{pmatrix}$$

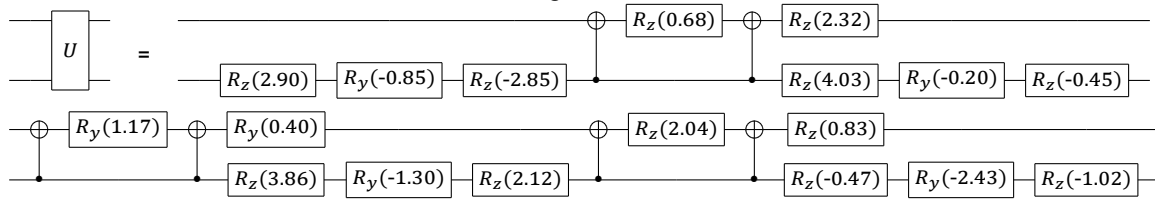
$$\begin{pmatrix} \alpha_0 \\ \alpha_1 \end{pmatrix}_{R_z v} = -2j \cdot \log \begin{pmatrix} 0.825 - 0.566j \\ 0.135 - 0.991j \end{pmatrix}$$

The single qubit gates can be transformed into three rotation gates, like this:

$$U_{1qubit} = R_z(\alpha)R_y(\beta)R_z(\gamma)$$

## Final circuit

So the total circuit, with the correct rotation angles in radians, is now: †



## In OpenQL

As can be seen, this process has a lot of steps, and a two-qubit gate, with just 16 matrix entries, already results in 18 rotation gates and 6 CNOT gates. This number increases exponentially with the number of qubits the input matrix affects. Therefore, any human involvement in this decomposition will make it very error-prone. And because the resulting circuits are very long, very unclear and inconvenient to use. So ideally, the decomposition is handled by a computer algorithm implemented in the compiler, so that a user can just use arbitrary unitary gates in any algorithm without having to deal with the decomposition.

With the unitary decomposition implemented in OpenQL, the way to generate this circuit as quantum assembly language (QASM), is very simple. First the matrix is defined. This can also be the output of some algorithm or function. This matrix is added to a "Unitary" object, and decomposed. Then, the "Unitary" is added to the kernel in the same way as normal gates.

```

1 U = [ 0.077+0.669j, -0.234+0.315j, 0.138+0.428j, -0.39 -0.196j,
2 -0.202+0.208j, 0.757+0.308j, -0.451+0.143j, 0.032+0.153j,
3 -0.533+0.091j, -0.094+0.05j, 0.451+0.192j, 0.192+0.648j,
4 0.412+0.029j, -0.393+0.111j, -0.494+0.299j, 0.407+0.404j]
5 u_unitary = ql.Unitary("Unitary",U)
6 u_unitary.decompose()
7 kernel.gate(u_unitary, [0,1])

```

OpenQL takes the matrix, goes through all the decomposition steps, and finally makes the same circuit. This circuit is printed as QASM:

```

1 version 1.0
2 # this file has been automatically generated by the OpenQL compiler please do not modify
  it manually.
3 qubits 2
4
5 .newKernel
6   rz q[0], 2.898309
7   ry q[0], -0.848687
8   rz q[0], -2.853675
9   rz q[1], 0.681902
10  cnot q[0],q[1]
11  rz q[1], 2.323443
12  cnot q[0],q[1]
13  rz q[0], 4.033960
14  ry q[0], -0.201912
15  rz q[0], -0.446115
16  ry q[1], 1.169904
17  cnot q[0],q[1]
18  ry q[1], 0.403249
19  cnot q[0],q[1]
20  rz q[0], 3.857529
21  ry q[0], -1.299610
22  rz q[0], 2.115405
23  rz q[1], 2.036688
24  cnot q[0],q[1]
25  rz q[1], 0.833905
26  cnot q[0],q[1]
27  rz q[0], 0.465892
28  ry q[0], -2.432474

```

†The angles are rounded to two decimals because otherwise the circuit does not fit on the page

```
29 rz q[0], -1.016344
```

### 1.3. The need for unitary decomposition

As can be seen in the example, unitary decomposition is the process of splitting a big unitary matrix into many other smaller unitary matrices, so that the product of the smaller matrices is equal to the initial big one. This is equivalent to taking a generic many-qubit quantum gate and splitting it into a specific combination of rotation gates and CNOTs. For algorithms which are specified as a generic unitary gate, this step is necessary to execute the algorithm using qubits. Therefore, these generic unitary gates need to be translated into a universal set of gates that can be executed on a quantum computer or a simulation of one.

One of the algorithms that uses unitary decomposition is Quantum Associative Memory, of which the Matlab implementation is shown in Listing B.1. Quantum Associative Memory is a pattern matching algorithm which matches short DNA sequences to a much bigger reference genome. The short reads are stored on the quantum memory using a unitary matrix, which needs to be decomposed into gates. When it is decomposed, a quantum computer can be used to find the solution to the pattern matching problem.

The previous workflow to execute the Quantum Associative Memory algorithm is shown in Figure 1.1. First, some small pattern of DNA is read or generated. This pattern is then used to assemble a unitary matrix, which is then decomposed into a set of unitary gates and printed to the console. This part is implemented in Matlab. The printed code is then manually copy-pasted into Python, and the OpenQL library is used to generate the cQASM output file. Which can then be executed using QX, a quantum simulator. The QX output is used to determine the solution.

Because of the copy-pasting of the potentially very long OpenQL code and the use of different programming languages, this workflow cannot be totally automated, is error prone and cumbersome.

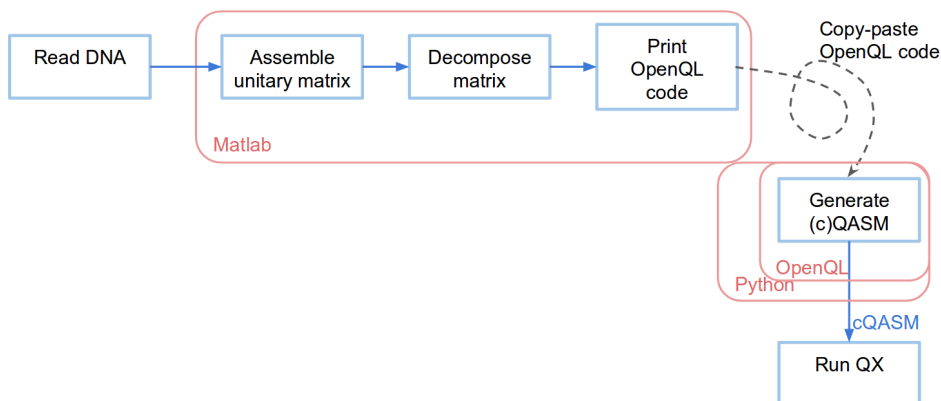


Figure 1.1: Workflow for the previous implementation of unitary decomposition, which used Matlab and Python.

When unitary decomposition is available inside OpenQL, this workflow can be condensed to the one shown in Figure 1.2. The DNA is still read or generated as before, but now Python can be directly used to assemble the unitary matrix. This matrix is then decomposed using OpenQL, which also generated the cQASM. The cQASM output file can then be executed using QX as before. This allows more automation, integration with other OpenQL functionalities and the circuit inside OpenQL is much shorter and more plain. And because all of the code is now in one place, debugging will become easier as well. This also provides opportunities for optimizations that combine OpenQL with the decomposition functionality, which can bring down the gate count of the final decomposition. Therefore, the goal and scope will be to implement unitary decomposition inside OpenQL, in an efficient way and with some optimization options.

To do this, the existing decomposition algorithm will need to be translated to C++ and implemented into the existing OpenQL framework. It will need to be validated and tested, and a suitable linear algebra library will need to be selected. Optimizations will need to be selected and implemented, and tested in a way that is independent of the specific circuit that is generated.

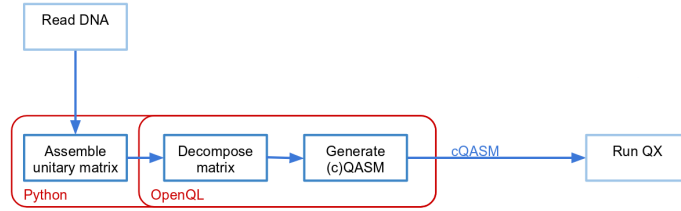


Figure 1.2: Workflow for new implementation of unitary decomposition, where the whole algorithm can be written in Python.

## 1.4. Existing implementations

Of the different quantum programming languages out there, the only one that currently implements full unitary decomposition up to any qubit size is Qubiter. It uses Cosine Sine Decomposition applied recursively to a tree of node matrices, so that the product of the node matrices is the input matrix. This is shown in Figure 1.3.

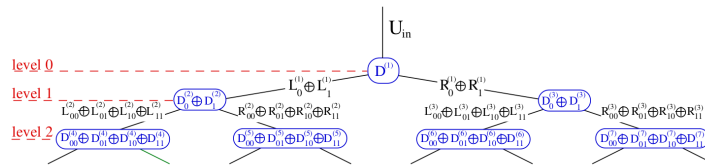


Figure 1.3: The CSD binary tree as used in Qubiter [12].

It first generates multiplexors and diagonal matrices. Two separate methods are used to transform these to a circuit consisting of CNOTs and qubit rotations only [12]. This is shown in Figure 1.4. It also supplies a method to check the expansion, by multiplying the gates of the circuit. More about Qubiter in Section 2.6.1.

```

1 num_bits = 3
2 init_unitary_mat = FouSEO_writer.fourier_trans_mat(1 << num_bits)
3 emb = CktEmbedder(num_bits, num_bits)
4 file_prefix = 'csd_test'
5
6 t = Tree(True, file_prefix, emb, init_unitary_mat, verbose=False)
7 t.close_files()
8
9 wr = MultiplexorExpander(file_prefix, nqubits, 'exact')
10 zr = DiagUnitaryExpander(file_prefix + '_X1', num_bits, 'exact')
  
```

Figure 1.4: Decomposition of a three qubit unitary matrix using Qubiter [12].

## 1.5. Scope of the thesis

The goal of the thesis is to implement unitary decomposition in OpenQL. This unitary decomposition will be applied to perfect qubits, and tested using a quantum simulator. Although the generated circuits can be run on real qubits, for the current state-of-the-art significant post-processing would be needed even for the cases where it would be possible. So therefore, the following assumptions are made for the implementation in this thesis:

- The qubits do not decohere: they keep their state indefinitely
- Any qubit can interact with any other qubit
- Quantum gates are instantaneous
- Quantum gates are exact



- It is possible to know the exact state of the system

So several things do not need to be taken into account for the generated quantum circuit. Because the qubits keep their state indefinitely, rather than go to some ground state after a certain time, the length of the generated circuit does not affect the outcome. And error-correcting mechanisms such as ancilla qubits are not necessary. Any qubit can interact with any other, so two-qubit gates can be between any two qubits. Therefore, no mapping of the qubits is necessary.

In addition to perfect qubits, perfect quantum gates are also assumed. The gates used in this thesis are instantaneous, so no timing constraints need to be met or generated. In addition, the gates do not introduce any errors, nor do they have any kind of limit to their accuracy.

Finally, and this is very important for checking the decomposition, the exact state of the system can be known. This is needed to verify the decomposition, but it is only possible on a simulator. Because real measurement on real qubits collapses the qubit state, but the correctness of the decomposition will be verified using the uncollapsed qubit states at the end of the decomposition.

The nature of the decomposition algorithm used is that it is exact, as long as no errors are introduced. But the gates that are generated can be run on real qubits, though in that case mapping, possibly gate swapping and error-correcting measures will need to be used in order to execute the generated circuit. These measures are all application specific, however, and are therefore not in the scope of this thesis.

## 1.6. Structure of the thesis

This chapter states the problem statement and the research question. It shows a decomposition example, demonstrates the need for unitary decomposition and shows existing implementations. Finally, it explains the scope of the thesis.

Chapter 2 starts with an introduction to quantum computation and which notations will be used. Then, it outlines several possible applications for quantum accelerators, and then shows several quantum annealers, gate-based quantum computers, a quantum machine learning platform and an example of a quantum programming language. Chapter 3 gives a short explanation of compilers, including front-end, back-end, possible optimizations and different kinds of compilers. Chapter 4 is about OpenQL, from the perspective of a user in Sections 4.2 to 4.4. It gives the structure of an OpenQL program, the components of such a program and which compilation steps are taken. Then it gives an overview of the Quantum Assembly Language (QASM) in Section 4.5. For people who need to change something in the OpenQL compiler, the structure is shown in Section 4.6. Chapter 5 first explains genome sequencing, and then the different algorithms for speeding up genome sequencing using quantum computers.

Chapter 6 is about the implementation of unitary decomposition in OpenQL. Section 6.1 gives the advantages and disadvantages of the decomposition, Sections 6.2 to 6.4 give the mathematical definition for several different quantum gates: the general definition of a gate, the universal gate set that will be used and some special gates that will be used as the intermediate values of the decomposition. In Section 6.5 several different algorithms for unitary decomposition are compared, and Quantum Shannon Decomposition (QSD) is selected. Minimum implementations for one- and two-qubit gates are shown in Sections 6.6 and 6.7. Then, the complete mathematical basis for QSD is shown in Section 6.8, and the OpenQL implementation in Section 6.9. This implementation was optimized with regards to gate count ) and execution speed in Sections 6.10 and 6.11. Finally, the number of gates generated by the decomposition is calculated in Section 6.12 and some conclusions are drawn in Section 6.13.

Chapter 7 shows the different unit-tests that were written to verify the decomposition. These different types are output file comparison in Section 7.2, exceptions and errors in Section 7.3, verification of the simulation results in Section 7.4 and finally several corner cases in Section 7.5.

Finally, the performance of the decomposition was measured, and the results are shown in Chapter 8. First the used and unused metrics are explained in Sections 8.1 and 8.2. Then the number of generated gates is shown in Section 8.3, the execution time in Section 8.4 and the memory allocation in Section 8.5. The execution time of the OpenQL decomposition is compared to that of Qubiter in Section 8.6 and some predictions are made for the decomposition of bigger gates in Section 8.7. Finally, some limitations are discussed in Section 8.8. These are then summarized in Chapter 9, along with other conclusions about the work. This chapter also includes several suggestions for future work.



# 2

## Background and related works

In this thesis, quantum computers will be exclusively looked at as accelerators working with classical computers to speed up specific parts of an application. Because current quantum processors are still only experimental devices, with too few qubits and too unreliable for the kinds of applications that this thesis will look at, any quantum algorithms will exclusively be executed using the QX-simulator. From this, it follows that a programming language is needed which translates high-level code into some hardware-agnostic quantum assembly language. OpenQL fulfills these requirements. For completeness and reference, this chapter includes some other quantum programming languages and hardware implementations and the various approaches to current quantum computers. First, some basics about quantum computation and notation are given. Then, various applications for quantum computers as accelerators are given. Several implementations are then given, from quantum annealing to gate based computation to quantum software such as Qubiter. The last category also includes OpenQL, but OpenQL is described in Chapter 4.

### 2.1. Introduction and Notation

In this section, I will explain some of the notation that will be used in this thesis. This includes qubits, ket notation  $|\_ \rangle$ , the Bloch sphere, quantum gates:  $\boxed{X}$ , kronecker product:  $\otimes$  and how these things are expressed mathematically.

#### 2.1.1. Qubits

A classical bit either has the value '0' or the value '1'. A quantum bit (qubit) has some probability amplitude  $\alpha$  of being in state '0', also written as  $|0\rangle$ , and some probability amplitude  $\beta$  of being in state '1', also  $|1\rangle$ .<sup>\*</sup> This is called *superposition*, and can be regarded as being simultaneously in both states. The total expression for such a superposition state is  $\phi = \alpha |0\rangle + \beta |1\rangle$ . When such a state is measured, the state collapses: the qubit becomes either  $|0\rangle$  or  $|1\rangle$ . A '0' is measured with a probability of  $\alpha^2$ , and the qubit state becomes  $|0\rangle$  after the measurement. The probability to measure a '1' is  $\beta^2$ , and then the qubit state becomes  $|1\rangle$ . The total probability of measuring something is 1, of course, so  $\alpha^2 + \beta^2 = 1$ .

One of the ways to show this superposition between  $|0\rangle$  and  $|1\rangle$  is as a sphere, shown in Figure 2.1. The qubit can be in all states that can be represented on the sphere. So qubit states are a 3 dimensional continuous spectrum. On the Z-axis is the position regarding the  $|0\rangle$  and  $|1\rangle$  states, on the Y-axis the position of the state with regards to states  $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$  and  $|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$ . The X-axis corresponds with the imaginary part of the complex state, so from  $\frac{1}{\sqrt{2}}(|0\rangle + i|1\rangle)$  to  $\frac{1}{\sqrt{2}}(|0\rangle - i|1\rangle)$ .

Besides the  $|\_ \rangle$  notation, called bra-ket notation, quantum states can also be represented as vectors:  $|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$  and  $|1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ . The state of multiple qubits can also be expressed as a vector, where the first row corresponds to the binary number "0" in as many bits as there are qubits. And the second row, to the number "1" etc. As an example, for a four qubit state, the first row corresponds to  $|0000\rangle$ , and the

<sup>\*</sup>There is one other base-state that is as frequently used which is based on the + or - values. This thesis focuses mostly on the 0/1 base-state.

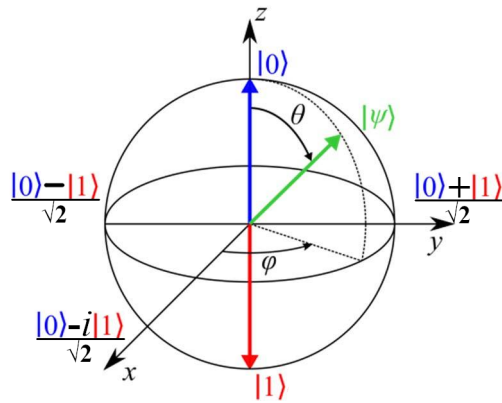


Figure 2.1: The Bloch sphere

second to  $|0001\rangle$ . This continues to the final row, of  $|1111\rangle$ . This means there are a total of  $2^n$  rows, with  $n$  as the number of qubits. For a four qubit state, this means a vector with a total of 16 rows.

### 2.1.2. Gates

Qubits can be manipulated using *gates*. These gates are usually represented as matrices which operate on the qubits in a vector form. These can operate on single or multiple qubits. The single qubit gates are shown in Table 2.1, and multi-qubit gates in Table 2.2.

As an example, the matrix for the Pauli X gate is:  $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ . This means that it transforms a  $|0\rangle$  state:  $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$  into a  $|1\rangle$  state:  $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$ . This is the first column of the matrix. And it transforms the  $|1\rangle$  state into the  $|0\rangle$  state, via the second column of the matrix. This is also illustrated by these equations:

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad (2.1)$$

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad (2.2)$$

This X-gate can also be drawn as a circuit similar to the way classical gates are drawn. With one major difference, quantum circuits are always reversible, except when a measurement is performed. So qubits and the information in them cannot be destroyed by any other means than measurement. Therefore, all qubit gates have the same number of outputs as they have inputs. The X-gate from before would look like this in a circuit:



Measurement of a quantum bit destroys any superposition of the qubit, and transforms (projects) it to the measurement result in the measurement basis. The measurement result is either '0' or '1', while the basis is  $|0\rangle$ ,  $|1\rangle$ . † Measurement looks like this in a quantum circuit:



When gates are applied to more than one qubit, the matrix size scales the same as the vector size. So a two qubit gate is represented by a four-by-four matrix, a three qubit gate by an eight-by-eight and so on.

To calculate the output when multiple gates are applied to the qubits, the matrix-representations of the gates are multiplied in reverse order. So when first a gate called "A" is applied to a qubit, and then afterwards a gate called "B" and then a gate called "C", then the new state of a qubit  $|\phi\rangle$  becomes:  $C \cdot B \cdot A \cdot |\phi\rangle$ .

Or as a circuit:

†or  $|+\rangle$ ,  $|-\rangle$

$$\boxed{A} \boxed{B} \boxed{C} = C \cdot B \cdot A \cdot |\phi\rangle$$

The combined effect of two gates on two different qubit is calculated using the Kronecker product of the two gates. So, if a gate called "A" is applied on the first qubit, and a gate called "B" on the second one, their combined effect is  $A \otimes B$  on the two qubits. If two separate one qubit gates are applied to two different qubits, then the combined effect they have on the qubit state is the Kronecker product of the first gate with the second:  $A \otimes B$ . This also goes if no gate is applied to one of the qubits, in which case the identity gate is used to calculate the total Kronecker product. So as circuits:

$$\begin{aligned} \boxed{A} \quad \text{is} \quad A \otimes B &= \begin{pmatrix} a_0 & a_1 \\ a_2 & a_3 \end{pmatrix} \otimes \begin{pmatrix} b_0 & b_1 \\ b_2 & b_3 \end{pmatrix} = \begin{pmatrix} a_0 \cdot b_0 & a_0 \cdot b_1 & a_1 \cdot b_0 & a_1 \cdot b_1 \\ a_0 \cdot b_2 & a_0 \cdot b_3 & a_1 \cdot b_2 & a_1 \cdot b_3 \\ a_2 \cdot b_0 & a_2 \cdot b_1 & a_2 \cdot b_0 & a_2 \cdot b_1 \\ a_2 \cdot b_2 & a_2 \cdot b_3 & a_3 \cdot b_2 & a_3 \cdot b_3 \end{pmatrix} \\ \text{---} \quad \boxed{B} \quad \text{is} \quad I \otimes B &= \begin{pmatrix} B & 0 \\ 0 & B \end{pmatrix} \\ \text{---} \quad \boxed{B} \quad \text{is} \quad I \otimes I &= \begin{pmatrix} I & 0 \\ 0 & I \end{pmatrix} \\ \text{---} \quad \boxed{A} \quad \text{is} \quad A \otimes I &= \begin{pmatrix} a_0 & a_1 \\ a_2 & a_3 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} a_0 & 0 & a_1 & 0 \\ 0 & a_0 & 0 & a_1 \\ a_2 & 0 & a_3 & 0 \\ 0 & a_2 & 0 & a_3 \end{pmatrix} \end{aligned}$$

This also goes if there are more than two qubits, and for multi-qubit gates, in which case the  $a_0$ 's stand for the blocks of the matrices. This way of calculating the outcome of circuits is used in quantum simulators, and will also be used to check the intermediate and final results of the decomposition.

### 2.1.3. Unitary matrices

For a quantum gate, both input and output can be an arbitrary mixture of possible states. Because they need to be reversible, quantum gates have the same number of input and output bits. The gate performs some permutation on the input state to produce the output state. This permutation can be mathematically described as a matrix, and to satisfy the reversibility, the reverse of this matrix needs to exist for it to describe a valid quantum gate. It also needs to preserve the probability amplitudes of the states, so that the total probability of the particle being in a state is still one. This means the matrix needs to be *unitary* [13].

This means the following things, for a unitary matrix  $U$  [4]:

- $U^\dagger = U^{-1}$
- $U$  is diagonalizable
- $|\det(U)| = 1$  for  $U$  in  $SU$  ‡
- For  $U = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$ ,  $\sqrt{A^2 + B^2} = 1$ .
- Or more general: the columns of  $U$  are orthonormal, so if we denote them as  $u_1, \dots, u_n$  then  $Ux = \sum_1^n x_i u_i$  and  $\|Ux\|^2 = \sum_1^n |x_i|^2 = \|x\|^2$  where  $x = (x_1, \dots, x_n)^T$

These properties will be used later to decompose these matrices.

## 2.2. Quantum Accelerator Applications

There are many possible applications for quantum computers, of which some are outlined below. These are split into two categories, applications for quantum annealers such as those in Section 2.2.1 and applications for gate-based quantum computers, such as those in Section 2.4.

### 2.2.1. Quantum annealing

Based on [23], quantum annealing is suitable for two types of problems, optimization and sampling:

‡An important distinction:  $\det(U) = 1$  for the special unitary group. Otherwise,  $\det(U) = \exp(i * \text{Phi})$

1. Optimization problems: finding the highest or lowest value of some metric for a problem with many different input variables. These problems are translated to the energy minimization of a specific system, where the lowest energy state corresponds to the best solution to the problem.
2. Sampling problems: characterizing the shape of an energy landscape through sampling. This is useful for machine learning problems where the goal is to make a probabilistic model of reality.

Examples of annealing include the Fujitsu digital annealer, and the D-wave quantum one.

### 2.2.2. Gate-based computation

Gate-based quantum computing can be used to solve many different kinds of problems, since the gates are used to program a quantum computer and therefore any problem that can be mapped to a classical computer can be solved on a quantum computer. However, not all classes of problems will be solved faster by a quantum computer. Here are some examples of things that might be solved by quantum computers in the future:

1. Encryption and cyber security: RSA encryption is widely used, while quantum computers could break this encryption much quicker than classical computers. Which is also why quantum computers might be the solution for new, secure communication that cannot be broken [25].
2. Autonomous cars: quantum computers can be used to optimize certain aspects of self-driving vehicles, such as the Ford partnership with NASA for route optimization of cars [32].
3. Financial services and other complicated modelling: computers are widely used to predict market behaviour and other such complicated problems. Quantum computing can be used to factor in more variables and process things in parallel, which might mean faster and better solutions to all kinds of problems [25].
4. Genome sequencing [38]: genome sequencing is also a problem of which certain steps can be executed in parallel, and which entails a lot of data. But more on this in Chapter 5.

Parties that are developing gate-based quantum computers or computer languages are Rigetti, IBM, Intel, Alibaba, the people from Qubiter and XanaduAI. These are compared in Table 2.3.

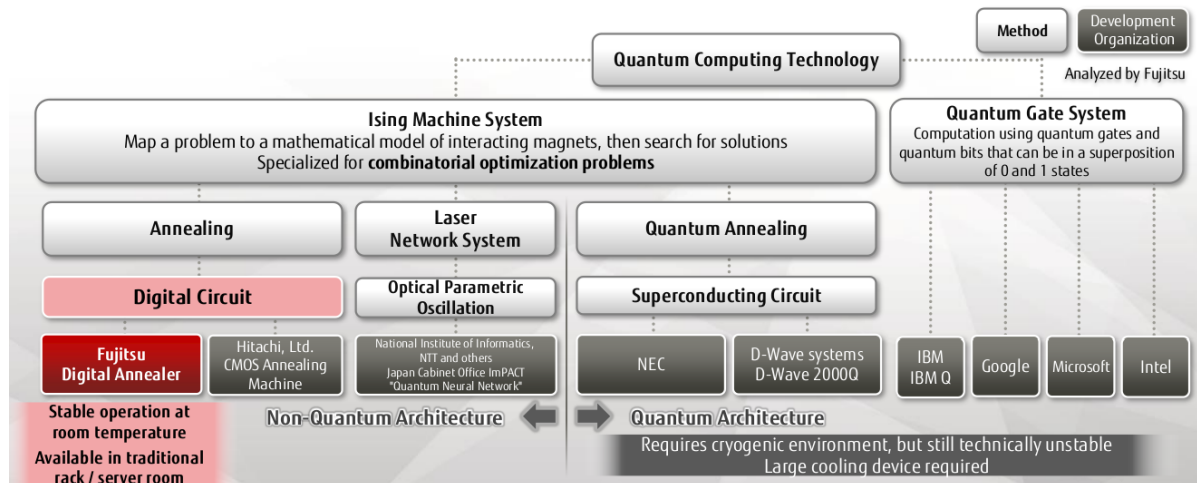


Figure 2.2: Positioning of different quantum computing technologies [30].

### 2.3. Quantum Annealing: example of a Quantum accelerator

The first set of quantum computers deal with a very important optimisation technique, where the annealing algorithm is used in a quantum or quantum-inspired way.

### 2.3.1. Fujitsu digital annealer

Fujitsu has developed a digital annealer, which runs on a classical computer. So it is not producing a real quantum computer but rather a quantum-inspired machine which is based on the annealing algorithm and implemented or inspired by the quantum version of quantum annealing. Annealing is used to find optimal solutions to problems with a lot of variables, and where the solution consists of a sequence or combination of these variables.

Physical annealing is the process which increases the ductility and decreases the hardness of a metal, by heating it above a certain (recrystallization) temperature. At this temperature, the atoms are free to move around, so the crystal structure of the metal is reformed. Then, the metal is cooled slowly so that the crystals can grow [41]. This is akin to finding a solution of a puzzle by shaking up the whole thing, so that pieces can move to the correct spot. Then, slowly decrease the amount of shaking, until the pieces fit inside the puzzle. This is shown in Figure 2.3 and 2.4. For the digital annealing, this means all the solutions are considered first, even those far away from the solution, and then gradually the system closes in on an optimal solution [30].

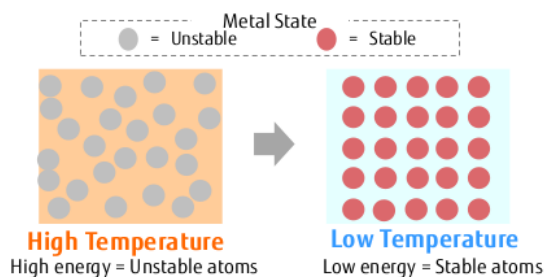


Figure 2.3: The annealing phenomenon in metal [30].

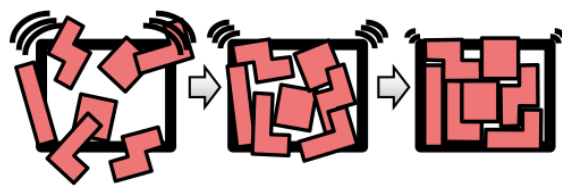


Figure 2.4: Annealing as a solution to a puzzle [30].

For problems where a combination or sequence of variables need to be found which satisfy the given constraints, annealing might be faster than other algorithms. One such problem is the traveling salesman problem: a salesman needs to visit some cities, and would like to take the shortest route possible. To calculate each possible sequence is very costly, there are already  $(7 - 1)! / 2 = 360$  options for just 7 cities. The starting city does not matter, so for  $n$  cities, there are  $(n - 1)!$  possible sequences. The options that need to be calculated are half of that, because it doesn't matter for the total distance whether the salesperson starts at the one end of a route or the other.

The traveling salesman problem can be solved much faster with digital annealing. The variables and constraints need to be defined first; variables are the order of the cities and which cities to visit. Constraints are that each city should be only visited once. This then needs to be represented as an Ising model: the interactions between the variables are expressed as the spins of atoms in a ferromagnetic material [30]. Spins can be 0 or 1, and like magnets, if two atoms have the same spin they are repelled by each other, while atoms with a different spin are attracted. The interactions between the atoms can be specified in a simulation or using external magnetic fields, which manipulate the system so that the lowest energy state of the system represents the solution of the problem [51].

Fujitsu implements this using a digital circuit, which consequently can operate at room temperature. The relation between the different technologies is shown in Figure 2.2. As opposed to quantum circuits, which need to be cooled to almost 0K. The circuit consists of 8192 bits, with total coupling and 64-bit gradation evaluation accuracy. A customer formulates the Ising model, the variables and the constraints, and sends this to Fujitsu. In the Fujitsu datacenter the annealing is performed, and the optimal solution is send back [30].

### 2.3.2. D-Wave quantum annealing

The D-wave system implements annealing as a quantum system. The problem is again mapped to a model where the solution is the lowest energy state of the system. Because D-wave uses multiple qubits, not one optimal solution is found, but rather simultaneously a bunch of good solutions. Running the problem multiple times and comparing the solution each time is needed to find the optimal solution, though the other answers can still be useful depending on the problem.

The Quantum Processing Unit is initialized into the ground state of the problem to be solved by

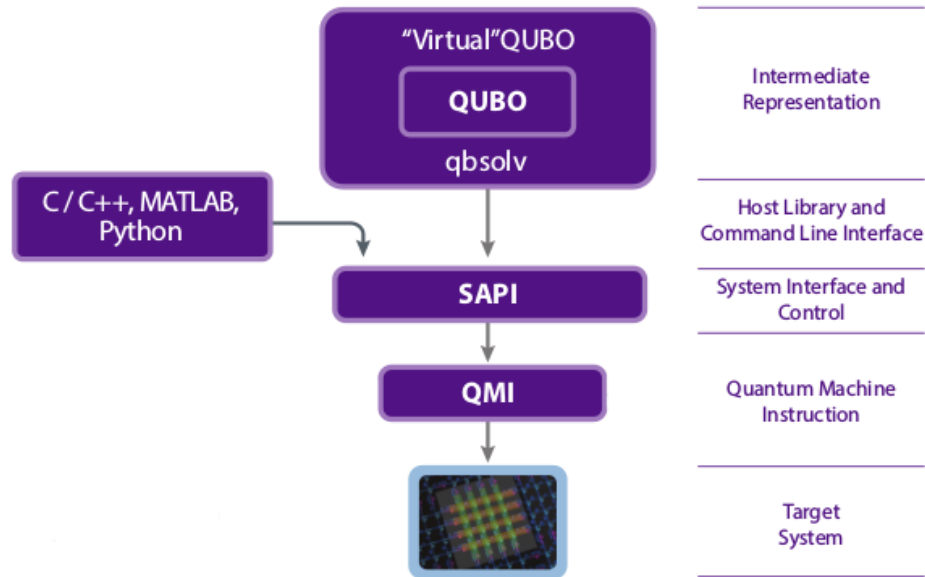


Figure 2.5: The D-wave software environment [11].

bringing the qubits into some superposition and entangling them. So that after annealing, the qubits all end up as either a 0 or a 1, at a low energy state of the system. This low energy state represents an optimal or near-optimal solution to the problem. By repeating this operation, many solutions can be found.

The D-wave 2000Q system has 2048 qubits and 5600 couplers. The couplers are used to link the state of two qubits together. The D-wave system operates inside a fridge at 15 milliKelvin. Like the Fujitsu machine, it is suited to solve complex optimization problems [11]. A representation of the D-wave system is shown in Figure 2.5.

## 2.4. Gate-based quantum computing

This section describes the different gate-based quantum computers that are being developed. There is not currently any company that has achieved so called "quantum supremacy", but many are working towards that goal. Not optimisation techniques are the basis for these quantum computers, but individual qubits that can be manipulated using gates.

### 2.4.1. Rigetti

Rigetti is the first of the true quantum computers that will be discussed. Rigetti offers two kinds of services. They provide Quantum Cloud Services and a quantum programming library, Forest SDK. In both cases, the compiled program can run on Rigetti's superconducting quantum chips, which has up to 19 qubits, or on the Rigetti Quantum Virtual Machine [9].

The Rigetti quantum language is called PyQuil, and is an interface into Python. It is compiled using the Quil compiler and compiles into Quil, their Quantum Instruction Language. PyQuil can be used to write hybrid quantum and classical applications.

### 2.4.2. IBM and Qiskit

Qiskit is the quantum framework of IBM. It is suited for working with noisy qubits, which can be simulated using their simulator, Qiskit Aer. This allows classical simulation of circuits compiled using the Qiskit compiler, Qiskit Terra. It further includes Qiskit Aqua, which is a library of cross-domain quantum algorithms, and Qiskit Ignis, which provides experiments that allow for characterization, verification and mitigation of noise in quantum circuits. Besides simulation, circuits compiled using Qiskit Terra can also be executed on real quantum devices using IBM Q.

Qiskit Terra is an open-source Python library, which compiles circuits written in Python to internal



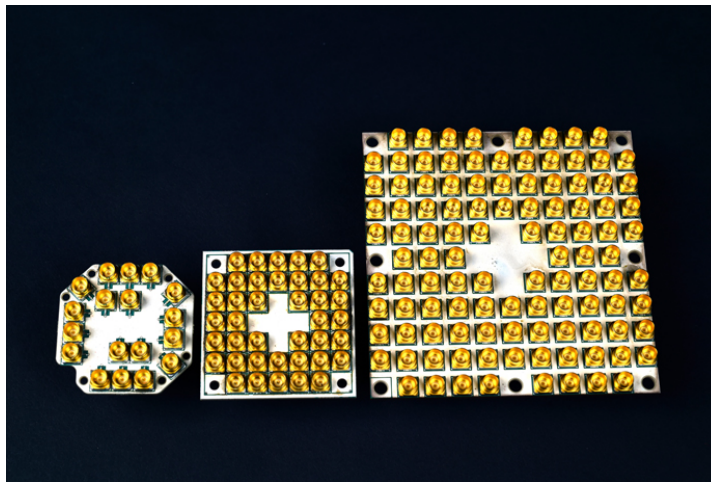


Figure 2.6: Intel Corporation quantum computing test chips with 7, 17 and 49 superconducting qubits [10].

objects or to OpenQASM. These circuits can then be executed using the Qiskit Aer, or using the in-the-cloud IBM-Q devices.

Qiskit Aer is written in c++, and includes tools to configure noise models. This means that realistically noisy circuits can be simulated, which can be configured to be similar to execution on real, noisy quantum devices.

Qiskit Ignis provides characterization experiments, such as for qubit lifetime and qubit dephasing, which can be used to measure noise parameters in the system. The supplied verification experiments can be used to do benchmarking and tomography to verify gates. Mitigation routines can be generated using the mitigation experiments of Ignis.

Qiskit Aqua and Chemistry are libraries that contain quantum algorithms that can be used to build near-term quantum applications. It is built to be extendable, so that different quantum algorithms can easily be added. It currently supplies chemistry AI, finance applications and optimization algorithms [22].

### 2.4.3. Intel: superconducting and spin-qubit quantum processor chips

Intel is developing a quantum processor called Tangle Lake, shown with its two predecessors in Figure 2.6. This processor incorporates 49 superconducting qubits, and must be operated at a temperature near absolute zero,  $-273.15\text{ }^{\circ}\text{C}$ . They are also investigating Spin Qubits, which can operate at slightly higher temperatures of 1 K instead of 0.020 K, and have longer coherence times. In collaboration with QuTech, the quantum framework OpenQL can be used to do gate-based operations on both the superconducting and the spin-qubits [10].

### 2.4.4. Alibaba: Cloud Quantum Computer

Alibaba created a quantum lab in order to develop a superconducting quantum processor. Besides that, they are looking into the balance between running programs on classical computers and on quantum processors [50], and developing a full-stack quantum computer [17]. The Alibaba Cloud quantum computer is capable of processing eleven quantum bits [52]. They are planning to reveal a quantum chip in the next couple of years, but current progress is being kept under wraps [42].

## 2.5. XanaduAI quantum machine learning

The third category of quantum technologies is machine learning techniques, in order to process the extremely large data sets that are currently being generated and assembled in many different fields and for many different applications.

