POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria Elettronica

Tesi di Laurea Magistrale

Proposal for a multi-technology, template-based quantum circuits compilation toolchain

Relatori:
Prof. Maurizio Zamboni
Prof. Mariagrazia Graziano
Prof. Giovanna Turvani

Candidato:
Manfredi Avitabile

Luglio 2021

Summary

The promising branch of quantum computing has made some significant advances in the last decade. With quantum hardware becoming larger in scale and more reliable, quantum circuits are growing in size and becoming more complex in their implementations. In order to face the challenge of achieving optimality for these circuits, the state-of-the-art in quantum computing has taken inspiration from classical computing by employing automated design tools for optimizing and mapping quantum circuits. Each of these tools are based on one specific strategy used to determine the most ideal improvements to be applied.

In this thesis work, the use of a template-based approach for quantum circuits optimization purposes is explored, and the proposal of a modular toolchain, compatible with three of the most mainstream quantum implementation technologies (NMR, Trapped Ions, Superconducting), is presented. The Toolchain optimizes quantum circuits described in OpenQASM intermediate language, and takes inspiration from the current state-of-the-art workflow of automated design applied to quantum computing. The Toolchain was divided in steps, designed to accommodate a comprehensive template database for quantum circuit identities and other smart optimization methods proposed in the last few years, and then implemented with a modular Python library. The performance of this toolchain was benchmarked on a set of quantum circuits and compared to the ones of other state-of-the art compilers. The toolchain was designed as a "first proposal", and as such it was made able to accommodate further expansion and support for new quantum technologies.
This work is not intended to be classified as "entry level" in the quantum computing's subject: thus, the reader should be advised that in this thesis no in-depth explanations of complex linear algebra applied to quantum mechanics, wavefunctions and quantum state's properties will be found, because this work is not intended to be an introduction to the mysteries of quantum computing for anyone.
With the perspective of analyzing the proposed toolchain, a quick overview of some basic concepts of quantum computing is presented in Chapter 1. This overview does not go into detail in some theoretical concepts in order to avoid redundancy, but is instead focused on making clear the "computer architecture-level" method of tackling the problem on which the Toolchain is based, and presents a description on the Toolchain's target technologies, which were picked for the mainstream role they had in the last years in the quantum computing industry and research field, and on the current state-of-the-art automated design Toolchains' structure.
In Chapter 2, after an introduction to the concept of template-based optimizations

on which the whole proposed design process revolves, the general structure of the implemented Quantum Toolchain, along with its caveats, and a detailed overview of its Step 1 are presented. Step 1 is the first, technology-agnostic part of the quantum circuit optimization procedure, and is described in detail along with its correlated Toolchain's libraries. This Step was implemented as the one in which most of the template identities of the database are exploited, through a reiterative structure. An accurate report of both the functions contained in its correlated libraries and of the circuit equivalences used in its database is presented.
In Chapter 3, the Toolchain's Step 2, concerning the technology-based translation process and the second part of the quantum circuit optimization procedure, and its libraries are presented. This step was designed in order to be able to handle the fundamental sets of gates of each target technology and to operate a circuit decomposition and translation into base gates. This step also features a series of fine-grain optimizations used to merge these base gates and consolidate the circuit's compaction. Once again, an in-depth description of this Step's functions and of their inner workings, alongside with an explanation of the used merging optimization methods, is presented.
Chapter 4 describes the Toolchain's final Step 3, that allows device-specific Layout Synthesis. This step was designed to tackle the arduous task of managing the optimizations regarding two-qubit gates in the target device. A description of the implemented methods of two-qubit gates decomposition into technology-specific gates and of their related template-based optimization process, involving their mirroring and distribution, is then presented.
In the final chapter, the results of each optimization step, as well as the overall performance of the Toolchain, are presented through a series of benchmarks. This benchmark procedure was carried out along the development of the Toolchain's software. A brief summary of the testing methods used to prove the toolchain's performance is reported, followed by graphical representations of a series of data obtained by testing a subset of complex quantum circuits both on the Toolchain and on two of the finest state-of-the-art compilers, IBM's Qiskit and Cambridge Quantum Computing's T-Ket. The results are then compared and analyzed (Chapter 5).
Even though the current Toolchain can still be expanded further and improved to support more quantum technologies and a fully-fledged Layout Synthesis, the results obtained are quite encouraging, and they prove how the Toolchain can be competitive in quantum circuits' optimization when compared to the state-of-the-art's finest compilers, especially when dealing with single-qubit gates. As a final note, some future perspectives are proposed to expand the Toolchain's actual capabilities in the future.

Table of contents

List of figures

Chapter 1

Basic Concepts

Quantum computing is a promising new branch in the computational architectures field. Encompassing several disciplines such as physics, mathematics, chemistry and computer science, quantum computing aims to exploit some peculiar properties of quantum mechanics to create systems capable of running certain kind of algorithms way faster and more efficiently than classical computers. While this fascinating and experimental research field is still limited by premature technologies, especially the ones concerning the relative material science and hardware implementations of the aforementioned systems, its potential is highly esteemed and the change of paradigm it could bring in several everyday life sectors is regarded as an auspicious way to overcome the recent limits of the dying Moore Law and to bring a revolution in modern computing. Such premises have spurred some of the biggest competitors in the computing industry to heavily invest in quantum computing and to start create their own customized hardware and working frameworks, and it is quite likely that in a not-so-distant future the industry will achieve reliable and competitive quantum-based machines and that they will start to utilize some of these in standard applications in the form of quantum accelerators.

To fully understand quantum computing it is though imperative to be familiar with some topics related to multiple scientific disciplines, from linear algebra to quantum-scale physical interactions. However, the aim of this chapter is not to present an overview on these arguments, but to assume a certain level of practicality with the quantum world and to concentrate instead on the topic of gates in quantum computers and of their implementation after a quick review of some basic properties of qubits. As quantum computing is nowadays a subject that has been studied for years, published literature (ex.: [1, 2]) and past degree dissertations (ex.:[3, 4]) covers quite well its basic concepts, so it was deemed to be redundant to approach the topic "from scratch" once again.

1.1 Qubits

A qubit, or "quantum bit", is a unit of information that describes a two-dimensional quantum system, and it is the most fundamental element of the quantum computing world. While similar to the classical bit, a qubit instead of defining either a 0 or a 1 can represent a potentially infinite plethora of different values obtained as a linear combination of its two basis states described by two complex numbers C_0 and C_1. This results in the state of a single qubit being a superposition of $|0\rangle$ and $|1\rangle$.

CLASSICAL BIT

$$|0\rangle = \begin{matrix} 0 \\ 1 \end{matrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \qquad |1\rangle = \begin{matrix} 0 \\ 1 \end{matrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

QUANTUM BIT

$$|\psi\rangle = C_0 |0\rangle + C_1 |1\rangle = \begin{bmatrix} C_0 \\ C_1 \end{bmatrix}$$

$$in\ \ which \quad |C_0|^2 + |C_1|^2 = 1$$

Of course, this "infinity" of values cannot be achieved when measuring a qubit. In fact, once the **measuring** operation is applied the state of the qubit will collapse to either the status $|0\rangle$ or $|1\rangle$ in an irreversible way. The probability to obtain $|0\rangle$ or $|1\rangle$ when measuring a qubit are respectively $|C_0|^2$ and $|C_1|^2$, which are their related probability amplitudes; this is also the reason why their sum must be equal to 1, because it cannot exist a case where the measured state is not one of these two possible states.

1.1.1 Superposition and Entanglement

Qubits have certain intrinsic properties that make them particularly powerful for computational purposes.
The first most interesting property is the one of **superposition**, which can be inferred from their basic description. If one of the two probability amplitudes is 0 and the other is 1, the contents of a qubit information-wise are alike to the ones of a classical bit. If that is not the case, though, the qubit effectively carries a "stronger" information package, handling both of its fundamental states as described by the complex numbers C_0 and C_1. This becomes a very powerful computational tool when a multi-qubit system is employed, since it can be demonstrated that a system composed by N different qubits can effectively represent 2^N different states at once. So, for instance, a two-qubit system would represent 4 states in superposition ($|00\rangle$, $|01\rangle$, $|10\rangle$, $|11\rangle$) as described by the four complex numbers that defines the system's matrix (C_0, C_1, C_2, C_3, in which $|C_0|^2 + |C_1|^2 + |C_2|^2 + |C_3|^2 = 1$).

So, generically speaking, one would obtain a system like this:

$$N\ Qubits \longrightarrow\ |\psi\rangle = \begin{matrix} 0..00 \\ 0..01 \\ ... \\ ... \\ ... \\ 1..10 \\ 1..11 \end{matrix} \begin{bmatrix} C_0 \\ C_1 \\ ... \\ C_k \\ ... \\ C_{N-2} \\ C_{N-1} \end{bmatrix} \quad where \quad \sum_{k=0}^{N-1} |C_k|^2 = 1$$

It is quite clear how in this propriety resides the key to make quantum systems more appealing than classical ones in certain applications, since it allows a potentially huge degree of *concurrent evalutation* of several possible states with a reduced number of qubits, while in large systems a classical computer would not even be able to store such a great amount of complex numbers to describe the system matrix.
However, superposition by itself is not enough. Even if a quantum system can

describe lots of multiple potential states simultaneously, once the measurement operation is applied one would obtain only one of those states: it is thus imperative to arrange the quantum algorithm in such a way to make it collapse with a probability as near to 100% as possible to the one state elected as "correct result". By managing the complex probability amplitudes accordingly, it is possible to exploit all of quantum computing's raw "parallel computation" capability while still ensuring to obtain the desired result.

Another useful property is the quantum **entanglement**.
Quantum entanglement is a kind of perfect correlation totally absent in the classical world, a resource that can only be exploited in quantum mechanics. Entangling two quantum systems results in a correlation which goes beyond space and time limitations between both of them and that brings an ad-hoc collapsing, once the measurement operation is performed on one of the two implied systems. In other words, if an entangled quantum system is described as:

$$|\psi\rangle = |a_0\rangle \otimes |b_0\rangle + |a_1\rangle \otimes |b_1\rangle$$

and the first quantum system is observed, once it is measured and found to collapse in the, say, $|a_1\rangle$ state, then no matter the distance between the two systems, the second system will immediately collapse to the state $|b_1\rangle$.
The possibility to "bound" so deeply two particles or systems is a very powerful tool to exploit, and it both aids in the management of complex probability amplitudes to obtain certain results and makes some otherwise unobtainable tasks possible and employable, such as quantum teleportation or quantum cryptography.

The most famous example of entangled states are the Bell States:

$$|\psi_1\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}} \quad |\psi_2\rangle = \frac{|00\rangle - |11\rangle}{\sqrt{2}}$$

$$|\psi_3\rangle = \frac{|01\rangle + |10\rangle}{\sqrt{2}} \quad |\psi_4\rangle = \frac{|01\rangle - |10\rangle}{\sqrt{2}}$$

In which if the first qubit is measured as $|1\rangle$ then the second qubit will collapse respectively to $|1\rangle$ ($|\psi_1\rangle$), $|1\rangle$ with a phase of $-180°$ ($|\psi_2\rangle$), $|0\rangle$ ($|\psi_3\rangle$), $|0\rangle$ with a phase of $-180°$ ($|\psi_4\rangle$).

1.1.2 Bloch Sphere

The most utilized graphical tool to represent qubit states and operations is the Bloch Sphere.
By rewriting the generic qubit form in another one which is dependent from a longitude and a latitude angles, it is possible to obtain some Cartesian coordinates that can be mapped on a sphere of unit radius:

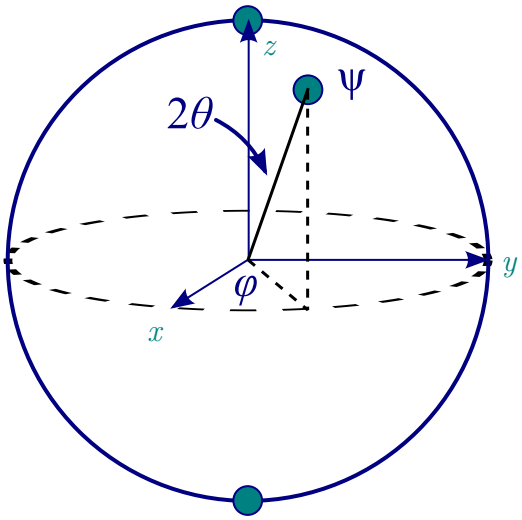


Figure 1.1: Bloch Sphere representation of a generic $|\psi\rangle$ state taken from [18]

The north pole of the sphere represents the state $|0\rangle$, the south pole represents the state $|1\rangle$, and each point of the sphere represents a pure state which is spanned by a Bloch vector.

This representation shows the qubit's superposition of states and how global phase changes do not affect the system's state. Once measured, the state will either collapse to the north pole $|0\rangle$ or the south pole $|1\rangle$. Another important hint given by this graphical representation is that the probability of a state to collapse to one of the two poles of the sphere is solely dependent on its latitude, and it is thus not affected by any rotation (i.e. change of phase) about the Z axis.

1.2 Quantum gates

Quantum gates are the singular elements that once combined compose a quantum circuit and are the key ingredient to manipulate the state vector of a qubit, in a way quite similar to the classical world of digital electronics. However along with the similarities to "traditional" gates, they also have some intrinsic properties that make them quite unique and differentiate them from the standard logic gates.

1.2.1 Definition and properties

A quantum gate is simply defined as an operator represented by an unitary matrix that acts on qubits. The manipulation performed by a quantum gate $\mathbf{G}$ on a standard qubit $|a\rangle$ is expressed as:

$$|b\rangle = \mathbf{G}\,|a\rangle$$

Quantum Gates have three fundamental properties: **linearity**, **unitarity** and **reversibility**.

The **linearity** of a quantum gate makes it fully definable by the effect it applies to the state vectors of the qubits it affects. A particular consequence of this characteristic is the **no cloning principle**, which states that making an exact copy of a given unknown quantum state is not possible, and thus the information contained in a quantum register cannot be copied in a second quantum register without losing it in the former. This principle only affects unknown states, so copying known states

such as $|0\rangle$ or $|1\rangle$ is still possible.

The **unitarity** of quantum gates is an intrinsic property of quantum mechanics' applications, and ensures that the state vector's norm remain unchanged after every manipulation. In other words, this peculiarity of the matrices that describes every quantum gate makes sure that no matter the evolution of the circuit, the overall probabilities remain the same, with no "leaks" nor "increments".

The **reversibility** of quantum gates is their property that separates them the most from our conventional take on logic gates, and that affects all operations pertaining to the quantum world except from measurements, which are irreversible. The reversibility states that it is always possible to reconstruct an input state once the output one is known, and viceversa. Or, in other words:

$$\mathbf{G}^{\dagger}\mathbf{G} = \mathbf{I}_n$$

where $\mathbf{I}_n$ the identity matrix on a n-dimensional vectorial space. This means that manipulations on qubits can be performed in "both directions", as opposed to standard logic gates, which once applied to an input generate an irreversible output, making impossible to "undo" their effect.

1.2.2 Most important quantum gates - single qubit

In this section a comprehensive list of the fundamental quantum gates used as "building blocks" for quantum circuits is presented, along with their defining matrices and circuital symbol representations [5].

PI-RADIANS PAULI GATES (1 QUBIT)

- X gate : -

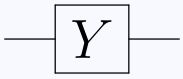
Performs a rotation of π radians around the x-axis, changing the $|0\rangle$ state in the $|1\rangle$, and viceversa.

$$\mathbf{X} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

- Y gate : -

Performs a rotation of π radians around the y-axis, changing the $|0\rangle$ state in the $|1\rangle$ and flipping its phase, and viceversa.

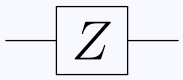
$$\mathbf{Y} = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$$



- Z gate : -

Performs a rotation of π radians around the z-axis, flipping the phase of the complex amplitude associated with $|1\rangle$ and leaving the one associated to $|0\rangle$ unchanged.

$$\mathbf{Z} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$



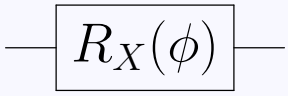
These gates are used to flip the state on the Bloch Sphere around a particular axis, with the **X** gate that is similar to the classical NOT gate, the **Z** that reverses the qubit's phase and the **Y** that is a combination of both effects.
These gates are particularly common cases applying a rotation of π radians of the general R_X, R_Y, R_Z arbitrary rotation Pauli gates.

ARBITRARY PAULI GATES (1 QUBIT)

- R_X gate : -

Performs an arbitrary rotation of ϕ radians around the x-axis.

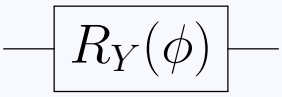
$$\boldsymbol{R_X} = \begin{bmatrix} \cos(\frac{\phi}{2}) & -i\sin(\frac{\phi}{2}) \\ -i\sin(\frac{\phi}{2}) & \cos(\frac{\phi}{2}) \end{bmatrix}$$



- R_Y gate : -

Performs an arbitrary rotation of ϕ radians around the y-axis.

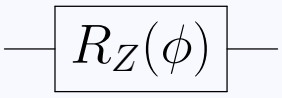
$$\boldsymbol{R_Y} = \begin{bmatrix} \cos(\frac{\phi}{2}) & -\sin(\frac{\phi}{2}) \\ \sin(\frac{\phi}{2}) & \cos(\frac{\phi}{2}) \end{bmatrix}$$



- R_Z gate : -

Performs an arbitrary rotation of ϕ radians around the z-axis.

$$\boldsymbol{R_Z} = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{bmatrix}$$



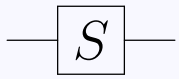
It has to be noted that only the application of R_X and R_Y gates change the output probabilities of the qubit states, because R_Z only change their relative phases modifying the longitude on the Bloch Sphere.

A particular case worthy of attention is the case of the **S** and **T**, both of which are commonly employed and are R_Z gates performing a rotation of $\frac{\pi}{2}$ radians and $\frac{\pi}{4}$ radians respectively.

- S gate : -

Performs an arbitrary rotation of $\frac{\pi}{2}$ radians around the z-axis.

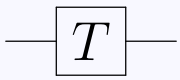
$$S = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{2}} \end{bmatrix}$$



- T gate : -

Performs an arbitrary rotation of $\frac{\pi}{4}$ radians around the z-axis.

$$T = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{4}} \end{bmatrix}$$



Another very important gate is the **Hadamard** gate, a single-qubit gate that sets up a $|00...0\rangle$ or $|11...1\rangle$ state in a perfect superposition of equally probable states. This manipulation is one of the staples of quantum computing, since it is its application that allows a concurrent evaluation of 2^N states starting from N qubits. So to speak, it is the "preparatory stage gate" for every qubit to be utilized in a quantum superposition computation and further manipulated in the circuit to obtain the target output.

- Hadamard gate : -
Sets the qubit in a superposition of states.

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

1.2.3 Most important quantum gates - Multiple qubits

Two-qubit gates in the quantum world are the last essential element alongside the single-qubit gates with which one can compose every kind of quantum circuit [20, 21]. Their typical form is the one of the **controlled gate**: basically this is a *"conditional"* manipulation that implies a **control qubit** and a **target qubit**, with the former that dictates when a certain operator has to be applied to the latter; specifically, the operation of the controlled gate is performed on the target qubit only if the state of the control qubit is $|1\rangle$, and that qubit is left untouched otherwise. This mechanism is compatible with the superposition property.
Of the many possible existing controlled gates, two in particular are relevant: the **Controlled X, CX or CNOT** gate, which in conjunction with the presented single-qubit gates defines the **Clifford + T Gate Set** and other universal gate sets which can be used to describe *every* kind of quantum circuit [19], and the **Controlled Z or CZ** gate. There is also the **SWAP** gate, a gate that is usually obtained as a combination of three CX gates and that describes the logical swap between two different qubits.

- CX or CNOT gate : -

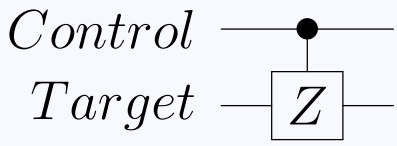
Changes the $|0\rangle$ state in $|1\rangle$ and viceversa on the target qubit if the control qubit's state is $|1\rangle$, and does nothing otherwise. In a quantum register, if we denote the control qubit as "C" and the target qubit as "T", the associated resulting state is $|CT\rangle$, in which the control act as a "MSB".

$$CX = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

- CZ gate : -

Flips the phase of the target qubit if the control qubit's state is $|1\rangle$, does nothing otherwise. In a quantum register, if we denote the control qubit as "C" and the target qubit as "T", the associated resulting state is $|CT\rangle$, in which the control act as a "MSB".

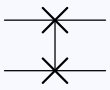
$$CZ = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$



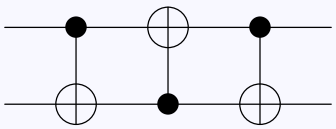
- SWAP gate : -

Logically swaps two qubits.

$$SWAP = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



OR



By using single-qubit and two-qubit gates, one can compose any custom quantum gate desired. A specific case of **multiple qubits gates** are the ones which have multiple control qubits. These more elaborate gates have a mechanism akin to their two-qubit counterpart, and they allow the implementation of any kind of logical operation in a quantum circuit. The most common gates of this category are the **Toffoli or CCNOT Gate**, that acts as a CNOT with two control qubits and that in this way can implement the logical function of the classical NAND gate, and the **Fredkin or CSWAP Gate**, that performs a logical swap depending on the control qubit's state.

Toffoli gate **Fredkin gate**

1.2.4 Physical implementation of quantum gates

Prior to a more in-depth overview on the hardware implementation part concerning quantum devices, it is worth introducing with a few words the topic of physical implementation of quantum gates, which is yet another element that differentiates them from standard logic gates.

Unlike the common gates typical of digital electronics that take inputs and perform Boolean functions generating an output, quantum gates are not physically-implemented devices inserted in a circuit. In fact **the structure of quantum circuits** does not present multiple registers, gates and devices interconnected, but is mainly composed by **quantum registers**, composed by a given amount of qubits on which one can *apply* the effect of certain quantum gates. In other words, quantum gates are effectively some "stimuli" that are applied to qubits to apply the desired manipulation, and much like sinusoidal pulses they are defined by their amplitude and phase and by their time of application on the qubit: by controlling these parameters, it is possible to implement all kind of quantum evolutions. Once again, this is deeply linked to the quantum concept of reversible transformations - since all evolutions can be applied and then undone by applying their opposite, gates by themselves lack a "material" implementation. An important aspect regarding quantum gates that changes in the spectrum of quantum technologies is the physical method with which the stimuli/pulses are applied in the quantum registers and the two-qubit gates, that presents a different mechanism and thus a different gate synthesis depending on the technology employed.

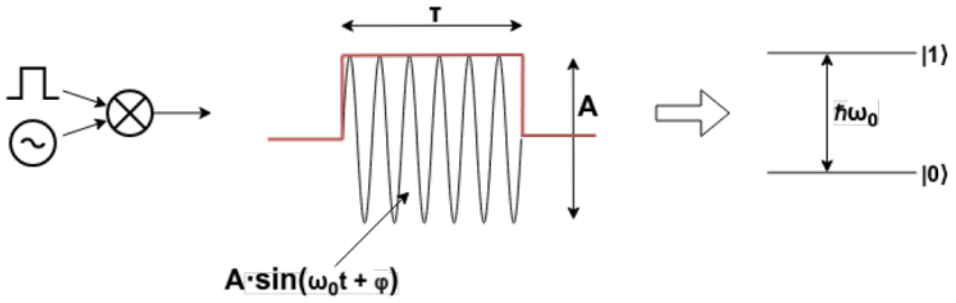


Figure 1.2: Example of a sinusoidal EM Pulse implementing a quantum gate's manipulation, as represented in [4]

1.2.5 Virtual implementation of RZ gates

It must be noted that RZ gates can be implemented in a peculiar and more efficient way in most quantum technologies [22]. Given that the effect of an RZ gate is all about changing the phase of the affected qubit state, instead of applying the gate in the "classical" way, one can resort to a **virtual implementation** that consists in a global phase change in the reference system equal to the phase change that the gate should introduce to be considered in all subsequent rotations in the Bloch Sphere generated by other gates' manipulations. By doing so, the evolution of the RZ gate is applied "for free" without an *actual implementation* of a quantum gate, and all of it happens instantaneously because no pulses or stimuli have to be applied at all to generate the z-axis rotation in the circuit. As will be explained in the next section, this virtual implementation method is particularly efficient to circumvent some of quantum devices' intrinsic constraints, and it should be maximized whenever is possible.

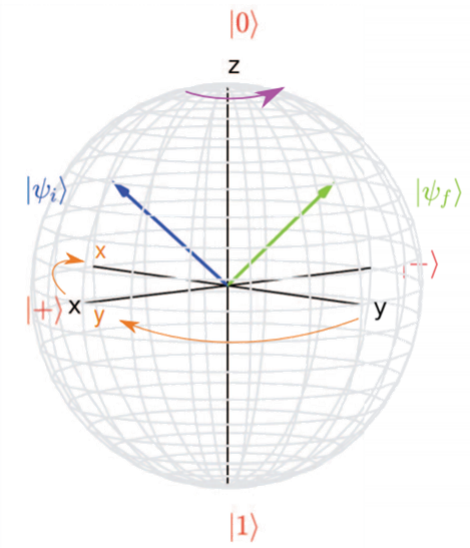


Figure 1.3: Virtual implementation of an RZ gate as a global phase change, as represented in [23]

1.3 Quantum technologies and devices

One thing must be clearly stated when discussing quantum technologies: to this day, all the branches concerning the hardware implementation of quantum computers are highly experimental, currently under development and affected by a lot of issues. Quantum implementation technologies revolves around the very complicated world of quantum mechanics and present some intrinsic problems that at present day are unresolved and heavily limit quantum devices' performance. So even if it is true that in the past two years some researchers claim to have achieved quantum supremacy in certain application, it is also true that existing quantum computers tend to still not possess a grade of reliability that would allow them to fully replace conventional computers, and that their implementation is so costly and complicated to shatter every possible thought of industrializing quantum devices. Even if some important breakthroughs have been achieved in the past years and even if the roadmaps of the major investors in the quantum computing business show that some promising milestones will be reached in the next years [16, 24], quantum computing is still a "promising bet", and any dreams of quantum hardware to be a common sight in

the sector of computing or to replace conventional hardware all together are very unlikely to be fulfilled in the near and even mid-term future. Still, the potential of this subject is very high, so much that it spurred many different realities in the sector to fund and experiment multiple studies on technological implementation: it is thus inevitable that someday the knowledge of quantum interactions and science material expertise will catch up to allow these devices to truly shine.

1.3.1 Fundamental criteria

Quantum computing devices must meet some fundamental criteria to be considered functional. These criteria were introduced in [6] and are known as **Di Vincenzo's Criteria**. While it would be redundant to re-list and define them, it is imperative to give a brief overview on some of the general fundamental criteria derived from Di Vincenzo's ones that poses some functional issues that still haunt quantum devices implementation to this very day:

- **Scalability constraints:** Scalability requires to be able to produce and route enough quantum registers' elements (i.e. qubits) to satisfy the needs of a specific computation, as to also be able to fully characterize these elements and reliably initialize their quantum state. With current quantum hardware, large numbers of reliable qubits are still quite difficult to obtain, and most state of the art technologies can only produce quantum registers with a number of elements in the order of the dozens. However experimental devices have achieved computers counting hundreds of qubits, and some existing roadmaps [16] points that in a couple of years quantum devices employing more than a thousand qubits for computation purposes will be achieved.

- **Layout constraints:** the quantum interactions on which most quantum implementation technologies are based on make the composed devices not **fully-connected**. This means that when two-qubit gates are involved, not all qubits in the register can reliably interact as the gate manipulation would require or can interact at all, imposing some restrictions on the physical layout of the

device that must be considered when computing. The connected qubits, i.e. the qubits that can mutually interact in two-qubit gates operations, are usually the ones with the least geometrical distance in the lattice or the strongest quantum interactions between them, and they are mapped in an ad-hoc layout which depends on the device. All technologies present some sort of constraints related to the layout. Fortunately, the problem can be resolved by using SWAP gates and by making "logically adjacent" all qubits that need to mutually interact. The downside of this solution is that complex layouts often require an hefty number of SWAPs involved, which are usually implemented with a combination of three CX gates and thus make the number of quantum gates in the circuit explode, making an efficient layout synthesis mandatory to reduce the overall complexity of the circuit.

- **Time constraints:** time constraints are actually the biggest obstacle for nowadays quantum computation. Quantum registers' elements, in fact, are not "ideally isolated" and presents some grade of interaction with other quantum systems. After a certain span of time has passed, these interactions alter the superposition states stored in the qubits due to the phenomena of **decoherence** and **relaxation**, resulting in a loss of information and invalidating the computation. For this reason, the overall circuit operation time dictated by the sequence of each gate application must be reasonably *shorter* than this decoherence/relaxation time, if output accuracy is to be achieved. These phenomena intrinsic to the quantum world make the execution of really complex quantum algorithms not always possible, and must be "fought back" by optimizing the overall latency of the circuit by reducing the number of gates applied or the duration of the time slot each gate requires (for example, as stated before, by maximizing the number of virtually-implemented RZ gates, if the quantum technology allows it). Another solution to this troublesome issue is to employ some **fault-tolerant protocols** in the circuit to counter the losses and the errors induced by decoherence/relaxation by utilizing redundancies and error correction codes when encoding information. This mechanism does not come cheap, though: implementing fault-tolerant protocols for even

simple to average quantum circuits requires a substantial increase in the number of used qubits because each single logic qubit is mapped onto multiple effective qubits, making the overall circuits incompatible with the actual scalability limits [25, 26]. As quantum hardware with larger number of qubits will be realized, this problem will eventually smooth out and these kind of protocol will probably become much more common in the quantum computation world.

As stated before, the field of quantum device implementation is still highly experimental. In the years many interesting technologies were discovered and are still being studied, but to the present day none of these has really "emerged" as the victor and proved itself unmistakeably superior to the others. Different major competitors in the sector have started focusing on a single particular technology that eventually became deeply connected to their own quantum environments, and new proposals regarding these technologies are still being developed all around the world.
The aim of these sections is to present the most promising technologies of nowadays' quantum implementation spectrum and to define their pros and cons. These technologies were selected also because they are the most mainstream implementations in today's quantum computing scene, and all of this thesis work is designed to be compatible with them. The description that follows will be only an overview of their most important characteristics: for an in-depth analysis, see [3, 7].

1.3.2 NMR - Nuclear Magnetic Resonance technology

Nuclear Magnetic resonance is a type of quantum molecular technology in which the information is encoded using the nuclear spins of some peculiar liquid-state diamagnetic molecules. One thing must be stated clearly whenever the NMR quantum technology in general is treated: most probably, it is not a *competitive* technology and basically no one of the main players in the quantum business has bet on it to ever bring about a direct and substantial breakthrough in the quantum computing world. In today's state of the art the biggest investors in this field chose to focus on other promising technologies, and the really limited scalability and the decoherence/relaxation time vs gate quantum gate duration ratio limits inherent to molecular technologies are likely to make so that this implementation technique

will never be mainstream hardware-wise when compared to Superconducting or Ion-Based quantum devices. So, why mention it at all? The whole point is that NMR's greatest pro is that it possess a huge *potential for research*, and for a time it has been the unparalleled choice to experiment on quantum protocols in small-scaled devices. While it is improbable that tomorrow's quantum computers will be built using NMR technologies, these present a plethora of properties that makes them really accessible, and more importantly are capable of operating at acceptable temperatures and even at *room temperature*, something that is currently unachievable for most state of the art quantum technologies, that require a work temperature in the order of the single Kelvin degrees. So to speak, NMR represents one of the best "quantum experimental technology" currently available, and could indirectly contribute to the evolution of the quantum computing scenario and to the understanding of quantum mechanics.

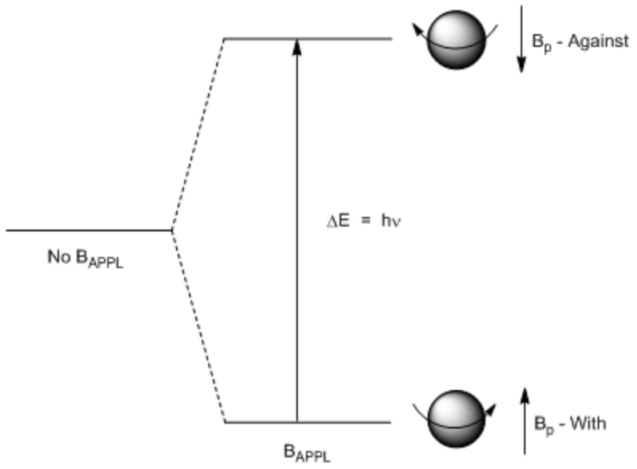


Figure 1.4: Arise of the superposition of energy states following the application of a magnetic field to a nucleus which is used to encode $|0\rangle$ and $|1\rangle$ states in NMR, as represented in [38]

- **Qubits** In NMR, qubits are implemented as nuclear spins [3, 27, 28]: by applying a magneto-static field B_0 two separate energy levels arise in superposition in these nuclei thanks to the **Zeeman Effect** and are used to encode

the $|0\rangle$ and $|1\rangle$ states. These spins can be represented as precessing around $\boldsymbol{B_0}\,\widehat{\boldsymbol{Z}}$ at the Larmor (resonance) frequency ω_0. **Quantum registers** are implemented as molecules composed by the aforementioned nuclei [3], each one possessing a slightly different resonance frequency. It is indeed due to this implementation method that NMR suffers an hefty scalability limit, because the signal used to perform readings suffers from a physical limit expressed in the form of an exponential decay proportional to the number of qubits, which makes measurements straightforward impossible to perform in large-scaled quantum circuits [28]. Another minor but non-negligible issue is that creating single molecules containing a large amount of different nuclei whose resonance frequency can be clearly differentiated by one another is really hard. The number of qubits can be increased by using *homonuclear* molecules, whose trade-off is that the difference between each nucleus' resonance frequency is not that great and thus it is harder to target the "right" nuclei, making a smart RF pulses management mandatory [3, 28].

- **Single-qubit gates** Single qubits manipulations are obtained in NMR technology by applying a RF magnetic field B_R to the molecules composing the quantum registers [3, 28]: by tuning this alternate field to the resonance frequency of the target nucleus, a rotation of its spin is applied and its measurement probabilities are altered. By changing this field's application parameters (phase amplitude and time duration) it is possible to implement two Pauli rotation gates ($\boldsymbol{R_X}$, $\boldsymbol{R_Y}$). The $\boldsymbol{R_Z}$ gates can be implemented as a combination of the former two gate, or by using a virtual implementation.

- **Two-qubit gates** To rotate spins depending on other spins' states the **J-Coupling** phenomenon is exploited [3, 28]. J-Coupling is a type of hyperfine interaction between different spins in which the nuclei's electrons couple making them interact between themselves. This influence dictated by the electrons' coupling is used to realize multi-qubit gates; although the **CX Gate** can be composed by using this mechanism, NMR's "true" two-qubit basis gate is the **CZ Gate**: in fact it can be demonstrated that when using J-Coupling by applying a slight compensation R_Z on bot spins (which can be implemented virtually as always) a CZ Gate is effectively obtained.

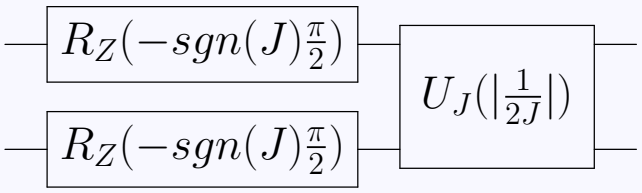


Figure 1.5: Implementation of base gate **CZ** in NMR technology by using J-Coupling, in which the coupling parameter $\frac{1}{2J}$ is the time in which the system is free to evolve with no RF fields applied as shown in [23]

Advantages/Disadvantages overview

+ Really cheap quantum technology: it is relatively easy to implement and does not require costly equipment to perform measurements.

+ Capability to operate at comfortable temperatures up to room temperature.

+ Theoretically Fully-Connected. This may not always be the case, as the coupling between two nuclei dictated by their J-Coupling factor may not be strong enough for certain pairs to allow the implementation of two-qubit gates.

– Very poor scalability: obtaining quantum register molecules capable of implementing more than a couple dozens of qubits at best is currently impossible.

– NMR's long computation times are far worse than the ones of other technologies. While decoherence and relaxation times are not short in themselves, the duration of the pulses which implements the quantum gates are long enough when compared to them to make these time constraints and information loss issues non-negligible.

1.3.3 Trapped Ions technology

The Trapped Ions implementation technology is one of the two most mainstream quantum technologies in the state of the art, and the one that has achieved the greatest stability and accuracy in computational results [9, 29]. This technology uses certain kinds of vaporized ions trapped in a vacuum and encodes $|0\rangle$ and $|1\rangle$ using their energy states. Trapped Ions were one of the first leading quantum technologies since Shor developed its renowned algorithm, and remained as such to this day, in which it is considered the most promising implementation method for quantum-based hardware alongside the Superconducting technology. The main advantage of quantum devices implemented using Ions is that, as told before, this is the most stable quantum technology known to this day. Decoherence times and states longevity are incredibly long, in the order of tens of minutes, easily surpassing all competitors and allowing to compute complex algorithms with no issues. Also their accuracy is unparalleled, and some qubits manipulations with an accuracy of up to 99.9999% have been reported [9], with only superconducting devices being able to obtain similar results when entangling gates are involved. Also, ion traps are a technology based on fully natural phenomena, while other implementation such as Superconductor require the manufactoring of man-made devices. What hampers these ion-based devices from truly shining is their intrinsic realization complexity and their abysmal multi-qubit gates speed, which makes achieving quantum advantage not a small feat. On the other hand, some advantages presented by Trapped Ions are unrivaled to this day by all of the competing technologies [10], and their status as one of the quantum technologies with most potential to succeed is as strong as ever.

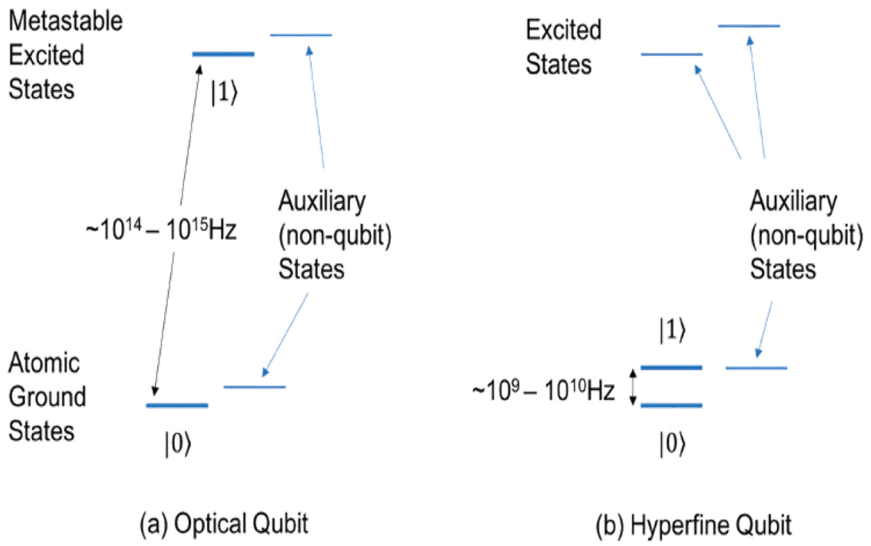


Figure 1.6: Quantum information encoding in Trapped Ions systems' states

- **Qubits** In Trapped Ions qubits are implemented by exploiting two separate energy levels in superposition within the ions themselves to encode the $|0\rangle$ and $|1\rangle$ states. Qubits can be implemented using two different methods: **hyperfine implementations** or **optical implementations**. **Hyperfine implementations** exploit some basic nuclei-electrons interactions and obtain rotations and initialization on single qubits by applying a RF Magnetic field B_R at the right frequency, and encoding quantum information on two different ground atomic states. Their states have a frequency difference in the order of tens of GHzs, and they tend to be more stable, with the longest decoherence times ever measured in the quantum hardware branch. These implementations became the flagship technology of IonQ [30, 32], which usually utilizes an implementation based on atomic clock technologies using $^{171}Yb^{+}$ [7, 11], or, more recently, $^{133}Ba^{+}$ [29] ions. **Optical implementations** perform rotations and initializations using a charge pump laser tuned to the right frequency, and encoding quantum information on different orbitals - $|0\rangle$ in a ground state and $|1\rangle$ in an excited state. Their states have a frequency difference in the order of the thousand of THzs, and they tend to be less accurate compared to Hyperfine

implementations. These became the flagship implementations of Alpine Quantum Technologies [31, 33], and are usually implemented using $^{40}Ca^{+}$ ions [7]. **Quantum registers** are implemented as ion chains suspended in a vacuum. These chains benefit of an important property: given the right controls, each ion in the chain can be moved in it without losing quantum information. This is in fact the reason why Trapped Ions devices are one of the few examples of quantum hardware in which true Full Connectivity is achieved.

