NExt ApplicationS of Quantum Computing

D3.4 myQLM special build

Document Properties

Contract Number	951821
Contractual Deadline	M27
Dissemination Level	Public
Nature	Report
Edited by :	Simon Martiel (Atos)
Authors	Simon Martiel (Atos)
Reviewers	Marko Rancic (Total), Andrés Gómez (CESGA)
Date	23/11/2022
Keywords	Quantum Circuit Compilation, Transpilation
Status	Final
Release	1.1

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 951821

History of Changes

Release	Date	Author, Organization	Description of Changes
1.0	23/11/22	Simon Martiel (ATOS)	Initial release for review
1.1	28/11/22	Simon Martiel (ATOS)	Implementation of the reviews

Table of Contents

1. Executive Summary

This report is the logical follow up of report D3.1: myQLM Specifications to support Hardware platforms. In D3.1, we detailed the software architecture behind myQLM and listed the various quantum circuit compilation and transpilation tools that can be used to efficiently target specific hardware platforms.
This new report provides a detailed presentation of a black-box compiler that combines quantum circuit optimization, compilation, and transpilation, all behind a user-friendly interface. After this presentation, we give results of execution and simulation of compiled quantum circuits and compare the resulting algorithmic performances against quantum circuits compiled with other frameworks.

2. Context

2.1. Project

One of the core objectives of the NEASQC project, is the development of applicative, quantum based, open-source, software libraries. Each of these libraries develops one range of applications of quantum computing to industrial use cases.
One of the roles of WP3 isthe development of the software bridge required to ensure that quantum accelerated applications developed in the QLM framework can be deployed on various hardware platforms. This report presents the final developments of a rather complete compilation tools that enables deployments of quantum applications on a wide variety of quantum hardware platforms.

2.2. Work package

As previously introduced WP3 activities serve the project as technical enablement activities that need to happen across the project's use cases. In addition to the points mentioned before and going in more detail, Tasks 3.3 and 3.1 deal with the provision of application centric benchmarks and the software bridge between applications and hardware platforms. On the other side, Task 3.3, responsible for the management and standardization of the applicative libraries of the project.

3. A single unified black-box compiler in QLM

This section details the structure of our new black-box compiler, designed in the QLM framework and called **NISQCompiler.** As its name suggests, this compiler is tailored to target NISQ platforms and can thus handle static compilation and transpilation of variational quantum circuits. Moreover, it is optimized for the compilation of energy sampling job where the energy computation is described as a sparse observable in the Pauli basis.
This compiler is included in the 1.7 release of the QLM framework, accessible by all the project partners. The documentation can be found here.

3.1. General architecture

The design of **NISQCompiler** heavily relies on the Plugin architecture of the QLM framework. **Plugins** are classical pre- and post- processings of quantum circuits and classical results that can be functionally composed in order to create powerful execution stacks attached to a QPU. Figure 1 depicts the general concept behind plugins and emphasizes their cross-languages capabilities.

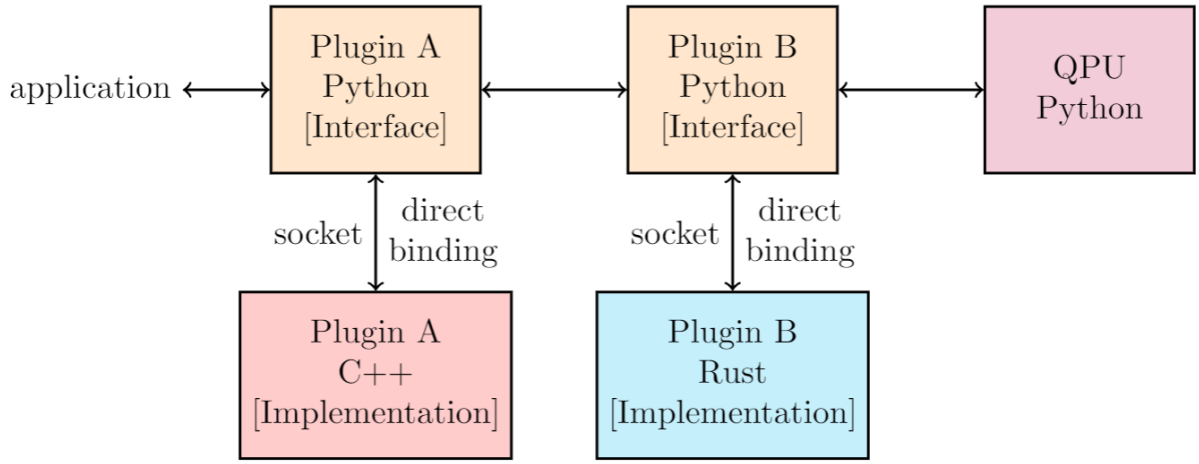

Figure 1: Concept of Plugins and their interaction with QPUs and applications

NISQCompiler is composed of different steps, each of them implemented via the use of one or several plugins. The stages are the following:

Pre-transpilation: The input quantum circuit is transpiled by expand gates in place in order to produce an output circuit containing solely: CNOT gates, one qubit Clifford gates in [H, $\sqrt{X}$, S] (or their inverse), and arbitrary RZ/RX/RY rotations.
This entails expanding common gates such as Toffoli and controlled phases using standard patterns. The purpose of this transpilation stage is to normalize the input gate set of the next compilation stages.

Compilation and optimization: This stage is the most critical one. The compiler is modifying the circuit in order to adapt it to the limited connectivity of the target hardware while attempting to reduce the overall 2-qubit gate count. At this stage, the compiler does not attempt to reduce the 1-qubit gate count since those can be reduced optimally in later stages. In practice, the compiler relies either on the ***lazy synthesis*** algorithm or on a SWAP insertion plugin. Since lazy synthesis outperforms the SWAP insertion approaches in most applications, we give more details on the various optimization stages it implements in the next section.

Two-qubit gates transpilation: The compiler then expands all the 2-qubit gates present in the circuit into equivalent subcircuits containing only the 2-qubit gates allowed in the target hardware. This is effectively implemented via the ***PatternManager*** plugin that can perform efficient circuit rewriting. After this stage, all 2-qubit gates are compatible with the target architecture as in they are both acting on neighboring qubits (thanks to the previous stage) and are part of the hardware's gate set.
At this stage of development, the supported entangling gates are CNOT, CZ, $ZZ\left(\frac{\pi}{2}\right)$, and $XX\left(\frac{\pi}{2}\right)$.

One-qubit gates compression and transpilation: In this last stage, the compiler compresses all the 1-qubit gate sequences into a sequence of 3 Pauli rotations (either Z-X-Z or X-Z-X) with a pattern that depends on the entangling gates generated at the previous stage. For instance, if the target entangling gates are CZ gates, the compiler will generate Z-X-Z patterns, thus allowing the first Z rotations to be commuted to the left of the previous entangling gate and merged with the last Z rotations of the previous 1-qubit gate sequence. This effectively reduces the 1-qubit gate count to a maximum of two 1-qubit gates (per wire) in between each entangling gate. Finally, the resulting sequences of one qubit gates are compressed again and redecomposed using the target gate set.

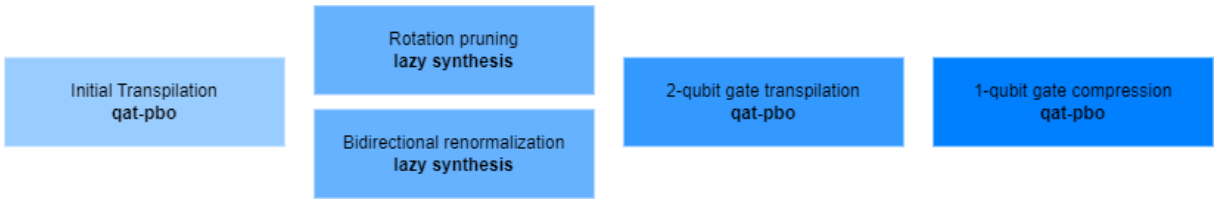

Figure 2: General structure of the compiler

3.2. Structure of the compilation stage

In this section, we detail the different compilation stages implemented in the ***lazy synthesis*** compiler.

3.2.1. Decomposition in Pauli rotation sequence

The compiler starts by decomposing the input circuit U in to a sequence (P_i, θ_i) of Pauli rotations and a final Clifford operator C_{glob}, such that:

$$U = C_{glob} \prod_i R_{P_i}(\theta_i)$$

This is done by pulling all Clifford gates to the end of the quantum circuit, aggregating them into a Clifford Tableau data structure representing C_{glob} and conjugating any non Clifford Pauli rotation with the current Clifford operator.