XanaduAI has a machine learning platform for quantum computers called PennyLane. It provides the ability to compute gradients of variational quantum circuits, which extends optimization and machine learning to quantum applications. The framework can be used to optimize variational quantum eigensolvers (VQE), quantum approximate optimization (QAOA) and quantum machine learning mod-

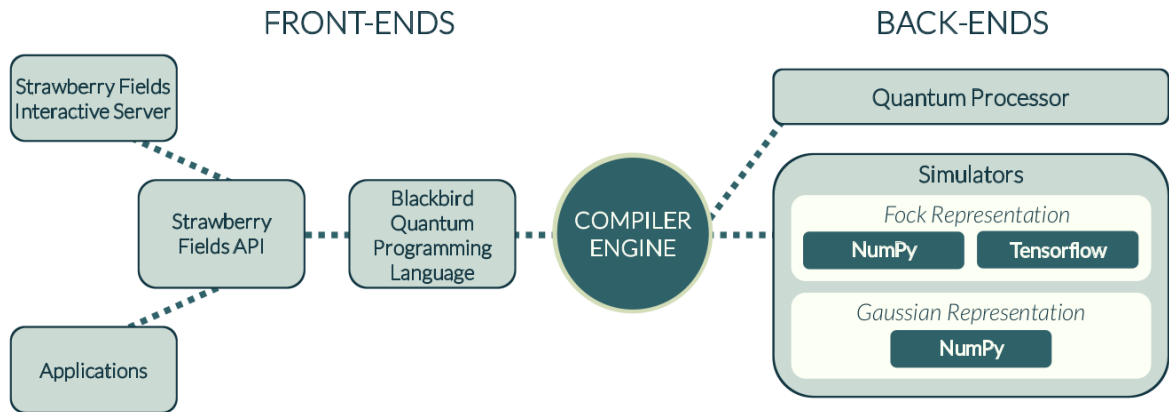


Figure 2.7: The Strawberry Fields software stack [29].

els. There are plugins for PennyLane so that it can be used with Rigetti Forest, Qiskit and ProjectQ, so that applications written using PennyLane can be executed on the quantum devices provided by Rigetti and IBMQ [7].

Xanadu also provides Strawberry Fields, a quantum programming architecture build in Python. Programming is done in the Blackbird programming language. Strawberry Fields targets the Continuous Variable (CV) model of quantum computing, which is suited for simulating bosonic systems or continuous quantum operators. Programs written in Blackbird can be simulated using three simulator back-ends that are included in Strawberry Fields. The first is a Gaussian back-end, which uses numpy to represent CV systems. This can efficiently simulate any computation which is fully Gaussian, but is not expressive enough for general quantum computation. The second back-end is the Fock back-end. This also uses numpy, but for Fock representation, which encodes quantum states using Hilbert Space. A limited number of states is enforced to allow computation. The final back-end is the TensorFlow back-end. It also uses the Fock basis, but uses the strong optimization from TensorFlow [29]. An outline of the software stack of Strawberry Fields is shown in Figure 2.7.

## 2.6. Quantum Software

There is not currently a quantum programming language that is widely used, nor any industry standard for a language that can be executed on any quantum computer. Rather, there are several hardware specific languages, such as those outlined before, and several that aim to be executable on a couple of different (hardware) implementations. One of these is Qubiter, and another is OpenQL, which will be elaborated on in Chapter 4.

### 2.6.1. Qubiter

Qubiter is a very basic set of python tools, which can be used to design and simulate quantum circuits. It is based on the earlier Qubiter from R.R. Tucci [44]. It includes a Cosine Sine Decomposition for quantum gates, so gate decomposition. Additionally, it includes reading and writing quantum files, expanding circuits from multiple controlled gates to CNOTs and single qubit rotation gates, and it includes a simulator [12]. It does not offer much in terms of built-in functionality or algorithms, but it is the basis for the unitary decomposition algorithm that will be implemented in OpenQL.

## 2.7. Conclusion

The focus of most of the languages and companies shown are looking at physical qubits, and the specific problems associated with executing programs on real qubits, such as mapping, error correction and ancilla qubits. Some also look at the algorithmic side of quantum computation, such as Rigetti and IBM with Qiskit, but these do not have unitary decomposition implemented, as can be seen in Table 2.3. So when Unitary Decomposition is implemented in OpenQL, it will be the only quantum programming languages with decomposition and extended algorithmic capabilities.

Table 2.1: Common single qubit gates






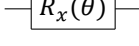
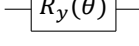
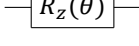

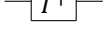








Name	Matrix	Circuit	Relations
Identity	$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$		
Hadamard	$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$		
Pauli X	$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$		$X = iZY = HZH$
Pauli Y	$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$		$Y = iXZ$
Pauli Z	$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$		$Z = iYX = HXH$
Rotation X	$R_x(\theta) = \begin{bmatrix} \cos \frac{\theta}{2} & -i \sin \frac{\theta}{2} \\ -i \sin \frac{\theta}{2} & \cos \frac{\theta}{2} \end{bmatrix}$		
Rotation Y	$R_y(\theta) = \begin{bmatrix} \cos \frac{\theta}{2} & -\sin \frac{\theta}{2} \\ \sin \frac{\theta}{2} & \cos \frac{\theta}{2} \end{bmatrix}$		
Rotation Z	$R_z(\theta) = \begin{bmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{bmatrix}$		
$\pi/4$ -phase gate	$T = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}$		
$T^\dagger$	$T^\dagger = \begin{bmatrix} 1 & 0 \\ 0 & e^{-i\pi/4} \end{bmatrix}$		$I = TT^\dagger$
Phase	$S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$		$S = T^2$

Table 2.2: Common multi qubit gates

Name	Matrix	Circuit	Relations
Identity	$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$		
Controlled not (Pauli X)	$CX_{01} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$		
Controlled not (Pauli X)	$CX_{10} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$		
Swap	$SWAP = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$		
Controlled phase (Pauli Z)	$CZ = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$		$CZ = H_1 CX_{01} H_1$
Toffoli (multicontrolled not)	$CCX = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$		
Fredkin (controlled swap)	$CSWAP = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$		

	Language	Assembly Language	Simulator	Physical system	Compiler	Noisy qubits?	Algorithmic capabilities	Decomposition
Fujitsu	Problem needs to be formulated as an Ising model	-		Classical computers (Digital circuit), on premises and in the cloud	-	-	Annealing	-
Dwave Systems	Python, c++, matlab	Quantum Machine Instructions (QMI)		D-Wave 2000Q	-	-	Quantum Annealing	-
Rigetti	Python	Quil	QVM	Rigetti chips up to 19Q	Quilic	Yes	VQE, QAOA, GFT, Phase estimation, histogram based tomography, grover's search, bernstein-Vazirani, Simon's, Deatsch-Jozsa, Arbitrary state generation	No
Qubiter	Python	Qubiter English	Internal simulator	-	Qubiter	No	<b>Cosine Sine Decomposition</b> , pretty printing	Up to arbitrary size
QisKit	Python	OpenQASM 2.0	QisKit Aer, HPC simulator	IBMQ	Qiskit	Yes	Pretty printing, VQE, QAOA, Eoh, QPE, IQPE, Amplitude estimation, grover's search, Deutsch-Jozsa, Bernstein-Vazirani, Simon	Two-qubit gates, and specific bigger gates
XanaduAI	Python	Blackbird, which is used to program the circuits in the front end	Internal simulator	-	Strawberry Fields	No	No	1 qubit gates
OpenQL	Python	(c/e)QASM	QX	Superconducting and spin qubits	OpenQL	No	(VQE, QAOA, max-cut, <b>Unitary decomposition</b> )	Specific gates (such as Toffoli) and up to arbitrary size

Table 2.3: Comparison of different quantum programming languages and frameworks



# 3

## How does a compiler work?

This chapter aims to give a short introduction on the topic of compilers, which will be useful for anyone unfamiliar with the concept and to provide the basics for the explanation of the OpenQL compiler in Chapter 4. Examples are placed in Appendix C.

### 3.1. What is a compiler?

A compiler translates a high level language into something that can be executed by a computer. They are generally split into two parts: the front end and the back end. The front end takes the program written by a human in a language like Python or C++, and generates an Intermediate Representation (IR). The front end is different for different programming languages, but not for different hardware (*hardware agnostic*). The back end takes the IR and translates it into executable code, so it is different for different types of hardware (*hardware specific*). The back end also manages the memory usage of the program [18].

The structure of a general compiler is shown in Figure 3.1.

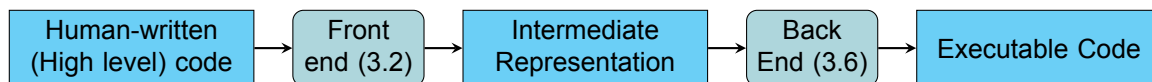


Figure 3.1: The structure of a general compiler

### 3.2. Front End

The front end translates from the target language to the Intermediate representation. This requires parsing the input, checking or resolving the types and resolving references. Some optimizations can be performed at this stage, and finally the generated Abstract Syntax Tree (AST) is transformed into the Intermediate Representation (IR).

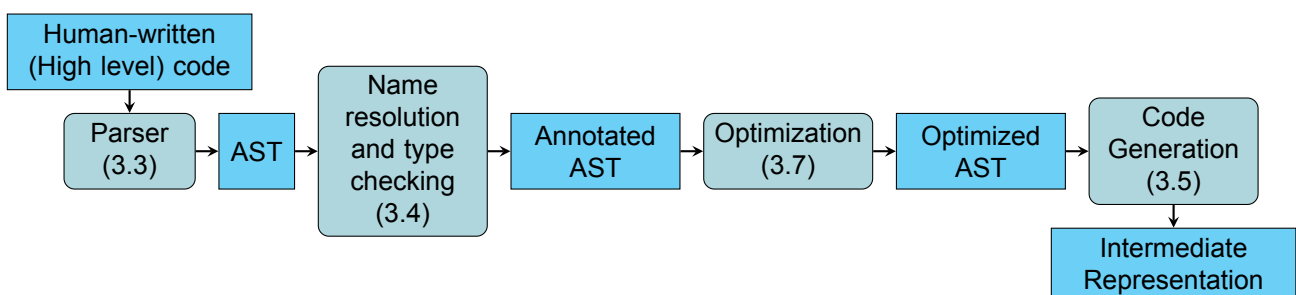


Figure 3.2: The structure of a the front end of a compiler

So, say we have a small program in c++ that we want to execute on some computer. Then the first

step is to translate it into the intermediate representation. An example of this transformation is shown in example C.1.

### 3.3. Parsing

The parser uses patterns to determine what the input text means. These patterns are defined in the language definition of a certain programming language and are called *syntax rules*. Using these rules, the parser then makes an Abstract Syntax Tree from the text. The AST is a representation of the meaning and structure of the written program.

There are two types of lexical syntax rules: lexical syntax and context-free syntax. The first concerns the layout, numbers and what counts as a valid name for variables, functions, classes etc. The second defines the rest of the language rules.

#### 3.3.1. Lexical syntax

When *parsing* the input text, the compiler first needs to recognize what counts as layout and what is the program. Spaces, tabs and new lines are a part of the input text (human-written program), and are important divisions between different words. But there is no difference for the meaning of the program whether there is a single space between two words or whether there are many. This step in parsing takes all the whitespace and replaces it with "layout" tokens. Comments also count as layout, as they do not change the meaning of the program, and are therefore also defined in lexical syntax.

Besides that, the lexical syntax also specifies allowed variable, type name and number formats. These rules can be things like: a variable name is not allowed to start with a number, or: words such as "for" and "int" are not allowed as function names. This rule is used to prevent confusion between an if-statement and a function call to "if" with a boolean parameter.

A processing pass of just the lexical syntax of the example code looks like this:

Listing 3.1: Parsing of text with lexical syntax

```
1 int a = 5;
  ↓
1 int LAYOUT a LAYOUT = LAYOUT 5 SEMICOLON
  ↓
1 typeName LAYOUT varName LAYOUT = LAYOUT num
  SEMICOLON
```

First, the layout is marked. Then, the parser tries to match all of the words it recognizes. The "int" and the "5" match the patterns for typeName and num, so are replaced with tokens that signify that. The "a" is evaluated as a varName, because it matches the varName pattern and it does not match any *reserved words*, such as "if" or "int".

For readability, the LAYOUT token is left out of further parsing examples.

#### 3.3.2. Context-free syntax

Context-free syntax parses the meaning of the rest of the program independent of layout. An example of a context-free pattern is the following: an expression can be a "num"-token from the lexical analysis. So the context-free analysis replaces all the "num" tokens from the lexical analysis with expression tokens: "Exp".

Another pattern in C++ is that an assignment consists of a type name ("typeName"), a variable name ("varName"), an "=" sign and an expression ("Exp"). So the "int a = 5" from the example is parsed as follows:



Listing 3.2: Parsing of text with context-free syntax

```

1 int a = 5;
  ↓
1 typeName varName = num
  ↓
1 typeName varName = Exp
  ↓
1 assignStatement

```

First the lexical patterns are applied. The "int" and the "a" are recognized as a type name and a variable name from the lexical syntax, which is explained in section 3.3.1. The single "=" does not match any pattern, so it is left as-is, and the 5 is recognized as a number. Then the "num" is matched to the "Exp" pattern and replaced. Finally, the parser recognizes and replaces the result with the "assignStatement" pattern.

For a correctly written program, the final result of the parser is a single, final pattern, such as "Program". This signals the compiler that the parsing step is done, as the complete input is now understood. If the parser cannot get to this symbol, then the program has *syntax errors*.

But the goal of parsing is not to replace text with symbols, it is to find the underlying structure of a program. This structure is represented as an Abstract Syntax Tree (AST). The AST from this example follows the same process as the replacing of the words:

Listing 3.3: Transforming text into an Abstract Syntax Tree

```

1 int a = 5;
  ↓
1 typeName("int") varName("a") = Num("5");
  ↓
1 assignStatement(typeName("int"), varName("a"), num("5"))

```

Graphically, this looks like Figure 3.3.

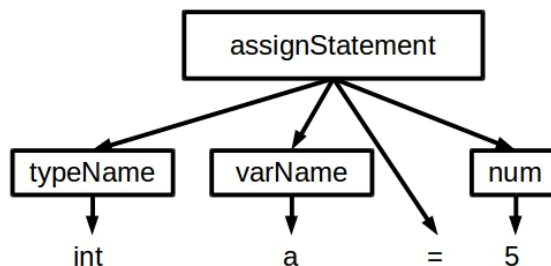


Figure 3.3: Abstract Syntax Tree of the example code.

## 3.4. Type Checking

The type checker takes the AST from the parsing, and checks it against the static semantic rules of the programming language. So it checks whether names can be (unambiguously) resolved, whether the arguments of all operations are as specified and of all variables have the correct type.

### 3.4.1. Name resolution

First, we will talk about name resolution. Say, you have a program like this:

```

1 int a;
2 a = 4;

```

Then the compiler needs to figure out what this "a" on the second line is and where it is defined. In the example, this is rather straightforward. But for bigger programs it is more complicated, as (variable) names can be defined in many places, such as in packages or functions. The rules of the language define where a variable can be referenced from, such a rule can be: "a variable can only be referenced after it is defined" or "a function from a package can only be referenced if the package is included in the program". The *scope* of a name definition is the place where the name is defined and mentions of that name are linked to that definition. This is illustrated in the example from before, where now the variable names are numbered so it is easier to follow. All the `i_number` are the same variable name, and the same goes for the `result_number` and `a_number`.

Listing 3.4: Small code example with numbered variables

```

1 int main() {
2     int result_1;
3     int a_1 = 5;
4     for(int i_1 = 0; i_2 < a_2; i_3++){
5         result_2 = result_3+i_4;
6     }
7     return result_4;
8 }

```

The variable "i" is defined inside the for-loop, and therefore the *scope* of this variable is only the for-loop. If "i" is referenced outside its *scope*, the compiler should give an error, because this is not allowed. The variable "result" is defined outside of the scope of the for-loop, but it is allowed to reference it inside the for-loop, as well as out of it.

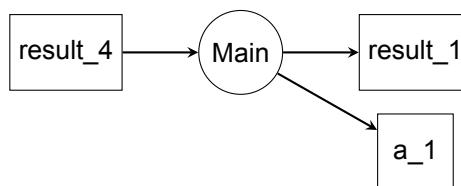


Figure 3.4: The main scope of the example program

The main scope of the example program is illustrated in Figure 3.4. This is without the for-loop, which we will get to later. The main scope has two variable definitions (`a_1` and `result_1`), which is shown by the arrows from the scope (the circle) to the variable names (rectangles). There is one variable reference, the `result_4` name in the return statement, which is shown by the arrow from the variable name to the scope. If there is a path from the reference to the definition by following arrows, the variable can be resolved. So the `result_4` reference can be resolved by following the arrow to the Main scope, and then to the `result_1` definition.

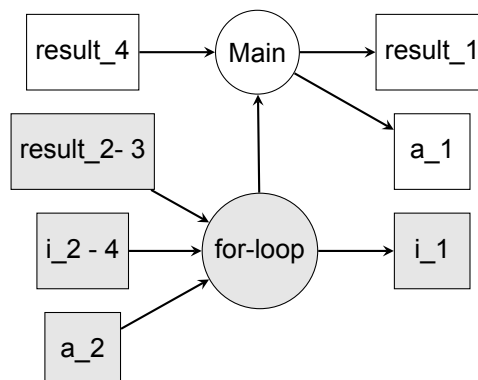


Figure 3.5: The full scope graph of the example program

But the example also has another scope, the for-loop. The full scope graph of the program is shown in Figure 3.5. In the for-loop the variable "i" is defined, and as can be seen from the arrow, this variable cannot be accessed from the Main scope. Accessing a variable from the main scope inside the for-loop is possible, as resolving the reference of "a\_2" can be done by following arrows from a\_2 to the for-loop scope, to the Main scope, to the a\_1 definition.

The language specification defines the scopes and where they can be reached from. The compiler keeps track of all variable declarations and references during the parsing stage, and then determines how to resolve the reference from the rules regarding scopes. If it cannot find a name definition to go along with a reference, it gives: "variable name not found" errors.

### 3.4.2. Types

Besides name resolution, the types of all operations need to be correct for a functioning program. This is where type checking comes in. The AST is analyzed with regards to types, and types for specific operations come from the parsing stage and the rules of the language. Type checking is also necessary for name resolution, because references can be type dependent.

As an example, we will use the same code as at the start of this section:

```
1 int a;  
2 a = 4;
```

From the first line, the compiler puts the type of variable "a" as "int". From the second line, the number also gets the type "int". The rules of the language specify that for assignment the things on both sides of the "=" sign need to be the same type (so "a" and "4"), and that variable references have the same type as the definition (so "int a" and "a"). The type checker determines whether that is the case. If it isn't, the compiler will give *type errors*.

Type checks can be implemented in two ways. As literal (static) checks, where after parsing the whole program is checked against type rules and then gives type errors when a type does not match. (Such as "true > 3") For most languages, however, not everything can be checked statically. This is where dynamic type checking comes in. This is verification of the types at runtime, where a type record is kept for each runtime object. This allows for types of objects to change during execution, and for more kinds of type-casting.

## 3.5. Code Generation

As the last step for the front-end of the compiler, the Intermediate Representation (IR) is generated. This last step takes the (annotated, typed) AST and translates it into IR, which is then passed on to a hardware specific back-end.

Intermediate representations are used to make the code hardware agnostic. It is a low-level representation of the program, but has abstractions for all things related to any specific hardware such as memory locations.

The code generation of the example before is quite lengthy, so we will break it down in parts.

### 3.5.1. Register Allocation

The first step is register allocation. Each variable needs to be associated with a symbolic location in the Intermediate Representation, which can be translated into real locations at the back-end. All the parts of the program that do something with a variable from the AST are transformed into operations that do something with the specific memory location in the IR.

At this stage, there are two levels of *memory hierarchy*. The registers, which are memory located near/on the processor, and the main memory. The main memory is further divided into multiple levels, but this is not relevant for this level of the compiler. A picture of the memory structure is shown in Figure 3.6.

The registers are much faster to access than the main memory, but also have a much lower capacity. Therefore, the values in the registers are used for computations, and for storage the values are written into memory. In the IR that is used in these examples, the registers are numbered like \$2 and the memory as 12(\$fp).

The transformation of the highest level of the example program is shown in Listing C.2.

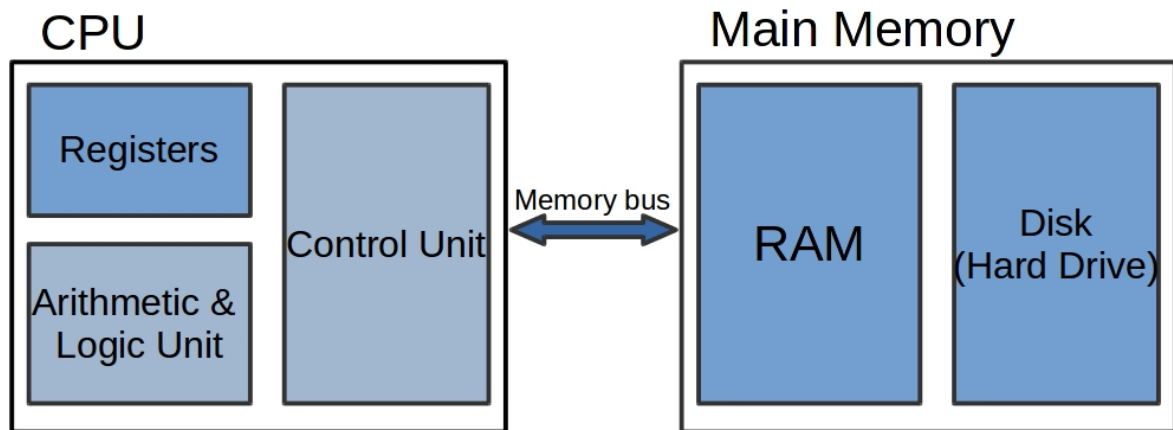


Figure 3.6: Typical memory structure of a computer

The compiler does not translate the input text, but rather the AST into IR. So it assigns a memory/register location to all variables and intermediate results, and then translates the variable assignments and references into load and store operations on these memory or register locations.

The program and return statement constructions are transformed into some IR-specific preamble and end, so that the back-end can use these to make executable code. In this example, this means that the compiler loads the return value into a specific register (\$2).

### 3.5.2. Loops

To execute loops on a low level, the compiler needs to implement the control flow of the program using linear instructions. This is done via labels. There are certain instructions that move the program counter (back) to a specific label based on a condition, these are called jump or branch instructions. The translation of the for-loop of the example in Listing C.1 is shown in example C.3.

This seems complicated, but it can be broken into parts. The first is the end-condition of the loop. So long as this condition is not met, the loop continues. Otherwise, it jumps to the end of the loop. In this case, this is  $(i < a)$ . Then, the code inside the loop is translated. This is  $(result = result + i)$ , which becomes load, add and store instructions. Finally, the end of the loop is evaluated. It increases the value "i" by 1, and jumps unconditionally to the start of the loop.

This IR code can then be further optimized, or directly passed on to a back-end compiler to generate code for a specific architecture, which translates it to zeroes and ones so that it can be executed on a processor.

## 3.6. Back end

The back end of a compiler takes the immediate representation and transforms it into *executable code*. This means translation of the instructions into hardware specific instructions, allocation of memory, transforming symbolic memory locations into real memory locations and outputting this all as zeroes and ones to a processor.

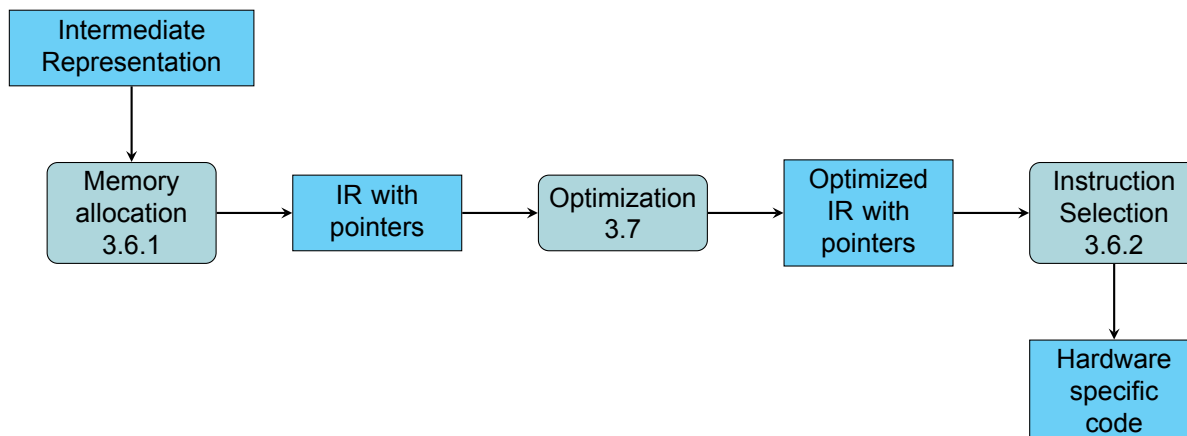


Figure 3.7: The structure of the back end of a compiler

### 3.6.1. Memory allocation and garbage collection

From the front end, each memory or register access has a symbolic identifier. These are translated into pointers to real memory locations. If there are not enough real registers for all of the symbolic registers, the compiler decides which values to put into the registers and which in the memory. Besides that, if there is not enough space on RAM or in the cache for the memory accesses, then some values need to be moved to the disk.

Garbage collection is also part of memory management. This means deleting or writing over values that will not be used anymore. Therefore, the compiler keeps track of all *live variables* (variables that will still be used somewhere) and marks any dead ones so that their memory locations can be reused.

### 3.6.2. Instruction selection

From the intermediate representation, transformation into machine code is fairly straightforward. If the IR is very close or tailored to the target architecture, all that needs to be done is to translate the instructions into *opcodes* and the numbers, registers and memory locations into bits. Opcodes are the bit representation of the instructions, such as "add" or "load word". Furthermore, the labels from jump instructions are replaced with line numbers, so "jump L2" becomes the opcode for the jump instruction and the instruction number of the label L2.

If the IR is not so tailored to the application, the compiler needs to do instruction selection. This means replacement of IR instructions which do not exist on the target platform. It can also replace instructions with the goal of optimizing the program, by replacing them with instructions that are much faster on the specific platform.

For execution of IR on a target system that is completely different from the intended system, a source-to-source compiler may be used. This is explained in Section 3.8.1.

## 3.7. Optimization

Compilers can also optimize the code before it is executed. Optimization of the program can be done in the front-end as well as the back-end.

For optimization in the front-end, the compiler takes the (typed) AST and tries to optimize it for execution time, memory consumption or energy consumption. Doing optimization at the front-end means that the structure of the program can still be used to find possible optimizations. This structure gets lost when the code is compiled to a lower level language. Furthermore, if the optimization results in less code, then the code generation step will be faster [35].

A lot of the optimization of the compiler can also be done after the code is generated, which are back-end optimizations. At this point, it is very clear what exactly the program is going to do, and many redundant, unnecessary or inefficient operations can be removed from the low level code. Besides that, the back-end also can perform optimizations based on the specific architecture the code will be run on. This can include parallelism for multi-core architectures or substituting instructions for faster, architecture specific instructions.

### 3.7.1. Loop optimizations

Code inside a loop will be executed every iteration. However, there might be code inside the loop that will be constant for each loop iteration. As an optimization, the compiler can move these constant operations outside the loop and thereby ensure that they will only be computed once.

```

1 for (i=0; i<10;i++) :
2     x = 4/5
3     y = y + i
    ↓
1 x = 4/5
2 for (i=0; i<10;i++) :
3     y = y + i

```

### 3.7.2. Reusing repeating code

If a computation is performed multiple times, then the result of the computation can be reused. This saves the program from computing something multiple times. Such as:

```

1 x = a + b
2 y = a + b
    ↓
1 x = a + b
2 y = x

```

### 3.7.3. Dead code elimination

With dead code elimination, the compiler tries to eliminate all the parts in the code that will never be executed, and remove them from the AST. This prevents them from translation into machine code, and makes the final compiled program smaller and possibly faster. This includes removing branches in if-else constructions that will never be taken (unreachable code), removing definitions that are never used and code that writes to variables that will never be used (dead variable).

Listing 3.5: Examples of dead code

```

1 #unreachable code
2 if(false):
3     #This code will never be executed
4
5 #assignment to dead variable: a is a dead variable
6 int a = 4
7
8 #assignment to dead variable: a is never used
9 a = a**4
10
11 #function definition that is never used
12 def functionname():
13     return 1

```

In this case, the complete example can be removed without changing the execution of the program, because it is all dead code. Dead code elimination is not always this simple, more complex loop definitions and code in other files that might call on a function definition make it difficult to determine if code is dead code. Therefore, there are more optimizations performed after code generation.

### 3.7.4. Loop unrolling

Conditional jump instructions are relatively slow, because processors usually implement pipelining. This means that the next instructions get executed before the previous instruction is done. Pipelining makes the program faster, but gives a problem for conditional jumps. The condition of the jump influences the next instruction, so the processor needs to wait for the calculation of the condition to finish before it can continue. In low level code, many conditional jumps can be avoided by loop unrolling.

### 3.7.5. Inlining

Function calls are usually computationally expensive (slow) operations. Memory needs to be copied, position in the code needs to be stored and new code needs to be loaded. This all can be avoided by simply moving the code inside a function into the main program.

### 3.7.6. Combining and removing instructions

Dead code can be eliminated just like before from the IR. Loop unrolling also might reveal reusable code, and instructions can be combined into fewer instructions. Depending on the hardware, instructions can also be replaced by faster instructions. This can include replacing the loading of integers with the value from a constant register if one is present. Multiplications by a power of 2 can also be replaced by a shift-operation, which is much faster.

## 3.8. Different kinds of compilers

Besides the "basic" compiler outlined above, there are many more kinds of compilers. Some of these will be outlined below, as they are useful to illustrate the wider use of compilers.

### 3.8.1. Source-to-source compilers

Source-to-source compilers are also known as transcompilers or transpilers. Compilers like these translate between high-level languages. For very similar languages, this might be a simple compiler pass that slightly changes the syntax and is done. But to translate a program written in one language into a completely different one, the compiler will need to make the (annotated, typed) AST for the source code, transform this into an AST for the target language, and then remake text from this AST.

Source-to-source compilers can be used to compile sequential programs into a program that supports parallelism, or other kinds of (hard) optimizations. The compiled code can easily be compiled by another compiler, and the output code is more readable for an end user than some low-level code [3].

Other uses for a source-to-source compiler are to update old code into a newer version of the language, or to quickly develop a new programming language, as then the back-end of another programming language can be used. This was done for early implementation of C++, for example [20].

### 3.8.2. Machine code compilers

A byte code compiler generates code for a virtual machine. It translates a high-level language into a simple low-level language. This allows for developing a program in a high level language, and execution of the resulting byte code is hardware agnostic. This is thus very similar to a "normal" compiler, except that the resulting code is meant to be interpreted and run on a virtual machine rather than directly compiled to hardware specific code [43].

### 3.8.3. Just-In-Time compilers

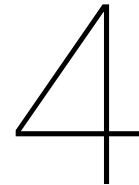
This is a compiler that compiles the code at runtime. This can be the source code (which is usually a high level language) or the intermediate representation. This improves the performance when the intermediate representation would otherwise need to be interpreted by a virtual machine. The Just-In-Time compiler can speed-up the application by removing the need for the interpretation, without needing additional time in the compilation step. It might need more time for start-up, when the application calls many classes or methods that need to be compiled for the first time [21].

### 3.8.4. Hardware compilers

These kinds of compilers do not produce a sequence of instructions as output, but rather the configuration for a certain type of hardware. These kinds of compilers target programmable hardware configurations, such as Field Programmable Gate Arrays [26].







# OpenQL

OpenQL is a library for C++ and Python developed by the QCA-group. It facilitates the use of both of quantum computation specific features and classical programming features from these languages. Quantum algorithms can be specified with quantum gates, which specify physical operations that are executed on (virtual) qubits. OpenQL is meant for hybrid computations, such classical programs which are used to assemble specific quantum circuits, or classical programs which call on and use the result of quantum circuits.

OpenQL can be used to manipulate qubits at different levels, from higher level operations to specifying specific gates. The high-level input code is translated into the common Quantum Assembly Language (cQASM), which can be executed on the QX simulator. It can also generate executable QASM (eQASM), which can be executed on any physical quantum processor that supports it. OpenQL aims to be a framework that can generate output in several types of QASM, so that the high-level implementation can be formulated independently from the execution platform. For this thesis, the code generated will be cQASM, which will be executed using the QX-simulator platform.

## 4.1. Quantum computers as accelerators

Although quantum computers can do everything a classical computer can do and more, the current focus is quantum computers as accelerators: to use a quantum computer to solve problems that take (too) long for a classical computer. In order to facilitate this, the OpenQL language functions as a package in high-level languages C++ and Python. This will allow classical code to run on the host computer, while using a quantum computer as solver. This is similar to the way other accelerators are used, such as Graphical Processing Units (GPUs), Tensor Processing Units (TPUs) and Field-Programmable Gate Arrays (FPGAs). Usually, the CPU is used to divide the problem into pieces and compute the sequential parts of the problem, and the pieces that can be computed parallel are computed on the accelerator. A graphical representation of external solvers like this is shown in Figure 4.2.

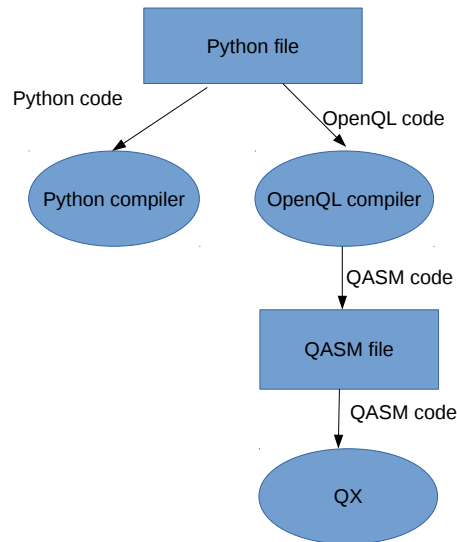


Figure 4.1: How a Python program which uses openql is handled

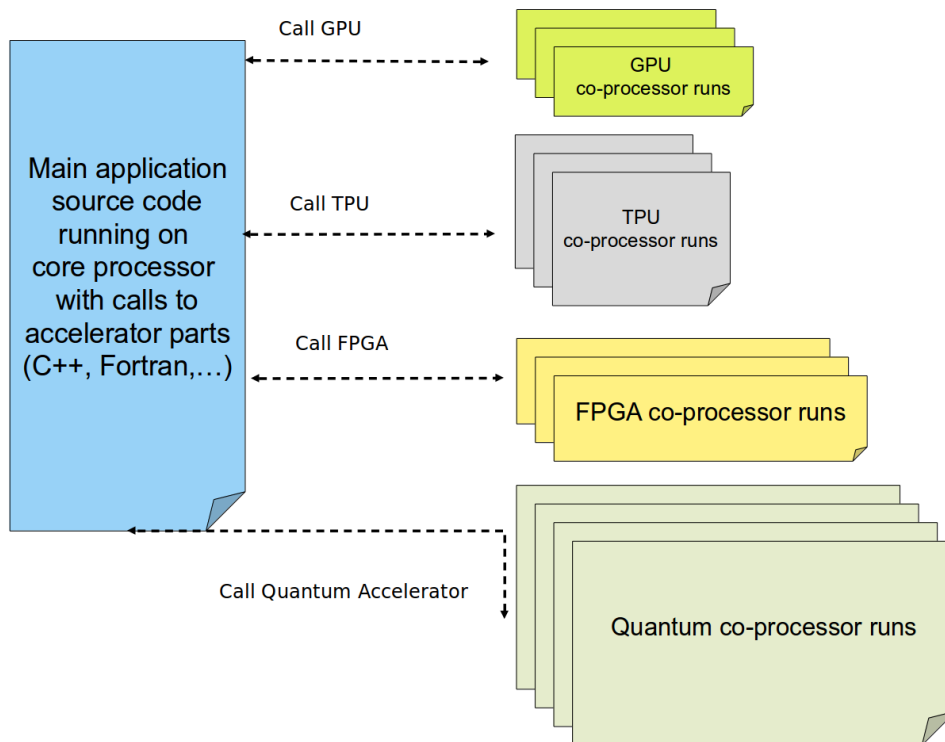


Figure 4.2: The main application can call on a different architecture as an accelerator

A quantum accelerator can be accomplished using similar systems as classical accelerators, since quantum computing functions in a way similar to parallel computing. Therefore, a framework that combines classical computations on the host computer with calls to the quantum accelerator is being developed.

## 4.2. An OpenQL program

In Listing 4.1, an example is shown of a simple Python program that uses OpenQL. In this example, OpenQL is used to make a qasm program that generates a Bell pair. This program is compiled to the code shown in Listing 4.2.

Listing 4.1: A generic program written using OpenQL

```

1 import os
2 from openql import openql as ql
3 import numpy as np
4
5 curdir = os.path.dirname(__file__)
6 config_fn = os.path.join(curdir, 'hardware_config_qx.json')
7 platf = ql.Platform("starmon", config_fn)
8 output_dir = os.path.join(curdir, 'test_output')
9 ql.set_option('output_dir', output_dir)
10
11 nqubits = 2
12 k = ql.Kernel("newKernel", platf, nqubits)
13 k.hadamard(0)
14 k.cnot(0,1)
15 k.display()
16
17 p = ql.Program('example', platf, nqubits)
18 p.add_kernel(k)
19 p.compile()

```

Listing 4.2: The output from the example program in Listing 4.1

```

1 version 1.0
2 # this file has been automatically generated by the OpenQL compiler
   please do not modify it manually.
3 qubits 2
4
5 .newKernel
6     h q[0]
7     cnot q[0],q[1]
8     display

```

### 4.3. Components of an OpenQL application

An OpenQL program consists of different parts. A separate configuration file is used to set the properties of the *Platform*. On this platform, a *Program* is defined. This program consists of one or more *Kernels* and has a set number of qubits. On each kernel the quantum gates are defined. The kernels are then added to the program, and the program is *compiled*. An output directory can be set as a general setting, although this is optional. If it is not set, a default output directory will be created and used. A graphical representation of this structure is shown in Figure 4.3. The explained components are the config file, the platform, the program and the kernel.

#### Config file

The configuration of the quantum system is specified in a JSON file, which has the information on every allowed operation and its arguments. If a configuration for a specific operation is not specified, it is not allowed. This is especially important for running on physical qubits, but for running on the QX simulator a configuration file is also mandatory. For each operation, it defines the duration, the latency, the input qubits, the transformation matrices, whether optimizations are allowed, the type of instruction, and the name for the specific instruction for the specific compiler.

#### Platform

The quantum platform specifies the target hardware of the compilation. A configuration file needs to be set, in which the eQASM compiler is set, as well as the hardware settings, all allowed instructions, gate decompositions, resources and the topology. These are used to compile the programs in a way where they can be executed on the specified platform. When executing using a simulator, very few restrictions

are present, but current quantum hardware does not allow all possible operations on all qubits, and can have many other directions [27]. In this way, the platform represents the quantum implementation on which the generated qasm code will be executed.

