UNIVERSIDAD NEBRIJA

Polytechnic Higher School

Measuring the Quantum Volume of Quantum Devices

Master in Quantum Computing

Author: **Guillermo Mijares Vilariño**

Director: Francisco Gálvez Ramírez

Codirector: Junye Huang

Madrid 2022

Contents

List of Figures

List of Tables

Acknowledgements

I want to sincerely thank Junye Huang for all his continuous support and guidance not only on this work but also on my inclusion in IBM. I also want to express my gratitude towards Doug McClure, Petar Jurcevic and Lev Bishop for taking some time to share their valuable insights on the Quantum Volume process with us. I'm also very grateful to IBM for giving me the opportunity to access its quantum devices for my research. I want to particularly thank my family and my girlfriend for being there to support me.

Abstract

Quantum computing has great potential in many areas like cybersecurity, artificial intelligence, logistics and chemistry. Because of this and the the wide array of proposed quantum systems that can act as quantum hardware, there is a growing interest in benchmarking the performance of each physical realization. There are many types of benchmark that prioritize different characteristics like speed or quality. In this work, we focused on the Quantum Volume benchmark, a single-number metric introduced by IBM to measure the quality of quantum devices by comparing the experimental results of random square quantum circuits with those of an ideal classically-simulated device. We provided an overview of the different ways a Quantum Volume test can be programmed using Qiskit, an open-source quantum Software Development Kit (SDK) created by IBM. Then, we studied algorithms like noise adaptive layout and SWAP-based BidiREctional (SABRE) heuristic algorithms, as well as the preset optimizers included in Qiskit and their implementation to improve the Quantum Volume result. We used these techniques to measure the Quantum Volume of 7 quantum devices, of which 3 had 27 qubits, 3 had 7 qubits and one had 5 qubits. We achieved Quantum Volume 16 in 3 devices and 8 in the others. We also compared the results of bigger and smaller devices for Quantum Volume 8 and 16 tests, where smaller devices tended to fare better in Quantum Volume 8 test while there was no difference for Quantum Volume 16 tests. The effects of SABRE and noise adaptive layouts were also compared, with SABRE being generally better than noise adaptive for smaller devices while bigger ones benefited more from noise adaptive. We also analyzed the effect of the optimization level and found out that for Quantum Volume 8 tests, it was less noticeable for smaller devices while for Quantum Volume 16 the results varied from device to device. The number of circuits used for each Quantum Volume experiment did not have a noticeable effect on the probability of getting outputs with higher probability than the mean but the width of the confidence interval used to determine if the test is passed is inversely proportional to the square root of the number of circuits.

Chapter 1

Introduction

1.1 Quantum computing

Quantum computing is a relatively new field of physics that has the power to shape our future. It has the potential to impact disciplines like cybersecurity, database search, artificial intelligence, logistics and chemistry.

Most of our personal information and even our bank account are protected by classical algorithms that are based on the assumed difficulty of solving some mathematical problems. One example of such algorithms is the ubiquitous RSA (Rivest et al. n.d.), invented by Rivest, Shamir and Adleman. This algorithm owes its security to the fact that, given an integer N that is the product of two sufficiently big prime numbers p and q, it is very hard to obtain p and q even with a cutting-edge supercomputer. However, Shor's factoring algorithms (Shor 1997) prove, at least theoretically, that a quantum computer can obtain these prime factors exponentially faster than any classical compute. The biggest challenges quantum computers are currently facing are scalability and noise. In other words, quantum computers are still too small and prone to errors to actually break RSA. Even though this can make some people consider quantum computing to be a future threat, it has opened a new world of possibilities in cryptography, a new discipline called post quantum cryptography (Campagna et al. 2021) (Kumar 2022), that aims to create new encryption schemes that are robust even against a quantum computer.

Database search can be enhanced with Grover's algorithm (Grover 1996). If we have a large database with no particular structure and a boolean function that indicates whether the answer is the correct one, this algorithm has shown that a quantum computer only needs to evaluate $\mathcal{O}(\sqrt{N})$ times the function instead of the $\mathcal{O}(N)$ a classical computer requires, where N is the number of elements in the database. A particular application of this algorithm can be cracking a password, where the database would be the set of every possible input.

Machine learning is also benefiting from quantum computing. Even though fully quantum machine learning is still in its infancy and it is not possible to implement it with the current devices (Dunjko et al. 2016) (Biamonte et al. 2017), hybrid approaches are giving promising results. There are two different types of hybrid machine learning algorithms: those that consist of applying classical machine learning tools to quantum data, like characterizing and controlling quantum bits (qubits) and those about processing classical data with quantum tools. One of the most famous examples of algorithms of the second kind is the Variational Quantum Classifier (VQC) (Havlíček et al. 2019) (Peruzzo et al. 2014), in which the data is encoded into a quantum state with a parametrized quantum circuit called feature map and then this quantumized data goes through another parametrized circuit called ansatz and is finally measured. The resulting bitstrings are used to assign labels to the data and a classical optimization algorithm is used to update the anzatz parameter and all the process is repeated until the optimal value of the parameter is found. Another well-known algorithm is the Quantum Support Vector Classifier (QSVC), in which feature maps are used to get a kernel matrix that is introduced into a classical support vector classification algorithm (Schuld & Killoran 2019) (Havlíček et al. 2019).

Logistics are being improved by the usage of the Quantum Approximate Optimization Algorithm (QAOA) (Farhi et al. 2014) to solve optimization problems like the vehicle routing problem (Utkarsh et al. 2020). Another problem of interest to logistics that can be solved with quantum computers is the travelling salesman problem, that can be solved by the phase estimation algorithm introduced in Kitaev (1995) (Srinivasan et al. 2018).

In chemistry, quantum computers can make a difference, given that a classical computer is unable to simulate quantum systems like molecules or atoms efficiently, while a quantum computer is a quantum system itself (Kandala et al. 2017). But this is not the only use of quantum simulation, as nanotechnology and solid state physics also rely heavily on quantum mechanics.

1.2 Comparing different quantum devices

Given that quantum computing can be used in a wide array of disciplines and can be implemented in very different ways, each one with disparate characteristics, an immediate question that arises is which quantum technology is the most useful and how we can compare them.

There are some properties that are desirable in a quantum computer such as a high number of qubits, a lower gate error rate, higher coherence times, better connectivity, a reliable initialization and dependable measures. However, each technology is better in some of these aspects and worse in others, so it is not straightforward to compare, for example, ion traps and superconducting qubits. That means we need a reliable way to benchmark the performance of these computers in the same way LINPACK (Dongarra et al. 1979) (Dongarra et al. 2003) is used to compare the fastest classical supercomputers in the TOP500 list (*TOP500 page* 2022). This benchmark measures how many double precision floating point operations per second (FLOPS) a computer can achieve when solving a linear equation system using the LU factorization with partial pivoting (*TOP500 page* 2022).

There are several proposed quantum benchmarks like IonQ's algorithmic qubits (*Algorithmic Qubits: A Better Single-Number Metric* 2022), IBM's CLOPS (Circuit Layer Operations Per Second) (Wack et al. 2021) and Quantum Volume (Cross et al. 2019), the latter of which will be the focus of this work.

1.3 Thesis outline

This work will be organized as follows: chapter 2 will provide some background information to justify the existence of Quantum Volume benchmark. We will start by discussing some of the different quantum hardware implementations and their characteristics. Then, we will go through the different ways to benchmark the performance of each quantum device in chapter 2.2, focusing on those that consider the system as a whole, that is, the holistic benchmarks (section 2.2.1). After that we will focus on the Quantum Volume benchmark in section 2.3, reviewing its steps in sections 2.3.1, 2.3.2, 2.3.3. Once we have reviewed the theory behind Quantum Volume, we will proceed to implement it using Qiskit in chapter 3, firstly step by step (section 3.1) and then using an encapsulated version from the module Qiskit Experiments (section 3.2). After going through the basic Quantum Volume test, we will discuss some algorithms that can be implemented with a circuit-to-circuit transpiler to improve the results in chapter 4. In particular, the noise adaptive layout, SABRE and preset transpiler pass managers will be explained in sections 4.1, 4.2 and 4.3, respectively. Then, in section 4.4 we will use Qiskit to implement all these optimization techniques. We will analyze the results in chapter 5, first comparing the Quantum Volume test results of some of IBM's 27-qubit, 7-qubit and 5-qubit quantum devices for Quantum Volume 8 and 16 tests without any optimization (section 5.1), then comparing the effects of SABRE and noise adaptive layout algorithms (section 5.2), the chosen preset pass manager (section 5.3), the number of circuits used for the test (section 5.4) and finally, a general appreciation of the obtained Quantum Volume results compared to the reported ones from IBM (section 5.5).

Chapter 2

Literature Review

2.1 Quantum hardware

2.1.1 Conditions

One of the particularities of quantum computers is that they can be built, in theory, from any kind of quantum system as long as it follows DiVicenzo's criteria (DiVincenzo 2000) for true quantum computation:

- Computer based on well-characterized and scalable qubits.
- System initializable to a well-defined state.
- Coherence maintained during operation time.
- Ability to transform the qubits with a universal set of quantum gates.
- Reliable state measures at the end of computation.

These conditions imply that the quantum system has to be divided in subsystems with 2 clearly defined states each. Those states will be labeled as $|0\rangle$ and $|1\rangle$, in order to keep the analogy with the classical bits. Given that computation, be it classical or quantum, is based on performing operations on the corresponding bits, it is essential to be able to reset the system to a specific state and that the environment does not make any undesired changes on the system until we are finished. The array of available quantum operators should be as wide as possible, so we will be interested in having a universal set of quantum gates, that is a set from which every quantum operator can be built. Once the computation is finished, for it to be of any use, the final state must be measured reliably.

2.1.2 Physical realizations

Over the years there have been a wide variety of physical systems proposed as qubit implementations. Some of them are:

- Photons.
  - Polarization (Kok et al. 2007).
  - Time of arrival (Pilnyak et al. 2019).
- Coherent states of light (Asbóth et al. 2004).
- Nuclear spin (Thiele et al. 2014)
- Energy levels of trapped ions (Cirac & Zoller 1995).
- Superconducting circuits (Makhlin et al. 2001).
  - Charge qubit.
  - Flux qubit.
  - Phase qubit.
- Quantum dots (Loss & DiVincenzo 1998).
- Vibrational states (Tesch & Vivie-Riedle 2002).

However, not all of them are equally successful in practice for many different reasons. Right now, the technologies that are showing the most promising results are superconducting circuits and ion traps (Bruzewicz et al. 2019) and (Makhlin et al. 2001).

Superconducting qubits

These qubits consist of circuits with no resistance to the flow of electrical current that are manipulated with microwave pulses.

Superconducting qubits have gate times of around 100 ns (Kjaergaard et al. 2020) and gate fidelity close to 99% for 2-qubit gates. However, they are very scalable thanks to the possibility of adapting existing technologies like microchips. IBM's latest quantum processor Eagle has 127 superconducting qubits (*IBM Quantum Roadmap* 2022). Nonetheless, the connectivity is limited to neighbouring qubits and the coherence time is low, around 50 or 100 μs (Kjaergaard et al. 2020). Other restrictions this type of qubit has are that it requires temperatures close to 10 mK (Kjaergaard et al. 2020) and that superconducting qubits are not all identical due to nanofabrication variations, so they need to be calibrated separately.

They are used by IBM, Google, Rigetti, Intel and others.

Ion traps

This type of qubit consists of ionized atoms trapped in a lattice by oscillating electromagnetic fields. The gates are implemented via laser pulses.

Ion traps have high coherence times, close to 1 s (Gaebler et al. 2016) and all-to-all connectivity. Their gates have better fidelity than those of superconducting qubits (around 99.9% for 2-qubit gates (Gaebler et al. 2016)). However, they have slower gates of around 10 μs for 2-qubit gates (Gaebler et al. 2016) and are not easily scalable because increasing the number of ions creates mutual interference. For instance, IonQ's latest quantum computer, IonQ Forte has 35 qubits (*Unveiling IonQ Forte: The First Software-Configurable Quantum Computer* 2022). The technologies necessary to implement them are not as mature as the ones in superconducting qubits.

They are used by IonQ, Quantinuum and Alpine Quantum Technologies among others.

2.2 Quantum benchmarking

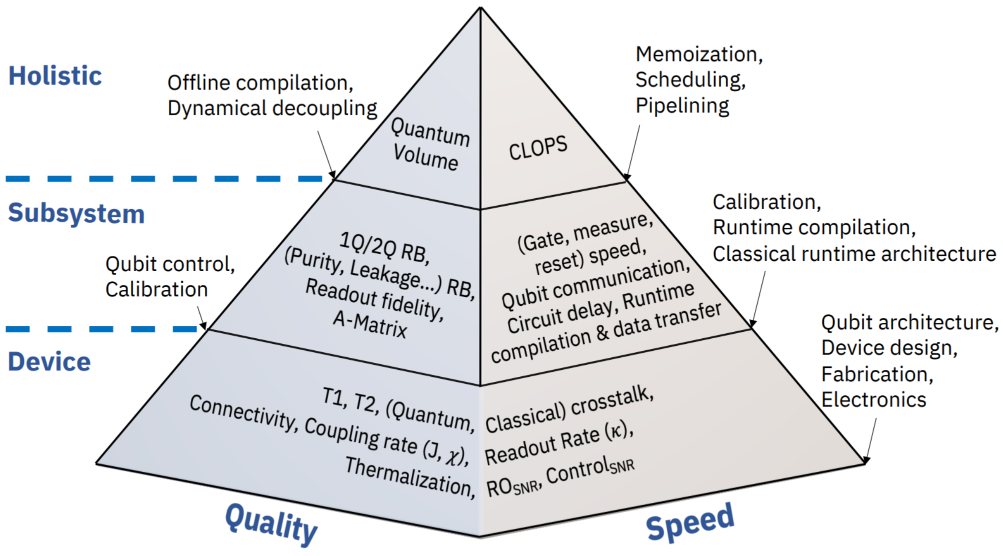
There are many different ways to benchmark the performance of a quantum computer, depending on whether we focus on quality or speed, between which there can be some tradeoffs, and the complexity. We can classify the benchmarks by complexity in device, subsystem and holistic:

- Device benchmarks: they consist of properties that are intrinsic to the device itself, like the coherence times T_1 and T_2, the crosstalk between qubits and qubit connectivity. These benchmarks don't consider any job or process performed by the device.
- Subsystem benchmarks: characterize the performance of part of the device, that is, a subset of qubits. They are influenced by the device benchmarks but subsystem benchmarks generally cannot be fully explained from them. However it is more straightforward to understand how subsystem benchmarks are influenced by device ones than how holistic benchmarks are affected, as fewer device benchmarks are present. Some examples of those are gate, measure and reset speed and readout fidelity.

- Holistic benchmarks: they measure how successfully the quantum computer as a whole can carry out a specific task. They are affected by the above benchmarks, but it is much harder to determine how each one affects the outcome and their importance. This type will be the focus of this work.

Device benchmark

A representation of some of the different benchmarks and how they are classified according to both complexity and priority can be seen in figure 2.1.



Holistic
Subsystem
Device
Offline compilation, Dynamical decoupling
Quantum Volume
CLOPS
Memoization, Scheduling, Pipelining
Qubit control, Calibration
1Q/2Q RB, (Purity, Leakage...) RB, Readout fidelity, A-Matrix
(Gate, measure, reset) speed, Qubit communication, Circuit delay, Runtime compilation & data transfer
Calibration, Runtime compilation, Classical runtime architecture
T1, T2, (Quantum, Connectivity, Coupling rate (J, χ), Thermalization,
Classical) crosstalk, Readout Rate (κ), RO_SNR, Control_SNR
Qubit architecture, Device design, Fabrication, Electronics
Quality
Speed

Figure 2.1: Benchmarking pyramid whose faces represent quality and speed. The higher a benchmark is, the more complex but less specific. Image taken from Wack et al. (2021).

2.2.1 Holistic benchmarks

There have been many different proposals for this type of benchmarks like cross-entropy benchmarking (Arute et al. 2019), IBM's Quantum Volume (Cross et al. 2019) and CLOPS (Circuit Layer Operations Per Second) (Wack et al. 2021), IonQ's algorithmic qubits (*Algorithmic Qubits: A Better Single-Number Metric* 2022).

The idea behind cross-entropy benchmarking is generating random circuits of any desired width and depth, then simulating their ideal probabilities, running it on a real device to obtain the experimental probabilities. Then, with those probabilities, it is possible to compute an approximation of the fidelity of the circuit (Arute et al. 2019)(*Cross Entropy Benchmarking Theory* 2022). So this test measures how well a quantum computer can run random circuits of any desired size and the result is not a success/failure but a number.

Algorithmic qubits, on the other hand, instead of random quantum circuits, focused on a specific set of known quantum algorithms like Shor's (Shor 1997), Grover's (Grover 1996) and Deutsch-Jozsa's (Deutsch & Jozsa 1992). Each circuit is then simulated classically to get the ideal probabilities and run on a real device without optimization to get the experimental probabilities. These probabilities are used to compute the classical fidelity, that has to pass the threshold $t = 1/e = 0.37$ within a σ confidence interval. The algorithmic qubits are then given by the highest number of qubits n for which all circuits of width $w \leq n$ and depth $d \leq n^2$ pass the threshold. This means that this is a single-number metric.

CLOPS measures how many random circuit layers (as in section 2.3.1) can be run by a quantum computer in a set amount of time, taking into account the repetition rate of the quantum processor, the speed at which gates run, the runtime compilation, the amount of time needed to generate the classical control instructions, and the data transfer rate among all units. This is also a single-number metric.

Quantum Volume measures the biggest random square circuit that can be reliably run on a particular quantum device. This benchmark will be the focus of this thesis.

2.3 Quantum Volume

The idea behind Quantum Volume is looking for the biggest random square circuits that can be successfully executed on a specific quantum computer. If d is the highest width (number of qubits) and depth achieved, then the Quantum Volume will be given by 2^d.

The protocol to obtain the Quantum Volume of a quantum computer (Cross et al. 2019) (Asfaw et al. 2020) consists of the following steps:

- Generate random square circuit.
- Simulate the circuit and record its results.
- Run the circuit on the real device and record its results.
- Check whether the results are statistically close to the simulated ones.

2.3.1 Random square circuit

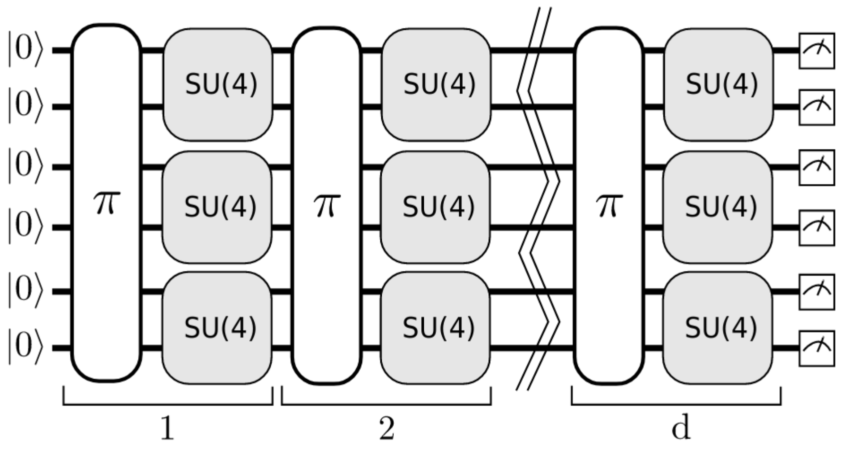


Figure 2.2: Model Quantum Volume circuit. Image taken from (Cross et al. 2019)

A random square circuit of width and depth d is a d-qubit circuit formed by d layers. Each one of these layers consists of a random permutation of all the qubits in the circuit and then a random 2-qubit unitary operator (i.e. a random $SU(4)$ operator) for each pair of adjacent qubits. If the number of qubits is odd, the last one will not be affected by any 2-qubit operator. However, given that at the start of each layer we permutate all the qubits, this does not constitute any problem.

Even though this structure can be perceived as too restrictive, the set of 2-qubit operators is actually universal (DiVincenzo 1995), so every quantum operator can be written with respect to them.

Mathematically, we will define a random square circuit of width and depth d as a sequence of d layers $U = U^{(d)}...U^{(2)}U^{(1)}$, where each $U^{(i)}$ is given by

$$U^{(i)} = U^{(i)}_{\pi_i(2\lfloor\frac{d}{2}\rfloor-1)\pi_i(2\lfloor\frac{d}{2}\rfloor)} \otimes ... \otimes U^{(i)}_{\pi_i(1)\pi_i(2)} = \bigotimes_{j=0}^{\lfloor\frac{d}{2}\rfloor-1} U^{(i)}_{\pi_i(2\lfloor\frac{d}{2}\rfloor-(2j+1))\pi_i(2\lfloor\frac{d}{2}\rfloor-2j)}, \tag{2.1}$$

where $\pi_i \in S_d$ is a uniformly random permutation of the qubits and each $U^{(i)}_{a,b}$ is a random element of $SU(4)$ acting on qubits a and b.

