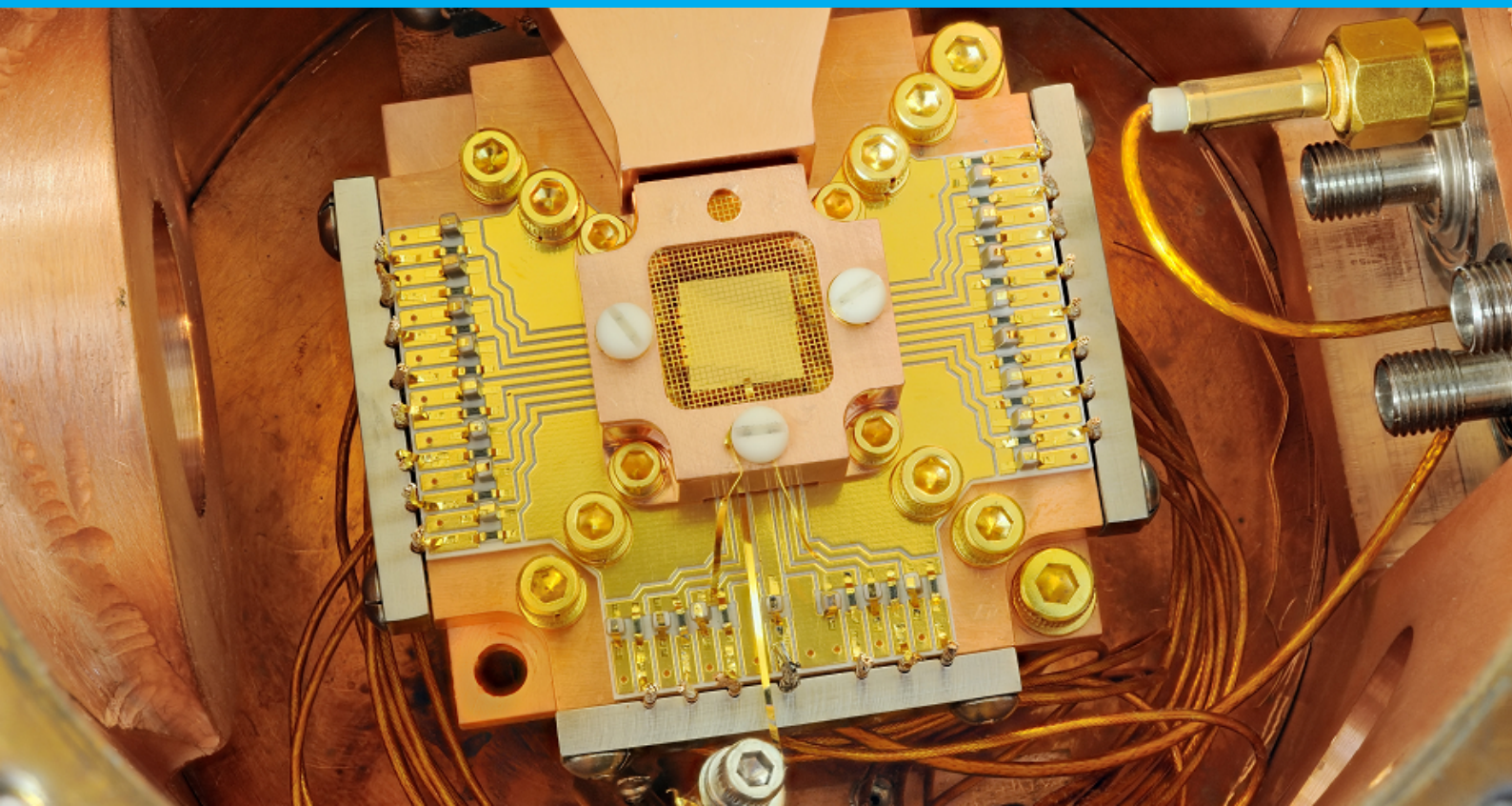


Visualization tools

for the OpenQL compiler

Tim van der Meer



Visualization tools

for the OpenQL compiler

by

Tim van der Meer

to obtain the degree of Master of Science

at the Delft University of Technology,

to be defended publicly on Wednesday June 16, 2021 at 10:30 AM.

Student number:	4012712
Project duration:	January 13, 2020 – June 16th, 2021
Thesis committee:	Dr. F. Sebastiano, AQUA, TU Delft Dr. C. G. Almudever, Senior Researcher Dr. L. DiCarlo, QuTech, TU Delft
Supervisor:	Dr. C. G. Almudever, Senior Researcher

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

Abstract

The rapid progress quantum devices have made in recent years has led to the need for systems that bridge the gap between quantum algorithms and quantum hardware. To this purpose different full-stack quantum programming platforms have been developed, providing high level languages for expressing quantum algorithms and providing compilers for making those quantum algorithms executable on a given quantum device. OpenQL is one such a platform, with support for compiling a variety of quantum program inputs into a quantum circuit, ready to be executed on the targeted quantum hardware.

OpenQL is an evolving tool in which new features are constantly added to improve its performance and extend its functionality. In order to enhance OpenQL and provide some extra support to the researchers using OpenQL for their experiments, a visualization tool with three main functionalities has been developed in this thesis project. First, a circuit visualizer, with support for both visualizing the circuit output of the OpenQL compiler as an abstract gate representation and a pulse representation, which has been made specifically for quantum hardware running on superconducting qubits. Secondly, a mapping graph visualizer, which allows displaying the logical to physical qubit mapping per program cycle. And lastly a qubit interaction graph generator, which shows the required interactions between qubits of a given quantum algorithm.

Acknowledgements

First of all I want to thank my supervisor, Dr. Carmina G. Almudéver, for the support she has given me throughout this project, even in the difficult year of 2020 with the entire world in lockdown due to the coronavirus.

I would also like to extend thanks to Hans van Someren en Razvan Nane for their knowledge of the OpenQL compiler, Hans specifically with regards to the mapper and schedule. We have spent quite some hours trying to get me to understand the workings of the mapper and to devise ways to have the mapper generate the required output for the mapping graph visualizer. Razvan for his help with setting up OpenQL on my machine and the work that was required to have the visualizer build on a Windows machine, and of course for his patience with my (sometimes) slow progress.

Jeroen van Straten provided me with a lot of help in the later stages of the project, especially as he himself worked tirelessly to refactor the code base of OpenQL to prepare it for future extensions and functionality.

The other people of the Quantum Computer Architecture group also deserve my fullest thanks: Matt Steinberg, Ahmed Abid Moueddene, Medina Bandic, Nikiforos Paraskevopoulos, and the new head of the group: Sebastian Feld.

Lastly, but not least, I want to thank my family and my parents in particular, who have supported me wherever needed during these trying times.

Tim van der Meer

Contents

1	Introduction	6
1.1	Motivation	6
1.1.1	The quantum stack	6
1.1.2	The OpenQL platform	7
1.2	Problem definition	8
1.3	Thesis outline.	8
2	Quantum computing	10
2.1	Quantum computing	10
2.1.1	A bit of history	10
2.1.2	From bits to qubits.	10
2.1.3	Quantum gates.	12
2.2	Quantum hardware	13
2.2.1	Robustness of quantum systems	13
2.2.2	Realisation of qubits	14
2.3	The progress of quantum computing	15
3	The OpenQL platform	17
3.1	Quantum computation platforms.	17
3.1.1	Qiskit	17
3.1.2	Cirq	17
3.1.3	Quantum Development Kit	18
3.1.4	Forest	18
3.1.5	Differences between platforms.	18
3.2	OpenQL.	19
3.2.1	A platform for quantum computation	19
3.2.2	Mapper	20
3.2.3	Scheduler	21
3.2.4	Motivation for the visualization tools	22
3.2.5	Future work	22

4	Choosing an implementation for the visualizer	23
4.1	Differences between C++ and Python	23
4.1.1	Object oriented	23
4.1.2	Typing	24
4.1.3	Memory control	24
4.1.4	Conclusion.	24
4.2	Requirements.	24
4.2.1	General requirements	24
4.2.2	Requirements for experiments.	25
4.2.3	Requirements for developers of OpenQL.	25
4.3	Existing code base	26
4.4	Comparison of existing visualizers	26
4.4.1	Qiskit	26
4.4.2	PyQuil	27
4.4.3	Cirq	27
4.4.4	QDK	27
4.4.5	QX simulator IDE	27
4.5	Code reuse	27
4.6	Image library	27
4.6.1	Python image libraries	28
4.6.2	C++ image libraries	28
4.7	Conclusion	29
5	Visualizer features	30
5.1	Preliminary work	30
5.1.1	Compiling OpenQL on Windows.	30
5.1.2	Development environment	30
5.2	General architecture	30
5.2.1	Setting up the visualizer	31
5.2.2	Source structure	31
5.2.3	Configuration	31
5.2.4	Example circuit	31
5.3	Circuit visualization.	32
5.3.1	Features	32
5.3.2	Architecture	34

- 5.3.3 Future work 35
- 5.4 Qubit interaction graph visualization 35
 - 5.4.1 Features 36
 - 5.4.2 Architecture 36
 - 5.4.3 Future work 36
- 5.5 Mapping graph visualization 37
 - 5.5.1 Features 37
 - 5.5.2 Architecture 37
 - 5.5.3 Future work 38
- 6 Conclusion** **39**
- References** **41**



Introduction

In this chapter the motivation behind the thesis is described in section 1.1, the problem is defined in section 1.2 and the structure of the thesis is outlined in section 1.3.

1.1. Motivation

1.1.1. The quantum stack

When computers were first invented, back in the early 20th century, computer programs were made by directly writing a program in the language the machine could understand. For larger programs this process was error-prone and would take very large amounts of time. To ease the burden on the computer programmer, specialized computer programs were invented, called compilers, that would take a human-readable language (but still written with a specific syntax: the programming language) and transform it to a format the computer could read and process. This enabled programmers to write larger programs much quicker and with less errors, provided the compiler was correct.

These days virtually all computer programs are written in a high-level language, and then transformed by a corresponding compiler into machine code. A compiler for a classical (in this context, classical means non-quantum) language typically involves most of the following steps (see Figure 1.1):

- Lexical analysis: breaking down the program into tokens
- Syntax analysis: checking whether the sequence of tokens adheres to the grammar of the programming language
- Semantic analysis: consists of constructing the symbol table and type checking
- Dataflow Analysis: this step analyses the flow of data in the program, code dependencies and much more
- Optimization: taking the results from the analysis and changing it in such a way that the code will run faster but the result will still be exactly the same
- Code generation: this step constructs a valid program for the target machine written in the language that machine can understand

Combined with many other tools that came into being since computers gained mainstream use, like IDE's, version control, build systems, automated test suites and much more, the quality of life of a programmer has vastly improved. None of the programs we use daily would exist without this extensive stack of tools.

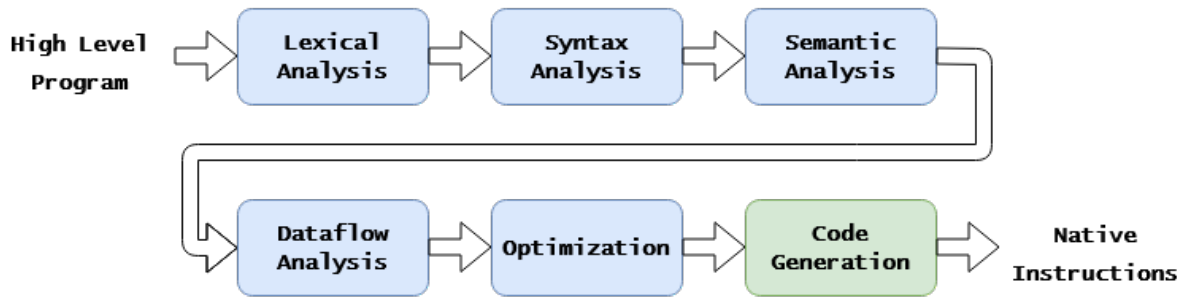


Figure 1.1: The general compilation process.

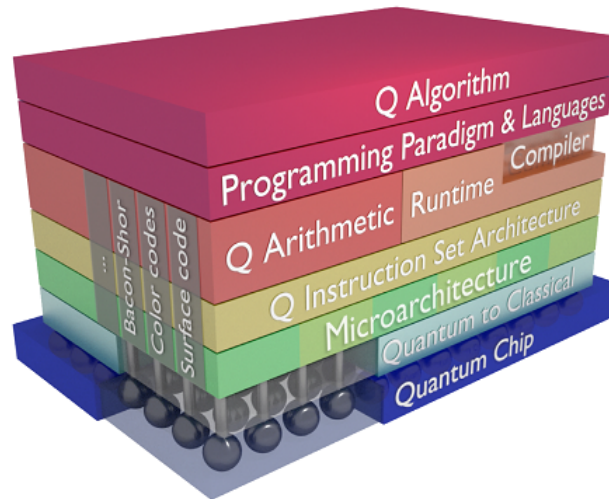


Figure 1.2: The full quantum stack [14].

With quantum computers gaining more and more accessible qubits and quantum algorithms becoming more complex, the need for such tools usable for quantum computation arises. Figure 1.2 shows the full stack for quantum computing. The base level consists of the physical realisation of the quantum hardware. The second level is the layer that serves as an interface between the classical part operating the quantum computer and the quantum hardware. This layer is responsible for sending the correct physical signals which enable the qubits to perform the required operation such as a specific gate, or to measure a qubit and store the classical result. The next two levels determine the instruction set for the quantum computer, for which the compiler will generate code from the high level description of the quantum algorithm that resides on the top level.

