FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

A Generic Micro-Architecture for Quantum Accelerators

João Lourenço Teixeira Vieira

U. PORTO

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Prof.dr. K. L. M. Bertels

July 16, 2021

A Generic Micro-Architecture for Quantum Accelerators

João Lourenço Teixeira Vieira

Mestrado Integrado em Engenharia Informática e Computação

July 16, 2021

Abstract

A comparison between quantum and its classical counterpart is an essential step to defining a blueprint for a quantum computer. While also analyzing their disparities, the one that strikes out the most is the error rates of qubits and quantum gates, being in the order of 10^{-3}, while for CMOS-technology are around 10^{-15}. Physicists are currently researching how to circumvent this problem, but estimations put solutions at least a decade away. K. Bertels compares our current period to the pre-transistor period of the classical computer building.

The error rates already mentioned result from physical qubits' inability to keep their state for long periods of time. This happens for every single experimental platform currently in existence. Quantum gates are an additional contributor to this problem, being also susceptible to error introduction. These factors bring us to distance from real qubits and isolate such problems by concentrating on manipulating a theoretical perfect quantum unit. Qubits in such a simulated system are also referred to as *perfect* since their behavior has no decoherence associated, and their gate operations are fault-proof.

In that context, this master thesis describes the development of a quantum digital micro-architecture that will serve as a medium between a quantum assembly language - cQASM -, and the simulation platform that deals in this kind of qubits - *QBeeSim* -, using C++. The *Quantum Micro-Architecture* here described is general-purposed, as it has no concrete solution oriented design, but should serve as an adaptable structure that requires minimal adjustments to fit any specific area of research. With it, we estimate what our current classical devices allow us in terms of circuit simulation, concluding that fifty qubits should be beyond our limits for a single isolated device.

This work brings us a step closer to having an implementation of the complete full-stack quantum accelerator[11], and to simplifying the process of quantum algorithm development.

Keywords: Computer systems organization, Quantum computing, Quantum micro-architecture, Quantum Assembly, Quantum accelerator

Resumo

A comparação entre quantum e o seu equivalente clássico é um passo essencial para projetar um computador quântico. Ao analisar também as suas disparidades, a mais evidente serão as taxas de erro de qubits e portas quânticas, na ordem dos 10^{-3}, enquanto que, para a tecnologia CMOS, estas estão à volta dos 10^{-15}. Actualmente, físicos investigam formas de contornar o problema, mas estimativas otimistas colocam soluções a uma década de distância. K. Bertels compara o nosso período actual com o período pré-transistor da construção dos computadores clássicos.

As taxas de erro já mencionadas resultam da incapacidade de qubits físicos manterem o seu estado durante longos períodos de tempo. Este fenómeno é comum a todas as plataformas experimentais actualmente existentes. As portas quânticas são um dos contribuintes para este problema, pois estão sujeitas à introdução de erros. Estes factores levam-nos a um afastamento de qubits físicos, de forma a isolar estes problemas, alterando o foco para a manipulação de uma unidade quântica teórica perfeita. Os qubits deste sistema simulado são simultaneamente denominados de *perfeitos*, uma vez que não existe qualquer limite de tempo para manter o estado do qubit válido, e as portas que os manipulam são à prova de erro.

Neste contexto, esta tese de mestrado descreve o desenvolvimento de uma micro-arquitectura digital quântica que servirá como meio entre uma linguagem assembly quântica - cQASM -, e o simulador que lida com este tipo de qubits - *QBeeSim* -, usando C++. A *Micro-Arquitetura Quântica* aqui descrita tem um propósito geral uma vez que não são tomadas decisões que beneficiem soluções concretas, podendo portanto servir como uma estrutura adaptável que requer alterações mínimas para se ajustar a uma área específica de investigação à escolha. Com esta, estimamos a capacidade dos dispositivos clássicos actuais na simulação de circuitos quânticos, concluindo que cinquenta qubits deverão estar além dos limites de um único dispositivo isolado.

Este trabalho aproxima-nos um pouco mais da implementação do acelerador quântico completo [11], e da simplificação do processo de desenvolvimento de algoritmos quânticos.

Keywords: Organização de sistemas de computadores, Computação Quântica, Micro-arquitetura quântica, Assembly quântico, Accelerador quântico

Acknowledgements

To Professor Koen Bertels, for all the patience for our mistakes, willingness to teach us what a good thesis is on his quest (and mine) for clarity, and the constant motivation provided, which helped me work more and better.

To Aritra Sarkar and Imran Ashraf, for shedding light in all not so trivial matters with patience that is hard to beat.

To Yasaman Samadi, without whom it would have been impossible to produce a paper in the tight deadlines we had.

To my friends, who let me blurt out, sometimes much more than they did.

To my family, for whom words would not be enough for any fitting description.

João Vieira

“ *I would rather have questions that can’t be answered than answers that can’t be questioned*”

Richard P. Feynman

Contents

List of Figures

List of Tables

Abbreviations

CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
FPGA	Field Programmable Gate Arrays
GPU	Graphics Processing Unit
ISA	Instruction Set Architecture
NISQ	Noisy Intermediate-scale Quantum
QEC	Quantum Error Correction
QISA	Quantum Instruction Set Architecture
QMA	Quantum Micro-Architecture
QPU	Quantum Processing Unit
SIMT	Single-instruction Multiple-thread
VPC	Virtual Private Cloud

Chapter 1

Introduction

Quantum computing was first introduced by R. Feynman, in 1982[24]. The prospective harnessing of quantum mechanical behaviours has long been an alluring one, but the increasingly obvious inability of classical computers to process the large amounts of data produced globally makes this technology that much more enticing.

1.1 Motivation

Moore's Laws is the observation of how classical computational capacity would improve over the years. If the trend kept going, we would see the number of transistors in a dense integrated circuit double every two years. It so happens that it no longer holds true. Although computational capacity falls short to the expected rate, data production worldwide has never been so high, requiring us to look for alternative ways to handle so much information.

History is kind enough to let us know there is no silver bullet, and optimal solutions require optimal problem-oriented ways to solve specific problems. For instance, when we were unable to increase the way our CPUs handle graphics, we introduced graphic-processing units (GPUs). In the mean time, we also noticed that, while we were focused on solving for an efficient way to process graphics, we were given a way to optimally handle algebraic calculations with large matrices. This shows us two things:

1. The better and more capable system is heterogeneous;

2. While solving a problem, it is unknown how many more we will be solving.

Nowadays, there is no shortage of end-users applications that perform better thanks to heterogeneity, and not only thanks to GPUs, showing how task delegation to specific hardware components is the best strategy.

When there was a lack of hardware capable for testing, in 1994, P. Shor already developed an algorithm[48] capable of surpassing even the most efficient classical algorithm, computational complexity wise. This already showed an optimal use-case for quantum technology. While we do not know what a quantum computer actually is, we do have some expectations on how the technology should evolve in the next decade or so (which is latter explored in Chapter 2). Those expectations leads to the belief that as we developed better quantum hardware, we can use it in the same way we use GPUs, as accelerators, enabling us to factorize a number much faster, as Shor hopped, while the classical system handles tasks that they are already quite good at. This hybrid system would be capable of running hybrid quantum applications.

1.2 Work's purpose

Similar to the development of transistors, quantum bit-wise, we may also call our current times a pre-transistor era. At the moment, we have a multitude of experimental platforms competing with none clearly outperforming the other. With any of those technologies, we have poor quality qubits with abysmal error rates, when compared to CMOS technology - error rates in the order of 10^{-3} against the 10^{-15} in CMOS.

As we wait for quantum hardware capable of handling qubit manipulation trustingly, we alternatively look at simulating this behavior using classical machines, which are not well suited for this task, only the next best thing. For the quantum accelerator model introduced in [11], the simulation only constitutes the very bottom layer, but it is, in reality, the only that will eventually be replaced by quantum hardware. The remaining layers use simulation to progress in their individual development, reducing substantially the expected span between quantum hardware availability and its usability. Additionally, the full-stack layer separation allows for layer abstraction, making it possible to develop quantum algorithms without concerns for qubit routing, for instance. Also, as the simulator in use handles perfect qubits, i.e. that are not susceptible to error (being by factor of time - decoherence - or gate introduced), the developer does not need to worry about faulty results beyond his/hers scope of research.

This thesis focus on developing a software-based *Quantum Micro-Architecture* layer for the full-stack model, while guaranteeing its integration with the surrounding layers, using C++. Here, we look at ways to process a specific kind of assembly language oriented to quantum operations - cQASM[36] -, its delivery method to the simulator, while storing real-time processing information, and the best ways to retrieve and store results away, reducing the memory footprint. Our final conclusions should also answer the question *what are our limits for quantum circuit's simulation*.

1.3 Thesis organization

Chapter 2 gives the essential concepts on quantum computing, that serve as basis for understanding the content of this thesis, as well as the many decisions taken through the course of the development of the micro-architecture. Also, the current state of quantum technology is explored, from quantum hardware to services available.

Chapter 3 presents equally important knowledge on classical and quantum acceleration. We start by understanding classical acceleration to make the bridge into the vision of the full-stack quantum accelerator. Next, the current state of that same stack is described, and its parts explained. Finally, we explain how distancing from NISQ hardware intends to be the way to further develop quantum software that will eventually make use of the full potential of quantum.

Chapter 4 describes the micro-architecture layer as seen from the outside. First, there is a description of how information goes in/out of the layer. Second, we make a first explanation on the possible memory impact that layer could have, without going into the specific implementation. Third, the test process that drove the development of the micro-architecture as is is explained, as well as additional knowledge needed to better understand that same process.

Chapter 5 has the first in-depth look of the micro-architecture, accompanied by an example that goes through every single component, in order.

Chapter 6 builds upon the previous chapter, by looking at the implementation there described, and optimizing several components in terms of runtime and memory usage.

Chapter 7 uses the implementation of the previous two chapters. It exposes the results of the tests described in Chapter 4, followed by a predictive analysis of our current simulation capacity using classical systems.

Chapter 8 offers the conclusions that resulted from this work, as well as future work that takes this thesis as its basis.

Chapter 2

Background

This chapter serves as an introduction to the concepts needed to understand the fundamentals of quantum computing. It also provides and analysis on the state of quantum technology around the world.

2.1 Notions on quantum computing

2.1.1 Qubits

A bit represents a two-state variable, being either 0 or 1, and there is no possibility for an intermediary value. The unitary element of quantum information - the quantum bit or *qubit* - is not as limited. A qubit is a two-level quantum system with two basis states. Its state depends on a linear combination of both basis states in a phenomenon designated *superposition*. Mathematically, this can be described as a combination of two orthogonal vectors

$$\overrightarrow{q} = \alpha \overrightarrow{u} + \beta \overrightarrow{v} = \alpha \begin{pmatrix} 1 \\ 0 \end{pmatrix} + \beta \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \tag{2.1}$$

with α and β being complex numbers associated with the state's vector positioning, in a 2D vector space.

Using Dirac notation, the vectors $\overrightarrow{u}$ and $\overrightarrow{v}$ correspond to $|0\rangle$ and $|1\rangle$, respectively, by analogy to classical bits,

$$|q\rangle = \alpha\,|0\rangle + \beta\,|1\rangle\,. \tag{2.2}$$

At the physical level, although we can manipulate a qubit's state, we are unable to pry into the values of α and β. Instead, when we force the reading of the quantum state, information is lost, as it collapses into one of the two basis. This way *measuring* a qubit is a destructive action.

Still, α and β are indicators that a measurement is a non-deterministic action. Its value is directly related to the probability of obtaining the bit 0 or 1, respectively. As the event space has two possible outcomes (or states) defined, the following equation must hold:

$$|\alpha|^2 + |\beta|^2 = 1 \tag{2.3}$$

When α (β) is equal to 0, we are faced with the pure state $|1\rangle$ ($|0\rangle$), with a 1 probability of obtaining the correspondent bit on measurement. Equation 2.3 also represents the qubit's vector magnitude, which allows us to say that every qubit is a unitary vector.

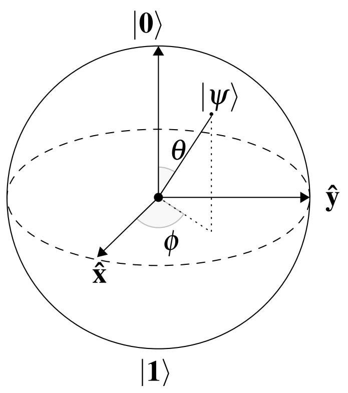


Figure 2.1: Bloch Sphere representation of qubit $|\psi\rangle$

The Bloch Sphere, in Figure 2.1, corresponds to the 3D representation of that last property. It allows for an intuitive visualization of a qubit's state. This representation additionally shows that a quantum state can be defined using the two angles θ and ϕ, according to

$$|q\rangle = \cos\frac{\theta}{2}\,|0\rangle + \sin\frac{\theta}{2}e^{i\phi}\,|1\rangle\,. \tag{2.4}$$

2.1.2 Entanglement and coherence

Similarly to classical systems, we increase the outcome event space by introducing more qubits. For a number of bits n, since every single one can take up to two different values, we have in total 2^n possible states, with only one active at a time. Quantum superposition negates this last fact, as multiple states can be active. Superposition results in an exponential state space, one of the pillars for quantum speedup. The application of the tensor product to each individual and independent qubit gives the representation of many-qubit states

$$|\psi\rangle = \bigotimes_{i=0}^{n-1} a_i|0\rangle + b_i|1\rangle = \bigotimes_{i=0}^{n-1} \begin{pmatrix} a_i \\ b_i \end{pmatrix}, \tag{2.5}$$

resulting in the association of a complex value α with each possible state

$$|\psi\rangle = \begin{pmatrix} \alpha_0 \\ \alpha_1 \\ \dots \\ \alpha_n \end{pmatrix}. \tag{2.6}$$

As for a single state, each value α squared corresponds to the probability of measuring that particular state. If no dependence is created between any qubits, i.e. all are independent, $|\psi\rangle$ is unitary. The same does not hold for entangled qubits. *Entanglement* is the association of multiple qubits making them indistinguishable, without independent description. This phenomenon does not affect our capability to represent a qubit's state as we previously did in 2.6, but it imposes limitations on manipulating those states. Transitioning from Equation 2.6 to 2.5, i.e., the decomposition as a qubit tensor product, becomes unfeasible.

For quantum hardware, creating entangled states requires the interaction of qubits, which poses a challenge in the development of usable qubits. We simultaneously expect inter-qubit communication, otherwise there would be no entanglement; and qubit isolation, since interactions with the environment affect the quantum state's maintainability[30]. The loss of its state, or *decoherence*, is an essential parameter in quantum computing. It places a time limit for qubit manipulation, consequently limiting computation capacity, since it is not possible to clone a quantum state, as stated per the *Noncloning theorem*[22].

2.1.3 Quantum gates

All qubit's state manipulations correspond to the application of rotations along the Bloch Sphere axes. In mathematics, these manipulations translate to having an operator A, capable of affecting a quantum state as follows

$$A|q\rangle = |q'\rangle. \tag{2.7}$$

To maintain the state's validity, $|q'\rangle$ must remain unitary. On the Bloch Sphere, the new state's representation must remains on its surface. For this reason, A must be a *unitary operator*, and we call the transformation of $|q\rangle$ to $|q'\rangle$ also *unitary*.

The *Pauli matrices* or I, X, Y, Z-gates are defined as

$$\begin{aligned} I \equiv \sigma_0 &= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \\ X \equiv \sigma_x = \sigma_1 &= \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \\ Y \equiv \sigma_y = \sigma_2 &= \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \\ Z \equiv \sigma_z = \sigma_3 &= \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, \end{aligned} \tag{2.8}$$

and are commonly used unitary operators or *quantum gates*. Take the X-gate as an example:

$$\begin{aligned} |q'\rangle &= X|q\rangle \\ &= X(\alpha|0\rangle + \beta|1\rangle) \\ &= \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \\ &= \begin{pmatrix} \beta \\ \alpha \end{pmatrix} = \beta|0\rangle + \alpha|1\rangle. \end{aligned} \tag{2.9}$$

It is clear that the vector's magnitude remains the same. This case is the quantum equivalent to the classical *NOT* gate, as it switches a qubit basis states' amplitudes.

Similar to the multi-qubit state representation in Equation 2.6, unitary operators are not limited to handling a single qubit at a time. For example, it is common for one qubit to act as a boolean controller to the application of a gate on another qubit, like in the *controlled-not-gate* (or *CNOT-gate*)

$$CNOT = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}, \tag{2.10}$$

with the application of the *X-gate* on the second qubit depending on the first one. In practice, this is the same of taking all states where the control qubit is set, and switching their amplitudes with those where only the non-control qubit is flipped.

The number of qubits involved in a single gate is unlimited. Any gate can be applied to a number n of qubits, as long as it still is a unitary operator of dimension $2^n * 2^n$. The *Toffoli gate* is a three-qubit gate that follows these rules, with similar function as the *CNOT gate*, but with yet another control qubit. For that reason, it is also known as a *CCNOT gate*:

$$CCNOT = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}. \tag{2.11}$$

When applying any quantum gate on physical qubits, interaction is required, i.e. there is a need to break qubit isolation, which affects the system's state and so, introduces errors. In a naive prediction, if we consider the probability of introducing an error p every time we apply a gate, we can estimate that using at least $1/p$ gates would most certainly produce a random result. Although there are ways to avoid such strict constraints, quantum gate error rates are one of the main problems quantum technology faces. The necessary time to apply a quantum gate is equally limiting. Even without the pseudo-limit number of gates $1/p$, if the total time needed to run all gates required exceeds the limit given by decoherence, the result is equally wrong.

Multi-qubit gates also play an important role demonstrating important quantum computing properties. The Deutch algorithm uses two-qubit gates to demonstrate quantum parallelism. The algorithm proves if a function f with boolean input is constant or not, without calculating $f(0)$ and $f(1)$ separately, as required in the classical case. This fundamental property is partially at fault for the belief in quantum speedup[22].

2.1.4 Quantum circuits

The notion of applying consecutive operators on qubits was introduced already. It happens to correspond to the definition of *quantum circuit*. The graphical depiction of quantum circuits has horizontal lines representing qubits. Specific gates have their own representation, and their placement indicates the qubits affected and order of operations.

Consider the *EPR-Bell states*, that are a particular example of quantum entanglement:

$$\begin{cases} |\Phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \\ |\Psi^+\rangle = \frac{1}{\sqrt{2}}(|01\rangle + |10\rangle) \\ |\Phi^-\rangle = \frac{1}{\sqrt{2}}(|00\rangle - |11\rangle) \\ |\Psi^-\rangle = \frac{1}{\sqrt{2}}(|01\rangle - |10\rangle) \end{cases} \tag{2.12}$$

The circuit corresponding to the creation of the first Bell state $|\Phi^+\rangle$ is shown in Figure 2.2.

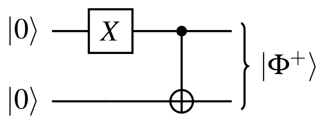


Figure 2.2: Quantum circuit to generate the first *EPR-Bell state*

The mathematical equivalent to the diagram is the following sequence:

$$CNOT_{q_0,q_1} \cdot H_{q_0}. \tag{2.13}$$

Note the inversion in the operations' order, which is consistent with ordering multiple consecutive matrices for multiplication with an input state vector.

Another important concept relates to different ways of representing the same circuit. Figure 2.3 shows two equivalent circuits. By placing gates that affect different qubits at the same depth level, we are compressing the circuit and identifying gate parallelism. Since the X-gate and the H-gate shown do not relate, they can operate at the same time, reducing the overall circuit depth. In the end, we keep respecting the pre-established order of operations, maintaining the final result, but reducing the execution time. This may look meaningless on a small scale but, as we increase circuit complexity, say, for example, up to twenty thousand (20.000) gates, the cumulative savings may be significant.

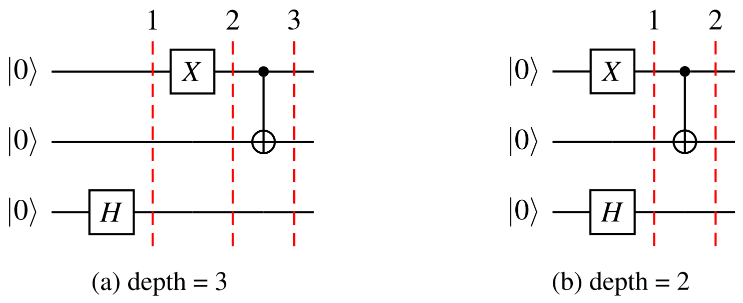


Figure 2.3: Circuit depth representation

Useful quantum circuits are bound to have at least two-qubit gates connecting multiple qubits. For physical qubits, this requires consideration for *Nearest-neighbour (NN) constraints*, as qubits' interactions require physical proximity. In those cases, additional procedures to physically approximate qubits may be necessary. Eliminating those completely may be impossible, but even reducing its necessity poses a challenge with a complexity degree proportional to the circuit's own complexity. Depending on the effort put into mapping virtual to physical qubits, these operations' overhead may be reduced substantially, which contributes to reducing the circuit runtime.

2.2 State of quantum hardware

Both the super-classical properties of quantum, and the development of algorithms showing classical-hard problems as quantum-easy, sustain quantum computing's potential. In turn, this estimated potential supports the effort to develop better quantum hardware and explore potential use-cases, on a global scale. Companies and governments alike play their part in the quantum race that intensified, at a steady pace, these last few years. Global giants like Google, IBM, Intel, Microsoft, and lesser known companies such as Xanadu, IonQ, Honeywell and Zapata are important contributors to the on-going research, as are the United States, Europe and China.

2.2.1 Qubit errors

Compared to CMOS-technology, which offers error rates of about $10^{-15/-16}$, quantum chips, independently of the underlying technology, display rates in the order of $10^{-2/-3}$. Making the parallelism to classical development, this places us in a pre-transistor or pre-qubit phase[11]. With many possible solutions for qubit implementation, it is unknown how many (if any) will remain in the future. Semi-conducting and superconducting qubits seem to be common approaches widely used. Nonetheless, research continues on other fronts like photonics, topological, NV center, graphene, and trapped-ion with identical results.

Qubit Type	**Gate Fidelity**		**Gate Time**		**Coherence Time**
	Single-qubit	**Two-qubit**	**Single-qubit**	**Two-qubit**	
Superconducting[37]	0.99+	0.997	10 - 40 ns	30 - 4600 ns	50 - 100 μs
Semi-Conducting[20]	0.99+	0.94+	0.25 - 100 ns	0.8 - 40 ns	30 ns - days
NV Centers[23][21][41]	0.999+	0.992	<8 μs	8 μs	1.8+ ms
Graphene[49]	-	-	-	-	10 μs*
Photonic[45][43][38]	0.997	0.84	-	-	100+ ms
Trapped-ion[18][17]	0.999999	0.996+	<1 μs	<100 μs	600 s
Topological[39]	-	-	-	-	-

Table 2.1: Qubit technology characteristics comparison

Table 2.1 shows how these experimental platforms relate to each other regarding essential reference values[1], that serve as indicators for its viability. In short, experimental platforms are far from the desirable level in all fronts, so the next few years should be a period of quantum hardware improvement. It is the enhancement of this underlying technology that will eventually allow us to explore the full potential of quantum.

2.2.2 Scaling the quantum processor

Some questions regarding how quantum processors will scale remain unanswered, and should only be answered with experimentation. For example, we still don't know how scaling up the number of qubits will impact qubit-qubit interactions. The noise produced by an increasing number of

[1]values marked with * are only theoretical

low-quality qubits could threaten accuracy levels. To solve this problem, qubit isolation needs refinement.

Improving qubit isolation, i.e. improving its quality, would diminish the overall error rates, and, consequently, the overhead that results from *Quantum Error Correction* (QEC). QEC techniques use groups of qubits to create a more reliable unit - the *logical qubit* - with one data storage qubit - *data qubit* - and many for error detection - *ancilla qubits*. Surface code[26, 15, 40] was the most popularized technique for QEC up to a few years ago[52, 51, 50]. The way to apply surface code may vary, as neither the shape or size of the logical qubit are fixed, since the number of qubits and their relative position changes. According to the version used in [12, 54], for a single data qubit, in a distance-*d* logical qubit, there are $n = d^2$ qubits in total. This shows that the authors are using square-shaped logical qubits. We know this because, for surface code in general, the total number of physical qubits is calculated as follows:

$$n = l_x * l_z, \tag{2.14}$$

where, for any rectangular-shaped logical qubit, l_x (l_z) corresponds to the length (width) of said logical qubit on the plane of reference. The variables l_x and l_z are the *weight* of the Pauli operators *X-gate* and *Z-gate*, respectively, hence the name. For the same general specification, the distance-*d* of the logical qubit also depends on these two values:

$$d = min(l_x, l_z). \tag{2.15}$$

An increase in the code distance represents a proportional increase in accuracy, but so do the classical computations, and delay introduced to apply error correction[16]. Now, let us consider square-shaped logical qubits, i.e. $l_x = l_z$. The table bellow shows how costly it is to increase the logical qubit distance, by associating the distance-*d* with the total number of physical qubits needed.

distance-*d*	**Number of physical qubits**
3	9
5	25
7	49
13	169
...	...
27	729

Table 2.2: Cost of surface code in physical qubits

In 2018, it was due to this extravagant cost that Preskill[46] dismissed surface code entirely. QEC research turned to small-codes, that require fewer physical qubits per logical qubit.