2.3.2 Simulate the circuit

Once we have a random square circuit, the next step is simulating it classically. This is done in order to know the ideal probability distribution of the circuit outputs. It is particularly important to note that, given that this process is completely classical, the computational resources required scale exponentially with t, so this can become unfeasible for bigger circuits in the long term. However we are still far from that, as the current highest reported Quantum Volume is 4096 (*Quantinuum Announces Quantum Volume 4096 Achievement* 2022) (equivalent to a 12-qubit circuit), achieved by Quantinuum by relaxing the usual Quantum Volume bounds, as explained in Baldwin et al. (2022) while IBM's general purpose simulator `ibmq_qasm_simulator` can simulate 32 qubits (*IBM Quantum simulators overview* 2022).

Moreover, if we consider that to store a complex number with double precision we need 16 bytes (16 B), 8 for real part and 8 for imaginary, and an n-qubit state requires 2^n complex numbers, we deduce that to simulate a system with n qubits, we need a memory M of:

$$M = 2^n \cdot 16\,\mathrm{B} \tag{2.2}$$

If we take, for example, IBM's Summit supercomputer, whose RAM is 2 801 664 GB (*Summit Oak Ridge National Laboratory's 200 petaflop supercomputer* 2022), it follows from equation 2.2 that

$$n = \log_2\left(\frac{M}{16}\right) = \log_2\left(\frac{2\,801\,664\,\mathrm{GB}}{16\,\mathrm{B}}\right) = \log_2\left(\frac{(1024)^3 \cdot 2\,801\,664\,\mathrm{B}}{16\,\mathrm{B}}\right) \approx 47, \tag{2.3}$$

so with a current supercomputer we can simulate a 47-qubit circuit.

After addressing the computational resources required for simulating quantum circuits, let's look at the results of the simulations. Given a random square circuit U of width and depth d, the ideal probability of measuring $x \in [0, 2^d - 1]$ is given by

$$p_U(x) = |\langle x| U |0\rangle|^2 \tag{2.4}$$

We are interested in the heavy outputs, that are the outputs with a higher probability than the median. Mathematically, the set of heavy outputs H_U from the circuit U are

$$H_U = \{x \in [0, 2^d - 1] \text{ such that } p_U(x) \geq p_{median}\}, \tag{2.5}$$

where, if we order the probabilities as $p_0 \leq p_1 \leq ... \leq p_{2^d-1}$, then

$$p_{median} = \frac{p_{2^{d-1}-1} + p_{2^{d-1}}}{2}. \tag{2.6}$$

A question that immediately arises is how often we should expect to find heavy outputs on a random circuit. It can be proven that for a random $SU(4)$ matrix, U, each $2^d |\langle x| U |0\rangle|^2$ follows an exponential distribution with parameter $\lambda = 1$, that is, $2^d |\langle x| U |0\rangle|^2 \equiv Exp(1)$ (Aaronson & Chen 2016).

So in order to find the median of these probabilities, we need to find the value m that satisfies

$$\int_0^m e^{-z} dz = \frac{1}{2} \tag{2.7}$$

Solving the integral on the left hand side of 2.7 gives

$$\int_0^m e^{-z} dz = [-e^{-z}]_0^m = 1 - e^{-m}, \tag{2.8}$$

so 2.7 becomes

$$1 - e^{-m} = \tfrac{1}{2} \quad \rightarrow \quad e^{-m} = \tfrac{1}{2} \quad \rightarrow \quad m = \ln(2) \tag{2.9}$$

The heavy outputs would be then

$$H_U = \{x \in [0, 2^d - 1 \text{ such that } 2^d |\langle x| U |0\rangle|^2 \geq \ln(2)]\} \tag{2.10}$$

And the mean value of $2^d |\langle x| U |0\rangle|^2$ over the heavy outputs would be

$$\int_{\ln(2)}^{\infty} z e^{-z} dz = [-z e^{-z}]_{\ln(2)}^{\infty} + \int_{\ln(2)}^{\infty} e^{-z} dz = \ln(2)\frac{1}{2} + \frac{1}{2} = \frac{1 + \ln(2)}{2} \approx 0.85 \tag{2.11}$$

The probability that a particular x is heavy is then the result of 2.11 divided by 2^d, that is, $\frac{1+\ln(2)}{2^{d+1}}$. Given that the corresponding distributions for the other $2^d - 1$ values are identical and independent, the probability of obtaining a heavy output would be the sum of the probabilities that each one of the 2^d x is heavy so we would recover the result from 2.11.

2.3.3 Run the circuit on a real quantum device

Once we have the list of heavy outputs of our quantum circuit, the next step is running it on an actual quantum computer and check what cumulative heavy output probability (HOP) we obtain.

In order to do that, first it is necessary to adapt the circuit to the limitations of the hardware. For example, if the device on which we want to run the circuit has limited connectivity, some

SWAPs may have to be added to swap quantum states among qubits. It is also possible that some gates in the circuit are not directly supported by the computer and have to be expressed in function of other gates.

In order to get the best possible Quantum Volume every trick is nevertheless allowed, as long as an honest attempt is made at recreating the circuit. Some of these possible tricks include the usage of a circuit-to-circuit transpiler to look for the best qubit placement and layout or cancelling and merging gates.

The objective is to get a high enough percentage of heavy outputs. The question that immediately arises is what is considered to be enough. In the previous section (2.3.2) it has been discussed that the probability of heavy outputs on an ideal device would be around 0.85. However, we cannot reasonably expect any current device to act exactly like an ideal one. On the other side, the worst-case scenario probability of heavy outputs, corresponding to a completely depolarized device, would be 0.5. One can envision this case as a circuit in which, because of noise, all outputs have more or less the same probability.

A middle ground between the best and worst cases would then be $\frac{0.85+0.5}{2} = 0.675 \approx 2/3$. In fact, it can be shown that none of the known polynomial time classical algorithms is able to, given a random quantum circuit $\mathcal{C}$, correctly predict whether any random string is a heavy ouput of $\mathcal{C}$ with a probability of success higher than $\frac{1}{2} + \Omega(2^{-n})$, where n is the number of qubits (Aaronson & Chen 2016).

Then if we run n_c circuits of dimension d for n_s times each and get n_h heavy outputs, a reasonable approximation of the probability h_d of getting a heavy output would be

$$\hat{h}_d = \frac{n_h}{n_s n_c} \tag{2.12}$$

However it is not enough to get $\hat{h} > 2/3$ to be sure our experiment is successful, given that we could be getting this result by chance. In order to ensure the significance of our outcomes, we need to resort to confidence intervals related to $\hat{h}$ and h.

Given a circuit labeled as $i \in \{1, ..., n_c\}$, when running it only once we would have a Bernoulli experiment with probability of success h. If we assume a worst-case scenario as in (Cross et al. 2019), in which for a given circuit all the outputs are heavy if and only if the first is heavy, then $\frac{n_{hi}}{n_s}$ would also be a Bernoulli experiment with probability of success h, where n_{hi} is the number of heavy outputs in circuit i, that is, $\frac{n_{hi}}{n_s} \equiv \mathcal{B}(h)$.

If we sum over all the circuits, using A.7, we would get

$$\frac{n_h}{n_s} = \sum_{i=1}^{n_c} \frac{n_{hi}}{n_s} \equiv \mathcal{B}(n_c, h) \tag{2.13}$$

Following section A.5, if we want to be sure with probability 0.975, corresponding to a 2σ confidence interval, that $h > 2/3$ we need to achieve the following inequality:

$$\frac{n_h - 2\sqrt{n_h(n_s - \frac{n_h}{n_c})}}{n_c n_s} > \frac{2}{3}. \tag{2.14}$$

Chapter 3

Obtain Quantum Volume with Qiskit

This chapter will give an overview of how to do a Quantum Volume test with Qiskit (ANIS et al. 2021) on IBM's quantum computers with two different approaches. The first one will be a detailed description of all the steps needed for the test and for the second an encapsulated version included in Qiskit's module `qiskit_experiments`. All the code from this section can be found on (Guillermo-Mijares-Vilarino 2022).

3.1 Step by step

In order to understand how to run a Quantum Volume test with Qiskit we will follow these steps:

- Create a random 2-qubit gate
- Create a gate layer
- Create a Quantum Volume circuit
- Simulate the circuit
- Get the heavy outputs of the circuit
- Run the circuit on a real quantum device
- Run the full Quantum Volume test

The first things we have to consider in order to calculate quantum volume is the amount of qubits and which ones we'll use. This amount of qubits will equal not only the width of the circuit but also its depth.

[1]:
```
# create a list with the qubits we want to use
qubit_list = [i for i in range(5)]

# save the length of the list = number of qubits = depth of the circuits =␣
 ↪width of the circuits
SIZE = len(qubit_list)
```

3.1.1 Create random 2-qubit gates

In order to measure quantum volume, we'll need to create random 2-qubit gates. That can be done with the `random_unitary` function from `qiskit.quantum_info`.

[2]:
```
from qiskit.quantum_info import random_unitary
# create a random 4x4 unitary matrix i.e. a random 2-qubit operator
random_unitary(4)
```

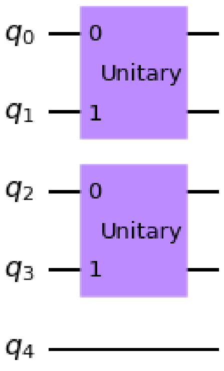
[2]:
```
Operator([[ 0.22135758-0.18654922j, -0.54570703+0.11067592j,
           -0.42865832-0.47181766j,  0.01161354+0.44683348j],
          [ 0.62744197+0.15964268j,  0.05480969+0.4823489j ,
           -0.1231835 +0.06476647j, -0.46251612-0.33447899j],
          [ 0.27109868-0.18618368j,  0.43031042-0.31670409j,
           -0.34289521-0.4415057j ,  0.35190968-0.41234425j],
          [ 0.15128686+0.60496646j,  0.27138866-0.308496j  ,
           -0.44568496+0.2512604j , -0.02823112+0.42396039j]],
         input_dims=(2, 2), output_dims=(2, 2))
```

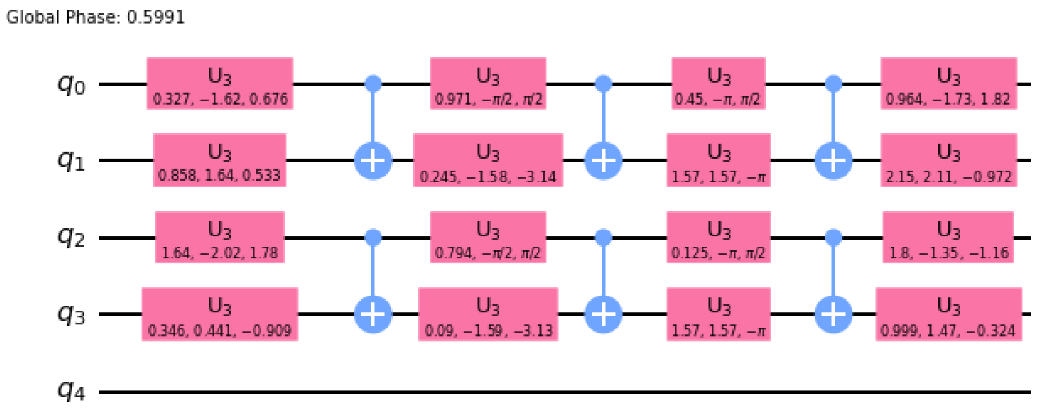
3.1.2 Create a gate layer

Now that we have the needed gates, let's start by using them with pairs of adjacent qubits. If we have an odd number of qubits, one of them will be left unchanged.

[3]:
```
from qiskit import QuantumCircuit

# create the circuit corresponding to the first layer
qv_layer = QuantumCircuit(SIZE)
```

```
# iterate over the pairs of qubits, of which there are the integer part of␣
↪SIZE/2
for pair in range(SIZE//2):
    # choose indexes
    qubit_indices = qubit_list[pair * 2 : pair * 2 + 2]
    # create gate
    gate = random_unitary(4).to_instruction()
    # add gate to circuit
    qv_layer.append(gate, qubit_indices)

# show the layer circuit
display(qv_layer.draw("mpl"))

# show a more detailed version of the layer circuit
display(qv_layer.decompose().draw("mpl"))
```

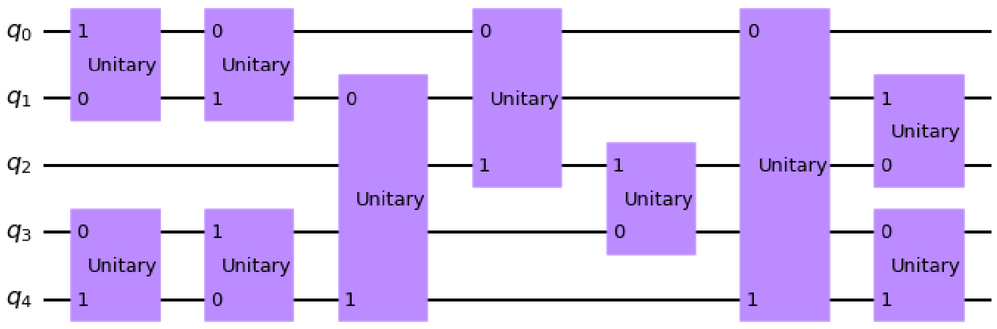




3.1.3 Create a Quantum Volume circuit

The next step will be creating a circuit with as many layers as qubits we have, that is `SIZE` layers, instead of only one. Before each layer we have to randomly permutate the qubits, what can be done with the `shuffle` function from `random`.

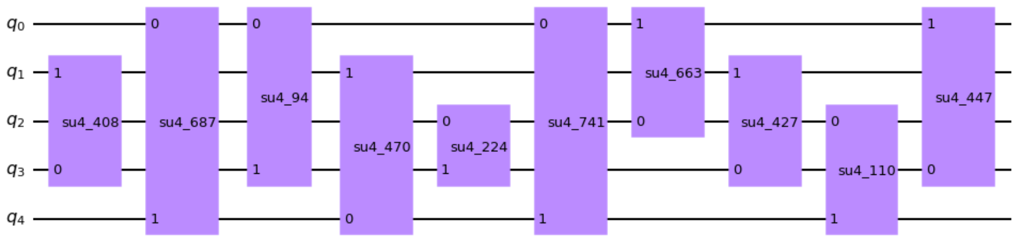
[4]:
```python
from random import shuffle

qlist = qubit_list[:]

# create the circuit
qv_circ = QuantumCircuit(SIZE)

# randomly permutate the list of qubits and add a layer SIZE = depth times
for i in range(SIZE):
    shuffle(qlist)
    for pair in range(SIZE//2):
        qubit_indices = qlist[pair * 2 : pair * 2 + 2]
        gate = random_unitary(4).to_instruction()
        qv_circ.append(gate, qubit_indices)

#show the circuit
display(qv_circ.draw("mpl"))
```



However, `qiskit` already has a class called `QuantumVolume` in its module `qiskit.circuit.library` that directly creates this kind of circuits for us so, in practice, we don't have to worry about how they are built.

[5]:
```python
from qiskit.circuit.library import QuantumVolume

qv_circ = QuantumVolume(SIZE)

qv_circ.decompose().draw("mpl")
```

[5]:



This is the circuit that will be used from now on.

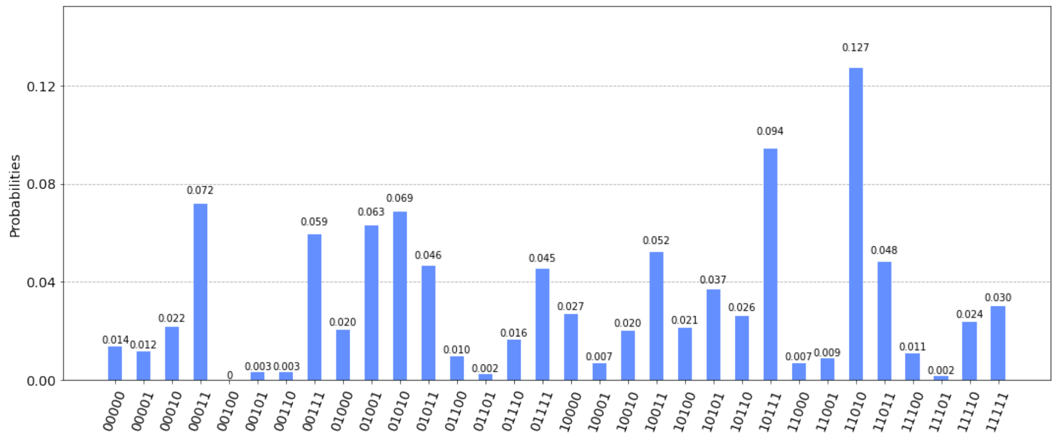
3.1.4 Simulate the circuit

Now we'll use the simulator `aer_simulator` from `Aer` to obtain the ideal probability distribution of the outputs of our circuit. In other words, we'll see what we should get if we ran our circuit with an ideal quantum computer.

[6]:
```python
from qiskit import Aer, transpile
from qiskit.visualization import plot_histogram

# define the simulator
simulator = Aer.get_backend("aer_simulator")

# transpile the circuit so it can be used with the simulator
t_qv_circ = transpile(qv_circ, simulator)

# save the statevector of the circuit
t_qv_circ.save_statevector()

# run the simulator to get the probability distribution from the statevector
counts = simulator.run(t_qv_circ).result().get_counts()

# represent the probability distribution with an histogram
```

```
plot_histogram(counts, figsize = (18,7))
```

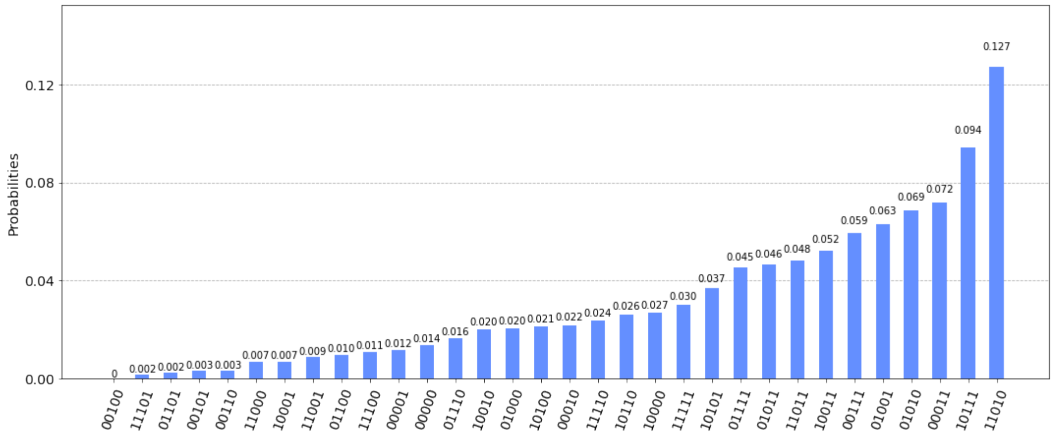
[6]:



3.1.5 Get the heavy outputs of the circuit

The part of the probability distribution that will be interesting to us is the heavy outputs, the outputs whose probability is higher than the median's. We can select these values by ordering our list of outputs by probability with `sorted` and then selecting the second half of the indices.