Several quantum programming platforms for quantum computation have been developed in the last years, many of which will be discussed in a later chapter. These platforms provide a fully incorporated programming suite that combines all or several layers of the quantum stack in one easy to use solution. Several notable [18] platforms are Qiskit, PyQuil, ProjectQ and the subject of this thesis: OpenQL.

1.1.2. The OpenQL platform

OpenQL is a software platform for quantum computation. It enables users to define a quantum algorithm in a high level programming language (the OpenQL language). This is in contrast to directly defining operations on specific qubits, which is difficult, if not impossible, to do for large programs and prone to errors even for smaller programs. The OpenQL compiler translates the high level language into instructions that the specified quantum hardware back-end can understand and execute.

Compiling a quantum program requires several special phases that are not necessary for classical compilation; such as the mapping and routing of qubits in which qubits in the circuit are assigned to physical qubits on the hardware and moved to adjacent positions when they need to interact; decomposition, which decom-

poses gates that are not native on the quantum hardware's instruction set unto gates that are native to that hardware; scheduling, in which quantum operations are scheduled to leverage their possible parallelism. Any existing dependencies between operations must be satisfied. In addition, resource constraints (for example, a quantum computer may only have a limited number of RF-generators to power the control lines for the qubits) may add other limitations in the way operations can be parallelized.

At the time of writing of this thesis and the work done within, OpenQL is mostly used by a research group using superconducting qubits (see section 2.2.2) as their physical quantum hardware platform.

1.2. Problem definition

The OpenQL compiler often works like a black-box, with the developers and physicists not knowing for certain the efficiency and correctness of the compiled program. Tools are needed to assist the developers and users of OpenQL with verifying the correctness and efficiency of the compiled solutions. This work focuses on developing three different visualization tools at different levels of the compilation process, a quantum circuit visualizer, a mapping graph visualizer and a qubit interaction graph generator:

- **Quantum circuit visualizer:** the final output of the compiler is a quantum circuit which gets run on the target machine. In order to help facilitate the development process of quantum circuits, a way to visualize the output of the compiler is required. The circuit can be visualized in two ways: an abstract representation of the gates and qubits as shown in Figure 1.3, and a physical representation of the operations (signals) to be sent to the target quantum processor. Note that this work will only develop the physical representation for superconducting quantum hardware. Superconducting qubits are controlled by three RF lines, the microwave, flux and readout lines, which control single-qubit gates, two-qubit gates and measurements respectively. Gates are represented as pulses on the control lines.

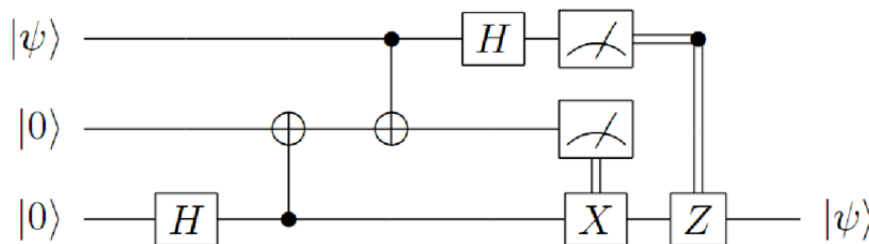


Figure 1.3: Example visualization of a quantum teleportation circuit [22].

- **Mapping graph visualizer:** the mapper pass maps the logical qubits to physical qubits, by way of a heuristic algorithm. This mapping is usually changed over time as qubits need to be moved to interact due to their limited connectivity. The mapping graph shows the path logical qubits take through the physical qubit grid during execution of the circuit. This provides insight into the way the mapper operates and can help to debug and improve the mapping process.
- **Qubit interaction graph:** in order to improve the mapping process it is very useful to see the amount and distribution of interactions between qubits, e.g. CNOT gates between qubits. The qubit interaction graph provides a way to visualize these interactions. This feature is also a subject of this work.

1.3. Thesis outline

The thesis is structured as follows:

- Chapter 2 discusses the basics of quantum computing, including the mathematics behind quantum bits and gates, superposition, entanglement, etc. The current approaches to quantum hardware are discussed and the chapter concludes with a discussion of the state of the art w.r.t. quantum computers.

-
- Chapter 3 opens with a short introduction on compiler theory and continues with a comparison between different existing compilers for quantum platforms. After this the inner workings of the relevant components of the OpenQL compiler are revealed, including a motivation for the work done in this thesis.
 - Chapter 4 describes the process of choosing an implementation of the visualizer (programming language, architecture, etc.).
 - Chapter 5 provides an overview of the work done on the visualization features, including the design choices that were made and the difficulties that were encountered.
 - Finally, Chapter 6 concludes the thesis with a discussion on future work for the visualizer of OpenQL and an overview of the work.

2

Quantum computing

In this chapter the background of quantum computing is discussed in a brief manner in section 2.1 to help the reader understand the concepts used in the rest of the thesis. A short overview of the different kind of physical implementations for quantum computers is given in section 2.2, and the chapter concludes with a discussion on the current state-of-the-art in quantum computing and the way forward from here in section 2.3.

2.1. Quantum computing

The concepts used in quantum computation are briefly discussed in this section, together with how these concepts are used to make computation possible.

2.1.1. A bit of history

For a long time classical physics was enough to describe the world around us. Maxwell told us how electromagnetic fields work and Newton's proverbial apple led to the theories that predict the movement of the planets in our solar system, including the amount of energy needed to escape Earth's gravity. However, at the start of the 20th century, it became apparent that these theories were not enough to accurately describe reality as it is. These models only provide us with rough answers that are suitable for everyday work, but not for the fine-grained analysis that is needed to understand the very small.

As Albert Einstein worked on the photoelectric effect, he realised that not the duration or intensity of light hitting a material determined the amount of electrons dislodged from it, but the *frequency* of the light did. In doing so he discovered that light was not only a continuous wave, but also a selection of particles at the same time (photons). This discovery, together with some others led to the development of quantum mechanics, where everything in the universe is quantized, and the world is described by a wave function with probability amplitudes. These facts are combined in the famous Schrödinger equation:

$$i\hbar \frac{d}{dt} |\phi(t)\rangle = \hat{H} |\phi(t)\rangle$$

This formula encapsulates how a closed system evolves with time, or put differently, it describes in what state a system will be after a certain period of time, given a set of initial conditions.

2.1.2. From bits to qubits

Classical computing uses bits that can either be 0 or 1 to make computations. Stringing operations together on collections of these bits allows computations to be done, which ultimately leads to computers as we have

them today. In this sense, the bit is the basic unit of information. Quantum computers use quantum bits as the basic unit of information. A quantum bit can be any two-level (or two-state) quantum system. Note that it is important that this system exhibits quantum mechanical behaviour. The 0 and the 1 state are then the two respective levels of that quantum system.

The attribute of a two-state quantum system that makes a qubit distinct from a normal bit (which is also a two-state system, but not quantum mechanical!) is that the qubit can be in a **superposition** of both states. In mathematical terms this means that a qubit is a linear superposition of two orthonormal basis vectors and it is written like so:

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

This is called the bra-ket notation. The basis vectors used here are $|0\rangle$ and $|1\rangle$, or equivalently $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$. The variables α and β are the probability amplitudes of the qubit and describe the probabilities of finding the qubit in either state after measurement. Measuring the state of a qubit will destroy the quantum mechanical properties of the physical realisation of the qubit and therefore collapses the qubit to either the 0 or the 1 state. Because the probability of finding the qubit in the 0 state is $|\alpha|^2$ and finding it in the 1 state is $|\beta|^2$, and probabilities add up to 1, the values of α and β are constrained by:

$$|\alpha|^2 + |\beta|^2 = 1$$

An easy way to visualize the quantum state of a two-level pure quantum system (pure in this context meaning a quantum state that cannot be written as a mixture of other quantum states) is the Bloch sphere representation, pictured in Figure 2.1. The three axes of the sphere constitute the X, Y and Z bases of the quantum system. Traditionally, the vertical axis is chosen to represent the Z basis, making the north and south pole of the sphere the $|0\rangle$ and $|1\rangle$ states respectively. In essence, the Bloch sphere makes it easy to visualize the position of the quantum state vector with respect to the three bases.

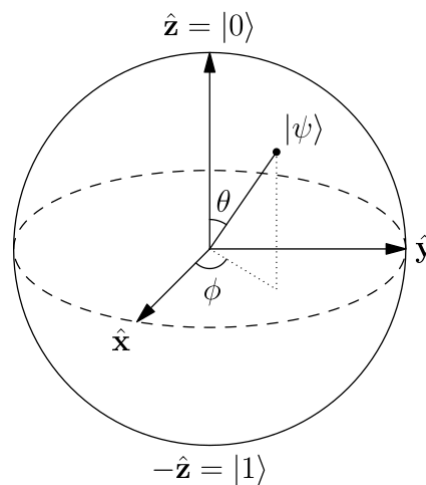


Figure 2.1: The Bloch sphere [7].

Multiple qubits can be denoted by vectors of length 2^n where n is the amount of qubits. For example, a description of two qubits in the 0 state would be $|00\rangle$, or equivalently $[0\ 0\ 0\ 1]^T$. Certain operations on two qubits allow for the creation of an *entangled* pair of qubits. This derives from the quantum mechanical property of entanglement, where two quantum systems are correlated in such a way that it is impossible to individually describe each system, i.e. only a description of *both* systems fully characterizes the two systems. The prototypical entangled state is the Bell state, given by the following formula:

$$\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

The mathematically inclined reader will note that it is impossible to separate this system into two individual systems that, when taken the tensor product of, will produce this entangled system. When one pair of an

entangled pair is measured, *both* pairs will collapse to a single state. Entanglement allows for applications that are not possible with classical bits, such as superdense coding, quantum teleportation and the creation of completely secure encryption algorithms.

2.1.3. Quantum gates

A computation on a classical computer is performed by logic circuits. The input of these circuits are bits and the output is determined by a Boolean logic (truth) table. This allows a circuit to perform basic AND, OR and NOT gates and putting multiple of these in sequence lets a computer perform complex algorithms. The quantum equivalent of these Boolean logic gates are quantum gates that are represented by unitary matrices. A unitary matrix U is a matrix whose conjugate transpose is also its own inverse: $U^*U = UU^* = I$. In mathematical terms, the effect of applying a quantum gate to a qubit is calculated by multiplying the state vector of the qubit with the unitary matrix of the gate. This is a reversible operation, therefore quantum gates are reversible, unlike classical Boolean logic.

Quantum gates can be performed on a single qubit. Notable examples of single qubit gates are the Pauli X , Y and Z gates, which perform rotations over the x , y and z axis of the Bloch sphere, and the Hadamard gate, which maps a base state $|0\rangle$ or $|1\rangle$ to $1/\sqrt{2}(|0\rangle + |1\rangle)$ and $1/\sqrt{2}(|0\rangle - |1\rangle)$ respectively. See Table 2.1 for more examples for single-qubit gates.








Gate name	X	Y	Z	H	S	T	I
Symbol							
Matrix	$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$	$\begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$	$\frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$	$\begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix}$

Table 2.1: Several notable single-qubit gates with their corresponding matrix representation. [21]

Quantum gates can also be performed on two qubits. Examples include the $CNOT$, $CPHASE$ or CZ gates, and $SWAP$ and \sqrt{SWAP} gates. $CNOT$ gates are used to entangle two qubits, and $SWAP$ gates are used to exchange the quantum state of two qubits. In addition, controlled two-qubit gates, such as the $CNOT$, consist of a control qubit and a target qubit. The control qubit is used to determine whether the corresponding operation should be performed or not on the target qubit. See Table 2.2 for some examples of the most important two-qubit gates.





Gate name	$CNOT$	$CPHASE$	$SWAP$	\sqrt{SWAP}
Symbol				
Matrix	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{1}{2}(1+i) & \frac{1}{2}(1+i) & 0 \\ 0 & \frac{1}{2}(1-i) & \frac{1}{2}(1-i) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

Table 2.2: Several notable two-qubit gates with their corresponding matrix representation. [21]

Quantum circuits can be built with qubits and gates and used to describe quantum algorithms and perform any kind of computation. An example of a quantum circuit is shown in Figure 1.3.

The output of a quantum circuit can be thought of as multiplying the matrix of each quantum gate together to compute the full unitary that describes the complete operation of the circuit. That unitary is then multiplied with the input quantum state vector to obtain the full output of the system.

Some relevant properties of a quantum circuit are the **cycle count**, which is the total amount of clock cycles the circuit takes to fully execute. Each gate has a duration (amount of time it takes to execute). In order to run the circuit, signals are sent to the hardware each cycle, where a single cycle is at least as long as the longest amount of time a gate takes to execute. Another useful property is the **circuit depth**, which is the number of time steps a circuit takes to execute (i.e. each time step is 1 unit).

2.2. Quantum hardware

To construct a functional quantum computer, several criteria have to be fulfilled. These criteria were first introduced by David DiVincenzo [12]. The following five criteria are necessary for quantum computation:

- A scalable system using properly characterized qubits with two well-defined states.
- The qubits should be able to be initialized into a reference state.
- The qubits should have long decoherence times.
- The operations that can be performed on the qubits should form a universal set.
- Each qubit's state should be measurable.

