PROJECT REPORT

Implementing and optimizing Shor's algorithm for native gate sets using Qiskit

Supervisor: David Fitzek

Examiners: Attila Geresdi
Simone Gasparinetti

Group members: Tobias Offermann
Nils Ciroth

Contents

1 Introduction and Motivation **2**

2 Shor's algorithm **2**
2.1 Mathematical background . . . 3
2.2 Quantum circuit of Shor's algorithm . . . 3
2.2.1 Quantum Fourier Transformation . . . 4
2.2.2 Unitary for modular exponentiation . . . 5
2.2.3 Quantum phase estimation . . . 5

3 Native gate sets **6**
3.1 Direct gate translation . . . 6
3.1.1 Necessary gates . . . 6
3.1.2 IBM hardware . . . 6
3.1.3 Chalmers hardware . . . 7
3.2 Fredkin gate discussion . . . 7
3.3 Predictions . . . 8

4 Decomposition of the whole circuit **9**
4.1 What is Qiskit and why do we use it? . . . 9
4.2 Transpile function and evaluation criteria . . . 10
4.2.1 IBM hardware . . . 11
4.2.2 Chalmers hardware . . . 11

5 Results of the circuit transpilation **12**
5.1 Results for the IBM hardware . . . 12
5.2 Results for the Chalmers hardware . . . 12

6 Conclusion **13**

7 Outlook **13**

Bibliography **14**

1 Introduction and Motivation

This project is about the implementation of Shor's algorithm with Qiskit and its optimization for different native gate sets. This topic lies at the interface of several different important branches in quantum computing. Shor's algorithm is probably the most important quantum algorithm to date since it is the only known one that grants an exponential speed up for a specific, practical problem compared to all classical algorithms. Moreover, this algorithm is still important in current research because once a large-scale application of Shor's algorithm is possible almost all of our modern cybersecurity protocols are not safe anymore. This is due to their reliance on the hardness of the factoring problem. The actual implementation is done with Qiskit, a quantum programming SDK that gives an impression of how the connection between the theoretical algorithms and the actual hardware could look like. Every quantum computer has a different hardware with different coupling maps and different gates. These come with individual difficulties of implementation and different run times on each device. Overall, we are going from the theoretical subject of the algorithm itself to programming the hardware with the question of which gates one should try to implement in the hardware. Or, to phrase it differently: We are going through different stages of building and programming a quantum computer.

2 Shor's algorithm

Shor's algorithm was found by Peter Shor in 1994 [1]. Its working principle is period finding using a Fourier transformation. In this light, a quantum speedup is possible by using the quantum Fourier transformation (QFT) within a phase estimation. We will be explaining the exact workings of both the algorithm and the quantum parts in the following chapters.

Peter Shor's current Twitter profile [2]

2.1 Mathematical background

The core concept of Shor's algorithm is the fact that a series of the form

$$\alpha_x = a^x \mod N \tag{1}$$

with arbitrary basis $a < N$ is periodic in x. This is easily traceable by the maximum of N different values of $\alpha \in \{0,1,...,N-1\}$ and the fact that each successor is only dependent on the value of the predecessor, since

$$\alpha_{x+1} = (a \cdot \alpha_x) \mod N \tag{2}$$

If the period r is known and even, the factors of N can be found easily:

$$a^0 = a^r \mod N = 1 \tag{3}$$
$$\implies (a^r - 1) \mod N = 0 \tag{4}$$
$$\implies c \cdot N = a^r - 1 = (a^{\frac{r}{2}} - 1)(a^{\frac{r}{2}} + 1) \tag{5}$$

The GCDs of $a^{\frac{r}{2}} - 1$ and $a^{\frac{r}{2}} + 1$ with N then reveals the factors, assuming one of them is not a full multiple of N.
Should one of those conditions be not fulfilled, a different basis a can be chosen and the algorithm can be re-run.