[7]:
```
# order the bars of the histogram by ascending probability
display(plot_histogram(counts, figsize=(18,7), sort='value'))

# sort the list of outputs by probability
sorted_counts = sorted(counts.keys(), key=counts.get)

# pick the heavy outputs
heavy_outputs = sorted_counts[len(sorted_counts)//2 : ]

# show the heavy outputs
print("The heavy outputs of this circuit are:", heavy_outputs)
```



```
The heavy outputs of this circuit are: ['00010', '11110', '10110', '10000',
'11111', '10101', '01111', '01011', '11011', '10011', '00111', '01001',␣
 ↪'01010',
'00011', '10111', '11010']
```

3.1.6 Run the circuit on a real quantum device

After getting our heavy output list we can start thinking about running our circuit with an actual quantum computer. One way of doing this is by using `IBMQ`. Keep in mind that for this to work it is necessary to have an IBM Quantum account, that can be created for free in (*IBM Quantum* 2022). It can be loaded with the following commands:

```
[8]: from qiskit import IBMQ

     # save IBMQ account
     #IBMQ.save_account(api_key)

     # load IBMQ account
     IBMQ.load_account()
```

```
[8]: <AccountProvider for IBMQ(hub='ibm-q', group='open', project='main')>
```

Once we have loaded the account, we need to choose a provider, from which we will be able to pick our desired device. As an IBM intern, I was granted access to the hub `'ibm-q-internal'`, which has a wider array of available computers than the free basic pack.

```
[9]: # select provider

     provider = IBMQ.get_provider(hub='ibm-q-internal')
```

Let's see what quantum systems we can have access to.

```
[10]: # show backends

      provider.backends()
```

```
[10]: [<IBMQSimulator('ibmq_qasm_simulator') from IBMQ(hub='ibm-q-internal',
      group='deployed', project='default')>,
       <IBMQBackend('ibmq_montreal') from IBMQ(hub='ibm-q-internal',␣
       ↪group='deployed',
      project='default')>,
       <IBMQBackend('ibmq_toronto') from IBMQ(hub='ibm-q-internal',␣
       ↪group='deployed',
      project='default')>,
       <IBMQBackend('ibmq_kolkata') from IBMQ(hub='ibm-q-internal',␣
       ↪group='deployed',
      project='default')>,
       <IBMQBackend('ibmq_mumbai') from IBMQ(hub='ibm-q-internal',␣
       ↪group='deployed',
      project='default')>,
       <IBMQBackend('ibmq_lima') from IBMQ(hub='ibm-q-internal', group='deployed',
      project='default')>,
       <IBMQBackend('ibmq_belem') from IBMQ(hub='ibm-q-internal',␣
       ↪group='deployed',
      project='default')>,
       <IBMQBackend('ibmq_quito') from IBMQ(hub='ibm-q-internal',␣
       ↪group='deployed',
      project='default')>,
       <IBMQBackend('ibmq_guadalupe') from IBMQ(hub='ibm-q-internal',
      group='deployed', project='default')>,
       <IBMQBackend('ibmq_jakarta') from IBMQ(hub='ibm-q-internal',␣
       ↪group='deployed',
project='default')>,
 <IBMQBackend('ibmq_manila') from IBMQ(hub='ibm-q-internal',␣
 ↪group='deployed',
project='default')>,
 <IBMQBackend('ibm_hanoi') from IBMQ(hub='ibm-q-internal', group='deployed',
project='default')>,
 <IBMQBackend('ibm_lagos') from IBMQ(hub='ibm-q-internal', group='deployed',
project='default')>,
 <IBMQBackend('ibm_nairobi') from IBMQ(hub='ibm-q-internal',␣
 ↪group='deployed',
project='default')>,
 <IBMQBackend('ibm_cairo') from IBMQ(hub='ibm-q-internal', group='deployed',
project='default')>,
 <IBMQBackend('ibm_auckland') from IBMQ(hub='ibm-q-internal',␣
 ↪group='deployed',
project='default')>,
 <IBMQBackend('ibm_perth') from IBMQ(hub='ibm-q-internal', group='deployed',
project='default')>,
 <IBMQBackend('ibm_washington') from IBMQ(hub='ibm-q-internal',
group='deployed', project='default')>,
 <IBMQBackend('ibm_oslo') from IBMQ(hub='ibm-q-internal', group='deployed',
project='default')>,
 <IBMQBackend('ibm_geneva') from IBMQ(hub='ibm-q-internal',␣
 ↪group='deployed',
project='default')>]
```

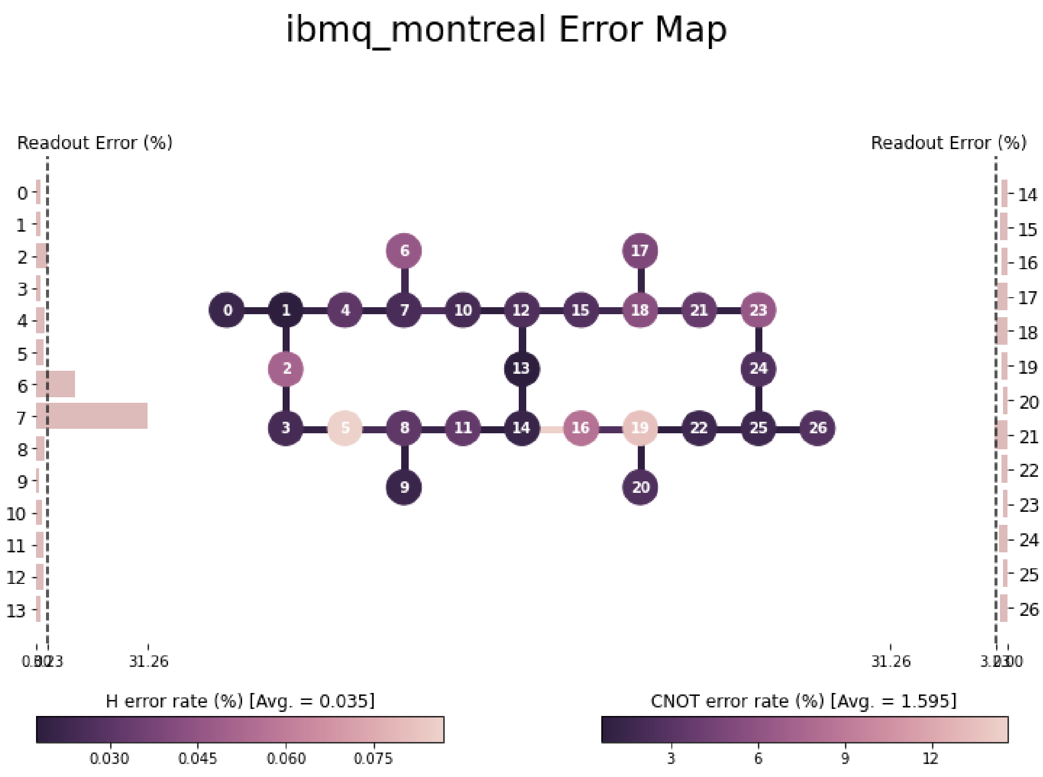
For this example, we will use the `'ibmq_montreal'` backend.

[11]:
```python
# choose quantum computer
backend = provider.get_backend('ibmq_montreal')
```

We can take a look into the coupling graph of the device, the error rate of each qubit, the CNOT error corresponding to each pair of qubits and the error rate of a Hadamard gate applied on each qubit using the `plot_error_map` function from the module `qiskit.visualization`.

[12]:
```python
from qiskit.visualization import plot_error_map

plot_error_map(backend)
```

[12]:



It is also possible to look into some of the properties of the each device by using the IBMQBackend.properties() method. For example, we can look into the error rates of each CNOT gate to get a more exact description than the colors from plot_error_map, that can be obscured if there are some extreme values.

[13]:
```python
for i in backend.properties().gates:
    if "cx" in i.name:
        print(f"{i.name}, that acts on qubits {i.qubits} with error rate {i.
↪parameters[0].value}")
```

```
cx22_19, that acts on qubits [22, 19] with error rate 0.00708146039993382
cx19_22, that acts on qubits [19, 22] with error rate 0.00708146039993382
cx21_18, that acts on qubits [21, 18] with error rate 0.013536296018506977
cx18_21, that acts on qubits [18, 21] with error rate 0.013536296018506977
cx16_19, that acts on qubits [16, 19] with error rate 0.026302695327298453
cx19_16, that acts on qubits [19, 16] with error rate 0.026302695327298453
cx15_18, that acts on qubits [15, 18] with error rate 0.015797993184110765
```

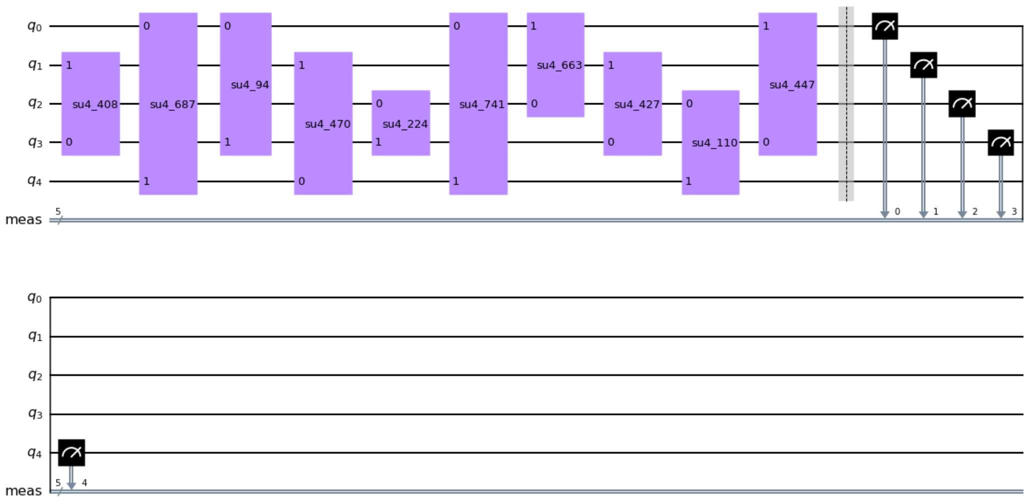
```
cx18_15, that acts on qubits [18, 15] with error rate 0.015797993184110765
cx11_14, that acts on qubits [11, 14] with error rate 0.0060997466690941304
cx14_11, that acts on qubits [14, 11] with error rate 0.0060997466690941304
cx23_21, that acts on qubits [23, 21] with error rate 0.00985424022906986
cx21_23, that acts on qubits [21, 23] with error rate 0.00985424022906986
cx10_7, that acts on qubits [10, 7] with error rate 0.022100143244592474
cx7_10, that acts on qubits [7, 10] with error rate 0.022100143244592474
cx7_6, that acts on qubits [7, 6] with error rate 0.017304717079566934
cx6_7, that acts on qubits [6, 7] with error rate 0.017304717079566934
cx25_22, that acts on qubits [25, 22] with error rate 0.008992448892042898
cx22_25, that acts on qubits [22, 25] with error rate 0.008992448892042898
cx5_8, that acts on qubits [5, 8] with error rate 0.020191319938492824
cx8_5, that acts on qubits [8, 5] with error rate 0.020191319938492824
cx17_18, that acts on qubits [17, 18] with error rate 0.011503405867402033
cx18_17, that acts on qubits [18, 17] with error rate 0.011503405867402033
cx20_19, that acts on qubits [20, 19] with error rate 0.007242342248744066
cx19_20, that acts on qubits [19, 20] with error rate 0.007242342248744066
cx4_7, that acts on qubits [4, 7] with error rate 0.009044494371092948
cx7_4, that acts on qubits [7, 4] with error rate 0.009044494371092948
cx16_14, that acts on qubits [16, 14] with error rate 0.14613891494225134
cx14_16, that acts on qubits [14, 16] with error rate 0.14613891494225134
cx5_3, that acts on qubits [5, 3] with error rate 0.01009856483524027
cx3_5, that acts on qubits [3, 5] with error rate 0.01009856483524027
cx12_13, that acts on qubits [12, 13] with error rate 0.007849394835374446
cx13_12, that acts on qubits [13, 12] with error rate 0.007849394835374446
cx3_2, that acts on qubits [3, 2] with error rate 0.009295332847434723
cx2_3, that acts on qubits [2, 3] with error rate 0.009295332847434723
cx12_10, that acts on qubits [12, 10] with error rate 0.006850408101404054
cx10_12, that acts on qubits [10, 12] with error rate 0.006850408101404054
cx23_24, that acts on qubits [23, 24] with error rate 0.007601349150858416
cx24_23, that acts on qubits [24, 23] with error rate 0.007601349150858416
cx1_4, that acts on qubits [1, 4] with error rate 0.010070635674277878
cx4_1, that acts on qubits [4, 1] with error rate 0.010070635674277878
cx11_8, that acts on qubits [11, 8] with error rate 0.013191992720322898
```

```
cx8_11, that acts on qubits [8, 11] with error rate 0.013191992720322898
cx2_1, that acts on qubits [2, 1] with error rate 0.00876053891962622
cx1_2, that acts on qubits [1, 2] with error rate 0.00876053891962622
cx13_14, that acts on qubits [13, 14] with error rate 0.008215552459451714
cx14_13, that acts on qubits [14, 13] with error rate 0.008215552459451714
cx25_26, that acts on qubits [25, 26] with error rate 0.008214387287877656
cx26_25, that acts on qubits [26, 25] with error rate 0.008214387287877656
cx0_1, that acts on qubits [0, 1] with error rate 0.007235113348252081
cx1_0, that acts on qubits [1, 0] with error rate 0.007235113348252081
cx9_8, that acts on qubits [9, 8] with error rate 0.007624649190245997
cx8_9, that acts on qubits [8, 9] with error rate 0.007624649190245997
cx15_12, that acts on qubits [15, 12] with error rate 0.013120605409075364
cx12_15, that acts on qubits [12, 15] with error rate 0.013120605409075364
cx24_25, that acts on qubits [24, 25] with error rate 0.007166937525262623
cx25_24, that acts on qubits [25, 24] with error rate 0.007166937525262623
```

In order to run the circuit on the quantum computer, we need to add measures to it. If this is not done, we will not get any significant result. A quick way to do this is using the `QuantumCircuit.measure_all()` method, that automatically adds a classical register with as many bits as qubits we have in our circuit and measures all of the qubits into their corresponding bit. After that, we run the circuit `qv_circ` several times, in this case `n_shots` = 1024, and check how many heavy outputs we got.

[14]:
```python
n_shots = 1024

# measure the circuit in order to get something meaningful from the quantum␣
↪computer.
qv_circ.measure_all()

# show the circuit with the measures
display(qv_circ.decompose().draw("mpl"))

# transpile the circuit according to the backend
transpiled_qv_circ = transpile(qv_circ, backend)
```

```
# run the circuit on the backend
job = backend.run(transpiled_qv_circ, shots=n_shots)

# take the results
device_counts = job.result().get_counts()

# initialize number of heavy outputs
n_heavy = 0
# check if they are heavy outputs
for heavy_output in heavy_outputs:
    if heavy_output in device_counts.keys():
        n_heavy += device_counts[heavy_output]
n_heavy = int(n_heavy)

print("After running the circuit", n_shots, "times, we got", n_heavy,␣
 ↪"heavy outputs")
```



```
After running the circuit 1024 times, we got 641 heavy outputs
```

3.1.7 Run the full Quantum Volume test

Now that we have seen how to create a quantum volume circuit, get its heavy outputs and run it on an actual quantum computer, it's time to do the same for `n_circuits` different circuits.

We will first start by creating a list of circuits and another list with their corresponding heavy outputs.

[15]:
```
from qiskit import Aer, transpile, QuantumCircuit, IBMQ
from qiskit.circuit.library import QuantumVolume
import numpy as np

# select the depth and width of the QV circuits
SIZE = 5

# number of times we will run each circuit
n_shots = 10000

# choose simulator
simulator = Aer.get_backend("aer_simulator")

# load IBMQ account
IBMQ.load_account()

# select provider
provider = IBMQ.get_provider(hub = 'ibm-q-internal')

# choose quantum computer
backend = provider.get_backend('ibmq_montreal')

# number of different circuits set to maximum permitted per job with this␣
 ↪particular backend
n_circuits = 1000

# initialize a list of all the circuits
circuit_list = []

# initialize list of heavy outputs
heavy_outputs_list = []
```

```python
for circuit in range(n_circuits):
    # create the QV circuit
    qv_circ = QuantumVolume(SIZE)
    # transpile circuit to use the simulator
    qv_circ_simulator = transpile(qv_circ, simulator)
    # save the statevector of the circuit
    qv_circ_simulator.save_statevector()
    # get probabilities
    counts = simulator.run(qv_circ_simulator).result().get_counts()
    # sort the results by probability
    sorted_counts = sorted(counts.keys(), key = counts.get)
    # pick the heavy outputs
    heavy_outputs = sorted_counts[len(sorted_counts)//2 : ]

    # add heavy outputs to list
    heavy_outputs_list.append(heavy_outputs)

    # measure the circuit before running it through the quantum computer
    qv_circ.measure_all()

    # add the circuit to the list
    circuit_list.append(qv_circ)

# transpile the circuits so they can be run on the quantum computer
transpiled_circuits = transpile(circuit_list, backend)
```

The next step is, as expected, running the circuits on the backend. In theory we would only need to run `backend.run(transpiled_circuits, shots=n_shots)` but there are some limitations that we have to take into account. Each device has a maximum limit of circuits allowed and, if we try to run more than that, we will get an error. This limit can be checked with the following command.

[16]:
```
max_circuits = backend.configuration().max_experiments
max_circuits
```

[16]:
```
900
```

A possible way to circumvent this limit would be breaking the circuit list `transpiled_circuits` into lists of size `max_circuits` and a final list with the rest of executions and then checking for each circuit the number of heavy outputs obtained.

[17]:
```
# initialize the number of heavy outputs
n_heavy = 0

# we split the executions according to the maximum number of circuits␣
↪allowed by the device
max_circuits = backend.configuration().max_experiments

for i in range(n_circuits//max_circuits):
    # run max_circuits circuits on the backend
    job = backend.run(transpiled_circuits[i * max_circuits : (i+1) *␣
↪max_circuits], shots=n_shots)
    # take the results
    device_counts = job.result().get_counts()
    # check if they are heavy outputs
    for j in range(max_circuits):
        for heavy_output in heavy_outputs_list[i * max_circuits + j]:
            if heavy_output in device_counts[j].keys():
                n_heavy += device_counts[j][heavy_output]

if (i+1) * max_circuits != n_circuits:
    # run rest of circuits on the backend if there are any
    job = backend.run(transpiled_circuits[(i+1) * max_circuits :␣
↪n_circuits], shots=n_shots)
    # take the results
    device_counts = job.result().get_counts()
    # check if they are heavy outputs
    for j in range(n_circuits - (i+1) * max_circuits):
        for heavy_output in heavy_outputs_list[(i+1) * max_circuits + j]:
            if heavy_output in device_counts[j].keys():
                n_heavy += device_counts[j][heavy_output]
n_heavy = int(n_heavy)
```

However, Qiskit also includes a job manager called `IBMQJobManager` that already handles this for us in a more optimal way.

[18]:
```
from qiskit.providers.ibmq.managed import IBMQJobManager

# initialize job manager
job_manager = IBMQJobManager()

# run the circuits with the job manager
job = job_manager.run(transpiled_circuits, backend=backend, shots=n_shots)
```

An impotant detail when using the `IBMQJobManager` is that the way to get the results has a slightly different syntax than usual. For a standard job, we would run `job.result().get_counts()` to get a list of dictionaries, each one corresponding to a circuit. However, for this managed job, we can only get the dictionaries one by one with `job.results().get_counts(i)` for the i-th circuit. Note that this time we change `result` for `results`.

Apart from counting the heavy outputs, this time we are going to keep track of the experimental probability of heavy outputs of each circuit, $\frac{n_{hj}}{n_s}$, the cumulated heavy output probability $\frac{\sum_{k=0}^{j} n_{hk}}{(j+1)n_s}$ and the cumulated 2σ given by $\frac{2\sqrt{\sum_{k=0}^{j} n_{hk}\left(n_s - \frac{\sum_{k=0}^{j} n_{hk}}{j+1}\right)}}{(j+1)n_s}$ in order to represent the results of the experiment.

[19]:
```
# initialize the number of heavy outputs
n_heavy = 0

# initialize cumulated heavy output probability (HOP), individual HOP and␣
 ↪2-sigma
hop_cumulated = np.array([])
```

```
hop_individual = []
two_sigma = np.array([])

for j in range(n_circuits):
    # initialize number of heavy outputs of circuit j
    n_heavy_j = 0
    # get dictionary of results from circuit j
    device_counts = job.results().get_counts(j)
    # check which results are heavy outputs
    for heavy_output in heavy_outputs_list[j]:
        if heavy_output in device_counts.keys():
            n_heavy_j += device_counts[heavy_output]
    # add the heavy outputs of the circuit to the total count
    n_heavy += n_heavy_j
    # update individual HOP, cumulated HOP and 2-sigma
    hop_individual.append(n_heavy_j/n_shots)
    hop_cumulated = np.append(hop_cumulated ,n_heavy/((j+1)*n_shots))
    two_sigma = np.append(two_sigma, 2 * np.sqrt( n_heavy * ( n_shots -␣
 ↪n_heavy/(j+1)) )/(n_shots*(j+1)))
```

Now that all the circuits have been run, it's time to check if the quantum volume is achieved. So we want to know whether

$$\frac{n_h - 2\sqrt{n_h(n_s - \frac{n_h}{n_c})}}{n_c n_s} > \frac{2}{3}.$$

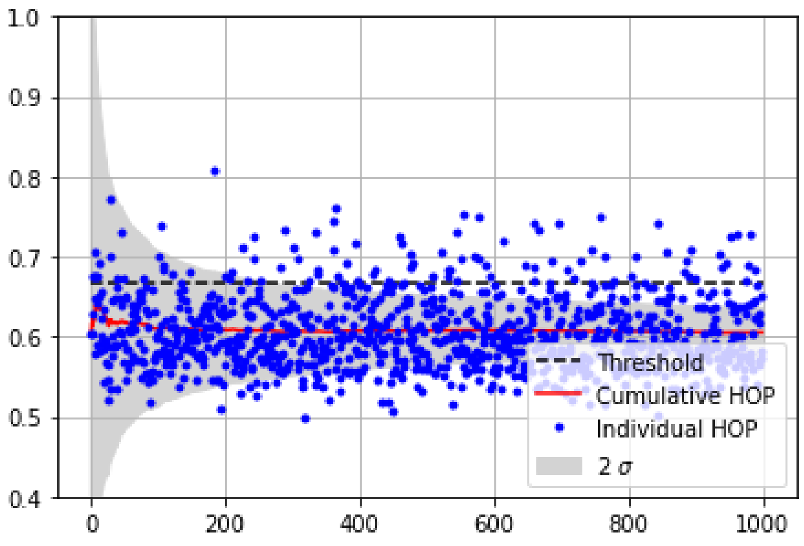
[20]:
```
# show the parameters we'll use to determine the quantum volume
print("n_c: ", n_circuits, "\nn_s: ", n_shots, "\nn_h: ", n_heavy)

# determine if the quantum volume is achieved
if (n_heavy - 2 * np.sqrt(n_heavy * (n_shots - n_heavy / n_circuits)))/
 ↪(n_shots * n_circuits) > 2/3:
    print("Success! Quantum Volume of", 2**SIZE, "achieved!")
else:
    print("Quantum Volume", 2**SIZE ,"test failed, ", (n_heavy - 2 * np.
 ↪sqrt(n_heavy * (n_shots - n_heavy / n_circuits)))/(n_shots * n_circuits),␣
 ↪"< 2/3")
```

```
n_c:  1000
n_s:  10000
n_h:  6052202
Quantum Volume 32 test failed,  0.5743055620747737 < 2/3
```

A way to see how close we got to actually passing the test, it is useful to visually represent the data. In particular we will plot the individual and cumulated heavy output probabilities, the latter of which will be surrounded by a shaded $\pm 2\sigma$ region.

```
[21]: import matplotlib.pyplot as plt


      fig, ax = plt.subplots(1,1)
      ax.plot([0, n_circuits], [2/3, 2/3], '--', color = 'black')
      ax.plot(range(n_circuits), hop_cumulated, '-', color = 'red')
      ax.plot(range(n_circuits), hop_individual, '.', color = 'blue')
      ax.fill_between(range(n_circuits), hop_cumulated - two_sigma, hop_cumulated␣
       ↪+ two_sigma, color = 'lightgrey')
      plt.grid()
      plt.ylim([0.4,1])
      ax.legend(['Threshold', 'Cumulative HOP', 'Individual HOP', '2 $\\sigma$'],␣
       ↪loc = 'lower right')
```

```
[21]: <matplotlib.legend.Legend at 0x7fc6311345e0>
```



The Quantum Volume test would be passed if the lower part of the interval goes above the 2/3 threshold. As we can see, the interval becomes narrower when the number of circuits increases. Generally, if the cumulated HOP is consistently above the threshold, it should be theoretically possible to pass the test with a big enough number of circuits. However, this decrease of width becomes less noticeable when the number of circuits increases, so it can sometimes become unfeasible in practice.

