Optimization Strategies for Quantum Computers in Distributed Systems

Lian Remme

Master's Thesis

Presented at the Chair of Operating Systems (HHU)
in Cooperation with
the Institute of Software Technology (German Aerospace Center, DLR).

Submission:	February 2025
First Reviewer:	Univ.-Prof. Dr. Michael Schöttner
Second Reviewer:	Dr. Andre Waschk

Abstract

Quantum computing has become an increasingly relevant topic in computer science due to the development of functional quantum computers in recent years. Quantum processing units (QPUs) use quantum particles in the form of qubits as information carriers. By using properties of quantum particles, quantum algorithms have been developed that can solve problems in polynomial time, for which only exponentially scaling classical algorithms are known.

QPUs will not run on their own in the foreseeable future, due to high error rates and low stability of qubits. Instead, quantum devices will be one component in a quantum-classical (hybrid) computing system (distributed system). Quantum computers embedded in distributed hybrid quantum-classical systems get instructions from a main CPU, execute them and report their results. To fully utilize quantum calculations, a QPU should be able to communicate with a CPU in "real-time". This means the qubits' information content should stay stable during communication time.

In this thesis, we will look at quantum computing systems with real-time feedback between a QPU and a CPU. As quantum computers are going to be used in real-time hybrid systems, we need optimization routines for code running on these systems. Existing works on the optimization of quantum code focus on the optimization of quantum circuits only, and neglect potential classical calculations that are necessary on a hybrid distributed system. We want to examine which kind of optimization routines are possible and sensible for hybrid quantum-classical computations.

We will evaluate some of today's quantum programming languages (QPLs), especially with respect to their ability to support and optimize real-time hybrid calculations. We will find that most languages support real-time calculations to some extent, but are more adapted to the programming approach of static circuit creation. Additionally, none of the QPLs offer optimization routines targeted at quantum-classical calculations.

Therefore, we will examine options for real-time quantum-classical optimization. For this, we use the low-level QPL Quil. We will introduce optimization operations and apply them to real-time quantum-classical algorithms. We will present metrics to evaluate the performance of hybrid calculations. The results of applying our optimization operations to Quil code will be evaluated against our performance metrics. We will find that we are in principle able to optimize hybrid quantum-classical programs.

We encourage more research in this field. QPLs will become more abstract in the future and need more compilation steps until they can be executed on hardware. This will make the optimization of quantum-classical hybrid code more relevant.

Contents

1. Introduction

In recent years, quantum computing has advanced from a theoretical concept in physics towards the development of working quantum computers. Thus quantum computers become an increasingly relevant topic in computer science. Quantum computing devices use quantum particles in the form of qubits as information carriers, instead of voltage differences used for classical bits. The calculations are done on a quantum processing unit (QPU) instead of on a central processing unit (CPU). By using unique properties of quantum particles, quantum algorithms have been developed that can solve problems in polynomial time, for which only exponentially scaling classical algorithms are known [1, 2]. One famous example is the Shor algorithm [1] which achieves prime factorization that scales in polynomial time.

Typically qubit numbers on a QPU are currently around a few hundred [3, 4, 5, 6]. The largest qubit numbers are 1180 qubits on a device by Atom Computing [7] and 1121 qubits on the IBM Condor chip [8].

The QPUs are programmed using a quantum programming language (QPL) (a domain-specific language (DSL) for a QPU). In recent years, multiple QPLs have been developed, with different properties, advantages and disadvantages.

The aim of quantum computing is to reach *quantum advantage* (or quantum supremacy). Quantum advantage is the ability "to perform tasks with controlled quantum systems going beyond what can be achieved with ordinary digital computers" [9]. I.e. we have reached quantum advantage when we have a QPU that can calculate a problem in a feasible amount of time that would take an unfeasible amount of time on a CPU, i.e. several months or even years.

Whether we reach quantum advantage or not, QPUs will not run on their own in the foreseeable future. The error rates on qubits are too high and the qubits are too unstable (as we will examine more closely in Section 2.1.3). Instead, quantum devices will be one component in a quantum-classical (hybrid) computing system (distributed system). In this scenario, the QPU merely calculates problems that are classically unfeasible. Other calculations are done by classical devices (e.g. CPU, or a graphics processing unit (GPU)), where the lower error rates and easier execution is beneficial.

Quantum computers embedded in distributed hybrid quantum-classical systems get instructions from a main CPU, execute them and report their results. To fully utilize quantum calculations, a QPU should be able to communicate with a CPU in "real-time". This means the communication time should be much shorter than the qubits' coherence time (cf. Section 2.2.1). The coherence time is the time the qubits remain stable and keep their information content. This concept is further looked into in Section 2.1.3.

In this thesis, we will look at quantum computing systems with real-time feedback between a QPU and a CPU. This kind of computing systems are the ones in which

quantum computers are going to be used in in the foreseeable future (cf. Section 2.2.1). Therefore, it will become necessary to enable developers to program these systems. This results in the need for optimization routines for code running on quantum computing systems with real-time feedback.

The optimization of code running on these kind of systems will be the main concern of this thesis. Works on the optimization of quantum code [10, 11, 12, 13] focus on the optimization of the quantum part only, and neglect potential classical calculations that are necessary on a hybrid distributed system.

In this work, we want to examine which kind of optimization routines are possible and sensible for hybrid quantum-classical computations. We will especially look at real-time calculations, i.e. those where a classical and a quantum component communicate within the qubits' coherence times.

To do this, we will first look at some key concepts of quantum computation, distributed computing and optimization techniques in Chapter 2, as well as at related work in Chapter 3.

In Chapter 4, we will evaluate some of today's QPLs, especially with respect to their ability to support and optimize real-time hybrid calculations. We will find that most languages support real-time calculations to some extent, but are more adapted to the programming approach of static circuit creation (cf. Section 4.1.2). Additionally, none of the QPLs offer optimization routines targeted at quantum-classical calculations.

Therefore, we will examine options for real-time quantum-classical optimization in Chapter 5. For this, we use the low-level QPL Quil [14]. We will introduce metrics to evaluate the performance of hybrid calculations. Our optimization operations will be applied to real-time quantum-classical algorithms. The results will be evaluated against our performance metrics. We will find that we are in principle able to optimize hybrid quantum-classical programs. For current algorithms, the difference between optimized and original code is not very high. However, in the future this field will become more relevant, when QPLs become more abstract and need more compilation steps until they can be executed on the hardware.

2. Background

In this section, we will examine some concepts needed for the remaining parts of the thesis. In Section 2.1, we introduce the key concepts of quantum computing. We will cover distributed computing and how quantum computers can be part of a distributed system in Section 2.2. Finally, we will look into compilation and optimization techniques in classical computing systems in Section 2.3.

2.1. Quantum Computing

In recent years, quantum computing has advanced from a theoretical concept in physics towards the development of working QPUs. Therefore, quantum computing becomes an increasingly relevant topic in computer science. Using unique properties of quantum mechanics, quantum computers promise to offer calculation speedup for certain problems like prime factorization [1], unstructured search [15], or chemical simulations [16].

Quantum computers are theoretically able to calculate the same problems as classical computers. This means they are capable of performing Turing-complete calculations, but cannot solve, e.g., the Halting-Problem [17, chapter 3.1.1]. Simulating QPUs with a CPU, in general, scales exponentially with the number of qubits [18]. No efficient algorithm to simulate a QPU with a CPU is known. On the other hand, CPUs can be efficiently simulated on a QPU, e.g. by simulating NAND gates [17, chapter 1.4.1]. However, QPUs are currently slower and much more error-prone than CPUs, meaning they will not replace CPUs in the foreseeable future.

In this section, we will look at concepts of quantum computing relevant for the remaining thesis. This will be a brief introduction into the topic of quantum computing and only cover what is necessary to understand the succeeding parts of this thesis. We refer the interested reader to Nielsen's and Chuang's book [17] for a more detailed introduction to quantum computation.

Section 2.1.1 introduces the properties of a qubit. Afterwards, Section 2.1.2 will be about calculations on a QPU, and Section 2.1.3 is about quantum hardware and today's challenges around it.

2.1.1. Properties of Qubits

The big difference between classical computers and quantum computers is that the former use bits for their calculations, while the latter use qubits.

Bits are expressed by voltage differences in a CPU. They hold *classical information*, which is restricted to either 0 or 1. Values between 0 and 1 are not defined.

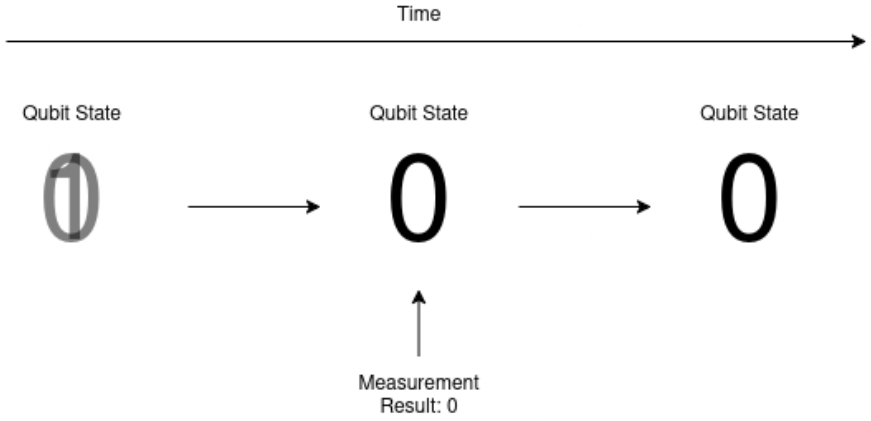


Figure 2.1.1.: An illustration how the measurement of the qubit influences its state. The qubit is originally in a superposition between two state. In the moment of the measurement, the qubit randomly "decides" on 0 or 1 (here 0) and is in this state afterwards, i.e. the superposition is lost.

Qubits consist of elements that can be in quantum states. There are different physical realizations for qubits available, e.g. superconducting qubits [19], or ion-trapped qubits [20]. In contrast to bits, a qubit cannot only be either 0 or 1, but also in a mixture (superposition) of 0 and 1, which is *quantum information*. The states that a qubit can take on are known as *quantum states*.

Measurements

To transfer quantum information to classical information, a qubit needs to be measured. While a qubit may be in a superposition between 0 and 1, the result of a measurement can only be either 0 or 1. The quantum state of the qubit influences the likelihood of the measurement outcome: E.g. the probability of the measurement outcome for a 0 can be 60% and the one for a 1 can be 40%.

However, while it is possible to predict the probability of a measurement outcome, the measurement itself is an inherently non-deterministic process. It is impossible to predict the measurement outcome (except if the probability for an outcome is 100%) due to the laws of quantum physics.

In quantum physics, the measurement of a system influences the state of said system. A measurement causes the quantum state to collapse into the result of the measurement. This is commonly referred to the quantum state "deciding" on either 0 or 1 at the moment of measuring. After the measurement, the qubit is not in superposition between 0 and 1 anymore, but at the state of 0 or 1 with 100% certainty. The state the qubit remains in is the one that has been measured beforehand. This is illustrated in Figure 2.1.1.

A measurement process can therefore also be described as a process where quantum information is destroyed in exchange for gaining one bit of classical information about the quantum system.

Formal Description of Qubits

To describe qubit states in a more formal matter, the braket notation [21] is used. This notations consists of *bras* and *kets*. A ket is a complex vector v written in the form of $|v\rangle$. A bra is the complex conjugate of $|v\rangle$, depicted as $\langle v|$. One can write a matrix multiplication between a bra and a ket as $\langle w|v\rangle$.

In this thesis, we will mostly need kets, as they are commonly used to describe qubits. A qubit in the zero-state is described by $|0\rangle$, a qubit in the one-state by $|1\rangle$. The states can also be expressed as vectors in $\mathbb{C}^2$:

$$|0\rangle = \begin{bmatrix}1\\0\end{bmatrix} \quad |1\rangle = \begin{bmatrix}0\\1\end{bmatrix}. \tag{2.1.1}$$

A (general) superposition of both states can be depicted by a summation:

$$\alpha\,|0\rangle + \beta\,|1\rangle, \quad \text{with } \alpha, \beta \in \mathbb{C}. \tag{2.1.2}$$

The probability to measure $|0\rangle$ is $|\alpha|^2$, the probability to measure $|1\rangle$ is $|\beta|^2$. From this arises the condition that

$$|\alpha|^2 + |\beta|^2 = 1. \tag{2.1.3}$$

A classical bit could be depicted similarly, though its seldom done due to the simplicity of bits. If the zero-state of a bit was depicted by B_0 and the one-state by B_1, a general bit-state would be:

$$a \cdot B_0 + b \cdot B_1, \quad \text{with } a, b \in \{0, 1\} \tag{2.1.4}$$

with the condition that $a + b = 1$.

We end up with a *general qubit state* $|\psi\rangle$ that is described by

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle, \quad \text{with } \alpha, \beta \in \mathbb{C}, \quad |\alpha|^2 + |\beta|^2 = 1 \tag{2.1.5}$$

with $|\alpha|^2$ being the measurement probability of 0 and $|\beta|^2$ being the measurement probability of 1.

To mathematically combine multiple qubits, a tensor product between the qubits is calculated: $|\psi\rangle \otimes |\varphi\rangle \otimes \cdots \otimes |\xi\rangle$. For further mathematical details, we refer the reader to [17, chapter 2]. Relevant for this thesis is to note that the $\otimes$ symbol can be omitted and multiple qubits can simply be described as $|\psi\varphi\ldots\xi\rangle$.

A general 2-qubits state is

$$\alpha\,|00\rangle + \beta\,|10\rangle + \gamma\,|01\rangle + \delta\,|11\rangle, \qquad \text{with } |\alpha|^2 + |\beta|^2 + |\gamma|^2 + |\delta|^2 = 1. \tag{2.1.6}$$

As we will later see, it is possible to *negate* qubits. Negating a classical bit means putting a zero-state to a one-state and vice versa. Applying this to Equation (2.1.4), the values of a and b would be switched.

Analogously, the negated qubit $\left|\overline{\psi}\right\rangle$ of $|\psi\rangle$ has $|\psi\rangle$'s values of α and β switched:

$$\left|\overline{\psi}\right\rangle = \beta\,|0\rangle + \alpha\,|1\rangle \tag{2.1.7}$$

$$\iff |\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle. \tag{2.1.8}$$

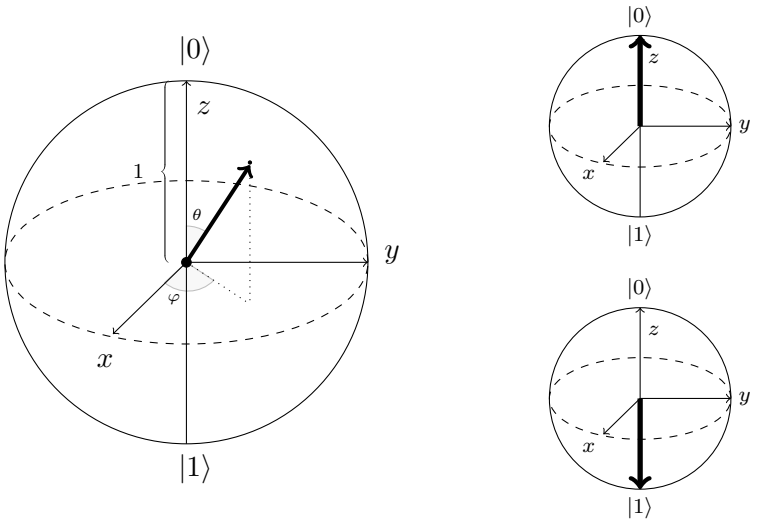


Figure 2.1.2.: Example for Bloch spheres. They have a radius of 1. The Bloch-vector (thick line) depending on φ and θ indicates a qubit state. As the vector of the left Bloch sphere points to the upper half of the sphere, a measurement is more likely to result in $|0\rangle$ than $|1\rangle$. The Bloch spheres on the right show the qubit in the basis states $|0\rangle$ (upper) and $|1\rangle$ (lower).

The Bloch Sphere

The Bloch sphere is a way to depict qubits. We will look into this concept as it can help understanding the effects of quantum gates.

Both α and β are complex numbers, meaning a priori the numbers have two degrees of freedom each, one for their real (a) and one of their imaginary (b) part: $\alpha = a + ib$. This would lead to four degrees of freedom to describe one qubit.

However, we have the condition $|\alpha|^2 + |\beta|^2 = 1$. Due to this restriction, the degrees of freedom for a qubit reduce to three.

Using the Hopf map, we can express α and β with the three free parameters ξ, θ and φ [22]:

$$\alpha = e^{i\xi} \cos\left(\frac{\theta}{2}\right) \tag{2.1.9}$$

$$\beta = e^{i(\xi+\varphi)} \sin\left(\frac{\theta}{2}\right). \tag{2.1.10}$$

Both α and β have a so-called *global phase* of $e^{i\xi}$. This global phase has no observable physical effects [17, chapter 2.2.7] and can therefore be set to 1.

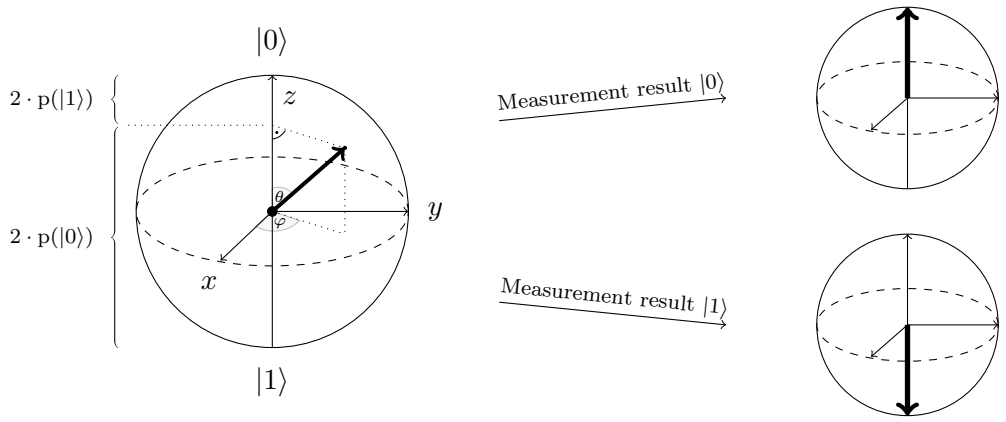


Figure 2.1.3.: How a measurement influences the Bloch sphere. The Bloch-vector's z-direction shows the probability for measuring $|0\rangle$ and $|1\rangle$. The probabilities are derived from Equations (2.1.9) and (2.1.10).

This leads to α and β having two degrees of freedom in total, θ and φ:

$$\alpha = \cos\left(\frac{\theta}{2}\right) \tag{2.1.11}$$

$$\beta = e^{i\varphi}\sin\left(\frac{\theta}{2}\right) \tag{2.1.12}$$

where $0 \leq \theta \leq \pi$ and $0 \leq \varphi < 2\pi$.

The variables φ and θ of Equations (2.1.11) and (2.1.12) can be interpreted as angles of a vector pointing to a spherical surface. This is used to depict a qubit value using the Bloch-sphere [17, chapter 1.2], a 3D sphere with radius 1.

An example of a Bloch sphere depiction of a qubit is given in Figure 2.1.2. The state of the qubit is shown by the position of the Bloch-vector. The z-direction of the Bloch-vector indicates the probability for the measurement outcomes. A Bloch-vector in the xy-plane represents a qubit that will be measured $|0\rangle$ and $|1\rangle$ with 50% probability each. If the vector is completely on the z-axis, it is either $|0\rangle$ (if $\theta = 0$) or $|1\rangle$ (if $\theta = \pi$) with 100% certainty. A measurement projects the vector onto the z-axis. This is depicted in Figure 2.1.3.

Pauli Basis States

If a Bloch-vector is on one of the coordinate axes, the qubit is in a *Pauli basis state*. They represent the eigenvectors of the Pauli matrices σ_x, σ_y and σ_z [17, chapter 2.1.3]:

$$\sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \qquad \sigma_y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \qquad \sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}. \tag{2.1.13}$$

There are six Pauli basis states, two for each of the x-, y- and z-basis. The basis states of the Pauli-z basis are $|0\rangle$ and $|1\rangle$. These are the states that typically correspond to the classical 0 and 1 bit values.

The basis states of the Pauli-x basis are:

$$|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \tag{2.1.14}$$

$$|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \tag{2.1.15}$$

and of the Pauli-y basis:

$$|\phi^+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + i\,|1\rangle) \tag{2.1.16}$$

$$|\phi^-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - i\,|1\rangle)\,. \tag{2.1.17}$$

The Pauli basis states are relevant for some theorems, e.g. the Gottesmann-Knill theorem [23], which we will look at in Section 2.1.2. We will also use the Pauli basis states in some optimization routines in Chapter 5.

No-Cloning Theorem

An important theorem about qubit states is the *no-cloning theorem* [24, 25]: An unknown quantum state cannot be copied (cloned) perfectly on another quantum particle. It is possible to transfer the state from one quantum particle to another, but as this always destroys the state on the first particle, the state is not copied. This theorem is especially relevant in quantum cryptography, where it prevents potential eavesdroppers from copying the qubits used for communication [17, chapter 12.6.3].

We will have to consider the no-cloning theorem during optimization. The copying of classical bits is relevant for some optimization routines, which we cannot do in the quantum case.

2.1.2. Calculations on a QPU

In this section, we will examine how one can run calculations on a QPU.

One way of doing a quantum calculation is to apply quantum gates on qubits. This is the most common approach (cf. Chapter 4) and the one we will look at in this thesis.