3.2.2. Rotation sequence pruning

The compiler then attempts to simply the rotation product using the following considerations:

- Any Pauli rotation applied on a Stabilizer state whose axis lies inside the stabilizer group of the state will have no effect on the state
- $|0\rangle$ is a Stabilizer state stabilized by $< Z_1, \dots, Z_n >$

These entails some possible simplications : initial Z rotations can be removed and X rotations can be simplified by using the fact that CNOT gate stabilizes $|0\rangle$.

3.2.3. Compilation and bidirectional Clifford renormalization

As introduced in (Martiel, 2020), the ***lazy synthesis*** algorithm can turn a circuit U into a new architecture-compliant circuit U_1 and a final Clifford operator C_1 such that:

$$U_0 = C_1 U_1$$

Using this black-box algorithm, one can iteratively compile a circuit and its inverse in order to produce the following sequence of equivalent circuits:

$$\begin{aligned} U_0 &= C_1 U_1 \\ &= C_1 U_2^\dagger \ C_2^\dagger \\ &= C_1 C_3 U_3 \ C_2^\dagger \end{aligned}$$

At each stage, one can track the cost of running the "middle" circuit U_i and the cost of running a Stabilizer state preparation $C|0\rangle$, with C being either the first aggregated Clifford operator, or the last one. At the end of the process (after a maximum number of iteration has been reached), the cheapest circuit is returned.
In practice, we use the number of entangling gate as a proxy for the implementation cost of the quantum circuits and Stabilizer state preprations.
Next subsection provides more details on how the final Clifford operator is dealt with.

3.2.4. Classical post-processing of the measurements

The lazy synthesis algorithm produces circuits that are equivalent to the input circuit up to some final Clifford operator. When sampling the energy of some observable specified in the Pauli basis, the compiler simply permutes the Pauli terms of the observable according to this Clifford operator, thus producing a quantum kernel strictly equivalent to the input kernel. If we are sampling observable $H = \sum_i \alpha_i P_i$ where P_i are Pauli operators, over a quantum state $C|\psi\rangle$ were C is Clifford, we can rewrite:

$$\begin{aligned} \langle\psi|C^\dagger H C|\psi\rangle &= \sum_i \alpha_i \langle\psi| \ C^\dagger P_i C \ |\psi\rangle \\ &= \sum_i \alpha_i \langle\psi|P_i'|\psi\rangle \end{aligned}$$

However, in the setting where one requires some samples/measurements in the computational basis, we can't rely on this method. In that setting, one would be tempted to append a circuit implementation of the final Clifford operator at the end of the compiled circuit. This is usually a quite costly quantum circuit (in $O(n^2)$ gates, depth of at least $O(n)$).
In practice, instead of synthesizing C in order to produce a quantum state $C|\psi\rangle$, we compute a Clifford basis change A such that A co-diagonalizes operators $\{C^\dagger Z_i C, i \in [1,n]\}$.
In other words: $D_i = AC^\dagger Z_i C A^\dagger$ are diagonal operators. When executing this (way simpler) basis change A on our quantum state $|\psi\rangle$, we get that sampling D_i is equivalent to sampling Z_i on the initial state:

$$\begin{aligned} \langle\psi|A^\dagger D_i A|\psi\rangle &= \langle\psi|A^\dagger A C^\dagger Z_i C A^\dagger A|\psi\rangle \\ &= \langle\psi|C^\dagger Z_i C|\psi\rangle \end{aligned}$$

Thus the statistics of the D_i operators on state $A|\psi\rangle$ are exactly equivalent to the statistics of operators Z_i over state $C|\psi\rangle$. Moreover, since the D_i are all diagonals, their values can be simultaneously sampled by measuring all the qubits and post-processing the results by counting qbit parities.

4. In-situ and simulated performances

This section is dedicated to benchmarking the NISQCompiler suite in realistic situations.
In order to assess the performances of our approach, we compiled several type of quantum circuits with state-of-the-art SWAP insertion techniques and with our compiler. The circuits were then emulated on a noisy emulator, and, in the case of QAOA circuits, on IQM's Adonis chip (a 5-qubits quantum chip based on superconducting qubits).
In order to make the comparison fair, we transpiled the circuits for IBM's initial gate set {CNOT, U3} which is close to most superconducting gate set's.
For these benchmarks, the bidirectional renormalization of the ***lazy synthesis*** backend was limited to a 1 second any time search, meaning that the compiler performed as many iterations as possible for 1 second and then picked the best resulting circuit. Section 5 provides details on the various parameters of the compiler and their effect on the runtime and performances.