Independent of which technique is used to correct qubit errors, improving qubit quality would mean an immediate increase in the trust of quantum devices results. For the long-term future, this means a higher number of qubits per chip, that is only limited by a comparatively lower noise to signal ratio. Of those chip qubits, more will partake in useful computation, considering logical qubits would be substantially smaller for the same degree of fidelity. In this phase, QEC will reach the peak of its foreseeable relevance, being a crucial part of reliably scaling a quantum chip.

2.2.3 NISQ-era technology

NISQ is the term designated by Preskill[46] to define quantum hardware for the next decade. The term comes from the erroneous behaviour of qubits (noisy), and our incapacity to make a large-scale integration of many qubits, expected to have around fifty to a hundred qubits (intermediate-scale). At the moment, the best devices registered have more than seventy (70) qubits[53, 55], which is well within those limits. These devices are seen as a necessary step towards a future where quantum reaches its true estimated potential. Until then, the usefulness of quantum devices will remain limited. Nevertheless, access to the technology could prove useful for quantum algorithmic development, similar to how first classical machines allowed for many advancements, independent from the lack of theoretical basis.

2.2.4 Quantum problem-solving

NP-hard problems are classical hard problems that will remain hard for quantum. There are particular cases of these problems that require only approximate solutions to be considered useful. For those, quantum devices could prove more effective in reaching the intended approximation, although this remains a theoretical idea, and NISQ technology could not be enough to prove it. In the short term, hybrid quantum-classical algorithms are another emerging possibility to make NISQ devices useful, with optimization problems fitting well the criteria to be adapted as such. For the long view, it remain essential to mark which problems are classical hard, but quantum easy.

2.2.5 Resources availability

Having an at-home quantum device is not an within our current reality. Cloud-based services like IBM's[2], Xanadu's[1], Alibaba's[5] allows quantum technology to reach the masses. They are an effort to push quantum algorithms development, by providing tools that deal with and access quantum systems. These services are connected to either quantum simulators or actual quantum devices that run on different experimental platforms.

2.3 Conclusion

Here, the basic notions required for understanding quantum computing have been presented. The concept of qubit was introduced, as were quantum gates, and its effects on qubits. Circuit representation was also explained, and how all those interactions can be understood with the help of linear algebra concepts and tensor mathematics. At the same time, superclassical properties were identified and explained. The remaining of the chapter described the current quantum technology in existence, identifying its weaknesses and what to expect in the years to come. Resources availability and their uses were also explored. In short, quantum technology does not meet the maturity level required for usability, and mixing its problems with computer engineering ones represents an insurmountable challenge. In that sense, an abstraction from those properties will be suggested in the next chapter, as the divide-and-conquer strategy seems to be the best fit.

Chapter 3

Quantum Acceleration

Chapter 2 explained how quantum hardware remains far from the desirable level. This area's advancements of the past fifteen years make us expect a Turing quantum computer's arrival in the next decade. The remaining time gives us enough leeway to consider the engineering problems involved in producing such a device. For one, the "quantum computer" term is loosely employed, since the technology is a better fit for accelerating a classical chip[11]. There are many tasks that the classical computer already performs efficiently and we don't expect quantum devices to outperform them. In that sense, both technologies should coexist, where quantum technology would be equivalent to other classical accelerators, responsible only for a designated group of tasks. In this chapter, we will start the analysis on the following:

- Classical accelerators, in order to make a bridge to the quantum one;
- The quantum accelerator full-stack, as presented in [11];
- What assumptions we can make to move forward in the engineering challenges involved in building a quantum accelerator.

3.1 Classical hardware heterogeneity

For computer architecture, it was long agreed that core homogeneity is far from ideal. With that in mind, accelerators like GPUs and FPGAs improved through the years. These components represent an extension of the computer architecture, and its processing capacity. During its design process, engineers look at a group of specific tasks intending to reduce their required computational time.

3.1.1 How GPUs work

Heterogeneous systems intend to break homogeneity bottlenecks, enhancing the performance of many applications. The type of the accelerator is oriented to the specific task it handles. Nevertheless, the inner workings of different accelerator types have commonalities, making this analysis relevant. The remaining of this section is focused on GPUs specifically.

Initially, by having task-specific processors, the applications of *graphical processing unit* (GPU) were limited to graphics and visual computing. The current implementation allowed for the introduction of scalable parallel programming models and software platforms, like CUDA. That development made it easier to benefit in other areas that thrive on the same implementation, making them proficient in large data-parallel problem-solving. The term unified GPU architecture refers to this latter development, consisting of a parallel array of many general programmable processors. Those are highly scalable and enable its parallelism capabilities to increase according to Moore's law.

Disconnecting from CPU design principles, the focus is on the efficiency of many-core parallel threads, whose cores are simpler but optimized for data-parallel behavior among groups of threads. Today, GPUs are multiprocessors composed of multiprocessors, as represented in Figure 3.1.

Each multiprocessor's parallelism provides powerful localized performance, and comprehensive multi-threading support. This constitution fits fine-grained parallel programming models incredibly well.

3.1.1.1 Implementation details

All *streaming processors* (SP) have a shared memory accessible by every thread using a low-latency interconnection. Its processors are multi-threaded multiprocessors as each has various *scalar processors* (SPs) responsible for the majority of operations. The SP is hardware multi-threaded, having a *register file* (RF) coupled. RF allocation is optimized by the compiler, balancing expenses of register splitting versus thread costs. For a well-defined set of functions, there are *special function units* (SFUs). Unique interfaces are also defined, allowing for external memory load, store, and atomic access operation - memory interface - for example, that plays a fundamental role in non-graphical computation.

The implementation described is associated with an execution model called *single-instruction multiple-thread* (SIMT), intended to manage and execute the available threads. The concept of

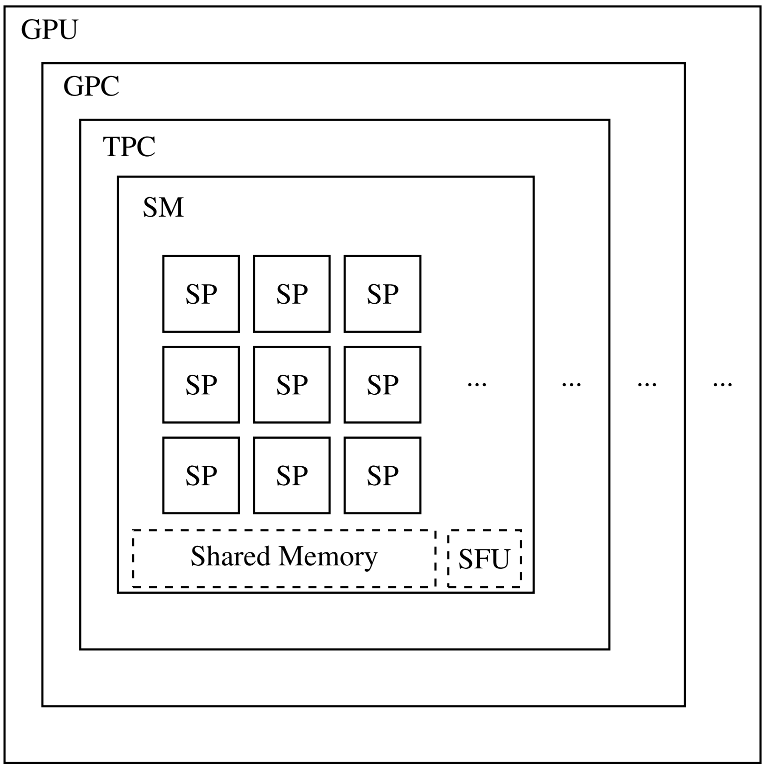


Figure 3.1: Basic unified GPU architecture core's representation

warps represents the association of concurrent threads, facilitating its overall operations. A *warp* can be the entire thread block, but it is also possible to divide a thread block into multiple *warps*. There is only a single thread type on any warp, and all its threads start together at the same program address; the rest of the execution is independent. The issue of an instruction to a *warp* is secured by the SIMT multi-threaded instruction unit, broadcasting the SIMT instruction, in a synchronous process, to the totality of active *warp* threads. The SIMT processor manages the individual threads, and is optimized to find data-level parallelism among threads at run-time. Full efficiency occurs when every thread on a *warp* takes the same execution path. Divergence due to data-dependent conditional branching requires execution serialization for each branch, and ultimately convergence on a single execution path. The resulting delays do not affect any other warps, as their executions are independent of each other. The only existent dependency relates to the same warp's sequential instructions.

According to its type, a multiprocessor controller accumulates and packs both work requests and input data into SIMT *warps* to allow its execution. Depending on the program's requirements, the controller manages the distribution of shared resources to warps, namely registers. These requirements can cause processors to stall, as allocation depends entirely on availability, which in turn depends on other *warps* completing the totality of its threads, enabling for resources to be unlocked.

3.1.1.2 Instruction execution

SP thread processors handle instruction execution for singular threads. The compiler generates intermediate assembler-level instructions to be later optimized and translated into binary GPU microinstructions. The definition of an *Instruction Set Architecture* (ISA), like the NVIDIA PTX, guarantees GPU generation compatibility, as hardware instructions evolution is isolated from ISA generator tools. This ISA works with virtual registers that are later analyzed for dependencies and allocated to real registers. It also passes through a process of code simplification, where unneeded parts vanish, instruction folding is applied whenever possible, and SIMT branch split points are optimized. NVIDIA PTX has a particularity of allowing for behavior specification with a single thread. Furthermore, its capabilities for memory load/store operations make it possible to support commonly used languages, like C/C++. To allow for this possibility, there are three memory spaces available on GPUs:

- Local memory per thread, for private temporary data, implemented on the external DRAM;
- Shared memory, intended for low-latency access to shared data for same thread block cooperating threads, on chip's SRAM;
- Global memory, allowing large data sets storage and sharing between all threads, also on the external DRAM.

The existence of these different memory spaces inevitably requires the definition of barrier synchronization instructions to avoid race conditions. These barriers block processor cycle consumption completely, and announce its waiting state to the scheduler, which modifies the barrier counter when all threads are in the same state, resuming individual execution.

3.1.1.3 Memory characteristics

Accessing external DRAM is expensive. Individual parallel thread requests from the same SIMT *warp* are coalesced to request for a single memory block when possible. This arrangement represents a significant improvement to operation costs.

A GPU's performance aligns with its memory subsystem capabilities, and so the following properties are crucial to avoid bottlenecks:

- Wide, referent to the number of data conveying pins, that have to be numerous;
- Memory array with many DRAM chips, supporting total data bus width;
- Fast, employing aggressive signaling techniques that maximize data rates per pin;
- Efficient, utilizing every cycle for successful data transference;
- Intended use of compression techniques, both lossless and lossy;

- Cache and work coalescence for off-chip traffic reduction and value of data transference cycles increase.

Even if we employ strategies for work coalescence, there will be numerous uncorrelated requests. That makes related request accumulation vital, as it increases the value of the data transference. On average, this strategy increases latency, and it may lead to the processing unit's starvation, resulting in neighboring processors becoming idle. This pitfall makes careful address selection for storage necessary.

Graphical related computation typically relies on large sets of data. In fact, those datasets' size makes it unthinkable to implement a large enough cache to hold them completely, opting for streamed cache. Compared to typical CPUs' hit rates, GPUs are much lower, at around 90%. The frequent necessity of in-flight misses handling is a hard problem. With a high number of threads and the frequent need to retrieve cache data, bandwidth is equally a concern, making an on-chip placement optimal.

Other implemented memories represent specific needs of GPU calculations. *Constant memory* enables the storage of read-only scalar values common to a SIMT *warp* on shared memory. *Texture memory* stores data in large arrays with read-only access, functioning as a throughput optimizer of texture fetches from concurrent threads. Independently of its intended use, it can serve as cache for any *global memory* data.

3.1.1.4 Case analysis and conclusion

O(n) algorithms speedup is a particular case deserving of attention. A clear obstacle relates to data transference. PCIe bus transference rates are different from CPU's memory access speeds, with the latter being multiple times lower. There are three options to overcome this bottleneck:

1. Maintaining the data on the GPU increases the value of each data movement;
2. Allowing for concurrent processing and transferring of data eliminates useless computation cycles;
3. Arguably, the most essential notion to retrieve from this analysis is related to work distribution to the processors available, treating the system as the heterogeneous platform and recognizing its advantages.

This section was based on [44].

3.2 Full-stack quantum accelerator

Quantum technology's logical next step is to become an accelerator for specific tasks where it excels. As shown in Figure 3.2, similar to the others of the same kind, a quantum accelerator or *quantum processing unit* (QPU) would function as a separate processing unit that connects to the main CPU, serving as an extension to its computational capabilities.

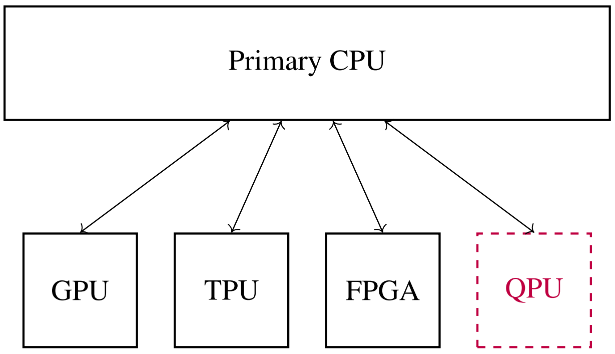


Figure 3.2: Heterogeneous computer system

When abstracting away from problem-oriented peculiarities, some components remain, and are the basis for any accelerator. This allows for a starting point when defining a full-stack in quantum. From the higher to lower levels, we can identify four elements:

- High-level logic that abstracts the low-level details;
- Compiler;
- Assembly kind of language;
- Micro-architecture closely linked to the acceleration hardware.

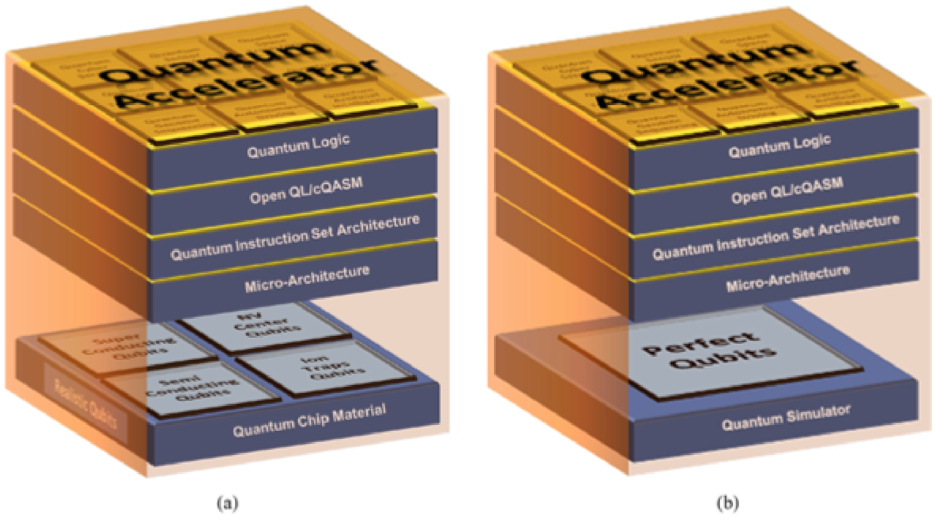


Figure 3.3: Quantum accelerator stack on [10]. a) supposes a quantum devices as bottom layer, while b) uses a simulator

Research that dates up to 2004[10] already express this structure, as in Figure 3.3. The same group updated its view on the topic with an evolved stack[11], that serves as the basis for this thesis, and is shown in Figure 3.4.

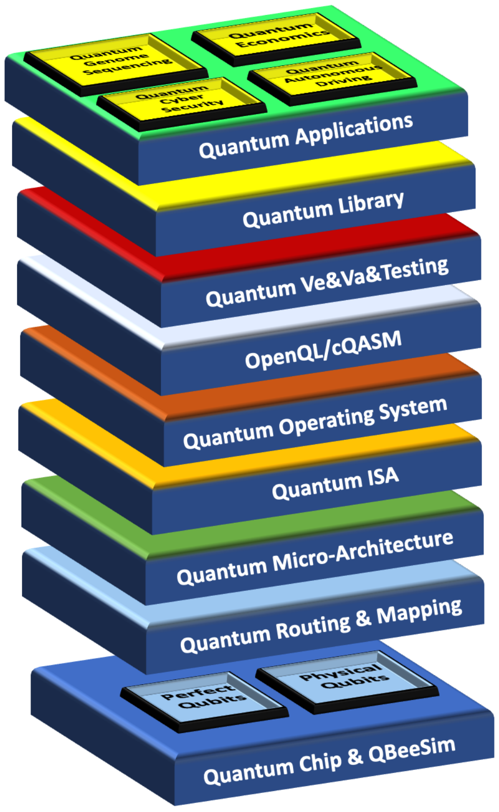


Figure 3.4: New quantum accelerator full-stack representation

From top to bottom, the degree of concern towards the implementation of qubits increases. This isolation by layer means, for example, that quantum applications, at the top, are concerned with algorithmic development for specific domain problem-solving while dismissing what qubits are specifically being used in the chip. This focus is vital in demonstrating commercial viability in many fields like genome sequencing[47], finance[14], and chemical and materials research[8].

3.2.1 Stack overview

To develop a commercial application, we need first a quantum programming language. This language must express the nuances of quantum computing in a human-friendly way, focused, not on the execution on specialised quantum hardware, but the application's demands. This is the meaning of the first few layers, that are also the most technology-agnostic. The high level description of the application must use a framework like OpenQL[35], that has a defined structure, allowing for validity and reliability confirmation of the logic defined. The *quantum library* would eventually contain various algorithms that still need to be developed world-wide and that can be re-used for various domains. So there is a clear link between the quantum application and the quantum library. This library can be developed inside one´s own organisation, or be in the public domain.

For the group's concrete implementation of the stack, OpenQL is associated with its own compiler. The compiler's final product may either be cQASM[36] or eQASM[28], only depending on the hardware that will run the application. cQASM is intended for simulators like *QBeeSim* to execute it. If ever we connect to a real physical quantum chip, we must translate the cQASM version in eQASM, where the prefix 'e' stands for the executable assembler. This separation came out of the team's research for Intel as they had to control both a semiconducting as well as a superconducting qubit[29, 28]. Both these QASM variations must abide by the underlying system requirements, but being already capable of execution, the *Quantum OS* only needs to adjust them to fit its criteria. Through this work, the *Quantum Instruction Set Architecture* (QISA) is expressed in cQASM (as this thesis focus on simulator runs), serving as input to the *Quantum Micro-Architecture*. In the micro-architecture layer, classical and quantum technology interactions are crucial. Here we solve the challenges that digital-analog interactions (and vice-versa) entail, like instructions execution timing, and results reading and storing.

Quantum technology-related constraints like qubit topology, specifically communication overhead or qubit's routing, require unique addressing, in the shape of instruction timing manipulation, or even insertion of new commands. Specifically, qubit proximity constraints imply a careful mapping of virtual qubits to logical ones and possibly, the introduction of routing operations that are ever-evolving along the application run. Even though there is the possibility to associate a quantum device at the lowest level, the focus will remain, as already expressed, on using *QBeeSim*.

The remaining of this section is dedicated to explaining the already developed components of the stack, due to their strong connection to the developments of this thesis.

3.2.2 OpenQL

OpenQL[35] is a framework that abstracts away the low-level requirements of quantum technology. It allows for algorithmic definition, in C++ or Python, that is translated by its associated compiler. The compiler allows for compilation and optimization of quantum code based on a configuration file, where the low-level specifications are detailed. The final result comes as low-level cQASM, that can be translated into eQASM, if needed.

Figure 3.5 represents a random quantum circuit that will be used as an example. Listing 3.1 describes that same example circuit using OpenQL.

```
from openql import openql as ql
# ...
# requires a platform definition in which the QASM will run
config_fn = os.path.join(curdir, 'hardware_config_qx.json')
platform = ql.Platform("platform_none", config_fn)
# number of qubits in the circuit
nqubits = 8
# create a program (container of kernels)
p = ql.Program("p1", platform, nqubits)
# create a kernel
```

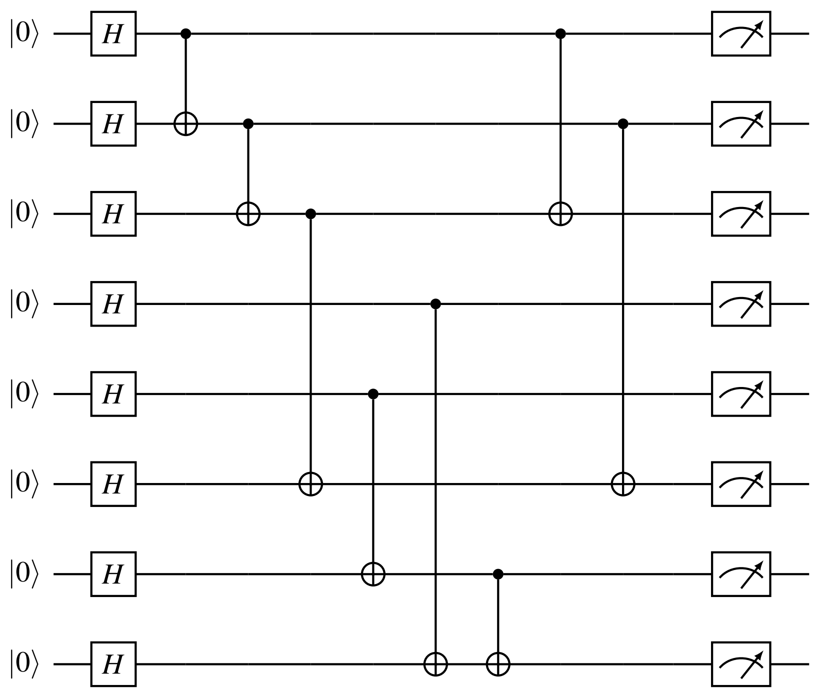


Figure 3.5: 8 qubits quantum circuit

```python
k = ql.Kernel("k1", platform, nqubits)

# populates the kernel
for i in range(nqubits):
    k.gate('prepz', [i])

for i in range(nqubits):
    k.gate('h', [i])

k.gate('cnot', [0,1])
k.gate('cnot', [1,2])
k.gate('cnot', [2,5])
k.gate('cnot', [4,6])
k.gate('cnot', [3,7])
k.gate('cnot', [6,7])
k.gate('cnot', [0,2])
k.gate('cnot', [1,5])

for i in range(nqubits):
    k.gate('measure', [i])

# add the kernel to the program
p.add_kernel(k)

# compile the program
```

```
p.compile()
```

Listing 3.1: Python example for the 8 qubits quantum circuit with OpenQL

The definitions that relate to Python configurations of OpenQL were omitted.

There are some circuit independent parts of the code that deserve attention. Particularly, the JSON configuration file imported in line 4, which associates the quantum code with the low-level hardware intended to run it. The following five sections split a valid configuration file:

1. `hardware_setting`, for the definition of hardware limitations, like the number of qubits available and clock cycle to consider
2. `topology`, for qubit position understanding
3. `resources`, to express qubit relations, like available interactions
4. `instructions`, where available quantum gates are expressed, and their properties identified
5. `gate_decomposition`, for the association of non-defined gates to an equivalent sequence of identified ones

To compile code intended for an actual quantum device is what makes the majority of these sections relevant. A quantum simulator like QX[34] or *QBeeSim* relieves most of the constraints, making all but sections numbered as 4. and 5. negligible.