- **Single-qubit gates** By changing the manipulation method's parameters it is possible to implement two of the Pauli rotation gates ($\boldsymbol{R_X}$ and $\boldsymbol{R_Y}$) through the generic rotation gate $\mathbf{R(\theta,\phi)}$. It has to be noted that $\boldsymbol{R_Z}$ gates can be universally implemented as a combination of R_X and R_Y, but that their virtual implementation in this technology is not always possible: only in the past few years it has been discovered an addressing system efficient enough to change the state relative phase on a single qubit basis [11, 12].

- **Two-qubit gates** Two-qubit manipulations are implemented by using the vibrational modes of the ion chain and by exploiting Coulombian interactions. This relative manipulation is transpiled using the **XX(χ) Gate**, in which the χ represents a phase that is dependent by the pair of ions on which the manipulation is enacted. This gate's implementation is based on the notorious Molmer-Sorensen interaction [13], and it is used to compose the **CX Gate**, as shown in Figure 1.7.

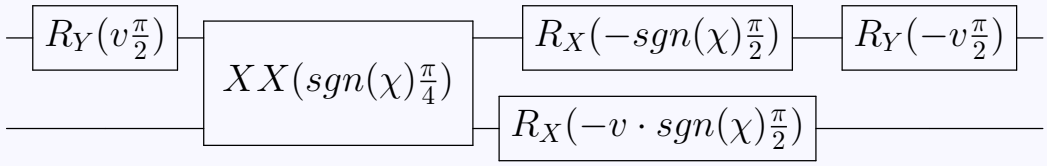


Figure 1.7: Example of implementation of a base gate **CX** in Trapped Ions technology by using the XX gate, in which v is an arbitrary parameter with the value of ∓ 1, as shown in [34]

Advantages/Disadvantages overview

+ The most accurate quantum technology currently known, with an exceptional stability provided by extremely long decoherence/relaxation times.

+ Can operate at comfortable temperatures, but very low temperatures produce better results in terms of noise-related performance.

+ Fully-connected, and with a excellent scalability.

– High complexity: the required instrumentation and pulse controls make this technology hard to implement and really expensive.

– While Trapped Ions yield good results in terms of quantum registers sizes, trapping very large amounts of ions is becoming ever more challenging and the scalability more limited.

– Abysmal gate speed, especially when compared with other quantum technologies such as the Superconducting one [7, 10]. Multi-qubit gates in particular tend to be very slow, and their speed tends to be inversely proportional to the ion chain's length: the more ions are present, the slower the manipulation times are [7].

1.3.4 Superconducting technology

The Superconducting implementation technology is the second most mainstream quantum technology in the state of the art, and the one that was elected as flagship technology by several major investors in the sector, such as IBM and Google. The Sycamore quantum processor, with which Google Quantum research team affirmed to have achieved actual quantum supremacy for the first time in human history in [14], was indeed built using Superconducting technology. The Superconducting technology uses the charge states of an LC Resonator in superconductive phase to encode the $|0\rangle$ and $|1\rangle$ states. The inductor in the aforementioned resonator is implemented as a *Josephson Junction* [4, 28]. This particular device is useful because it changes the energy gap between the $|0\rangle$ and $|1\rangle$ states in such manner that it becomes different from all the other energy gaps relative to other higher-energy states; in this way an ad-hoc pulse at the resonance frequency relative to that energy gap can be generated, charging the junction and making all other states off-resonance and allowing only the two basis states as possible states. Josephson Junctions are also engineered in order to resonate at a frequency that can be easily produced with the state of the art microwave signal generators. Superconducting technology has proved itself quite versatile in the last decade, boasting an excellent accuracy, an overall fine stability and good scalability potential. What really makes this technology valuable is its inherent working speed, with manipulation on multiple qubits performed in the order of nanoseconds, and the fact that is partially compatible with today's state of the art semiconductor fabrication processes, allowing for an easier production of high scalability and quality devices and improving the overall designability of the technology, with many different viable qubit implementations

techniques each with its own pros and cons ([15]). The main disadvantages of this technology are the limited longevity of its quantum states, that makes decoherence and relaxation phenomena non negligible, the fact that it is not fully-connected and thus requires a well-thought layout to adapt the computation to the hardware in use, and finally that it requires abysmal operative temperatures in the order of the dozens of mK degrees, making it expensive to implement and unfit for industrialization. Even with its own limits, Superconducting quantum devices are the quantum hardware that, to this day, have probably been more successful in terms of results. Moreover, the major investments of the big names in the industry makes this a fast evolving technology, with roadmaps such as the one published by IBM that states that quantum devices with more than 1000 qubits will be delivered by the end of 2023 [16], and this has encouraged the blooming of several environments such as Google's CircQ and IBM's Quantum Experience. As of now, the future of Superconducting devices seems all but radiant.

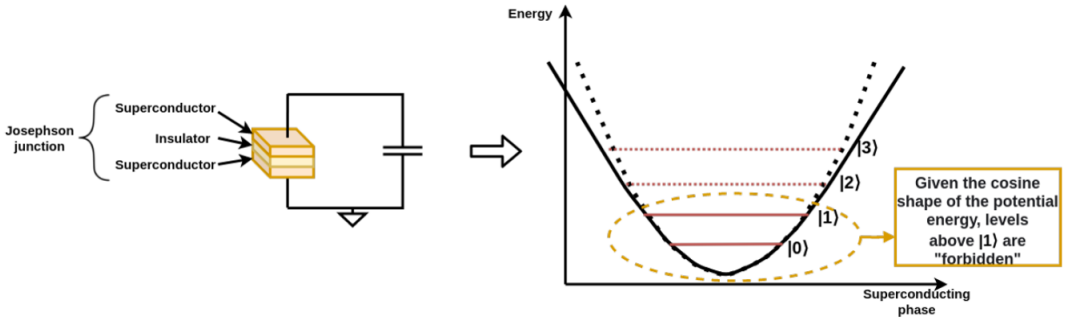


Figure 1.8: Josephson Junction and its effect on the energy level in a superconducting LC Resonator, as represented in [4]

- **Qubits** Superconducting qubits are implemented as LC Resonators that are quantum oscillators that behave similarly to the default harmonic oscillator. The inductors of these resonators are implemented using a Josephson Junction which introduces anharmonicity in the system, thus permitting to facilitate the addressing of only two energy states which are then used to encode $|0\rangle$ and $|1\rangle$, as represented in Figure **??**. Superconducting **quantum**

registers are implemented as ensembles of these resonators in loop [35]. Usually the Josephson Junctions that compose different qubits are engineered to posses intrinsic resistances that slightly differ from each other [7] in order to make the corresponding single resonance frequencies clearly identifiable and different between them, but without straying too much from the optimal control frequency chosen for the device. Implementing this optimization is not trivial, and the current fabrication processes do not allow to create quantum registers with a lot of qubits each with a slighlty diverse resonance frequency. The intrinsic scalability limits of this technology originate from this and other issues.

- **Single-qubit gates** Qubits initialization and manipulations are performed through magnetic resonance by applying a microwave pulse at the right resonance frequency, which excites the current within the resonator. By changing this pulse's application parameters it is possible to directly implement two of the Pauli rotation gates ($\boldsymbol{R_X}$ and $\boldsymbol{R_Y}$). The $\boldsymbol{R_Z}$ can still be implemented as a combination of the other two Pauli gates or virtually.

- **Two-qubit gates** Two-qubit manipulations are implemented by coupling different qubits and by exploiting the *crossresonance* phenomenon [4]: by exciting a Superconducting qubit with a pulse at the resonance frequency of its coupled other qubit, a manipulation on the former is performed depending on the latter's state. It is also possible to enact a direct capacitive coupling between adjacent qubits, but doing so, crosstalk issues tend to arise in devices that uses great numbers of qubits. All this can be used to implement the basic **CX Gate**.

IBM's Superconducting native set

IBM's Superconducting quantum computers are based on a different native set with respect to the standard base gates: it employs the **CX Gate** as the two-qubit fundamental gate, and then the **U1, U2, U3 Gates** as universal single-qubit gates

[36]. Since this thesis work largely revolves around the usage of IBM's open environment comprising the IBM Quantum Experience and the Qiskit transpiler and optimization open-source framework, it is worth spending a few words on this particular native set and its characteristics.

IBM'S NATIVE SET

- U1(λ) gate : -

Also known as "phase gate", it is equivalent to the $\boldsymbol{R_Z(\lambda)}$ Gate. It is a subcase of the U3 gate ($\boldsymbol{U1 = U3(0, 0, \lambda)}$).

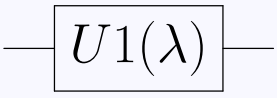


Figure 1.9: U1 gate symbol

- U2(ϕ, λ) gate : -

Equivalent to a $\boldsymbol{R_Z(\phi) \times R_Y(\frac{\pi}{2}) \times R_Z(\lambda)}$ gates combination, its duration is half of the U3 Gate's one, and thus it can be used to implement gates such as the Hadamard more efficiently. It is a subcase of the U3 gate ($\boldsymbol{U2 = U3(\frac{\pi}{2}, \phi, \lambda)}$).

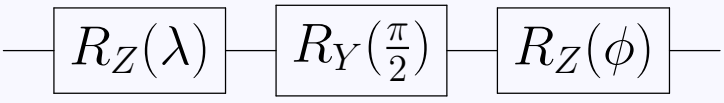


Figure 1.10: U2 gate equivalence

- U3(θ, ϕ, λ) Gate : -

General case gate that describes all possible single-qubit rotations, its duration is twice the one of the U2, so it is advisable to use its subcases U1 and U2 whenever is possible. It is equivalent to a $\boldsymbol{R_Z(\phi) \times R_Y(\theta) \times R_Z(\lambda)}$ gates combination, which can also be expressed as $\boldsymbol{R_Z(\phi) \times R_X(-\frac{\pi}{2}) \times R_Z(\theta) \times R_X(\frac{\pi}{2}) \times R_Z(\lambda)}$, which is indeed equal to two U2 gates. Because of this, it is clear why U3 gates' duration lasts twice the duration of U2 gates.

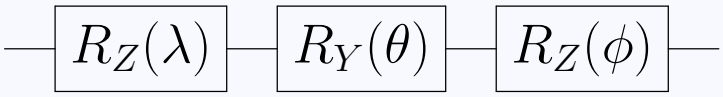


Figure 1.11: U3 Gate Equivalence

- CX Gate : -

Standard CX Gate.

Advantages/Disadvantages overview

+ Excellent scalability, it is the technology that was used to implement several large-scale quantum circuits, such as Google's Sycamore [14].

+ Good reliability and really fast operation times.

+ Compatible with today's microelectronics fabrication processes in the hardware creation phase and with today's state of the art microwave pulse generation techniques in the gate implementation phase.

– States' coherence tends to collapse quite easily when compared to other technologies, so decoherence and relaxation issues become non negligible.

– Require abysmal operative temperatures, which make this technology expensive to implement and complex to manage.

– Superconducting devices cannot be fully-connected, and this means that when dealing with multi-qubit operations several SWAP gates may be required for the circuit to logically function, increasing overall complexity and latency issues.

1.4 Quantum computing's state of the art Design Toolchain

Because of the ever expanding interest in quantum computing and its steady development in recent years, researchers have started interrogating themselves on which is the most convenient way to implement such challenging devices. Taking inspiration from classical computing, the quantum state of the art rapidly adopted the philosophy of the automated design to produce reliable architecture. The process of defining an optimal toolchain to define and optimize a quantum processor is still a matter of trial and error in development, but in the last decade the comprehension of quantum algorithms and quantum circuits has steadily improved in the industry, and nowadays the steps most crucial to efficiently optimize a quantum circuit are known, defined and implemented in every design process in the state of the art.
Below is represented what is considered to be the current state of the art's quantum architecture Design Toolchain [17], and its various steps are described:

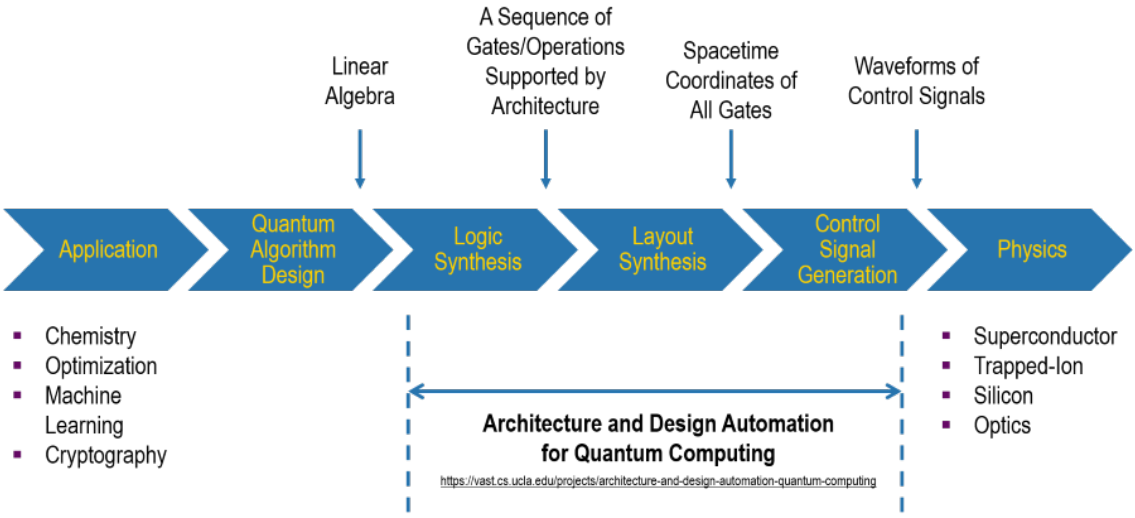


Figure 1.12: Current state of the art quantum Design Toolchain, as represented in [17]

- **Application:** In this step the target application for which the quantum device will be used is defined. Depending on the field of interest, different kind of algorithms might be considered.

- **Quantum Algorithm Design:** Once the purpose of the algorithm to be executed is set, the quantum circuit that will implement the algorithm itself must be devised. In this step the operations that must be performed in the circuit are stated and put in the desired order, and then the algorithm to be implemented is described in algebric terms. Most of the time algorithms expressed in algebric terms are then translated using **intermediate representation languages for quantum instructions**, such as OpenQASM, Quil, Blackbird and others. These languages are used to easily describe the quantum architecture to be implemented, and they are very similar to classical Hardware Description Languages.

- **Logic Synthesis:** One of the most crucial steps in automated quantum design. In this step the circuit expressed in intermediate quantum languages is translated using the target quantum technology's own library, and all the gates are transpiled using the technology's proper basis gates. It is most common that CX gates are left untouched, since in the next step most Layout Synthesis tools tend to work using non-decomposed CX/CZ gates [37]. A lot of optimizations can be applied in this step, where it is possible to obtain the greatest reduction of single-qubit gates amount and to manipulate them in order to employ the most efficient and cost-effective gates. This sequence of operations proper of the target device is then produced as output.

- **Layout Synthesis:** This is the other most crucial step in automated quantum design, and the one that currently presents the most criticalities to be overcome when working with non fully-connected technologies. In this step the circuit is mapped and scheduled on the hardware of the device to be used, working on the spacetime coordinates of all the involved gates. In other words, this is the step in which the critical multi-qubit gates are adapted to the device connectivity map and the required SWAP gates are inserted, and in which all the optimizations regarding the logical shifting of qubits and the two-qubit base gates are performed. CX Gates are then decomposed using the target technology's own library. The ouput of this step may be represented using intermediate languages as a ordered circuit containing all the optimization and logical swaps required to work on the device. Since multi-qubit gates

are often very useful in quantum computing but also very troublesome if the stability and logical success ratio of the circuit are to be guaranteed, the step of the Layout Synthesis is the one on which usually most efforts are made in the optimization phase, and the one that can potentially yield the best improvements in the toolchain.

- **Control Signals Generation:** This is the step in which the gates are effectively "programmed" by setting the pulse and stimuli generation devices required by the target technology, and in which the decided schedule is actually enacted. In some technologies the gate pulses management can be actually harder than handling the quantum hardware itself.

- **Physics:** In this step the quantum circuit is physically implemented. All the techniques relative to the target technology are employed, qubits are initialized, gates are applied in the set order and then finally measurements are performed.

It is clear that different steps in the toolchain are prone to different kinds of optimizations. In the *Algorithm Design* step, the target algorithm to be performed can be improved in a technology-agnostic way, but no action can be taken on how it will perform on actual quantum hardware. The *Control Signals Generation* and *Physics* step are the most important step to consider if one aims to perfect the hardware involved and to devise new smart and efficient ways to employ the target technology with no control at all on the sequence of operations to be performed. The *Logic Synthesis* and *Layout Synthesis* steps are the crucial link between the algorithm and the hardware, and are the best source of improvements architecturally speaking.

Chapter 2

Proposal for an Optimized Quantum Toolchain and overview of its Step 1

This chapter is divided in two sections. In Section 2.1, a proposal for a new Design Toolchain based on the current state-of-the-art one presented in Chapter 1 [17] and represented in Figure 1.12 is explained, along with its basic concepts, its bias and its peculiar optimization strategy which employs a template-based approach. A general overview of the strategies regarding the Toolchain's exact implementation and modularity is also provided. After this first description of the proposed Toolchain's structure, Section 2.2 features an in-depth overview of its correlated Step 1. In general, in this part the exact purpose and the inner structure of the first part of the template-based Logic Synthesis is explained, and brief descriptions of each of the functions that compose the related Python libraries are provided. Each of these descriptions are presented along with a complete list of the circuital identities and templates which are employed in the optimization process.

2.1 The Optimized Quantum Toolchain

The aim of this thesis work is to present, implement and test **a new Quantum Design Toolchain** capable of performing a certain set of optimizations on a quantum circuit and to elaborate it through various step of design automation to finally produce an output file that describes an optimized QC translated according to the chosen technology's own library and, in its final version, adapted to the chosen device layout. Both the input and output circuits involved are meant to be described at high level with an intermediate representation language.
Before fully explaining the approach on which this Toolchain is based and seeing in detail its inner workings, two things must be pointed out: first, the Toolchain is designed to work with - and therefore produce as output - files that describe quantum circuits using **IBM's OpenQASM 2.0 language** (or **Open Quantum Assembly language**) [39]. The language itself was chosen because of its common use in the state-of-the-art as intermediate description language for quantum computing purposes, and because most of this thesis work revolved around the usage of IBM's open source resources such as **Quantum Experience** or **Qiskit** for testing and benchmarking purposes. Recently IBM released a new iteration of the language, **OpenQASM 3.0** [40], which implements significant syntax changes and that trades the pseudo-similarity of the 2.0 iteration to a classical HDL language for a new form which is more akin to a programming language. The compatibility with this iteration was discarded for two reasons: first, it was officially presented right in the middle of the development of the Optimized Toolchain, and its hefty differences with OpenQASM 2.0 made impossible to allow a 100% compatibility without redesigning from scratch the Pyhton libraries. Second, it was considered preferable to employ a well-established language instead of an experimental one to gain access to a series of resources and repository regarding quantum circuits described in OpenQASM 2.0 and to make the Toolchain itself a useful tool for each user who has "quantum computing need" but still is not experienced with this new OpenQASM iteration. Of course, the abstract concepts on which the whole Toolchain is based could be made compatible with OpenQASM 3.0 described files, and as a future work an *ad-hoc* implementation for this purpose could very well be possible. An in-depth description of OpenQASM 2.0's basic syntax and mechanics will not be presented

nor described in the following Chapters, since it is not required to understand the basic conceptual operations performed in the Toolchain. For further details on the matter, see [4, 39].
Second, the Toolchain was designed to be compatible with the three important quantum implementation technologies that were presented earlier in Chapter 1: the **NMR**, the **Trapped Ions**, and the **Superconducting** technologies. When using the Toolchain, optimizing a quantum circuit for a certain target technology implies the usage of a specific gate set and an application of a subset of possible optimizations, all of which will be further explained in Chapter 3. These technologies were chosen because, as explained in Chapter 1, they are the most mainstream implementation technologies in the state-of-the-art (Trapped Ions, Superconducting), and because they retain an high value for research purposes (NMR). Of course, once again, the Toolchain's libraries could be expanded to be compatible with more niche or specific target quantum technologies.

2.1.1 The template-based approach

Nowadays, quantum circuits have reached such large scales and complexities that resorting to a manual implementation effort is utterly unpractical. As described in Section 1.4, the industry took inspiration from the past of classical computing to adapt new strategies for the future of quantum computing and resorted to *automated design methods* to produce refined, reliable quantum circuits to be adapted on the target devices. The goal is always the same: producing an optimized quantum circuit based on an implementation of a certain quantum algorithm. The entity of the applied improvements and their focus may vary depending on the desired performance, the target device and many other variables. These optimizations are usually performed by **compilers** [42], which take as input the high-level, abstract description of a certain application of a quantum algorithm or of a quantum circuit performing a certain algorithm and automatically "translate" and adapt it following the given specifications, generating a quantum circuit built to work smoothly with the target technology or on the target device and making sure to optimize its circuit depth, latency, critical operations number and/or other parameters. There are many philosophies on which the current state-of-the-art design methods are based. Some

prefer to apply **optimal** implementations of given reversible circuital structures defined by a certain number of qubits whenever possible [41], but in more general case other approaches are commonly devised. Some of the most common approaches are the **heuristic** methods that rely on a plethora of possible different logic mechanisms (transformations, Binary Decision Diagrams or BDDs, unitary matrices evaluation or search algorithms such as the A* [41, 42]) to produce reasonable solutions starting from a fixed amount of available computational resources [44], and that are one of the few feasible ways to solve NP-hard problems. Some others are other, more advanced **meta-heuristic** approaches that implements more unique and complex solutions, like the **Quantum Annealing** approach [43].
To optimize quantum circuits in this thesis work's proposal, none of the more "classical" approaches was chosen. Instead of resorting to a theory-rich, algebraic evaluation-based or branching diagram-based complex approach, a more "circuital" method was deemed worthy of being explored: the **template-based approach**. According to this approach, the input quantum circuit is scanned, and a series of equivalences and circuital identities denoted as templates, which are described in the Toolchain's libraries, are identified. Once a template is detected, the quantum gates adjacent to it are identified and, if deemed convenient, the template structure is "switched" to its equivalent form to obtain a compaction or optimization in the circuit.

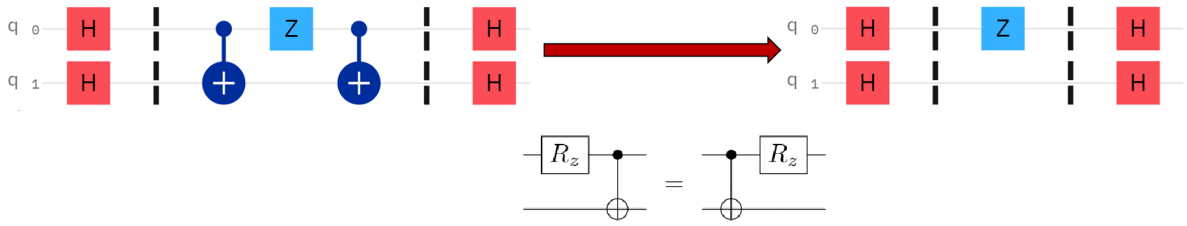


Figure 2.1: Example of application of a template as described in [47] to perform a circuital optimization, represented using IBM's Quantum Experience [48]

This approach is based on the very simple and intuitive concept of *circuital equivalences* and does not require complex mathematics or algebraic evaluation tools to be implemented, thus potentially requiring a minor average amount of computational

resources to be compiled in acceptable times. The intrinsic flaw of this approach is that by being based on case-by-case applications performed while scanning the circuit, it suffers from a **limited foreseeing** and is incapable of looking for multiple steps ahead. In other words, this approach implies that no initial disadvantage, even if that could yield a greater payout in the future, will ever be accepted. Moreover, because the quantum circuits are optimized by applying a remodeling based on a purely circuital evaluation, without analyzing the exact state of given qubits after or before each operations, the template-based approach does not consider all optimizations or equivalences based on knowing the state of certain qubits, favouring general optimizations instead.

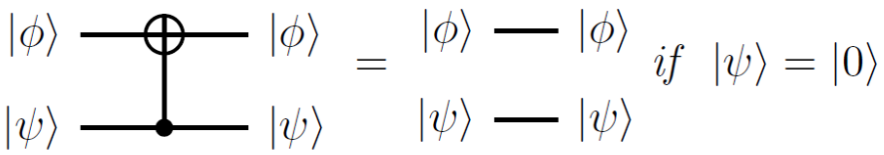


Figure 2.2: Example of an "idle" CX gate which is redundant if the control qubit's state is $|0\rangle$ as represented in [46], and that can be targeted using a state-based optimization method

2.1.2 The Toolchain's structure

The proposed Quantum Toolchain works by taking as input a given quantum circuit described in OpenQASM 2.0 and produces as output an optimized, compacted circuit decomposed using a target technology's own set of gate. This output is also described in OpenQASM 2.0. Using the state-of-the-art toolchain represented in Figure 1.12 and described in [17], the Toolchain operates by implementing the optimizations proper of the **Logic Synthesis** and a segment of the **Layout Synthesis** blocks by using a sequence of three different scripts labeled as **"Steps"**. The Toolchain operates in what could be described as the "central" part of the stet-of-the-art design toolchain, and it does not dabble with the **Algorithm Design** block (whose algorithm's selection and implementation in a quantum circuit is left to the

user's choice) and with the **Control Signals Generation** block (which as it was described in Section 1.4 involves a more "physical" and technological implementation of the circuit on the target device, and it is thus the field of interest of different, more specific applications [45]).

Logic Synthesis block

- **Step 1: QASM template-based optimization -** The aim of this step is to apply the main bulk of the circuital equivalences and template-based substitutions described in the Toolchain's libraries and to compact the input circuit as much as possible. This is achieved by generating wherever it is possible some **circuital null operations**, thus allowing to reduce the number of quantum gates, all without hampering the logic function of the circuit. When a straightforward elimination of redundant gates is not feasible, this Step implements some transformations in order to maximize the use of a certain preferable kind of gates. The output produced by this Step is an optimized OpenQASM-described circuit in which all gates have been decomposed to the $\boldsymbol{R_X}$, $\boldsymbol{R_Y}$, $\boldsymbol{R_Z}$, **CX**, **CZ** subset of gates, and in which each Pauli gate is transformed in its rotational form using floating point notation.

- **Step 2: Technology-dependent gates compaction -** The aim of this step is to translate the output circuit of Step 1 into the proper set of gates relative to the target technology, which could be specified as input and chosen from the **NMR, Trapped Ions and Superconducting technologies**. This translation is applied universally, but CX gates are not decomposed in their basic constituting gates yet: this does happen in Step 3. Along with this translation, Step 2 employs a rotation-based compaction, which aims to consolidate the optimizations of Step 1, followed by a series of technology-dependent optimizations created by smartly managing peculiar gates, like CZs transformed into CXs and the IBM's set U gates.

Layout Synthesis block

- **Step 3: Distribution/Mirroring-based optimizations and CX gates decomposition -** The aim of this step is double: the exploit of a certain subset of templates which may ensure the generation of a more suitable layout of CX gates, and the decomposition of each two-qubit gate by using the target technology's own native library. Other reiterative applications of the rotation-based optimizations of Step 2 are also applied. As of now, this Step covers the fully-connected technologies but does not implement yet an hardware-mapping system of the circuit for the Superconducting technology. However, it was created as a modular part in which this and other layout-related functions could be implemented in the future to expand the Toolchain's utilities.

Implementation overview

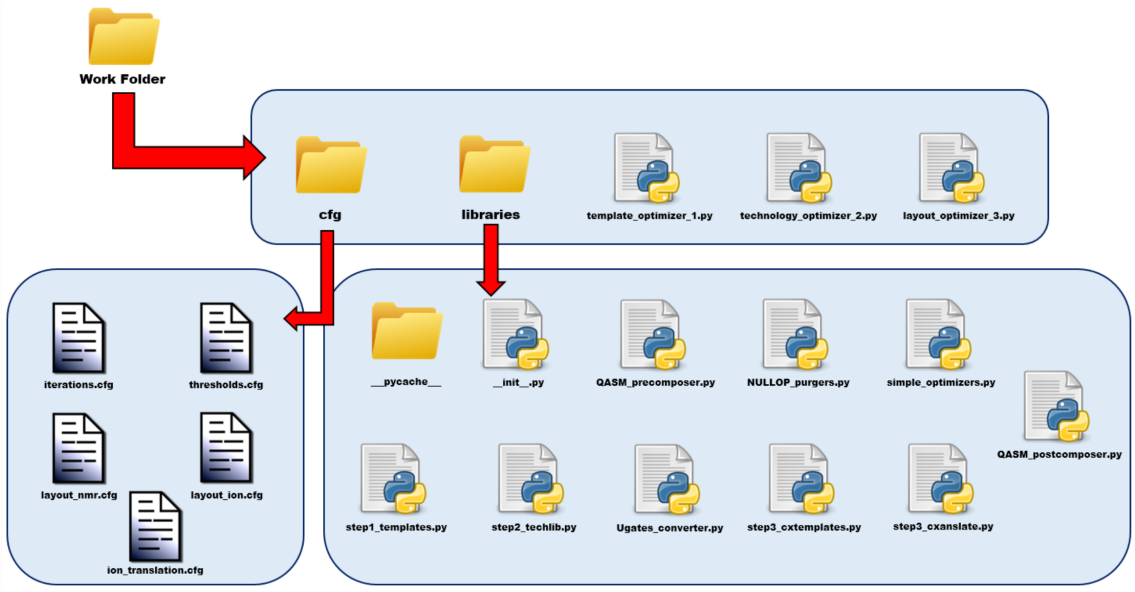


Figure 2.3: Overall structure and contents of the Toolchain

The Toolchain was implemented using a series of **Python 3.x scripts**. Each of the three main scripts ("optimizer_1", "optimizer_2", "optimizer_3") implements one of the three Steps and takes as input an OpenQASM 2.0-described .qasm file. To ensure the complete application of the Toolchain as intended, each step must take

as input the file produced as output by the previous step or, in Step 1's case, the original quantum circuit file to be optimized. It is also possible to apply the specific optimization described into one of the three main scripts to an *ad-hoc*, custom file. In fact, the whole Toolchain was designed in a completely **modular** way through a **libraries-based implementation**, which allows total control on which steps are applied to a certain circuit, high flexibility in the functions' usage, the capability to set some specific parameters through the edit of specific files and the faculty to allow future modifications and expansions.
Each step requires specific inputs when they are to be executed, and these inputs will be covered and described for each step in Section 2.2 and in Chapter 3, Chapter 4. The operations performed by each step is *circuit-wide*, and the subpartitions of a given circuit by using *Barriers* are not considered optimization-wise. Also, each qubit is treated as ideal and thus differences in performance between qubits are not considered, and at the time of writing each architecture is treated as fully-connected. Finally, the Toolchain requires the installation of the **python-math, NumPy, SciPy and Configparser libraries** to work, and for easiness of parsing reasons it requires that each .qasm file to be used as input starts with the following lines (which means that only a single quantum register and a single classic register are supported, and that the quantum register have to be named with the standard **"q[...]"** name):

```
OPENQASM 2.0;
include \enquote{qelib1.inc};
qreg q[..];
creg c[..];
```

- **Libraries:** Following the philosophy of total modularity, several libraries in form of scripts were created, each containing a subset of functions designed to tackle a certain specific part of the optimization process. This allows the main script for each Step to remain "lean" and easily customizable, and it also facilitates the nested usage of the functions in multiple occurrences, while making each library easily readable. Each library with its related set of functions will be described in detail, along with the inner workings of each step, in Section

2.2 and in Chapter 3, Chapter 4, and they are all contained in the **"libraries"** subfolder.

- **.cfg Files:** To easily allow the edit of certain parameters that are particularly uncomfortable to pass as shell inputs, each step supports the reading of **.cfg files** to determine such parameters and to act accordingly. These files are contained in the **"cfg"** subfolder. Once again, this system is fit for a further customization and expansion of the Toolchain's capabilities, and many more parameters could be easily set using these very configuration files. All these files require the **Configparser library** to be interpreted, and thus this Python library is required for the Toolchain to work. As of now, five .cfg files are implemented:

  - The **thresholds.cfg** file, which defines in a "10^X form" both **the approximation of** π and of its dividends to be employed when using rotational gates in floating-point notation (this is labeled as **Threshold 1**, and its allowed maximum precision is 10^{-15}) and **the rotation value under which a gate is considered a null operation** when combining multiple rotational gates, with a lower threshold corresponding to a stricter criteria to approximate a rotation to 0 (this is labeled as **Threshold 2**, and its allowed maximum precision is 10^{-12}).
  - The **iterations.cfg** file. which defines the iterative parameters **IT1, IT2** used in Step 1, which describe how many times certain portions of optimizing code are called and are somewhat akin to a rough "optimization grade" indication (as explained in Section 2.2.1) and the iterative parameter **IT3** used in Step 3, which describe how many iterations of the code section that applies optimizations based on templates using CX gates are performed (as explained in Section 4.1.1).
  - The **ion_translation.cfg**, which defines all generic translation parameters for Step 2 and Step 3 and, as of now, contains a boolean parameter labeled as **Iontran**, that in the case of Trapped Ions technology defines

if R_X and R_Y gates are to be left in their original form or translated in R(θ,ϕ) form.

- The **layout_nmr.cfg**, which regulates how CX gates have to be managed in Step 3 in the **NMR technology** case, as explained in Section 4.1.2. This file contains the **CZTranslation** parameter, which defines if CZ gates have to be decomposed in the structure represented in Figure 1.5 or left untouched, and the **NMR Layout** parameter, a custom list of lists which defines the J coupling sign of each couple of interacting qubits in the circuit. This sign is used to implement decomposed two-qubit gates.
- The **layout_ion.cfg**, which regulates how CX gates have to be managed in Step 3 in the **Trapped Ion** case, as explained in Section 4.1.3. This file contains the **CZTranslation** parameter, which defines if CZ gates have to be decomposed in the structure represented in Figure 1.5 or left untouched, and the **Ion Layout** parameter, a custom list of lists which defines the J coupling sign of each couple of interacting qubits in the circuit. This sign is used to implement decomposed two-qubit gates.

2.2 Step 1 - QASM template-based optimization

As stated before in Section 2.1.2, in the first step of the Toolchain the main bulk of template-based optimizations is performed, with the aim of optimizing as much as possible any redundancy or suboptimality present in the circuit through the reiterative application of a series of "coarse" compactions based on simpler circuital equivalences followed by specific, fine-grain remodelings on each detected template. Step 1 is designed to be completely **technology-agnostic**: in fact, it does work at an high-level of description, not considering implementations that use technology-specific gate sets or constraints dictated by an hypothetical target device's layout or ideality issues. The workflow of this Step is:

- trying to maximize the creation of circuital null operations in order to maintain the logic functionality of the circuit while at the same time reducing the number of involved quantum gates.

- Trying to compact as much as possible the quantum circuit through the usage of less gates that apply a rotation on the qubits' states equivalent to the ones applied by multiple gates.
- If neither of these two operations above are feasible, trying to maximize the number of $\boldsymbol{R_Z}$ **gates** through specific circuital forms and substitutions in order to incentivize an extended use of **virtual implementations** to reduce the circuit's latency all across the board.

Step 1's optimizations are particularly efficient in reducing single-qubit gates, but they also encompass several templates which contain CX and CZ gates, thus allowing a situational and yet efficient set of circuital improvements. As of now, Step 1 is compatible with an extended *Clifford + T* gate set, and it supports the usage of $\boldsymbol{R_X}$, $\boldsymbol{R_Y}$, $\boldsymbol{R_Z}$, **X, Y, Z, S, T,** $\boldsymbol{S^\dagger}$, $\boldsymbol{T^\dagger}$, **H, CX, CZ and CCX (or Toffoli) gates** in the input quantum circuit. IBM's gate set comprising the **U1, U2, U3** gates is also passively supported, but no optimization are performed on such gates in this step (that passage is delegated to the technology-specific Step 2, as explained in Chapter 3). No other quantum gates usage is supported at the moment, such as the definition of custom circuital structures, but while the current supported gates are more than capable of covering most of the common QASM circuits, both the supported gate set and the feature involving the definition of custom gates/structures could be easily expanded and implemented in the future.
Step 1 requires two input when executed from shell: the **input .qasm circuit file**, which describes the reference quantum circuit that as to be optimized, and the **Subcircuit parameter**, which is a boolean flag that defines if the circuit is indeed a *subcircuit* to be used in conjunction with other QASM-described entities and thus if certain optimization regarding R_Z gates can be employed.
The templates and identities contained in the Toolchain's libraries were extrapolated from [8, 20, 49, 50, 51] or obtained through calculation and experimentation.
All the benchmarking procedures, the results and their related analysis are reported in Chapter 5 in the appropriate section.

2.2.1 Step 1's structure

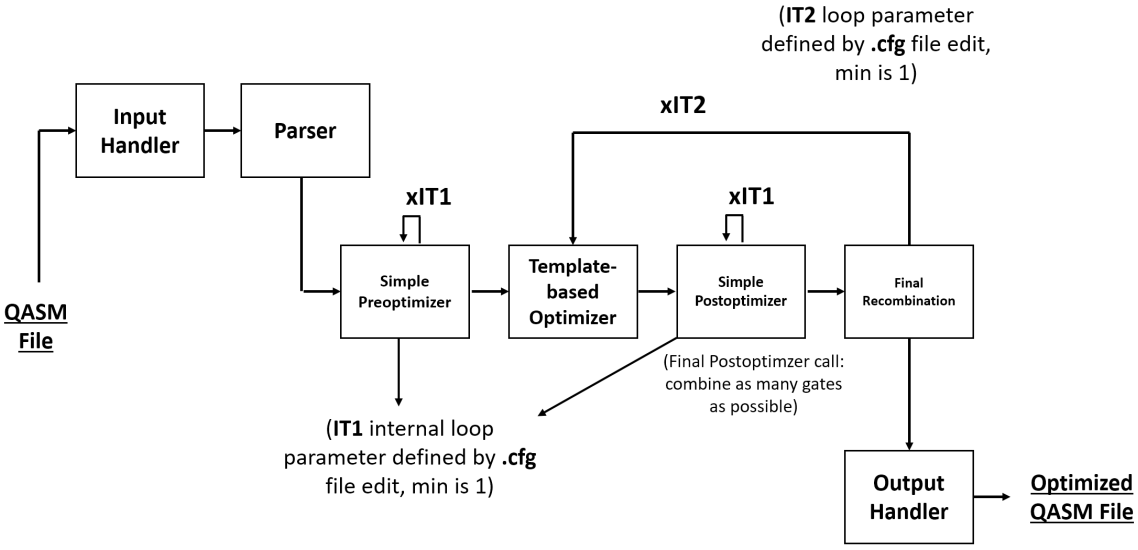


Figure 2.4: Representation of the overall structure of the Toolchain's Step 1

Step 1 is implemented with the **template_optmizer_1 script**, and is divided in the following sections:

1. **Input Handler section:** This section handles the correct reception of inputs and provides error messages if a wrong number of parameters or a wrong file extension are passed as inputs. It also read the .cfg files to define the thresholds and approximation accuracy to be used in the optimization process and to set the iteration parameters IT1 and IT2 used in the looped section of the code. For what concerns the thresholds, the script provides an error message if an approximation accuracy equal or greater than 1 is provided, and a warning message if a null operation identifier threshold is defined, since this could bring to "overzealous" optimizations that could compromise the logic functionality of the circuit. If a non-integer number is passed for the iterative parameters it proceeds to round it, and if a number which is less than 1 is set in the .cfg files for these loops it automatically sets it to "1".

2. **Parser + Precomposer section:** The **Parser** section generates the circuit list on which the script will work based on the input file and rearranges

it, "cleaning" it from endline spaces, blank lines, barriers and comments. It also identifies the number of qubits involved in the target circuit and saves all the measurements performed in the input file on a temporary list, which will be appended to the output circuit list once the optimization is completed. The **Precomposer** section employs the functions contained in the **QASM_precomposer library** to decompose universally-applied gates and Toffoli gates in basic gates applied to each involved qubit, to convert all Pauli gates in floating-point notation (ex: $R_Z(\frac{\pi}{3})$ gates to $R_Z(1.04666..)$ gates) or, when possible, in "non-rotational" form (ex: $R_Z(\frac{\pi}{2}$ gates to S gates) and to then perform some preliminary combinations on S-type and T-type quantum gates.

3. **Null Operations Purger section:** In between the Precomposer and the first Simple Optimizer section there is a first "ghost section" that calls the basic functions contained in the **NULLOP_purgers library**. This serves as a first coarse optimization able to easily purge the most straightforward null operations in the circuit. A double call of the InitialZ and FinalZ functions is also performed to cover the cases in which this first optimization brings some R_Z gates eligibile to be purged right after initialization or right before measurement in the circuit.

4. **Simple Preoptimizer section:** This section uses the functions contained in the **simple_optimizers library** to exploit some less complex circuit identities involving single-qubit gates labeled as "*simple optimizations*" to perform an ubiquitous and thorough compaction of the circuit. These optimizations are reiterated in a loop defined by the **IT1** parameter in order to achieve maximum compaction throughout the compilation. Each gate detected as the target of a possible simple optimization is left untouched if it is susceptible of being part of a more complex, higher-gain optimization-wise template; these gates are usually identified by being single-qubit gates of the "right" type near a CX or a CZ gate.