4.1. QAOA MaxCut quantum circuits

The first set of benchmarks consists of 100 QAOA MaxCut depth-1 circuits generated from an Erdos-Renyi distribution with parameters $n = 5, p = \frac{1}{2}$. We picked 100 graphs according to this distribution and compiled the resulting depth-1 QAOA MaxCut Ansätze with a SWAP insertion based compiler and our compiler. Both compilation stacks rely on the same transpilation procedures, only the compilation/optimization stage differs from one circuit to the other. For a fair comparison, the SWAP based compiler was allowed to permute the initial position of the qubits.

This benchmark was also run on IQM's Adonis 5-qubit quantum chip (see Figure 4). Adonis gate set is composed of CZ gates (locally equivalent to CNOT gates), and phased $RX(\frac{\pi}{2})$ gates. RZ gates are performed virtually by dephasing the following phase RX gates (similarly to most superconducting setups).

We ran the 100 circuits on four different architectures, each time performing a parameter optimization in order to minimize the energy of the MaxCut cost Hamiltonian (using the gradient-free COBYLA optimization algorithm of scipy, with a tolerance of $1e-5$ and a random initial point). Each energy sampling was performed using 1024 shot. The first architecture consists in a perfect emulator, giving us a baseline for comparing with more realistic runs. The second and third architectures are emulators emulating a depolarizing noise with a 2-qubit error rate of 2% and 3%, and a measurement error of 5%, both of them with an emulated connectivity similar to IBM's Yorktown quantum chip. The last

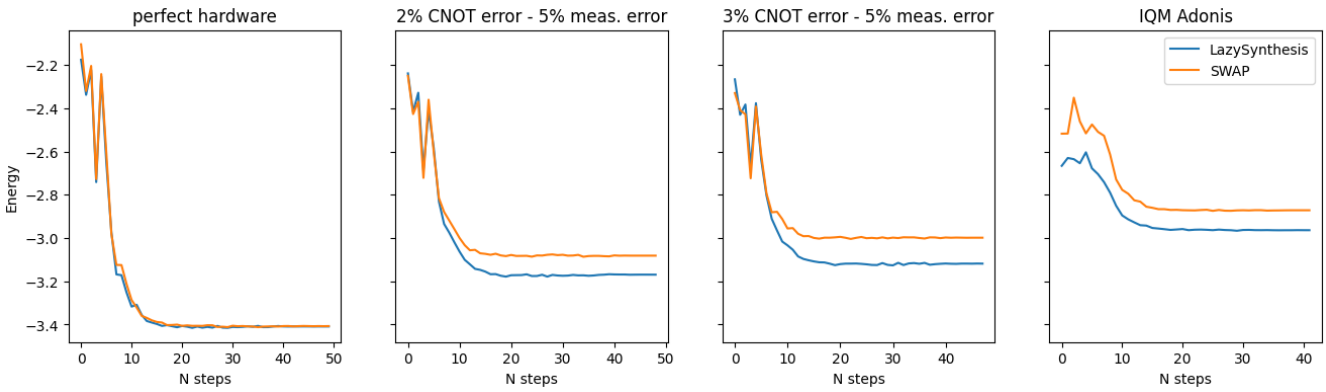

Figure 3: Evolution of the Ansatz energy as a function of the number of optimization steps

(avg. over 100 circuits)

architecture is IQM's Adonis chip; a superconducting quantum chip with a star shaped connectivity over 5-qubits. The results are depicted in Figure 3.

Without surprise, the depolarizing and measurement noise degrades significantly the performances, as in that the measured energy grows with the amount of noise. Moreover, we can observe that in the presence of noise, the lazy synthesis backend (in blue) performs far better than the standard SWAP insertion backend (in orange).
Interestingly, the run on IQM's quantum chip is significantly worse than the emulated hardware with 3% CNOT error rate. This might be due the more constained hardware topology (see Figure 4).
Overall, these results can be easily predicted by simply looking at the average entangling gate count produced by the two compilers on this benchmark set. Table 1 sums up the average counts obtained after compilation.

