Quantum Compilation for NISQ computers

Hasan Saygınel[1]

[1]*Department of Physics and Astronomy, University College London, London, WC1E 6BT, United Kingdom*
[1]*Email: hasan.sayginel.17@ucl.ac.uk*

Abstract

Noisy-intermediate scale quantum (NISQ) devices have not converged to one type of technology for qubit fabrication. Each quantum processor has its own set of hardware constraints besides limited qubit count due to the different qubit technologies. Quantum algorithms, on the other hand, are written using abstract quantum circuits without a specific device implementation in mind. Quantum compilers bridge this gap by mapping the logical qubits of an algorithm to the physical qubits of the device in a way that the output circuit complies with the hardware limitations such as qubit connectivity and the native gate set. Furthermore, a compiler has various optimisation steps to shorten the circuit depth as much as possible in order to reduce errors accumulated due to error-prone quantum operations. This paper discusses the compilation requirements of qubit architectures, introduces the various steps in quantum compilers using Cambridge Quantum's t|ket⟩ as an example and evaluates current research on improving Qubit Mapping based on proposed application-specific compilers and noise-aware techniques.

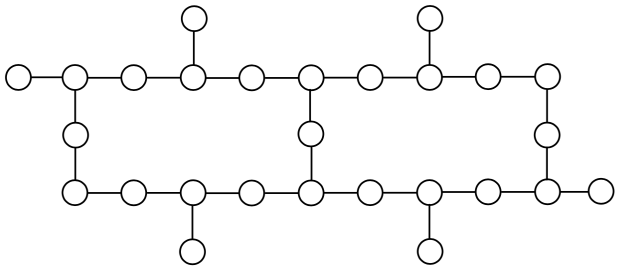
1 Introduction

Compilation is standard practice in classical programming and is used to map an algorithm written in a high-level user-friendly programming language into the low-level machine language instructions that are executed step by step on the device hardware [1]. Classical computers are built from relay circuits or transistors that work as switches and therefore take as input Boolean instructions (0 & 1). These instructions are abstracted in several layers so that programmers can use intuitive programming languages and run algorithms without needing to know the inner workings of the hardware they are using [2]. Abstraction is also desired in quantum algorithms to allow a programmer to focus on applications without requiring a deep knowledge regarding the quantum technologies underlying quantum processors and the physics behind qubit operations.

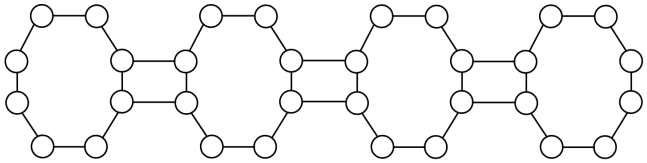
Quantum algorithms are conventionally written using idealised quantum circuits that are intuitive to the physicist, however, include no information about how the algorithm can be implemented on a device. The realisation and advancements of Noisy Intermediate Scale Quantum (NISQ) computers [3] has motivated the development of quantum compilers in order to map abstract quantum circuits into hardware executable circuits so that algorithms can be tested and benchmarked for the proposed near-term applications such as quantum chemistry simulations [4] and machine learning [5]. The hardware limitations of current quantum devices, where even the state-of-the-art processor announced by IBM contains only 127 qubits with limited connectivity [6], means that quantum compiler toolflows must ensure logical qubits and operations of a suitable algorithm are mapped to the physical qubits of a device in a feasible and optimal way.

The primary aim of a quantum compiler is to make sure that the algorithm respects the hardware constraints including the qubit count, native gate set and the connectivity limitations of the device so that it can be mapped into corresponding instructions. Furthermore, quantum compilation involves circuit optimisation steps that aim to shorten the circuit depth where possible, and replace error-prone gates with alternative gates with higher fidelity. This is particularly important in the near future where the limited qubit number in available hardware does not allow for error correction meaning that noise has to be considered and every available qubit matters.