5. **Template-based Optimizer section:** The "core" of the optimization process of this step, this is the section in which the functions contained in the

step1_templates library are used to exploit some complex templates and advanced circuital identities to achieve optimizations of both single-qubit and two-qubit basic gates. Some of these functions are call themselves a second time in order to try to perform a compaction of gates of the same type if a stronger optimization could not be achieved. This section followed by another set of simple optimizations and a final recombination is reiterated through a loop defined by the **IT2** parameter to maximize the number of optimizations applied. The recommended value of IT2 is actually "1", because a single iteration of this core section of the optimization process is usually enough to detect and perform all possible template-based optimizations. That being said, the parameter gives the possibility to try to achieve an higher grade of optimization, trading off computation time in order to increase the probabilities of all possible templates actually being detected and exploited.

6. **Simple Postoptimizer section:** This section employs once again the functions used in the Simple Preoptimizer section in order to exploit the set of simple optimizations. This section's unique behaviour is that on the very last application of the simple optimizations' loop it tries to enact its peculiar compaction even on gates that was previously detected as susceptible of being part of a template, since at this point of the process all the complex template-based identities should have already been exploited.

7. **Final Recombination:** This section uses the **QASM_postcomposer library** to translate all gates in their floating-point rotational notation and to attempt a last "brute force" compaction between them. This section makes the output circuit composed by **CX, CZ, R_X, R_Y, R_Z gates only** and ensures that each single-qubit gate is **of a different kind with respect to the gates adjacent to it**.

8. **Output Handler section:** This section uses the final circuit list to generate a .qasm file in the working directory that has the same name of the original input file plus the suffix "_optimized", appends to it the measurements contained in the original circuit (if it is not a subcircuit, of course) and provides a message announcing that the optimizations were completed successfully and the name

of the generated file.

2.2.2 QASM_precomposer library

QASM_PRECOMPOSER

- **UniversalDecomposer function:** This function scans the circuit and when it finds universally applied gates it decomposes them into the same kind of gate applied to each qubit.

  Example in a four-qubit circuit case:

  $$h\ q; \quad \Longrightarrow \quad \begin{array}{l} h\ q[0]; \\ h\ q[1]; \\ h\ q[2]; \\ h\ q[3]; \end{array}$$

- **Translator function:** This function transform all Pauli gates in a "non-rotational" form. When these known forms are not exploitable, it translates each Pauli gate whose rotation parameter is expressed as a multiple of "pi" into its floating-point notation by evaluating the parameter in pi-form and recalculating it using the π value approximated using the .cfg accuracy threshold in the Input Handler section.

Examples:

rz(pi/4) **or** *rz(0.78...)* *q[n];*	$\Longrightarrow$	*t q[n];*
rz(pi/2) **or** *rz(1.57...)* *q[n];*	$\Longrightarrow$	*s q[n];*
rz(+/- pi) **or** *rz(3.14...)* *q[n];*	$\Longrightarrow$	*z q[n];*
rx(+/- pi) **or** *rx(3.14...)* *q[n];*	$\Longrightarrow$	*x q[n];*
ry(+/- pi) **or** *ry(3.14...)* *q[n];*	$\Longrightarrow$	*y q[n];*
*ry(3*pi/2)* *q[n];*	$\Longrightarrow$	*ry(4.71...)* *q[n];*

- **ToffoliDecomposer function:** This function detects each Toffoli gate in the circuit (labeled as "*ccx q[a],q[b],q[c];*") and replaces it using its decomposition in Clifford + T gates.

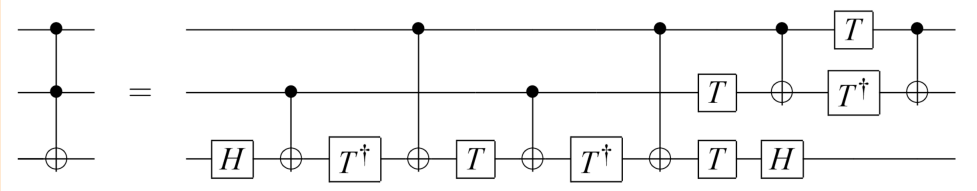


Figure 2.5: Decomposition of a Toffoli gate in Clifford + T gates, as represented in [49]

- **Precombo_ST function:** This function scans the circuit and performs a preliminary combination of S-type and T-type gates into Z gates and S-type gates, respectively.

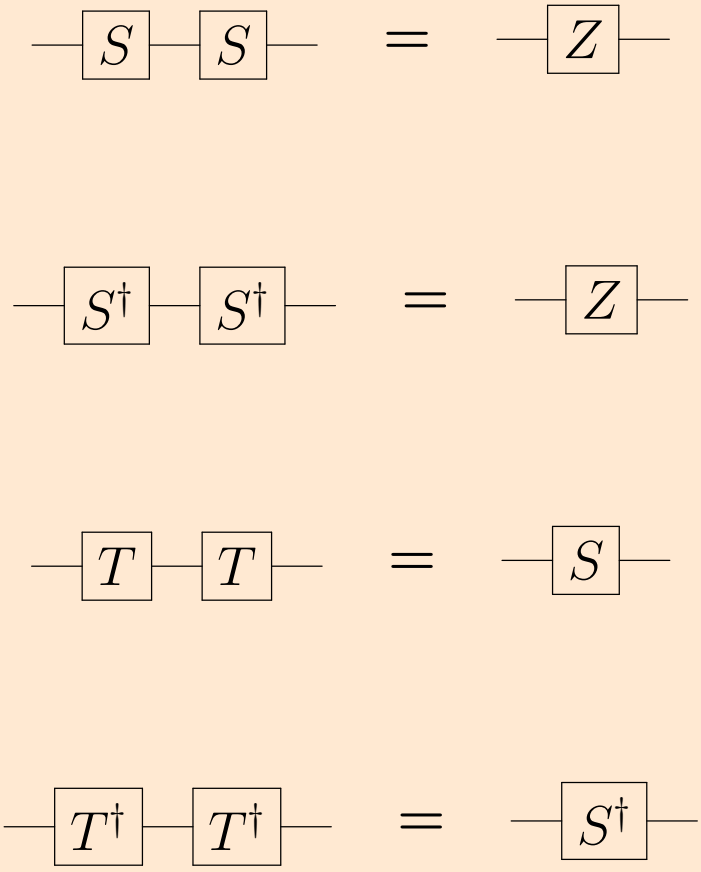


Figure 2.6: Preliminary recombinations of S-type and T-type gates

2.2.3 NULLOP_purgers library

NULLOP_PURGERS

- **FinalZ function:** This function scans each qubit line in the circuit backwards, starting from the circuit's end, and checks if the **last gate** is a R_Z: if it is not, it proceeds with the next qubit line; if it is, it purges it and resumes from the new circuit's end. This optimization exploit the fact that ***R_Z* gates right before measurement are negligible**, since they only insert a phase variation. If the circuit is a Subcircuit and thus it is not followed by measurement operations, this functions does nothing.

- **InitialZ function:** This function scans each qubit line in the circuit forwards, starting from the circuit's beginning, and checks if the **first gate** is a R_Z: if it is not, it proceeds with the next qubit line; if it is, it purges it and resumes from the new circuit's start. This optimization exploit the fact that ***R_Z* gates right after qubit initialization to $|0\rangle$ are negligible**, since they do not insert any phase variation. If the circuit is a subcircuit and thus it is not preceded by qubits' initializations, this functions does nothing.

- **NullPurge function:** This function is **the main building block** on which all optimizations in the Toolchain are based on. While the other two functions of this library are situational, the NullPurge function offer a powerful tool that is invoked after exploiting most templates, compactions and identities. This function scans each qubit line in the circuit backwards, starting from the circuit's end, checking each couple of adjacent gates, detecting eventual **null operations** and purging them. In case of Pauli gates in rotational form, it combines them in a single equivalent gate. If the resulting rotation parameter is lower or equal to the **Threshold 2 parameter** read from the .cfg files, the combined gate is considered as a null operation and thus purged. Once the purge or the combination is applied, the scan on the qubit line resumes from the last gate before the gates that were optimized, and from the end of the circuit if there are none left.

- **Standard case:** For **H, X, Y, Z gates** a null operation is detected each time two gates of this type are adjacent. The gates are then purged.

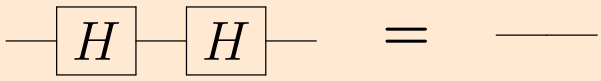


Figure 2.7: Example of standard null operation

- **S-type, T-type case:** For **S, $S^\dagger$, T, $T^\dagger$ gates** a null operation is detected each time an S-type or T-type gate is adjacent to a gate that describes the conjugate transpose of its unitary matrix (*Example:* $SS^\dagger$). The gates are then purged.

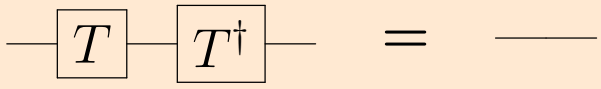


Figure 2.8: Example of null operation in the case of S-type or T-type gates

- **Pauli gates case:** For R_X, R_Y, R_Z **gates** a null operation is detected when the evaluated absolute value of the algebraic sum of the rotation parameters of two subsequent Pauli gates of the same type is equal or less than the .cfg parameter Threshold 2. If a null operation is not detected but two Pauli gates of the same type are adjacent, the gates are combined in an equivalent single gate of the aforementioned type.

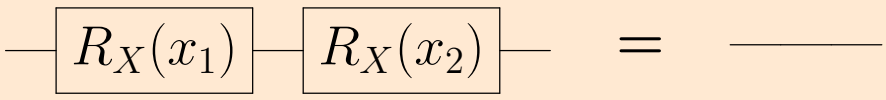


Figure 2.9: Example of null operation in the case of rotational Pauli gates, in which $|x_1 + x_2| \leq Threshold\_2$

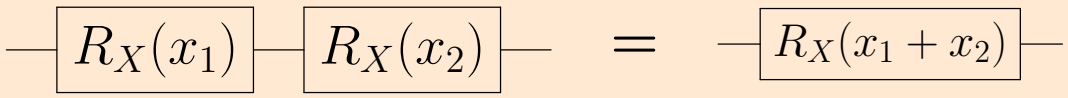


Figure 2.10: Example of adjacent gates combination in the case of rotational Pauli gates, in which $|x_1 + x_2| > Threshold\_2$

- **Two-qubit gates case:** For **CX and CZ gates** a null operation is detected whenever two of these gates are perfectly adjacent. If a null operation is detected the gates are then purged. If a "fake" null operation is detected due to the gates not being perfectly adjacent on both their qubit lines, no gate is modified to preserve the logical function of the circuit.

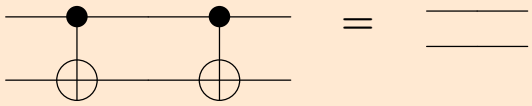


Figure 2.11: Example of null operation in the case of two-qubit gates

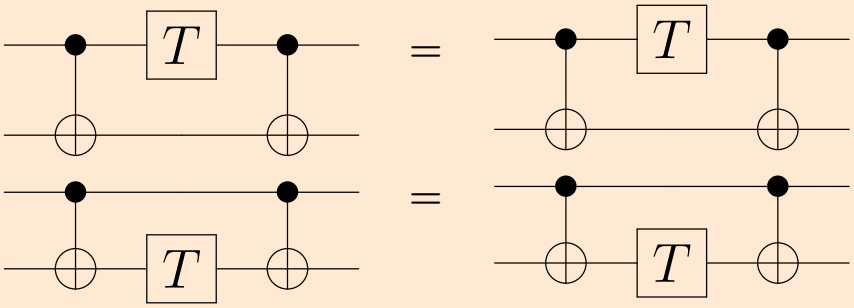


Figure 2.12: Examples of "fake"null operations in the case of two-qubit gates, which do not allow for any kind of straightforward purge

2.2.4 simple_optimizers library

Employed simple optimizations and identities

SIMPLE_EQUIVALENCES

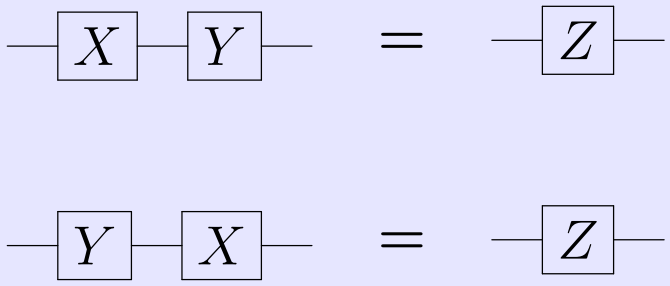


Figure 2.13: X-Y, Y-X to Z simple equivalences (considered valid because of the global phase change which results as irrelevant to the final overall state)

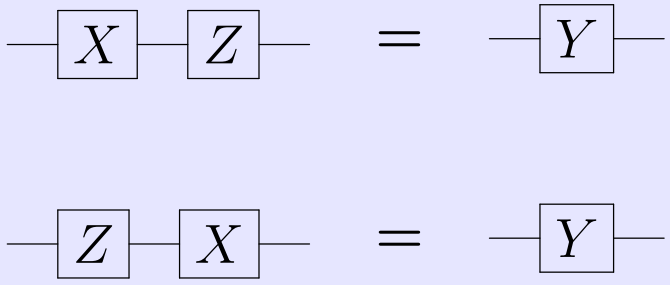


Figure 2.14: X-Z, Z-X to Y simple equivalences (considered valid because of the global phase change which results as irrelevant to the final overall state)

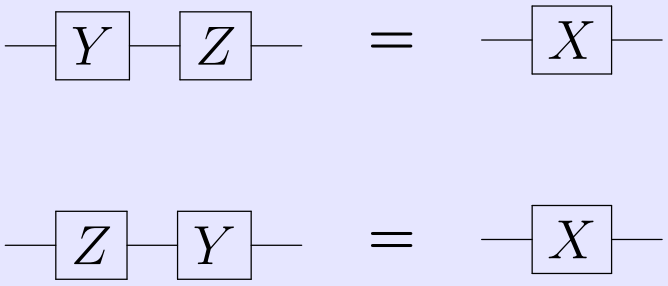


Figure 2.15: Y-Z, Z-Y to X simple equivalences (considered valid because of the global phase change which results as irrelevant to the final overall state)

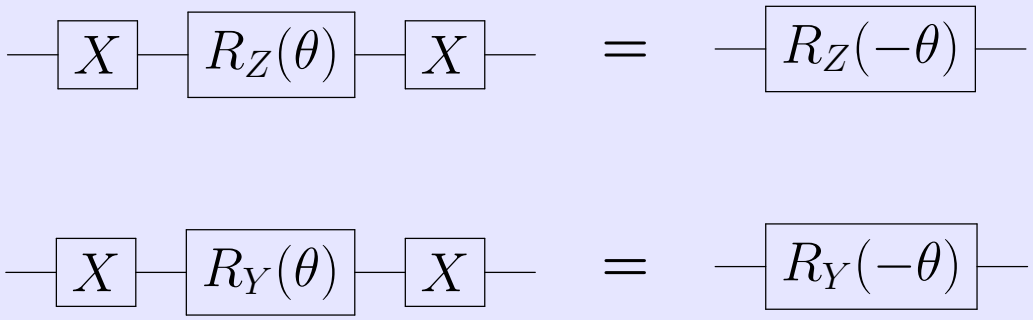


Figure 2.16: X-R_Z-X to -R_Z, X-R_Y-X to -R_Y simple equivalences

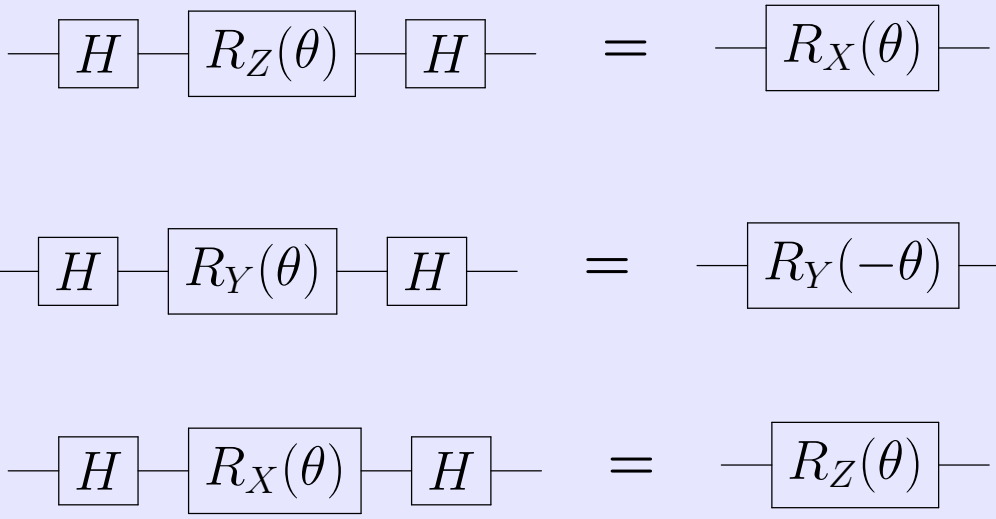


Figure 2.17: H-R_Z-H to R_X, H-R_Y-H to -R_Y, H-R_X-H to R_Z simple equivalences

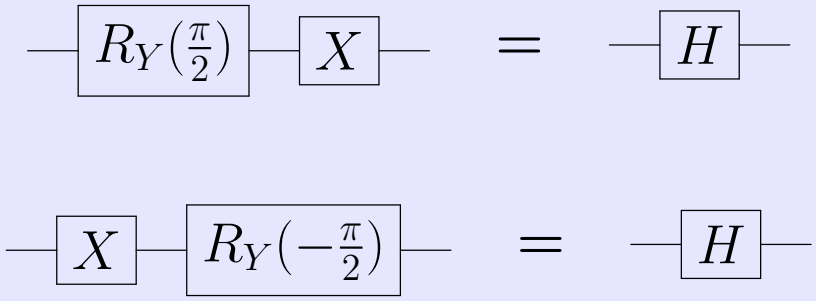


Figure 2.18: $Y^{\frac{1}{2}}$-X, X-$Y^{-\frac{1}{2}}$ to H and viceversa simple equivalences

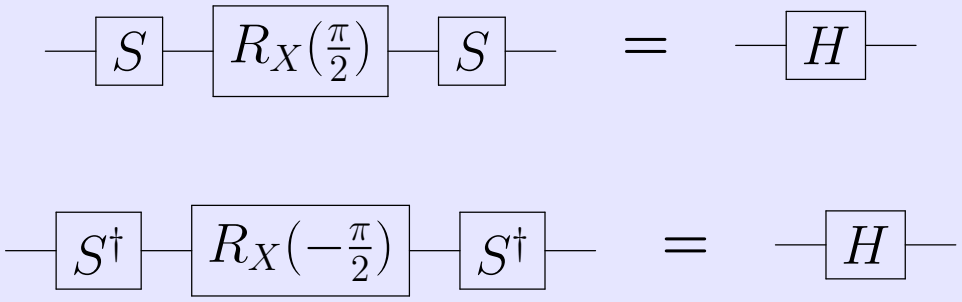


Figure 2.19: S-$X^{\frac{1}{2}}$-S, $S^\dagger$-$X^{-\frac{1}{2}}$-$S^\dagger$ to H and viceversa simple equivalences

Library description

SIMPLE_OPTIMIZERS

- **Eq_XY function:** This function scans each qubit line in the circuit backwards, starting from the circuit's end, and tries to exploit the equivalences represented in Figure 2.13 to compact each couple of adjcent X and Y gates into Z gates.

- **Eq_XZ function:** This function scans each qubit line in the circuit backwards, starting from the circuit's end, and tries to exploit the equivalences represented in Figure 2.14 to compact each couple of adjcent X and Z gates into Y gates.

- **Eq_YZ function:** This function scans each qubit line in the circuit backwards, starting from the circuit's end, and tries to exploit the equivalences represented in Figure 2.15 to compact each couple of adjcent Y and Z gates into X gates.

- **Eq_XRX function:** This function scans each qubit line in the circuit backwards, starting from the circuit's end, and tries to exploit the equivalences represented in Figure 2.16. The application of this function yields a gain which makes it always worth it.

The following functions accept a parameter that defines if step1_templates library's functions will be called after the current Simple Optimizer section:

- **Eq_HRH function:** This function scans each qubit line in the circuit backwards, starting from the circuit's end, and tries to exploit the equivalences represented in Figure 2.17.
  When **trying to compact the gates**, it checks if templates are to be called later: if so, it never transforms an H gate if it is adjacent to a CX gate in order to try to obtain a "stronger" template-based optimization, since H-based templates are particularly effective. Otherwise, it always tries to compact the circuit. If the result of this compaction is a R_Z gate, it calls the **Translator** function to translate it in an eventual "non-rotational" form.

- **Eq_H1 function:** This function scans each qubit line in the circuit backwards, starting from the circuit's end, and tries to exploit the equivalences represented in Figure 2.18.
  When **transforming gates into H gates**, it checks if templates should be called later: if so, it always transforms the couple of gates if it is adjacent to a CX gate in order to try to obtain a "stronger" template-based optimization, since H-based templates are particularly effective. Otherwise, it transforms the couple if a null operation can be achieved. If not, once all the template-based optimizations have been performed, on the last iteration it tries to compact the circuit as much as possible in order to allow the transformation of H gates into the more convenient form of Figure 2.19.
  When **expanding H gates**, it checks if templates are to be called later: if so, it never transforms the H gate if it is adjacent to a CX gate in order to try to obtain a "stronger" template-based optimization. Otherwise, it transforms the couple if a null operation can be achieved. If not, it never tries to pointlessly expand the circuit.

- **Eq_H2 function:** This function scans each qubit line in the circuit backwards, starting from the circuit's end, and tries to exploit the equivalences represented in Figure 2.19.

When **transforming gates into H gates**, it checks if templates should be called later: if so, it always transforms the triplet of gates if it is adjacent to a CX gate in order to try to obtain a "stronger" template-based optimization, since H-based templates are particularly effective. Otherwise, it transforms the triplet if a null operation can be achieved. If not, once all the template-based optimizations have been performed, on the last iteration it tries to compact the circuit as much as possible.

When **expanding H gates**, it checks if templates are to be called later: if so, it never transforms the H gate if it is adjacent to a CX gate in order to try to obtain a "stronger" template-based optimization. Otherwise, it transforms the couple if a null operation can be achieved, preferring the first form in Figure 2.19. If not, it never tries to pointlessly expand the circuit.

2.2.5 step1_templates library

Employed templates

TEMPLATES

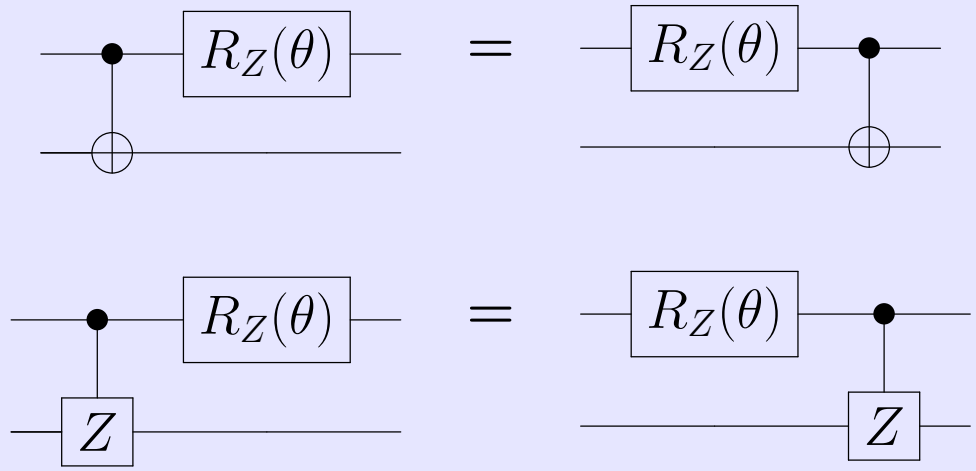


Figure 2.20: Template 1: CX(i,j)-R_Z(i) / CZ(i,j)-R_Z(i) to R_Z(i)-CX(i,j) / R_Z(i)-CZ(i,j) and viceversa

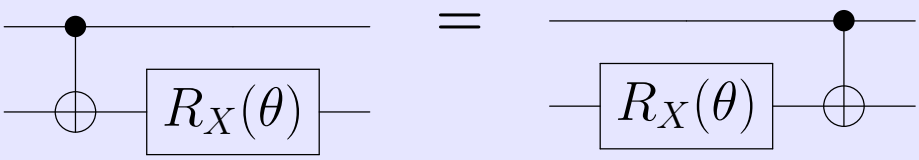


Figure 2.21: Template 2: CX(i,j)-R_X(j) to R_X(j)-CX(i,j) and viceversa

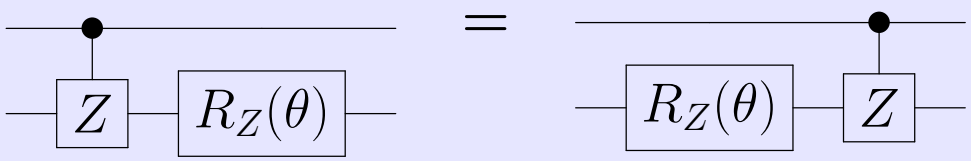


Figure 2.22: Template 3: CZ(i,j)-R_Z(j) to R_Z(j)-CZ(i,j) and viceversa

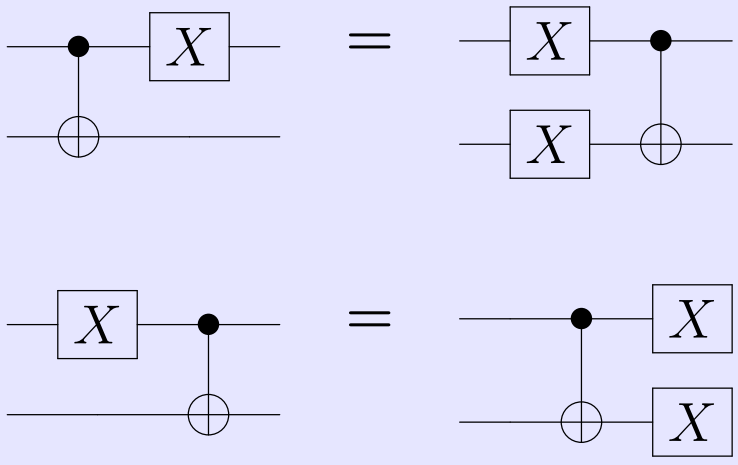


Figure 2.23: Template 4: CX(i,j)-X(i) to X(i)-X(j)-CX(i,j) / X(i)-CX(i,j) to CX(i,j)-X(i)-X(j) and viceversa

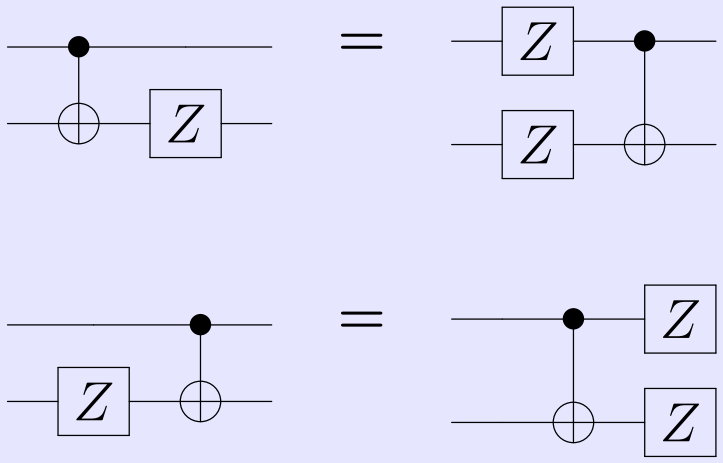


Figure 2.24: Template 5: CX(i,j)-Z(j) to Z(i)-Z(j)-CX(i,j) / Z(j)-CX(i,j) to CX(i,j)-Z(i)-Z(j) and viceversa

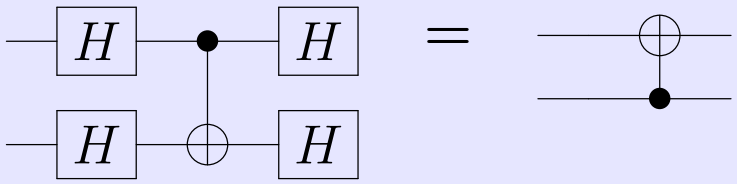


Figure 2.25: Template H1: H(i)-H(j)-CX(i,j)-H(i)-H(j) to CX(j,i)

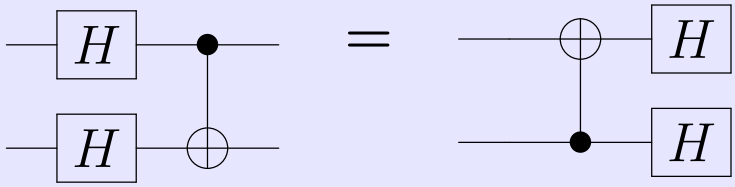


Figure 2.26: Template H2: H(i)-H(j)-CX(i,j) to CX(j,i)-H(i)-H(j) and viceversa

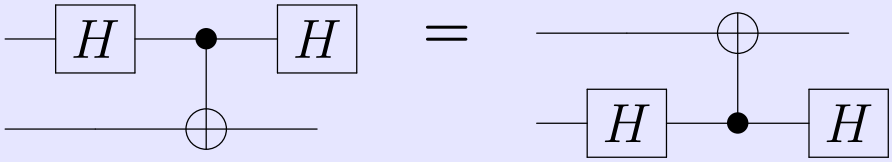


Figure 2.27: Template H3: H(i)-CX(i,j)-H(i) to H(j)-CX(j,i)-H(j) and viceversa

Library description

HELPER FUNCTIONS

These functions are employed to identify the single-qubit gates adjacent to particular two-qubits gates that are on a qubit line which is not the one that is being scanned. Overall, they are a tool used to detect if certain complex templates equivalences can actually be exploited.

- **CheckCXSwapDown function:** This function accepts as input the qubit line under scanning, the string of the target CX or CZ gate in the circuit list, and the indexes of the two gates involved in a circuital equivalence. From the two-qubit gate's string it identifies its second, non-scanned qubit, *i.e.* the control qubit or the target qubit, and checks if on such qubit line there is any gate which is between the two gates whose position in the circuit list is defined by the input indexes. This evaluation is done by scanning **from the bottom of the circuit**. If there is at least one such gate, it returns the index of the nearest adjacent gate to the CX or CZ on the non-scanned qubit line. If there is none, it returns a flag with the value "0".

The basic use of this is checking if **"simple swaps"** directed towards the end of the QC ("right direction") in the circuit list are allowed when applying a template that utilizes a two-qubit gate. In fact, by recklessly applying the "simple swaps" one may compromise the logical functionality of the circuit by "distorting" the equivalence, as represented in figure 2.28. In such cases, a more complex kind of swap is performed.

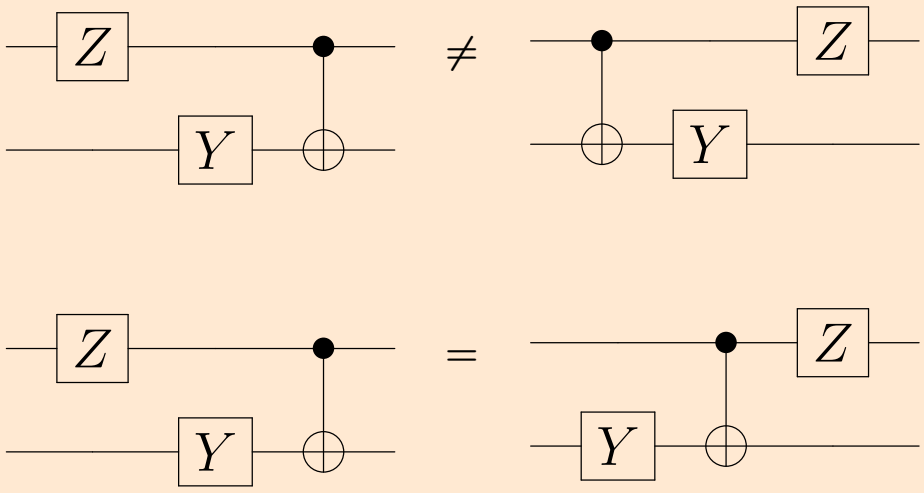
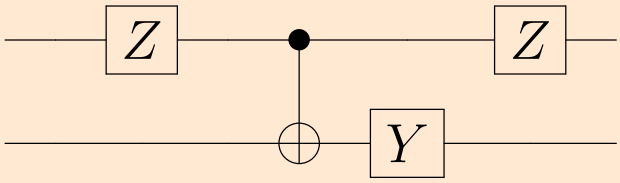


Figure 2.28: Generic example in which exploiting a template-based equivalence (the first one in Figure 2.20) can result in a "logical distortion" of the identity if elements of the circuit list are just subjected to a "simple swap"

- **CheckCXSwapUp function:** This function accepts the same inputs as the previous function, but it performs its evaluation on the non-scanned qubit by scanning **from the beginning of the circuit**. It also returns a flag with the index of the nearest adjacent gate to the CX or CZ on the non-scanned qubit line, or, if there is none, a flag with the value "0".
  The basic use of this is checking if **"simple swaps"** directed towards the beginning of the QC ("left direction") in the circuit list are allowed when applying a template that utilizes a two-qubit gate, for the same reason of performing more complex kind of swaps when required.



Example of Helper functions usage, in which the gates in the circuit are represented with the same order they have in the circuit list. The Z gates and the CX gate are capable of being exploited as a template in both directions, and the qubit line under scanning is the one above. By using the CheckCXSwapUp function with the CX and the left Z's indexes as input, the flag returned is "0": no gates are present on the non-scanned qubit, and so a "simple swap" between the CX and the left Z gate may be performed. By using the CheckCXSwapDown function with the CX and the right Z's indexes as input, the flag returned is the index of the Y gate: that very gate on the non-scanned qubit line implies that no "simple swaps" may be performed, and that a more complex swap is required.

STEP 1_TEMPLATES

- **Templ1 function:** This function scans each qubit line in the circuit backwards, starting from the circuit's end, and tries to exploit the templates represented in Figure 2.20, in one verse or another. Each time this function is called, it has two iterations. This function support both the known form and the rotational form of the R_Z gates.
  In the first iteration, it tries to "swap" the gates in order to obtain one or multiple null operations. If no null operation can be achieved, it tries to move each R_Z-type gate towards the beginning or the end of the circuit if it is not a subcircuit and then attempt an optimization through the call of **InitialZ and FinalZ functions**. Each time that a swap is required, it makes sure that the logical functionality of the circuit is maintained when applying the template.
  In the second iteration it performs the same exact operations but when no null operations can be achieved it also tries to make all possible R_Z-type gates adjacent in order to obtain a further compaction of the circuit if no gate purging was possible.

- **Templ2 function:** This function scans each qubit line in the circuit backwards, starting from the circuit's end, and tries to exploit the template represented in Figure 2.21, in one verse or another. Each time this function is called, it has two iterations. This function support both the known form and the rotational form of the R_X gates.
  In the first iteration, it tries to "swap" the gates in order to obtain one or multiple null operations.

In the second iteration it performs exactly the same operations but when no null operations can be achieved it also tries to make all possible R_X-type gates adjacent in order to obtain a further compaction of the circuit if no gate purging was possible.

- **Templ3 function:** This function scans each qubit line in the circuit backwards, starting from the circuit's end, and tries to exploit the template represented in Figure 2.22, in one verse or another. This template is actually a subcase of the second template in Figure 2.20 caused by **the symmetry of CZ gates**. Each time this function is called, it has two iterations. This function supports both the known form and the rotational form of the R_Z gates.
  In the first iteration, it tries to "swap" the gates in order to obtain one or multiple null operations. If no null operation can be achieved, it tries to move each R_Z-type gate towards the beginning or the end of the circuit if it is not a subcircuit and then attempt an optimization through the call of **InitialZ and FinalZ functions**. Each time that a swap is required, it makes sure that the logical functionality of the circuit is maintained when applying the template.
  In the second iteration it performs exactly the same operations but when no null operations can be achieved it also tries to make all possible R_Z-type gates adjacent in order to obtain a further compaction of the circuit if no gate purging was possible.

- **Templ4 function:** This function scans each qubit line in the circuit backwards, starting from the circuit's end, and tries to exploit the template represented in Figure 2.23, in one verse or another. This function support both the known form and the rotational form of the

X gates.
When **detecting the template in the form with both X gates**, it automatically tries to apply it.
When **detecting the template in the form with a single X gate**, it tries to apply it only if the insertion of a new X gate can create a null operation on the control qubit, on the target qubit or on both qubits. Otherwise, since it yields no gain, so the function does nothing. Each time that a swap is required, it makes sure that the logical functionality of the circuit is maintained when applying the template.

- **Templ5 function:** This function scans each qubit line in the circuit backwards, starting from the circuit's end, and tries to exploit the template represented in Figure 2.24, in one verse or another. This function support both the known form and the rotational form of the Z gates.
  When **detecting the template in the form with both Z gates**, it automatically tries to apply it, and it also cover the case in which the Z gate is pushed to the end or beginning of the circuit, where it can be purged through **InitialZ and FinalZ functions**
  When **detecting the template in the form with a single Z gate**, it tries to apply it only if the insertion of a new Z gate can create a null operation on the control qubit, on the target qubit or on both qubits, or, when it is not a subcircuit, if this results in both the Z gates being pushed to the end or the beginning of the circuit. Each time that a swap is required, it makes sure that the logical functionality of the circuit is maintained when applying the template.

- **TemplH1 function:** This function scans each qubit line in the circuit backwards, starting from the circuit's end, and tries to exploit the template represented in Figure 2.25. This template is the most powerful in terms of gain, and it is actually **a subcase of the TemplH2 function** which uses the template in Figure 2.26. It was separated in a standalone case because it is easy to detect and apply preemptively. Whenever this template is applicable, this function tries to exploit it.

- **TemplH2 function:** This function scans each qubit line in the circuit backwards, starting from the circuit's end, and tries to exploit the template represented in Figure 2.26. This template is very powerful in terms of gain, because optimizing H gates yields a higher benefit with respect to other single-qubit gates. This function tries to exploit the template if it can create null operations. Otherwise, it tries to apply the template whenever **two CX gates with inverted control and target qubits are adjacent**. This particular case may result in a scenario similar to the one represented in Figure 2.24, allowing an optimization if H gates are present right before the cluster in the circuit; its application is harmless otherwise. The function performs a reiterative call of itself whenever after the application of this particular case it detects a possible null operation obtainable through another **TemplH2 function** call or if it identifies another "alternated" CX gate: by doing so, the couple of H gates is moved through whole clusters of "alternated CXs", as represented in figure 2.29.

The function though makes this movement of gates possible only **backwards**, towards the beginning of the circuit, in order to avoid endless loops in particular situations.

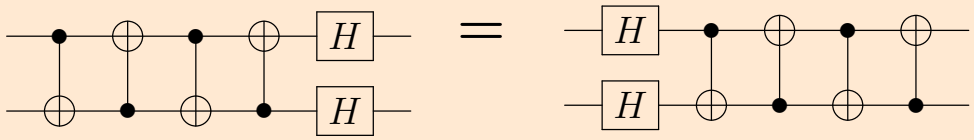


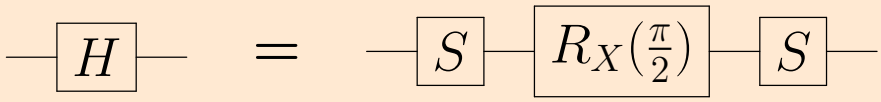
Figure 2.29: Examples of template application through a cluster of "alternated" CX gates, with H gates moved backwards

- **TemplH3 function:** This function scans each qubit line in the circuit backwards, starting from the circuit's end, and tries to exploit the template represented in Figure 2.27. This template is very powerful in terms of gain, because optimizing H gates yields a higher benefit with respect to other single-qubit gates. This function tries to exploit the template if it can create null operations. Otherwise, it tries to exploit it whenever its application results in a R_X-type gate being near to a CX gate on its target qubit, as this scenario may result in a possible optimization through the call of **Templ2 function**.

2.2.6 QASM_postcomposer library

QASM_POSTCOMPOSER

- **FinalCombo function:** This function has the important task of rebasing all the circuit using the R_X, R_Y, R_Z gates in preparation for Step 2 and of compacting in such a way that each gate becomes adjacent to gates of a different kind. To do this, it scans each qubit line in the circuit and transform every single-qubit gate which is in a known form into its rotational form with floating-point notation, and then calls in sequence all the functions in the **NULLOP_purgers library** to compact the circuit as much as possible.
  This function accepts a parameter that defines if step1_templates library's functions will be called after the current of the Simple Optimizer section. When dealing with **H gates**, it checks if templates should be called later: if not, on the last iteration it decomposes all H gates using the first equivalence in Figure 2.19 (also reported below). This equivalence is particularly advantageous for H gates, since when compared to the ones in Figure 2.18 it only employs a **single non-R_Z gate**, while the other two can be **implemented virtually** thus not encumbering the circuit.