3.2.3 cQASM

Common QASM, or cQASM, is intended for algorithmic description, distancing itself from qubit technology needs. For instance, qubit routing is often needed to respect *NN-constraints*, allowing for multi-qubit gates to work properly. The degree of proximity needed may be highly linked to the quantum hardware technology in use, requiring different manipulations in accordance. This is why such operations should be left alone until we reach the QISA level. To sum it up, cQASM specifies the quantum gates that need to be applied, while on-chip qubit movement instructions are introduced at the QISA level. Listing 3.2 shows the generated cQASM code for the circuit in Figure 3.5.

```
qubits 8

.k1
    { prep_z q[0] | prep_z q[1] }
    { h q[0] | h q[1] }
    { prep_z q[2] | prep_z q[3] | prep_z q[4] | prep_z q[6] | prep_z q[7] | cnot q
        [0],q[1] }
    { h q[2] | h q[3] | h q[4] | h q[6] | h q[7] }
```

```
8    { prep_z q[5] | cnot q[1],q[2] | cnot q[4],q[6] | cnot q[3],q[7] }
9    h q[5]
10   { cnot q[2],q[5] | cnot q[6],q[7] }
11   { measure q[0] | measure q[1] | measure q[2] | measure q[3] | measure q[4] |
         measure q[5] | measure q[6] | measure q[7] }
```

Listing 3.2: cQASM code generated for the 8 qubits quantum circuit

Like its Python counterpart, a program or circuit is a sequence of *kernels*, that requires an initial definition of the number of qubits involved through its entirety. *kernels* are instructions' groups, defined as seen in line 3. An array-like fashion is used to reference qubits by their index, for example, in gate application. For optimization, OpenQL joins independent operations, involving them in curly brackets. Line 4 illustrates the possibility of parallelizing qubits 0 and 1 preparation, i.e. setting them to the $|0\rangle$ state at the same time. On a circuit, this is equivalent to placing both operation at the same depth level. While there is no circuit symbol to indicate qubit preparation, consider that qubits are only prepared at the points where its circuit line starts, as shown in Figure 3.6, where the original circuit of Figure 3.5 is adapted to better represent OpenQL's optimizations of Listing 3.2.

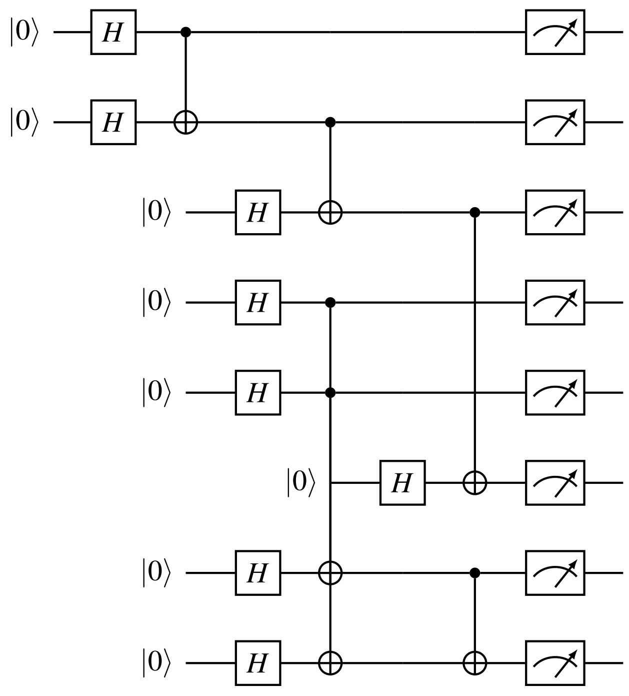


Figure 3.6: Adaptation of the circuit in Figure 3.5 according to cQASM optimizations in Listing 3.2

Additionally, quantum simulators allow us to have access to the qubit states' amplitudes at any moment, using the `display` command, which is impossible for physical devices. This possibility represents an advantage for quantum algorithm development, specially for debugging purposes.

3.2.4 QBeeSim

QBeeSim was developed in C++. Similar to how we need to translate low-level instructions into analog signals for quantum hardware, the simulator requires us to do the parsing of cQASM, and translate it into C++ instructions that it understands. Every instruction will then manipulate complex numbers, since they constitute the matrices and vectors for quantum gate operators representation and quantum states storing, respectively. Those complex numbers make use of the C++ `std::complex<T>`. When instantiating a *complex*, choosing its elements' type becomes a decision between precision and memory cost, as the more memory its type needs, the more precise it is. This decision is an extremely important one, as state memorization is the most expensive process done in quantum simulation. To understand the why behind this statement, consider the following: for a selected data type size S_{type}, considering an n number of qubits, quantum states occupy a total size calculated by

$$S_{state} = 2 * S_{type} * 2^n, \tag{3.1}$$

while each gate's matrix would result in a total memory expense of

$$S_{gate} = 2 * S_{type} * (2^n)^2. \tag{3.2}$$

In its C++ implementation, S_{type} should range from 4 bytes to 12 bytes, corresponding to choosing the *float* data type and the *long double*, respectively. This means the values of S_{state} and S_{gate} are also in bytes.

Using 1D arrays - the sequence of its rows - to store this elements improves locality and results in lower memory-related overhead. This is advantageous for the overall simulator's performance[19]. Gate operations are efficient as QBeeSim quickly dismisses unimportant states with no amplitude - *Zero-state skipping*. Each operation also runs in parallel for the remaining states, which is possible since all qubit storage is done in a super-positioned state of all initialized qubits.

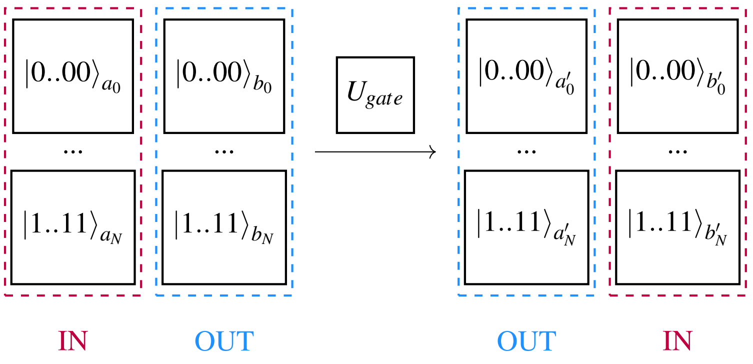


Figure 3.7: QBeeSim quantum states memory model

Its use of two different arrays for operations' input and output storage results in a shrinkage of delays associated to moving data around after almost every operation, and facilitates parallelism. This advantage comes from changing each array's purpose post every operation, as shown in

Figure 3.7. There, our current state, or the gate's input, is on the left (IN) vector, and the amplitudes that result from the application of the *U-gate*, or its output, are on the right (OUT) vector; next the IN (OUT) vector is labeled as the OUT (IN) vector, and all states in the new OUT vector are made into *zero-states*, becoming ready for the application of another gate. This setup avoids an otherwise necessary move of results from one array to the other. As a result, memory usage for state storage is constant through the simulator's run and corresponds to

$$2 * S_{state}. \tag{3.3}$$

Note that zero-states are also preserved during runs, which allows for the direct association of corresponding states to their storage index. This avoids the allocation of space for explicit state indication, which is why we simply use arrays instead of maps.

3.3 Distancing from NISQ quantum hardware

The immaturity of quantum technology leaves us with to many problems to handle at the same time. With real qubits, even if we are focused on solving high-level problems, our results are influenced by the underlying technology. The developments here proposed are made under the assumption of *perfect or virtual qubits*. Perfect qubits are tailored for testing the correctness of the quantum logic, by not factoring decoherence or errors introduced by gate application or measurement. With them, the focus remains on engineering problems, allowing for the advancement of the rest of the stack's layers while not waiting for a good enough experimental platform. Virtual qubits go a step further, dismissing *NN-constraints*.

One day, we expect qubits to be sufficiently good to allow quantum hardware usability, with error rates that reach at least $10^{-4/-5}$. At that point, they can be called *realistic qubits*[11], and we may even be able to use QEC to help us scale quantum accelerators. Despite being far from that stage, with simulators like QBeeSim, it is possible to introduce limitations, like *NN-constraints* and topology restrictions. These restrictions make the behavior of perfect qubits comparable to logical qubits, allowing for a bridge to be maid, where the advancements of the former can be applied to the latter.

To reiterate, by deciding to level in perfect qubits, we abstract ourselves away from the QEC methodologies and experimental platforms, that are currently sub-optimal. Nonetheless, we can still achieve breakthrough in terms of manipulation of logical qubits, giving a solid basis on how logical qubits are routed and mapped. This would ultimately make the transition to physical qubits smother, as the remaining steps would be almost limited to translating operations associated to logical qubits into operations linked to physical qubits.

3.4 Conclusion

This chapter starts by presenting the relevance of computer heterogeneity. The inner works of a classical accelerator, the GPU, was explained to identify common points expected in any kind of accelerator. Those points were enumerated and the parallelism was made to define a QPU. The full-stack of the quantum accelerator and its evolution was explained, as were its core components described in a more in-depth manner. At last, the assumptions for the development of quantum software were laid.

To reiterate, a quantum machine is optimal for a certain group of tasks, as are classical systems. This lead to the believe that a quantum computer should instead be a quantum accelerator. To allow the developments that will be presented in the next chapters, we assume *perfect qubits*. This assumption allows the development of quantum software without waiting for reliable quantum hardware.

Chapter 4

Quantum Micro-Architecture Layer Framing

In Chapter 3, we explained how to define a quantum accelerator. From here on, the content becomes more specific, with the *micro-architecture layer* being the object of research. To be able to develop a generic quantum micro-architecture, first we need to have a concrete understanding of how we expect it to operate. Specifically, we need to detail how a developer's application code should be described to serve as input for the micro-architecture. Understanding its output is equally important, and considering that we are using *QBeeSim*, the options are expanded. Here, we consider the micro-architecture a black-box, so we can define input and output, understand how it impacts the system, and also make an early definition of our expectations, without going to deeply into how it works. With that in mind, the following sections are divided as follows:

- Analysis of the information flow of the micro-architecture;
- Micro-architecture's memory impact;
- Testing process explanation: experimental setup, and concrete approach to testing.

Our analysis takes into account not only the number of qubits and gates, but also the overall circuit design on its impact on memory usage and circuit processing time.

4.1 Information flow description

Previously, we concluded that quantum is a good fit for hardware acceleration, so we see the *quantum accelerator* as the first real-world implementation of the technology, and most likely the dominant one on how we use quantum in the next decade or so. That analysis is based on quantum being the way to speedup only specific tasks. With that principle in mind, the application's developer should have a way to differentiate between classical and quantum components of the application. C++ already integrates the `#pragma` statement, allowing for additional information to be passed to the compiler. In that sense, a `#pragma quantum` should precede a group of QPU intended instructions:

```
[C++ operations]

#pragma quantum ...
{
    [cQASM operations]
}

[C++ operations]
```

In Figure 4.1, we have two main areas: the blue one (micro-architecture) and the green one (accelerator's classical controller). The specified C++ operations are intended for Host CPU, in the green area. In that same area, the arbiter function is to allow those same instructions to reach the Host CPU, while all cQASM operations (inside a `#pragma quantum`) are redirected to the Micro-Architecture, with its results being retrieved by the Exchange Register File, which allows for their access on the Host CPU.

Additionally, the `#pragma` statement should be able to specify some kind of storage, to where the result of the quantum operation is loaded at the end:

```
#pragma quantum store(results) ...
{
    [cQASM operations]
}
```

Similarly, an option for results loading should also exist, opening the possibility to have cQASM code that depends on the results previously calculated:

```
#pragma quantum store(results) ...
{
    [cQASM operations]
```

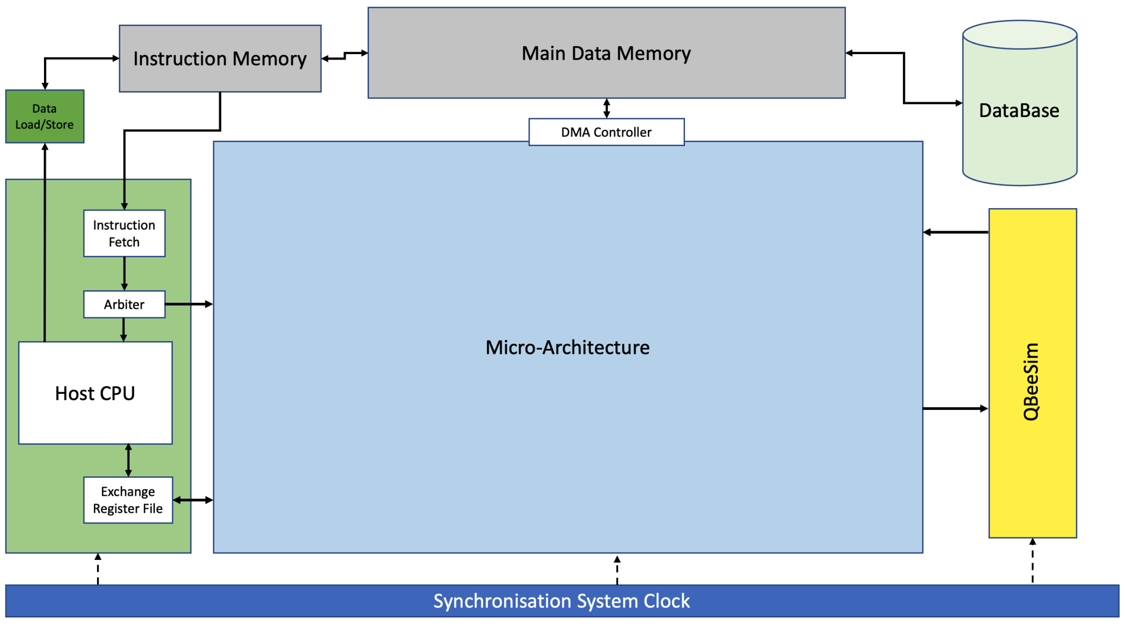


Figure 4.1: Quantum Micro-Architecture as a black-box

```
}

[C++ operations]

#pragma quantum load(results) store(new_results)
{
    [cQASM operations]
}
```

With a concrete example, let us consider an initial search over a database, using Grover's algorithm[1], followed by the application of our result oriented circuit, based on the intermediate result that we got from Grover's:

```
#pragma quantum store(results) ...
{
    [Grover's Algorithm cQASM operations]
}

std::bitset<S> result;

// Processing the most probable result
```

[1]The algorithm is latter explored near the end of this chapter

```
std::map<std::bitset<S>, float>::iterator it = results.begin();
while (it != results.end())
{
    std::bitset<S> state = it->first;
    float prob = it->second;

    //Assuming 51% as a good limit
    if(prob >= 0.51) {
        result = state;
        break;
    }
    it++;
}
// End of processing

#pragma quantum load(result) store(new_results)
{
    [cQASM operations]
}
```

The result loading on the last `#pragma` statement - in the end - should be equivalent to manipulating cQASM code. For example, if the search yields the state $|00101\rangle$, we prepare all qubits to state $|0\rangle$, and apply the *X-gate* to all qubits in the $|1\rangle$ state, as follows:

```
//State preparation is implicit
#pragma quantum store(new_results)
{
    { X q[0] | X q[2] }
    [Original cQASM operations]
}
```

4.1.1 Results representation

To transmit the results back to the developer, we see two obvious options:

1. Storage in an array;
2. Storage in an associative container (hash map).

(1) has a clear downside that requires us to record every single amplitude that the simulator has access to, as the only indicator for the state comes as the array's index. This means that the order of storage represents the state that the stored amplitude refers to.

(2) stores elements formed by a combination of a *key* value and a *mapped* value. Since each element specifies both state (as key) and amplitude (as mapped), we loose the dependency to the storage order. This approach allows us more flexibility to only save part of the simulator's stored state amplitudes.

For now, know that we opted for (2), and will refer to it from now on as *map*. The why behind this approach is object of analysis in the next section.

4.1.2 QBeeSim versatility

When using QBeeSim, the processing and execution of quantum instructions remains digital. Despite all downsides of simulation, we are able to access the amplitudes associated with each state. While developing an algorithm, it is useful to have access to such details, making it logical that the first possible output are those same amplitudes.

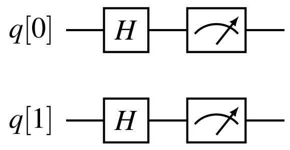


Figure 4.2: Simple example circuit

For the example circuit of Figure 4.2, we get the following amplitudes:

```
results = {
    0: (0.5, 0),
    1: (0.5, 0),
    2: (0.5, 0),
    3: (0.5, 0)
}
```

In the above results *map*, the *key* - on the left - identifies the state and the *mapped* value - on the right - is its amplitude. For example, in `2: (0.5, 0)`, `2` represents the state $|10\rangle$, while `(0.5, 0)` is its amplitude $\alpha = 0.5 + 0i$. By retrieving both state and amplitude, we are eliminating the need to save all states' amplitude. As quantum algorithms are supposed to produce results that are highly disperse in the total event space, meaning that we will only have a few non-zero or relevant states, we are ultimately saving memory.

As a second option, to imitate the results given by a physical device, we should only retrieve the probability of each state:

```
results = {
    0: 0.25,
    1: 0.25,
        2: 0.25,
        3: 0.25
    }
```

Here, `2: 0.25`, for example, shows that state $|10\rangle$ has a probability of 25%. We have the option to completely replicate the process of a physical chip, and run the algorithm multiple times, until we are satisfied with the results. On the other hand, digital simulations simplify the process, since we can derive the probabilities from the amplitudes after a single run. This last method would be equivalent to running the algorithms on a quantum chip n times, with $n \to \infty$.

4.2 Memory usage

The *Quantum Micro-Architecture* developed processes cQASM code to run on a quantum simulator. In Chapter 3, we already described how we can calculate the total memory used by *QBeeSim*. It remain true that, for any quantum circuit of reasonable size, *QBeeSim* is the main memory consumer. Figure 4.3 shows the evolution of the number of states during a circuit run. The amount of memory required is on par with the number of states stored. So, the state's count ceiling comes from superpositioning all qubits in the circuit, as we have to store a total of 2^n state's amplitudes, with n being the total number of qubits. In contrast, the number of states (and, consequently, expected memory usage) floor logically happens when we don't have any qubit superposition. At those times, we only have to record each qubit basis states' amplitudes, giving a total of $2 * n$ state's amplitudes. Due to *QBeeSim* own optimizations, this behavior is not followed, as it would require both state and amplitude storage, making it more efficient to have a stable memory use during the circuit run, which only happens with constant full superposition. This means that instead of having the gradual growth shown in Figure 4.3, if we were to place it in a memory graph, we get an instantaneous rise to the maximum memory required in the beginning, and an equally quick drop at the end of the circuit. In comparison, the *QBeeSim* peaks are always considerable lower than what we get from gradually fluctuating the number of qubits in superposition.

Overall, the *Quantum Micro-Architecture* memory needs are much lower. Even if we ran a 100.000 gates circuit, we can opt for processing these gates in limited-sized *batches*, meaning that at each moment, the total memory required for instruction parsing and processing is capped. For the *Quantum Micro-Architecture*, the worst-case scenario in terms of memory requirements happens when we need to store intermediate or final results, triggered by a `display` instruction, i.e. when we access *QBeeSim* states and pull them to the *Quantum Micro-Architecture*. Considering that *QBeeSim*[19] works with two arrays for memory storage, with each having enough space to store all states; if we apply the same result storing strategy, we will need at maximum, half the memory used by *QBeeSim*. Still, this would be extremely inefficient, given that we are targeting useful or purposeful *quantum algorithms*, which are most commonly very sparse in their results space, as opposed to random *quantum algorithms*. As we direct the *Quantum Micro-Architecture* to be optimized in this direction, we chose to store the amplitudes and the state they

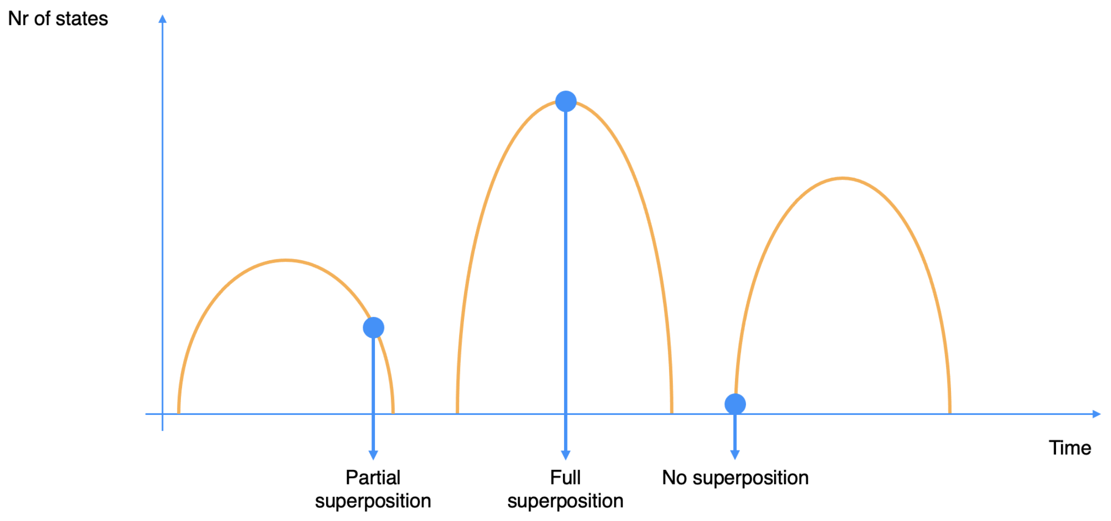


Figure 4.3: Optimal variation of the number of stored states

refer to in a *map*. This way, we can simply dismiss allocated space for non-valuable states, as was presented in the previous section. The memory used S_{state} in this situation is calculated by

$$S_{state} = (2^n - z) * (2 * S_{type1} + S_{type2}). \tag{4.1}$$

The equation has two key points:

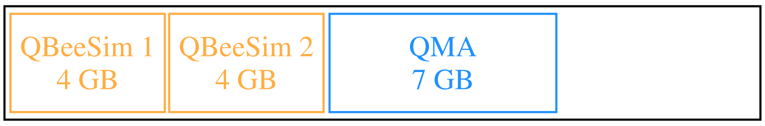
- For the total number of states resulting from the superposition of all qubits 2^n, we are only interested in the non-zero amplitude states $2^n - z$, with z being the number of zero-amplitude states;
- For each of those states, we have S_{type1} representing the size of the data type chosen to store both the real and imaginary parts (that being the why the value is doubled) of the complex amplitudes, and S_{type2} for the data type chosen for state storing; so each state+amplitude combo amounts to $2 * S_{type1} + S_{type2}$.

This option is undoubtedly worst for *non-useful* or *random quantum algorithms*. For instance, if applied the *H-gate* on every qubit and immediately after we record the results, all states would have a non-zero amplitude, $z = 0$. In this case the total memory would still be inferior to *QBeeSim*, but would reach up to three fourths of its memory. This fact becomes more obvious if we consider $S_{type} = S_{type1} = S_{type2}$, so the total memory for state storing in the *Quantum Micro-Architecture* can be calculated by

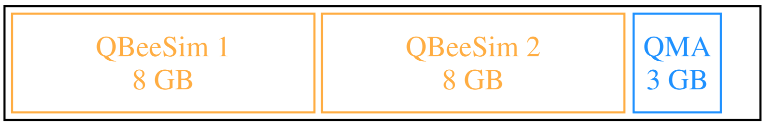
$$S_{state} = 2^n * 3 * S_{type}. \tag{4.2}$$

But again, we aim for usefulness, meaning we try to reduce the impact of the *Quantum Micro-Architecture* in terms of memory usage, allowing *QBeeSim* to use a higher number of qubits.

Figure 4.4 illustrates what we mean by this. In 4.4a we represent the worst case possible, where all state's amplitudes are recorded, even the zero valued ones. This ends up consuming three fourths of the total space of *QBeeSim* (sum of the two state vectors of the simulator). Applying the optimization for useful *quantum algorithms*, we only record the useful state, as is represented in 4.4b, reducing the impact of the *Quantum Micro-Architecture*, and even allowing the expansion of the circuit to one more qubit.



(a) Worst case-scenario with n qubits



(b) Best case-scenario with $n+1$ qubits

Figure 4.4: Proportional RAM consumption by the *Quantum Micro-Architecture* and *QBeeSim*, on 20 GB of RAM

4.2.1 Practical demonstration of sparsity

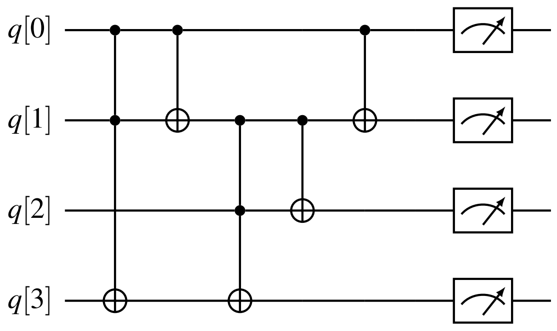


Figure 4.5: Quantum full-adder

Let us consider the quantum full-adder[4], shown in Figure 4.5. In this example, *q[0]*, *q[1]* and *q[2]* are the input bits, with the first two being the bits to be summed and the last one being the carry in bit. Given the reversibility of quantum circuits, we also need an additional qubit *q[3]*. For the same reason, on output, *q[0]* and *q[1]* still have the same values but *q[2]* is now storing the sum of the first three qubits, and *q[3]* holds the carry out.