## Program

The quantum program is the container that implements the whole quantum algorithm. It executes on a specific platform, and the maximum number of qubits are set for the whole program. A program can consist of one or more quantum kernels. Kernels can be added multiple times to the program, in which case the corresponding QASM will be duplicated. The program is compiled to produce QASM with the "compile()" function, which also generates the QASM code and prints it to an output file which is placed in the (specified) output directory [27].

## Kernel

The quantum kernels implement the functional blocks. Qubits, gates and other operations are specified on the kernel. This allows for reuse of recurring blocks of code, and more modular programming. To compile a kernel, it needs to be added to a program [27]. It is also possible to add display commands, which mean the current state of the qubits is shown when the QASM is executed using the QX simulator.

## Gates

These are the allowed operations to add to a kernel.

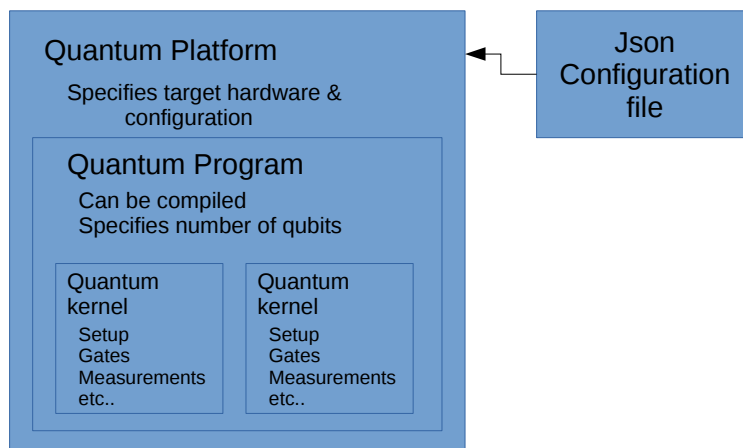


Figure 4.3: The structure of an OpenQL program

## 4.4. Compilation steps

OpenQL consists of two main layers. The first is the high level programming interface, which translates C++ or Python instructions into cQASM. This is accomplished by several compilation passes which decompose, optimize, schedule, map and produce QASM. This QASM can then be executed by the QX simulator, or be passed on to the next layer. This layer translates cQASM into hardware specific eQASM for specific hardware implementations.

This means that high level code does not need to be changed for different hardware implementations and that the lower layer of hardware specific compilation can be adapted to or added to changing architectures and needs. The basic structure of the OpenQL compiler is shown in Figure 4.4, and the compile order in Figure 4.5. The compilation steps are, in order: decomposition, optimization, scheduling, mapping and QASM generation.

### 4.4.1. Decomposition

If the gate decomposition of a gate is specified in the configuration file, the gate is decomposed before adding it to the kernel. For each gate, it checks whether there is a specialized decomposition available.

Table 4.1: Supported OpenQL kernel operations [28].

Operation	Parameters	Explanation
barrier	qubits	Insert a specific barrier on the specified qubits
classical	CReg, Operation	Add a classical operation kernel
clifford	id, q0	Add a clifford operation of the specified ID
cnot	q0, q1	Add a controlled not gate
conjugate	k	Generate conjugate version of the kernel
controlled	k, control qubits, ancilla qubits	Generate a controlled version of the kernel
cphase	q0, q1	Add a controlled phase gate
cz	q0, q1	Add a controlled phase gate
display	-	Display the current state (only for simulators)
gate	*args	Add a gate
get_custom_instructions	-	Get all possible custom instructions
hadamard	q0	Add a hadamard gate
identity	q0	Add an identity gate
measure	q0	Measure the specified qubit
mrx90	q0	
prepz	q0	Prepare the specified qubit in a zero state
rx	q0, angle	Add a rotation-x gate
rx180	q0	Add a rotation-x gate with a rotation angle of 180°
rx90	q0	Add a rotation-x gate with a rotation angle of 90°
ry	q0, angle	
ry180	q0	Add a rotation-y gate with a rotation angle of 180°
ry90	q0	Add a rotation-y gate with a rotation angle of 90°
rz	q0, angle	Add a rotation-z gate
s	q0	Add an s-gate
sdag	q0	Add an s-dagger gate
t	q0	Add a t-gate
tdag	q0	Add a t-dagger gate
toffoli	q0, q1, q2	Add a toffoli gate
wait	qubits, duration	Add a wait of the specified duration
x	q0	Add a Pauli-x gate
y	q0	Add a Pauli-y gate
z	q0	Add a Pauli-z gate

First it transforms the instruction into a parameterized name, so that the decomposition is independent from the qubit it acts upon. Then, it tries to find a specialized composite gate in the gate definitions of the kernel. If the gate is a composite gate type, the sub-instructions are loaded. For each sub-instruction, either a custom gate is added if it is available, or a default gate is added if it is available and the option for it is set by the user.

If the option is set, the Toffoli gate is also decomposed into a defined set of one- and two-qubit gates, as shown in Figure 4.6. It is necessary to decompose multi-qubit gates for the later mapping stage, which can only map one and two-qubit gates. This step is not necessary for execution on a simulator which allows multi-qubit gates and does not require mapping, such as QX.

The decomposing stage also tries to replace any operation that is not supported by the target platform with an equivalent operation if one is available.

#### 4.4.2. Optimization

In this step, the compiler optimizes the whole circuit, except for measurement instructions. If a circuit contains measurement operations, it is decomposed into basic blocks consisting entirely of either prepz gates (preparation in the zero state), measurement gates or all other gates. Each circuit is evaluated

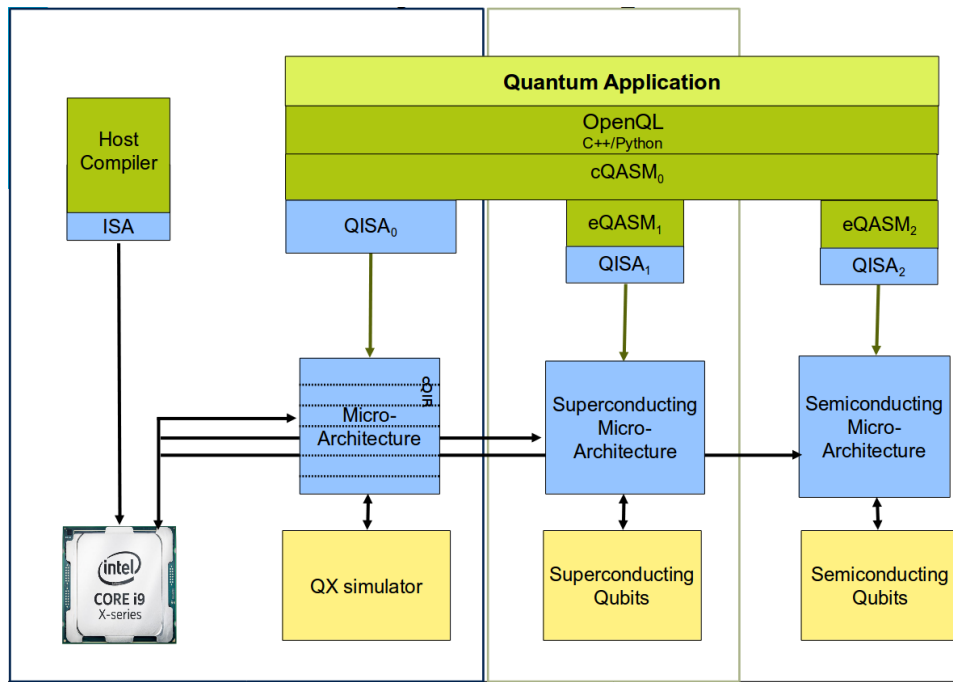


Figure 4.4: OpenQL Compiler Architecture

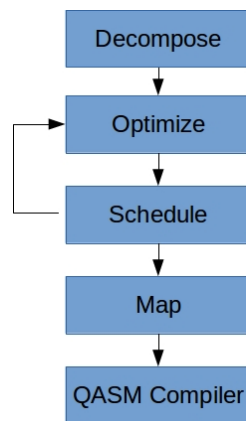


Figure 4.5: OpenQL compile order

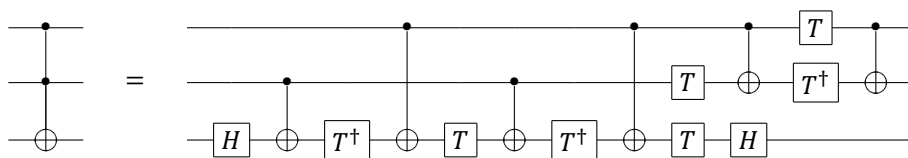


Figure 4.6: Toffoli gate decomposition [27].

and all the non-measurement instructions are passed to the optimize function. All gate dependencies are analyzed and transformed into a Direct Acyclic Graph. This allows for extracting parallelism, as well as reordering gates with respect to dependencies and to extract local gate sequences which can be fused into smaller sequences to reduce the overall gate count. The latter is done through simple matrix multiplication and then approximated using a new equivalent operation. The precision of this approximation can be set by the user. It uses a sliding window in which the gates are multiplied, and if this results in the identity matrix, the gates are removed from the circuit. Applying an identity gate to a qubit does not change its state, and removing the gate does not affect the outcome of the circuit.

### 4.4.3. Scheduling

The gate dependency can also be used for the scheduling, to determine which operations can be scheduled in parallel. There are three options for gate scheduling, As Soon As Possible, As Late as Possible and Uniform As Late As Possible. ASAP scheduling may result in many gates executed at the start of the circuit and lower fidelity due to longer cycles between successive gates operating on the same qubit. ALAP scheduling may result in a lot of gates at the end of the circuit, which may result in buffer overflows and local feedback systems holding more gates off, thus resulting in a longer total execution time. Uniform ALAP scheduling addresses this problem by starting with an ASAP schedule and then filling cycles with gates by moving them towards the end while respecting the dependencies. The duration of each gate is taken from the platform's configuration file. The scheduler puts a wait statement after each instruction with the duration of that instruction.

### 4.4.4. Mapping

The physical layout of the quantum chip restricts the possible interactions between the qubits, which can make mapping necessary. Since two-qubit gates can only be done between neighbouring qubits, routing is often required to allow these operations. OpenQL supports this using two algorithms: one that aims to find the optimal initial placement of the qubits, and one that replaced two-qubit gates on non-neighbouring qubits with a set of two-qubit gates on neighbouring qubits.

### 4.4.5. QASM Compilation

The QASM is produced by calling the `qasm()` function for the whole program, which generates the preamble of the program. Then, the `qasm` function of each kernel is called. This makes the kernel preamble, calls the `qasm()` function for each circuit in the kernel, and then generates the kernel end code depending on the kernel type. Each instruction definition has their own QASM definition, which returns a string that is the QASM equivalent of the OpenQL instruction. This string includes the qubits the operation acts upon.

Each intermediate step in the compilation produces valid QASM code, and the compiler writes these files to the specified output directory.

### 4.4.6. Compilation to specific back-ends

Current available back-ends are CBox and CCLight, which compile eQASM for a quantum processor based on spin qubits and superconducting qubits respectively. Which (if any) eQASM compiler is used is specified in the json configuration file. This also specifies the allowed instructions for the specific hardware implementation, as well as their eQASM-equivalent, duration and microcode specification.

## 4.5. QASM

The umbrella-term Quantum Assembly Language contains two different types of QASM: cQASM and eQASM. cQASM is the common quantum assembly language. It is a hardware agnostic language, so it is independent of the specific (micro-)architecture or allowable operations of different quantum implementations. It can be executed on QX, a quantum simulator, without any modifications.

cQASM can also be transpiled to executable QASM (eQASM). eQASM encompasses several assembly languages specific to different hardware implementations. To compile cQASM to eQASM, instructions might need to be replaced by several others when they cannot be executed on the hardware. Additionally, the code needs to be scheduled in accordance with the execution times for each instruction, and mapped, because two-qubit operations can only happen between neighbouring qubits. Therefore, SWAP operations need to be inserted, or qubits need to be arranged differently at the start of the application.

A list of supported quantum gates in cQASM 1.0 is shown in Table 4.2. For cQASM 2.0, support will be added for classical instructions, registers, branching and possibly dynamic or parameterized qubit addressing. Because cQASM 2.0 is still being worked upon, and it is therefore not supported yet, this thesis will concern itself only with cQASM 1.0. The aim is to implement unitary decomposition in such a way that it can be easily ported to cQASM 2.0.

Table 4.2: Supported Quantum Gates in cQASM 1.0 [46].

Gate	Description	Example
I	Identity	i q[2]
H	Hadamard	h q[0]
X	Pauli-X	x q[1]
Y	Pauli-Y	y q[3]
Z	Pauli-Z	z q[7]
Rx	Arbitrary x-rotation	rx q[0], 3.14
Ry	Arbitrary y-rotation	ry q[3], 3.14
Rz	Arbitrary z-rotation	rz q[2], 0.71
X90	X90	x90 q[1]
Y90	Y90	y90 q[0]
mX90	mX90	mx90 q[1]
mY90	mY90	my90 q[0]
S	Phase	s q[6]
Sdag	Phase dagger	sdag q[6]
T	T	t q[1]
Tdag	T dagger	tdag q[3]
CNOT	CNOT	cnot q[0],q[1]
Toffoli	Toffoli	toffoli q[3],q[5],q[7]
CZ	CPHASE	cz q[1],q[2]
SWAP	Swap	swap q[0],q[3]
CRK	Controlled Phase Shift ( $\pi/2^k$ )	crk q[0],q[1],k
CR	Controlled Phase Shift (arbitrary angle)	cr q[0],q[1],angle
c-X	Binary-Controlled X	c-x b[0],q[2]
c-Z	Binary-Controlled Z	c-z b[1],q[2]
Measure	Measurement	measure q[4]

A QASM file looks like this:

```

1 version 1.0
2 # this file has been automatically generated by the OpenQL compiler
  please do not modify it manually.
3 qubits 2
4
5 .newKernel
6   prep_z q[0]
7   prep_z q[1]
8   h q[0]
9   y90 q[0]
10  cz q[0],q[1]
11  y90 q[1]
12  measure q[0]
13  measure q[1]
```

It specifies the version of the cQASM and the number of qubits, and then one or more quantum kernels. The single kernel in this example has the name "newKernel". In this kernel, the operations on the specific qubits are defined.

## 4.6. Compiler structure

This section concerns itself with the nitty-gritty details of the compiler itself. The structure here is thus not the high-level compiler structure detailed before, but rather the structure of the compiler files itself.

OpenQL is a module written in C++, that is connected using SWIG. SWIG parses the C++ interface and generates "glue code" that enables Python or C++ applications to use OpenQL as an importable module. Besides this, SWIG also enables testing of the OpenQL compiler using the high-level python



test-bench [45].

To use OpenQL functionality in python, the module "openql" is imported from OpenQL (openql.openql). This interface passes calls of these classes and methods to the next layer. This is the ql-namespace, which describes the functionality of the compiler. Including the compile() function, which defines all the compilation steps as described in section 4.4.

### 4.6.1. openql/openql.h

This openql.h file defines the interface between Python/C++ and the compiler. It defines the Platform, CReg, Operation, Kernel and Program classes and their submethods.

```
1 from openql import openql as ql
```

This module gives direct access to the following classes: Platform, CReg, Operation, Kernel and Program. And these functions: get\_version(), set\_option(..), get\_option(..) and print\_options(). A minimal example of how these are used is down below:

```
1 ql.get_version()
2
3 config_fn = "path/of/config/file"
4 platf = ql.Platform("starmon", config_fn)
5
6 output_dir = "path/of/output/directory"
7 ql.set_option('output_dir', output_dir)
8 ql.get_option('output_dir')
9 ql.print_options()
10
11 creg = ql.CReg()
12 operation = ql.Operation(0)
13
14 nqubits = 2
15 k = ql.Kernel("newKernel", platf, nqubits)
16
17 p = ql.Program('test_cqasm_test', platf, nqubits)
```

These classes and functions are defined in "openql.h" in the openql folder. This folder is used by Swig to compile the files into a Python/C++ package. In order to do this, Swig needs an interface file. This file is called "openql.i" and defines the interface to all functions and classes defined in "openql.h".

The function of openql/openql.h is to pass calls to the package on to the "ql" namespace, where all the functionality is defined. So openql/openql.h defines the functions, classes and methods of these classes as they are used from a Python or C++ program. To illustrate this, below a code snippet from the definition of the Program class. When openql.openql.Platform(...) is called, a new quantum\_platform (from the ql namespace) is defined. Calls to methods of the Program class are translated to calls to the quantum\_platform.

Listing 4.3: Definition of the "Platform" class

```
1 class Platform
2 {
3 public:
4     std::string      name;
5     std::string      config_file;
6     ql::quantum_platform * platform;
7
8     Platform() {}
9     Platform(std::string name, std::string config_file) : name(name),
10    config_file(config_file)
11    {
12        platform = new ql::quantum_platform(name, config_file);
13    }
```

```

13     size_t get_qubit_number()
14     {
15         return platform->get_qubit_number();
16     }
17 };

```

### 4.6.2. The definition of a gate

The `ql` namespace, in the "ql" folder, contains the definitions of all of the functionality of OpenQL. So the previous `ql::quantum_platform` call is defined in the `platform.h` file. Generally, each class has their own header file in which the functionality is implemented.

Different functionalities are: defining the various settings and fields of the class, types and methods of these classes. The structure from the `openql` namespace is preserved, and all elements that need to be compiled to QASM have a `qasm()` method which returns the appropriate QASM code when called.

Special case: when a gate is called from a specific kernel, like so:

```

1 kernel.hadamard(0)

```

The `hadamard` function from the `kernel` class gets called, which is defined like this:

```

1 void hadamard(size_t qubit)
2 {
3     gate("hadamard", {qubit} );
4 }

```

So calling a specific gate gets translated to the general "gate"-function. This function checks whether the qubit number is valid, and then whether there is a decomposition available for this gate. Finally it tries to add the gate, "hadamard" in this example, as a custom gate. This means that it checks it against the gates defined in the configuration file. If that is not possible, it tries to add it as a default gate, or this function fails and the compiler returns an exception.

It adds custom gates to the kernel by instantiating a new `custom_gate` object, and setting the parameters it gets from the configuration file. For a default gate, it instantiates a new instance of the class associated with the gate. After getting the new instance, it adds the gate to the "circuit" of the kernel. When the `qasm()` function of the kernel is called, it iterates through all the objects in the circuit and calls the `qasm()` function for each. It concatenates all the strings it gets from each gate, as well as kernel pre- and post-amble. This string is returned to the Program, and printed to generate QASM files.

For a default gate, like the `hadamard` gate, this looks like this:

```

1 if(gname == "hadamard")
2 {
3     c.push_back(new ql::hadamard(qubits[0]) );
4 }

```

Each default gate is a derivation of the "gate" class and has an initialisation, `qasm()`, `micro_code()`, `type()` and `mat()` method. These are used in generating the cQASM and eQASM, generating micro-code which is used for hardware, and for supporting compiler functionality and optimizations.

### 4.6.3. Compilation

To compile the quantum part of the program, and to generate the `.qasm` files, the `compile()` function is called on the Program object. This function goes through the main compiler steps from Section 4.4. First, it will check if the kernels are empty, and then whether they should be optimized. If so, it optimizes all the kernels. The same goes for the decomposition of any Toffoli gates. Then, it writes the cQASM to the specified output file. After that, it calls the `schedule` function to schedule the `qasm` according to the specifications in the configuration file. This is done by inserting a wait statement for each gate. If an eQASM compiler has been specified, that compiler is called to generate the specified eQASM. If not, the compiler checks for and writes any sweep points to the file. This concludes the compilation.

# 5

## Quantum Genome Sequencing

In this chapter, DNA and genome sequencing will be explained first. Then, a recap of different quantum genome sequencing algorithms will be given.

### 5.1. Genome Sequencing

One thing that all of the life on earth has in common is that it is based on genetic code. For classical computers, computer code is represented as zeroes and ones. For quantum computers, the quantum code are the energy levels and phases of single atoms: qubits. But for organic life, this code is made of four bases, or two pairs: Adenine (A) and Thymine (T), Cytosine (C) and Guanine (G). These are what is called: Deoxyribose Nucleic Acid: DNA, and it is stored as a double helix in every single living cell. Each strand consists of a sequence of A, T, G and C, and connected to the complementary base on the other strand. So A to T, and G to C. This is illustrated in the top strand in Figure 5.1. The strands are split from each other when copying the DNA, which is done when a cell divides into two, or when encoding the genetic code into Ribo Nucleic Acid (RNA). RNA is used to facilitate chemical reactions in cells and to synthesize proteins, which is how the DNA determines everything going on in (and around) the cell [8]. A basic representation of this process is illustrated in Figure 5.1.

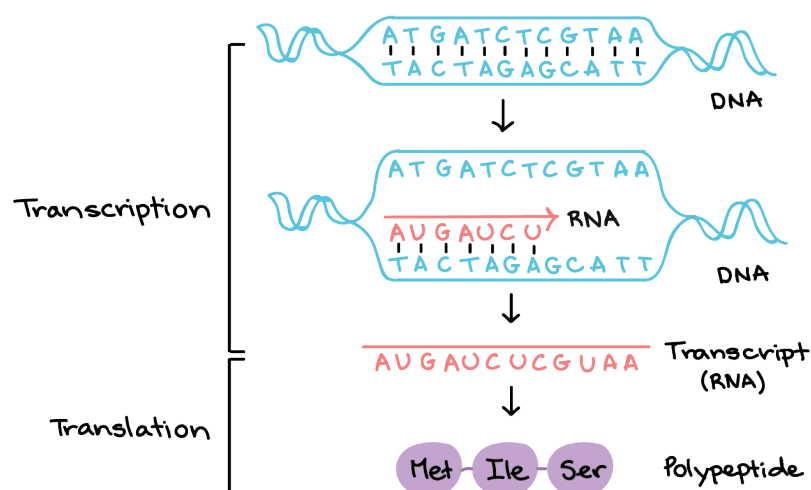


Figure 5.1: Synthesis from DNA to RNA to a protein [2].

In order to find out the specific order of the A C, T and Gs in a DNA sample, the DNA is made into many copies of short pieces of the sequence. These short reads range from 50-700 nucleotides in length [1]. Various methods can be used to read the sequence of the short reads, such as the

454 GenomeSequencer FLX instrument, the Illumina Genome Analyzer, the Applied Biosystems ABI SOLiD system, or the Helicos single-molecule sequencing device from HeliScope [5].

After acquiring the short reads, the original sequence of the DNA needs to be puzzled back together. Algorithms that do this fall into two basic categories: De Novo sequencing and Ab-Initio sequencing. Ab-Initio sequencing maps the short-reads to a reference genome. It tries to find the location on the reference with which the short read shares the most base-pairs. This is possible because the DNA between two individuals of the same species is extremely similar. On a total base-pair count of 3.2 million, a typical human genome is only different from another typical human genome in 20 million places. So only 0.6% of all the base-pairs are actually different between (most) people [15].

De-Novo sequencing is used if there are no reference genomes available or to remove the bias introduced by using a reference genome. The DNA is assembled from the short reads by looking at the overlap between the sequences [37]. This is possible because the DNA is copied multiple times before it is chopped into the short reads, so the sequence of a single location is present into multiple short reads. By comparing the sequences, the overlapping parts can be found and then used to stitch the DNA sequence together.

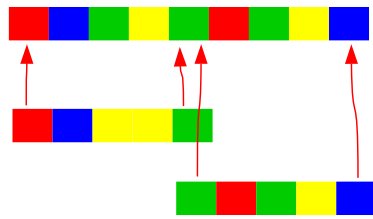


Figure 5.2: Sequencing DNA using a reference genome

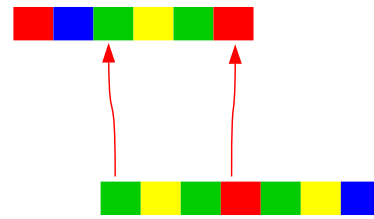


Figure 5.3: Sequencing DNA De-Novo

After the DNA has been read and the short reads have been assembled into a full genome, the DNA analysis can be started. This process is illustrated in Figure 5.4 and encompasses pre-processing, variant calling, recalibration and evaluation. The pre-processing starts with the DNA, making and reading the short reads, and then the assembly into a full DNA using Map-To-Reference or De-Novo sequencing. The probably overlap between different short reads is identified, and used for Base (Quality Score) Recalibration to find and correct any systematic errors in the confidence score coming from the sequencer. This is important for the decision making during the variant discovery process.

During this process, the full assembled sequence is compared to a reference. And rather than look at the similarities, the differences are marked. The calibration is used to filter the variants based on confidence level, which is needed to reduce false positives. The last step is annotating the variants, refinement and evaluation.

Finally, the variants are annotated, refined and evaluated. From these steps, the one that is most easily computed in parallel is the data pre-processing. Getting the raw-reads can be parallel depending on the implementation. A technique like Illumina reads multiple DNA fragments in parallel, for example [5]. The Map-To-Reference or De-Novo assembly step is a computing intensive step that can be computed in parallel, so this is the step that is being looked at to implement as a quantum algorithm [38].

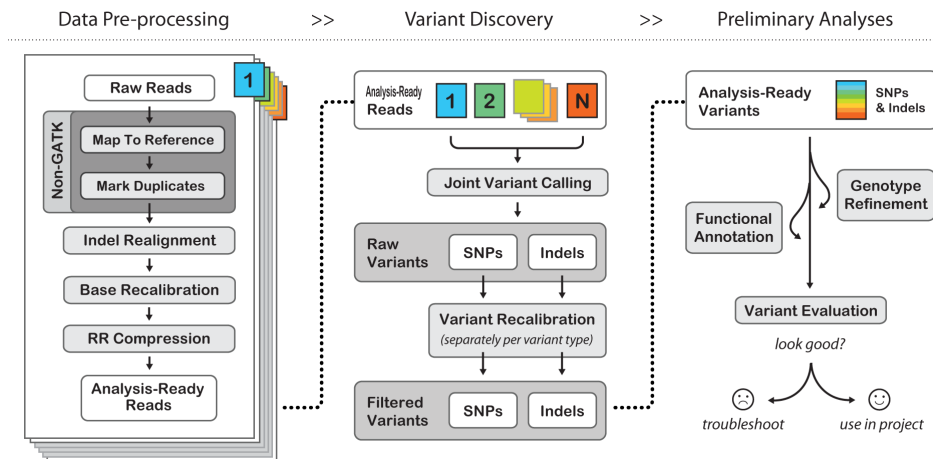


Figure 5.4: Broad Institute GATK best practices pipeline

## 5.2. Quantum algorithms for genome sequencing

Genome sequencing is in its essence a big-data problem, simply because full genomes are very long. Executing the whole data-pipeline can take multiple days for human DNA. But it is also a problem that can be executed very well in parallel. For both De-Novo and Map-To-Reference algorithms, the short-reads need to be compared to many other parts of the DNA. Either all locations in the reference genome, or many other short-reads. Since quantum computing means evaluating many possible answers at the same time, Quantum Genome Sequencing looks at ways to accelerate the DNA sequence reconstruction using quantum computing. In [38], the following algorithms were considered and evaluated.

### 5.2.1. Grover's search

In Grover's search, the initial state is a superposition of all the possible strings the size of the search pattern. If there is exactly one item which matches the search criteria, the number of queries to the algorithm is reduced to the square root of the worst case for the classical number of queries. So it takes a search from  $O(N)$  to  $O(\sqrt{N})$  queries. As stated before, the search starts out with an equal superposition of all states. Then, an Oracle function is called, which produces a yes or no answer regarding whether the given input is the correct one. In the case of Grover's search, the Oracle rotates the phase of the answer by  $\pi$  radians. The Oracle is thus an assumption of a perfect algorithm that can somehow determine whether the input is the correct one, without any of the limits of using an actual algorithm.

The next step in the Grover's search is an inversion about the mean value of all of the states. This provides the amplitude amplification of the result around which this search functions. In this way, it makes the state of the correct answer more likely. The Oracle call and the amplitude amplification step are then repeated  $\sqrt{N}$  times. As the final step, the states are measured. Because of the amplitude amplification, the probability of the measurement outcome being the correct answer is very high.

What makes Grover's search useful for Map-to-Reference genome sequencing is that it also allows for multiple unknown solutions. This is the case for genome pattern matching, as a short read can match multiple locations in the DNA within the set error margin.

For the simplest implementation of Grover's search, however, the answer to the pattern matching problem needs to be known to make the Oracle. Which is the answer we were hoping to get from the whole search algorithm [38]. The next two methods consist of implementations of the oracle, while the last method considers in-memory computing.

### 5.2.2. Conditional Oracle call

This method defines a different Oracle for each different letter of the used alphabet. For binary applications, the size of the alphabet is two: '0' and '1'. For DNA pattern matching, the alphabet is size 4: 'C', 'G', 'A' and 'T'. Then, depending on the characters of the search pattern, the matching Oracle is called. This requires real-time interpretation of the circuit by the micro-architecture and classical

controls, which is possible if these are fast enough. These conditional oracle calls are used inside a Grover's search pipeline to amplify the correct answer [38].

This algorithm converges on the correct solution only on repetitions of the experiment, so it might not be suitable for general pattern matching. According to [31], the algorithm's efficiency can be easily verified when the alphabet consists of many symbols, and the symbols in the pattern only occur once. Neither is the case for DNA sequence matching.

### 5.2.3. Quantum phone directory

In this case, the initial state is two qubit registers, the index and the search pattern, similar to a phone directory where each entry has a name and a number. The patterns are sorted according to the second register. In the second step of the algorithm, the data qubits are evolved according to their Hamming distance to the search pattern. For a perfect match, the oracle matches the states with the value of 0. This state is then amplified using Grover's search. For approximate matching, the oracle is modified so the minimum value is found, marking the most optimal solution. This is done by having the oracle mark incremental Hamming distances [19, 38].

### 5.2.4. Quantum associative memory

(B.1) To retrieve an element in associative memory, a partial description of the element is passed as the input query. This is in contrast to Random Access Memory (RAM), where just the index of the element is passed. In an associative memory, the nearest match to the query is retrieved. In quantum computing, the search operation becomes operations on a superposition of states which represent the memory. Quantum associative memory consists of two steps, storing the pattern and recalling the pattern. To store the binary patterns, it uses custom unitary gates. A Hadamard-like transform operator  $\hat{S}_{p^p}^s$  is used, one for each associated pattern used to store the reference in the associative memory [49]. These are given by:

$$\hat{S}_{p^p}^s = CR_y(2\sin^{-1}(-1/\sqrt{p})) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \sqrt{\frac{p-1}{p}} & \frac{-s}{\sqrt{p}} \\ 0 & 0 & \frac{s}{\sqrt{p}} & \sqrt{\frac{p-1}{p}} \end{bmatrix}$$

Two oracles are used, one for marking the states in memory for which the qubits from the search query match. The second oracle marks all of the stored states, because there can be more possible states than there are possible patterns [49]. A Matlab implementation of QAM is shown in Listing B.1.

It is also possible to directly incorporate measurement into the recall step, which might allow for faster retrieval. but this would come at the cost of added computing energy, and the translation of a non-unitary matrix to quantum-classical hybrid code is outside of the scope of this thesis [38].

# Unitary Decomposition Implementation

There are several already existing algorithms which result in unitary gates, such as Quantum Associative Memory (Section 5.2.4, code in Listing B.1). This unitary gate is executed on a quantum computer or simulator to generate the result from the algorithm. Therefore, the arbitrarily big unitary matrix needs to be translated into a circuit that can be executed on a quantum computer. This is where unitary decomposition comes in. As can be seen in Figure 6.1, the quantum algorithm specifies an arbitrary quantum gate. The matrix corresponding to this gate is decomposed into a set of matrices, which are the equivalent of some quantum circuit. This circuit can then be used to run the algorithm on the quantum computer.

When the circuit is executed, the effect on the qubits needs to be the same as if the arbitrary quantum gate is executed. This means that the product of all the quantum gates needs to be equal to the matrix coming from the algorithm. So the beginning matrix needs to be decomposed into a set of matrices, where each of the matrices represents a quantum gate from some universal set.

First, the advantages and disadvantages of unitary decomposition are explained in Section 6.1. Then, unitary and quantum logic gates are defined in Sections 6.2 and 6.3, and some special gates that are in between results from the decomposition in Section 6.4. Several unitary decomposition algorithms are compared with respect to gate count in Section 6.5, and special cases for one and two qubit gates are shown in Sections 6.6 and 6.7. The complete Shannon Decomposition is laid out in Section 6.8. The implementation in OpenQL can be found in Section 6.9, and this was optimized, as written in Section 6.10. Finally, the number of gates generated by the decomposition is calculated in Section 6.12, and some concluding remarks are made in Section 6.13.

## 6.1. Advantages and disadvantages of unitary decomposition

The main advantage of unitary decomposition is that it allows a user to specify arbitrary quantum gates as unitary matrices, which can then be executed on quantum systems.

This facilitates a specific type of quantum algorithms, and could also allow short-cuts in the design of new algorithms. In the latter case, a part of an algorithm could be specified as an arbitrary unitary gate while the developer works on the rest of the algorithm, rather than having to make the whole algorithm out of known quantum gates before it can be run. This allows the full algorithm to be tested and checked much earlier in the development process.

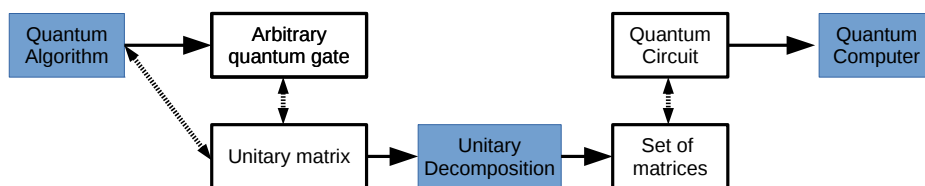


Figure 6.1: How a quantum algorithm uses unitary decomposition

Furthermore, unitary decomposition decomposes to a number of known quantum gates, and the maximum number of gates is known and can be calculated easily. The maximum number of gates is thus known beforehand, and is only dependent on the size of the unitary matrix. This also means that unitary decomposition could be implemented as some very aggressive optimization for circuits that consists of more gates than this maximum. For example, if a user specifies a three-qubit circuit consisting of 180 gates, then reassembling it into a unitary matrix and decomposing that will always result in a circuit with a maximum of 84 rotation gates and 36 CNOT gates. Although the scaling of the number of gates makes this only probable for few qubits.

Finally, the decomposition is numerically exact, so the only limit to the accuracy is in the implementation and in the input matrices. The implementation uses the "double" number type, so the inaccuracy introduced by this is in the neighbourhood of  $10^{-16}$  [24]. Therefore, the accuracy of the decomposition is bounded by the accuracy of the input.

The downside of unitary decomposition is that it takes a lot of separate algorithms, which make it very computationally expensive for higher qubit counts. It includes several mathematical algorithms, such as Singular Value Decomposition and QR decomposition. So if the goal of the algorithm is, for example, to find the eigenvalues of the unitary gate using a quantum computer, then during the decomposition process the eigenvalues are also needed. So in some cases, the answer of the algorithm might already be calculated before the circuit is run on a quantum computer.

Furthermore, unitary decomposition results in a set number of gates, but this number is higher than the theoretical minimum for a unitary gate. Even so, even the minimum number of gates resulting from the decomposition of an arbitrary unitary matrix becomes quite large very quickly. For a specific quantum algorithm, a hand-optimized and application specific circuit will thus generally be shorter than one resulting from universal unitary decomposition.

## 6.2. Unitary gates

Before unitary gates can be explained, first qubits and qubit vectors will be elaborated on. A qubit is a formalization of an ideal quantum-mechanical system with two basic states:  $|0\rangle$  and  $|1\rangle$ . These are written as the following vectors:  $|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$  and  $|1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$  [39]. A single qubit state consists of some combination of these two vectors. So if in bracket notation, a qubit state  $|\phi\rangle$  is  $\alpha|0\rangle + \beta|1\rangle$ , then in vector notation this qubit is  $|\phi\rangle = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}$ , where  $\alpha$  and  $\beta$  can be complex numbers. The combined state of multiple qubits can be expressed as the sum of the possible states. For  $n$  qubits, the total number of possible states is  $2^n$ . In vector notation, the first row corresponds to the all-zeroes state:  $|0..00\rangle$ , and the second row to the  $|0..01\rangle$  state, all the way to the  $|1..11\rangle$  state. This vector notation is the one which will be used for the rest of this chapter. and quantum gates are expressed as matrices.

A quantum gate that applies to  $n$  qubits is a  $2^n$  by  $2^n$  matrix. When the gate is applied to some qubits, the matrix is multiplied with the total qubit vector. The result of this multiplication will be the new qubit state. An example with a one qubit gate is:

$$U = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \quad (6.1)$$

$$|\phi_0\rangle = \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \quad (6.2)$$

$$|\phi_1\rangle = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} a \cdot \alpha + c \cdot \beta \\ b \cdot \alpha + d \cdot \beta \end{pmatrix} \quad (6.3)$$

After the gate, the new qubit state is  $\begin{pmatrix} a \cdot \alpha + c \cdot \beta \\ b \cdot \alpha + d \cdot \beta \end{pmatrix}$ . This illustrates some requirements on quantum gates. The first is that quantum gates need to be square and of the same dimension as the vectors representing the quantum state, because quantum gates have the same number of in and output bits.

The second requirement is that the total measurement probability needs to be preserved. For any qubit, when measured, the total probability of measuring something is 1. Before the gate is applied, this total probability is  $\alpha^2 + \beta^2 = 1$ . After the gate,  $(a \cdot \alpha + c \cdot \beta)^2 + (b \cdot \alpha + d \cdot \beta)^2 = 1$ . This expands to  $(a^2 + b^2)\alpha^2 + (c^2 + d^2)\beta^2 + 2(ac + bd)\alpha\beta = 1$ . When the qubit starts in state  $|0\rangle$ ,  $\alpha$  is 1, and  $\beta$  is



0. The new measurement probability then becomes  $(a^2 + b^2) \cdot 1 = 1$ , so  $a^2 + b^2$  needs to be 1. The same goes for  $c^2 + d^2$ , when the qubit starts out in state  $|1\rangle$ . When these numbers are plugged into the equation, it becomes:

$$\alpha^2 + \beta^2 = 1 \quad (6.4)$$

$$(a \cdot \alpha + c \cdot \beta)^2 + (b \cdot \alpha + d \cdot \beta)^2 = 1 \quad (6.5)$$

$$(a^2 + b^2)\alpha^2 + (c^2 + d^2)\beta^2 + 2(ac + bd)\alpha\beta = 1 \quad (6.6)$$

$$a^2 + b^2 = 1 \quad (6.7)$$

$$c^2 + d^2 = 1 \quad (6.8)$$

$$1 \cdot \alpha^2 + 1 \cdot \beta^2 + 2(ac + bd)\alpha\beta = 1 \quad (6.9)$$

$$1 + 2(ac + bd)\alpha\beta = 1 \quad (6.10)$$

$$ac + bd = 0 \quad (6.11)$$

So, this set of equations imposes some requirements on the 2-by-2 matrix. When put more generally, for n-qubit gates, it means that quantum gates need to be *unitary*. More specifically, when the global phase is not relevant for the outcome, then the quantum gate is a part of the Special unitary (*SU*) group. This is allowed because the global phase has no effect on the measurement probabilities of the different states in  $|0\rangle, |1\rangle$  basis. So discarding the global phase does not influence the outcome of a quantum algorithm.