3.2 Qiskit Experiments implementation

In the previous section it has been discussed how to manually create a Quantum Volume test. However, Qiskit already offers an implementation of this test as part of the module `qiskit_experiments`. Like in the previous section, it is necessary to load the IBM Quantum account and select a particular quantum computer, in this case `ibmq_jakarta`.

[1]:
```python
from qiskit import IBMQ

# load IBMQ account
IBMQ.load_account()

# select provider
provider = IBMQ.get_provider(hub = 'ibm-q-internal')

# choose backend
```

```
backend = provider.get_backend('ibmq_jakarta')
```

One of the main differences between this method and the previous one is that to define a `QuantumVolume` experiment from `qiskit_experiments.library` it is required to choose a list of physical qubits to use for the experiment beforehand. It is also possible to choose the number of circuits for the experiment with the `trials` argument, whose default value is 100.

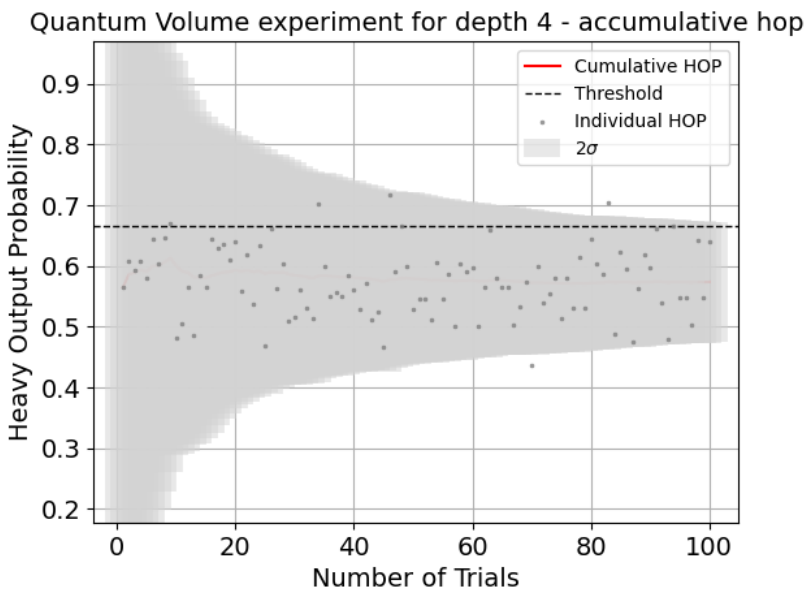
[2]:
```
from qiskit_experiments.library import QuantumVolume

# select physical qubits
qubits = range(4)

# define QV experiment
qv_exp = QuantumVolume(qubits, trials=100)
```

The experiment can then be executed on the device using the `QuantumVolume.run()` method, passing the backend as an argument.

[3]:
```
# run the experiment on the device
expdata = qv_exp.run(backend).block_for_results()
```

Once the experiment has been run, we can get a representation of the data using the `figure()` method with our experiment data. This representation is similar to the one in the previous section, in which the individual and cumulated heavy output probabilities, as well as the $\pm 2\sigma$ interval and the 2/3 thereshold are plotted.

[4]:
```
%matplotlib inline
display(expdata.figure(0))
```



We can access the experimental results with the `analysis_results()` method. There are two types of analysis: `mean_HOP` and `quantum_volume`. The value of the former is given by $\frac{n_h}{n_s n_c} \pm \sigma = \frac{n_h \pm \sqrt{n_h (n_s - \frac{n_h}{n_c})}}{n_s n_c}$ and the latter's is the achieved Quantum Volume, that is, if the experiment was successful we get 2^d, where d is the depth and witdth of the Quantum Volume circuit and if the experiment was not successful, the value is 1. The quality of the result means if the experiment can be considered successful, so it is good in that case and otherwise bad while verified means whether some other person has verified the experimental results.

[5]:
```
for result in expdata.analysis_results():
    print(result)
```

```
DbAnalysisResultV1
- name: mean_HOP
- value: 0.57+/-0.05
- quality: bad
- extra: <4 items>
- device_components: ['Q0', 'Q1', 'Q2', 'Q3']
- verified: False
DbAnalysisResultV1
- name: quantum_volume
- value: 1
- quality: bad
- extra: <4 items>
- device_components: ['Q0', 'Q1', 'Q2', 'Q3']
- verified: False
```

The extra items of each one of the two analysis can be seen with `result.extra.items()`. For `mean_HOP` we get a list formed by the heavy output probabilities of each individual circuit, the value of 2σ, the depth (and width) of the Quantum Volume circuit and the number of trials. For `quantum_volume` what it is obtained is whether the experiment was successful, the confidence level and again the depth and trials.

```
[6]: for result in expdata.analysis_results():
         print(f"\n{result.name} extra:")
         for key, val in result.extra.items():
             print(f"- {key}: {val}")
```

```
mean_HOP extra:
- HOPs: [0.5645, 0.60775, 0.5935, 0.60875, 0.57975, 0.64325, 0.60275, 0.6465,
0.669, 0.48225, 0.505, 0.56575, 0.485, 0.58475, 0.5645, 0.6445, 0.6285, 0.
 ↪63575,
0.60975, 0.6395, 0.5585, 0.619, 0.53625, 0.63375, 0.46875, 0.662, 0.562, 0.
 ↪6035,
0.509, 0.51675, 0.56025, 0.53025, 0.5145, 0.702, 0.59975, 0.55025, 0.55725,
0.55, 0.584, 0.5605, 0.52775, 0.5705, 0.5115, 0.52375, 0.46625, 0.71625,
0.59075, 0.6665, 0.60025, 0.52775, 0.54575, 0.546, 0.51225, 0.60675, 0.5465,
0.58675, 0.49975, 0.604, 0.5915, 0.59725, 0.5015, 0.56475, 0.65875, 0.581,
0.56425, 0.564, 0.50325, 0.5335, 0.57425, 0.4355, 0.599, 0.53975, 0.555, 0.
 ↪581,
0.51325, 0.5795, 0.53125, 0.615, 0.53125, 0.64325, 0.604, 0.58575, 0.704,
0.48725, 0.62325, 0.59425, 0.47425, 0.5635, 0.6175, 0.59675, 0.66225, 0.5385,
0.48025, 0.66525, 0.548, 0.54775, 0.503, 0.64275, 0.54875, 0.64075]
- two_sigma: 0.09890657836438384
- depth: 4
- trials: 100


quantum_volume extra:
```

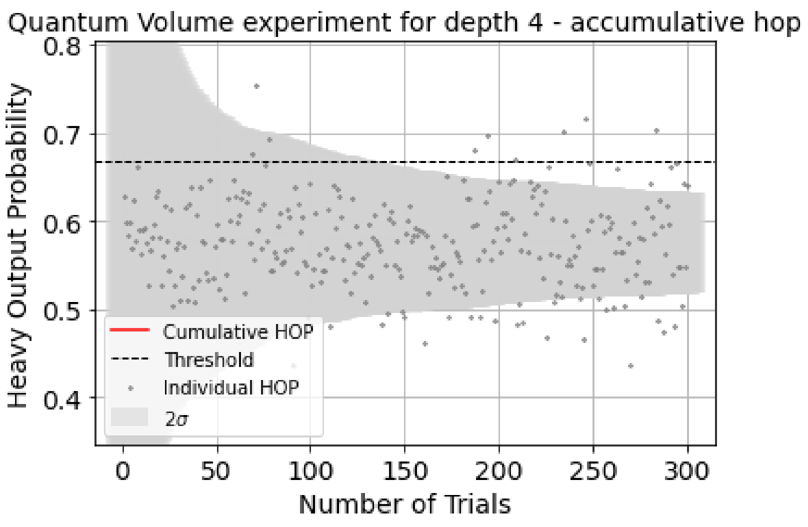
```
- success: False
- confidence: 0.030113362199722604
- depth: 4
- trials: 100
```

The parameters of a `QuantumVolume` experiment can be modified using the `QuantumVolume.set_experiment_options` method. For this example we will change the number of trials, run the experiment again and then add the result data that of the first experiment with `expdata.add_data()`. After merging the data, the result analysis can be redone with `QuantumVolume.analysis.run(data)`. That is why the `analysis` argument was set to `None` when running the experiment the second time.

[7]:
```python
# change number of trials
qv_exp.set_experiment_options(trials = 200)

# run experiment again
expdata2 = qv_exp.run(backend, analysis = None).block_for_results()

# combine data of the two experiments
expdata2.add_data(expdata.data())

# analyze the new joint data
qv_exp.analysis.run(expdata2).block_for_results()
```

[7]:
```
<ExperimentData[QuantumVolume], backend: ibmq_jakarta, status:
ExperimentStatus.DONE, experiment_id: d95d7f53-72bc-4973-ba72-78953517c854>
```

[8]:
```python
display(expdata2.figure(0))
for result in expdata2.analysis_results():
    print(result)
```



```
DbAnalysisResultV1
- name: mean_HOP
- value: 0.575+/-0.029
- quality: bad
- extra: <4 items>
- device_components: ['Q0', 'Q1', 'Q2', 'Q3']
- verified: False
DbAnalysisResultV1
- name: quantum_volume
- value: 1
- quality: bad
- extra: <4 items>
- device_components: ['Q0', 'Q1', 'Q2', 'Q3']
- verified: False
```

It is also possible to create and run several different `QuantumVolume` experiments using `BatchExperiment`. The syntax used to run these experiments is the same as with individual experiments.

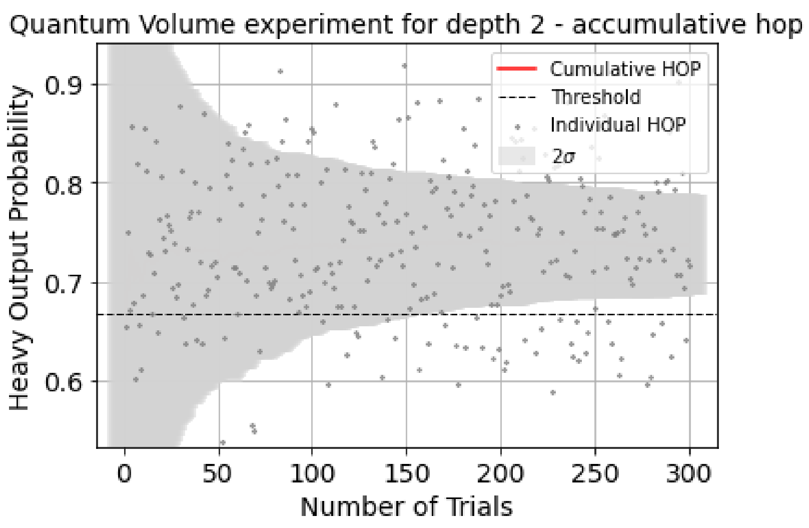
[9]:
```
from qiskit_experiments.framework import BatchExperiment

exps = [QuantumVolume(range(i), trials = 300) for i in range(2,5)]

batch_exp = BatchExperiment(exps)
```

```
batch_expdata = batch_exp.run(backend).block_for_results()
```

In order to obtain the results of the *i*-th experiment, if suffices to use `batch_expdata.child_data(i)`. The rest works in the exact same way as in the individual case.

```
[10]: for i in range(batch_exp.num_experiments):
          print(f"\nComponent experiment {i}")
          sub_data = batch_expdata.child_data(i)
          display(sub_data.figure(0))
          for result in sub_data.analysis_results():
              print(result)
```

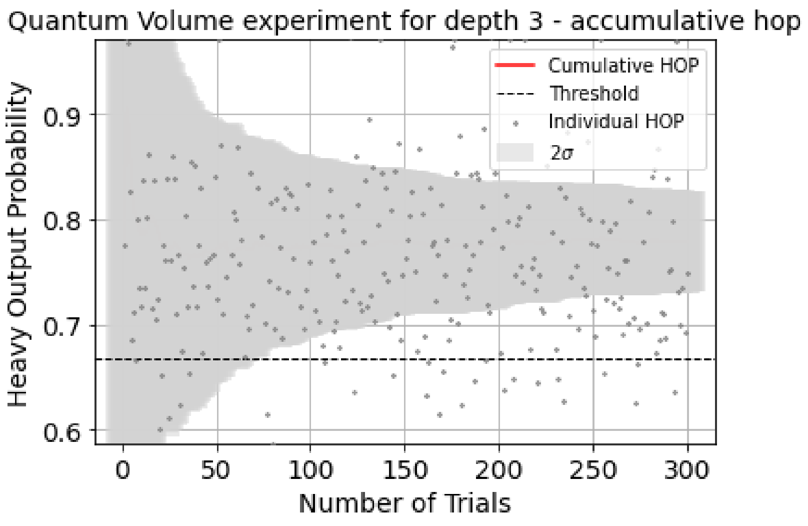
```
Component experiment 0
```



```
DbAnalysisResultV1
- name: mean_HOP
- value: 0.737+/-0.025
- quality: good
- extra: <4 items>
- device_components: ['Q0', 'Q1']
- verified: False
DbAnalysisResultV1
```

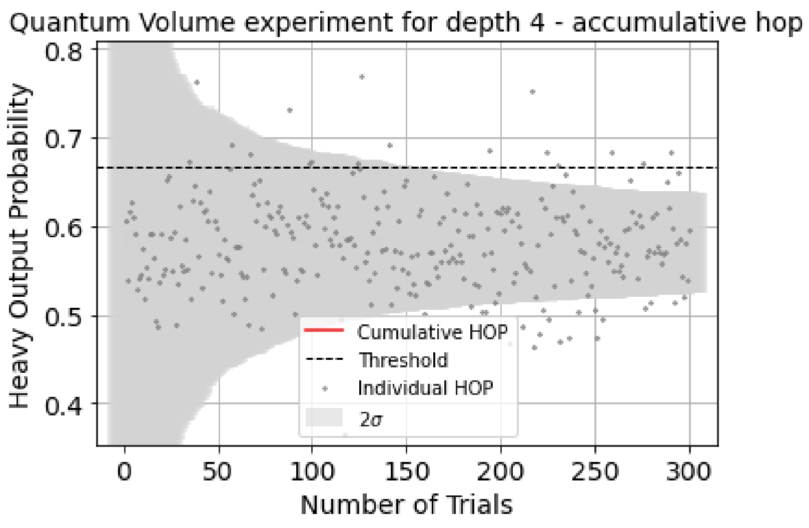
```
- name: quantum_volume

- value: 4

- quality: good

- extra: <4 items>

- device_components: ['Q0', 'Q1']

- verified: False
```

```
Component experiment 1
```



```
DbAnalysisResultV1

- name: mean_HOP

- value: 0.778+/-0.024

- quality: good

- extra: <4 items>

- device_components: ['Q0', 'Q1', 'Q2']

- verified: False

DbAnalysisResultV1

- name: quantum_volume

- value: 8

- quality: good

- extra: <4 items>

- device_components: ['Q0', 'Q1', 'Q2']

- verified: False
```

```
Component experiment 2
```



```
DbAnalysisResultV1
- name: mean_HOP
- value: 0.581+/-0.028
- quality: bad
- extra: <4 items>
- device_components: ['Q0', 'Q1', 'Q2', 'Q3']
- verified: False
DbAnalysisResultV1
- name: quantum_volume
- value: 1
- quality: bad
- extra: <4 items>
- device_components: ['Q0', 'Q1', 'Q2', 'Q3']
- verified: False
```

[11]:
```
qv_values = [batch_expdata.child_data(i).analysis_results("quantum_volume").
 ↪value for i in range(batch_exp.num_experiments)]


print(f"Max quantum volume is: {max(qv_values)}")
```

```
Max quantum volume is: 8
```

Chapter 4

Improve the Results Using a Transpiler

In the previous chapter we have only used the transpiler to insert the circuit into the desired device, without further optimizing the circuit. We have seen that this result is nowhere close to the Quantum Volume reported by IBM. In this section we will cover some algorithms that can improve the results we have obtained.

4.1 Noise adaptive layout

This algorithm (GreedyE* from Murali et al. (2019)) uses the properties of the device in order to get the best possible selection of qubits. In particular, it uses the error rates of the qubits and each CNOT gate.

It starts by creating a graph whose nodes are the indices of the physical qubits and whose edges are the available CNOTs of the device. These edges are given the weights

$$w_{ij} = -\ln\Big(\text{Reliability}_{SWAP_{ij}}\Big), \tag{4.1}$$

where

$$\text{Reliability}_{SWAP_{ij}} = (\text{Reliability}_{CNOT_{ij}})^3 = (1 - \text{error}_{CNOT_{ij}})^3 \tag{4.2}$$

Then, the Floyd-Warshall algorithm (explained in appendix B) is used to get the shortest path from each node to the other. That means that if the shortest path from nodes i and j

is the one that goes through nodes $\{k_l\}_{l=0}^m$, with $m < N$, then

$$M_{ij} = -\ln(w_{ik_0}) - \sum_{l=0}^{m-1} \ln\big(w_{k_l k_{l+1}}\big) - \ln(w_{k_m j}) = -\ln\Bigg(w_{ik_0}\big(\prod_{l=0}^{m-1} w_{k_l k_{l+1}}\big)w_{k_m j}\Bigg) \tag{4.3}$$

Given that the function e^{-x} is strictly decreasing, if we take

$$\begin{aligned} e^{-M_{ij}} &= w_{ik_0}\Bigg(\prod_{l=0}^{m-1} w_{k_l k_{l+1}}\Bigg)w_{k_m j} \\ &= \text{Reliability}_{SWAP_{ik_0}}\Bigg(\prod_{l=0}^{m-1} \text{Reliability}_{SWAP_{k_l k_{l+1}}}\Bigg)\text{Reliability}_{SWAP_{k_m j}} \end{aligned}$$

we are getting the maximum product of SWAP reliabilities, that is, the reliability corresponding to applying all these SWAPs.

In order to get the reliability corresponding to a CNOT from node i to j, if there is no direct CNOT that connects them, the algorithm looks into the different neighbours n of j and takes the one with the best reliability, that is the product of the reliability of the SWAP that exchanges i for n, that is $e^{-M_{in}}$, multiplied by the reliability of the CNOT that links n and j. In other words, it searches

$$\text{Reliability}_{CNOT_{ij}} = max_{n\in N(j)}(e^{-M_{in}} \cdot \text{Reliability}_{CNOT_{nj}}), \tag{4.4}$$

where $N(k)$ denotes the neighbours of node k.

In case there already exists a CNOT between nodes i and j, then we can directly pick the reliability from the device specifications.