If all qubits are initialized, similar to how we would do it in a classical machine, to basis states, the output of the circuit would be a single state. For example, if we initialize the qubits to the state $|0101\rangle$, by applying an X-gate on *q[0]* and *q[2]*, we expect the output to be the single state $|1001\rangle$.

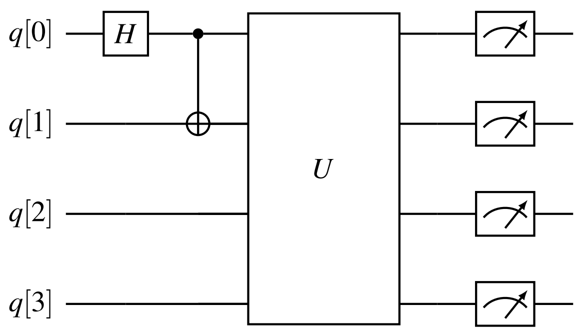


Figure 4.6: Bell state preparation for the quantum full-adder represented as U

Now we will see, in Figure 4.6, that the result space remains sparse if we do the initialization process with operations only available for quantum:

- We start by preparing an *EPR-Bell state* between *q[0]* and *q[1]* prior to starting the quantum full-adder, here represented by U;
- If we maintain the prior initialization of *q[2]* and *q[3]*, i.e. states $|1\rangle$ and $|0\rangle$, respectively, the final result will be either $|0100\rangle$ or $|1111\rangle$, with equal probabilities;
- Compared to the sixteen possible states generated by four qubits, the result space is still very thin, demonstrating the usefulness of the optimization.

4.3 Testing process

In this context, testing refers not to the verification and validation of the software-based micro-architecture developed, but to experimenting with different circuits, by changing its number of gates and/or qubits. This allows us to have a notion on how the bottom layers of the stack react to different inputs, making it possible to draw conclusions mainly related to scalability.

4.3.1 Hardware-independent analysis

It is clear, by looking at Table 4.1, that a classical machine with more memory would allow an increase in the number of qubits that *QBeeSim* can simulate. Also, having defined an optimization for the *Quantum Micro-Architecture*, we also do not expect that said layer would act as the system's bottleneck, especially on useful algorithms. Hardware-independent testing is based on these two facts: since, for any random hardware used for testing, the impact in the system caused by the *Quantum Micro-Architecture* is always a tier lower than one cause by the simulator, the specifications of the classical computer used are irrelevant for the task of analysing the micro-architecture's performance. This leads to dismissing strict time-based performance, preferring to evaluate the relative performance of running *QBeeSim* alone against running it with the *Quantum Micro-Architecture* on top, while using the same hardware. The objective of such tests is to have

Qubits	**Memory Usage**	**Available Computational Capacity**
5	1 024 B	
10	32 768 B	
15	1.04 MB	
20	33.56 MB	
25	1.08 GB	Base-model Raspberry Pi 4 can run it
30	34.36 GB	Surpasses common 16 GB laptops
35	1.10 TB	
40	35.18 TB	Limit of AWS EC2 High-memory[7]
45	1.12 PB	
50	36.02 PB	Above supercomputers (like Fugaku[6])

Table 4.1: Estimated memory usage for state storage of *QBeeSim* using *double* data type

a certain degree of comparison against our most time-consuming layer - *QBeeSim*. Similarly, it is also important to analyse the overall memory consumption variation through time, for those same situations.

4.3.2 Connecting to a database

Although a database connection causes additional overhead, hindering performance and making the previous comparison flawed, the benefits outweigh the costs as we need a way to store algorithmic processing information, to help on overall layer analysis. Therefore, a previously developed database was used, with an interaction layer placed directly on top of *QBeeSim*. Along with other capabilities, a connection to this database allows for storing system properties, like memory consumption, and recording intermediary and final states. This is an important step to guarantee solution quality.

Originally, the database, whose EER Diagram is shown in Figure 4.7, was intended for local use, i.e. having the *MariaDB* server on the same system used to run *QBeeSim*. To make it scalable, we opted to move it to *Amazon AWS*, placing it in a *RDS instance*. By having this stand-alone RDS instance for the database, we are capable of recording data that comes from multiple systems in the same place. This way we are facilitating the tracking and comparison between different hardware solutions. Note that, although we said that we are not interested in comparing hardware, this is crucial as a business requirement. That same RDS instance was placed in an *Amazon Virtual Private Cloud* or VPC, becoming accessible from multiple Amazon *EC2 instances* created in the same physical location, that can run the full-stack quantum accelerator, and connect to it simultaneously, without being exposed to the internet. The solution described is represented in Figure 4.8.

4.3.3 Random circuit testing

Random circuit generation goes a long way when we want to validate the processing of quantum instructions. It is also the simplest solution from which we can draw conclusion related to

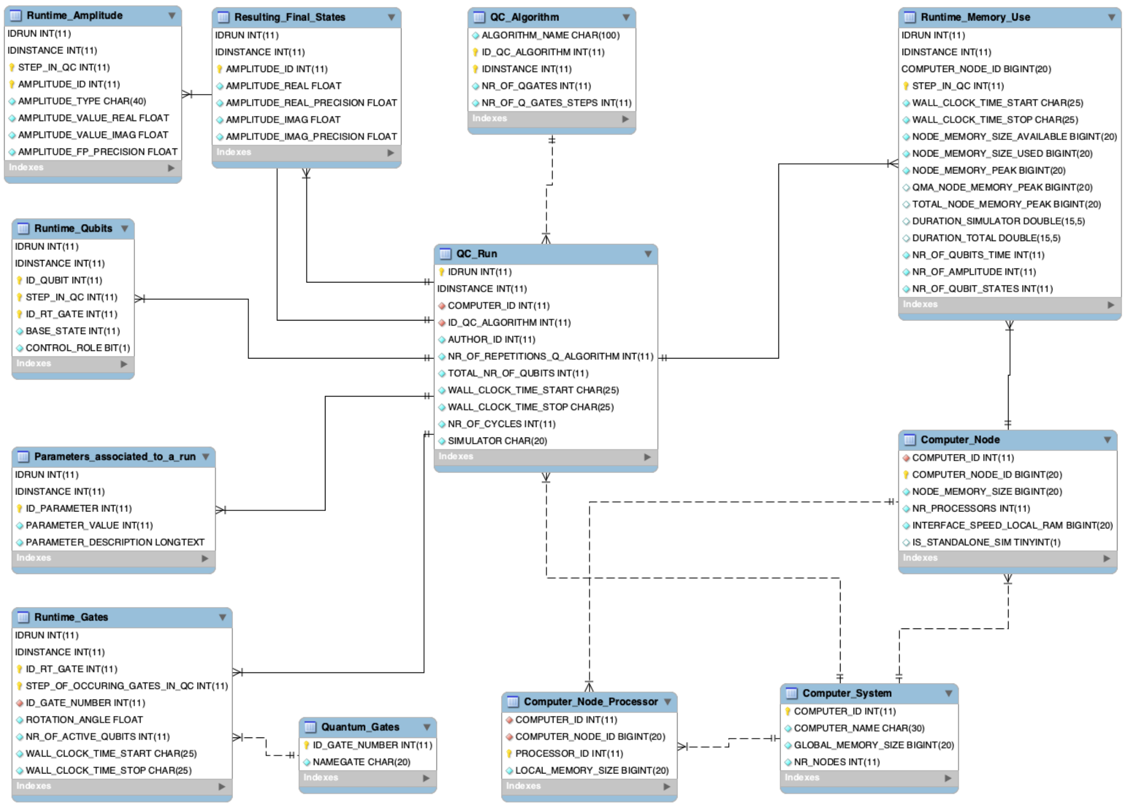


Figure 4.7: Database EER Diagram

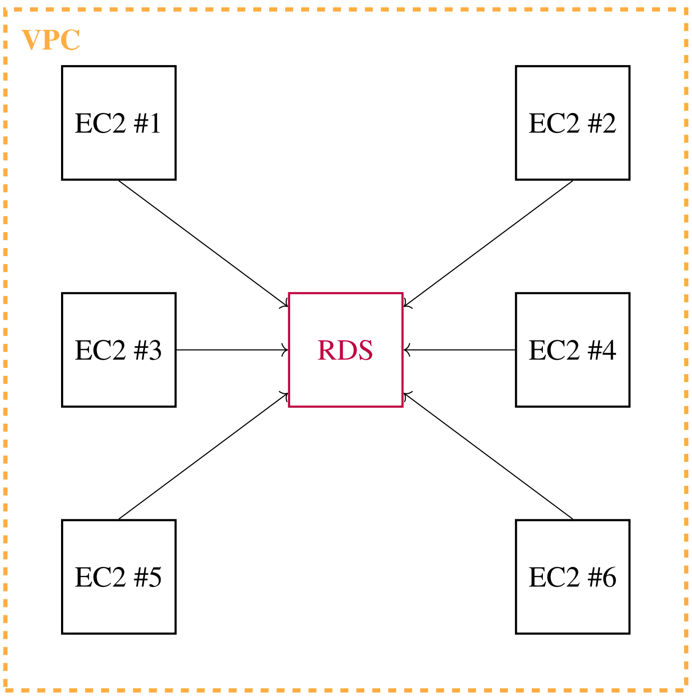


Figure 4.8: AWS setup

scalability of both gates and qubits. In that sense, we generated multiple circuits that have a variable number of qubits, and go from a few gates up to multiple thousands. These conclusions will remain limited, nonetheless, as random circuits do not explore the results storage optimization envisioned, making them more indicated for the time-based analysis. For a memory analysis, using a concrete algorithms seems more appropriate.

4.3.4 Grover's algorithm for useful circuit testing

Grover's algorithm[31, 13] is commonly used due to its ability to perform searches with a quadratic speedup. This kind of speedup is essentially optimal[9], and has proven itself useful as a subroutine for more complex quantum algorithms[33]. Essentially, it can be divided into three distinct phases:

- **Initialization**, to create an uniform superposition, by applying the *H-gate* on all qubits;
- **Oracle** U_w, that encodes the information, marking which state(s) we want to find - *winner state(s)* $|w\rangle$[3];
- **Diffuser** U_s, responsible for the amplification of the probability of *winner state(s)*.

There are two distinct possibilities on how to encode the Oracle[25], with their own denomination: you can either have a *boolean oracle* or a *phase oracle*, using the $|0\rangle$ and $|1\rangle$, or the $|+\rangle$ and $|-\rangle$ basis, respectively, hence the names. Compared to the phase oracle, the boolean oracle requires an additional ancilla qubit, which is similar to what we expect when using the classical analog of the Grover's algorithm. In the end, they are equivalent in function, and so, for our tests, it is indifferent which one we choose. Due to the practicality of scaling the algorithm, we opted for a phase oracle, which will now be explained.

4.3.4.1 Using a phase oracle

To describe how we can implement a phase oracle U_w, let us consider that we have a three qubit system. As described above, the first step places the system in uniform superposition

$$|\psi\rangle = \frac{1}{\sqrt{8}}(|000\rangle + |001\rangle + |010\rangle + |011\rangle + |100\rangle + |101\rangle + |110\rangle + |111\rangle). \tag{4.3}$$

Take note that we will now define the current superposition as

$$|s\rangle = |\psi\rangle\,. \tag{4.4}$$

Next, we mark the winner state(s). Let us, for this example consider $|w\rangle = |110\rangle$. The application of the oracle results in a phase flip of state(s) $|w\rangle$, essentially switching its signal

$$|\psi\rangle = \frac{1}{\sqrt{8}}(|000\rangle + |001\rangle + |010\rangle + |011\rangle + |100\rangle + |101\rangle - |110\rangle + |111\rangle). \tag{4.5}$$

As expected, the oracle defined is highly dependable on the solution or $|w\rangle$ we are looking for. For this concrete example, Figure 4.9 shows its quantum circuit representation.

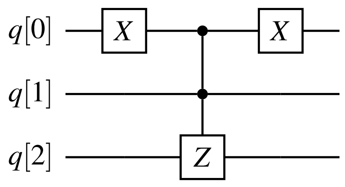


Figure 4.9: Phase oracle U_w for $|\psi\rangle$ where $|w\rangle = |110\rangle$

4.3.4.2 Amplifying the $|w\rangle$ states

The last step, the Diffuser's U_s application, refers back to the already defined superposition $|s\rangle$, with

$$U_s = 2|s\rangle\langle s| - I. \tag{4.6}$$

If we expand the equality, with N being the number of states originated from n qubits

$$N = 2^n, \tag{4.7}$$

we are able to see that it corresponds to doing an *inversion about the mean* of the amplitude of $|\psi\rangle$

$$U_s = \begin{pmatrix} \frac{2}{N}-1 & \frac{2}{N} & \dots & \frac{2}{N} \\ \frac{2}{N} & \dots & \dots & \dots \\ \dots & \dots & \dots & \frac{2}{N} \\ \frac{2}{N} & \dots & \frac{2}{N} & \frac{2}{N}-1 \end{pmatrix}, \tag{4.8}$$

since, for every state i of $|\psi\rangle$, it calculates its new amplitude $\alpha_i^{'}$ by doubling the mean μ, and subtracting its current amplitude α_i:

$$\begin{aligned} \alpha_i^{'} &= \frac{2\alpha_0}{N} + \frac{2\alpha_1}{N} + \dots + \alpha_i(\frac{2}{N} - 1) + \dots + \frac{2\alpha_{N-1}}{N} \\ &= \frac{2}{N}(\sum_{y=0}^{N-1} \alpha_y) - \alpha_i \\ &= 2\mu - \alpha_i. \end{aligned} \tag{4.9}$$

Note that positive phased states or *non-winner states* are above the mean as μ is affected negatively by winner-states, which are, at this point, negative. Figure 4.10 shows the circuit for a Diffuser U_s that can be used to complete our example.

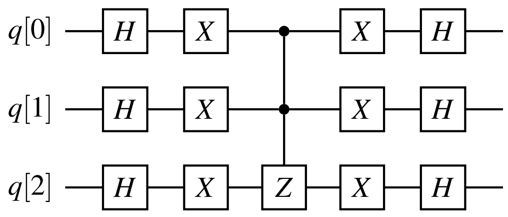


Figure 4.10: Diffuser U_s for $|\psi\rangle$

4.3.4.3 Grover iteration

The operators U_s and U_w above describe, together, form the *Grover iteration G*

$$G = U_s U_w, \tag{4.10}$$

so each time we apply G, we are initially reducing the mean, by inverting the phase of $|w\rangle$, which corresponds to the application of U_s. U_w subtracts from each state's amplitude its distance to the new found mean μ. Taking into account the phase of every state, positive phased states end up being suppressed, with their amplitude reduced, while negative phased states, or $|w\rangle$ states, increase their amplitude, keeping μ stable. So, by using G for a number of times t, we are growing the $|w\rangle$ state(s) linearly with the number of applications of approximately

$$tN^{-1/2}, \tag{4.11}$$

with N, in our example case, being eight (8). From this equation, we take that the number t of Grover iterations increase the differentiation between winner and non-winner states, while the dimension of the space of possible result reduces the differentiation seen per iteration. This is logical as impact of inversion around the mean is diluted the bigger the results' space. Its application can be described as

$$G^t |\psi\rangle . \tag{4.12}$$

For Equation 4.11, if we use $t = \sqrt{N}$ number of Grover iterations, it is highly expected for us to measure the correct result. With a M number of winner states $|w\rangle$, it is possible to generalize this equation[42], so

$$t = \sqrt{N/M}. \tag{4.13}$$

In our tests, we ran the algorithm using a $t = \frac{\pi}{4}\sqrt{N}$ number of Grover iterations [32].

4.3.4.4 Building the complete circuit

For our previous example, the complete circuit for a single Grover iteration is shown in Figure 4.11.

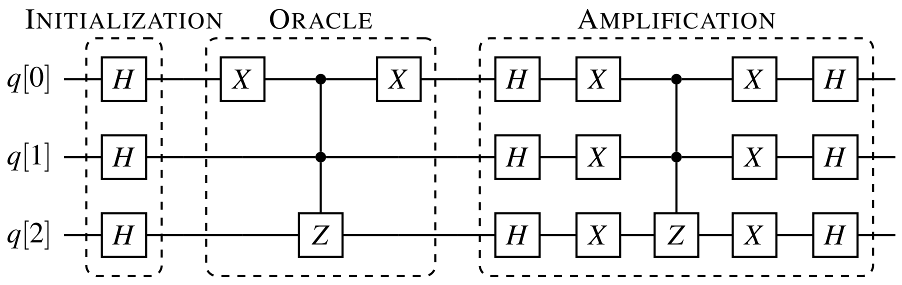


Figure 4.11: Complete circuit to do a single Grover iteration for $|\psi\rangle$

If we take into account the number of Grover iterations required according to the previously established $t = \frac{\pi}{4}\sqrt{N}$[2], with

$$t = \lfloor \frac{\pi}{4}\sqrt{N} \rfloor = \lfloor \frac{\pi}{4}\sqrt{8} \rfloor = 2, \tag{4.14}$$

the final circuit to be translated into cQASM code would look as in Figure 4.12.

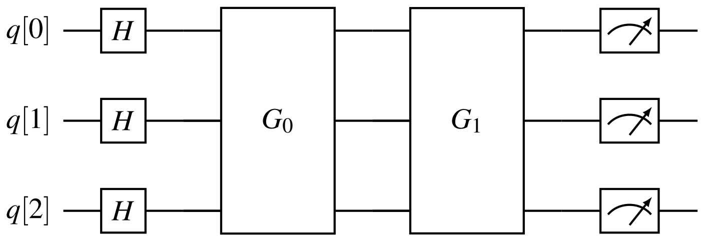


Figure 4.12: Complete circuit to find $|w\rangle$ for $|\psi\rangle$

4.3.4.5 Scaling the algorithm

In here, we explore how we can use the Grover's algorithm for different result spaces, i.e. how is it affected by increasing or decreasing the number of qubits involved. Since the *Initialization* step remains constant, independently of the number of qubits used, we should focus on how to scale the other two steps. The way we build a *phase oracle* for a single winner state $|w\rangle$ is actually fairly simple if we are able use a *Z-gate* controlled by every other non-target qubit in the circuit.

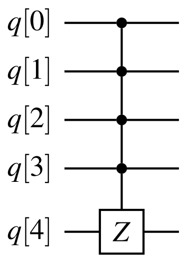


Figure 4.13: Phase oracle U_w where $|w\rangle = |11111\rangle$

[2]Non-integer values will always be rounded down.

Consider now Figure 4.13. By applying this single gate, we are tagging all qubits to the state $|1\rangle$, which means the state $|11111\rangle$ is the winner. If, for instance, we intend to tag a different solution like $|10101\rangle$, we only need to apply two *X-gate*, one before and another after the *multiple-control-Z-gate*, on each qubit that is not in state $|1\rangle$ on the winner $|w\rangle$. Here, we would apply those gates to qubits 1 and 3, like Figure 4.14 is showing.

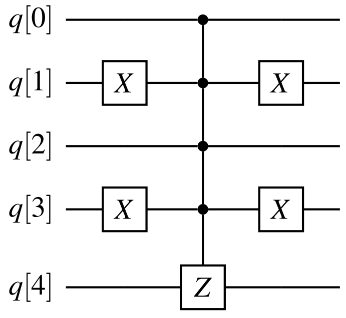


Figure 4.14: Phase oracle U_w where $|w\rangle = |10101\rangle$

Scaling the *Diffuser* is equally simple. For every new qubit added to the circuit, we need it to also act as a control qubit for the *multiple-control-Z-gate* of this step. The remaining gates are the same as for every qubit: the *H-gate* and *X-gate* prior to that same *multiple-control-Z-gate*, that should also be followed by the *X-gate* and *H-gate*, in this order. For the same 5-qubit circuit, the *Diffuser* is presented in Figure 4.15.

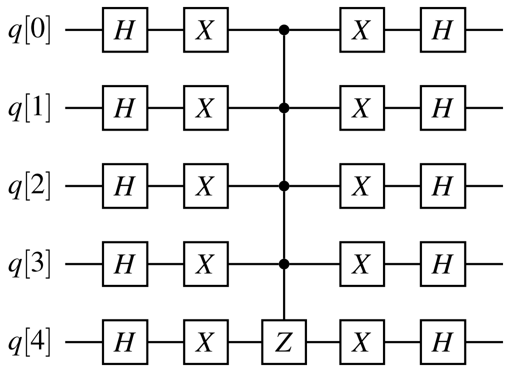


Figure 4.15: Diffuser U_s for a 5-qubit circuit

In total, on every new qubit added to the search, for $|w\rangle$ respecting

$$\forall i \in [0,n], q[i] = |1\rangle\,, \tag{4.15}$$

with n being the number of qubits in the circuit, we are adding one more gate to the circuit plus four gates on each existing Grover iterations. If $q[i] = |0\rangle$, the number of added gates rises to three plus the same four on each existing iteration. Note that we are excluding the two additional *control* for the two *multiple-control-Z-gates*, as they represent gate changes, not additions. If there are further Grover iterations due to the addition of the new qubit, the total number of added gates is calculated with

$$\begin{cases} 1+4*(t-diff(t))+(4n+1)*diff(t), & \text{if } q[i]=|1\rangle \\ 3+4*(t-diff(t))+(4n+1)*diff(t), & \text{otherwise} \end{cases} \tag{4.16}$$

with $diff(t)$ as the number of additional iteration. Again, know that Grover's algorithm is intended for memory testing, and we are focusing on knowing the impact of *useful quantum algorithms* on the *Quantum Micro-Architecture* by comparing it to *QBeeSim*. For that reason, the variation on gate quantity should not have a significant impact in overall the analysis.

4.4 Conclusion

This chapter started with an explanation on how we see the the top-level layers interacting with the lower-layers of the full-stack, i.e. what a programmer developing a *quantum algorithm* should expect from the *Quantum Micro-Architecture*. For that, we presented our vision on how cQASM integrates with classical languages, using `#pragma` statements, and how the results are accessible from classical languages by storing them in associative containers. Also, an important differentiation was made from what to expect from a simulator and from an actual quantum processor. From that, we concluded that the simulator allows us to have two types of results, one using state's amplitudes, which is appropriate for application development, and other using state's probabilities, emulating the response from quantum hardware. Next, we explained how the number of states evolves during a *quantum algorithm*, and related those same states to memory cost, identifying state's representation as the main memory consumer. This topic continued with an explanation on how we decided to optimize the storage required by the *Quantum Micro-Architecture*, given our focus on *useful quantum algorithms*, opting by only storing non-zero amplitude states. Finally, we described our test process for the *Quantum Micro-Architecture*, taking into account what kind of parameters are important for us, given the intended deployment setup. Specifically, we look for ways to measure the overhead caused by the *Quantum Micro-Architecture* on the simulator, and methods to assess correctness of instruction processing. That being the case, we differentiated between random circuits and *useful quantum algorithms*, even describing Grover's algorithm, since we plan to use it for memory analysis, while the former are preferred for a time-based analysis.

Restating, we analysed the *Quantum Micro-Architecture* from an outside perspective, looking at the components that surround it to see what are good practices that we can adopt, and what optimizations are clear from the start. This overview also served to clearly define what to expect as both input and output, laying all the fundamentals that allow us to understand the inner-workings of the *Quantum Micro-Architecture*. With this in mind, we can start its explanation, on the next chapter.

Chapter 5

Describing the Quantum Micro-Architecture

In Chapter 4, we described the *Quantum Micro-Architecture* essentially as a black-box: both the input and output are known, and the test process is also defined. Some insights were also given when it comes to memory conservation, which will be further explained here. Section 5.1 introduces our most recent diagram for the *Quantum Micro-Architecture*. Next, we go inside the black-box, describing each component in the order cQASM code interacts with the structure, again, focusing on its connection with *QBeeSim* for *quantum algorithm* development. Note that we are describing a *Generic Quantum Micro-Architecture*. This means that, in the future, for *quantum accelerators* that target a specific problem, this design might suffer slight variations to better adapt it to the problem to be solved. In that sense, we try to define it in a way that makes such adaptations as minimal as possible, for the great majority of applications. Particularly, we see that happening for applications like finance, and genetic analysis, as those are the areas QBeeX is most proactively looking, at the moment. A sample circuit expressed in cQASM is also used to better visualize how each component works, as it accompanies their own description.

5.1 Overview

The developments on thesis are focused on a software-based implementation of the *Quantum Micro-Architecture*. It is the same as saying that the entire development is described in classical logic, particularly in C++. In the future, we will develop an experimental *quantum accelerator* but, the driving logic will remains classical, nonetheless. This is already implying that any *quantum accelerator* will use digital hardware up to the quantum chip, which delves in the analog world. Since the quantum chip is only reactive to whatever instruction we send, its control remains digital.