2.2 Quantum circuit of Shor's algorithm

The actual quantum part of the algorithm is just a single step of it, but still the most important one since it provides the exponential speed-up. We need to construct a circuit that returns the order r. For this, we need the unitary of modular exponentiation, which can be defined by

$$U|y\rangle = \begin{cases} |x \cdot y \mod N\rangle\,, & 0 \leqslant y \leqslant N-1 \\ |y\rangle\,, & N \leqslant y \leqslant 2^L - 1 \end{cases}, \tag{6}$$

where L is the number of qubits involved in the unitary. One can show [3] that the states

$$|u_s\rangle = \frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} \exp\Big[-2\pi i \frac{s}{r} k\Big] |x^k \mod N\rangle \tag{7}$$

are eigenstates of U, where

$$U|u_s\rangle = \exp\Big[2\pi i \frac{s}{r}\Big] |u_s\rangle \tag{8}$$

is the corresponding eigenvalue equation. Here, s with $0 \leqslant s \leqslant r-1$ can be an arbitrary integer. Since we can find r out of the fraction s/r with the classical continued fractions algorithm efficiently, it is sufficient to find a circuit that returns the phase s/r of the unitary U. Here is where phase estimation comes into play, which will be explained later. At this point, we will just show the complete circuit used for Shor's algorithm, which is depicted in figure 2. To further discuss the gates and the optimization of the circuit for different native gate sets, it is enough to look at the final circuit.

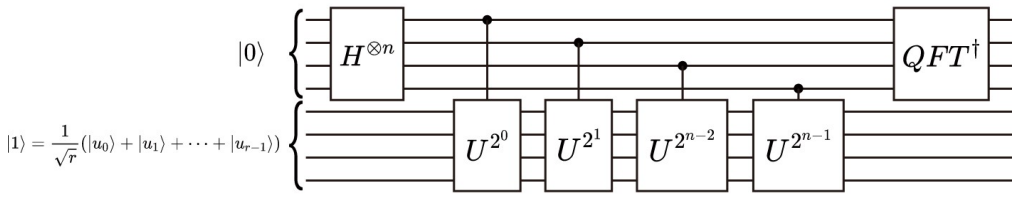

Figure 2: Full general circuit that performs Shor's algorithm. [4]

One can see that the circuit consists of two registers, single qubit gates, controlled unitary operators and a quantum Fourier transformation (QFT). The qubits in the first register are initialized using Hadamard gates, which is the first type of gates we need. The other parts of the circuit (namely the unitary and the quantum Fourier transform) and the gates we need to perform them are discussed in the following. For further reading we suggest reference [4].

2.2.1 Quantum Fourier Transformation

The quantum Fourier transformation can be seen as the quantum version of the classical discrete Fourier transformation, even though it has a slightly different purpose. It can be expressed as a unitary multi-qubit operation that applies the transformation

$$|j\rangle \rightarrow \frac{1}{\sqrt{N}} \sum_{m=0}^{N-1} e^{2\pi i j k/N} |k\rangle \tag{9}$$

which can be seen as a basis transformation of a basis state $|j\rangle$ to a new basis that is mathematically equivalent to the classical Fourier transform. We can use the binary representation of a multi-qubit state, defined by

$$k = k_1\, 2^{n-1} + ... + k_n\, 2^0 \tag{10}$$

to write the effect of the QFT in a more intuitive form, namely

$$|j\rangle \rightarrow \frac{1}{\sqrt{N}} \left(|0\rangle + e^{2\pi i \cdot 0.j_n} |1\rangle\right)\left(|0\rangle + e^{2\pi i \cdot 0.j_{n-1} j_n} |1\rangle\right) \cdot ... \cdot \left(|0\rangle + e^{2\pi i \cdot 0.j_1 ... j_n} |1\rangle\right). \tag{11}$$

It is relatively easy to construct a circuit that performs exactly this operation. We can define the single qubit gate

$$R_k = \begin{bmatrix} 1 & 0 \\ 0 & e^{2\pi i/2^k} \end{bmatrix} \tag{12}$$

and use its controlled version in the circuit shown in figure 3 to perform the transformation of relation (11). Therefore, the CR_k gate is the second type of gate we need to execute. Besides this, the QFT only contains Hadamard gates, of which we already know that they will be needed. The exact mathematical operations involved to show that equation (11) holds and is performed by the circuit in figure 3 can be found in reference [3].

