Quantum Computing: Rethinking Implementations and Performance

Louis Qin Keqin Yan William Zhu
University of California, Los Angeles
{louisqin, kyan233, williamzhu}@g.ucla.edu

Abstract

We implement six quantum algorithms on both Google's Sycamore and IBM's quantum computers and experiment with different quantum error correction methods. In comparison, IBM's quantum computer demonstrated a compelling superiority over Google's Sycamore when the number of qubits is large.

Limited by the optimization provided by quantum compiler, we had to rethink our implementations of the algorithms, drastically changing our previous projects. In this process, we discover several novel ideas in our implementation, including 1. a generalized oracle for Grover's algorithm, a new error corrective action that does not depend on measuring the helper qubits, 2. dynamically saving the number of helper qubits in an instance of Simon's algorithm.

To evaluate the performance of a quantum computer, we explain why a metric based on error rate is not sufficient. In this project, we rethink the performance metric as a statistics problem. Here, we draw inspirations from information theory, introducing and modifying KL divergence as our performance metric.

1. Introduction

1.1. Tasks

In this project, we focus on transplanting our previous quantum circuit implementations on simulators onto actual quantum computers. We implement the following six algorithms: Deutsch Jozsa Algorithm, Bernstein Vazirani Algorithm, Simon's Algorithm, Grover's Algorithm, QAOA Algorithm, and Shor's Algorithm.

1.2. Programming Languages and Platforms

We work with two quantum programming languages, Google's Cirq and IBM's Qiskit in our implementations. We run our programs on Google's Sycamore and IBM's Melbourne, as well as some of IBM's 5 qubits machines like Athens and Santiago for less demanding circuits.

Since we have already written all six algorithms in Cirq to run on simulators as part of our previous assignments, modifying these Cirq implementations to work on Google's quantum computer is an obvious starting point for this project. In addition, since Cirq and Qiskit both support circuits in QASM language format, we write scripts to automatically convert Cirq circuits into Qiskit circuits, saving ourselves the time to re-implement them from scratch in Qiskit.

1.3. Difference from Previous Implementations

Previously, we made heavy use of `cirq.MatrixGate` to convert unitary matrix into simulator-executable gates. However, since Cirq does not support converting `cirq.MatrixGate` into basic gates, we have to design our own heuristics to decompose the unitary matrix into a cascade of basic gates.

2. Experiment

2.1. Deutsch Jozsa Algorithm

2.1.1 Algorithm Overview

Deutsch Jozsa's Algorithm solves the following problem: Let function f takes in a bit string of length n, and outputs a binary value (0 or 1). Assume f is either constant or balanced, i.e. f either outputs only one value for all inputs, or outputs 0 for half of the inputs and 1 for the other.

Our program will take in the function f and determines if it is a constant or a balanced function. Specifically, if the quantum circuit outputs 0^n, f is a constant function, and if the output is anything else, f is a balanced function. We apply Hadamard gates on all input qubits and one helper qubit. Then, after going through the oracle, we apply another round of Hadamard gates on all qubits except the helper qubit. We then measure all qubits except the helper qubit.

2.1.2 Oracle Implementation

Adapting IBM's design for Deutsch Jozsa oracle [Asfaw et al., 2020], our oracles are generalized as follows: For constant function f, the oracle randomly generates NOT gates and sets the output value of f as 0 or 1. For balanced function f, the oracle consists of a set of CNOT gates and NOT gates. For CNOT gates, the target is the helper qubit while the controls are the rest. NOT gates are randomly applied on both sides of some qubits before and after the CNOTs.

2.2. Bernstein Vazirani Algorithm

2.2.1 Algorithm Overview

Bernstein Vazirani's Algorithm solves the following problem: Let function f takes in a bit string of length n, and outputs a binary value. Assume $f(x) = a \cdot x + b$, in which a is a bit string, $\cdot$ is mod 2 inner product, and b is a binary value. We are looking for the values a and b.

Our program will take in a function f and determines a and b accordingly. It shares the same circuit as Deutsch Jozsa program.

2.2.2 Oracle Implementation

Due to the similarity in circuit, Bernstein Vazirani shares the same oracle with the Deutsch Jozsa algorithm.

2.3. Simon's Algorithm

2.3.1 Algorithm Overview

Simon's Algorithm solves the following problem: Let function f takes in a bit string of length n, and outputs another bit string of length n. Assume that there exists a secret bit string, s, of length n, and for any two arbitrary bit string with length n, say x and y, f(x) = f(y) if and only if x + y equals to 0 or s.