The algorithm continues by creating a program graph P_{graph} whose nodes are the indices of the virtual qubits and whose edges correspond to the CNOTs of the circuit. These edges have an assigned weight that is the number of times this particular CNOT appears in the circuit. The edges of the program graph are then ordered by descending weight and in case of draw, by ascending edge vertex corresponding to the gate's control qubit and if there is another draw, by ascending target qubit vertex. Then, if there is an edge for which only one of the qubits is mapped to a physical one, this one is the next edge and if it's not the case, the next in the aforementioned ordered list is chosen.

If none of the nodes of the current program edge has been mapped to a physical qubit, the algorithm checks for the available CNOT in the device with the highest total reliability (lowest total error), given by the product of the reliabilities of the corresponding qubits and the CNOT itself,

$$\text{Reliability}_i \cdot \text{Reliability}_{CNOT_{ij}} \cdot \text{Reliability}_j \tag{4.5}$$

and maps the virtual qubits to these physical ones. It's important to note that for this we are only considering the CNOTs that are directly implemented on the device, so the reliabilities are not taken from 4.4 but only from the device specifications.

If only one of the qubits is already mapped, without loss of generality we can suppose that the first qubit is mapped to physical node i, then the algorithm looks for the best possible remaining physical qubit j, that is the one that gives the highest total reliability over all the already mapped neighbours of the program qubit q multiplied by the fidelity of the hardware qubit, that is

$$j = \arg\max_k \left(\prod_{\substack{n \in N(q) \\ n \text{ mapped}}} \text{Reliability}_{CNOT_{hw(n)k}} \right) \cdot \text{Reliability}_k, \tag{4.6}$$

where $hw(n)$ is the hardware qubit to which program qubit n was mapped.

4.2 SABRE

The SWAP-based BidiREctional (SABRE) heuristic algorithm (Li et al. 2018) can be used to both efficiently add any necessary SWAP gates to a circuit in order to run it on a device with limited connectivity but it can also be used to determine a mapping of program qubits into physical ones.

This algorithm starts by ordering all the 2-qubit gates into layers according to qubit dependencies, that is, each qubit can be used at most once on each layer and we intend to use them as soon as possible. We will also need an initial random mapping π of the qubits.

We then consider the first layer, that we will call F and check whether it is empty. If that is the case, the algorithm ends and if not, it looks for any gates that can be directly executed without any extra SWAPs and removes them from F. Then it looks for any possible successor gates and if there is no unresolved dependency, they are added to F. For the gates that require SWAPs, the next step is to look for every SWAP that affects at least one qubit from the gates in F, create a new temporal mapping π_{temp} according to each one of them and assign a score by evaluating a cost function H. The SWAP with the minimal score is then applied and we start again.

The cost function H should be able to:

- indicate the SWAP that can place the qubits in F closer to execute the 2-qubit gates.
- consider some more 2-qubit gates apart from those in F.
- control the parallel execution of SWAPs.

For this purpose, we consider a graph whose nodes are the physical qubits on the device and whose edges are the pairs with available CNOTs. We will then use the Floyd-Warshall algorithm (appendix B) to get the distance matrix D, whose element $D[i, j]$ is the number of needed SWAPs to go from node i to j. That means that the weights of all the edges are set to 1.

Our basic heuristic cost function will consist of the summation of the distances between the qubits in F, that is:

$$H_{basic} = \sum_{gate \in F} D[\pi(gate.q_1), \pi(gate.q_2)], \tag{4.7}$$

where $gate.q_i$ is the i-th program qubit of $gate$.

This cost function fulfills the first requirement stated above, but still needs to consider some gates outside F and parallelism. The former can be solved by selecting a small set of gates E outside of F that contains some close successors of the gates in F and extend the above summation over them. The size of E can be chosen according to the desired look-ahead ability. However, it is not necessary to pick too many gates for E, as these gates are only used to get a not too accurate estimation of the effect the SWAPs can have and if there are too many of them, the computation effort can increase dramatically. Because the sizes of F and E will be generally different, we will normalize the sums over F and E over their respective number of gates. We will nevertheless give some priority to the gates in F, as they are the ones that will be executed first, so we will add a weight parameter $W \in (0, 1)$ to the summation over E. Our next cost function will then be:

$$H_{lookahead} = \frac{1}{|F|} \sum_{gate \in F} D[\pi(gate.q_1), \pi(gate.q_2)] + \frac{W}{|E|} \sum_{gate \in E} D[\pi(gate.q_1), \pi(gate.q_2)], \tag{4.8}$$

In order to give preference to SWAPs that can be run in parallel, we introduce a decay term to the heuristic cost function. The idea behind this term is that if a qubit q_i was recently used in a SWAP, its decay parameter will increase by δ, that is, $decay(q_i) = 1 + \delta$. That way there will be some preference towards non-overlapping SWAPs. Similarly to the selection of $|E|$ and W for $H_{lookahead}$, we can choose δ according to how much priority we want to give to the parallelism. Our final cost function will then be:

$$H_{decay} = \max(decay(SWAP.q_1), decay(SWAP.q_2))\Bigg(\frac{1}{|F|}\sum_{gate\in F} D[\pi(gate.q_1), \pi(gate.q_2)] + \frac{W}{|E|}\sum_{gate\in E} D[\pi(gate.q_1), \pi(gate.q_2)]\Bigg), \quad (4.9)$$

By default, Qiskit's implementation of this algorithm sets $|E| = 20$, $W = 0.5$, $\delta = 0.001$ and the decay is reset after 5 SWAPs (*Source code for SabreSWAP, Qiskit* 2022).

The full description of this routing algorithm is then:

Algorithm 1 SABRE's SWAP-based Heuristic Search

```
    Input: Front layer $F$, Mapping $\pi$, Distance Matrix $D$, Quantum Circuit $qc$, Device Coupling Graph $G$
    Output: Inserted SWAPs, Final Mapping $\pi_f$
  while $F$ is not empty do
      Execute_gate_list = ∅
      for gate in $F$ do
          if gate can be executed on device then
              Execute_gate_list.append(gate)
          end if
      end for
      if Execute_gate_list ≠ ∅ then
          for gate in Execute_gate_list do
              F.remove(gate)
              Obtain successor gates
              if successor gates' dependencies are resolved then
                  F.append(gate)
              end if
          end for
          Continue
      else
          score = []
          SWAP_candidate_list = Obtain_SWAPs($F$,$G$)
          for SWAP in SWAP_candidate_list do
              $\pi_{temp} = \pi$.update(SWAP)
              score[SWAP] = H($F$, $qc$, $\pi_{temp}$, $D$, SWAP)
          end for
          Find the SWAP with minimal score;
          $\pi = \pi$.update(SWAP)
      end if
  end while
```

This circuit routing algorithm can also be applied several times to get an initial layout. This can be done by taking into account the reversibility of the quantum circuits and the fact that after the routing you get as output a final mapping. That way it is possible to use the final mapping obtained after running the routing algorithm once as an initial mapping for the reverse circuit, that is, the circuit with its gates in the reverse order. After going forward and backwards through the circuit several times, you will get an increasingly improved initial mapping. Qiskit's implementation of this layout algorithm does 3 forward-backward iterations (*Source code for SabreLayout, Qiskit* 2022).

4.3 Preset transpiler pass managers

The algorithms or transpiler passes from the previous sections are nevertheless only a small part of the transpilation process, that includes rewriting all gates in function of the native gates from the device, choose an initial qubit layout, map the circuit to the hardware and gate optimization. A full analysis of all the different transpiler passes that can be applied to a quantum circuit and how to schedule them is out of the scope of this thesis. Fortunately, Qiskit includes 4 different preset pass managers classified by optimization level (*Preset Passmanagers* 2022) that let us improve the performance of our quantum circuits without delving too deep into the details. As of Qiskit version 0.37 and Qiskit Terra version 0.21, each preset pass manager acts the following way:

- `optimization_level=0`: does the bare minimum to ensure that the circuit can be run on the chosen quantum backend. If an initial qubit layout is given, it is used and if no layout is given, the trivial layout consisting of assigning the i-th virtual qubit to the i-th physical qubit is used. Then the gates from the circuit are unrolled to the basis of native gates from the hardware and SWAPs are added to match the coupling map, that is, the available qubit connections. It is the default level when using the `transpile()` function in Qiskit.

- `optimization_level=1`: does light optimization. If an initial qubit layout is given, it is used and if no layout is given, then the trivial layout is used only if it is already compatible with the coupling map. If that is not the case, the circuit is mapped to the most densely connected subset of physical qubits. The circuit is then unrolled to the basis of native gates and SWAPs are added to match the coupling map if needed. Then adjacent gates are combined and any redundant reset in the circuit is removed. It is the default level when using `qiskit-experiments`.

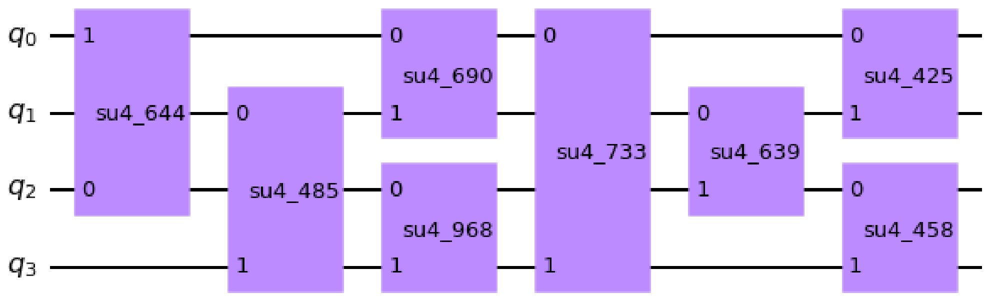
- `optimization_level=2`: does medium optimization. If an initial layout is given, it is used and if none is given, looks for a layout that makes the circuit satisfy the coupling map. If this is not possible, the most densely and reliably connected subset of physical qubits. SWAPs are then added if necessary to match the coupling map, the circuit is unrolled to the basis gates
- `optimization_level=3`: does heavy optimization. If an initial layout is given, it is applied. If it is not the case, it looks for a layout that makes the circuit satisfy the coupling map. If it is not possible, uses SABRE to get the best possible layout. Then, SWAPs are added to ensure that the circuit matches the coupling map if needed, unrolls to the basis gates, uses commutation rules to remove redundant gates, syntetizes 2-qubit blocks and redundant resets are removed.

4.4 Applying the algorithms with Qiskit

This section will cover how to apply the algorithms covered in this chapter using Qiskit. We will start by creating a 4-qubit `QuantumVolume` circuit, to which we will apply each algorithm.

[19]:
```
from qiskit.circuit.library import QuantumVolume


qv_circ = QuantumVolume(4)
qv_circ.decompose().draw("mpl")
```

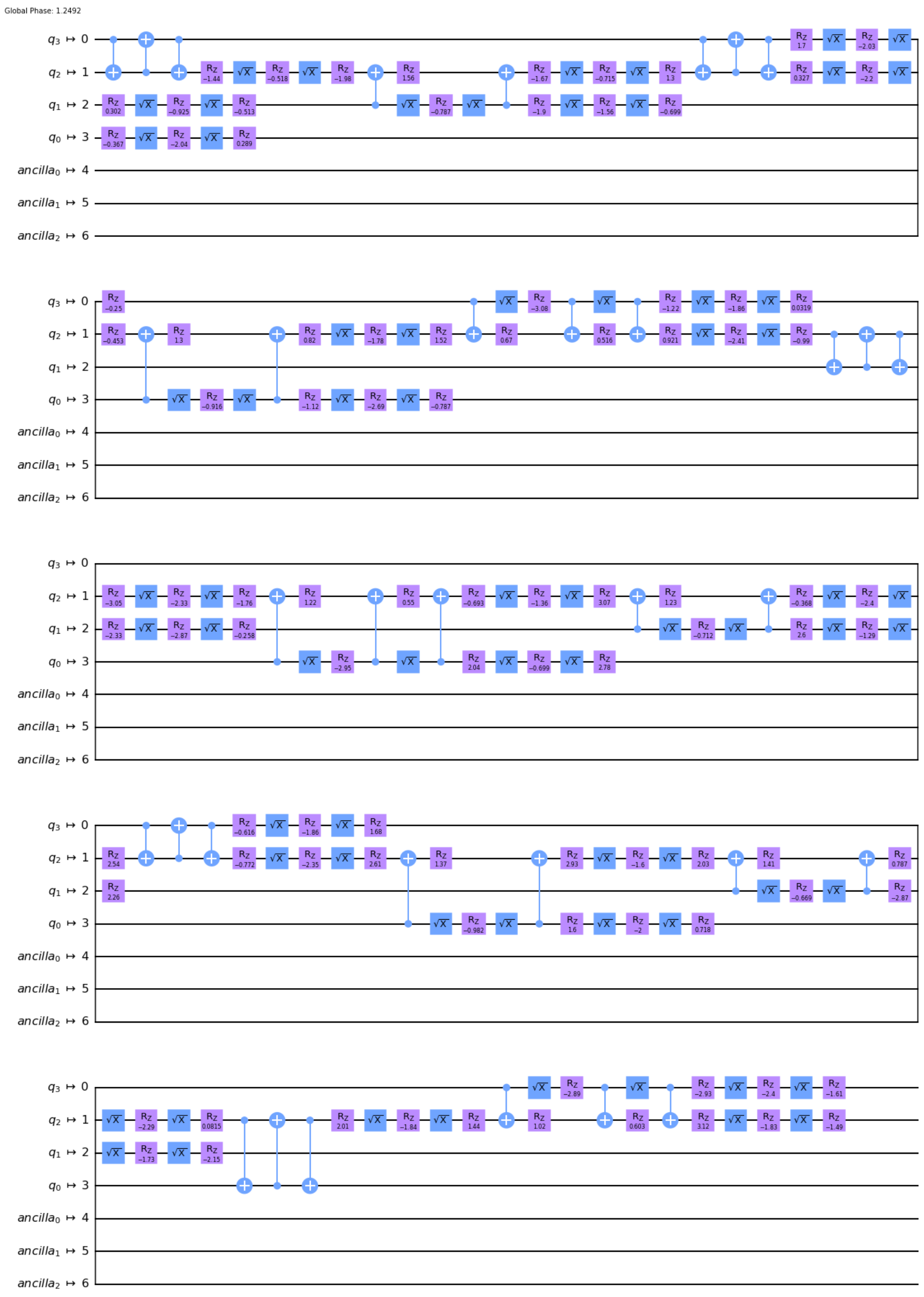
[19]:



We will compare SABRE and noise adaptive layout algorithms to the default options from the `transpile` function. For this comparison the quantum computer `ibm_perth` will be used. This device has 7 qubits, so we will be able to see the difference that each layout algorithm can make. We start transpiling the circuit to the backend with the default options.

[23]:
```python
from qiskit import IBMQ, transpile

# load IBMQ account
IBMQ.load_account()

# select provider
provider = IBMQ.get_provider(hub='ibm-q-internal')

# choose quantum computer
backend = provider.get_backend('ibm_perth')

# transpile the circuit with default layout method
transpiled_qv_circ = transpile(qv_circ, backend)
transpiled_qv_circ.draw("mpl")
```

[23]:

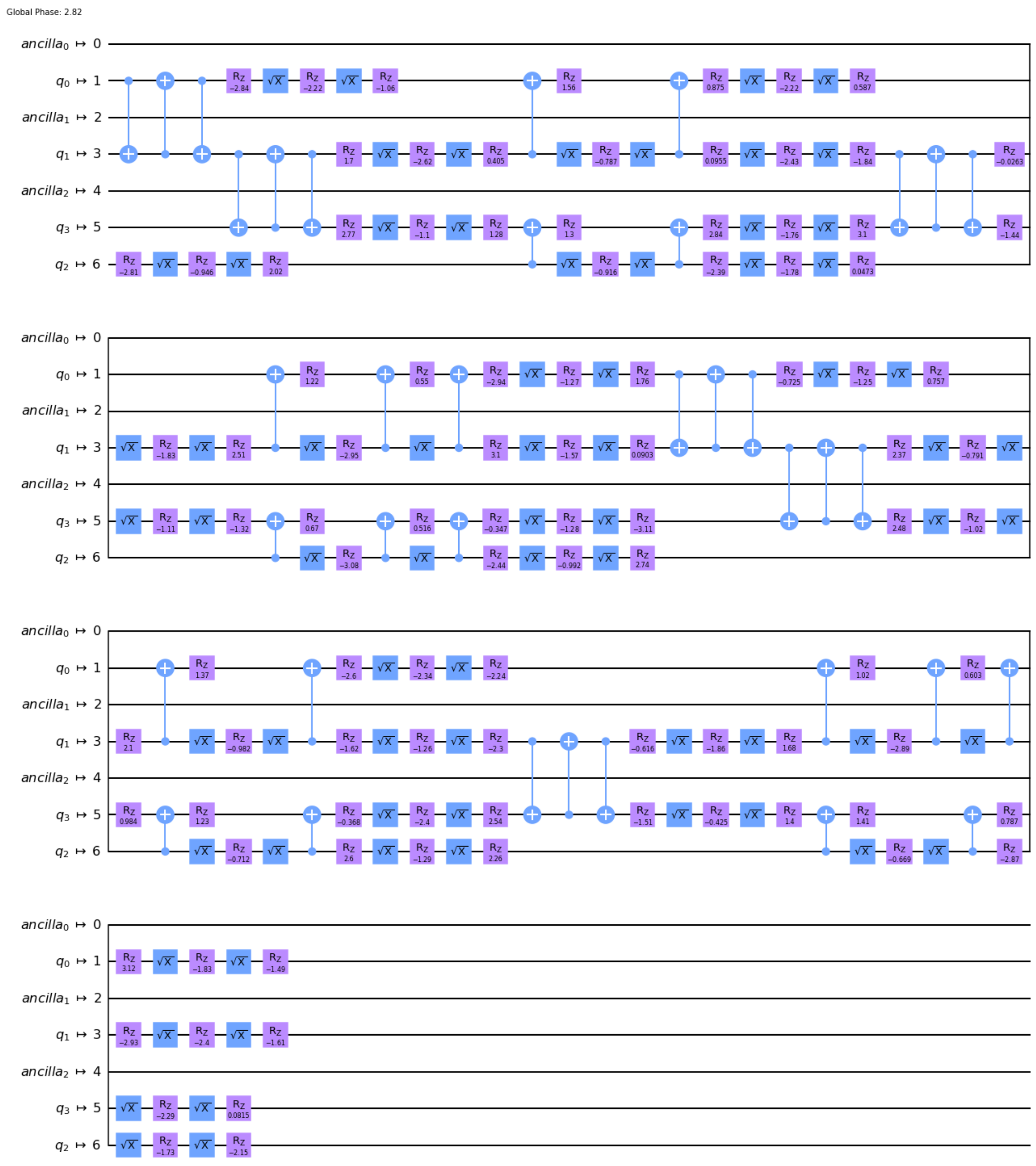


In order to apply SABRE to the circuit, it suffices to set the `layout_method` argument from `transpile` to `'sabre'`.

[24]:
```
# transpile with sabre
transpiled_qv_circ = transpile(qv_circ, backend, layout_method='sabre',␣
↪seed_transpiler=123)
```

```
transpiled_qv_circ.draw("mpl")
```

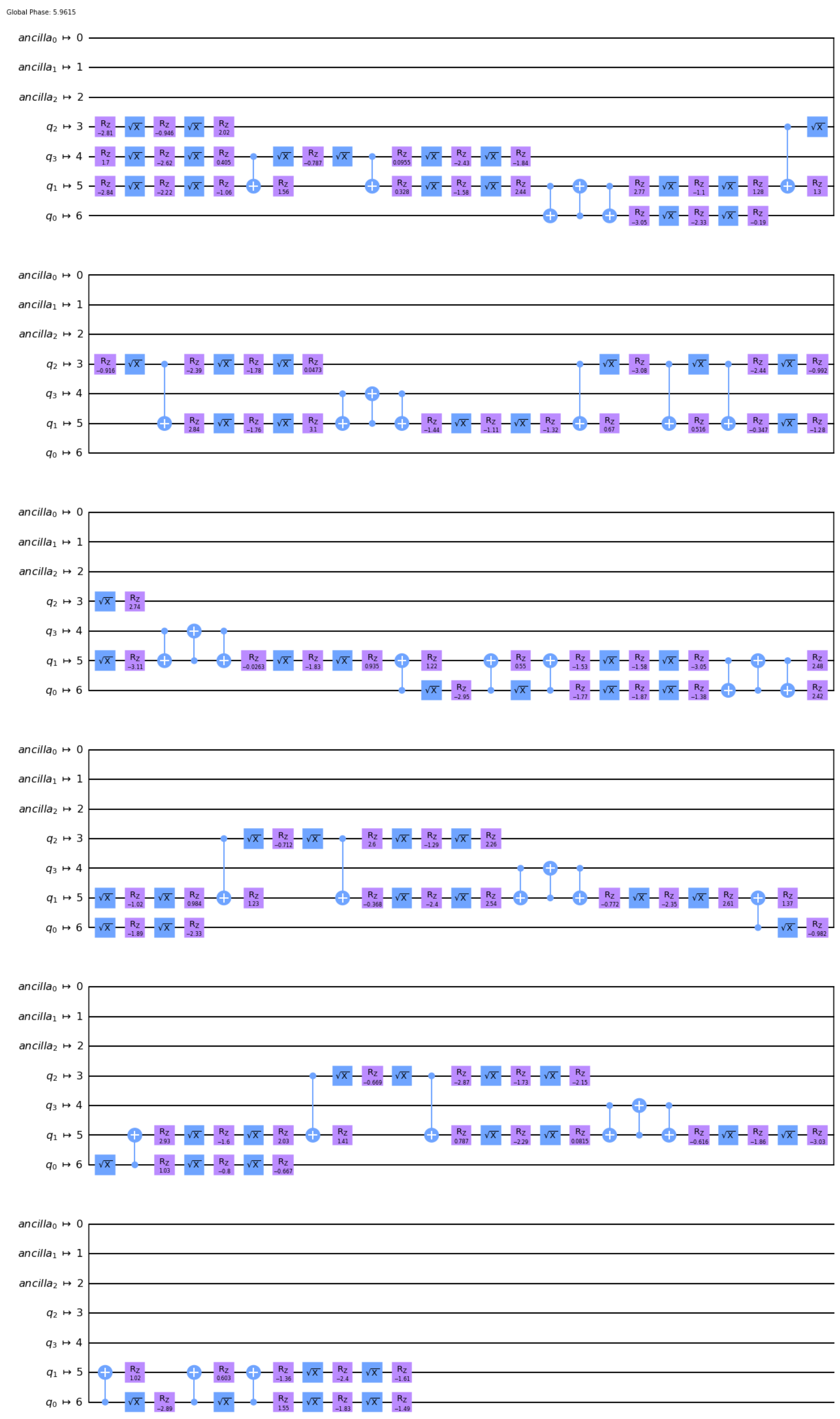
[24]:



Similarly, the noise adaptive layout algorithm can be applied setting `layout_method` to `'noise_adaptive'`.