Quantum Gates

In classical computation, we do calculations by applying a series of gates onto bits and checking the result. This procedure is usually abstracted away from software developers: The gates are arranged in a CPU and the application of the gates is decided by CPU instructions.

In gate-based quantum computation, we apply quantum gates to the qubits and eventually do measurements. We will look at some quantum gates that are needed to do

Table 2.1.1.: Gates acting on a single qubit, their symbols and their effects on $|0\rangle$ and $|1\rangle$. As $|0\rangle$ and $|1\rangle$ are a basis and the gates are linear, their effects on $|0\rangle$ and $|1\rangle$ completely describe the gates.

Gate	Symbol	Effect on $\|0\rangle$	Effect on $\|1\rangle$
Pauli-X / Not	X	$\|1\rangle$	$\|0\rangle$
Pauli-Y	Y	$i\|1\rangle$	$-i\|0\rangle$
Pauli-Z / $P(\pi)$	Z	$\|0\rangle$	$-\|1\rangle$
Phase shift	$P(\varphi)$	$\|0\rangle$	$e^{i\varphi}\|1\rangle$
S / $P\left(\frac{\pi}{2}\right)$	S	$\|0\rangle$	$e^{i\frac{\pi}{2}}\|1\rangle$
T / $P\left(\frac{\pi}{4}\right)$	T	$\|0\rangle$	$e^{i\frac{\pi}{4}}\|1\rangle$
Hadamard	H	$\frac{1}{\sqrt{2}}(\|0\rangle + \|1\rangle)$	$\frac{1}{\sqrt{2}}(\|0\rangle - \|1\rangle)$

gate-based quantum calculations. We do this to get an intuition for how quantum calculations work. Additionally, many current QPLs directly use quantum gates that are applied on qubits, and an understanding for the gates is necessary to understand quantum code.

To do arbitrary calculations, gate-based quantum calculations require a set of universal quantum gates [17, chapter 4.5]. This is analogous to the classical case, where a functionally complete set of gates is needed.

There are several sets of universal quantum gate sets. We will look at some commonly used gates in scientific publications and literature.

In general, quantum gates can be described as matrices that act on the vector representation of a qubit (see Equation (2.1.1)). An alternative description is that gates are rotations of the Bloch-vector on the Bloch-sphere around an axis.

As quantum gates depict the transformation of a quantum state, they have to satisfy some conditions to be physically possible:

Every quantum gate needs to be reversible. This, in particular, means that a quantum gate needs to have as many input as output qubits. A quantum gate can neither create nor destroy a qubit, but only change the state of the qubit(s). A gate like the AND gate would therefore not be possible. If one wants to simulate a non-reversible classical gate, we need to use ancilla qubits that are discarded without being measured. From these requirements follows the mathematical condition that the matrix representation of a gate U has to be unitary, i.e. $U^\dagger U = \mathbb{1}$, where $U^\dagger$ is the result of transposing and complex conjugating U. This is the only mathematical requirement on a quantum gate [17, chapter 1.3.1].

A summary of gates that act on a single qubit is given in Table 2.1.1. As $|0\rangle$ and $|1\rangle$ are a basis and the gates are linear, their effects on $|0\rangle$ and $|1\rangle$ completely describe the gates. In particular, the effects on $|0\rangle$ and $|1\rangle$ can be applied to a superposition by applying the gate to each of the states, because the gate operations are linear.

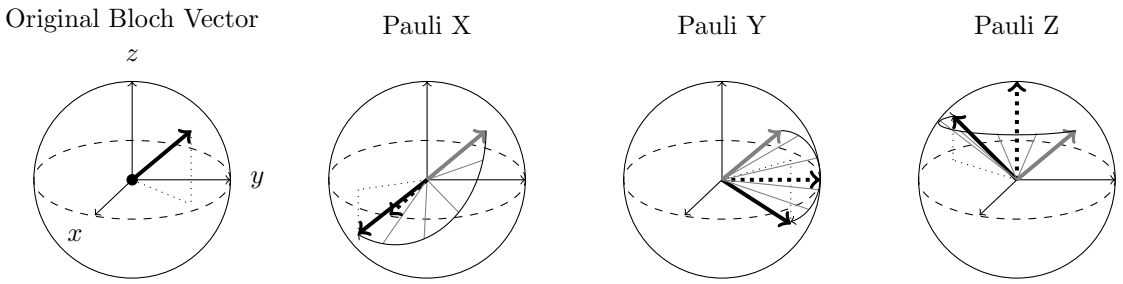


Figure 2.1.4.: An example of how Pauli gates influence the Bloch vector. Each gate rotates the Bloch vector a half around the respective coordinate. The thick line indicates the Bloch vector, the gray line its original position and the dotted line the axis around which the Bloch vector is rotated.

An example of a Hadamard gate applied to the state $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ is:

$$H \cdot \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = \frac{1}{\sqrt{2}}(H|0\rangle + H|1\rangle) \tag{2.1.18}$$

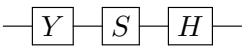
$$= \frac{1}{\sqrt{2}}\left(\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) + \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)\right) \tag{2.1.19}$$

$$= \frac{1}{\sqrt{2}}\left(\left(\frac{1}{\sqrt{2}} + \frac{1}{\sqrt{2}}\right)|0\rangle + \left(\frac{1}{\sqrt{2}} - \frac{1}{\sqrt{2}}\right)|1\rangle\right) \tag{2.1.20}$$

$$= \frac{1}{\sqrt{2}}\left(\frac{2}{\sqrt{2}}|0\rangle\right) \tag{2.1.21}$$

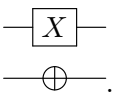
$$= |0\rangle\,. \tag{2.1.22}$$

To depict single qubit gates in a quantum circuit, we use lines to depict qubits, and labeled boxes with the symbols of the gates to depict gates. E.g., the circuit



applies first a Pauli-Y gate, then an S gate, and then a Hadamard gate to the qubit.

The Pauli-X gate has two common depictions:



.

The first three gates given in Table 2.1.1 are the Pauli gates. They are the application of the three Pauli matrices (see Equation (2.1.13)) on the vector representation of the qubit (see Equation (2.1.1)).

The Pauli gates rotate the Bloch vector representing the qubit half a circle around their respective axis in the Bloch sphere: The Pauli-X gate rotates the vector around the x-axis, the Pauli-Y gate around the y-axis, and the Pauli-Z gate around the z-axis. This is depicted in Figure 2.1.4.

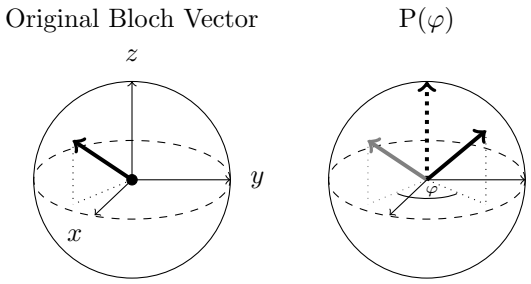


Figure 2.1.5.: An example of how a phase gate $P(\varphi)$ is applied to the Bloch vector and rotates the vector φ radians around the z-axis. The thick line indicates the Bloch vector, the gray line its original position and the dotted line the axis around which the Bloch vector is rotated.

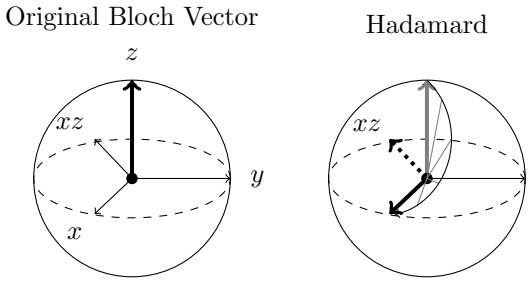


Figure 2.1.6.: An example of how a Hadamard gate is applied to the Bloch vector and rotates the vector. The rotations is done a half around the xz-axis (dotted line).

The Pauli-X gate can be interpreted as NOT-gate, because it swaps the coefficients of $|0\rangle$ and $|1\rangle$. The multiplication of two equal Pauli gates results in $\mathbb{1}$, meaning that two equal Pauli gates after each other cause a qubit to be in its initial state.

The general phase shift gate $P(\varphi)$ takes a parameter φ and adds a phase $e^{i\varphi}$ to $|1\rangle$. This is equivalent to rotating the Bloch vector φ radians around the z-axis, parallel to the xy-plane. This is shown in Figure 2.1.5.

The parameters $\varphi = \frac{\pi}{4}, \frac{\pi}{2}, \pi$ describe special cases of the phase gate. $\varphi = \pi$ is the already mentioned Pauli-Z gate. $\varphi = \frac{\pi}{2}$ describes the S-gate. It rotates the Bloch vector a quarter around the z-axis. $\varphi = \frac{\pi}{4}$ describes the T-gate. It rotates the Bloch vector one eighth around the z-axis.

The matrix representations of the gates are

$$P(\varphi) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\varphi} \end{pmatrix}, \qquad S = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{2}} \end{pmatrix}, \qquad T = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{4}} \end{pmatrix}. \tag{2.1.23}$$

Table 2.1.2.: Truth table of a CNOT gate. The target qubit is flipped if the control qubit is $|1\rangle$.

Control (Input)	**Target (Input)**	**Control (Output)**	**Target (Output)**
$\|0\rangle$	$\|0\rangle$	$\|0\rangle$	$\|0\rangle$
$\|0\rangle$	$\|1\rangle$	$\|0\rangle$	$\|1\rangle$
$\|1\rangle$	$\|0\rangle$	$\|1\rangle$	$\|1\rangle$
$\|1\rangle$	$\|1\rangle$	$\|1\rangle$	$\|0\rangle$

The Hadamard gate puts the basis states $|0\rangle$ and $|1\rangle$ into superposition. It can be interpreted as a rotation around an axis in the xz-plane that has an equal distance from x- and z-axis. This can be seen in Figure 2.1.6. Its matrix representation is

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}. \tag{2.1.24}$$

As $H \cdot H = \mathbb{1}$, two Hadamard gates after each other cause a qubit to be in its initial state.

In addition to single qubit gates, a universal set of quantum gates requires at least one multi-qubit gate. One of the most common multi-qubit gates is the controlled NOT (CNOT) gate. Its truth table can be found in Table 2.1.2. A CNOT gate is applied to two qubits: A target and a control qubit. The target qubit is negated if the control qubit is $|1\rangle$ and left unchanged if the control qubit is $|0\rangle$.

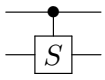
It is represented in a quantum circuit like this:

 .

The two horizontal lines are the qubits the CNOT gate acts on. The black dot indicates the control qubit, the bigger circle the target qubit.

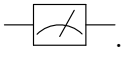
A universal set of quantum gates contains the Hadamard gate, phase gate, and CNOT [17, chapter 4.5.3].

One can add control qubits to arbitrary gates. For example, the circuit



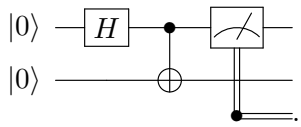
applies an S gate to the second qubit if the first qubit is $|1\rangle$.

The symbol to indicate a qubit measurement is

 .

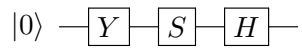
Quantum Circuits

To perform a quantum calculation, the different quantum gates have to be put together into a *quantum circuit*. One example of a circuit like this is:



The single horizontal lines represent qubits and the double lines classical bits. In this circuit, a Hadamard gate is applied to the first qubit. Afterward, a CNOT gate is applied between both qubits and at the end the first qubit is measured. The result of the measurement is saved in the classical bit of the circuit.

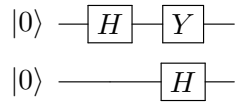
A quantum circuit consisting of gates (not measurements) can be depicted as a formula. For this, the matrix representations of the gates are used. The gates that act on the qubits first (left in the circuit) are written right in the formula, i.e. closest to the qubit in the formula. For example, the circuit



is represented by

$$\mathrm{H} \cdot \mathrm{S} \cdot \mathrm{Y} \left|0\right\rangle . \tag{2.1.25}$$

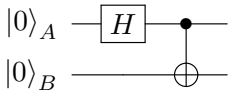
If a circuit consists of multiple qubits, a tensor product between two gates at the same time on different qubits is used. The identity $\mathbb{1}$ is used to indicate that no gate is applied to a qubit. E.g.



is represented by

$$(\mathrm{Y} \otimes \mathrm{H}) \cdot (\mathrm{H} \otimes \mathbb{1}) \left|00\right\rangle . \tag{2.1.26}$$

A subscript can be used to indicate which gates act on which qubits:

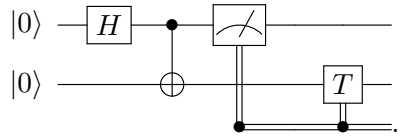


is represented by

$$\mathrm{CNOT}_{AB} \cdot (\mathrm{H}_A \otimes \mathbb{1}_B) \left|00\right\rangle . \tag{2.1.27}$$

An important part of quantum computing is classical feedback, e.g. classically controlled gates. As the name suggests, these are quantum gates which are only applied if a

classical bit is set to 1. These gates are especially useful if the classical bit has received its value by a previous measurement, as in this circuit:



In this circuit, the first qubit is measured after a Hadamard- and CNOT-gate. Afterwards, a T-gate is applied to the second qubit if the measurement outcome of the first bit is 1. Measurements write classical bits and other gates read classical bits. The quantum circuit notation does not differentiate between reading and writing classical bits.

Gottesman-Knill Theorem

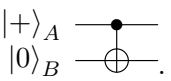
An important set of quantum gates are the elements of the Clifford group [26], the so-called *Clifford gates*. These are gates that normalize the Pauli group, i.e. they map Pauli basis states to Pauli basis states. A set of quantum gates that generate the Clifford group are the Hadamard gate, the S gate and CNOT [23]. The Pauli gates are Clifford gates as well by definition.

Clifford gates are interesting to us because of the Gottesman-Knill theorem [23]: It states that any quantum circuit that consists only of Clifford gates, preparation of qubits in computational-basis states and measurements in said basis can be simulated efficiently on a classical computer. This means that we have to use non-Clifford gates in calculations for a chance to gain quantum advantage. Any quantum algorithm that consists of Clifford gates only can better be executed on a CPU than on a QPU, as CPUs are much less error-prone and bits much more stable than qubits.

We will use this theorem for optimization routines in Chapter 5. For some of the optimization routines, it is relevant to know which quantum gates can be simulated efficiently by a classical computer.

Entanglement

Entanglement is a qubit property that causes the measurement of one qubit to influence another qubit's state. For example, we will look at a qubit that is in a superposition between $|0\rangle$ and $|1\rangle$: $|+\rangle_A = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and use it as control of a CNOT gate:



As already established, the CNOT puts qubit B to $|1\rangle_B$ if qubit A is $|1\rangle_A$. However, in this scenario, qubit A is both $|0\rangle_A$ and $|1\rangle_A$. Therefore, the CNOT puts qubit B into a superposition that depends on qubit A. If qubit A is measured to be a $|0\rangle_A$, qubit B collapses to the state $|0\rangle_B$ as well and will be measured as such. The reverse is the case if qubit A is measured to be a $|1\rangle_A$. It is also possible to measure qubit B first, causing

qubit A to collapse to either $|0\rangle_B$ or $|1\rangle_B$. This happens because the qubits are in an entangled state.

From a mathematical point of view, an *entangled state* is a state for which no factorized product can be found. A state for which a factorized product can be found would be called a *separable state*. No existing factorized product is the mathematical definition for an entangled state.

We will examine what this means. The mathematical expression of the above circuit is

$$\text{CNOT} \cdot (|+\rangle_A \otimes |0\rangle_B) = \text{CNOT} \cdot \left(\frac{1}{\sqrt{2}} (|00\rangle_{AB} + |10\rangle_{AB}) \right) \tag{2.1.28}$$

$$= \frac{1}{\sqrt{2}} (|00\rangle_{AB} + |11\rangle_{AB}) \tag{2.1.29}$$

$$= \frac{1}{\sqrt{2}} \Big(|0\rangle_A \otimes |0\rangle_B \Big) + \frac{1}{\sqrt{2}} \Big(|1\rangle_A \otimes |1\rangle_B \Big). \tag{2.1.30}$$

In Equation (2.1.30), the states of qubit A and B cannot be factorized to a form of $|\psi\rangle_A \otimes |\phi\rangle_B$. If that was the case, the equation would show a *separable* or *product state*. An example for a separable state is

$$\frac{1}{\sqrt{2}} (|00\rangle_{AB} + |01\rangle_{AB}) = |0\rangle_A \otimes \frac{1}{\sqrt{2}} (|0\rangle + |1\rangle)_B \tag{2.1.31}$$

with $|\psi\rangle_A = |0\rangle_A$ and $|\Phi\rangle_B = \frac{1}{\sqrt{2}} (|0\rangle + |1\rangle)_B$.

In an entanglement like Equation (2.1.30), physically, the first measurement "decides" on one of the factors for all of the qubits in the equation. If we measure $|1\rangle_A$ for qubit A, the coefficient for $|0\rangle_A$ becomes zero. Therefore, it becomes mathematically impossible for the coefficient of $|0\rangle_B$ to be something else than zero, and qubit B collapses to $|1\rangle_B$. In general it is an NP-hard problem to determine whether a quantum state is entangled or not [27].

No-Communication Theorem

Entangled states have caused discussion in physics, as the measurement on one qubit instantaneously causes the other qubit's state to collapse, seemingly transferring the information about a measurement faster than the speed of light. However, it is consensus in physics that any communication faster than the speed of light is impossible [28]. This problem is solved by the fact that one cannot tell by measurement of one entangled qubit whether the other entangled qubit(s) have already been measured [29, II.E]. A classical communication channel (e.g. via internet) is needed to check that the qubits' measurement results correlate. This classical communication is again limited by the speed of light.

The *no-communication theorem* is a theorem restricting the information exchange between entangled qubits. As information exchange is restricted by the speed of light, quantum particles should not be able to exceed this principle, even if they are entangled. The no-communication theorem underlines this. It says that even if two qubits

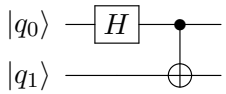


Figure 2.1.7.: A quantum circuit to create one of the four Bell states. Depending on the initial states of $|q_0\rangle$ and $|q_1\rangle$, one of the four Bell states is created (see Table 2.1.3).

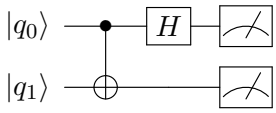


Figure 2.1.8.: A quantum circuit to measure one of the four Bell states. Depending on the Bell state, one of the four combinations of $|q_0 q_1\rangle$ is measured (see Table 2.1.3).

are entangled, local operations on only one qubit cannot affect the statistics of the measurement on the other qubit [29, II.E]. This means: If two people (Alice and Bob) both have a qubit of one entangled system, Alice cannot receive information from Bob by only working on her local qubit, no matter what Bob does with his qubit.

In a quantum computer, this is relevant for some optimization operations, as we have to know which qubits can influence other qubits in successive operations.

Bell States

The *Bell-states* [17, chapter 1.3.6] are four important examples of entangled 2-qubit states. We will look at them as their creation and measurement is relevant in some quantum circuits, including one we will look at in Section 4.2.1.

The Bell states are defined as:

$$|\Phi^+\rangle = \frac{1}{\sqrt{2}}\left(|00\rangle + |11\rangle\right) \tag{2.1.32}$$

$$|\Phi^-\rangle = \frac{1}{\sqrt{2}}\left(|00\rangle - |11\rangle\right) \tag{2.1.33}$$

$$|\Psi^+\rangle = \frac{1}{\sqrt{2}}\left(|01\rangle + |10\rangle\right) \tag{2.1.34}$$

$$|\Psi^-\rangle = \frac{1}{\sqrt{2}}\left(|01\rangle - |10\rangle\right). \tag{2.1.35}$$

Figure 2.1.7 shows a circuit that creates a Bell state. Depending on the initial states of $|q_0\rangle$ and $|q_1\rangle$, one of the four Bell states is being created. The created Bell states depending on the initial values can be taken from Table 2.1.3.

Figure 2.1.8 shows the circuit to measure a Bell state. The measured $|q_0 q_1\rangle$ corresponding to the Bell states can be seen in Table 2.1.3.

Table 2.1.3.: Initial and measured values of $|q_0 q_1\rangle$ if a Bell-state is initialized/measured by the circuits in Figures 2.1.7 and 2.1.8.

Initial $\|q_0 q_1\rangle$	Bell-State	Measured $\|q_0 q_1\rangle$
$\|00\rangle$	$\|\Phi^+\rangle$	$\|00\rangle$
$\|10\rangle$	$\|\Phi^-\rangle$	$\|10\rangle$
$\|01\rangle$	$\|\Psi^+\rangle$	$\|01\rangle$
$\|11\rangle$	$\|\Psi^-\rangle$	$\|11\rangle$

2.1.3. Quantum Hardware

Creating quantum devices with arbitrary many qubits that stay stable for an arbitrary long time is one of the goals of quantum hardware developers.

Currently, we are in an era that Preskill called the noisy intermediate-scale quantum (NISQ) era [30]. NISQ QPUs are characterized by 50 to a few hundred qubits. The application of gates in these QPUs is error-prone, meaning the intended applied gate and the actual applied gate can differ. Additionally, the qubits stay stable for only a limited amount of time. NISQ QPUs can be useful for proof-of-concepts and may surpass CPUs in some regards, but the ultimate goal is to leave the NISQ era in the foreseeable future.

Decoherence in Quantum Devices

One challenge on the way to surpass the NISQ era is quantum decoherence. This is the process of qubits loosing information to the environment, i.e., they become *decoherent* over time [31]. The *coherence time* is the time a qubit holds its superposition (i.e. state) [32, chapter 4.1] and stays useful for calculations.