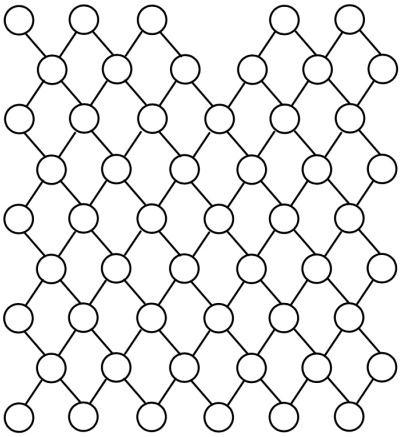
A number of quantum programming languages including t|ket⟩ by Cambridge Quantum [7], Qiskit by IBM [8], Cirq by Google [9] and Quil by Rigetti [10] equipped with compiler software are already available. The purpose of this paper is to review different candidates for quantum processor architectures that are used as backends, discuss the techniques in NISQ compilers, present key challenges in compilation and explore possible avenues for further research.



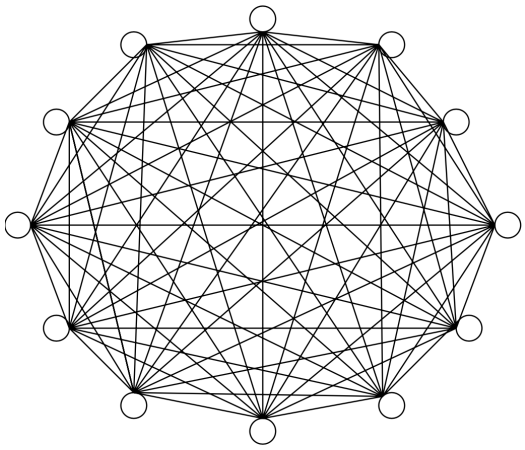
(a) IBMQ Montreal [11]



(b) Rigetti Aspen-9 [12]



(c) Google Sycamore Computer [13][14]



(d) IONQ [15]

Figure 1: Device topologies of different quantum hardware where the nodes represent qubits and edges represent connectivity.

2 Qubit Architectures

Quantum computing has a number of candidates and has not yet converged to one type of universal technology for qubit production [2]. Leading technologies include superconducting qubits [16] and trapped ion qubits [17] as well as other interesting candidates such as spin qubits [18]. For each different technology the fundamental gate operations that can be applied on the hardware, known as the gate set of the device, varies. Typically, each different hardware contains at least one 1-qubit (1Q) and one 2-qubit (2Q) gate as part of a universal set meaning that it is possible to express any algorithm using only the gate set of the device.

The underlying technology of the quantum devices also determines the physical implementation of 2-qubit interactions leading to different device topologies. In superconducting devices, 2Q operations are achieved via adjustable couplers which need to be placed between the two qubits involved in the operation. IBM, Rigetti and Google AI systems, all consisting of superconducting qubits, therefore have sparse near-neighbour connectivity where 2-qubit operations can only be performed between two physically adjacent qubits. This necessitates the addition of SWAP operations during compilation such that the desired control and target qubits in an algorithm are brought next to each other where necessary [14]. Trapped ion qubits do not need SWAP operations as they have all-to-all connectivity. In such systems, the two qubit gate is achieved via irradiating any two ions on the grid with laser pulses [19]. Figure 1 illustrates the connectivity graphs of four quantum processors from different providers for which the device topology and fidelity characteristics of the qubits are publicly available. The nodes in the graphs represent the physical qubits of the device and edges represent connectivity.

In the NISQ era, there are a number of error sources in the hardware limiting the success rate of the computation. The errors include operation, communication and measurement errors (also known as readout errors) [20]. The implementation of an algorithm is further constrained by the relaxation, $_1$, and decoherence times, $_2$. $_1$ characterises the timescale in which an excited state $|1\rangle$ decays into the ground state $|0\rangle$ and $_2$ indicates the timescale in which the superposition state collapses to a classical value due to deco-

Machine	Qubit Count	Gates	Gate Errors	Readout Error	$_1$ (s)	$_2$ (s)
IBMQ Montreal [11]	27	1Q: I, X, RZ, SX	0:02	1:7	90:1	81:1
		2Q: CX	1:20			
Rigetti Aspen-9 [12]	32	1Q: RX, RZ	0:7	3:8	29:5	17:5
		2Q: CPHASE, CZ, XY	9:3			
Google Sycamore [13]	53	1Q: Phased XZ, Z	0:1	5 − 7	15	
		2Q: Sycamore, $\sqrt{iSWAP}$, CZ ^{y}	0:9 − 1:4			
IONQ [15]	11	1Q: GP I( ), GP I2( ), RZ*	0:5	0:7		
		2Q: MS	2:5			