Our program takes in an oracle that represents a function with a secret string s, and outputs either the secret string or all zero. The regular Simon's circuit works as follows: first applies Hadamard gates to all main qubits, then after applying the oracle, it applies another round of Hadamard gates to the main qubits, and measures them afterwards. There is an option of measuring the helper qubits before measuring the main qubits, but our simulations returns the same results regardless of this step.

2.3.2 Oracle Implementation

We develop a novel algorithm that could potentially decrease the need for the helper qubits; for instance, for a secret string 100, our circuit only needs 1 helper qubit, which is a rather big improvement compared to the 3 qubits needed for the version presented in class.

Our oracle, inspired by the IBM's design [Asfaw et al., 2020], is constructed according to the following thinking process: After receiving the secret string, the algorithm examines the string and identify which locations would be 1. As it turns out, we only need the same number of helper qubits as the number of 1s in our secret string. The oracle then applies CNOT between each of the 1 qubits and the helper qubits as needed, eventually giving us the correct results in the simulator. Since we do not have enough time to fully investigate how to generalize this process, we mainly investigate the case of having 2 and 3 main qubits and with no more than 2 helper qubits.

2.4. Grover's Algorithm

2.4.1 Algorithm Overview

Grover's Algorithm solves the following problem: Let function f takes in a bit string of length n, and outputs another bit value (0 or 1). The task is to figure out whether there is a unique input, x, such that f(x) will return 1.

Our program takes in the function f and return 1 if there is a unique input, x, that will result in $f(x) = 1$.

2.4.2 Oracle Implementation

The main oracle in Grover's Algorithm is the Zf matrix. Zf is a diagonal matrix with 1 everywhere and -1 at the diagonal entry that corresponds to basis x, where x is the unique input that makes $= 1$. By observing Zf's properties, we derive a general algorithm that constructs a quantum computer-compatible oracle for a given f using NOT gates and controlled-Z gates.

We observe that applying the Zf matrix flips the phase of the unique x-basis while having no effect on any other basis. As a result, we apply NOT gates to qubits where the target x has 0 in its bit representation, so that only the x-basis would be mapped to the state with 1 in all qubits. Then, we apply a controlled-Z gate to flip the phase of the state only if all qubits have 1. For one-qubit circuits, this gate would simply be a normal Z gate, for two-qubit circuits a CZ gate, and for three-qubit circuits a CCZ gate, and so on and so forth. Then, NOT gates are applied on the same qubits to undo the effect of previous NOT gates.

In practice, however, since such CC..CCZ gates with large number of C's are not native to quantum computers and must be decomposed into elementary gates, and that the number of elementary gates required grows exponentially as a function of the number of C, in practice this algorithm only works with n=1,2,3, n being the number of qubits.

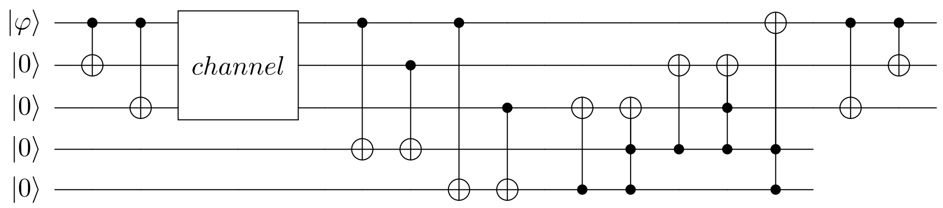

Table 1. General bit flip error correction

2.5. QAOA Algorithm

2.5.1 Algorithm Overview

Our approach for QAOA algorithm in this assignment is realized with the Max Cut problem, which asks about when using a line to divide a group of dots into half, what would be the maximal number of the intersections this line can have with the connection lines between the dots. During our experiments, we found out that if we allow any arbitrary dot inputs, it would lead to the qubit mapping problem, which we will discuss in our Challenge section (Section 5.1). Therefore, although our program is relatively generalized in this experiment, we focus on investigating Max Cut problems involves 4 qubits due to the qubit mapping issue.

2.5.2 Oracle Implementation

The oracle of QAOA involves two parts: separator and mixer. And specifically, we adapted the design of the oracale from Google's Max Cut solution [Sung et al., 2021].

For our separator, we took the approach of using the Z-parity gate of Google raised to an power decided by the random angle factor we generate. Mixer is done by the similar process yet through the Rx gate.

2.6. Shor's Algorithm

2.6.1 Algorithm Overview

Shor's Algorithm takes in an odd, composite number N that is not a prime or a power of prime, and attempts to return a non-trivial factor of N, i.e., a factor named d such that $1 < d < N$ and d divides N. The process goes as such: first, the program generates a random integer, a, from 2, …, N-1 and finds the greatest common divisor of a and N, sets it as d. At this point, if $d > 1$, the program returns it. Otherwise, it uses the FindOrder program which includes the modular exponential and the quantum Fourier transform. Then, let $r = FindOrder(a, N)$, and if r is even, the program checks to see if the greatest common divisor of $(a^{r/2-1}, N)$ is larger than 1, if so, it returns it; otherwise, the program has no result in this loop, and depending on the given number of tries, it either goes for another round or exits the loop. The chance of not able to find a good r is less than 50% each loop.