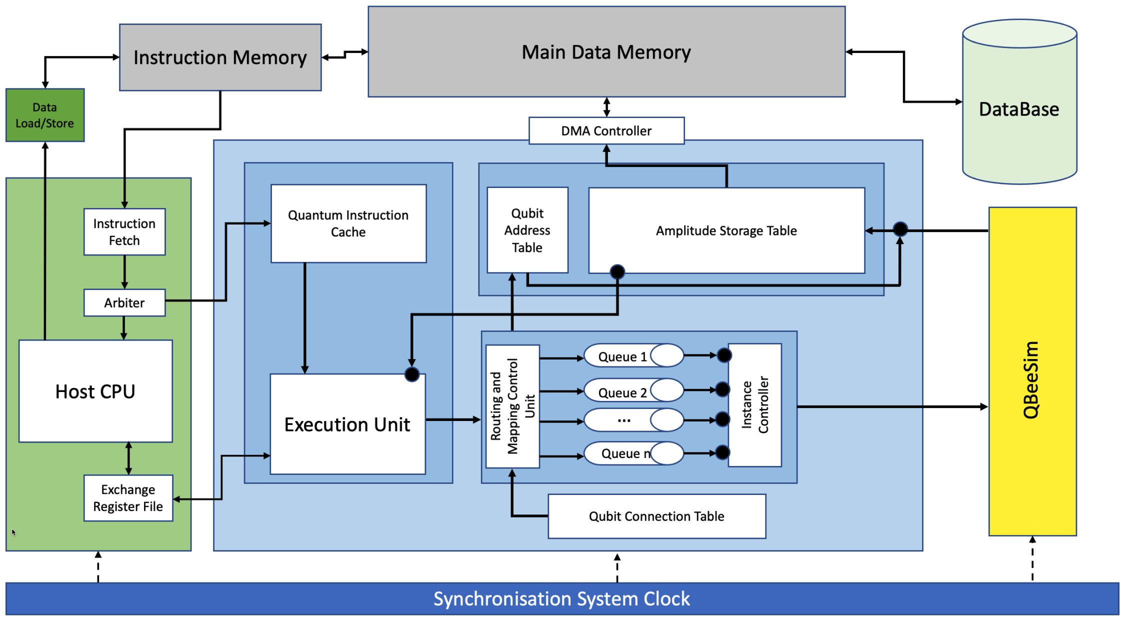


Figure 5.1: Generic *Quantum Micro-Architecture* for *quantum algorithm* development

Also, instead of a *physical qubits*, we deal with *perfect qubits*, requiring us to abandon the idea using a quantum chip, at the moment. Still, by using *QBeeSim*, we can emulate its behavior, and develop the remaining classical components around it, dissipating the limitation of not having any quantum hardware ready. Figure 5.1 shows, in blue, our representation of the *Quantum Micro-Architecture* as explained, while attached to our simulator *QBeeSim*.

Any optimized application, makes use of the heterogeneity of modern computer systems. This optimization requires the routing of dedicated instructions towards particular co-processors, to be executed. It is also implicit that such application can be written in several programming languages that communicate using the machine's main memory, showing the versatility of modern processors. Behind these processors, there is a specific architecture able to handle any sequence of instructions, if expressed in an understandable way. Similarly, the QPU recognizes both classical and quantum instructions, having two distinct processors. For classical instructions, i.e. for the accelerator's logic, we should use a traditional processor, while quantum operations should be redirected to the quantum chip. In essence, these architectures are a bridge connecting applications to the hardware that runs them. For the understandable language we speak off, we have the *Quantum Instruction Set Architecture* or QISA, that should be highly adaptable, according to the accelerator's logic needs. Still, for the actual development of such a structure, a number of properties can only be guessed, like the pipeline depth, instruction-length, or the effect of each cQASM instruction on platform control channels. Our view on this estimations will be given as we describe each block, since they are the base for them.

This section was based on [11].

5.2 Components analysis

5.2.1 cQASM and Quantum Instruction Cache

The *Instruction Memory* holds both classical and quantum instructions, fetched and passed on to the *Arbiter*, to be separated and redirected to the corresponding processor. While classical instructions should be redirected to the *Host CPU*, quantum instructions are passed to the *Quantum Micro-Architecture*, arriving at the *Quantum Instruction Cache*. The *Quantum Instruction Cache* serves as an on-site storage for these instructions, meaning that at this point, there is no processing of its individual information. Also, take into consideration that there is a size limit for the number of instructions read, meaning that for a multi-hundred non-parallel instructions circuit, only a small portion will be taken at a time, to be processed. After its processing, another group or *batch* of instructions is pulled, and so on until the circuit is terminated.

```
qubits 4

x q[0]
{y q[1] | z q[2]}
cnot q[1], q[3]
h q[0]
cnot q[0], q[2]
z q[0]
{y q[1] | h q[2] | x q[3]}
x q[3]
x q[1]
display
```

Listing 5.1: Example circuit to demonstrate how the *Quantum Micro-Architecture*works

Given that this is the only component that does not make any processing beyond identifying each line as a valid instruction, the original cQASM code is extremely similar to the way it is stored here, except that comments are already removed. For the explanation of each remaining component, the random circuit described in Listing 5.1 will be used. In that way, it is possible to visualize how the information is processed at each individual stage.

5.2.2 Execution Unit

On arrival at the *Execution Unit*, the instructions are interpreted, resulting on the identification of the key points of each instruction, like its *type*, *group*, *qubits* it affects, and the *arguments* it has, like rotation angles, if any. For example, consider Listing 5.1 line 5. That instruction, at this stage, is represented as follows:

cQASM	**Type**	**Group**	**Qubits**	**Arguments**
cnot q[1], q[3]	CNOT	NonPauli	[1,3]	[]

Table 5.1: Instruction parsing example

At the same time we do the parsing of individual instructions, it is also possible to identify parallelism. We expect *OpenQL* to already place parallel instructions together, as is shown in line 4, for example. Nonetheless, we can still verify if sequential instructions are not targeting the same qubits. If we go back to how we represent quantum circuits in a diagram, we are simply saying that parallel instructions are placed at the same depth level. It is clear that gates affecting the same qubits must be placed one after the other, so at different depths. For Listing 5.1 lines 4 and 5, we then have the following representation:

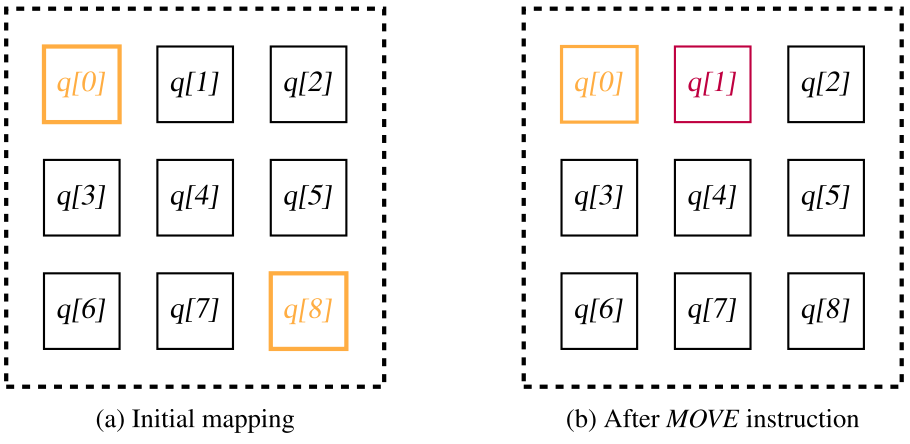
cQASM	**Type**	**Group**	**Qubits**	**Arguments**	**Depth**
y q[1]	Y	Pauli	[1]	[]	2
z q[2]	Z	Pauli	[2]	[]	2
cnot q[1], q[3]	CNOT	NonPauli	[1,3]	[]	3

Table 5.2: Instruction parsing example with parallelism identification

with the first two operations to be executed at the same time, considering how they don't interfere with each other. This is signaled by the same *Depth* value. The third instruction (CNOT) must take place after the first two, since both the first and third instructions require *q[1]*.

5.2.3 Qubit Address Table and the Routing and Mapping Unit

The *Qubit Address Table* is a requirement brought by the physical constraints imposed on real qubits. Since we want to emulate those constraints, in order to make the simulation as close as



(a) Initial mapping (b) After *MOVE* instruction

Figure 5.2: Example of simulator's qubit mapping

possible to reality, we need a way to associate the name given to qubits on *cQASM* to their position on the simulator.

Consider again Listing 5.1 line 5. The *CNOT-gate* in question tells us that qubits *q[1]* and *q[3]* must interact. If we impose *NN-constraints*, the mapping of qubits on the simulator becomes substantially more relevant. Figure 5.2 shows us two of the possible cases for the mapping of these two qubits. If we say that qubit interaction must only happen between adjacent qubits, it is clear that situation on the left, i.e.

cQASM index	**QBeeSim index**
q[1]	q[0]
q[3]	q[8]

Table 5.3: Partial *Qubit Address Table* for Figure 5.2a

is impossible unless we introduce position manipulation instructions, like the `MOVE` or `SWAP`[1] instructions. This first instruction takes two (QBeeSim) qubit indexes, first the one to be moved, and second the destination index. So, if we introduce a `MOVE q[8], q[1]`, which is done by the *Routing and Mapping Control Unit*, we would then have the following association:

cQASM index	**QBeeSim index**
q[1]	q[0]
q[3]	q[1]

Table 5.4: Partial *Qubit Address Table* for Figure 5.2b

Consider also that such addition would be unnecessary if the initial mapping took into consideration these interactions, meaning that we could have, from the beginning, mapped cQASM *q[3]* to QBeeSim *q[1]*. Additionally, if we decided to ignore such constraints, the mapping could be simplified by having *QBeeSim* indexes corresponding to *cQASM* indexes. This option seems reasonable if we are intending to purely test the results of a quantum algorithm while dismissing the natural implications of running it on an experimental platform.

5.2.4 Queues and Instance Controller

To understand how we define *Instruction Queues* and the *Instance Controller* for our specific case, we need to look at the current state of development of *QBeeSim*. The most important characteristic of the simulator is how each quantum gate requires a manipulation to a vast amount of states' amplitudes, since it places all qubits in *superposition* from the beginning[2]. The negative consequence of this behavior is that we are limited to having a sequential execution of quantum

[1]Notice that a *SWAP* operation is essentially 3 *MOVE* operations: (1) storing the destination qubit in a vacant qubit, (2) moving the origin qubit to the destination qubit and (3) moving the qubit stored away in the previously vacant qubit to the origin qubit

[2]The explanations as to why this is done goes beyond the focus of this chapter, nonetheless, this was proven more memory efficient when compared to creating qubits superposition when needed, since this last method requires us to label the amplitudes with the states they refer to, while the former does not. See Chapter 3 for more information on *QBeeSim*

gates, otherwise, in case we decided to parallelise same-depth instructions, we would face race conditions. This does not mean that, for each gate, we are unable to parallelise their effect on the multiple states, as this is already implemented. To reinforce, QBeeSim:

- Places qubits in superposition, from the beginning until the end of the circuit;
- Quantum gates must be executed sequentially.

This conditions tells us that all instructions can be loaded into the same *Instruction Queue*, as there is no benefit in having multiple ones, like there would be for a real quantum chip. Also, the *Instance Controller* only functions are to make sure that the previous operation was concluded, so that it can pull the next instruction from the *Instruction Queue* and deliver it to *QBeeSim*.

When we apply a *Quantum Micro-Architecture* on top of a quantum chip, it is at this point that timing becomes deterministic[27]. As the simulator does not have the same time-wise constraints as a quantum chip, we don't have to set a specific timing for each operation as long as we keep relative timing, i.e. the order of instructions is preserved.

5.2.5 Amplitude Storage Table

Considering that we are using a simulator, we have constant access to qubit state's data, without interfering with their value, meaning we don't collapse the state by accessing it. This means that instead of just knowing to which state the circuit collapses to, we can retrieve amplitudes, allowing us to bypass the process of making multiple circuit runs, since we can build the correspondent *probability density function* or PDF, for short, by extrapolating the probabilities. So, in terms of algorithmic runs, the simulator is actually more efficient. The *Amplitude Storage Table* serves as an intermediate holder for those amplitudes, before we store them away. In this block, we don't keep a register on all states produced for the circuit. Remember that we are targeting useful *quantum algorithms*, so we expect the result space to be sparse. For those cases, contrarily to what happens to the simulator, it is more efficient to know both the state we are referring to and its amplitude, since in that way, we can discard what should be a great amount of irrelevant states, i.e. state's whose amplitude are zero, also known as *zero-states*. Table 5.5 shows what the *Amplitude Storage Table* looks like for our example circuit, in Listing 5.1. Given that amplitudes are complex numbers we store pairs of values for said column: on the left we have the real component, while on the right we have the imaginary part. Also, while the state in the table uses ket notation, we only store the correspondent integer value, so $|1110\rangle$, for example, is only recorded as 14.

State	**Amplitude**
$\|1010\rangle$	(0.5, 0.0)
$\|1011\rangle$	(0.5, 0.0)
$\|1110\rangle$	(0.5, 0.0)
$\|1111\rangle$	(-0.5, 0.0)

Table 5.5: *Amplitude Storage Table* corresponding to example circuit

5.2.6 DMA Controller and Exchange Register File

When we have access to the final results, we have two data paths:

1. Passing them along to the main memory, using a *DMA Controller*;
2. Sending them to the *Host CPU*, through the *Exchange Register File*.

The first option is optimal for terminal computations, meaning that the work of the *quantum accelerator* is done. The second option allows the results to be kept inside the *Quantum Micro-Architecture*, so we can do intermediary calculations without having to pull its results from main memory in order to continue. A similar scenario happens for results loading. While intermediary results can be directly loaded from the *Quantum Micro-Architecture*, again through the *Exchange Register File*, results that were stored away to main memory can also be loaded using the *DMA Controller*.

As we are working with a simulator, the process of loading results operates in a different way of how it might happen on a quantum chip. While we can reestablish the quantum states fully for the simulator, again, due to the capacity to access its data without causing any losses, it has another meaning for quantum chips. Quantum chips need multiple runs of the same algorithm, so we can build the circuit's PDF. If, for some reason, we are required to interrupt the mid run cycle, we may store the measurements from the already executed runs away, to latter load them, continuing the cycle without losses.

5.2.7 Qubit Connection Table

While explaining how the *Routing and Mapping Control Unit* works, we exemplified with a specific topology, shown in Figure 5.2. This last component answers the question *How do we define that topology?*, which is the same as asking how can we make the *Quantum Micro-Architecture* adaptable to simulate any kind of experimental platform/topology combination. In its essence, the *Qubit Connection Table* brings into the *Quantum Micro-Architecture* two important pieces of information:

- Which qubits interact with each-other;
- The cost of performing a `SWAP` operation between two qubits.

`SWAP` operations, like the name indicates, changes the information stored in the qubits. The impact produced can be seen using the *Qubit Address Table*, as shown in Table 5.6.

cQASM index	**QBeeSim index**
q[1]	q[0]→q[1]
q[3]	q[1]→q[0]

Table 5.6: Partial *Qubit Address Table* for a *SWAP* operation between *q[0]* and *q[1]*

For multi-qubit gates, it is important to respect *NN-constraints*, so it makes sense to know if the interaction of the qubits involved is instantly feasible, or if we are required to introduce `SWAP` instructions to bring qubits to an adequate position. This information, for a four qubits chip, for example, can can be seen in the two matrices of Figure 5.3.

	q[0]	*q[1]*	*q[2]*	*q[3]*
q[0]	▨	1	0	0
q[1]	1	▨	0	0
q[2]	0	0	▨	0
q[3]	0	0	0	▨

(a) Interactions information

	q[0]	*q[1]*	*q[2]*	*q[3]*
q[0]	▨	1	2	5
q[1]	1	▨	3	4
q[2]	2	3	▨	2
q[3]	5	4	2	▨

(b) *SWAP* Cost information

Figure 5.3: Information stored on the *Qubit Connection Table*

Figure 5.3a, for example, tells us that *q[0]* and *q[1]* are the only ones capable of performing two-qubit gates, while Figure 5.3b lets us know that swapping qubits *q[1]* and *q[3]* has a cost of 4. The costs don't have units, as they should be taken as a measure of comparison for us to know which `SWAP` operations are more or less demanding to the chip. In that way, our objective is to use *mapping and routing* cautiously, so we can reduce the total cost of the circuit, which means a reduction to both number of instructions executed and total circuit time for chips and simulators. The two matrices can be compiled into a single table, which is the normal form of the *Qubit Connection Table*:

	q[0]	**q[1]**	**q[2]**	**q[3]**
q[0]	-	(1,1)	(0,2)	(0,5)
q[1]	(1,1)	-	(0,3)	(0,4)
q[2]	(0,2)	(0,3)	-	(0,2)
q[3]	(0,5)	(0,4)	(0,2)	-

Table 5.7: Example *Qubit Connection Table* according to Figure 5.3

In this table we record both matrices' values in a pair, with *interactions* as the first element and `SWAP` *cost* as the second.

5.3 Conclusion

This chapter shows how the *Quantum Micro-Architecture*is used to process cQASM, and how it redirects its results:

1. Classical and quantum instructions are fetched together, from *Instruction Memory*;

2. The *Arbiter* separates classical from quantum instructions, sending the former to the *Host CPU*, and the latter to the *Quantum Micro-Architecture*;
3. Quantum instructions are cached in the *Quantum Instruction Cache*;
4. The *Execution Unit* pulls instructions, to parse them, and identifies instruction parallelism;
5. cQASM qubits are mapped into *QBeeSim* qubits in the *Routing and Mapping Control Unit*, and this information is stored in the *Qubit Address Table*;
6. The instructions are loaded into the *Instruction Queues*, from which they are pulled by the *Instance Controller*, so they can be passed on to *QBeeSim*, when it is available;
7. For every *batch* of processed instructions, some may need the addition of auxiliary instructions to route qubits; the *Routing and Mapping Control Unit* is responsible for the addition of said instructions, based on the information provided by the *Qubit Connection Table*;
8. At the end of the entire circuit processing (or if called explicitly), the relevant amplitudes are passed to the *Amplitude Storage Table*;
9. Before storing the results in the *Amplitude Storage Table*, we must access the *Qubit Address Table* to translate *QBeeSim* qubits to cQASM qubits;
10. Intermediary results may go from the *Amplitude Storage Table* to the *Exchange Register File*, so they can be reused later;
11. Final results use the *DMA Controller* to be stored directly on main memory;
12. It is also possible to restore main memory results to *QBeeSim*, using the *DMA Controller*.

Chapter 6

Functional Design

In the last chapter, we explained how cQASM instructions are processed in the *Quantum Micro-Architecture*. Along its development, we noticed some areas where we can improve performance, and/or reduce the total memory footprint, which is most relevant. In this chapter, we explain those improvements, from the simplest to the most complex, in terms of implementation:

1. We look at perfect qubits connection's, to detect unnecessary/duplicated information;
2. The barriers created for result acceptance are analysed, as they are a way to reduce the *Quantum Micro-Architecture* memory footprint;
3. A new cQASM command is introduced, enabling the reduction of the qubit states in analysis, without interfering with the solution.

6.1 Reducing the Connection Table size

Figure 6.1 is a copy of Figure 5.3 just to improve readability. From the figure, we take that both matrices are symmetric, since every element $a_{i,j}$, on each one, is equal to $a_{j,i}$. This is the same as saying that each matrix is equal to its transposed:

$$A = A^T \tag{6.1}$$

	q[0]	*q[1]*	*q[2]*	*q[3]*
q[0]	▨	1	0	0
q[1]	1	▨	0	0
q[2]	0	0	▨	0
q[3]	0	0	0	▨

(a) Interactions information

	q[0]	*q[1]*	*q[2]*	*q[3]*
q[0]	▨	1	2	5
q[1]	1	▨	3	4
q[2]	2	3	▨	2
q[3]	5	4	2	▨

(b) *SWAP* Cost information

Figure 6.1: Information stored on the *Qubit Connection Table*- copy of Figure 5.3

It is now obvious that we are storing the same amount of information while using double the amount of space required. To reduce said space, we can cut half the table and sort qubit indexes each time we want to retrieve any information. For example, accessing the information for *q[3]* and *q[0]* ($a_{3,0}$) is the same as accessing the information for qubits *q[0]* and *q[3]* ($a_{0,3}$), since the indices of the former are now sorted, becoming the latter. For the *interactions information*, the symmetry may seem obvious, but we are also assuming that the *SWAP cost* is similarly symmetric, which may not be the case. Note that this reflects a very minor improvement in the overall program cost, given that, for each matrix, instead of storing a total of $q^2 - q$ values, for a q number of qubits, we store

$$\frac{q^2}{2} - q \tag{6.2}$$

values, which is a very small amount, when compared to the number of states produced by the same number of qubits q. Additionally, this is a trade-off that may only make sense when we are trying to save memory, as for this purpose, we introduce the sorting operation, which may have an costly impact on a more time-sensitive platform, i.e. a quantum processor, due to the accumulation of small time increments.

6.2 Restricting the result space

We already explained how useful *quantum algorithms* tend towards a more sparse result space compared to random walks. For those cases, let us look at the two barriers we can implement to exclude a state, i.e. leave it inside *QBeeSim*.

6.2.1 Elimination of zero-states

The possibility to eliminate *zero-states* was already explored in Chapter 5. In essence, every time we want to store away *QBeeSim*'s results, we have to iterate through its vector of states, or copy the entire memory block. If we exclude the latest option, for each state, we can check if their amplitude is zero, meaning that both the real and imaginary components of the amplitude are zero valued, or at least close enough to zero, since small differences can be the result of classical computation limitations.

6.2.2 Cutting improbable states

To further reduce the results space, we can establish a minimum acceptable probability, making inferior amplitudes irrelevant. This option was mostly implemented thinking on the specific case of Grover's Algorithm. On each Grover iteration, we are inverting the amplitudes based on the mean amplitude at the beginning of the iteration. Through successive iterations, it is highly unlikely that we can reduce *non-winner states* to *zero-states*. Considering that all non-winners have equal probability, the result space would have the same size as the total number of states produced by the circuit. This makes the way we store results in the *Quantum Micro-Architecture* inappropriate, even for a useful algorithm like this one. Similar to how we verify the amplitude of the state, when iterating through *QBeeSim* results, to dismiss *zero-states*, we can make the same verification based on a pre-established minimum probability value. If this value is high enough, we know that, for Grover's, for instance, the result space is only as large as the number of winner-states.

Figure 6.2 shows how this barriers function, having *QBeeSim*'s storage on top, and passing through all barriers sequentially, until the remaining results are stored away in the *Amplitude Storage Table*.

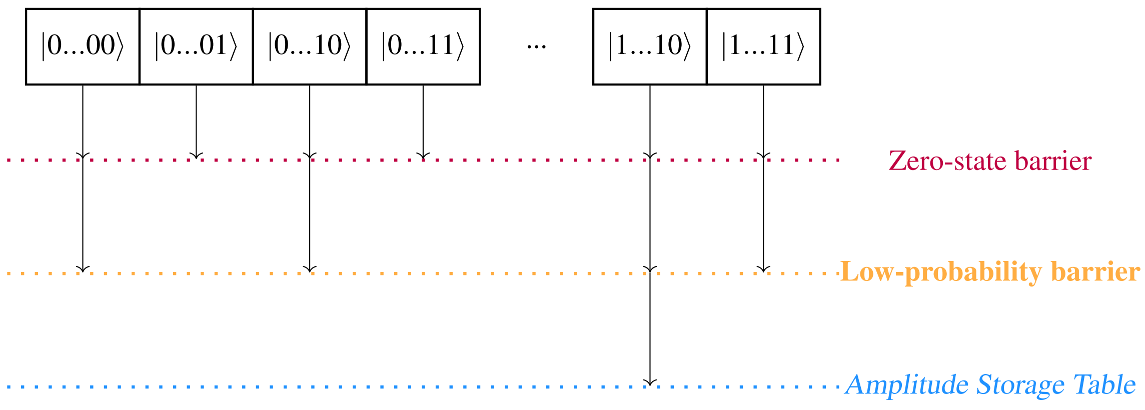


Figure 6.2: Results space restriction

In practice we know that a low-probability check eliminates a zero-state check, given that all zero-states are low-probability, although we cannot say the same for the opposite case, since not all low-probability states are zero-states.