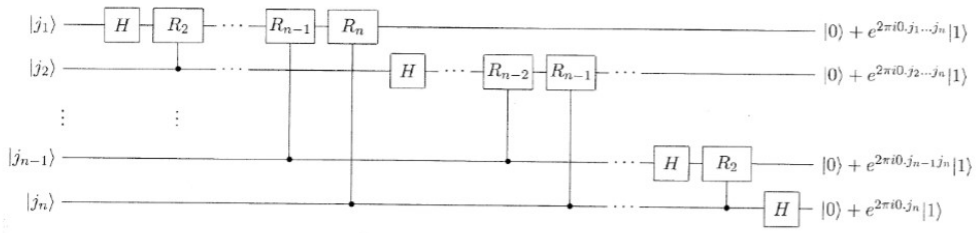

Figure 3: Circuit for the quantum Fourier transform. [4]

2.2.2 Unitary for modular exponentiation

In a last step, we need to decompose the modular exponentiation unitary defined in equation (6) into known gates before putting all pieces together again. It is quite hard to find a general decomposition summed up in a circuit, which is why we focus on our example of factoring 15 with basis $a = 2$ here. For this example, the corresponding unitary can be easily decomposed into multiple Fredkin gates, which can be seen in figure 4. For different numbers

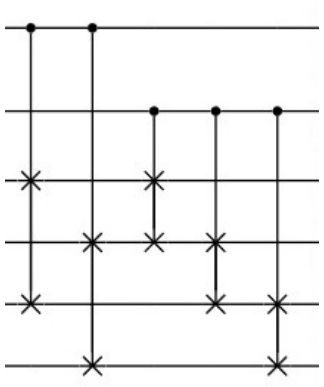

Figure 4: Decomposition of unitary for modular exponentiation of $N = 15$ into Fredkin gates.

than 15 and 2 only the amount of Fredkin gates changes or CX gates are added, which makes Shor's algorithm in general not much harder to decompose. However, here we will focus on factoring 15.

2.2.3 Quantum phase estimation

To sum things up, the quantum circuit for Shor's algorithm is s special form of the phase estimation for the modular exponentiation unitary defined in equation (6). A general circuit for phase estimation can be seen in figure 5. One can show that this circuit finds the phase φ of the eigenvalue $\exp(2\pi i\varphi)$ for an arbitrary unitary operator. This is possible because

$$U\,|u\rangle = e^{2\pi i\varphi}\,|u\rangle \tag{13}$$

holds for any unitary operator. As before, the exact mathematical background of why phase estimation works is not immediately relevant for our case, but can be found in reference [3] for further reading.

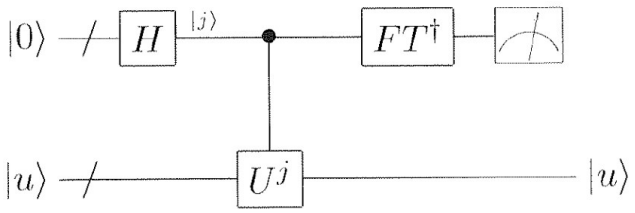

Figure 5: General circuit for phase estimation. One can see that Shor's algorithms circuit is a Phase estimation with a special unitary. [4]

3 Native gate sets

Native gate sets are the sets of quantum gates that can be applied by a given hardware. Any algorithm that is supposed to be ran on a quantum computer has to be translated into that computer's native gate set.

3.1 Direct gate translation

The most elementary way to translate any gate set into another is to translate each individual gate into suitable gates of the target architecture. This however can still be optimised, since certain steps may not be necessary when applying multiple gates after another. Nevertheless it is a good starting point and also allows for an elementary study on which gates are tough to translate. We will be analysing the translation of the necessary gates into the hardware of IBM and Chalmers.

3.1.1 Necessary gates

The gates we will need to be able to compute in order to perform our specific case of Shor's algorithm are the Hadamard gate, the CROT gate and the Fredkin gate.

3.1.2 IBM hardware

The gate set that is available on IBM hardware is consisting out of the gates CX, RZ, SX and X, where SX is the square root of the X gate. Fundamental translations into these gates are depicted in the following figures 6 for the Hadamard gate, 7 for the CROT gate and 8 for the Fredkin gate.

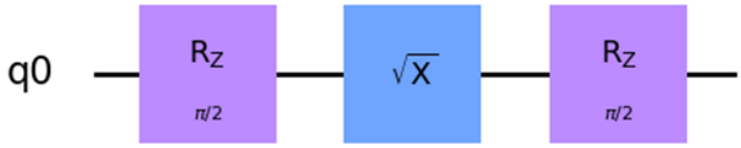

Figure 6: Hadamard gate in IBM native gate set.