Another important parameter is the *fidelity*. Fidelity is a metric about the "closeness" between two quantum states. Operation fidelity shows how successful quantum gates and quantum measurements on qubits are. This means it quantifies how close gates and measurements are to the ideal operation [17, chapter 9.3].

Decoherence and low-fidelity are two factors that pose a challenge in QPU development.

IBM's Marrakesh [4], Fez [3] and Torino [5] have a coherence time of around 100 µs. Their error rates are between 10^{-2} and 10^{-4}, with read-out being especially error-prone and single qubit gates being most robust. IonQ's Aria [6] has a coherence time of around 1000 ms and error rates between 10^{-3} and 10^{-4} [6]. The error rate of CPUs is much lower and usually negligible.

DiVincenzo [33] formulated five requirements that have to be met for a quantum computer that can offer quantum advantage (cf. Chapter 1). The QPU needs to ...

1. consist of scalable, well-defined qubits.
2. be able to put all qubits into a well-defined start state, e.g. $|0\rangle$.
3. have small error rates. How small exactly depends on the error correction abilities. The errors should be corrected faster than they come up, or at least fast enough to calculate all desired quantum circuits.
4. be able to apply an universal (functionally complete) set of unitary transformations (gates) onto arbitrary qubits.
5. be able to measure the qubits in an orthogonal basis, e.g. $|0\rangle$ and $|1\rangle$.

Requirement 1 and 3 are particularly tricky to fulfill at the same time, as error rates on qubits typically increase with the number of qubits in the QPU [8].

Concrete Hardware Implementations

There are different ways to physically realize a QPU. For example: Superconducting qubits, realized by IBM [19] and Google [34], ion-trapped qubits, realized by IonQ [20], trapped neutral atoms, realized by Atom Computing [35], qubits created by nitrogen-vacancy centres in diamonds [36], and photonic qubits [37].

The different hardware technologies have different advantages and disadvantages. For example, superconducting qubits are sensitive to decoherence and therefore need to be cooled down to a temperature close to 0 K. Additionally, qubits in superconducting QPUs influence each other easily [38]. On the other hand, superconducting qubits are implemented by macroscopic electric circuits, while other qubit technologies use microscopic quantum particles. This leads to advantages in coupling qubits and producing QPUs [39]. Ion trap qubits have a rather long coherence time and a good fidelity, but the operations on the ion qubits are comparably slow [40] (see the following subsection). Photonic qubits are fast and scalable, but rather sensitive to noise [40]. Nitrogen-vacancy centres in diamonds create qubits with long coherence times. Additionally, these kind of QPUs can be operated at room temperature. On the other hand, the behavior of these qubits is difficult to predict [40].

QPU Speeds

While QPUs based on different technologies have different calculation speeds, they are generally all slower than a CPU. IBM indicates the speed of their QPUs in circuit layer operations per second (CLOPS). CLOPS say how many layers of operations can be applied to the qubits in one second [41]. It can be roughly compared to the tact frequency of classical CPUs, at least in our case, as we will end up at several orders of magnitude difference.

Typical clock frequencies of an Intel processor are in the GHz range [42]. Meanwhile, IBM's superconducting QPUs Marrakesh [4], Fez [3] and Torino [5] have around a few hundred kCLOPS. IonQ's trapped ion QPU Aria [6] takes 135 µs for one-qubit gate operations and 600 µs for two-qubit gate operations. This results into a rate in the order of magnitude of a few kHz.

Executing Gate-Based Quantum Hardware

There are two different state-of-the-art execution models for QPUs: *Gate-based* and *annealing*.

In this work we only consider *gate-based* quantum hardware, i.e. QPUs that apply gates on qubits to receive a result. Gate-based QPUs typically have a set of native gates, i.e. gates that they can directly execute [43]. The QPUs can only execute circuits consisting of their native gates. This is why the circuits need to be transpiled before submitting to QPUs. This is a functionality that is provided by many QPLs (cf. Section 4.1.2).

A quantum circuit can be executed by submitting it to a QPU. Notable vendors are, e.g., IBM [19], Amazon Braket [44], or IonQ [45]. A difficulty for direct programming of quantum hardware is that the vendors do not publicly share their QPUs' architectures. After submission of the circuit, the execution is not transparent to the quantum developer.

Next to gate-based hardware there is also annealing hardware [46]. A notable vendor is D-Wave [47]. When using such hardware, quantum annealing is used to find the minima of functions. Qubits are initially put into superpositions between $|0\rangle$ and $|1\rangle$. The function that needs to be minimized is slowly applied to the qubits. Values of the qubits that correspond to small function values are made to be more energetically favorable. As physical systems naturally strive to states of low energy, the qubits will end up in a state that corresponds to the minimum of the function.

2.2. Distributed Computing

Van Steen and Tanenbaum define: "A distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system." [48]. These computing elements can be a hardware device or a software.

To appear as a single system, the computing elements need to collaborate with each other. Ensuring that the collaboration works error-free is one of the most important tasks in computing distributed systems.

We carefully distinguish between hybrid and distributed computing. Hybrid computing is computation using both classical and quantum components, while distributed computation is what Van Steen and Tanenbaum defined [48] and does not need to contain quantum devices.

Examples for distributed computing are high performance computing (HPC) clusters, distributed information systems, or pervasive systems. It should be noted that there

is a difference between distributed and parallel computing: Distributed systems consist of multiple computing nodes connected through a network and do not share main memory. Parallel computing/processing uses multiple processors that access the same memory [48].

Distributed systems can be created by different communicating hardware components, like CPUs and GPUs. This kind of system can be found in high performance clusters.

In the future, QPUs could be a component in distributed systems as well. In the next section, we will discuss how QPUs can be added into a distributed system.

2.2.1. Distributed Quantum Computing

A QPU is much more error-prone than a CPU and more difficult to build and run (cf. Section 2.1.3). This is one of the reasons why QPUs will not "replace" CPUs in the foreseeable future. Instead, QPUs will probably become a new hardware component in distributed systems, similar to GPUs. In such systems, a QPU can be used to calculate specific problems and gets its instruction by (a) main CPU(s).

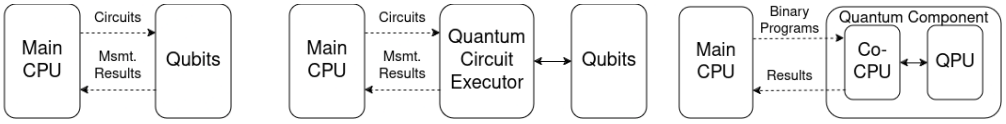
However, the communication between QPU and CPU would be too slow to happen during the coherence time of the qubits. IBM's Marrakesh [4], Fez [3] and Torino [5] have a coherence time of around 100 µs, while IonQ's Aria [6] has one of around 1000 ms.

Fu et al. [49] give a calculation example for the reaction time of a CPU: A CPU that controls the qubits and is not close to the QPU has to communicate with the QPU in some sort of way. The communication via bus or ethernet is not possible without latency. The communication alone can take milliseconds. Additionally, due to the operating system and scheduling, there is no guarantee as of how long the CPU takes to process requests for a QPU.

A communication time of several milliseconds makes the distributed computation between CPUs and QPUs a non feasible option for the current superconducting QPUs' coherence times. This is particularly inconvenient, as superconducting QPUs are one of the most advanced technologies for quantum computation at the moment. In theory, ion-trapped QPUs have a coherence time long enough to wait for communication lasting milliseconds. However, ion-trapped QPUs are still in early stage of development compared to superconducting QPUs, and do not offer as many qubits to work on [50].

Therefore, it is worthwhile to consider how to work with *real-time* feedback that is not possible between a QPU and a (main) CPU. Real-time communication is communication between CPU and QPU during the coherence time of a qubit, e.g. applying gates to qubits depending on the output of a measurement. This kind of quantum communication allows to dynamically adjust a quantum circuit depending on measurements. Therefore, this kind of computation is called *dynamic quantum circuit calculation* [51].

Dynamic quantum circuit computing is needed for some algorithms and can be used to reduce quantum resources of a calculation. Examples for dynamic circuit algorithms are quantum teleportation [52], active reset [53], magic state distillation [54, 55, 56], repeat-until-success [57, 58], iterative phase estimation [49, 59], and Shor's algorithm for $2n + 1$ qubits [60, 61]. Some of these algorithms are explained in Section 4.2.1.



(a) QRAM model (b) Restricted HQCC model (c) Refined HQCC model

Figure 2.2.1.: The execution models for HQCC (from [49, Fig. 2]). Dotted lines indicate slow communication (i.e. communication that takes longer than the coherence time of the qubits), continuous lines indicate fast communication (shorter than qubit coherence time).

Currently, the possibility to do dynamic circuit calculations on real quantum hardware is very restricted, even though vendors seem to be aware of the demand. IBM points out "several considerations and limitations to be aware of" if one uses classical feed forward and control flow, namely limited working memory and latency of the classical computation [62]. Amazon's Braket only supports classical operations and control on their `LocalSimulator`, not on the QPUs [63, p. 67], which are provided by IonQ, IQM, QuEra and Rigetti [64].

To implement dynamic circuit calculations, we do not only need the appropriate hardware, but also a QPL that allows quantum-classical calculations. Fu et al. [49] summarized three different execution models a QPL can be designed on with respect to quantum-classical calculations. The models are depicted in Figure 2.2.1.

We will now look at the properties of the three different architectures.

The *QRAM model* (see Figure 2.2.1a) has been proposed by Knill [65] in 1996. In this model, a CPU controls qubits by applying operations in form of circuits on them. The CPU can do measurements on the qubits and gets the measurement results. With the CPU, only "slow" communication (longer than the coherence time) is possible.

The *restricted heterogeneous quantum-classical computation (HQCC) model* puts a hardware component between CPU and qubits [49]. This hardware component is a quantum circuit executor, which can apply a *fixed* quantum circuit onto the qubits. This does not allow real-time feedback, but prevents the CPU from having to control the qubits itself. The quantum circuit executor can communicate with the qubits within their coherence time ("quickly").

The *refined HQCC model* upgrades the quantum circuit executor to a *quantum control processor* (co-CPU). This processor can apply real-time feedback on the qubits and execute classical instructions. The classical calculations are limited by the coherence time of the qubits, but not by the abilities of the quantum control processor. The quantum control processor and the qubits form a quantum component, which gets a quantum program from the main CPU and returns the results to said main CPU.

Both hardware and software should be moving towards the refined HQCC architecture, as it allows highest control and flexibility on quantum calculations. This is why we will look at how programs intended for the refined HQCC architecture can be optimized. In the remaining thesis, we will refer to the co-CPU as CPU and indicate when we are specifically talking about the main CPU. We will check to which extent current pro-

gramming languages already offer support for the refined HQCC model (cf. Chapter 4) and what kind of optimizations could be possible (cf. Chapter 5).

To be able to do so, we will first look at already established optimization routines done by classical compilers in the next section.

2.3. Compilation and Optimization Techniques

Aho et al. define a compiler as "a program that can read a program in one language – the source language – and translate it into an equivalent program in another language – the target language" [66].

A compiler typically consists of multiple phases. Two required phases are *analysis* and *synthesis*. The analysis phase creates an intermediate representation (IR) of the program and checks whether the source program aligns with the syntactic and semantic requirements of the source language. The synthesis phase creates the target language from the IR.

2.3.1. Abstraction Levels

Compilers typically compile from a high-abstraction programming language to a low-abstraction programming language [66, chapter 1.5]. "More abstract" means that more of the actual hardware functionality is "hidden" and not relevant for the programmer.

Su and Yan [67, chapter 1.2] differentiate between high-abstraction and low-abstraction by calling high-abstraction mankind-oriented and low-abstraction machine-oriented. They describe Assembly and machine instruction sets as low-level (low abstraction). They call the common feature of high-level (high abstraction) languages that "they broke away from the restriction of the computer instruction set" [67, chapter 1.2].

It is rather challenging to formally define abstraction levels, and to make a fine-grained decision about which languages are "how" abstract (in a quantitative sense). Classical programming languages can use different abstraction options, e.g. instructions, functions, objects, or first class functions. Typically, language developers choose an as coherent as possible set of abstractions for their languages, which provides a consistent picture of the language's abstraction level. QPLs most commonly work with the application of gates to a program, as we will see in Chapter 4, and do not have such clear differences between the abstraction level as classical programming languages do.

2.3.2. Optimization Steps

One or multiple optimization phases are optional phases for a compiler. A typical sequence of phases in a compiler is given in Figure 2.3.1. It shows that during the synthesis of a program, a machine-independent and machine-dependent optimization phase can take place. The effectiveness of the optimizations can be evaluated against different metrics, for example wall-time or number of instructions.

A way of analyzing a program for different compilation steps is to do a control flow analysis [68]. This can be done using a control-flow graph (CFG). A CFG represents all

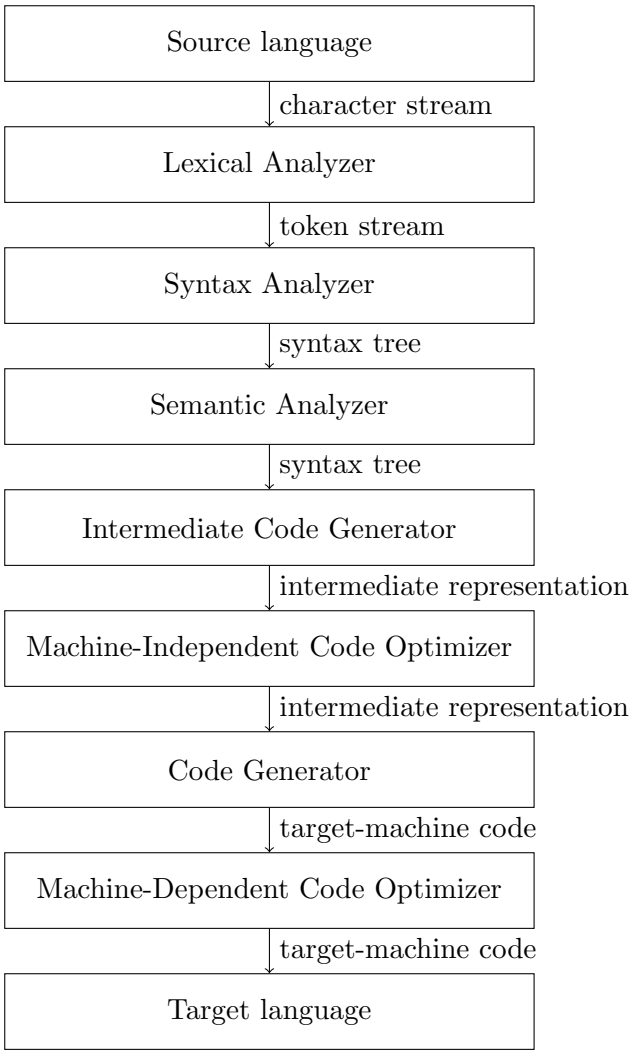


Figure 2.3.1.: Typical phases of a compiler [66, fig. 1.6].

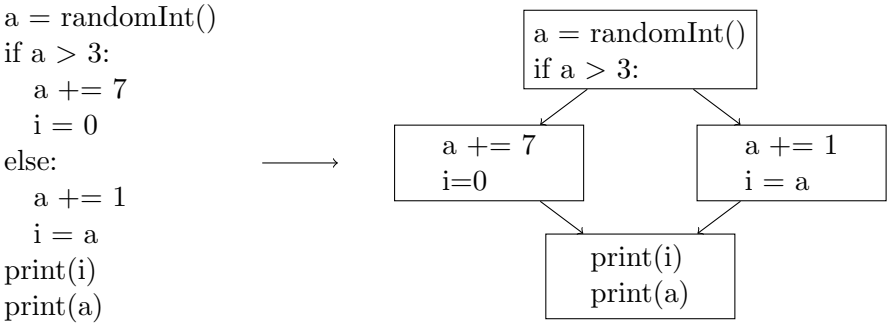


Figure 2.3.2.: An example for the creation of a CFG.

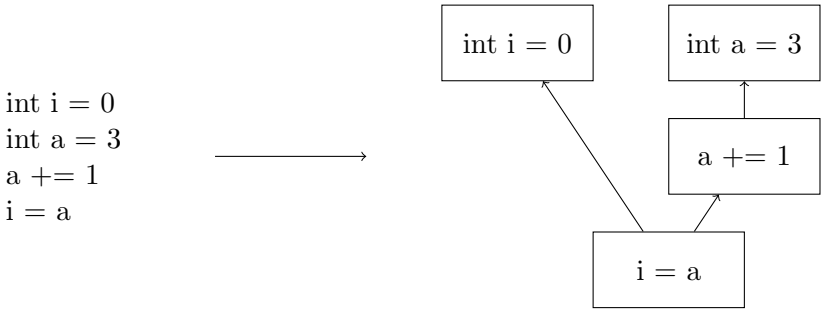


Figure 2.3.3.: An example for the creation of a DDG.

possible execution paths of a program. The directed edges of the graphs represent jumps. The nodes describe a basic block, which are "sequences of statements that are always executed one-after-the-other, with no branching" [66, chapter 2.8.1]. This means that a jump on the current instruction as well as a label (jump target) at the next instruction ends a basis block [66, chapter 8.4.1]. An example for the creation of a CFG is given in Figure 2.3.2.

Most optimization techniques depend on data-flow analysis [66, chapter 9.2]. One way to do data-flow analysis is to create a data-dependence graph (DDG)[1] [69, chapter 5.3.2] of the program. A DDG is a directed graph. Nodes represent basic blocks (or, in our case, single instructions) of a program. An instruction A is a successor of instruction B in the graph, if A has to be executed after B in order for the program to be correct, e.g. if A and B access the same variable. Edges only appear between instructions/nodes if no other instruction C has to be executed between A and B. An example for a DDG created from code can be seen in Figure 2.3.3.

To optimize code, a set of optimization methods are applied on the code. The operations change the instructions of the program without changing the semantics. This is done to improve the program with respect to one or several metrics [66].

The operations are divided into analysation and transformation. *Analysations* are operations that extract information about the program, but do not change it. *Transformations* change the program. Typically, transformations use the information analysations extracted.

The optimization operations we look at are given in Table 2.3.1 and taken from [66]. Optimization techniques targeted at loops are out of scope for this work and left out.

Available expressions is about checking whether an expression $a \bullet b$ is available at a program point p. An expression is available if it is evaluated on every path between the start node and p. Additionally, there must not be new assignments to a or b between the last evaluation and p.

Constant-propagation means to check whether a variable holds a unique constant value at a program point p.

[1] Also called program-dependence graph [66, chapter 11.8.2].

Table 2.3.1.: Analysation and transformation techniques to optimize code, taken from [66].

Analysation operations	Transformations
Available expressions	Constant folding
Constant propagation	Copy propagation
Live-variable analysis	Dead code elimination
Reaching definitions	

Live-variable analysis is used to determine which variables are still "in use" (alive) at a point p of the program. If there is any usage of a variable's value in the DDG between p and the program halt, it is alive. Otherwise, it is dead. Values of dead variables have no influence on the remaining program, and they do not need to be considered any further.

Reaching definitions is used to determine which definitions of a variable can reach a program point p.

Constant folding is evaluating constant expressions and replacing the expressions by their values.

Copy propagation checks when a copy statement (e.g. `a = b`) is given, whether a can be removed and b be used instead of a. Copy propagation is especially useful followed by dead code elimination, as it can turn copy statements into dead code.

Dead code elimination removes "dead" (or useless) code from the program. This affects unreachable code as well as code that computes values which are never used.

Most QPLs currently have a very low abstraction level, as we will see in Chapter 4. Programmers program by deciding which gate(s) to apply in which order onto which qubit.

Research on optimization routines for quantum programs is usually restricted to the quantum circuit of the program [10]. A main goal of these optimizations is to reduce the quantum resources, e.g. by reducing the qubit count, the circuit depth or the two-qubit gate count.

The quantum circuit optimization routines generally transform the quantum circuit and work with the mathematical properties of the applied gates. For example, gates whose respective matrices commute can be swapped. Alternatively, if it holds: $AB = BC$, the gate corresponding to A can be swapped with the gate B if it is changed to the gate corresponding to C. Additionally, the optimization routines are used to find an optimal mapping of the programmed quantum circuit to the hardware, e.g. to have minimal distance between qubits that have multi-qubit gates applied onto them simultaneously or to minimize error rates.

When programming a refined HQCC architecture, we have hardware that has inherent hybrid characteristics (the CPU and the QPU). The compilation and optimization stages for programs on this devices have to consider this hybrid nature to do more efficient optimization. We will examine this requirement and the optimization possibilities for this device in the remaining thesis.

3. Related Work

Optimizing quantum circuits is an active field of research [10, 11, 12, 13]. First approaches have been done in the 2000s [70, 71]. In recent years, the field as gotten more attention due to the grown availability of functioning quantum hardware.

Optimization procedures typically work with the mathematical properties of the matrices representing quantum gates. The gates can be swapped, removed or exchanged. Additionally, a hardware-dependent optimization becomes necessary when the quantum circuit is supposed to be executed by physical hardware. In this case, the qubits on the programmed circuit need to be mapped to the physical qubits on the QPU. For the mapping, parameters like error rates of single qubits and the distance between qubits need to be considered.

In this thesis, we want to examine another approach. Our optimization procedures are orientated on optimization procedures of classical compilers (cf. Chapter 5) and not on the mathematical properties of quantum gates.

The optimization of quantum circuits brings up NP-hard problems, like T-count optimization [72], a fault-tolerant implementation of topological error-correction [73], or parameter optimization with specific requirements on the circuit [74]. Research about quantum circuit optimizations usually does not consider the interference between classical and quantum components of a larger computing system. This is something intended to do in this thesis.