Table 1: Survey of device characteristics of four different quantum processors based on public information. The errors in 2-qubit gates are an order of magnitude worse than 1-qubit errors. ^{y} in development.

herence. This places a limit on the number of gate operations that can be executed accurately on a device before decoherence. Table 1 summarises the device characteristics of the same four quantum machines, including the qubit count, gate set, error rates and coherence times. The typical 2Q gate error rates are an order of magnitude worse than the 1Q gate errors. 2Q gates resulting in entanglement make an algorithm more expressive and are essential to non-classical behaviour. However, due to worse fidelities, the number of 2Q gates should be optimised to avoid redundancies and replaced with alternative sub-circuits with a lower number of 2Q gates where this increases the overall fidelity of the circuit.

3 Quantum Compilation

Quantum compilation has a two-fold purpose; to map an abstract algorithm into an executable one and to optimise the new circuit while still satisfying all the constraints of a particular hardware. Figure 2 illustrates the typical pipeline of a quantum compiler. It takes as input a quantum computing program, device-specific information and applies a number of passes, returning code in the appropriate low-level language, such as IBM OpenQASM that satisfies the hardware constraints of IBM machines. This framework describes a general compiler that can take as input code from all (or a number of) quantum software languages and compile it onto any device. Cambridge Quantum's t|ket⟩[1] is one such compiler and can be used to run algorithms written in different quantum software languages on a number of different platforms. Therefore, t|ket⟩ allows for cross-platform benchmarks [21]. In the following, various compilation passes of t|ket⟩ will be explained as an example, however other compilers such as Qiskit involve similar passes albeit with different techniques in implementing them.

[1]documentation available at [7].

3.1 Circuit optimisation

Each qubit operation in a quantum circuit has a cost when executed on hardware as the quantum gates in NISQ devices are error-prone. Therefore, t|ket⟩ has a number of circuit optimisation techniques in order to shorten the circuit depth as much as possible. Circuit optimisation is carried out both before the circuit is mapped to the device and after because the other compilation passes introduce additional gates which may lead to redundant sequences that can be optimised. In the latter case, circuits are only modified in a way that the output still respects the various hardware constraints. Circuit optimisations by t|ket⟩ can be divided into two categories; namely peephole and macroscopic optimisations.

3.1.1 Peephole optimisations

Peephole optimisations look for small patterns of unitaries and substitutes sub-circuits where an equivalent sub-circuit with a lower gate count or depth exists. This procedure also eliminates any redundant gates with no physical effect, such as identities, gate-inverse pairs, adjacent rotation gates in the same basis and diagonal gates before measurement [21]. Gate commutation is also considered, where via commuting gates around further redundant patterns are identified and eliminated.

Furthermore, the Cartan decomposition based on the Lie algebra of the unitary group provides a way of decomposing arbitrary unitaries $U \in SU(N)$ into local unitaries [22] while also stating an upper bound for how many 2Q operations are needed to construct an arbitrary unitary of dimension N [23]. For single and two-qubit unitaries Cartan decomposition gives the commonly used Euler-angle decomposition and KAK decomposition.

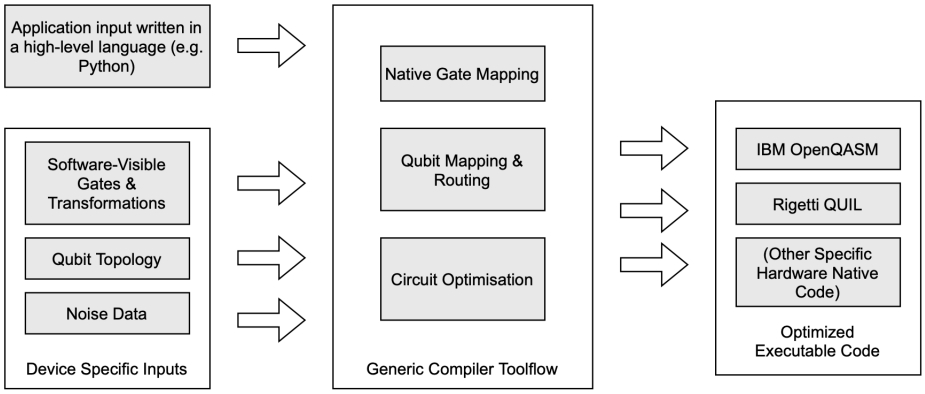
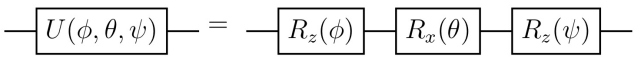
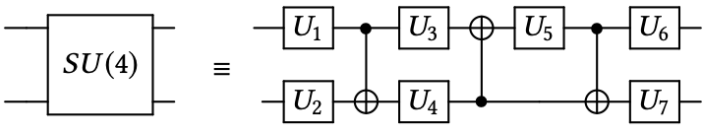