Note that a universal set of quantum gates is the minimum set of gates that can perform all possible operations of a quantum computer, i.e. any unitary operation can be expressed as a finite sequence of gates from that universal set.

An additional two criteria must be satisfied to allow for quantum communication:

- Stationary and flying qubits must be able to be interconverted.
- Flying qubits should be able to be transmitted without loss of state between two locations.

Several notable technologies for which all of the DiVincenzo criteria hold (or are thought to hold) will be discussed in this section. This is not an exhaustive list, it is only meant to give the reader an impression of the many different ways in which a physical quantum computer can be realised.

2.2.1. Robustness of quantum systems

Systems which exhibit the useful quantum properties needed for quantum computation, such as superposition and entanglement, are notoriously easy to disturb. Any interference from outside will cause a qubit to collapse to one of its two states, which is called decoherence. It is the process where the qubit gets entangled with the outside environment and loses its quantum information.

Notably, we distinguish between two different decoherence times, the T1 and T2 times. The T1 time is the time it takes for a qubit to relax from its excited $|1\rangle$ state to its ground $|0\rangle$ state. The T2 time is the time it takes for the qubit to acquire a phase difference from the environment.

The possibility of a qubit losing its coherence necessitates the application of error correction to perform fault-tolerant quantum computation. In a quantum error correction scheme, more than one physical qubits together form one logical qubit. In addition, using error syndrome measurements certain types of errors can then be detected and corrected.

In general, the performance of a quantum computer implemented in a specific hardware type can be characterized by the following variables [25]:

- **Coherence time:** the duration in which the state of the qubit remains valid. When a qubit experiences decoherence its state is no longer valid. Important parameters are the T1 and T2 times described above.
- **Gate latency:** the amount of time executing a quantum gate takes. The shorter the gate latency, the more operations that can be performed on a qubit before it decoheres.
- **Gate fidelity:** the probability that a quantum gate executes successfully, without experiencing errors. Gate fidelity for quantum computing is much lower than for classical computing. This is part of why quantum error correction is necessary.

- **Qubit connectivity:** how qubits are arranged and interconnected. Quantum processors usually have limited connectivity that makes it necessary to transport quantum states, or even move the qubits themselves (depending on the physical technology). This transportation decreases the chance of success of the quantum algorithm.

2.2.2. Realisation of qubits

Trapped ion quantum computing

Trapped ion quantum computers [5] work by manipulating ions in such a way that a two-level quantum state is achieved, which constitutes the qubit in this system. An ion is a charged atom or molecule, or in other words, an atom or molecule that has an unequal number of protons and electrons. They are ubiquitous in nature, easily created and responsible for many everyday phenomena. Because an ion is electrically charged, they can be trapped inside an electromagnetic field.

The electronic states of a trapped ion are used to construct the two-level quantum system necessary to form a qubit. Using lasers with specific frequencies single qubit gates can be performed on individual ions. Multi-qubit gates can be performed by coupling the internal ion state to its external motion state, and then using shared motional states to exchange quantum information between different qubits to achieve a multi-qubit gate. Figure 2.2 shows several configurations of magnesium ions in an ion trap.

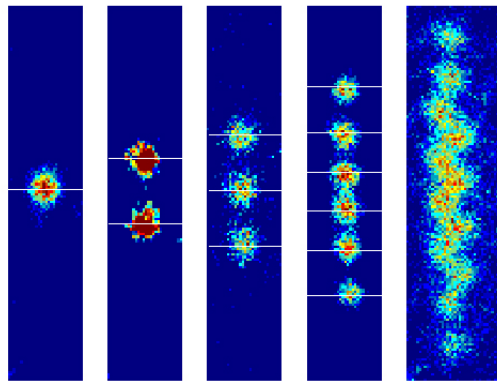


Figure 2.2: Trapped magnesium ions in a planar ion trap produced by NIST (National Institute for Standards and Technology) [8].

Notable advantages of using trapped ions as qubits are their very long coherence times, in the order of seconds or even another order of magnitude higher in some cases. Another benefit is the very high fidelity with which operations can be applied to the qubits. The most notable disadvantage of using trapped ions as qubits is that coming up with a design to scale the amount of ions to very high amounts, while still retaining control over each individual ion is very difficult.

Quantum dot quantum computing

Quantum dots are artificially constructed particles inside of a semiconducting material. They effectively behave as if they are an artificial atom, having distinctly defined electronic states. Electrons can be trapped inside of a quantum dot, after which the spin property of the electron can be used as a two-level quantum system, constituting a qubit [19].

Single qubit gates can be achieved by applying a local magnetic field to an individual quantum dot, thereby changing the spin of the enclosed electron. Two-qubit gates in this model can be performed by allowing the electrons inside two neighbouring quantum dots to perform quantum tunneling, thereby coupling the two quantum dots and enabling the qubits to perform a two-qubit operation. Quantum tunneling is the process whereby the wave function of a particle leaks to the other side of a normally insurmountable barrier. Readout is done by spin-to-charge conversion.

Advantages of using electron spin quantum dots include very long relaxation times (T_1 decoherence), easy application of gates through the use of localized magnetic fields and high gate fidelity. Disadvantages are that quantum dots are that two-qubit gates have relatively low gate fidelity, due to charge noise induced by electric fluctuations [26].

Superconducting quantum computing

A promising way to implement quantum hardware is to use superconducting qubits. These qubits can be implemented in a few ways, but in this work we will focus on transmon qubits, which are a form of superconducting charge qubits [17]. Superconducting qubits are cheap to manufacture, and are easy to control. However, superconducting quantum hardware needs to operate at temperatures close to absolute zero, which requires specialized fridges.

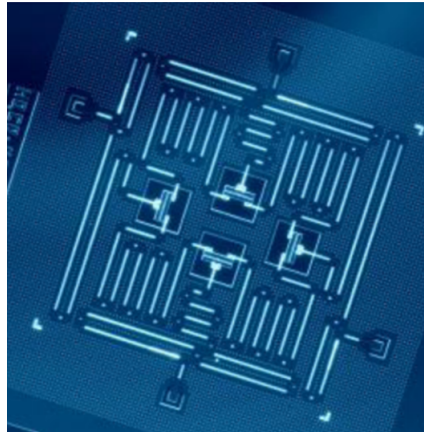


Figure 2.3: A chip with four transmon qubits, fabricated by IBM [6].

The circuit implementing this type of qubit consists of two superconducting islands with two Josephson junctions coupling those islands. A Josephson junction is formed by placing two superconducting wires next to each other in such a way that electrons are allowed to tunnel through the barrier, thereby inducing a current. The ratio between the Josephson energy and the charging energy, E_J/E_C , in the transmon qubit is responsible for its insensitivity to charge noise. The anharmonicity of the energy levels of the quantum states creates the two-level quantum system needed to implement a proper qubit. Specifically shaped microwave pulses can then be used to excite the system in one of the two quantum states of a qubit.

2.3. The progress of quantum computing

Quantum computing has moved beyond the purely experimental stage with quantum chips just having a handful of functional qubits. Current state-of-the-art quantum computers possess several dozens of qubits. With qubit numbers for quantum computers now in the 50 to 100 range, quantum computing has entered the noisy intermediate-scale quantum (NISQ) era [25]. The NISQ era is a significant milestone because these quantum devices have too many qubits to be simulated by classical computing using brute force. However, because these devices still only have a small amount of physical qubits, their practical use remains limited. Researchers are trying to find applications beyond the experimental and engineering relevant applications these NISQ devices provide [23].

In order for a quantum computer to have an insurmountable advantage over a classical computer, quantum advantage needs to be demonstrated. Quantum advantage is achieved when a quantum computer is able to solve a problem that when solved on a classical computer would be intractable, i.e. a problem that a quantum computer could solve with an exponential speedup over solving that problem on a classical computer. To achieve this goal, a quantum computer with a notable amount of qubits has to be built. Significant progress has been made towards demonstrating quantum advantage. In 2019 Google claimed that it had achieved quantum advantage [3]. However, these results are disputed by various sources. Nevertheless, even if disre-

garding Google's claim, it is reasonable to assume that quantum advantage will be achieved within the next few years.

Current notable quantum computers include Google's Sycamore computer, with 53 superconducting qubits. In late December 2020 a team of Chinese researchers working for the University of Science and Technology of China had built a photon-based quantum computer with 76 qubits called Jiuzhang [32]. In this computer the quantum information is encoded in the position and polarization of the photons. It managed to perform the boson-sampling problem [1] in 200 seconds. They claimed solving the same problem on a state-of-the-art classical supercomputer would take 2.5 billion years. IBM's flagship quantum computer currently has 65 qubits. Both Google and IBM plan to have a one-million qubit quantum computer within 10 years. Whether this is feasible remains to be seen.

To solve practical problems that have a real application, a quantum computer would need millions of qubits. Because quantum computers are inherently noisy, error-correcting schemes will require many additional physical qubits to form one logical qubit that has a reasonably low error rate. Hundreds of thousands of those logical qubits are needed to implement a quantum algorithm with real-world application. Once these qubit numbers are reached, internet encryption as we know it will be broken by Shor's algorithm (an algorithm for efficiently factoring prime numbers, the factoring problem being difficult is the basis for many encryption algorithms). Many quantum mechanical phenomena will be simulatable and new molecules for use in e.g. material and medicinal science will be easily simulated. Many technical hurdles will have to be solved still before realizing the full potential of quantum computing, but the first steps have been made.

3

The OpenQL platform

In this chapter a brief overview of the most important current quantum computation platforms is given in section 3.1. The OpenQL platform is discussed in section 3.2.

3.1. Quantum computation platforms

With the remarkable progress of quantum devices in the last few years, full-stack quantum computing systems already exist for which quantum software frameworks have been developed. In this section several of these platforms will be discussed. The visualization features of the mentioned platforms will be discussed in section 4.4.

3.1.1. Qiskit

Qiskit [2] is an open-source platform developed by IBM for use with their cloud quantum computing service, called IBM Q Experience. It uses Python as the programming interface, but it can also accept OpenQASM (Open Quantum Assembly language) as input. It consists of four modules, each providing a specific function:

- Qiskit Terra is the main foundation of Qiskit, providing the tools necessary to describe a quantum program. It functions as the compiler of Qiskit programs, handling optimizing and transforming the input program to the specifications of the backend.
- Qiskit Aer provides a framework for using Qiskit with simulators. It also allows for the use of realistic noise models with simulations.
- Qiskit Ignis lets the user verify the performance of a circuit, characterize the error model for a circuit and provides measures to help mitigate the effect of errors on a circuit.
- Qiskit Aqua gives support for domain-specific algorithm writing, such as chemistry and finance.

3.1.2. Cirq

Cirq [10] is an open-source Python library for quantum computing developed by Google. Cirq can be used with real quantum hardware or with a built-in simulator.

Qubits in Cirq can be defined as a set of names, a linear array of numbers or connected on a grid topology. Cirq also defines a Device structure that can be used to store a topology of qubits together with rules on how they can interact and be accessed, emulating real quantum hardware. Cirq differentiates between gates and operations. A gate is a traditional gate that be applied to a qubit, while an operation is a gate that is applied

to a specific set of qubits. A clock cycle is called a moment, and each moment contains a set of operations. Together, all the moments in a program form the quantum circuit.

With a fully defined circuit, the results can be obtained by running it on a simulator or by running it any number of times on a real quantum device.

Cirq further supports the use of unitary matrices for gates and the decomposition of complex gates into a series of simpler gates, or, if a given device (the set of rules governing the constraints on qubit accessibility and connectivity) does not natively support the used quantum gates into a series of supported gates.

3.1.3. Quantum Development Kit

The Quantum Development Kit [20] is a development kit for quantum programs written in the Q# quantum programming language, developed by Microsoft. It uses a hybrid quantum-classical programming model in which Q# is used to program the quantum parts of the program, and C# is used when classical computations are done.

The QDK contains Q# libraries, simulators and support for integrating Q# with other programming languages, Python and .NET languages like C#. The QDK does not provide support for running a program written in Q# on real quantum hardware however. A program written in Q# can either be compiled as a standalone program to be ran on the connected quantum simulator, or as a quantum accelerator, i.e. used inside of a host language which can then use the results of the quantum program for further processing.

3.1.4. Forest

The Forest software development kit [28] is a quantum programming platform developed by Rigetti. It consists of PyQuil, a Python library for creating quantum programs with the Quil (quantum instruction language), quilc - the compiler for Quil and the Quantum Virtual Machine (QVM) which provides a simulator for the programs produced in Quil.

A program written in Quil can be executed in the cloud on the QVM or on a QPU (quantum processing unit) provided by Rigetti.

3.1.5. Differences between platforms

Each of the aforementioned quantum programming platforms more or less do the same, but there do exist differences between them. Several differences will be discussed below [18].

Language differences: A distinction in the abstraction level of the quantum programming language can be made. Cirq, OpenQASM and Quil are all quantum instruction languages, somewhat equivalent with assembly languages on classical computers (although not quite the same). PyQuil, Q# and Qiskit's Python wrapper are higher-level quantum programming languages, more similar to contemporary classical high-level programming languages. Of these three, Q# resembles a classical high-level programming language the most, due to its relation with C#.

Quantum hardware connectivity: Qiskit, Cirq and Forest all have their own corresponding quantum hardware platforms that their programs can be executed on. Microsoft's QDK does not provide such a connection to a quantum hardware platform.