There have been multiple surveys and reviews about the current state-of-the-art of quantum computing.

Jimnez-Navajas et al. [75] did a survey asking which quantum programming tool (QPT) quantum researchers and developers use during the quantum software lifecycle. They found that quantum-classical hybrid software is most commonly created by using Python that implements classical calculations alongside quantum circuits. We will see in Section 4.1 that many current quantum programming languages support these implementations, as they are often Python frameworks themselves.

Elsharkawy et al. [76] examined how current programming tools can be used to integrate quantum computing into an HPC environment. They categorize the potential of different QPTs to integrate QPUs into HPC environments. However, the review does not look at the optimization measures of the QPTs.

Barral et al. [77] did an extensive review on the aspects of distributed quantum computing. They describe the relevant components from hardware to software, and look at the current state-of-the-art. The survey examines optimization of distributed quantum software, but not with respect to quantum-classical hybrid calculations.

Multiple QPTs have been published to support the implementation of quantum calculations in hybrid environments. We will examine today's QPLs in detail in Chapter 4.

Here, we will only look at some tools that are not mentioned or used in other parts of this thesis:

Tweedledum [78] is an open-source compiler companion for quantum computation, written in C++. The developers of tweedledum intend to support more abstract QPLs with it. It can translate classical logic functions into quantum circuits. An important part of tweedledum is its IR, which is used to support different abstraction levels of one quantum circuit. The last commit in tweedledum's GitHub repository was in November 2022 [79] (as of February 2025).

Qualtran [80] is an open-source Python library that can be used to analyze and present quantum algorithms. It can create diagrams, e.g. circuit diagrams or compute graphs and estimate the resources a program needs. Qualtran supports classical logic, but classical data (e.g. from measurement results) is not yet considered in the compute graph. This is a feature intended for future releases [81].

Quantum Intermediate Representation for Optimization (QIRO) [82, 83] is an multi-level intermediate representation (MLIR) that has been designed to find quantum and classical data dependencies, as well as determining the control flow. The authors bring up the idea that QIRO could enable quantum-classical co-optimization. The GitHub QIRO project has not been updated since November 2022 [84] (as of February 2025).

4. Evaluation of Today's Quantum Languages

In recent years, many different embedded and stand-alone DSLs for quantum computing have been developed. The embedded DSLs are not a regular language with syntax. We will still refer to them as language for the sake of simplicity. We want to examine which optimization steps for quantum-classical heterogeneous architectures the languages provide.

The support of QPLs for a HQCC architecture, as well as general quantum-classical programming, differs. In this section, we will first introduce different quantum languages and compare them by their properties (cf. Section 4.1). Afterwards, we will look at the support of a subset of these languages for quantum-classical computation in Section 4.2.

4.1. Properties of the Quantum Languages

We want to look at the properties of some QPLs in more detail. The QPLs we assess are taken from Elsharkawy et al.'s review of quantum programming tools [76], Barral et al.'s review of distributed quantum computing [77] and popular languages from Jimnez-Navajas et al.'s survey [75], i.e. languages that were used by more than 10 respondents. Additionally, we will look at the language Silq (0.0.39) [100], which we found during our research. It has been developed as a more high-level programming language than the other languages we look at. As "[h]igh-level language constructs can introduce substantial run-time overhead if we naively translate each construct independently into machine code" [66, chapter 9], optimization becomes especially important for high-level languages. This makes Silq interesting for our research question, as we want to find optimization steps languages provide for quantum-classical computation.

This results into the QPLs given in Table 4.1.1.

4.1.1. Introduction of Known Quantum Programming Languages

Braket [44] is a cloud quantum computing service by Amazon. It includes a Python software development kit (SDK) and a service to execute quantum circuits on quantum hardware or simulators. The Python framework is a gate programming language (GPL), meaning a developer has to individually decide which gates to apply to qubits. One can submit a quantum circuit to the Braket cloud service. Additionally, the Amazon cloud service can be addressed via Qiskit, PennyLane and OpenQASM.

Cirq [101] is an open-source Python framework developed by Google Quantum AI. It is a GPL and can be run on QPUs or QPU simulators. A developer defines a circuit,

Table 4.1.1.: Comparison of Quantum Computing Languages with respect to the implementation type, developer, and version. The developers are categorized as major tech company (MTC), quantum tech company (QTC), or uni/research institute (U/R). If no version number/last public activity is given, the programming language has no official release. The last public activity has been checked in February 2025.

Language	**DSL Type**	**Developer**	**Version**	**Last Public Activity**
Braket	Python-Emb.	MTC (Amazon)	1.84.0	February 2025 [85]
Cirq	Python-Emb.	MTC (Google)	1.4.4	February 2025 [86]
CUDA-Quantum	Python/C++-Emb.	MTC (NVIDIA)	0.8.0	February 2025 [87]
D-Wave Ocean	Python-Emb.	QTC (D-Wave)	8.0.1	December 2024 [88]
InQuIR	Stand-Alone	U/R (Sōkendai)	-	February 2023 [89]
NetQASM	Stand-Alone	U/R (QuTech)	-	January 2025 [90]
OpenQASM 3	Stand-Alone	MTC (IBM)	3.1.0	January 2025 [91]
OpenQL	Python/C++-Emb.	U/R (QuTech)	0.12.2	January 2024 [92]
PennyLane	Python-Emb.	QTC (Xanadu)	0.38.0	February 2025 [93]
Q#	Stand-Alone	MTC (Microsoft)	1.8.0	February 2025 [94]
Qiskit	Python-Emb.	MTC (IBM)	1.2.4	February 2025 [95]
QMPI	C++-Emb.	MTC (Microsoft)	-	-
Quil	Stand-Alone	QTC (Rigetti)	2021.1	September 2024 [96]
QWIRE	Coq Impl.	U/R (Univ. of Pennsylvania)	-	December 2023 [97]
Silq	Stand-Alone	U/R (ETH Zürich)	0.0.39	February 2022 [98]
XACC	Python/C++-Emb.	U/R (Oak Ridge Nat. Lab.)	1.0.0	August 2023 [99]

which is optimized and transformed. The transformed circuit is sent to a quantum device (or simulator) afterward.

CUDA-Q [102] by NVIDIA is a platform for hybrid quantum-classical computing. It provides the `nvq++` compiler, which maps quantum expressions to a MLIR-based IR. The compiler transforms and optimizes the IR, before lowering it to a Low Level Virtual Machine (LLVM).

D-Wave Ocean [88] is a Python framework developed by D-Wave. It is not a GPL, instead, it implements quantum annealing (cf. Section 2.1.3), which makes it useful for optimization problems. D-Wave Ocean uses D-Wave's Advantage QPU [47], which can be accessed through an API.

InQuIR [103] is an IR for distributed quantum computing. It supports quantum and classical communication between different devices. It allows quantum communication and entanglement across devices. At the time of writing, InQuIR's GitHub repository was last updated in February 2023 [89].

NetQASM [104] is a Quantum Assembly Language (QASM) variant that supports distributed structure of quantum devices. It provides an instruction set architecture (ISA) for quantum network processing units (QNPUs), with quantum instructions as well as classical control and memory operations. The QNPUs are end-nodes in a quantum network, e.g. quantum clients and servers. Next to writing NetQASM code directly, one can use a Python SDK for programming NetQASM applications.

OpenQASM 3 [105] is an attempt to create a QASM. It build on OpenQASM 2 [106], but extends its functionality by the creation of real-time calculations, timing, pulse control, and gate modifiers. In this work, if we refer to OpenQASM we mean OpenQASM 3, unless stated otherwise. OpenQASM is designed to be used as IR, like Assembly in classical compilation procedures. It is a GPL. Arbitrary classical functions can be computed in addition to quantum circuits.

It is important to note that OpenQASM provides no own compilation or execution structure. However, e.g. IBM Quantum [107] or Amazon Braket [108] support it.

OpenQL [109] is a quantum programming framework for C++ or Python. It is a GPL. OpenQL provides two compilation stages: A hardware independent compilation to an IR, and a hardware dependent compilation from the IR. It supports different backend architectures and a developer can also specify a completely new backend.

PennyLane [110] by Xanadu is a Python quantum software framework. It is a GPL. A developer can create circuits and submits them to a backend hardware, e.g. IBM's or Rigetti's. New backend hardware can be added if desired. PennyLane allows simplifying and transformation of quantum circuit [111].

Q# [112] by Microsoft is part of Microsoft's Quantum Development Kit and an open-source, stand-alone quantum language. Q# is a GPL and its compiler performs optimizations on the code. It is possible to execute the code on a simulated QPU or on a real QPU, of which Microsoft provides some through Azure Quantum.

Qiskit [113] by IBM Quantum is a very well known open-source development kit for quantum hardware, embedded in Python. It claims to be the world's most popular quantum software.

Qiskit is a GPL and provides out-of-the-box circuits for certain algorithms directly. It can submit the quantum circuits to a real backend (or simulator). This functionality is mainly designed to connect to IBM's QPUs, but other hardware is possible as well.

Quantum MPI [114] is an extension to Message Passing Interface (MPI) [115] to support quantum hardware. MPI is a message-passing standard to support parallel computing architectures. It implements point-to-point and collective operations. According to [114], a C++ prototype of Quantum MPI has been implemented. However, to the best of our knowledge, no implementation of Quantum MPI is publicly available.

Quil [14] by Rigetti is an instruction set created for quantum-classical computation on an abstract machine architecture. It is an assembly-style low-level language and a GPL. Single quantum instructions are gates applied to one or multiple qubit(s).

Quil's developers provide some additional tools to work with: PyQuil [116], a Python library to generate and execute Quil code. Quilc [117], a compiler that compiles Quil circuits to circuits a defined hardware can execute. It also performs optimization. And the Quil-Lang quantum virtual machine (QVM) [118], that can simulate the execution of Quil code.

QWIRE [119] is a language focused on formally verifying quantum circuits. It implements the QRAM model. QWIRE was implemented in the Coq proof assistant [120] but can be embedded in other host languages as well. At the time of writing, the Coq implementation was last updated in December 2023 [97].

Silq [100] is a quantum language developed at the ETH Zürich. It is a comparably high-level quantum programming language: It works less with directly applying gates to qubits, though applying gates to qubits is still possible in this language. Silq allows logical operations on qubits (e.g. negations, conjugations) as well as conditional branching. Conditional branching on a superposition can be translated to the CNOT gate acting on a superposition. This means the conditional instructions only have an effect on the states which fulfill the condition. Silq does not yet provide a possibility to execute its code on real quantum hardware. It is only executed by a Silq interpreter.

XACC [121] is a quantum compilation framework. It is divided in a front- and backend as well as a middle layer. The frontend compiles quantum code to an IR. The middle layer optimizes and transforms the IR. The backend submits the code to an executing hardware. XACC supports IBM, Rigetti and D-Wave hardware, as well as some simulators.

XACC's main goal is device interoperability. Its backend can target gate-based and annealing QPU models. Its frontend code can be written in any language for which a XACC compiler exists, e.g. C++ or Python. At the time of writing, XACC's GitHub repository was last updated in August 2023 [99].

```python
from braket.circuits import Circuit

qc = Circuit()
qc.h(1)
qc.cnot(1, 0)
qc.measure(1)
```

Listing 4.1.1: Braket bell entanglement.

```python
from cirq import Circuit, LineQubit, H, CNOT, measure

qc = Circuit()
qubits = LineQubit.range(2)
qc.append(H(qubits[1]))
qc.append(CNOT(qubits[1], qubits[0]))
qc.append(measure(qubits[1]))
```

Listing 4.1.2: Cirq bell entanglement.

4.1.2. Comparison of Quantum Language Properties

In this section, we will compare the QPLs listed in the last subsection in more detail. For this, we focus on quantum languages developed for implementing quantum algorithms in gate-based quantum computers. This work's scope is the embedding of one QPU into a distributed, classical system. Therefore, we do not consider languages specifically designed to implement communication of multiple QPUs.

Due to these criteria, we remove the following languages from our comparison:

- **D-Wave Ocean**, as it is developed for quantum annealers.
- **QWIRE**, as it is a language to formally verify quantum algorithms, not to execute them.
- **QuantumMPI**, **InQuIR** and **NetQASM**, as they have been created to implement communication between QPUs.

This leaves us to compare Braket, Cirq, CUDA-Quantum, OpenQASM 3, OpenQL, PennyLane, Q#, Quil, and XACC.

Example implementations of a bell entanglement are shown in Listings 4.1.1 to 4.1.11. Of these eleven languages, seven are frameworks implemented in Python (and occasionally also C++). This kind of implementation has the advantage of re-using existing

```
from cudaq import QuantumCircuit, QuantumRegister

qr = QuantumRegister(2)
qc = QuantumCircuit(qr)
qc.h(1)
qc.cx(1, 0)
qc.measure(1)
```

Listing 4.1.3: CUDA-Q bell entanglement.

```
OPENQASM 3.0;
include "stdgates.inc";

qubit[2] q;
bit[1] c;
h q[1];
cx q[1], q[0];
c[0] = measure q[1];
```

Listing 4.1.4: OpenQASM bell entanglement.

```
from cudaq import QuantumCircuit, QuantumRegister

platform = ql.Platform('openql_platform', 'none')
kernel = ql.Kernel('bell_kernel', platform, 2)
kernel.h(1)
kernel.cnot(1, 0)
kernel.measure(1, 0)
```

Listing 4.1.5: OpenQL bell entanglement.

```
import pennylane as qml

qml.Hadamard(wires=1)
qml.CNOT(wires=[1, 0])
m = qml.measure(1)
```

Listing 4.1.6: PennyLane bell entanglement.

```
namespace BellStateExample {
        open Microsoft.Quantum.Intrinsic;
        open Microsoft.Quantum.Canon;
        open Microsoft.Quantum.Measurement;

        operation CreateBellStateAndMeasure() : Result {
                use q = Qubit[2];
                H(q[1]);
                CNOT(q[1], q[0]);
                result = M(q[1]);
                ResetAll(q);
                return result;
        }
}
```

Listing 4.1.7: Q# bell entanglement.

```
from qiskit import QuantumCircuit

qc = QuantumCircuit(2, 1)
qc.h(1)
qc.cnot(1, 0)
qc.measure(1, 0)
```

Listing 4.1.8: Qiskit bell entanglement.

```
DECLARE c BIT[1]

H 1
CNOT 1 0
MEASURE 1 c[0]
```

Listing 4.1.9: Quil bell entanglement.

```
def main() {
        q0 := 0:B;
        q1 := 0:B;

        q1 := H(q1);
        if q1 {
                q0 := X(q0);
        }
        m := measure(q1);
        measure(q0);
}
```

Listing 4.1.10: Silq bell entanglement. Note that instead of using CNOT, the high-level if-statement is used.

```
import xacc

provider = xacc.getIRProvider('quantum')
program = provider.createComposite('initial-state')
program.addInstruction(createInstruction('H', [1]))
program.addInstruction(createInstruction('CX', [1, 0]))
prog.addInstruction(xacc.gate.create("Measure", [0]))
```

Listing 4.1.11: XACC bell entanglement.

coding infrastructure. Additionally, aspiring quantum developers that are already familiar with Python (or C++) do not have to learn a completely new programming language if their QPL of choice is a framework. Also, the developers of the quantum languages have to put less thought into the syntax if they do not create a completely new language from scratch. This is especially handy if the focus of a framework lays elsewhere, e.g. the compilation and optimization of circuits (e.g. OpenQL) or interoperability (e.g. XACC).

The challenge of creating a framework for an existing classical programming language is the dependence on said programming language. The developers have to work with the syntax and execution model said languages offer, and are less flexible in the creation of their code.

The developers of the languages we look at can be divided into three groups:

- **Major tech companies** which have been founded for another purpose than quantum technologies, e.g. Microsoft.
- **Quantum tech companies** which have been founded with the intention of developing and enhancing quantum technologies, e.g. Rigetti.
- **Universities** or **research institutes**, e.g. ETH Zürich.

Six of eleven of the quantum languages we look at are developed by major tech companies, three by universities or research institutes and two by companies that have been founded to work on quantum technologies.

The quantum languages that are frameworks of Python and C++ follow a similar programming approach, which Q# follows as well. We will call it *object-oriented static circuit creation*: The programmer defines a programming structure (generally an object) that represents a quantum circuit. One can apply different operations, e.g. quantum gates or measurements, on the qubits in the circuit. When the complete circuit has been created, the programmer can submit it to a QPU or a quantum simulator. An example for the programming process of an object-oriented static circuit creation program written in Qiskit can be found in Listing 4.1.12.

The quantum frameworks in general offer to transpile or compile circuits, as most QPUs accept quantum circuits consisting of only their native gates only [122, 123].

This way of implementing quantum algorithms can be perceived as low abstraction level compared to common classical programming languages (cf. Section 2.3). It directly depicts the way gate-based QPUs work and is not developed to implement more abstract routines that do not require direct understanding of quantum gates by the developer. However, some languages offer methods to apply multiple gates at once in order to implement well-known quantum algorithms, e.g. the variational quantum eigensolver or the quantum phase estimation [16, 59]. This is for example possible in Qiskit [124] and PennyLane [125].

Three of the examined quantum language follow different programming approaches: OpenQASM, Quil and Silq.

Silq offers the possibility to directly add gate operations to quantum circuits, but also offers higher levels of abstraction. For example, one can apply an `or`-operator on qubits as well as `if`- and `else`-structures (see Listing 4.1.10).

```python
from qiskit import QuantumCircuit, transpile

# Initialize a quantum circuit
qc = QuantumCircuit(number_qubits, number_classical_bits)

# Apply gates to circuit
qc.h(0)
...

# Transpile circuit to native gate set of used QPU/Simulator
simulator = AerSimulator()
circ = transpile(qc, simulator)

# Run circuit and receive result
result = simulator.run(circ).result()
```

Listing 4.1.12: An example of an object-oriented static circuit creation program written in Qiskit.

```
OPENQASM 3;
include "stdgates.inc";

qubit[1] q;
bit[1] creg;
int[32] t = 10;

h q[0];
creg[0] = measure qubit[0];

if (creg[0] == 1) {
        t = t * t;
} else {
        t = t - 5;
}
```

Listing 4.1.13: An example of OpenQASM measuring a qubit and applying a classical calculation on a bit depending on the measurement outcome.

```
DECLARE creg BIT[1]
DECLARE t INTEGER[1]

MOVE t 10

H 0
MEASURE 0 creg[0]

JUMP-UNLESS @ELSE creg[0]
MUL t[0] t[0]
JUMP @END
LABEL @ELSE
SUB t[0] 5
LABEL @END
```

Listing 4.1.14: An example of Quil measuring a qubit and applying a classical calculation on a bit depending on the measurement outcome.

On the other hand, OpenQASM's and Quil's abstraction levels can be perceived as lower than the object-oriented static circuit creation quantum languages'. They can be found at Listings 4.1.13 and 4.1.14 for comparison.

OpenQASM has been proposed as IR. It requires the declaration of the qubits and classical bits used in its program. One can define gates and measurements on the qubits, as well as arithmetic operations, branching and looping on the classical bits. Some of its functionalities can be compared to object-oriented static circuit creation, especially the application of gates to qubit variables for quantum calculations. However, contrary to languages implementing the static circuit creation model, it does not create a circuit object. OpenQASM also does not offer functions or methods to transpile or execute a quantum program, as it is not necessary for its intended use as IR. This is why we describe it of lower abstraction level than object-oriented static circuit creation languages.

Quil has been proposed as an instruction set architecture. It is comparable to classical Assembly and offers (conditional) jumps only as classical control structures. The lack of `if` and `while` control structures as well as the lack of an execution or transpilation functionality is the reason we describe Quil as the language of the lowest abstraction among those we examine.

Quil and OpenQASM provide two options of low-level programming languages that a compiler could compile towards. Nevertheless, none of them are a fixed standard yet. The object-oriented static circuit creation languages offer transpilation functionalities, but they do not necessarily compile to Quil, OpenQASM or another IR. This is amplified by the fact that QPU hardware has also no standard way of accepting execution instructions yet. For example, IBM's devices can be addressed with OpenQASM [128],

Rigetti's devices accepts Quil programs and QIR bitcode [129] and IonQ's devices can be addressed by expressing quantum circuits in `.json` format [130]. Additionally, any code modification after submitting to a cloud QPU service is usually not disclosed. This makes an understanding of the quantum code compilation and possible standards more difficult.

However, some of the static circuit creation languages accept input written in Quil or OpenQASM and transpile it to the language's respective circuit representation. This is for example the case with Braket [108], Qiskit [126] or XACC [127]. These transpilation functionalities are typically restricted to a modification of the quantum circuit. Optimizations on classical control or hybrid structures in a refined HQCC architecture are, to the best of our knowledge, not yet available. This even holds for Silq, which we expected to offer optimization steps due to its high level of abstraction.

Most programming languages are more adapted to the QRAM than the (refined) HQCC model. They work with the static creation of a circuit that is supposed to be completely executed, before a CPU receives the result or reacts to it. Optimizations on the circuits do not consider classical instructions.

However, Silq, OpenQASM, Quil and some of the static circuit creation languages allow adding classical control structures in their control flow. We will look at the support for heterogeneous architectures of the different languages in the next section.

4.2. Support for Heterogeneous Architectures

We want to examine which quantum languages allow real-time feedback of quantum-classical computing (dynamic circuit creation). Not all languages allow unrestricted application of classical control structures and classical calculations next to the quantum calculations. To check which languages do allow dynamic circuit creation, we will look at two algorithms that require real-time classical control structures and check which languages allow the implementation of these algorithms.

4.2.1. Real-Time Feedback Algorithms

To check which languages support dynamic circuit creation, we implemented two examples of heterogeneous algorithms on the QPLs: **Quantum teleportation** and **active reset**.