Figure 2: A general compiler receives application inputs containing the algorithm written a high-level language as well as device specific inputs and applies compilation passes in order to output executable code that complies with low-level machine specific languages.

Theorem 1 *Any single qubit gate* $U \in SU(2)$ *can be decomposed into a sequence of at most three rotations using any choice of Rx, Ry, and Rz gates. The angles of rotation are given by the Euler-angle decomposition of the combined rotation of the Bloch sphere [24].*



Theorem 2 *Any two-qubit unitary* $U \in SU(4)$ *can be synthesised using at most three CX gates and* 15 *parameterised single-qubit gates (from any choice of Rx, Ry, and Rz), given by the KAK decomposition [25] [26].*



Euler and KAK decompositions are used by t|ket⟩ to reduce long sequences of 1Q gates on a single qubit into three 1Q rotation gates and to decrease the 2Q gate count on identified 2-qubit sub-circuits via KAK representation which at most contains three 2Q gates.

3.1.2 Macroscopic optimisations

While the peephole optimisations focus on local redundancies by considering individual gate commutations, macroscopic optimisations look for larger structures or it converts the circuit completely into an alternative algebraic representation based on ZX-calculus [27][28] to find simplifications in that representation and resynthesise into the basic gate model. One such approach is Pauli gadgets [29]. Pauli gadgets can be used to represent the whole circuit as a sequence of exponentials of Pauli operators. This expresses non-Clifford gates as rotations in a basis determined by the Clifford gates and allows for commutations of non-Clifford gates so that they can be merged.

3.2 Mapping into native gate set

An abstract quantum circuit comprised of any gates must be re-expressed using only the hardware qubit gates when it will be run on a backend. t|ket⟩ implements this step naively using known gate decompositions, deferring optimisation to the optimisation pass. Therefore, this procedure typically increases circuit depth [7].

It is up to the choice of a software programmer to decide on which gates will be software-visible in the high-level language that a user has access to. Exposing as much of the native gate set to the user as possible to encourage them to write algorithms using the native gates could reduce overhead from compilation [2]. On the other hand, there is a trade-off between abstraction and overhead. While the CX gate is a native gate only in the IBMQ machine given in Table 1, it might be helpful to the user to have it as a software-visible gate on all platforms since CX is a ubiquitous gate in quantum computation. CX gate decompositions into the gate sets of the Rigetti and Google devices are given below:

- Rigetti: [Rz(0.5) q[1];, Rx(0.5) q[1];, Rz(0.5) q[1];, CZ q[0], q[1];, Rz(0.5) q[1];, Rx(0.5) q[1];, Rz(0.5) q[1];]
- Google: [Rz(1) q[1];, PhasedX(0.5, 0.5) q[1];, CZ q[0], q[1];, Rz(1) q[1];, PhasedX(0.5, 0.5) q[1];]

where q[0] represents qubit 0, q[1] represents qubit 1 and the arguments of the gates are multiples of . Therefore, a single CX gate introduces many gates after decomposition. An extensive use of the CX gate on a device that does not have it as a native gate could lead to less reliable algorithms.

3.3 Qubit mapping

Qubit mapping is the task of finding an optimal mapping of the logical qubits into the physical qubits in a way that introduces a minimal number of SWAP (or CX) operations. SWAP operations have a high cost in terms of noise because the implementation of SWAP operations typically requires three 2Q gates which are an order of magnitude more error prone than 1Q gates.

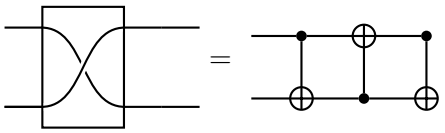


Figure 3: Decomposition of SWAP gate into CX gates

An alternative to SWAP is bridged-CX operations which introduces four CXs. A compiler may opt for this option if some of the introduced CX gates cancel with other CX gates in the circuit and the final circuit has fewer 2Q gates in comparison to one that would be produced by introducing a SWAP gate.

