Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

COM SCI 239 Run on a Quantum Computer Report

Deutsch-Jozsa

Statistics of Results

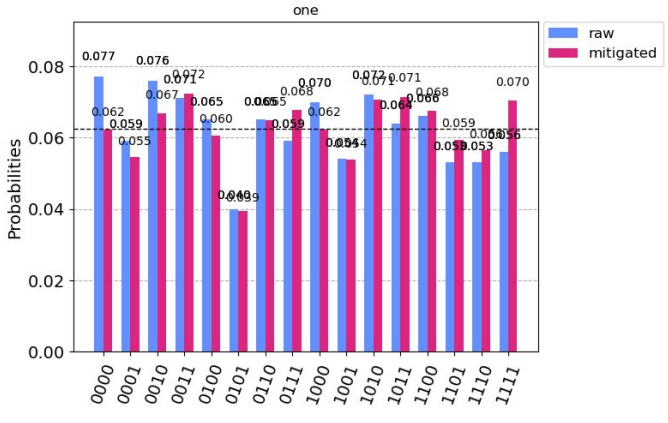
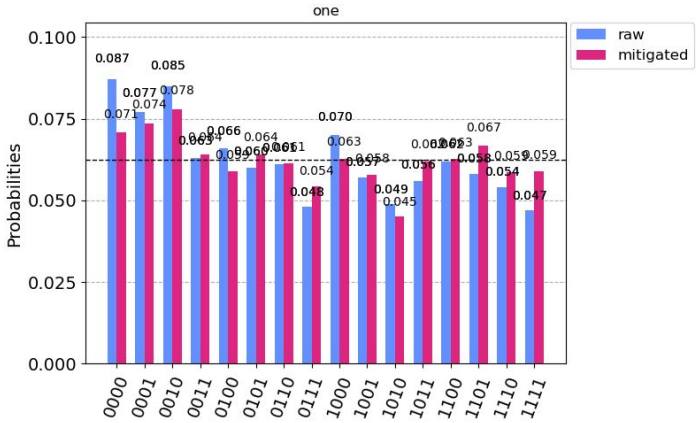
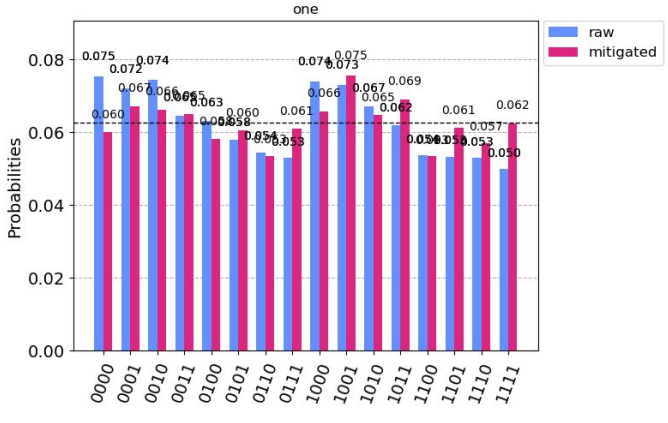
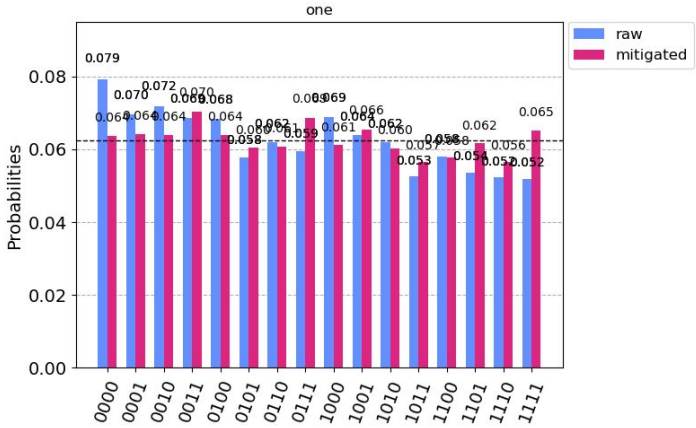
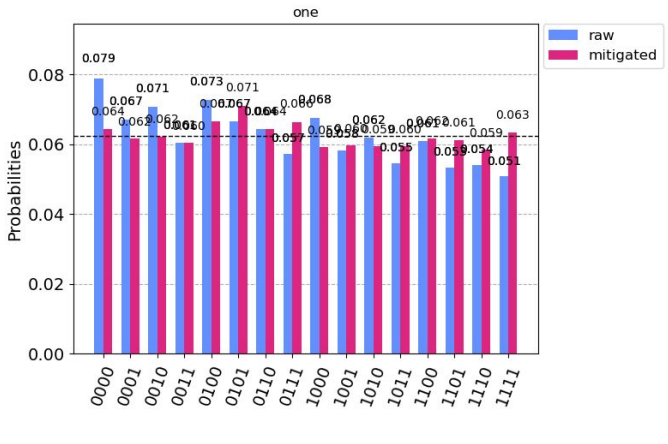
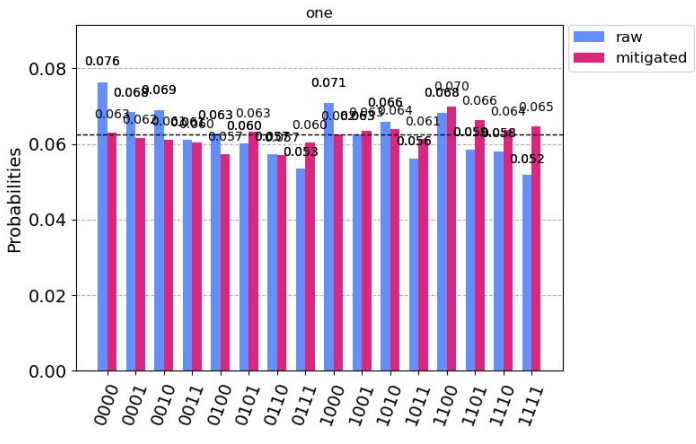
We tested our Deutsch-Jozsa implementation on the functions constant 1 (constant) and XOR (balanced) . We set up our test driver to accept the number of trials to run as a command line argument. Furthermore, prior to running any circuit, we first transpiled it, i.e. we optimally unrolled the gates in the circuit into the universal basis gates which, as we discussed in class, form the instruction set of the quantum device. Our program allows the user to specify the Qiskit level of optimization (integer between 0 and 3) used in transpilation as a command line argument: 0 indicates no optimization, and the higher the optimization level, the more optimized the circuit is, that is, the compiler further reduces the number of basis gates into which the circuit is decomposed. As stated in the Qiskit documentation, “higher optimization levels generate more optimized circuits, at the expense of longer transpilation time.”

For each test, we print out the results in ordering of decreasing frequency across all the shots; we use nice formatting to clearly label the test to which the trials correspond and print, separately, the transpilation time, the total number of gates in the transpiled circuit, and average run time (across all the shots), and subsequently, for each result, present 1) the frequency of the result, 2) the error-mitigated frequency of the result, 3) the actual measurements of the n input qubits and 4) the interpretation of the measurements (i.e. “Constant” if all the qubits are measured to be zero and “Balanced” otherwise). Furthermore, we plot a histogram of the raw and error-mitigated frequencies. The histogram contains a horizontal, black dashed line indicating the probability of random guessing a result given *n* (i.e. 1/2^*n*), to provide reference for the distribution relative to the uniform distribution. We do not provide any single-number statistics of the results (e.g. mean, standard deviation, etc.), but opted instead for well-labeled histograms, as our results were not normally-distributed. (**Note:** we provide more details on our implementation of measurement error mitigation in the **Experience** section)

Presented below are measurement results for running deutsch_jozsa.py with 4 qubits with optimization levels 1 and 3, shots in [1000, 8092, 30000], and 2 different functions (constant 0, XOR).

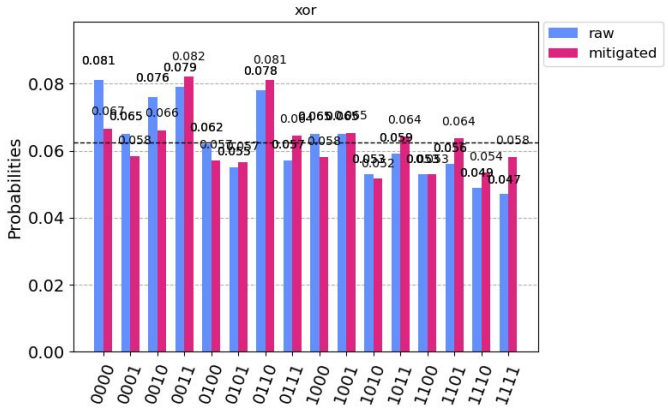
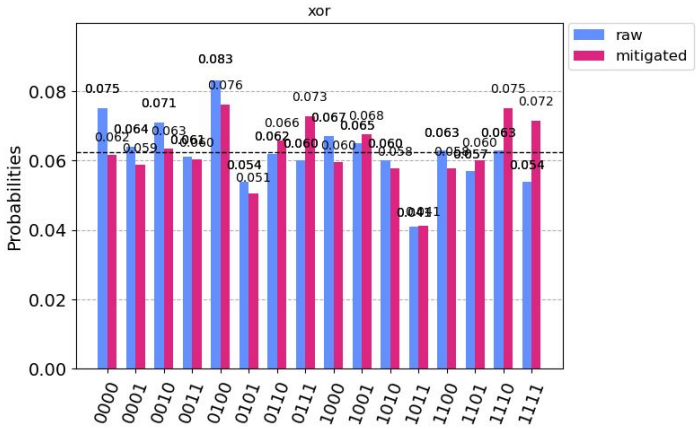
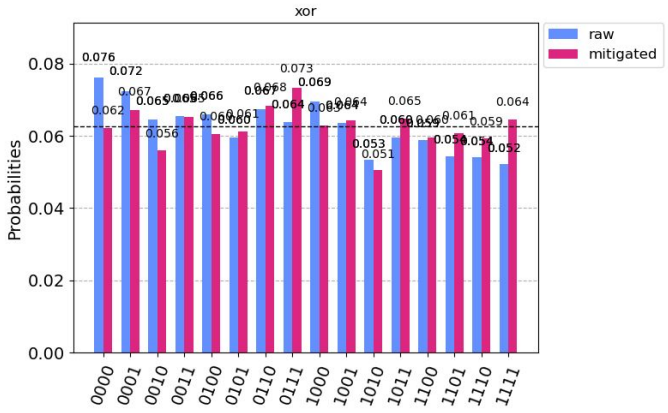
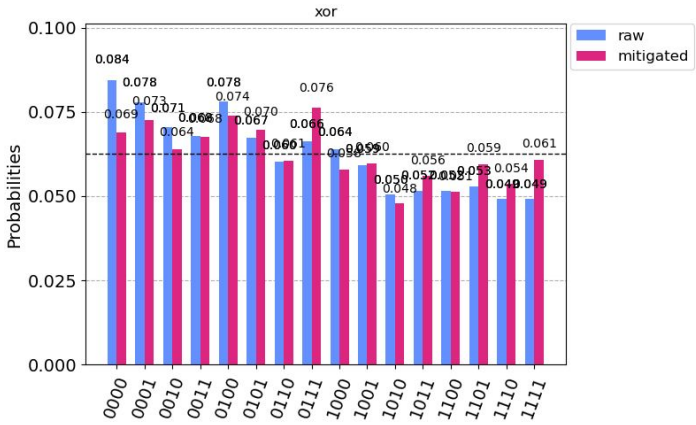
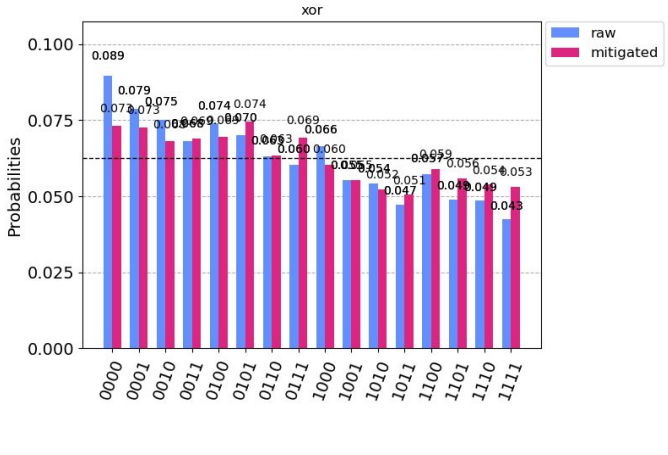
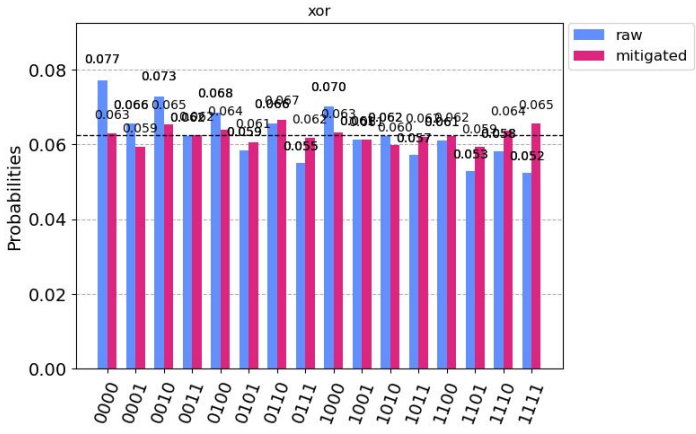
Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

function: one, n: 4

Shots	Optimization Level = 1	Optimization Level = 3
1000		
8092		
30000		

Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

function: xor, n: 4

Shots	Optimization Level = 1	Optimization Level = 3
1000		
8092		
30000		

We were surprised to see that even for a high number of trials and an optimization level of 3, the frequencies are almost uniform in distribution. Moreover, the correct result almost never has the highest frequency (**note:** please review the histograms for *n* = 2 and *n* = 3 in the Bernstein-Vazirani section for a discussion of accuracy of results as the number of qubits increases). We discovered that the noise in real quantum devices is significantly higher than the noise in noisy quantum simulators. The decoherence in the real quantum devices produced a nearly uniform distribution of results over multiple shots for greater than 2 qubits, rather than a unimodal distribution with the mode at the correct result. To resolve this issue, we employed measurement calibration to mitigate measurement errors. The main idea is to prepare all 2^n basis input states and compute the probability of measuring counts in the other basis states. From these calibrations, it is possible to correct the average results of another experiment of interest.

In the XOR function's trial at optimization level 1, the correct output '0000' becomes more evident with higher shots. The expected output for the constant one function '1000' on the other hand wasn't clear even at 30000 shots. This makes sense as measurements on quantum machines are known to have state-dependent bias where '1' is misread as '0' more often than the other way around. As a result, we would expect more error from states with '1's.