Quantum teleportation [52] is the process of transferring quantum information from one qubit to the other using one ancilla qubit. We will look at a brief introduction to the workings of the algorithm. For a more detailed explanation, we refer to Nielsen and Chuang [17, chapter 1.3.7].

The quantum state is transferred without any gates that act directly on original and target qubit at the same time, which is why the term teleportation is being used. The state might be in a superposition, meaning we would not be able to measure it classically and transfer the information using bits. It is important to note that the state of the original qubit is not copied, which is impossible according to the no-cloning theorem [24,

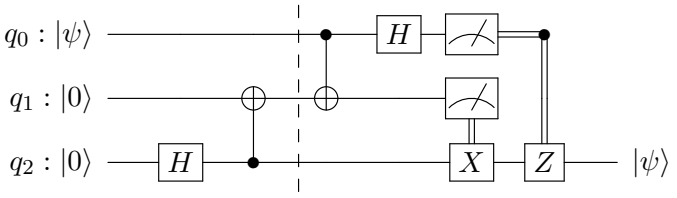


Figure 4.2.1.: The quantum circuit that teleports the quantum state $|\psi\rangle$ from q_0 to q_2.

Table 4.2.1.: The measured states of the first two qubits in the quantum teleportation circuits, and the succeeding required gates on $|q_2\rangle$.

Measured Bell State	Measured $\|q_0 q_1\rangle$	$\|q_2\rangle$	Necessary gates on $\|q_2\rangle$
$\|\Phi^+\rangle$	$\|00\rangle$	$(\alpha\|0\rangle + \beta\|1\rangle)$	
$\|\Phi^-\rangle$	$\|10\rangle$	$(\alpha\|0\rangle - \beta\|1\rangle)$	Z
$\|\Psi^+\rangle$	$\|01\rangle$	$(\alpha\|1\rangle + \beta\|0\rangle)$	X
$\|\Psi^-\rangle$	$\|11\rangle$	$(\alpha\|1\rangle - \beta\|0\rangle)$	XZ

25] (cf. Section 2.1.1). Instead, the quantum state of the original qubit is destroyed by measurement during the process.

The quantum circuit used for teleportation is given in Figure 4.2.1. The arbitrary state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ is teleported from q_0 to q_2. We calculate the equation depicted by the teleportation circuit until the first barrier (dotted line):

$$(\mathbb{1} \otimes \mathrm{CNOT}_{21}) \cdot (\mathbb{1} \otimes \mathbb{1} \otimes H)\,|\psi 00\rangle \tag{4.2.1}$$

$$= (\mathbb{1} \otimes \mathrm{CNOT}_{21}) \left(\frac{1}{\sqrt{2}}\,|\psi 0\rangle \otimes (|0\rangle + |1\rangle) \right) \tag{4.2.2}$$

$$= \frac{1}{\sqrt{2}}\,|\psi\rangle \otimes (|00\rangle + |11\rangle) \tag{4.2.3}$$

$$= (\alpha|0\rangle + \beta|1\rangle) \otimes \frac{1}{\sqrt{2}}\,(|00\rangle + |11\rangle) \tag{4.2.4}$$

The quantum state of the three qubits can be reordered and depicted using Bell states (cf. Section 2.1.2).

We use the properties

$$|00\rangle = \frac{1}{\sqrt{2}}\left(|\Phi^+\rangle + |\Phi^-\rangle\right) \tag{4.2.5}$$

$$|01\rangle = \frac{1}{\sqrt{2}}\left(|\Psi^+\rangle - |\Psi^-\rangle\right) \tag{4.2.6}$$

$$|10\rangle = \frac{1}{\sqrt{2}}\left(|\Psi^+\rangle + |\Psi^-\rangle\right) \tag{4.2.7}$$

$$|11\rangle = \frac{1}{\sqrt{2}}\left(|\Phi^+\rangle - |\Phi^-\rangle\right) \tag{4.2.8}$$

to calculate

$$(\alpha|0\rangle + \beta|1\rangle) \otimes \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \tag{4.2.9}$$

$$= \frac{1}{\sqrt{2}}(\alpha|000\rangle + \alpha|011\rangle + \beta|100\rangle + \beta|111\rangle) \tag{4.2.10}$$

$$\begin{aligned} = \frac{1}{2}\Big(&\left(|\Phi^+\rangle + |\Phi^-\rangle\right)\alpha|0\rangle + \left(|\Psi^+\rangle - |\Psi^-\rangle\right)\alpha|1\rangle \\ &+ \left(|\Psi^+\rangle + |\Psi^-\rangle\right)\beta|0\rangle + \left(|\Phi^+\rangle - |\Phi^-\rangle\right)\beta|1\rangle \Big) \end{aligned} \tag{4.2.11}$$

$$\begin{aligned} = \frac{1}{2}\Big(&|\Phi^+\rangle(\alpha|0\rangle + \beta|1\rangle) + |\Phi^-\rangle(\alpha|0\rangle - \beta|1\rangle) \\ &+ |\Psi^+\rangle(\alpha|1\rangle + \beta|0\rangle) - |\Psi^-\rangle(\alpha|1\rangle - \beta|0\rangle) \Big). \end{aligned} \tag{4.2.12}$$

We can see from Equation (4.2.12) that measuring the first two qubits in the Bell basis results into the last qubit taking one of four specific forms which can be seen in Table 4.2.1. The Bell measurement is done by a CNOT and Hadamard gate with a following measurement, as explained in Section 2.1.2. Depending on the Bell state measurement outcome, we need to apply certain gates to the target qubit to recreate $|\psi\rangle$.

Table 4.2.1 shows the necessary gates to apply on $|q_2\rangle$ to re-create $|\psi\rangle$. These gates correspond with the classically controlled gates applied to $|q_2\rangle$ in Figure 4.2.1.

The application of gates depending on measurement outcomes while $|q_2\rangle$ has to stay coherent causes quantum teleportation to be an algorithm needing dynamic circuit creation.

Active reset [53] is used to achieve high-fidelity zero-state qubits. Resetting a qubit to state 0 has a small probability of failure. This is why the active reset requires having measured the qubit in state 0 twice, as shown in Listing 4.2.1. If the qubit is measured to be in state 1, the qubit is flipped and afterwards measured again.

Even though being rather primitive, this algorithm has high requirements for its programming language. A measurement has to be possible mid-circuit, and a classical count variable has to be used, which is dependent on measurement outcomes. The calculation whether to execute the next loop hast to be done in real-time, as well as the branching inside the loop.

```
numberOfSuccesses = 0
while(numberOfSuccesses < 2):
        measure qubit, store result in bit
        if bit == 0:
                numberOfSuccesses++
        else:
                numberOfSuccesses = 0
                apply X on qubit
```

Listing 4.2.1: A pseudocode of the active Reset Algorithm.

4.2.2. Evaluating Languages

In this section, we will look at the languages we examined in Section 4.1.2. We tried to implement quantum teleportation and active reset to check which ones support real-time hybrid calculations.

The languages must be executable to check if the classical control structures can be executed correctly. This results in removal of four languages from this section of the thesis, because their code could not be executed:

- We were unable to get **XACC**'s interpreter running following the installation process given in the XACC documentation [131]. We could not use the pre-configured integrated development environment (IDE) offered on XACC's GitHub page [99] either, as this IDE lacked Python modules needed for XACC execution.
- **CUDA-Q**'s semantic is restrictive and forbids many operations. The documentation and error messages of the Python interpreter were not helpful enough to execute a program in this language.
- **OpenQASM** delivers no native way to execute its code.
- **OpenQL** delivers no native way to execute its code, it only compiles towards OpenQASM.

Table 4.2.2 summarizes which languages can implement quantum teleportation and active reset.

Braket is the only quantum language which cannot implement either of the algorithms. Braket does not support classical control structures in any way, and neither do Amazon's QPUs support HQCC computing models [63, p. 67].

All other languages allow implementing quantum teleportation, but only Q# and Quil additionally allow active reset.

Quil and **Q#** offer a turing-complete classical instruction set, thus one can implement any executable quantum-classical algorithm.

Table 4.2.2.: Ability of QPLs to implement real-time hybrid quantum algorithms.

Language	Quantum teleportation	Active reset
Braket	✗	✗
Cirq	✓	✗
PennyLane	✓	✗
Q#	✓	✓
Qiskit	✓	✗
Quil	✓	✓
Silq	✓	✗

Table 4.2.3.: Operations of the object-oriented static circuit creation languages to implement classical branching and looping.

Language	Conditional Branching	Looping
Cirq		`CircuitOperation` with `repeat_until`.
PennyLane	`cond`	`while_loop`
Qiskit	`c_if`	`while_loop`

Silq allows conditional logic on classical variables, which means one can implement quantum teleportation in it. We can also implement loops that depend on classical logic and classical arithmetic in Silq. However, when we measured a qubit once, we cannot use it again without redefining it. Therefore, while we can in principle implement loops depending on measurement results, we cannot implement the active reset algorithm.

The remaining object-oriented static circuit creation languages **Cirq**, **PennyLane** and **Qiskit** have operations to add gates depending on classical variables to the circuit. They offer both conditional application of gates and looping. The necessary instructions are listed in Table 4.2.3. However, while the branching allows to implement the quantum teleportation algorithm, the loops fail to implement active reset. The reasons are:

- Cirq cannot do arithmetic on two classical values which are afterwards used in the `repeat_until` condition.
- PennyLane's loop cannot iterate over a classical value that is updated by a measurement in the loop.
- Qiskit offers a loop, but it cannot assign a value to a classical register without a measurement or do arithmetic on two classical values.

4.3. Summary

In this section, we examined the properties of existing QPLs. We found that most languages follow the programming approach of object-oriented static circuit creation and are implemented as Python framework. All languages of different approaches than object-oriented static circuit creation are stand-alone languages and not frameworks. This could indicate that stand-alone languages work better than embedded languages on programming approaches that are different from object-oriented static circuit-creation.

We saw that most quantum languages lack not only optimization specifically for heterogeneous quantum-classical architectures, but also the possibility to implement arbitrary quantum-classical algorithms. Programming languages that support quantum-classical algorithms are still in early development, but crucial for the progress of quantum computation.

In the next chapter, we will examine possibilities of optimizing code for heterogeneous quantum-classical architectures in the future.

5. Optimizing Quil-Programs

As we have seen in Chapter 4, existing QPLs offer little to no optimizations for heterogeneous architectures. In this section, we want to assess which static optimizations can be done by the main CPU of a refined HQCC architecture, before the program is sent to the QPU/CPU component (cf. Section 2.2.1).

Classical compilers typically apply many small optimization strategies to a program, such as constant-propagation, constant-folding, liveness analysis, reaching definitions or available expressions [66] (cf. Section 2.3.2). We aim to create a similar set of optimization methods for heterogeneous quantum-classical architectures. For this, we wrote a program that parses, analyzes, optimizes, and evaluates Quil code.

Quil has been used for these examinations, as Quil with its assembly-like syntax and `goto` branching is straightforward to analyze. Additionally, Quil programs can be created with PyQuil [116] and executed with Quilc [117] and the Quil-Lang QVM [118] (cf. Section 4.1), which is helpful for our work.

We will first look at Quil's instructions in Section 5.1 and examine how to execute Quil on a HQCC architecture (cf. Section 5.2). Afterwards we will explain how we analyzed Quil programs (cf. Section 5.3) and how we evaluated them in Section 5.4.

In Section 5.5, we will propose a set of optimization methods to optimize a Quil program with respect to heterogeneous architectures. We will evaluate the optimization methods in Section 5.6 and discuss the results in Section 5.7.

The GitHub repository containing the code we developed is given in Appendix A.1.

5.1. Quil Instructions

In this subsection, we will give an overview of the instructions that Quil provides, to the extent that it is relevant in this thesis. For a more extensive overview, we refer the reader to the paper introducing Quil [14].

In Quil, integer indices are used to refer to qubits. As integers can also occur in other parts of the program (e.g. as classical parameters), this leads to an ambiguity considering the usage of integers. However, the position of an integer clearly indicates if the integer is used as qubit or a parameter, as qubits follow after gate instructions or measurements. Qubits do not need to be declared upfront.

Quil supports four classical variable types: `BIT`, `OCTET`, `INTEGER`, and `REAL`. Classical variables need to be declared before usage, e.g. `DECLARE a BIT` to declare a bit value of the name `a`. Afterward, the value can receive a value (`MOVE a 0`) and be used in classical instructions.

Classical variables can be declared as an array, e.g. `DECLARE a BIT[2]`. The values can then be accessed by `[n]`, e.g. `MOVE a[1] 0`.

Quil offers different categories of instructions, namely

- **quantum gates**: Instructions that name a gate and at least one qubit that is applied to the gate, e.g. `H 0`, or `CNOT 0 1`.
- **classical instructions**: Calculations applied on classical variables. The classical instruction set is turing-complete.
- **parameterized quantum gates**: Quantum gate instructions with gates that receive a classical parameter, e.g. `RZ(angle) 0`, with the classical parameter `angle`. Quantum gates that receive a fixed value (e.g. `RZ(1.57)`) do not count as parameterized.
- **measurements**: Instructions that measure the value of a qubit. The value can be saved in a classical parameter. E.g.: `MEASURE 0 ro[0]`.
- **control structures**: One can define labels (`LABEL @label_name`) and jumps in Quil. Jumps can be conditional (e.g. `JUMP-WHEN @label_name cond`) or unconditional (e.g. `JUMP @label_name`).

Quantum gates that are not parameterized can be exclusively executed by a QPU and are therefore *quantum instructions*. Classical instructions can be exclusively executed by a CPU and are therefore *classical instructions*. All other instructions need to be executed by QPU and CPU at the same time, and are therefore *hybrid instructions*.

5.2. Naive Quil Execution

Quil provides instructions for both quantum and classical instructions. We will assume that each Quil instruction takes one time step to be executed for our proof-of-concept. We neglect the communication time between QPU and CPU, as we do not know its speed and it depends on the concrete hardware.

Purely quantum or classical instructions can be executed in parallel, as they are executed by two different devices. Hybrid instructions (like measurements, parameterized gates, or branching) need to be executed by both devices at the same time.

We assume the following execution process for the CPU – QPU component:

- If the next instruction is a hybrid instruction, it is executed by the CPU and QPU in parallel.
- If the next instruction is a classical (or quantum) instruction, it is sent to the CPU (or QPU). Additionally, if there is a quantum (or classical) instruction in the code before the next hybrid instruction, it is sent to the QPU (or CPU) as well. Both instructions are then executed in parallel.

CPU and QPU can work in parallel, and at the next hybrid instruction, the devices have to wait for each other.

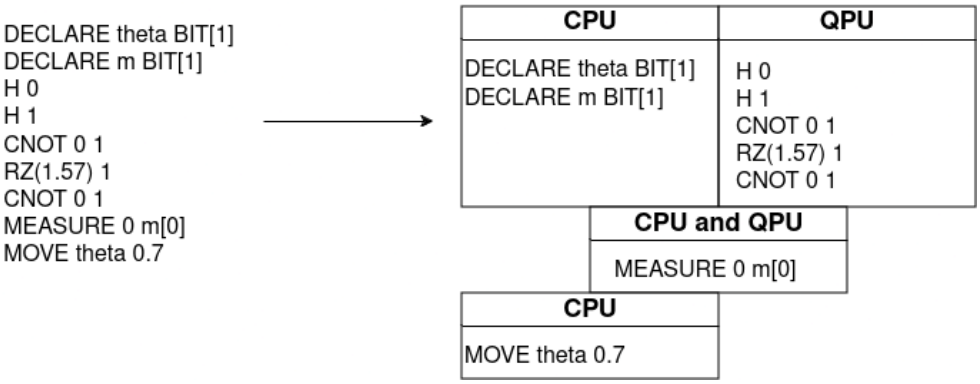


Figure 5.2.1.: An example how a naive execution of Quil would cause the CPU to wait for a hybrid instruction (`MEASURE`), even though the succeeding `MOVE` instruction would be executable.

This execution model can cause unnecessary idling of a device. An example for this can be seen in Figure 5.2.1. It shows a CPU that needs to wait for the QPU, even though a succeeding classical instruction would be executable. A previous optimization routine could find that an instruction is executable before the hybrid instruction and change the order of the instructions in the program.

We want to explore the possibility of optimizing Quil code with respect to heterogeneous architecture. At the moment, this architecture is not considered during Quil compilation or the execution on the Quil QVM to the best of our knowledge.

5.3. Analyzing Quil Programs

We did analyzation procedures that can be applied to arbitrary Quil programs. In this work, we will use a few algorithms as ongoing example, namely:

- Quantum teleportation [52]
- Magic state distillation [54, 55, 56]
- Repeat-until-success [57, 58]
- Iterative phase estimation (IPE) [49, 59]

We will not go into detail about how the algorithms work. This is not necessary for this work, and we refer the reader to the given sources for further reading. The important point about the algorithms for this thesis is that they all require dynamic circuit creation in order to work, and thus communication between CPU and QPU.

We created a CFG [68] for all programs we evaluate. Recall that a CFG is used to depict all possible execution paths of a program (cf. Section 2.3). Each node/basic block of the CFG consists of either only quantum instructions, classical instructions, or hybrid/control structures. Quantum and classical instructions that are executed without

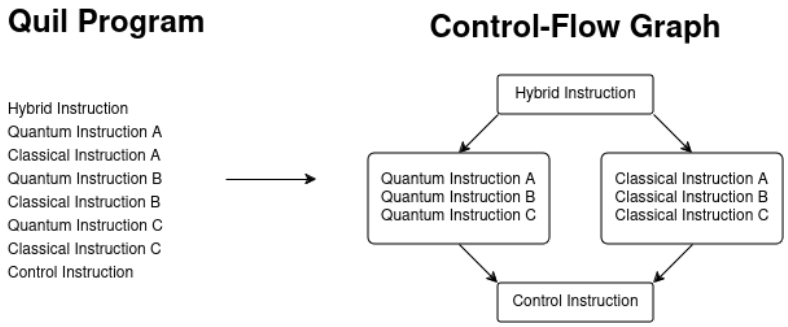


Figure 5.3.1.: Creating basic blocks in the CFG from alternating quantum and classical instructions.

a hybrid/control instruction in between are written into two parallel basic blocks. An example of this is shown in Figure 5.3.1. By creating a CFG in this way, one can easily see which parts of the code are executed by which device(s).

Recall that a DDG is a graph that shows which instructions need to be executed before another instruction (cf. Section 2.3). We create a DDG using information from the Quil program and the CFG. Every node of the DDG holds a single Quil instruction. The edges indicate which instructions have to be executed before their instruction can be executed. This is done by checking variable dependency. Unconditional jumps are resolved before creating the DDG and not listed as nodes.

Conditional jumps in the program pose a problem to the DDG. If a conditional jump targets an already executed line, it introduces circular dependencies. This could only be resolved exactly if we knew the number of iterations, which would be analogous to solving the Halting problem, and therefore not generally possible.

We resolve this issue by creating a DDG only up to the next conditional jump. By that, we can receive multiple DDGs for a single program, all depicting a part of the program. One of the DDGs is the start DDG, which is the DDG describing the entry of the program. Additionally, we have one or multiple halt DDGs, which include the program instructions last executed before the program terminates. An example for a program that results in multiple DDGs is given in Listing 5.3.1, and the corresponding DDGs in Figure 5.3.2.

As an example, the code for the IPE is given in Listing 5.3.2, and the corresponding CFG in Figure 5.3.3. The given code calculates two bits of the phase φ of a given matrix, with the matrix' eigenvalue $\lambda = e^{i\varphi}$.

In the algorithms we implemented to evaluate our optimizations, an algorithm that calculates five bits was used. The shorter algorithm is given here for simplicity.

An DDG of the IPE code (Listing 5.3.2) is given in Figure 5.3.4. For the IPE, we resolved conditional jumps before DDG creation. In the executable IPE, we use the code given in Listing 5.3.3 to add 1 to `param_no_pi[0]` if `lastMeasurement[0]` is 1.

Conditional jumps have to be used, as Quil's semantic does not allow adding a bit-value to a real value. In the IPE that is used for the DDG and during the optimization,

```
DECLARE m BIT
H 0
MEASURE 0 m
JUMP-WHEN @label m
Y 0
LABEL @label
Z 0
MEASURE 0 m
```

Listing 5.3.1: An example for Quil code that results into multiple DDGs. The corresponding DDGs can be found in Figure 5.3.2.