Measured helper bits 4,5	Corrective Actions
00	Do Nothing
10	Flip qubit 2
01	Flip qubit 3
11	Flip the main qubit

Table 2. Corrective Actions in Lecture

2.6.2 Modular Exponentiation and Quantum Fourier Transform

Quantum Order Finding is the core of Shor's Algorithm, and it is composed of two parts, Modular Exponentiation and Quantum Fourier Transform. The circuit for Modular Exponentiation is often cited as the primary bottleneck in Quantum Order Finding, as a different circuit is needed for each choice of N and a. In this project, we focus on implementing one particular circuit with N=15 and a=2. We borrow the compiled architecture for this circuit from Gamel and James [2013], which, by observing the behavior of modular exponentiation with these specific parameters, reduce the 8 qubits origininally needed to just 2 qubits. This is because the lower 6 qubits of the output are identically zero given any quantum state inputs, and therefore it is only necessary to consider how modular exponentiation behaves on the top 2 qubits. For Quantum Fourier Transform, we use the provided cft gate in the Cirq language, which fortunately can be compiled to be run on Sycamore.

3. Error Correction

3.1. Unconditional Corrective Action

In the lecture, the corrective actions for bit flip and phase flip errors are outlined in Table 2, with qubit 2, 3, 4, 5 as the helper qubits. However, this method is conditional on the measurement of qubits 4, 5, which is not possible in cirq. Instead we introduce a method that is logically equivalent but not contingent on the measurements qubits 4, 5.

We realize that qubit 4 and 5 are not in superposition but rather in a single state, and as a result, we decide to go with CNOT and CCNOT gates instead of the corrective actions based on measurements. Table 1 presents a design that replicates the logic of the corrective actions in the lecture. However, as it requires too many CCNOT gates which

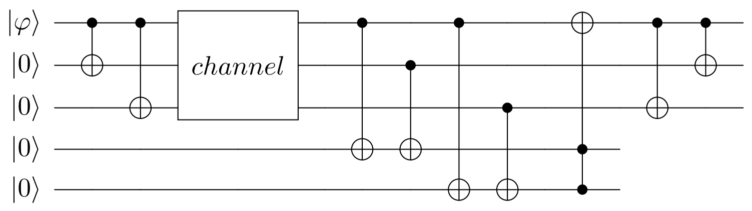

Table 3. Bit flip error correction on only $|\varphi\rangle$

decompose into prohibitive amount of elementary gates on Sycamore, we introduce a more bare-bone version in Table 3 which only corrects qubit 1, but not qubit 2, 3.

3.2. Deutsch Jozsa and Bernstein Vazirani

Because we obtained low accuracy on a circuit of 2 qubits, a main qubit and a helper qubit, we implement error corrections on our main qubit. Since there are two types of error corrections, bit flip and phase flip, we implement 2 versions separately.

As bit flip correction would conflict with the purpose of having gates like X or H, we decide to implement the bit flip correction around the DJ oracle, which only involves some CNOT gate and does not change the bit value of the main qubit. The process of the bit flip correction is as follows: we first encode the state of the main qubit with two helper qubits (different from the DJ helper qubit, which are qubit 2 and 3), then after the channel, where potential error could happen, we detect if the main qubit or the two helper qubits's information changed with the help of two other helper qubits (qubit 4 and 5). And lastly, we apply corrective actions correspondingly.

Phase flip circuit is similar to the bit flip circuit, except that it wraps the channel with Hadamard gates. Yet, since DJ algorithm does not really have a suitable channel option, we implement an artificial phase flip error to test our error correction implementation, which successfully recovered the correct information.

Since BV and DJ share the majority of the circuits, we implement the same error correction for BV and achieve very similar results.

3.3. Simon

For error correction in Simon, since there isn't really a good place to insert the error correction algorithms, we implement an artificial bit flip error to test our error correction implementation, which successfully recovered the correct information in the simulation.

3.4. Grover

For Grover's algorithm, the Zf oracle provides a suitable option for testing the bit flip algorithm as it does not cause any change to the bit value. By implementing error correction around the channel, Zf, we achieved the expected results in the simulations. Unfortunately, we do not see a good place to place the channel of the phase flip algorithm, and we had to implement an artificial phase flip error to test the algorithm, which gave us the correct results in the simulations.