Chapter 3

Overview of the Toolchain's Step 2

As explained in Section 1.4, the required output of the Logic Synthesis step in the state-of-the-art quantum design toolchain must be, as it is described in [17], "a sequence of gates/operations supported by the architecture". The aim of this chapter is to provide an in-depth overview of the Toolchain's Step 2, which is the step delegated to tackle this particular task and finalize the Logic Synthesis in the optimization process. A description of the exact purpose of this step is reported, followed by a summary of the conceptual blocks that define its structure and of their inner workings. After that, the exact sequence of operations performed for each target technology is reported, along with some explanations about the design choices made in the implementation of each technology-specific workflow. Finally, brief descriptions of each of the functions that compose the related Python libraries are provided.

3.1 Step 2 - Technology-dependent gates compaction

As stated before in Section 2.1.2, in this step of the Toolchain the translation process of each gate in the input quantum circuit into the **NMR, Trapped Ions and Superconducting technologies'** own set of gates is performed. This step is designed to take as inputs the optimized circuit .qasm files generated by Step 1 to work at maximum efficiency, but it can also be used on custom, unoptimized .qasm files. The translation process, which is applied universally to single-qubit gates, does not implement a decomposition of two-qubit gates such as the CX gates, which are left untouched in order to be better exploited in Step 3, where they will eventually be decomposed into their basic constituting gates. As explained in Section 4.1, this procrastination is due to the fact that all the templates that involve multiple two-qubit gates and all Layout Synthesis operations in general can be detected and performed much more easily with non-decomposed CX and CZ gates.
Clearly, to perform these operations, Step 2 forsake the technology-agnosticism of Step 1 to employ a **technology-specific approach**, taking in consideration only the chosen target technology and its inherent optimization process. Generally speaking, the workflow of this step is:

- Smartly disposing of **CZ gates** in such a way that their conversion is made as efficiently as possible according to the target technology's specific needs.
- Employing a peculiar manipulation on triplets of adjacent Pauli gates of different type to achieve a further compaction of the circuit and to increase the number of $\boldsymbol{R_Z}$ **gates** when possible. This situational manipulation is quite powerful, and it is based on coordinate transformations using *Euler angles*.
- Translating each single-qubit gate into one of the target technology's base set of gates following technology-specific criteria.
- If it is possible, trying to compact the resulting gates as much as possible.

While Step 2's role in the optimization process is fundamental in order to obtain quantum circuits that are tailored to a specific implementation technology, its optimizations are mostly situational, when compared to Step 1's powerful set of circuit improvements. Optimization-wise, Step 2 introduces a "consolidation" of the reduction in the single-qubit gates' number, and it is particularly effective in dealing with long streaks of these kind of gates uninterrumpted by two-qubit gates, streaks that, as seen in the results in Chapter 5, are usually quite rare in layered quantum circuits, especially after the manipulations performed in Step 1.

As of now, Step 2 supports the usage of $\boldsymbol{R_X}$, $\boldsymbol{R_Y}$, $\boldsymbol{R_Z}$, **CX and CZ gates** in the input quantum circuit and requires **all single-qubit gates to be adjacent to gates of different type**. These are condition easily achieved in Step 1 through the translation and compaction resulting from the usage of the FinalCombo function (as explained in Section 2.2.6), but in case of usage of an input quantum circuit which was not previously optimized through Step 1 they have to be enforced by the user. Once again, the modular structure of the libraries allows for an expansion to support other quantum technology with their specific workflows.

Step 2 requires two inputs when executed from shell: the **input .qasm circuit file**, which describes the reference quantum circuit that has to be optimized, and the **Subcircuit parameter**, which is a boolean flag that defines if the circuit is indeed a *subcircuit* to be used in conjunction with other QASM-described entities and thus if certain optimizations regarding R_Z gates can be employed. Once the script is started, it asks the user for a parameter to define **the target technology**: **"M"** for the **NMR technology**, **"I"** for the **Trapped Ions technology** and **"S"** for the **Superconducting technology**.

All the manipulations involving *Euler angles* are performed through the usage of the **SciPy Python library** [52] and of the **NumPy Python library** [53]. All the benchmarking procedures, the results and their related analysis are reported in Chapter 5 in the appropriate section.

3.1.1 Step 2's structure

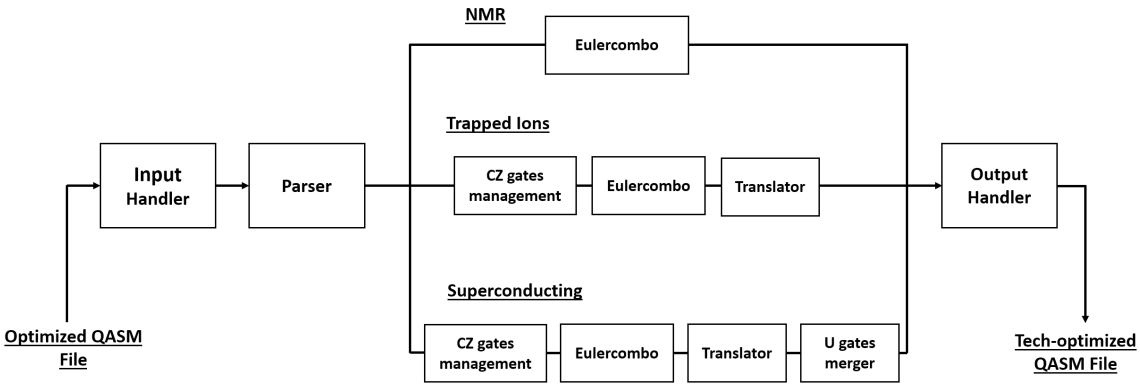


Figure 3.1: Representation of the overall structure of the Toolchain's Step 2 (assuming that the input circuit was previously optimized by Step 1)

Step 2 is implemented with the **technology_optimizer_2 script**, and depending on the target technology calls the **NMRLib, IonsLib or SupercondLib functions.** The script is divided in the following sections:

1. **Input Handler section:** This section handles the correct reception of inputs and provides error messages if a wrong number of parameters or a wrong file extension are passed as inputs. It also reads some of the .cfg files described in 2.1.2 to define the thresholds and approximation accuracy to be used in the optimization process and to set the translation parameter **Iontran** in the Trapped Ions technology case. For what concerns the thresholds, the script provides an error message if an approximation accuracy equal or greater than 1 is provided and a warning message if a null operation identifier threshold is defined, since this could bring to "overzealous" optimizations that could compromise the logic functionality of the circuit. If a non-integer number is passed for the iterative parameters it proceeds to round it, and if a number which is less than 1 is set in the .cfg files for these loops, it automatically sets it to "1".

2. **Parser section:** The **Parser** section generates the circuit list on which the script will work based on the input file and rearranges it, "cleaning" it from

endline spaces, blank lines, barriers and comments. It also identifies the number of qubits involved in the target circuit and saves all the measurements performed in the input file on a temporary list, which will be appended to the output circuit list once the optimization is completed.

3. **CZ Management section (Trapped Ions, Superconducting):** This section performs a smart translation of **CZ gates into CX gates** in the technologies where they are not natively supported by exploiting the equivalence represented in Figure 3.2. Since this transformation implies the insertion of **H gates** in the circuit, the functions of this section are programmed to account for the **symmetry propriety** of CZ gates and try to achieve some optimizations by generating the most convenient null operations. To do this, a "score system" is employed, as described in Section 3.1.5. If no null operation can be achieved, they simply insert the H gates using their most convenient decomposed form represented in Figure 2.19. The template used to translate CZ gates into CX gates was extrapolated from [8, 50].

4. **Eulercombo section (NMR, Trapped Ions, Superconducting):** This section tries to achieve a further single-qubit gates compaction through the usage of coordinate transformations applied on triplets of different Pauli gates. These transformations are based on the principle that each arbitrary rotation in the Bloch Sphere described by a triplet of single-qubit gates and their rotation parameters (*i.e.* their **Euler angles**) can be perfectly described by another triplet of different gates with different Euler angles obtained through a transformation of the three-dimensional coordinates [54]. These angles are calculated through the multiplication by a matrix performed by the SciPy library, and are the only parameters in the toolchain which have to be mathematically evaluated and cannot be extracted from a database of templates - since the sheer number of possibilities would be simply too big to handle. By performing specific coordinate transformation, it is possible to make so that the external gates in the triplets are adjacent to Pauli gates of the same type, thus allowing for a compaction. If the detected triplets are not adjacent to any other single-qubit gate, this section uses the same method to transform them in a form which maximizes the usage of virtually-implementable R_Z gates.

A more in-depth description of the mechanism around which the Eulercombo section revolves is presented in 3.1.5.

5. **Translator section (Trapped Ions, Superconducting):** This is the section in which the effective translation into the technology-specific gate set is performed. In the case of the Trapped Ions technology, this consists in an optional, sheer translation process, while in the Superconducting case a smarter and more refined approach is used to maximize the usage of IBM's native gate set's optimal U gates.

6. **U gates Merger section (Superconducting):** This section is uniquely dedicated to the Superconducting technology, and it performs the efficient optimizing scheme proposed in [55] in order to merge single-qubit gates to the extreme. This section uses functions from a dedicated library, the **Ugates_converter library**.

7. **Output Handler section:** This section uses the final circuit list to generate a .qasm file in the working directory. If the input file is detected as an output of Step 1 through the parsing of its name, the suffix (_optimized) is replaced. Otherwise, a new suffix is simply appended to the original file name. In either case, the suffix "_X_techoptimized" in which X is the input parameter describing the target technology ("M", "I", "S") is added, thus making explicit in the file name the chosen target technology. Then, it appends to the file the measurements contained in the original circuit (if it is not a subcircuit, of course) and provides a message announcing that the optimizations were completed successfully and the name of the generated file.

3.1.2 NMR - Specific workflow

- **CZ gates:** CZ gates are actually supported in NMR technology. In fact, they are the most convenient way to implement two-qubit gates, as explained in Section 1.3.2. Thus, instead of a transformation from CZs to CXs, it would be preferable a transformation the other way around, from CXs to CZs. However, most template-based optimizations applicable in the Layout Synthesis block are based on CX gates and not on CZ gates. To leave the possibility of exploiting these templates, present CX gates are left untouched, but since CZ gates are preferable in this technology they are left untouched as well. Basically, **no conversion is performed in either case**, and the translation into a single type of two-qubit gate is performed later in Step 3, as explained in Chapter 4.

- **Translation:** The base gates of the **NMR technology** are the $\boldsymbol{R_X}$**,** $\boldsymbol{R_Y}$**,** $\boldsymbol{R_Z}$**, CX and CZ gates**. This gate set is already required for Step 2 to work and its usage is ensured in each circuit optimized by the Toolchain's Step 1, so **no translation is necessary**.

- **Special functions:** Since CZ gates, like R_Z gates, can be neglected when at the very end or very beginning of the circuit, an *ad-hoc* version of FinalZ and InitialZ functions (**SpecialFinalZ and SpecialInitialZ**) are called to try to purge every redundant CZ once all other optimizations are applied.

- **Function calls: (contained in the NMRLib function)**

  - *Eulercombo function*
  - *EulerZYZ function*
  - *SpecialFinalZ function*
  - *SpecialInitialZ function*

3.1.3 Trapped Ions - Specific workflow

- **CZ gates:** As CZ gates are not supported by this technology, they are translated into CX gates using the equivalence represented in Figure 3.2. This is done optimally by the *ad-hoc* functions described in Section 3.1.5.

- **Translation:** As explained in Section 1.3.3, the Trapped Ions technology usually supports the implementation of single-qubit gates through the usage of the $\mathbf{R(\theta,\phi)}$ generic rotation gate. Since this notation may not always be preferred, as for example in benchmarking operations, the translation of the $\boldsymbol{R_X}$ and $\boldsymbol{R_Y}$ gates in the $\mathbf{R(\theta,\phi)}$ gate is performed only if the **Iontran** boolean parameter in the **ion_translations.cfg** file is set as true ("T"). Otherwise, the gates are left untouched. $\boldsymbol{R_Z}$ **gates** are left in their original form in order to be clearly identifiable and to make the **virtual implementation** process easier. This choice was made because recently virtual implementations of these gates became available for Trapped Ions-based devices ([11, 12]) and because in this way the Toolchain's philosophy of maximizing the number of R_Z gates is matched. If one were to decide to make the translation compatible with non-virtual implementations, it could be easily changed to be capable of replacing each R_Z gate with an equivalent combination of R_X and R_Y gates.

- **Function calls: (contained in the IonsLib function)**

  - *CZReadjust function*
  - *CZtoCX function*
  - *Eulercombo function*
  - *EulerZYZ function*
  - *If required: IonTranslator function*

3.1.4 Superconducting - Specific workflow

- **CZ gates:** As CZ gates are not supported by this technology, they are translated into CX gates using the equivalence represented in Figure 3.2. This is done optimally by the *ad-hoc* functions described in Section 3.1.5.

- **Translation:** As explained in Section 1.3.4, the Superconducting technology supports IBM's native gate set of **U1, U2, U3 and CX gates**, so a translation of single-qubit gates is required. Since **the usage of U1 and U2 gates is preferable** with respect to U3 gates because it is more convenient in terms of gate latency, this translation is performed in order to maximize their number in the circuit. This is done by using the **Ugates_converter library**, whose functions translate detected H gates into U2 gates and then, after the Eulercombo optimizations, try to resort to U3 implementations only in the cases in which U1 and U2 gates are not employable. These U gates are then merged by using the same mechanism on which the Eulercombo optimizations are based to implement the scheme proposed in [55] and compact the circuit as much as possible.

- **Function calls: (contained in the SupercondLib function)**

  - *CZReadjust function*
  - *CZtoCX function*
  - *H_to_U2 function*
  - *Eulercombo function*
  - *EulerZYZ function*
  - *ZRZ_to_U2 function*
  - *HalfR_to_U2 function*
  - *ZRZ_to_U3 function*
  - *GenericConverter function*
  - *SupercondMerge function*

3.1.5 step2_techlib library

STEP 2_TECHLIB - CZ MANAGEMENT

CZ gates can be transformed into CX gates by using the following equivalence:

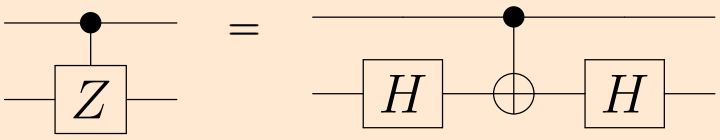
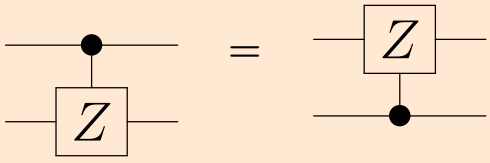


Figure 3.2: Template to transform a CZ gate into a CX gate through the insertion of H gates

However, CZ gates are **symmetrical**, and thus a single CZ when transformed can actually result in **two** different templates with a CX:

Because



both equivalences are true:

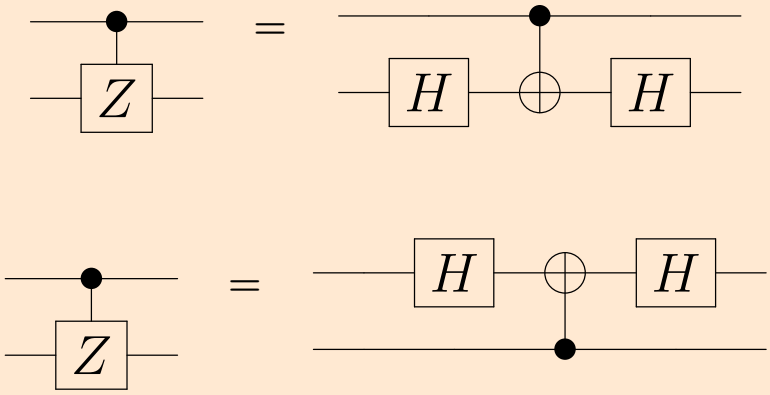


Figure 3.3: Double equivalence when transforming CZ gates into CX gates due to CZs' symmetry

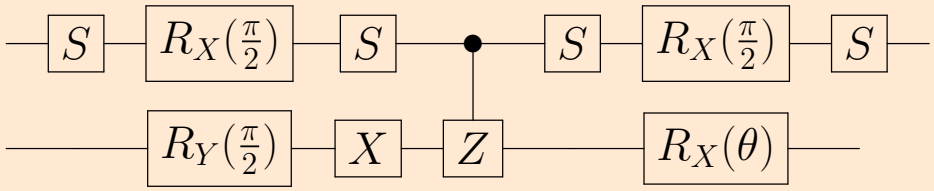
To transform CZ gates into CX gates in an optimal way, both templates must be taken into account to insert the H gates in the qubit line where they yield the greatest benefits in terms of circuit compaction, because purging an H gate is a particularly strong optimization for single-qubit gates. To do this, two functions are employed.
The **CZReadjust** function evaluates which qubit line is more suitable for the H gates insertion through the "score system" described below and shifts the CZ in order to have its target qubit on the aforementioned qubit line.
Once this is done, the **CZtoCX** function simply applies the correct template by inserting the required H gates in the qubit line where the CZ's target qubit is located.
The H gates are inserted in a decomposed form, which is determined by the compactable gates adjacent to the original CZ.

- **CZReadjust function:** This function performs a preemptive translation of all eligible single-qubit gates into known forms of Pauli gates by invoking the **Translator function**. Then, it scans each qubit line in the circuit backwards, starting from the circuit's end, and identifies all CZ gates. As first thing, the function checks if the CZ's control qubit is adjacent to decomposed H gates and the target qubits is not: if so, it inverts the CZ gate. Viceversa, if the CZ's target qubit is adjacent to decomposed H gates and the control qubit is not, the function identifies a best-case scenario and does nothing. If both qubit lines present potentially compactable single-qubit gates as adjacent gates, both sides on the qubit lines are checked and the following "score system" is employed, with the target qubit line and the control qubit line having separate scores:

  - **H gates in the R_Y-X / X-R_X form** score **3 points each**. This is due to the fact that purging a decomposed H gate is a strong optimization, and that purging two non-R_Z gates yields the most gain in the circuit.
  - **H gates in the S-R_X-S / $S^\dagger$-R_X-$S^\dagger$ form** score **2 points each**.
  - **Non-H gates** which can be compacted with a decomposed H gate score **0.5 points each**. This is due to the fact that when no H gate is available on either side it is still optimal to try to compact the circuit as much as possible, but the presence of these single-qubit gates should never compromise the evaluation of present H gates.

  At the end of the evaluation, the scores of the target qubit line and of the control qubit lines are compared. If the target score is greater, nothing is done. If the control score is greater, the CZ is inverted. When there's a draw, it is useless to take any action and so nothing is done.

For instance:



In this example, the target qubit line presents a non-ideal decomposed H gate on the left (3 points) and a compactable non-H gate on the right (0.5 points). The control qubit line presents an ideal decomposed H gate (2 points) on both sides. The target qubit line's total score is **3.5**, the control qubit line's total score is **4**. Since the control qubit line "won", it is more convenient to **invert the CZ gate**.

- **CZtoCX function:** This function scans each qubit line in the circuit backwards, starting from the circuit's end, identifies all CZ gates and then translates them into CXs. Since after the call of the **CZReadjust function** each CZ is already placed in order to have the target qubit on the qubit line in which an insertion of H gates is most convenient, this function simply transforms the CZ into a CX using the template in Figure 3.2 by ensuring that the CX's target qubit match the CZ's one, in order to effectively insert the H gates where they are most beneficial.

The decomposed form of the H gates to use is evaluated for both sides by identifying the adjacent gates. Then, when two or more single-qubit gates of the same type are adjacent, an optimization is performed through the **NullPurge, InitialZ and FinalZ functions**. Once the purge or the combination is applied, the scan on the qubit line resumes from the last gate before the gates that were optimized, and from the end of the circuit if there are none left.

STEP 2_TECHLIB - EULERCOMBO

When using these functions with a very accurate approximation of π, it is possible to incur in the problem known as *Gimbal Lock* [56]. This involves no issues when compiling the code nor when manipulating the circuit list, but can severely increase the computation time required. For a more in-depth explanation on the mathematical mechanism on which these functions are based, refer to [54].

- **Eulercombo function:** This function scans each qubit line in the circuit backwards, starting from the circuit's end, and identifies triplets of consecutive single-qubit gates. Once a triplet is detected, it identifies its adjacent gates and evaluates which side is suitable to allow a combination (*i.e.* which side has no CX, CZ, RXX, RZZ or no gate adjacent to the triplet). Then, it applies the most convenient coordinate transformations to manipulate the triplet in a way to change its external gates in the same type of their adjacent gates, in order to allow a compaction. When a template is used, the function automatically detects what two-qubit gates are involved (**CXs, CZs, RXXs or RZZs**) and calls only the optimization functions that can be exploited on those particular gates.

These manipulations consist in a change from an **"ABC" triplet**, in which each letter defines a rotation around the X, Y or Z axis and in which each gate describing a rotation is adjacent to another single-qubit gate of different type (a triplet in an "AAC" form for instance could not be transformed), to an **"UVW" triplet**, whose rotations are determined on the principle of compacting the circuit as much as possible and are different from the adjacent rotations in the triplet. After the manipulation, the function also purges all the gates in the triplet with a rotation parameter of "0", because they are null operation: this is done in order to avoid the insertion of redundant gates when a Gimbal Lock happens.

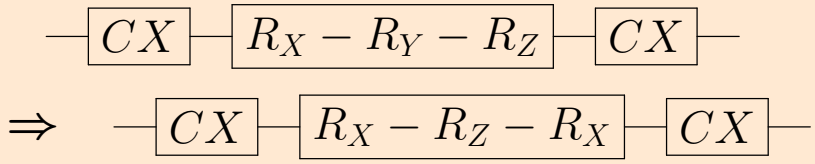
To achieve this, the following criteria are adopted:

- Try to maximize R_Z gates for optimal virtual implementation.
- When both the gates adjacent to the triplet are R_Z gates or whenever applicable, prefer a **Z-Y-Z form**. This is ideal both because it maximizes the number of involved R_Z gates and because it makes translation into U gates easier in the Superconducting technology.
- Whenever a side is not suitable for combination (*i.e.* a CX, CZ, RXX or RZZ gate is adjacent to the triplet), try to transform its adjacent external gate of the triplet in a $\boldsymbol{R_X}$ **gate** in order to exploit the advantageous template in Figure 2.21 and 4.9 as much as possible. If not possible, transform it in a $\boldsymbol{R_Z}$ **gate** to exploit the templates in Figure 2.20, 2.22 and 4.8 instead (which are considered as less advantageous because they optimize R_Z gates, which, if virtually implemented, do not hamper the circuit). It must be noted that the exploitation of these three last templates is not the focus of this section. In fact, they are employed only when a combination of single-qubit gates is already possible and no template

involving R_X gates can be exploited, as a "bonus". Indeed, at this stage of optimizations, R_Z gates are considered "harmless" because of their virtual implementation.

By labeling as **"C"** an external gate in the triplet whose rotation is of the same type of a gate adjacent to the triplet (and that thus allows a combination) and as **"G"** a generic gate in the middle of the triplet which is of different type with respect to the other two external gates, the following cases arise:

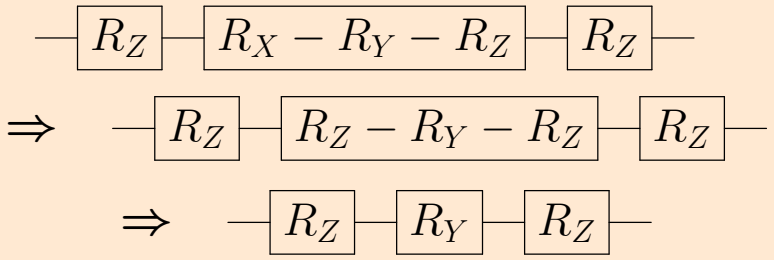
– **Both sides not suitable for combination:** if both sides feature a **CX or RXX gate**, then the triplet is transformed into the **X-Z-X form**. If not, no action is performed on the triplet (as discussed in the adopted criteria, when both adjacent gates are **CZ or RZZ gates** no effort is made to try to exploit their correlated templates, because of the harmlessness of R_Z gates). For example:



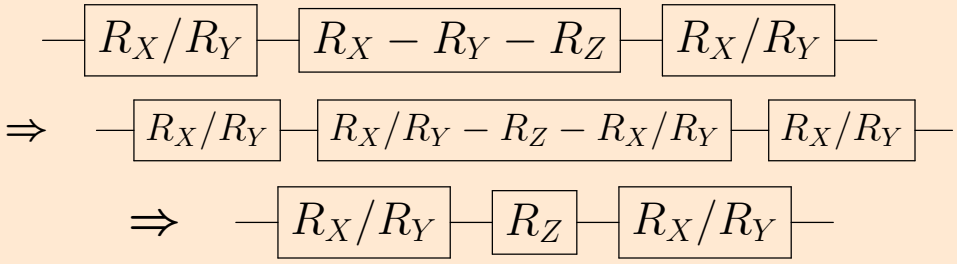
Both sides are not suitable for combination: a X-Z-X form is applied to try to exploit some advantageous templates that involve CX and R_X gates.

– **Both sides suitable for combination:** the triplet is preferably transformed into the **Z-Y-Z form**. If it is evaluated that more combinations are achievable by not using a couple of external R_Z gates, the triplet is transformed into the **C-Z-C form**. If not

possible because two R_Z gates would be adjacent in the triplet, the generic **C-G-C form** is applied. For example:



Both sides are suitable for combination (best case scenario): since using a Z-Y-Z form actually yields the highest number possible of combinations, it is preferred above all other forms.

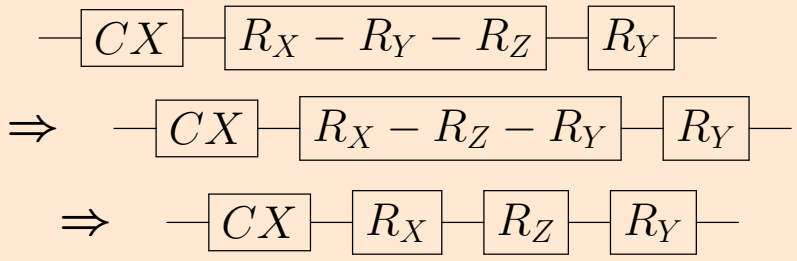


Both sides are suitable for combination: A C-Z-C is used to obtain the highest number possible of combinations. The usage of a central R_Z gate is preferred since none of the other two external gates is of the same type.

– **At least one side suitable for combination:** the triplet is preferably transformed into the **C-Z-X or X-Z-C form**, in order to allow a compaction of gates on the suitable side and to try to exploit the template in Figure 2.21 or 4.8 on the other.

There are cases in which the adjacent gate to the triplet is a R_Z gate and thus the usage of these two forms would not allow both the possibility to exploit a template that is considered "interesting", *i.e.* one that involves CX and RXX gates, and a combination on the suitable side, because that would result in a triplet containing two adjacent internal R_Z gates. In those cases the triplet is transformed into the **C-X-Z or Z-X-C form**, in order to compact the gates on the suitable side and to try to exploit the templates in Figure 2.20, 2.22 and 4.8 on the other at the same time, as a "bonus" effort alongside the combination on the suitable side. Also, whenever the side not suitable for combination coincides with the circuit's end or beginning, a **C-Y-Z or Z-Y-C form** is adopted to try to place a R_Z gate at the end or beginning of the circuit, where it can be purged. When optimizing, the function calls the related functions from the **step1_templates library** and from the **Special Functions section of this library** according to the detected exploitable template, and the **InitialZ and FinalZ functions** when R_Z gates might be at the beginning or the end of the circuit.

For example:



Left side not suitable for combination: A X-Z-C is used to obtain both the combination of the rightmost adjacent R_Y gate and the chance to exploit the CX + R_X template on the left side.

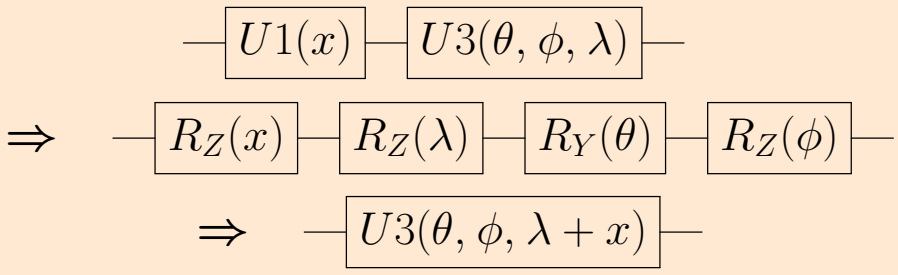
- **EulerZYZ function:** Once all existing triplets are compacted as much as possible in a single one by the **Eulercombo function call**, or if there are triplets which could not be compacted any further or partially employed in a template, this function detects them by scanning each qubit line in the circuit backwards, starting from the circuit's end. Once this is done, it transforms them in the optimal **Z-Y-Z form**. After that, the function also purges all the gates in the triplet with a rotation parameter of "0", because they are null operation: this is done in order to avoid the insertion of redundant gates when a Gimbal Lock happens. Finally, it tries a last optimization effort by calling the **InitialZ and FinalZ functions**.

STEP 2_TECHLIB - TRANSLATION

- **IonTranslator function:** This function is called only when the **Iontran parameter is set to "true"**. When this happens, it scans the circuit and translates each $\boldsymbol{R_X}$ and $\boldsymbol{R_Y}$ gate in an equivalent $\mathbf{R(\theta,\phi)}$ gate. Of course, to ensure correctness with decomposed two-qubit gates all **RXX** gates are detected and left untouched in the translation process.

STEP 2_TECHLIB - U GATES MERGE

- **SupercondMerge function:** This function scans each qubit line in the circuit backwards, starting from the circuit's end, and identifies each U gate in the circuit. The merging mechanism is based on the U gates equivalences of IBM's native gate set described in Section 1.3.4 and represented in Figure 1.9, 1.10, 1.11. By exploiting these equivalences, U gates are merged into a single U gate.
  Merging U1 gates with other U gates is trivial, because it is performed by merging two R_Z gates:



When combining U2 and U3 gates between them or with each other, things are more complicated. By employing the optimization scheme proposed in [55], by applying an *ad-hoc* coordinate transformation in the central triplet of gates and by considering each U2 as a particular case of U3, it is possible to solve each **generic U3 + U3 case and merge the gates into a single U3** by exploiting a **Y-Z-Y form to Z-Y-Z form transformation**:

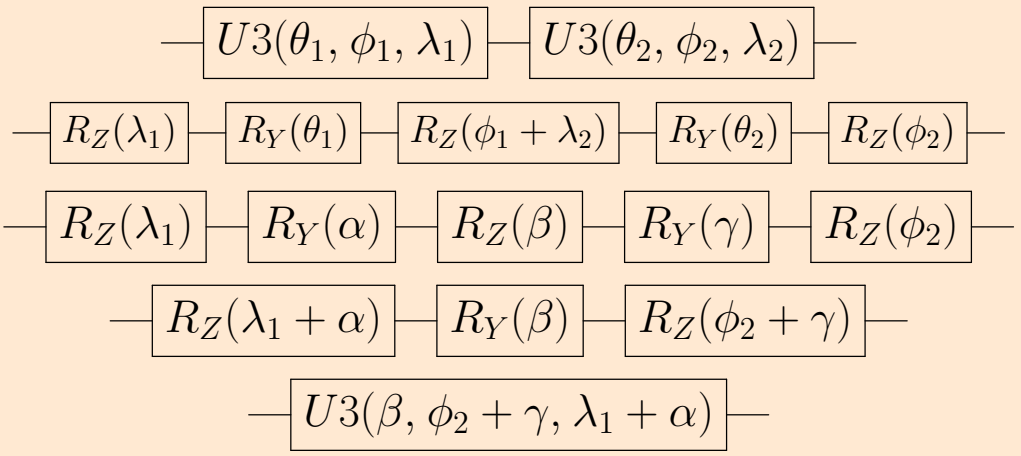


Figure 3.4: Generic U3 merging scheme into a single U3 gate by exploiting equivalences, as proposed in [55]

STEP 2_TECHLIB - SPECIAL FUNCTIONS

- **SpecialFinalZ function:** This function act as an *ad-hoc* version of the **FinalZ function** described in Section 2.2.3 to purge all **CZ and RZZ gates** at the very beginning of the circuit. This function scans each qubit line in the circuit backwards, starting from the circuit's end, and checks if the **last gate** is a CZ or RZZ: if it is not, it proceeds with the next qubit line; if it is, it checks the other of the two qubits involved in the gate and detects if the gate is at the very end of the circuit on both qubit lines. If so, it purges it and resumes from the new circuit's end. Otherwise, it proceeds with the next qubit line. This optimization exploit the fact that, like R_Z gates, **CZ and RZZ gates right before measurement are negligible**, since they only insert a phase variation. If the circuit is a Subcircuit and thus it is not followed by measurement operations, this functions does nothing.

- **SpecialInitiallZ function:** This function act as an *ad-hoc* version of the **InitialZ function** described in Section 2.2.3 to purge all **CZ and RZZ gates** at the very beginning of the circuit. This function scans each qubit line in the circuit forwards, starting from the circuit's beginning, and checks if the **first gate** is a CZ or RZZ: if it is not, it proceeds with the next qubit line; if it is, it checks the other of the two qubits involved in the gate and detects if the gate is at the very beginning of the circuit on both qubit lines. If so, it purges it and resumes from the new circuit's start. Otherwise, it proceeds with the next qubit line.

This optimization exploit the fact that, like R_Z gates, **CZ and RZZ gates right after qubit initialization to $|0\rangle$ are negligible**, since they do not insert any phase variation. If the circuit is a Subcircuit and thus it is not followed by measurement operations, this functions does nothing.

- **NMRLib, IonsLib, SupercondLib:** Wrappers that call the detailed workflow reported in Sections 3.1.2, 3.1.3 and 3.1.4 for the **NMR, Trapped Ions and Superconducting technology** respectively. These are the functions called in the **technology_optimizer_2 script** that implements Step 2.

3.1.6 Ugates_converter library

U GATES_CONVERTER

- **H_to_U2 function:** This function scans each qubit line in the circuit backwards, starting from the circuit's end, and detects decomposed H gates both in ideal and non-ideal form. It then translates them in **U2(0, π) gates** by exploiting the equivalence in Figure 3.5. This function is called before eventual Eulercombo optimizations to ensure H gates are transformed into U2 right away, instead of risking less optimal U3 implementations after triplets manipulations.

Figure 3.5: H to U2 equivalence

- **ZRZ_to_U2 function:** This function scans each qubit line in the circuit backwards, starting from the circuit's end, and tries to exploit the easily detectable equivalences represented in Figure 3.6 to maximize the number of employed U2 gates.

The R_Z-R_X-R_Z case is actually unused thanks to the usage of **EulerZYZ function**, but was still added for completion.

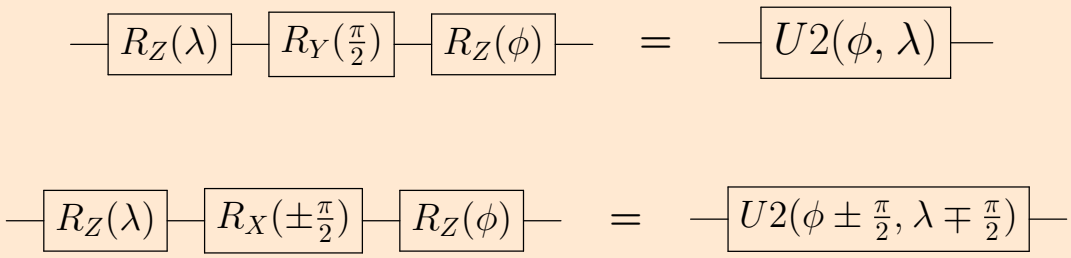


Figure 3.6: R_Z-R_X/R_Y-R_Z to U2 equivalences

- **HalfR_to_U2 function:** This function scans each qubit line in the circuit backwards, starting from the circuit's end, and tries to exploit the easily detectable equivalences represented in Figure 3.7 to maximize the number of employed U2 gates.

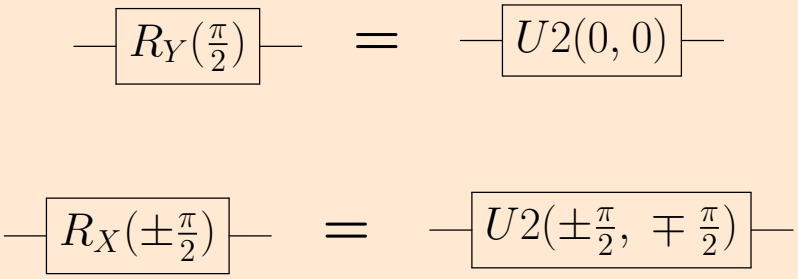


Figure 3.7: $R_X(\pm\frac{\pi}{2})$/$R_Y(\frac{\pi}{2})$ to U2 equivalences

- **ZRZ_to_U3 function:** This function scans each qubit line in the circuit backwards, starting from the circuit's end, and tries to exploit the easily detectable equivalences represented in Figure 3.8 to translate the gates into U3s right away. The R_Z-R_X-R_Z case is actually unused thanks to the usage of **EulerZYZ function**, but was still added for completion.

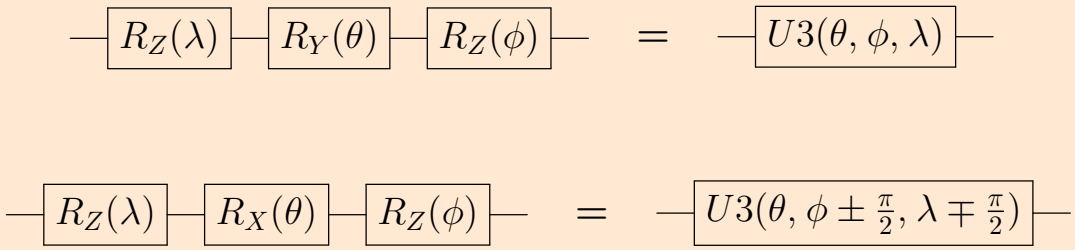


Figure 3.8: Easily detectable U3 equivalences

- **GenericConverter function:** This function comes into play when all existing equivalences for U2s have already been exploited, and when all easily detectable equivalences for U3s have already been identified and translated. The function scans each qubit line in the circuit backwards, starting from the circuit's end, and translates **all remaining R_Z gates into U1 gates** and **all remaining R_X, R_Y gates into U3 gates** by using the equivalences represented in Figure 3.9

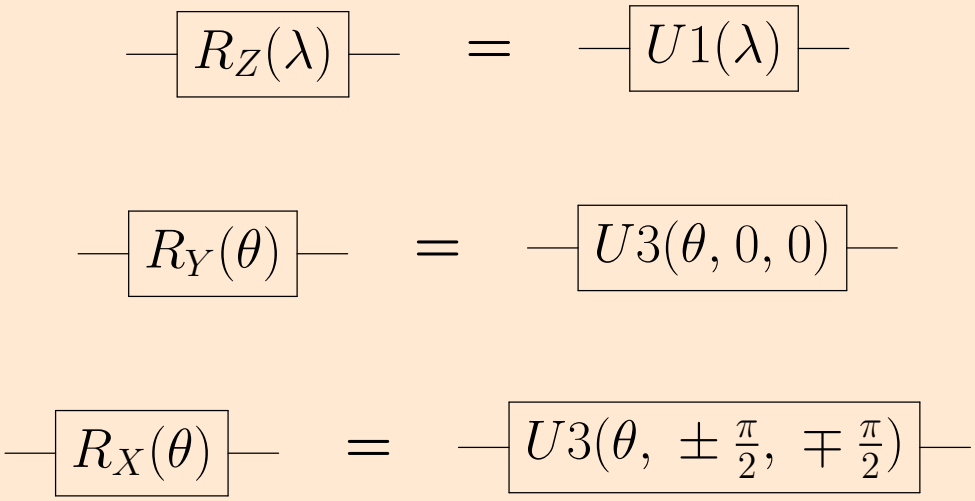


Figure 3.9: Generic R_X, R_Y, R_Z to U1, U3 equivalences

Chapter 4

Overview of the Toolchain's Step 3

The final step of the Toolchain manages the most complex operations involving templates. In fact, it takes care of a part of the Layout Synthesis block described in Section 1.4. At the time of writing this thesis, it does not implement a proper routing and mapping mechanism to support proper adaptation to quantum devices in non-fully-connected technologies (*i.e.* the Superconducting one) however, it already performs a multitude of template-based optimizations targeting clusters of two-qubit gates and finalizes the circuit translation in the technology's own gate set by smartly decomposing all those gates. The aim of this chapter is to provide an in-depth overview of the Toolchain's Step 3 and of its implemented technology-based optimization strategies, both for decomposing the involved two-qubit gates and for compacting the circuit as much as possible.

A description of the exact purpose of this step is reported, followed by a summary of the blocks that define its structure and of their inner workings. Then, the exact sequence of operations performed for each target technology is reported, along with some explanations about the design choices made in the implementation of each technology-specific workflow. Finally, brief descriptions of each function composing the technology-specific sections of the related Python libraries are provided. The proposed descriptions are presented with a complete list of the circuit identities and templates employed in the optimization process.