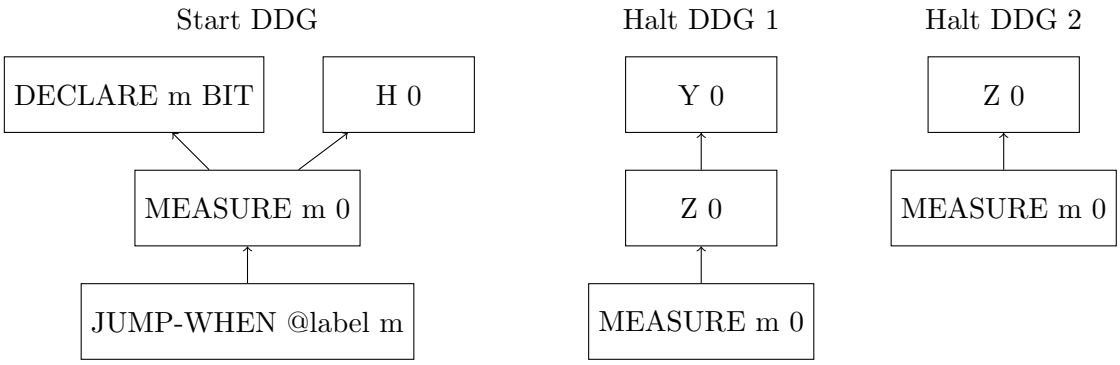


Figure 5.3.2.: An example for multiple DDGs originating from one Quil code. The code can be found in Listing 5.3.1.

```
DECLARE theta REAL[1]
DECLARE result REAL[1]
DECLARE param_no_pi REAL[1]
DECLARE lastMeasurement BIT[1]

MOVE theta[0] 0
MOVE param_no_pi[0] 0
MOVE lastMeasurement[0] 0

H 0
CONTROLLED targetMatrix_two 0 1 2
RZ(theta[0]) 0
H 0
MEASURE 0 lastMeasurement[0]
RESET 0
MOVE theta[0] 3.1415
JUMP-UNLESS @noadd1 lastMeasurement[0]
ADD param_no_pi[0] 1
LABEL @noadd1
DIV param_no_pi[0] 2
MUL theta[0] param_no_pi[0]

H 0
CONTROLLED targetMatrix_zero 0 1 2
RZ(theta[0]) 0
H 0
MEASURE 0 lastMeasurement[0]
MOVE theta[0] 3.1415
RESET 0
JUMP-UNLESS @noadd2 lastMeasurement[0]
ADD param_no_pi[0] 1
LABEL @noadd2
DIV param_no_pi[0] 2
MUL theta[0] param_no_pi[0]

MOVE result[0] 3.1415
MUL result[0] 2
MUL result[0] param_no_pi[0]
```

Listing 5.3.2: Calculating two bits of `targetMatrix_zero`'s phase φ, which is stored in `result[0]`. The matrix' eigenvalue is $\lambda = e^{i\varphi}$. `targetMatrix_two` is `targetMatrix_zero` squared.

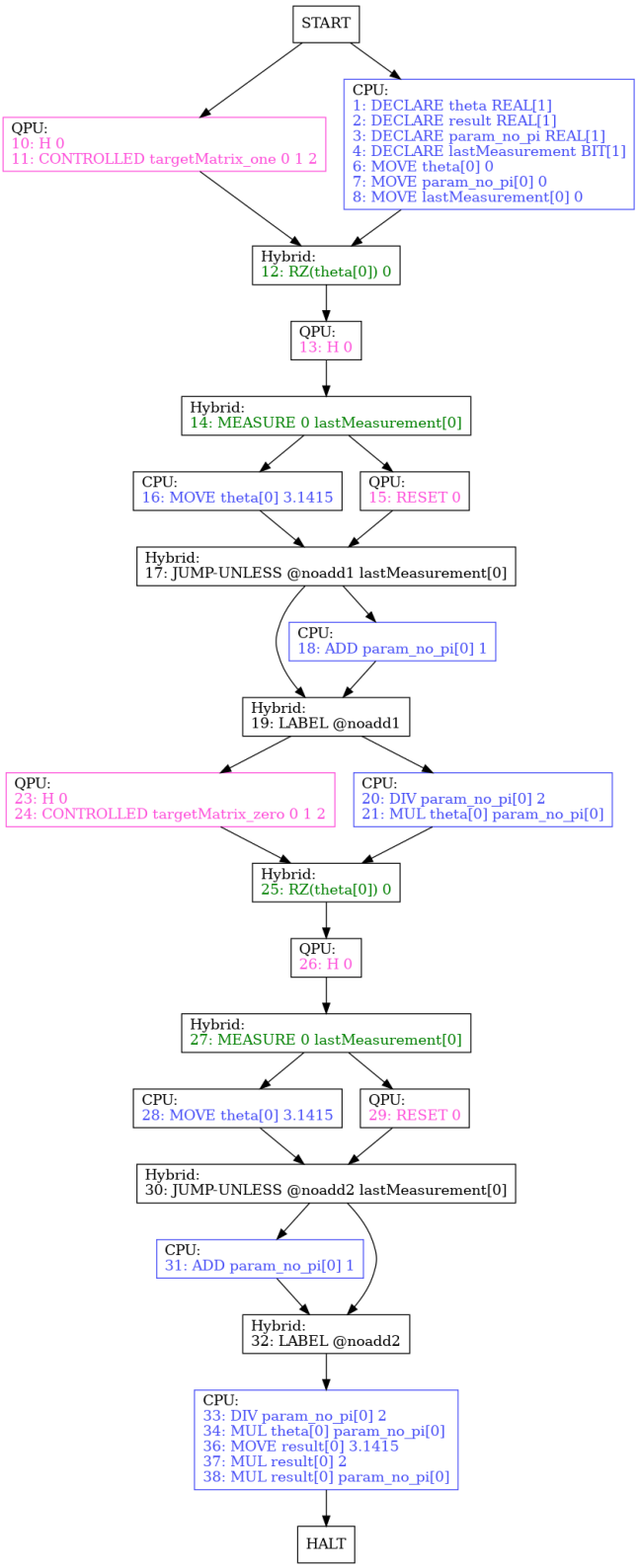


Figure 5.3.3.: The CFG created from Listing 5.3.2.

```
DECLARE param_no_pi REAL[1]
DECLARE lastMeasurement BIT[1]

JUMP-UNLESS @noadd1 lastMeasurement[0]
ADD param_no_pi[0] 1
LABEL @noadd1
```

Listing 5.3.3: Executable version of adding 1 to param_no_pip[0] if lastMeasurement[0] is true. Quil's semantic does not allow to add a BIT variable to a REAL variable. Therefore, we need conditional jumps.

```
DECLARE param_no_pi REAL[1]
DECLARE lastMeasurement BIT[1]

ADD param_no_pi[0] lastMeasurement[0]
```

Listing 5.3.4: Version we use for analyzation of adding 1 to param_no_pip[0] if lastMeasurement[0] is true. The BIT value is directly added to param_no_pi[0] to avoid conditional jumps.

the above code is changed to the one in Listing 5.3.4: The bit value is directly added to the real value.

While Quil's semantic forbids this construction, it has the same logical effect: Adding 1 to `param_no_pi[0]` if `lastMeasurement[0]` is 1. This is done to prevent conditional jumps in the IPE algorithms, which makes the optimization and the evaluation of the optimization simpler. This has only been done for IPE, not for the other algorithms. The CFGs and DDGs for the other algorithms can be found in the GitHub repository with our code (cf. Appendix A.1).

The properties of the created DDG are listed in Table 5.3.1. As the IPE algorithm contains no conditional jumps, it is depicted by one DDG only, while the other algorithms need at least three DDGs. We calculated a naive execution time for every DDG by using the execution procedure from Section 5.2. We assume an execution time of 1 per instruction. We make this assumption as we do not have sufficient insight in real hardware. Depending on the hardware, we could have large differences between the execution time of quantum and classical instructions.

In the following, we will look at evaluation metrics for Quil programs and try to optimize the programs with respect to the metrics.

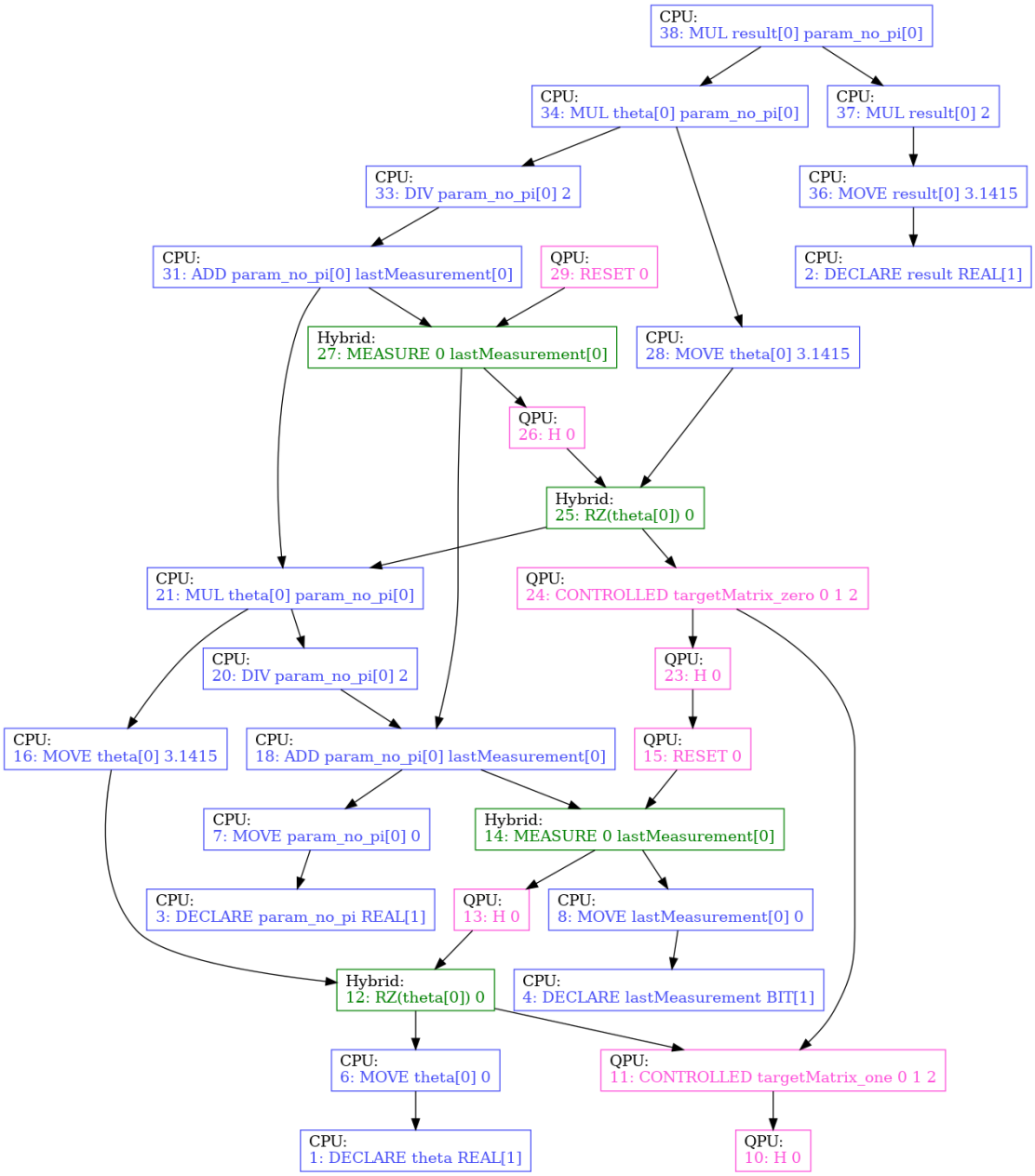


Figure 5.3.4.: The DDG created from Listing 5.3.2. The conditional jumps have been changed to an addition of `param_no_pi[0]` with `lastMeasurement[0]`, to avoid conditional jumps.

Table 5.3.1.: Properties of the example programs and their respective DDG(s). Instruction number and naive wall time are given per DDG. The wall time is calculated using the naive algorithm explained in Section 5.2 and assuming a time of 1 for each instruction.

Example program	Lines	Instruction Number	Naive Wall Time
Quantum teleportation	14	8, 2, 2	6, 2, 1
Magic state distillation	76	67, 63, 6	62, 62, 6
Repeat-until-success	24	12, 11, 10, 7	9, 10, 9, 6
IPE	67	55	45

5.4. Metrics to Evaluate Quil Programs

We will look at three metrics statically to evaluate Quil programs. The metrics can be used to optimize Quil programs against or to evaluate how well optimization methods worked. We implemented this functionality and evaluate our optimization methods against these metrics. The results are given in Section 5.6.

Wall time: Considering that all instructions have the execution time 1, it is assessed how long the program would take to execute. Whenever possible, the CPU and QPU execute instructions in parallel. Two parallel executed instructions receive execution time 1 together. At a hybrid or control instruction, one device waits for the other. For example, the program in Figure 5.2.1 has a wall time of 7.

The wall time is calculated for all DDGs of a program and summed up for comparison of syntactically different representations of the same program.

Quantum instruction number (QIN): Count the number of quantum instructions per DDG and sum the result up for comparison to other codes. Hybrid instructions are included in the counting, as the QPU and the quantum state are considered during hybrid calculations. Minimizing the QIN is sensible, as quantum instructions are more likely to introduce errors than classical instructions (cf. Section 2.1.3).

Quantum calculation time (QCT): As quantum states in a QPU have a limited coherence time, it makes sense to minimize the time the QPU has to keep the qubits coherent. This value is the QCT. As we assume that each instruction needs a time value of 1, the time is given in units of the instruction number.

For the calculation of QCT, we assume the best case: The QPU starts with the first quantum instruction on the latest possible time such that the CPU does not have to wait for the QPU at the first hybrid instruction. Additionally, the QPU calculates the quantum instructions after the last hybrid instruction directly after the hybrid instruction. Between the first and the last hybrid instruction, the wall time is equal to the time the qubits have to stay coherent.

It should be noted that a reasonable program has no quantum instructions after the last hybrid instruction. This would only create quantum information which is destroyed again, as it would have to be measured for further usage. Measurement, again, would be a hybrid instruction.

$n_{q\_before} = 2$	DECLARE r REAL	Classical
	DECLARE m BIT	Classical
	H 0	Quantum
	Y 0	Quantum
$\delta t_{between} = 6$	MEASURE 0 m	Hybrid (first)
	H 0	Quantum
	Z 0	Quantum
	MOVE r 2	Classical
	RZ(r) 0	Hybrid
	H 0	Classical
	MEASURE 0 m	Hybrid (last)
$n_{q\_after} = 1$	Z 0	Quantum

Figure 5.4.1.: An example of how to determine $\delta t_{between}$, n_{q_before} and n_{q_after} from Quil code. The QCT is the sum of these three values.

This results in a QCT τ_Q of

$$\tau_Q = \delta t_{between} + n_{q\_before} + n_{q\_after} \tag{5.4.1}$$

with the wall time between first and last hybrid instruction (including these hybrid instructions) $\delta t_{between}$, the number of quantum instructions before the first hybrid instruction n_{q_before}, and the number of hybrid instruction after the last hybrid instruction n_{q_after}. An example can be seen in Figure 5.4.1.

A difficulty of this metric arises when the program is divided into more than one DDG. The first hybrid instruction in the start DDG is the globally first hybrid instruction. The last hybrid instruction in the halt DDG is the globally last hybrid instruction. If there are multiple possible last DDGs, we take the one that causes the longest QCT (worst-case). The QCT is calculated by the number of quantum instructions before the first hybrid instruction and after the last hybrid instruction. Additionally, the wall time of all other DDGs and between the first and the last hybrid instruction (included) are added.

If the last DDG has no quantum instruction, it does not add to the QCT. The wall time of all other DDGs is simply added, as every DDG necessarily ends at a hybrid conditional jump.

Note that, in an ideally optimized program, a DDG with only classical instructions should not exist, as this could be calculated completely by the main CPU and does not have to be sent to a CPU-QPU complex.

Table 5.4.1 shows the values of the different metrics for our initial implementations of the algorithms. We will look at strategies to improve (i.e., reduce) the values in the next sections.

Table 5.4.1.: Wall time, QIN (quantum instruction number), and QCT (quantum calculation time) of the original Quil files of the different algorithms. The Wall time is calculated by the naive execution time described in Section 5.2 and per DDG.

Example program	Wall time	QIN	QCT
Quantum teleportation	6, 2, 1	9	10
Magic state distillation	62, 62, 6	66	130
Repeat-until-success	9, 10, 9, 6	34	35
IPE	45	25	33

5.5. Optimization Strategies

In this section, we aim to create a set of optimization methods for heterogeneous quantum-classical architectures.

In the beginning, we will derive some optimization strategies from already well established classical (machine-independent) strategies. As introduced in Section 5.3, loops play a critical role when analyzing program structures. Nevertheless, they are out of scope for this work and we will not target optimization strategies that aim to work with loops.

5.5.1. Adapted Classical Optimization Operations

We will examine if and how the classical machine-independent optimizations mentioned in Section 2.3 can be applied to the quantum-classical heterogeneous case. Some optimization methods are not sensible for the quantum case, but could in principle be implemented for the classical part of the algorithms. We will not implement these optimizations, to keep our focus on quantum-classical hybrid code.

It is insensible to try to determine **available expressions** in Quil. Instructions that calculate an expression $a \bullet b$ and assign the result to another variable do not exist. Instead, quantum operators act directly on qubits and the results of classical calculations are stored in one of the involved variables. For example,

```
ADD a b
```
(5.5.1)

is interpreted as

$$a \coloneqq a + b. \tag{5.5.2}$$

```
1 DECLARE a INTEGER
2 DECLARE b INTEGER
3 MOVE a 3
4 ADD a 10
5 MOVE b 7
6 MOVE a 10
```

Listing 5.5.1: An example of Quil code with live and dead variables. Assuming `a` is a readout variable, it counts as dead in line 4, as the addition result is not read anywhere before a is written to in line 6. `b` is dead in line 5, as it is not read afterward.

We apply **constant propagation** to the classical and the quantum part of the code.

For the classical part of the code, the algorithm works as follows: A classical value is recognized as constant by a `MOVE` instruction if the instruction moves a constant value onto it. E.g

```
MOVE a 10
```
(5.5.3)

results into `a` having the constant value 10.

A variable remains constant until it is being written to again.

```
ADD b a
```
(5.5.4)

writes and reads `b`, but only reads `a`, thus `a` remains constant. This means the algorithms recognizes `b` to be constant at the reading part of the instruction, but `b` looses its constant state afterward. This would even hold if `a` was replaced by a constant value. Constant folding will replace the instruction by a `MOVE` instruction if `a` and `b` were constant.

For the quantum part of the code, we restrict ourselves to the Pauli-basis and single qubit gates. A qubit is constant at the start of the program and after a `RESET` with the value `Pauli X Zero`. The constant propagation checks at which instructions a constant qubit "arrives" with a specific value, either because it has been initially used or reset before, or because constant folding has shown that it remained in a constant Pauli basis after a gate application.

Live-variable analysis can be used for the classical and quantum parts of heterogeneous programs.

A classical variable counts as dead if it is written to and the value of the variable is not read until the variable's value is overwritten. For the application of the algorithm, we need to know which classical variables hold *readout values*, i.e. which values need to be reported back to the main CPU at the end of the quantum program. These variables are by default alive at the end of the program, which is handled as if there was an additional read of the variable after the halt of the program. An example is given in Listing 5.5.1.

A quantum variable counts as dead if its information content does not influence any classical information from the current instruction until the end of the program.

The quantum variable can safely be called dead at a program point p if it is

- not used by any multi-qubit gate, **and**
- not measured

until

- the end of the program, **or**
- the next reset.

These conditions even hold if the instruction at p is a measurement, as long as this measurement is not saved in a classical variable.

The variable may be entangled with other variables. Due to the no-communication theorem [29] (cf. Section 2.1.1), operations on a qubit A cannot influence the measurement result on other qubits without using a measurement result of A. This even holds if A and the other qubits are entangled. Single qubit operations on an entangled qubit cannot influence other qubit states either (cf. Section 2.1.2). Therefore, we do not have to check for entanglement if a qubit will not be used in any multi-qubit gates and if the result of a measurement on it will not be considered in the program.

We can only apply live-variable analysis at the halt DDG of the program, as we cannot say whether the variable will be used in future DDGs. This limits the usage of the live-variable analysis. Live-variable analysis could be applied to non-halt DDGs in a restricted manner: All values count as alive at the end of the DDG (handled as if a read-instruction is about to follow). If a variable is dead because its value is overwritten within one DDG, it can still safely be called dead. However, our current implementation does not support this.

Reaching definitions can be implemented for the classical Quil variables by checking `DECLARE` and `MOVE` statements. Quil programs do not offer define/declare statements for qubits, neither direct assignment of a value to a qubit. Therefore, applying reaching definitions to the quantum part of the program is not sensible.

Constant folding can be applied on classical and quantum parts of our hybrid code. In classical instructions, all constant values that are read are replaced by their constant values. This means, e.g.,

```
ADD a b
```
(5.5.5)

is replaced by

```
ADD a 10
```
(5.5.6)

if `b` has a constant value of 10. The same holds for parameterized quantum gates. If the classical parameter is constant, it is replaced by the constant value.

If a value that is written to is constant and additionally, all read values in the instructions are constant, the instruction is replaced by a `MOVE` instruction. As example: If `a` is 5, `b` is 7 and the instruction is

```
ADD a b,
```
(5.5.7)

it is replaced by

```
MOVE 12
```
(5.5.8)

as `a` would be 12 after the addition.

For the quantum part of the program, constant folding calculates the result of the application of a gate on a qubit. This is only calculated if the qubit is constant beforehand (i.e. in a Pauli basis, as defined by constant propagation) and a single-qubit Clifford gate acts on it. We calculate the next Pauli state and still assume the qubit as constant (in the respective Pauli state).

Constant folding for quantum variables can be done efficiently for Clifford gates (cf. Section 2.1.2). To avoid dealing with entangled states during constant propagation, we generally assume qubits to be non-constant after a multi-qubit gate.

Due to this, we only trace the constant value of a qubit until the first non single-qubit or non-Clifford gate acts on it. This means we are tracing the six Pauli basis state as values of the qubits.

Measurements are in general considered to be random and write non-constant values in the classical values. The only exception is if we know that a qubit is either in the $|0\rangle$ or $|1\rangle$ state. In that case, the classical value receives the constant value 0 or 1.

Copy propagation can be used in the classical part of heterogeneous code. It is not sensible for quantum code due to the no-cloning theorem [24, 25] (cf. Section 2.1.1).

Dead code elimination can be applied to classical and quantum instructions. We use the results of the live-variable analysis of which variables are dead. Classical code is dead if all variables written to in the instruction are dead. Quantum code is dead if all quantum variables an instruction acts on are dead All dead instructions are deleted.