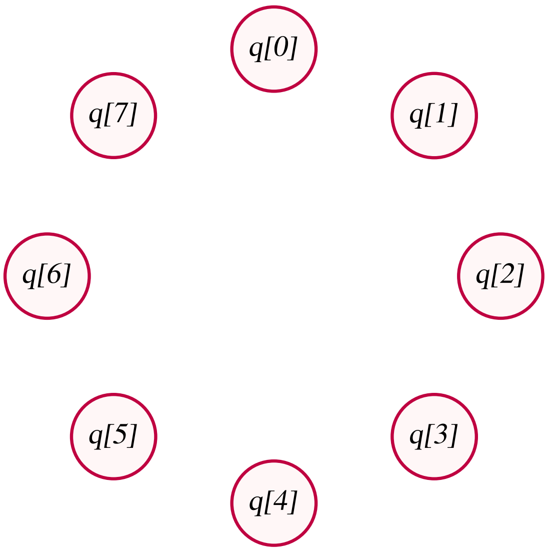


Figure 6.3: 8-qubits circuit nodes

6.3 Instance command

6.3.1 Defining an instance

Consider a graph built from a random quantum circuit with 8 qubits, where its qubits are the nodes, as shown in Figure 6.3. *QBeeSim* executes every circuit by placing all its qubits in superposition. This means that, for every gate, the number of states we have to access is exponentially correlated to the number of qubits on the circuit. Superposition is needed only for multi-qubit gates, which means that applying it from the beginning to non-interacting qubits is a waist of resources. Now, let us consider the extreme case for the 8-qubit circuit, by using the graph's edges to represent multi-qubit gates, as shown in Figure 6.4.

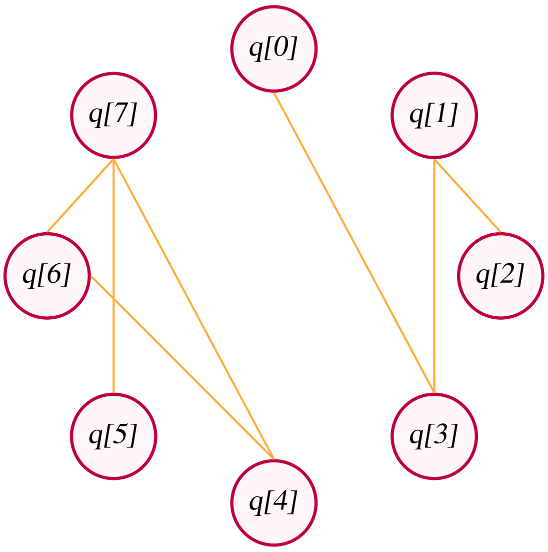


Figure 6.4: 8-qubits circuit qubit interactions graph

The separation of *q[0]* to *q[3]*, and *q[4]* to *q[7]* is clear. We call this groups of connected qubits *instances*, which are independent from each other. What this means is that these qubits (and the correspondent circuit gates) can be separated in two different circuits. By doing so, we can run the two circuits in parallel, reducing the total running time from the time it takes to run the initial circuit, to the time it takes to run the larger of the newly built circuits. This speedup can not be extrapolated directly from the sum of the time it takes to run the larger circuit's gates on the initial circuit. The gate time variation comes from the reduction of the total amount of states, as we are superpositioning an inferior number of qubits, which in turn reduces each gate's running time.

At this point, a question like *who would build a circuit that interacts in this way?* might arise. For one, in large circuits, this distinction is not very clear, unless we do this kind of graph. Also, this case can represent simply a starting point: say, we have these same eight qubits in a thousand gates circuit; maybe the first nine hundred gates can be represented using the graph in Figure 6.4, and only the last hundred gates required another kind of configuration, by connecting *q[1]* to *q[5]*, for example. This means that we can considerably speedup 90% of the circuit, while only 10% remaining sequential. *But why do the last 10% remain sequential?* Can we apply the same strategy used to group qubits, and also speed it up? To apply the same strategy, we would have to decompose the superposition in the amplitudes associated to the basis states of each qubit. So if we have two qubits:

$$q[0] = \gamma_0 |0\rangle + \gamma_1 |1\rangle , \tag{6.3}$$

$$q[1] = \upsilon_0 |0\rangle + \upsilon_1 |1\rangle , \tag{6.4}$$

and these two qubits where in superposition, they could be described as follow:

$$\Psi = \alpha_0 |00\rangle + \alpha_1 |01\rangle + \alpha_2 |10\rangle + \alpha_3 |11\rangle , \tag{6.5}$$

with the amplitude values α agreeing with the following association:

$$\begin{cases} \alpha_0 = \gamma_0 * \upsilon_0 \\ \alpha_1 = \gamma_1 * \upsilon_0 \\ \alpha_2 = \gamma_0 * \upsilon_1 \\ \alpha_3 = \gamma_1 * \upsilon_1 \end{cases} \tag{6.6}$$

We must not forget that all these variables represent complex numbers, which means that we are doubling the amount of variables in the system. Although we know the components of the α named variables, γ_0, γ_1, υ_0, and υ_1 make up a total of eight unknown real variables, which are impossible to determine in a system with four equations[1]. Since we are unable to decompose the

[1]Note that we may not be able to find the amplitudes but the probability associated to each basis state is achievable, which does not have much impact for our intended solution

superposition, we can only join both superpositions, by applying the tensor product. This option leaves us with the same number of states as making the superposition with all circuit's qubits, meaning that the last 10% of the circuit are, indeed, unchanged.

6.3.2 Instance command implementation

We see the `instance` command as an informative command that tell us which qubits belong in that same *instance*. For a circuit corresponding to the graph in Figure 6.4, we would required the following instructions:

```
instance {0,1,2,3}
instance {4,5,6,7}
```

By listing all qubits that belong to each *instance*, we add versatility to the way they are named, since this does not require qubits to be named sequentially, as it would if we used ranges.

As instructions are read in *batches*, it would be impossible to assess the constitution of said *instances* correctly in the *Quantum Micro-Architecture* alone. We propose a change to OpenQL's compiler, since it already as to go through all circuit's instructions, when building the cQASM code. That way we guarantee the correctness of the instance instruction, and allows it to change according to circuit's needs. In turn, this results in the possibility to introduce a new *instance* command in any part of the circuit, if we want to change which qubits are interacting. The definition of new instances is, of course, ruled by the limitations presented above, meaning that changing an instance actually means joining it with another one.

6.3.3 Impact on the micro-architecture's design

The *instance* command translates directly into how we group qubits in *QBeeSim*. For that reason, we can say that *QBeeSim* itself also has *instances*. The first observable change to the *Quantum Micro-Architecture* components comes to the *Qubit Address Table*, where we have not only a new qubit address, but also an *instance index*, indicating in which simulator's instance are we placing said qubit. The *QBeeSim index* now indicates the qubit index inside the *QBeeSim*'s instance. Table 6.1 shows how we make this association based on the graph shown in Figure 6.4.

cQASM index	Instance index	QBeeSim index
q[0]	0	q[0]
q[7]	1	q[3]

Table 6.1: Partial *Qubit Address Table* based on the qubit graph of Figure 6.4

In the end of the circuit, we have multiple groups of independent results, one for each *instance*. Similar to how we would do in any other superposition, we apply the tensor product to build the results in the expected format. One more step is required so we have accurate results, and that is rebuilding the states to which the amplitudes correspond to. This process is done by looking at the

Qubit Address Table, and doing the decoding of the states involved in the specific superposition in question. For example, based on the partial table in Table 6.1, if *q[0]* on instance 0 is active, *q[0]* is also active on the original state. Similarly, if *q[3]* on instance 1 is active, so is *q[7]* on the original state, and so on.

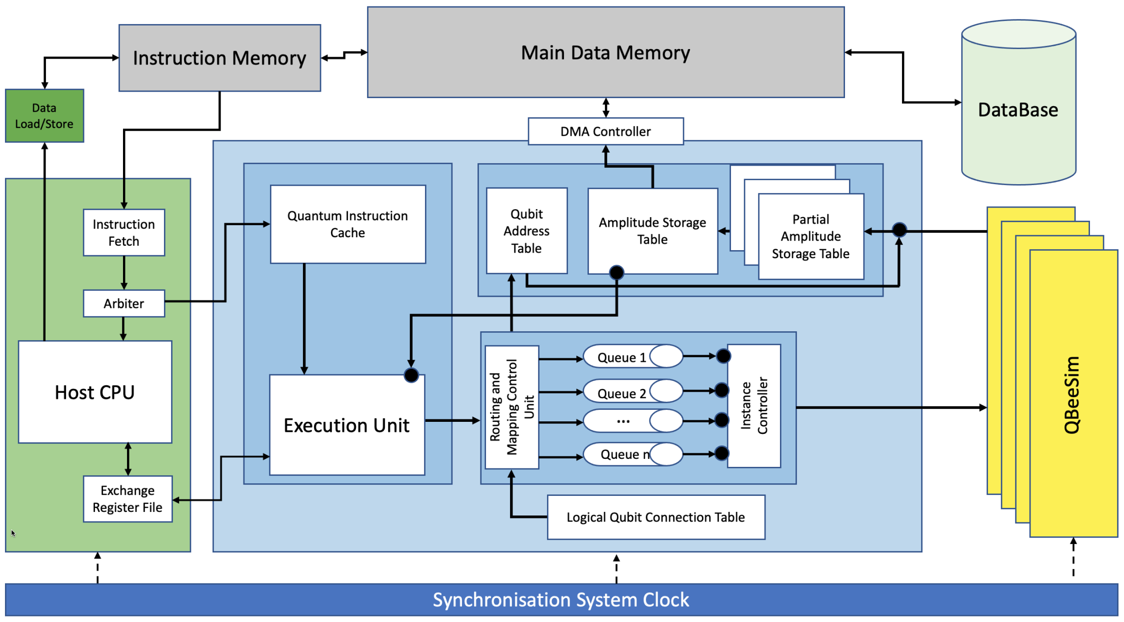


Figure 6.5: Generic *Quantum Micro-Architecture* for *quantum algorithm* development - *instance* based

Figure 6.5 shows how this changes affect the design of the *Quantum Micro-Architecture*. Essentially, instead of directly placing the results in the *Amplitude Storage Table*, each instance places its results in one of the *Partial Amplitude Storage Tables* (based on the same index value). The phrasing "placing the results on *Partial Amplitude Storage Tables*" ends up being false, in the strict sense of the word. The *Partial Amplitude Storage Tables* serve only as a bridge to *QBeeSim*, so we don't actually copy the results from each *QBeeSim*'s instance, we simply access them, saving memory. Those results are then reshaped, based on the *Qubit Address Table*, as explained before, and only then are they stored away in the *Amplitude Storage Table*.

6.4 Conclusion

In this chapter, we explore three ways on how to make the *Quantum Micro-Architecture* more efficient. We begin on the less relevant up to the most impactful. First, we showed that if *SWAP* operations are symmetric quantum chip operations, meaning that a `SWAP q[a], q[b]` is the same as a `SWAP q[b], q[a]`, with $a, b \in \mathbb{N}_0$, we can half the size of the *Qubit Connection Table*, by sorting the qubits involved. Second, we placed two barriers with the purpose of validating the results obtained. The looser one blocks the storage of states with amplitude zero - *zero-states*. The other and stricter one, has defined a minimum acceptable probability for the result to be considered valid. Third, we presented a way to optimize how quantum circuits are simulated, by dividing unrelated qubits in different *instances*, which is expected to save both memory and time. Both savings come from the reduced number of superpositions to be handled, that result from this new representation. In the next chapter, we look at the actual impact this changes produce.

Chapter 7

Experimental Results and Limits Analysis

Chapters 5 and 6 serve as an explanation on the inner working of the *Quantum Micro-Architecture* with its current improvements, respectively. In the former, we went component by component, following the code's path with a small circuit accompanying their respective explanation. The latter introduced three improvements to the architecture's implementation defined in the former, and it is with those in mind that we realize all tests that follow. Here, we look at the impact of said improvements according to its use context and testing methodology, as explained in Chapter 4. We start by analysing the impact caused by the micro-architecture in the existing stack layers. Next, we look at what amount of instructions should be processed at a time, i.e. *batch* sizing. The remaining experiments look at existing structures in the micro-architecture, to find weak links, and direct our research for future improvements:

1. Analysis of the impact of database storage on the system;

2. Validation of its orientations towards useful *quantum algorithms*;

3. Limits testing and extrapolations of what circuits we are able to simulate.

7.1 Benchmark system setup

The stack layers in use during this research were placed in an AWS EC2 instance, specifically a *c5a.4xlarge* instance, which provides eight (8) cores and sixteen (16) vCPUs, and thirty-two (32) GiB of memory, following the orientations in Chapter 4. The purpose of the setup is to allow the following experiments to be made, both locally, and with external database storage, running independently on a separate RDS instance:

1. Grover's algorithm (as a useful *quantum algorithm*), with increasing number of qubits (and gates, as a consequence);

2. Random algorithms, with increasing number of qubits for a set amount of gates;

3. Random algorithms, with fixed number of qubits and increasing number of gates.

Examples of circuits used for testing are provided in Appendices A and B, corresponding to Grover's algorithm and a random five-hundred (500) gates algorithm, respectively, both with five (5) qubits.

Note that random algorithms represent the worst-case scenario of any algorithm, and it is in that way that we use them for our tests. Also, take into account Grover's dependency between the number of qubits and gates, which makes it impossible to make an analysis following the same structure that will be employed during the following sections[1].

7.1.1 Results retrieving steps

There were, in total, three steps for recording peak memory usage during an algorithm's run. The first one, what we here call as *micro-architecture only memory usage*, takes place before the initialization of *QBeeSim*, which means this only considers the first stage of usage of the *Quantum Micro-Architecture*, i.e. does not account for final results storage. The second, or *simulator's memory usage*, happens at the end of every batch processing. The third and final, or *total memory usage*, takes place just before the stack process is closed.

7.2 Results analysis

The next subsections contain all insights taken from the data collected, both by the information stored away in the database, and from process tracking. Its conclusions build upon each other, being presented in a way where previous subsections serve as the basis for conclusions taken ahead.

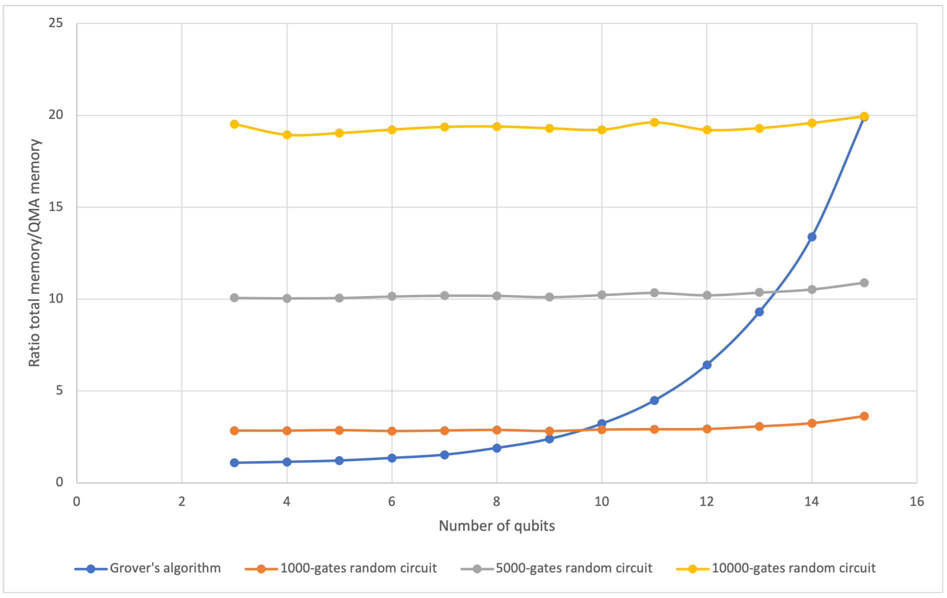


Figure 7.1: Ratio of total memory usage to micro-architecture only memory usage

7.2.1 Impact on the stack process

Figure 7.1 compares the first and final steps, corresponding to the micro-architecture only with total memory usage, respectively:

$$y = \frac{\text{Total Memory}}{\text{QMA Memory}} \tag{7.1}$$

To do so, we use three fixed gates numbers, representing the random algorithms, being a thousand (1.000), five-thousand (5.000), and ten-thousand (10.000) gates, plus Grover's algorithm, where the number of gates depend on the number of qubits, as previously established in Chapter 4. These four groups are varying the number of qubits.

The graph tells us that the ratio's link is much stronger to the number of gates than to the number of qubits. With this results alone, it is very hard to understand why the total memory usage is growing with the number of gates (since the *micro-architecture only memory usage* is almost constant for each of the random algorithms groups). The tests that follow should clarify this question by relating them to database usage.

7.2.2 Runtime impact of instruction's cache size on in database runs

In Chapter 5, the concept of instruction's *batch* was introduced. A *batch* is a group of quantum instructions pulled into the micro-architecture's cache to be processed, which is predefined in its size. Increasing the batch size means an overall increase in program memory consumption for instructions only. As expected, at this cost, we end up with smaller running times (in most of

[1]See Chapter 4 for a detailed explanation on Grover's

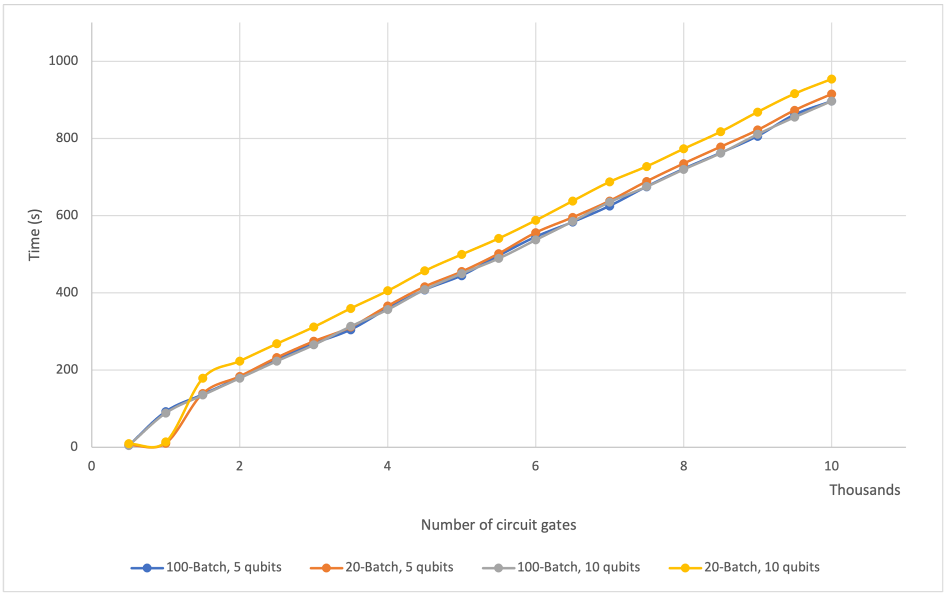


Figure 7.2: Batch size comparison for 100 and 20 instructions batches

the cases), as shown in Figure 7.2. These results may come from the reduction of the amount of times we need to open the cQASM input file, although this seems highly unlikely. Note that discrepancy between the two possible batch sizes is more noticeable the more qubits we are taking into consideration. Nonetheless, note that the discrepancy is kept almost constant, as we increase the number of gates, which is expected due to the constant rate kept between the size of the batches being compared. At last, we should note how smaller batches actually result in faster runtimes when the circuits are small (< 1.000 gates). As before, the information here analysed should relate to database usage, and will be further explained in the next sections, as for now it still remains inconclusive.

7.2.3 In database & local run comparison

7.2.3.1 Process runtime

Looking at Figures 7.3 and 7.4, we notice two great contrasts:

1. Local and in database runtimes are widely separated on its runtimes;
2. Runtimes, either local or in database, are highly affected by the addition of extra qubits, as in both cases we can see linear growth.

First, we now know that the communication with an external database is very costly, in terms of process runtime. For this reason, we have to weigh the advantages of such connection, i.e. storage of every detail that we can collect during circuit runs, like intermediary states for example, versus solutions collection speed. Second, having more qubits brings more states, which, consequently, affects gate performance since *QBeeSim* has to go through every existing state to either dismiss it,

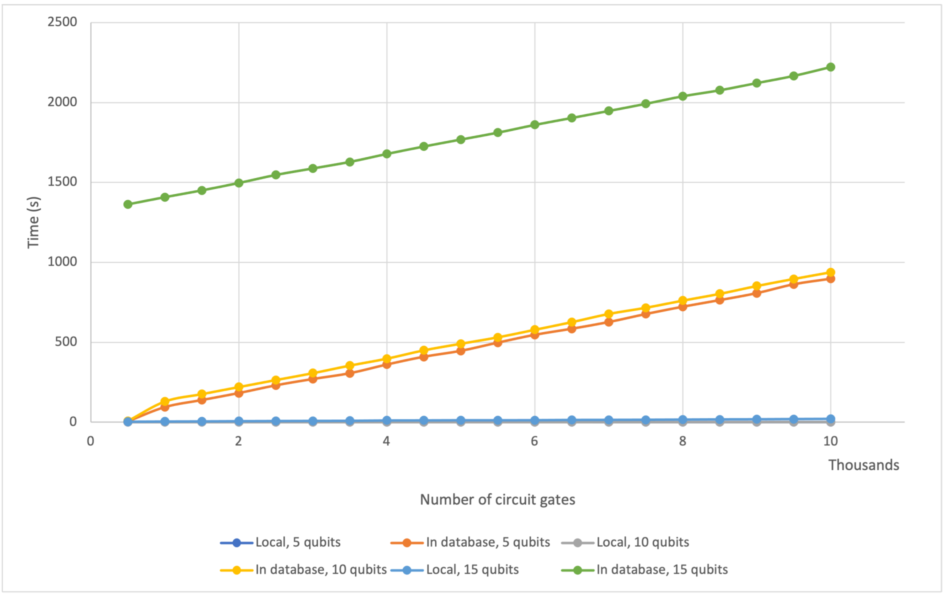


Figure 7.3: Local and in database process runtime by circuit size

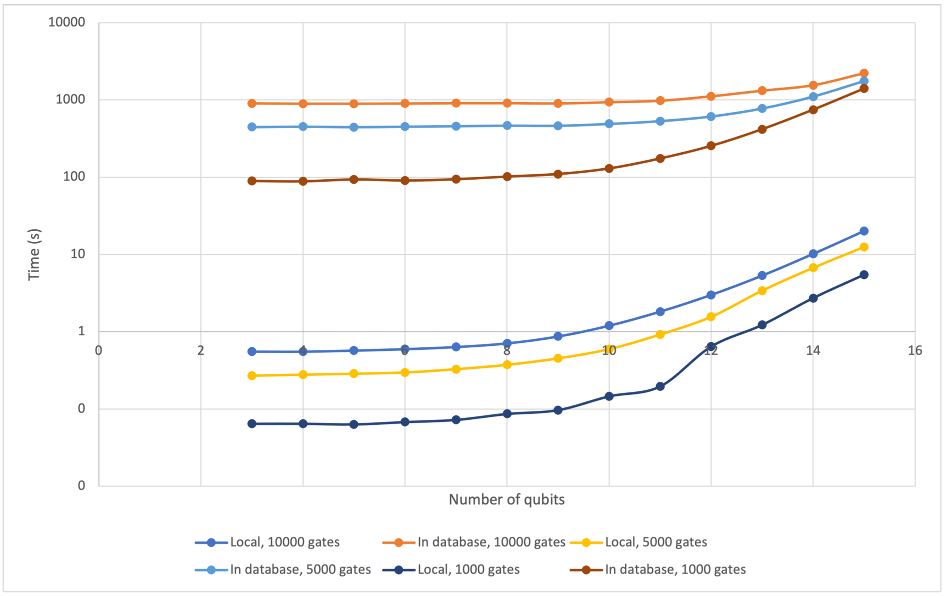


Figure 7.4: Local and in database process runtime by circuit size (logarithmic scale)

or apply the respective operation. In this way, the average gate time is highly linked to the number of qubits in the circuit. Considering that, with each new qubit, the number of states doubles - $N_{q+1} = 2 * N_q$, with N being the number of states resulting from q qubits -, it comes as no surprise that process runtime is also affected exponentially.

Additionally, Figure 7.5 shows us that, either in local or in database runs, the effect of adding qubits follow the same patterns. Despite the growth rate being more accentuated for in database runs, qubit addition affect process runtimes exponentially, which is seen by comparing different number of qubits while keeping the same number of gates.