3.5. Shor: Potential Generalization of Error Correction

Shor's specific problem is interesting because as we have been purposefully using the same group of qubits, we could save the time of manually trace the swapping of qubits again due to hardware constraint, if our target correcting qubit is the same one. Therefore, we test it out with Shor's algorithm and set the target qubit to the (1, 5) in the Sycamore grid. As the code presents, we did successfully reuse our code with only minor modifications. The strategy of correcting the same qubit can potentially be reused as long as we keep track of two things: 1. the order of the qubits in each program, and 2. the selection of the channel.

4. Results

4.1. Error metrics: Simulator v.s. Quantum Computer

In lectures, the most frequently discussed error metric for quantum computer is the error rate, i.e. $1 - accuracy$. In practice, however, we find this metric to be unrepresentative of the performance of a quantum computer. Rather, we formulate the performance of a quantum computer as a statistics problem:

For a given circuit C, the measurements of a noiseless simulator is modeled by a discrete random variable S_C while the measurements of a quantum computer is modeled by a discrete random variable Q_C.

Recall the definition of support for a discrete distribution X:

$$Support(X) = \{x \in \{0,1\}^n : P(X = x) > 0\}$$

where n is the number of qubits measured.

$Dnorm_{KL}(S_C \| Q_C)$	Interpretation
$= 0$	Noiseless
$\in (0, 1)$	Good
$= 1$	Equivalent to a random number generator
> 1	Quantum computer is very confident about something wrong

Table 4. Interpreting normalized KL-Divergence

4.1.1 Accuracy

We formulate accuracy of a quantum computer as:

$$Accuracy = \sum_{x \in Support(S_C)} P(Q_C = x)$$

However, this error metric is especially problematic when the final state before measurement is a superposition, i.e. $Support(S_C) = \{0, 1\}^n$. In this case, the accuracy of a quantum computer is always 1.

Accuracy is also not helpful for interpretation. Consider the following example, where we measure one qubit in a circuit. In a simulator, we obtain a deterministic measurement of 0, and in a quantum computer, we get 0 or 1 50% of the time. By the definition of accuracy, the quantum computer is 50% accurate. However, in reality, this performance is merely on par with that of a random number generator. In other words, the metric of accuracy portrays the performance to be better than it seems.

4.1.2 Normalized KL-divergence

Motivated by the aforementioned issues, we borrow and modify an idea from information theory - KL-divergence. Formally, KL-divergence is defined as:

$$D_{KL}(P \| Q) = \sum_{x \in X} P(x) \log \frac{P(x)}{Q(x)}$$

Informally, we can think of $D_{KL}(P \| Q)$ as the distance of a random variable Q to a true random variable P. If $D_{KL}(P \| Q) = 0$, two distributions match exactly. Therefore, we use $D_{KL}(S_C \| Q_C)$ to denote how far off is the noisy quantum computer measurements to the noiseless simulator measurements.

Since the empirical results suggest the quantum computer is extremely noisy, we introduce a normalized version of KL-divergence to compare the performance of a quantum computer to that of a uniform random number generator:

$$Dnorm_{KL}(S_C \| Q_C) = \frac{D_{KL}(S_C \| Q_C)}{D_{KL}(S_C \| U(0, 2^n))}$$

where $U(0, 2^n)$ is a discrete uniform distribution and n is the number of qubits measured. Informally, it normalizes the distance of S_C and Q_C with the distance of S_C and $U(0, 2^n)$. We provide an intuitive interpretation of normalized KL-divergence in Table 4. However, note that the validity of this metric hinges on S_C being far from $U(0, 2^n)$. When S_C is too close to $U(0, 2^n)$, the metric will explode.

4.2. Error Correction

We sum up our results of error correction in Table 5. As we expected, since the error correction circuit greatly increases the number of operations (e.g. volume), and in real life, there is no error-free operation, this increase directly leads to the significant increase in the error rate.

Specifically, in the Deutsch Jozsa circuits, not only do we compare the results with and without error correction, we also analyze the results with correcting the main qubit and the helper qubit. In addition, phase flip has a slight higher error rate, since most of the time, we had to add an artificial phase error. Although the error correction in general does not work as well as on the simulator, we still confirmed that there isn't an obvious difference between correcting the main qubit and the helper qubit. Continuing this thought, we tested our implementation of error corrections on the helper qubit of Shor's algorithm. The result, however, seems suboptimal. This is mainly because our Shor's algorithm takes a special approach in its compilation and gives an output that is close to the uniform distribution, which causes the normalized KL metric to explode.

For Simon's algorithm, we only tested the bit flip algorithm as we found out that the error correction is not very useful in this situation. Although the volume increased from 65 to 1296, the accuracy and kl-normalized values do not get much worse. This is potentially due to the fact that we selected a pretty noise circuit and that error correction does not make it much noisier.