These were optimization methods taken from the compilation routine of purely classical code. We end up with implementing *constant propagation*, *live-variable analysis*, *constant folding*, and *dead code elimination*.

5.5.2. Optimizations for Quantum-Classical Calculations

Besides the classical methods, we can apply algorithms that are specific for the heterogeneous quantum-classical architecture. We introduce three algorithms targeting quantum-classical architecture in this thesis: An analysis operation that finds hybrid-dependencies, and the two transformation operations instruction reordering and latest possible quantum execution. All three operations have been implemented in our code.

Finding hybrid-dependencies checks which instructions have to be executed before a hybrid node becomes executable. This analysis first finds all hybrid instructions, and afterwards determines which instructions have to be executed before the respective

```
1 DECLARE m INTEGER[1]
2 H 0
3 MEASURE 0 m
4 RZ(m) 0
```

Listing 5.5.2: An example of Quil code with hybrid instructions that depend on previous lines.

hybrid instruction by using the DDG. A hybrid instruction can have other hybrid instructions as dependencies. A hybrid instruction "blocks" the instruction dependencies for all other hybrid instructions.

For example, Listing 5.5.2 contains the hybrid instructions `MEASURE` and `RZ(m)`. All other lines have to be executed before `RZ(m)`. But the only direct dependency we save for it is `MEASURE`, as `MEASURE` is a hybrid instruction and depends on `H` and the `DECLARE` instruction as well. For `MEASURE`, the dependencies `H` and `DECLARE` are saved.

Instruction reordering can be done as long as instructions dependent on each other are still in the same order. The DDG provides information about the dependent instructions.

The goal is to reorder the instructions to maximize the number of parallel executions of the CPU and QPU. In other words, the time in which only one device is calculating and the other one is idling should be minimized.

The two devices only influence each other through hybrid nodes. To prevent long waiting times of one device for the other, we use the knowledge we gained from the DDG and the finding hybrid-dependency analysis.

Listing 5.5.3 shows the pseudocode of the instruction reordering algorithm. If the CPU or the QPU have to wait for the other device before a hybrid instruction, the algorithm aims to avoid that the waiting device does not execute instructions. It is checked whether there are instructions for the waiting device that can already be executed, i.e. for which all dependencies have been executed. If this is the case, the waiting device can to execute the instructions while waiting instead of doing nothing.

Latest possible quantum execution is one method to keep the total execution time of the quantum device small.

The goal of this algorithm is to execute as many classical instructions as possible before the QPU starts to work. For this, the execution of instructions is reordered in the following way:

- All classical instructions that are not dependent on quantum device are executed at the start.
- The first quantum instruction is executed such that QPU and CPU reach the first hybrid node simultaneously. This is again done with the assumption that each instruction has an execution time of 1.

```
Input:
instruction_ddg: The instructions of the programm with the dependencies of the instructions.

Output:
execution_queue: The (probably new) order in which the instructions should be executed. Must keep the topological order of instruction_ddg.

execution_queue ← empty list
relevant_instructions_list ← hybrid instructions and last instruction of instruction_ddg
next relevant_instruction to be executed:
        dependencies ← instructions that still need to be executed before relevant_instruction in instruction_ddg
        quantum_number ← amount of quantum instructions in dependencies
        classical_number ← amount of classical instructions in dependencies
        Add dependencies to execution_queue
        while new instructions in instruction_ddg are excutable wrt current execution queue and quantum_number != classical_number:
                if quantum_number > classical_number:
                        Add executable classical instructions to execution queue
                else:
                        Add executable quantum instructions to execution queue
                Update quantum_number and classical_number according to the number of added instructions
        add relevant_instruction to execution_queue
        repeat
```

Listing 5.5.3: Instruction reordering algorithm.

Table 5.6.1.: Best (i.e. minimal) wall time, instruction number, and QCT that could be reached by random application of 50 optimizations. No given number indicates that no improvement could be done compared to the original program. The QIN has never been improved. The original values can be found in Tables 5.3.1 and 5.4.1.

Example program	Wall time	Instr. no.	QCT
Quantum teleportation	–	–	–
Magic state distillation	53, 53, 6	–	112
Repeat-until-success	–	–	–
IPE	35	51	30

After setting up this set of optimization operations, we will check in the next section how these operations could optimize the example algorithms introduced in Section 5.3.

5.6. Evaluate Optimization Methods

In this section, we will examine how the optimization operations introduced in Section 5.5 affect quantum algorithms. We use our own implementation for this.

Determining the best order to apply optimization operations on a given code (phase-ordering) is an open question in classical compiler architecture [132, 133]. Finding a sensible order is out of scope for this work, and we will simply draw operations to apply to the codes at random. We do this 500 times for each algorithm and apply 50 optimization operations every time. While the operation drawing was random, certain analysis-transformation routines had to follow directly after each other, as certain analysis operations lay the groundwork for succeeding transformation operations. The optimization routine consisted of a permutation of the following operations succeeding each other:

- Constant propagation → constant folding.
- Live-variable analysis → dead code elimination.
- Finding hybrid-dependencies → instruction reordering.
- Finding hybrid-dependencies → Latest possible quantum reordering.

Which means we did 25 random draws per optimization routine.

After all optimization operations had been applied, the metrics of Section 5.4 were calculated for the optimized Quil program. The best resulting values for the metrics to evaluate Quil programs against can be seen in Table 5.6.1.

For quantum teleportation and repeat-until-success, no improvement compared to the original programs could be found. The iterative phase estimation's wall time, instruction

Table 5.6.2.: The results of optimizing the iterative phase estimation with respect to the metrics we evaluate the code against. 500 runs have been done. The frequencies of the different combinations of wall time, instruction number, QIN and QCT are given. The bold results are the best ones for this metric (only given if the metric does not have the same result for all sets).

Wall time	Instr. no.	QIN	QCT	Result frequency
35	**51**	25	**30**	49.2%
36	**51**	25	**30**	43%
36	**51**	25	31	6.6%
37	52	25	**30**	0.6%
37	53	25	31	0.2%
39	**51**	25	33	0.2%
39	55	25	32	0.2%

number, and QCT could be reduced by 22% (wall time), 7.3% (instruction number), and 9.1% (QCT).

The magic state distillation's wall time (summed over all DDGs) could be reduced by 14%, its QCT dropped by 14% as well. It yields the same values for all metrics in every optimization iteration.

The different result sets of the IPE algorithm and their frequencies can be found in Table 5.6.2. The most frequent results are a wall time of 35, an instruction number of 51, and a QCT of 30. This is not only the most frequent result set, but also the result set with most optimal values for all metrics.

The second most frequent result has a value of 35 for the wall time.

5.7. Summary

In this section, we could see that we are able to optimize code for heterogeneous quantum-classical architecture. We implemented parsing, analysation, optimization, and evaluation procedures for algorithms written in Quil. The optimization procedures were able to improve some of the metrics we evaluate the code against.

It could be shown that the order of instructions has to be carefully considered in an HQCC architecture. The integration of two devices (CPU and QPU) into one component results into new properties that have to be taken into account for optimization.

Two of the algorithms we looked at could not be improved, and only some parameters improved for the examples that did improve at all. This is most likely a consequence of the fact that the programming language we are looking at has very low abstraction. This means we do not have the necessity for optimization that is the result of more abstract programming languages being compiled to a less abstract language.

The suggested optimization routines did not manage to reduce the total number of quantum instructions for any algorithm. This raises the question how sensible it is

to try and optimize a quantum circuit by routines derived from classical optimization. Another way to optimize quantum circuits is to re-order and exchange quantum gates based on their physical properties [10] (cf. Chapter 3). This optimization only considers the quantum part of a program and is possibly the more sensible way of optimizing quantum circuits.

To sum up, we showed that we are able to optimize programs for the heterogeneous case. The order of the instructions is especially relevant for this kind of programs. However, we have also seen that current quantum algorithm have only a limited potential for being optimized. Nevertheless we can expect this field to become more relevant in the future. QPLs will most likely become more abstract, like classical languages did. This will also cause more necessity for code optimization, for which heterogeneous properties will have to be considered.

6. Conclusion

In this thesis, we looked at QPUs in heterogeneous quantum-classical systems. Our focus was on the refined HQCC architecture [49]. This architecture defines a quantum component which is part of a distributed computing system. The quantum component consists of a CPU and a QPU, which can communicate within the coherence time of the qubits. We examined how programs dedicated to the refined HQCC architecture can be optimized.

We will now discuss the results and examine the possibilities for future work.

6.1. Discussion

The aim of this thesis was to examine optimization routines for heterogeneous calculations. There is already research about the optimization of quantum circuits, but the existing research neglects the classical part of real-time quantum-classical calculations. We wanted to find out which kind of optimization routines are possible and which already exist. We looked at some of today's QPLs in Chapter 4. We found that most QPLs are best adapted to static circuit creation, i.e. one creates a quantum circuit that does not have to be updated by a CPU during execution on a QPU. Additionally, none of the languages we looked at support optimizations specifically targeting heterogeneous quantum-classical architectures.

Due to that finding, we examined some optimization options ourselves in Chapter 5. We used the programming language Quil [14] for that, as it is low in abstraction and there are some tools that support developing and executing Quil. We introduced performance metrics to evaluate a program in Section 5.4, namely wall time, quantum instruction number and quantum calculation time. In Section 5.5 we suggested some optimization operations for heterogeneous Quil programs, namely constant propagation, live-variable analysis, constant folding, dead code elimination, finding hybrid-dependencies, instruction reordering, and latest possible quantum execution. We tried out and evaluated the operations in Section 5.6. We found that we are in principle able to optimize programs with respect to our performance metrics with the operations we suggested.

The optimization operations we worked with were derived from classical optimizations during Chapter 5. In Chapter 3, we examined that there is a research field about optimizing quantum circuits using physical and mathematical properties. Working with these kind of optimizations was out of scope of this thesis.

The programs we used were only improved a small amount. This could be because the programs we used were programmed on a very low-level of abstraction. Nevertheless, it is highly probable that the optimization of heterogeneous quantum-classical programs

becomes more relevant in the future. The development of more abstract quantum programming languages will lead to more compilation steps between the written program and the hardware, which will also lead to a bigger need of optimization routines. Therefore, the optimization of quantum-classical code is something that should be investigated in more detail. We will look into some possibilities for this in the next section.

Within the evaluation process of this thesis in Section 5.3, multiple DDGs of a Quil program were created, as we split the program after each conditional jump. We did this to avoid having to work with circular dependencies and loops. Classical programs execute loops most of the time, which is why optimizing loops has a high impact on program performance [66].

During Chapter 5, we assumed that the instruction time of one instruction would be 1, independent on the type of measurement. In reality, CPUs execute an instruction much faster than QPUs, and the execution time for one instruction in a QPU varies greatly with the qubit technology used (cf. Section 2.1.3). Additionally, the execution time of one-qubit and multiple-qubit gates often differs [3, 4, 5, 6] and one gets latency times due to communication between classical and quantum hardware.

6.2. Future Work

In this thesis, we examined some principles for the optimization of quantum software running on heterogeneous quantum-classical architecture. There are many ways to do further research in this direction.

As previously discussed, the limitation on loops and circular dependencies opens additional fields for future research. It would be interesting to examine how loop optimizations and branch prediction can be applied to a quantum-classical program. One of the simplest additions would be to remove a condition if we can say during static analysis that a result is certain. However, this case can be expected to be the minority of cases, due to the non-deterministic nature of quantum mechanics.

In Chapter 3 we mentioned that there is research on circuit optimization focused solemnly on the quantum circuit. Combining these quantum circuit optimizations with the optimizations we derived from classical procedures is another non-trivial problem that could be interesting to look into.

As we assumed an execution time of 1 per instruction, future research could be about considering real execution times. Additionally, the speed of the hardware is just one property of the hardware to be considered. One challenge of this thesis was that we have only little insight into the concrete hardware architecture of QPUs. When sufficient information about QPU hardware gets available, it would be interesting to consider these as well during the optimization process.

One of the quantum languages we looked at in Chapter 4 was Silq [100]. Silq is special in the way that it is the most abstract language we examined. However, at the moment, Silq cannot be compiled to quantum circuits. Silq combines quantum and classical calculations and is comparably abstract which makes it interesting for future works. Compiling Silq – or a comparable language – to a less abstract form and optimizing it

during compilation would be one step towards abstract quantum languages that demand less quantum mechanical insights of the developer.

7. Statutory Declaration

I hereby state that I have written this master's thesis independently and that I have not used any sources or aids other than those declared. All passages taken from the literature have been marked as such. This thesis has not yet been submitted to any examination authority in the same or a similar form.

Düsseldorf, February 6, 2025

Lian Remme

Declaration on the Usage of Generative AI

Generative AI was used for the following purposes in this master's thesis:

- GitHub Copilot while working on the code as an integrated tool in the integrated development environments (IDEs). It has been used to assist during code-writing, helped writing documentation strings, deciding on variable/-function names and wrote first drafts of some methods/functions.
- GitHub Copilot as an integrated tool in the IDE to write the `README.md` of the GitHub repository corresponding to this thesis [134]. It has been used for formulation suggestions and for Markdown formatting.

A. Appendix

A.1. Code and Data Availability

The code wrote to analyze, optimize and evaluate Quil code in Chapter 5 can be found at [134] (release v0.0.1). The repository contains descriptions of how to compile and execute the code for reproducibility. It provides the functionalities of creating CFGs, creating DDGs and applying the optimization techniques introduced in this thesis.

The repository contains the Quil codes as well as the CFGs and DDGs for quantum teleportation, magic state distillation, repeat-until-success, and IPE, which have been used for the optimizations evaluated in Chapter 5. Additionally, the detailed optimization results can be found at [135].

A.2. List of Abbreviations

CFG	control-flow graph
CLOPS	circuit layer operations per second
CPU	central processing unit
DDG	data-dependence graph
DSL	domain-specific language
GPL	gate programming language
GPU	gate processing unit
HPC	high performance computing
HQCC	heterogeneous quantum-classical computation
IDE	integrated development environment
IR	intermediate representation
ISA	instruction set architecture
IPE	iterative phase estimation
LLVM	low level virtual machine
MLIR	multi-level intermediate representation
MPI	message passing interface
NISQ	noisy intermediate-scale quantum
QASM	quantum assembly language
QCT	quantum calculation time
QIN	quantum instruction number
QIRO	quantum intermediate representation for optimization
QNPUs	quantum network processing units
QPL	quantum programming language
QPU	quantum processing unit
QPT	quantum programming tool
QVM	quantum virtual machine
SDK	software development kit

Bibliography

[1] Peter W Shor. "Algorithms for Quantum Computation: Discrete Logarithms and Factoring". In: *Proceedings 35th annual symposium on foundations of computer science*. Ieee. 1994, pp. 124–134.

[2] Stephen Jordan. *Quantum Algorithm Zoo*. Aug. 23, 2024. URL: https://quantumalgorithmzoo.org/ (visited on 12/29/2024).

[3] IBMQuantum. *ibm_fez*. 2024. URL: https://quantum.ibm.com/services/resources?system=ibm_fez (visited on 12/20/2024).

[4] IBMQuantum. *ibm_marrakesh*. 2024. URL: https://quantum.ibm.com/services/resources?system=ibm_marrakesh (visited on 12/20/2024).

[5] IBMQuantum. *ibm_torino*. 2024. URL: https://quantum.ibm.com/services/resources?system=ibm_torino (visited on 12/20/2024).

[6] Inc. IonQ. *IonQ Aria: Practical Performance*. Jan. 18, 2024. URL: https://ionq.com/resources/ionq-aria-practical-performance (visited on 12/20/2024).

[7] Inc. Atom Computing. *Quantum startup Atom Computing first to exceed 1,000 qubits*. Oct. 24, 2024. URL: https://atom-computing.com/quantum-startup-atom-computing-first-to-exceed-1000-qubits/ (visited on 12/29/2024).

[8] Davide Castelvecchi. "IBM releases first-ever 1,000-qubit quantum chip". In: *Nature* 624.7991 (2023), pp. 238–238.

[9] John Preskill. "Quantum computing and the entanglement frontier". In: *Bulletin of the American Physical Society* 58 (2013).

[10] Krishnageetha Karuppasamy et al. "Quantum Circuit Optimization: Current trends and future direction". In: arXiv:2408.08941 (Aug. 2024), arXiv:2408.08941. DOI: 10.48550/arXiv.2408.08941. arXiv: 2408.08941.

[11] Thomas Fösel et al. "Quantum circuit optimization with deep reinforcement learning". In: *arXiv e-prints*, arXiv:2103.07585 (Mar. 2021), arXiv:2103.07585. DOI: 10.48550/arXiv.2103.07585. arXiv: 2103.07585 [quant-ph].

[12] Francisco JR Ruiz et al. "Quantum Circuit Optimization with AlphaTensor". In: *arXiv e-prints* (2024), arXiv–2402.

[13] Zikun Li et al. "Quarl: A Larning-Based Quantum Circuit Optimizer". In: *Proceedings of the ACM on Programming Languages* 8.OOPSLA1 (Apr. 2024). DOI: 10.1145/3649831. URL: https://doi.org/10.1145/3649831.

[14] Robert S. Smith, Michael J. Curtis, and William J. Zeng. "A Practical Quantum Instruction Set Architecture". In: *arXiv e-prints*, arXiv:1608.03355 (Aug. 2016), arXiv:1608.03355. DOI: 10.48550/arXiv.1608.03355. arXiv: 1608.03355 [quant-ph].

[15] Lov K. Grover. "A fast quantum mechanical algorithm for database search". In: *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*. STOC '96. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1996, pp. 212–219. ISBN: 0897917855. DOI: 10.1145/237814.237866. URL: https://doi.org/10.1145/237814.237866.

[16] Alberto Peruzzo et al. "A variational eigenvalue solver on a photonic quantum processor". In: *Nature Communications* 5.4213 (2014). DOI: 10.1038/ncomms5213.

[17] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information - 10th Anniversary Edition*. 10th ed. Cambridge University Press, 2010. ISBN: 978-1-107-00217-3.

[18] Xiaosi Xu et al. "A Herculean task: Classical simulation of quantum computers". In: *arXiv e-prints*, arXiv:2302.08880 (Feb. 2023), arXiv:2302.08880. DOI: 10.48550/arXiv.2302.08880. arXiv: 2302.08880 [quant-ph].

[19] IBMQuantum. *QPU information — IBM Quantum Documentation*. 2024. URL: https://docs.quantum.ibm.com/guides/qpu-information (visited on 12/16/2024).

[20] Inc. IonQ. *IonQ — Our Trapped Ion Technology*. 2024. URL: https://ionq.com/technology (visited on 12/16/2024).

[21] Paul Adrien Maurice Dirac. "A new notation for quantum mechanics". In: *Mathematical Proceedings of the Cambridge Philosophical Society* 35.3 (1939), pp. 416–418. DOI: 10.1017/S0305004100021162.

[22] Rémy Mosseri and Rossen Dandoloff. "Geometry of entangled states, Bloch spheres and Hopf fibrations". In: *Journal of Physics A: Mathematical and General* 34.47 (Nov. 2001), pp. 10243–10252. DOI: 10.1088/0305-4470/34/47/324.

[23] Daniel Gottesman. "Theory of fault-tolerant quantum computation". In: *Physical Review A* 57 (1 Jan. 1998), pp. 127–137. DOI: 10.1103/PhysRevA.57.127.

[24] William K Wootters and Wojciech H Zurek. "A single quantum cannot be cloned". In: *Nature* 299 (1982), pp. 802–803. DOI: 10.1038/299802a0.

[25] Dennis Dieks. "Communication by EPR devices". In: *Physics Letters A* 92.6 (1982), pp. 271–272. ISSN: 0375-9601. DOI: https://doi.org/10.1016/0375-9601(82)90084-6. URL: https://www.sciencedirect.com/science/article/pii/0375960182900846.

[26] A Robert Calderbank et al. "Quantum Error Correction and Orthogonal Geometry". In: *Physical Review Letters* 78 (3 Jan. 1997), pp. 405–408. DOI: 10.1103/PhysRevLett.78.405.

[27] Leonid Gurvits. "Classical Deterministic Complexity of Edmonds' Problem and Quantum Entanglement". In: *Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing*. STOC '03. San Diego, CA, USA: Association for Computing Machinery, 2003, pp. 10–19. ISBN: 1581136749. DOI: 10.1145/780542.780545. URL: https://doi.org/10.1145/780542.780545.

[28] Roger G. Newton. "Relativity and the Order of Cause and Effect in Time". In: *AIP Conference Proceedings*. Vol. 16. American Institute of Physics. 1974, pp. 49–64. DOI: 10.1063/1.2948448.

[29] Asher Peres and Daniel R. Terno. "Quantum Information and Relativity Theory". In: *Reviews of Modern Physics* 76 (1 Jan. 2004), pp. 93–123. DOI: 10.1103/RevModPhys.76.93.

[30] John Preskill. "Quantum Computing in the NISQ era and beyond". In: *Quantum* 2 (Aug. 2018), p. 79. DOI: 10.22331/q-2018-08-06-79.

[31] Maximilian Schlosshauer. "Quantum decoherence". In: *Physics Reports* 831 (2019), pp. 1–57. DOI: 10.1016/j.physrep.2019.10.001.

[32] Tzvetan Metodi, Arvin I. Faruque, and Frederic T. Chong. *Quantum Computing for Computer Architects*. 2nd ed. Morgan & Claypool Publishers, 2011. ISBN: 9781608456208.

[33] David P DiVincenzo. "Topics in quantum computers". In: *Mesoscopic electron transport*. Springer Netherlands, 1997, pp. 657–677.