Quantum simulator: The Forest quantum simulator runs in the cloud and allows the user to apply sophisticated noise models. The Qiskit simulator has several different modes, which differ in simulation strategy. The unitary simulation simulates the full unitary matrix of the quantum circuit, but because the matrices grow exponentially with the amount of qubits, this simulation mode consumes a lot of memory. The other simulation mode only stores the state vectors and therefore uses much less memory. QDK contains several simulators, with the most complete one being the full state simulator. This simulator can run full quantum algorithms, although it is limited to programs with about 30 qubits. QDK also has two resource-estimation simulators, which can be used to estimate the amount of resources needed to run a given quantum program.

Both support up to thousands of qubits. Lastly, the QDK also has a Toffoli simulator, which runs on a limited set of gates, X , $CNOT$ and multi-controlled X gates. Because of this limited set of gates, this simulator supports millions of qubits. The drawback is that it is not a proper full quantum circuit simulator.

3.2. OpenQL

In this section a general overview of the OpenQL quantum platform [16] is given, together with a more in depth look at the relevant components for this work.

3.2.1. A platform for quantum computation

Similar to the platforms discussed in section 3.1, OpenQL provides a high-level framework for the development of quantum programs. OpenQL accepts programs written in Python or C++, as well as cQASM (an assembly-like language for programming a quantum computer). OpenQL takes care of the compilation of such a quantum program into a form that the connected back-end can understand. This means that OpenQL has to know what type of instructions the back-end supports. Not all quantum gates are natively implementable by all types of quantum hardware. Instructions that do not have a direct physical counterpart have to be decomposed into a sequence of native instructions that together perform the desired operation. This is always possible if the hardware supports a universal set of quantum gates, as every arbitrary unitary quantum gate can be decomposed into a series of gates from this universal set. Figure 3.1 provides an overview of the general compiler and framework structure of OpenQL.

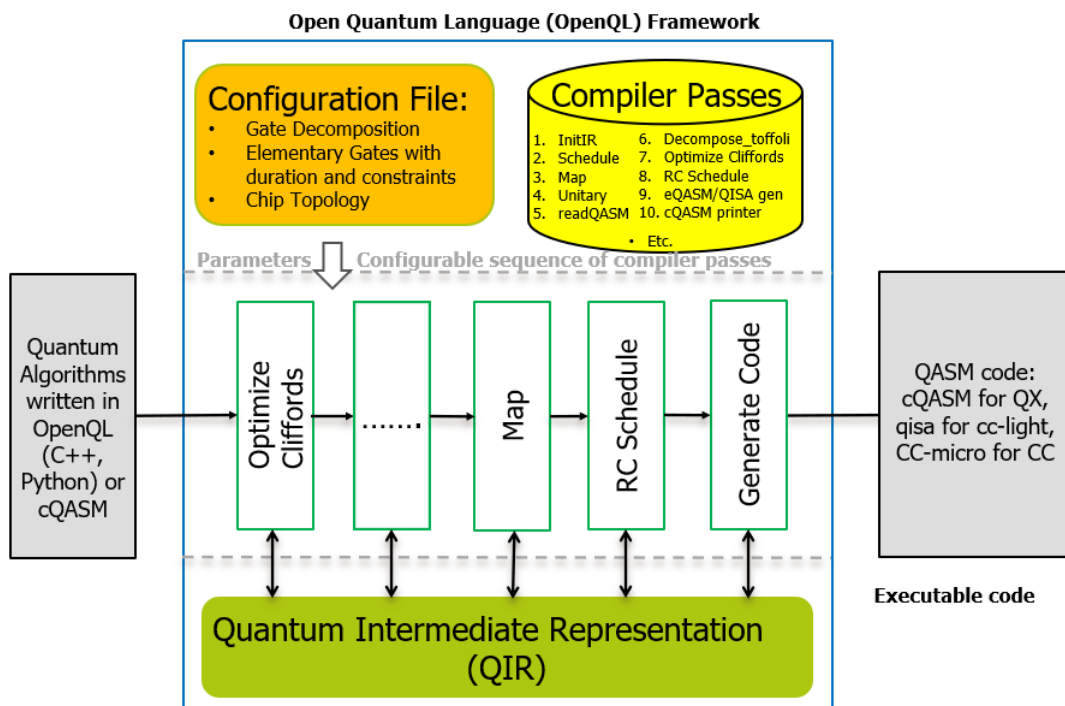


Figure 3.1: The OpenQL compiler framework [29].

A circuit in OpenQL consists of a program object, that takes a platform as input. The platform describes everything that OpenQL needs to know to perform the compilation. A program object can have one or more kernels. A kernel is a vector of gates and optional classical instructions representing one quantum circuit. A kernel can also be responsible for the control flow of the program, i.e. it can transfer control from one kernel to another, possibly based on the result of some measurement.

OpenQL needs to know about several parameters of the back-end in order to do its job:

- The hardware settings, like the time it takes to complete one clock cycle and several hardware dependent settings.
- The topology of the qubit layout. The mapper needs this information to calculate where *SWAP* gates should be included to bring qubits next to each other in order to perform two-qubit gates (remember that two-qubit gates can often only be performed on neighbouring qubits).
- The resources that are available to the quantum hardware, e.g. microwave drives that provide the pulses for addressing qubits. This has a classical analogue in the sense of classical hardware also having resources, like adders and dividers.
- The available instructions on the back-end. The parameters for each instruction include the duration (in time), latency (to account for electronic control time differences) and the qubits the instruction operates on (note that physical control commands for gates can differ per qubit, e.g. qubit 1 and qubit 2 require different microwave pulse shapes to perform the same gate)
- Any manual decomposition rules. The programmer can explicitly define decomposition rules.

Not all of these are mandatory, e.g. the topology and resources sections can be omitted if the user wishes to compile a quantum program without constraints on the topology or number and type of resources available.

The OpenQL compiler takes an external program and transforms it to a program in the desired output format, while taking a set of predefined rules into account. To do this, the compiler follows a sequence of steps, or passes, that each take the output of the previous pass, transform it and output it to the next pass. In order to facilitate the interfacing between each of the passes, a common representation of the quantum program being compiled is used: the internal representation (IR). This representation contains all the information necessary to represent the quantum circuit in each stage of compilation.

Relevant compiler passes include:

- Optimizer - optimizes the circuit by replacing gate sequences by shorter sequences that result in the same operation.
- Scheduler - schedules the gates in the circuit in a way that adheres to the constraints imposed by the gate dependency graph, gate durations and several other conditions.
- Backend compiler - transforms the final IR into instructions usable by the backend
- Report statistics - a summary of the IR is written to the file, including number of qubits, kernels, one and two-qubit gates, used qubits and more.
- Mapper - maps the gates in the circuit in such a way that two-qubit gates can be performed by inserting *SWAP* gates in places to bring virtual (logical) qubits together. The mapper employs an exact initial placement algorithm, and a heuristic algorithm to map the circuit.
- Resource constrained scheduler - schedules the gates in the circuit in such a way that also adheres to the resource constraints imposed by the hardware platform

The mapper and scheduler passes are particularly relevant for this work and will be expanded upon in the next sections.

3.2.2. Mapper

Physical quantum hardware often imposes constraints on the connectivity between qubits. These constraints limit the application of two-qubit gates between qubits in the topology. Mapping is the process of moving the logical state of qubits next to each other so that two-qubit gates can be performed on those qubits. Depending on the amount and operands of the two-qubit gates in the circuit, the mapping process can become quite difficult and time-consuming. Figure 3.2 shows a minimal example of a circuit that needs to be mapped based on the qubit connectivity constraints.

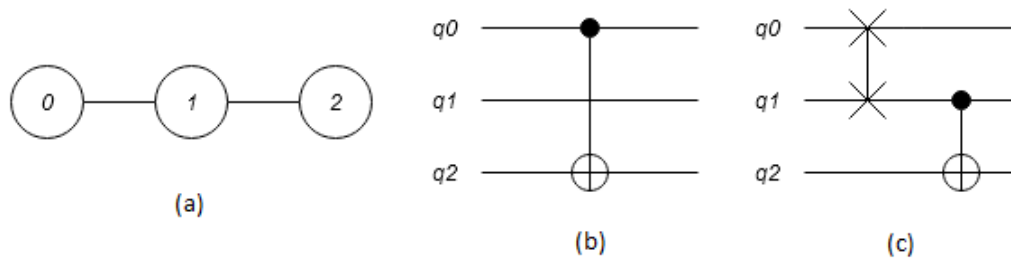


Figure 3.2: (a) The physical qubit topology. Note the lack of a direct connection between qubit 0 and qubit 2. (b) The quantum circuit, consisting of a *CNOT* between logical qubits 0 and 2. This gate is impossible to execute with a direct mapping of each logical qubit to the physical qubit with the same index, since physical qubits 0 and 2 are not connected. (c) The same circuit, but a *SWAP* has been inserted to swap the states of physical qubits 0 and 1, therefore making the *CNOT* between physical qubits 1 and 2 logically a *CNOT* between the states of logical qubits 0 and 2.

Each logical qubit gets mapped to a physical qubit and this mapping is computed during the mapping process. The mapping is computed by an exact initial placement method, that minimizes the amount of swaps necessary, or more formally, it minimizes the sum of the distances between the operands of all two-qubit gates of the circuit. This method uses a Mixed Integer Linear Programming approach to find the most optimal result. However, the time-complexity of this method increases exponentially with the circuit size, so it is infeasible for larger circuits.

If initial placement fails, or is not enabled, the mapper employs a heuristic algorithm to find a mapping that is (hopefully) close to the optimal solution. This method walks over the circuit and insert swaps before each encountered two-qubit gate. Many options exist to control the parameters of this algorithm. At best it has linear time complexity, but with the most advanced options enabled its complexity becomes cubic.

The visualizer project expands the mapper with a method to keep track of the virtual to physical qubit mapping throughout the compilation. Normally this virtual to physical mapping information is discarded at the end of the mapper because it is not useful any longer. However, this mapping information is necessary in order to visualize the mapping graph, both immediately after having done the mapping and later after the final scheduling, to also be able to visualize the mapping of the circuit then.

3.2.3. Scheduler

Gates in a quantum circuit, just like in a classical circuit, have dependencies on the outputs of other gates. A gate with a direct or transitive dependency cannot be executed until the gates that it is directly or indirectly dependent on have finished execution. The process of ordering operations in such a manner that for each operation, all operations it depends on are executed before that operation, is called scheduling.

The dependencies between gates in the gate dependency graph are not necessarily the only constraints imposed upon the scheduling process. When relevant, resource constraints also have to be taken into account when scheduling gates. A resource can be anything that a gate requires for execution of which there is a limited quantity. E.g. for superconducting qubits an example of a resource is a microwave drive that powers the pulses that address the qubits. One such a drive can be responsible for multiple qubits, therefore those qubits cannot be operated upon in parallel and the scheduler must schedule gates on those qubits in different cycles.

The scheduler has two different methods by which the gates can be scheduled: ASAP and ALAP. ASAP stands for As Soon As Possible and means that a gate will be scheduled as soon as possible. ALAP stands for As Late As Possible and means that a gate will be scheduled as late possible. The difference between the two is shown in Figure 3.3.

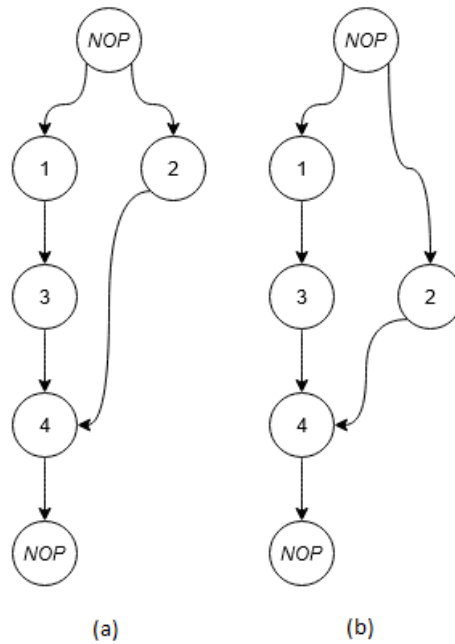


Figure 3.3: (a) ASAP scheduling. Operation 2 is started as soon as possible. (b) ALAP scheduling. Operation 2 is started in the last cycle before operation 4 is executed. It cannot be started any later because operation 4 depends on the result of operation 2.

3.2.4. Motivation for the visualization tools

Three distinct visualization tools will be developed for OpenQL. The list below lists the three tools, together with the motivation for developing that specific tool:

- **Circuit visualizer:** The experiments of the physicists using OpenQL will be improved by the development of a circuit visualizer, showing both the abstract visualization and pulse visualization of the OpenQL program. Having a concrete visualization (abstract and pulse representation) of the output circuit that the OpenQL compiler produces for a given quantum program is beneficial for the design and fine-tuning of experiments done on real quantum hardware.
- **Mapping graph visualizer:** The mapper and scheduler mostly work as a black box, it is not always clear what decisions they make and why. In order to improve the visibility of the decision making process of the mapper, a tool will be developed in this work called the mapping graph. The mapping graph shows the logical to physical qubit mapping at each cycle of the circuit, thereby allowing the developer and/or user to verify the paths each logical qubit takes through the physical qubit topology and ascertain its effectiveness, and thereby the effectiveness of the mapper.
- **Qubit interaction graph visualizer:** A qubit interaction graph will help visualize the interactions between the qubits in a quantum program. As with the above two visualization tools, this will help the users and developers of OpenQL gain insight into the compiled output circuits that OpenQL generates.