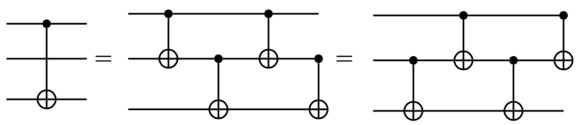


Figure 4: Decomposition of distance-2 CX into four distance-1 CXs.

The initial *qubit placement* problem is equivalent to sub-graph isomorphism while the *qubit routing* (moving qubits around using SWAP operations) is equivalent to token-swapping which has been shown to be at least NP-hard [30][31]. There exists exact solutions to the mapping problem however they scale with complexity exponential in the number of qubits. The exact methods include an exact dynamic programming method by [32] and a constrained linear optimisation method by [33]. In the latter case, a satisfiability modulo (SMT) solver is used where the variables are the program qubit locations, gate start times, routing paths and the constraints specify qubit mappings should be distinct, gates should start in program dependency order and routing paths should be non-overlapping.

t|ket⟩, on the other hand, opts for a heuristic method that scales well in terms of circuit depth and total qubit gate count. The heuristic method used by t|ket⟩ is discussed in depth below.

3.3.1 Qubit Routing by t|ket⟩

The objective function of the t|ket⟩ algorithm is to output a circuit with optimal CX count due to higher error rates of 2Q gates. Therefore, during routing only the 2Q gates in the circuit are considered but the algorithm still respects the semantic order of the 1Q and 2Q gates when they do not commute. The connectivity map of a device is represented as an undirected graph $G_D = (V_D; E_D)$ where V_D are the vertices representing qubits and E_D are the edges representing the possible 2Q interactions. An interaction graph is constructed for the logical circuit given as $G_I = (V_I; E_I)$. The placement problem is solved when for every two-qubit gate, $(q; q')$, the map $p : V_I \to V_D$ respects $(q; q') \in E_I \Rightarrow (p(q); p(q')) \in E_D$. However, due to limited connectivity of most quantum processors, naive placement usually does not satisfy $(p(q); p(q')) \in E_D$ necessitating the qubit routing step.

The routing algorithm iterates through the 2Q gates and finds the first 2Q gate, $(q; q')$, such that $(p(q); p(q'))$ does not respect G_D and no q is involved in more than one interaction. This splits the circuit into timeslices within which the operations can be simultaneously performed. The first slice is denoted S_0 and permuted by adding SWAP operations resulting in a temporary placement p'. The permutation of p and p' is logged and the algorithm picks the optimal edge $e \in E_D$ to implement a SWAP operation on. A set of different placement options $\{p''_0; : : : ; p''_n\}$ is constructed based on permutations of p' with SWAP operations on different edges in E_D. Each placement candidate is scored based on the distance between interactions in S_0 and p''. If there is a tie in the scores, the placements are scored for a new slice S_1 where S_1 is the next set of 2Q gate $(q; q')$ where $(p(q); p(q'))$ does not respect G_D and q interacts once. The same process is repeated scoring for higher slices if necessary until there is a winning placement for some S_n.

3.4 Compilation example

See appendix for a compilation example where the Bernstein-Vazirani algorithm with 5 qubits in its abstract representation is compiled onto the IBMQ-Quito device which is publicly available at [11] for experiments.

4 Conclusions & Further work

Compilers are essential to map abstract algorithms into executable code. Due to the presence of noise and coherence limitations in the NISQ era, how well a compiler optimises a circuit may well determine the success rate of an algorithm. General purpose compilers, such as t|ket⟩, help abstract the qubit technologies in order to allow a programmer to focus on applications.

While the retargetability of t|ket⟩ is a major advantage and allows for benchmarking across different devices, recent studies have shown that application-specific compilers such as 2QAN [34] may outperform general compilers. The obvious disadvantage of such compilers is that they are application-limited and therefore do not allow for much abstraction. However, if efficient compilers are constructed for a particular class of circuits/algorithms, such as 2-local qubit Hamiltonians in the 2QAN case, then application-specific compilers can prove to be more useful in the NISQ era provided that they outperform general-purpose compilers.