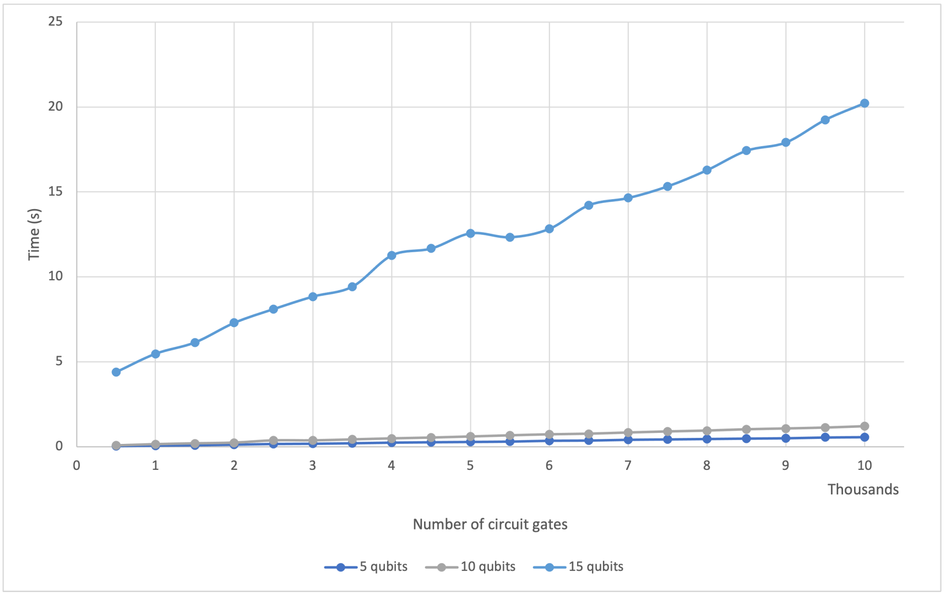


Figure 7.5: Local process runtime by circuit size

7.2.3.2 Average gate time

Figure 7.6 shows an analysis of the average gate time for fifteen (15) qubits circuits, with varying number of gates.

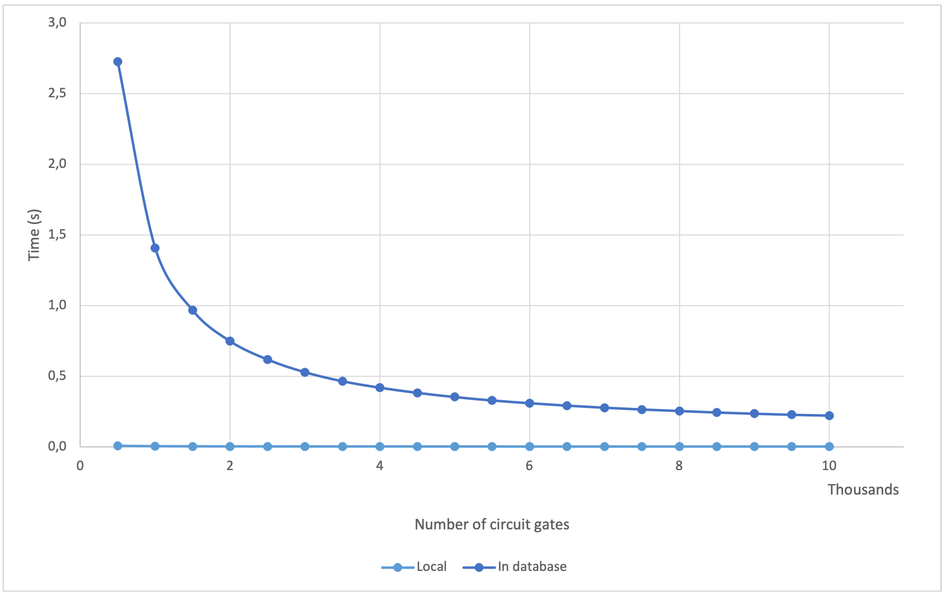


Figure 7.6: Average gate time for 15 qubits circuits

For the local runs, the results are as expected: we see the average gate time being reduced as we grow the number of gates, meaning that the non-circuit processing operations, or operations unrelated to *QBeeSim*'s simulations, become less relevant in terms of process runtime, until the average gate time stabilizes on seven-thousand five-hundred (7.500) gates. Although graphically, with in database results, this is not visually clear for the local runs, due to the scale displayed. Multiple database runs show us approximately the same effect (shape of the graphic), but on a much bigger scale. This means that for each gate, there are still database operations associated, and those can not be eliminated, resulting in average gate times that are at least two orders of magnitude greater than on local runs (+100 times longer).

To sum it up, for both types of runs, after a certain number of gates threshold, the average gate time stabilizes, so increasing the number of gates should produce runtimes that fit linear estimations. Only by increasing the number of qubits, considering that each gate will take more time to go through all the superpositioned states, we should see this stabilization threshold happening faster.

Note that the analysis of average gate times are highly connected to the constitution of the circuit being analysed. We did not take into account maintaining gate proportion constant, i.e. the ratio between different gates in the circuits were not taken into consideration, so, as different gates have different runtimes, variations from a exact estimations are expected. What we mean by this is that, for example, if we want to calculate the time it would take to run a twenty-thousand (20.000) gates circuit locally, we should expect our error, compared to to double the time it takes to run a ten-thousand (10.000) gates circuit also locally, to increase the more different the gate ratios are. Independently of this factors, a correlation between local and in database runs is possible, as the circuits ran are the same.

7.2.3.3 Qubit count based analysis

As mentioned already in this chapter, but primarily in Chapter 3, as we grow the number of qubits, so do the number of possible states we need to represent. Figure 7.7 shows how the exponential increase of states, by a factor of two (2) on each new qubit, results in an equally exponentially increase in the time it takes to run a circuit. For the latter, the growth factor is also increasing with the number of qubits, but has the tendency to stabilize equally at two (2). Again, we see a distinction between local and in database runs, although it is clear that, in both cases, the growth is exponential, only with the in database runs presenting a much quicker growth. Note that we are also showing three different circuit sizes, for both local and in database runs and from this we conclude that time increases linearly with the number of gates.

An increase in the number of states obviously results in a need for more memory, since all states are kept in memory through the circuit's run. Figure 7.8 similarly keeps the same parameters as the previous one, but, this time, showing what that parameters combination represents in terms of memory usage. Given the low amount of qubits, we see a very small growth in memory, which is obfuscated by the impact that increasing the number of gates has on in database runs.

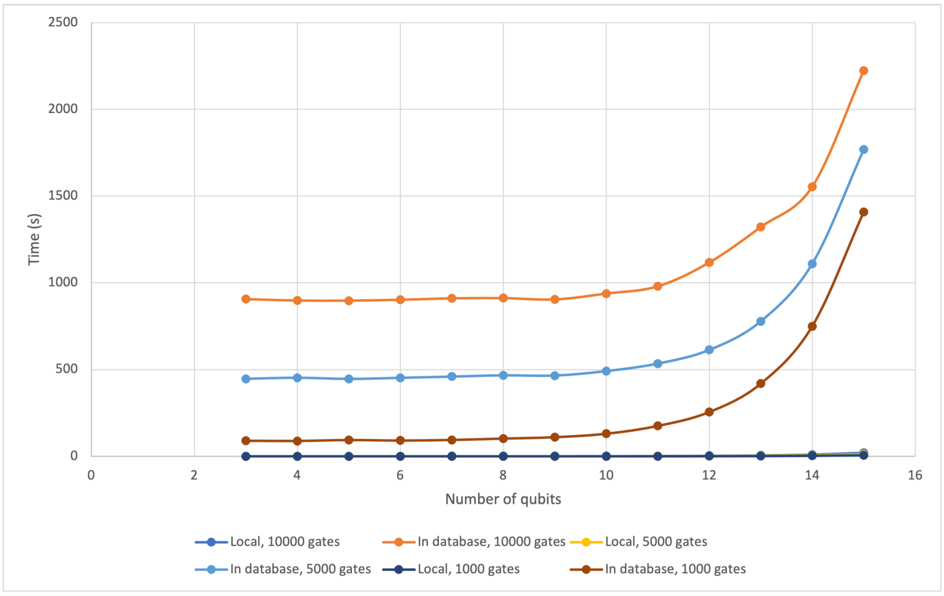


Figure 7.7: Local and in database process runtime by number of qubits

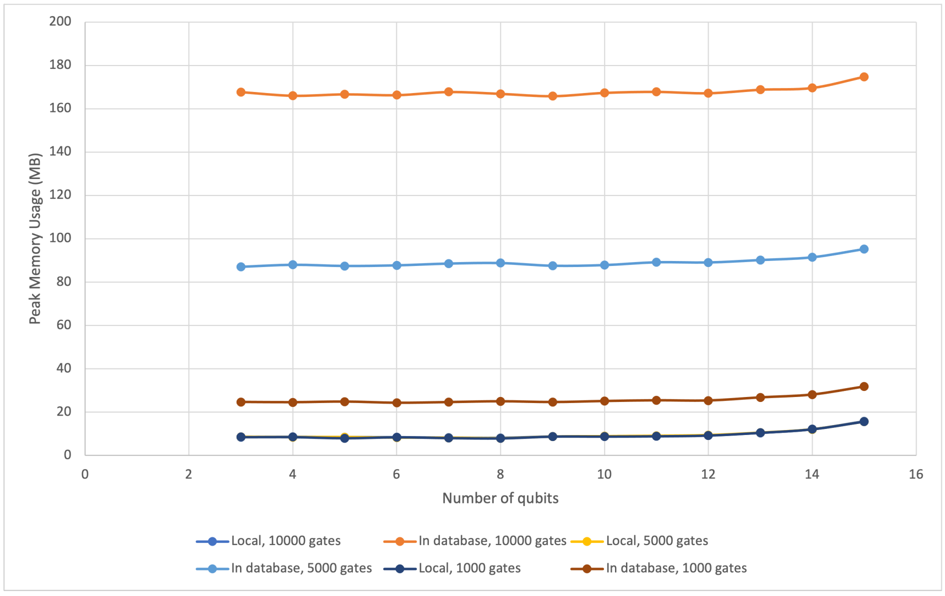


Figure 7.8: Local and in database process peak memory (MB) by number of qubits

Note that Figure 7.9 shows the same information, but only for local runs. The exponential increase in memory usage brought by the increase in the number of qubits, as calculated by Equation 3.1, is more obvious here, while there is no significant impact that comes from increasing the number of gates. This proves how the impact on memory usage comes solely from the database connection layer, which caches each *step*'s operation[2] and only at its end does it release them. On

[2]With *steps* being the naming given on the database connection layer to group multiple instructions

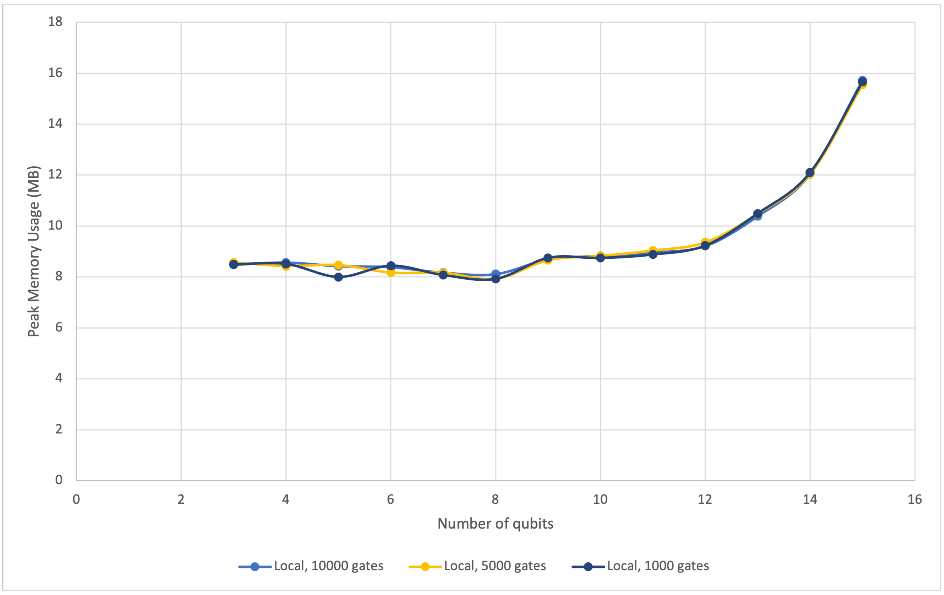


Figure 7.9: Local process peak memory (MB) by number of qubits

the other hand, the *Quantum Micro-Architecture* keeps its expected scalable behaviour of minimalist as possible footprint, only majorly affected by the number of produced results.

7.2.4 Runtime and memory impact of connecting to a database

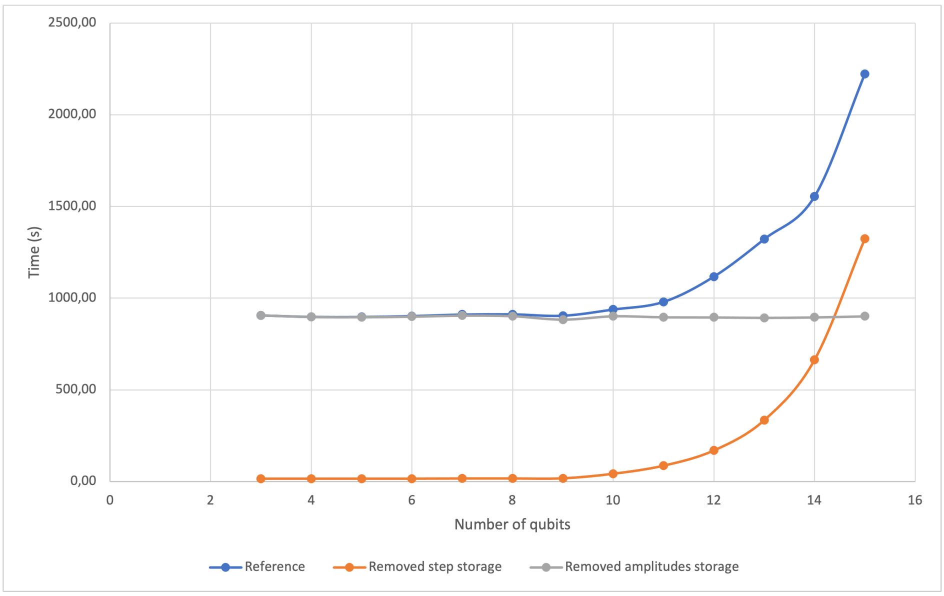


Figure 7.10: Database connection layer individual components removal

Figure 7.10 purpose is to show the effects of removing specific actions from the database connection, and see how it affects the overall process runtime. Prior to making this chart, we already identified in Subsection 7.2.3 the two major causes of performance deterioration:

- Need to store intermediary state's amplitudes;
- Need to store groups of instructions (*step* instructions).

As expected, when removing *step* storage, we only get the increased delay brought by saving state's amplitudes, which has an exponential shape, that was also expected due to the way the number of states grows when qubits are increased. Doing the opposite, i.e. removing amplitude storage, we get a nearly constant additional delay. Such delays also falls within expectations, considering how batch sizes are fixed at the beginning of the circuit's run, making the number of *step* instructions that need to be loaded into the database also constant. We also plot a reference curve, with all database connection steps included, i.e. as it is intended to run, being essentially the sum of the two previously described curves.

Considering that we are only loading instructions into the database at the end of the *step* (as we do for amplitudes), for large amounts of qubits, bigger batches make more sense, as we store those same amplitudes less frequently (assuming the time it takes to store the amplitudes alone is bigger than the time to store the instructions, which happens at fourteen qubits onward). For smaller amounts of qubits, where the amplitude storage time is inconsequential, we should opt for smaller batches to get more efficient runs - smaller batches reduce the time it takes to store instructions, but the time for amplitude storage is increased. That being said, choosing a batch size is highly dependent on the number of qubits in use. Only in that way can we reach optimal runtime conditions, balancing memory usage and runtime constraints to fit our needs.

7.2.5 Random & useful algorithms comparison

In Chapter 4, we concluded that the expensive memory cost brought by the micro-architecture comes from the need to retrieve results from the *QBeeSim*. In Table 7.1 we present the constitution of both Grover's and random algorithms used for this comparison, where we can see the disparity in the amount of results produced. Since the simulator is intended to be interchangeable with a quantum chip, it should come as no surprise that we need to somehow retrieve results. With *QBeeSim*, the moment of results retrieval to the micro-architecture represents the peak in terms of memory use, for the process. At that moment, we have a data duplicate. The method chosen to minimize this impact for useful *quantum algorithms*is in full display in Figure 7.11. We leverage how sparse the results space is for this kind of algorithms by recording both state and amplitude, instead of having to depend on position for state association and, consequently, storing all states in the micro-architecture. From this point onward, we are able to release the simulator's resources.

In the figure mentioned, we see that random algorithms tend to have a higher peak for the same amount of qubits (scale on the left). Given our lack of time to experiment with multiple random algorithms, the orange curve (using the scale on the right) is also presented in the same figure. It represents the proportion of results states against all possible states. When analysing the graph, we should take into consideration that a high proportion is essentially meaningless for lower number

Qubits	**Gates**		**Results**	
	Grover's	**Random**	**Grover's**	**Random**
3	31	500	1	7
4	58	500	1	16
5	93	500	1	32
6	162	500	1	62
7	247	500	1	124
8	416	500	1	249
9	655	500	1	483
10	1060	500	1	949
11	1621	500	1	1835
12	2512	500	1	3449
13	3847	500	1	6364
14	5814	500	1	11190
15	8819	500	1	19032
16	13282	500	1	27291
17	19897	500	1	31841
18	29766	500	1	27711
19	44323	500	1	21115
20	65948	500	1	2258
21	97803	500	1	23
22	144742	500	1	1
23	213779	500	1	0
24	315192	500	1	0
25	464023	500	1	0

Table 7.1: Circuit constitution used when comparing Grover's algorithm and random walks

of qubits, as the memory expense of duplicating the results space is minimal. Likewise, a low ratio in higher number of qubits can represent a huge amount of states and memory usage, which is visible after fifteen (15) qubits, for example, since there is a continuous separation between both peak memory curves while the proportion curve is reducing.

7.2.6 Predicting simulation limits

The estimation of the number of qubits we can simulate with existing classical technology is an important question that can help in the advance of quantum algorithms. As our last analysis, we answer this question for the stack in question. We decided to use the data collected from random algorithms as they represent the worst-case scenario that make the stack more resource-hungry. That same data is shown in Figure 7.12a. Although with slight deviations, again, by not keeping gate ratios from one algorithm to the other, it is possible to notice the approximation to exponential curves for both time and memory, which goes in accord to this chapter's earlier remarks. By looking at the growth factor of successive memory results, we notice that same factor also growing, becoming closer and closer to two (2). The time growth factor comes a litter shorter in comparison, stabilizing at one and a half (1.5), indicating that states' manipulation (*QBeeSim* operations)

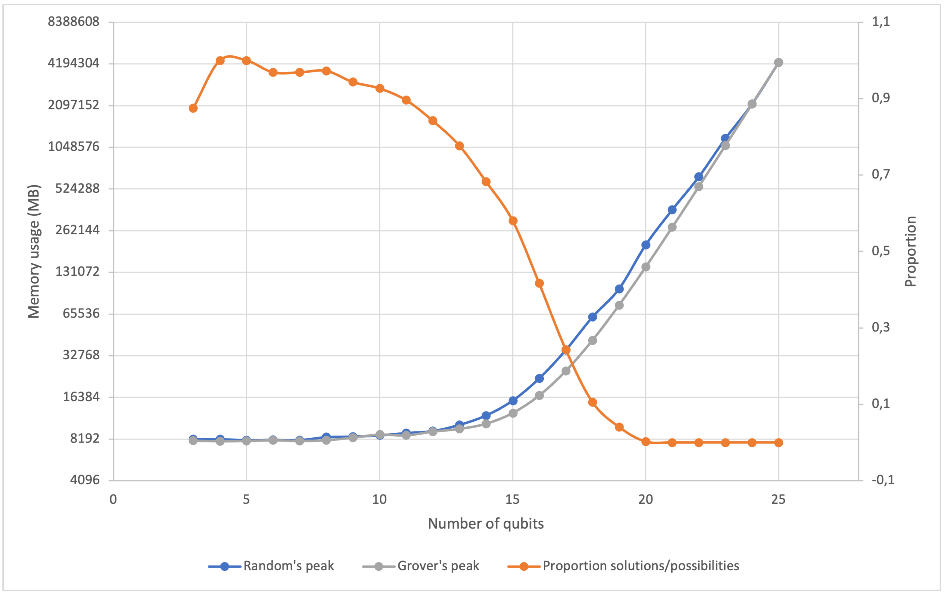
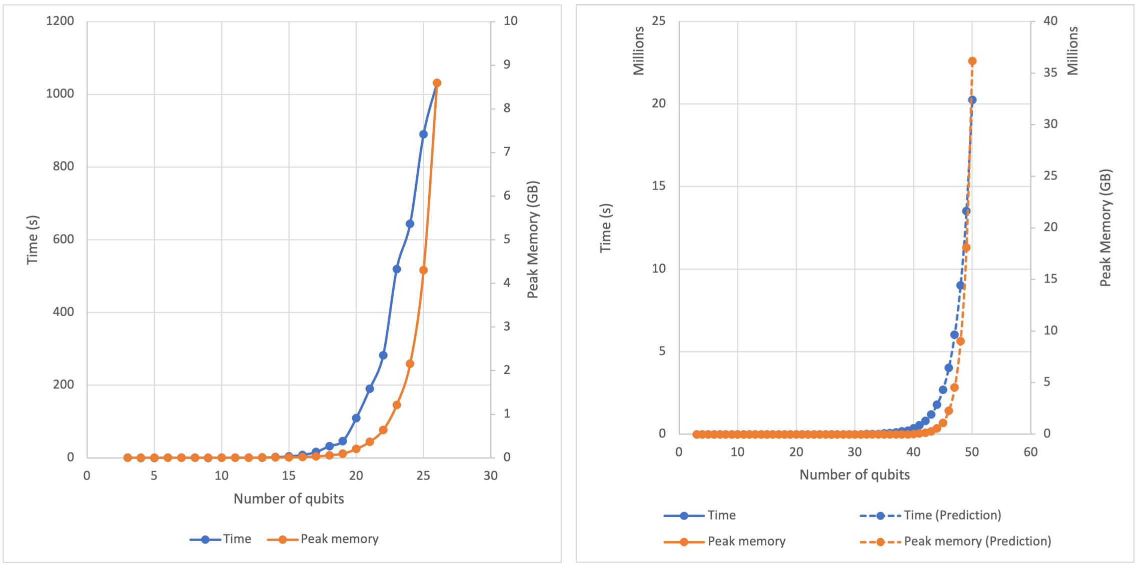


Figure 7.11: Peak process memory usage (logarithmic scale) for random algorithms and Grover's



(a) Measured (b) Estimated

Figure 7.12: Runtime and peak memory usage for 500 gates random circuits running locally

make the bulk but not the entirety of process costly operations. The knowledge of how time and peak memory grow, enables us to predict the consumption of higher qubits circuits on the same conditions by using those same growth rates mentioned, as shown in Figure 7.12b. According to the predictions made, at best, AWS[7] would allow us to simulate up to thirty-nine (39) qubits, while the supercomputer Fugaku[6] should be capable of running forty-seven (47) qubits circuits.

7.3 Conclusion

In this chapter, we present the results for all tests ran. For that, we identify the benchmark system as an AWS EC2 *c5a.4xlarge* instance, used for both the Grover's Algorithm, and random algorithms with a variety of gates and qubits. When comparing the memory usage of the *Quantum Micro-Architecture* and the entire stack in-use, we see a strong correlation of the entire stack's memory to the number of circuit's gates. While analysing the optimal instructions *batch* size, we noticed that small batches result in a longer total runtime. It was only when looking specifically at the difference between local and in database runs that we could explain certain particularities above mentioned. Both local and in database runs are highly affected by qubit additions. Nonetheless, there is a great discrepancy between local and in database runs for the same number of qubits. Calculated average gate times for in database runs are more than a hundred times longer that local average gate times. Despite this notorious difference, we could verify the micro-architecture's intended scalability. The section that followed, studied the impact of the database connection by separating its costly components. From its analysis, we concluded that the previous hunt for an overall optimal batch size is indeed pointless, since its optimally is highly connected to the number of qubits in use, system's memory availability, and needed detail depth when saving intermediary results. Next, we tested the validity of the optimization implemented to target useful algorithms, and, indeed, identified optimality towards its intended purpose, despite the small tests pool. Finally, we made an estimation of our capability to simulate large numbers of qubits using current technology, using worst-case scenario data. To conclude, we should be able to use AWS[7] to simulate up to thirty-nine (39) qubits, while the supercomputer Fugaku[6] would allow for forty-seven (47) qubits.

Chapter 8

Conclusion and Future Work

The last few decades where single-mindedly focus on improving quantum hardware, growing the disparity between it and quantum software. In recent years, that tendency as shown a shift as we see more investigation directed towards quantum algorithms. The work with perfect qubits serves to further close that gap, as the inherent defects associated to any quantum technology currently available can be utterly ignored, bringing a new level of freedom to quantum algorithm developers. It is generous to simply call simulation as sub-optimal, nevertheless, it makes it possible so that software validation and verification does not wait for the perfecting of quantum technology.