3.2.5. Future work

Much work remains to be done on OpenQL. A full rewrite of the compiler pass structure is now in progress at the time of writing, further cementing the modularization of the compiler passes and providing a more user-friendly way to write and add new passes.

Furthermore, work is in progress on implementing conditional execution of quantum gates based on the outcomes of measurements earlier in the quantum circuit. In addition, support for classical instructions in an OpenQL program has been added.

4

Choosing an implementation for the visualizer

The OpenQL compiler is written in C++, with a Python wrapper (SWIG [11]) that makes the compiler callable from Python code. To accomplish visualizing the circuit both after and during the compiler passes, the visualizer will need access to the compiler's internal representation of the circuit state. For the mapping graph the visualizer needs access to the virtual (or logical) to physical qubit mapping. Given this information, the two most logical choices for the implementation language of the visualizer are C++ and Python.

The advantages and disadvantages of both approaches will be discussed in this chapter. In section 4.1 the differences between the C++ and Python programming languages will be discussed. Section 4.2 outlines the requirements for the visualizer, both general requirements and requirements from both the OpenQL developers and the physicists who actively use OpenQL. Section 4.3 deals with the relevant existing OpenQL codebase. Section 4.4 provides a comparison between the visualizers of other quantum platforms with the intent to determine whether code reuse is possible. Section 4.6 describes the process of choosing an image library. Finally, in section 4.7 the chapter is concluded.

4.1. Differences between C++ and Python

Python is a language in which it is easy to quickly prototype some functionality. This comes at the price of reduced performance and no direct control over the hardware. C++ gives the programmer much more control over the hardware, including explicit memory management. C++ gets compiled to native code for the chosen computer architecture which makes it much faster than C++. Python, on the other hand, is an interpreted language which adds an additional step between the program and execution on the target platform. That intermediary layer translates the Python byte code in real time to native code of the platform it is running on.

4.1.1. Object oriented

Both C++ and Python are object oriented languages, which means that they provide support for the object-oriented paradigms. An object-oriented program consists of classes, where a class is a coherent collection of data. The class *encapsulates* the data, shielding it from external influence. The class provides functions (or methods) to act upon the data the object contains. These methods are the class' interface to the outside world, again shielding the interior of the class from external influence. This also shields users of the class from changes to the internal workings of the class. As long as the class' public interface remains intact, users can be guaranteed compatibility.

A class can be instantiated into an object, i.e. the class provides the blueprint for the object. The properties of these objects as defined in the class they were created from are then filled with actual values. Classes can

inherit properties and functionality from a base class, thereby creating a hierarchy of classes, cutting down on code duplication. The objects in an object-oriented program interact with each other through the use of their methods, thereby making up the program.

Note that while C++ is primarily an object-oriented language, it still retains all of the procedural features of C (the language it evolved from).

4.1.2. Typing

Python is a dynamically typed language. Variables have no defined type at time of compilation. Note that compiling for Python means transforming the program to byte code (instructions) for a virtual machine (interpreter) that converts the byte code to the native code of the platform the interpreter runs on at run time. Incompatible variable types are thus only discovered during a run of a program. In contrast, each variable in C++ has a clear, defined type at compile time, allowing the compiler to fail compilation if variable types are not compatible.

Both approaches have their advantages and disadvantages. Python code is easy to write, no need to declare the type of each variable. This makes the program more like a human language and thus more readable. However, this also makes a Python program more prone to (subtle) bugs. C++ programs do not have this problem because all variables are required to have a defined type. The compiler checks this at compile-time. However, having to define a type for each variable slows down programming and therefore development time.

4.1.3. Memory control

In Python, unused objects are collected and discarded by the garbage collector. When programming in C++, memory must be explicitly allocated and later on manually deallocated when the program no longer needs it. This can lead to all kinds of memory leaks when the programmer forgets to free up memory that is no longer used. On the other hand, this provides absolute control for the programmer, allowing the programmer to decide at which moment memory is no longer needed. In Python the garbage collector has non-deterministic behaviour, i.e. the programmer has no control over when unused memory is freed up, introducing small 'hick-ups' in the performance of a program. Performance critical programs therefore benefit from having explicit memory control. In case memory is limited, having manual control over the memory is also beneficial.

4.1.4. Conclusion

In the end both Python and C++ have applications for which they are better than the other. In Python a program can be rapidly prototyped due to a less strict syntax, whereas in C++ you have more performance and more control over the hardware. As such it is not immediately clear which programming language to choose without knowing the visualizer requirements and design considerations. In the next sections we will examine both languages with respect to the requirements and design considerations.

4.2. Requirements

This section provides a list of the different requirements from the different stakeholders interested in, or using OpenQL.

4.2.1. General requirements

These requirements are general requirements that are necessary or otherwise useful for all stakeholders.

- The visualizer should be able to visualize the quantum circuit, i.e. visualize the IR (the Internal Representation, which is an easily manipulable representation of the quantum circuit that is being compiled). The visualizable components of the circuit are:

- qubit wires
 - classical wires (these should be able to be collapsed to a single 'thick' wire with connections indicating the classical bit being addressed)
 - single qubit gates
 - multiple qubit gates
 - measurements
 - measurement result wires
- The visualizer should adhere to the timing information provided by the scheduler, provided the scheduler pass has been performed already.

4.2.2. Requirements for experiments

These requirements pertain specifically to the experiments of the DiCarlo lab, which uses OpenQL to program, control and operate their superconducting quantum processors.

- The visualization should be zoomable and scrollable. Experimental circuits are often very large and viewing them in an unzoomed image is infeasible.
- A toggle should be provided to turn circuit components on or off: (either interactive or as parameter list to static image)
 - wires
 - gates
 - measurements

Additionally, or alternatively, the visualization could be divided into sub-circuits and allow sub-circuits to be collapsed into a single line. This to allow easy viewing of smaller parts of the circuits because the experimental circuits are often very large.

- Visualization should provide support for visualizing the waveform equivalent of each gate (specific quantum technology to the superconducting qubits used in the DiCarlo lab). This probably requires a 1-to-1 correspondence between gates and waveform pulses. This correspondence would be described in the platform configuration file, just like the available gates are described in the platform configuration file. The waveforms are dependant on both the qubit being addressed and the gate being performed.

4.2.3. Requirements for developers of OpenQL

These requirements are specifically aimed at the interests of the developers of OpenQL.

- The visualizer should be able to take a Directed Acyclic Graph (DAG - used for representing the control flow of an OpenQL program) as input and present its structure as an image.
- The visualizer should be able to take the compiler's internal mapper state as input and visualize it. This enables the developer to easily gain insight into the way the mapper works.
- The visualizer should provide support for visualizing the quantum circuit after each compiler pass. (Requires compiler passes to be modular, either by calling the visualizer after each pass manually or providing information about which passes should be visualized beforehand)

The implementation of these requirements will be explored in Chapter 5.

4.3. Existing code base

The OpenQL compiler is written in C++. The Python part of the compiler consists of a wrapper that allows the compiler to be called from Python. This wrapper is implemented using SWIG [11]. SWIG automatically generates the wrapper code necessary to connect a Python function call with a C/C++ function declaration. It does this by using a user-specified interface file that connects to a C/C++ header file and instructs SWIG where to find the function declarations.

The C++ program maintains the state of the quantum circuit during compilation in the internal representation (IR). This IR is currently not available to the Python wrapper. If Python were to be chosen to implement the visualizer, additional functionality to expose the IR to Python from C++ would need to be written. Furthermore, providing access to different structures used by the different compiler passes to visualize them would introduce even more work, e.g. the mapping graph visualizer would need access to the internal state of the mapper during the mapper process. This functionality does not currently exist. Implementing the visualizer in C++ would not have these problems, every structure would immediately be available to the implementation, although additional code would need to be written to make some internal structures available to the visualizer.

Implementing the visualizer in C++ would mean that future extension of the functionality of the visualizer beyond the scope of this project would require access to the C++ code, as well as knowledge of C++. Because C++ is a more difficult language to program in, this would impose additional constraints on the future developer(s) working on the visualizer. This is not a problem for a Python implementation, as that could be designed in a way to be easily extendable for anyone with Python knowledge, using well-known image library packages. In addition, many researchers who are not necessarily programmers often do have some knowledge of Python programming, as Python usage is ubiquitous in the academic world.

4.4. Comparison of existing visualizers

Many existing quantum platforms have their own visualizers built-in. In this section the features of several of these visualizers will be discussed.

4.4.1. Qiskit

The Qiskit (section 3.1.1) visualizer provides support for plotting the output distribution of a quantum circuit in several different ways, including a histogram and a Bloch sphere drawer. Qiskit is also able to visualize the gate map, error map and final physical gate layout of a quantum circuit. Lastly, it provides a circuit and DAG (directed acyclic graph) drawer to visualize the compiled quantum circuit and the operation dependencies respectively.

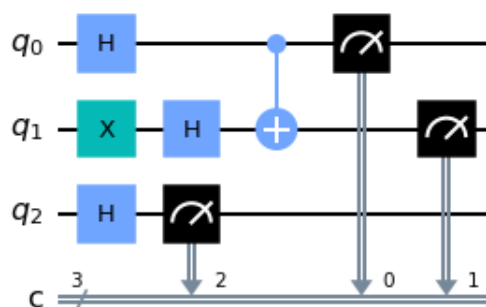


Figure 4.1: Example circuit visualization from Qiskit's visualizer [24].

The circuit visualizer is of greatest interest for the purposes of this project. It has several different parameters

for controlling the way in which the circuit is drawn. Scale of the image, reverse bit order, plotting of barriers (which subdivide a circuit by imposing the requirement that all gates before the barrier must execute before the gates after the barrier), whether to display idle wires or not and several other layout-related parameters. Furthermore, the circuit drawer allows the circuit to be drawn as ASCII, a Matplotlib figure, a LaTeX image or directly produce LaTeX source code. See Figure 4.1 for an example of the Qiskit visualizer.

4.4.2. PyQuil

The PyQuil (section 3.1.4) visualizer has comparatively few features. It can only draw the quantum circuit and has no support for drawing output distributions or internal compiler representations of the circuit during the different compiler passes. Its visualization parameters include being able to show indexed qubits between the used indices and writing controlled rotations in compact forms.

4.4.3. Cirq

Cirq (section 3.1.2) does not have a proper quantum circuit visualizer. Cirq is only able to print a simple text-based visualization of a circuit. It does have a heatmap feature, showing a user-specified value for some variable for each qubit.

4.4.4. QDK

The Quantum Development Kit (section 3.1.3) does not have a visualizer.

4.4.5. QX simulator IDE

OpenQL and QX simulator IDE developed by Aritra Sarkar [27]. A simple IDE developed by Aritra Sarkar for the TU Delft, combining writing OpenQL programs with simulation on QX-Sim. The visualizer works by parsing each line of the OpenQL code and if a quantum gate is present, drawing the gate in a column. This approach is quite different from drawing a circuit from the IR of OpenQL, which is a graph and far more dynamic than line by line parsing. The IDE also seems to have limited gate support (even though many gates are defined in a data structure of the code, only a few are actually implemented), nor can it draw multiple gates in a single time step. Drawing multiple gates per column should be easier when the IR is used to parse the circuit.

4.5. Code reuse

Depending on which programming language is chosen to implement the OpenQL visualizer, code could be reused from existing open-source visualizers (if the licensing allows it). Note that no open-source quantum circuit visualizers have been found that use C++ as implementation language. Therefore, code reuse would only be possible if the chosen implementation language for the OpenQL visualizer is Python. If code can efficiently be reused this would greatly speed up initial development of the visualizer. The decision to reuse code therefore definitively impacts the choice of programming language.

4.6. Image library

The task of generating an image from code involves a lot of work. In order to speed up this process, an image library will be used that will handle all the overhead needed to create and optionally display the generated image. In this section candidate image libraries are considered for both Python and C++. These libraries were selected on being stable and up-to-date. For each library several parameters will be checked and compared:

- the license (which has to be compatible with OpenQL's license)
- whether the library has built-in image manipulation, image manipulation here meaning being able to

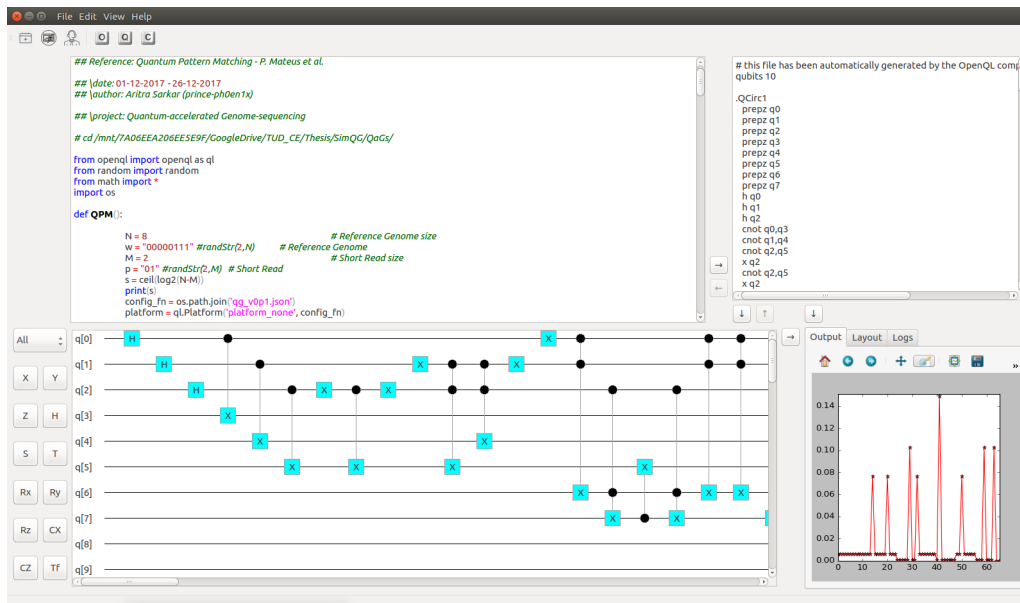


Figure 4.2: Image of QPM IDE [27].

draw lines on the image

- whether the library is able to read and write images to and from disk
- what platforms the library runs on (note: only relevant for C++ libraries)
- the footprint on the library in terms of runtime overhead, memory requirements and size on disk

4.6.1. Python image libraries

For Python, realistically only one package can be considered for the requirements of the visualizer: Pillow [13]. It is licensed under the HPND license, can read and write images, but unfortunately has no built-in image manipulation. Image manipulation would need to be done through manipulation of the individual pixels, which means that generic routines would need to be written to draw shapes and lines.