### 6.3. Quantum Logic Gates

There are several types of quantum operations which correspond to a physical phenomenon, and can be actually executed on quantum bits. Five of these are shown here, along with their corresponding matrices [39]:

- The rotation-X gate:  $R_x(\theta) = \begin{pmatrix} \cos\theta/2 & i\sin\theta/2 \\ i\sin\theta/2 & \cos\theta/2 \end{pmatrix}$

- The rotation-Y gate:  $R_y(\theta) = \begin{pmatrix} \cos\theta/2 & \sin\theta/2 \\ -\sin\theta/2 & \cos\theta/2 \end{pmatrix}$

- The rotation-Z gate:  $R_z(\theta) = \begin{pmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{pmatrix}$

- The Pauli-X gate  $\sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$

- The controlled not gate:  $CNOT = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$

In order for the decomposition of arbitrary unitary gates (matrices) to be universal, they need to be decomposed to some universal set of gates. In this case, all matrices will be decomposed to rotation-Z, rotation-Y and controlled-Not gates. The effects of the rotation gates on a qubit are shown on a Bloch sphere in Figures 6.2 and 6.3.

The controlled-Not gate is a two-qubit gate, where the value of the first qubit influences the state of the second qubit. If the first qubit is  $|0\rangle$ , the second qubit is unaffected. However, if it is  $|1\rangle$ , then a Pauli-X gate is done on the second qubit. This gate is a 180° rotation around the x-axis of the Bloch sphere, which switches the probability amplitudes of the  $|0\rangle$  and  $|1\rangle$  states. This is why it is also called a "not" gate.

These three gates will be used as the universal set, but many other options are possible such as controlled-Z for the two-qubit gate or a rotation-X instead of the rotation-Y or -Z gates.. Generally speaking, any set of gates can be a universal set. As long as this set includes a multi-qubit gate and a way of manipulating a single qubit to all possible locations on the Bloch sphere. This can be multiple gates used in combination, or a single gate which is repeated many times.

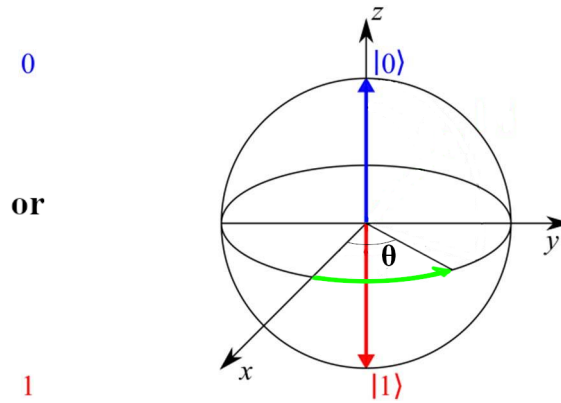


Figure 6.2: Effect of a rotation z gate shown in a Bloch sphere

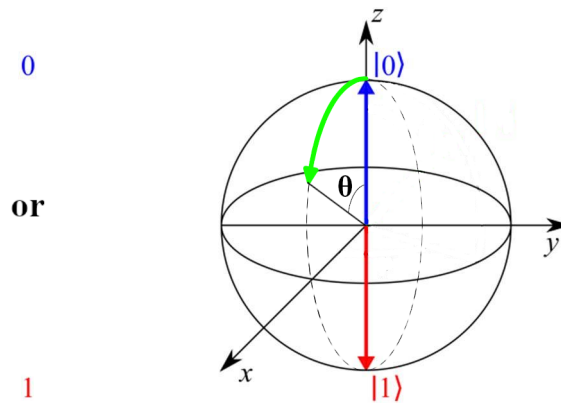


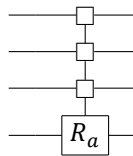
Figure 6.3: Effect of a rotation y gate shown on a Bloch sphere

### 6.4. Special gates

Besides the logic gates, there are also some gates that cannot be executed on a real quantum system, but which are the in-between results of the decomposition algorithm. These are multi-controlled gates and quantum multiplexors.

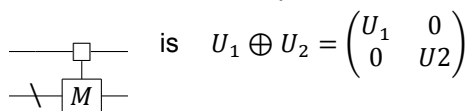
Multi-controlled (rotation) gates are single qubit gates that do a different operation or rotation for each combination of states of the control bits. So if a single qubit gate is controlled by three qubits, then it can execute eight different operations. In the case of multi-controlled rotation gates, only the angle of the rotation depends on the state of the input qubits.

The circuit representation of a multi-controlled rotation gate is:



Quantum multiplexors are multi qubit or one qubit gates, which are controlled by a single qubit. If the state of the control qubit is "0", one gate is applied, and if the state of the qubit is "1", another gate is applied. The matrices corresponding to these gates are either the upper left block of matrix for the total multiplexor, or the lower left.

The circuit and matrix representations are thus:



Number of qubits	1	2	3	4	5	6	n
Quantum circuits for general multi-qubit gates: 6.5.3	0	8	48	224	960	3968	$n - n \cdot 2^n$
Efficient decomposition of quantum gates: 6.5.1	0	4	64	536	4156	22618	$O(4^n)$
Theorem 9: 6.5.2	0	8	62	344	1642	7244	$2 \cdot 4^n - (2n + 3) \cdot 2^n + 2n$
Decomposition of general Q gates (CSD): 6.5.4	-1	4	26	118	494	2014	$(1/4) \cdot 4^n - (1/2) \cdot 2^n - 2$
Decomposition of general Q gates (NQ): 6.5.4	0	3	21	105	465	1953	$(1/4) \cdot 4^n - (3/2) \cdot 2^n + 1$
<b>QSD: 6.8</b>	<b>0</b>	<b>6</b>	<b>36</b>	<b>168</b>	<b>720</b>	<b>2976</b>	$(3/4) \cdot 4^n - (3/2) \cdot 2^n$
QSD optimized: 6.8	0	3	20	100	444	1868	$(23/48) \cdot 4^n - (3/2) \cdot 2^n + 4/3$
Lower bounds: 6.5.5	0	3	14	61	252	1020	$(1/4) \cdot (4^n - 3n - 1)$

Table 6.1: CNOT counts for different implementations of unitary decomposition

## 6.5. Comparison of different algorithms for unitary decomposition

There are many different ways to decompose unitary matrices, so one had to be selected from literature. The various techniques will be compared on their resulting CNOT counts, rather than their single qubit rotations. This is done because not all of the papers present their rotation gate count, and some are unclear whether the number they present for the gate count is that for general single-qubit gates or for single rotation gates. As this means potentially three times as many gates, it was decided to not use these numbers but rather to use the CNOT count to compare the algorithms. This number gives a reasonable indication of rotation gate count as well.

The compared algorithms are shown in Table 6.1, and will be explained in the next sections.

### 6.5.1. QR decomposition: Efficient decomposition of Quantum Gates

In [47], unitary decomposition is described, which uses Gray code and QR decomposition. The total number of controlled gates is  $g_n(k) = g_n^0(k) + g_{n-1}(k) + g_{n-1}(k-1)$ . The total number of CNOT gates needed for the decomposition is of the order  $O(4^n)$ .

### 6.5.2. QR decomposition (Theorem 9 from [39])

This uses a different technique for unentangling the qubits. Since a unitary gate can be completely defined by its effect on base vectors, this technique implements the correct behaviour for each vector iteratively, while leaving previous assessed vectors preserved. This leads to a total CNOT count of  $2 \cdot 4^n - (2n + 3) \cdot 2^n + 2n$ .

### 6.5.3. CSD: Quantum circuits for general multi-qubit gates

The circuit presented in [33] applies Cosine Sine Decomposition (CSD) recursively. Each of the  $2^{n-1} - 1$  sections consists of  $2^{n+1}$  elementary one-qubit rotations and  $2^{n+1} - 4$  CNOT gates. Some gates can be canceled, so the total cost is  $4^n$  elementary one-qubit gates and  $4^n - 2^{n+1}$  CNOT gates.

### 6.5.4. CSD and NQ: Decomposition of general quantum gates

This paper [34] shows two ways to decompose arbitrary unitary matrices. The first is using QR decomposition to transform the matrix into a product of Givens rotations. These rotations correspond to uniformly controlled one-qubit gates. These are decomposed to a combination of CNOT gates and rotation gates using singular value decomposition. This yields a total of  $\frac{1}{4}4^n - \frac{3}{2}2^n + 1$  CNOT gates.

The other decomposition uses CSD, applying NQ decomposition [39]. This decomposition stops at the two-qubit gate level, which are decomposed using the minimal elementary gate construction from [40]. This results in an implementation which has  $\frac{1}{4}4^n - \frac{1}{2}2^n - 2$  CNOT gates.

So although these methods seem considerably better than the Quantum Shannon Decomposition, they should actually only be compared to the optimized version of the QSD. This is because they implement partly the same optimizations, such as stopping the decomposition at an optimized two-qubit circuit, and these algorithms perform worse than QSD for that level of optimization.

### 6.5.5. Theoretical lower bounds

The theoretical lower bound for CNOT count for n-qubit gates is proven to be  $\frac{1}{4}(4^n - 3n - 1)$  [40]. There is not currently an automated implementation that reaches this theoretical count, however, although

there are implementations that reach this number for 2 [40] and 3 qubit gates [48]. For 1 qubit gates, no CNOT gates are necessary. The minimum rotation gate count for a single qubit gate could be 1 if any possible rotation gate is allowed. But when using physically executable gates, all implementations use a decomposition that generates 3 single rotation gates when decomposing an arbitrary single-qubit gate.

## 6.6. One qubit gate: ZYZ decomposition

For one qubit gates, there is a single way to decompose them into rotation gates. This is called ZYZ-decomposition, and is shown below:

$$U = \begin{pmatrix} e^{i\delta} & 0 \\ 0 & e^{i\delta} \end{pmatrix} \begin{pmatrix} e^{i\alpha/2} & 0 \\ 0 & e^{-i\alpha/2} \end{pmatrix} \begin{pmatrix} \cos(\gamma/2) & \sin(\gamma/2) \\ -\sin(\gamma/2) & \cos(\gamma/2) \end{pmatrix} \begin{pmatrix} e^{i\beta/2} & 0 \\ 0 & e^{-i\beta/2} \end{pmatrix}$$

where  $\delta$ ,  $\alpha$ ,  $\beta$  and  $\gamma$  are real angles [6]. This corresponds to two rotation-Z gates, and one rotation-y gate. For any 2x2 unitary matrix U, the angles  $\Phi$ ,  $\alpha$ ,  $\beta$ ,  $\gamma$  can be found so that the following equation is satisfied:

$$U = e^{-i\Phi} * \begin{pmatrix} A & B \\ C & D \end{pmatrix} \quad (6.12)$$

$$U = e^{-i\Phi} R_z(\alpha) R_y(\beta) R_z(\gamma) \quad (6.13)$$

$$SU = R_z(\alpha) R_y(\beta) R_z(\gamma) \quad (6.14)$$

Or as a circuit:

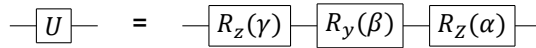


Figure 6.4: ZYZ decomposition of an arbitrary (special) unitary gate

The decomposition of an arbitrary unitary  $U$  with size  $n$  is then as follows:

$$\Phi = \frac{1}{n} * \arctan\left(\frac{\text{Im}|U|}{\text{Re}|U|}\right) \quad (6.15)$$

$$SU = U * e^{i\Phi} \equiv \begin{bmatrix} A & B \\ C & D \end{bmatrix} \quad (6.16)$$

$$sw = \sqrt{\text{Im}(B)^2 + \text{Re}(B)^2 + \text{Im}(A)^2} \quad (6.17)$$

$$wx = \text{Im}(B)/sw \quad (6.18)$$

$$wy = \text{Re}(B)/sw \quad (6.19)$$

$$wz = \text{Im}(A)/sw \quad (6.20)$$

$$t1 = \arctan\left(\frac{\text{Im}(A)}{\text{Re}(A)}\right) \quad (6.21)$$

$$t2 = \arctan\left(\frac{\text{Im}(B)}{\text{Re}(B)}\right) \quad (6.22)$$

$$\alpha = t1 + t2 \quad (6.23)$$

$$\gamma = t1 - t2 \quad (6.24)$$

$$\beta = 2 \cdot \arctan\left(\frac{sw * \sqrt{wx^2 + wy^2}}{\sqrt{cw^2 + (wz * sw)^2}}\right) \quad (6.25)$$

First, the angle  $\Phi$  is calculated from the angle between the real and imaginary parts of the determinant of matrix U. Then, the factor  $e^{i\Phi}$  is removed from U to get SU, the special unitary group. This means that the global phase is not taken into account [38, 44]. From this, the angles for the ZYZ decomposition can be found.

### 6.7. Minimum universal quantum circuit for a two qubit gate

There also exist minimum implementations for two-qubit gates. The minimum number of gates to correctly specify an arbitrary unitary matrix of size 4x4 up to the global phase is 18. This amounts to 3 CNOT gates, 3 rotation gates, and 4 arbitrary single qubit gates which become 12 single rotation gates using ZYZ decomposition: 3+3+12 = 18 gates [40]. A possible implementation using this minimal number of gates is shown in Figure 6.5, where  $a, b, c$  and  $d$  are arbitrary one-qubit gates. This extends to  $n$ -qubit gates, with the theoretical bound being  $\frac{1}{4}(4^n - 3n - 1)$  [40].

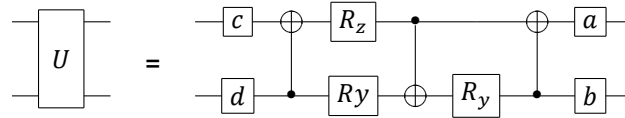


Figure 6.5: Minimal universal quantum circuit using 18 elementary gates[40]

### 6.8. Complete Quantum Shannon Decomposition algorithm

For bigger unitary gates of arbitrary size, the Quantum Shannon Decomposition is selected, because it results in the fewest generated CNOT and rotation gates of the compared circuits when it is implemented in an optimized way. In the most basic implementation, it results in the lowest gate-count compared to the simplest versions of the other compared algorithms.

#### 6.8.1. Quantum Shannon decomposition

From the different decomposition methods compared in [39], Quantum Shannon Decomposition uses less than twice the number of gates compared to the ideal method. The advantage of this Quantum Shannon Decomposition is that it is easy to implement using recursion: the function calls itself to decompose the sub-matrices. The circuit for the decomposition is shown in Figure 6.6, where the “ $n$ ” means an arbitrary number of qubits. Gate  $U$  is a unitary gate applied to at least two qubits, and the  $R_y$  and  $R_z$  gates are uniformly controlled rotation gates. The  $G_1 - G_4$  are unitary gates of one size smaller than  $U$ . These gates can be further decomposed using the Quantum Shannon Decomposition. QSD is easy to optimize because it recurses on unitary gates, so each step of the recursion ends with four smaller unitary gates on which the function gets called. In the current implementation, the recursion stops at one qubit gates, but this could easily be adjusted to two-, three- or even four qubit gates if universal circuits for these are discovered and/or implemented.

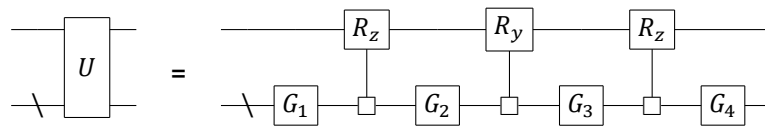


Figure 6.6: Quantum Shannon Decomposition[39]

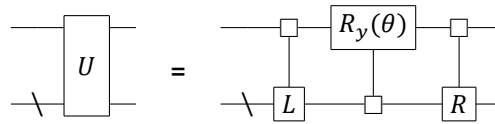
To get to this decomposition, the matrix is first decomposed using cosine sine decomposition, and then the uniformly controlled arbitrary unitary gates are demultiplexed, as well as the uniformly controlled rotation gates.

#### 6.8.2. Cosine Sine decomposition

The first decomposition step is Cosine Sine Decomposition (CSD). Any even-dimensional unitary matrix  $U$  can be decomposed into real diagonal matrices  $C$  and  $S$ , and smaller unitary matrices  $L_0, L_1, R_0, R_1$  using CSD. See equation 6.26, where  $C^2 + S^2 = I$ , and  $I$  is the identity matrix.

$$U = \begin{pmatrix} R_0 & 0 \\ 0 & R_1 \end{pmatrix} \begin{pmatrix} C & -S \\ S & C \end{pmatrix} \begin{pmatrix} L_0 & 0 \\ 0 & L_1 \end{pmatrix} \tag{6.26}$$

As a quantum circuit, this looks like this:



Where  $G_j$  are uniformly controlled gates, which execute either  $A_j$  or  $B_j$  depending on the value of the control bit, and  $R_y$  is a uniformly controlled rotation around the y-axis.

For any unitary matrix  $U$ , there exist four unitary matrices  $L_0, L_1, R_0$  and  $R_1$  such that: [36]

$$U = \begin{pmatrix} U_{00} & U_{01} \\ U_{10} & U_{11} \end{pmatrix} \quad (6.27)$$

$$L^\dagger UR = \begin{pmatrix} L_0^\dagger & 0 \\ 0 & L_1^\dagger \end{pmatrix} \quad (6.28)$$

$$\begin{pmatrix} U_{00} & U_{01} \\ U_{10} & U_{11} \end{pmatrix} \begin{pmatrix} R_0 & 0 \\ 0 & R_1 \end{pmatrix} = D = \begin{pmatrix} C & -S \\ S & C \end{pmatrix} \quad (6.29)$$

$$(6.30)$$

This leads to the following equalities for the submatrices of  $U$ :

$$U_{00} = L_0 C R_0^\dagger \quad (6.31)$$

$$U_{10} = L_1 S R_0^\dagger \quad (6.32)$$

$$U_{01} = L_0 C R_1^\dagger \quad (6.33)$$

$$U_{11} = L_1 S R_1^\dagger \quad (6.34)$$

$$(6.35)$$

$C$  and  $S$  are diagonal matrices with respectively the cosines and sines of  $\theta$  as the diagonal elements. These angles are the angles between the subspaces, and can be obtained using Singular Value Decomposition [16].

$$L_0^\dagger U_{00} R_0 = C = \text{diag}(c_1, \dots, c_n) \quad (6.36)$$

$$L_1^\dagger U_{10} R_0 = S = \text{diag}(s_1, \dots, s_n) \quad (6.37)$$

where the values  $c_j$  are ordered from small to large, and are between 0 and 1.

### 6.8.3. Singular Value Decomposition

The Singular Value Decomposition of a matrix is:

$$U = A \Sigma B^\dagger \quad (6.38)$$

Where  $A$  and  $B$  are unitary matrices, and the diagonal entries of  $\Sigma$  are the singular values of  $U$  [16]. This means that using SVD, the matrices in this equation can be obtained:

$$U_{00} = L_0 C R_0^\dagger \quad (6.39)$$

$$(6.40)$$

And the matrices  $L_0, L_1, R_0, C$  and  $S$  are obtained with:

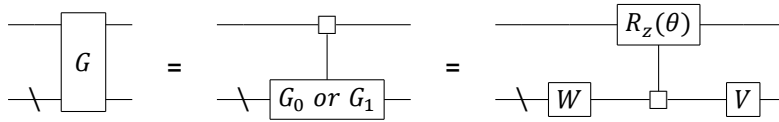
$$L_1^\dagger = U_{10} * R_0 \quad (6.41)$$

$$L_1 = U_{10} * R_0 * C^{-1} \quad (6.42)$$

For the efficient calculation of the SVD, several algorithms exist. These are implemented in software packages such as LAPACK [24].

### 6.8.4. Decomposing a uniformly controlled arbitrary gate

The matrix  $G = \begin{pmatrix} G_0 & 0 \\ 0 & G_1 \end{pmatrix}$ , when applied to a quantum state, is equal to a *uniformly controlled gate*. This gate executes unitary gate  $G_0$  when the control bit is 0, and the gate  $G_1$  when it is 1. This can not generally be executed as a physical process, so it needs to be decomposed to a circuit like this.



Which can be done using eigenvalue decomposition, using this matrix equality:

$$G = \begin{pmatrix} G_0 & 0 \\ 0 & G_1 \end{pmatrix} = \begin{pmatrix} V & 0 \\ 0 & V \end{pmatrix} \begin{pmatrix} D & 0 \\ 0 & D^\dagger \end{pmatrix} \begin{pmatrix} W & 0 \\ 0 & W \end{pmatrix} \tag{6.43}$$

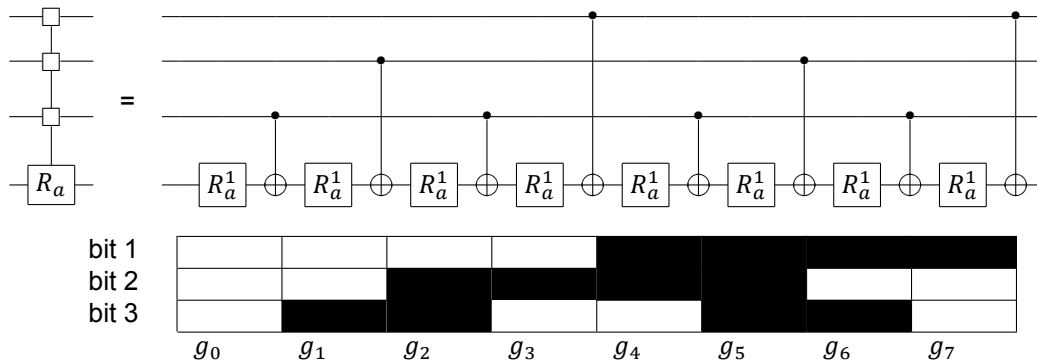
$$G_0 \cdot G_1^\dagger = VD^2V^\dagger \tag{6.44}$$

$$W = DV^\dagger G_1 \tag{6.45}$$

Where  $V$  consists of the eigenvalues of the product of  $G_0$  and  $G_1^\dagger$ , and  $D^2$  has the eigenvalues of this matrix product on the diagonal. The matrix  $D$  is obtained by taking the square root of this matrix.

### 6.8.5. Decomposing multi-controlled rotation gates

The multi-controlled rotation gates can be decomposed into a sequence of CNOT gates and rotation gates. A quantum circuit for a rotation gate controlled by 3 qubits is shown below [33]:



Which qubit is the controlling bit of each CNOT is determined using Gray code, which is shown in the table below the figure. The number of the bit that gets changed in the Gray code is the number of the qubit that will be the control qubit. The angles for the rotation gates,  $R_a(\theta_i)$  are calculated using Equation (6.46), where  $M^k$  is calculated using Equation (6.47). The values for  $\alpha_i$  are calculated using the diagonal values of the matrices corresponding to the multi-controlled rotation gates, using Equation (6.48) for a multi-controlled-Y and Equation (6.49) for the multi-controlled-Z matrices.

$$M^k \begin{pmatrix} \theta_1 \\ \vdots \\ \theta_{2^k} \end{pmatrix} = \begin{pmatrix} \alpha_1 \\ \vdots \\ \alpha_{2^k} \end{pmatrix} \tag{6.46}$$

$M^k$  is a square matrix where all the entries are either "+1" or "-1", which are calculated using Gray code. The exponent is the bit-wise inner product of the binary vectors for the standard binary code representation of the integer  $i$  ( $b_i$ ) and the binary representation of the  $i$ th value of the gray code up to a value of  $2^k$ . The  $j$ th value of the gray code is calculated using the bit-wise XOR of the unsigned binary  $j$  and a single shift right of the value of  $j$ , like so: `j^(j>>1)` for C++ code.

$$M_{ij}^k = (-1)^{b_{i-1} \cdot g_{j-1}} \quad (6.47)$$

The entries for  $\alpha_i$  are for controlled Y, from the arcsin of the diagonal entries of the S-matrix from the Cosine Sine decomposition.

$$\begin{pmatrix} \alpha_1 \\ \vdots \\ \alpha_{2^k} \end{pmatrix} = 2 \cdot \arcsin(\text{diagonal}(S)) \quad (6.48)$$

And for controlled a multi-controlled rotation-Z gate, using the natural logarithm of the diagonal entries of the D-matrix coming from the demultiplexing of the arbitrary controlled unitary gate.

$$\begin{pmatrix} \alpha_1 \\ \vdots \\ \alpha_{2^k} \end{pmatrix} = -2 \cdot \sqrt{-1} \cdot \ln(\text{diagonal}(D)) \quad (6.49)$$

## 6.9. OpenQL implementation

The implementation of the decomposition in OpenQL is split into two pieces. The first is the matrix decomposition, which calculates all of the rotation angles for all the rotation gates. There are two reasons for this: it is not known at the "Unitary" level what the specific qubits are that the gate will be applied, and the costly matrix decompositions will not have to be done again if the same arbitrary unitary gate is applied to a different set of qubits.

This split is very clear when using the unitary decomposition in OpenQL. First, the "Unitary" object is defined. This object is added to a kernel, with the qubits that the unitary gate will be applied to. A graphic representation of the function calls and the program structure involved in the decomposition is shown in Figure 6.7.

### 6.9.1. Unitary

When "decompose()" is called on the "Unitary" object, the complete Shannon Decomposition is computed, and the calculated angles for the resulting rotation gates are added to a list called "instructionlist". A check is performed to see if the matrix is unitary at the highest level of the decomposition, and only called once. All the calculated matrices will also be unitary if the input matrix is, so this check is only done once. Furthermore, all the sizes of the matrices used to calculate the values the multi-controlled rotation gates ( $M^k$ ) are calculated and added to a lookup table, so they don't need to be recalculated every time a multi-controlled gate is computed.

To make certain that the decomposition is correct, each single intermediate decomposition is checked. This saves any calculations that might be done on an incorrect matrix. And checking at each step means that only three matrices need to be multiplied rather than having to assemble and store matrices for all the steps of the decomposition.

The `decomp_function` is the main function of the recursion, in which all the intermediate matrices are initialized and passed by reference to the Cosine Sine Decomposition function (CSD) and the demultiplexing function. This way there is no unnecessary copying of these matrices [14]. The `decomp_function` is called for the new unitary matrices. With each call of the `decomp_function`, the matrix size is halved, until finally the function is called for two-by-two matrices. The size of the matrices is checked every recursion, and if it is two, the `zyz_decomp()` function is called. Besides that, the matrices for the multi-controlled rotation gates are passed to the appropriate functions. This is done in vector form, as these are diagonal matrices of which only the diagonal is needed, again, to save any unnecessary copying and storing of data.

The `zyz_decomp` function takes two-by-two matrices, and decomposes them into three rotation gates. The angles for these gates are added to the variable length vector "instructionlist" as doubles. The multicontrolled-Y and -Z functions take diagonal matrices as vectors, and use the function as described in Section 6.8.5 to calculate the angles for the rotation gates. These are also added to the "instructionlist" vector.



Besides function calls, more measures were taken to minimize the memory use of the algorithm, and therefore optimize the execution speed. The linear algebra library used, Eigen, assumes aliasing for every matrix multiplication, and therefore copies the result into a temporary matrix before assigning it [14]. Therefore, any cases where aliasing would not be an issue were fitted with the `.noalias()` function in order to avoid this copying.

Besides that, any cases with actual aliasing, such as square roots and adjoint operations, were replaced with their alias-proof versions, or the operation was moved to the first assigning operation where the result was needed. So `v1 = v1.adjoint();` was replaced with `v1 = adjointInPlace();`, and `D = eigslv.eigenvalues(); D = D.sqrt` was replaced with `D = eigslv.eigenvalues().sqrt();` in order to avoid aliasing issues.

### 6.9.2. Kernel

The angles are passed on via the "instructionlist" from the decompose function, and assembled into a circuit with the appropriate CNOT gates placed in between the rotation gates. At the kernel level, when the (decomposed) gate is added to the circuit, the gates and CNOTs are assembled and added to the circuit list. Because the circuit is the same for each qubit count, which angle corresponds to a gate is based on the order in which they were added. When optimizations result in less overall gates, this needs to be signalled to the kernel so that each rotation angle still gets applied to the correct gate.

It is also checked whether the gate is applied to the correct number of qubits, and whether it is decomposed at all. If either of these is the case, an exception is thrown and the compilation is stopped.

The kernel only has the list of rotation counts, and calls either the main recursive function, or the functions that calculate the control qubit numbers for the multicontrolled rotation gates. And this process needs to be sequential, because the total number of gates is not known beforehand due to the gatecount optimizations outlined in Section 6.10. Therefore the index for the "instructionlist" can only be known by knowing exactly how many gates happened before it. This is solved by having each call to the main recursive function return the number of gates used in that level and all of the ones below it, and each call to the main function includes the current index. This way, the kernel adds the rotation gates to the circuit in the correct order, which is the same order in which they have been calculated in the unitary decomposition.

### 6.9.3. Compilation

When the "compile()" function is called on the OpenQL program, the list of gates resulting from the decomposition is handled exactly the same as any manually added gates, and therefore passes through all of the same compilation steps as outlined in Chapter 4. The gates are transformed into QASM, and written to some output file.

## 6.10. Gate count Optimization

Two main optimizations were implemented to bring down the gate count, both geared to skipping steps in the recursion. They are the detection of quantum multiplexors, and the detection of unaffected qubit gates. Because parts of the recursion are skipped, the pattern of rotation gates and CNOTs is affected. Therefore, if one of these optimizations is used, it needs to be signalled to the kernel, and the same steps can be skipped in the assigning of the rotation angles to the gates. This is done via specific angles that are pushed to the "instructionlist", and these angles are checked in the kernel. Values of "100.0", "200.0" and "300.0" were chosen because they are outside of the range of values that can be calculated for the rotation angles, which are calculated in radians. Therefore, no actual gates with these specific angles are generated. Besides that, the round values are easy to recognize by any humans who need to verify the instructions, and are not easily mistaken for actual rotation gate values. It is necessary to signal using numbers, because that is the type of the elements inside the "instructionlist", and signalling using the "instructionlist" provides not just the type of the optimization, but also the specific place the optimization occurs in a way that makes it easy to implement the optimizations in the kernel.

### 6.10.1. Detection of multiplexors

When At any step, the unitary gate is already the shape of a multiplexed unitary gate, the Cosine Sine Decomposition of this matrix can be skipped. If the upper right and lower left quarters of the matrix are zero-matrices, the matrix is a multiplexed unitary gate, and the submatrices are passed

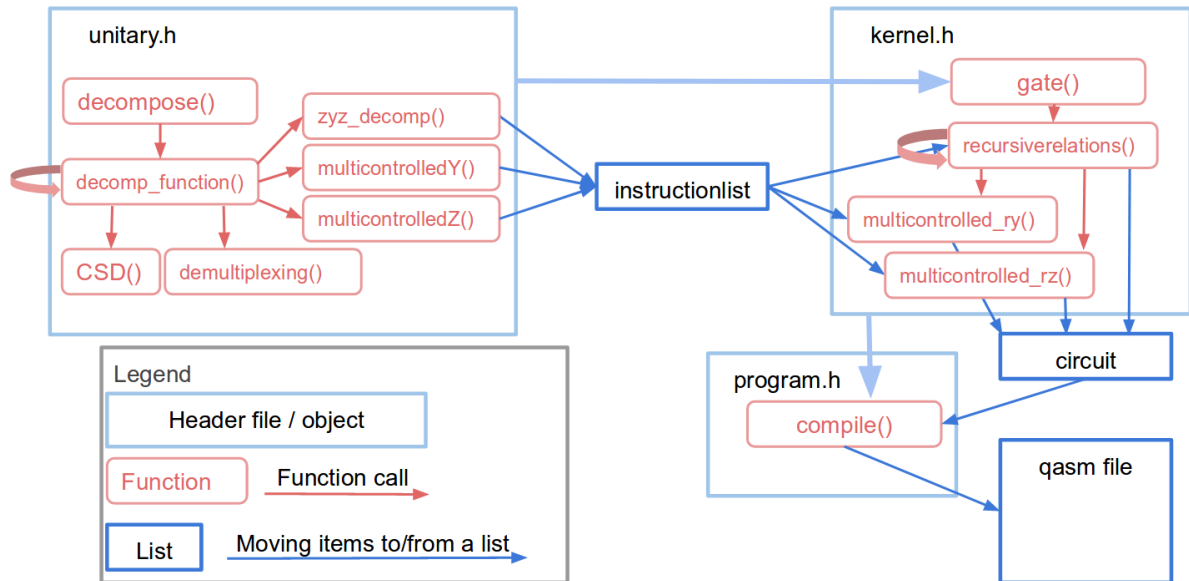


Figure 6.7: The program structure and function calls of the various parts of the decomposition algorithm

on to the demultiplexing function. To signal this to the kernel, a value of "200.0" is passed on to the "instructionlist".

### 6.10.2. Unaffected qubits

If any qubits are not affected by a unitary gate, then it is unnecessary to apply any gates to this qubit. And as the total gate count is exponential with regards to the number of qubits, treating an effectively  $(n - 1)$ -qubit gate as an  $n$ -qubit gate results in a lot of unnecessary gates.

Therefore, any non-affected qubits need to be detected. If a qubit is not affected, the gate is the Kronecker product of a smaller gate and the identity matrix. Either the first qubit is not affected or the last qubit is not affected. Because of the way the decomposition works, any unaffected qubits in the middle of the circuit are evaluated in later steps of the recursion and become the first or last qubit in that stage.

When the first qubit is not affected, the unitary matrix corresponding to the gate looks like this:

$$\begin{array}{c} \text{---} \\ \diagdown \\ \text{---} \\ \boxed{B} \\ \text{---} \end{array} \quad \text{is} \quad I \otimes B = \begin{pmatrix} B & 0 \\ 0 & B \end{pmatrix}$$

Where  $B$  is an arbitrarily big unitary gate, and the zeroes are correspondingly big zero-matrices. In this case, the gate is a special case of a multiplexor, where the lower right and upper left matrices are equal. This is checked, and a signalling value of "30.0" is pushed to the instructionlist, and the recursion proceeds with the upper left of the matrix, which is  $B$  in the example.

If the last qubit is not affected, the unitary matrix corresponding to the gate looks like this:

$$\begin{array}{c} \text{---} \\ \boxed{A} \\ \text{---} \\ \diagdown \\ \text{---} \end{array} \quad \text{is} \quad A \otimes I = \begin{pmatrix} a_0 & a_1 \\ a_2 & a_3 \end{pmatrix} \otimes \begin{pmatrix} a_0 & 0 & a_1 & 0 \\ 0 & a_0 & 0 & a_1 \\ a_2 & 0 & a_3 & 0 \\ 0 & a_2 & 0 & a_3 \end{pmatrix}$$

This is also checked, and in that case a value of "200.0" is passed to the "instructionlist". The matrix  $A$  from the example is assembled from all the non-zero values, and passed on to the next level of the recursion.

## 6.11. Execution time optimizations

When the execution time for the decomposition of a unitary matrix into QASM was tested for the first time, it was very noticeable that something should be improved. Mainly in "Adding the gate to the kernel", which should not take too much time, but which scaled exponentially with the number of matrix

elements. So therefore, several optimizations were implemented to improve the speed of the decomposition.

### 6.11.1. Adding the gate to the kernel

In the first implementation, adding the decomposed to the kernel scaled exponentially with input matrix entries. This scaling is shown in Figure 6.8, the "Adding to kernel" line. This is the wall-cock execution time per line of code, generated using the program shown in Listing 8.1. Adding the unitary to the kernel started taking more time per matrix entry or per added rotation gate as the size increased. This can have several explanations: the first is that this part of the code needs to be executed sequentially, because the "instructionlist" gets translated into gates one-by-one. Because of the optimizations, the recursive step of the algorithm needs the total number of rotation gates that are added one level lower, before it can call on the next step. This means parallel execution of this part of the algorithm is rather limited. Another possible explanation is that instantiating all the added gates fill up the first level of cache, so that they need to be written to memory. This is much slower than keeping them in cache, and therefore the whole algorithm becomes slower. In this case, adding to kernel is not an exponential process, but rather two different linear relations. The point where adding the gate to the kernel seemed to become slower is around the 5-qubit mark. For a 5-qubit unitary, the total number of gates is 512.

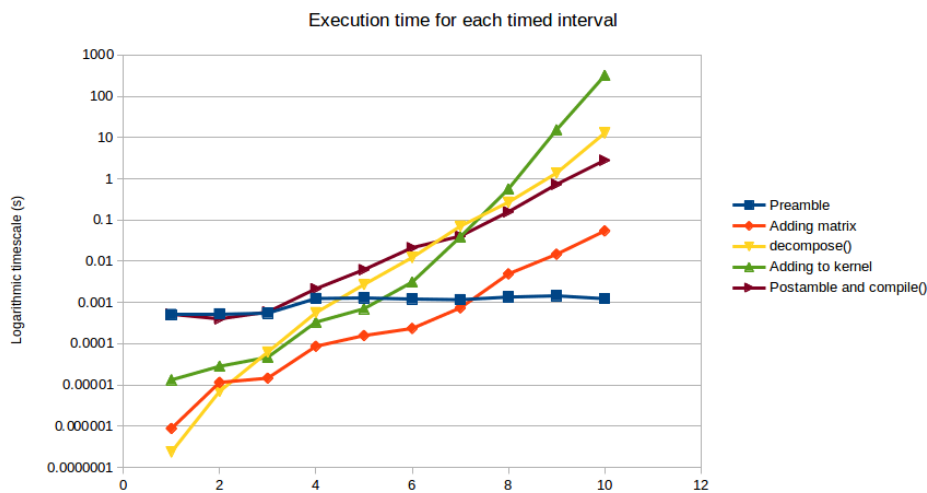


Figure 6.8: Execution time per timed interval of the first test of the program

Several ways to optimize the code in kernel.h were implemented. These are:

- To calculate some of the values for the "instructionlist" counter (start\_counter) rather than have them be returned from the function itself. It is not possible to do this for all the components, as the optimization means that it is not known how many gates certain parts will be decomposed into beforehand.
- Removal of the if-statements for zero-gates, when not many will be generated for most general unitary matrices.
- To pass the "instructionlist" as a pointer rather than copying the whole list for each function call, which might also explain part of the big memory allocation for this part of the code. This was determined to be the cause of the extreme growth in execution time.