4.1 Step 3 - Distribution/Mirroring-based optimizations and CX gates decomposition

As stated in Section 2.1.2, Step 3 has a double aim:

- Exploiting a subset of templates which can potentially reduce the number of two-qubit gates in a circuit, especially CX gates
- Subsequently decomposing them according to the target technology and then trying a final optimization effort

This decomposition is delayed until Step 3 because, as it is common in the state-of-the art [37], the choice to work with non-decomposed CX/CZ gates in the Layout Synthesis block was made. This is usually done in order to perform routing operations in a simpler way by considering these gates in compacted form, to then decompose them once they are mapped on the target device. In the case of a template-based approach, it should be noted that this procrastinated decomposition also makes much easier to detect and exploit the templates involving cluster of CXs reported in Section 4.1.5.
This step currently covers only a fraction of the effective operations needed to finalize the Layout Synthesis block described in the state-of-the-art toolchain in [17]. In particular, it is focused on what we could name "the first part", involving the optimization of potentially redundant multi-qubit gates, and "the last part", which is the one regarding the decomposition of the aforementioned gates that is usually performed at the very end of the Layout process. While a mechanism to specify some of the properties of a target device is implemented (namely some files to describe the sign of the interactions between two given qubits in the NMR and Trapped Ions technology, as reported in Sections 4.1.2 and 4.1.3), for this first prototype of Toolchain it was preferred to focus on general-purpose optimizations and on the adaptation to theoretically fully-connected technologies. Moreover, the actual issue of inserting SWAP gates to match the circuit's usage of multi-qubit gate with a non-fully-connected target device is left as a potential future evolution of the project, as

reported in the Conclusions and future perspectives.
Step 3 is designed to take as inputs the optimized circuit .qasm files generated by Step 2 to work at maximum efficiency, but it can also be used on custom, unoptimized .qasm files. This step is **technology-specific** exactly like Step 2, and the performed manipulations of the circuit differ greatly depending on the target technology. Generally speaking, the workflow of this step is:

- Employing the templates based on **parallelization, distribution and mirroring of CX gates** in order to reduce their number in the circuit.
- Decomposing two-qubit gates using a technology-specific gate set. This decomposition is enacted in a way that aims to compact the circuit further whenever is possible.
- If it is possible, trying to exploit the special properties of newly inserted two-qubit gates (namely the **RXX and RZZ gates**) and a final iteration of the **Eulercombo** mechanism described in Sections 3.1.1, 3.1.5 in order to compact the circuit as much as possible.

Step 3 has the important task of handling the complex templates that involve clusters of **CX gates**, and is the Toolchain's primary source of **multi-qubit gates optimizations**. This task is followed by another important one: a smart **decomposition of two-qubit gates** and some other minor optimization, all in order to fully adapt the circuit to the target technology with the least impact on the circuits' gates. Both roles performed by Step 3 are essential to complete the Toolchain's proposed compilation process, but the template-based approach on which they are based on is theoretically **suboptimal** for such tasks. While two-qubit gates decomposition can still be handled efficiently, This is especially true in the case of the exploiting of **CX templates**, which on paper seem quite limited when compared to more complex evaluation algorithms. Also, they are less suited to be effective in the Layout Synthesis' phase of mapping circuits on target devices, which is currently not implemented in the Toolchain.
Step 3 supports the usage of the following gate sets depending on the target technology:

- **NMR technology: R_X, R_Y, R_Z, CX and CZ gates**
- **Trapped Ions technology: R_X, R_Y OR R(θ,ϕ) gates, R_Z, CX gates**
- **Superconducting technology: U1, U2, U3 and CX gates**

Because of the usage of the *Eulercombo* functions, this step also requires **all single-qubit gates to be adjacent to gates of different type**. This condition is guaranteed after the circuit optimization based on Step 1 and Step 2, as reported in Chapter 3, and specifically by the previous usage of the *Eulercombo* itself, but in case of usage of an custom or not previously optimized input circuit they have to be enforced by the user. Once again, the modular structure of the libraries allows to support in the future other quantum technologies with their specific workflows and to upgrade the existing ones with new features.
Step 3 requires two inputs when executed from shell: the **input .qasm circuit file**, which describes the reference quantum circuit that has to be optimized, and the **Subcircuit parameter**, which is a boolean flag that defines if the circuit is indeed a *subcircuit* to be used in conjunction with other QASM-described entities and thus if certain optimizations regarding R_Z gates can be employed. If the input circuit is the output of Step 2, the script is able to parse the file name and to recognize the employed **target technology**, which uses the same parameters as in Step 2 (**"M"** for the **NMR technology**, **"I"** for the **Trapped Ions technology** and **"S"** for the **Superconducting technology**). If the file name was changed or if the input file is custom or not previously optimized, the script displays a message and then asks the user the technology parameter.
All the manipulations involving *Euler angles* are performed through the usage of the **SciPy Python library** [52] and of the **NumPy Python library** [53]. All the benchmarking procedures, the results and their related analysis are reported in Chapter 5 in the appropriate section.

4.1.1 Step 3's structure

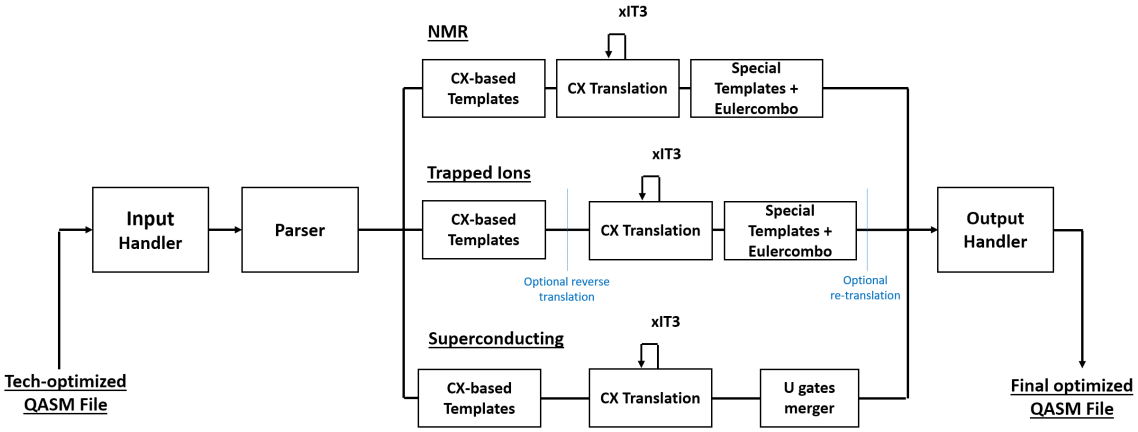


Figure 4.1: Representation of the overall structure of the Toolchain's Step 3 (assuming that the input circuit was previously optimized by Step 2)

Step 3 is implemented with the **layout_optimizer_3 script**, and is divided in the following sections:

1. **Input Handler section:** This section handles the correct reception of inputs and provides error messages if a wrong number of parameters or a wrong file extension are passed as inputs. It also reads some of the .cfg files described in Section 2.1.2 to define the thresholds and approximation accuracy to be used in the optimization process, to set the translation parameter **Iontran** in the Trapped Ions technology case and to set the **IT3** iteration parameter. For what concerns the thresholds, the script provides an error message if an approximation accuracy equal or greater than 1 is provided and a warning message if a null operation identifier threshold is defined, since this could bring to "overzealous" optimizations that could compromise the logic functionality of the circuit. If a non-integer number is passed for the iterative parameter it proceeds to round it, and if a number which is less than 1 is set in the .cfg files for these loops, it automatically sets it to "1".

2. **Parser section:** The **Parser** section generates the circuit list on which the script will work based on the input file and rearranges it, "cleaning" it from endline spaces, blank lines, barriers and comments. It also identifies the number of qubits involved in the target circuit and saves all the measurements performed in the input file on a temporary list, which will be appended to the output circuit list once the optimization is completed.

3. **CX Templates section (NMR, Trapped Ions, Superconducting):** This section is in charge of performing the main "core" of optimizations in this step by using the functions contained in the **step3_cxtemplates library**. These functions detect and employ a subset of templates extrapolated from [50] which are based on **parallelization, distribution and mirroring of CX gates** to achieve some potentially powerful compactions that are unobtainable when using the simple equivalences or the template-based optimization contained in Step 1. As of now, these are the most complex template-based equivalences exploited in the Toolchain. This section is reiterated through a loop defined by the **IT3** parameter to maximize the number of optimizations applied, because some of the manipulations applied through the usage of these templates based on two-qubit gates could reshape the CX clusters in the circuit in such a way that another CX-based template is created. The recommended value of IT3 is actually "2", because while it is true that new templates may be generated after the first iteration of the section, it is quite unlikely to see this reshape in another iteration, so a double loop is usually enough to detect and perform all possible template-based optimizations. In any case, the parameter gives the possibility to achieve an higher grade of optimization, trading off computation time in order to increase the probabilities of all possible templates actually being detected and exploited.

4. **CX Translation section (NMR, Trapped Ions):** This is the section in which the effective decomposition of multi-qubit gates into the technology-specific gate set is performed, if required. In the case of the NMR technology, this involves **RZZ gates** (which can be used as an OpenQASM-supported version of the $\boldsymbol{U_J}$ **gates**) to implement the circuit structure represented in Figure 1.5. In the case of the Trapped Ions technology, this involves **RXX**

gates to implement the circuit structure represented in Figure 1.7. In both cases, the translation process is designed to achieve maximum compaction in the circuit. In both cases, one can use the parameters in the .cfg files described in Section 2.1.2 to forsake this decomposition and use untranslated CX gates.

5. **Special Templates + Eulercombo section (NMR, Trapped Ions):** In this section the last optimization effort is performed. Firstly, the commutation properties with certain single-qubit gates of the newly inserted RXXs and RZZs are exploited in order to apply some **"special templates"** which are *de facto* a particular case of the **Templates 1, 2 and 3** represented in Figure 2.20, 2.21, 2.22. Following this optimization, the **Eulercombo** mechanism described in Sections 3.1.1 and 3.1.5 is called in order to capitalize on eventual streaks of single-qubit gates created after the application of the last templates and to compact as much as possible all the single-qubit gates inserted during the decomposition of CX gates.

6. **Output Handler section:** This section uses the final circuit list to generate a .qasm file in the working directory. If the input file is detected as an output of Step 2 through the parsing of its name, the suffix (_finaloptimized) is replaced. Otherwise, a new suffix is simply appended to the original file name. In either case, the suffix "_X_finaloptimized" in which X is the input parameter describing the target technology ("M", "I", "S") is added, thus making explicit in the file name the chosen target technology. Then, it appends to the file the measurements contained in the original circuit (if it is not a subcircuit, of course) and provides a message announcing that the optimizations were completed successfully and the name of the generated file.

4.1.2 NMR - Specific workflow

- **CX and CZ gates:** Since CZ gates are the most convenient way to implement two-qubit gates, as explained in Section 1.3.2, all CX gates are translated in a form which uses them by using the equivalence represented in Figure 4.6. The two-qubit gate itself is transformed using a **RZZ gate**, a type of gate supported in OpenQASM that can be used to implement the $\boldsymbol{U_J}$ **gate** by using a rotation of $\frac{\pi}{2}$. **The sign of the rotation** in the RZZ gate depends on the target qubits (nuclear spins) involved, and is determined by reading the **NMR_layout parameter** from the **layout_nmr.cfg file** described in Section 2.1.2. Basically the **NMR Layout parameter** is read, and the J coupling sign for each couple of interacting qubits is determined. The parameter itself must be written in form of a **list of lists**, with each list detailing the J coupling sign that each qubit has when interacting with others. In a quantum circuit containing N qubits, Step 3 to work would require a list containing N lists each containing N elements, which can be "$\mp$1" or "0". The "0" value is employed since the qubit interaction with itself is clearly null, so an i-th qubit will feature in the i-th list an i-th element which is equal to 0. In this way, it is possible to implement the circuit structure represented in Figure 1.5. The CX gates are decomposed only if the **CZ Translation parameter** in the **layout_nmr.cfg file** is set as "true", and no operation is performed otherwise. The potential phenomenon of weak J coupling that can impede two-qubit interactions between specific, weakly-coupled qubits is currently not taken into account, and all qubits are considered as ideally coupled between each other.

- **Special Templates:** It can be demonstrated that the **RZZ gates** share the same commutation property of CZ gates with respect to $\boldsymbol{R_Z}$ **gates**. Because of this, it is possible to apply an equivalent to the Template 1 and the Template 3 described in Section 2.2.5 in order to try to combine as many R_Z and RZZ gates as possible. Both templates are merged under a single template named **"Special template 1"**, and described in Section 4.1.6. An **Eulercombo section** is called immediately afterwards to try to compact the circuit even further.

- **Special functions:** Since RZZ gates benefit of the same property of R_Z gates of being negligible when at the very end or very beginning of the circuit, the same *ad-hoc* version of FinalZ and InitialZ functions used in Step 2 and described in Section 3.1.5, (**SpecialFinalZ and SpecialInitialZ**), are called to try to purge every redundant RZZ once all other optimization are applied.

- **Function calls:**

  - ***CX optimization: (repeated IT3 times)***
  - *CXDistribution function*
  - *CXParallel function*
  - *CXMirror1 function*
  - *CXMirror2 function*

  - ***CX translation: (if required)***
  - *CXtoCZ function*
  - *CZHarmonize function*

  - ***Specialtemplate + Eulercombo:***
  - *Specialtempl1 function*
  - *Eulercombo function*
  - *Eulerzyz function*
  - *SpecialFinalZ function*
  - *SpecialInitialZ function*

4.1.3 Trapped Ions - Specific workflow

- **CX gates:** As explained in Section 1.3.3, the Trapped Ions technology supports the **RXX gate** as fundamental two-qubit gate, so a decomposition is applied to CX gates in order to implement the circuit structure represented in Figure 1.7 by using a RXX gate with a rotation of $\frac{\pi}{4}$. **The sign of the rotation** in the RXX gate depends on the target qubits (ions in the chain) involved, and is determined by reading the **Ion_layout parameter** from the **layout_ion.cfg file** described in Section 2.1.2. The **Ion Layout parameter** is then read and the χ interaction sign for each couple of interacting qubits is determined. The parameter itself must be written in form of a **list of lists**, with each list detailing the χ interaction sign that each qubit has when interacting with others. In a quantum circuit containing N qubits, Step 3 to work would require a list containing N lists each containing N elements, which can be "$\mp$1" or "0". The "0" value is employed since the qubit interaction with itself is clearly null, so an i-th qubit will feature in the i-th list an i-th element which is equal to 0. Furthermore, as explained in [34], some rotation parameter in the circuit equivalence depend on the arbitrary parameter v, that can be equal to $\mp$1. To ensure maximum compaction, this parameter is chosen depending on the single-qubit gates adjacent to the decomposed CX in order to allow the highest number of combinations with those gates possible. The CX gates are decomposed only if the **CX Translation parameter** in the **layout_ion.cfg file** is set as "true", and no operation is performed otherwise.

- **Ghost translations:** If in Step 2 all the $\boldsymbol{R_X}$ **and** $\boldsymbol{R_Y}$ gates were translated in the $\mathbf{R(\theta,\phi)}$ generic rotation gate form, an "hidden" translation is performed in the optimization process. Parsing and detecting combinable gates in the $\boldsymbol{R_X}$, $\boldsymbol{R_Y}$ **form** is much easier with respect to the generic rotation form, so before enacting the effective decomposition of CX gates, if the circuit shows that generic form, it is **preemptively translated** using the regular rotation forms. Once all optimization efforts have been performed, all R_X and R_Y gates, both "old" and newly inserted, are re-translated in the generic rotation gate form.

- **Special Templates:** It can be demonstrated that the **RXX gates** shows a commutation property with respect to $\boldsymbol{R_X}$ **gates** [34], so it is possible to apply a template equivalent to the Template 2 described in Section 2.2.5. Moreover, a new template which is similar to Template 1 but uses R_X gates instead of R_Z gates can be employed in order to try to combine as many R_X and RXX gates as possible. Both templates are merged under a single template named **"Special template 2"**, and described in Section 4.1.6. An **Eulercombo section** is called immediately afterwards to try to compact the circuit even further.

- **Function calls:**

  - ***CX optimization: (repeated IT3 times)***
  - *CXDistribution function*
  - *CXParallel function*
  - *CXMirror1 function*
  - *CXMirror2 function*

  - ***Ghost reverse translation: (if required)***
  - *IonTranslator_Reverse function*

  - ***CX translation: (if required)***
  - *IonCX function*

  - ***Specialtemplate + Eulercombo:***
  - *Specialtempl2 function*
  - *Eulercombo function*
  - *Eulerzyz function*

  - ***Ghost re-translation: (if required)***
  - *IonTranslator function*

4.1.4 Superconducting - Specific workflow

- **CX gates:** Since CX gates are supported in their undecomposed form in the Superconducting technology, they are left untouched and **no translation operations are performed**. The layout constraints intrinsic to this technology are not currently taken into account, and the architecture is considered as fully-connected.
- **Special Templates:** Regular CX do not benefit from any "special template", but only from the regular ones described in Chapter 2 and in this Chapter. Thus, **no particular operation is performed**. A **SupercondMerge function** is called immediately after the regular CX optimizations in order to try to capitalize on potentially newly generated streaks of U gates and to compact the circuit even further.
- **Function calls:**
  - ***CX optimization: (repeated IT3 times)***
  - *CXDistribution function*
  - *CXParallel function*
  - *CXMirror1 function*
  - *CXMirror2 function*
  - *SupercondMerge function*

4.1.5 step3_cxtemplates library

Employed CX templates

CX TEMPLATES

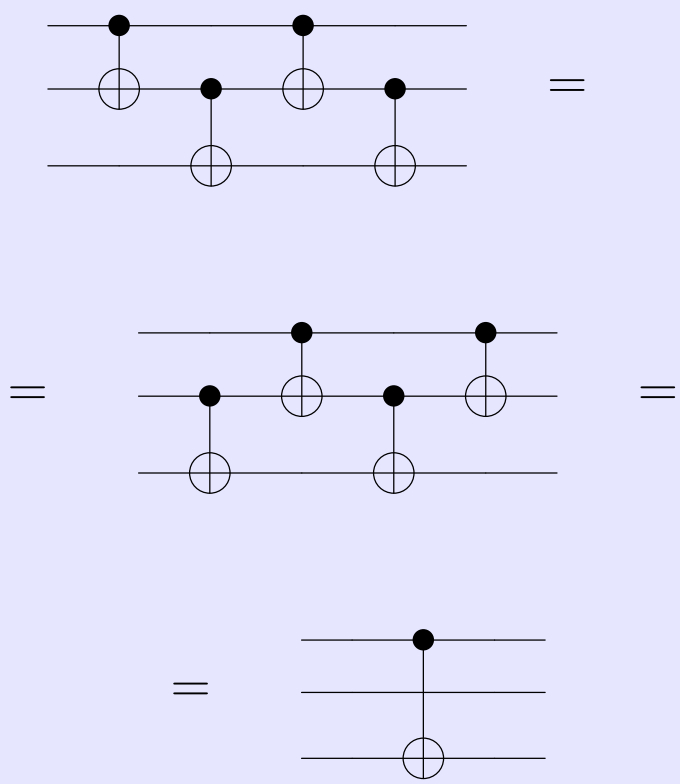


Figure 4.2: CX Template 1: transformation of a distribution of four alternated CXs into a single long-range CX

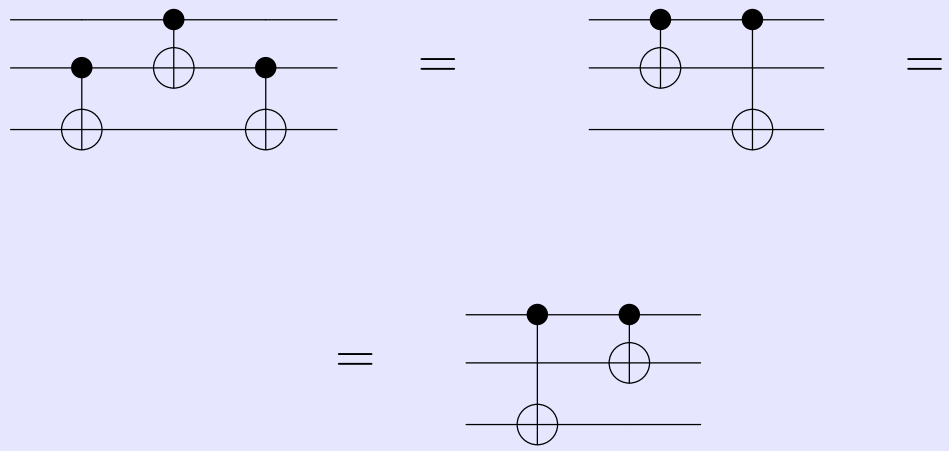


Figure 4.3: CX Template 2: transformation of a parallel cluster of CXs into a CX and a long-range CX

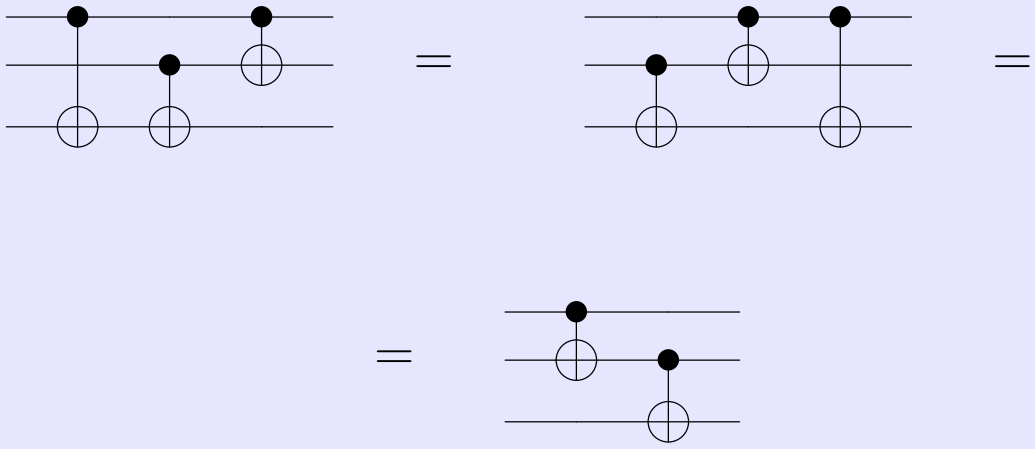


Figure 4.4: CX Template 3: transformation of a mirrored cluster of CXs into two CX gates (Form 1)

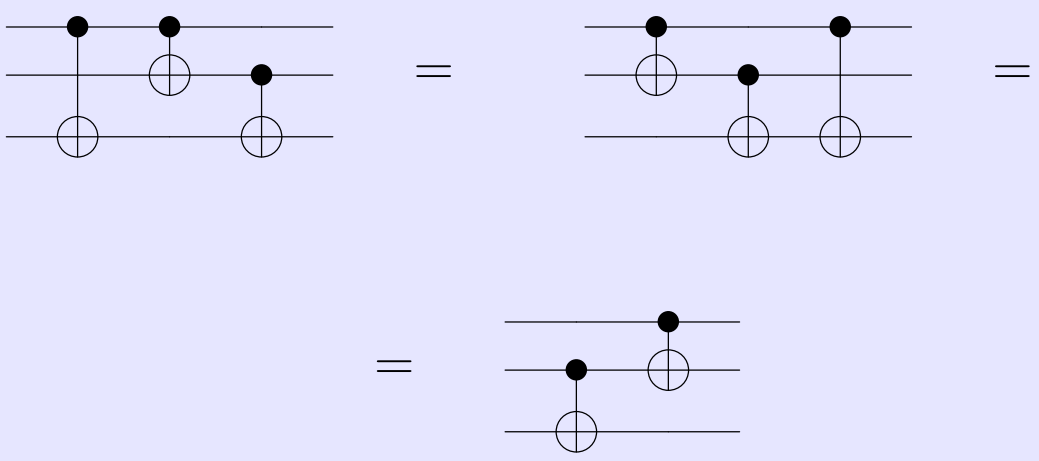


Figure 4.5: CX Template 4: transformation of a mirrored cluster of CXs into two CX gates (Form 2)

Library description

CX TEMPLATES

- **CXDistribution function:** This function scans each qubit line in the circuit backwards, starting from the circuit's end, and tries to exploit the template represented in Figure 4.2. The detection of the template takes place on the qubit line in the middle. Once the template has been detected, the function checks if the distribution of CX gates is uninterrupted by single-qubit gates; if not, the template cannot be applied, so no operation is performed. Otherwise, it implements the long-range CX on the right qubits.

- **CXParallel function:** This function scans each qubit line in the circuit backwards, starting from the circuit's end, and tries to exploit the template represented in Figure 4.3. The detection of the template takes place on the qubit line in the middle. Once the template has been detected, the function checks if the cluster of CX gates is uninterrupted by single-qubit gates; if not, the template cannot be applied, so no operation is performed. Otherwise, it checks the gates adjacent to the template and transforms the latter in the form which can yield a possible null operation. If no null operation can be achieved, it automatically chooses the first form in Figure 4.3 (a CX followed by a long-range CX).

- **CXMirror1 function:** This function scans each qubit line in the circuit backwards, starting from the circuit's end, and tries to exploit the template represented in Figure 4.4. The detection of the template takes place on the qubit line in the middle for the two short-range CXs, and is completed by checking if they are adjacent to a long-range CX. Once the template has been detected, the function checks if the cluster of CX gates is uninterrupted by single-qubit gates; if not, the template cannot be applied, so no operation is performed.

- **CXMirror2 function:** This function works exactly like the CXMirror1 function, but implements instead the template represented in Figure 4.5.

4.1.6 step3_cxtranslate library

CX TRANSLATE - NMR CASE

- **CXtoCZ function:** This function scans each qubit line in the circuit backwards, starting from the circuit's end, identifies all CX gates and then translates them into CZs using the template reported in Figure 4.6. Since CX gates do not benefit from the symmetry property of CZs, the translation in this case is straightforward. The decomposed form of the H gates to be used is evaluated for both sides by identifying the adjacent gates. Each gate is directly inserted in a rotational form with floating-point notation. Then, when two or more single-qubit gates of the same type are adjacent, an optimization is performed through the **NullPurge, InitialZ and FinalZ functions**. Once the purge or the combination is applied, the scan on the qubit line resumes from the last gate before the gates that were optimized, and from the end of the circuit if there are none left.

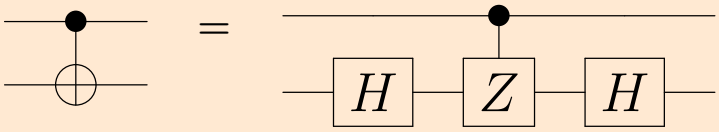


Figure 4.6: Equivalence when transforming CX gates into CZ gates

- **CZHarmonizer function:** This function scans each qubit line in the circuit backwards, starting from the circuit's end, and tries to exploit the symmetry of CZ gates, both "old" or newly inserted, to try to generate null operations between them by making them detectable by

the **NullPurge function**. That function, in fact, does not detect a couple of mirrored CZs as a null operation, exactly as in the CX gates case. In order to compact the circuit as much as possible, the CZHarmonizer function detects all adjacent CZ gates which involve the same two qubits. At this point of the process the only potentially removable couples of CZs are the mirrored ones, since the others have already been dealt with previously. Once a mirrored couple is detected, the function checks if the two gates are perfectly adjacent, with no single-qubit gates between them. If so, it proceeds to purge them.

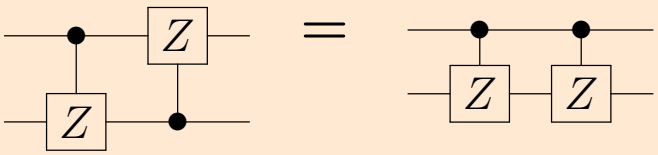


Figure 4.7: Exploitation of CZ gates' symmetry to generate a null operation

- **NMRCZ function:** This function detects all CZ gates in the circuit and transforms them in the form represented in 1.5 using two R_Z gates and a technology-specific RZZ gate. To evaluate the sign of the J coupling interaction, it reads the **NMR Layout parameter** from the .cfg files, and if a mismatch between the signs in the list of lists is detected an error is displayed to signal it and no other operations are performed. If the decomposition happens, the function also tries an optimization using the **NullPurge and InitialZ** functions. The **FinalZ** function is not called because the structure used in the decomposition does not insert any new R_Z gate towards the end of the circuit.

CX TRANSLATE - TRAPPED IONS CASE

- **IonTranslator_Reverse function:** This function is called only when the **Iontran parameter is set to "true"**. When this happens, it performs the same translation of the **IonTranslator function**, but in the opposite verse. The function scans the circuit and translates each generic rotation $\mathbf{R(\theta,\phi)}$ gate in an equivalent $\boldsymbol{R_X}$ and $\boldsymbol{R_Y}$ gate. This is done to ensure that when applying the **Special Template + Eulercombo section** in the workflow all gates are not in the generic rotation form, since the functions involved do not support it. This is due to the fact that a generic rotation form makes optimizations harder to perform. Once the section is executed, the **IonTranslator** function is eventually called once again to ensure the circuit gates are in the desired form.

- **IonCX function:** This function detects all CX gates in the circuit and transforms them in the form represented in 1.7 using two R_X gates, two R_Y gates and a technology-specific RXX gate (which must have a rotation parameter equal to $\frac{\pi}{2}$ in order to implement the desired $\frac{\pi}{4}$ manipulation using OpenQASM's RXX gates [**?**]). To evaluate the sign of the χ interaction, it reads the **Ion Layout parameter** from the .cfg files, and if a mismatch between the signs in the list of lists is detected an error is displayed to signal it and no other operations are performed. As Figure [**?**] shows, the involved gates also depend from an arbitrary parameter v, which can be equal to ∓ 1. To evaluate the most convenient parameter to use, the function checks the gates adjacent to the CX, and chooses a v parameter which ensures the lowest rotation parameters in the potentially combined gates in order to reduce overall circuit latency. To do so it first checks for the best case scenarios, the ones in which two gates can be combined (both gates on the right or a gate on the

right and a gate on the left). In the regular scenarios in which only one gate can be combined, the function prioritizes an efficient combination of the R_Y gates, starting from the one on the right. If no of the newly-inserted gates can be combined with other adjacent gates, the function automatically set a v parameter equal to "1". Then, if the decomposition happens, the function also tries an optimization using the **NullPurge** function.

CX TRANSLATE - SPECIAL FUNCTIONS

- **SpecialNullPurge function:** This function act as an *ad-hoc* version of the **NullPurge function** described in Section 2.2.3 to combine all adjacent **RXX and RZZ gates**. The mechanism is almost the same: this function scans each qubit line in the circuit backwards, starting from the circuit's end, checking each couple of adjacent RZZ and RXX gates. It then detects if the gates are acting on the same two qubits and checks if they are perfectly adjacent, with no single-qubit gate between them. If so, they can be combined in a manner which is similar to the Pauli gates case of the original NullPurge. If the resulting rotation parameter is lower or equal to the **Threshold 2 parameter** read from the .cfg files, the combined gate is considered as a null operation and thus purged. Once the purge or the combination is applied, the scan on the qubit line resumes from the last gate before the gates that were optimized, and from the end of the circuit if there are none left. This function is employed in the **SpecialTempl1 and SpecialTempl2 functions**, which are the only and last attempt to obtain further combinations between RXX and RZZ gates once the decomposition is enacted.

- **SpecialTemplate1 function:** This function act as an *ad-hoc* version of the **Templ1 function** described in Section 2.2.5 to exploit the commutation of **RZZ gates** with R_Z gates. This function scans each qubit line in the circuit backwards, starting from the circuit's end, and tries to exploit the templates represented in Figure 4.8, in one verse or another. The function tries to "swap" the gates in order to obtain one or multiple combinations between gates of the same type. If no null operation can be achieved, it tries to move each R_Z-type gate towards the beginning or the end of the circuit if it is not a subcircuit and then attempt an optimization through the call of **InitialZ and FinalZ functions**. Each time that a swap is required, it makes sure that the logical functionality of the circuit is maintained when applying the template. The template is exploited on both the qubit lines involved.

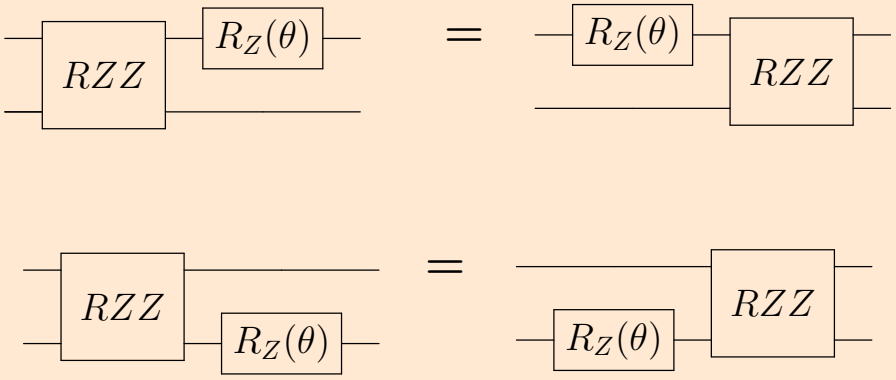


Figure 4.8: RZZ gate commutation properties with R_Z gate

- **SpecialTemplate2 function:** This function act as an *ad-hoc* version of the **Templ2 function** described in Section 2.2.5 to exploit the commutation of **RXX gates** with R_X gates. This function scans each qubit line in the circuit backwards, starting from the circuit's end, and tries to exploit the templates represented in Figure 4.9, in one verse or another. The function tries to "swap" the gates in order to obtain one or multiple combinations between gates of the same type. Each time that a swap is required, it makes sure that the logical functionality of the circuit is maintained when applying the template. The template is exploited on both the qubit lines involved.

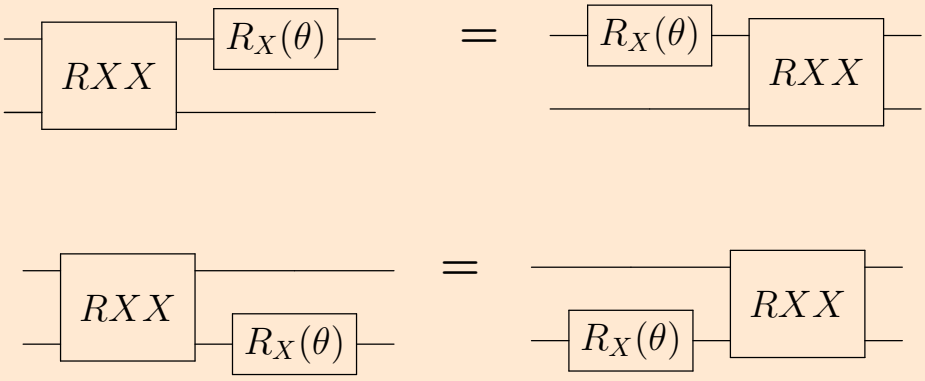


Figure 4.9: RXX gate commutation properties with R_X gate

Chapter 5

Benchmarks

In order to ensure the correctness and efficiency of the Toolchain's optimization, several tests and benchmarks have been progressively performed during its development. The aim of this chapter is to provide a full explanation about the procedures and the tools used to test the generated optimized circuits and to verify that the identities were satisfied. In the first Section, an overview of the procedures used to set the benchmarks is provided, along with a brief description of the two state-of-the-art transpilers that were chosen as reference, IBM's Qiskit and Cambridge Quantum Computing's T-KET. The sections that follow are dedicated to the tests performed on each of the Toolchain's steps, and they present an explanation of the benchmarking methods, along with a brief analysis of the obtained results. The final section is dedicated to Step 3, the last segment of the Toolchain, whose results correspond to the Toolchain's final results. Each section also reports the results obtained by testing quantum circuits of different scale in the form of comparative histograms.

5.1 General benchmarking procedures

Generally speaking, in multiple occasions during the development of the Toolchain a quick, reliable way to generate simple quantum circuits and to test circuit identities was needed. To perform this kind of trivial yet useful tasks, both **IBM's Quantum Experience Composer** [48] and **Strilanc's Quirk Simulator** [58] were used. They are both quantum circuit simulators that allow an user-friendly generation of custom circuits in the form of toolboxes or, in the case of Quantum Experience, also in the form of an OpenQASM-defined circuit. These two tools were used extensively throughout the development, and their role must not be neglected.

To test the circuits generated by the Toolchain's steps, some benchmark scripts were created in **Python language**. The first aim of these script was to verify that the introduced optimizations were actually correct, and that each circuit's logic outcome was the same as the reference, untouched quantum circuit. To do so both the .qasm file describing the reference quantum circuit and the .qasm files generated by the Toolchain were used in conjunction with **IBM's Qiskit** to create quantum circuits based on them in it and then simulate them using the **Aer Libraries' Simulator**. initially, an approach in which **Qiskit's Unitary Simulator** was used to evaluate each circuit's overall unitary matrix was employed.

This first approach revolved around generating the matrices and then evaluating the product between the first matrix and the conjugate transpose of the second matrix. The product between the first matrix and the transposed conjugate of the second one must be the identity matrix $\mathbf{I}_n$.

$$\mathbf{R} = \mathbf{M'}^{\dagger}\mathbf{M} = \mathbf{I}_n$$

For this reason, the grade of correctness of the optimized version was evaluated as the "maximum deviation" between the resulting matrix and the identity matrix, calculated as the greatest difference between the **squared magnitude of the elements in diagonal** of the resulting matrix and "1", which is the value of all

elements on the main diagonal of the identity matrix.

$$\textbf{Max Deviation} = \max(1 - |R[ii]|^2)$$

If this **"maximum deviation" was low enough** (initially lower than 10^{-3}), the optimization was marked as "correct". This approach was later abandoned because calculating each matrix required a huge amount of computational resources and was thus painstakingly slow. When dealing with large-sized circuits in particular the computational load was too much to handle, and Qiskit usually crashed, thus making this approach unfeasible.

The second approach, which became the one used to test all generated circuits, involved the usage of **Qiskit's QASM Simulator**. By using this simulator's standard settings (such as 1024 shots and others) and the *aer.QasmSimulator* backend and by ensuring that every qubit line in each tested circuit had a measurement performed at the end, it was possible to simulate the circuit's outcome and to visualize it in form of an histogram representing the **simulated output states with their associated probabilities**. In case of circuits in which a single output state was expected with a probability of 100%, the relative optimized circuit was deemed as "correct" if the output matched completely with a probability of also 100%, as represented in Figure 5.1. In case of circuits in which multiple output states with different probabilities were expected, the obtained results were compared and the optimization was marked as "correct" only if the difference between each state probability did not exceed 5% of the related reference's probability, as represented in Figure 5.2. A threshold of 5 % was chosen because in these cases randomness and number of executed shots can significantly impact the results (so it is generally unlikely to have very small differences in probabilities' percentages), but a good grade of fidelity is still required. The generated graphs representing the output states and output states' probabilities of both the original reference and its optimized version were checked and validated for each tested circuit, but are not reported in this thesis.

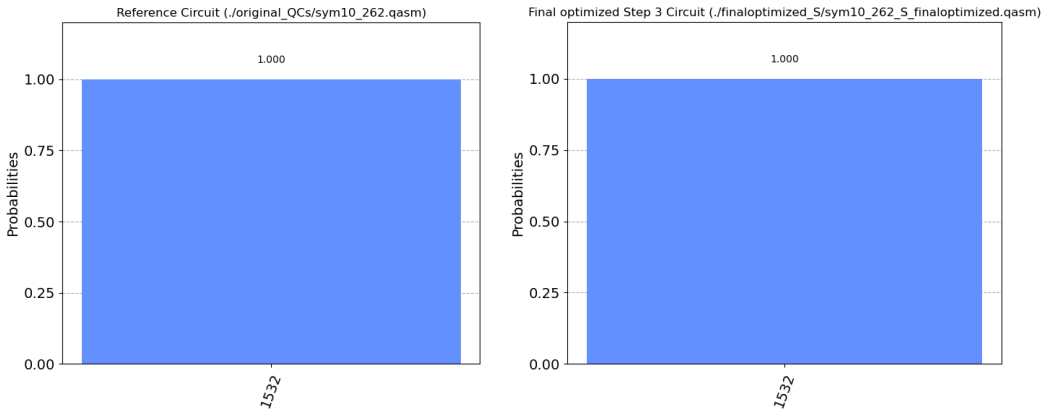


Figure 5.1: Example of an acceptable output probability of the optimized circuit compared to the reference circuit's one. In circuits that produces a single outcome state, the state and probabilities must totally match to be considered acceptable

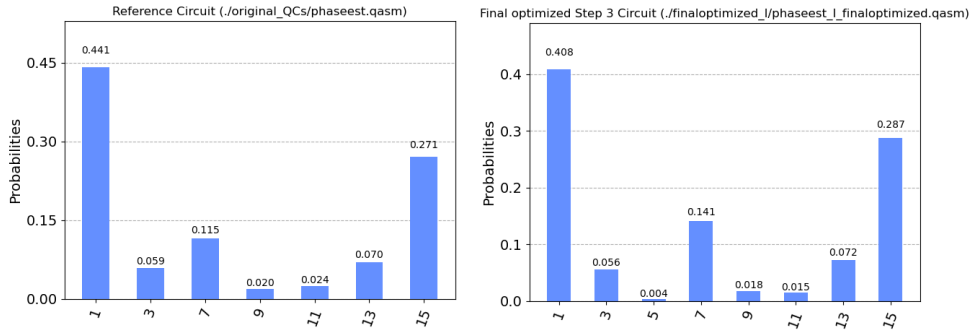


Figure 5.2: Example of an acceptable output probability of the optimized circuit compared to the reference circuit's one. In circuits that produces multiple outcome states, the state and probabilities are considered acceptable if they are very similar (no differences greater than 5%)

Once the optimized circuits were verified as "correct", the scripts' aim became to calculate its relevant parameters and to compare them with the ones generated by the other state-of-the-art compilers.