[25]:
```
# transpile with noise adaptive layout
transpiled_qv_circ = transpile(qv_circ, backend,␣
↪layout_method='noise_adaptive', seed_transpiler=123)
transpiled_qv_circ.draw("mpl")
```

[25]:


Global Phase: 5.9615
$ancilla_0 \mapsto 0$
$ancilla_1 \mapsto 1$
$ancilla_2 \mapsto 2$
$q_2 \mapsto 3$
$q_3 \mapsto 4$
$q_1 \mapsto 5$
$q_0 \mapsto 6$

The different preset pass managers can be applied to the circuit setting the `optimization_level` algorithm to the desired level: 0, 1, 2 or 3. We can compare the number of each operation with `QuantumCircuit.count_ops()`.

[37]:
```
for i in range(4):
    tc = transpile(qv_circ, backend, optimization_level=i)
    print("optimization level", i, list(tc.count_ops().items()))
```

```
optimization level 0 [('rz', 115), ('sx', 80), ('cx', 34)]
optimization level 1 [('rz', 100), ('sx', 70), ('cx', 40)]
optimization level 2 [('rz', 91), ('sx', 64), ('cx', 34)]
optimization level 3 [('rz', 97), ('sx', 68), ('cx', 28)]
```

As we can see, the higher the optimization level, the lower the number of gates.

With `qiskit_experiments`, it is also possible to choose the optimization level. It can be done by using the `QuantumVolume.set_transpile_options` method, that works nearly identically to the `transpile` function for circuits, that is, it has the same arguments. However, as discussed in the previous chapter, given that you have to choose the qubits beforehand, the `layout_method` argument will be ignored, so it is not possible to add SABRE or noise adaptive layout algorithms.

[ ]:
```
from qiskit_experiments.library import QuantumVolume

# create experiment
qubits = range(4)
qv_exp = QuantumVolume(qubits, seed = 42)

qv_exp.set_transpile_options(optimization_level=3)
```

Chapter 5

Results

Using the processes described in chapter 3 with the improvements mentioned in chapter 4, we tried to measure the Quantum Volume of 7 different devices. 3 of these devices have a noticeably larger number (27) than the others (5-7). The devices studied with their number of qubits, reported Quantum Volume and the Quantum Volume obtained in this work are in table 5.1. A full disclosure of the experimental results can be found in appendix C.

5.1 Comparing smaller and bigger devices

5.1.1 Quantum Volume 8

Even though the Quantum Volumes obtained for all the studied quantum devices are similar, the behavior is a bit different for the larger devices. Generally speaking, with the smaller devices a Quantum Volume of 8 can be easily achieved even with 300 circuits `optimization_level=0` and without using SABRE or noise adaptive layout algorithms, achieving heavy output probabilities above 0.74 in all cases with a $2\sigma < 0.06$, which is enough to pass the test in all cases. However, the bigger ones don't always pass the test under these conditions, with `ibmq_mumbai` having a heavy output probability (HOP) slightly higher than the 2/3 threshold (0.672) but not enough for the 2σ to surpass it. However, `ibmq_montreal` does pass the Quantum Volume test but barely and still getting an HOP of 0.720, that is lower than that of the smaller devices. The results for each quantum computer can be seen in table 5.2. A particular case is that of `ibm_cairo`, for which the run with index 1 (taken from table C.6) gives an HOP and 2σ that enable it to pass the Quantum Volume test but the run with index 11 doesn't pass the test. This can be happening because more than a month passed

Device	qubits	reported Quantum Volume	obtained Quantum Volume
`ibm_cairo`	27	64	16
`ibmq_mumbai`	27	128	8
`ibmq_montreal`	27	128	16
`ibmq_jakarta`	7	16	16
`ibm_perth`	7	32	8
`ibm_lagos`	7	32	8
`ibmq_lima`	5	8	8

Table 5.1: Quantum devices studied with their number of qubits, their reported Quantum Volume and the Quantum Volume obtained in this work

Device	id	total qubits	n_c	n_s	HOP	2σ
`ibmq_lima`	3	5	200	4000	0.782	0.058
`ibmq_jakarta`	2	7	200	4000	0.781	0.058
`ibm_perth`	1	7	300	5000	0.768	0.048
`ibm_lagos`	1	7	300	5000	0.744	0.050
`ibm_cairo`	1	27	200	4000	0.774	0.059
`ibmq_cairo`	11	27	300	5000	0.672	0.054
`ibmq_mumbai`	1	27	300	5000	0.669	0.054
`ibmq_montreal`	1	27	300	5000	0.720	0.052

Table 5.2: Results of Quantum Volume 8 test with optimization level 0 and trivial layout for the different devices.

between the Qiskit Experiments executions and the rest, while device properties change over time and calibration is done weekly.

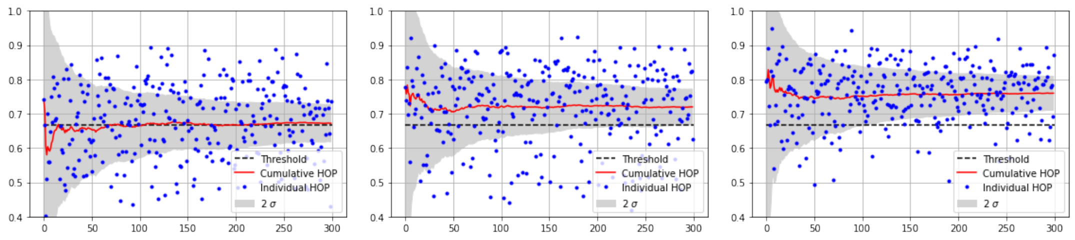


Figure 5.1: Results of Quantum Volume 8 experiment on `ibm_cairo` with `optimization_level=0` and trivial (left), SABRE (center) and noise adaptive (right) layout methods.

Device	id	total qubits	n_c	n_s	HOP	2σ
`ibmq_jakarta`	3	7	200	4000	0.561	0.070
`ibmq_jakarta`	9	7	1500	5000	0.587	0.025
`ibm_perth`	3	7	300	5000	0.561	0.057
`ibm_lagos`	5	7	300	5000	0.595	0.057
`ibm_cairo`	3	27	200	4000	0.629	0.068
`ibm_cairo`	14	27	300	5000	0.57	0.057
`ibmq_mumbai`	6	27	300	5000	0.548	0.057
`ibmq_montreal`	3	27	300	5000	0.621	0.056

Table 5.3: Results of Quantum Volume 16 test with optimization level 0 and trivial layout for the different devices.

id	qubits	n_c	n_s	n_h	HOP	2σ	layout
4	4	200	4000	538400	0.673	0.066	[0,1,2,3]
5	4	1000	4000	2248000	0.562	0.031	[2,1,4,7]
6	4	1000	4000	2796000	0.699	0.029	[13,14,16,19]

Table 5.4: Results of different choice of qubits for Quantum Volume 16 tests run on `ibm_cairo` with optimization level 3.

5.1.2 Quantum Volume 16

When running Quantum Volume 16 tests with optimization level 0, there is not a noticeable difference between small and big devices, as all of them are getting results that are relatively close to the worst possible result of HOP$= 0.5$. According to the data in table 5.3 `ibmq_mumbai` has an HOP of 0.548, that is below that of the small devices (between 0.561 and 0.595), while `ibmq_montreal` surpasses all of these devices in terms of HOP with a value of 0.621.

5.2 Layout

The behavior of the layout algorithms also differs between bigger and smaller devices, whose data can be seen in tables 5.5 and 5.6, respectively. For the smaller devices, both SABRE and noise adaptive algorithms tend to actually give a worse HOP than the default algorithm, with SABRE giving better results than noise adaptive in all these devices. For example, for `ibm_perth` and `ibm_lagos`, SABRE reduces HOP by 0.005 and 0.008 respectively while noise adaptive layout does it by 0.052 and 0.022 with 4 qubits and optimization level 3. However, if instead of 4 qubits we use 3 for `ibm_lagos`, we got that the decrease in HOP is less noticeable for noise adaptive (0.006) than for SABRE (0.014). Another exception is `ibmq_jakarta`, for which SABRE actually improves the HOP by 0.011 while noise adaptive lowers it by 0.026 for the Quantum Volume 16 test with optimization level 3.

However, this is not the case for the bigger devices, in which both layout algorithms give better results than the default, with noise adaptive having some advantage over SABRE. For example, for `ibmq_mumbai`, noise adaptive layout increases HOP by 0.074 while SABRE gives a much smaller improvement (0.014) for optimization level 3. However, for `ibmq_montreal` SABRE seems to actually reduce HOP by 0.010 while noise adaptive increases it by 0.060. A graphical comparison of the effect of trivial, SABRE and noise adaptive layout on `ibm_cairo` with optimization level 0 and 300 3-qubit circuits can be seen in figure 5.1.

An alternative to the aforementioned layout algorithms is manually selecting the qubits based on their properties. Table 5.4 shows the result of choosing qubits with lower CNOT error rate

device	id	qubits	n_c	n_s	HOP	2σ	optimization level	layout
ibmq_jakarta	9	4	1500	5000	0.587	0.025	0	
ibmq_jakarta	10	4	300	5000	0.588	0.057	0	noise adaptive
ibmq_jakarta	11	4	300	2000	0.654	0.055	0	SABRE
ibmq_jakarta	12	4	2400	2000	0.676	0.019	3	
ibmq_jakarta	13	4	900	2000	0.650	0.032	3	noise adaptive
ibmq_jakarta	14	4	3600	2000	0.687	0.015	3	SABRE
ibm_perth	4	4	300	5000	0.684	0.054	3	
ibm_perth	5	4	300	5000	0.632	0.056	3	noise adaptive
ibm_perth	6	4	300	5000	0.679	0.054	3	SABRE
ibm_lagos	2	3	300	5000	0.758	0.049	3	
ibm_lagos	3	3	300	5000	0.752	0.050	3	noise adaptive
ibm_lagos	4	3	300	5000	0.744	0.050	3	SABRE
ibm_lagos	6	4	300	5000	0.654	0.054	3	
ibm_lagos	7	4	300	5000	0.646	0.055	3	SABRE
ibm_lagos	8	4	300	5000	0.632	0.056	3	noise adaptive

Table 5.5: Behavior of the layout algorithms on smaller devices

device	id	qubits	n_c	n_s	HOP	2σ	optimization level	layout
ibmq_mumbai	1	3	300	5000	0.669	0.054	0	
ibmq_mumbai	2	3	300	5000	0.761	0.049	0	noise adaptive
ibmq_mumbai	3	3	300	5000	0.699	0.052	0	SABRE
ibmq_mumbai	4	3	2400	10000	0.691	0.019	0	SABRE
ibmq_mumbai	7	4	300	5000	0.609	0.056	3	
ibmq_mumbai	8	4	300	5000	0.683	0.054	3	noise adaptive
ibmq_mumbai	9	4	300	5000	0.623	0.056	3	SABRE
ibmq_montreal	3	4	300	5000	0.621	0.056	0	
ibmq_montreal	4	4	300	5000	0.613	0.056	0	SABRE
ibmq_montreal	5	4	300	5000	0.705	0.053	0	noise adaptive
ibmq_montreal	6	4	1200	5000	0.705	0.026	0	noise adaptive
ibmq_montreal	7	4	300	5000	0.654	0.055	3	
ibmq_montreal	8	4	300	5000	0.714	0.052	3	noise adaptive
ibmq_montreal	10	4	300	5000	0.645	0.055	3	SABRE
ibm_cairo	11	3	300	5000	0.672	0.054	0	
ibm_cairo	12	3	300	5000	0.76	0.049	0	noise adaptive
ibm_cairo	13	3	300	10000	0.72	0.052	0	SABRE

Table 5.6: Behavior of the layout algorithms on bigger devices

id	qubits	n_c	n_s	n_h	HOP	2σ
3	3	300	5000	1048728	0.699	0.052
4	3	2400	10000	16577121	0.691	0.019

Table 5.7: Effect of changing the number of circuits n_c on `ibmq_mumbai` using SABRE and optimization level 0

id	qubits	n_c	n_s	n_h	HOP	2σ	optimization level
5	4	300	5000	1057260	0.705	0.053	0
6	4	1200	5000	4229444	0.705	0.026	0
8	4	300	5000	1070713	0.714	0.052	3
9	4	900	5000	3192246	0.709	0.030	3

Table 5.8: Effect of changing the number of circuits n_c on `ibmq_montreal` using noise adaptive layout

([13,14,16,19]) than the first qubits ([0,1,2,3]) or qubits with a higher error rate ([2,1,4,7]). In the first case we get an HOP increase of 0.026 that enables `ibm_cairo` to achieve Quantum Volume 16.

5.3 Optimization level

The impact of changing the optimization level from 0 to 3 depends heavily on the number of qubits. For 3-qubit circuits (Quantum Volume 8), as the smaller devices tend to have a higher HOP, the increase is much less noticeable when setting the optimization level to 3, getting only 0.016 and 0.014 for `ibm_perth` and `ibm_lagos` respectively. For the bigger devices, however, the increase of the HOP is much more significant, with it being 0.097 for `ibmq_mumbai` and 0.048 for `ibmq_montreal`. For 4-qubit circuits (Quantum Volume 16) the results are very different. In this case the differences seem to change from device to device, with, for example, `ibmq_mumbai` and `ibm_lagos` getting a HOP increase close to 0.060, while `ibm_cairo` and `ibmq_jakarta` are closer to 0.080-0.090, `ibmq_montreal` is close to 0.030 and `ibm_perth` reaches 0.120.

5.4 Number of circuits

Tables 5.7 and 5.8 show how the number of circuits can affect the results of the Quantum Volume test for `ibmq_mumbai` and `ibmq_montreal` respectively. In both cases, the HOP difference is small enought to be explained by the intrinsic randomness of the Quantum Volume experiment, while the 2σ is notably reduced. Figure 5.2 shows that the HOP becomes practically constant after around 50 circuits.

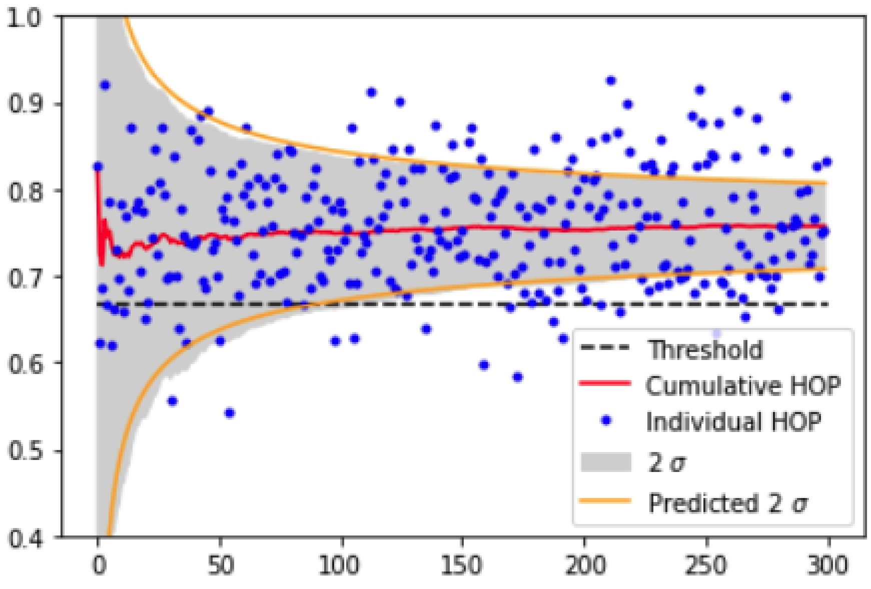


Figure 5.2: Results of Quantum Volume 8 experiment on `ibmq_montreal` with optimization level 0, including a theoretical prediction of 2σ from equation 5.1

This behavior of the 2σ can be predicted mathematically if we consider HOP to be constant. Taking into account that

$$2\sigma = \frac{2\sqrt{n_h(n_s - \frac{n_h}{n_s})}}{n_c n_s} = 2\sqrt{\frac{\frac{n_h}{n_s n_c}(1 - \frac{n_h}{n_s n_c})}{n_c}}$$

and

$$\text{HOP} = \frac{n_h}{n_s n_c}$$

we get that

$$2\sigma = 2\sqrt{\frac{\text{HOP}(1 - \text{HOP})}{n_c}} \tag{5.1}$$

Figure 5.2 shows a prediction of the 2σ taken from equation 5.1 over the experimental data. We can see that the prediction is very close to the real data.

If we try to predict, for example, the 2σ from the second row of 5.7 using the first one we would get that $2\sigma = 2\sqrt{\frac{0.699(1-0.699)}{2400}} = 0.019$, that is exactly the real value.

That implies that, if we run more than 50 circuits, we can predict the number of circuits needed to pass the threshold, as long as the HOP surpasses it

$$\text{HOP} - 2\sigma = \text{HOP} - 2\sqrt{\frac{\text{HOP}(1-\text{HOP})}{n_c}} = 2/3 \rightarrow 4\frac{\text{HOP}(1-\text{HOP})}{n_c} = (2/3 - \text{HOP})^2$$
$$\rightarrow n_c = \frac{4\text{HOP}(1-\text{HOP})}{(2/3-\text{HOP})^2}$$

So if we take, the fourth entry of C.3, in which we have HOP $= 0.684$ and a $2\sigma = 0.054$ so HOP $- 2\sigma = 0.63 < 2/3$ we can obtain the circuits we need for `ibm_perth` to pass the Quantum Volume 16 as

$$n_c = \frac{4 \cdot 0.684(1-0.684)}{(2/3-0.684)^2} = 2878.$$

5.5 Reaching the reported Quantum Volume

The results from table 5.1 show that only preset transpiler passmanagers and layout methods are not generally enough to come even close to reaching the reported Quantum Volume of IBM's quantum devices even though they generally improve the HOP. There are nevertheless some exceptions like `ibmq_jakarta` and `ibmq_lima` for which we could reach their true Quantum Volume with only these tools. In general, the experimental Quantum Volumes we obtained are much closer to the reported ones for the smaller devices than for the bigger ones. This can be happening because a higher number of qubits implies more variables that affect the Quantum Volume like crosstalk, for which more advanced techniques that are out of the scope of this thesis can become critical. Some examples of these are pulse-efficient SU(4) gate decomposition, idle qubit noise mitigation via dynamical decoupling and readout improvement with excited state promotion, that were used in Jurcevic et al. (2020) to reach Quantum Volume 64 with `ibmq_montreal`. It is also possible that the smaller devices were not studied in as much depth as bigger ones, as they have a clear limit on the Quantum Volume they can achieve, so they cannot be used to show any breakthrough about the limits of Quantum Volume.

Chapter 6

Conclusions

In this work we have discussed how the many applications and hardware implementations of quantum computing have given birth to several different benchmarks to measure the performance of the different quantum systems. We have gone into the classification of benchmarks according to quality, speed and complexity, focusing on holistic bechmarks like IBM's Quantum Volume, that take into account the quantum device as a whole and is focused on quality. Then, we moved our focus to explaining the Quantum Volume protocol, first in theory and then in practice using Qiskit and IBM's quantum devices. The idea behind the protocol was to simulate the circuit, get the outputs with higher probability than the expected value (heavy outputs), run the circuit on a real quantum device and check whether the heavy output probability (HOP) was higher than 2/3 with confidence 0.975 (2σ interval).