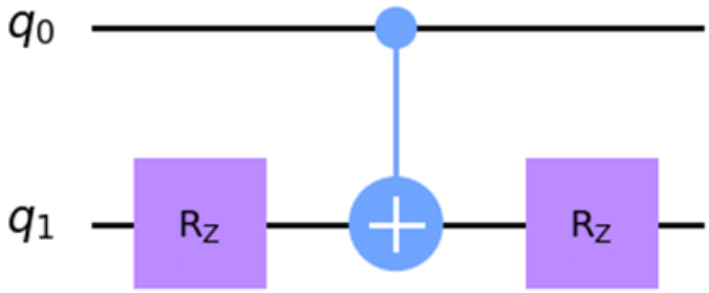

Figure 7: CROT gate in IBM native gate set. The angles of the first and second RZ gate are $-\frac{\theta}{2}$ and $\frac{\theta}{2}$ respectively to translate to a CROT with an angle of θ.

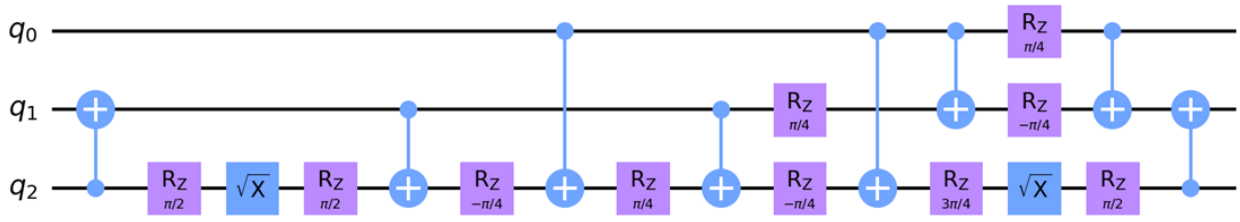

Figure 8: Fredkin gate in IBM native gate set.

3.1.3 Chalmers hardware

The Chalmers hardware has a different native gate set, it is given by iSWAP, CZ, RZ, RX($\pm\frac{\pi}{2}$) and RY($\pm\frac{\pi}{2}$). The translation of relevant gates except for the Fredkin gate has already been done [5], the Hadamard gate can be constructed with a depth of 2, the CROT gate with 3. The translation for the Fredkin gate can be achieved only with a very long depth of 23.

3.2 Fredkin gate discussion

From both decompositions it becomes clear that a decomposition into individual gates is usually only connected to a small factor of 2 or 3 for 1- and 2-qubit-gates, but for the 3-qubit Fredkin gate, the number of gates seems to be much higher. The reason for this is that there are only 1- and 2-qubit gates in both hardware gate sets, but they need to be linked in a way that a 3-qubit gate can be represented. With a 3-qubit gate, for example by adding a Toffoli

gate as current research efforts at Chalmers try to achieve [5], the translation of the Fredkin gate with CX and one Toffoli gate would have a length of 3 and looks as shown in figure 9.

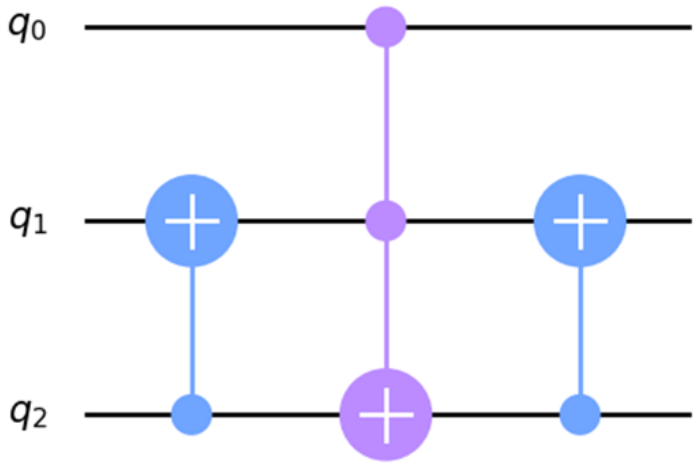

Figure 9: Fredkin gate decomposition using a Toffoli gate and CX gates.