Execution Times

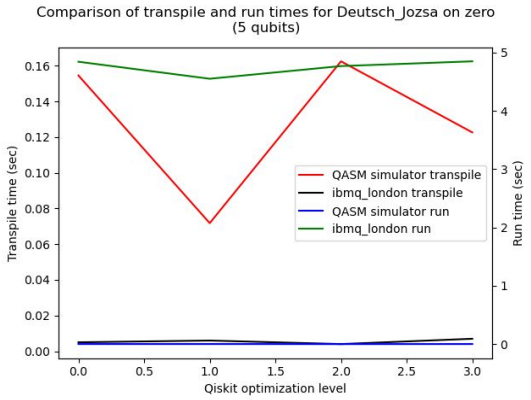
Different cases of U_f definitely lead to different execution times. As stated earlier, for each test, we separately compute the transpilation time and run time (including all the trials). In Qiskit, the qiskit.execute function, which executes circuits, combines transpilation and running. However, we elected not to use the transpilation functionality built into qiskit.execute because that would prevent us from measuring the transpilation time and run time separately. Instead, we first used qiskit.compiler.transpile to transpile the circuit with the appropriate optimization level and timed this operation, then we called qiskit.execute on the transpiled circuit with an optimization level of 0 (so that qiskit.execute wouldn't apply light optimization by default to the already transpiled circuit). We chose to investigate optimization levels in this report to gauge the ability of optimized transpilation to mitigate the long run times associated with a large number of qubits. Hopefully, with more research into optimized transpilation, we can reduce run times and reduce errors in quantum computation.

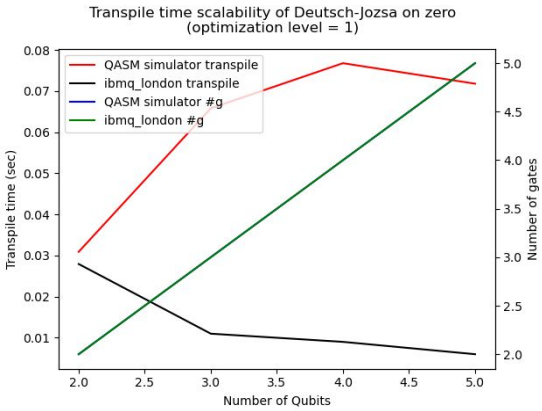
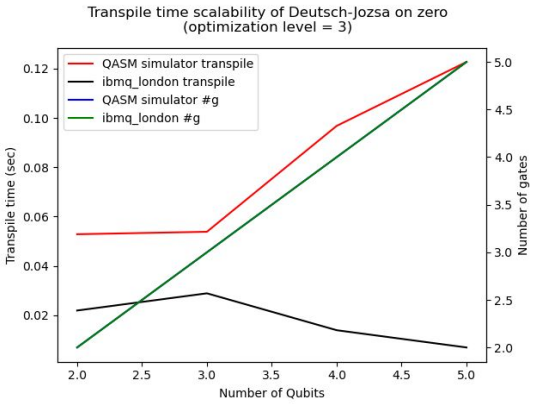
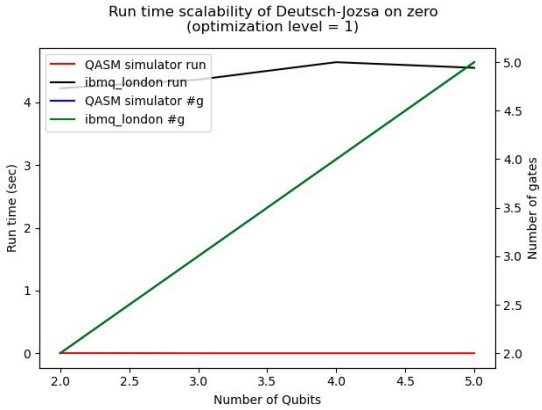
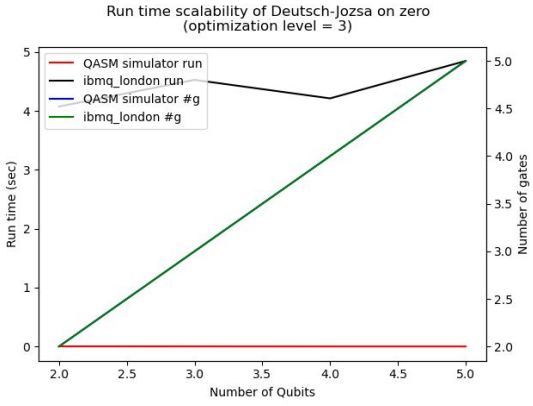
Neither the transpilation time nor the run time includes the construction of the black-box U_f (since Deutsch-Jozsa assumes U_f is available). Presented below are plots comparing the transpilation and execution times and number of gates for running deutsch_jozsa.py with 4 qubits 1-shot with optimization levels 1 and 3, on 4 different functions (constant 0, constant 1, XOR, and XNOR). We decided to include the number of gates on a twin y-axis in the plots. This is because the number of gates greatly affected the error in the results. Furthermore, a circuit that was too large could cause the job to terminate with the following error: "Circuit runtime is

greater than the device repetition rate, exceeded maximum call stack size.” This error prevented us from doing the analysis with a larger number of qubits and a low level of optimization.

Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

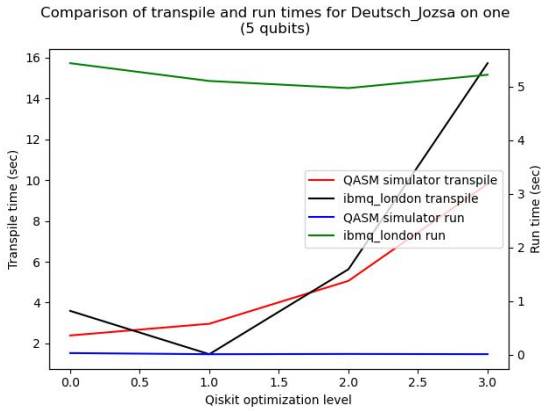
function: zero, n: 4

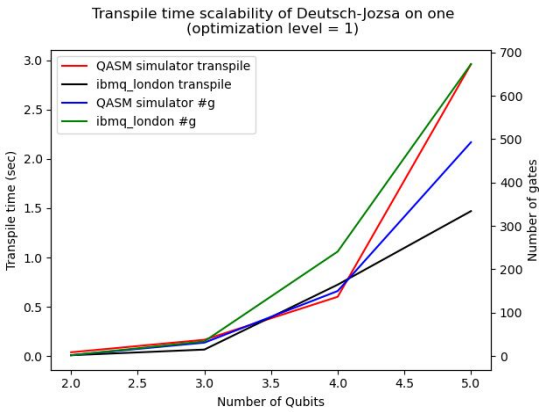
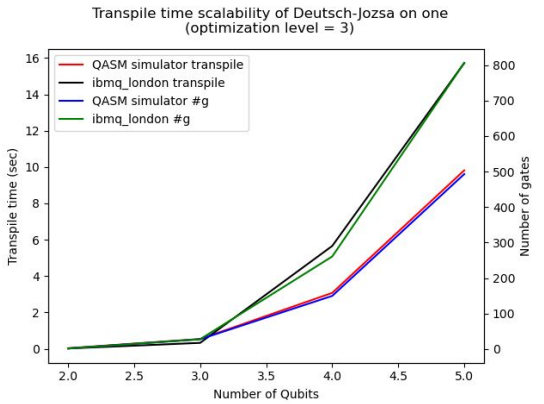
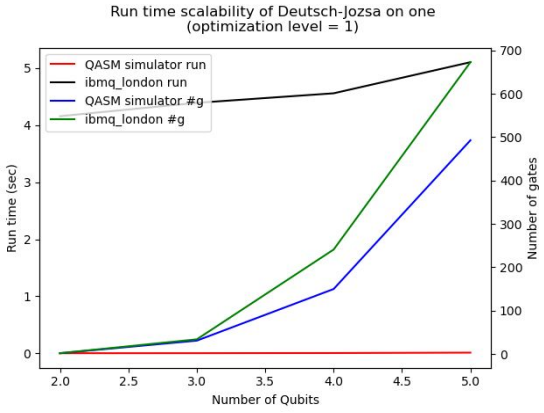
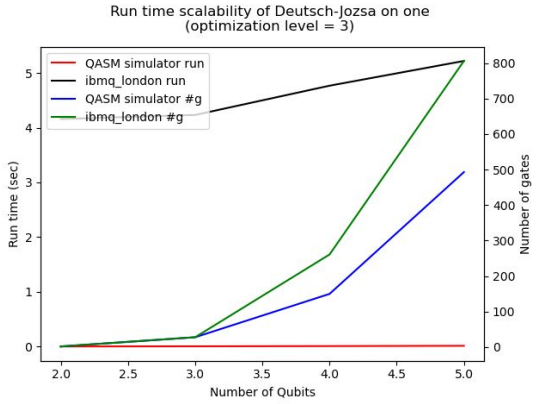
Comparison of Transpile Time and Run Time	

	Optimization Level = 1	Optimization Level = 3
Transpile Time Scalability		
Run Time Scalability		

Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

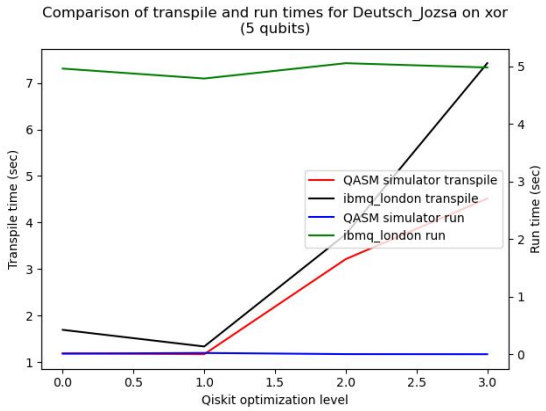
function: one, n: 4

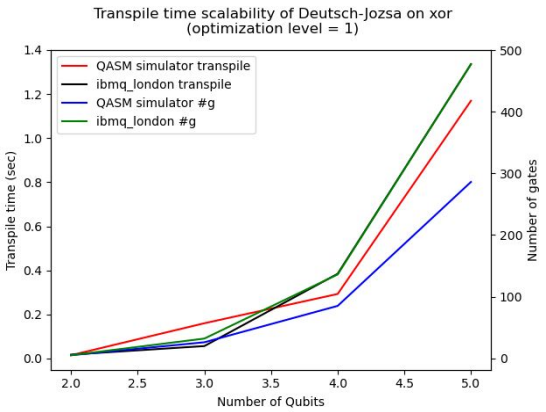
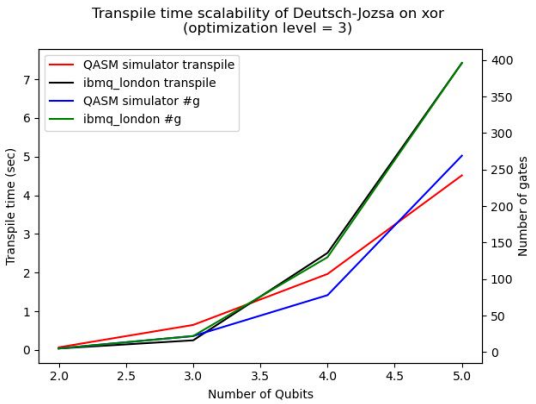
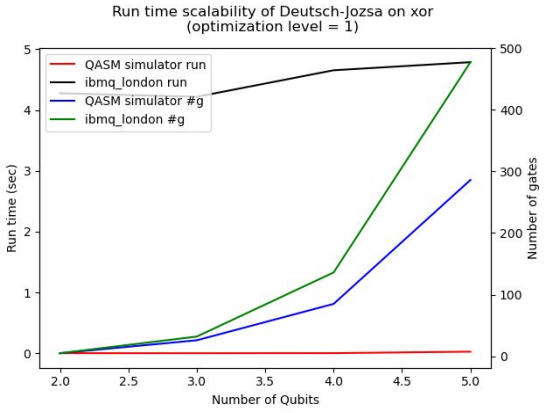
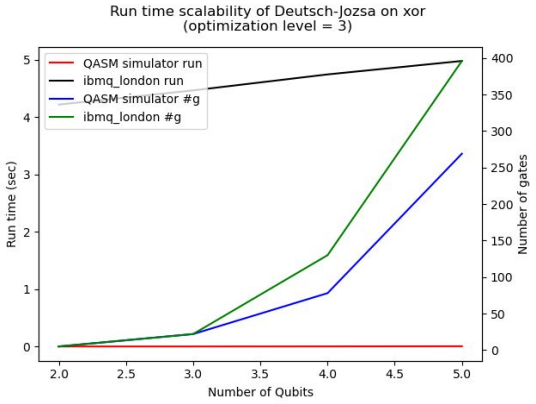
Comparison of Transpile Time and Run Time	

	Optimization Level = 1	Optimization Level = 3
Transpile Time Scalability		
Run Time Scalability		

Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

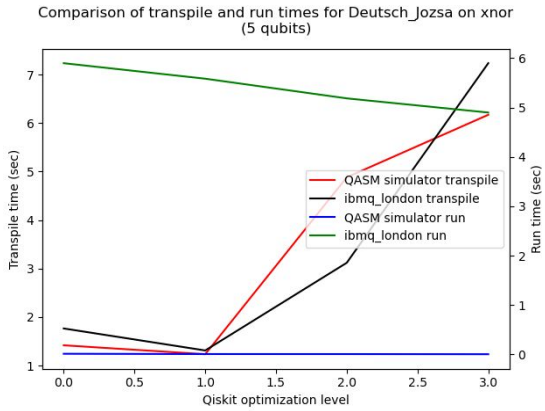
function: xor, n: 4

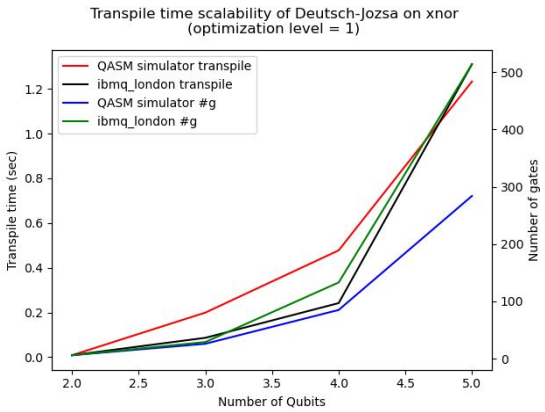
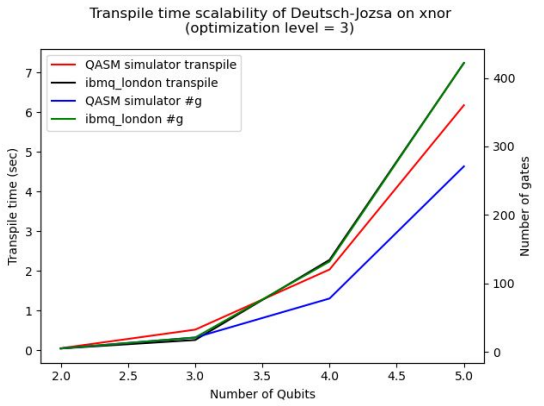
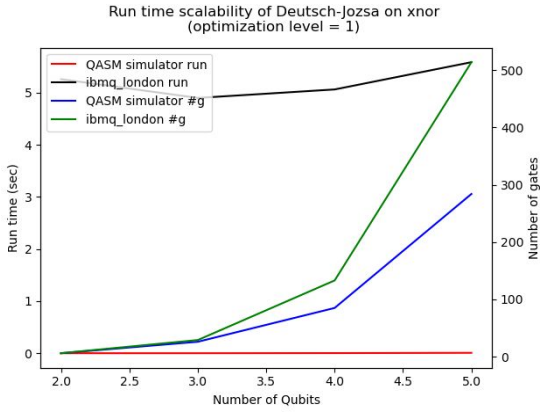
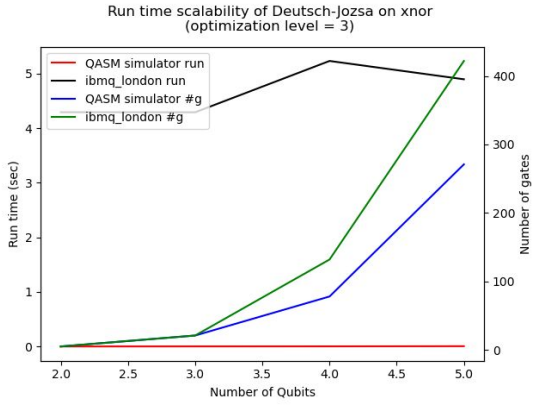
Comparison of Transpile Time and Run Time	

	Optimization Level = 1	Optimization Level = 3
Transpile Time Scalability		
Run Time Scalability		

Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

function: xnor, n: 4

Comparison of Transpile Time and Run Time	

	Optimization Level = 1	Optimization Level = 3
Transpile Time Scalability		
Run Time Scalability		

Importantly, in our analysis below, while the Hadamard gates contribute to the total transpile time and run time of the circuit, we assume that their contribution is minimal and relatively constant across all tests, and that transpiling and running U_f takes significantly longer.