In addition, we could not get the results from Grover's error corrections and Shor's bit flip correction in time due to various reasons such as the Sycamore server breakdown and exceeding the operation limits.

4.3. Scalability

We hypothesized that the performance of a quantum computer is explained by the following variables:

1. number of qubits

2. number of moments

Algorithm	Accuracy	kl-normalized	Notes
DJ	0.6919	0.531365	no correction
DJ	0.4682	1.094803	bit flip - correction strategy 1
DJ	0.4682	0.997117	bit flip - correct main qubit state, strategy 2
DJ	0.5257	0.927688	phase flip - correct main qubit, strategy 2
DJ	0.494	1.017417	phase flip - correct helper qubit, strategy 2
Simon	0.0359	2.455858	no correction
Simon	0.0336	2.545284	bit flip - correction strategy 1
Shor	1	1	no correction
Shor	1	6.108683	phase flip corrected, strategy 2

Table 5. Error Correction Results

3. number of gates

4. quantum volume

Indeed, analyzing the metrics across 28 experiments on Sycamore (excluding QAOA), we observe positive correlations between these variables with all performance metrics in Figure 1. Among these variables, number of qubits attains the highest correlation.

We exclude 18 QAOA experiments because all of them share the same variables therefore will skew the data heavily. This effect is reported in Figure 3. However, the high variance in QAOA performance suggests other hidden variables to explain the variance in performance. Therefore, although the aforementioned variables explain the performance to a certain degree, more experiments are needed to understand what determine the performance of a quantum computer.

4.4. Sycamore vs IBM

In general, we observe that Sycamore has a better performance than IBM with smaller number of qubits, but IBM has a more stable performance than Sycamore across different numbers of qubits. In Appendix B, we present our full comparison across the two platforms. Here, as an example, the plots for the accuracy and normalized kl values are shown for Sycamore and IBM 2.

5. Challenges

In this section, we discuss the challenges we face when porting implementations from previous projects to Sycamore. Specifically, we find the circuit optimizer provided by cirq to be insufficient when 1. mapping logical qubits to Sycamore grid qubits, 2. swapping qubits to support logical connection when physical connection does not exist, and 3. decomposing an arbitrary unitary matrix to a composition of basic gates.

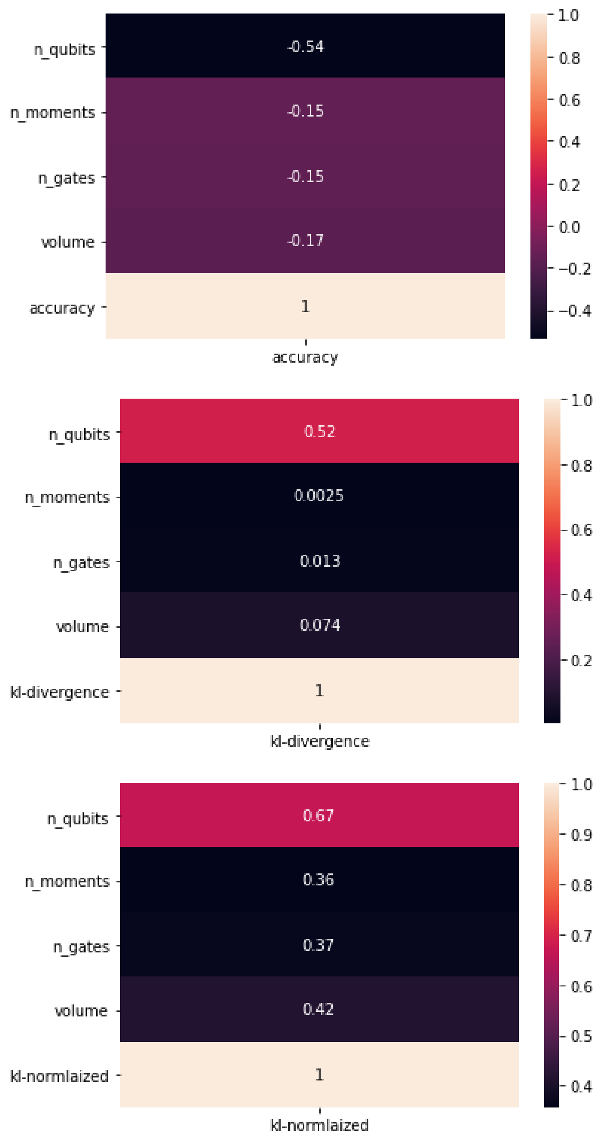

Figure 1. Correlations between variables and metrics on experiments excluding QAOA