After that, we jumped to explaining SABRE, noise adaptive layout algorithm and the preset pass managers to improve the Quantum Volume results and how to implement them with Qiskit's circuit-to-circuit transpiler.

Then, we compared the Quantum Volume 8 and 16 test results for several 27 and 7-qubit devices. We observed that for Quantum Volume 8, the smaller devices tended to give a higher heavy output probability (HOP), passing the corresponding test while the bigger devices tended to fail. However for Quantum Volume 16 there was not a significant difference between bigger and smaller devices, most of which gave an HOP close to the worst-case scenario of 0.5.

Analyzing the effect of the SABRE and noise adaptive algorithms, we obtained that for smaller devices, both algorithms actually worsened the results, with SABRE being generally better than noise adaptive layout. However, for bigger devices they both gave an improvement in HOP and in this case the best algorithm was noise adaptive. We also discussed the possibility of handpicking qubits based on their CNOT error rate and could observe that it could make the difference between achieving a specific Quantum Volume and not.

Our next step was comparing the effect of the different preset optimizers included in Qiskit, sorted by optimization level. We found out that changing the optimization level from 0 to 3 had a very different impact depending on the number of qubits from the circuits. When running Quantum Volume 8 tests, the smaller devices tended to barely benefit from this change while the bigger ones had significant changes, with improvements of HOP around 0.015 for the smaller ones and above 0.045 for the bigger ones. This can be explained by the higher HOP the smaller devices had for this test without optimizing, so there was less room for improvement. For Quantum Volume 16 tests, the results seemed to vary from device to device, without any particular trend for bigger or smaller ones.

After that, we analyzed how the number of circuits affected the Quantum Volume test outcome. Generally speaking, the HOP tended to only vary significantly until around circuits had been run, after which it tended to remain stable. However, the width of the 2σ HOP confidence interval was reduced by a factor $1/\sqrt{n_c}$, where n_c is the number of circuits.

In general we obtained Quantum Volumes of 8 or 16 for all the studied quantum devices. However, those values were significantly closer to the reported ones for the smaller ones, for two of which even the reported Quantum Volume was achieved. It is important to note that in this work only 2 layout algorithms (SABRE and noise adaptive) and the preset transpiler pass managers were used while there are many other advanced techniques, like those from (Jurcevic et al. 2020), that were out of the scope of this thesis. Given that more qubits imply more variables that can affect the Quantum Volume, it is reasonable to expect this unused techniques to become critical to get closer to the reported Quantum Volume of the bigger devices while for the smaller ones it is possible that they do not make such an impact. Another possibility is that the smaller devices were not studied in so much detail as the bigger ones.

Some future investigations could try to determine how Quantum Volume experimental results can change over time, given that calibration data changes in a weekly basis. Another possible line of investigation could be trying to apply some of the most advanced optimization techniques to the smaller devices to see if the reported Quantum Volume can be surpassed.

Appendix A

Statistics Overview

This chapter will give an overview of the statistical concepts and results needed to completely understand the Quantum Volume protocol. It is based on (Bertsekas & Tsitsiklis 2002) and (Hogg et al. 1977).

A.1 Bernoulli distribution

If we consider a random experiments with only two possible outcomes, success and failure, where the probability of success is $p \in (0, 1)$ and then we define the random variable X that takes the value 1 if the experiment was successful and 0 otherwise, this random variable X will follow a Bernoulli distribution with parameter p, i.e., $X \equiv \mathcal{B}(p)$.

The probability function of X is then given by

$$P(X = k) = \begin{cases} p & \text{if } k = 1 \\ 1 - p & \text{if } k = 0 \end{cases} \tag{A.1}$$

And its expected value and variance are:

$$E[X] = p \tag{A.2}$$

$$Var[X] = p(1 - p) \tag{A.3}$$

A.2 Binomial distribution

If instead of only one Bernoulli experiment, we consider n successive identical and independent experiments and we denote by Y the random variable that indicates the number of successful experiments, then Y follows a binomial distribution with parameters n and p, that is $Y \equiv \mathcal{B}(n, p)$.

Its probability function is

$$P(Y = k) = \binom{n}{k} p^k (1-p)^{n-k}, \tag{A.4}$$

and its expected value and variance are given by

$$E[Y] = np \tag{A.5}$$

$$Var[Y] = np(1-p) \tag{A.6}$$

A very important property of the binomial distribution with is that it can be expressed as a sum of Bernoullis. In particular, if $Y \equiv \mathcal{B}(n, p)$, then

$$Y = \sum_{i=1}^{n} X_i, \qquad \text{where } X_i \equiv \mathcal{B}(p)\ \forall i. \tag{A.7}$$

A.3 Normal distribution

One of the most well-known continuous probability distributions is the normal distribution. The density function of a random variable Z that follows a normal with parameters μ and σ, $Z \equiv \mathcal{N}(\mu, \sigma)$, is given by

$$f(z) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{z-\mu}{\sigma})^2}, \tag{A.8}$$

and its expected value and variance are

$$E[Z] = \mu \tag{A.9}$$

$$Var[Z] = \sigma^2. \tag{A.10}$$

An interesting property the normal distribution has is that, if $Z \equiv \mathcal{N}(\mu, \sigma)$, then

$$\frac{Z-\mu}{\sigma} \equiv \mathcal{N}(0,1) \tag{A.11}$$

This can be straightforwardly proven with the change of variable $u = \frac{z-\mu}{\sigma} \rightarrow dz = \sigma du$:

$$1 = \int_{-\infty}^{\infty} f(z)dz = \int_{-\infty}^{\infty} \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{z-\mu}{\sigma})^2} dz = \int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}u^2} = \int_{-\infty}^{\infty} f_{\mathcal{N}(0,1)}(u)du$$

A.4 Central Limit Theorem

One of the reasons the normal distribution is so ubiquitous in statistics is because of the Central Limit Theorem. It states that, given the independent and identically distributed random variables $\{X_i\}_{i=1}^n$ with $E[X_i] = \mu$ and $Var[X_i] = \sigma^2$ then

$$\frac{\overline{X}_n - \mu}{\sigma/\sqrt{n}} \xrightarrow[n\rightarrow\infty]{} \mathcal{N}(0,1), \tag{A.12}$$

where $\overline{X}_n = \frac{1}{n}\sum_{i=1}^n X_i$ In most practical cases $n \geq 30$ in considered big enough.

In particular, if $X_i \equiv \mathcal{B}(p)$, then $\mu = p$ and $\sigma = \sqrt{p(1-p)}$ and the Central Limit Theorem reads

$$\frac{\overline{X}_n - p}{\sqrt{p(1-p)}/\sqrt{n}} = \frac{n\overline{X}_n - np}{\sqrt{p(1-p)}\sqrt{n}} = \frac{\sum_{i=1}^n X_i - np}{\sqrt{np(1-p)}} \rightarrow \mathcal{N}(0,1) \tag{A.13}$$

Then if we consider the random variable $Y = \sum_{i=1}^n X_i$, by A.7, we know $Y \equiv \mathcal{B}(n,p)$ then, by A.2 and A.3, we have

$$\frac{Y - np}{\sqrt{np(1-p)}} \rightarrow \mathcal{N}(0,1) \tag{A.14}$$

If we take into account A.5 and A.6 then we have that

$$\frac{Y - E[Y]}{\sqrt{Var[Y]}} \rightarrow \mathcal{N}(0,1),$$

which is analogous to A.11. That is why it is often considered that with a big enough n, a binomial $\mathcal{B}(n,p)$ behaves like a $\mathcal{N}(np, \sqrt{np(1-p)})$.

A.5 Confidence interval

Given the binomial $Y \equiv \mathcal{B}(n,p)$, the objective of this section is to and find a one-sided confidence interval for p with confidence $1-\alpha$ to obtain a tight lower bound of this parameter.

By applying A.14, we have

$$1-\alpha = P\left(\frac{Y-np}{\sqrt{np(1-p)}} < z_\alpha\right), \tag{A.15}$$

where z_α is such that $P(\mathcal{N}(0,1) < z_\alpha) = 1-\alpha$. Finding p from this expression can be too difficult unless we introduce some simplifications. In this case, we will use the estimation $\hat{p}$ of p for the terms in the square root, so we have:

$$\begin{aligned} 1-\alpha &= P\left(\frac{Y-np}{\sqrt{n\hat{p}(1-\hat{p})}} < z_\alpha\right) = P(Y-np < z_\alpha\sqrt{n\hat{p}(1-\hat{p})}) \\ &= P(Y - z_\alpha\sqrt{n\hat{p}(1-\hat{p})} < np) = P\left(\frac{Y-z_\alpha\sqrt{n\hat{p}(1-\hat{p})}}{n} < p\right) \end{aligned} \tag{A.16}$$

Then, for our particular case, $p = h$, $Y = \frac{n_h}{n_s}$, $n = n_c$ and $\hat{p} = \frac{n_h}{n_s n_c}$ so A.16 becomes

$$\begin{aligned} 1-\alpha = P\left(\frac{\frac{n_h}{n_s} - z_\alpha\sqrt{n_c\frac{n_h}{n_s n_c}(1-\frac{n_h}{n_s n_c})}}{n_c} < h\right) &= P\left(\frac{n_h - z_\alpha\sqrt{n_s^2\frac{n_h}{n_s}(1-\frac{n_h}{n_s n_c})}}{n_c n_s} < h\right) \\ &= P\left(\frac{n_h - z_\alpha\sqrt{n_h(n_s-\frac{n_h}{n_c})}}{n_c n_s} < h\right) \end{aligned} \tag{A.17}$$

Then, if we want a confidence of 97.5%, $\alpha = 0.25$ and $z_{0.25} = 1.96 \approx 2$ so

$$0.975 \approx P\left(\frac{n_h - 2\sqrt{n_h(n_s-\frac{n_h}{n_c})}}{n_c n_s} < h\right) \tag{A.18}$$

So, if $\frac{2}{3} < \frac{n_h - 2\sqrt{n_h(n_s-\frac{n_h}{n_c})}}{n_c n_s}$, then using A.18, we conclude that $\frac{2}{3} < h$ with probability 0.975.

Appendix B

Floyd-Warshall Algorithm

B.1 Graphs

A graph is a pair $G = (V, E)$ where V and E are the sets of vertices and edges, respectively. We will label the vertices as integers from 0 to $N - 1$, so N is the number of vertices, and for the edge that links vertex i to vertex j we will use the set $\{i, j\}$.

B.1.1 Directed graph

This algorithm is used for a specific type of graphs that are called directed graphs. That means that the pairs $(i, j) \in E$ are ordered. In other words, if $(i, j) \in E$ it is possible that $(j, i) \notin E$, as they are not considered to be the same.

Any undirected graph can be seen as a directed graph in which $(i, j) \in E \iff (j, i) \in E$.

B.1.2 Weighted graph

A weighted graph is a graph whose edges have numerical value or weight assigned. That means that in this case we can express an edge as (i, j, w), where i is the starting node, j the ending node and w its weight.

B.2 Description of the algorithm

Given a directed weighted graph (V, E) with no negative cycles, the Floyd-Warshall algorithm (Floyd 1962) outputs a matrix M whose element (i, j) is the shortest path from node i to node j.

In order to do so we first initialize the square matrix M so that

$$M_{ij} = \begin{cases} 0 & \text{if } i = j \\ w & \text{if } \exists (i,j,w) \in E \\ \infty & \text{rest of cases} \end{cases} \tag{B.1}$$

Then for each pair of vertices (i, j) it recursively checks whether there is any vertex $x \in S_k := \{0, ..., k\}$ for each $k \in \{0, ..., N-1\}$ for which the path $i \to x \to j$ is shorter than $i \to j$. If this condition is met, then the value M_{ij} is updated to match $M_{ix} + M_{xj}$.

In Python this can be written as:

```
def floydwarshall(M):
    N = M.shape[0]
    for i in range(N):
        for j in range(N):
            for k in range(N):
                if M[i,j] > M[i,k] + M[k,j]:
                    M[i,j] = M[i,k] + M[k,j]
    return M
```

Appendix C

Experimental Results

Tables C.1, C.2, C.3, C.4, C.5, C.6 and C.7 show a more detailed rundown of the results obtained for each quantum computer. Number of qubits (qubits), circuits (n_c), heavy outputs (n_h), heavy output probability (HOP), 2σ, optimization level and layout are specified. When no layout is given it means the default value for that optimization level was used and if a list of qubits is given, it means Qiskit Experiments was used.

id	qubits	n_c	n_s	n_h	HOP	2σ	optimization level	layout
1	2	200	4000	584000	0.730	0.063	0	[0,1]
2	3	200	4000	624800	0.781	0.058	0	[0,1,2]
3	4	200	4000	448800	0.561	0.070	0	[0,1,2,3]
4	2	200	4000	600000	0.750	0.061	3	[0,1]
5	3	200	4000	622400	0.778	0.059	3	[0,1,2]
6	4	200	4000	499200	0.624	0.069	3	[0,1,2,3]
7	3	2000	10000	14846201	0.742	0.020	1	
8	3	1500	5000	5544901	0.739	0.023	3	
9	4	1500	5000	4399880	0.587	0.025	0	
10	4	300	5000	882433	0.588	0.057	0	noise adaptive
11	4	300	2000	392228	0.654	0.055	0	SABRE
12	4	2400	2000	3245666	0.676	0.019	3	
13	4	900	2000	1170232	0.650	0.032	3	noise adaptive
14	4	3600	2000	1170232	0.687	0.015	3	SABRE
15	4	3600	2000	4846366	0.673	0.016	3	

Table C.1: Results of experiments on `ibmq_jakarta`

id	qubits	n_c	n_s	n_h	HOP	2σ	optimization level	layout
1	3	300	5000	1003913	0.669	0.054	0	
2	3	300	5000	1142031	0.761	0.049	0	noise adaptive
3	3	300	5000	1048728	0.699	0.052	0	SABRE
4	3	2400	10000	16577121	0.691	0.019	0	SABRE
5	3	300	5000	1148896	0.766	0.049	3	
6	4	300	5000	821270	0.548	0.057	0	
7	4	300	5000	914065	0.609	0.056	3	
8	4	300	5000	1024818	0.683	0.054	3	noise adaptive
9	4	300	5000	935864	0.623	0.056	3	SABRE

Table C.2: Results of experiments on `ibmq_mumbai`

id	qubits	n_c	n_s	n_h	HOP	2σ	optimization level	layout
1	3	300	5000	1152212	0.768	0.048	0	
2	3	300	5000	1176044	0.784	0.048	3	
3	4	300	5000	840820	0.561	0.057	0	
4	4	300	5000	1026202	0.684	0.054	3	
5	4	300	5000	948192	0.632	0.056	3	noise adaptive
6	4	300	5000	1018638	0.679	0.054	3	SABRE
7	5	300	5000	928167	0.619	0.056	3	

Table C.3: Results of experiments on `ibm_perth`

id	qubits	n_c	n_s	n_h	HOP	2σ	optimization level	layout
1	3	300	5000	1116578	0.744	0.050	0	
2	3	300	5000	1137768	0.758	0.049	3	
3	3	300	5000	1128567	0.752	0.050	3	noise adaptive
4	3	300	5000	1115980	0.744	0.050	3	SABRE
5	4	300	5000	892467	0.595	0.057	0	
6	4	300	5000	981012	0.654	0.054	3	
7	4	300	5000	968659	0.646	0.055	3	SABRE
8	4	300	5000	948192	0.632	0.056	3	noise adaptive

Table C.4: Results of experiments on `ibm_lagos`

id	qubits	n_c	n_s	n_h	HOP	2σ	optimization level	layout
1	3	300	5000	1080261	0.720	0.052	0	
2	3	300	5000	1151278	0.768	0.049	3	
3	4	300	5000	931147	0.621	0.056	0	
4	4	300	5000	919402	0.613	0.056	0	SABRE
5	4	300	5000	1057260	0.705	0.053	0	noise adaptive
6	4	1200	5000	4229444	0.705	0.026	0	noise adaptive
7	4	300	5000	981390	0.654	0.055	3	
8	4	300	5000	1070713	0.714	0.052	3	noise adaptive
9	4	900	5000	3192246	0.709	0.030	3	noise adaptive
10	4	300	5000	967170	0.645	0.055	3	SABRE
11	5	300	5000	929889	0.620	0.056	3	noise adaptive

Table C.5: Results of experiments on `ibmq_montreal`

id	qubits	n_c	n_s	n_h	HOP	2σ	optimization level	layout
1	3	200	4000	619200	0.774	0.059	0	[0,1,2]
2	3	200	4000	636800	0.796	0.057	3	[0,1,2]
3	4	200	4000	503200	0.629	0.068	0	[0,1,2,3]
4	4	200	4000	538400	0.673	0.066	3	[0,1,2,3]
5	4	1000	4000	2248000	0.562	0.031	3	[2,1,4,7]
6	4	1000	4000	2796000	0.699	0.029	3	[13,14,16,19]
7	5	200	4000	446400	0.558	0.07	0	[0,1,2,3,4]
8	5	200	4000	479200	0.599	0.069	3	[0,1,2,3,4]
9	6	200	4000	404800	0.506	0.071	0	[0,1,2,3,4,5]
10	6	200	4000	417600	0.522	0.071	3	[0,1,2,3,4,5]
11	3	300	5000	1008311	0.672	0.054	0	
12	3	300	5000	1139592	0.76	0.049	0	noise adaptive
13	3	300	10000	2159402	0.72	0.052	0	SABRE
14	4	300	5000	854944	0.57	0.057	0	
15	4	1200	10000	7825121	0.652	0.027	3	
16	4	1200	10000	7842315	0.654	0.027	3	SABRE

Table C.6: Results of experiments on `ibm_cairo`

id	qubits	n_c	n_s	n_h	HOP	2σ	optimization level	layout
1	2	200	4000	597600	0.747	0.061	0	[0,1]
2	2	200	4000	612800	0.766	0.060	3	[0,1]
3	3	200	4000	625600	0.782	0.058	0	[0,1,2]
4	3	200	4000	624000	0.78	0.059	3	[0,1,2]
5	4	900	2000	1181392	0.656	0.032	3	noise adaptive

Table C.7: Results of experiments on `ibmq_lima`

Bibliography

Aaronson, S. & Chen, L. (2016), 'Complexity-theoretic foundations of quantum supremacy experiments'.
URL: *https://arxiv.org/abs/1612.05903*

Algorithmic Qubits: A Better Single-Number Metric (2022).
URL: *https://ionq.com/posts/february-23-2022-algorithmic-qubits*