Additionally, the given gates seem to be quite sub-optimal to translate the Fredkin gate, since the theoretical limit would be a 6 gate long translation [6]. A sensible 7 gate long translation would be given by introducing the CSX gate, a controlled SX gate, the translation were to look as depicted in figure 10.

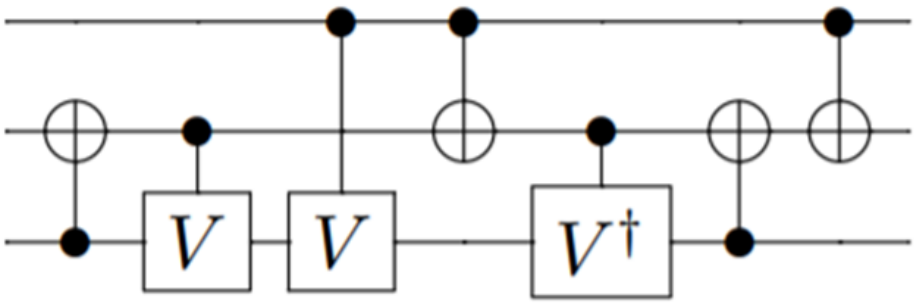

Figure 10: Representation of the Fredkin gate with depth of 7, using CSX gates (V) and CX gates. Taken from [6].

3.3 Predictions

Before tackling the whole circuit in the following chapters, we can predict that the Chalmers hardware will yield a longer circuit depth due to the bigger length of the Fredkin gate translation. Additionally, we can assume that the circuit depth will further increase as soon as the actual connectivities of the hardware is taken into account. The architecture with a denser connectivity map should be expected to have a less impactful increase due to this.

4 Decomposition of the whole circuit

As seen before, it is generally possible to find a translation from the theoretical universal gate sets to the native gate sets of a certain hardware by hand. However, this translation is far from being the optimal. Especially when multiple gates are used it is possible to drastically simplify certain gate combinations to reduce the number of gates that need to be performed (and therefore the run time of our algorithm). As an example one could look at the relation $HH = I$, which means that any two adjacent Hadamard gates can be simplified to the identity matrix. It is obvious that the application of two Hadamard gates on a qubit takes longer than not applying a gate at all. This is where the "optimization" part of our work becomes important again. We need to find the optimal circuit for our example of factoring 15 with Shor's algorithm, not just a direct translation. Or to rephrase it in a mathematical way: We need to decompose the unitary matrix for Shor's algorithm into the most efficient number of native gates. Matrix decomposition is a NP problem [7], which is why it is almost impossible to find any appropriate solution by hand. Because of that we make use of the computer (namely the Qiskit environment) for the implementation of our circuit of relatively small size, for bigger circuits, this method will also take exponentially more time.

4.1 What is Qiskit and why do we use it?

Qiskit is an open-source software development kit (SDK) that allows the circuit-based implementation of quantum gates and algorithms. It is developed by IBM Research with the original purpose to allow an easy development of quantum programs on the IBM cloud quantum computers. This is also one of the reasons why we used the IBM hardware to optimize our circuit. Qiskit uses Python as programming language, which means that it can be imported to a normal Python file and all the functions it provides can be used with the Python syntax. An example of how to create a basic circuit in Qiskit is shown in figure 11. After importing all the necessary libraries in the first line, a qubit register and a classical register is created. The function `QuantumCircuit()` takes both registers as an input and creates a variable that has the quantum circuit datatype. This variable represents the blank circuit. To add gates to the circuit, one just calls the function representing the specific gate on the quantum circuit variable. The input op the gate function depends on the type of gate. The single-qubit R_X-gate used in this circuit takes the rotation angle and the qubit it acts on as an input, while the following CX-gates needs the control and target qubit as input. The last parts of the circuits are measurements performed on both registers. This example also shows that Qiskit provides examples for visualizing circuits. There are more types of functions that do not serve the sole purpose of creating a circuit. Especially the `transpile()` function, which will be discussed in the following section, is essential for any optimization processes.

```
In [7]: from qiskit import QuantumRegister, ClassicalRegister, QuantumCircuit
        from qiskit.tools.visualization import circuit_drawer
        import numpy as np

        qr = QuantumRegister(2)
        cr = ClassicalRegister(2)
        qp = QuantumCircuit(qr,cr)

        qp.rx( np.pi/2,qr[0])
        qp.cx(qr[0],qr[1])

        qp.measure(qr,cr)

        circuit_drawer(qp)
```