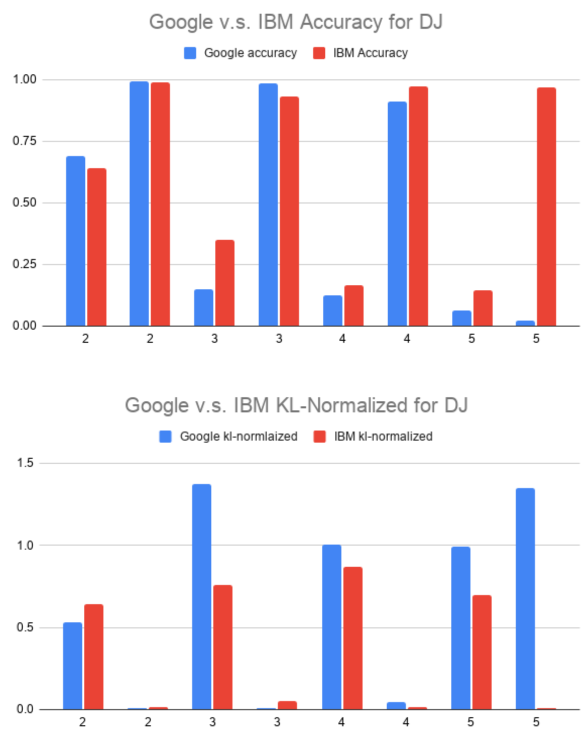

Figure 2. Google v.s. IBM metrics for Deutsch-Jozsa

5.1. Qubit mapping

Since Cirq does not support mapping logical `cirq.LineQubit`. We identify the topology of our problems and strategically place our qubits on the corresponding Sycamore lattice. Sometimes, such a mapping is not immediately possible because each physical qubit on Sycamore is only connected to 4 other qubits. In this case, we introduce swapping operations to our circuit.

5.2. Swapping

Due to the limits of quantum circuit compilers, we have to manually swap the qubits for each operations that need logical connections between two or more qubits to ensure that they have physical connections as well. This need for extra operations greatly limits our scope of experiments under time constraints as it significantly increases the number of operations and prevents us from testing certain circuits. For instance, some of our error corrections exceeded the operation limits of Sycamore, and due to the different physical layouts of qubits, some of them cannot be ran on IBM quantum computers. Facing this challenge, however, we took advantage of the fact that the arrangement of the qubits are mostly the same, and successfully reused one of our swapping code for another program (DJ to Shor).

5.3. Oracle decomposition

In previous projects, we were able to generalize oracles by using `cirq.MatrixGate`. However, in this project, due to the limited resources on decomposing an arbitrary unitary matrix into basic gates, our circuits are only generalized to a certain degree, often limited to some special instances of the problems. The details of oracle decomposition are reported in Section 2.

6. Conclusion

In this project, we explore implementing six quantum algorithms on two different families of quantum hardware and software, namely Google's Sycamore with Cirq and IBM's Melbroune, Athens, etc. with Qiskit. We face numerous challenges, such as decomposing complex operations, deriving generalizable oracles, and fitting the circuit to a physical machine, and successfully overcome them with creative solutions. Some problems cannot be solved due the limits of technologies we are using, but we obtain meaningful insights into how to eventually solve them.

References

A. Asfaw, L. Bello, Y. Ben-Haim, M. Bozzo-Rey, S. Bravyi, N. Bronn, L. Capelluto, A. C. Vazquez, J. Ceroni, R. Chen, A. Frisch, J. Gambetta, S. Garion, L. Gil, S. D. L. P. Gonzalez, F. Harkins, T. Imamichi, H. Kang, A. h. Karamlou, R. Loredo, D. McKay, A. Mezzacapo, Z. Minev, R. Movassagh, G. Nannicni, P. Nation, A. Phan, M. Pistoia, A. Rattew, J. Schaefer, J. Shabani, J. Smolin, J. Stenger, K. Temme, M. Tod, S. Wood, and J. Wootton. Learn quantum computation using qiskit, 2020. URL http://community.qiskit.org/textbook. 2

O. Gamel and D. F. V. James. Simplified factoring algorithms for validating small-scale quantum information processing technologies. *arXiv preprint arXiv:1310.6446*, 2013. 3

K. Sung, V. Omole, M. Neeley, D. Bacon, C. Gidney, and A. Zalcman. Cirq/examples/qaoa.py, 2021. URL https://github.com/quantumlib/Cirq/blob/master/examples/qaoa.py. 3