The main goal of this thesis was to build a software-based micro-architecture capable of receiving any kind of cQASM instructions, so they could be run on either simulators or quantum chips, as part of a quantum accelerator. With it, we would be empowering the development of quantum software. Despite the hurdles that come from connecting multiple components to said micro-architecture, this goal was achieved, as we now have a platform able to serve as a medium between OpenQL and the layer responsible for running the circuit - *QBeeSim*, while storing huge amounts of relevant circuit metrics.

However, Chapter 7 also tells us our ability to run large circuits is compromised. Despite tinkering with the connection layer, for example, by adding a naive parallel implementation using OpenMP to improve its performance, local circuit runs still perform more than a hundred times faster. And this is on the 8 cores machine used for testing. This proves that there is still work to be done on said layer, so that its performance can be improved, despite mid-algorithm state storage inherently hindering the circuit's runtime.

In addition, note that the simulator is also a work in progress. For the version we used during this thesis development, functionalities like initialization with a user-defined state are still to be integrated, which would allow for partial algorithms runs. With this new developments, corresponding modifications will also need to be integrated in the micro-architecture, and all the way up the stack layers, showing how highly related they are with each other. This high proximity ends up making them all works in progress, as a consequence. Also, we looked at the limits of what our current setup can eventually achieve, and even using the now most capable supercomputer - Fugaku -, we fall short to even fifty (50) qubits. Note that, even if we built a distributed system capable of joining multiple of those same supercomputers, a doubling into our capacity only represents an one qubit increase, showing its scalability limitations. That could lead us to use a different kind of simulator, that does not require all possible states to be handle simultaneously, which would also require changes to the way the micro-architecture works.

Finally, OpenQL simply exports cQASM code that serves as the input for the micro-architecture. We would like to see the inter-layers connection improved, so this no longer is a manual step, enhancing the experience of circuit running.

To summarise, the *Quantum Micro-Architecture* developed during this thesis is in accord to all other layers QBeeX offers. In that way, its development is somewhat stagnant now, but with the other layers' development according to the company's vision, adaptations to it will be required.

Appendix A

Grover's Algorithm Example

```
qubits 5
instance {0,1,2,3,4}
# Initialization
h q[0]
h q[1]
h q[2]
h q[3]
h q[4]
.grover_iteration(4)
# Oracle
cz q[0], q[1], q[2], q[3], q[4]
# Amplification
h q[0]
x q[0]
h q[1]
x q[1]
h q[2]
x q[2]
h q[3]
x q[3]
h q[4]
x q[4]
cz q[0], q[1], q[2], q[3], q[4]
x q[0]
h q[0]
x q[1]
h q[1]
x q[2]
h q[2]
x q[3]
h q[3]
x q[4]
h q[4]
```

```
.measure
display
```

Listing A.1: cQASM code generated for a 5 qubits Grover’s algorithm quantum circuit

Appendix B

Random Algorithm Example

```
qubits 5
instance {0,1,2,3,4}
h q[1]
x q[4]
h q[4]
cnot q[2], q[4]
z q[2]
cnot q[0], q[3]
z q[4]
t q[2]
z q[0]
t q[0]
toffoli q[4], q[1], q[0]
cnot q[0], q[1]
y q[3]
cnot q[2], q[0]
h q[4]
cphase q[0], q[4]
h q[4]
cphase q[2], q[4]
toffoli q[0], q[1], q[3]
t q[2]
h q[2]
h q[2]
x q[4]
cphase q[1], q[3]
cnot q[1], q[4]
cnot q[3], q[4]
x q[0]
cnot q[1], q[2]
x q[0]
toffoli q[3], q[0], q[2]
toffoli q[0], q[3], q[1]
```

```
t q[3]
z q[4]
t q[2]
t q[1]
x q[2]
t q[0]
cphase q[0], q[2]
t q[1]
z q[2]
y q[1]
cnot q[4], q[0]
h q[3]
t q[4]
t q[2]
z q[2]
z q[1]
z q[1]
z q[3]
t q[1]
z q[3]
h q[1]
z q[2]
t q[1]
t q[3]
toffoli q[0], q[2], q[3]
toffoli q[4], q[3], q[0]
cnot q[0], q[3]
cnot q[4], q[1]
x q[4]
t q[1]
x q[0]
cphase q[1], q[0]
cnot q[2], q[0]
x q[3]
x q[2]
y q[1]
h q[4]
t q[2]
x q[2]
cnot q[1], q[2]
x q[1]
cphase q[3], q[0]
t q[1]
y q[0]
z q[3]
h q[2]
cnot q[4], q[3]
x q[1]
h q[0]
```

```
t q[3]
h q[4]
z q[3]
cphase q[1], q[2]
z q[2]
x q[0]
cphase q[4], q[3]
cnot q[2], q[1]
y q[4]
cnot q[1], q[2]
cnot q[2], q[0]
toffoli q[2], q[0], q[1]
t q[1]
t q[1]
x q[3]
cphase q[3], q[2]
z q[4]
y q[3]
cnot q[4], q[0]
toffoli q[1], q[4], q[0]
x q[2]
cphase q[2], q[3]
z q[1]
z q[2]
cphase q[4], q[1]
x q[4]
y q[2]
cphase q[1], q[2]
cphase q[3], q[1]
x q[1]
t q[3]
cnot q[0], q[1]
x q[0]
cnot q[0], q[3]
toffoli q[1], q[4], q[2]
z q[0]
y q[1]
t q[4]
t q[1]
y q[0]
x q[0]
x q[1]
t q[0]
x q[3]
z q[4]
z q[0]
y q[4]
x q[2]
t q[4]
```

```
cnot q[1], q[0]
cnot q[4], q[3]
h q[0]
cnot q[0], q[2]
cnot q[2], q[4]
h q[1]
cphase q[3], q[0]
cphase q[4], q[0]
y q[4]
x q[2]
z q[0]
h q[2]
h q[4]
cphase q[1], q[4]
t q[4]
cnot q[4], q[3]
cphase q[3], q[2]
toffoli q[0], q[3], q[2]
cphase q[2], q[4]
toffoli q[0], q[3], q[2]
h q[0]
y q[2]
t q[1]
toffoli q[3], q[1], q[2]
cnot q[1], q[4]
z q[0]
t q[2]
z q[0]
h q[2]
cphase q[1], q[3]
toffoli q[2], q[3], q[0]
z q[1]
y q[0]
t q[0]
y q[0]
t q[1]
toffoli q[4], q[2], q[1]
t q[3]
h q[2]
t q[0]
cphase q[3], q[0]
cphase q[3], q[4]
h q[3]
toffoli q[0], q[3], q[1]
y q[0]
t q[4]
x q[4]
z q[2]
x q[4]
```

```
z q[4]
cnot q[0], q[2]
cnot q[1], q[2]
toffoli q[4], q[3], q[1]
z q[2]
y q[3]
y q[2]
cnot q[3], q[1]
y q[3]
y q[2]
h q[1]
toffoli q[4], q[1], q[0]
h q[1]
cphase q[4], q[0]
y q[1]
x q[4]
cnot q[0], q[2]
cnot q[1], q[3]
t q[4]
cnot q[0], q[4]
x q[1]
x q[3]
cphase q[1], q[3]
cnot q[1], q[2]
cnot q[4], q[2]
z q[3]
x q[4]
cnot q[2], q[3]
h q[3]
cphase q[4], q[2]
cphase q[3], q[4]
toffoli q[2], q[4], q[1]
cphase q[3], q[4]
cnot q[1], q[0]
y q[0]
h q[4]
t q[0]
toffoli q[1], q[0], q[2]
cphase q[4], q[3]
toffoli q[2], q[0], q[3]
cnot q[1], q[2]
x q[4]
x q[0]
h q[3]
toffoli q[1], q[0], q[2]
h q[2]
h q[4]
h q[1]
y q[1]
```

```
z q[1]
y q[1]
cnot q[1], q[4]
t q[0]
x q[2]
y q[1]
t q[2]
h q[0]
x q[3]
z q[0]
cphase q[1], q[2]
toffoli q[3], q[4], q[2]
y q[2]
h q[3]
y q[4]
x q[3]
t q[4]
z q[0]
toffoli q[1], q[3], q[4]
z q[0]
t q[1]
x q[2]
t q[1]
x q[3]
cnot q[4], q[3]
h q[1]
t q[1]
cphase q[3], q[1]
cnot q[1], q[4]
toffoli q[0], q[2], q[1]
h q[2]
cnot q[4], q[3]
h q[1]
cnot q[4], q[1]
y q[4]
toffoli q[0], q[4], q[3]
cnot q[0], q[2]
x q[0]
z q[4]
toffoli q[3], q[0], q[1]
cphase q[1], q[4]
h q[2]
x q[3]
toffoli q[1], q[0], q[2]
h q[2]
cnot q[3], q[4]
cnot q[3], q[1]
x q[0]
y q[1]
```

```
h q[0]
t q[0]
toffoli q[0], q[3], q[2]
h q[1]
x q[0]
h q[2]
toffoli q[1], q[4], q[2]
t q[3]
y q[2]
z q[0]
cphase q[2], q[0]
t q[3]
cnot q[4], q[3]
toffoli q[4], q[1], q[3]
t q[3]
t q[0]
cphase q[2], q[4]
cnot q[4], q[2]
y q[0]
x q[1]
x q[3]
y q[1]
h q[1]
toffoli q[3], q[2], q[1]
x q[2]
h q[0]
x q[2]
x q[0]
y q[2]
y q[3]
y q[4]
z q[3]
cnot q[1], q[3]
y q[3]
z q[0]
y q[3]
cphase q[4], q[2]
y q[2]
y q[0]
h q[0]
t q[2]
cnot q[2], q[1]
cphase q[3], q[0]
z q[4]
y q[2]
x q[1]
t q[3]
t q[0]
t q[1]
```

```
h q[2]
cphase q[1], q[0]
cphase q[4], q[3]
toffoli q[0], q[3], q[4]
z q[1]
cphase q[0], q[2]
cphase q[1], q[0]
cnot q[2], q[4]
t q[0]
toffoli q[4], q[3], q[0]
x q[1]
cphase q[1], q[3]
cphase q[4], q[0]
x q[2]
toffoli q[1], q[2], q[0]
h q[2]
y q[0]
cnot q[3], q[2]
toffoli q[3], q[4], q[1]
cphase q[3], q[2]
cnot q[1], q[4]
x q[4]
y q[1]
x q[1]
t q[4]
z q[2]
h q[3]
y q[1]
cphase q[3], q[1]
cphase q[2], q[1]
cphase q[1], q[3]
t q[1]
y q[3]
h q[4]
x q[3]
cphase q[2], q[3]
h q[1]
y q[0]
t q[0]
x q[4]
cnot q[4], q[1]
toffoli q[2], q[1], q[0]
t q[0]
y q[1]
z q[4]
z q[4]
z q[2]
x q[2]
toffoli q[1], q[3], q[4]
```

```
cnot q[0], q[4]
t q[0]
t q[0]
h q[4]
cnot q[2], q[1]
cnot q[3], q[1]
z q[4]
y q[4]
cnot q[3], q[2]
cphase q[0], q[1]
h q[2]
h q[4]
x q[3]
x q[4]
z q[1]
toffoli q[0], q[4], q[3]
cnot q[1], q[0]
z q[2]
z q[1]
cnot q[4], q[2]
t q[2]
toffoli q[0], q[4], q[1]
h q[1]
t q[2]
z q[0]
x q[0]
toffoli q[3], q[0], q[2]
x q[3]
toffoli q[2], q[3], q[4]
h q[0]
cnot q[4], q[0]
t q[4]
y q[1]
toffoli q[1], q[0], q[4]
t q[2]
toffoli q[2], q[0], q[3]
h q[2]
t q[1]
z q[3]
cnot q[3], q[0]
cphase q[0], q[1]
cnot q[0], q[3]
cphase q[2], q[3]
cnot q[3], q[2]
x q[3]
t q[3]
toffoli q[1], q[4], q[2]
cphase q[3], q[4]
t q[0]
```

```
cphase q[1], q[2]
cphase q[0], q[4]
x q[1]
y q[2]
cnot q[2], q[3]
cnot q[4], q[1]
x q[1]
z q[3]
y q[3]
z q[3]
y q[2]
y q[2]
x q[4]
y q[4]
y q[0]
y q[3]
x q[0]
cnot q[4], q[3]
t q[2]
y q[2]
x q[2]
h q[3]
cnot q[1], q[3]
z q[3]
cnot q[2], q[1]
h q[3]
z q[4]
cnot q[0], q[1]
toffoli q[2], q[4], q[3]
cnot q[4], q[3]
y q[2]
h q[4]
cphase q[1], q[4]
toffoli q[0], q[1], q[3]
t q[0]
cphase q[3], q[2]
x q[0]
cnot q[4], q[1]
y q[2]
toffoli q[0], q[2], q[3]
toffoli q[2], q[1], q[4]
t q[1]
cphase q[0], q[2]
toffoli q[0], q[2], q[3]
h q[1]
toffoli q[2], q[1], q[3]
y q[3]
y q[4]
cnot q[1], q[3]
```

```
toffoli q[3], q[4], q[0]
h q[0]
cnot q[1], q[3]
cnot q[1], q[2]
h q[2]
x q[4]
x q[0]
t q[2]
h q[4]
toffoli q[1], q[3], q[0]
cnot q[2], q[1]
y q[2]
toffoli q[0], q[4], q[2]
t q[2]
toffoli q[0], q[4], q[1]
x q[2]
cnot q[4], q[3]
cphase q[1], q[3]
t q[3]
y q[4]
h q[3]
z q[4]
cphase q[3], q[1]
t q[3]
cnot q[0], q[4]
z q[2]
z q[1]
t q[1]
display
```

Listing B.1: cQASM code generated for a 5 qubits random quantum circuit

References

[1] Cloud platform - xanadu. https://xanadu.ai/cloud. (Online; accessed on 02/01/2021).

[2] Ibm quantum experience | quantum cloud computer | ibm. https://www.ibm.com/quantum-computing/experience/. (Online; accessed on 02/01/2021).

[3] Qiskit. https://qiskit.org/. (Online; accessed on 02/08/2021).

[4] Quantum inspire - code example: Quantum full adder. https://www.quantum-inspire.com/kbase/full-adder/. (Online; accessed on 14/03/2021).

[5] Quantum lab - damo academy. https://damo.alibaba.com/labs/quantum. (Online; accessed on 02/01/2021).

[6] Top500: Supercomputer fugaku. https://www.top500.org/system/179807/, 2020. (Online; accessed on 2 May 2021).

[7] Amazon ec2 high memory instances. https://aws.amazon.com/ec2/instance-types/high-memory/, 2021. (Online; accessed on 2 Jun 2021).

[8] Bela Bauer, Sergey Bravyi, Mario Motta, and Garnet Kin-Lic Chan. Quantum algorithms for quantum chemistry and quantum materials science. *Chemical Reviews*, 120(22):12685–12717, Oct 2020.

[9] Charles H. Bennett, Ethan Bernstein, Gilles Brassard, and Umesh Vazirani. Strengths and weaknesses of quantum computing. *SIAM Journal on Computing*, 26(5):1510–1523, Oct 1997.

[10] K. Bertels, A. Sarkar, T. Hubregtsen, M. Serrao, A. A. Mouedenne, A. Yadav, A. Krol, I. Ashraf, and C. G. Almudever. Quantum computer architecture toward full-stack quantum accelerators. *IEEE Transactions on Quantum Engineering*, 1:1–17, 2020.

[11] K. Bertels, A. Sarkar, A. Krol, R. Budhrani, J. Samadi, E. Geoffroy, J. Matos, R. Abreu, G. Gielen, and I. Ashraf. Quantum accelerator stack: A research roadmap, 2021.

[12] H. Bombin and M. A. Martin-Delgado. Optimal resources for topological two-dimensional stabilizer codes: Comparative study. *Physical Review A*, 76(1), Jul 2007.

[13] Michel Boyer, Gilles Brassard, Peter Høyer, and Alain Tapp. Tight bounds on quantum searching. *Fortschritte der Physik*, 46(4-5):493–505, Jun 1998.

[14] Adam Brandenburger and Pierfrancesco La Mura. Team decision problems with classical and quantum signals, 2015.

[15] S. B. Bravyi and A. Yu Kitaev. Quantum codes on a lattice with boundary. *arXiv:quant-ph/9811052*, November 1998. arXiv: quant-ph/9811052.

[16] Sergey Bravyi, Matthias Englbrecht, Robert König, and Nolan Peard. Correcting coherent errors with surface codes. *npj Quantum Information*, 4(1), Oct 2018.

[17] K. R. Brown, A. C. Wilson, Y. Colombe, C. Ospelkaus, A. M. Meier, E. Knill, D. Leibfried, and D. J. Wineland. Single-qubit-gate error below104in a trapped ion. *Physical Review A*, 84(3), Sep 2011.

[18] Colin D. Bruzewicz, John Chiaverini, Robert McConnell, and Jeremy M. Sage. Trapped-ion quantum computing: Progress and challenges. *Applied Physics Reviews*, 6(2):021314, Jun 2019.

[19] Ravish Budhrani. Quantumsim: A memory efficient simulator for quantum computing. Master's thesis, Delft University of Technology, 2020. https://repository.tudelft.nl/islandora/object/uuid%3A8d0d0375-f35c-472f-bdd7-ad0012b22c91.

[20] Anasua Chatterjee, Paul Stevenson, Silvano De Franceschi, Andrea Morello, Nathalie de Leon, and Ferdinand Kuemmeth. Semiconductor Qubits In Practice. *arXiv:2005.06564 [cond-mat, physics:quant-ph]*, May 2020. arXiv: 2005.06564.

[21] Xing-Yan Chen and Zhang-qi Yin. Universal quantum gates between nitrogen-vacancy centers in a levitated nanodiamond. *Physical Review A*, 99(2), Feb 2019.

[22] Emmanuel Desurvire. *Classical and Quantum Information Theory: An Introduction for the Telecom Scientist*. Cambridge University Press, 2009.

[23] Marcus W. Doherty, Neil B. Manson, Paul Delaney, Fedor Jelezko, Jörg Wrachtrup, and Lloyd C. L. Hollenberg. The nitrogen-vacancy colour centre in diamond. *Physics Reports*, 528(1):1 – 45, 2013.

[24] Richard P. Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21(6):467–488, Jun 1982.

[25] C. Figgatt, D. Maslov, K. A. Landsman, N. M. Linke, S. Debnath, and C. Monroe. Complete 3-qubit grover search on a programmable quantum computer. *Nature Communications*, 8(1), Dec 2017.

[26] Michael H. Freedman and David A. Meyer. Projective plane and planar quantum codes. *arXiv:quant-ph/9810055*, October 1998. arXiv: quant-ph/9810055.

[27] X. Fu. *Quantum Control Architecture*. PhD thesis, Delft University of Technology, 2018. https://doi.org/10.4233/uuid:8205cc34-30df-45f0-b6eb-8081bdb765b8.

[28] X. Fu, L. Riesebos, M. A. Rol, J. van Straten, J. van Someren, N. Khammassi, I. Ashraf, R. F. L. Vermeulen, V. Newsum, K. K. L. Loh, J. C. de Sterke, W. J. Vlothuizen, R. N. Schouten, C. G. Almudever, L. DiCarlo, and K. Bertels. eqasm: An executable quantum instruction set architecture, 2019.

[29] X. Fu, M. A. Rol, C. C. Bultink, J. van Someren, N. Khammassi, I. Ashraf, R. F. L. Vermeulen, J. C. de Sterke, W. J. Vlothuizen, R. N. Schouten, and et al. An experimental microarchitecture for a superconducting quantum processor. *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, Oct 2017.

[30] Daniel Greenberger, Friedel Weinert, and Head of Section for the History of Science. *Compendium of Quantum Physics: Concepts, Experiments, History and Philosophy*. Springer Publishing Company, Incorporated, 2016.

[31] 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, page 212–219, New York, NY, USA, 1996. Association for Computing Machinery.

[32] Lov K. Grover and Jaikumar Radhakrishnan. Is partial quantum search of a database any easier? In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '05, page 186–194, New York, NY, USA, 2005. Association for Computing Machinery.

[33] K. R. Khadiev and L. I. Safina. Quantum algorithm for shortest path search in directed acyclic graph. *Moscow University Computational Mathematics and Cybernetics*, 43(1):47–51, Jan 2019.

[34] N. Khammassi, I. Ashraf, X. Fu, C. G. Almudever, and K. Bertels. Qx: A high-performance quantum computer simulation platform. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 464–469, 2017.

[35] N. Khammassi, I. Ashraf, J. v. Someren, R. Nane, A. M. Krol, M. A. Rol, L. Lao, K. Bertels, and C. G. Almudever. Openql : A portable quantum programming framework for quantum accelerators, 2020.

[36] N. Khammassi, G. G. Guerreschi, I. Ashraf, J. W. Hogaboam, C. G. Almudever, and K. Bertels. cqasm v1.0: Towards a common quantum assembly language, 2018.

[37] Morten Kjaergaard, Mollie E. Schwartz, Jochen Braumüller, Philip Krantz, Joel I.-Jan Wang, Simon Gustavsson, and William D. Oliver. Superconducting Qubits: Current State of Play. *Annual Review of Condensed Matter Physics*, 11(1):369–395, March 2020. arXiv: 1905.13641.

[38] M. Körber, O. Morin, S. Langenfeld, A. Neuzner, S. Ritter, and G. Rempe. Decoherence-protected memory for a single-photon qubit. *Nature Photonics*, 12(1):18–21, Dec 2017.

[39] Ville Lahtinen and Jiannis Pachos. A short introduction to topological quantum computation. *SciPost Physics*, 3(3), Sep 2017.

[40] Brun T.A. (eds.) Lidar D.A. *Quantum Error Correction*. Cambridge University Press, 2013.

[41] Gang-Qin Liu and Xin-Yu Pan. Quantum information processing with nitrogen–vacancy centers in diamond. *Chinese Physics B*, 27(2):020304, feb 2018.

[42] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, 2010.

[43] J. L. O'Brien, G. J. Pryde, A. G. White, T. C. Ralph, and D. Branning. Demonstration of an all-optical quantum controlled-NOT gate. *Nature*, 426(6964):264–267, November 2003.

[44] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2013.

[45] Nicholas Peters, Joseph Altepeter, Evan Jeffrey, David Branning, and Paul Kwiat. Precise creation, characterization, and manipulation of single optical qubits. *Quantum Info. Comput.*, 3(7):503–517, October 2003.

[46] John Preskill. Quantum Computing in the NISQ era and beyond. *Quantum*, 2:79, August 2018.

[47] Aritra Sarkar, Zaid Al-Ars, Carmen G. Almudever, and Koen Bertels. An algorithm for dna read alignment on quantum accelerators, 2019.

[48] P.W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994.

[49] B. Trauzettel, Denis V. Bulaev, Daniel Loss, and Guido Burkard. Spin qubits in graphene quantum dots. *Nature Physics*, 3(3):192–196, March 2007. arXiv: cond-mat/0611252.

[50] Savvas Varsamopoulos, Koen Bertels, and Carmen Garcia Almudever. Comparing neural network based decoders for the surface code. *IEEE Transactions on Computers*, 69(2):300–311, Feb 2020.

[51] Savvas Varsamopoulos, Ben Criger, and Koen Bertels. Decoding small surface codes with feedforward neural networks. *Quantum Science and Technology*, 3(1):015004, Nov 2017.

[52] R. Versluis, S. Poletto, N. Khammassi, B. Tarasinski, N. Haider, D.J. Michalak, A. Bruno, K. Bertels, and L. DiCarlo. Scalable quantum circuit and control for a superconducting surface code. *Physical Review Applied*, 8(3), Sep 2017.

[53] Benjamin Villalonga, Sergio Boixo, Bron Nelson, Christopher Henze, Eleanor Rieffel, Rupak Biswas, and Salvatore Mandrà. A flexible high-performance simulator for verifying and benchmarking quantum circuits implemented on real hardware. *npj Quantum Information*, 5(1), Oct 2019.

[54] Xiao-Gang Wen. Quantum orders in an exact soluble model. *Physical Review Letters*, 90(1), Jan 2003.

[55] Han-Sen Zhong, Hui Wang, Yu-Hao Deng, Ming-Cheng Chen, Li-Chao Peng, Yi-Han Luo, Jian Qin, Dian Wu, Xing Ding, Yi Hu, Peng Hu, Xiao-Yan Yang, Wei-Jun Zhang, Hao Li, Yuxuan Li, Xiao Jiang, Lin Gan, Guangwen Yang, Lixing You, Zhen Wang, Li Li, Nai-Le Liu, Chao-Yang Lu, and Jian-Wei Pan. Quantum computational advantage using photons, 2020.