4.6.2. C++ image libraries

See Table 4.1 for an overview of several notable C++ image libraries.

Below are the license compatibilities per library. Note that OpenQL is licensed under the Apache 2.0 license:

- **ImageMagick**: compatible. It does not adhere to a standard license, but the terms of the license are compatible with the Apache 2.0 license.
- **stb_image**: compatible. It is public domain, and can therefore be used in any project.
- **CImg**: compatible. The header file is licensed under the CeCILL-C license, instead of the full CeCILL license and is compatible with OpenQL's license. The rest of the CImg project cannot be used in OpenQL but this is not a project, as only the header will be included.
- **FreeImage**: incompatible. A GPLv3 licensed project cannot be included in an Apache 2.0 licensed project. In the reverse situation the licenses are compatible, but unfortunately the situation is not reversed.
- **DevIL**: compatible. A library licensed under the LGPLv2.1 license can be included in an Apache 2.0 project without requiring that project to also be licensed under the LGPLv2.1 license, with respect to the visualizer's intended usage of DevIL.

Library	License	Built-in manipulation	Reading and writing	Platforms	Footprint
ImageMagick [30]	compatible with GPL V3	yes	yes	linux, windows, mac	large library
stb_image [4]	public domain	no	yes	linux, windows, mac	single small header
CImg [31]	CeCILL-C	yes	yes	linux, windows, mac	single large header
FreeImage [15]	GPL V3	no	yes	linux, windows, mac	medium library
DevIL [9]	LGPLv2.1	no	yes	linux, windows, mac	small library

Table 4.1: Comparison of several C++ image libraries.

Generating images with libraries without built-in image manipulation will take more development time, as common drawing routines for drawing lines and circles and other shapes, including text, will need to be implemented. Of the above image libraries, only two have built-in image manipulation routines: ImageMagick and CImg. While using any of the other libraries is not out of the question, having predefined drawing routines cuts down on development time and complexity.

Looking at the differences between ImageMagick and CImg, ImageMagick has more advanced and better documented functionality than CImg. However, the ImageMagick library has a very large footprint, whereas CImg does not. Library footprint is important because the build process of OpenQL should not get bogged down.

4.7. Conclusion

Many different parameters affect the choice of programming language and image library to use. Making a choice was not an easy process. However, for one aspect of the puzzle no feasible solution could be found. Due to the requirement of visualisation of several internal OpenQL compiler processes, **the choice has been made to implement the visualizer in C++**, in order to have direct access to the internal data structures of the OpenQL compiler. Implementing support to make these internal structures available for a Python-based visualizer is not feasible.

The choice of C++ as implementation language for the visualizer has two consequences. First, no code can be reused, because all reusable code is Python code. The cost of having to build functionality to provide the visualizer written in Python with access to the C++ internal compiler structures outweighs the benefit from possibly being able to reuse code from other Python visualizers. Secondly, this means that a C++ library has to be chosen as the image library.

The choice of image library to use has fallen on CImg, a lightweight, easy to use and small basic image manipulation and display library. It runs on all three required platforms for OpenQL: Windows, Linux and MacOS. It has built-in image manipulation, therefore not necessitating the need for custom pixel drawing routines. When only using the header file, the CeCILL-C license is applicable, which is compatible with OpenQL's license.

5

Visualizer features

In this chapter the development process of the visualizer is discussed. Section 5.1 describes the process of setting up OpenQL for development. Then, the general architecture of the visualizer is discussed in section 5.2. The chapter continues with section 5.3 on the development of the circuit visualizer. Section 5.4 discusses the implementation of the qubit interaction graph. The chapter concludes with section 5.5 on the mapping graph.

5.1. Preliminary work

5.1.1. Compiling OpenQL on Windows

When the visualizer project was started, there were some difficulties with getting OpenQL to compile on a machine running Windows 10 with the MSVC compiler. Several parts of the general OpenQL code were not portable to the MSVC compiler, e.g. code transforming JSON objects into C++ standard strings was accepted by GCC, but not by MSVC.

OpenQL supports a pre-built Python package, as well as compilation from source on Linux, Mac and Windows. To install from source, dependencies need to be installed first and on Windows, the shell environment needs to be setup in such a way to have access to these dependencies, Python and MSVC during the build process.

Work was done on the code to make it compilable with MSVC, including adding several definitions of constants that were taken for granted by the GCC compiler. This process took quite a while. Later on during development of the visualization features, the build process was completely rewritten, making it inherently compatible with Windows and the MSVC compiler.

5.1.2. Development environment

The visualizer was mostly developed on a Windows desktop with Windows 10 installed, using PowerShell as terminal and VSCode as text editor. MSVC version 16.5 was used to compile the project. A Linux virtual machine was available to test changes on Linux and also for comparing errors in the build and installation process while setting up OpenQL on Windows.

5.2. General architecture

This section will elaborate on the general architecture of the visualizer.

5.2.1. Setting up the visualizer

On Windows the visualizer does not require any additional packages to be installed. On Linux and Mac however, the visualizer relies on the X11 library in order to generate and display the image of the visualized circuit. If this library is not found, the visualizer will gracefully disable itself during the build process of OpenQL.

The entry point for the visualizer is the calling of the visualizer pass in the OpenQL program. It does not matter at which point of compilation the visualizer is called, it has provisions to make sure it will not crash, even if scheduling and/or mapping is not done yet. The pass manager calls the visualizer and forwards the pass options to it. Then, the correct visualization is chosen based on the visualization type requested by the user.

5.2.2. Source structure

General types used throughout the source code of the visualizer are stored in `visualizer_types.h`. The visualizer's entry point is declared in `visualizer.h` and defined in `visualizer_common.cc`, together with the definitions of commonly used functions throughout the visualizer from `visualizer_common.h`. Each of the visualization types has its own header and source file combination, except for pulse visualization, which is grouped under the circuit visualizer.

The image library used in the visualizer, `Clmg` has a wrapper around it to reduce compilation times and to customise commonly used drawing functions used throughout the visualizer.

5.2.3. Configuration

The visualizer has many parameters that can be configured in order to enable or disable functionality, to choose the type of visualization and to change the look-and-feel of the generated image. Some of these parameters have to be passed as compiler pass options to the visualizer, while others are stored in separate configuration files.

```
compiler.set_pass_option("Visualizer", "visualizer_type", "CIRCUIT")
compiler.set_pass_option("Visualizer", "visualizer_config_path", "visualizer_config.json")
compiler.set_pass_option("Visualizer", "visualizer_waveform_mapping_path", "waveform_mapping.json")
```

Figure 5.1: The pass options for the visualizer. The type determines what should be visualizer: the circuit, the mapping graph or the qubit interaction graph. The config path is the path to the general visualizer configuration file. The waveform mapping path is the path to the file containing the waveform mappings for the pulse visualization feature of the circuit visualizer.

The visualizer has three pass options (see Figure 5.1), one for determining the type of visualization and two containing the paths to the visualizer configuration file and the waveform mapping file for pulse visualization. The visualizer configuration file contains all the parameters controlling the functionality and look-and-feel of the visualizer, including options for turning on pulse visualization and grouped classical bit lines to reduce clutter. An example is shown in Figure 5.2. The different functionalities and look-and-feel options will be discussed in the next sections of this chapter. The waveform mapping configuration file stores the different waveforms that correspond to each gate on each qubit. The pulse visualization feature uses this waveform mapping to visualize the pulses.

5.2.4. Example circuit

Throughout the rest of the chapter, the visualization features will be demonstrated by use of an example OpenQL program. This program is shown in Figure 5.3. The corresponding topology of the qubits on the quantum hardware this program will be compiled for is shown in Figure 5.4. The program contains two X and Y gates on logical qubits 2 and 3. Between those two sets of single-qubit gates, a $CNOT$ is inserted. However, there is no physical connection between qubits 2 and 3. This results in the mapper producing a series of $SWAP$ gates to bring the logical states of qubits 2 and 3 next to each other, so that the $CNOT$ can be executed.

```

{
  "saveImage": true,
  "backgroundColor": [179, 235, 255],
  "mappingGraph":
  {
    "initDefaultVirtuals": false,
    "showVirtualColors": true,
    "showRealIndices": true,
    "useTopology": true,
    "qubitRadius": 15,
    "qubitSpacing": 7,
    "fontHeightReal": 13,
    "fontHeightVirtual": 13,
    "textColorReal": [0, 0, 255],
    "textColorVirtual": [255, 0, 0],
    "realIndexSpacing": 1,
    "qubitFillColor": [255, 255, 255],
    "qubitOutlineColor": [0, 0, 0]
  },
  "interactionGraph":
  {
    "outputDotFile": true,
    "borderWidth": 32,
    "minInteractionCircleRadius": 100,
    "interactionCircleRadiusModifier": 3.0,
    "qubitRadius": 17,
    "labelFontHeight": 13,
    "circleOutlineColor": [0, 0, 0],
    "circleFillColor": [255, 255, 255],
    "labelColor": [0, 0, 0],
    "edgeColor": [0, 0, 0]
  },
  "circuit":
  {
    "cycles":
    {
      "labels":
      {
        "show": true,
        "inNanoSeconds": false,
        "rowHeight": 24,
        "fontHeight": 13,
        "fontColor": [0, 0, 0]
      }
    }
  }
}

```

Figure 5.2: A section of an example visualizer configuration file. This configuration file controls the functionality and look-and-feel of the visualizer. The configuration for each visualization type is separated into its own section, while the general visualization parameters are stored at the top level.

5.3. Circuit visualization

A quantum program is implemented by way of a quantum circuit that performs the specified operations in the most optimal order as defined by the quantum compiler. This process produces a final sequence of quantum gates that are executed on their target qubit(s). The circuit visualizer captures this sequence of gates on qubits in an image. This image lets the user of an OpenQL program inspect the quantum circuit that the OpenQL compiler produces. This is useful for research, compiler design and educational purposes.

5.3.1. Features

Abstract circuit visualization

The abstract representation of the visualized quantum circuit visualizes the gates as abstract, platform-independent representations, instead of the signals sent to the actual quantum hardware. Figure 5.5 shows the abstract circuit visualization of the example OpenQL program discussed in section 5.2.4.

List of features:

- cutting cycles

```
kernel.gate("x", [2])
kernel.gate("y", [3])
kernel.gate("cnot", [2,3])
kernel.gate("x", [2])
kernel.gate("y", [3])
kernel.gate('measure', [2])
kernel.gate('measure', [3])
```

Figure 5.3: Example program that will be used throughout this chapter to demonstrate the visualization features of OpenQL.

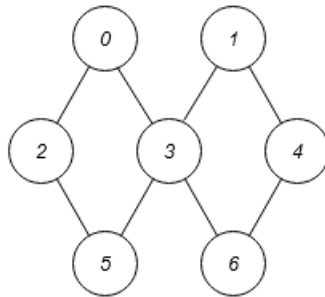


Figure 5.4: The topology of the quantum hardware used for the example program of Figure 5.3.

- compressing cycles
- grouping classical lines

Not included in this list are options to show cycle edges, bit line edges, change the labels of qubits and cycles, changing colors, alpha and sizes of elements and more look-and-feel options.

Cycle cutting

Gates can take many cycles to execute without anything else happening in the circuit. Cycles where a gate is executing, but nothing is happening, are called 'empty' cycles. Functionality is provided by the circuit visualizer to 'cut' these cycles out of the diagram, squeezing multiple empty cycles together into one cycle. A threshold can be set to indicate the minimum required amount of empty cycles before cycles are cut. Figure 5.6 shows an example of cycle cutting in action. The durations of the *SWAP* and *CNOT* gates are compressed into a small column, minimizing the amount of screen space needed to show those gates.

Compressing cycles

In contrast to cycle cutting, where empty cycles are squeezed together, compressing cycles reduces the duration of each gate to one cycle. This then produces a compressed visualization of the quantum circuit, where sequencing is preserved, but cycle information is thrown away. This can be useful to create a general overview of the circuit, without being interested in the cycle information.

Grouping classical lines

Instead of each classical bit register having its own bit line representing it, the classical bit lines can be grouped together. Measurements going into the classical registers then have a number associated with them, indicating the classical register index the measurement stores its result in.