The selected parameters were:

- The **total number of single-qubit gates**.
- The **total number of non-R_Z/U1 single-qubit gates** (which are the *de facto* relevant single-qubit gates latency-wise when RZ gates are implemented virtually).
- The **total number of two-qubit gates**.
- The **approximated circuit latency weighted for single-qubit gates**.

More information about the criteria used to evaluate these parameters in each step's test are reported at the beginning of the related section.

5.1.1 Tested circuits

Several custom quantum circuits were made to evaluate the correct application of templates and identites in the developed libraries, but none of them were used to evaluate the Toolchain's performance. Instead, it was preferred to use general, different-sized quantum circuits that were made available on **GitHub**. These circuit were taken from existing repositories: in particular, form a repository of an existing QASM Benchmark Suite, **QASMBench** [59], [60], and a repository of testing circuits belonging to **Prof. Dr. R. Wille's IIC Group from the Johannes Kepler University of Linz** [61], [62]. The set of circuits chosen for the benchmark was composed with the idea of involving small to large-scaled circuits. Only the latest version of each chosen circuit in the repositories was taken in consideration. The circuits were mostly left untouched: the only actions that were performed on them was adding a measurement on each involved qubit line at the end and, in a few circuits taken form QASMBench's repository, to manually decompose custom-defined gates (which the Toolchain at the moment does not support). The circuits used in the benchmarks have a quantum register size that spans from 3 qubits to 18 qubits.

The main quantum circuits used in the benchmarks were:

Circuit name	Number of qubits	Number of gates
Linearsolver	3	43
Adder_small	4	27
Phaseest (formerly pea)	5	108
Adder_medium	10	174
Alu-bdd_288	16	104
Rd84_253	16	17062
Sym10_262	16	61269
Urf5_280	16	80331
Adder_large	18	346

All circuits optimized by the Toolchain were generated by setting an **approximation of π and its dividends at 10^{-10}** and a **threshold of rotation parameters for a gate to be considered a null operations at 10^{-12}** in the configuration (.cfg) files.

5.1.2 Comparing the Toolchain to the state-of-the-art

The Toolchain's performance was validated by comparing it to two of the state-of-the-art most successful compilers.

The first chosen compiler is **Qiskit Transpiler**. Qiskit is an open-source framework for quantum computing developed by **IBM Research**, and is fully implemented in **Python** language. What was specifically used to generate comparison circuits was the **Qiskit Terra Transpiler**, which allows an optimized transpiling of quantum circuits described in Open QASM using a user-defined set of gates. It also allows multiple optimization levels (**none, 1, 2, 3**, with an higher levels that correspond to a finer optimization. Qiskit was an obvious choice for these benchmarks. This compiler has been widely used in the quantum research community for years, and IBM's policy of open-sourcing it and of making available tools like Quantum Experience for free made it one one of the most employed quantum compilers, especially when dealing with Superconducting devices, which are IBM's flagship quantum technology. Also, the whole Toolchain is based on **IBM's OpenQASM language** and the

benchmarks were realized using exactly Qiskit, so involving IBM's core quantum framework was deemed as a must. To provide multiple comparisons with different grades of optimization, each circuit was transpiled both with an **optimization grade of "1"** and a finer **optimization grade of "3"**. The benchmarks were performed using **Qiskit version 0.27**, with **Qiskit Aer Libraries version 0.8.2** and **Qiskit Terra Libraries version 0.17.4** (the latest releases at the time of writing).

The second chosen compiler is **T-KET**, a closed-source architecture-agnostic quantum software developed by **Cambridge Quantum Computing**, implemented in **C ++** language. This compiler allows to transpile machine-independent quantum algorithms into optimized quantum circuits, and it supports multiple intermediate languages (including OpenQASM). It also allows two optimization levels, **standard and maximum**. Because of it being closed-source, to access it and interfacing it with Qiskit an *ad-hoc* module called **pytket-qiskit** implemented in **Python** language was employed. T-KET was chosen because in the past few years it proved to be a particularly efficient compiler in the state-of-the-art, outperforming most competitors in several published benchmarks. T-KET is particularly renowned for its capability to smartly adapt a circuit to a given target device and to manage multi-qubit gates.
Once again, choosing T-KET in the comparisons was a key choice to determine the Toolchain's competitiveness in the state-of-the-art. The **pytket** module was used to interface with Qiskit in order to generate a quantum circuit in it starting from T-KET's optimized output for benchmarking purposes, as done for the Toolchain's generated circuits. In order to provide multiple comparisons with different grades of optimization, each circuit was transpiled both with the **standard optimization grade** and the **maximum optimization grade**. The benchmarks were performed using **pytket-qiskit version 0.15.1**.

The **general trend noted in the benchmarks** is that **Qiskit** features a versatile management of **single-qubit gates**, being able to optimize them well using the gates basis of multiple target technologies. **T-KET**, on the other hand, proved to be more **unpredictable** when dealing with single-qubit gates, because it alternates very good optimizations to completely suboptimal handles. At the same time,

T-KET proved to be the best in the **reduction of two-qubit gates**, as it was able to reduce their number substantially by accepting some tradeoffs on the single-qubit gates' number.

5.2 Step 1 intermediate benchmarks

Since Step 1 is the Toolchain's step in which the main bulk of template-based optimizations is performed, its benchmarks were the ones in which most circuits were tested to ensure of the correcteness of each optimization. To maintain the step's philosophy of technology-agnosticism, all qubits were considered as **ideal** and all architectures as **fully-connected**, with no specific backend device. The **initial basis of quantum gates** used in **Qiskit** were two: the first using **X, Y, Z, S, T, $S^\dagger$, $T^\dagger$, R_X, R_Y, R_Z, CX, CZ and undecomposed H gates** and the second **X, Y, Z, S, T, $S^\dagger$, $T^\dagger$, R_X, R_Y, R_Z, CX and CZ gates**. This solution was employed because, if this Clifford + T expanded gate set with decomposed H gates was used, **Qiskit** sometimes showed some anomalies in the compilation and used a non-smart optimization for single-qubit gates. When comparing those transpiled circuits with the others, only the **best implementations** among these two cases were considered. In the case of non-decomposed H gates, the effective count of gates was calculated by using H gates' efficient decomposition represented in Figure 2.19.
The **initial basis of quantum gates** used in **T-KET** was the **ProjectQ one**, which involved the same gates of Qiskit's first used gate basis plus **SWAP, CRZ and V gates**. A first batch of benchmarks was performed using these gate basis.
To determine the final comparison, it was decided to "equalize" the circuits generated by the state-of-the-art's compilers to the ones generated by the Toolchain, so a common **gate basis** using only **R_X, R_Y, R_Z, CX and CZ gates** was employed for both compilers.
To evaluate the single-qubit gates' overall weighted latency, it was decided that each **non-R_Z gate** with a rotation parameter of $\boldsymbol{\theta}$ would introduce a latency equal to $\mathbf{2}\frac{|\boldsymbol{\theta}|}{\boldsymbol{\pi}}$. Hence, gates with a rotation of $\mp\pi$ introduce a weighted latency equal to "2" and gates with a rotation of $\mp\frac{\pi}{2}$ introduce a weighted latency equal to "1". The calculated latencies were approximated at two decimals. R_Z gates were not considered because a virtual "instantaneous" implementation is assumed.

Quantum circuits - First used gate basis

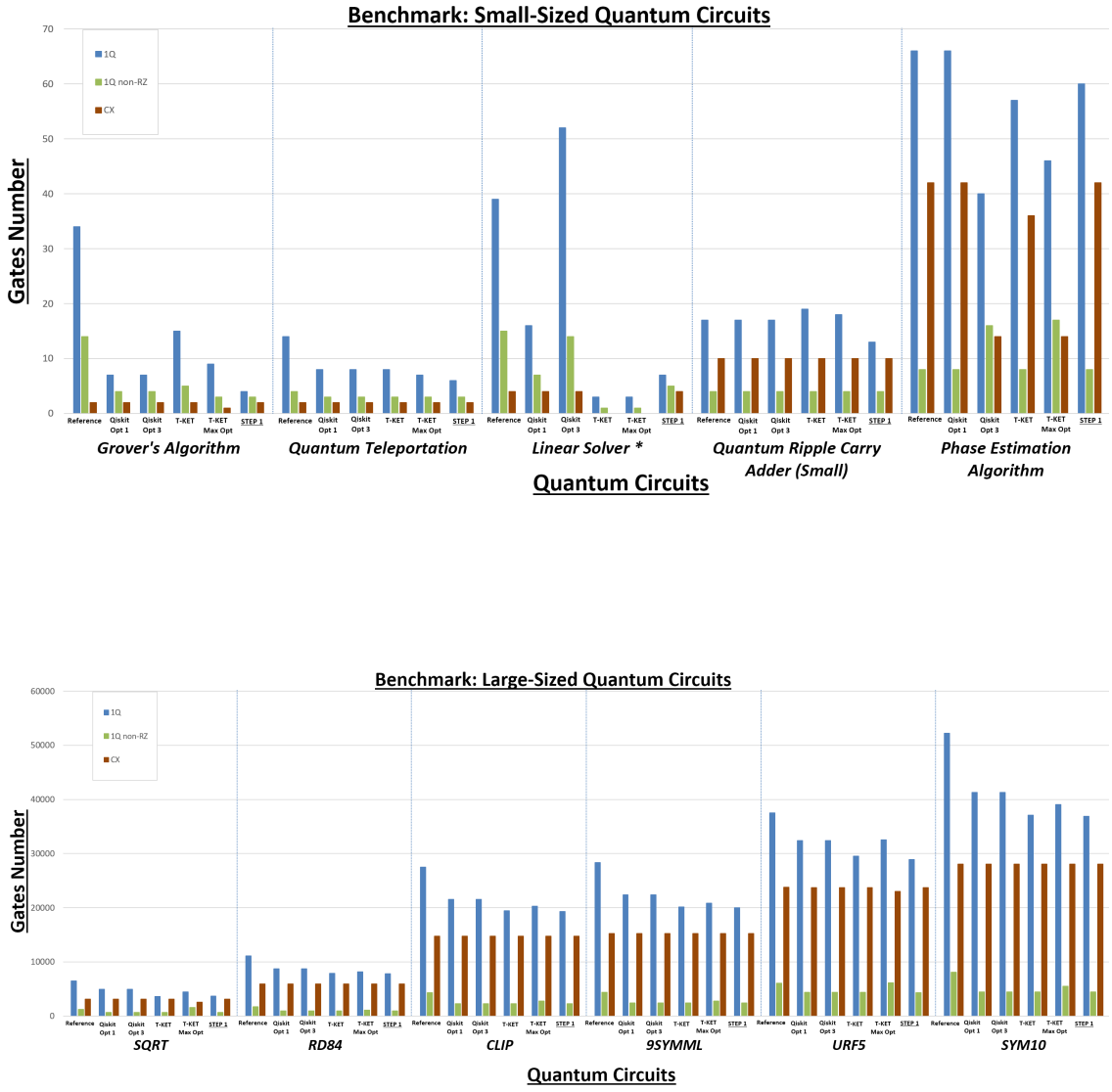


Figure 5.3: General benchmarks of Step 1 using the first proposed gate basis

Small-sized quantum circuits

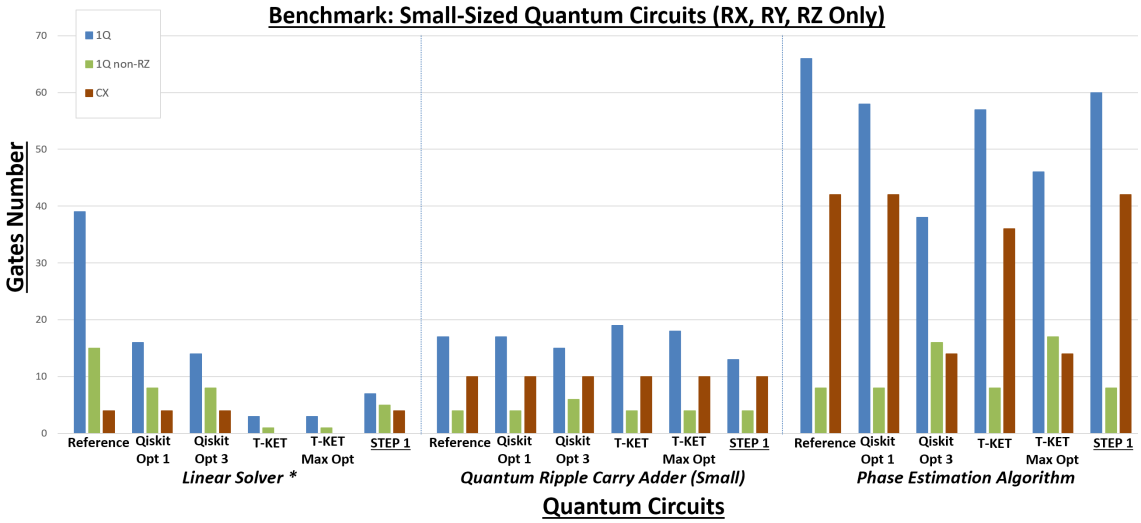


Figure 5.4: Benchmarks of gates number in small-sized circuits in Step 1 using the equalized gate basis

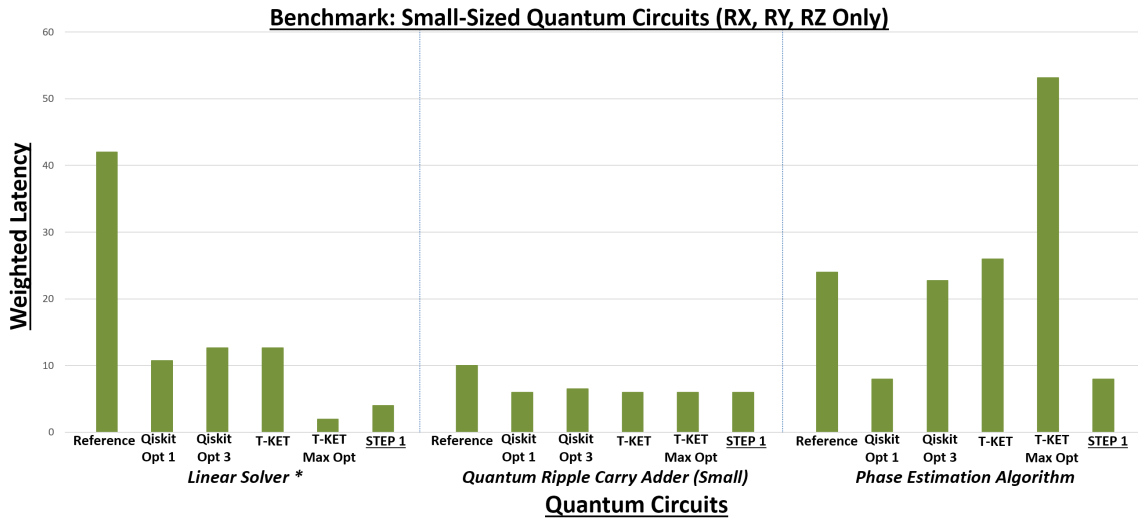


Figure 5.5: Benchmarks of latency in small-sized circuits in Step 1 using the equalized gate basis

Medium-sized quantum circuits

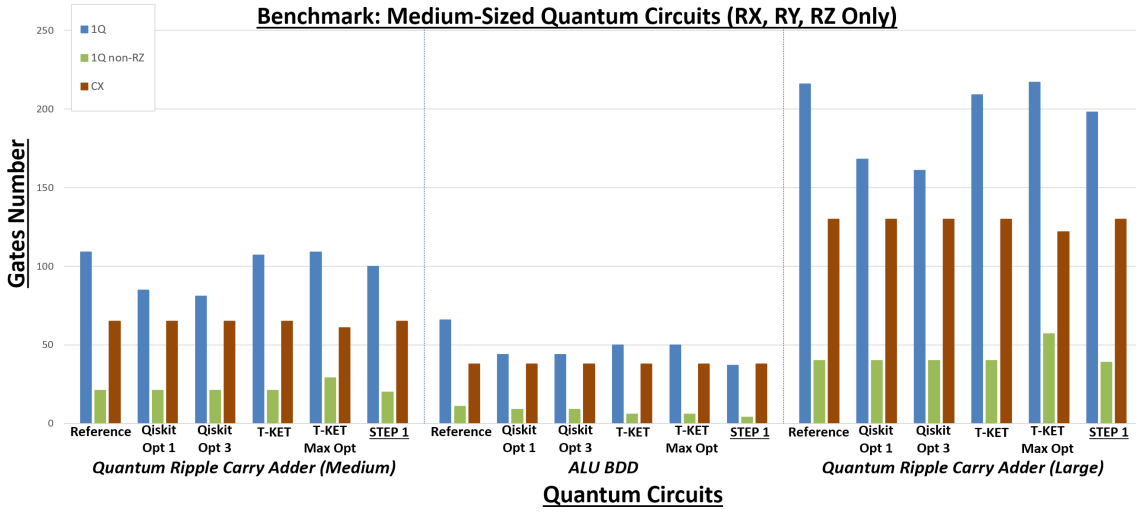


Figure 5.6: Benchmarks of gates number in medium-sized circuits in Step 1 using the equalized gate basis

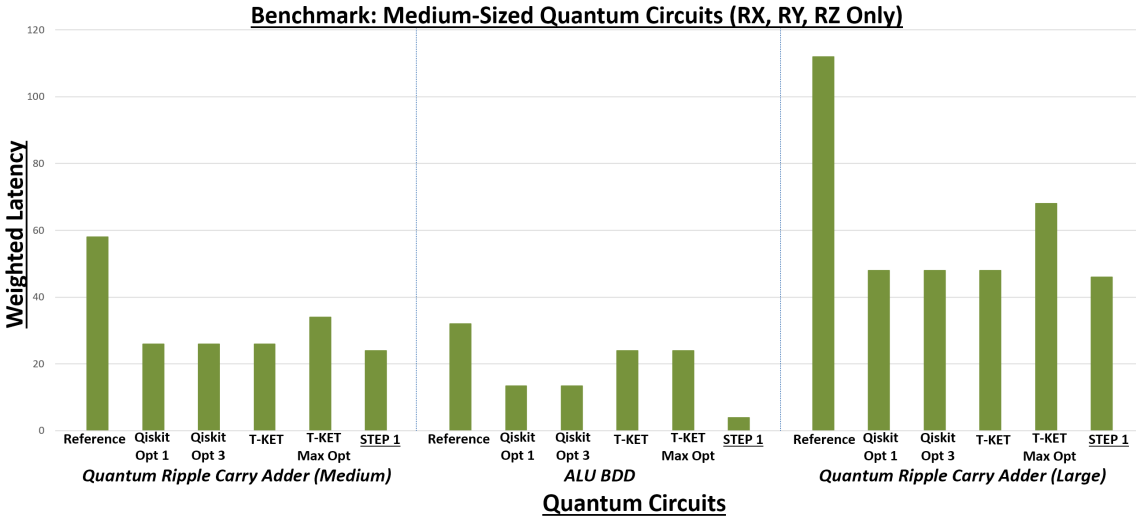


Figure 5.7: Benchmarks of latency in medium-sized circuits in Step 1 using the equalized gate basis

Large-sized quantum circuits

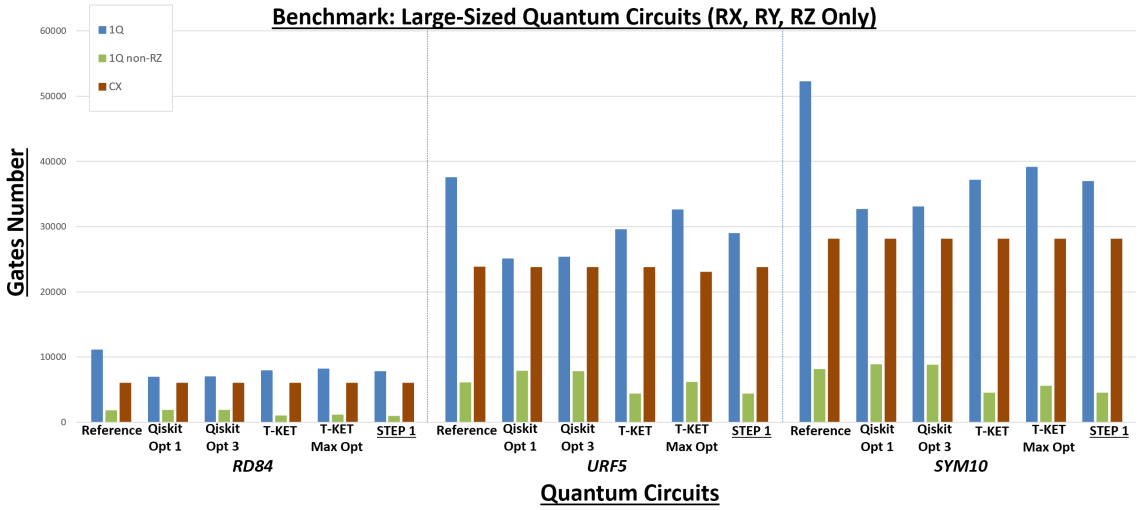


Figure 5.8: Benchmarks of gates number in large-sized circuits in Step 1 using the equalized gate basis

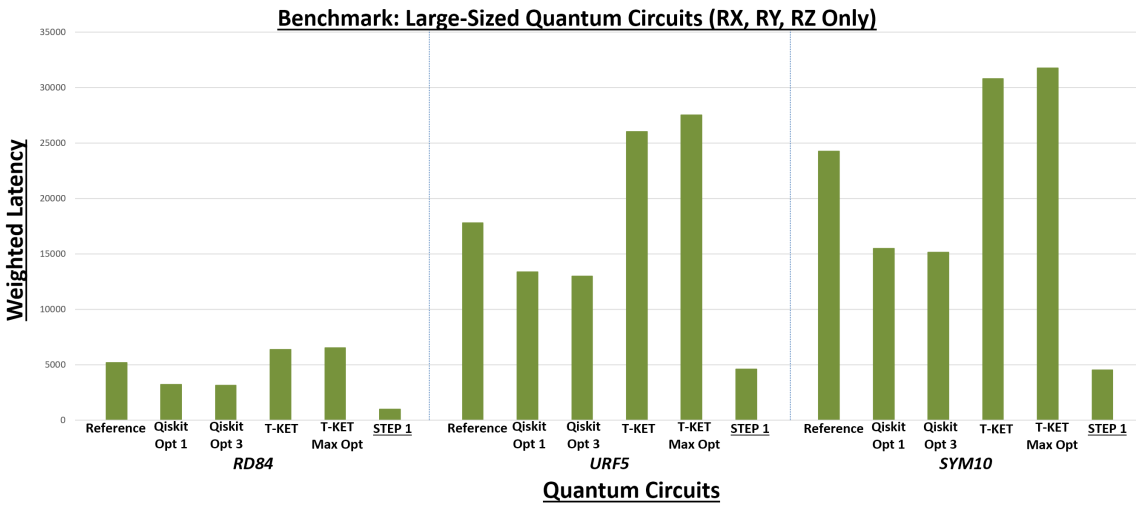


Figure 5.9: Benchmarks of latency in large-sized circuits in Step 1 using the equalized gate basis

5.2.1 Intermediate results analysis

The obtained results show that Step 1 fares consistently well when **optimizing single-qubit gates**, competing in almost every scenario with T-KET's optimization (and in some cases even surpassing it in non-R_Z gates usage) and outperforming Qiskit all over the chart in this branch; this results in an overall better weighted latency of single-qubit gates in the circuit. At the same time, it is clear how the introduced **two-qubit gates optimizations** are minimal. This was expected because, as reported in Chapter 4, the main source of optimizations in the Toolchain is Step 3, while Step 1 templates' purging capabilities on CX gates are pretty weak. For what concerns **Qiskit**, when comparing the results obtained with the first dual gate basis and the R_X, R_Y, R_Z only basis, it can be observed how the number of employed single-qubit gates is lower in the second case, but the number of employed non-R_Z single-qubit gates is quite higher, even doubled in some circuits. This probably means that Qiskit does not use the efficient decomposition of H gates, but the **classical decomposition using R_X and R_Y gates** instead.
The compilation time of Step 1 was really short when compared to the other compilers. This was expected as well, because of the step's lack of more complex optimizations, involving Euler transformations and two-qubit gates.

5.3 Step 2 intermediate benchmarks

In order to test Step 2, only circuits previously optimized by **Step 1** were used. During the tests, all qubits were considered once again as **ideal** and all architectures as **fully-connected**, with no specific backend device. The **gate basis** used in each technology implementation for all compilers were the **technology's legal set of gates**, with the only exception of the **NMR technology** in which CX are still treated as "legal" because they are decomposed in Step 3, as explained in Chapter 3 and Chapter 4:

- **NMR Technology: R_X, R_Y, R_Z, CX and CZ gates**
- **Trapped Ions Technology: R_X, R_Y, R_Z and CX gates**
- **Superconducting Technology: U1, U2, U3 and CX gates**

Since the set of tested quantum circuits did not actually employ any **CZ gate** in it, the handler functions managing the efficient translation of CZ gates into CX gates were tested with some **custom circuits**. The correct usage of the functions' mechanism was ensured. Moreover, in order to test the optimization capabilities of the Eulercombo functions described in Section 3.1.5, a **randomized custom circuit** featuring a single qubit line with 10000 random single-qubit gates, each one of different type with respect to tits adjacent gates, was employed. The random circuit was tested multiple times, each time with a different randomizer seed.
To evaluate the single-qubit gates' overall weighted latency, in the **NMR and Trapped Ions technology cases** it was decided that each **non-R_Z gate** with a rotation parameter of θ would introduce a latency equal to $\mathbf{2}\frac{|\theta|}{\pi}$. Hence, gates with a rotation of $\mp\pi$ introduce a weighted latency equal to "2" and gates with a rotation of $\mp\frac{\pi}{2}$ introduce a weighted latency equal to "1". The latencies were approximated at two decimals. In the **Superconducting technology case**, a more straightforward approach was used instead, and the weighted latency was calculated by assuming that each **U2 gate** introduces a latency equal to **"1"** and that each **U3 gate** introduces a latency equal to **"2"**. This criteria was used to emphasize the fact that an implementation using a U3 gate last **twice** the time of an implementation using a U2 gate. R_Z and U1 gates were not considered because a virtual instantaneous implementation is assumed.

Randomized circuits test

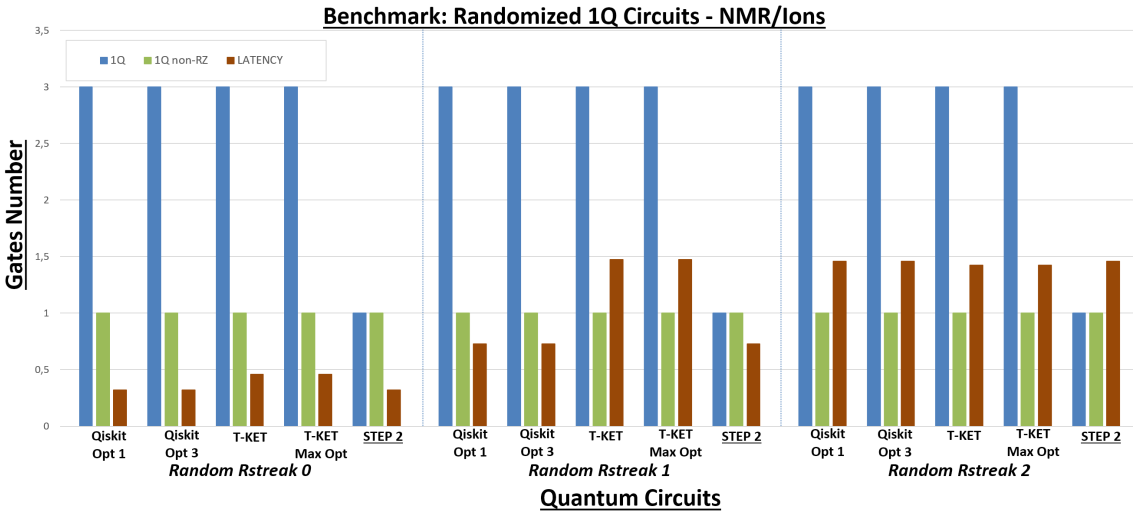


Figure 5.10: Benchmarks of Eulercombo functions with a randomized circuit 10000 gates long

Small-sized quantum circuits - NMR

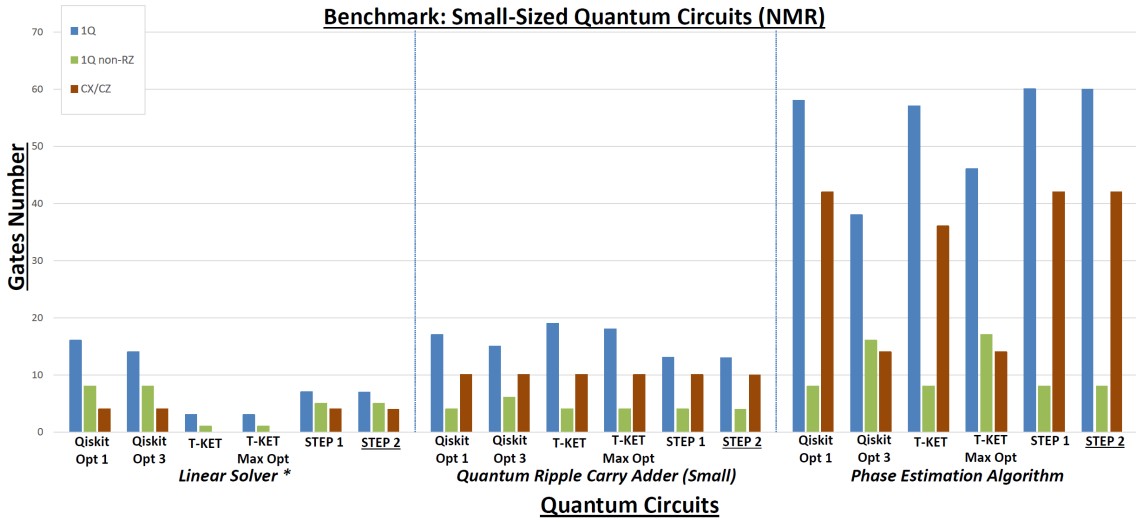


Figure 5.11: Benchmarks of gates number in small-sized circuits in Step 2 using NMR technology

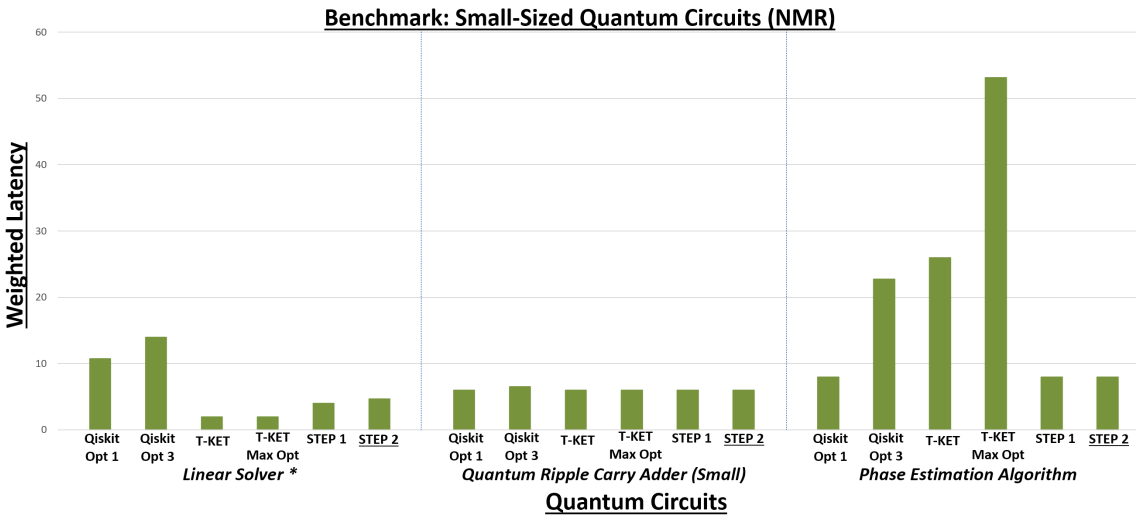


Figure 5.12: Benchmarks of latency in small-sized circuits in Step 2 using NMR technology

Medium-sized quantum circuits - NMR

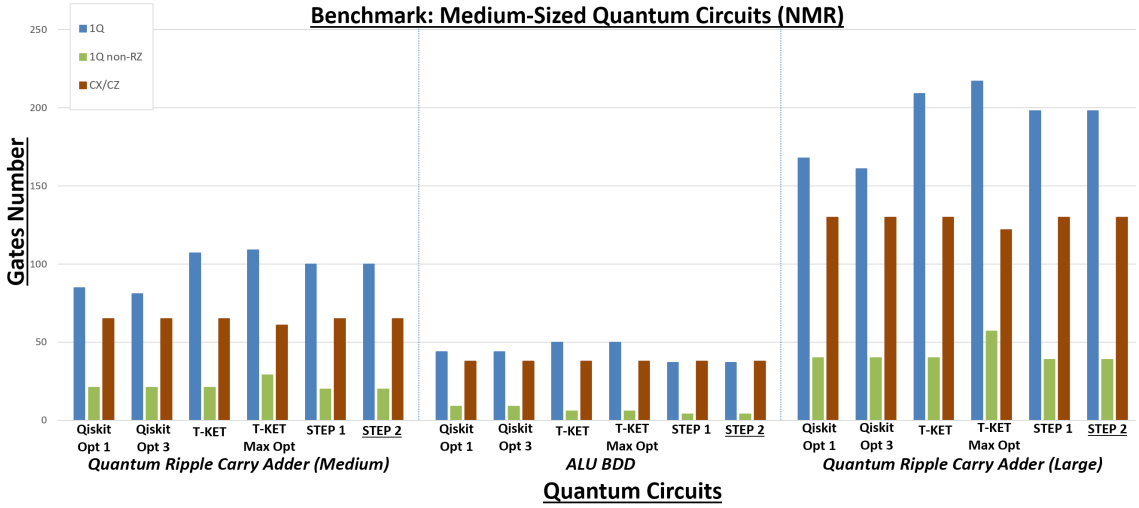


Figure 5.13: Benchmarks of gates number in medium-sized circuits in Step 2 using NMR technology

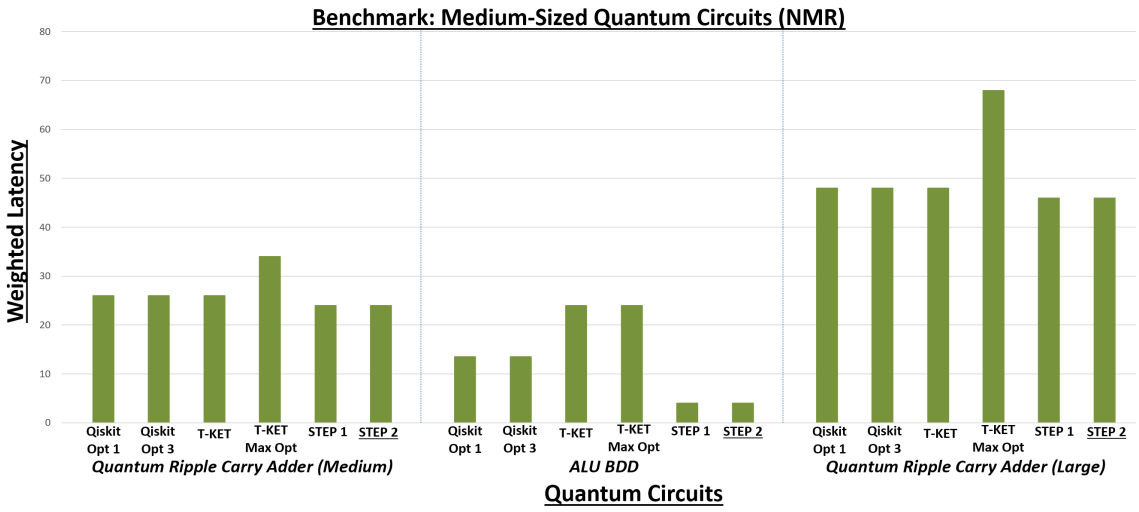


Figure 5.14: Benchmarks of latency in medium-sized circuits in Step 2 using NMR technology

Large-sized quantum circuits - NMR

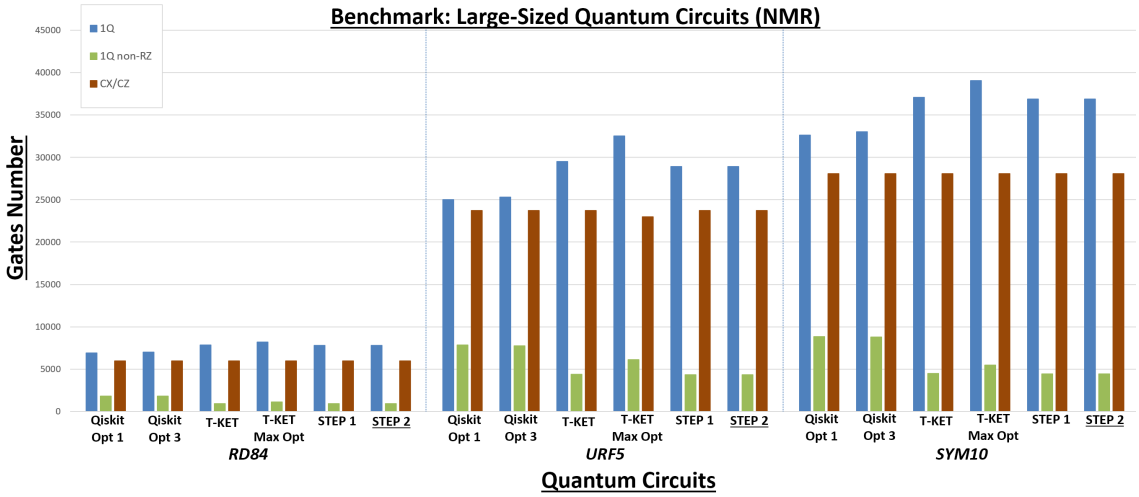


Figure 5.15: Benchmarks of gates number in large-sized circuits in Step 2 using NMR technology

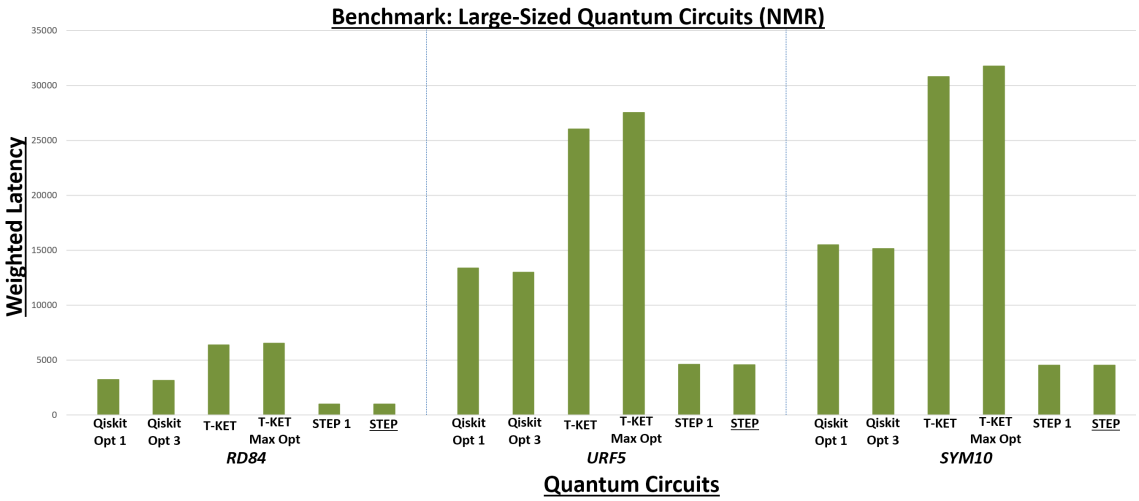


Figure 5.16: Benchmarks of latency in large-sized circuits in Step 2 using NMR technology