For each function, as the optimization level increases, the transpiling time increases, as the compiler must do more work to more optimally decompose U_f and the other gates in the circuit into the basis gates, while the run time should decrease (obviously, a more optimized circuit with fewer gates will run more quickly).

We noticed that, in general, constant one led to a much faster transpilation time than constant 1, XOR and XNOR. This is the same result we obtained with the simulator. We thought this was due to the fact that the constant functions make it easier for the compiler to decompose U_f because U_f for constant 0 is just the identity $I^{\otimes(n+1)}$ (i.e. no basis gates are applied) and U_f for constant 1 is just $I^{\otimes n} \otimes X$ (i.e. CCNOT, which is a well-known gate), while the balanced functions are more complex since they have an equal amount of and alternating on-diagonal and off-diagonal non-zero entries. Since U_f for constant 0 is simply the identity, it obviously makes it the easiest for the compiler to decompose U_f, since $I^{\otimes(n+1)}$ translates to no basis gates being applied.

Scalability an n Grows

With regards to the scalability of n, for each test U_f, regardless of the optimization level, the transpile time appears to be exponential in number of qubits; this makes sense since, as we discussed in class, transpiling is NP-complete and hence the compiler will have an exponentially harder time transpiling larger matrices whose dimension grows exponentially in the number of qubits. Additionally, for each test U_f besides constant 0, regardless of the optimization level, the run time seems to be exponential in the number of qubits; this also makes sense as larger matrices whose dimension grows exponentially in the number of qubits will likely be transpiled into more basis gates, which is confirmed by our plots. (The run time trends for constant 0 are not significant because, regardless of the level of optimization, the identity is extremely easy for the transpiler to decompose and $I^{\otimes(n+1)}$ translates to no basis gates being applied.)

Regardless of the number of qubits, higher optimization levels generate circuits with a shorter run time, at the expense of longer transpilation time, which substantiates Qiskit's documentation. Again, we hope that, with more research into optimized transpilation, we can reduce run times for large numbers of qubits and reduce errors in quantum computation.

Lastly, as we stated earlier, a circuit that was too large could cause the job to terminate with the following error: "Circuit runtime is greater than the device repetition rate, exceeded maximum call stack size." This error prevented us from doing the analysis with a larger number of qubits. Even though accuracy of results remains an issue with a large number of qubits, limiting the

size of circuits that can be run on real quantum devices is a huge hindrance to scalability as *n* grows.

Simulator and Quantum Computer Comparison

We initially wrote our programs to find the least busy backend to run on, but then realized that potentially using different backends makes the execution time comparisons between different functions unreliable. For instance, different quantum devices could have different topologies, which could change the execution time, as that would affect how our program is compiled. Furthermore, there are many differences on the hardware level, e.g. measurement errors will be different on different backends.To eliminate this confounding variable, we used ibmq_london for all runs of deutsch_jozsa. The transpiling time was greater for the QASM simulator than the quantum machine in most cases. This may be because of more quantum device instruction-set specific transpilation optimization techniques used. The execution times for the quantum machine and the simulator increased as the number of qubits increased. The execution times for the simulator were almost zero while those of the quantum machine were much higher. Indeed, we get a longer execution time on the actual quantum computer than on the simulator because the problem scale is so small that local simulation is easy.

Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

Bernstein-Vazirani (bernstein_vazirani.py)

Statistics of Results

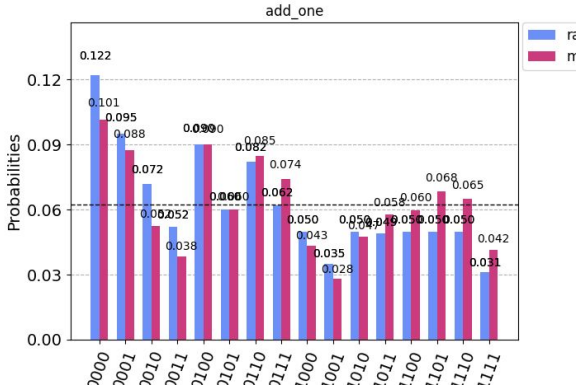
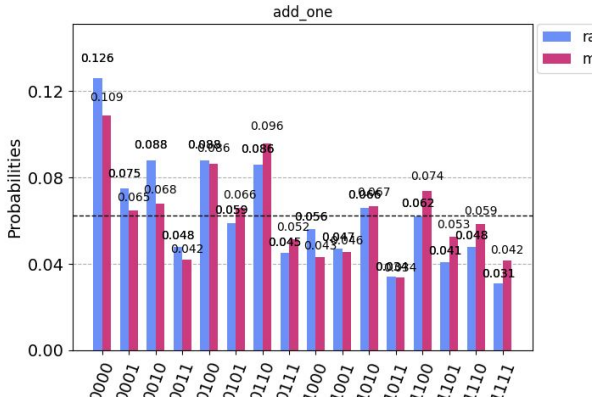
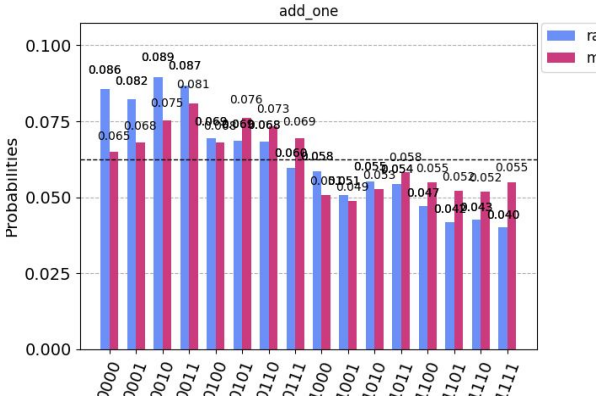
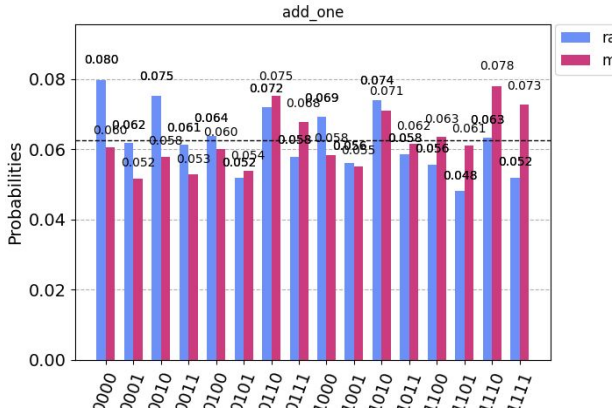
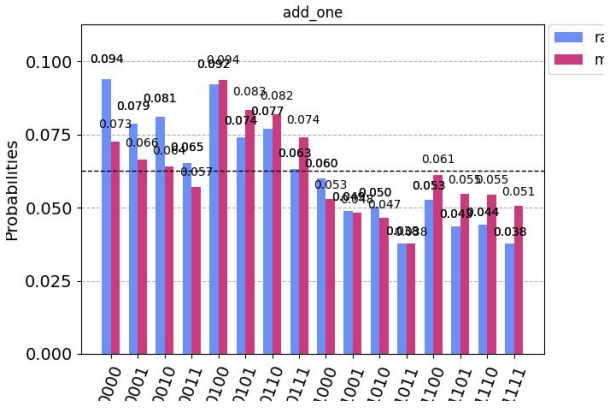
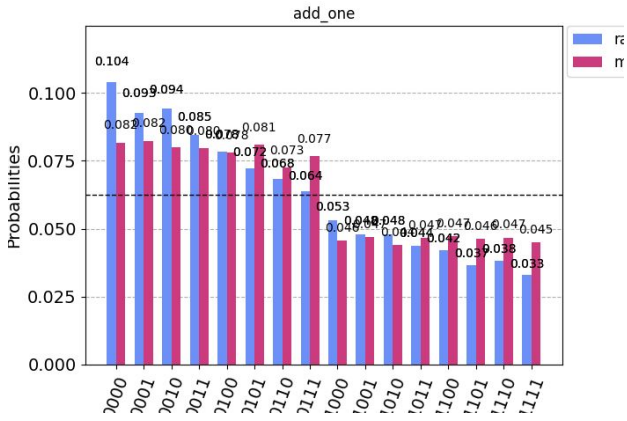
We tested our Bernstein-Vazirani implementation on the following functions:

- add_one: returns the sum of all bits in the bit string plus 1 (mod 2)
    - This test allows us to additionally validate that the classical computation of b is correct
- bottom_half: returns the sum of the second half, rounded up, of the bit string (mod 2)
- first_bit: returns the value of the most significant bit in the bit string
- top_half: returns the sum of the first half, rounded down, of the bit string (mod 2)

The presentation of the results is the same as for Deutsch-Jozsa. For each test, we print out the results in order of decreasing frequency; we use nice formatting to clearly label the test to which the trials correspond and print, separately, the transpilation time, the total number of gates in the transpiled circuit, and run time for the entire test (including all the trials), and subsequently, for each result, present 1) the frequency of the result, 2) the error-mitigated frequency of the result, 3) the actual measurements of the n input qubits and 4) the interpretation of the measurements (i.e. a is just the measurements of the n input qubits and b is computed classically by evaluating the function on {0}^n). Furthermore, we plot a histogram of the raw and error-mitigated frequencies. The histogram contains a horizontal, black dashed line indicating the probability of random guessing a result given *n* (i.e. 1/2^*n*), to provide reference for the distribution relative to the uniform distribution. Presented below are measurement results for running bernstein_vazarani.py with 4 qubits with optimization levels 1 and 3, shots in [1000, 8092, 25000], and 4 different functions (add_one, bottom_half, first_bit, top_half).

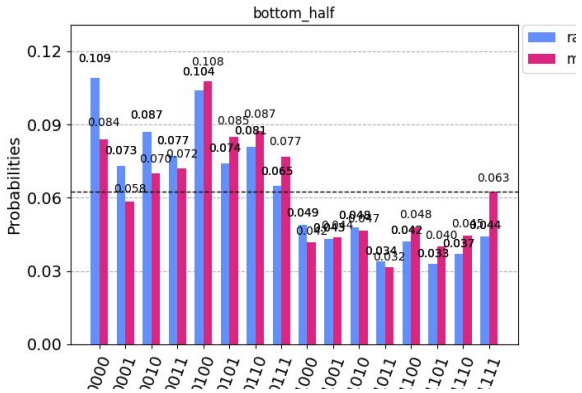
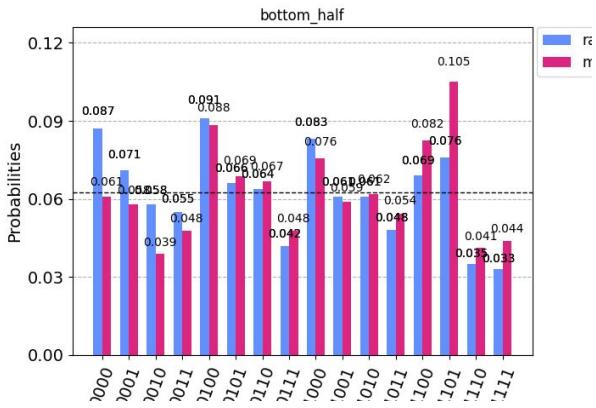
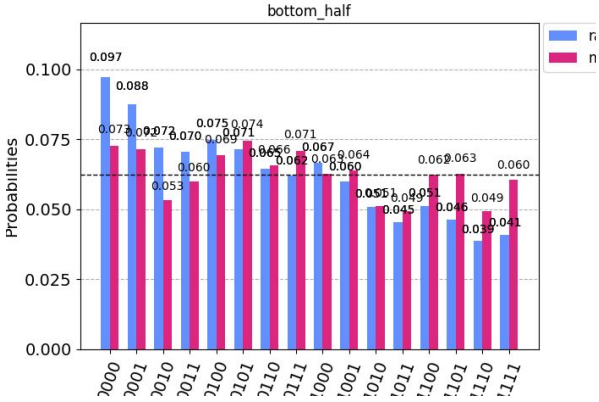
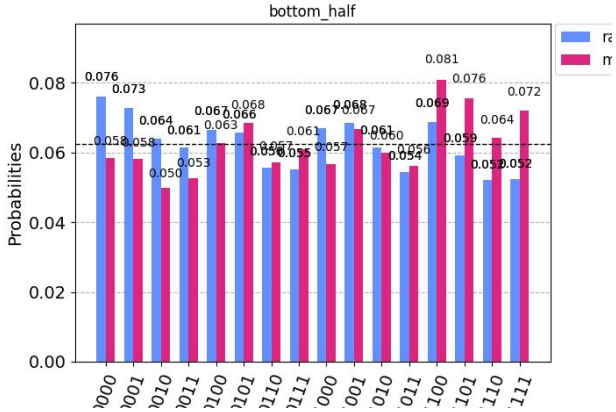
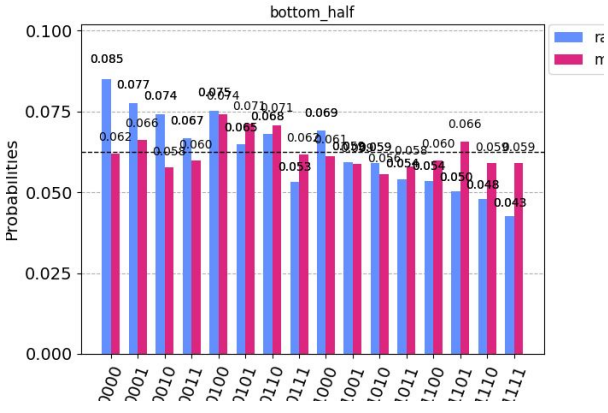
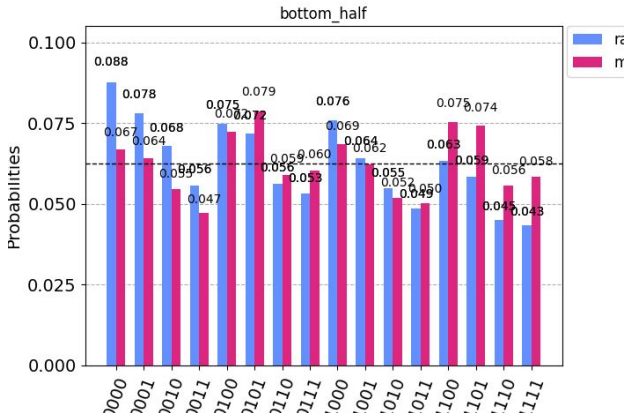
Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