Pulse circuit visualization

In addition to the abstract representation of a circuit, a quantum circuit can also be represented by the pulses operating on the physical quantum hardware. Note that this feature has specifically been built for the quan-

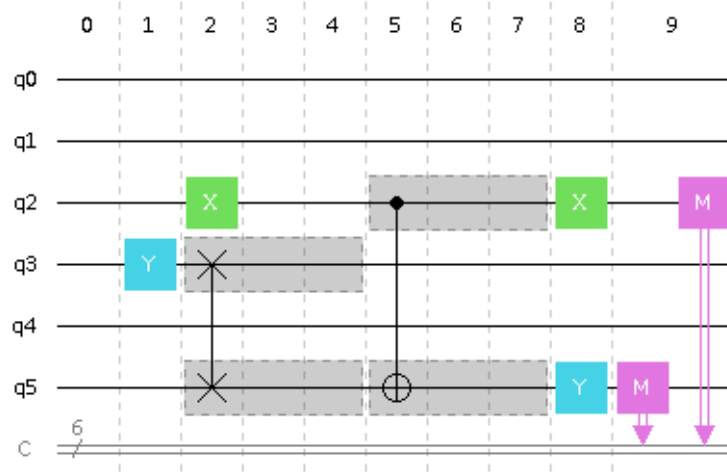


Figure 5.5: Abstract circuit visualization of the example program discussed in section 5.2.4. Note the outlines around the durations of the *SWAP* and *CNOT* gates, indicating exactly how long those gates take to execute.

tum hardware used in the DiCarlo lab, and may not be useful for other quantum hardware. Pulse visualization can be enabled and configured with parameters in the circuit visualizer configuration. See Figure 5.7 for an example of pulse visualization.

A qubit in the pulse visualizer is represented by three lines, the microwave, flux and readout lines, responsible for single-qubit, two-qubit and measurement gates respectively. A gate has a specific pulse (waveform) associated with it, which can also differ for each qubit even though the gate is the same. These waveforms are stored in the waveform mapping configuration file. A waveform is stored as a sequence of samples, and together with the sample rate can be reconstructed into a visualized pulse by the visualizer. This produces an image with the actual waveforms passed to the quantum hardware at each moment of the circuit.

5.3.2. Architecture

The circuit visualizer works on a grid whose cells are formed by the intersection of bit lines (both quantum and classical) and cycles. Bit lines are drawn from the first cycle to the last, and gates are drawn on their corresponding bit lines with a length corresponding to the amount of cycles the gate takes to execute.

Two main classes form the heart of the circuit visualizer: the `CircuitData` class and the `Structure` class. The `Structure` class calculates and stores the screen positions of the various elements used in the visualized circuits. The drawing code in the visualizer uses the positions stored in this class to properly position gates and other elements on the screen. The `CircuitData` class contains data on the properties of the circuit, such as amount of qubits, and the (gate) contents of each cycle, including information about whether that cycle is cut or not.

The code that draws the actual image on the screen takes the information provided by the `CircuitData` and `Structure` classes and uses that together with the layout information contained in the `CircuitLayout` class to determine what to draw, where to draw it and how to draw it respectively. The `CircuitLayout` class contains the parameters that control the look-and-feel of the visualized circuit.

Each gate has its own visuals associated with it. In the hardware configuration file, the custom instructions should have a special parameter, `visual_type`, that maps to the visual parameters of that associated type in the visualizer configuration file. These parameters determine the look-and-feel of each of the gates and are used to draw the abstract representation of the circuit.

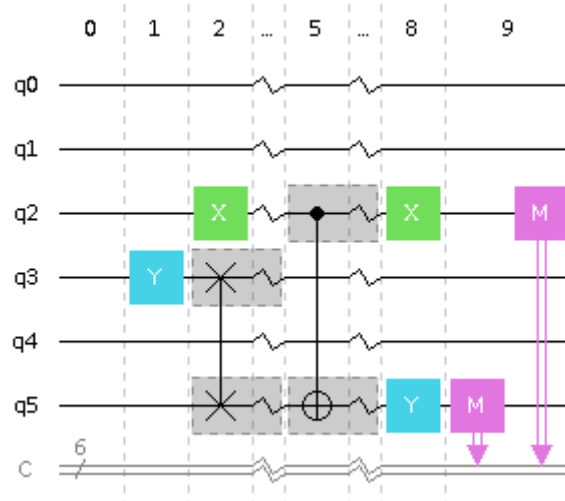


Figure 5.6: Abstract circuit visualization of the example program discussed in section 5.2.4. Gates with durations longer than two cycles are 'cut', i.e. each cycle beyond the first cycle of that gate is compressed into a dashed line. Many gates have long durations during which nothing else is happening on any of the other qubits in the circuit. Cycle cutting provides a way to minimize the amount of screen space needed to display those gates.

5.3.3. Future work

The basic functionality of the circuit and pulse visualizers is completed. Several smaller improvements are planned for the future, including:

- Cycle cutting working with the mapping graph and pulse visualization: cycles currently cannot be cut when using the mapping graph or the pulse visualization.
- Pulse visualization working with mapping graph: the mapping graph only works with the abstract gate representation of the quantum circuit.
- Collapsing of three lines for qubits in pulse visualization to one line: to reduce visual clutter, the microwave, flux and readout lines of the pulse visualization for each qubit could be collapsed into a single lines, with different colors indicating the different types of pulses for that qubit.
- Barrier gates cannot be displayed: barrier gates separate parts of the circuit. Each gate started before a barrier gates has to finish executing before any of the gates after the barrier gate can be started.
- Generating default gate visuals for custom instructions without a specified visual type or corresponding visual type in the visualizer configuration file: currently, when a custom instruction does not have visual parameters associated with it, the visualization of that instruction is skipped. A better solution would be to automatically generate these parameters when they are missing, in order to preserve the correctness of the visualized circuit.

5.4. Qubit interaction graph visualization

The qubit interaction graph visualizes the interactions between the qubits in the quantum circuit. Both the involved qubits and the amount of interactions throughout the execution of the circuit are visualized. Note that the order in which the OpenQL compiler passes are executed influences the output of the interaction graph visualizer. For example, when ran after the mapper, the interaction graph represents the connections between the real qubits as the mapper mapped those. When ran before the mapper, the interaction graph represents the interactions between the logical qubits, as they have not yet been mapped, and the quantum hardware's topology has not yet been taken into account.

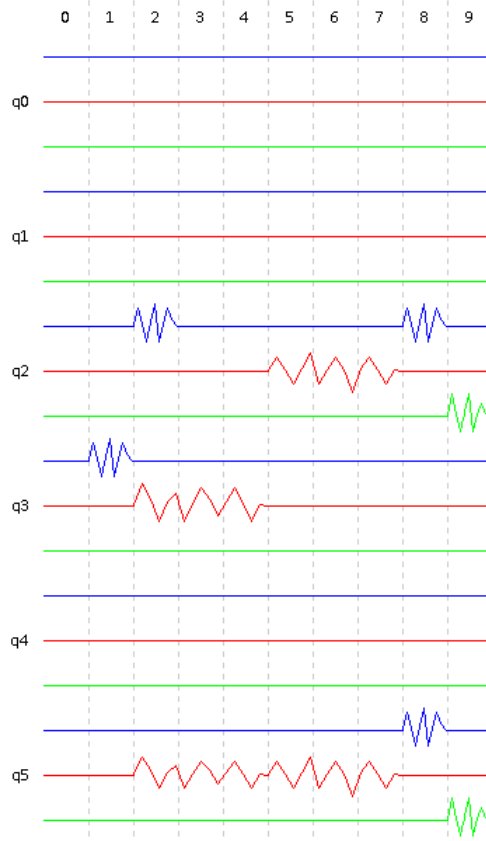


Figure 5.7: Pulse level circuit visualization of the example program discussed in section 5.2.4. Here, the gates are rendered as waveforms on RF lines. Note that the waveforms themselves are just example pulses. The duration, shape and addressed RF line of each pulse determines what gate gets executed. Single-qubit gates use the blue microwave line, two-qubit gates the red flux lines on both corresponding qubits and measurements use the green readout line.

5.4.1. Features

The qubit interaction graph has a very simple built-in visualization, shown in Figure 5.8. The qubits used in the quantum circuit are placed on a circle, and lines with numbers between them indicate the interactions and the number of interactions. This visualization is not intended to be used in place of heuristics that can elegantly place the qubits and their relations on a grid. To facilitate this, the qubit interaction graph visualizer also has an option to output a DOT file. An example DOT file is shown in Figure 5.9 This DOT file adheres to the DOT file standard and can be used with third party graphing software to produce a properly visualized interaction graph. In essence, this means that the DOT file production of the qubit interaction graph visualizer is the main output, while the simple visualization is secondary.

5.4.2. Architecture

The architecture of the qubit interaction graph visualizer is simple. It uses simple math to produce its own visualization. The DOT file is generated by calculating the edges between qubits and outputting them (including the amount of times an edge was encountered) in the DOT file standard format.

5.4.3. Future work

Because of the simplicity of the qubit interaction graph, no major work remains to be done on it. One could think of allowing it to be compiler pass order independent. That would mean it can produce an interaction graph or coupling graph (when ran after the mapper pass) at any point during compilation, controlled with an option. A second possibility is to improve the simple visualization, implement a heuristic that tries to

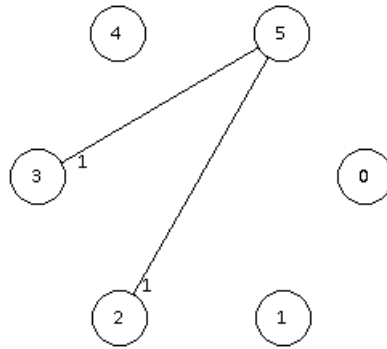


Figure 5.8: Basic qubit interaction graph visualization of the example program discussed in section 5.2.4. Note that this built-in visualization is very basic.

```
graph qubit_interaction_graph {
  node [shape=circle];
  2 -- 5 [label=1];
  3 -- 5 [label=1];
}
```

Figure 5.9: DOT file of the qubit interaction graph of the example program discussed in section 5.2.4. This interaction graph only has two edges. These two edges represent the *CNOT* between logical qubits 2 and 3. Physical qubit 5 is involved because there is no direct connection between physical qubits 2 and 3.

place the qubits and edges in an elegant way.

5.5. Mapping graph visualization

Often it is the case that a two-qubit gate takes place between two qubits that are not physically connected on the topology of the quantum hardware. When a gate like this is encountered, the mapper calculates a path to bring one or both of the logical qubits involved in the gate together through a series of swap gates that swap the states of the qubits involved. This sequence of swap gates has the effect of bringing the logical state of the two qubits in the original two-qubit gate together in different physical qubits. The mapping graph visualizes this process, shown in Figure 5.10.

5.5.1. Features

The mapping graph visualizes the mapping of logical (virtual) qubits to physical (real) qubits. This mapping changes per cycle as two-qubit gates that are not connected on the topology of the quantum hardware are requested by the user.

The mapping graph visualizer can assign colors to each individual logical qubit to easily follow the evolution of the qubit mapping throughout the execution of the circuit. Physical qubits can be initialized with a logical qubit, or only those physical qubits that have had a gate act upon them can be initialized with a logical qubit.

If the hardware configuration file includes a topology section, the layout of the physical qubit topology will be taken and drawn from that section. If there is no defined topology, the mapping graph visualizer defaults to simply placing the physical qubits in a grid structure.

5.5.2. Architecture

At the start of the mapping graph visualizer part of the visualizer project, there was no functionality present in OpenQL that could track the movement of the logical qubits throughout the circuit. Internally, the mapper

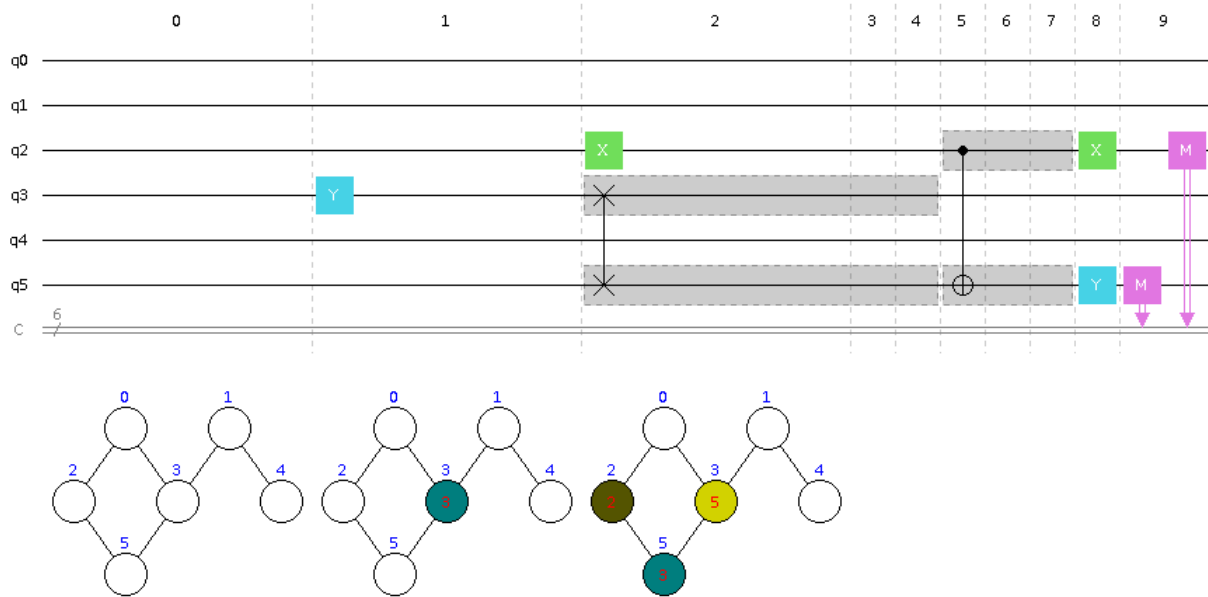


Figure 5.10: Mapping graph visualization of the example program discussed in section 5.2.4. Each circle represents a physical qubit. The index of the physical qubits is displayed above the qubits. The index of the logical qubit stored in that physical qubit is displayed inside of each activated physical qubit. Each of the physical qubits fills with the index of the logical qubit currently stored in the first cycle that physical qubit is used. Note that the topology is only shown for cycles in which the mapping has changed.

stores a mapping that tracks this, but there was no satisfactory way to bring this internal mapping outside of the mapper.