Small-sized quantum circuits - Trapped Ions

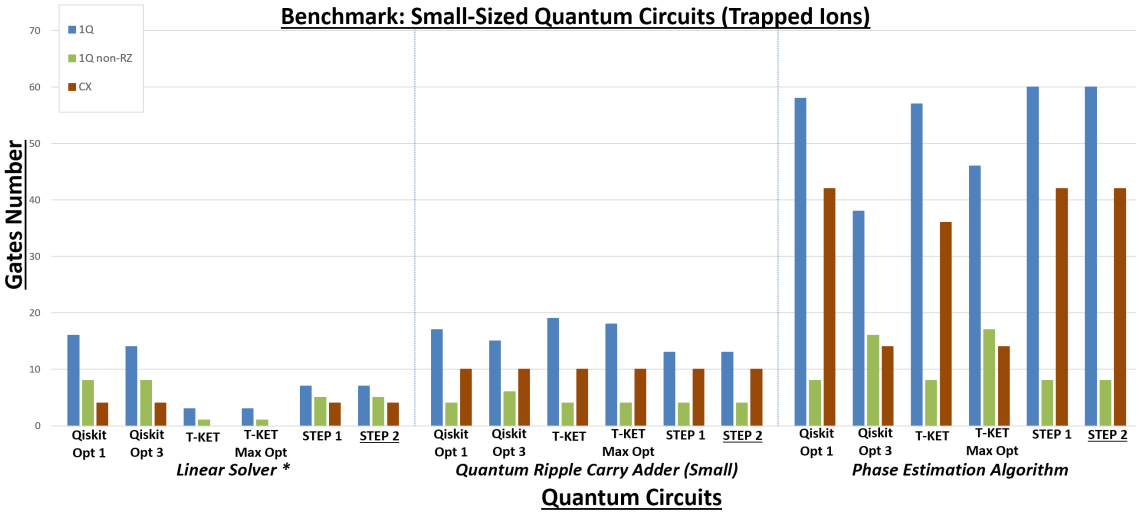


Figure 5.17: Benchmarks of gates number in small-sized circuits in Step 2 using Trapped Ions technology

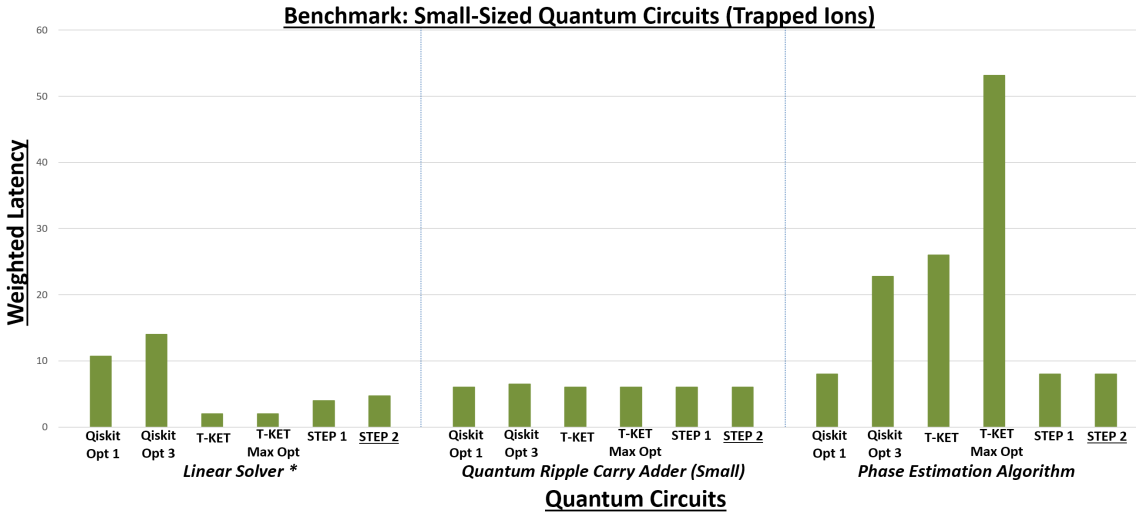


Figure 5.18: Benchmarks of latency in small-sized circuits in Step 2 using Trapped Ions technology

Medium-sized quantum circuits - Trapped Ions

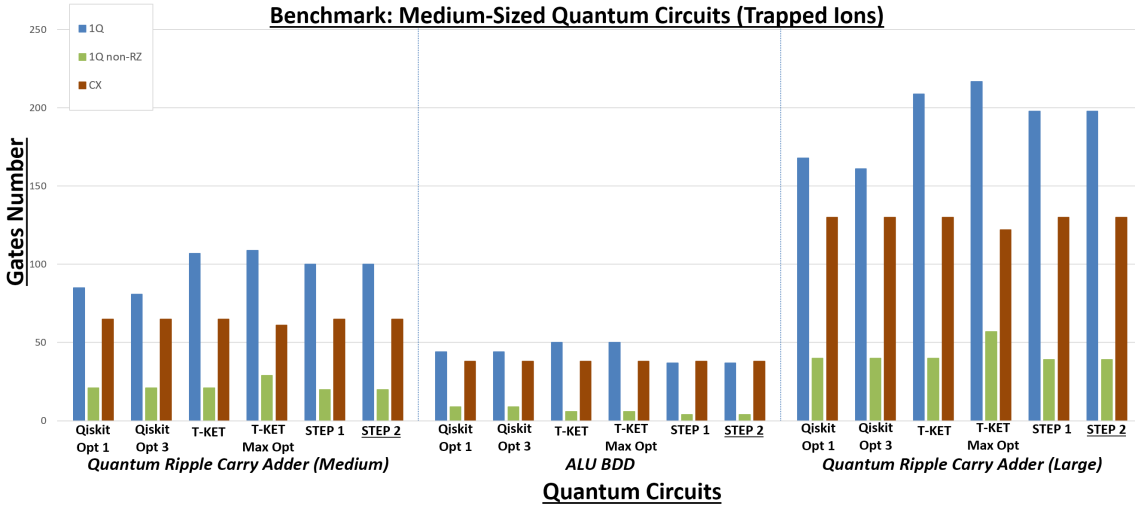


Figure 5.19: Benchmarks of gates number in medium-sized circuits in Step 2 using Trapped Ions technology

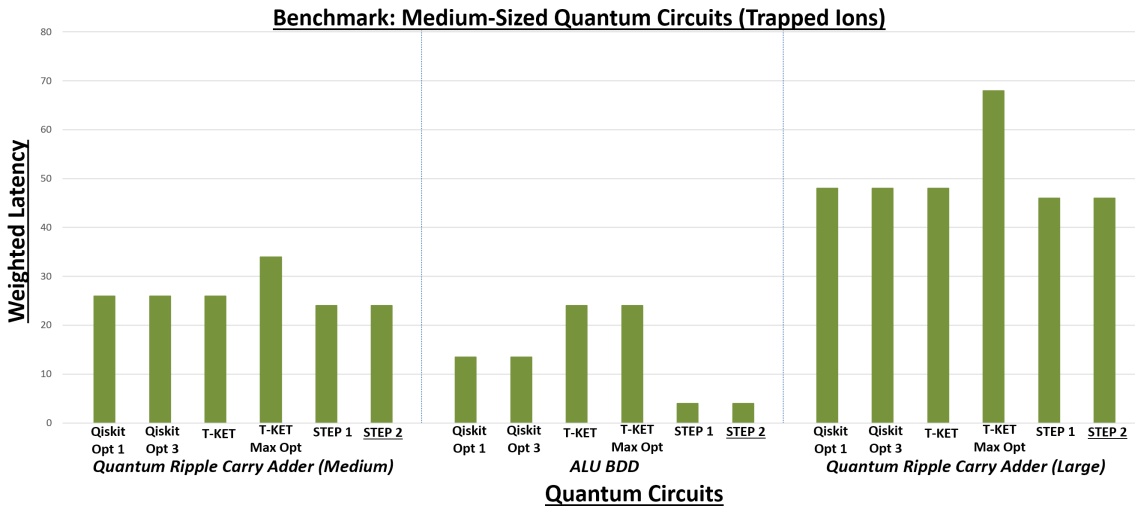


Figure 5.20: Benchmarks of latency in medium-sized circuits in Step 2 using Trapped Ions technology

Large-sized quantum circuits - Trapped Ions

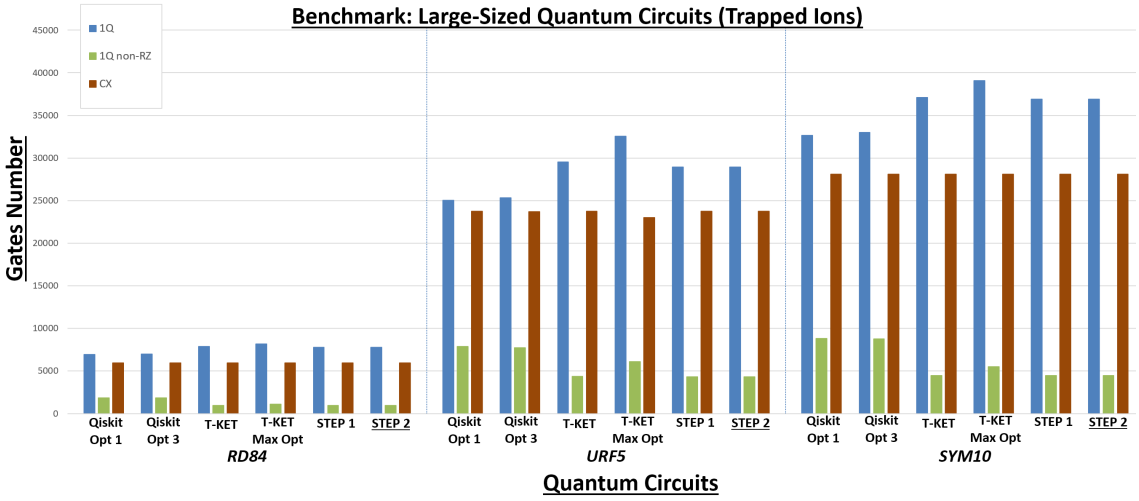


Figure 5.21: Benchmarks of gates number in large-sized circuits in Step 2 using Trapped Ions technology

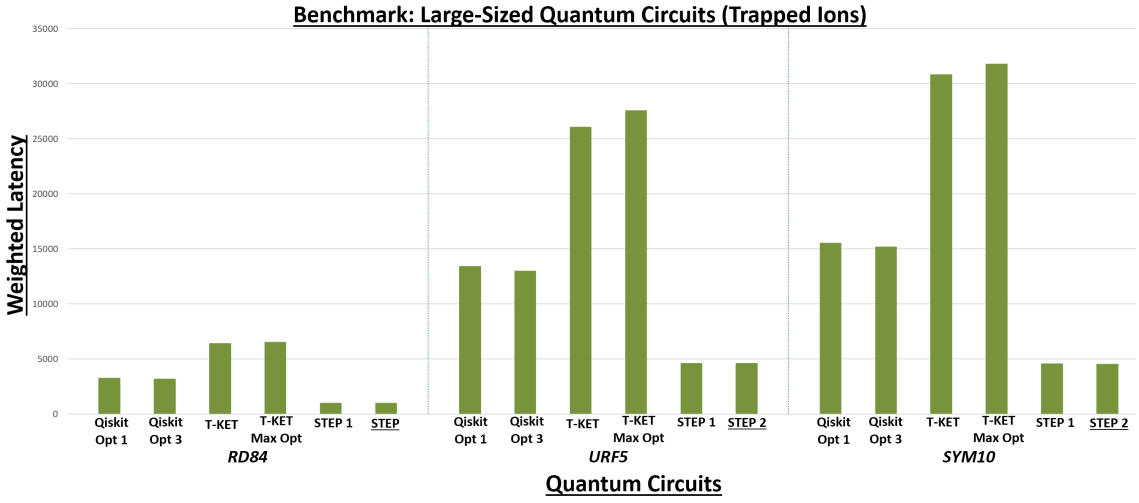


Figure 5.22: Benchmarks of latency in large-sized circuits in Step 2 using Trapped Ions technology

Small-sized quantum circuits - Superconducting

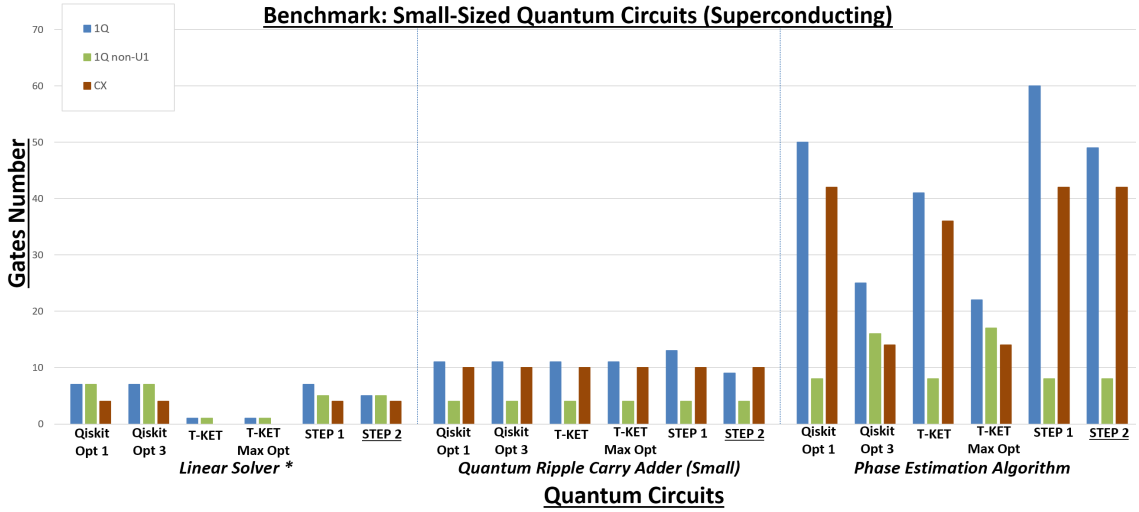


Figure 5.23: Benchmarks of gates number in small-sized circuits in Step 2 using Superconducting technology

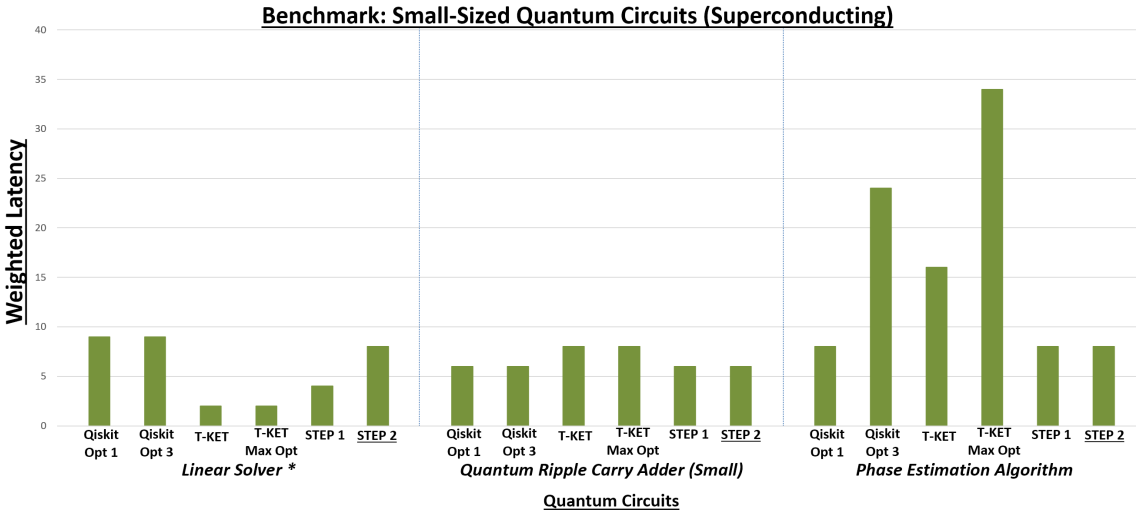


Figure 5.24: Benchmarks of latency in small-sized circuits in Step 2 using Superconducting technology

Medium-sized quantum circuits - Superconducting

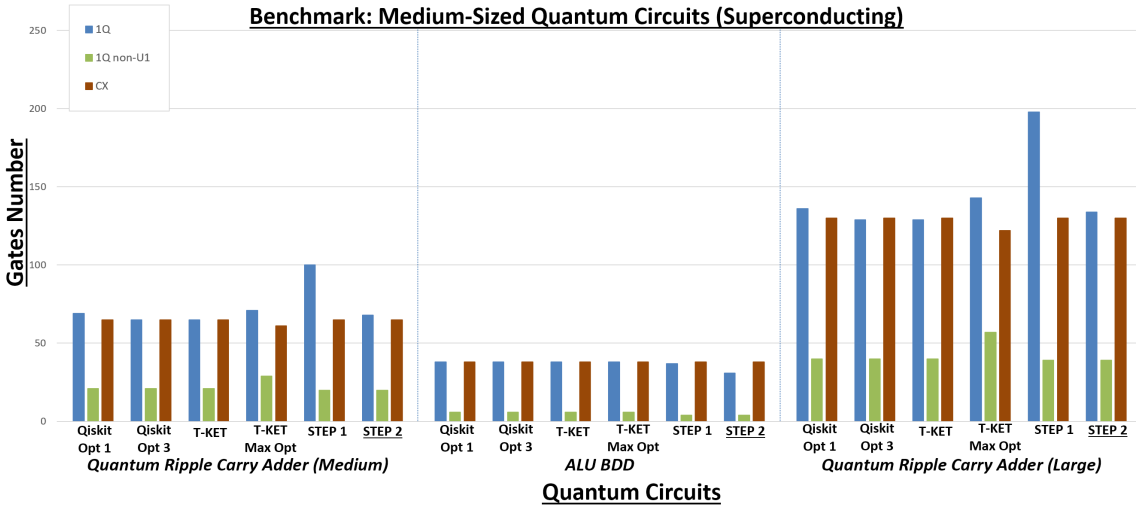


Figure 5.25: Benchmarks of gates number in medium-sized circuits in Step 2 using Superconducting technology

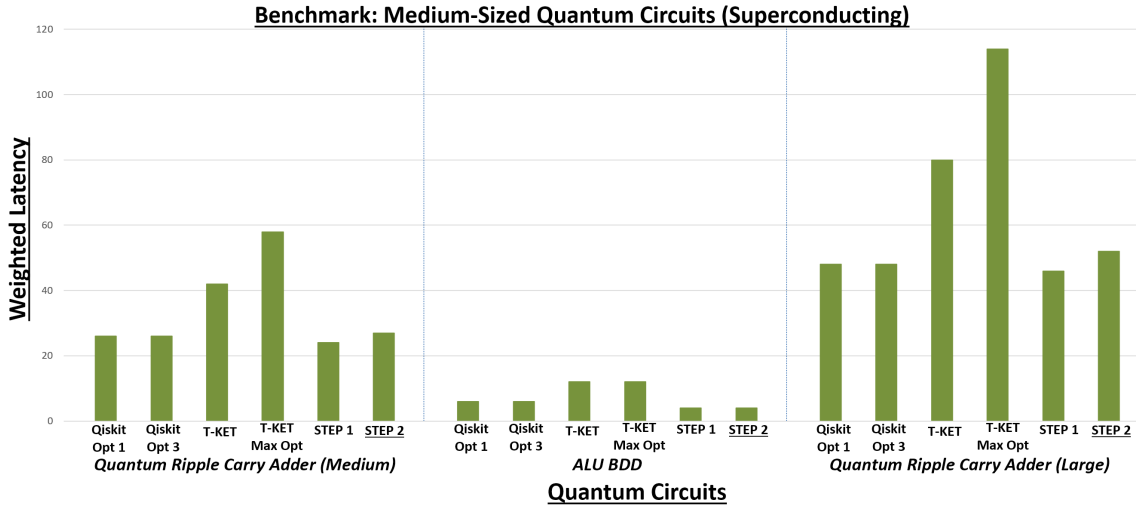


Figure 5.26: Benchmarks of latency in medium-sized circuits in Step 2 using Superconducting technology

Large-sized quantum circuits - Superconducting

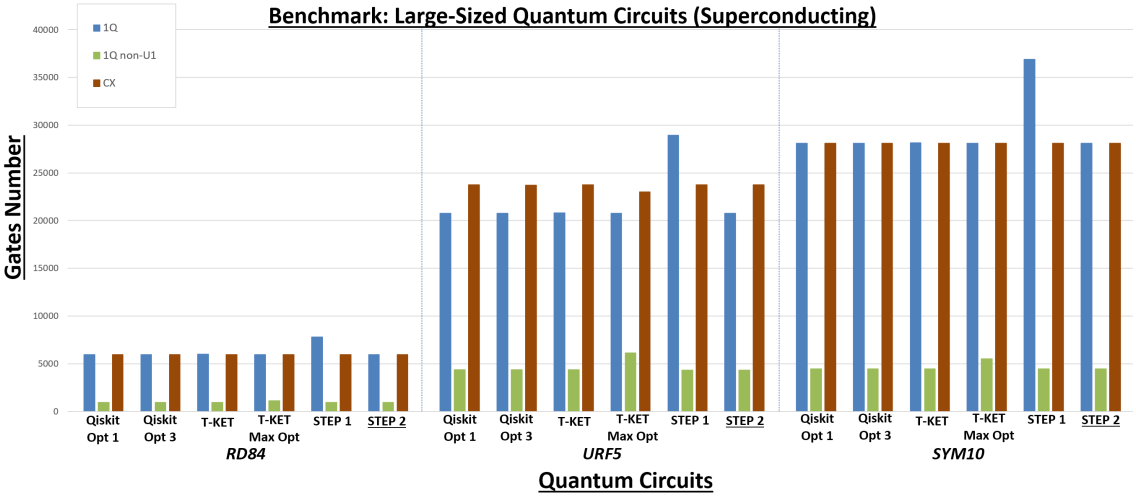


Figure 5.27: Benchmarks of gates number in large-sized circuits in Step 2 using Superconducting technology

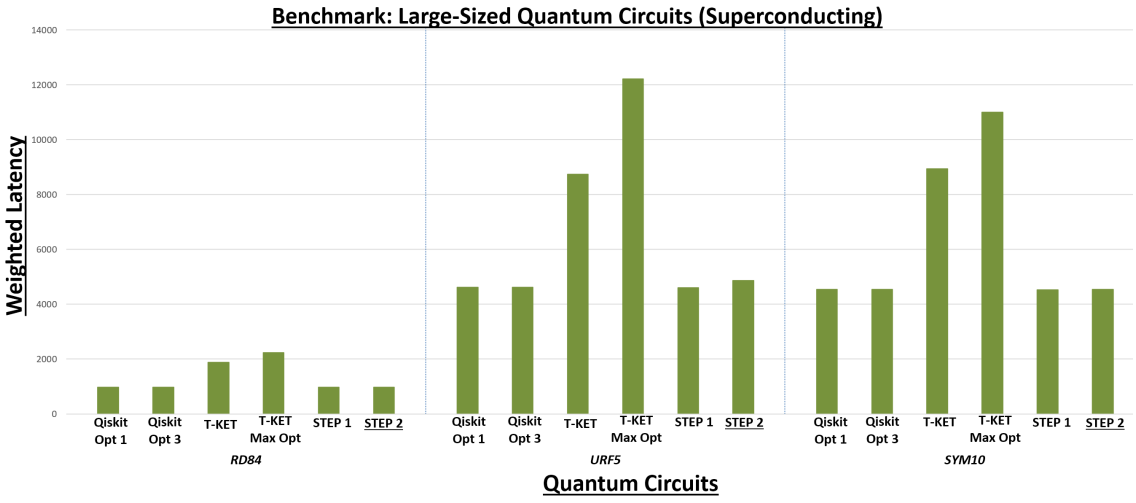


Figure 5.28: Benchmarks of latency in large-sized circuits in Step 2 using Superconducting technology

5.3.1 Intermediate results analysis

The results obtained in Step 2 were somewhat disappointing. In the **NMR and Trapped Ions technologies case**, Step 2 yields basically **no improvements when compared to Step 1**. While it is true that Step 1 is undoubtedly the step whose functions introduce the heaviest optimizations, it was estimated that a versatile tool such as the Eulercombo would mostly consolidate the results achieved in Step 1, and in some cases improve them even further. The tests of the Eulercombo on the randomized circuits were successful, and in every case achieved a drastic circuit compaction, by collapsing the streak of 10000 subsequent gates into a single triplet. This matched or outperformed the single-qubit streak management of the other two compilers. However, what Step 2 basically does in these two technologies is translating the circuit with negligible improvements. This is probably due to the fact that the Eulercombo optimization, albeit strong, is **very situational**: in simple circuits and in more sophisticated circuits using large numbers of two-qubit gates, the probability of having long streaks of single-qubit gates to be compacted is quite unlikely. For this reason, it was deemed as redundant to show the histograms regarding **weighted latency in NMR and Trapped Ions technologies**, since they are basically identical to the graphs represented in Section 5.2.
On the other hand, Step 2 performed really well when using the **Superconducting technology**, showing how the optimized merging scheme proposed in [55] is actually very efficient. This is proved by the fact that Step 2 was able to **outperform Qiskit** at "its own game" when using IBM's gate set. It must also be noted that, on the other hand, T-KET's optimization using this gate set proved **not that great**, because it tends to completely neglect the use of **U1 and U2 gates** when possible and resorts instead to a "brute force" translation into a large number of U3 gates that hampers the **overall circuit latency**. However, T-KET also features a **two-qubit gates optimization** which outperform even Qiskit.
The compilation time of Step 2 was non negligible in the case of large-sized circuits. This is mostly due to the fact that when using very precise approximations of π it is very easy when using the Eulercombo to incur into **Gimbal Locks**, which slow drastically the compilation.

5.4 Step 3 final benchmarks

In order to test Step 3, only circuits previously optimized by **Step 2** were used. During the tests, all qubits were considered once again as **ideal** and all architectures as **fully-connected**, with no specific backend device. However, this time, as explained in Chapter 4, the **decomposition of two-qubit gates** depended on the sign of interactions between certain qubits. Since having mixed sets of interaction signs was deemed as unnecessary for the purpose of evaluating the Toolchain's performance, these signs were "equalized" and the same decomposition was applied in each of the tested compilers. In other words, all the interaction signs in the .cfg files were set as **"1"** (except for the ones referring to the interaction of a qubit with itself of course, which were still equal to "0"). The **technology's legal set of gates** was adopted as **gate basis**, with the $\boldsymbol{U_J}$ **gate** of the **NMR technology** translated using the OpenQASM-supported **RZZ gate**:

- **NMR Technology: $\boldsymbol{R_X}$, $\boldsymbol{R_Y}$, $\boldsymbol{R_Z}$ and RZZ gates**
- **Trapped Ions Technology: $\boldsymbol{R_X}$, $\boldsymbol{R_Y}$, $\boldsymbol{R_Z}$ and RXX gates**
- **Superconducting Technology: U1, U2, U3 and CX gates**

In **T-KET**, the decompositions were allowed when generating the optimized circuit were set as the same one used by the Toolchain and represented in Figure 1.5, 1.7. In **Qiskit**, since **Qiskit Terra** does not support circuit decomposition using only **RZZ gates** as multi-qubit gates, a **decomposition in CZ gates** was applied instead, and in the gate count each CZ was counted as a RZZ gate and two R_Z gates. This was a good approximation, since inserted R_Z gates do not influence the overall latency of single-qubit gates anyway.
To evaluate the single-qubit gates' overall weighted latency, in the **NMR and Trapped Ions technology cases** it was decided that each **non-$\boldsymbol{R_Z}$ gate** with a rotation parameter of $\boldsymbol{\theta}$ would introduce a latency equal to $\mathbf{2}\frac{|\boldsymbol{\theta}|}{\boldsymbol{\pi}}$. Hence, gates with a rotation of π introduce a weighted latency equal to "2" and gates with a rotation of $\frac{\pi}{2}$ introduce a weighted latency equal to "1". The calculated latencies were approximated at two decimals. In the **Superconducting technology case**, a more straightforward approach was used instead, and the weighted latency was

calculated by assuming that each **U2 gate** introduces a latency equal to **"1"** and that each **U3 gate** introduces a latency equal to **"2"**. This criteria was used to emphasize the fact that an implementation using a U3 gate last **twice** the time of an implementation using a U2 gate. R_Z and U1 gates were not considered because a virtual instantaneous implementation is assumed.

Small-sized quantum circuits - NMR

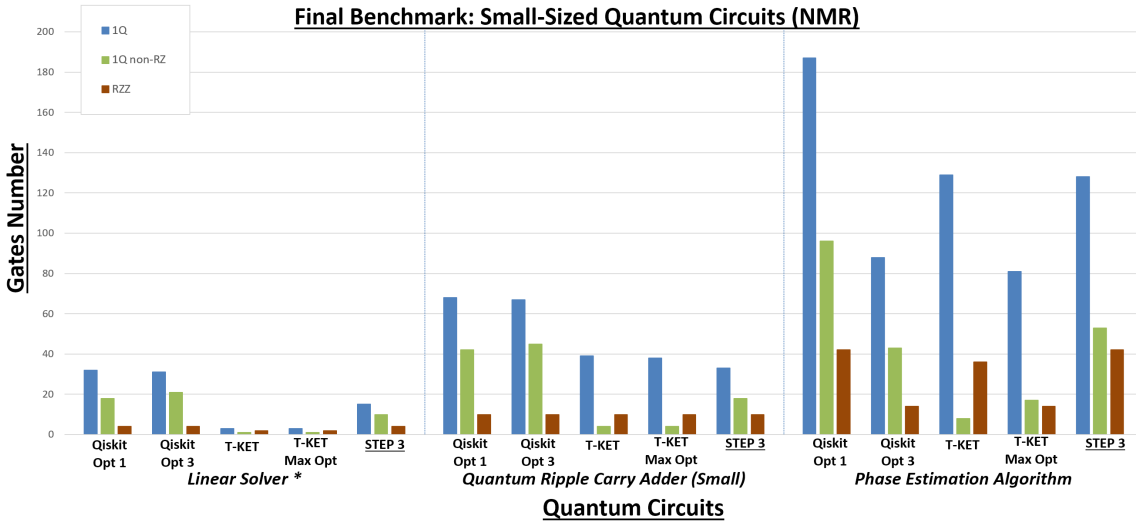


Figure 5.29: Final benchmarks of gates number in small-sized circuits in Step 3 using NMR technology

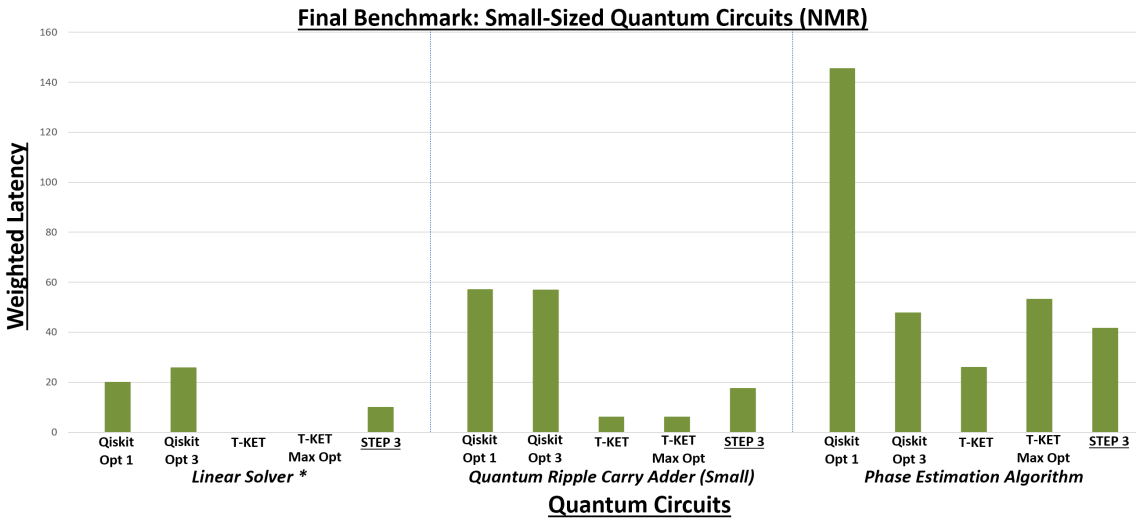


Figure 5.30: Final benchmarks of latency in small-sized circuits in Step 3 using NMR technology

Medium-sized quantum circuits - NMR

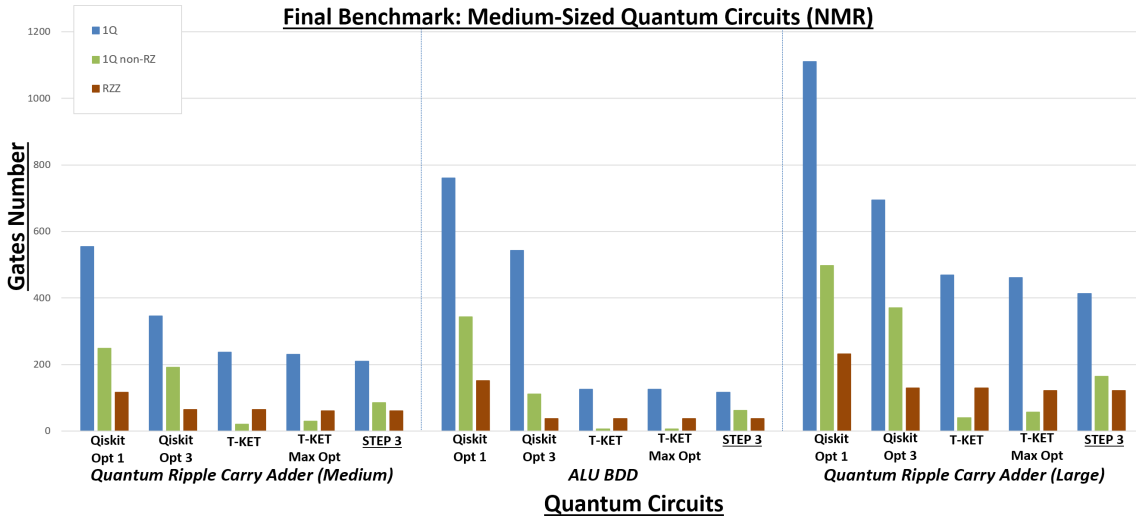


Figure 5.31: Final benchmarks of gates number in medium-sized circuits in Step 3 using NMR technology

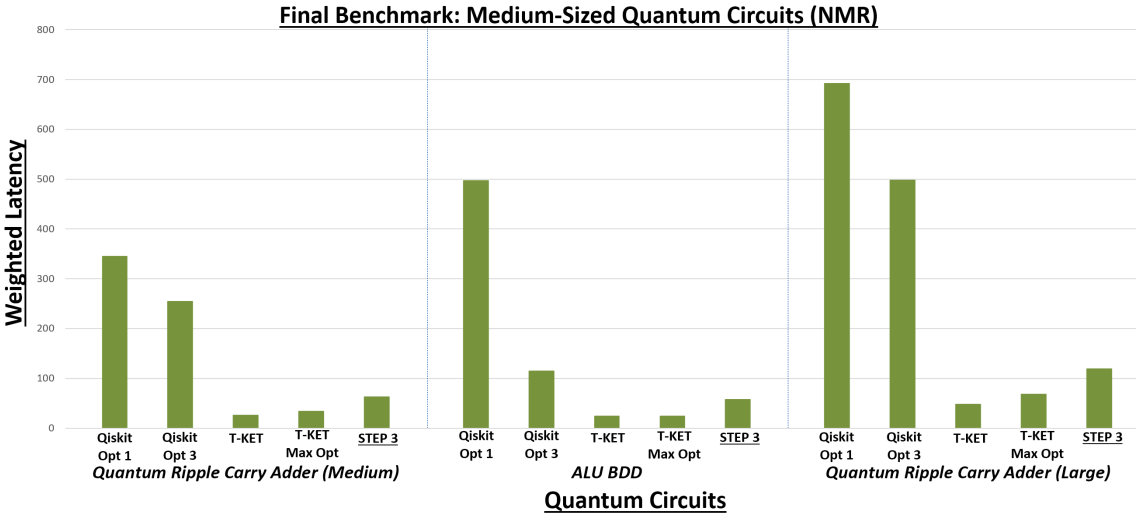


Figure 5.32: Final benchmarks of latency in medium-sized circuits in Step 3 using NMR technology

Large-sized quantum circuits - NMR

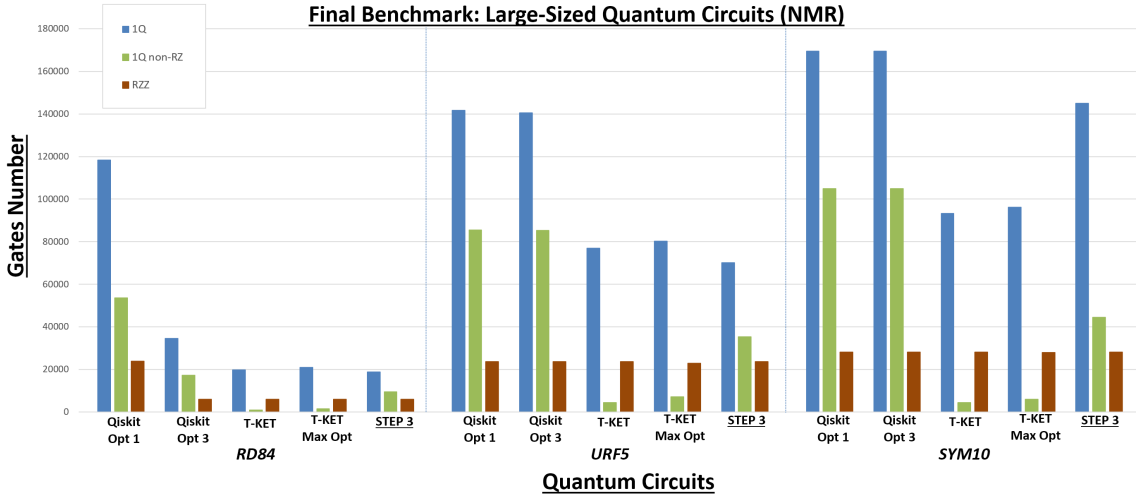


Figure 5.33: Final benchmarks of gates number in large-sized circuits in Step 3 using NMR technology

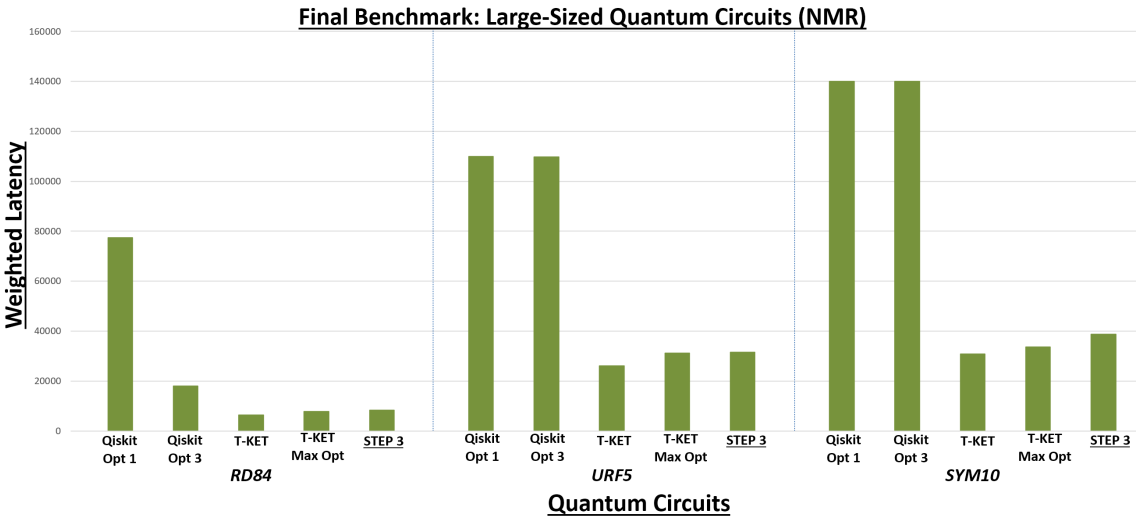


Figure 5.34: Final benchmarks of latency in large-sized circuits in Step 3 using NMR technology