function: add_one, n: 4

Shots	Optimization Level = 1	Optimization Level = 3
1000		
8092		
25000		

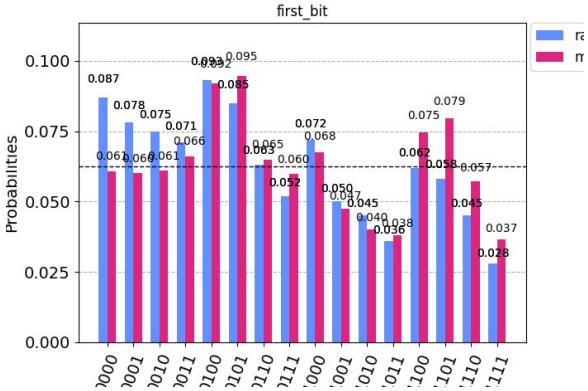
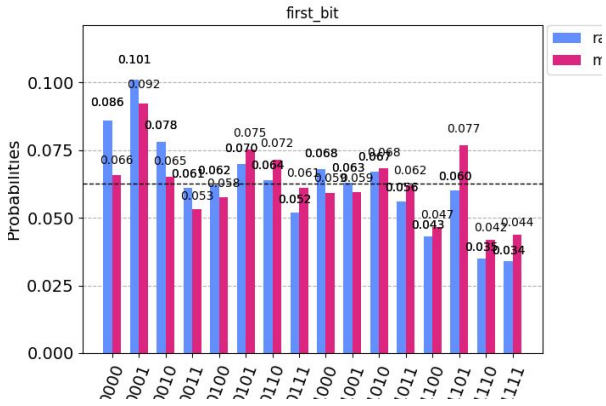
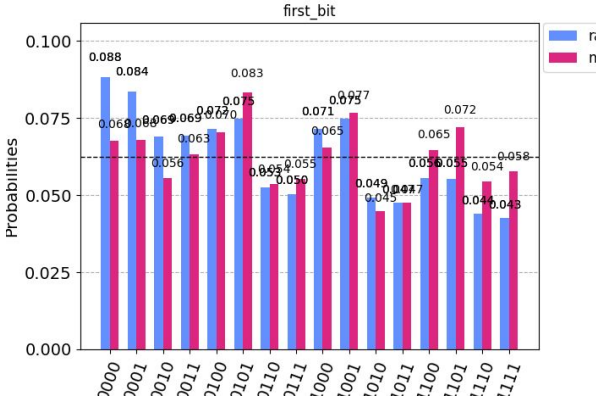
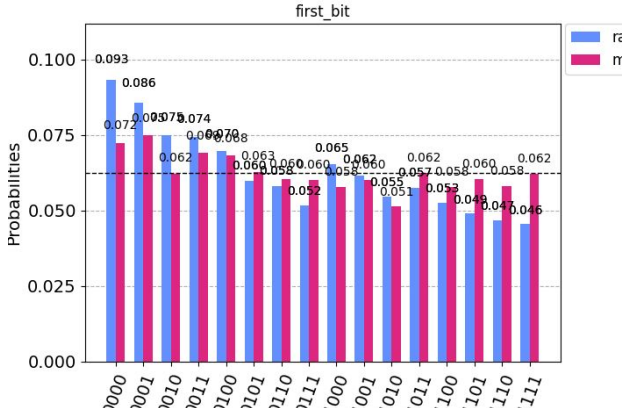
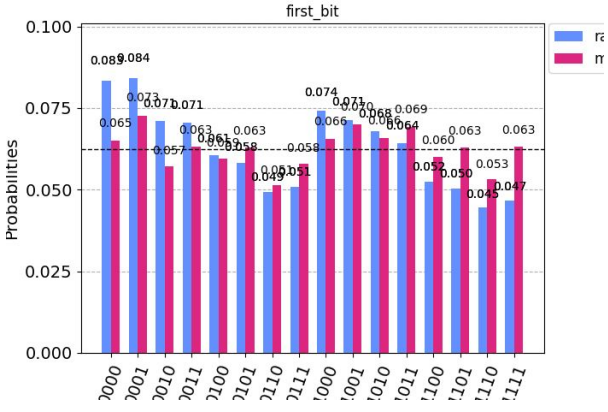
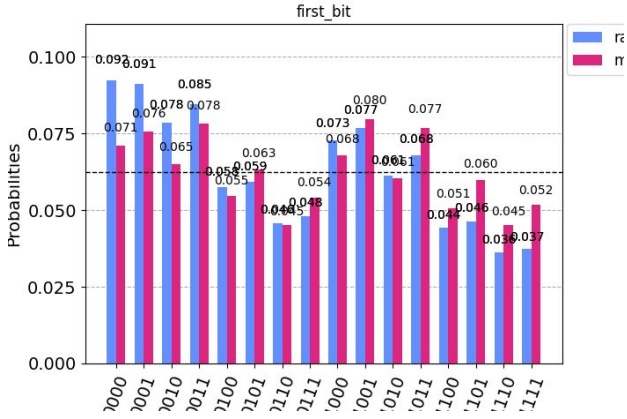
Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

function: bottom_half, n: 4

Shots	Optimization Level = 1	Optimization Level = 3
1000		
8092		
25000		

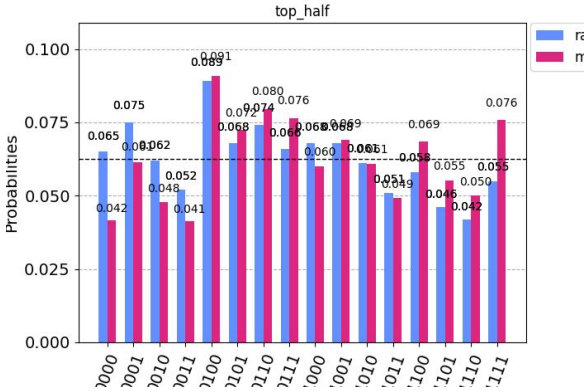
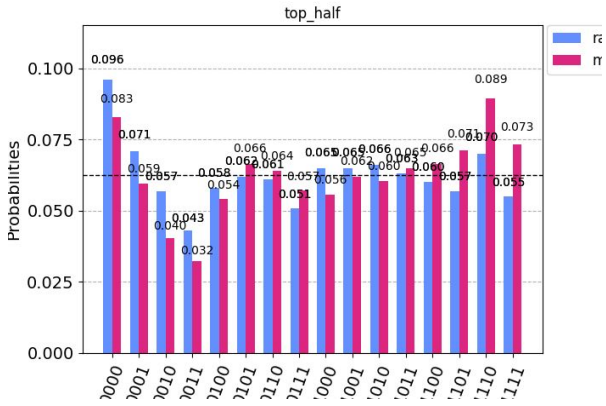
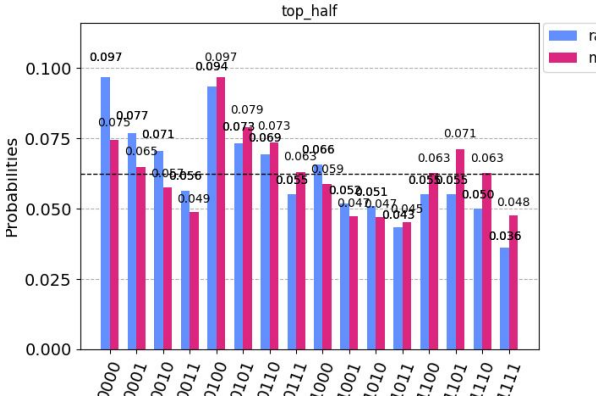
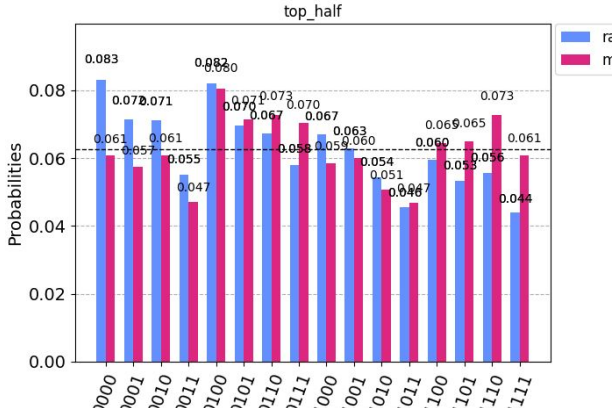
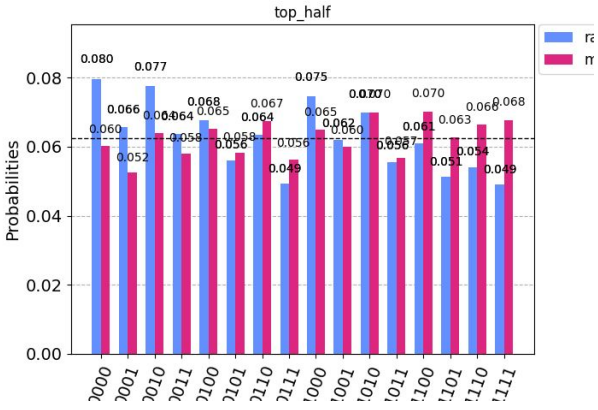
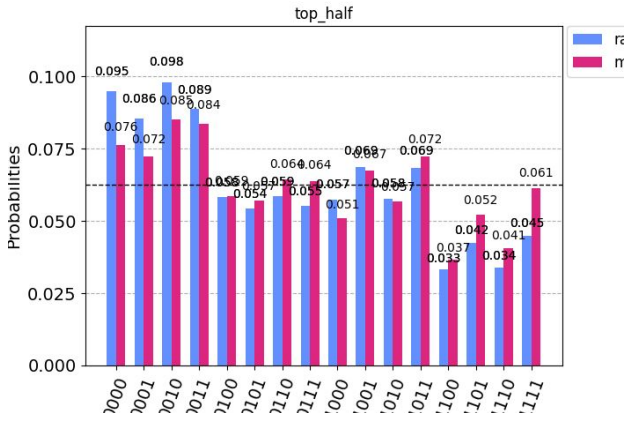
Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

function: first_bit, n: 4

Shots	Optimization Level = 1	Optimization Level = 3
1000		
8092		
25000		

Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

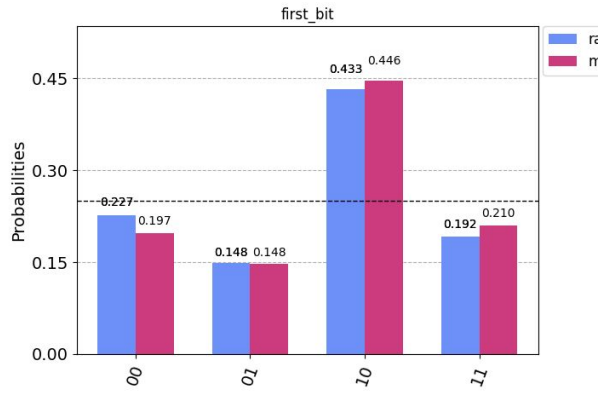
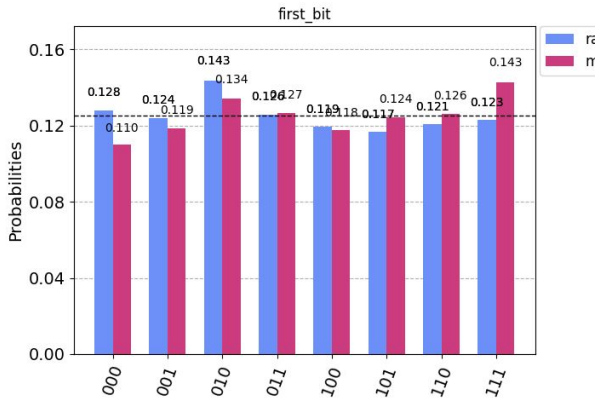
function: top_half, n: 4

Shots	Optimization Level = 1	Optimization Level = 3
1000		
8092		
25000		

Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

Again, we were surprised to see that even for a high number of trials and an optimization level of 3, the frequencies are almost uniform in distribution. Moreover, the correct result almost never has the highest frequency. As with Deutsch-Jozsa, we employed measurement calibration to mitigate measurement errors. The results are biased towards zero, similar to Deutsch-Jozsa.

The graphs below highlight the drastic difference in accuracy for Bernstein-Vazarani when the qubit value increases from 2 to 3. When n = 2, the correct result has a significantly higher frequency than the remaining output values. Furthermore, this high frequency is correctly further enhanced by error mitigation, which substantiated that our error mitigation was working correctly. When n = 3, the results form an almost even distribution, and exhibit this same pattern at higher qubit values. Apart from the necessary backend changes, we used the same U_f and circuit implementations for our algorithms as we did in the last Qiskit project, for which the simulator yielded the expected results. Therefore, we concluded the even distributions are a consequence of decoherence-induced errors in the quantum computer's execution.

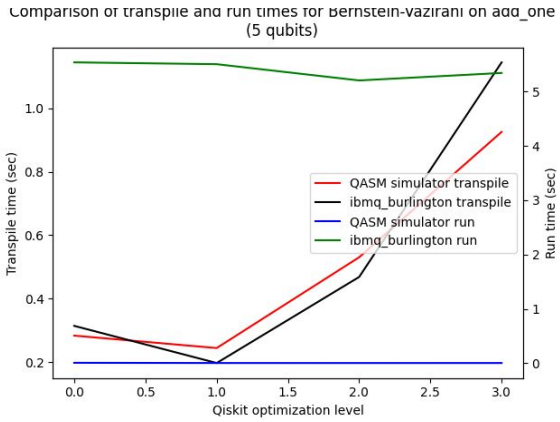
n = 2	n = 3
	

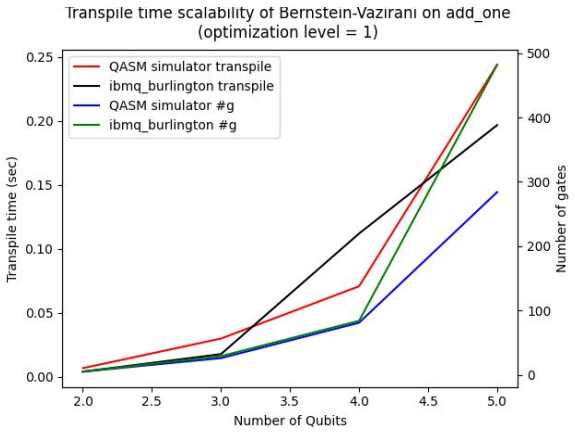
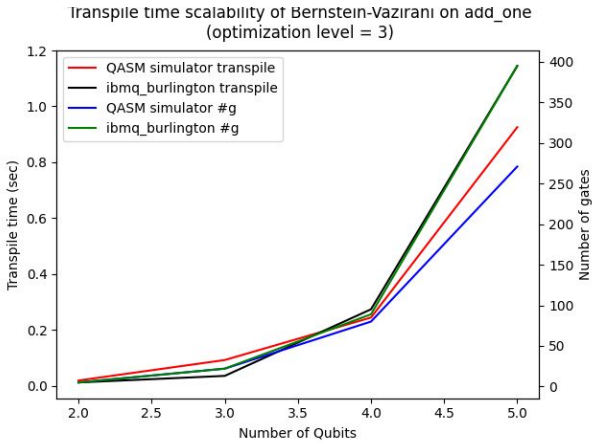
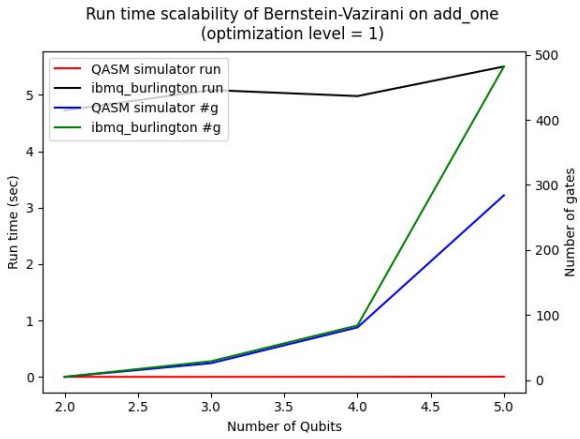
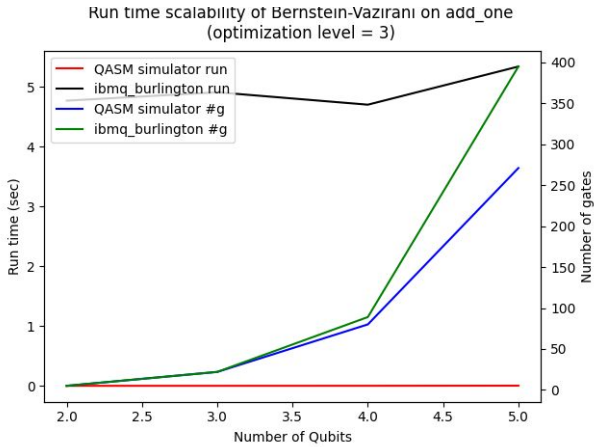
Execution Times

We use the same methodology as for Deutsch-Jozsa to compute transpilation and execution times. Presented below are plots comparing the transpilation and execution times for running bernstein_vazarani.py with 4 qubits 1-shot with all 4 optimization levels, on the four aforementioned test functions.

Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

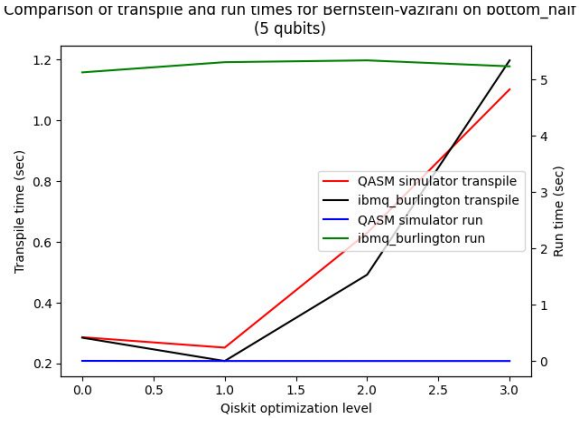
function: add_one, n: 4

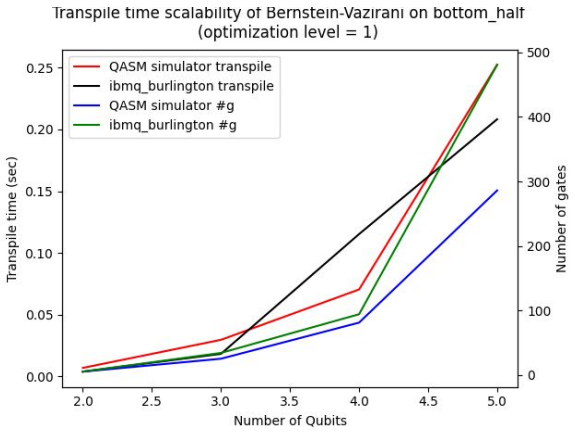
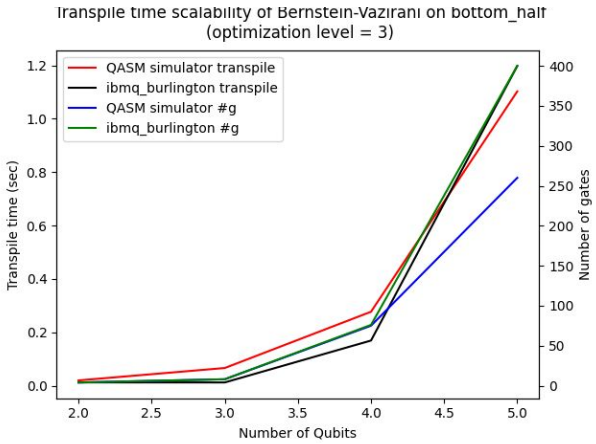
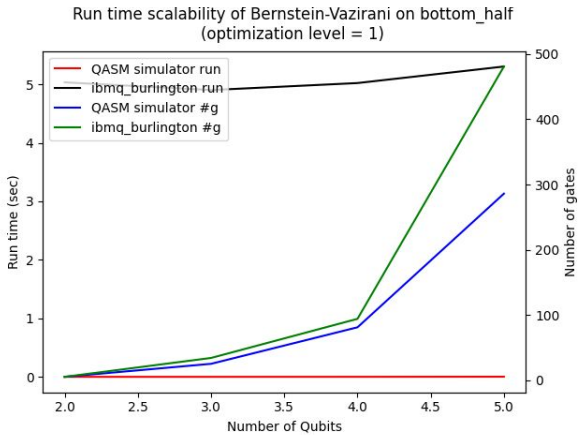
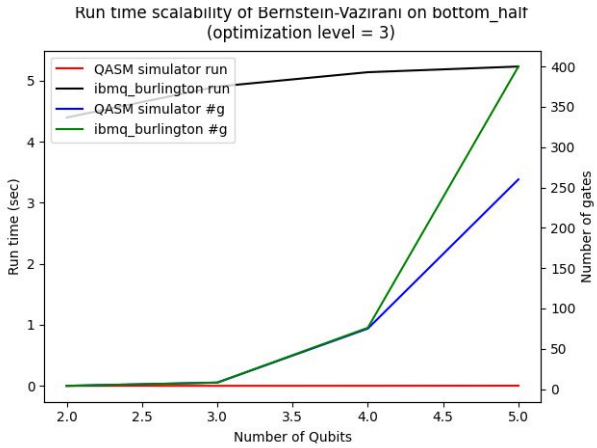


	Optimization Level = 1	Optimization Level = 3
Transpile time Scalability		
Run time Scalability		

Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

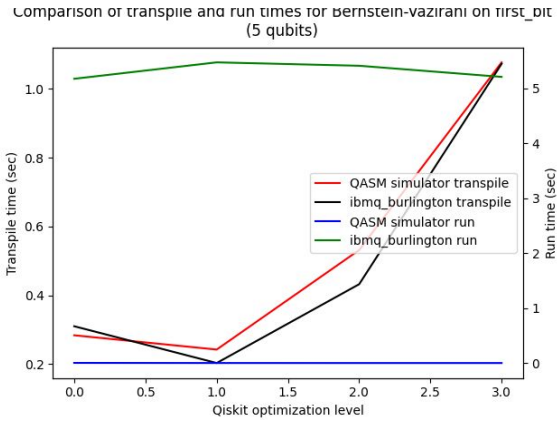
function: bottom_half, n: 4

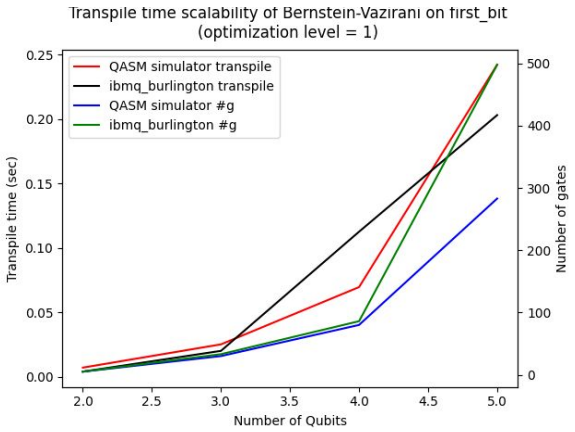
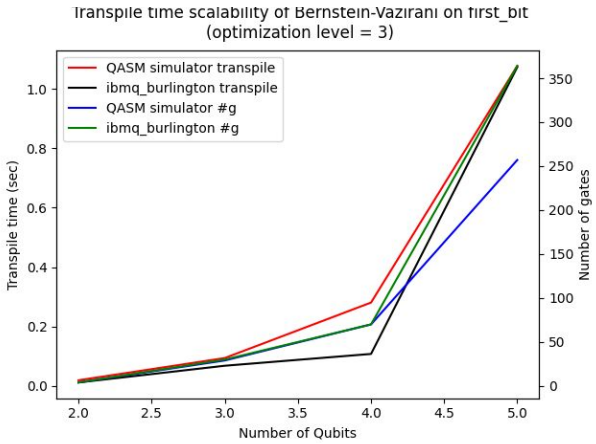
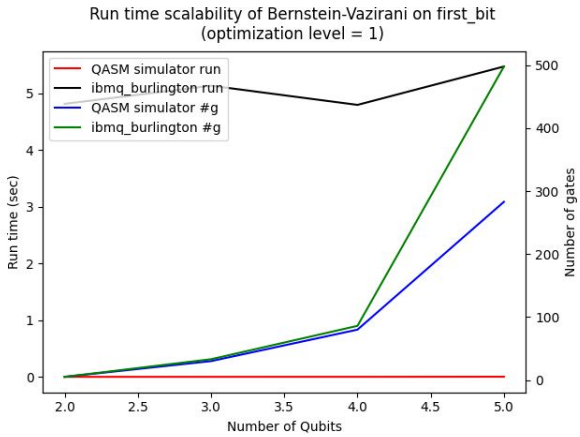
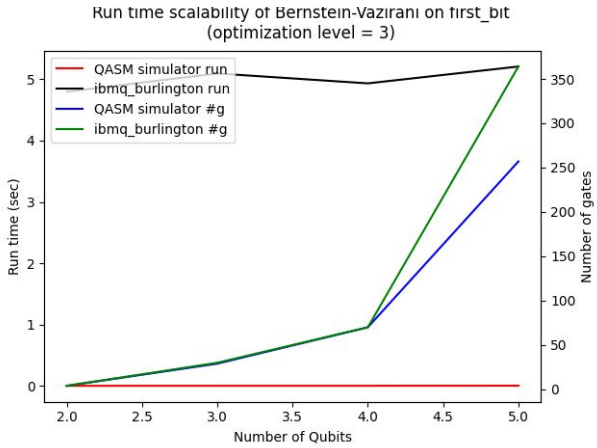


	Optimization Level = 1	Optimization Level = 3
Transpile time Scalability		
Run time Scalability		

Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

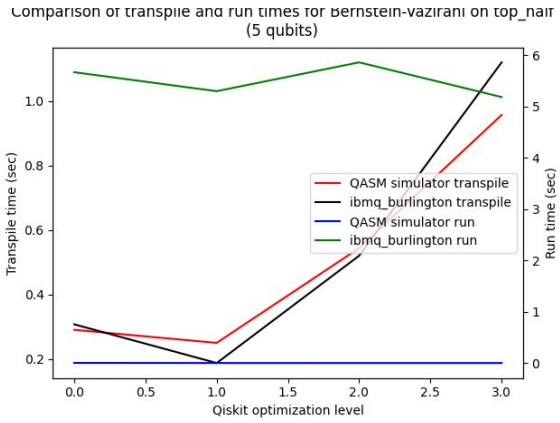
function: first_bit, n: 4

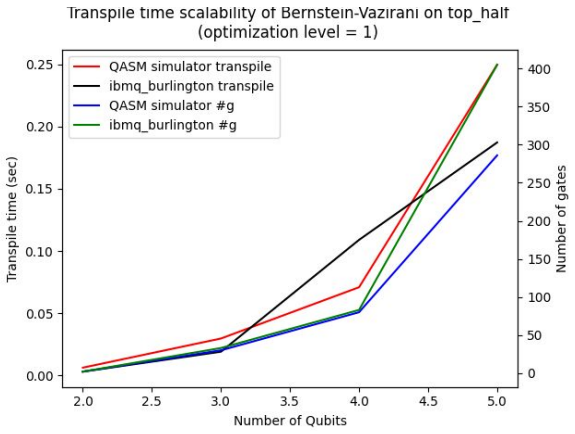
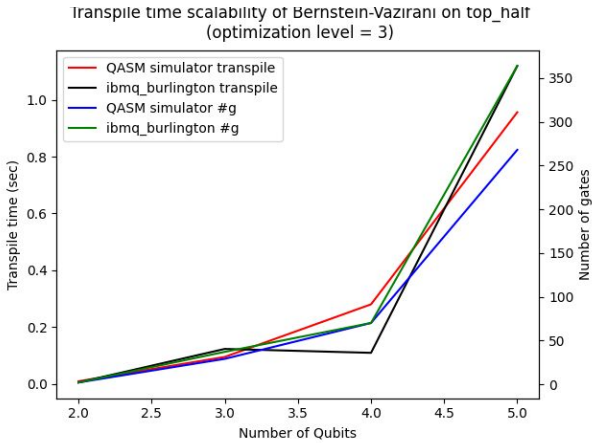
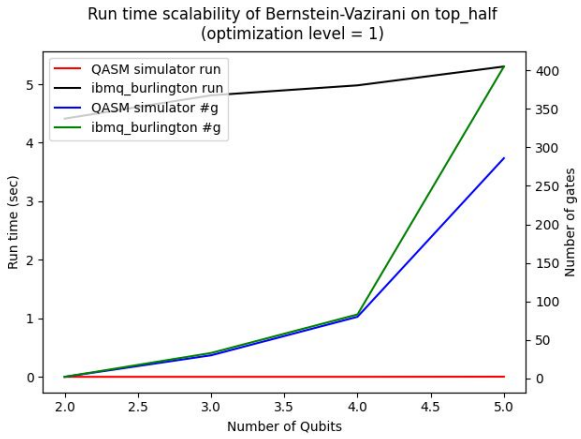
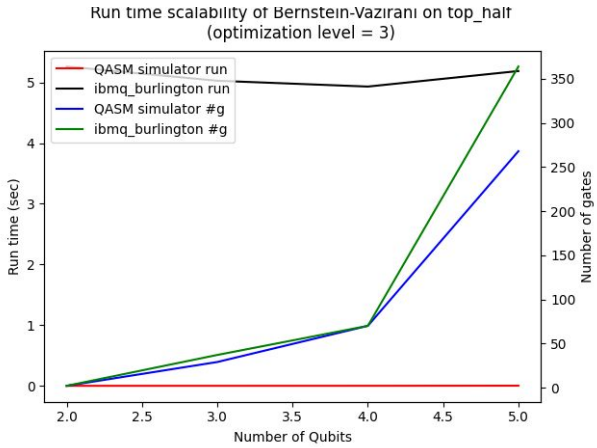


	Optimization Level = 1	Optimization Level = 3
Transpile time Scalability		
Run time Scalability		

Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

function: top_half, n: 4



	Optimization Level = 1	Optimization Level = 3
Transpile time Scalability		
Run time Scalability		

Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

For each function, as the optimization level increases, the transpiling time increases, as the compiler must do more work to more optimally decompose U_f and the other gates in the circuit into the basis gates, while the run time should decrease (obviously, a more optimized circuit with fewer gates will run more quickly). All test functions exhibited relatively similar transpile and run times for each optimization level, with add_one having the smallest transpile time and top_half having the largest transpile times when optimization_level = 3.