The first idea that was tried was to add a special gate, called a remap gate, each time a swap or move (a move is generated when the state of the qubits that need to be swapped are equal) gate is generated by the mapper. This remap gate has as parameters the two real qubits and the two virtual qubits involved in the swap or move. However, this again did not provide a satisfactory way to track the movement of the virtual qubits throughout the execution of the quantum circuit. The remap gates altered the behaviour of the mapper, whereas they should have had no effect on the mapping.

Instead of trying to fix the altering of the behaviour of the mapper by the addition of the remap gates, a simpler solution was adopted. Each time a swap or move is generated, the logical and real qubits involved are added to that swap/move gate in a special register. Then, whenever that swap or move is decomposed into a set of gates that are native on the quantum hardware platform, the swap parameters are propagated to each decomposed gate. The mapping graph visualizer then uses this information to construct the mapping graph.

In addition to visualizing the topology of the quantum hardware and the mapping between physical and logical qubits, the abstract representation of the quantum circuit is shown above the mapping graph. This is a direct copy of the visualization produced by the circuit visualizer.

5.5.3. Future work

Some future work for the mapping graph visualizer includes:

- Better color picking for logical qubits. The current color picker does not pick very distinct colors. A better way would be to let the user define a color scheme for each of the logical qubits involved in the circuit, and/or have the automatic color picker pick out more natural colors.
- Making the mapping graph work with pulse visualization.
- Making the mapping graph work with cycle cutting in the abstract circuit representation.

6

Conclusion

Large strides have been made in the scalability of qubits from many kinds of different physical quantum hardware implementations over the past few years. This has resulted in the need for full-stack developments for quantum computing. Many attempts have been made by many different companies and research groups.

One of these quantum programming platforms has been developed by the Quantum and Computer Architecture group of the Delft University of Technology, called OpenQL. The OpenQL compiler translates programs written in CQASM or the OpenQL quantum programming language to native instructions for the targeted quantum hardware, taking a set of constraints on the qubit connectivity and other resource constraints into account. OpenQL is used for running experiments with superconducting qubit chips at the university's DiCarlo lab.

For the compiler team working on the OpenQL compiler the inner workings of the mapper compiler pass have been proven to be quite opaque. The mapper compiler pass maps logical qubits onto physical qubits taking topology constraints into account. One of the wishes of the compiler team was to develop a mapping graph visualizer which shows the logical to physical qubit mapping per cycle. In addition, the research team working with the OpenQL platform in the DiCarlo lab has expressed the need for functionality that can express the generated quantum circuit by the OpenQL compiler in a visual way. Having visualization features will help them to better understand the results of their experiments with their superconducting qubit quantum chips.

In order to fulfill the requirements of both the compiler team and the physicists working with OpenQL, a visualizer compiler pass has been developed that has three main functionalities.

Firstly, the visualizer is able to generate a visual representation of the internal state of the quantum circuit or quantum circuit output of the OpenQL compiler. In essence, this circuit visualizer can produce an image of the quantum circuit at any point during the compilation process, including the final output. Furthermore, the circuit visualizer can output both an abstract representation of the quantum circuit and a pulse level representation of that same circuit. The pulse level visualization has been developed specifically for use by the DiCarlo lab as it is focused on their specific quantum hardware implementation of superconducting qubits.

Secondly, the visualizer can generate a qubit interaction graph, showing the interactions and amount of them between the different qubits in the quantum circuit. By applying the visualizer pass before mapping has been done, this visualizer can generate a qubit interaction graph of the logical qubits. When done after the mapper pass, the visualizer produces a coupling graph, showing the interactions between qubits as they are performed in the actual physical quantum hardware. Since the qubit interaction graph visualizer only has a simple algorithm for the visualization of these interactions, a DOT file representation can also be produced for use by third-party graphing software to generate a better looking image.

Lastly, the mapping graph visualizer generates a per cycle overview of the logical to physical qubit mapping. This provides a way to inspect how the mapping evolves as the quantum hardware executes the cycles of the circuit. Functionality is provided to track the path the logical qubits take through the physical qubit topology.

A view of the abstract representation of the quantum circuit is provided for each cycle to keep track of what the circuit is doing in each cycle.

With the visualizer project having been completed, the OpenQL quantum platform now has functionality to visualize several aspects of the compiler pipeline and generated quantum circuit. This will allow researchers working on the development of OpenQL to gain more insight into the inner workings of several of the compiler's internal processes, thereby allowing them to improve those functionalities. Furthermore, the visualization features will help the physicists working with OpenQL to better understand the results of their experiments, helping them to fine-tune their experiments.

References

- [1] Scott Aaronson and Alex Arkhipov. “The Computational Complexity of Linear Optics”. In: *Theory of Computing* 9.4 (2013), pp. 143–252. DOI: 10.4086/toc.2013.v009a004. URL: <http://www.theoryofcomputing.org/articles/v009a004>.
- [2] Héctor Abraham et al. *Qiskit: An Open-source Framework for Quantum Computing*. 2019. DOI: 10.5281/zenodo.2562110.
- [3] Frank Arute et al. “Quantum supremacy using a programmable superconducting processor”. In: *Nature* 574 (Oct. 2019), pp. 505–510. DOI: 10.1038/s41586-019-1666-5.
- [4] Sean Barret. *stbimage*. URL: <https://github.com/nothings/stb>.
- [5] Colin D. Bruzewicz et al. “Trapped-ion quantum computing: Progress and challenges”. In: *Applied Physics Reviews* 6.2 (June 2019), p. 021314. ISSN: 1931-9401. DOI: 10.1063/1.5088164. URL: <http://dx.doi.org/10.1063/1.5088164>.
- [6] Wikimedia Commons. *File:4 Qubit, 4 Bus, 4 Resonator IBM Device (Jay M. Gambetta, Jerry M. Chow, and Matthias Steffen, 2017).png* — *Wikimedia Commons, the free media repository*. [Online; accessed 26-May-2021]. 2020. URL: [https://commons.wikimedia.org/w/index.php?title=File:4_Qubit,_4_Bus,_4_Resonator_IBM_Device_\(Jay_M._Gambetta,_Jerry_M._Chow,_and_Matthias_Steffen,_2017\).png&oldid=466776966](https://commons.wikimedia.org/w/index.php?title=File:4_Qubit,_4_Bus,_4_Resonator_IBM_Device_(Jay_M._Gambetta,_Jerry_M._Chow,_and_Matthias_Steffen,_2017).png&oldid=466776966).
- [7] Wikimedia Commons. *File:Bloch Sphere.svg* — *Wikimedia Commons, the free media repository*. [Online; accessed 14-April-2021]. 2020. URL: https://commons.wikimedia.org/w/index.php?title=File:Bloch_Sphere.svg&oldid=514662857.
- [8] Wikimedia Commons. *File:Planar Ion Trap; Magnesium Ions (5884514798).jpg* — *Wikimedia Commons, the free media repository*. [Online; accessed 20-April-2021]. 2017. URL: [https://commons.wikimedia.org/w/index.php?title=File:Planar_Ion_Trap;_Magnesium_Ions_\(5884514798\).jpg&oldid=236875680](https://commons.wikimedia.org/w/index.php?title=File:Planar_Ion_Trap;_Magnesium_Ions_(5884514798).jpg&oldid=236875680).
- [9] Meloni Dario Denton Woods Nicolas Weber. *DevIL*. URL: <http://openil.sourceforge.net/>.
- [10] Cirq Developers. *Cirq*. Version v0.10.0. See full list of authors on Github: <https://github.com/quantumlib/Cirq/graphs/contributors>. Mar. 2021. DOI: 10.5281/zenodo.4586899. URL: <https://doi.org/10.5281/zenodo.4586899>.
- [11] SWIG Developers. *SWIG*. See full list of authors on: <http://www.swig.org/guilty.html>. URL: <http://www.swig.org/>.
- [12] David P. DiVincenzo. “The Physical Implementation of Quantum Computation”. In: *Fortschritte der Physik* 48.9-11 (Sept. 2000), pp. 771–783. ISSN: 1521-3978. DOI: 10.1002/1521-3978(200009)48:9/11<771::aid-prop771>3.0.co;2-e. URL: [http://dx.doi.org/10.1002/1521-3978\(200009\)48:9/11%3C771::AID-PROP771%3E3.0.CO;2-E](http://dx.doi.org/10.1002/1521-3978(200009)48:9/11%3C771::AID-PROP771%3E3.0.CO;2-E).
- [13] Alex Clark Fredrik Lundh. *Pillow (PIL Fork)*. 2021. URL: <https://python-pillow.org/>.
- [14] X. Fu et al. “A Heterogeneous Quantum Computer Architecture”. In: *CF '16* (2016), pp. 323–330. DOI: 10.1145/2903150.2906827. URL: <https://doi.org/10.1145/2903150.2906827>.
- [15] Floris van den Berg Hervé Drolon. *FreeImage*. URL: <https://freeimage.sourceforge.io/>.
- [16] N. Khammassi et al. *OpenQL: A Portable Quantum Programming Framework for Quantum Accelerators*. 2020. arXiv: 2005.13283 [quant-ph].
- [17] Jens Koch et al. “Charge-insensitive qubit design derived from the Cooper pair box”. In: *Physical Review A* 76.4 (Oct. 2007). ISSN: 1094-1622. DOI: 10.1103/physreva.76.042319. URL: <http://dx.doi.org/10.1103/PhysRevA.76.042319>.
- [18] Ryan LaRose. “Overview and Comparison of Gate Level Quantum Software Platforms”. In: *Quantum* 3 (Mar. 2019), p. 130. ISSN: 2521-327X. DOI: 10.22331/q-2019-03-25-130. URL: <http://dx.doi.org/10.22331/q-2019-03-25-130>.

- [19] Daniel Loss and David P. DiVincenzo. "Quantum computation with quantum dots". In: *Phys. Rev. A* 57 (1 Jan. 1998), pp. 120–126. DOI: 10.1103/PhysRevA.57.120. URL: <https://link.aps.org/doi/10.1103/PhysRevA.57.120>.
- [20] Microsoft. *Quantum Development Kit*. URL: <https://azure.microsoft.com/en-us/resources/development-kit/quantum-computing/>.
- [21] Alejandro Morais. "Mapping quantum algorithms in a crossbar architecture". MA thesis. Delft University of Technology, Sept. 2019.
- [22] Maximilian Plenert. "A Hollow Beam Trap for Atomic Ensembles of Rubidium - Towards a Long Life Time Quantum Memory for Long Distance Quantum Communication". PhD thesis. Apr. 2009. DOI: 10.13140/RG.2.1.4577.2002.
- [23] John Preskill. "Quantum Computing in the NISQ era and beyond". In: *Quantum* 2 (Jan. 2018). DOI: 10.22331/q-2018-08-06-79.
- [24] Qiskit. *Visualized circuit*. [Online; accessed 26-April-2021]. 2021. URL: https://qiskit.org/documentation/tutorials/circuits_advanced/03_advanced_circuit_visualization.html.
- [25] Salonik Resch and Ulya R. Karpuzcu. *Quantum Computing: An Overview Across the System Stack*. 2019. arXiv: 1905.07240 [quant-ph].
- [26] Maximilian Russ et al. "High-fidelity quantum gates in Si/SiGe double quantum dots". In: *Physical Review B* 97 (Nov. 2017). DOI: 10.1103/PhysRevB.97.085421.
- [27] Aritra Sarkar. *QX simulator IDE*. [accessed 26-April-2021]. URL: <https://gitlab.com/prince-ph0en1x/QuInE>.
- [28] Robert S. Smith, Michael J. Curtis, and William J. Zeng. *A Practical Quantum Instruction Set Architecture*. 2016. arXiv: 1608.03355 [quant-ph].
- [29] J. van Someren. *OpenQL Compiler, Modularization and beyond*. 2021.
- [30] The ImageMagick Development Team. *ImageMagick*. Version 7.0.10. URL: <https://imagemagick.org>.
- [31] David Tschumperlé. *CImg*. URL: <https://cimg.eu/index.html>.
- [32] Han-Sen Zhong et al. "Quantum computational advantage using photons". In: *Science* 370.6523 (2020), pp. 1460–1463. ISSN: 0036-8075. DOI: 10.1126/science.abe8770. eprint: <https://science.sciencemag.org/content/370/6523/1460.full.pdf>. URL: <https://science.sciencemag.org/content/370/6523/1460>.