With the new optimizations, the execution time for specifically the decomposition and kernel parts of the program are shown in Figure 6.9. As you can see, these optimizations have made the full decomposition a lot faster, and now it scales, as expected, with the matrix size shown as the  $4/n$  line. The new longest part of the decomposition is now the decompose() function, which still seems to become slower for bigger matrix sizes at a rate worse than expected.

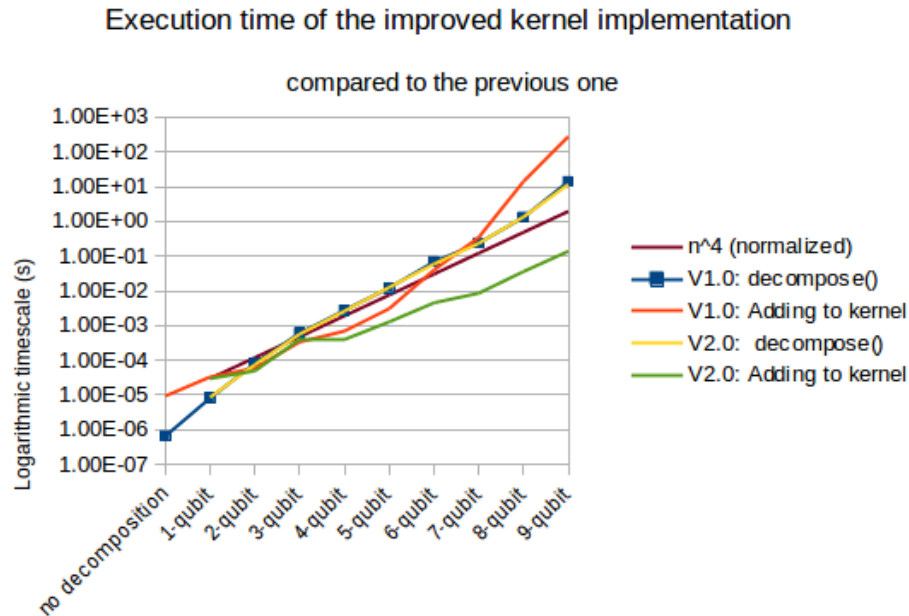


Figure 6.9: Execution time of the new kernel and decomposition code (V2.0) compared to the old one (V1.0)

### 6.11.2. The decompose() function

As can be seen in Figure 6.9, the new kernel implementation scales with the matrix size (so with  $4^n$ ), but the decomposition does not. This seems to indicate that there is some optimization left to do.

As the decomposition function consists of several different function calls, additional timing information was needed to determine exactly which part of the code was causing the bad scaling. Because of the recursion, the timing calls needed to be placed carefully, so they would not also measure lower levels of the recursion. The following things were timed: the total CSD function, the total `zyz_decomp` function and the combined time of both `multiplexed_` functions. Times were measured using the "chrono" library of c++, using code like this:

```
1 std::chrono::duration<double> zyz_time;
2 auto start = std::chrono::high_resolution_clock::now();
3 // Code to be measured
4 zyz_time += std::chrono::high_resolution_clock::now() - start;
```

The time that comes from these functions is the accumulated total time spend inside the function for each decomposition call. The total execution time of the `decompose()` function was scaling worse than the input matrix size, so if any of these functions caused this scaling then that should show up in the total execution time. Especially for functions that only get called for small matrices like the `zyz-decomposition`, the total time spend inside these functions reflects that they get called more often if the input matrix is bigger. The results of these tests are shown in Figure 6.10. As can be seen in the figure, the time spend inside the Cosine Sine Decomposition increases faster than the matrix size ( $n^4$ ), while the other functions scale with the matrix size. This indicates that the issue is located inside the CSD function.

Another timing test was done with multiple timing statements inside the CSD function, and the code shown below was the cause of the increase in execution time:

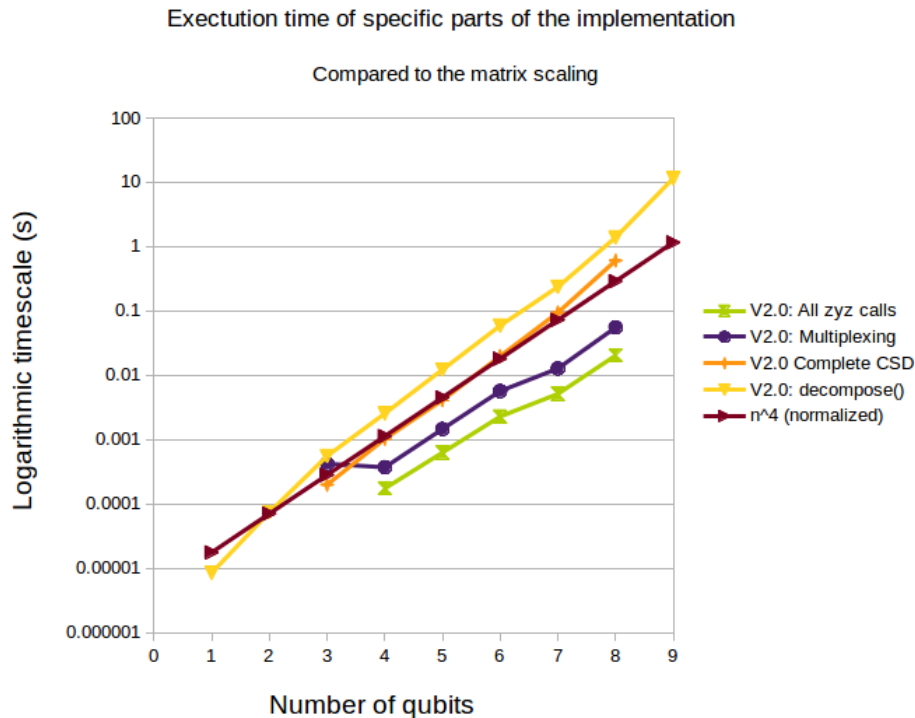


Figure 6.10: The total execution time for different functions that get called by the `decompose()` function in c++

```

1 for(int i = 0; i < p; i++){
2     if(std::abs(s(i,i)) > std::abs(c(i,i))){
3         complex_matrix tmp = u1.adjoint()*U.topRightCorner(p,p);
4         v2.row(i) = tmp.row(i)/s(i,i);
5     }else{
6         complex_matrix tmp = u2.adjoint()*U.bottomRightCorner(p,p);
7         v2.row(i) = tmp.row(i)/c(i,i);
8     }}

```

This for-loop generates a new temporary matrix for every iteration. When the matrices are computed just once, before the for-loop, the execution time start behaving again, and just scales with the input matrix size, so with  $4^n$ . Assigning the rows every for-loop is not an issue because of how Eigen handles blocks of matrices, it only copies the specific row. However, it assumes aliasing [14], so another optimization it to add a `.noalias()` to these operations so no additional matrices get allocated but the result gets immediately assigned to the matrix `v2`.

The demultiplexing, a function that was not timed in the first tests, has a final component that is not scaling well, which is the generation of the eigenvectors and eigenvalues. As this growth in execution time is not the case for the Qubiter implementation (shown in Section 8.6), there must be a better way to do this. The cause might be the `ComplexEigenSolver` functionality from Eigen, as it is classified as being very slow [14]. However, when comparing the execution times for other methods of diagonalizing the matrix, the `ComplexEigenSolver` was determined to be the fastest. The other options, `ComplexSchur` and `Hessenberg` decomposition, either scaled worse than the eigenvalue decomposition, or required a matrix-wide square root operation rather than calculating the square roots for each eigenvalue individually. This meant that of the options available, the complex `EigenSolver` still performs the best, though the matrix square root operation could be replaced which resulted in slightly improved scaling. Additionally the `Schur` decomposition is used for the smaller matrices, although it does not seem to have much of an impact on the total decomposition time, possible due to internal optimizations from the linear algebra library.

This results in the new total execution time for the complete decomposition, shown in Figure 6.11. The Cosine Sine Decomposition is faster than the initial implementation, noticeable from about 8 qubits onwards. The demultiplexing is slightly better than the first try, although it seems that there is some improvement left to make.

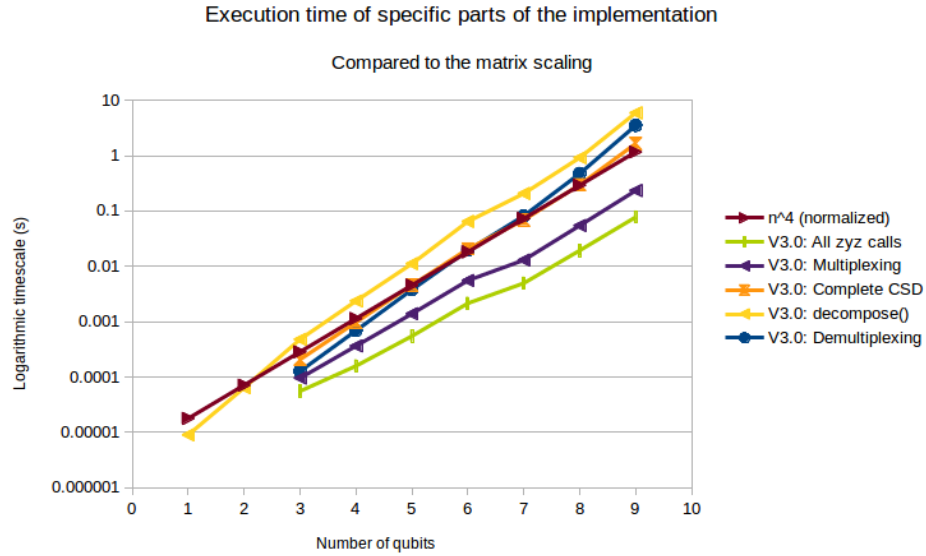


Figure 6.11: The total execution time of the different functions called by the `decompose()` function, with the optimizations for both the Cosine Sine Decomposition (faster from 8 qubits on) and the demultiplexing

These optimizations led to the total execution times as shown in Figure 8.2. Note that the total execution time for the 9 qubit unitary has improved by about a factor 30, going from 300 seconds to 9 seconds of execution time.

## 6.12. Number of gates

The initial implementation of the QSD decomposes each arbitrary  $n$ -qubit unitary gate into 4  $(n-1)$ -qubit unitary gates, and 3  $n$ -qubit uniformly controlled rotation gates. The  $(n-1)$  unitary gates are then decomposed again, setting up a recursive relation until only single qubit unitary gates are left. The total number of single qubit unitary gates, which are almost the final step of the decomposition, can be expressed like this:

$$G(n) = 4 * G(n - 1) \quad (6.50)$$

Each single qubit unitary can be decomposed into 3 single rotation gates, which makes the more accurate gate count:

$$Z(n) = 3 * 4 * Z(n - 1) \quad (6.51)$$

For the uniformly controlled gates, with 1 qubit for the rotations and  $(n-1)$  qubits that control it, each gate gets decomposed into  $2^{n-1}$  CNOT gates and  $2^{n-1}$  single rotation gates. The recursive relation for both of these is:

$$R(n) = 3 * 2^{n-1} + 4 * R(n - 1) \quad (6.52)$$

Which means that the total rotation gate count is:

$$U(1) = 3 \quad (6.53)$$

$$U(n) = 3 * 2^{n-1} + 4 * U(n - 1) \quad (6.54)$$

And the total CNOT count is:

$$C(1) = 0 \quad (6.55)$$

$$C(n) = 3 * 2^{n-1} + 4 * C(n - 1) \quad (6.56)$$

Which can be made into these formula's:

$$U(n) = 3 * 2^{n-1} * (2^n - 1) \quad (6.57)$$

$$U(n) = 3/2 * 4^n - 3/2 * 2^n \quad (6.58)$$

$$C(n) = 3 * 2^{n-2} * (2^n - 2) \quad (6.59)$$

$$C(n) = 3/4 * 4^n - 3/2 * 2^n \quad (6.60)$$

## 6.13. Conclusion

The Quantum Shannon Decomposition as implemented in OpenQL generates the expected number of rotation gates and CNOTs. However, without implementation of the optimized two-qubit circuit, and the other optimizations from the compared decomposition algorithms, it will not match the lowest gate counts from these algorithms. But the chosen decomposition can easily be optimized within the existing program structure. None of the compared algorithms reach the lowest gate count, so future optimizations might surpass the current best known gate count.

Besides that, the implemented optimizations mean that decomposition of specific applications can result in significantly fewer gates than the one calculated here, as these are all worst-case counts.

But still, using this exact unitary decomposition in an algorithm should be done with care. For one, as has been mentioned before, to decompose a unitary matrix many different classical mathematical algorithms are used, such as Singular Value Decomposition and Eigenvalue Decomposition. So if the purpose of the quantum algorithm is to find the eigenvalues of the matrix by executing it as a unitary gate, then the translation of the input matrix to an executable circuit already includes getting the eigenvalues. Users should therefore examine why they want to execute the general unitary gate before using decomposition.

Any algorithm that includes unitary decomposition will also include generation of a unitary matrix to decompose. This means that the user probably has more information on the resulting matrix than is assumed by the decomposition algorithm, and may therefore make a much more efficient quantum circuit by hand than the one resulting from the unitary decomposition in OpenQL. If the matrix generating algorithm includes calculation of the (Kronecker) product of multiple matrices, then the decomposition should be used for the separate matrices rather than the product as this will result in a shorter total circuit. The same goes for any operation that includes smaller arbitrary unitary gates inside bigger circuits, in which case the decomposition should only be applied to the arbitrary gates and not to the total matrix corresponding to the circuit. In short, any information besides "this generates a unitary matrix" should be used by users of the unitary decomposition in order to generate shorter circuits.







# Verification and unit tests

Unit tests verify code, but can also be used to design the final product. When tests are developed first, they determine what the exact implementation will look like and what the functionality should be. This means that the input and output of the feature can and need to be completely specified before starting to write the code that will transform the input to the output. Then, the code can be written with these things clear in mind, and immediately verified as well.

Tests for the compiler implementations proposed in this thesis will be simple examples as well as potential full algorithms that use unitary decomposition. Both of these are necessary, as the simple tests verify single parts of the functionality, and help narrow down any potential bugs or errors in the implementation. While the full algorithms need to be tested to verify the total decomposition functionality in a way similar to how they will be used.

A quantum genome sequencing algorithm that uses matrix decomposition can be found in Listing D.1. This code calls the `QAS_opql` function, which generates the cQASM code by printing it to the program output. This can be found in Listing D.2.

## 7.1. The different types of unit tests

A unit test is a small application that aims to test a single thing. It consists of the code that will be tested and some test-output to which it will be compared. In this project, the existing unit-test structure will be used, which is `pytest`. The tests are run by executing `pytest`, and each test is defined as a function with some pass condition.

The pass conditions of the tests fall in these categories:

- Output file is the same as a reference file
- Executes without any exceptions or errors
- Raises the expected exception
- When run using a simulator, the circuit generates the expected result

## 7.2. Output file comparison

When comparing the output file, the program output is compared to some golden reference file. As OpenQL outputs `.qasm` files, these can easily be compared to another `.qasm` file. These reference files can be handwritten, when the output is determined beforehand by a requirement. The reference files can also be generated by running the program and verifying the output file. This is useful when writing tests for code that will be changed later, as it provides a check that the code still gives the same, correct, output. Because unit tests are used to continually keep checking the complete compiler, any changes or new functionalities that affect the compiler output are continually checked.

The output file comparisons are written for single qubit 90 degree rotation gates that will be decomposed. The exact decomposition of these gates is easy to check, since they only describe a 90 degree rotation on the Bloch sphere.

A test for a simple gate looks like this:

```

1     def test_unitary_decompose_X(self):
2         config_fn = os.path.join(curdir, 'test_cfg_none.json')
3         platform = ql.Platform('platform_none', config_fn)
4         num_qubits = 1
5         p = ql.Program('test_unitary_pass', platform, num_qubits)
6         k = ql.Kernel('akernel', platform, num_qubits)
7
8         u = ql.Unitary('u1', [ complex(0.0, 0.0), complex(1.0, 0.0),
9                                complex(1.0, 0.0), complex(0.0, 0.0)])
10        u.decompose()
11        k.gate(u, [0])
12
13        p.add_kernel(k)
14        p.compile()
15
16        gold_fn = rootDir + '/golden/test_unitary-decomp_1qubit_X.qasm'
17        qasm_fn = os.path.join(output_dir, p.name+'.qasm')
18        self.assertTrue( file_compare(qasm_fn, gold_fn) )

```

Which compares the written output to the following file:

```

1 version 1.0
2 # this file has been automatically generated by the OpenQL compiler
3   please do not modify it manually.
4
5 qubits 1
6
7 .akernel
8   rz q[0], -1.57080
9   ry q[0], 3.141593
10  rz q[0], 1.570796

```

It is also possible to check these decompositions manually, by calculating the reconstructed unitary using these angles. If this matches the original unitary up to the global phase then the decomposition is correct. For a single unitary, the decomposition of a I, Y and Z gate is also checked. These tests are shown in Listings D.9 to D.12.

### 7.3. Exceptions and errors

The second type of test is rather straightforward, it consists of compiling and executing an example program and verifying that it doesn't throw an exception. The other option, which is just as important to check, is running some code that should generate an exception and verifying that it does and that it is the correct kind of exception.

Most of the other tests that are showcased in this section also implicitly test whether exceptions occur, as they test for output. And when an exception occurs, no output is generated and therefore these other tests will also fail. Then, there are several tests that just exist to verify that no exception occurs when the code is executed. One of these is Listing D.5, which exists to define the input and output of the decomposition functionality. It verifies whether the decomposition of an identity gate is allowed. The other test that just verifies no exception occurs is Listing D.25, where a 6 qubit unitary is added but not decomposed. This test is implemented because decomposing a 6 qubit unitary becomes already very computationally expensive, but it is useful to verify that it can be added to the program. This is the biggest matrix that is tested in the test suite, as a 7 qubit unitary means a matrix of 16k entries ( $(2^7)^2$ ), which was determined to use too much memory to be used in a test suite.

Besides checking that no errors occur, there are also many tests that check whether the correct exception occurs when specific things are done which are not allowed. Such as:

- adding a non-decomposed unitary to the kernel (Listing D.6).
- when a unitary is applied to a wrong number of qubits (Listings D.7 and D.8).
- when a non-unitary matrix is decomposed (Listing D.15).
- a matrix is decomposed instead of an array (Listing D.16).

## 7.4. Verifying the result

Finally, the last kind of test uses the quantum simulator (QX). In this case, it checks whether the output of the test, a qasm file, gives the expected result. Such a test is shown in Listing 7.1. For unitary decomposition, these tests are structured as follows: the test program is made as normal, and a qasm file containing the gates from the decomposition of the unitary matrix is generated. This file is set as the input file for QXelerator, which is a python library that is used to call the QX simulator inside a python file. QXelerator then executes the file, and outputs the final state of the qubits. This is then parsed using a regular expression to get a list of the probabilities of each state, code shown in code fragment D.3. This is then compared to the input matrix. This comparison is based on the mathematics from the application of the unitary gate.

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \cdot \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} A \cdot \alpha + B \cdot \beta \\ C \cdot \alpha + D \cdot \beta \end{pmatrix} \quad (7.1)$$

$$P_{meas} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = Real(A \cdot \alpha + B \cdot \beta)^2 + Imag(A \cdot \alpha + B \cdot \beta)^2 \quad (7.2)$$

$$P_{meas} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = Real(C \cdot \alpha + D \cdot \beta)^2 + Imag(C \cdot \alpha + D \cdot \beta)^2 \quad (7.3)$$

So the test compares the total probability of measuring a state as calculated by QX, with the mathematical probability of measuring the state as calculated using the equations above, using the input matrix and the initial state of the qubits.

So for two qubits in the initial state '00' or  $\begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$ , and unitary matrix (gate) of  $\begin{pmatrix} A & B & C & D \\ E & F & G & H \\ I & J & K & L \\ M & N & O & P \end{pmatrix}$ , the

probability of measuring a '00' after applying the unitary gate is  $(Real(A)^2 + Imaginary(A)^2)$ . This is calculated using the helper\_prob function shown in code fragment D.4. So if this matches the probability of measuring a '00' state that comes out of QX, then the decomposition of (at least part of) the unitary matrix is correct.

Listing 7.1: Unit test using QX to verify the output

```

1 def test_usingqx(self):
2     config_fn = os.path.join(curdir, 'test_cfg_none_simple.json')
3     platform = ql.Platform('platform_none', config_fn)
4     num_qubits = 1
5     p = ql.Program('test_usingqx', platform, num_qubits)
6     k = ql.Kernel('akernel', platform, num_qubits)
7
8     matrix = [ 0.30279949-0.60010283j, -0.58058628-0.45946559j
9               , 0.04481146-0.73904059j,    0.64910478+0.17456782j]
10    u1 = ql.Unitary("testname", matrix)
11    u1.decompose()
12    k.gate(u1, [0])
13    k.display()
14
15    p.add_kernel(k)
16    p.compile()
17    qx.set(os.path.join(output_dir, p.name+'.qasm'))

```

```

18     qx.execute()
19     c0 = qx.get_state()
20
21     self.assertAlmostEqual(helper_prob(matrix[0]), helper_regex(c0)[0],
22                             5)
23     self.assertAlmostEqual(helper_prob(matrix[1]), helper_regex(c0)[1],
24                             5)

```

These tests include:

- All entries of a two qubit unitary gate, by starting the qubits in all four possible states: '00', '01', '10', '11' (Listing D.18).
- The same two qubit gate but the qubits are first put into the bell state:  $0.707|00\rangle + 0.707|11\rangle$  (Listing D.19).
- The same two qubit gate, but the qubits are first put into a full superposition state:  $0.5|00\rangle + 0.5|01\rangle + 0.5|10\rangle + 0.5|11\rangle$  (Listing D.20).
- A three qubit gate, where the qubits are first put into a fully entangled state in order to check that the full matrix has been correctly decomposed (Listing D.21).
- The same, but for a four qubit gate (Listing D.22).
- A five qubit gate, but for two different start conditions ( $|00000\rangle$  and  $|10011\rangle$ ) in order to spot check these two columns. Since the second start condition is from the right half of the matrix, this guarantees at least the first two steps of the recursion are completely correct\* (Listings D.23 and D.24).

## 7.5. Corner cases

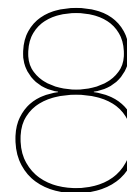
In addition to functionality of error messages and the correctness of the decomposition, the compiler is also tested using cases that could potentially give issues. Because the code should keep functioning even if people use more than one gate, a very long gate name, interleave different unitaries or other technically allowed things.

The first of these is adding multiple unitaries, and decomposing them one after the other. This could give issues if the unitary class is not handling the different unitaries separately. So two tests are implemented, one where three times a unitary is defined, decomposed and added to the kernel, and one where three times a unitary is defined, then all three are decomposed and all three are added to the kernel. Both of these tests should give the same output, and not give any issues or errors. The tests are shown in Listings D.13 and D.14.

Furthermore, because of the mathematical nature of the decomposition, how it handles matrices that consists mostly of zeroes should also be checked. When a unitary matrix only changes the state of a subset of the qubits it is applied to, part of the matrix becomes zero. And since the decomposition of the matrix means it is split up into parts, this can result into zero matrices being exposed and used as input for further decomposition steps. This can lead to issues. In fact, when these "very sparse matrix" tests were first implemented, the decomposition gave "demultiplexing is not correct" errors. When these were ignored (commented out), then the resulting circuit gave almost the same measuring probabilities as would be expected from the original matrix. But due to the decomposition errors, the measurement probabilities of qubits that should not be affected were affected slightly. So a qubit that starts out as a '0', and that should not be affected by the unitary gate, ended up with a small probability of being a '1' nonetheless. The implementation of these tests led to an overhaul of how these non-affected qubits were handled. First, a two qubit unitary where only one of the qubits is affected were tested separately, to verify both sides of the optimization (Listings D.28 and D.29). Then, a four qubit unitary where only two qubits are affected, tested using a fully entangled state. This verifies that the optimizations work even when multiple are applied (Listing D.27).

---

\*The first step of the decomposition only uses the left half of the matrix



# Performance

This chapter will outline the various metrics that were used to measure how "good" the decomposition implementation is, by measuring the gate count, execution time and memory use, the latter also compared to Qubiter.

Execution time and memory are from tests run on my HP EliteBook from 2013\*. If a better computer is used to execute the unitary decomposition it will take less time, but these tests are still useful to demonstrate the execution time compared to other implementations, such as Qubiter. As well as to estimate the speed of the algorithm on faster systems, and to estimate how the performance scales for bigger input matrices.

First, in Section 8.1 the metrics are outlined that will be used to judge the performance of the decomposition. Then, Section 8.2 explains why some metrics will not be used, because they would only be valid for specific implementations. Sections 8.3 to 8.5 show the calculated and measured gates, execution time and memory use of the current implementation, up to 9-qubit gates. The execution time was compared to Qubiter, in Section 8.6. Finally, some predictions for the execution time and memory use scaling are made in Section 8.7, and some limitations are discussed in Section 8.8.

## 8.1. Metrics

The metrics used to judge the quality of the decomposition are:

- Number of generated gates
- Execution time of the decomposition of a matrix to QASM
- Memory use of the decomposition algorithm

The number of generated gates relates to execution of the circuit on real qubits. When fewer gates are generated, it follows that the execution of the decomposition on quantum computer will be faster. This holds up perfectly for the execution on a simulator, where the number of generated gates relates directly to the number of matrices that need to be multiplied to get the final result, and therefore to the time needed to run the circuit.

The other side of the decomposition is the resources needed by the compiler, which are relevant for the users of OpenQL. These are the execution time for the decomposition, which happens when a program written Python or C++ is compiled that uses it. The execution time is measured as the wall-clock time from the start to the end of the decomposition. This will influence the compile time of any algorithms written in OpenQL that uses unitary decomposition.

The other relevant metric for users of unitary decomposition is the memory that is needed while decomposing. Either this or the execution time will be the limit for the size of the unitary matrices that can be decomposed.

---

\*6 years ago

## 8.2. Metrics that are not used

Some other potential metrics are **not** used to judge the decomposition, mostly because they relate to specific potential implementations on real qubits. These are included to show why number of generated gates will be the only metric relating to real execution of the generated circuits. These are:

- Potential for parallel operations
- How many additional operations are needed from support operations before execution of the circuit<sup>†</sup>.
- Speed of the specific universal set of gates that was chosen

The parallelism of all the compared algorithms (Section 6.5) is very similar, because all of them implement recursion in which the leading qubit is discarded at each step. This means that for all of these, there are very few operations performed on the leading qubit, some more on the second, etc. This limits any parallelism that might arise from these types of circuits. Besides that, there might exist further physical limitations on the parallelism on a real quantum implementation, which will not be accounted for in this high-level, general implementation.

When unitary decomposition is used on real quantum circuits or more realistic simulations, then it is very probably that some support operations will be needed. This includes everything from mapping of the circuit to replacing the generated gates with (multiple) different ones. These are all quantum implementation specific, as each implementation differs on which gates are possible, and which sets of qubits allow multiple qubit gates such as CNOTs. Because they are implementation specific, and because these concerns are something for the lower layers of the full stack, things like that will not be taken into account.

Finally, the universal set of gates for the implemented unitary decomposition are Rotation-Y, Rotation-Z and CNOT gates. Depending on the implementation, this might be a slower implementation than another universal gate set. This again dependent on the specific implementation, and not relevant for execution on a simulator on perfect qubits.

Therefore, the number of generated gates is the only metric relating to the generation of circuits that will be used.

## 8.3. Number of generated gates

The total number of CNOT gates generated by OpenQL, Qubiter and the theoretical minimum are shown in Figure 8.1.

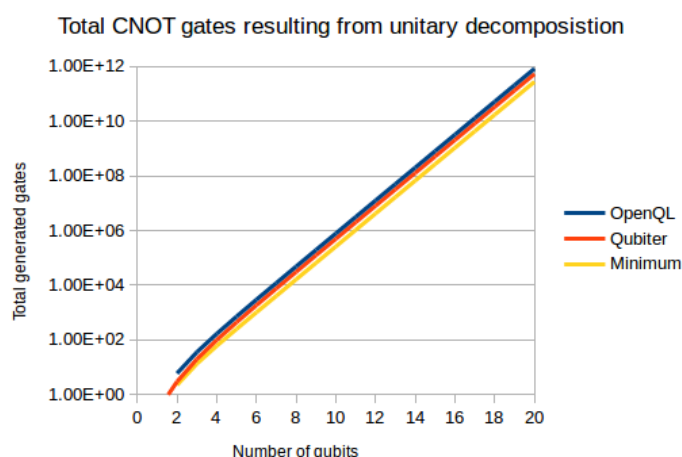


Figure 8.1: The number of CNOT gates generated by OpenQL, Qubiter and the theoretical minimum for unitary matrices up to 20-qubit gates

<sup>†</sup>Such as mapping or replacement of gates

Number of qubits	OpenQL	Qubiter	Minimum
1	0	0	0
2	6	3	3
3	36	20	14
4	168	100	61
5	720	444	252
6	2,976	1,868	1,020
7	12,096	7,660	4,091
8	48,768	31,020	16,378
9	195,840	124,844	65,529
10	784,896	500,908	262,137

Table 8.1: The number of generated CNOTs for decomposition of  $n$ -qubit gates, for OpenQL, Qubiter and the theoretical minimum

These all scale with  $4^n$ , where  $n$  is the number of qubits that the unitary gate is applied to. And while the number of generated gates end up in the same ball-park, the number of gates generated by OpenQL for a decomposition is about three times the minimum. Qubiter fares slightly better, generating 1.9 times as many gates as the minimum. So there is quite a bit of improvement possible to bring the OpenQL implementation to the level of Qubiter, and for both to get to the minimum required number of gates.

However, even the minimum number of gates still becomes quite large very quickly, which might make execution of the generated circuit not realistic on a real implementation or even on a simulator for bigger gates. But if exact unitary decomposition is the best possible option for a specific algorithm, such as Quantum Associative Memory, then all methods for unitary decomposition will generate a similar number of gates.

## 8.4. Execution time

The most important metric considered will be the execution time of the decomposition. This is the wall-clock time between the start and the end of the code that translates a unitary matrix and outputs it as a QASM file. This is the compilation time of a Python or C++ program that uses OpenQL to generate QASM, not the time needed to compile the OpenQL compiler. How much time the execution of the decomposition takes, gives an indication of the total efficiency, computational cost and well, total time to execute the decomposition.

The code for timing the different parts of the decomposition is shown in Listing 8.1. The timing is split into 5 parts. The first consists of all the preamble, such as setting the configuration file and instantiating the platform and the kernel. This is expected to take a constant amount of time for any input matrix size. The second is adding the matrix, which instantiates the unitary class. This is expected to increase for bigger matrix size, as this involves copying the input matrix into the new "Unitary" object. The third is the `decompose()` function, which checks whether the matrix is unitary, does the decomposition of the matrix and generates the list of rotation-gate angles. This step is recursive, and is expected to increase quite a lot for bigger matrix sizes. The third is adding the matrix to the kernel, which recursively generates the gray-code to calculate the CNOTs, and adds the rotation gates to the circuit using the angles from the `decompose()` function. This should scale with the number of generated gates as this makes a new gate object for each new gate. Finally, the time taken by the last generic parts of the program, so making the Program-object, adding the kernel to it, and compiling the program to generate the QASM. This should also scale with the number of generated gates, as each one is handled by OpenQL to print as QASM.

Listing 8.1: Testing the execution time for the decomposition for varying iterations and number of qubits

```

1 import os; from openql import openql as ql; import numpy as np; import time;
  import gen_unitary as gu
2
3 aggregated_matrix = 0; aggregated_decompose = 0; aggregated_kernel = 0;
  aggregated_program = 0; aggregated_start = 0
4 iterations = 1
5 nqubits = 8

```

```

6 initial_total_time = time.time()
7 for i in range(0, iterations):
8     matrix = gu.gen_unitary_array(nqubits)
9     initial_time = time.time()
10    curdir = os.path.dirname(__file__)
11    config_fn = os.path.join(curdir, 'hardware_config_qx.json')
12    platf = ql.Platform("starmon", config_fn)
13    k = ql.Kernel("newKernel", platf, nqubits)
14    aggregated_start += time.time() - initial_time
15    u1 = ql.Unitary("testname", matrix)
16    aggregated_matrix += time.time() - initial_time
17    u1.decompose()
18    aggregated_decompose += time.time() - initial_time
19    k.gate(u1, range(0, nqubits))
20    aggregated_kernel += time.time() - initial_time
21
22    k.display()
23    p = ql.Program('example', platf, nqubits)
24    p.add_kernel(k)
25    p.compile()
26    aggregated_program += time.time() - initial_time
27 print("average time preamble:\n", aggregated_start/iterations, "\n", "average
    time adding matrix:\n", aggregated_matrix/iterations, "\n", "average time
    decomposing:\n", aggregated_decompose/iterations, "\n", "average time
    adding to kernel:\n", aggregated_kernel/iterations, "\n", "average time
    total program:\n", aggregated_program/iterations)

```

If there are any significant trend-breaks in the execution time, this might indicate that certain sub-algorithms are only optimized up to a certain matrix-size, or that the matrices become too big to hold in a level of the cache, at which point writing to- and from disk starts taking up more time.

The results from running this test with varying number of iterations for each qubit size from 1 through 9 is shown in Figure 8.2, generated from Table 8.2. For reference, a circuit without decomposition was also timed, in which case a single Hadamard gate and a CNOT gate were added to the kernel and compiled. The latter will give an indication of the timing of certain components without decomposition, and whether these increase when decomposition becomes involved. The scale was chosen to be logarithmic on the y-axis, as each bigger size of unitary matrix resulted in an order of magnitude longer execution time for the total decomposition. However, the matrix sizes also grows exponentially, as  $4^n$ . Therefore, any straight lines on the plots actually mean something scales linearly with matrix size.

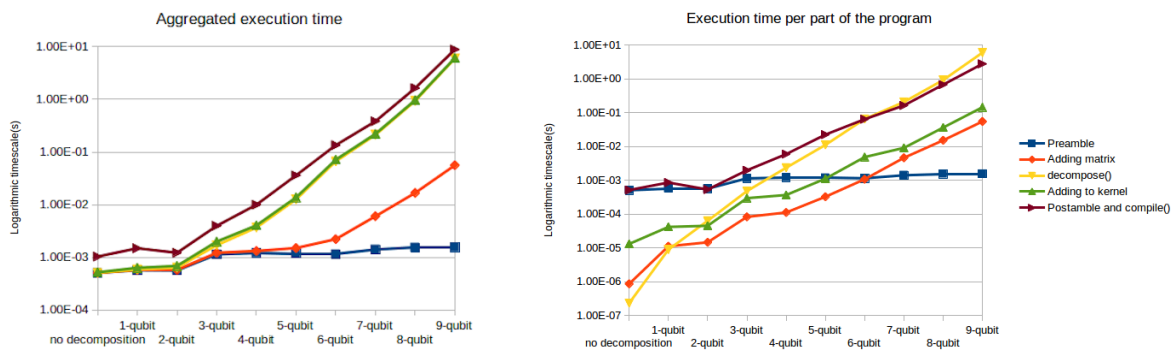


Figure 8.2: Execution time for the timed intervals, for different sizes of unitary matrices

As can be seen in the figure, the constant part of any OpenQL program, the preamble, takes approximately 1 millisecond. The postamble starts taking up more time, when the unitary matrix that gets decomposed gets bigger. This is due to the instantiating and adding of each generated gate to the circuit. Besides that, adding the matrices, decomposing them and adding them to the kernel all start very fast, and then start taking linearly more time when the input matrix gets bigger. This is according to expectations, as both matrix size and the size of the generated circuit grow with  $4^n$ , and therefore



the time needed for these functions also grows with  $4^n$ .

Size of gate	Preamble	Adding matrix	decompose()	Adding to kernel	Postamble and compile()
no decomposition	0.0005069602	5.08E-04	5.08E-04	5.21E-04	0.0010369518
1-qubit	0.0005723572	5.84E-04	5.93E-04	6.35E-04	0.001491971
2-qubit	0.0005655217	5.80E-04	6.44E-04	0.0006894064	0.0012233925
3-qubit	0.0011408281	0.0012237811	1.70E-03	0.0019996858	0.0039744115
4-qubit	0.0012120962	0.0013241529	0.0036828542	0.0040509653	0.0100028992
5-qubit	0.0011809111	0.0015077591	0.0126505136	0.0137670994	0.0361521006
6-qubit	0.0011463165	0.0022214055	0.0667135119	0.0715714097	0.1353835464
7-qubit	0.0014121532	0.0060400963	0.2124438286	0.2215316296	0.3842484951
8-qubit	0.001537323	0.0167648792	0.9336073399	0.9697782993	1.6395030022
9-qubit	0.0015523434	0.0562000275	5.9757013321	6.1196198463	8.8728892803

Table 8.2: Aggregated execution time for the decomposition per line of code in seconds

## 8.5. Memory allocation

To measure how much memory the decomposition was using, at first a Python package called "memory\_alloc" was used. But this package gave a steady use of 584 bytes for allocating the unitary object, and similar values for all other lines of code in the test program. Even for big unitary matrices, where the matrix alone takes more than 584 bytes to store.

As this obviously was not the correct memory usage of the program, a different library for measuring memory use was used, called memory\_profiler. The test code used with this profiler is shown in Listing 8.2, and the results are shown in Figure 8.3, generated from Table 8.3. As you can see, Python allocates memory in blocks. When the further parts of the program don't use up more memory, no more memory is allocated. All the python programs take at least 39 MB, and as expected, start taking exponentially more memory when the gates get bigger. From about a 6 qubit gate, the total memory use starts rising exponentially. Although this is not noticeable in the execution time of the decomposition, as the execution time per line of the program does not suddenly start increasing at this point, as seen in Figure 8.2.

Below 6 qubit unitary matrices, so for input matrices smaller than 1024 items, the memory allocation seems to be somewhat constant. After the initial 40 MB, not a lot of additional memory is allocated, if at all. From a 6 qubit unitary, so 4096 matrix elements of each 16 bytes (complex doubles), so storing just the input matrix already takes 65KB. The memory use above that is taken up by the in between matrices, calculations, the list of rotation angles and the generated quantum circuit. All of these start taking up memory at the same rate as the input matrix and output circuits grow in size, namely  $4^n$ .

The thing that take up the main chunk of the memory is not decomposition specific, but because of the length of the resulting circuit. This is adding the unitary to the kernel, when each single rotation gate and CNOT is instantiated and added to the circuit. Besides that, the compilation of the whole circuit takes up a big chunk, as each gate gets translated to QASM with a few steps in between. The decomposition function is relatively lightweight compared to these bigger operations, at least when looking at the allocated memory.