Scalability an n Grows

With regards to the scalability of n, for each test U_f, regardless of the optimization level, the transpile time appears to be exponential in number of qubits; this makes sense since, as we discussed in class, transpiling is NP-complete and hence the compiler will have an exponentially harder time transpiling larger matrices whose dimension grows exponentially in the number of qubits. Additionally, regardless of the optimization level, the run time seems to be exponential in the number of qubits; this also makes sense as larger matrices whose dimension grows exponentially in the number of qubits will likely be transpiled into more basis gates.

Furthermore, higher optimization levels generate circuits with a shorter run time, at the expense of longer transpilation time. Lastly, as we stated earlier, even though accuracy of results remains an issue with a large number of qubits, limiting the size of circuits that can be run on real quantum devices is a huge hindrance to scalability as *n* grows.

Simulator and Quantum Computer Comparison

Our simulator vs. quantum computer comparison for Bernstein-Vazarani is nearly identical to that for Deutsch-Jozsa. This is because the algorithms have a similar structure (with the primary differences being the construction of U_f and interpretation of the qubit measurements).

Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

Grover (grover.py)

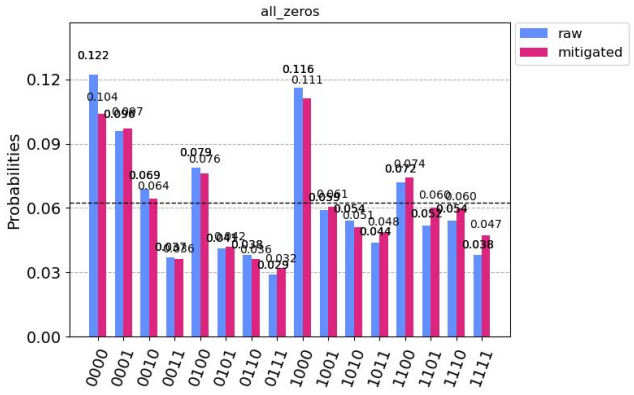
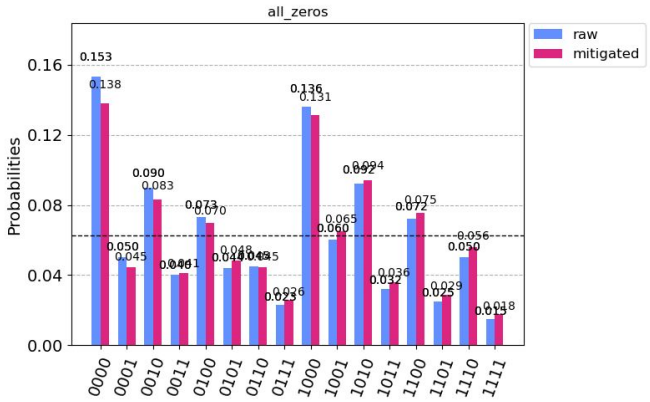
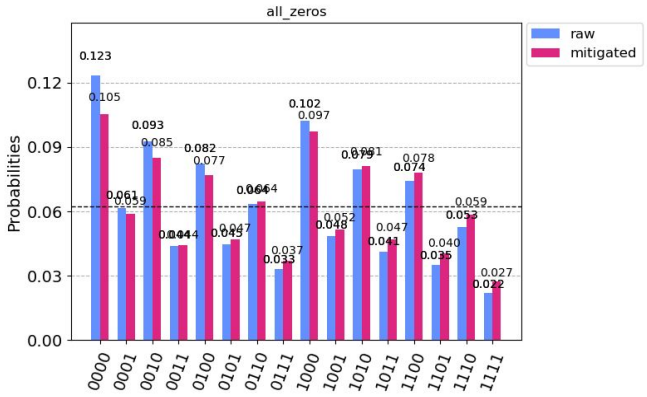
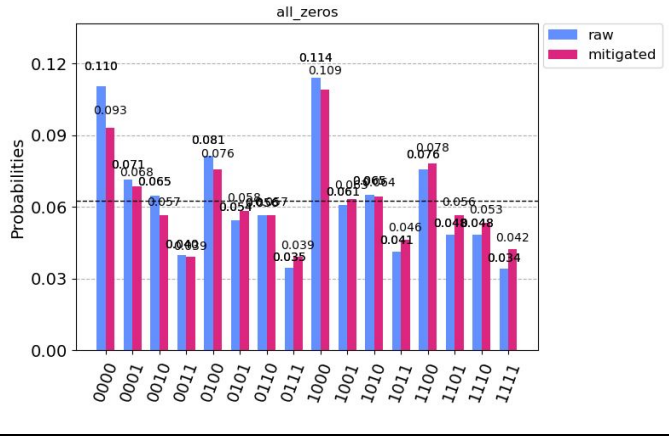
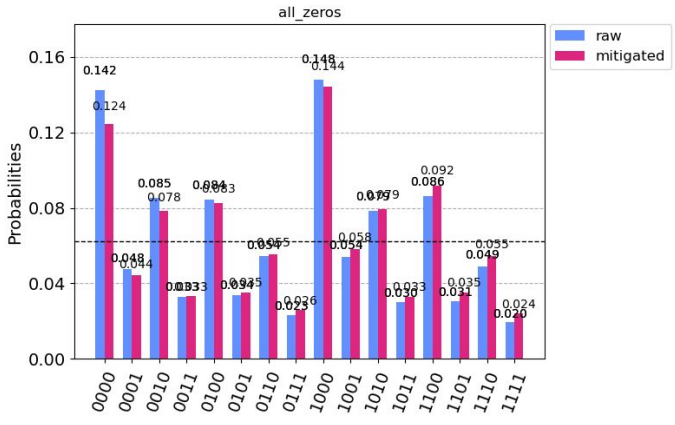
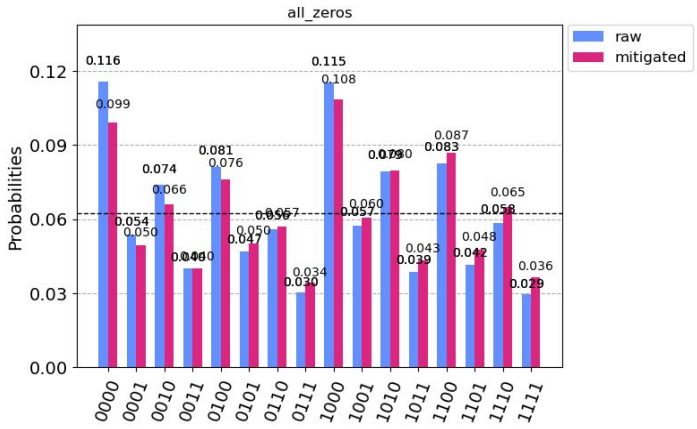
Statistics of Results

We tested our Grover implementation on the functions constant 0, All 0's (evaluates to 1 only when all input bits are 0), All 1's (evaluates to 1 only when all input bits are 1), and XNOR. Since Grover is not a deterministic algorithm and real quantum devices are noisy, we set up our test driver to accept the number of trials to run as a command line argument. The presentation of the results is the same as for Deutsch-Jozsa. Presented below are measurement results for running grover.py with 4 qubits with optimization levels 1 and 3, shots in [1000, 8092, 25000], and 2 different functions:

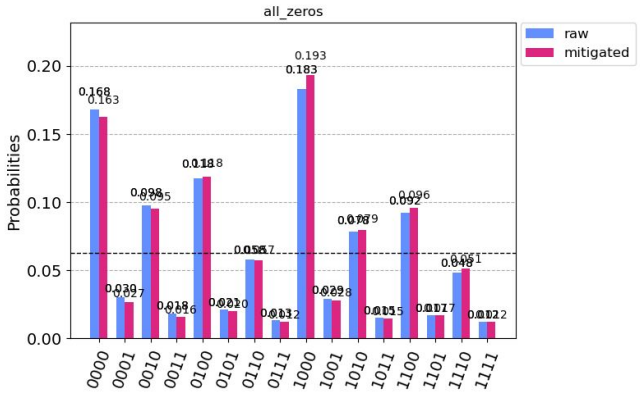
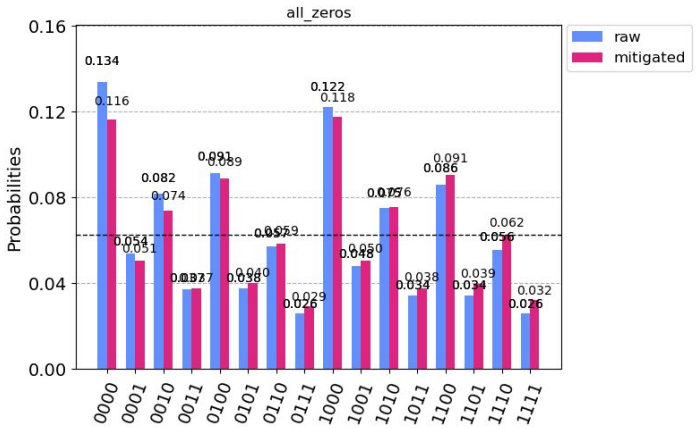
- all_zeros: f(x) = 1 when x consists of all zeros, f(x) = 0 otherwise
- zero: f(x) = 0 for any x

Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

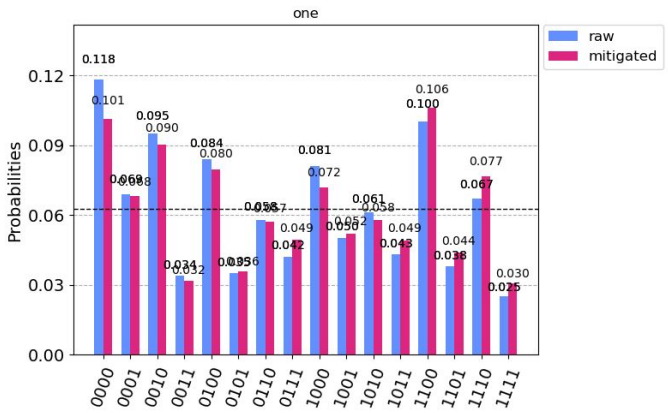
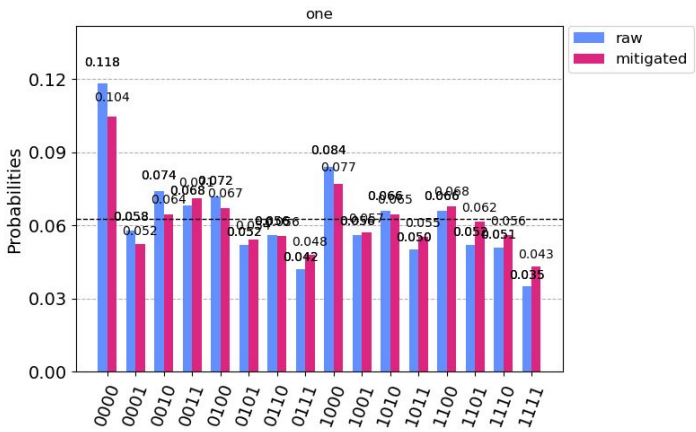
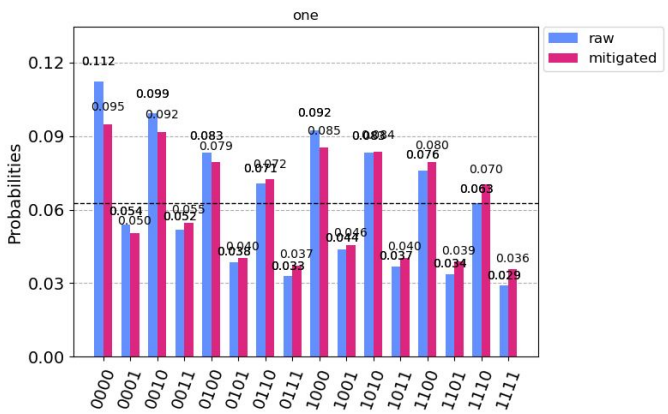
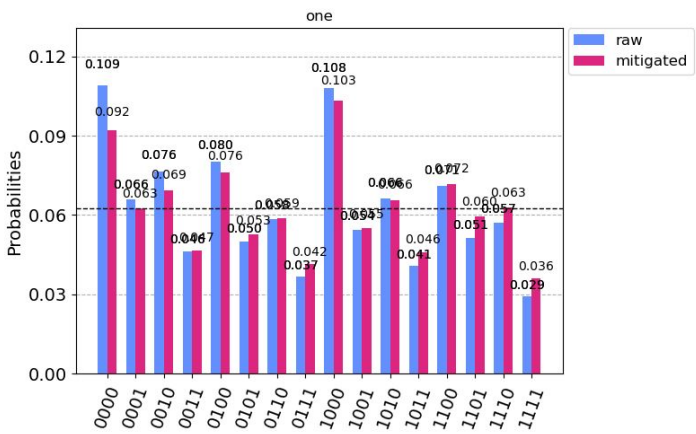
function: all_zeros, n: 4

Shots	Optimization Level = 1	Optimization Level = 3
1000		
8092		
15000		

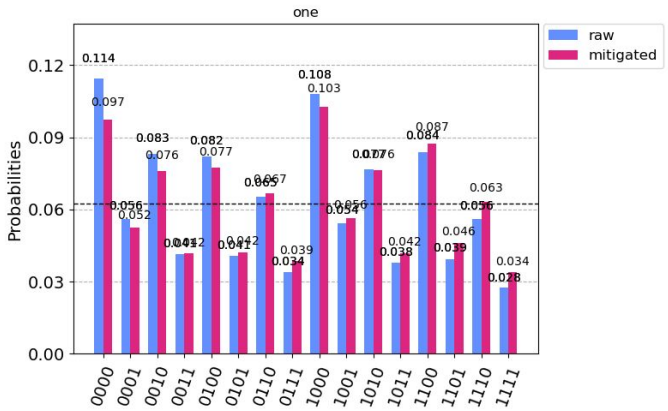
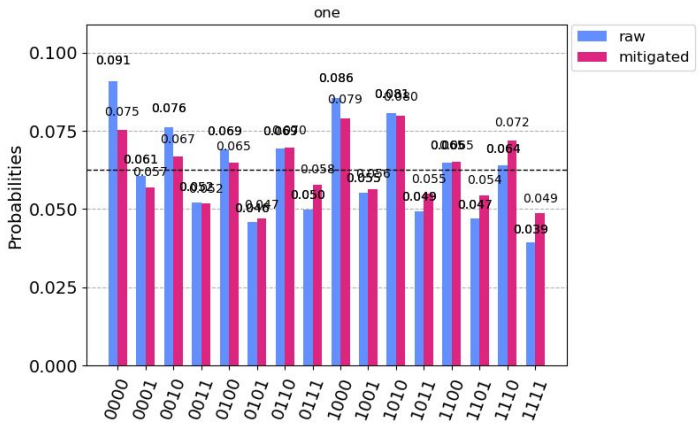
Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

30000		

function: zero, n: 4

Shots	Optimization Level = 1	Optimization Level = 3
1000		
8092		

Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

30000		

Compared to Deutsch-Jozsa, the results for Grover were much more indicative of the correct output. The results are still biased towards zero in the cases the expected output is ‘1000’ . This makes sense as measurements on IBM quantum machines are known to have state-dependent bias where ‘1’ is misread as ‘0’ more often than the other way around.