Out[7]:

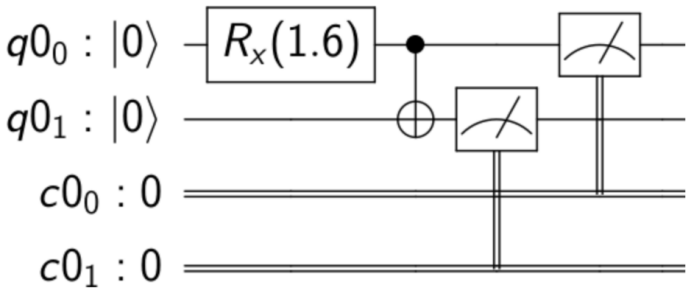

Figure 11: Example of a program in Qiskit that creates a quantum circuit, adds gates to the circuit and draws this circuit as output.

4.2 Transpile function and evaluation criteria

The `transpile()` function is a function included in Qiskit that allows to input a quantum circuit and get a circuit transpiled to a specific hardware as an output. It allows the implemetation of a certain native gate set as well as a coupling map, which both have a big impact on the circuit. Moreover, we can set an optimization level between 0 (no optimization) and 3 (full optimization). This optimization level determines how good the optimization described in the previous section is. Level 0 is just translating, while level 3 is our standard choice for optimizing. The output of this function is probabilistic, which means that all numbers that are presented in the following are just to demonstrate the order of magnitude or rough factor of change when enabling different optimization options. For our analysis of the optimized circuits, we will use two criteria to evaluate how good the optimization was. First, we will look at the circuit depth. It is defined by the length of the longest path from the input (or from a preparation) to the output (or a measurement gate). Intuitively, it makes sense that a deeper circuit corresponds to a longer run time, to that we are interested in decreasing the circuit depth. Second, we will also look at the number of two qubit operations. It is always easier to perform single qubit operations, because if you have more qubits they might have to be swapped before the gate can be applied. Therefore, the circuit depth is not enough to make statements about the runtime and fidelity of a program. A longer circuit could be easier to perform if the number of two qubit operations is significantly lower. However, it is not so easy to reduce both the circuit depth and the number of two qubit operations to a single, weighted number that quantifies the performance of a circuit.

4.2.1 IBM hardware

For the IBM device we used IBM Nairobi as reference device. The native gates for this device are explained above. This device is a 7-qubit quantum computer. The corresponding coupling map is shown in figure 12. Since our circuit only needs 6 qubits and this hardware provides 7, we will be using the qubit numbers 0 to 5 to encode the core algorithm. The additional qubit is, however, available to the optimiser in order to find more efficient routings.

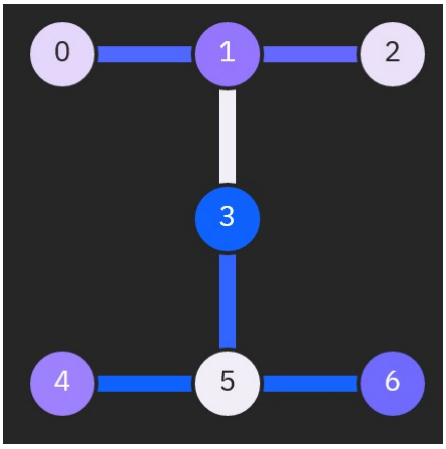

Figure 12: Coupling map of the IBM Nairobi quantum computer.

4.2.2 Chalmers hardware

For the upcoming quantum computer at Chalmers we used a 9-qubit fake backend. The native gate sets can be found above as well, the coupling map is depicted in figure 13. Analogous to the IBM hardware, we will be initially encoding the problem on qubits 0 to 5, with the other qubits also being available to the optimiser.

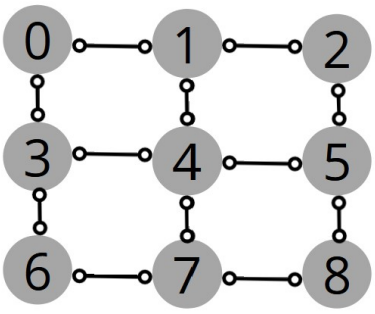

Figure 13: Fake coupling map of the upcoming Chalmers quantum computer.