This paper did not address noise-aware techniques in compilation, however there have been studies in integrating noise-data available from hardware providers into compilation methods [33] [35]. A noise-aware compiler, TriQ, constructs a reliability matrix where the (i^{th}, j^{th}) entry of the matrix summarises the end-to-end reliability of an operation between any pairs of qubits based on both the number of SWAP gates required and also the fidelity of operations on the edges [2]. Optimising solely for SWAP gates could sometimes lead to less reliable use of hardware if the logical qubits are mapped to physical qubits that have higher error rates. t|ket⟩ hosts noise-aware initial qubit placement but does not utilise noise data during routing. There could be further studies investigating how well noise-aware techniques scale in comparison to ones that optimise solely on communication distance.

References

[1] Frederic T. Chong, Diana Franklin, and Margaret Martonosi. Programming languages and compiler design for realistic quantum hardware. *Nature*, 549(7671):180–187, 2017.

[2] Prakash Murali, Norbert Matthias Linke, Margaret Martonosi, Ali Javadi Abhari, Nhung Hong Nguyen, and Cinthia Huerta Alderete. Full-Stack, Real-System Quantum Computer Studies: Architectural Comparisons and Design Insights. *arXiv e-prints*, page arXiv:1905.11349, May 2019.

[3] John Preskill. Quantum Computing in the NISQ era and beyond. *arXiv e-prints*, page arXiv:1801.00862, January 2018.

[4] Abhinav Kandala, Antonio Mezzacapo, Kristan Temme, Maika Takita, Markus Brink, Jerry M. Chow, and Jay M. Gambetta. Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets. *Nature*, 549(7671):242–246, 2017.

[5] Jacob Biamonte, Peter Wittek, Nicola Pancotti, Patrick Rebentrost, Nathan Wiebe, and Seth Lloyd. Quantum machine learning. *Nature*, 549(7671):195–202, 2017.

[6] IBM Quantum. 127-qubit Eagle processor. `https://newsroom.ibm.com/2021-11-16-IBM-Unveils-Breakthrough-127-Qubit-Quantum-Processor`, 2021. Online; accessed 29 November 2021.

[7] Cambridge Quantum. pytket 0.15.0 documentation. `https://cqcl.github.io/tket/pytket/api/index.html#`, 2021. Online; accessed 20 October 2021.

[8] Qiskit. Qiskit 0.32.1. `https://qiskit.org`, 2021. Online; accessed 29 November 2021.

[9] Google Quantum AI. Cirq. `https://quantumai.google/cirq`, 2021. Online; accessed 29 November 2021.

[10] Rigetti Computing. pyquil 3.0.0 documentation. `https://pyquil-docs.rigetti.com/en/latest/quilt_raw_capture.html`, 2021.

[11] IBM Quantum. ibmq_montreal machine. `https://quantum-computing.ibm.com/`, 2021. Online; accessed 03 November 2021.

[12] Rigetti. Rigetti Systems, Aspen-9 Quantum Processor. `https://qcs.rigetti.com/qpus/`, 2021. Online; accessed 04 November 2021.

[13] Quantum AI, Google. Quantum Computer Datasheet. `https://quantumai.google/hardware/datasheet/weber.pdf`, 2021.

[14] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C. Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando G. S. L. Brandao, David A. Buell, Brian Burkett, Yu Chen, Zijun Chen, Ben Chiaro, Roberto Collins, William

Courtney, Andrew Dunsworth, Edward Farhi, Brooks Foxen, Austin Fowler, Craig Gidney, Marissa Giustina, Rob Graff, Keith Guerin, Steve Habegger, Matthew P. Harrigan, Michael J. Hartmann, Alan Ho, Markus Hoffmann, Trent Huang, Travis S. Humble, Sergei V. Isakov, Evan Jeffrey, Zhang Jiang, Dvir Kafri, Kostyantyn Kechedzhi, Julian Kelly, Paul V. Klimov, Sergey Knysh, Alexander Korotkov, Fedor Kostritsa, David Landhuis, Mike Lindmark, Erik Lucero, Dmitry Lyakh, Salvatore Mandrà, Jarrod R. McClean, Matthew McEwen, Anthony Megrant, Xiao Mi, Kristel Michielsen, Masoud Mohseni, Josh Mutus, Ofer Naaman, Matthew Neeley, Charles Neill, Murphy Yuezhen Niu, Eric Ostby, Andre Petukhov, John C. Platt, Chris Quintana, Eleanor G. Rieffel, Pedram Roushan, Nicholas C. Rubin, Daniel Sank, Kevin J. Satzinger, Vadim Smelyanskiy, Kevin J. Sung, Matthew D. Trevithick, Amit Vainsencher, Benjamin Villalonga, Theodore White, Z. Jamie Yao, Ping Yeh, Adam Zalcman, Hartmut Neven, and John M. Martinis. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, 2019.