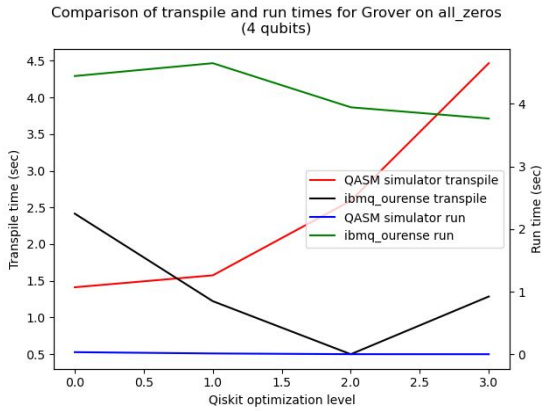
For the all_zeros function, at 30000 shots and optimization level 1, the mitigated results increase the probability of ‘1000’ (the expected result) and decrease the probability of ‘0000’, indicating that the error mitigation is helping clarify the correct result. Interestingly, the same trial run with an optimization level of 3 produced less accurate results, where the probabilities for ‘1000’ and ‘0000’ are almost the same. Higher optimization levels should mean fewer gates which should lead to fewer decoherence-induced errors, but our results suggest otherwise - we are not sure why.

Execution Times

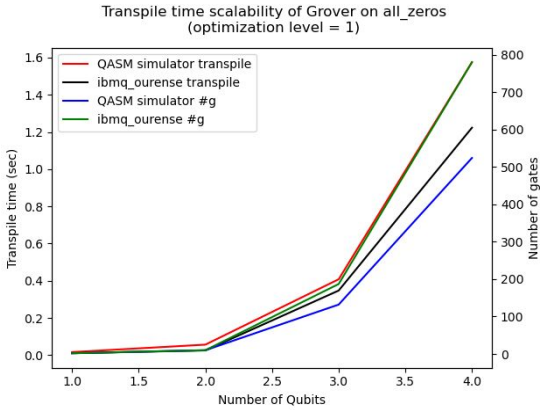
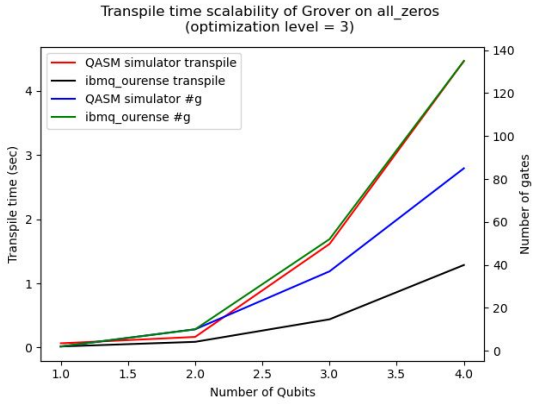
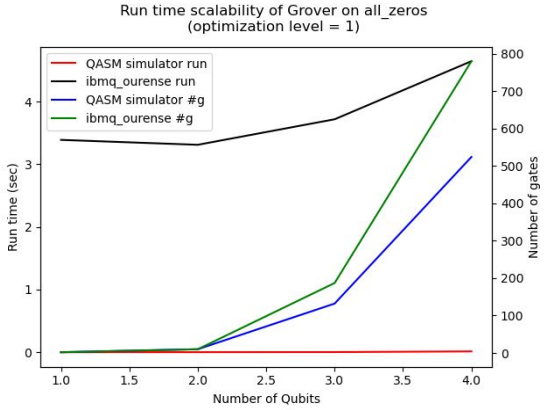
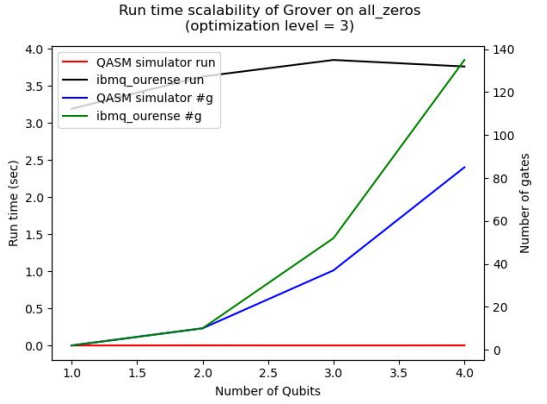
We use the same methodology as for Deutsch-Jozsa to compute transpilation and execution times. Presented below are plots comparing the transpilation and execution times for running grover.py with 4 qubits 1-shot with all 4 optimization levels, on the two aforementioned test functions.

Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

function: all_zeros, n: 4

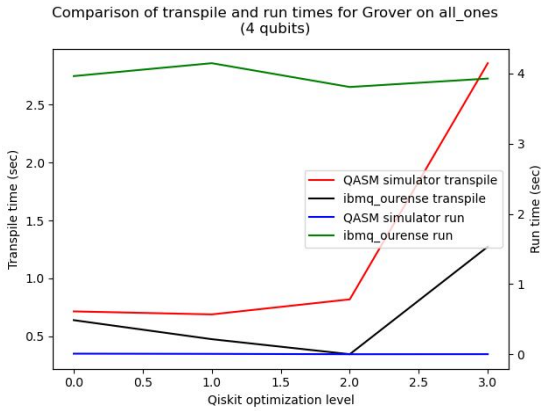
Comparison of Transpile Time and Run Time	

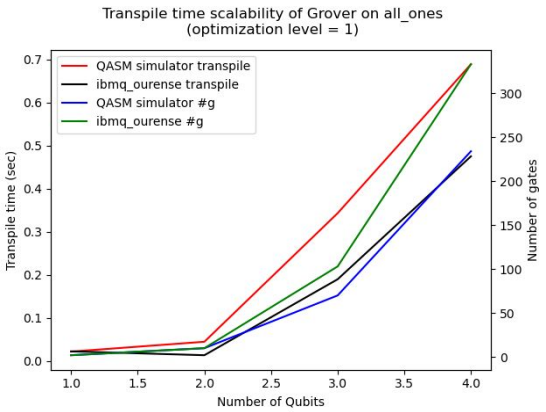
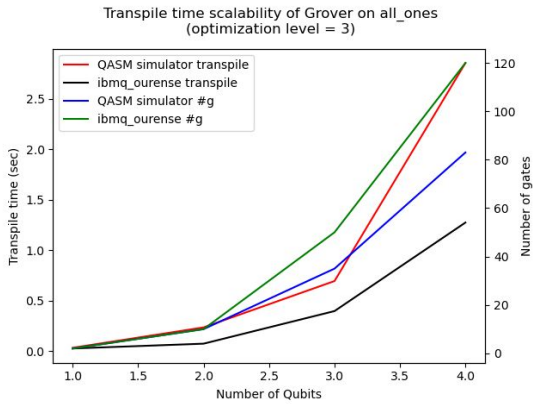
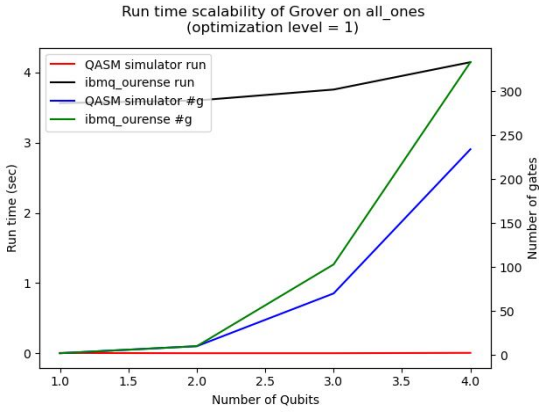
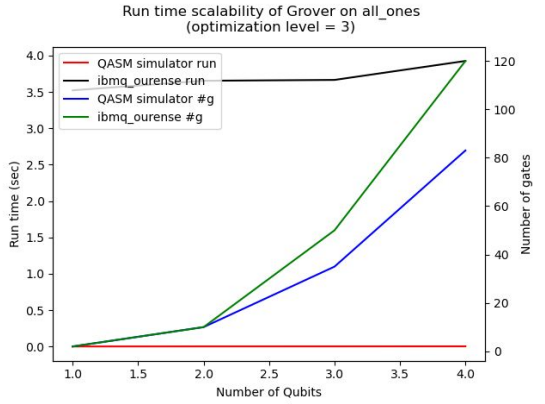
Note: the number of qubits n does not include the helper qubit

	Optimization Level = 1	Optimization Level = 3
Transpile Time Scalability		
Run Time Scalability		

Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

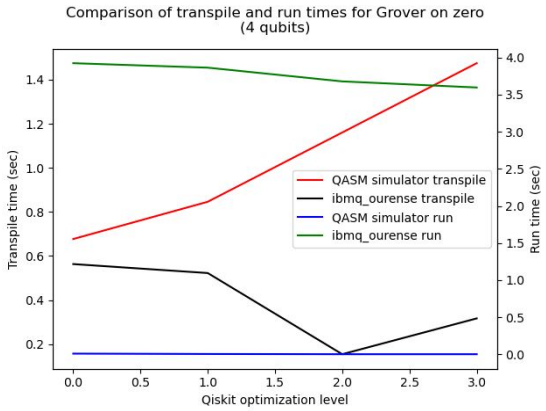
function: all_ones, n: 4

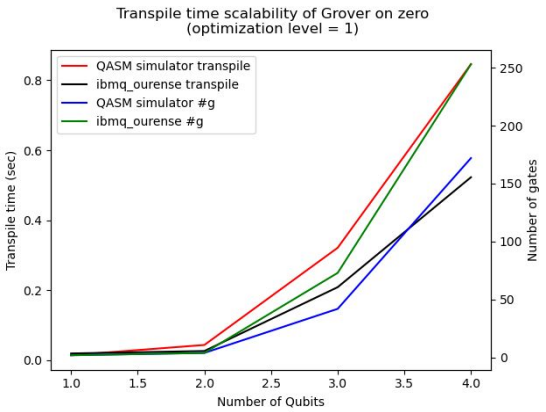
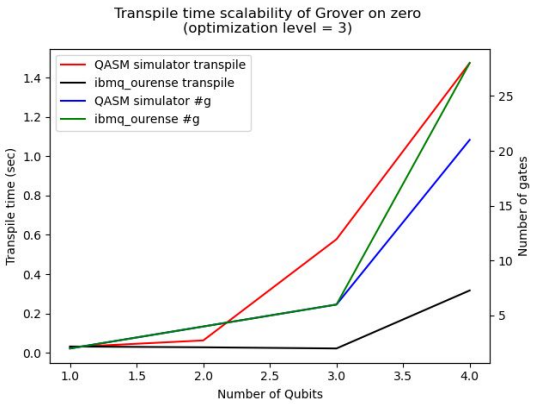
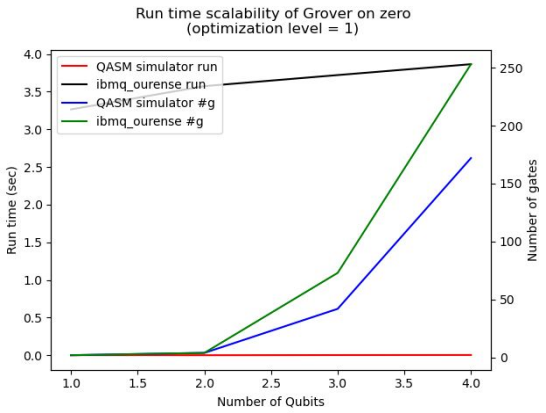
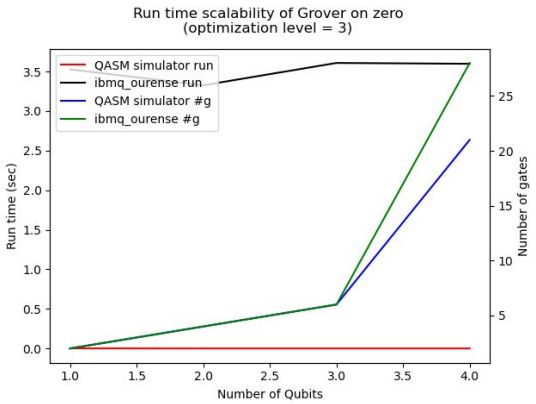
Comparison of Transpile Time and Run Time	

	Optimization Level = 1	Optimization Level = 3
Transpile Time Scalability		
Run Time Scalability		

Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

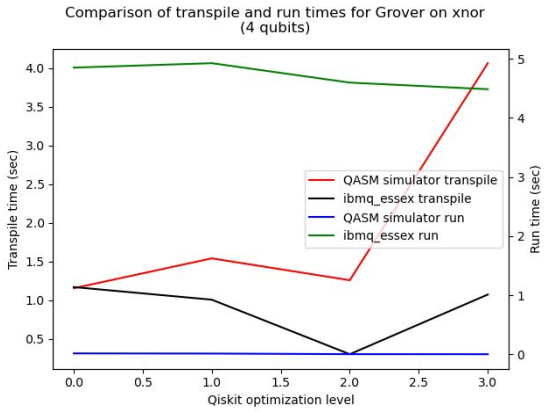
function: zero, n: 4

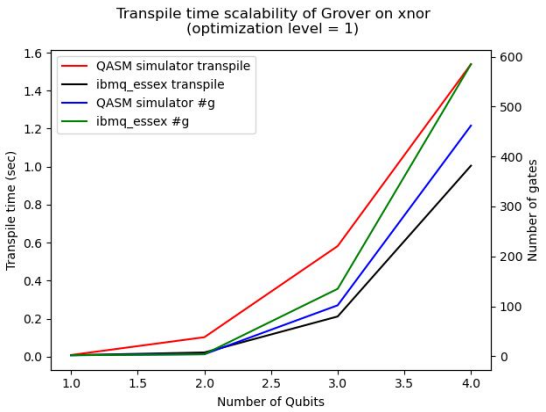
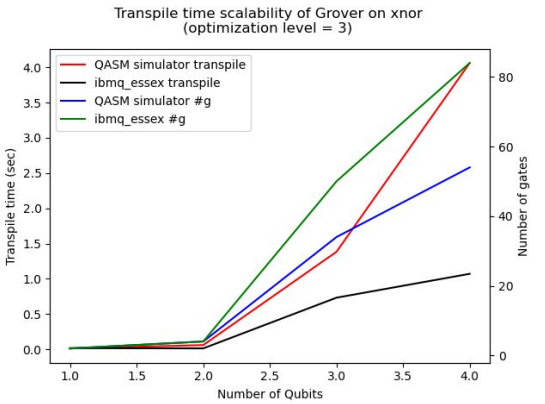
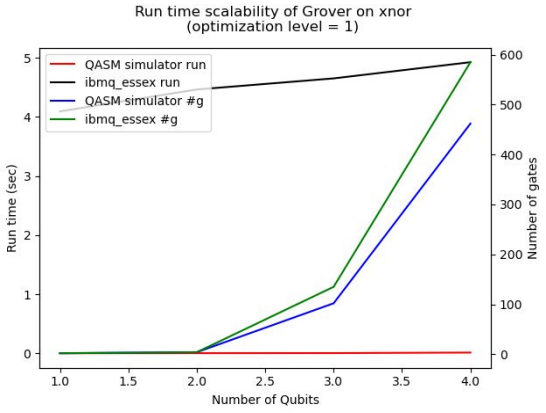
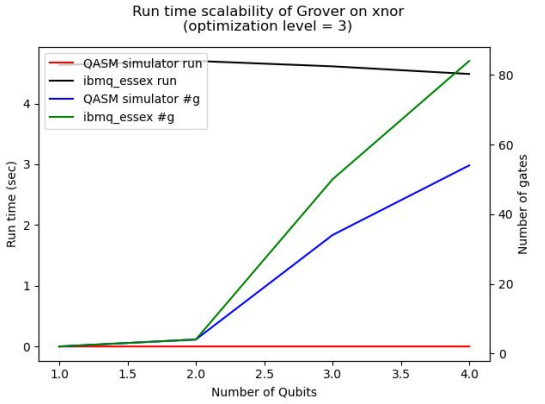
Comparison of Transpile Time and Run Time	

	Optimization Level = 1	Optimization Level = 3
Transpile Time Scalability		
Run Time Scalability		

Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

function: xnor, n: 4

Comparison of Transpile Time and Run Time	

	Optimization Level = 1	Optimization Level = 3
Transpile Time Scalability		
Run Time Scalability		

Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

Different cases of Z_f definitely lead to different execution times. As stated earlier, for each test, we compute the execution time including all the trials. This execution time does not include the construction of the black-boxes Z_f and Z_0 (since Grover assumes Z_f and Z_0 are available). We noticed that with 4 qubits, the constant zero function led to a faster execution time than for the other functions. This is likely due to the fact that the constant function makes it easy for the compiler to decompose Z_f into simpler gates, because Z_f for constant 0 is just the identity, while the other functions admit a Z_f that is more complex due to having -1 entries along their diagonal. The disparity in the execution time between the functions seems to grow exponentially as the number of qubits increases. Notably, the disparity is not due to the construction of Z_0 or the floor(pi/4 * sqrt(2^n)) number of iterations Grover requires because these are consistent across all the test functions in the 4-qubit case. Furthermore, compared to Deutsch-Jozsa, for the same number of qubits, Grover has much faster execution times since all the gates involved have diagonal matrices, which are much easier for the transpiler to decompose than matrices with off-diagonal non-zero entries.