Appendix A. Tables and Plots

job_id	algorithm	n_qubits	n_moments	n_gates	volume	accuracy	kl-divergence	kl-normlaized
5742161945427968	DJ	2	6	8	12	0.6919	0.368314	0.531365
4590568369815552	DJ	2	1	2	2	0.9925	0.007528	0.010861
4938701238960128	DJ	3	9	13	27	0.1492	1.902468	1.372340
5298307073048576	DJ	3	1	2	3	0.9839	0.016231	0.011708
4754422546563072	DJ	4	12	18	48	0.1242	2.085862	1.003088
6481114759692288	DJ	4	1	2	4	0.9112	0.092993	0.044720
5355214852849664	DJ	5	15	23	75	0.0637	2.753571	0.993141
6064601145802752	DJ	5	1	2	5	0.0239	3.733877	1.346711
5707058573737984	BV	2	1	2	2	0.9929	0.007125	0.010280
6053026779365376	BV	2	6	8	12	0.6861	0.376732	0.543509
4669038596718592	BV	3	1	2	3	0.9855	0.014606	0.010536
5101908284932096	BV	3	9	13	27	0.1512	1.889152	1.362735
6701630695145472	BV	4	1	2	4	0.915	0.088831	0.042719
5144108620316672	BV	4	12	18	48	0.0656	2.724180	1.310053
6270008527159296	BV	5	1	2	5	0.023	3.772261	1.360556
5490076825944064	BV	5	15	23	75	0.0737	2.607752	0.940548
6509283168485376	Simon	4	13	23	52	0.6581	0.498900	0.719752
6101486324940800	Simon	4	13	24	52	0.4309	0.899241	0.648484
6235436053692416	Simon	5	13	23	65	0.0359	3.404873	2.455858
4836858974437376	Simon	5	13	24	65	0.0129	4.431845	2.131247
5959457158725632	Simon	9	144	213	1296	0.0336	3.531207	2.545284
6342774399959040	DJ	6	130	189	780	0.4682	0.758860	1.094803
6687685506760704	DJ	6	110	164	660	0.4682	0.691149	0.997117
6569342514757632	DJ	6	132	189	792	0.5257	0.643025	0.927688
5722891836456960	DJ	6	132	189	792	0.494	0.705220	1.017417
5396507205304320	QAOA	4	22	43	88	1	0.092322	14.197192
6363668308557824	QAOA	4	22	43	88	1	0.348013	1.482356
6285841789878272	QAOA	4	22	43	88	1	0.576886	1.135292
4878466906324992	QAOA	4	22	43	88	1	0.584252	1.487900
5372807038894080	QAOA	4	22	43	88	1	0.738099	1.142578
6214614052241408	QAOA	4	22	43	88	1	0.408506	2.019842
6511442127749120	QAOA	4	22	43	88	1	0.734923	1.199553
6498706945736704	QAOA	4	22	43	88	1	0.754549	1.118609
6729668241653760	QAOA	4	22	43	88	1	0.268086	1.419629
6514821663031296	QAOA	4	22	43	88	1	0.418187	2.248439
4685234314412032	QAOA	4	22	43	88	1	0.389264	1.685002
5232069550538752	QAOA	4	22	43	88	1	0.242746	2.200161
6522407112146944	QAOA	4	22	43	88	1	0.479931	1.628281
6208354875604992	QAOA	4	22	43	88	1	1.020501	1.005864
5865655072980992	QAOA	4	22	43	88	1	0.682216	1.371753
4800979992051712	QAOA	4	22	43	88	1	0.411648	0.773346
6004366813167616	QAOA	4	22	43	88	1	0.639576	1.070752
4956293425004544	QAOA	4	22	43	88	1	0.393981	1.611281
4825478485508096	Grover	2	10	15	20	0.64	0.445818	0.321590
6455338379247616	Grover	2	10	15	20	0.6343	0.455233	0.328381
5047963495694336	Grover	3	66	83	198	1	1.520093	1.360118
6173863402536960	Grover	3	66	83	198	1	1.105981	0.983079
5511543609360384	Grover	7	177	288	1239	N/A	N/A	N/A
6348684274958336	Grover	7	189	288	1239	N/A	N/A	N/A
4973603317612544	Shor	4	18	32	72	1	0.053728	1
5811512178376704	Shor	8	144	213	1152	1	0.022514	6.108683