5 Results of the circuit transpilation

In this chapter, we will show exemplary data for the circuit depth and number of two qubit gates for both hardware and different optimization levels are presented.

5.1 Results for the IBM hardware

As expected, we can see that the consideration of the coupling map increases both evaluation numbers significantly. Moreover, the fully optimized circuit decreases all evaluation numbers. Especially in the case with coupling map the reduction of two qubit operations is quite high, which is a result of the optimization of multiple swap gates.

Table 1: Results of the transpiled circuit for the IBM hardware.

	No optimization		Full optimization	
	Circuit depth	2qb gates	Circuit depth	2qb gates
Just native gates	362	167	308	165
Native gates and coupling map	486	386	576	359

5.2 Results for the Chalmers hardware

For the Chalmers hardware, we can make the same observations as for the IBM hardwere in terms of the influence of the coupling map and optimization level. We can see that in general the numbers in all categories are higher than for the IBM hardware, which means that the IBM hardware is better for factoring 15 with our 6-qubit version of Shor's algorithm The coupling maps do not seem to play as big of a role in creating a difference in circuit depth and 2-qubit gate count between the two architectures, which is most likely due to both maps being sufficient for the scale of the problem.

Table 2: Results of the transpiled circuit for the fake Chalmers hardware.

	No optimization		Full optimization	
	Circuit depth	2qb gates	Circuit depth	2qb gates
Just native gates	786	165	370	165
Native gates and coupling map	4853	699	694	303

6 Conclusion

In summary, we found an implementation of Shor's algorithm in Qiskit and used different optimization approaches to tailor this algorithm for the IBM and the Chalmers hardware. First, we searched for translations of the theoretical gates into the native gate sets of the corresponding hardware. We found out that native 3-qubit gates could drastically improve the performance in our algorithms. Moreover, we analyzed the transpiled circuit in terms of circuit depth and number of two-qubit operations and found out that the IBM hardware is more suitable for factoring 15. In addition to that, we observed that the density of the coupling map as well as the implementation of a coupling map itself has a strong impact on our evaluation parameters.

7 Outlook

We have found 3 main ways to improve the performance of a given hardware, either by improving the connectivity map, by adding gates to the native gate set that make the hardware better at translating common gates shorter and by introducing 3-qubit-gates like a Toffoli gate which significantly reduces the required depth to translate 3-qubit gates, up to a level where potentially much lower fidelities of the 3-qubit-gate would result in a higher overall fidelity since it saves so many steps. Especially for scaled architectures it will be important to take great care in optimising the connectivity map, since more and more swaps would otherwise be necessary. Significant research efforts are spent in all of these directions, so a good improvement not only in raw hardware, but also in compilation potential is expected to be observable in the mid-term future.

References

1. Shor, P. *Algorithms for quantum computation: discrete logarithms and factoring* in *Proceedings 35th Annual Symposium on Foundations of Computer Science* (1994), 124–134. doi:10.1109/SFCS.1994.365700 (cit. on p. 2).
2. Peter Shor's Twitter Profile. https://twitter.com/PeterShor1 (cit. on p. 2).
3. Ferrini, G., Kockum, A. F., García-Álvarez, L. & Vikstål, P. Advanced Quantum Algorithms (cit. on pp. 3, 4, 6).
4. Nielsen, M. A. & Chuang, I. L. *Quantum Computation and Quantum Information: 10th Anniversary Edition* 10th. ISBN: 1107002176 (Cambridge University Press, USA, 2011) (cit. on pp. 4–6).
5. *Chalmers internal* Private communication through David Fitzek about unpublished work within the research groups of Chalmers (cit. on pp. 7, 8).
6. Yu, N. & Ying, M. *Optimal simulation of three-qubit gates* 2013. arXiv: 1301.3727 [quant-ph] (cit. on p. 8).
7. Çivril, A. & Magdon-Ismail, M. On selecting a maximum volume sub-matrix of a matrix and related problems. *Theoretical Computer Science* **410,** 4801–4811. ISSN: 0304-3975. doi:https://doi.org/10.1016/j.tcs.2009.06.018. https://www.sciencedirect.com/science/article/pii/S0304397509004101 (2009) (cit. on p. 9).