[15] IonQ Inc. Best Practice for Using IonQ Hardware. `https://ionq.com/best-practices`, 2021.

[16] J. Majer, J. M. Chow, J. M. Gambetta, Jens Koch, B. R. Johnson, J. A. Schreier, L. Frunzio, D. I. Schuster, A. A. Houck, A. Wallraff, A. Blais, M. H. Devoret, S. M. Girvin, and R. J. Schoelkopf. Coupling superconducting qubits via a cavity bus. *Nature*, 449(7161):443–447, 2007.

[17] J. I. Cirac and P. Zoller. Quantum computations with cold trapped ions. *Phys. Rev. Lett.*, 74:4091–4094, May 1995.

[18] Hans-Andreas Engel, L. P. Kouwenhoven, Daniel Loss, and C. M. Marcus. Controlling spin qubits in quantum dots. *Quantum Information Processing*, 3(1):115–132, 2004.

[19] K. Wright, K. M. Beck, S. Debnath, J. M. Amini, Y. Nam, N. Grzesiak, J. S. Chen, N. C. Pisenti, M. Chmielewski, C. Collins, K. M. Hudek, J. Mizrahi, J. D. Wong-Campos, S. Allen, J. Apisdorf, P. Solomon, M. Williams, A. M. Ducore, A. Blinov, S. M. Kreikemeier, V. Chaplin, M. Keesan, C. Monroe, and J. Kim. Benchmarking an 11-qubit quantum computer. *Nature Communications*, 10(1):5464, 2019.

[20] Swamit S. Tannu and Moinuddin K. Qureshi. A Case for Variability-Aware Policies for NISQ-Era Quantum Computers. *arXiv e-prints*, page arXiv:1805.10224, may 2018.

[21] Seyon Sivarajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington, and Ross Duncan. t—ket□: a retargetable compiler for NISQ devices. *Quantum Science and Technology*, 6(1):014003, January 2021.

[22] Navin Khaneja and Steffen Glaser. Cartan decomposition of su(2n) and control of spin systems. *Chemical Physics*, 267:11–23, 06 2001.

[23] Farrokh Vatan and Colin Williams. Optimal quantum circuits for general two-qubit gates. *Physical Review A*, 69(3):032315, March 2004.

[24] Wikipedia. Euler-angles. `https://en.wikipedia.org/wiki/Euler_angles`, 2021. Online; accessed 03 December 2021.

[25] M. Blaauboer and R. Visser. An analytical decomposition protocol for optimal implementation of two-qubit entangling gates. *Journal of Physics A: Mathematical and Theoretical*, 41, 10 2006.

[26] Alwin Zulehner and Robert Wille. Compiling SU(4) Quantum Circuits to IBM QX Architectures. *arXiv e-prints*, page arXiv:1808.05661, August 2018.

[27] Hector Miller-Bakewell, John van de Wetering. ZX Calculus. `https://zxcalculus.com`, 2021. Online; accessed 01 December 2021.

[28] Bob Coecke and Aleks Kissinger. Picturing quantum processes. In Peter Chapman, Gem Stapleton, Amirouche Moktefi, Sarah Perez-Kriz, and Francesco Bellucci, editors, *Diagrammatic Representation and Inference*, pages 28–31, Cham, 2018. Springer International Publishing.

[29] Alexander Cowtan, Silas Dilkes, Ross Duncan, Will Simmons, and Seyon Sivarajah. Phase Gadget Synthesis for Shallow Circuits. *arXiv e-prints*, page arXiv:1906.01734, June 2019.

[30] Katsuhisa Yamanaka, Erik D. Demaine, Takehiro Ito, Jun Kawahara, Masashi Kiyomi, Yoshio Okamoto, Toshiki Saitoh, Akira Suzuki, Kei Uchizawa, and Takeaki Uno. Swapping labeled tokens on graphs. *Theoretical Computer Science*, 586:81–94, 2015. Fun with Algorithms.

[31] Edouard Bonnet, Tillmann Miltzow, and Pawel Rzazewski. Complexity of Token Swapping and its Variants. *arXiv e-prints*, page arXiv:1607.07676, July 2016.