Size of gate	start	Matrix =	U1 = ql.Unitary	u1.decompose	k.gate(	p.compile
no decomposition	40.476562	0.00	0.00	0.00	0.00	0.00
1-qubit	40.703125	0.00	0.00	7.23E-01	0.00	0.00
2-qubit	39.62	0.00	6.02E-01	0.00	0.00	0.00
3-qubit	40.589844	0.00	0.00	0.722656	0	0.00
4-qubit	40.277344	0.00	0.00	0.66	0.00	0.558594
5-qubit	39.79	0.00	0.00	0.00	0.85	0.73
6-qubit	40.703125	0.433694	0.546875	0.316406	1.875	2.117188
7-qubit	40.628906	0.570312	1.320312	1.359	7.476562	4.679688
8-qubit	40.546875	1.265625	5.125	5.191406	30.124062	22.207
9-qubit	40.195312	4.203125	20.136719	11.91	126.129	87.55

Table 8.3: Additional memory use of the decomposition algorithm in MB per line of code

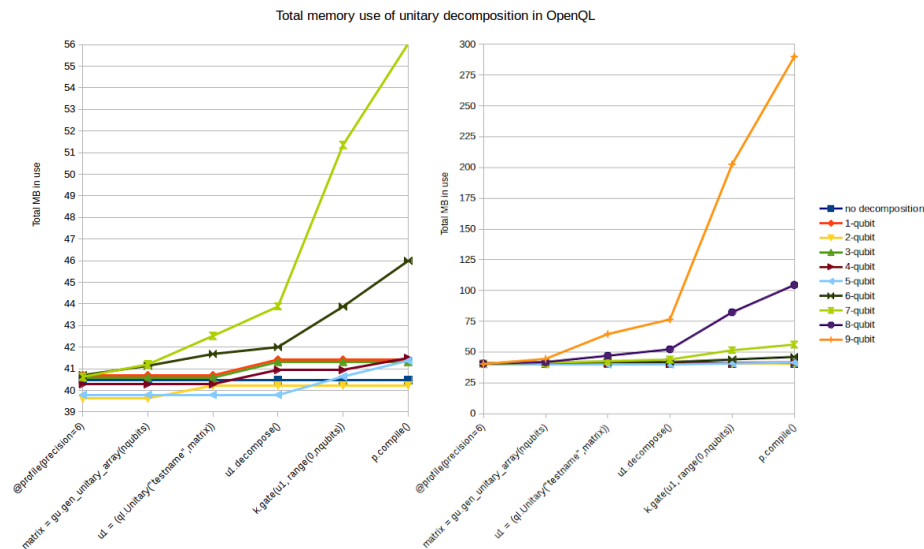


Figure 8.3: Total memory allocated by the program described in Listing 8.2 in MB

Listing 8.2: Test program used to measure the memory use of the decomposition

```

1 import linecache; import os; from openq1 import openq1 as ql; import numpy as
  np; import time; import gen_unitary as gu; from memory_profiler import
  profile
2
3 @profile(precision=6)
4 def my_func():
5     nqubits = 4
6     matrix = np.load('outputfile_' + str(nqubits) + ".np1").flatten()
7     curdir = os.path.dirname("/home/anneriet/Afstuderen/Results/")
8     config_fn = os.path.join(curdir, 'hardware_config_qx.json')
9     platf = ql.Platform("starmon", config_fn)
10
11     k = ql.Kernel("newKernel", platf, nqubits)
12     u1 = (ql.Unitary("testname", matrix))
13     u1.decompose()
14     k.gate(u1, range(0, nqubits))
15     p = ql.Program('example', platf, nqubits)
16     p.add_kernel(k)
17     p.compile()
18
19 if __name__ == '__main__':
20     my_func()

```

## 8.6. Execution time compared to Qubiter

The test program in Listing 8.3 was used to compare the execution time of the unitary decomposition of Qubiter to the execution time using OpenQL, resulting in the total execution times shown in Figure 8.4 and Figure 8.5.

As can be seen in the figures, the total execution time of the decomposition is much lower for the openQL implementation, by about an order of magnitude. This clearly shows that the C++-implementation is faster than the Python-based Qubiter implementation of the decomposition algorithm.

This could be because Qubiter makes a file for the intermediate representations of the circuit, and then has to open that file for the next iteration. Other potential causes are that the execution of the algorithm is slower because C++ code is compiled to native code when the program is compiled, while Python code is interpreted at runtime. Because both are used as libraries, the compile time of the

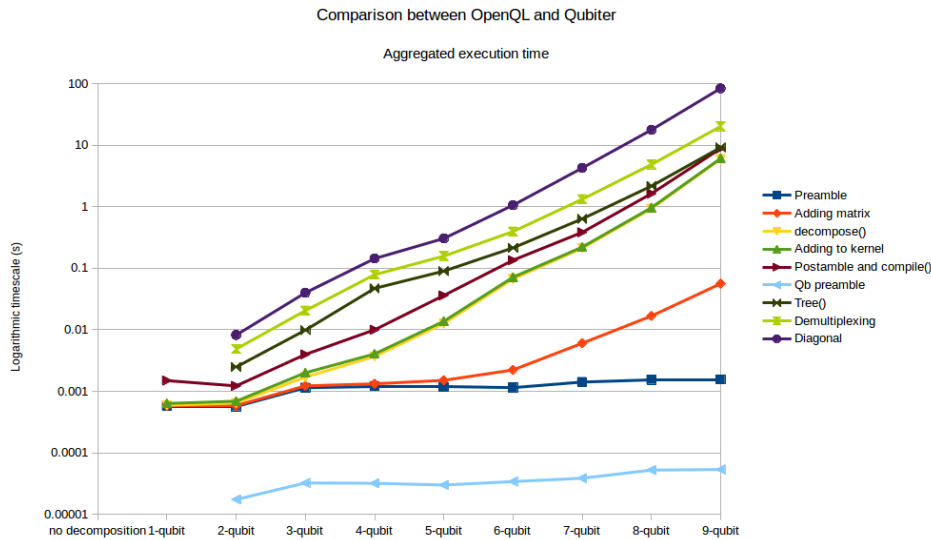


Figure 8.4: Comparison of aggregated execution times between OpenQL and Qubiter for decomposition of unitary gates of different sizes

OpenQL (C++) implementation does not influence the compilation or runtime of an application using the library. Finally, the cause of the longer runtime might be in the execution of the time-intensive mathematical algorithms in the decomposition, but this is unlikely as both Eigen, the math library used in OpenQL, and numpy, the math library used in Qubiter, run on the same internal math libraries, LAPACK and BLAS.

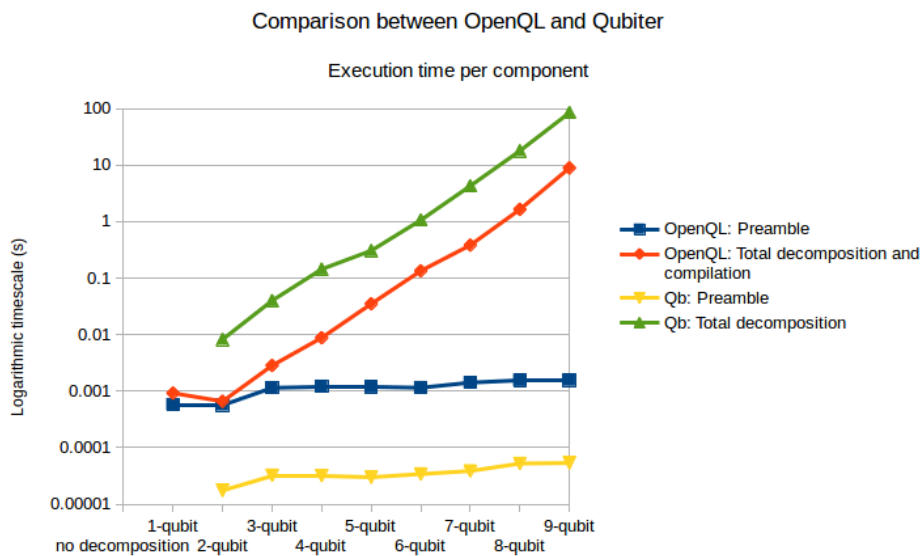


Figure 8.5: Comparison of execution times for decomposition and preamble between OpenQL and Qubiter

Listing 8.3: Test program used to measure execution time for Qubiter and OpenQL decomposition

```

1 import numpy as np; import gen_unitary as gu ;import os ; import sys; import
  time; import math
2 rootDir = os.path.dirname(os.path.realpath(__file__)); curdir = os.getcwd();
  sys.path.insert(0,os.getcwd()); sys.path.append(os.path.abspath(rootDir))
3
4 from quantum_CSD_compiler.Tree import *
```

```

5 from quantum_CSD_compiler.DiagUnitarySEO_writer import *
6 from quantum_CSD_compiler.MultiplexorSEO_writer import *
7 from quantum_CSD_compiler.MultiplexorExpander import *
8 from quantum_CSD_compiler.DiagUnitaryExpander import *
9 import pandas as pd;
10
11
12 initial_time_qb = 0; aggregated_preamble_qb =0; aggregated_tree = ;
    aggregated_mp = 0; aggregated_diag = 0
13
14 iterations = 1
15 nqubits = 7
16 initial_total_time = time.time()
17
18 for i in range(0, iterations):
19     init_unitary_mat = gu.gen_unitary(nqubits)
20     initial_time_qb = time.time()
21     emb = CktEmbedder(nqubits, nqubits)
22     file_prefix = 'qubiter_timing'
23
24     aggregated_preamble_qb += time.time() - initial_time_qb
25     t = Tree(True, file_prefix, emb, init_unitary_mat, verbose=False)
26     t.close_files()
27     aggregated_tree +=time.time()-initial_time_qb
28     wr = MultiplexorExpander(file_prefix, nqubits, 'exact')
29     aggregated_mp += time.time() - initial_time_qb
30
31     zr = DiagUnitaryExpander(file_prefix + '_X1', nqubits, 'exact')
32     aggregated_diag += time.time() - initial_time_qb
33
34 print("qb: average time preamble:\n", aggregated_preamble_qb/iterations, "\n",
    "qb: average time tree:\n", aggregated_tree/iterations, "\n", "qb: average
    time demultiplexing:\n", aggregated_mp/iterations, "\n", "qb: average time
    diagonal:\n", aggregated_diag/iterations)

```

Size of gate	Preamble	Tree()	Demultiplexing	Diagonal
no decomposition				
1-qubit	gives error			
2-qubit	1.74E-05	0.0024805548	0.0048608384	0.0082185013
3-qubit	3.23E-05	0.0098646755	0.0204878032	0.0399154189
4-qubit	3.20E-05	0.0471440945	0.0786870394	0.1438712831
5-qubit	2.99E-05	0.0899846458	0.1578747511	0.3058763647
6-qubit	3.40E-05	0.2151622057	0.3959766078	1.0626960349
7-qubit	3.86E-05	0.6363847971	1.3273614645	4.2765287399
8-qubit	5.23E-05	2.1768739223	4.870875597	17.8260270357
9-qubit	5.34E-05	9.230694294	20.3065607548	84.283356905

Table 8.4: Aggregated execution time for the decomposition per line of code using Qubiter in seconds

## 8.7. Expected scaling for bigger gates

My laptop has a Intel Core i7-3610QM, running on 2.3 GHz, with 4 cores and 6MB of cache. AWS c5 instances have a sustained frequency of 3.6GHz, and say, we use a c5.4xlarge of 16 cores, then this system is about 6 times as fast as my laptop. Probably more, as my laptop is also running this overleaf file, for example, and therefore does not use the complete CPU for the decomposition.

So, approximate execution time on my laptop is 10 seconds for a 9 qubit gate, and every added qubit means the whole decomposition takes about 4 times as much time. So a 10 qubit gate takes 40 seconds, etc. For the c5.4xlarge, a nine-qubit gate would take 3.5 seconds, and the 10-qubit gate an estimated 6.5. Taking this line of reasoning further, that means the total execution time will be similar to the one shown in Figure 8.6. So decomposing a 20 qubit gate on a single c5.4xlarge node, assuming

perfect memory scaling and parallelism, would take about 9000000 seconds, or 104 days. As this is a very long time, it is probably wise to avoid decomposition from such big unitary matrices. For algorithms where it is needed, however, decomposition up to 14 qubit gates will probably take less than one hour.

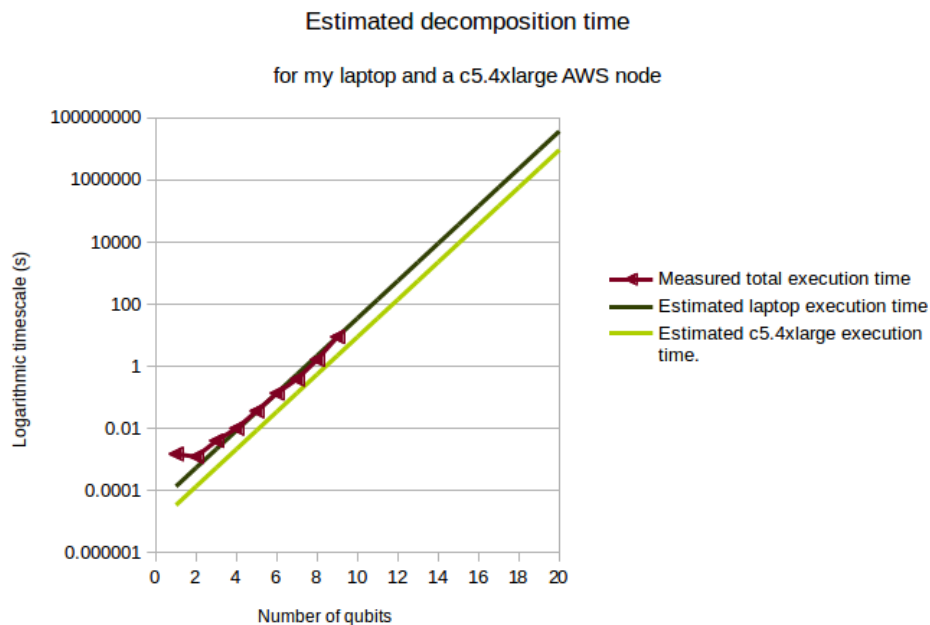


Figure 8.6: Real and estimated execution time for my laptop (HP Elitebook 8570w from 2013) and a c5.4xlarge AWS node.

When the memory use is extrapolated, as shown in Figure 8.7, it is clear that the total decomposition of bigger gates takes a lot of memory. Decomposing a 16 qubit gate takes one terabyte, and a twenty qubit gate one zettabyte, or  $10^{15}$  bytes. This is more than the current biggest supercomputer. The real and predicted memory use for the complete decomposition are shown in Figure 8.7. The estimate was made by taking the real memory use from 5 qubit unitary gates and bigger, because from then on the matrix size starts to have an influence on the memory allocation. The base memory allocation of about 40 MB was deducted, and the result scaled up with  $4^n$ , just like the matrix size. After adding the base memory allocation again, this estimate matched up well with the memory use from 5 to 9 qubit gates. Though most of this memory use is taken up by the storing of the individual gates in the final circuit.

## 8.8. Limitations

Even the minimum number of generated gates scales with  $4^n$ , so each bigger size unitary gate results in approximately four times as many gates, and as illustrated by the memory use and execution time measurements and calculations, unitary decomposition takes a lot of computational resources. This way of doing unitary decomposition is thus viable up to a certain number of qubits, but anything bigger than that becomes unfeasible very quickly for bigger input matrices.

### 8.8.1. Size of the circuit

The decomposition can be hindered by all of the shown metrics, but for short-term implementation the number of generated gates will be the biggest issue. If the assumption of perfect qubits is even slightly relaxed, then mapping operations, gate errors and decohering of the qubits become big issues, because of the size of the generated circuit. Even three qubit gates already take 84 rotation gates and 36 CNOTs.

This can be mitigated by modifying the decomposition to a specific implementation, by using the physical layout of the qubits to implement nearest-neighbour CNOTs in the multi-controlled rotation gates. Or by using specific gates other than the universal gateset that was chosen, such as controlled-Z gates instead of CNOTs, and modifying the circuit so that more operations might be performed in parallel to minimizing the total execution time of the circuit on a real or more realistic quantum system.

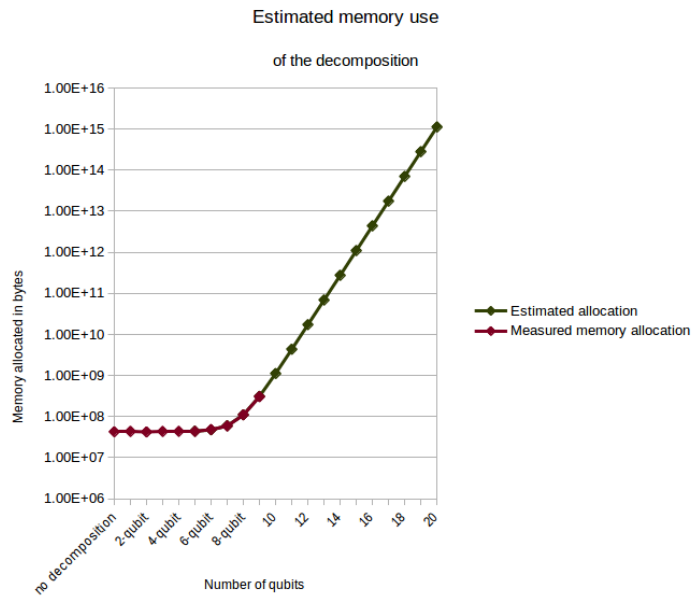


Figure 8.7: Real and estimated memory allocation for decomposition up to 20 qubits.

Besides that, the decomposition can be tuned to the specific algorithm, by using the structure of the algorithm to minimize the size of the unitary matrices that will need to be decomposed. This can be done by manually implementing some parts of the circuit and using unitary decomposition only for small parts, or by using unitary decomposition on the separate matrices of which a Kronecker product is desired. This is only useful when the size can be reduced of the input matrix to the decomposition, or when specific optimizations can be performed on all the individual matrices of a matrix product.

### 8.8.2. Execution time and memory use

If the circuit itself is not an issue, then the execution time and memory use can become the limit of the decomposition. This can be mitigated by executing on a supercomputer, cluster or just waiting a long time. But even so, decomposition of 20-qubit gates will probably not be doable. Decomposing an arbitrary unitary gate bigger than 15 qubits will start taking more than a petabyte of memory, so decomposing gates bigger than this is not really feasible on most current systems. For supercomputers, decompositions of 19 qubit gates would just be possible, with an estimated memory use of 280 petabyte, but the resulting 800 trillion gates will be an issue much earlier.

The memory use of the decomposition can be reduced, however, because it is not the decomposition algorithm but rather storing all of the gates that uses the most memory. This can be reduced by handling the compilation in a different way, where part of the gates are completely compiled and then written to a QASM file, and then the next part is compiled, and then the next, etc. This will reduce the total number of gates held in memory, but this will only be an issue for circuits as long as the ones coming from unitary decomposition.

Overall, the size of the generated circuit will be the biggest issue for short term implementations. But at some point in the future, memory use can become the limiting factor, in which case the memory use of OpenQL as a whole needs to be reduced.

# 9

## Conclusion and future work

In this chapter, a summary and the assembled conclusions from the whole thesis are given. Then, some possible avenues for future improvements are given, both for the unitary decomposition and OpenQL as a whole.

### 9.1. Conclusion

In conclusion, the unitary decomposition as implemented in OpenQL gives correct results, in a reasonable amount of time. The implementation being completely in C++, means that it is about 10 times as fast as currently the only other implementation of unitary decomposition. With this decomposition, algorithms such as Quantum Associative Memory can be done completely inside OpenQL, and do not require anymore manual copying and pasting of code. This makes the whole process faster and provides fewer opportunities for human error than before.

More gates are generated than the minimum required number by the current implementation, and also more than some other algorithms. However, the structure of the decomposition means that further optimizations can be easily integrated with the current program. The optimizations that were implemented can further bring down the gate count in specific cases, so that the total gate count is much lower than the illustrated worst case numbers.

The unit tests allow continuous verification of the compiler, including whether the correct errors and exceptions are thrown. And they guarantee that the decomposition algorithm gives correct results, and also allow for continuous checking for future changes to the algorithm. This makes it easier to implement future optimizations, as any issues with the current decomposition functionality will become clear immediately.

This implementation of unitary decomposition in OpenQL is exact, to the point where any inaccuracies in the decomposition are caused by the precision of the input matrix. Or that of the simulator or quantum computer the generated circuit is executed on. It might therefore be used in development of algorithms, so that a developer can translate parts of an algorithm to a quantum circuit while keeping other parts as a matrix, and still test the complete algorithm.

And when decomposing a unitary gate of a certain size, the maximum number of resulting gates can easily be calculated beforehand. So any user can determine whether they find that number acceptable before using it. This also means that, if a user has a circuit which consists of more gates than would be required by an arbitrary unitary gate of the same size, then multiplying all the gates in the circuit to get a unitary matrix and decomposing that matrix will result in a circuit with fewer gates.

But for algorithms that do require unitary decomposition, but with less precision than provided by this implementation in OpenQL, an approximate decomposition algorithm will probably result in a shorter circuit that still fulfills the requirements.

Besides the number of generated gates, the decomposition also requires a lot of advanced mathematical decompositions such as eigenvalue decomposition. Therefore, any users of the decomposition should be careful that they do not use the decomposition to get the eigenvalues of a unitary matrix. Or any other result of the unitary decomposition. In this case, the "answer" wanted by the user of OpenQL

is already calculated when they decompose the unitary. Which means that the execution of the circuit on a quantum computer is redundant.

Algorithms that use unitary decomposition also include the generation of the unitary matrix. So a user of unitary decomposition has probably more information about the resulting unitary matrix than is assumed by OpenQL, which should be used to either hand-optimize the circuit or parts of the circuit. If possible, this can significantly decrease the number of gates in the final circuit.

As the calculations for the performance metrics show, implementing the optimizations for the gate-count will result in at most 3-times less gates than the current implementation, and only 1.9 times for the currently best-known implementation. So any implementation will result in more gates than is feasibly possible in any near-term quantum applications. For longer circuits the memory use of OpenQL becomes an issue, which can be mitigated by writing the circuit immediately to QASM during the compilation.

## 9.2. Future work

There are several avenues to make this implementation of unitary decomposition better. The first is to improve the execution time for bigger matrices, by implementing a different way to diagonalize the matrices inside the demultiplexing step. If a diagonalization method is found that scales linearly with matrix elements, then the time needed to decompose bigger matrices will decrease.

To minimize execution time of the resulting circuit on a simulator or quantum computer, it is more important that the decomposition generates as few gates as possible. There are several avenues for that, such as:

- Implementing the minimum two-qubit circuit described in [39].
- Or implementing immediately also a universal three qubit gate, such as the one in [48].
- Implementing the multiplexed rotation-Y gate with a controlled-Z gate, as expressed in [39].
- Reworking the Quantum Shannon Decomposition so that the intermediate matrices cancel out, as the input matrix has fewer degrees of freedom than the matrices coming from the QSD. Therefore, it might be possible to choose some of these intermediate matrices in such a way that they can be decomposed to fewer gates.
- Other specific efficient implementations, such as controlled unitary gates (as opposed to uniformly controlled gates), or specifically optimized multi-controlled rotation gates and quantum multiplexors for certain numbers of qubits.

Finally, it might be possible to implement detailed, application specific optimizations. When the input for the unitary decomposition is known to have more constraints, then the decomposition algorithm can be tailored to take advantage of these constraints. Such as decomposition of arbitrary unitary gates that only apply right-angle operations to the qubits, or matrices that are Hermitian as well as unitary. This can then result in much lower gate-counts for these specific cases.

For Quantum Associative Memory, the quantum genome sequencing algorithm which requires unitary decomposition, it might also be possible to incorporate measurement into the recall step. This might allow for faster retrieval, but cost added computing energy. It also means that a non-unitary matrix will be translated to some version of quantum-classical hybrid code. This is not possible with the current unitary decomposition, but could be a potential future feature.

Furthermore, for near-term quantum applications, the unitary decomposition generates too many gates for unitary matrices bigger than a certain size. Depending on the error rate and accuracy of the gates, this might be as low as 3-qubit gates, to 6- or 7-qubit gates. A lot of things can be done to make unitary decomposition possible for non-perfect qubits, such as implementing a more parallel circuit, splitting it in some way so that first one half and then the other can be executed or something else.

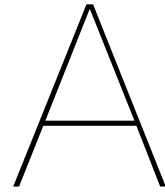
When relaxing the requirement for perfect qubits, mapping, gate accuracy and qubit decoherence also become relevant. To minimize mapping, nearest-neighbour circuits can be used, but to deal with gate accuracy and decohering of qubits some more impactful things will need to be used, in order to change the whole structure of the resulting circuit. Due to the identical structure for each decomposed circuits, it might also be possible to adjust the structure of a real quantum system so that it perfectly fits unitary decomposition, which might also mitigate these issues.



Other than improving the unitary decomposition, there are also quite some things to improve in OpenQL as a language. The first is to update the current version of cQASM (common QASM), from cQASM 1.0 to cQASM 2.0. In this second version, it is possible to use classical instructions next to the quantum instructions. This is currently not supported in OpenQL, and will increase the capabilities of OpenQL. Secondly, although OpenQL currently has a compilation step, it does not currently include full, correct optimization of all circuits. New optimizations could include automatic parallelization, removing zero rotations or otherwise redundant operations and cancelling or merging gates with their neighbours. The same goes for improving the scheduling and mapping operations. Finally, when using OpenQL for larger circuits of more than several thousand gates, the memory use starts to grow as keeping the complete circuit in memory starts taking up significant space. Even so, decomposing gates up to 15 qubits is feasible with respect to memory use, but anything bigger than that will run into issues. So to use OpenQL in a future where such long circuits become more standard, the memory use will need to be adjusted in some way.

Ultimately, the goal of all of these suggestions is to keep unitary decomposition and OpenQL relevant and useful both for near-term and future quantum applications.





## Full Code

This chapter contains the code inside the unitary.h header file and the relevant parts of the kernel.h header file. This is all the relevant c++ code pertaining to unitary decomposition inside OpenQL.

Listing A.1: Unitary decomposition code inside kernel.h

```
1 /**
2  * @file   kernel.h
3  * @date   09/2019
4  * @author Nader Khammassi
5  *         Imran Ashraf
6  *         Anneriet Krol
7  * @brief  openql kernel
8  */
9
10 // to add unitary to kernel
11 void gate(ql::unitary u, std::vector<size_t> qubits)
12 {
13     double u_size = uint64_log2((int) u.size())/2;
14     if(u_size != qubits.size())
15     {
16         EOUT("Unitary " << u.name <<" has been applied to the wrong number
17 of qubits! " << qubits.size() << " and not " << u_size);
18         throw ql::exception("Unitary '"+u.name+"' has been applied to the
19 wrong number of qubits. Cannot be added to kernel! " + std::to_string(
20 qubits.size()) + " and not "+ std::to_string(u_size), false);
21     }
22     for(uint i = 0; i < qubits.size()-1; i++)
23     {
24         for(uint j = i+1; j < qubits.size(); j++)
25         {
26             if(qubits[i] == qubits[j])
27             {
28                 EOUT("Qubit numbers used more than once in Unitary: " << u.name
29 << ". Double qubit is number " << qubits[j]);
30                 throw ql::exception("Qubit numbers used more than once in
31 Unitary: " + u.name + ". Double qubit is number " + std::to_string(qubits[j
32 ]), false);
33             }
34         }
35     }
36     // applying unitary to gates
```

```

33     COUT("Applying unitary '" << u.name << "' to " << ql::utils::to_string(
qubits, "qubits: ") );
34     if(u.is_decomposed)
35     {
36
37         COUT("Adding decomposed unitary to kernel ...");
38         DOUT("The list is this many items long: " << u.instructionlist.size
());
39         //COUT("Instructionlist" << ql::utils::to_string(u.instructionlist)
);
40         int end_index = recursiveRelationsForUnitaryDecomposition(u,qubits,
u_size, 0);
41         DOUT("Total number of gates added: " << end_index);
42     }
43     else
44     {
45         EOUT("Unitary " << u.name <<" not decomposed. Cannot be added to
kernel!");
46         throw ql::exception("Unitary '"+u.name+"' not decomposed. Cannot be
added to kernel!", false);
47     }
48 }
49
50 //recursive gate count function
51 //n is number of qubits
52 //i is the start point for the instructionlist
53 int recursiveRelationsForUnitaryDecomposition(ql::unitary &u, std::vector<
size_t> qubits, int n, int i)
54 {
55     // DOUT("Adding a new unitary starting at index: "<< i << ", to " << n
<< ql::utils::to_string(qubits, " qubits: "));
56     if (n > 1)
57     {
58         // Need to be checked here because it changes the structure of the
decomposition.
59         // This checks whether the first qubit is affected, if not, it
applies a unitary to the all qubits except the first one.
60         int numberforcontrolledrotation = std::pow(2, n - 1);
//number of gates per rotation
61
62         // code for last one not affected
63         if (u.instructionlist[i] == 100.0)
64         {
65             COUT("[kernel.h] Optimization: last qubit is not affected, skip
one step in the recursion. New start_index: " << i+1);
66             std::vector<size_t> subvector(qubits.begin() + 1, qubits.end())
;
67             return recursiveRelationsForUnitaryDecomposition(u, subvector,
n - 1, i + 1) + 1; // for the number 10.0
68         }
69         else if (u.instructionlist[i] == 200.0)
70         {
71             std::vector<size_t> subvector(qubits.begin(), qubits.end() - 1)
;
72
73             // This is a special case of only demultiplexing
74             if (u.instructionlist[i+1] == 300.0)
75             {
76
77                 // Two numbers that aren't rotation gate angles
78                 int start_counter = i + 2;

```

```

79         COUT("[kernel.h] Optimization: first qubit not affected,
skip one step in the recursion. New start_index: " << start_counter);
80
81         return recursiveRelationsForUnitaryDecomposition(u,
subvector, n - 1, start_counter) + 2; //for the numbers 20 and 30
82     }
83     else
84     {
85         int start_counter = i + 1;
86         COUT("[kernel.h] Optimization: only demultiplexing will be
performed. New start_index: " << start_counter);
87
88         start_counter += recursiveRelationsForUnitaryDecomposition(
u, subvector, n - 1, start_counter);
89         multicontrolled_rz(u.instructionlist, start_counter,
start_counter + numberforcontrolledrotation - 1, qubits);
90         start_counter += numberforcontrolledrotation; //
multicontrolled rotation always has the same number of gates
91         start_counter += recursiveRelationsForUnitaryDecomposition(
u, subvector, n - 1, start_counter);
92         return start_counter - i;
93     }
94 }
95 else
96 {
97     // The new qubit vector that is passed to the recursive
function
98     std::vector<size_t> subvector(qubits.begin(), qubits.end() - 1)
;
99     int start_counter = i;
100    start_counter += recursiveRelationsForUnitaryDecomposition(u,
subvector, n - 1, start_counter);
101    multicontrolled_rz(u.instructionlist, start_counter,
start_counter + numberforcontrolledrotation - 1, qubits);
102    start_counter += numberforcontrolledrotation;
103    start_counter += recursiveRelationsForUnitaryDecomposition(u,
subvector, n - 1, start_counter);
104    multicontrolled_ry(u.instructionlist, start_counter,
start_counter + numberforcontrolledrotation - 1, qubits);
105    start_counter += numberforcontrolledrotation;
106    start_counter += recursiveRelationsForUnitaryDecomposition(u,
subvector, n - 1, start_counter);
107    multicontrolled_rz(u.instructionlist, start_counter,
start_counter + numberforcontrolledrotation - 1, qubits);
108    start_counter += numberforcontrolledrotation;
109    start_counter += recursiveRelationsForUnitaryDecomposition(u,
subvector, n - 1, start_counter);
110    return start_counter - i; //it is just the total
111 }
112 }
113 else //n=1
114 {
115     // DOUT("Adding the zyz decomposition gates at index: "<< i);
116     // zyz gates happen on the only qubit in the list.
117     c.push_back(new ql::rz(qubits.back(), u.instructionlist[i]));
118     c.push_back(new ql::ry(qubits.back(), u.instructionlist[i + 1]));
119     c.push_back(new ql::rz(qubits.back(), u.instructionlist[i + 2]));
120     // How many gates this took
121     return 3;
122 }
123 }

```

```

124
125 //controlled qubit is the first in the list.
126 void multicontrolled_rz(std::vector<double> &instruction_list, int
start_index, int end_index, std::vector<size_t> qubits)
127 {
128     // DOUT("Adding a multicontrolled rz-gate at start index " <<
start_index << ", to " << ql::utils::to_string(qubits, "qubits: ");
129     int idx;
130     //The first one is always controlled from the last to the first qubit.
131     c.push_back(new ql::rz(qubits.back(),-instruction_list[start_index]));
132     c.push_back(new ql::cnot(qubits[0], qubits.back()));
133     for(int i = 1; i < end_index - start_index; i++)
134     {
135         idx = uint64_log2(((i)^((i)>>1))^((i+1)^((i+1)>>1)));
136         c.push_back(new ql::rz(qubits.back(),-instruction_list[i+
start_index]));
137         c.push_back(new ql::cnot(qubits[idx], qubits.back()));
138     }
139     // The last one is always controlled from the next qubit to the first
qubit
140     c.push_back(new ql::rz(qubits.back(),-instruction_list[end_index]));
141     c.push_back(new ql::cnot(qubits.end()[-2], qubits.back()));
142 }
143
144 //controlled qubit is the first in the list.
145 void multicontrolled_ry( std::vector<double> &instruction_list, int
start_index, int end_index, std::vector<size_t> qubits)
146 {
147     // DOUT("Adding a multicontrolled ry-gate at start index "<<
start_index << ", to " << ql::utils::to_string(qubits, "qubits: ");
148     int idx;
149
150     //The first one is always controlled from the last to the first qubit.
151     c.push_back(new ql::ry(qubits.back(),-instruction_list[start_index]));
152     c.push_back(new ql::cnot(qubits[0], qubits.back()));
153
154     for(int i = 1; i < end_index - start_index; i++)
155     {
156         idx = uint64_log2 ((i)^((i)>>1))^((i+1)^((i+1)>>1)));
157         c.push_back(new ql::ry(qubits.back(),-instruction_list[i+
start_index]));
158         c.push_back(new ql::cnot(qubits[idx], qubits.back()));
159     }
160     // Last one is controlled from the next qubit to the first one.
161     c.push_back(new ql::ry(qubits.back(),-instruction_list[end_index]));
162     c.push_back(new ql::cnot(qubits.end()[-2], qubits.back()));
163 }
164 // source: https://stackoverflow.com/questions/994593/how-to-do-an-integer-
log2-in-c user Todd Lehman
165 int uint64_log2(uint64_t n)
166 {
167     #define S(k) if (n >= (UINT64_C(1) << k)) { i += k; n >>= k; }
168
169     int i = -(n == 0); S(32); S(16); S(8); S(4); S(2); S(1); return i;
170
171     #undef S
172 }

```

Listing A.2: Unitary decomposition code inside unitary.h

```

1 /**
2  * @file   unitary.h
3  * @date   12/2018
4  * @author Imran Ashraf
5  * @author Anneriet Krol
6  * @brief  unitary matrix (decomposition) implementation
7  */
8
9 #ifndef _UNITARY_H
10 #define _UNITARY_H
11
12 #include <complex>
13 #include <string>
14
15 #include <ql/utils.h>
16 #include <ql/str.h>
17 #include <ql/gate.h>
18 #include <ql/exception.h>
19 #include <Eigen>
20 #include <unsupported/Eigen/MatrixFunctions>
21 #include <src/misc/lapacke.h>
22
23 namespace ql
24 {
25
26 class unitary
27 {
28 private:
29     Eigen::Matrix<std::complex<double>, Eigen::Dynamic, Eigen::Dynamic> _matrix
30     ;
31 public:
32     std::string name;
33     std::vector<std::complex<double>> array;
34     std::vector<std::complex<double>> SU;
35     double delta;
36     double alpha;
37     double beta;
38     double gamma;
39     bool is_decomposed;
40     std::vector<double> instructionlist;
41
42     typedef Eigen::Matrix<std::complex<double>, Eigen::Dynamic, Eigen::Dynamic>
43     complex_matrix ;
44
45     unitary() : name(""), is_decomposed(false) {}
46
47     unitary(std::string name, std::vector<std::complex<double>> array) :
48         name(name), array(array), is_decomposed(false)
49     {
50         DOUT("constructing unitary: " << name
51             << ", containing: " << array.size() << " elements");
52     }
53
54     double size()
55     {
56         if(!array.empty())
57             return (double) array.size();
58         else
59             return (double) _matrix.size();
60     }
61
62 };
63
64 }

```

```

59     }
60
61     complex_matrix getMatrix()
62     {
63         if (!array.empty())
64         {
65             int matrix_size = (int)std::pow(array.size(), 0.5);
66
67             Eigen::Map<complex_matrix> matrix(array.data(), matrix_size,
matrix_size);
68             _matrix = matrix.transpose();
69         }
70         return _matrix;
71     }
72
73     void decompose()
74     {
75         DOUT("decomposing Unitary: " << name);
76
77         getMatrix();
78         int matrix_size = _matrix.rows();
79
80         // compute the number of qubits: length of array is columns*rows, so
log2(sqrt(array.size))
81         int numberofbits = uint64_log2(matrix_size);
82
83         Eigen::MatrixXcd identity = Eigen::MatrixXcd::Identity(matrix_size,
matrix_size);
84         Eigen::MatrixXcd matmatadjoint = (_matrix.adjoint()*_matrix);
85         // very little accuracy because of tests using printed-from-matlab code
that does not have many digits after the comma
86         if( !matmatadjoint.isApprox(identity, 0.001))
87         {
88             //Throw an error
89             EOUT("Unitary " << name <<" is not a unitary matrix!");
90
91             throw ql::exception("Error: Unitary '"+ name+"' is not a unitary
matrix. Cannot be decomposed!" + to_string(matmatadjoint), false);
92         }
93         // initialize the general M^k lookuptable
94         genMk();
95
96         decomp_function(_matrix, numberofbits); //needed because the matrix is
read in columnmajor
97
98         DOUT("Done decomposing");
99         is_decomposed = true;
100     }
101
102     std::string to_string(complex_matrix m, std::string vector_prefix = "",
103                          std::string elem_sep = ", ")
104     {
105         std::ostringstream ss;
106         ss << m << "\n";
107         return ss.str();
108     }
109
110     void decomp_function(const Eigen::Ref<const complex_matrix>& matrix, int
numberofbits)
111     {
112         DOUT("decomp_function: \n" << to_string(matrix));