ANIS, M. S., Abby-Mitchell, Abraham, H., AduOffei, Agarwal, R., Agliardi, G., Aharoni, M., Ajith, V., Akhalwaya, I. Y., Aleksandrowicz, G., Alexander, T., Amy, M., Anagolum, S., Anthony-Gandon, Araujo, I. F., Arbel, E., Asfaw, A., Athalye, A., Avkhadiev, A., Azaustre, C., BHOLE, P., Banerjee, A., Banerjee, S., Bang, W., Bansal, A., Barkoutsos, P., Barnawal, A., Barron, G., Barron, G. S., Bello, L., Ben-Haim, Y., Bennett, M. C., Bevenius, D., Bhatnagar, D., Bhatnagar, P., Bhobe, A., Bianchini, P., Bishop, L. S., Blank, C., Bolos, S., Bopardikar, S., Bosch, S., Brandhofer, S., Brandon, Bravyi, S., Bronn, N., Bryce-Fuller, Bucher, D., Burov, A., Cabrera, F., Calpin, P., Capelluto, L., Carballo, J., Carrascal, G., Carriker, A., Carvalho, I., Chen, A., Chen, C.-F., Chen, E., Chen, J. C., Chen, R., Chevallier, F., Chinda, K., Cholarajan, R., Chow, J. M., Churchill, S., CisterMoke, Claus, C., Clauss, C., Clothier, C., Cocking, R., Cocuzzo, R., Connor, J., Correa, F., Crockett, Z., Cross, A. J., Cross, A. W., Cross, S., Cruz-Benito, J., Culver, C., Córcoles-Gonzales, A. D., D, N., Dague, S., Dandachi, T. E., Dangwal, A. N., Daniel, J., Daniels, M., Dartiailh, M., Davila, A. R., Debouni, F., Dekusar, A., Deshmukh, A., Deshpande, M., Ding, D., Doi, J., Dow, E. M., Downing, P., Drechsler, E., Dumitrescu, E., Dumon, K., Duran, I., EL-Safty, K., Eastman, E., Eberle, G., Ebrahimi, A., Eendebak, P., Egger, D., ElePT, Emilio, Espiricueta, A., Everitt, M., Facoetti, D., Farida, Fernández, P. M., Ferracin, S., Ferrari, D., Ferrera, A. H., Fouilland, R., Frisch, A., Fuhrer, A., Fuller, B., GEORGE, M., Gacon, J., Gago, B. G., Gambella, C., Gambetta, J. M., Gammanpila, A., Garcia, L., Garg, T., Garion, S., Garrison, J. R., Garrison, J., Gates, T., Georgiev, H., Gil, L., Gilliam, A., Giridharan, A., Glen, Gomez-Mosquera, J., Gonzalo, de la Puente González, S., Gorzinski, J., Gould, I., Greenberg, D., Grinko, D., Guan, W., Guijo, D., Guillermo-Mijares-Vilarino, Gunnels, J. A., Gupta, H., Gupta, N., Günther, J. M., Haglund, M., Haide, I., Hamamura, I., Hamido, O. C., Harkins, F., Hartman, K., Hasan, A., Havlicek, V., Hellmers, J., Herok, Ł., Hillmich, S., Hong, C., Horii, H., Howington, C., Hu, S., Hu, W., Huang, C.-H., Huang, J., Huisman, R., Imai, H., Imamichi, T., Ishizaki, K., Ishwor, Iten, R., Itoko, T., Ivrii, A., Javadi, A., Javadi-Abhari, A., Javed, W., Jianhua, Q., Jivrajani, M., Johns, K., Johnstun, S., Jonathan-Shoemaker, JosDenmark, JoshDumo, Judge, J., Kachmann, T., Kale, A., Kanazawa, N., Kane, J., Kang-Bae, Kapila, A., Karazeev, A., Kassebaum, P., Kehrer, T., Kelso, J., Kelso, S., van Kemenade, H., Khanderao, V., King, S., Kobayashi, Y., Kovi11Day, Kovyrshin, A., Krishnakumar, R., Krishnamurthy, P., Krishnan, V., Krsulich, K., Kumkar, P., Kus, G., LaRose, R., Lacal, E., Lambert, R., Landa, H., Lapeyre, J., Latone, J., Lawrence, S., Lee, C., Li, G., Liang, T. J., Lishman, J., Liu, D., Liu, P., Lolcroc, M, A. K., Madden, L., Maeng, Y., Maheshkar, S., Majmudar, K., Malyshev, A., Mandouh, M. E., Manela, J., Manjula, Marecek, J., Marques, M., Marwaha, K., Maslov, D., Maszota, P., Mathews, D., Matsuo, A., Mazhandu, F., McClure, D., McElaney, M., McElroy, J., McGarry, C., McKay, D., McPherson, D., Meesala, S., Meirom, D., Mendell, C., Metcalfe, T., Mevissen, M., Meyer, A., Mezzacapo, A., Midha, R., Miller, D., Miller, H., Minev, Z., Mitchell, A., Moll, N., Montanez, A., Monteiro, G., Mooring, M. D., Morales, R., Moran, N., Morcuende, D., Mostafa, S., Motta, M., Moyard, R., Murali, P., Murata, D., Müggenburg, J., NEMOZ, T., Nadlinger, D., Nakanishi, K., Nannicini, G., Nation, P., Navarro, E., Naveh, Y., Neagle, S. W., Neuweiler, P., Ngoueya, A., Nguyen, T., Nicander, J., Nick-Singstock, Niroula, P., Norlen, H., NuoWenLei, O'Riordan, L. J., Ogunbayo, O., Ollitrault, P., Onodera, T., Otaolea, R., Oud, S., Padilha, D., Paik, H., Pal, S., Pang, Y., Panigrahi, A., Pascuzzi, V. R., Perriello, S., Peterson, E., Phan, A., Pilch, K., Piro, F., Pistoia, M., Piveteau, C., Plewa, J., Pocreau, P., Pozas-Kerstjens, A., Pracht, R., Prokop, M., Prutyanov, V., Puri, S., Puzzuoli, D., Pythonix, Pérez, J., Quant02, Quintiii, Rahman, R. I., Raja, A., Rajeev, R., Rajput, I., Ramagiri, N., Rao, A., Raymond, R., Reardon-Smith, O., Redondo, R. M.-C., Reuter, M., Rice, J., Riedemann, M., Rietesh, Risinger, D., Rivero, P., Rocca, M. L., Rodríguez, D. M., RohithKarur, Rosand, B., Rossmannek, M., Ryu, M., SAPV, T., Sa, N. R. C., Saha, A., Ash-Saki, A., Sanand, S., Sandberg, M., Sandesara, H., Sapra, R., Sargsyan, H., Sarkar, A., Sathaye, N., Savola, N., Schmitt, B., Schnabel, C., Schoenfeld, Z., Scholten, T. L., Schoute, E., Schulterbrandt, M., Schwarm, J., Seaward, J., Sergi, Sertage, I. F., Setia, K., Shah, F., Shammah, N., Shanks, W., Sharma, R., Shaw, P., Shi, Y., Shoemaker, J., Silva, A., Simonetto, A., Singh, D., Singh, D., Singh, P., Singkanipa, P., Siraichi, Y., Siri, Sistos, J., Sitdikov, I., Sivarajah, S., Slavikmew, Sletfjerding, M. B., Smolin, J. A., Soeken, M., Sokolov, I. O., Sokolov, I., Soloviev, V. P., SooluThomas, Starfish, Steenken, D., Stypulkoski, M., Suau, A., Sun, S., Sung, K. J., Suwama, M., Słowik, O., Taeja, R., Takahashi, H., Takawale, T., Tavernelli, I., Taylor, C., Taylour, P., Thomas, S., Tian, K., Tillet, M., Tod, M., Tomasik, M., Tornow, C., de la Torre, E., Toural, J. L. S., Trabing, K., Treinish, M., Trenev, D., TrishaPe, Truger, F., Tsilimigkounakis, G., Tulsi, D., Tuna, D., Turner, W., Vaknin, Y., Valcarce, C. R., Varchon, F., Vartak, A., Vazquez, A. C., Vijaywargiya, P., Villar, V., Vishnu, B., Vogt-Lee, D., Vuillot, C., Weaver, J., Weidenfeller, J., Wieczorek, R., Wildstrom, J. A., Wilson, J., Winston, E., WinterSoldier, Woehr, J. J., Woerner, S., Woo, R., Wood, C. J., Wood, R., Wood, S., Wootton, J., Wright, M., Xing, L., YU, J., Yaiza, Yang, B., Yang, U., Yao, J., Yeralin, D., Yonekura, R., Yonge-Mallo, D., Yoshida, R., Young, R., Yu, J., Yu, L., Yuma-Nakamura, Zachow, C., Zdanski, L., Zhang, H., Zidaru, I., Zimmermann, B., Zoufal, C., aeddins ibm, alexzhang13, b63, bartek bartlomiej, bcamorrison, brandhsn, chetmurthy, choerst ibm, deeplokhande, dekel.meirom, dime10, dlasecki, ehchen, ewinston, fanizzamarco, fs1132429, gadial, galeinston, georgezhou20, georgios ts, gruu, hhorii, hhyap, hykavitha, itoko, jeppevinkel, jessica angel7, jezerjojo14, jliu45, johannesgreiner, jscott2, kUmezawa, klinvill, krutik2966, ma5x, michelle4654, msuwama, nico lgrs, nrhawkins, ntgiwsvp, ordmoj, sagar pahwa, pritamsinha2304, rithikaadiga, ryancocuzzo, saktar unr, saswati qiskit, septembrr, sethmerkel, sg495, shaashwat, smturro2, sternparky, strickroman, tigerjack, tsura crisaldo, upsideon, vadebayo49, welien, willhbang, wmurphy collabstar, yang.luh, yuri@FreeBSD & Čepulkovskis, M. (2021), 'Qiskit: An open-source framework for quantum computing'.

Arute, F., Arya, K., Babbush, R., Bacon, D., Bardin, J., Barends, R., Biswas, R., Boixo, S., Brandao, F., Buell, D., Burkett, B., Chen, Y., Chen, J., Chiaro, B., Collins, R., Courtney, W., Dunsworth, A., Farhi, E., Foxen, B., Fowler, A., Gidney, C. M., Giustina, M., Graff, R., Guerin, K., Habegger, S., Harrigan, M., Hartmann, M., Ho, A., Hoffmann, M. R., Huang, T., Humble, T., Isakov, S., Jeffrey, E., Jiang, Z., Kafri, D., Kechedzhi, K., Kelly, J., Klimov, P., Knysh, S., Korotkov, A., Kostritsa, F., Landhuis, D., Lindmark, M., Lucero, E., Lyakh, D., Mandrà, S., McClean, J. R., McEwen, M., Megrant, A., Mi, X., Michielsen, K., Mohseni, M., Mutus, J., Naaman, O., Neeley, M., Neill, C., Niu, M. Y., Ostby, E., Petukhov, A., Platt, J., Quintana, C., Rieffel, E. G., Roushan, P., Rubin, N., Sank, D.,

Satzinger, K. J., Smelyanskiy, V., Sung, K. J., Trevithick, M., Vainsencher, A., Villalonga, B., White, T., Yao, Z. J., Yeh, P., Zalcman, A., Neven, H. & Martinis, J. (2019), 'Quantum supremacy using a programmable superconducting processor', *Nature* **574**, 505–510.
URL: *https://www.nature.com/articles/s41586-019-1666-5*

Asbóth, J., Adam, P., Koniorczyk, M. & Janszky, J. (2004), 'Coherent-state qubits: Entanglement and decoherence', *The European Physical Journal D* **30**, 403–410.

Asfaw, A., Bello, L., Ben-Haim, Y., Bravyi, S., Capelluto, L., Vazquez, A. C., Ceroni, J., Harkins, F., Gambetta, J., Garion, S., Gil, L., Gonzalez, S. D. L. P., McKay, D., Minev, Z., Nation, P., Phan, A., Rattew, A., Schaefer, J., Shabani, J., Smolin, J., Temme, K., Tod, M. & Wootton., J. (2020), 'Learn quantum computation using qiskit'.
URL: *http://community.qiskit.org/textbook*

Baldwin, C. H., Mayer, K., Brown, N. C., Ryan-Anderson, C. & Hayes, D. (2022), 'Re-examining the quantum volume test: Ideal distributions, compiler optimizations, confidence intervals, and scalable resource estimations', *Quantum* **6**, 707.
URL: *https://doi.org/10.22331%2Fq-2022-05-09-707*

Bertsekas, D. & Tsitsiklis, J. (2002), *Introduction to Probability*, Athena Scientific books, Athena Scientific.
URL: *https://books.google.es/books?id=bcHaAAAAMAAJ*

Biamonte, J., Wittek, P., Pancotti, N., Rebentrost, P., Wiebe, N. & Lloyd, S. (2017), 'Quantum machine learning', *Nature* **549**(7671), 195–202.
URL: *https://doi.org/10.1038%2Fnature23474*

Bruzewicz, C. D., Chiaverini, J., McConnell, R. & Sage, J. M. (2019), 'Trapped-ion quantum computing: Progress and challenges', *Applied Physics Reviews* **6**(2), 021314.
URL: *https://doi.org/10.1063%2F1.5088164*

Campagna, M., LaMacchia, B. & Ott, D. (2021), 'Post quantum cryptography: Readiness challenges and the approaching storm'.
URL: *https://arxiv.org/abs/2101.01269*

Cirac, J. I. & Zoller, P. (1995), 'Quantum computations with cold trapped ions', *Phys. Rev. Lett.* **74**, 4091–4094.
URL: *https://link.aps.org/doi/10.1103/PhysRevLett.74.4091*

Cross, A. W., Bishop, L. S., Sheldon, S., Nation, P. D. & Gambetta, J. M. (2019), 'Validating quantum computers using randomized model circuits', *Physical Review A* **100**(3).
URL: *https://doi.org/10.1103%2Fphysreva.100.032328*

Cross Entropy Benchmarking Theory (2022).
URL: *https://quantumai.google/cirq/noise/qcvv/xeb_theory*

Deutsch, D. & Jozsa, R. (1992), 'Rapid solution of problems by quantum computation', **439**.
URL: *https://doi.org/10.1098/rspa.1992.0167*

DiVincenzo, D. P. (1995), 'Two-bit gates are universal for quantum computation', *Physical Review A* **51**(2), 1015–1022.
URL: *https://doi.org/10.1103%2Fphysreva.51.1015*

DiVincenzo, D. P. (2000), 'The physical implementation of quantum computation', *Fortschritte der Physik* **48**(9-11), 771–783.
URL: *https://doi.org/10.1002%2F1521-3978%28200009%2948%3A9%2F11%3C771%3A%3Aaid-prop771%3E3.0.co%3B2-e*

Dongarra, J. J., Moler, C. B., Bunch, J. R. & Stewart, G. W. (1979), *LINPACK Users' Guide*, Society for Industrial and Applied Mathematics.
URL: *https://epubs.siam.org/doi/abs/10.1137/1.9781611971811*

Dongarra, J., Luszczek, P. & Petitet, A. (2003), 'The linpack benchmark: past, present and future', *Concurrency and Computation: Practice and Experience* **15**, 803–820.

Dunjko, V., Taylor, J. M. & Briegel, H. J. (2016), 'Quantum-enhanced machine learning', *Physical Review Letters* **117**(13).
URL: *https://doi.org/10.1103%2Fphysrevlett.117.130501*

Farhi, E., Goldstone, J. & Gutmann, S. (2014), 'A quantum approximate optimization algorithm'.
URL: *https://arxiv.org/abs/1411.4028*

Floyd, R. W. (1962), 'Algorithm 97: Shortest path', *Commun. ACM* **5**(6), 345.
URL: *https://doi.org/10.1145/367766.368168*

Gaebler, J. P., Tan, T. R., Lin, Y., Wan, Y., Bowler, R., Keith, A. C., Glancy, S., Coakley, K., Knill, E., Leibfried, D. & Wineland, D. J. (2016), 'High-fidelity universal gate set for

^{9}Be$^+$ ion qubits', *Phys. Rev. Lett.* **117**, 060505.
URL: *https://link.aps.org/doi/10.1103/PhysRevLett.117.060505*

Grover, L. K. (1996), 'A fast quantum mechanical algorithm for database search'.
URL: *https://arxiv.org/abs/quant-ph/9605043*

Guillermo-Mijares-Vilarino (2022), 'Guillermo-Mijares-Vilarino/Master-thesis: Master thesis code'.
URL: *https://doi.org/10.5281/zenodo.7079951*

Havlíček, V., Córcoles, A. D., Temme, K., Harrow, A. W., Kandala, A., Chow, J. M. & Gambetta, J. M. (2019), 'Supervised learning with quantum-enhanced feature spaces', *Nature* **567**(7747), 209–212.
URL: *https://doi.org/10.1038%2Fs41586-019-0980-2*

Hogg, R., Tanis, E. & Zimmerman, D. (1977), *Probability and Statistical Inference*, MacMillan Publishing Company.

IBM Quantum (2022).
URL: *https://quantum-computing.ibm.com/*

IBM Quantum Roadmap (2022).
URL: *https://www.ibm.com/quantum/roadmap*

IBM Quantum simulators overview (2022).
URL: *https://quantum-computing.ibm.com/services/resources/docs/resources/manage/simulator*

Jurcevic, P., Javadi-Abhari, A., Bishop, L. S., Lauer, I., Bogorin, D. F., Brink, M., Capelluto, L., Günlük, O., Itoko, T., Kanazawa, N., Kandala, A., Keefe, G. A., Krsulich, K., Landers, W., Lewandowski, E. P., McClure, D. T., Nannicini, G., Narasgond, A., Nayfeh, H. M., Pritchett, E., Rothwell, M. B., Srinivasan, S., Sundaresan, N., Wang, C., Wei, K. X., Wood, C. J., Yau, J.-B., Zhang, E. J., Dial, O. E., Chow, J. M. & Gambetta, J. M. (2020), 'Demonstration of quantum volume 64 on a superconducting quantum computing system'.
URL: *https://arxiv.org/abs/2008.08571*

Kandala, A., Mezzacapo, A., Temme, K., Takita, M., Brink, M., Chow, J. M. & Gambetta, J. M. (2017), 'Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets', *Nature* **549**(7671), 242–246.
URL: *https://doi.org/10.1038%2Fnature23879*

Kitaev, A. Y. (1995), ‘Quantum measurements and the abelian stabilizer problem’.
URL: *https://arxiv.org/abs/quant-ph/9511026*

Kjaergaard, M., Schwartz, M. E., Braumüller, J., Krantz, P., Wang, J. I.-J., Gustavsson, S. & Oliver, W. D. (2020), ‘Superconducting qubits: Current state of play’, *Annual Review of Condensed Matter Physics* **11**(1), 369–395.
URL: *https://doi.org/10.1146%2Fannurev-conmatphys-031119-050605*

Kok, P., Munro, W. J., Nemoto, K., Ralph, T. C., Dowling, J. P. & Milburn, G. J. (2007), ‘Linear optical quantum computing with photonic qubits’, *Reviews of Modern Physics* **79**(1), 135–174.
URL: *https://doi.org/10.1103%2Frevmodphys.79.135*

Kumar, M. (2022), ‘Post-quantum cryptography algorithms standardization and performance analysis’.
URL: *https://arxiv.org/abs/2204.02571*

Li, G., Ding, Y. & Xie, Y. (2018), ‘Tackling the qubit mapping problem for nisq-era quantum devices’.
URL: *https://arxiv.org/abs/1809.02573*

Loss, D. & DiVincenzo, D. P. (1998), ‘Quantum computation with quantum dots’, *Physical Review A* **57**(1), 120–126.
URL: *https://doi.org/10.1103%2Fphysreva.57.120*

Makhlin, Y., Schön, G. & Shnirman, A. (2001), ‘Quantum-state engineering with josephson-junction devices’, *Reviews of Modern Physics* **73**(2), 357–400.
URL: *https://doi.org/10.1103%2Frevmodphys.73.357*

Murali, P., Baker, J. M., Abhari, A. J., Chong, F. T. & Martonosi, M. (2019), ‘Noise-adaptive compiler mappings for noisy intermediate-scale quantum computers’.
URL: *https://arxiv.org/abs/1901.11054*

Peruzzo, A., McClean, J., Shadbolt, P., Yung, M.-H., Zhou, X.-Q., Love, P. J., Aspuru-Guzik, A. & O’Brien, J. L. (2014), ‘A variational eigenvalue solver on a photonic quantum processor’, *Nature Communications* **5**(1).
URL: *https://doi.org/10.1038%2Fncomms5213*

Pilnyak, Y., Zilber, P., Cohen, L. & Eisenberg, H. S. (2019), 'Quantum tomography of photon states encoded in polarization and picosecond time bins', *Physical Review A* **100**(4).
URL: *https://doi.org/10.1103%2Fphysreva.100.043826*

Preset Passmanagers (2022).
URL: *https://qiskit.org/documentation/apidoc/transpiler_preset.html*

Quantinuum Announces Quantum Volume 4096 Achievement (2022).
URL: *https://www.quantinuum.com/pressrelease/quantinuum-announces-quantum-volume-4096-achievement*

Rivest, R. L., Shamir, A. & Adleman, L. M. (n.d.), 'A method for obtaining digital signatures and public-key cryptosystems', *CACM* **26**(1), 96–99.

Schuld, M. & Killoran, N. (2019), 'Quantum machine learning in feature hilbert spaces', *Physical Review Letters* **122**(4).
URL: *https://doi.org/10.1103%2Fphysrevlett.122.040504*

Shor, P. W. (1997), 'Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer', *SIAM Journal on Computing* **26**(5), 1484–1509.
URL: *https://doi.org/10.1137%2Fs0097539795293172*

Source code for SabreLayout, Qiskit (2022).
URL: *https://qiskit.org/documentation/_modules/qiskit/transpiler/passes/layout/sabre_layout.html*

Source code for SabreSWAP, Qiskit (2022).
URL: *https://qiskit.org/documentation/_modules/qiskit/transpiler/passes/routing/sabre_swap.html*

Srinivasan, K., Satyajit, S., Behera, B. K. & Panigrahi, P. K. (2018), 'Efficient quantum algorithm for solving travelling salesman problem: An ibm quantum experience'.
URL: *https://arxiv.org/abs/1805.10928*

Summit Oak Ridge National Laboratory's 200 petaflop supercomputer (2022).
URL: *https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/*

Tesch, C. & Vivie-Riedle, R. (2002), 'Quantum computation with vibrationally excited molecules', *Physical review letters* **89**, 157901.

Thiele, S., Balestro, F., Ballou, R., Klyatskaya, S., Ruben, M. & Wernsdorfer, W. (2014), 'Electrically driven nuclear spin resonance in single-molecule magnets', *Science* **344**, 1135–8.

TOP500 page (2022).
URL: *https://www.top500.org/project/linpack/*

Unveiling IonQ Forte: The First Software-Configurable Quantum Computer (2022).
URL: *https://ionq.com/posts/may-17-2022-ionq-forte*

Utkarsh, Behera, B. K. & Panigrahi, P. K. (2020), 'Solving vehicle routing problem using quantum approximate optimization algorithm'.
URL: *https://arxiv.org/abs/2002.01351*

Wack, A., Paik, H., Javadi-Abhari, A., Jurcevic, P., Faro, I., Gambetta, J. M. & Johnson, B. R. (2021), 'Quality, speed, and scale: three key attributes to measure the performance of near-term quantum computers'.
URL: *https://arxiv.org/abs/2110.14108*