Table 6. Sycamore experiment results

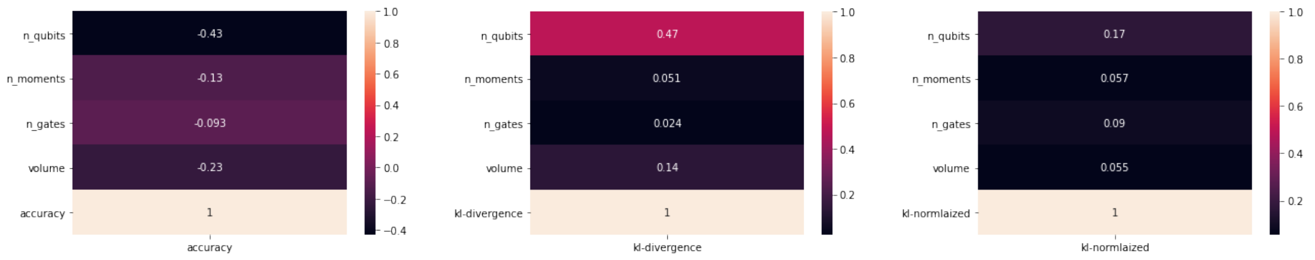

Figure 3. Correlations between variables and metrics on all experiments (skewed by QAOA)

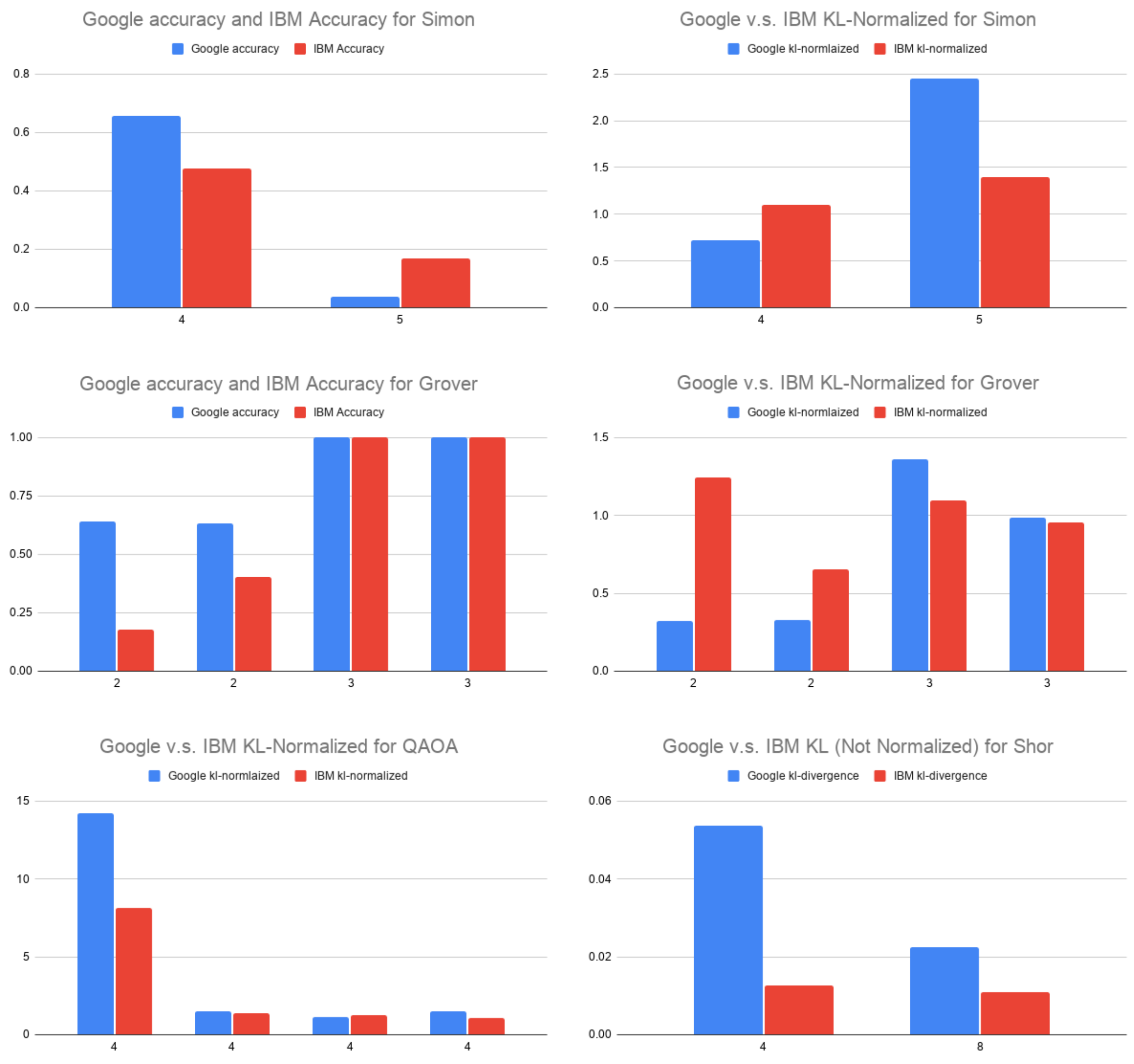

Figure 4. Google v.s. IBM performance plots