```



```

113     if(numberofbits == 1)
114     {
115         zyz_decomp(matrix);
116     }
117     else
118     {
119         int n = matrix.rows()/2;
120
121         complex_matrix V(n,n);
122         complex_matrix W(n,n);
123         Eigen::VectorXcd D(n);
124         // if q2 is zero, the whole thing is a demultiplexing problem
instead of full CSD
125         if(matrix.bottomLeftCorner(n,n).isZero(10e-14) && matrix.
topRightCorner(n,n).isZero(10e-14))
126         {
127             DOUT("Optimization: q2 is zero, only demultiplexing will be
performed.");
128             instructionlist.push_back(200.0);
129             if(matrix.topLeftCorner(n, n).isApprox(matrix.bottomRightCorner
(n,n),10e-4))
130             {
131                 DOUT("Optimization: Unitaries are equal, skip one step in
the recursion for unitaries of size: " << n << " They are both: " << matrix
.topLeftCorner(n, n));
132                 instructionlist.push_back(300.0);
133                 decomp_function(matrix.topLeftCorner(n, n), numberOfbits-1)
;
134             }
135             else
136             {
137                 demultiplexing(matrix.topLeftCorner(n, n), matrix.
bottomRightCorner(n,n), V, D, W, numberOfbits-1);
138
139                 decomp_function(W, numberOfbits-1);
140                 multicontrolledZ(D, D.rows());
141                 decomp_function(V, numberOfbits-1);
142             }
143         }
144         // Check to see if it the kronecker product of a bigger matrix and
the identity matrix.
145         // By checking if the first row is equal to the second row one over
, and if the last two rows are equal
146         // Which means the last qubit is not affected by this gate
147         else if (matrix(Eigen::seqN(0, n, 2), Eigen::seqN(1, n, 2)).isZero
() && matrix(Eigen::seqN(1, n, 2), Eigen::seqN(0, n, 2)).isZero() &&
matrix.block(0,0,1,2*n-1) == matrix.block(1,1,1,2*n-1) && matrix.block(2*n
-2,0,1,2*n-1) == matrix.block(2*n-1,1,1,2*n-1))
148         {
149             DOUT("Optimization: last qubit is not affected, skip one step
in the recursion.");
150             // Code for last qubit not affected
151             instructionlist.push_back(100.0);
152             decomp_function(matrix(Eigen::seqN(0, n, 2), Eigen::seqN(0, n,
2)), numberOfbits-1);
153
154         }
155         else
156         {
157             complex_matrix ss(n,n);
158             complex_matrix L0(n,n);

```

```

159     complex_matrix L1(n,n);
160     complex_matrix R0(n,n);
161     complex_matrix R1(n,n);
162     CSD(matrix, L0, L1, R0, R1, ss);
163     demultiplexing(R0, R1, V, D, W, numberofbits-1);
164     decomp_function(W, numberofbits-1);
165     multicontrolledZ(D, D.rows());
166     decomp_function(V, numberofbits-1);
167
168     multicontrolledY(ss.diagonal(), n);
169
170     demultiplexing(L0, L1, V, D, W, numberofbits-1);
171     decomp_function(W, numberofbits-1);
172     multicontrolledZ(D, D.rows());
173     decomp_function(V, numberofbits-1);
174     }
175 }
176
177
178 void CSD(const Eigen::Ref<const complex_matrix>& U, Eigen::Ref<
complex_matrix> u1, Eigen::Ref<complex_matrix> u2, Eigen::Ref<
complex_matrix> v1, Eigen::Ref<complex_matrix> v2, Eigen::Ref<
complex_matrix> s)
179 {
180     //Cosine sine decomposition
181     // U = [q1, U01] = [u1   ][c  s][v1  ]
182     //      [q2, U11] = [   u2][-s c][  v2]
183
184     int n = U.rows();
185     Eigen::BDCSVD<complex_matrix> svd(n/2,n/2);
186     svd.compute(U.topLeftCorner(n/2,n/2), Eigen::ComputeThinU | Eigen::
ComputeThinV); // possible because it's square anyway
187
188
189     // thinCSD: q1 = u1*c*v1.adjoint()
190     //             q2 = u2*s*v1.adjoint()
191     int p = n/2;
192     complex_matrix c(svd.singularValues().reverse().asDiagonal());
193     u1.noalias() = svd.matrixU().rowwise().reverse();
194     v1.noalias() = svd.matrixV().rowwise().reverse(); // Same v as in
matlab: u*s*v.adjoint() = q1
195
196     complex_matrix q2 = U.bottomLeftCorner(p,p)*v1;
197
198     int k = 0;
199     for(int j = 1; j < p; j++)
200     {
201         if(c(j,j).real() <= 0.70710678119)
202         {
203             k = j;
204         }
205     }
206
207     Eigen::HouseholderQR<complex_matrix> qr(p,k+1);
208     qr.compute(q2.block( 0,0, p, k+1));
209     u2 = qr.householderQ();
210     s.noalias() = u2.adjoint()*q2;
211     if(k < p-1)
212     {
213         DOUT("k is smaller than size of q1 = "<< p << ", adjustments will
be made, k = " << k);

```

```

214         k = k+1;
215         Eigen::BDCSVD<complex_matrix> svd2(p-k, p-k);
216         svd2.compute(s.block(k, k, p-k, p-k), Eigen::ComputeThinU | Eigen::
ComputeThinV);
217         s.block(k, k, p-k, p-k) = svd2.singularValues().asDiagonal();
218         c.block(0,k, p,p-k) = c.block(0,k, p,p-k)*svd2.matrixV();
219         u2.block(0,k, p,p-k) = u2.block(0,k, p,p-k)*svd2.matrixU();
220         v1.block(0,k, p,p-k) = v1.block(0,k, p,p-k)*svd2.matrixV();
221
222         Eigen::HouseholderQR<complex_matrix> qr2(p-k, p-k);
223
224         qr2.compute(c.block(k,k, p-k,p-k));
225         c.block(k,k,p-k,p-k) = qr2.matrixQR().triangularView<Eigen::Upper
>();
226         u1.block(0,k, p,p-k) = u1.block(0,k, p,p-k)*qr2.householderQ();
227     }
228
229     std::vector<int> c_ind;
230     std::vector<int> s_ind;
231     for(int j = 0; j < p; j++)
232     {
233         if(c(j,j).real() < 0)
234         {
235             c_ind.push_back(j);
236         }
237         if(s(j,j).real() < 0)
238         {
239             s_ind.push_back(j);
240         }
241     }
242
243     c(c_ind,c_ind) = -c(c_ind,c_ind);
244     u1(Eigen::all, c_ind) = -u1(Eigen::all, c_ind);
245     s(s_ind,s_ind) = -s(s_ind,s_ind);
246     u2(Eigen::all, s_ind) = -u2(Eigen::all, s_ind);
247
248     if(!U.topLeftCorner(p,p).isApprox(u1*c*v1.adjoint(), 10e-8) || !U.
bottomLeftCorner(p,p).isApprox(u2*s*v1.adjoint(), 10e-8))
249     {
250         if(U.topLeftCorner(p,p).isApprox(u1*c*v1.adjoint(), 10e-8))
251         {
252             DOUT("q1 is correct");
253         }
254         else
255         {
256             DOUT("q1 is not correct! (is not usually an issue)");
257             DOUT("q1: \n" << U.topLeftCorner(p,p));
258             DOUT("reconstructed q1: \n" << u1*c*v1.adjoint());
259         }
260     }
261     if(U.bottomLeftCorner(p,p).isApprox(u2*s*v1.adjoint(), 10e-8))
262     {
263         DOUT("q2 is correct");
264     }
265     else
266     {
267         DOUT("q2 is not correct! (is not usually an issue)");
268         DOUT("q2: " << U.bottomLeftCorner(p,p));
269         DOUT("reconstructed q2: " << u2*s*v1.adjoint());
270     }
271 }

```

```

272     v1.adjointInPlace(); // Use this instead of = v1.adjoint (to avoid
aliasing issues)
273     s = -s;
274
275     complex_matrix tmp_s = u1.adjoint()*U.topRightCorner(p,p);
276     complex_matrix tmp_c = u2.adjoint()*U.bottomRightCorner(p,p);
277     for(int i = 0; i < p; i++)
278     {
279         if(std::abs(s(i,i)) > std::abs(c(i,i)))
280         {
281             v2.row(i).noalias() = tmp_s.row(i)/s(i,i);
282         }
283         else
284         {
285             v2.row(i).noalias() = tmp_c.row(i)/c(i,i);
286         }
287     }
288
289     // U = [q1, U01] = [u1   ][c  s][v1  ]
290     //      [q2, U11] = [   u2][-s c][   v2]
291
292     complex_matrix tmp(n,n);
293     tmp.topLeftCorner(p,p) = u1*c*v1;
294     tmp.bottomLeftCorner(p,p) = -u2*s*v1;
295     tmp.topRightCorner(p,p) = u1*s*v2;
296     tmp.bottomRightCorner(p,p) = u2*c*v2;
297     // Just to see if it kinda matches
298     if(!tmp.isApprox(U, 10e-2))
299     {
300         throw ql::exception("CSD of unitary '"+ name+"' is wrong! Failed at
matrix: \n"+to_string(tmp) + "\nwhich should be: \n" + to_string(U), false
);
301     }
302 }
303
304
305 void zyz_decomp(const Eigen::Ref<const complex_matrix>& matrix)
306 {
307     ql::complex_t det = matrix.determinant(); // matrix(0,0)*matrix(1,1)-
matrix(1,0)*matrix(0,1);
308
309     double delta = atan2(det.imag(), det.real())/matrix.rows();
310     std::complex<double> A = exp(std::complex<double>(0,-1)*delta)*matrix
(0,0);
311     std::complex<double> B = exp(std::complex<double>(0,-1)*delta)*matrix
(0,1); //to comply with the other y-gate definition
312
313     double sw = sqrt(pow((double) B.imag(),2) + pow((double) B.real(),2) +
pow((double) A.imag(),2));
314     double wx = 0;
315     double wy = 0;
316     double wz = 0;
317
318     if(sw > 0)
319     {
320         wx = B.imag()/sw;
321         wy = B.real()/sw;
322         wz = A.imag()/sw;
323     }
324
325     double t1 = atan2(A.imag(),A.real());

```

```

326     double t2 = atan2(B.imag(), B.real());
327     alpha = t1+t2;
328     gamma = t1-t2;
329     beta = 2*atan2(sw*sqrt(pow((double) wx,2)+pow((double) wy,2)),sqrt(pow
((double) A.real(),2)+pow((wz*sw),2)));
330     instructionlist.push_back(-gamma);
331     instructionlist.push_back(-beta);
332     instructionlist.push_back(-alpha);
333 }
334
335 void demultiplexing(const Eigen::Ref<const complex_matrix> &U1, const Eigen
::Ref<const complex_matrix> &U2, Eigen::Ref<complex_matrix> V, Eigen::Ref
<Eigen::VectorXcd> D, Eigen::Ref<complex_matrix> W, int
numberofcontrolbits)
336 {
337     // [U1 0 ] = [V 0][D 0 ][W 0]
338     // [0 U2]   [0 V][0 D*][0 W]
339     complex_matrix check = U1*U2.adjoint();
340     if(check == check.adjoint())
341     {
342         COU<("Demultiplexing matrix is self-adjoint()");
343         Eigen::SelfAdjointEigenSolver<Eigen::MatrixXcd> eigslv(check);
344         D.noalias() = ((complex_matrix) eigslv.eigenvalues()).cwiseSqrt();
345         COU<("D"<< D);
346         V.noalias() = eigslv.eigenvectors();
347         COU<("V"<< V);
348         W.noalias() = D.asDiagonal()*V.adjoint()*U2;
349     }
350     else
351     {
352         if (numberofcontrolbits < 5) //schur is faster for small matrices
353         {
354             Eigen::ComplexSchur<complex_matrix> decomposition(check);
355             D.noalias() = decomposition.matrixT().diagonal().cwiseSqrt();
356             V.noalias() = decomposition.matrixU();
357             W.noalias() = D.asDiagonal() * V.adjoint() * U2;
358         }
359         else
360         {
361             Eigen::ComplexEigenSolver<complex_matrix> decomposition(check);
362             D.noalias() = decomposition.eigenvalues().cwiseSqrt();
363             V.noalias() = decomposition.eigenvectors();
364             W.noalias() = D.asDiagonal() * V.adjoint() * U2;
365         }
366     }
367     if(!(V*V.adjoint()).isApprox(Eigen::MatrixXd::Identity(V.rows(), V.rows
()), 10e-3))
368     {
369         DOUT("Eigenvalue decomposition incorrect: V is not unitary,
adjustments will be made");
370         Eigen::BDCSVD<complex_matrix> svd3(V.block(0,0,V.rows(),2), Eigen::
ComputeFullU);
371         V.block(0,0,V.rows(),2) = svd3.matrixU();
372         svd3.compute(V(Eigen::all,Eigen::seq(Eigen::last-1,Eigen::last)),
Eigen::ComputeFullU);
373         V(Eigen::all,Eigen::seq(Eigen::last-1,Eigen::last)) = svd3.matrixU
());
374     }
375     complex_matrix Dtemp = D.asDiagonal();
376     if(!U1.isApprox(V*Dtemp*W, 10e-2) || !U2.isApprox(V*Dtemp.adjoint()*W,

```

```

10e-2))
378     {
379         EOUT("Demultiplexing not correct!");
380         throw ql::exception("Demultiplexing of unitary '"+ name+"' not
correct! Failed at matrix U1: \n"+to_string(U1)+ "and matrix U2: \n" +
to_string(U2) + "\nwhile they are: \n" + to_string(V*D.asDiagonal()*W) + "\
nand \n" + to_string(V*D.conjugate().asDiagonal()*W), false);
381     }
382 }
383
384
385 std::vector<Eigen::MatrixXd> genMk_lookupable;
386 // returns M^k = (-1)^(b_(i-1)*g_(i-1)), where * is bitwise inner product,
g = binary gray code, b = binary code.
387 void genMk()
388 {
389     int numberqubits = uint64_log2(_matrix.rows());
390     for(int n = 1; n <= numberqubits; n++)
391     {
392         int size=1<<n;
393         Eigen::MatrixXd Mk(size,size);
394         for(int i = 0; i < size; i++)
395         {
396             for(int j = 0; j < size ;j++)
397             {
398                 Mk(i,j) =std::pow(-1, bitParity(i&(j^(j>>1))));
399             }
400         }
401         genMk_lookupable.push_back(Mk);
402     }
403     // return genMk_lookupable[numberqubits-1];
404 }
405
406 // source: https://stackoverflow.com/questions/994593/how-to-do-an-integer-log2-in-c user Todd Lehman
407 int uint64_log2(uint64_t n)
408 {
409     #define S(k) if (n >= (UINT64_C(1) << k)) { i += k; n >>= k; }
410     int i = -(n == 0); S(32); S(16); S(8); S(4); S(2); S(1); return i;
411     #undef S
412 }
413
414 int bitParity(int i)
415 {
416     if (i < 2 << 16)
417     {
418         i = (i >> 16) ^ i;
419         i = (i >> 8) ^ i;
420         i = (i >> 4) ^ i;
421         i = (i >> 2) ^ i;
422         i = (i >> 1) ^ i;
423         return i % 2;
424     }
425     else
426     {
427         throw ql::exception("Bit parity number too big!", false);
428     }
429 }
430
431 void multicontrolledY(const Eigen::Ref<const Eigen::VectorXcd> &ss, int
halfthesizeofthematrix)

```

```

432     {
433         Eigen::VectorXd temp = 2*Eigen::asin(ss.array()).real();
434         Eigen::CompleteOrthogonalDecomposition<Eigen::MatrixXd> dec(
genMk_lookuptable[uint64_log2(halfthesizeofthematrix)-1]);
435         Eigen::VectorXd tr = dec.solve(temp);
436         // Check is very approximate to account for low-precision input
matrices
437         if(!temp.isApprox(genMk_lookuptable[uint64_log2(halfthesizeofthematrix)
-1]*tr, 10e-2))
438             {
439                 EOUT("Multicontrolled Y not correct!");
440                 throw ql::exception("Demultiplexing of unitary '"+ name+"' not
correct! Failed at demultiplexing of matrix ss: \n" + to_string(ss), false
);
441             }
442
443         instructionlist.insert(instructionlist.end(), &tr[0], &tr[
halfthesizeofthematrix]);
444     }
445
446 void multicontrolledZ(const Eigen::Ref<const Eigen::VectorXcd> &D, int
halfthesizeofthematrix)
447 {
448     Eigen::VectorXd temp = (std::complex<double>(0,-2)*Eigen::log(D.array
())).real();
449     Eigen::CompleteOrthogonalDecomposition<Eigen::MatrixXd> dec(
genMk_lookuptable[uint64_log2(halfthesizeofthematrix)-1]);
450     Eigen::VectorXd tr = dec.solve(temp);
451
452     // Check is very approximate to account for low-precision input
matrices
453     if(!temp.isApprox(genMk_lookuptable[uint64_log2(halfthesizeofthematrix)
-1]*tr, 10e-2))
454         {
455             EOUT("Multicontrolled Z not correct!");
456             throw ql::exception("Demultiplexing of unitary '"+ name+"' not
correct! Failed at demultiplexing of matrix D: \n"+ to_string(D), false);
457         }
458     instructionlist.insert(instructionlist.end(), &tr[0], &tr[
halfthesizeofthematrix]);
459 }
460
461 ~unitary()
462 {
463     // destroy unitary
464     DOUT("destructing unitary: " << name);
465 }
466 };
467 }
468 #endif // _UNITARY_H

```





# B

## Code from A.Sarkar

This is code from Aritra Sarkar. The MatLab implementation of Quantum Associative Memory is taken from his Master's thesis [38], and the `gen_unitary` was written specifically to generate dense unitary matrices. Both of these small programs have just been used to test and verify the functionality of the Unitary Decomposition.

Listing B.1: Quantum Associative Memory [38]

```
1 % Reference : Quantum associative memory with improved distributed
  queries - J.P.T. Njafa, S.G.N. Engo, P. Woafu
2 % Reference : Quantum algorithms for pattern matching in genomic
  sequences - A. Sarkar
3 % \author: Aritra Sarkar (-princeph0enlx)
4 % \project: -Quantumaccelerated -Genomesequencing
5 % \repo: https: //gitlab.com/-princeph0enlx/QaGs
6
7 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
8
9 function mtlb_qam_a4_idq()
10     close all
11     clear all
12     clc
13
14     AS = {'A','C','G','T'}; %alphabet set {0,1,2,3} := {ACGT} for Dna
  Nucleotide bases
15     A = size{AS,2};         %Alphabet size
16     Qa = ceil(log2(A));    %Number of qubits to encode a character of the
  alphabet
17
18     R = 'TTGTCTAGGCGACCA';
19     N = size{R,2};        %reference genome size
20     M = 2;                %Short read size
21     P = 'AA';            %Search pattern (always a series of A, due to
  minimal Hamming distance as the query center)
22     Pb = '0000';         %Binary encoding for P
23     Qd = Qa*M;           %Number of qubits to encode the quantum genomic
  database
24     SS = 2^Qd;           %State space
25
26     qbodq = 0.25         %q for the Binomial distribution for
  distributed query
27     bp = ones(1,SS)
```

```

28     for i = 1:SS
29         hd = sum(sprintf('%s', dec2bin(i-1,Qd)) ~=Pb);
30         bp(i) = sqrt((qbodq^(hd)) * ((1-qbodq)^(Qd-hd)));
31     end
32     plot([0:SS-1],bp,'v-.b')
33     hold on
34     BO = eye(SS) - 2*bp'*bp;
35     BOD = QSD_opq1(BO,3,[0:Qd-1]); %Arg2: 1 - no qas, np AP; 2 - qas,,
AP, 3+ - qasm, no AP
36     maxerrabs = max(max(abs(BOD)-abs(BO))) %check decomposition error

```

Listing B.2: Program to generate dense and complex unitary matrices used to unit test and performance test the decomposition algorithm.

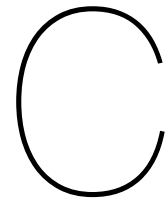
```

1 # File: gen_unitary.py
2 # Author: A. Sarkar
3 import numpy as np
4 from scipy.linalg import expm
5 import sys
6 np.set_printoptions(threshold=sys.maxsize)
7 np.set_printoptions(linewidth=1000)
8 def check_hermitian(A):
9     adjoint = A.conj().T # a.k.a. conjugate-transpose, transjugate,
dagger
10    assert(np.allclose(A,adjoint))
11
12 def gen_wsopp(n_qubits = 1):
13    H = np.zeros([2**n_qubits,2**n_qubits])
14    I = np.array([[1,0],[0,1]])
15    X = np.array([[0,1],[1,0]])
16    Y = np.array([[0,complex(0,-1)],[complex(0,1),0]])
17    Z = np.array([[1,0],[0,-1]])
18
19    for i in range(4**n_qubits):
20        pt = format(i,"0"+str(2*n_qubits)+"b")
21        sopp = [1]
22        for j in range(0,len(pt),2):
23            k = pt[j:j+2]
24            if k == '00':
25                sopp = np.kron(sopp,I)
26            elif k == '01':
27                sopp = np.kron(sopp,X)
28            elif k == '10':
29                sopp = np.kron(sopp,Y)
30            else:
31                sopp = np.kron(sopp,Z)
32        w = np.random.uniform(0,1)
33        H = H + w*sopp
34    check_hermitian(H)
35    return H
36
37 def check_unitary(U):
38    adjoint = U.conj().T # a.k.a. conjugate-transpose, transjugate,
dagger
39    assert(np.allclose(U.dot(adjoint),adjoint.dot(U)))
40    assert(np.allclose(U.dot(adjoint),np.eye(U.shape[0])))
41    return

```

```
42
43 def gen_unitary(n_qubit = 1):
44     H = gen_wsopp(n_qubit)
45     U = expm(complex(0, -1) * H)
46     check_unitary(U)
47     return U
48
49 def gen_unitary_array(n_qubit = 1):
50     H = gen_wsopp(n_qubit)
51     U = expm(complex(0, -1) * H)
52     check_unitary(U)
53     I = np.array([[1, 0], [0, 1]])
54     #print(np.kron(I, U))
55     return U.flatten()
```





## Compilation examples

Listing C.1: This c++ program adds all integers lower than "a" to the result.

```
1 int main()
2 {
3     int result;
4     int a = 5;
5     for(int i = 0; i < a; i++)
6     {
7         result = result+i;
8     }
9     return result;
10 }
```

This compiles into:

```
1      li      $2,5           # a = 5
2      sw      $2,16($fp)     # int a = 5
3      sw      $0,12($fp)     # int i = 0;
4 $L3:
5      lw      $3,12($fp)     # load i
6      lw      $2,16($fp)     # load a
7
8      slt     $2,$3,$2       # i < a
9      beq     $2,$0,$L2     # go to end of for-loop if i<a==false
10
11     lw      $3,8($fp)      # load result
12     lw      $2,12($fp)    # load i
13
14     addu    $2,$3,$2       # result + 1
15     sw      $2,8($fp)     # result = result + i
16
17     lw      $2,12($fp)    # load i
18     addiu   $2,$2,1       # i+1
19     sw      $2,12($fp)    # store i
20     b       $L3           # go to start of for-loop
21 $L2:
22     lw      $2,8($fp)     # return result;
```

Listing C.2: Register allocation for the highest level of the program from example C.1

```
1 int main() {}
2     int result;
```

```

3   int a = 5;
4   ...    // This is the for-loop
5   return result;}

```

⇓

```

1 Program(
2   statementList([
3     varDef("result"),
4     varDefAssignment("a",5),
5     ...
6   ],
7   returnStatement(
8     varRef("result")
9   )
10  )

```

⇓

```

1   store word    $0, 8($fp)
2   load immediate $2, 5
3   store word    $2, 16($fp)
4   ...
5   load word     $2, 8($fp)

```

Listing C.3: Register allocation for the for loop of the program from example C.1

```

1 for(int i = 0; i < a; i++){
2   result = result+i;
3 }

```

⇓

```

1 forStatement(
2   varDefAssignment("i",0),
3   lessThan("i", "a"),
4   increase("i"),
5   statementList([
6     varAssignment("result",
7       Add(varRef("result"), varRef("i"))
8     )
9   ])
10 )

```

⇓

Listing C.4: Transformation of a for-loop

```

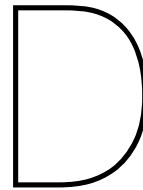
1 # store i (in memory location 12($fp)
2   store word    $0, 12($fp)
3
4 #label $L3 (start of for-loop)
5 $L3:
6 # load i into register 3 (stored in memory location 12($fp)
7   load word     $3, 12($fp)
8 # load a into register 2 (stored in memory location 16($fp)
9   load word     $2, 16($fp)
10 # the result of "set less than" is stored in register 2, 1 if i < a and
    0 if not.
11   set less than $2, $3, $2
12 # branch to label $L2 if the result in register 2 is the same as in
    register 0 (which is the register which always has the value 0)

```

```
13  branch if equal    $2, $0, $L2
14
15 #load result into register 3 (stored in memory location 8($fp)
16  load word         $3, 8($fp)
17 # load i into register 2 (stored in memory location 12($fp)
18  load word         $2, 12($fp)
19 # add the values in registers 2 and 3 together and store in register 2
20  add               $2,$3,$2
21 #store the value from register 2 to the memory location of "result"
22  store word        $2, 8($fp)
23
24 #load i into register 2 (stored in memory location 16($fp)
25  load word         $2, 12($fp)
26 #add 1 to the value in register 2
27  add immediate     $2, $2,1
28 #store the value from register 2 to the memory location of i
29  store word        $2, 12($fp)
30 #branch to label $L3
31  branch            $L3
32
33 #label $L2 (end of for-loop)
34 $L2:
```







## Tests

Listing D.1: High level Matlab algorithm for Quantum Associative Memory [38]

```
1 function mtlb_qam_a4_idq()
2     close all
3     clear all
4     clc
5
6     AS = {'A','C','G','T'}; %alphabet set {0,1,2,3} := {ACGT} for Dna
    Nucleotide bases
7     A = size(AS,2); %Alphabet size
8     Qa = ceil(log2(A)); %Number of qubits to encode a character of the
    alphabet
9
10    R = 'TTGTCTAGGCGACCA';
11    N = size(R,2); %reference genome size
12    M = 2; %Short read size
13    P = 'AA'; %Search pattern (always a series of A, due to
    minimal Hamming distance as the query center)
14    Pb = '0000'; %Binary encoding for P
15    Qd = Qa*M; %Number of qubits to encode the quantum genomic
    database
16    SS = 2^Qd; %State space
17
18    qbodq = 0.25 %q for the Binomial distribution for
    distributed query
19    bp = ones(1,SS)
20    for i = 1:SS)
21        hd = sum(spintf('%s', dec2bin(i-aQd)) ~=Pb);
22        bp(i) = sqrt((qbodq^(hd)) * ((1-qbodq)^(Qd-hd)));
23    end
24    plot([0:SS-1],bp,'v-.b')
25    hold on
26    BO = eye(SS) - 2*bp'*bp;
27    BOD = QSD_opq1(BO,3,[0:Qd-1]); %Arg2: 1 - no qas, np AP; 2 - qas,,
    AP, 3+ - qasm, no AP
28    maxerrabs = max(max(abs(BOD)-abs(BO))) %check decomposition error
```

Listing D.2: The QSD\_opq1 function from Listing D.1

```
1 function Ud = QSD_opq1(U,prnt,qbsp)
2     %% Decompose U to AB1, CS, AB2
```

```

3
4 dim = log2(size(U,1));
5 splitpt = 2^(dim-1);
6 OU = zeros(splitpt, splitpt);
7 I = eye(2);
8 [L0, L1, cc, ss, R0, R1] = fatCSD(U);
9
10 %~~~~~ STEP 1 ~~~~~
11 %% Decompose AB2 to V, D, W (lower dimension)
12 AB2 = [R0 zeroes(size(R0, 1), size(R1,2)); zeroes(size(R1,1),size(
R0,2) R1)];
13 U1 = AB2(1:splitpt, 1:splitpt);
14 U2 = AB2(splitpt+1:end, splitpt+1:end);
15
16 %etc.

```

Listing D.3: Extraction function for the total probability from the QX output

```

1 def helper_regex(measurementstring):
2     regex = re.findall('[0-9.]+' , measurementstring)
3     i = 0
4     array = []
5     while i < len(regex):
6         array.append(float(regex[i])**2+float(regex[i+1])**2)
7         i +=3 # so we skip every third occurrence, which is the bit
string representing the qubit combination
8     return array

```

Listing D.4: The calculation of the total probability of a state from a matrix entry.

```

1 def helper_prob(qubitstate):
2     return qubitstate.real**2+qubitstate.imag**2

```

Listing D.5: Test of the basic functionality of the unitary decomposition

```

1 def test_unitary_basic(self):
2     config_fn = os.path.join(curdir, 'test_cfg_none_simple.json')
3     platform = ql.Platform('platform_none', config_fn)
4     num_qubits = 3
5     p = ql.Program('test_unitary_pass', platform, num_qubits)
6     k = ql.Kernel('akernel', platform, num_qubits)
7
8     u = ql.Unitary('u1', [ complex(1.0, 0.0), complex(0.0, 0.0),
9                             complex(0.0, 0.0), complex(0.0, 1.0)])
10    u.decompose()
11
12    k.gate("i", [0])
13    k.gate("s", [0])
14    k.gate(u, [2])
15
16    p.add_kernel(k)
17    p.compile()

```

Listing D.6: Test for raising an error when a non-decomposed unitary is added to the kernel

```

1 def test_unitary_undecomposed(self):
2     config_fn = os.path.join(curdir, 'test_cfg_none_simple.json')
3     platform = ql.Platform('platform_none', config_fn)
4     num_qubits = 3

```

```

5 p = ql.Program('test_unitary_pass', platform, num_qubits)
6 k = ql.Kernel('akernel', platform, num_qubits)
7
8 u = ql.Unitary('u1', [ complex(1.0, 0.0), complex(0.0, 0.0),
9                        complex(0.0, 0.0), complex(0.0, 1.0)])
10 k.gate("s", [0])
11
12 # adding un-decomposed u to kernel should raise error
13 with self.assertRaises(Exception) as cm:
14     k.gate(u, [2])
15
16 self.assertEqual(str(cm.exception), 'Unitary \'u1\' not decomposed.
    Cannot be added to kernel!')

```

Listing D.7: Test for raising an error when a unitary is applied to too many qubits

```

1 def test_unitary_wrongnumberofqubits(self):
2     config_fn = os.path.join(curdir, 'test_cfg_none_simple.json')
3     platform = ql.Platform('platform_none', config_fn)
4     num_qubits = 3
5     p = ql.Program('test_unitary_pass', platform, num_qubits)
6     k = ql.Kernel('akernel', platform, num_qubits)
7
8     u = ql.Unitary('u1', [ complex(1.0, 0.0), complex(0.0, 0.0),
9                            complex(0.0, 0.0), complex(0.0, 1.0)])
10    k.gate("s", [0])
11
12    # adding un-decomposed u to kernel should raise error
13    with self.assertRaises(Exception) as cm:
14        k.gate(u, [1,2])
15
16    self.assertEqual(str(cm.exception), 'Unitary \'u1\' has been
    applied to the wrong number of qubits. Cannot be added to kernel! 2
    and not 1.000000')

```

Listing D.8: Test for raising an error when a unitary is applied to too few qubits

```

1 def test_unitary_wrongnumberofqubits_toofew(self):
2     config_fn = os.path.join(curdir, 'test_cfg_none_simple.json')
3     platform = ql.Platform('platform_none', config_fn)
4     num_qubits = 3
5     p = ql.Program('test_unitary_pass', platform, num_qubits)
6     k = ql.Kernel('akernel', platform, num_qubits)
7
8
9     u = ql.Unitary('u1', [-0.43874989-0.10659111j,
10    -0.47325212+0.12917344j, -0.58227163+0.20750072j,
11    -0.29075334+0.29807585j,
12    0.30168601-0.22307459j, 0.32626 +0.4534935j ,
13    -0.20523265-0.42403593j, -0.01012565+0.5701683j ,
14    -0.40954341-0.49946371j, 0.28560698-0.06740801j,
15    0.52146754+0.1833513j , -0.37248653+0.22891636j,
16    0.03113162-0.48703302j, -0.57180014+0.18486244j,
    0.2943625 -0.06148912j, 0.55533888+0.04322811j])
13    # adding un-decomposed u to kernel should raise error
14    with self.assertRaises(Exception) as cm:
15        k.gate(u, [0])
16

```

```

17 self.assertEqual(str(cm.exception), 'Unitary \'u1\' has been
    applied to the wrong number of qubits. Cannot be added to kernel! 1
    and not 2.000000')

```

Listing D.9: Test with file comparison of decomposition of an identity gate

```

1 def test_unitary_decompose_I(self):
2     config_fn = os.path.join(curdir, 'test_cfg_none_simple.json')
3     platform = ql.Platform('platform_none', config_fn)
4     num_qubits = 1
5     p = ql.Program('test_unitary_I', platform, num_qubits)
6     k = ql.Kernel('akernel', platform, num_qubits)
7
8     u = ql.Unitary('u1', [ complex(1.0, 0.0), complex(0.0, 0.0),
9                             complex(0.0, 0.0), complex(1.0, 0.0)])
10    u.decompose()
11    k.gate(u, [0])
12
13    p.add_kernel(k)
14    p.compile()
15
16    gold_fn = rootDir + '/golden/test_unitary-decomp_1qubit_I.qasm'
17    qasm_fn = os.path.join(output_dir, p.name+'.qasm')
18    self.assertTrue( file_compare(qasm_fn, gold_fn) )

```

Listing D.10: Test with file comparison of decomposition of a pauli-X gate

```

1 def test_unitary_decompose_X(self):
2     config_fn = os.path.join(curdir, 'test_cfg_none_simple.json')
3     platform = ql.Platform('platform_none', config_fn)
4     num_qubits = 1
5     p = ql.Program('test_unitary_X', platform, num_qubits)
6     k = ql.Kernel('akernel', platform, num_qubits)
7
8     u = ql.Unitary('u1', [ complex(0.0, 0.0), complex(1.0, 0.0),
9                             complex(1.0, 0.0), complex(0.0, 0.0)])
10    u.decompose()
11    k.gate(u, [0])
12
13    p.add_kernel(k)
14    p.compile()
15
16    gold_fn = rootDir + '/golden/test_unitary-decomp_1qubit_X.qasm'
17    qasm_fn = os.path.join(output_dir, p.name+'.qasm')
18    self.assertTrue( file_compare(qasm_fn, gold_fn) )

```

Listing D.11: Test with file comparison of decomposition of a pauli-Y gate

```

1 def test_unitary_decompose_Y(self):
2     config_fn = os.path.join(curdir, 'test_cfg_none_simple.json')
3     platform = ql.Platform('platform_none', config_fn)
4     num_qubits = 1
5     p = ql.Program('test_unitary_Y', platform, num_qubits)
6     k = ql.Kernel('akernel', platform, num_qubits)
7
8     u = ql.Unitary('u1', [ complex(0.0, 0.0), complex(0.0, -1.0),
9                             complex(0.0, 1.0), complex(0.0, 0.0)])
10    u.decompose()

```

```

11 k.gate(u, [0])
12
13 p.add_kernel(k)
14 p.compile()
15
16 gold_fn = rootDir + '/golden/test_unitary-decomp_1qubit_Y.qasm'
17 qasm_fn = os.path.join(output_dir, p.name+'.qasm')
18 self.assertTrue( file_compare(qasm_fn, gold_fn) )

```

Listing D.12: Test with file comparison of decomposition of a pauli-Z gate

```

1 def test_unitary_decompose_Z(self):
2     config_fn = os.path.join(curdir, 'test_cfg_none_simple.json')
3     platform = ql.Platform('platform_none', config_fn)
4     num_qubits = 1
5     p = ql.Program('test_unitary_Z', platform, num_qubits)
6     k = ql.Kernel('akernel', platform, num_qubits)
7
8     u = ql.Unitary('u1', [ complex(1.0, 0.0), complex(0.0, 0.0),
9                             complex(0.0, 0.0), complex(-1.0, 0.0)])
10    u.decompose()
11    k.gate(u, [0])
12
13    p.add_kernel(k)
14    p.compile()
15
16    gold_fn = rootDir + '/golden/test_unitary-decomp_1qubit_Z.qasm'
17    qasm_fn = os.path.join(output_dir, p.name+'.qasm')
18    self.assertTrue( file_compare(qasm_fn, gold_fn) )

```

Listing D.13: Test with file comparison of decomposition of three gates

```

1 def test_unitary_decompose_IYZ(self):
2     config_fn = os.path.join(curdir, 'test_cfg_none_simple.json')
3     platform = ql.Platform('platform_none', config_fn)
4     num_qubits = 1
5     p = ql.Program('test_unitary_IYZ', platform, num_qubits)
6     k = ql.Kernel('akernel', platform, num_qubits)
7
8     u = ql.Unitary('X', [ complex(0.0, 0.0), complex(1.0, 0.0),
9                             complex(1.0, 0.0), complex(0.0, 0.0)])
9     u.decompose()
10    k.gate(u, [0])
11    u = ql.Unitary('y', [ complex(0.0, 0.0), complex(0.0, -1.0),
12                             complex(0.0, 1.0), complex(0.0, 0.0)])
13    u.decompose()
14    k.gate(u, [0])
15    u = ql.Unitary('Z', [ complex(1.0, 0.0), complex(0.0, 0.0),
16                             complex(0.0, 0.0), complex(-1.0, 0.0)])
17    u.decompose()
18    k.gate(u, [0])
19
20    p.add_kernel(k)
21    p.compile()
22
23    gold_fn = rootDir + '/golden/test_unitary-decomp_1qubit_IYZ.
qasm'
24    qasm_fn = os.path.join(output_dir, p.name+'.qasm')

```

```
23 self.assertTrue( file_compare(qasm_fn, gold_fn) )
```

Listing D.14: Test with file comparison of decomposition of three gates in a different order

```
1 def test_unitary_decompose_IYZ_differentorder(self):
2     config_fn = os.path.join(curdir, 'test_cfg_none_simple.json')
3     platform = ql.Platform('platform_none', config_fn)
4     num_qubits = 1
5     p = ql.Program('test_unitary_IYZ', platform, num_qubits)
6     k = ql.Kernel('akernel', platform, num_qubits)
7     u = ql.Unitary('X', [ complex(0.0, 0.0), complex(1.0, 0.0),
8                           complex(1.0, 0.0), complex(0.0, 0.0)])
9     u1 = ql.Unitary('y', [ complex(0.0, 0.0), complex(0.0, -1.0),
10                          complex(0.0, 1.0), complex(0.0, 0.0)])
11     u2 = ql.Unitary('Z', [ complex(1.0, 0.0), complex(0.0, 0.0),
12                          complex(0.0, 0.0), complex(-1.0, 0.0)])
13     u.decompose()
14     u1.decompose()
15     u2.decompose()
16     k.gate(u, [0])
17     k.gate(u1, [0])
18     k.gate(u2, [0])
19
20     p.add_kernel(k)
21     p.compile()
22
23     gold_fn = rootDir + '/golden/test_unitary-decomp_1qubit_IYZ.qasm'
24     qasm_fn = os.path.join(output_dir, p.name+'.qasm')
25     self.assertTrue( file_compare(qasm_fn, gold_fn) )
```