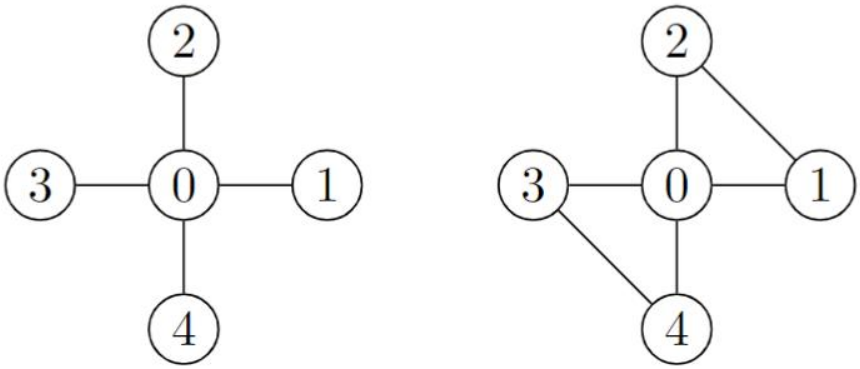

Figure 4: Left: IQM's Adonis topology. Right: IBM's Yorktown topology

Table 1: 2-qubit and 1-qubit gate count averaged over 100 QAOA MaxCut circuit of 5 qubits

Compiler	Avg. 2-qubit count	Avg. 1-qubit count
SWAP insertion	16.82	14.77
Lazy synthesis	7.46	11.32

4.2. UCCSD quantum circuits

The second benchmark input is a UCCSD Ansatz used for minimizing the energy of a LiH molecule. The circuits acts on 5 qubits and initially contains 64 CNOT gates. After compilation with the ***lazy synthesis*** backend, the compiled circuit contains 16 CNOTs and 18 U3 gates (reached at the 13th bidirectional iteration). When using the SWAP insertion backend, the resulting circuit contains 68 CNOTs and 43 U3 gates. The emulation results are presented in Figure 5.

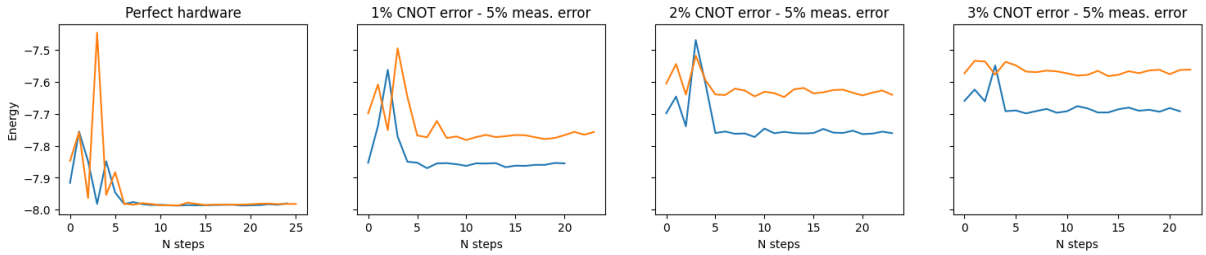

Figure 5: Optimization of an UCCSD Ansatz for LiH

As expected, in the presence of noise, the algorithm sees a performances decrease. UCCSD Ansätze being quite deep quantum circuits, the effect of noise becomes quickly noticeable. Once again, the fact that the lazy synthesis backend manages to only use 16 CNOT gates produces a much more resistant quantum circuit.

4.3. Quantum Arithmetic circuits

We also ran compilation benchmarks for a QFT based adder over 2 + 3 qubits. The quantum circuit consists in preparing a uniform distribution over a 2-qubit register and performing an addition into a second 3-qubit register. The expected measurement outcomes are classical states $|i\rangle|i\rangle$ for i ranging from 0 to 3, each of these outcomes having probability $1/4$.

The gate count comparison of the compiled circuits can be found in Table 2. Notice that the lazy synthesis backend is able to understand the structure of the circuit and delegates all the bit correlation to its classical post-processing of the measurements (see subsection 3.2.4). In practice, this results in a circuit with 0 CNOT gates.

Table 2: 2-qubit and 1-qubit gate count after compilation of a 2-3 QFT adder

Compiler	CNOT count	U3 count
SWAP insertion	28	29
Lazy synthesis	0	2

Consequently, the success probability of the circuits diverge rapidly when introducing noise in the system (see Figure 6).