Small-sized quantum circuits - Trapped Ions

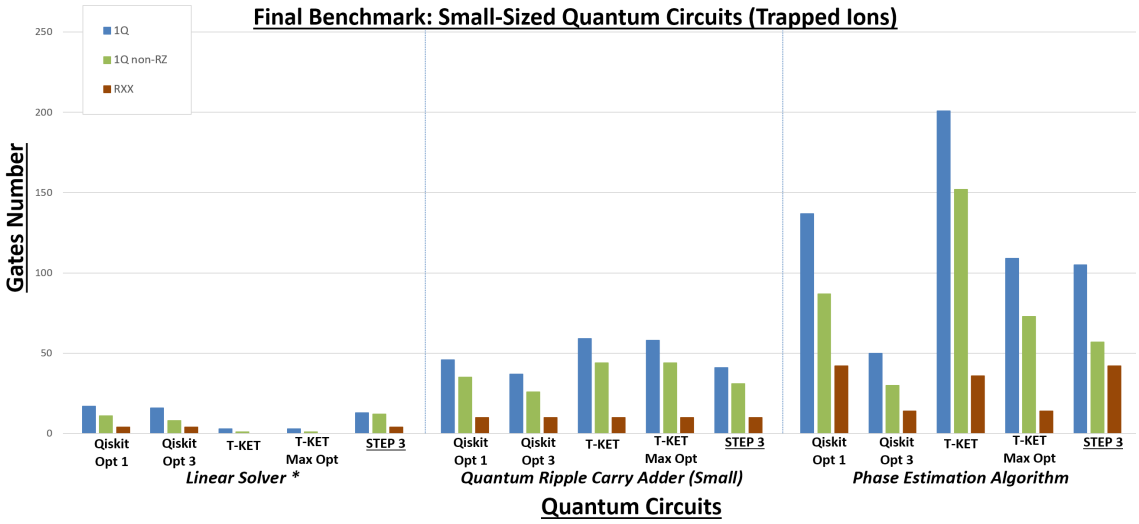


Figure 5.35: Final benchmarks of gates number in small-sized circuits in Step 3 using Trapped Ions technology

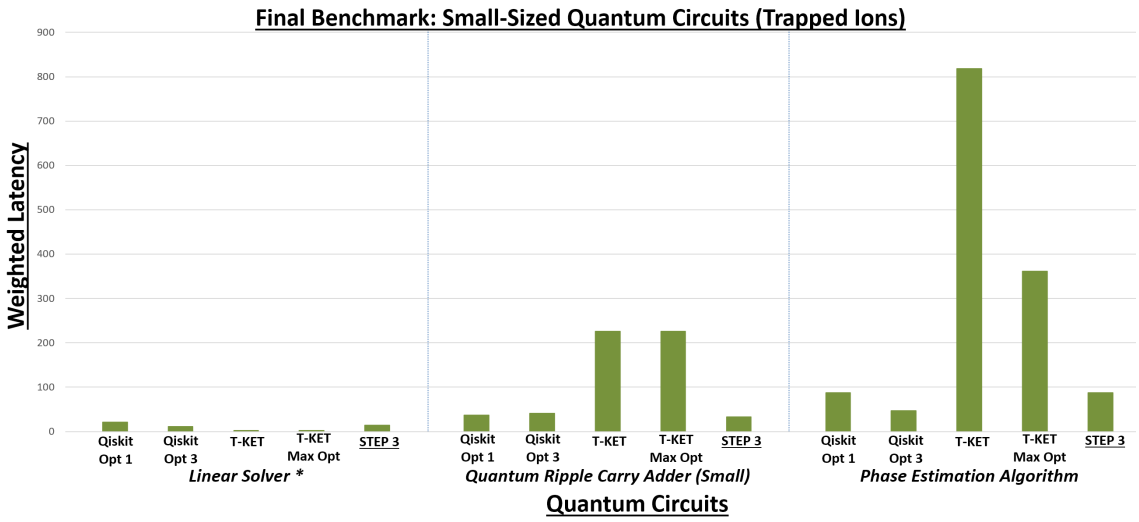


Figure 5.36: Final benchmarks of latency in small-sized circuits in Step 3 using Trapped Ions technology

Medium-sized quantum circuits - Trapped Ions

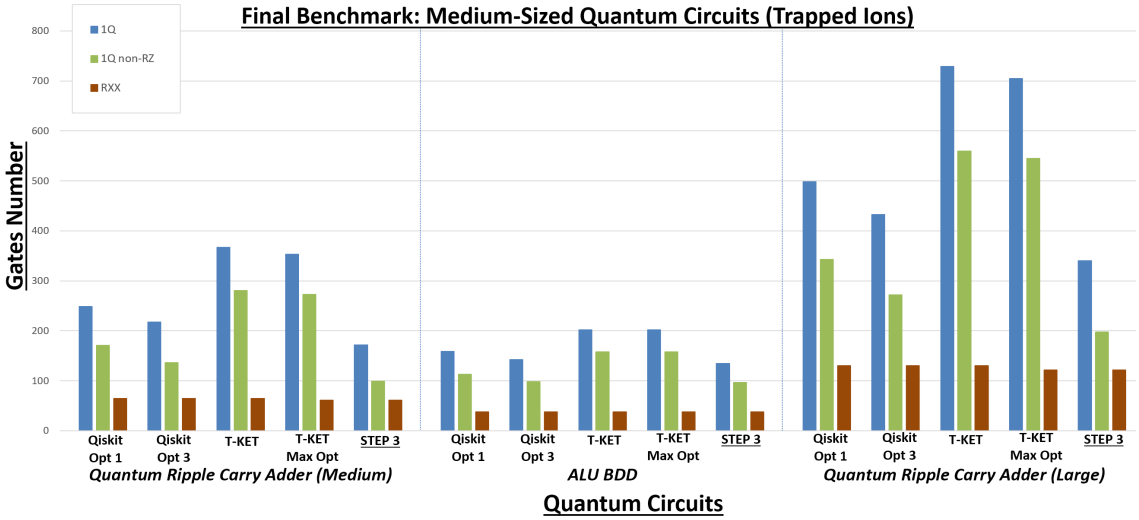


Figure 5.37: Final benchmarks of gates number in medium-sized circuits in Step 3 using Trapped Ions technology

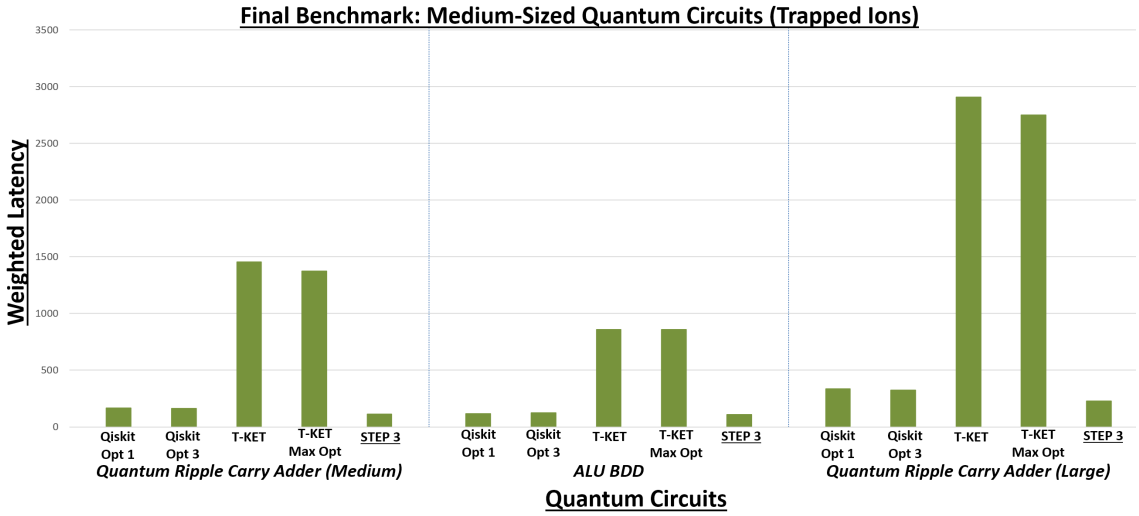


Figure 5.38: Final benchmarks of latency in medium-sized circuits in Step 3 using Trapped Ions technology

Large-sized quantum circuits - Trapped Ions

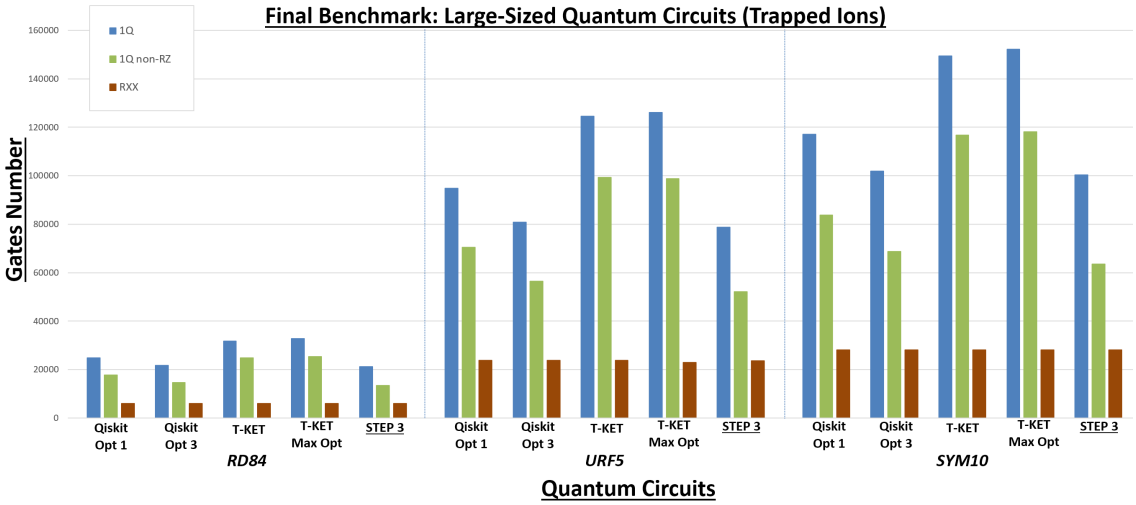


Figure 5.39: Final benchmarks of gates number in large-sized circuits in Step 3 using Trapped Ions technology

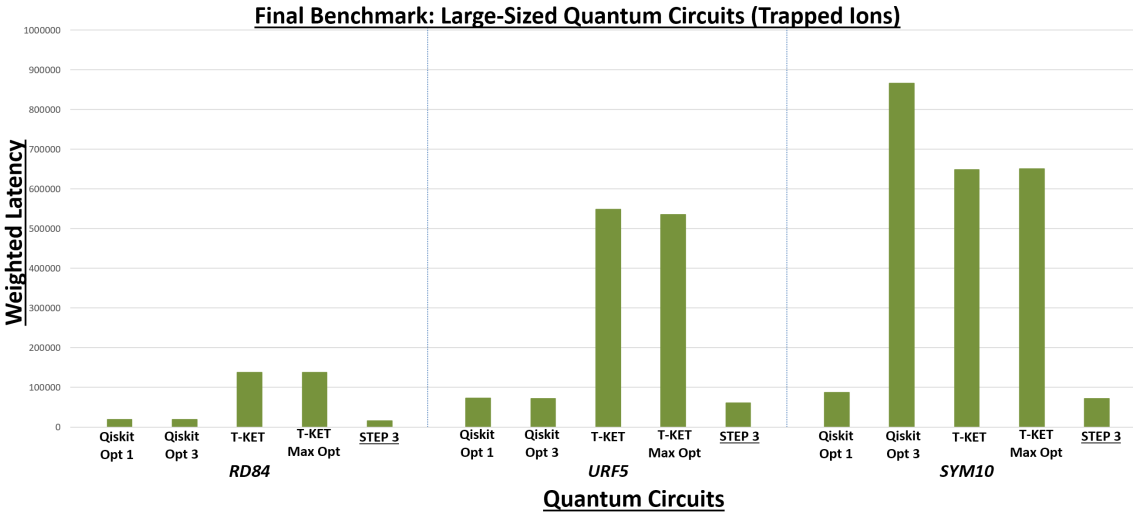


Figure 5.40: Final benchmarks of latency in large-sized circuits in Step 3 using Trapped Ions technology

Small-sized quantum circuits - Superconducting

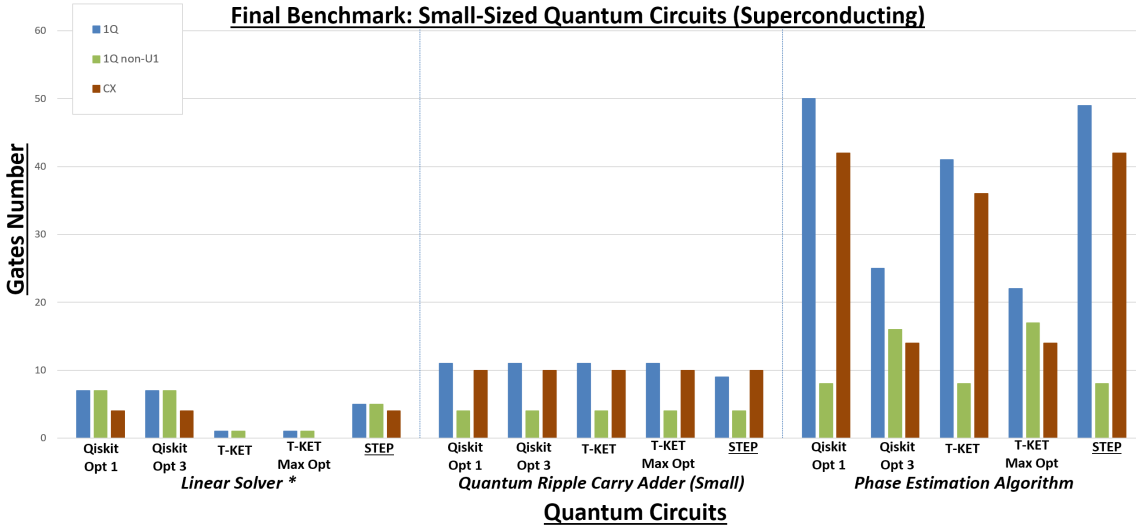


Figure 5.41: Final benchmarks of gates number in small-sized circuits in Step 3 using Superconducting technology

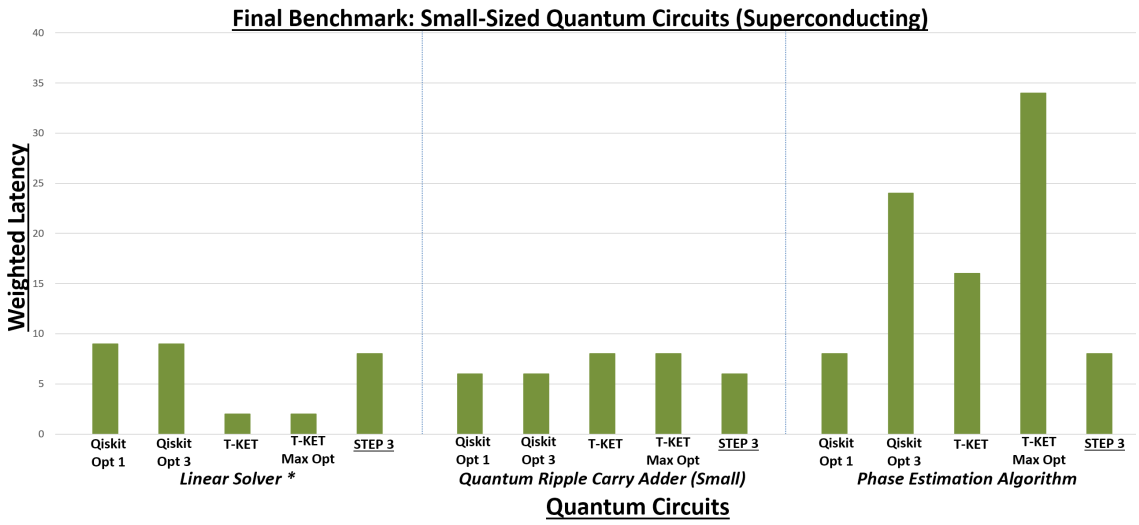


Figure 5.42: Final benchmarks of latency in small-sized circuits in Step 3 using Superconducting technology

Medium-sized quantum circuits - Superconducting

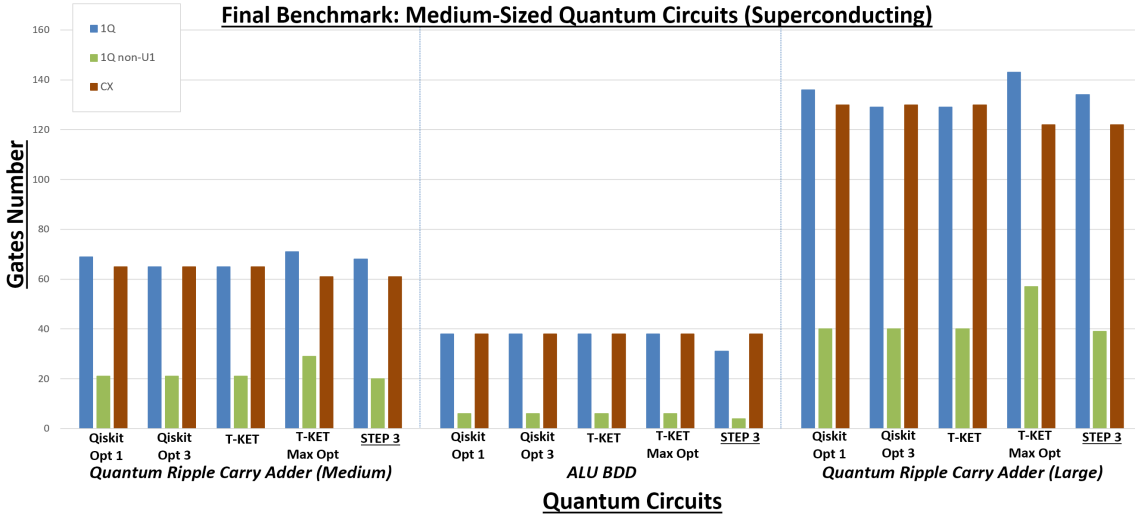


Figure 5.43: Final benchmarks of gates number in medium-sized circuits in Step 3 using Superconducting technology

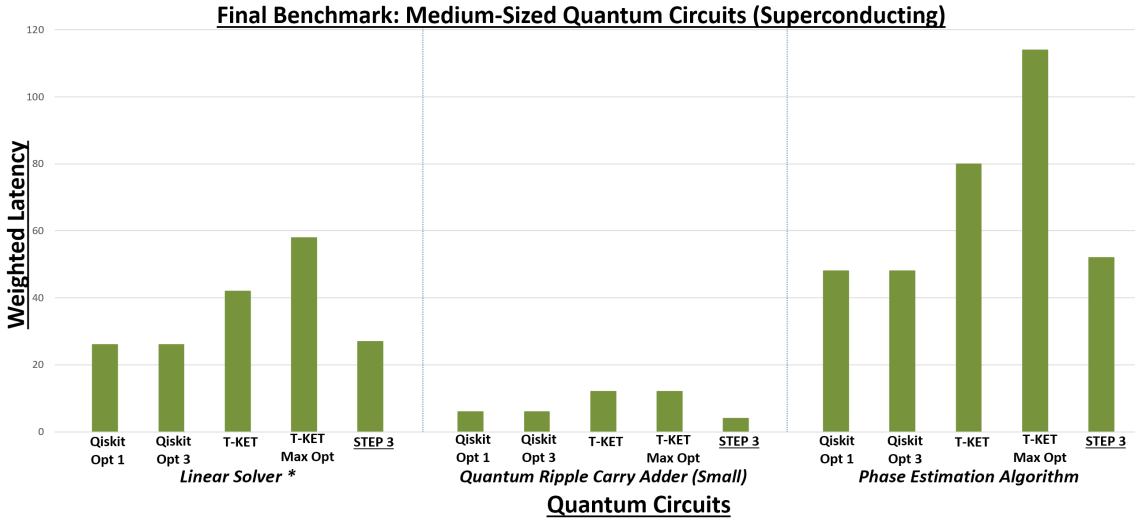


Figure 5.44: Final benchmarks of latency in medium-sized circuits in Step 3 using Superconducting technology

Large-sized quantum circuits - Superconducting

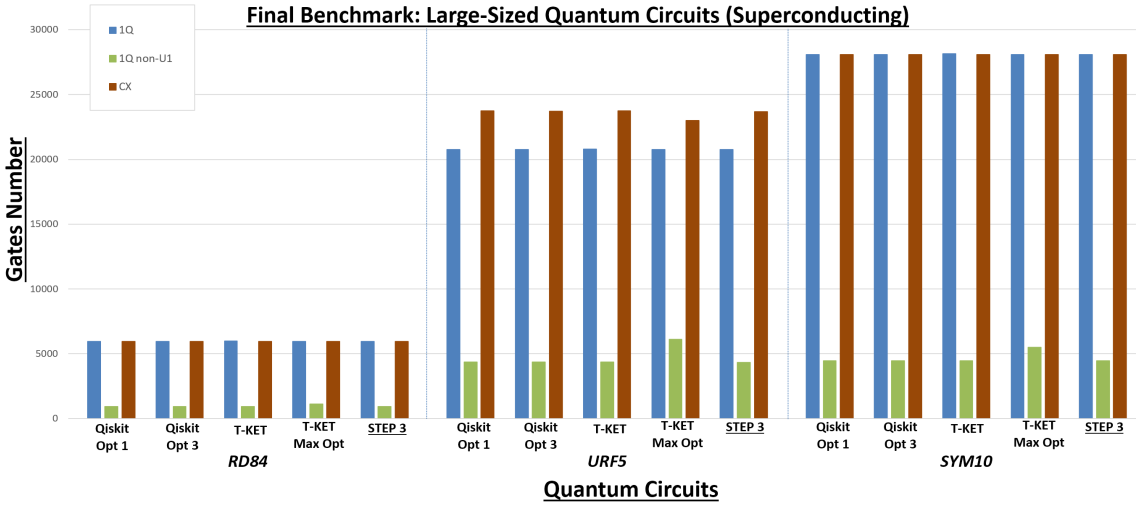


Figure 5.45: Final benchmarks of gates number in large-sized circuits in Step 3 using Superconducting technology

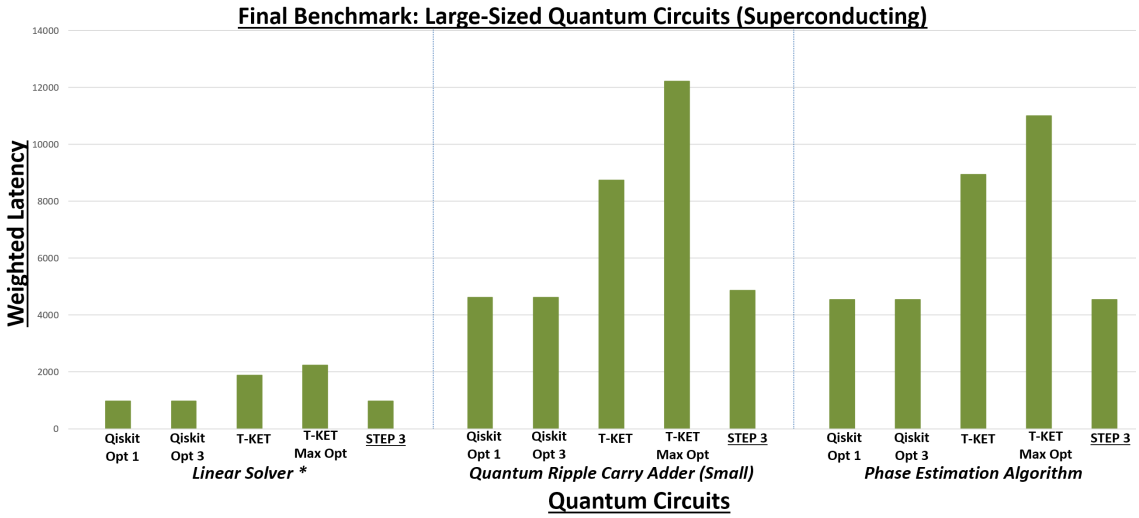


Figure 5.46: Final benchmarks of latency in large-sized circuits in Step 3 using Superconducting technology

5.4.1 Final results analysis

Generally speaking, the final results obtained were **promising** and surprising in several ways. The most unexpected thing was that the Step 3 proved capable of **handling two-qubit gates quite well**, introducing optimizations capable of outperforming Qiskit and, most importantly, of **competing with T-KET** in most situations, the very T-KET whose strong point is actually the capability of reducing the number of multi-qubit gates involved in the circuit. This went against the expectation of the template-based approach being unsuited to tackle efficiently the management of clusters of CX gates, as the templates that involve them are not numerous and as other methods, such as the heuristic-based ones, seem theoretically more prone to detect advantageous circuit restructurations. As expected, the number of single-qubit gates **drammatically increases** in some cases, since each decomposition of two-qubit gates introduces several single-qubit gates in the circuit and can bring to a lot of new inserted gates in large-sized circuits. The insertion of the new single-qubit gates also makes the Eulercombo optimization more useful, as it makes long streaks of adjacent single-qubit less rare.
The specific results obtained for each technology are:

- **NMR technology:** Of all the supported technologies, the generated circuits with the NMR as target technology proved to be the ones in which the Toolchain's optimization **is least competitive**. When compared to **Qiskit**, the Toolchain performs really well, introducing a significant reduction on **both single-qubit and two-qubit gates**. On the other hand, **T-KET completely outperforms the other compilers, and by far**. While its optimizations regarding **two-qubit gates** prove once again to be the best ones, they are generally not that different from the ones the Toolchain is capable of implementing. When managing **single-qubit gates**, though, T-KET's optimization is unrivaled and boasts a reduction of gates and a readaptation of non-R_Z gates into R_Z gates so strong that both Qiskit and the Toolchain do not come even close to competing with it. While the Toolchain performance with this technology is still good, the benchmarks (Figure from 5.29 to 5.34) showed that in no way the NMR technology would perform better on the currently-implemented Toolchain than on other state-of-the-art compilers. It

has to be noted that the NMR technology was considered as fully-connected, but that it might not always be. As higlighted in Section 1.3.2, in some cases the J coupling between two qubits might be too weak to allow an actual two-qubit gate interaction. This issue was not taken in consideration, but it could easily be resolved by adding new handling cases based on the **NMR Layout parameter**.

- **Trapped Ions technology:** The benchmarks (Figure from 5.35 to 5.40) showed that the Trapped Ions technology is the one managed **most successfully** in the Toolchain. Along with excellent improvements on the number of two-qubit gates in the circuit, the Toolchain achieves a consistent reduction of the number of single-qubit gates and significantly **outperforms both the other compilers** in terms of overall single-qubit gates, non-R_Z single-qubit gates and weighted latency. When using this target technology, **Qiskit** is capable of holding its ground quite well, while the same cannot be said for **T-KET**, which features a lackluster implementation: even if it manages as well as always two-qubit gates, it is, in fact, incapable of severely reducing single-qubit gates, and it also employs non-R_Z gates with rotation parameters such that the overall weighted latency results very high. This results in both a **generated circuit gate count and overall latency** ensured by the Toolchain which is far superior with respect to the other compilers' work.

- **Superconducting technology:** The Toolchain performs **really well** when using this target technology. In Step 3, all of Step 2's significant improvements are consolidated with the attainment of a good reduction of two-qubit gates in the circuit. In this technology, as seen in Step 2, **T-KET** is able to achieve a reduction of two-qubit gates all cross the chart while at the same time managing in a suboptimal way the single-qubit U gates; **Qiskit** handles instead IBM's gate set really well in terms of single-qubit gates number, but it is not capable of competing with T-KET in CX optimization. After Step 3, the Toolchain **takes the best of both worlds**, outperforming Qiskit in the management of U gates and nearly matching T-KET in CX optimizations. The benchmarks (Figure from 5.41 to 5.46) show that latency-wise the Toolchain

basically matches Qiskit's optimization, but by using less gates. Generally-speaking, the advantages over the other compilers' results are not very strong, but are still significant. However, it has to be noted that, in this specific technology, the critical layout mapping phase was not taken in consideration, and that the smart management of SWAP gates to adapt circuits to target devices is one of Qiskit's and T-KET's strong point: perhaps, with such feature implemented the Toolchain would prove less ideal than Qiskit, T-KET or both.

In the Superconducting technology case, the compilation time of Step 3 was non negligible in the case of large-sized circuits, but not long. Combining the running times of all the steps, the Toolchain takes longer than both Qiskit and T-KET, but in small to average circuits this difference is negligible and is still acceptable in large circuits. In the NMR and Trapped Ions technologies cases, the compilation time of Step 3 was generally speaking significant and absolutely crippling when dealing with large-sized circuits. This is probably due to the employment of the second Eulercombo mechanism in a circuit whose number of gates drammatically increased after the decomposition of CX and CZ gates: what would be a long but acceptable compilation, once very large circuits are decomposed, snowballs into an hours-long process.
When considering the overall running time of the Toolchain, it is clear that the template-based approach can be fast enough for small to average circuits, but that it is also extremely slower than other heuristic methods when dealing with very large circuits.

Conclusions and future perspectives

The main goal of this thesis was to create a prototype of a toolchain capable of optimally compiling quantum circuits for more target quantum technologies. As main principle of the Toolchain's inner workings, it was deemed as interesting to explore a less conventional template-based approach in the optimization process.
The Toolchain was developed using modular libraries in order to allow expansions and future support for other features and technologies, and it was divided in steps to allow the exploiting of specific layers of the optimization process. During and after development, the Toolchain was tested by using small to large-sized quantum circuits as inputs and its results, both final and intermediate, were compared to two of the state-of-the-art most efficient compilers.
The results obtained show that the Toolchain and its core philosophy are actually competitive in the state-of-the-art, and that the designed optimization process can introduce some fine-grain technology-dependent optimizations that allow the Toolchain to outperform its competitors, especially when dealing with single-qubit gates. It is, in fact, capable of steadily reduce the number of these gate, and to prioritize an abundant use of advantageous and virtually-implementable R_Z gates. What the Toolchain prototype disappoints in is the overall compilation time, that in certain technologies becomes an unbearable factor when dealing with very large circuits. The largest circuits used in the benchmarks are probably too large in scale for actual quantum computers to be able to handle them and actually use them for computation, but still, this shows a limit of some of the proposed optimizations, that are easily prone to trigger very long computations in extreme cases when compared to other compilers.

As reported multiple times through this thesis, the Toolchain is, however, still a prototype. Its structure could allow it to support even more quantum technologies, enhancing its already good versatility.

The most important part currently missing in the Toolchain is the capability of handling the whole Layout Synthesis process for non-fully-connected technologies such as the Superconducting one. This would require to implement a tool capable of adapting the compilation to a given device's layout, and to logically map each qubit line with a smart insertion of SWAP gates. It is unlikely for the template-based approach to be competitive in this kind of quantum gates management when compared to other more "farsighted" approaches, but it could still prove good enough. The way to implement this device-mapping could easily be inserted in the specific part of the Toolchain's Step 3. Multiple ways of implementing the method with which to define the target device could be devised, from using the existing .cfg files infrastructure, to other more radical reshapes of the process. Moreover, in order to improve the compilation other steps could be implemented in the Toolchain. Existing steps may also be modified in order to accommodate new features and currently-implemented functions could be moved in the Toolchain to improve its overall process. Last but not least, some precautions could be taken to specifically reduce compilation time.

Even if this Toolchain and all the potential works inspired from it will not be considered as worthy to supersede other commonly employed optimization mechanism, or even if they will eventually become obsolete, the perspective and analysis offered by this thesis work about the usage of non-heuristic methods tailored on the intrinsic dynamics of quantum technologies and about the approach of balancing versatility and technology-specific efficiency in optimization processes, will remain one of the many steps forward in the direction of the paradigm-changing quantum advantage.

Bibliography

[1] Yanofsky, N., Mannucci, M. (2008). *Quantum Computing for Computer Scientists*. Cambridge: Cambridge University Press. DOI:10.1017/CBO9780511813887

[2] Hidary, J. (2019). *Quantum Computing: An Applied Approach*. Springer Nature. DOI:10.1007/978-3-030-23922-0

[3] Simoni Mario. *Modelling Molecular Technologies for Nuclear Magnetic Resonance Quantum Computing*. Consultable at https://webthesis.biblio.polito.it/14446/, Politecnico di Torino, April 2020

[4] Raggi Lorenzo. *Arithmetic circuits for quantum computing: a software library*. Consultable at https://webthesis.biblio.polito.it/15853/, Politecnico di Torino, October 2020

[5] Eastin Bryan, Flammia Steven T. *Q-Circuit Tutorial*. Department of Physics and Astronomy, University of New Mexico. Online at https://physics.unm.edu/CQuIC/Qcircuit/Qtutorial.pdf

[6] D. P. DiVincenzo et al. *The Physical Implementation of Quantum Computation*. arXiv preprint quant-ph/0002077, 2000.

[7] T. S. Humble et al. *Quantum Computing Circuits and Devices*. arXiv preprint 1804.10648, 2018

[8] G. A. Cirillo, G. Turvani and M. Graziano. *A Quantum Computation Model for Molecular Nanomagnets*. in IEEE Transactions on Nanotechnology, vol. 18, pp. 1027-1039, 2019, DOI:10.1109/TNANO.2019.2939910.

[9] T.P. Harty et al. Phys. Rev. Lett.113, 220501 (2014) DOI:10.1103/PhysRevLett.113.220501

[10] C. D. Bruzewicz et al. *Trapped-Ion Quantum Computing: Progress and Challenges*. arXiv preprint 1904.04178, 2019

[11] C. Figgatt. *Building and Programming a Universal Ion Trap Quantum Computer*. DOI:10.13016/M2K35MH5F, 2018

[12] A. C. Lee et al. *Engineering Large Stark Shifts for Control of Individual Clock State Qubits*. arXiv preprint 1604.08840, 2016

[13] A. Sorensen, K. Molmer. *Quantum computation with ions in thermal motion*.

arXiv preprint quant-ph/9810039, 1999

[14] Arute, F., Arya, K., Babbush, R. et al. *Quantum supremacy using a programmable superconducting processor*. Nature 574, 505-510, DOI:10.1038/s41586-019-1666-5, 2019

[15] H. L. Huang, D. Wu, D. Fan, X. Zhu. *Superconducting Quantum Computing: A Review*. arXiv preprint 2006.10433, 2020

[16] J.M. Gambetta. *IBM's Roadmap For Scaling Quantum Technology* Online 15/09/2020 at https://www.ibm.com/blogs/research/2020/09/ibm-quantum-roadmap/

[17] *Architecture and Design Automation for Quantum Computing* Online at https://vast.cs.ucla.edu/projects/architecture-and-design-automation-quantum-computing University of California, Los Angeles

[18] QMBlochSphere.svg by CSTAR Online 20/06/2007 at https://commons.wikimedia.org/wiki/

[19] S. Bravyi, A. Kitaev. *Universal Quantum Computation with ideal Clifford gates and noisy ancillas*. arXiv preprint quant-ph/0403025, 2004

[20] A. Barenco, D.P. DiVincenzo, P. Shor et al. *Elementary gates for quantum computation*. arXiv preprint quant-ph/9503016, 1995

[21] D.P. DiVincenzo *Two-Bit Gates are Universal for Quantum Computation*. arXiv preprint cond-mat/9407022, 1994

[22] D.C. McKay, C.J. Wood, S. Sheldon, J.M. Chow, J.M. Gambetta. *Efficient Z-Gates for Quantum Computing*. arXiv preprint 1612.00858, 2016

[23] G. A. Cirillo, G. Turvani, M. Simoni, M. Graziano. *Advances in Molecular Quantum Computing: from Technological Modeling to Circuit Design*. 2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), 2020, pp. 132-137 DOI:10.1109/ISVLSI49217.2020.00033.

[24] E. Lucero. *Unveiling our new Quantum AI campus* Online 18/05/2021 at https://blog.google/technology/ai/unveiling-our-new-quantum-ai-campus/

[25] R. Chao, B.W. Reichardt. *Fault-tolerant quantum computation with few qubits*. npj Quantum Inf 4, 42 (2018). DOI:10.1038/s41534-018-0085-z

[26] M. Suchara et al. *Comparing the Overhead of Topological and Concatenated Quantum Error Correction*. arXiv preprint 1312.2316, 2013

[27] L.M.K. Vandersypen, M. Steffen et al. *Experimental realization of Shor's quantum factoring algorithm using nuclear magnetic resonance*. arXiv preprint quant-ph/0112176, 2001

[28] M. Nakahara, T. Ohmi (2008). *Quantum Computing: From Linear Algebra to Physical Realizations*. CRC Press. DOI:10.1201/9781420012293

[29] Christensen, J.E., Hucul, D., Campbell, W.C. et al. *High-fidelity manipulation of a qubit enabled by a manufactured nucleus*. npj Quantum Inf 6, 35 (2020). DOI:10.1038/s41534-020-0265-5

[30] From IonQ's site "Technology" section, Online at https://ionq.com/technology

[31] From Alpine Quantum Technologies' site "Technology" section, Online at https://www.aqt.eu/technology/

[32] S. Allen, J. Kim, D. L. Moehring and C. R. Monroe. *Reconfigurable and Programmable Ion Trap Quantum Computer*. 2017 IEEE International Conference on Rebooting Computing (ICRC), 2017, pp. 1-3, DOI:10.1109/ICRC.2017.8123665.

[33] T. Monz et al. *Realization of a scalable Shor algorithm*. arXiv preprint 1507.08852, 2015

[34] D. Maslov. *Basic circuit compilation techniques for an ion-trap quantum machine*. arXiv preprint 1603.07678, 2016

[35] V.V. Aristov, A.V. Nikulov. *Chain of Superconducting Loops as a Possible Quantum Register*. arXiv preprint cond-mat/0412573, 2004

[36] *General U-gates*, 1.4.7 section, online at https://qiskit.org/textbook/ch-states/single-qubit-gates.html by The Jupiter Book Community

[37] B. Tan, J. Cong. *Optimality Study of Existing Quantum Computing Layout Synthesis Tools*. arXiv preprint 2002.09783, 2020

[38] T. Wenzel. *Quantum Mechanical Description of NMR Spectroscopy*. Chemistry LibreTexts, 2020.

[39] A.W Cross, L.S. Bishop, J.A. Smolin, J.M. Gambetta. *Open Quantum Assembly Language*. arXiv preprint 1707.03429, 2017

[40] A.W Cross, L.S. Bishop, J.A. Smolin, J.M. Gambetta et al. *OpenQASM 3: A broader and deeper quantum assembly language*. arXiv preprint 2104.14722, 2021

[41] M. Saeedi, I.L. Markov. *Synthesis and Optimization of Reversible Circuits - A Survey*. arXiv preprint 1110.2574, 2011

[42] M. Soeken, T. Haner, M. Roetteler. *Programming Quantum Computers Using*

Design Automation. arXiv preprint 1803.01022, 2018

[43] S.E. Venegas-Andraca et al. *A cross-disciplinary introduction to quantum annealing-based algorithms*. arXiv preprint 1803.03372, 2018

[44] Y.R. Sanders et al. *Compilation of Fault-Tolerant Quantum Heuristics for Combinatorial Optimization*. arXiv preprint 2007.07391, 2020

[45] C. Altafini, F. Ticozzi. *Modeling and Control of Quantum Systems: An Introduction*. arXiv preprint 1210.7127, 2012

[46] J. Liu, L. Bello, H. Zhou. *Relaxed Peephole Optimization: A Novel Compiler Optimization for Quantum Circuits*. arXiv preprint 2012.07711, 2020

[47] T. Itoko, R. Raymond, T. Imamichi, A. Matsuo. *Optimization of Quantum Circuit Mapping using Gate Transformation and Commutation*. arXiv preprint 1907.02686, 2019

[48] IBM's Quantum Experience, Quantum Composer, Online at https://quantum-computing.ibm.com/composer/

[49] V.V Shende, I.L. Markov. *On the CNOT-cost of TOFFOLI gates*. arXiv preprint 0803.2316, 2008

[50] J.C. Garcia-Escartin, P. Chamorro-Posada. *Equivalent Quantum Circuits*. arXiv preprint 1110.2998, 2011

[51] IBM's "Learn Quantum Computation using Qiskit", "2 - Multiple Qubits and Entanglement", Online at https://qiskit.org/textbook/ch-gates/more-circuit-identities.html by The Jupiter Book Community

[52] The SciPy Community's "API reference - SciPy API", Online at https://docs.scipy.org/doc/scipy/reference/

[53] The NumPy Community's "API reference - NumPy Manual", Online at https://numpy.org/doc/stable/

[54] M. Ben-Ari. *A Tutorial on Euler Angles and Quaternions*, version 2.0.1. Online at https://www.weizmann.ac.il/sci-tea/benari/sites/sci-tea.benari/files/uploads/softwareAndLearningMaterials/

[55] X. Zhang, H. Xiang, T. Xiang, L. Fu, J. Sang. *An efficient quantum circuits optimizing scheme compared with QISKit*. arXiv preprint 1807.01703, 2018

[56] D. S. Brezov, C. D. Mladenova, I. M. Mladenov. *New Perspective on the Gimbal Lock Problem*. AIP Conference Proceedings, 1570. 367-374, 2013

[57] IBM's "Qiskit Circuit Library - RXX Gate". Online at

https://qiskit.org/documentation/stubs/qiskit.circuit.library.RXXGate.html

[58] C. "Strilanc" Gidney. *Quirk Simulator*. Online at https://algassert.com/quirk

[59] A. Li, S. Stein, S. Krishnamoorthy, J. Ang. *QASMBench: A Low-level QASM Benchmark Suite for NISQ Evaluation and Simulation*. arXiv preprint 2005.13018, 2020

[60] *QASMBench circuits repository*, Online at https://github.com/uuudown/QASMBench

[61] Johannes Kepler University Linz's *Institute for integrated Circuits and System Design Group* Team online at https://iic.jku.at/eda/team/

[62] *JKU IIC circuits repository*, Online at https://github.com/iic-jku/ibm_qx_mapping/tree/master/examples