[34] Frank Arute et al. "Quantum supremacy using a programmable superconducting processor". In: *Nature* 574.7779 (2019), pp. 505–510. DOI: 10.1038/s41586-019-1666-5.

[35] Inc. Atom Computing. *Technology - Atom Computing*. 2024. URL: https://atom-computing.com/quantum-computing-technology/ (visited on 12/16/2024).

[36] Ya-Chi Liu, Yi-Chung Dzeng, and Chao-Cheng Ting. "Nitrogen Vacancy-Centered Diamond Qubit: The fabrication, design, and application in quantum computing". In: *IEEE Nanotechnology Magazine* 16.4 (2022), pp. 37–43. DOI: 10.1109/MNANO.2022.3175405.

[37] Sergei Slussarenko and Geoff J. Pryde. "Photonic quantum information processing: A concise review". In: *Applied Physics Reviews* 6 (2019). DOI: 10.1063/1.5115814.

[38] Siddharth Chander. "The Current Landscape of Quantum Hardware Development - An Overview". In: *Intersect: The Stanford Journal of Science, Technology, and Society* 17.3 (2024).

[39] Michel H. Devoret and John M. Martinis. "Implementing Qubits with Superconducting Integrated Circuits". In: *Quantum Information Processing* 3.1–5 (Oct. 2004), pp. 163–203. ISSN: 1570-0755. DOI: 10.1007/s11128-004-3101-5.

[40] Eby Sebastian and Ramesh Chandra Poonia. "Compendium of Qubit Technologies in Quantum Computing". In: *International Conference on Communication and Intelligent Systems*. Springer. 2022, pp. 91–100.

[41] Andrew Wack et al. "Quality, Speed, and Scale: three key attributes to measure the performance of near-term quantum computers". In: *arXiv preprint arXiv:2110.14108* (2021). DOI: 10.48550/arXiv.2110.14108.

[42] Intel Corporation. *Intel® Core™, Intel® Core Ultra, and Intel Processor Comparison Chart for Laptops*. Dec. 18, 2024. URL: https://www.intel.com/content/www/us/en/content-details/843334/intel-core-intel-core-ultra-and-intel-processor-comparison-chart-for-laptops.html (visited on 12/20/2024).

[43] IBMQuantum. *Native gates and operations*. 2024. URL: https://docs.quantum.ibm.com/guides/native-gates (visited on 12/23/2024).

[44] Amazon Web Services. *Cloud Quantum Computing Service - Amazon Braket - AWS*. 2024. URL: https://aws.amazon.com/braket/ (visited on 07/22/2024).

[45] Inc. IonQ. *QPU Submission Checklist*. 2024. URL: https://docs.ionq.com/guides/qpu-submission-checklist (visited on 12/23/2024).

[46] Giuseppe E. Santoro and Erio Tosatti. "Optimization using quantum mechanics: quantum annealing through adiabatic evolution". In: *Journal of Physics A: Mathematical and General* 39.36 (2006). DOI: 10.1088/0305-4470/39/36/R01.

[47] Catherine McGeoch and Pau Farré. "The D-Wave Advantage System: An Overview". In: *D-Wave Systems Inc., Burnaby, BC, Canada, Tech. Rep* (2020).

[48] Maarten Van Steen and Andrew S. Tanenbaum. *Distributed Systems*. 3rd ed. Maarten van Steen Leiden, The Netherlands, 2020. ISBN: 978-90-815406-2-9.

[49] Xiang Fu et al. "Quingo: A Programming Framework for Heterogeneous Quantum-Classical Computing with NISQ Features". In: *ACM Transactions on Quantum Computing* 2 (4 2021). DOI: 10.1145/3483528.

[50] Alexander Geng et al. "Quantum image processing on real superconducting and trapped-ion based quantum computers". In: *tm - Technisches Messen* 90.7-8 (2023), pp. 445–454. DOI: 10.1515/teme-2023-0008.

[51] Kun Fang et al. "Dynamic quantum circuit compilation". In: *arXiv preprint arXiv:2310.11021* (2023).

[52] Charles H Bennett et al. "Teleporting an Unknown Quantum State via Dual Classical and Einstein-Podolsky-Rosen Channels". In: *Physical Review Letters* 70 (13 1993), pp. 1895–1899. DOI: 10.1103/PhysRevLett.70.1895.

[53] Thomas Lubinski et al. "Advancing hybrid quantum–classical computation with real-time execution". In: *Frontiers in Physics* 10 (2022). DOI: 10.3389/fphy.2022.940293.

[54] Emanuel Knill. "Fault-Tolerant Postselected Quantum Computation: Schemes". In: *arXiv preprint quant-ph/0402171* (2004). DOI: 10.48550/arXiv.quant-ph/0402171.

[55] Sergey Bravyi and Alexei Kitaev. "Universal quantum computation with ideal Clifford gates and noisy ancillas". In: *Physical Review A* 71 (2 2005). DOI: 10.1103/PhysRevA.71.022316.

[56] Austin G. Fowler et al. "Surface codes: Towards practical large-scale quantum computation". In: *Physical Review A* 86 (3 Sept. 2012). DOI: 10.1103/PhysRevA.86.032324.

[57] Adam Paetznick and Krysta M. Svore. "Repeat-Until-Success: Non-deterministic decomposition of single-qubit unitaries". In: *Quantum Information & Computation* 14.15–16 (Nov. 2014), pp. 1277–1301.

[58] Peng Fu et al. "Proto-Quipper with Dynamic Lifting". In: *Proceedings of the ACM on Programming Languages* 7.11 (POPL Jan. 2023), pp. 309–334. DOI: 10.1145/3571204.

[59] Miroslav Dobšíček et al. "Arbitrary accuracy iterative quantum phase estimation algorithm using a single ancillary qubit: A two-qubit benchmark". In: *Physical Review A* 76 (3 Sept. 2007). DOI: 10.1103/PhysRevA.76.030306.

[60] Stephane Beauregard. "Circuit for Shor's algorithm using 2n+ 3 qubits". In: *Quantum Information & Computation* 3.2 (Mar. 2003), pp. 175–185.

[61] Martina Rossi et al. "Using Shor's algorithm on near term Quantum computers: a reduced version". In: *Quantum Machine Intelligence* 4.18 (July 2022). DOI: 10.1007/s42484-022-00072-2.

[62] IBMQuantum. *Hardware considerations and limitations for classical feedforward and control flow*. 2024. URL: https://docs.quantum.ibm.com/guides/dynamic-circuits-considerations (visited on 12/26/2024).

[63] Amazon Web Services. *Developer Guide Amazon Braket*. 2024. URL: https://docs.aws.amazon.com/pdfs/braket/latest/developerguide/braket-developer-guide.pdf (visited on 09/10/2024).

[64] Amazon Web Services. *Amazon Braket Quantum Computers - AWS*. 2024. URL: https://aws.amazon.com/braket/quantum-computers/ (visited on 12/26/2024).

[65] E. Knill. *Conventions for Quantum Pseudocode*. Tech. rep. Los Alamos National Lab. (LANL), Los Alamos, NM (United States), June 1996. DOI: 10.2172/366453. URL: https://www.osti.gov/biblio/366453.

[66] Alfred V. Aho et al. *Compilers: Principles, Techniques and Tools*. 2nd ed. Pearson Education, 2007. ISBN: 0-321-48681-1.

[67] Yunlin Su and Song Y. Yan. *Principles of Compilers*. Springer, 2011. ISBN: 978-3-642-20834-8.

[68] Frances E. Allen. "Control flow analysis". In: *ACM SIGPLAN Notices* 5 (7 July 1970), pp. 1–19. DOI: 10.1145/390013.808479. URL: https://doi.org/10.1145/390013.808479.

[69] Keith D. Cooper and Linda Torczon. *Engineering a Compiler*. Morgan Kaufmann, 2006. ISBN: 1–55860–699–8.

[70] Dmitri Maslov, Gerhard W. Dueck, and D. Michael Miller. "Simplification of Toffoli Networks via Templates". In: *16th Symposium on Integrated Circuits and Systems Design, 2003. SBCCI 2003. Proceedings.* IEEE. 2003, pp. 53–58. DOI: 10.1109/SBCCI.2003.1232806.

[71] Dmitri Maslov et al. "Quantum Circuit Simplification Using Templates". In: *Design, Automation and Test in Europe*. IEEE. 2005. DOI: 10.1109/DATE.2005.249.

[72] John van de Wetering and Matt Amy. "Optimising T-count is NP-hard". In: *arXiv e-prints* (Sept. 2023). DOI: 10.48550/arXiv.2310.05958.

[73] Daniel Herr, Franco Nori, and Simon J Devitt. "Optimization of lattice surgery is NP-hard". In: *npj Quantum Information* 3.35 (2017). DOI: 10.1038/s41534-017-0035-1.

[74] John van de Wetering et al. "Optimal compilation of parametrised quantum circuits". In: *arXiv e-prints* (Jan. 2024). DOI: 10.48550/arXiv.2401.12877.

[75] Luis Jimnez-Navajas et al. "Quantum Software Development: A Survey". In: 24.7&8 (2024), pp. 609–642.

[76] Amr Elsharkawy et al. "Integration of Quantum Accelerators with High Performance Computing – A Review of Quantum Programming Tools". In: *arXiv preprint arXiv:2309.06167* (Sept. 2023).

[77] David Barral et al. "Review of Distributed Quantum Computing. From single QPU to High Performance Quantum Computing". In: *arXiv e-print* (Apr. 2024). DOI: 10.48550/arXiv.2404.01265.

[78] Bruno Schmitt and Giovanni De Micheli. "Tweedledum: A Compiler Companion for Quantum Computing". In: *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. Mar. 2022, pp. 7–12. DOI: 10.23919/DATE54114.2022.9774510.

[79] boschmitt. *boschmitt/tweedledum: C++17 Library for analysis, compilation/synthesis, and optimization of quantum circuits.* 2025. URL: https://github.com/boschmitt/tweedledum (visited on 02/05/2025).

[80] Matthew P Harrigan et al. "Expressing and Analyzing Quantum Algorithms with Qualtran". In: *arXiv e-prints* (Sept. 2024). DOI: 10.48550/arXiv.2409.04643.

[81] quantumlib. *Measurement and Classical Data · Issue #445 · quantumlib/Qualtran.* Oct. 25, 2023. URL: https://github.com/quantumlib/Qualtran/issues/445 (visited on 12/28/2024).

[82] David Ittah et al. "Enabling Dataflow Optimization for Quantum Programs". In: *arXiv e-prints* (Jan. 2021). DOI: 10.48550/arXiv.2101.11030.

[83] David Ittah et al. "QIRO: A Static Single Assignment-based Quantum Program Representation for Optimization". In: *ACM Transactions on Quantum Computing* 3.14 (3 June 2022), pp. 1–32. DOI: 10.1145/3491247.

[84] dime10. *dime10/QIRO: Source code for the QIRO research project - a novel IR for hybrid quantum program optimization.* 2025. URL: https://github.com/dime10/QIRO (visited on 02/05/2025).

[85] amazon-braket. *amazon-braket/amazon-braket-sdk-python: A Python SDK for interacting with quantum devices on Amazon Braket.* 2025. URL: https://github.com/amazon-braket/amazon-braket-sdk-python (visited on 02/05/2025).

[86] quantumlib. *quantumlib/Cirq: A Python framework for creating, editing, and invoking Noisy Intermediate-Scale Quantum (NISQ) circuits.* 2025. URL: https://github.com/quantumlib/Cirq (visited on 02/05/2025).

[87] NVIDIA. *NVIDIA/cuda-quantum: C++ and Python support for the CUDA Quantum programming model for heterogeneous quantum-classical workflows.* 2025. URL: https://github.com/NVIDIA/cuda-quantum (visited on 02/05/2025).

[88] dwavesystems. *GitHub - dwavesystems/dwave-ocean-sdk: Installer for D-Wave's Ocean tools.* 2024. URL: https://github.com/dwavesystems/dwave-ocean-sdk (visited on 02/05/2025).

[89] team-InQuIR. *team-InQuIR/InQuIR: OpenQL: InQuIR: Intermediate Representation for Interconnected Quantum Compters.* 2025. URL: https://github.com/team-InQuIR/InQuIR (visited on 02/05/2025).

[90] QuTech-Delft. *QuTech-Delft/netqasm.* 2025. URL: https://github.com/QuTech-Delft/netqasm (visited on 02/05/2025).

[91] openqasm. *openqasm/openqasm: Quantum assembly language for extended quantum circuits.* 2025. URL: https://github.com/openqasm/openqasm (visited on 02/05/2025).

[92] QuTech-Delft. *QuTech-Delft/OpenQL: OpenQL: A Portable Quantum Programming Framework for Quantum Accelerators.* 2025. DOI: 10.1145/3474222. URL: https://github.com/QuTech-Delft/OpenQL (visited on 02/05/2025).

[93] quantumlib. *PennyLaneAI/pennylane: PennyLane is a cross-platform Python library for quantum computing, quantum machine learning, and quantum chemistry. Train a quantum computer the same way as a neural network.* URL: https://github.com/PennyLaneAI/pennylane (visited on 02/05/2025).

[94] microsoft. *microsoft/qsharp: Azure Quantum Development Kit, including the Q# programming language, resource estimator, and Quantum Katas.* 2025. URL: https://github.com/microsoft/qsharp (visited on 02/05/2025).

[95] Qiskit. *Qiskit/qiskit: Qiskit is an open-source SDK for working with quantum computers at the level of extended quantum circuits, operators, and primitives.* 2025. URL: https://github.com/Qiskit/qiskit (visited on 02/05/2025).

[96] quil-lang. *quil-lang/quil: Specification of Quil: A Practical Quantum Instruction Set Architecture*. 2025. URL: https://github.com/quil-lang/quil (visited on 02/05/2025).

[97] inQWIRE. *GitHub - inQWIRE/QWIRE: A quantum circuit language and formal verification tool*. 2025. URL: https://github.com/inQWIRE/QWIRE (visited on 02/05/2025).

[98] silq-lang. *silq-lang/vscode-silq*. 2025. URL: https://github.com/silq-lang/vscode-silq (visited on 02/05/2025).

[99] eclipse. *eclipse/xacc: XACC - eXtreme-scale Accelerator programming framework*. 2025. URL: https://github.com/eclipse/xacc (visited on 02/05/2025).

[100] Benjamin Bichsel et al. "Silq: A High-Level Quantum Language with Safe Uncomputation and Intuitive Semantics". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 286–300. DOI: 10.1145/3385412.3386007.

[101] Google Quantum AI. *Cirq — Google Quantum AI*. 2024. URL: https://quantumai.google/cirq (visited on 04/22/2024).

[102] The CUDA-Q development team. *CUDA-Q*. Version 0.7.1. May 2024. DOI: 10.5281/zenodo.11236586. URL: https://doi.org/10.5281/zenodo.11236586.

[103] Shin Nishio and Ryo Wakizaka. "InQuIR: Intermediate Representation for Interconnected Quantum Computers". In: *arXiv e-prints* (Feb. 2023). DOI: 10.48550/arXiv.2302.00267.

[104] Axel Dahlberg et al. "NetQASM—A low-level instruction set architecture for hybrid quantum–classical programs in a quantum internet". In: *Quantum Science and Technology* 7.035023 (June 2022). DOI: 10.1088/2058-9565/ac753f.

[105] Andrew Cross et al. "OpenQASM 3: A broader and deeper quantum assembly language". In: *ACM Transactions on Quantum Computing* 3.12 (3 Sept. 2022), pp. 1–50. DOI: 10.1145/3505636.

[106] Andrew W. Cross et al. "Open Quantum Assembly Language". In: *arXiv e-prints* (July 2017). DOI: 10.48550/arXiv.1707.03429.

[107] IBMQuantum. *OpenQASM 3 feature table — IBM Quantum Documentation*. 2024. URL: https://docs.quantum.ibm.com/guides/qasm-feature-table (visited on 07/18/2024).

[108] Amazon Web Services. *Run your circuits with OpenQASM 3.0 - Amazon Braket*. 2024. URL: https://docs.aws.amazon.com/braket/latest/developerguide/braket-openqasm.html (visited on 12/08/2024).

[109] Nader Khammassi et al. "OpenQL: A Portable Quantum Programming Framework for Quantum Accelerators". In: *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 18.13 (1 Dec. 2021), pp. 1–24. DOI: 10.1145/3474222.

[110] Ville Bergholm et al. "PennyLane: Automatic differentiation of hybrid quantum-classical computations". In: *arXiv e-prints* (Nov. 2018). DOI: 10.48550/arXiv.1811.04968.

[111] Xanadu. *Compiling circuits – PennyLane 0.38.1 documentation*. 2024. URL: https://docs.pennylane.ai/en/stable/introduction/compiling_circuits.html (visited on 10/31/2024).

[112] Krysta Svore et al. "Q#: Enabling Scalable Quantum Computing and Development with a High-level DSL". In: *Proceedings of the Real World Domain Specific Languages Workshop 2018*. RWDSL2018. Association for Computing Machinery, Feb. 2018. DOI: 10.1145/3183895.3183901.

[113] IBMQuantum. *Qiskit — IBM Quantum Computing*. 2024. URL: https://www.ibm.com/quantum/qiskit (visited on 07/14/2024).

[114] Thomas Häner et al. "Distributed quantum computing with QMPI". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. New York, NY, USA: Association for Computing Machinery, Nov. 2021, pp. 1–13. DOI: 10.1145/3458817.3476172.

[115] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.1*. Nov. 2023. URL: https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf.

[116] Rigetti Computing. *Pyquil documentation*. Version 2.3.0. Jan. 2019. URL: https://readthedocs.org/projects/pyquil/downloads/pdf/v2.3.0/.

[117] R. S. Smith et al. "An open-source, industrial-strength optimizing compiler for quantum programs". In: *Quantum Science and Technology* 5.4 (July 2020). DOI: 10.1088/2058-9565/ab9acb.

[118] quil-lang. *qvm: A High-Performance Quantum Virtual Machine*. 2024. URL: https://github.com/quil-lang/qvm (visited on 10/30/2024).

[119] Jennifer Paykin, Robert Rand, and Steve Zdancewic. "QWIRE: A Core Language for Quantum Circuits". In: *ACM SIGPLAN Notices* 52 (1 Jan. 2017), pp. 846–858. DOI: 10.1145/3093333.3009894.

[120] Robert Rand, Jennifer Paykin, and Steve Zdancewic. "QWIRE Practice: Formal Verification of Quantum Circuits in Coq". In: *arXiv e-prints* (Mar. 2018). DOI: 10.4204/EPTCS.266.8.

[121] Alexander J McCaskey et al. "XACC: A System-Level Software Infrastructure for Heterogeneous Quantum–Classical Computing". In: *Quantum Science and Technology* 5.2 (Feb. 2020). DOI: 10.1088/2058-9565/ab6bf6.

[122] IBMQuantum. *Native gates and operations — IBM Quantum Documentation*. 2024. URL: https://docs.quantum.ibm.com/guides/native-gates (visited on 12/06/2024).

[123] Google Quantum AI. *IonQ API Circuits — Cirq — Google Quantum AI*. 2024. URL: https://quantumai.google/cirq/hardware/ionq/circuits (visited on 12/06/2024).

[124] IBMQuantum. *PhaseEstimation (v1.2) — IBM Quantum Documentation*. 2025. URL: https://docs.quantum.ibm.com/api/qiskit/1.2/qiskit.circuit.library.PhaseEstimation#phaseestimation (visited on 01/04/2025).

[125] Xanadu. *qml.QuantumPhaseEstimation – PennyLane 0.39.0 documentation*. 2025. URL: https://docs.pennylane.ai/en/stable/code/api/pennylane.QuantumPhaseEstimation.html (visited on 01/04/2025).

[126] IBMQuantum. *qasm3 (latest version) — IBM Quantum Documentation*. 2024. URL: https://docs.quantum.ibm.com/api/qiskit/qasm3 (visited on 12/08/2024).

[127] Alex McCaskey. *Extensions — XACC 1.0.0 documentation*. 2024. URL: https://xacc.readthedocs.io/en/latest/extensions.html (visited on 12/08/2024).

[128] IBMQuantum. *Qiskit Runtime REST API — IBM Quantum Documentation*. 2024. URL: https://docs.quantum.ibm.com/api/runtime/index (visited on 12/06/2024).

[129] Azure Quantum. *Rigetti provider - Azure Quantum*. 2024. URL: https://learn.microsoft.com/en-us/azure/quantum/provider-rigetti (visited on 11/06/2024).

[130] Inc. IonQ. *Writing Quantum Programs - IonQ Quantum Cloud Documentation*. 2024. URL: https://docs.ionq.com/api-reference/v0.3/writing-quantum-programs (visited on 12/06/2024).

[131] Alex McCaskey. *XACC 1.0.0 documentation*. 2019. URL: https://xacc.readthedocs.io (visited on 01/23/2025).

[132] Amir H Ashouri et al. "A Survey on Compiler Autotuning using Machine Learning". In: *ACM Computing Surveys (CSUR)* 51.96 (5 Sept. 2018), pp. 1–42. DOI: 10.1145/3197978.

[133] Spyridon Triantafyllis et al. "Compiler optimization-space exploration". In: *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* IEEE. Mar. 2003, pp. 204–215. DOI: 10.1109/CGO.2003.1191546.

[134] Lian Remme. *LiRem101/parser-analyser: Analysation and optimization of Quil programs*. Feb. 6, 2025. URL: https://github.com/LiRem101/parser-analyser (visited on 02/06/2025).

[135] Lian Remme. *Supplementary Material for Optimization Strategies for Quantum Computers in Distributed Systems*. Version 0.0.1. Feb. 2025. DOI: 10.5281/zenodo.14823715. URL: https://doi.org/10.5281/zenodo.14823715.