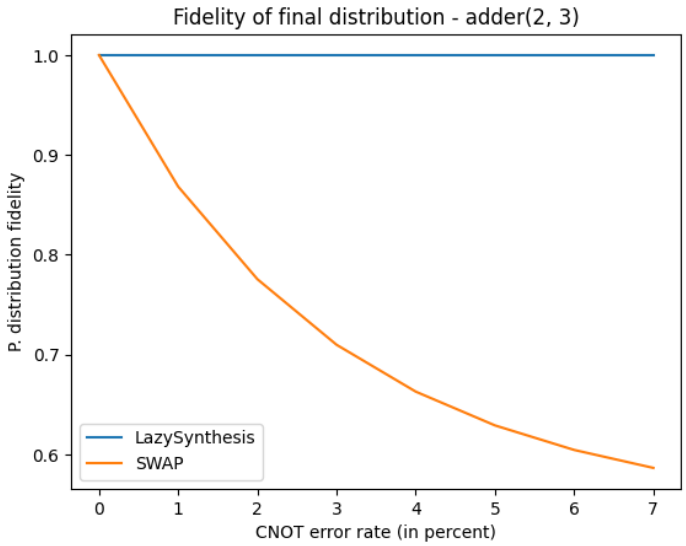

Figure 6: Final distribution fidelity w.r.t. to the theoretical result

5. NISQCompiler from a user point of view

The resulting NISQCompiler plugin comes with various parameters that can be fine-tuned by the user. Most of these parameters allow the user to tune the time/quality tradeoff of the compiler. Here, we only provide details on the **lazy synthesis** backend since it is the most performant.

Lazy Synthesis parameters:

- Search depth (key **"depth"**): everytime the algorithm has to make a choice, it recursively looks at the next "depth" choices to make and locally takes the best decision according to the future induced cost (i.e. it minimizes the future number of CNOT gates in the output circuit). This parameter induce an exponential cost increase. The overall complexity is $O(n^{2+3d}m)$ where n is the number of qubits, d the depth, and m the number of non-Clifford rotations. See (S. Martiel, 2022).
- Rotations meging and reordering (keys "**reorder**" and "**merge**"). When "**merge"** is set, the algorithm attempts to merge equivalent rotations to reduce the total rotation count. This might induce a large reduction since merging non-Clifford gates might result in a Clifford rotation, which is then pulled to the end of the circuit. When "**reorder**" is set, the algorithm takes the liberty of commuting rotation that can be commuted in order to reduce the CNOT count.
- Bidirectional normalization (keys "**bidirectional**", "**max_iter**", "**timeout**"). When "**bidirectional"** is set, the algorithm attempts several passes (see Subsection 3.2.3). "**max_iter**" then specifies the number of passes to perform. Alternatively, "**timeout**" can be used to specify a timeout in seconds in order to perform an any time search. The algorithm will iterate renormalization steps until the timeout is reached. Overall, the compilation cost grows linearly with the number of steps.
- Rotation pruning (key "**optimize_initial**"). When set, the algorithm will perform the rotation pruning procedure described in subsection 3.2.2.

Since all these options are parameters passed to the LazySynthesis plugin, they should be specified inside the **compiler_options** parameter of the NISQCompiler plugin:

```
NISQCompiler(
    compiler_options={
        'depth': 5,
        "bidirectional": True, "timeout": 600,
        "optimize_initial": True
        "reorder": True,
        "merge": True,
    }
)
```

Figure 7: Example of compiler options setup when building a NISQCompiler plugin

Gate set specification: The only other option of the compiler is the target gate set to transpile to. The compilers supports some presets that can be specified via a string:

- "IBM": CNOT + U3
- "IQM": CZ + $RX(\frac{\pi}{2})$ + $RZ(\theta)$
- "ions": $XX\left(\frac{\pi}{2}\right)$ + $RX\left(\frac{\pi}{2}\right)$ + $RZ(\theta)$

Other than that, the user can specify a list of gates in the following set:

- CNOT, CZ, ZZ (corresponding to $ZZ\left(\frac{\pi}{2}\right)$), XX (corresponding to $XX\left(\frac{\pi}{2}\right)$)
- RX (for $RX(\theta)$), RZ (for $RZ(\theta)$), RX+ (for $RX(\frac{\pi}{2})$), U3

6. Conclusion and perspectives

In conclusion, our NISQCompiler plugin remains simple to use for an end user while being more than competitive of NISQ quantum circuits. It can be simply adapted to target various hardware technologies, including all technologies developed in the quantum flagship projects (trapped ions and superconducting qubits).
The benchmark results show that using this compiler can directly improve the algorithmic performances of the compiled applications.

The general compilation approach described here is enough to outperform standard compilation approach, mainly thanks to the entangling gate count reduction.
It is however possible to further improve the in-situ performances of the compiled circuits by adapting the choices made at the compilation stage to a finer hardware model.

7. References

Martiel, S. a. (2020). Architecture aware compilation of quantum circuits via lazy synthesis. *Quantum,* doi: 10.22331/q-2022-06-07-729

8. List of figures

9. List of tables