[32] Marcos Yukio Siraichi, Vinícius Fernandes dos Santos, Caroline Collange, and Fernando Magno Quintão Pereira. Qubit allocation as a combination of subgraph isomorphism and token swapping. *Proc. ACM Program. Lang.*, 3(OOPSLA), oct 2019.

[33] Prakash Murali, Jonathan M. Baker, Ali Javadi Abhari, Frederic T. Chong, and Margaret Martonosi. Noise-Adaptive Compiler Mappings for Noisy Intermediate-Scale Quantum Computers. *arXiv e-prints*, page arXiv:1901.11054, January 2019.

[34] Lingling Lao and Dan Browne. 2QAN: A quantum compiler for 2-local qubit Hamiltonian simulation algorithms. *arXiv e-prints*, page arXiv:2108.02099, August 2021.

[35] Joel J. Wallman and Joseph Emerson. Noise tailoring for scalable quantum computation via randomized compiling. *arXiv e-prints*, 94(5):052325, nov 2016.

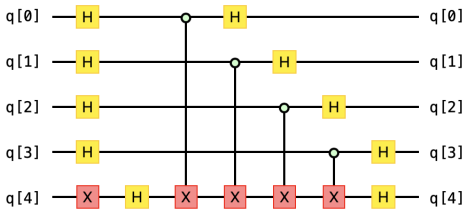
Appendix - Compiling BV5 algorithm on IBMQ-Quito

The following code snippet illustrates how the Bernstein-Vazirani algorithm with 5 qubits can be compiled to satisfy the device characteristics of a real 5-qubit superconducting processor by IBM, namely IBMQ-Quito.

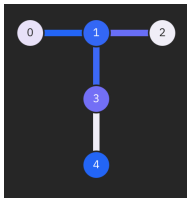
```
from pytket import Circuit
from pytket.circuit.display import render_circuit_jupyter
import numpy as np

def BV_circuit(qubit_num):
    qubits = np.arange(0,qubit_num,1)
    BV = Circuit(qubit_num)
    BV.X(qubits[-1])
    for i in range(qubit_num):
        BV.H(i)
    for control in qubits[:-1]:
        BV.CX(control,qubits[-1])
    for i in range(qubit_num):
        BV.H(i)
    return BV

BV5 = BV_circuit(5)
render_circuit_jupyter(BV5) # plot circuit
```



(a) Bernstein-Vazirani algorithm with 5 qubits.



(b) Connectivity map of IBMQ-Quito.

Figure 5: Compiling the BV5 circuit.

```
# IBMQ account credentials
from qiskit import IBMQ
IBMQ.load_account()
IBMQ.providers()

# pytket-qiskit plug-in
from pytket.extensions.qiskit import IBMQBackend
backend = IBMQBackend("ibmq_quito") # import backend data

backend.compile_circuit(BV5) # full compilation pass
render_circuit_jupyter(BV5)
```

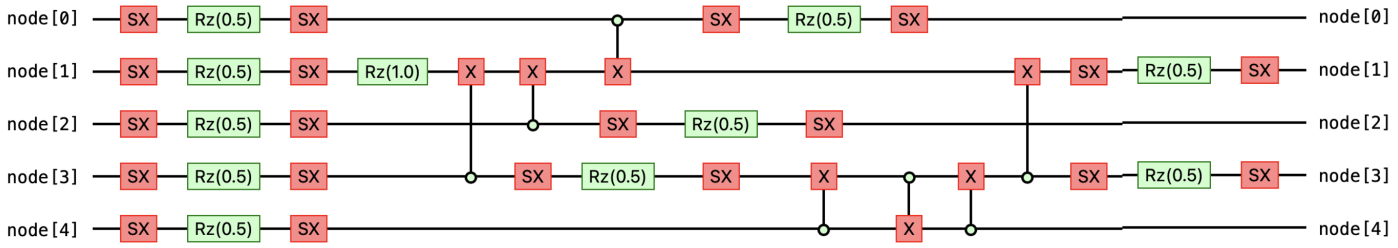


Figure 6: BV5 circuit post-compilation. All the gates are expressed in the basis gate set of the device and as the qubit that interacts the most with other qubits, q[4] is mapped to node[1] of the IBMQ Quito device to minimise SWAPs.