Listing D.15: Test for raising an error when a non unitary matrix is decomposed

```
1 def test_unitary_decompose_nonunitary(self):
2     config_fn = os.path.join(curdir, 'test_cfg_none_simple.json')
3     platform = ql.Platform('platform_none', config_fn)
4     num_qubits = 1
5     p = ql.Program('test_unitary_nonunitary', platform, num_qubits)
6     k = ql.Kernel('akernel', platform, num_qubits)
7     u = ql.Unitary('WRONG', [ complex(0.0, 0.0), complex(0.0, 0.0),
8                              complex(0.0, 0.0), complex(0.0, 0.0)])
9
10     with self.assertRaises(Exception) as cm:
11         u.decompose()
12         k.gate(u, [0])
13         add_kernel(k)
14         p.compile()
15
16     self.assertEqual(str(cm.exception), "Error: Unitary 'WRONG' is not
17 a unitary matrix. Cannot be decomposed!")
```

Listing D.16: Test for raising an error when a matrix is decomposed instead of an array

```
1 def test_unitary_decompose_matrixinsteadofarray(self):
2     config_fn = os.path.join(curdir, 'test_cfg_none_simple.json')
3     platform = ql.Platform('platform_none', config_fn)
4     num_qubits = 1
5     p = ql.Program('test_unitary_wrongtype', platform, num_qubits)
6     k = ql.Kernel('akernel', platform, num_qubits)
```

```

7
8 with self.assertRaises(TypeError) as cm:
9     u = ql.Unitary('TypeError', [[1, 0], [0, 1]])

```

Listing D.17: Test with file comparison of decomposition of an arbitrary angle unitary gate

```

1 def test_non_90_degree_angle(self):
2     config_fn = os.path.join(curdir, 'test_cfg_none_simple.json')
3     platform = ql.Platform('platform_none', config_fn)
4     num_qubits = 2
5     p = ql.Program('test_unitary_non_90_degree_angle', platform,
6 num_qubits)
7     k = ql.Kernel('akernel', platform, num_qubits)
8
9     matrix = [-0.43874989-0.10659111j, -0.47325212+0.12917344j, ...
10 0.30168601-0.22307459j, 0.32626 +0.4534935j, ...
11 -0.40954341-0.49946371j, 0.28560698-0.06740801j, ...
12 0.03113162-0.48703302j, -0.57180014+0.18486244j, ...]
13
14 u1 = ql.Unitary("testname", matrix)
15 u1.decompose()
16 k.gate(u1, [0, 1])
17
18 p.add_kernel(k)
19 p.compile()
20
21 # Tested for correctness using QX
22 gold_fn = rootDir + '/golden/test_unitary-decomp_non_90+
23 degree_angle.qasm'
24 qasm_fn = os.path.join(output_dir, p.name+'.qasm')
25 self.assertTrue(file_compare(qasm_fn, gold_fn))

```

Listing D.18: Test using QX of all the entries in the matrix

```

1 def test_usingqx_00(self):
2     config_fn = os.path.join(curdir, 'test_cfg_none_simple.json')
3     platform = ql.Platform('platform_none', config_fn)
4     num_qubits = 4
5     p = ql.Program('test_usingqx00', platform, num_qubits)
6     k = ql.Kernel('akernel', platform, num_qubits)
7
8     matrix = [-0.15050486+0.32164259j, -0.29086861+0.76699622j, ...
9 -0.65629705+0.20915109j, 0.32782708+0.16363753j, ...
10 -0.38935965-0.47787084j, 0.30279699-0.10056307j, ...
11 0.13874319-0.01460122j, -0.27256915+0.12950497j, ...]
12
13 u1 = ql.Unitary("testname", matrix)
14 u1.decompose()
15 k.gate(u1, [0, 1])
16
17 p.add_kernel(k)
18 p.compile()
19
20 qx.set(os.path.join(output_dir, p.name+'.qasm'))
21 qx.execute()
22 c0 = qx.get_state()
23 self.assertAlmostEqual(helper_prob(matrix[0]), helper_regex(c0)
24 [0], 5)

```

```

24     self.assertAlmostEqual(helper_prob(matrix[4]), helper_regex(c0
25     [1],5)
26     self.assertAlmostEqual(helper_prob(matrix[8]), helper_regex(c0
27     [2],5)
28     self.assertAlmostEqual(helper_prob(matrix[12]), helper_regex(c0
29     [3],5)
30
31 def test_usingqx_01(self):
32     config_fn = os.path.join(curdir, 'test_cfg_none_simple.json')
33     platform = ql.Platform('platform_none', config_fn)
34     num_qubits = 4
35     p = ql.Program('test_usingqx01', platform, num_qubits)
36     k = ql.Kernel('akernel', platform, num_qubits)
37
38     matrix = [-0.15050486+0.32164259j, -0.29086861+0.76699622j, ...
39     -0.65629705+0.20915109j, 0.32782708+0.16363753j, ...
40     -0.38935965-0.47787084j, 0.30279699-0.10056307j, ...
41     0.13874319-0.01460122j, -0.27256915+0.12950497j, ..]
42
43     k.x(0)
44     u1 = ql.Unitary("testname",matrix)
45     u1.decompose()
46     k.gate(u1, [0,1])
47     k.display()
48     p.add_kernel(k)
49     p.compile()
50
51     qx.set(os.path.join(output_dir, p.name+'.qasm'))
52     qx.execute()
53     c0 = qx.get_state()
54     self.assertAlmostEqual(helper_prob(matrix[1]), helper_regex(c0
55     [0],5)
56     self.assertAlmostEqual(helper_prob(matrix[5]), helper_regex(c0 [1]
57     5)
58     self.assertAlmostEqual(helper_prob(matrix[9]), helper_regex(c0
59     [2],5)
60     self.assertAlmostEqual(helper_prob(matrix[13]), helper_regex(c0
61     [3],5)
62
63 def test_usingqx_10(self):
64     config_fn = os.path.join(curdir, 'test_cfg_none_simple.json')
65     platform = ql.Platform('platform_none', config_fn)
66     num_qubits = 4
67     p = ql.Program('test_usingqx10', platform, num_qubits)
68     k = ql.Kernel('akernel', platform, num_qubits)
69
70     matrix = [-0.15050486+0.32164259j, -0.29086861+0.76699622j, ...
71     -0.65629705+0.20915109j, 0.32782708+0.16363753j, ...
72     -0.38935965-0.47787084j, 0.30279699-0.10056307j, ...
73     0.13874319-0.01460122j, -0.27256915+0.12950497j, ..]
74
75     u1 = ql.Unitary("testname",matrix)
76     u1.decompose()
77     k.x(1)
78     k.gate(u1, [0,1])
79     p.add_kernel(k)
80     p.compile()

```



```

73     qx.set(os.path.join(output_dir, p.name+'.qasm'))
74     qx.execute()
75     c0 = qx.get_state()
76     self.assertAlmostEqual(helper_prob(matrix[2]), helper_regex(c0)
77     [0],5)
77     self.assertAlmostEqual(helper_prob(matrix[6]), helper_regex(c0)
78     [1],5)
78     self.assertAlmostEqual(helper_prob(matrix[10]), helper_regex(c0)
79     [2],5)
79     self.assertAlmostEqual(helper_prob(matrix[14]), helper_regex(c0)
80     [3],5)
80
81 def test_usingqx_11(self):
82     config_fn = os.path.join(curdir, 'test_cfg_none_simple.json')
83     platform = ql.Platform('platform_none', config_fn)
84     num_qubits = 4
85     p = ql.Program('test_usingqx11', platform, num_qubits)
86     k = ql.Kernel('akernel', platform, num_qubits)
87
88     matrix = [-0.15050486+0.32164259j, -0.29086861+0.76699622j, ...
89     -0.65629705+0.20915109j, 0.32782708+0.16363753j, ...
90     -0.38935965-0.47787084j, 0.30279699-0.10056307j, ...
91     0.13874319-0.01460122j, -0.27256915+0.12950497j, ..]
92     u1 = ql.Unitary("testname",matrix)
93     u1.decompose()
94     k.x(0)
95     k.x(1)
96     k.gate(u1, [0,1])
97
98     p.add_kernel(k)
99     p.compile()
100    qx.set(os.path.join(output_dir, p.name+'.qasm'))
101    qx.execute()
102    c0 = qx.get_state()
103
104    self.assertAlmostEqual(helper_prob(matrix[3]), helper_regex(c0)[0],
105    5)
105    self.assertAlmostEqual(helper_prob(matrix[7]), helper_regex(c0)[1],
106    5)
106    self.assertAlmostEqual(helper_prob(matrix[11]), helper_regex(c0)
107    [2], 5)
107    self.assertAlmostEqual(helper_prob(matrix[15]), helper_regex(c0)
108    [3], 5)

```

Listing D.19: Test complete matrix using the bell state

```

1 def test_usingqx_bellstate(self):
2     config_fn = os.path.join(curdir, 'test_cfg_none_simple.json')
3     platform = ql.Platform('platform_none', config_fn)
4     num_qubits = 4
5     p = ql.Program('test_usingqxbellstate', platform, num_qubits)
6     k = ql.Kernel('akernel', platform, num_qubits)
7
8     matrix = [-0.15050486+0.32164259j, -0.29086861+0.76699622j, ...
9     -0.65629705+0.20915109j, 0.32782708+0.16363753j, ...
10    -0.38935965-0.47787084j, 0.30279699-0.10056307j, ...
11    0.13874319-0.01460122j, -0.27256915+0.12950497j, ..]

```

```

12     u1 = ql.Unitary("testname",matrix)
13     u1.decompose()
14     k.hadamard(0)
15     k.cnot(0, 1)
16     k.display()
17
18     k.gate(u1, [0,1])
19     k.display()
20
21     p.add_kernel(k)
22     p.compile()
23     qx.set(os.path.join(output_dir, p.name+'.qasm'))
24     qx.execute()
25     c0 = qx.get_state()
26
27     self.assertAlmostEqual(0.5*helper_prob((matrix[0]+ matrix[3])),
28     helper_regex(c0)[0], 5)
29     self.assertAlmostEqual(0.5*helper_prob((matrix[4]+matrix[7])),
30     helper_regex(c0)[1], 5)
31     self.assertAlmostEqual(0.5*helper_prob((matrix[8]+ matrix[11])),
32     helper_regex(c0)[2], 5)
33     self.assertAlmostEqual(0.5*helper_prob((matrix[12]+ matrix[15])),
34     helper_regex(c0)[3], 5)

```

Listing D.20: Test using QX of a fully entangled state

```

1 def test_usingqx_fullyentangled(self):
2     config_fn = os.path.join(curdir, 'test_cfg_none_simple.json')
3     platform = ql.Platform('platform_none', config_fn)
4     num_qubits = 4
5     p = ql.Program('test_usingqxfullyentangled', platform, num_qubits)
6     k = ql.Kernel('akernel', platform, num_qubits)
7
8     matrix = [-0.15050486+0.32164259j, -0.29086861+0.76699622j,...
9     -0.65629705+0.20915109j,  0.32782708+0.16363753j,...
10    -0.38935965-0.47787084j,  0.30279699-0.10056307j,...
11    0.13874319-0.01460122j, -0.27256915+0.12950497j,..]
12    u1 = ql.Unitary("testname",matrix)
13    u1.decompose()
14    k.hadamard(0)
15    k.hadamard(1)
16    k.cnot(0, 1)
17    k.display()
18
19    k.gate(u1, [0,1])
20    k.display()
21
22    p.add_kernel(k)
23    p.compile()
24    qx.set(os.path.join(output_dir, p.name+'.qasm'))
25    qx.execute()
26    c0 = qx.get_state()
27
28    self.assertAlmostEqual(0.25*helper_prob((matrix[0] + matrix[1] +
29    matrix[2] + matrix[3] )), helper_regex(c0)[0], 5)
30    self.assertAlmostEqual(0.25*helper_prob((matrix[4] + matrix[5] +
31    matrix[6] + matrix[7] )), helper_regex(c0)[1], 5)

```

```

30 self.assertAlmostEqual(0.25*helper_prob((matrix[8] + matrix[9] +
matrix[10]+ matrix[11])), helper_regex(c0)[2], 5)
31 self.assertAlmostEqual(0.25*helper_prob((matrix[12]+ matrix[13]+
matrix[14]+ matrix[15])), helper_regex(c0)[3], 5)

```

Listing D.21: Test a full three qubit unitary using a fully entangled state and QX

```

1 def test_usingqx_fullyentangled_3qubit(self):
2     config_fn = os.path.join(curdir, 'test_cfg_none_simple.json')
3     platform = ql.Platform('platform_none', config_fn)
4     num_qubits = 3
5     p = ql.Program('test_usingqxfullyentangled_3qubit', platform,
num_qubits)
6     k = ql.Kernel('akernel', platform, num_qubits)
7
8     matrix =
9     [-6.71424156e-01+0.20337111j, -1.99816541e-02+0.41660484j, ...
10 -2.53660227e-01+0.17691007j, 3.13791732e-01+0.27041001j, ...
11 1.00349815e-01-0.11576965j, 2.96991063e-01+0.12974051j, ...
12 -1.47285402e-01+0.02010648j, -1.02440846e-01-0.36282662j, ..]
13
14     u1 = ql.Unitary("testname", matrix)
15     u1.decompose()
16     k.hadamard(0)
17     k.hadamard(1)
18     k.hadamard(2)
19     k.cnot(0, 1)
20     k.cnot(0, 2)
21     k.cnot(1, 2)
22     k.display()
23
24     k.gate(u1, [0,1, 2])
25     k.display()
26
27     p.add_kernel(k)
28     p.compile()
29     qx.set(os.path.join(output_dir, p.name+'.qasm'))
30     qx.execute()
31     c0 = qx.get_state()
32
33     self.assertAlmostEqual(0.125*helper_prob((matrix[0] + matrix[1] +
matrix[2] + matrix[3] + matrix[4] + matrix[5] + matrix[6] + matrix
[7])), helper_regex(c0)[0], 5)
34     self.assertAlmostEqual(0.125*helper_prob((matrix[8] + matrix[9] +
matrix[10]+ matrix[11]+ matrix[12]+ matrix[13]+ matrix[14]+ matrix
[15])), helper_regex(c0)[1], 5)
35     ...# full comparison snipped for length

```

Listing D.22: Test with file comparison of decomposition of a 4 qubit gate

```

1 def test_usingqx_fullyentangled_4qubit(self):
2     config_fn = os.path.join(curdir, 'test_cfg_none_simple.json')
3     platform = ql.Platform('platform_none', config_fn)
4     num_qubits = 4
5     p = ql.Program('test_usingqxfullyentangled_4qubit', platform,
num_qubits)
6     k = ql.Kernel('akernel', platform, num_qubits)
7

```

```

8     matrix = [-0.11921901-0.30382154j, -0.10645804-0.11760155j,    ...
9     -0.12435741-0.07223017j,  0.00178745+0.14813263j,    ...
10    ...# matrix snipped for length
11
12    u1 = ql.Unitary("testname",matrix)
13    u1.decompose()
14    k.hadamard(0)
15    k.hadamard(1)
16    k.hadamard(2)
17    k.hadamard(3)
18    k.cnot(0, 1)
19    k.cnot(0, 2)
20    k.cnot(0, 3)
21    k.cnot(1, 2)
22    k.cnot(1, 3)
23    k.cnot(2, 3)
24    k.gate(u1, [0, 1, 2, 3])
25    k.display()
26
27    p.add_kernel(k)
28    p.compile()
29    qx.set(os.path.join(output_dir, p.name+'.qasm'))
30    qx.execute()
31    c0 = qx.get_state()
32
33    self.assertAlmostEqual(0.0625*helper_prob((matrix[0] + matrix[1]
34    + matrix[2] + matrix[3] + matrix[4] + matrix[5] + matrix[6] +
35    matrix[7] + matrix[8] + matrix[9] + matrix[10]+ matrix[11]+
36    matrix[12]+ matrix[13]+ matrix[14]+ matrix[15])), helper_regex(c0
37    )[0], 5)
38    self.assertAlmostEqual(0.0625*helper_prob((matrix[16] + matrix
39    [17]+ matrix[18]+ matrix[19]+ matrix[20]+ matrix[21]+ matrix
40    [22]+ matrix[23]+ matrix[24] + matrix[25]+ matrix[26]+ matrix
41    [27]+ matrix[28]+
42    ...# full comparison snipped for length

```

Listing D.23: Test using QX of 5 qubit unitary

```

1 def test_usingqx_5qubit(self):
2     config_fn = os.path.join(curdir, 'test_cfg_none_simple.json')
3     platform = ql.Platform('platform_none', config_fn)
4     num_qubits = 5
5     p = ql.Program('test_usingqx_5qubit', platform, num_qubits)
6     k = ql.Kernel('akernel', platform, num_qubits)
7
8     matrix =[0.31869031-9.28734622e-02j, 0.38980148+2.14592427e-01j,
9     ...
10    ...# matrix snipped for length
11
12    k.display()
13    u1 = ql.Unitary("testname",matrix)
14    u1.decompose()
15    k.gate(u1, [0, 1, 2, 3, 4])
16    k.display()
17
18    p.add_kernel(k)
19    p.compile()

```

```

19 qx.set(os.path.join(output_dir, p.name+'.qasm'))
20 qx.execute()
21 c0 = qx.get_state()
22
23
24 self.assertAlmostEqual(helper_prob(matrix[0]), helper_regex(c0)
[0], 5)
25 self.assertAlmostEqual(helper_prob(matrix[32]), helper_regex(c0)
[1], 5)
26 ...# full comparison snipped for length

```

Listing D.24: Test using QX of 5 qubit unitary, while starting with a different qubit state

```

1 def test_usingqx_fullyentangled_5qubit_10011(self):
2     config_fn = os.path.join(curdir, 'test_cfg_none_simple.json')
3     platform = ql.Platform('platform_none', config_fn)
4     num_qubits = 5
5     p = ql.Program('test_usingqxfullyentangled_5qubit_10011', platform,
num_qubits)
6     k = ql.Kernel('akernel', platform, num_qubits)
7
8     matrix = [ 0.31869031-9.28734622e-02j, 0.38980148+2.14592427e-01j,
0.20279154-1.56580826e-01j, 0.00210273-2.83970875e-01j,
9     ...# full matrix snipped for length
10    u1 = ql.Unitary("testname",matrix)
11    u1.decompose()
12    k.x(4)
13    k.x(1)
14    k.x(0)
15    k.display()
16    k.gate(u1, [0, 1, 2, 3, 4])
17    k.display()
18
19    p.add_kernel(k)
20    p.compile()
21    qx.set(os.path.join(output_dir, p.name+'.qasm'))
22    qx.execute()
23    c0 = qx.get_state()
24
25
26    self.assertAlmostEqual(helper_prob(matrix[19+0]), helper_regex(c0)
) [0], 5)
27    self.assertAlmostEqual(helper_prob(matrix[19+32]), helper_regex(c0)
) [1], 5)
28    ...# full comparison snipped for length

```

Listing D.25: Test for adding a 6 qubit unitary matrix. No decomposition because that is too computationally heavy for a test suite.

```

1 def test_adding2tothepower6unitary(self):
2     config_fn = os.path.join(curdir, 'test_cfg_none_simple.json')
3     platform = ql.Platform('platform_none', config_fn)
4     num_qubits = 6
5     p = ql.Program('test_6qubitjustadding', platform, num_qubits)
6     k = ql.Kernel('akernel', platform, num_qubits)
7
8     matrix = [ 3.66034993e-01+2.16918726e-01j, 4.63119758e
-02-1.25284236e-01j, 3.23689480e-01-1.67028180e-02j, -4.01291192e

```

```

-02-1.53117445e-01j,
9   ...#full matrix snipped for length
10
11   u1 = ql.Unitary("big6qubitone", matrix)

```

Listing D.26: Test with QX for full 4 qubit matrix

```

1 def test_usingqx_fullunitary(self):
2     config_fn = os.path.join(curdir, 'test_cfg_none_simple.json')
3     platform = ql.Platform('platform_none', config_fn)
4     num_qubits = 5
5     p = ql.Program('test_usingqxfullunitary', platform, num_qubits)
6     k = ql.Kernel('akernel', platform, num_qubits)
7
8     matrix = [-0.11921901-0.30382154j, -0.10645804-0.11760155j,
9     -0.09639953-0.0353926j , -0.32605797+0.19552924j,
10    ...#full matrix snipped for length
11
12    u1 = ql.Unitary("testname",matrix)
13    u1.decompose()
14    k.hadamard(0)
15    k.hadamard(1)
16    k.hadamard(2)
17    k.hadamard(3)
18    k.cnot(0, 1)
19    k.cnot(0, 2)
20    k.cnot(0, 3)
21    k.cnot(1, 2)
22    k.cnot(1, 3)
23    k.cnot(2, 3)
24    k.gate(u1, [0, 1, 2, 3])
25    k.display()
26
27    p.add_kernel(k)
28    p.compile()
29    qx.set(os.path.join(output_dir, p.name+'.qasm'))
30    qx.execute()
31    c0 = qx.get_state()
32
33    self.assertAlmostEqual(0.0625*helper_prob((matrix[0] + matrix[1]
34    + matrix[2] + matrix[3] + matrix[4] + matrix[5] + matrix[6] +
35    matrix[7] + matrix[8] + matrix[9] + matrix[10]+ matrix[11]+
36    matrix[12]+ matrix[13]+ matrix[14]+ matrix[15])), helper_regex(c0
37    )[0], 5)
38
39    self.assertAlmostEqual(0.0625*helper_prob((matrix[16] + matrix
40    [17]+ matrix[18]+ matrix[19]+ matrix[20]+ matrix[21]+ matrix
41    [22]+ matrix[23]+ matrix[24] + matrix[25]+ matrix[26]+ matrix
42    [27]+ matrix[28]+ matrix[29]+ matrix[30]+ matrix[31])),
43    helper_regex(c0)[1], 5)
44    ...# full comparison snipped for length

```

Listing D.27: Test with extremely sparse unitary

```

1 def test_extremelysparseunitary(self):
2     config_fn = os.path.join(curdir, 'test_cfg_none_simple.json')
3     platform = ql.Platform('platform_none', config_fn)
4     num_qubits = 4

```

```

5  p = ql.Program('test_usingqx_extremelysparseunitary_newname',
platform, num_qubits)
6  k = ql.Kernel('akernel', platform, num_qubits)
7
8  matrix = [-0.59111943+0.15726005j, -0.          +0.j          ,
-0.15509793+0.32339668j, -0.          +0.j          ,
-0.33317562+0.00860528j, -0.          +0.j          , -0.5566068
+0.27625195j, -0.          +0.j          , -0.          +0.j          , -0.
          +0.j          , -0.          +0.j          , -0.          +0.j          , -0.
          +0.j          , -0.          +0.j          , -0.          +0.j          , -0.          +0.j
          , -0.          +0.j          , -0.          +0.j          , -0.59111943+0.15726005j, -0.
          +0.j          , -0.15509793+0.32339668j, -0.          +0.j          ,
-0.33317562+0.00860528j, -0.          +0.j          , -0.5566068
+0.27625195j,
9  ...# matrix snipped for length
10 u1 = ql.Unitary("testname",matrix)
11 u1.decompose()
12 k.hadamard(0)
13 k.hadamard(1)
14 k.hadamard(2)
15 k.hadamard(3)
16 k.cnot(0, 1)
17 k.cnot(0, 2)
18 k.cnot(0, 3)
19 k.cnot(1, 2)
20 k.cnot(1, 3)
21 k.cnot(2, 3)
22 k.display()
23
24
25 k.gate(u1, [0, 1, 2, 3])
26 k.display()
27
28 p.add_kernel(k)
29 p.compile()
30 qx.set(os.path.join(output_dir, p.name+'.qasm'))
31 qx.execute()
32 c0 = qx.get_state()
33
34 self.assertAlmostEqual(0.0625*helper_prob((matrix[0] + matrix[1]
+ matrix[2] + matrix[3] + matrix[4] + matrix[5] + matrix[6] +
matrix[7] + matrix[8] + matrix[9] + matrix[10]+ matrix[11]+
matrix[12]+ matrix[13]+ matrix[14]+ matrix[15])), helper_regex(c0
)[0], 5)
35 self.assertAlmostEqual(0.0625*helper_prob((matrix[16] + matrix
[17]+ matrix[18]+ matrix[19]+ matrix[20]+ matrix[21]+ matrix
[22]+ matrix[23]+ matrix[24] + matrix[25]+ matrix[26]+ matrix
[27]+ matrix[28]+ matrix[29]+ matrix[30]+ matrix[31])),
helper_regex(c0)[1], 5)
36 ...# full comparison snipped for length

```

Listing D.28: Test with file comparison of decomposition of a sparse matrix. In this case, the second qubit is not affected.

```

1 def test_sparse2qubitunitary(self):
2     config_fn = os.path.join(curdir, 'test_cfg_none_simple.json')
3     platform = ql.Platform('platform_none', config_fn)
4     num_qubits = 2

```

```

5 p = ql.Program('test_usingqx_sparse2qubit', platform, num_qubits)
6 k = ql.Kernel('akernel', platform, num_qubits)
7
8 matrix = [ 0.2309453 -0.79746147j, -0.53683301+0.15009925j, 0.
9           +0.j           , -0.           +0.j
10          , 0.39434916-0.39396473j, 0.80810853+0.19037107j, 0.           +0.j
11          , 0.           +0.j           , -0.           +0.j           , 0.2309453
12          -0.79746147j, -0.53683301+0.15009925j
13          , 0.           +0.j           , 0.           +0.j           ,
14          0.39434916-0.39396473j, 0.80810853+0.19037107j]
15
16 u1 = ql.Unitary("testname",matrix)
17 u1.decompose()
18 k.gate(u1, [0, 1])
19 k.display()
20
21 p.add_kernel(k)
22 p.compile()
23 qx.set(os.path.join(output_dir, p.name+'.qasm'))
24 qx.execute()
25 c0 = qx.get_state()
26
27 self.assertAlmostEqual(helper_prob(matrix[0]), helper_regex(c0)[0],
28                          5)
29 self.assertAlmostEqual(helper_prob(matrix[4]), helper_regex(c0)[1],
30                          5)
31 self.assertAlmostEqual(helper_prob(matrix[8]), 0, 5) # Zero
32 probabilities do not show up in the output list
33 self.assertAlmostEqual(helper_prob(matrix[12]), 0, 5)

```

Listing D.29: Test with file comparison of decomposition of a sparse matrix. In this case, the first qubit is not affected

```

1 def test_sparse2qubitunitaryotherqubit(self):
2     config_fn = os.path.join(curdir, 'test_cfg_none_simple.json')
3     platform = ql.Platform('platform_none', config_fn)
4     num_qubits = 2
5     p = ql.Program('test_usingqx_sparse2qubitotherqubit', platform,
6                   num_qubits)
7     k = ql.Kernel('akernel', platform, num_qubits)
8
9     matrix = [ 0.30279949-0.60010283j, 0.           +0.j           ,
10             -0.58058628-0.45946559j, 0.           -0.j           ,
11             , 0.           +0.j           , 0.30279949-0.60010283j, 0.
12             -0.j           , -0.58058628-0.45946559j
13             , 0.04481146-0.73904059j, 0.           +0.j           ,
14             0.64910478+0.17456782j, 0.           +0.j           ,
15             , 0.           +0.j           , 0.04481146-0.73904059j, 0.
16             +0.j           , 0.64910478+0.17456782j]
17
18     u1 = ql.Unitary("testname",matrix)
19     u1.decompose()
20     k.gate(u1, [0, 1])
21
22     p.add_kernel(k)
23     p.compile()
24     qx.set(os.path.join(output_dir, p.name+'.qasm'))

```



```
20 qx.execute()
21 c0 = qx.get_state()
22 self.assertAlmostEqual(helper_prob(matrix[0]), helper_regex(c0)[0],
23                          5)
23 self.assertAlmostEqual(helper_prob(matrix[4]), 0, 5) # Zero
probabilities do not show up in the output list
24 self.assertAlmostEqual(helper_prob(matrix[8]), helper_regex(c0)[2],
25                          5)
25 self.assertAlmostEqual(helper_prob(matrix[12]), 0, 5)
```



# Bibliography

- [1] Khan Academy. Dna sequencing. URL <https://www.khanacademy.org/science/high-school-biology/hs-molecular-genetics/hs-biotechnology/a/dna-sequencing>. Accessed on: 08-07-2019.
- [2] Steve Acquaro. From dna to protein. <https://sites.google.com/site/obenscience7e/unit-7/from-dna-to-protein>. Accessed on: 10-01-2019.
- [3] D. Adamski, G. Jablonski, P. Perek, and A. Napieralski. Polyhedral source-to-source compiler. *2016 MIXDES - 23rd International Conference Mixed Design of Integrated Circuits and Systems*, pages 458–463, 2016.
- [4] G. Donald Allen. Unitary matrices. In *Lectures on Linear Algebra and Matrices*, chapter 4, pages 157-180. Texas A&M University, College Station, TX, 2003.
- [5] Wilhelm J. Ansorge. Next-generation dna sequencing techniques. *New Biotechnology*, 25(4): 195–203, April 2009.
- [6] A. Barenco, C. H. Bennett, R. Cleve, D. P. DiVincenzo, N. Margolus, P. Shor, T. Sleator, J. Smolin, and H. Weinfurter. Elementary gates for quantum computation. *Physical Review A*, November 1995.
- [7] Ville Bergholm, Josh Izaac, Maria Schuld, Christian Gogolin, and Nathan Killoran. Pennylane: Automatic differentiation of hybrid quantum-classical computations. Technical report, 2019. URL [arXiv:1811.04968v2](https://arxiv.org/abs/1811.04968v2).
- [8] A. Bruce, A. Johnson, J. Lewis, M. Raff, K. Roberts, and P. Walter. *Molecular Biology of the Cell*. Garland Science, 2002.
- [9] Rigetti Computing. Welcome to quantum cloud services. URL <https://www.rigetti.com/qcs>. Accessed on: 08-07-2019.
- [10] Intel Corporation. Reinventing data processing with quantum computing, 2019. URL [www.intel.com/content/www/us/en/research/quantum-computing.html](http://www.intel.com/content/www/us/en/research/quantum-computing.html). Accessed on: 22-08-2019.
- [11] D-wave. The D-Wave 2000Q quantum computer technology overview 0117F. <https://www.dwavesys.com/quantum-computing>. Accessed on: 26-04-2019.
- [12] Henning Dekant, Henry Tregillus, Robert Tucci, and Tao Yin. Qubiter at github. URL <https://github.com/artiste-qb-net/qubiter>. Accessed on: 27-05-2019.
- [13] David Eliesser Deutsch and Roger Penrose. Quantum computational networks. *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences*, 425(1868):73–90, 1989. doi: 10.1098/rspa.1989.0099.
- [14] Benoît Jacob (founder), Gaël Guennebaud (guru), and many more. The eigen documentation, 2019. URL [http://eigen.tuxfamily.org/index.php?title=Main\\_Page](http://eigen.tuxfamily.org/index.php?title=Main_Page). Accessed on: 04-09-2019.
- [15] The Genomes Project Consortium. A global reference for human genetic variation, the 1000 genomes project consortium. *Nature*, 526:68–74, 01 2015.
- [16] G.H. Golub and C. F. Van Loan. *Matrix Computations*. The Jogn Hopkins University Press.

- [17] Alibaba Group. Alibaba quantum laboratory(aql)-process engineer, quantum computing-hangzhou, 2019. URL [https://job.alibaba.com/zhaopin/position\\_detail.htm?positionId=68103](https://job.alibaba.com/zhaopin/position_detail.htm?positionId=68103). Accessed on: 22-08-2019.
- [18] D. Grume, H. E. Bal, C. J. H. Jacobs, and K. G. Langendoen. *Modern Compiler Design*. Wiley, 2000.
- [19] Lloyd C. L. Hollenberg. Fast quantum search algorithms in protein sequence comparisons: Quantum bioinformatics. *Phys. Rev. E*, 62:7532–7535, Nov 2000.
- [20] Computer Hope. Transcompiler. <https://www.computerhope.com/jargon/t/transcompiler.htm>, 2017. Accessed: 08-01-2019.
- [21] IBM. The JIT compiler. [https://www.ibm.com/support/knowledgecenter/en/SSYKE2\\_8.0.0/com.ibm.java.vm.80.doc/docs/jit\\_overview.html](https://www.ibm.com/support/knowledgecenter/en/SSYKE2_8.0.0/com.ibm.java.vm.80.doc/docs/jit_overview.html). Accessed on: 08-01-2019.
- [22] IBM. Qiskit documentation. <https://qiskit.org/>, 2019. Accessed on: 27-05-2019.
- [23] D-Wave Systems Inc. Introduction to quantum annealing. URL [https://docs.dwavesys.com/docs/latest/c\\_gs\\_2.html](https://docs.dwavesys.com/docs/latest/c_gs_2.html). Accessed on: 22-08-2019.
- [24] Intel. <https://software.intel.com/en-us/mkl-developer-reference-c-singular-value-de>. Accessed on: 21-03-2019.
- [25] Interesting Engineering John Loeffler. 5 intractable problems quantum computing will solve, 2018. URL <https://interestingengineering.com/5-intractable-problems-quantum-computing-will-solve>. Accessed on: 22-08-2019.
- [26] K. Keutzer and W. Wolf. Anatomy of a hardware compiler. *SIGPLAN Not.*, 23(7):95–104, June 1988. ISSN 0362-1340. doi: 10.1145/960116.54000. URL <http://doi.acm.org/10.1145/960116.54000>.
- [27] N. Khamassie, I. Ashraf, J. v. Someren, X. Rol, Fu, D.M. Manzano, L. Lao, C.G. Almudever, L. DiCarlo, and K. Bertels. Openq1 : A portable quantum programming framework for quantum accelerators. 2018.
- [28] N. Khammassi, I Ashraf, A. Rol, X Fu, and W. Vlothuizen. Openq1 at github. URL <https://github.com/QE-Lab/OpenQL>. Accessed on: 08-09-2019.
- [29] Nathan Killoran, Josh Izaac, Nicolás Quesada, Ville Bergholm, Matthew Amy, and Christian Weedbrook. Strawberry Fields: A Software Platform for Photonic Quantum Computing. Technical report, March 2019. URL <https://doi.org/10.22331/q-2019-03-11-129>.
- [30] Fujitsu Limited. Digital annealer introduction. <https://www.fujitsu.com/global/digitalannealer/superiority/>, February 2019. accessed on: 25-04-2019.
- [31] P. Mateus and Y. Omar. Quantum Pattern Matching. *eprint arXiv:quant-ph/0508237*, August 2005.
- [32] FutureCar Michael Cheng. Ford-nasa partnership applies quantum computing to autonomous vehicle research, 2018. URL <https://www.futurecar.com/2713/Ford-NASA-Partnership-Applies-Quantum-Computing-to-Autonomous-Vehicle-Research>. Accessed on: 22-08-2019.
- [33] Mikko Möttönen, Juha J. Vartiainen, Ville Bergholm, and Martti M. Salomaa. Quantum circuits for general multiqubit gates. *Phys. Rev. Lett.*, 93:130502, Sep 2004.
- [34] Mikko Möttönen and Juha Vartiainen. Decompositions of general quantum gates. *Frontiers in Artificial Intelligence and Applications*, 05 2005.
- [35] J. Nordlander and M. P. Jones. Slides of lecture 11: Introduction to optimization. Computer Science and Electrical Engineering, Lulea University of Technology.

- [36] C.C. Paige and M Wei. History and generality of the cs decomposition. *Linear Algebra and its Applications*, 208-209:303–326, 09 1994. doi: 10.1016/0024-3795(94)90446-4.
- [37] Jeffrey M. Perkel. De novo sequencing: Assembling a genome from scratch. *Biocompare*. <https://www.biocompare.com/Editorial-Articles/140207-De-Novo-Sequencing/>, Accessed on: 08-07-2019.
- [38] Aritra Sarkar. Quantum algorithms for pattern-matching in genomic sequences. Master’s thesis, 2018.
- [39] V. Shende, S. S. Bullock, and I. Markov. Synthesis of quantum logic circuits. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 25:1000 – 1010, July 2006. doi: 10.1109/TCAD.2005.855930.
- [40] Vivek V. Shende, Igor L. Markov, and Stephen S. Bullock. Minimal universal two-qubit cnot-based circuits.
- [41] Metal Supermarkets. What is annealing?, 2018. URL <https://www.metalsupermarkets.com/what-is-annealing/>. Accessed on: 08-07-2019.
- [42] Synced. Alibaba launches chip company “ping-tou-ge”; pledges quantum chip, 2018. URL <https://medium.com/syncedreview/chinese-internet-mogul-jack-ma-has-a-flair-for-naming-new-businesses-alibaba-originat>. Accessed on: 22-08-2019.
- [43] Luke Tierney. Byte code compiler. <http://homepage.divms.uiowa.edu/~luke/xls/interface98/paper/node3.html>, 1998. Accessed: 08-01-2019.
- [44] Robert R. Tucci. A rudimentary quantum compiler (qubiter), 2008.
- [45] University of Chicago University of Utah and the Arizona Board of Regents. [www.swig.org](http://www.swig.org). Accessed on: 18-03-2019.
- [46] J. van Straaten. Towards a common quantum assembly language cqasm 2.0. Accessed on: 19-03-2019.
- [47] J.J. Vartiainen, Möttönen, and M.M. Salomaa. Efficient decomposition of quantum gates. *Physical Review Letters*, 92(17):177902, april 2004.
- [48] Farrokh Vatan and Colin P. Williams. Realization of a general three-qubit quantum gate. 02 2004.
- [49] Dan Ventura and Tony Martinez. Initializing the amplitude distribution of a quantum state. *Foundations of Physics Letters*, 12(6):547–559, Dec 1999.
- [50] MIT Technology Review Yiting Sun. Why alibaba is betting big on ai chips and quantum computing. URL <https://www.technologyreview.com/s/612190/why-alibaba-is-investing-in-ai-chips-and-quantum-computing/>.
- [51] C. Yoshimura, M. Hayashi, T. Okuyama, and M. Yamaoka. Fpga-based annealing processor for ising model. pages 436–442, Nov 2016.
- [52] South China Morning Post Zen Soo. Alibaba cloud steps up its game as it offers quantum computing service, 2018. URL <https://www.scmp.com/tech/china-tech/article/2134520/alibaba-cloud-steps-its-game-it-offers-quantum-computing-service>. Accessed on: 22-08-2019.