Scalability as n Grows

With regards to the scalability of n, for each test Z_f, regardless of the optimization level, the transpile time appears to be exponential in number of qubits; this makes sense since, as we discussed in class, transpiling is NP-complete and hence the compiler will have an exponentially harder time transpiling larger matrices whose dimension grows exponentially in the number of qubits. Additionally, regardless of the optimization level, the run time seems to be exponential in the number of qubits; this also makes sense as larger matrices whose dimension grows exponentially in the number of qubits will likely be transpiled into more basis gates.

Furthermore, higher optimization levels generate circuits with a shorter run time, at the expense of longer transpilation time. Lastly, as we stated earlier, even though accuracy of results remains an issue with a large number of qubits, limiting the size of circuits that can be run on real quantum devices is a huge hindrance to scalability as *n* grows.

Simulator and Quantum Computer Comparison

As with Deutsch-Jozsa, the execution times for the quantum machine and the simulator increased as the number of qubits increased. The execution times for the simulator were almost zero while those of the quantum machine were much higher. Indeed, we get a longer execution time on the actual quantum computer than on the simulator because the problem scale is so small that local simulation is easy.

Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

Simon (simon.py)

We tested our Simon's implementation on the functions identity (s = 00...0) and two_to_one (s = 1010…). As with the other algorithms, we set up our test driver to accept the number of trials as a command line input. However, this was treated slightly differently for Simon's algorithm, in that we used it as the number of 4-cycle iterations of the equation generation process. Also in line with the implementation of the other algorithms, we transpiled all circuits before executing them. We allowed the user to provide the level of optimization they wanted to transpile the circuit at, with valid values being integers between 0 and 3, and the default being 1. Higher optimization levels meant more optimized circuits at the expense of a longer transpilation time.

For each run of the algorithm, we print out the total number of gates in the transpiled circuit, the predicted value of s, the time taken for transpilation, and the average time taken to run a single shot. Furthermore, we plotted a histogram showing the frequency of each of the observed quantum states, both as the raw data and the error-mitigated data, along with a dotted 'reference line' that showed the mean probability of a random state (assuming uniform distribution). Further discussion of error mitigation can be found in the **Experience** section.

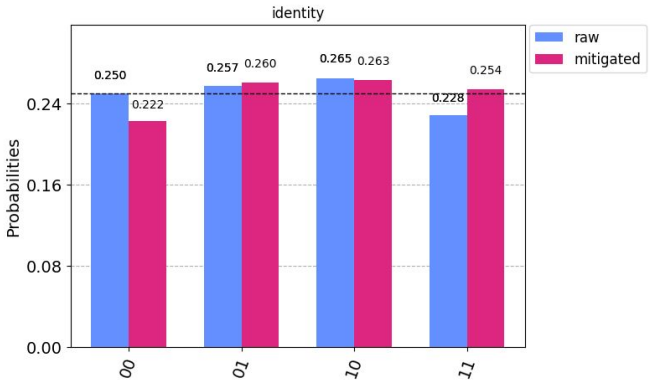
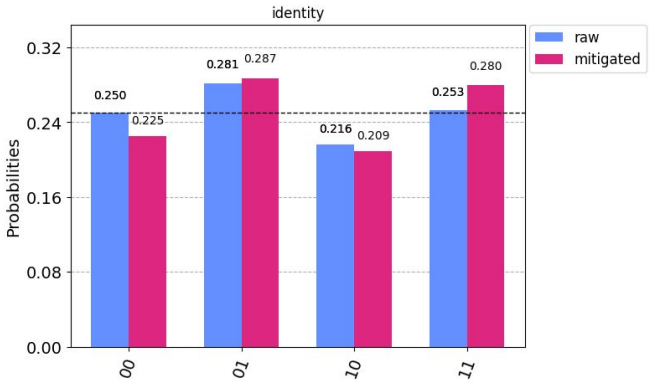
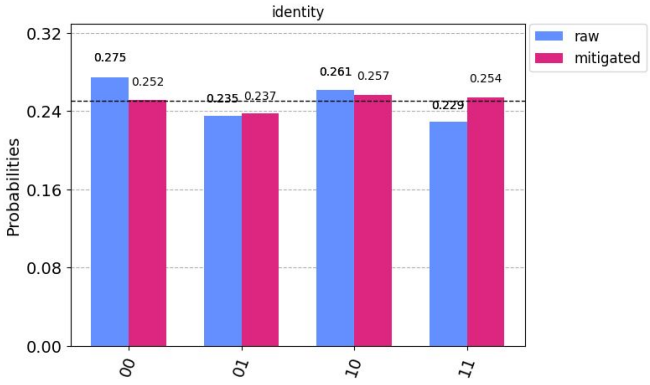
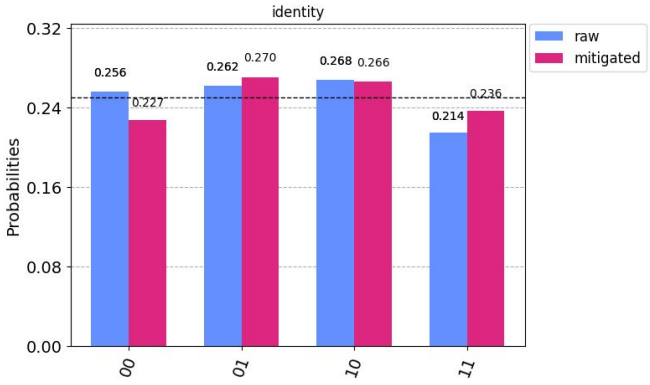
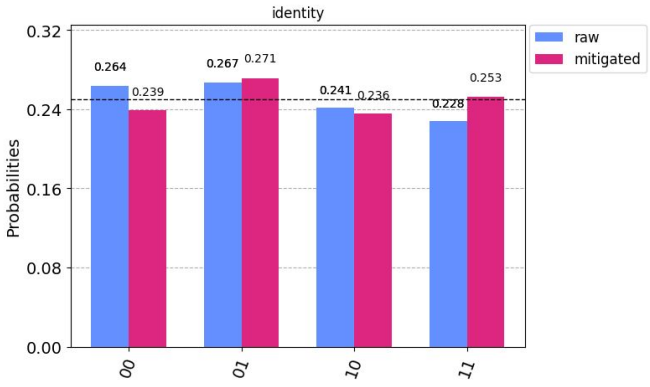
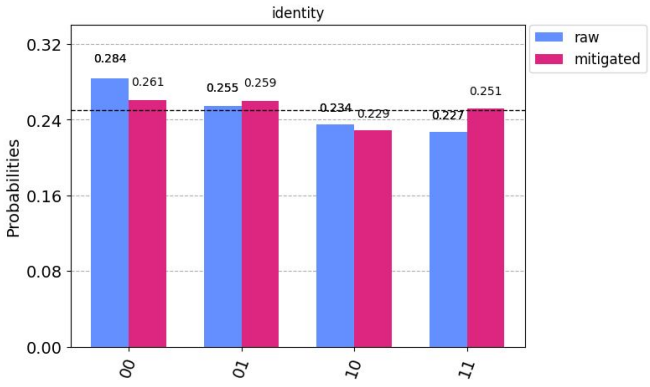
Presented in the next few pages are histograms of the qubit states we observed, as well as relevant graphs on the scalability of the transpilation and running of the algorithm as the number of qubits increases.

As an additional note, we were unable to run Simon's algorithm for more than 2 (non-helper) qubits as we got the following error: "Circuit runtime is greater than the device repetition rate [8020]". Therefore, all trials of the algorithm were run for bit strings of length 1 or 2 only. Furthermore, the 'number of qubits' on the axes in the plot is only the number of non-helper qubits, and the total number of qubits including the helper qubits would be double.

Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

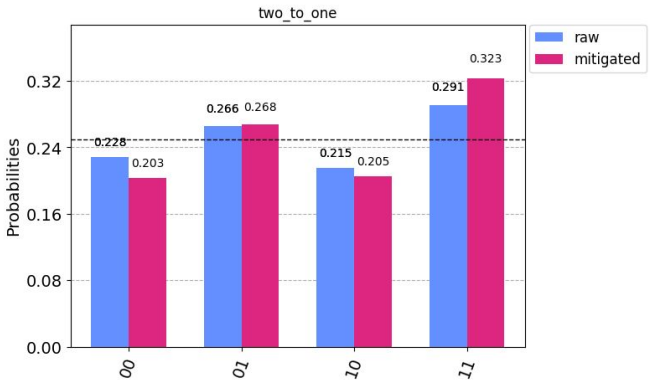
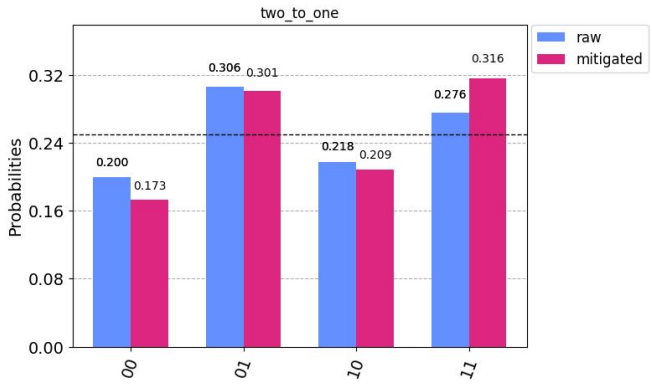
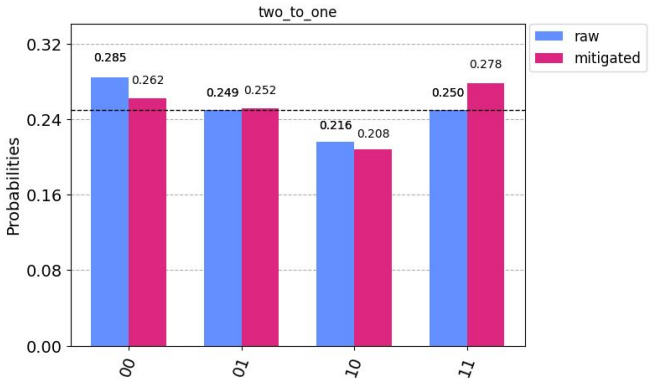
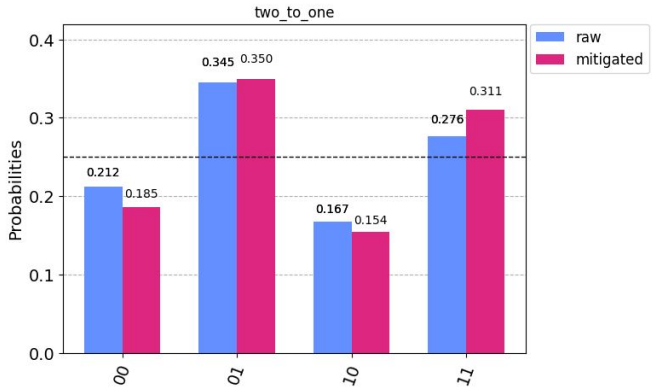
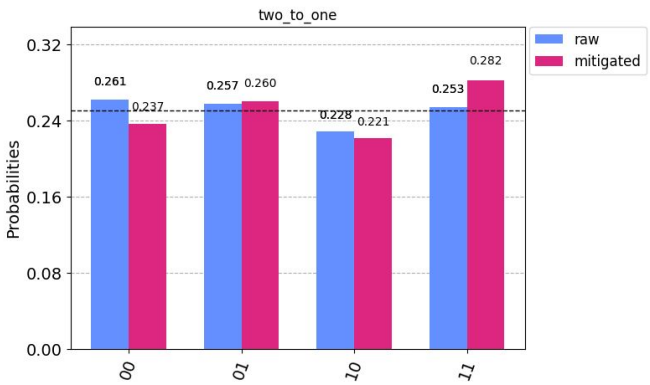
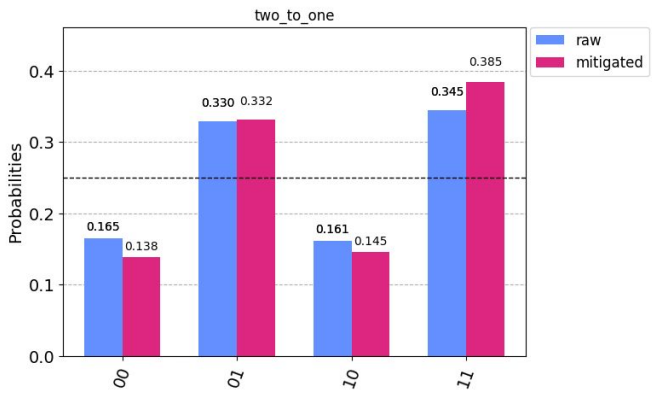
Statistics of Results

function: identity, n: 2 (s = 0 0)

# of Shots	Optimization Level = 1	Optimization Level = 3
1000		
8092		
30000		

Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

function: two_to_one, n: 2 (s = 1 0)

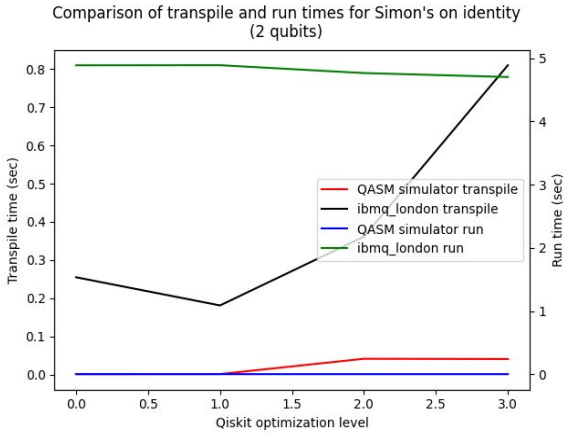
# of Shots	Optimization Level = 1	Optimization Level = 3
1000		
8092		
30000		

Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

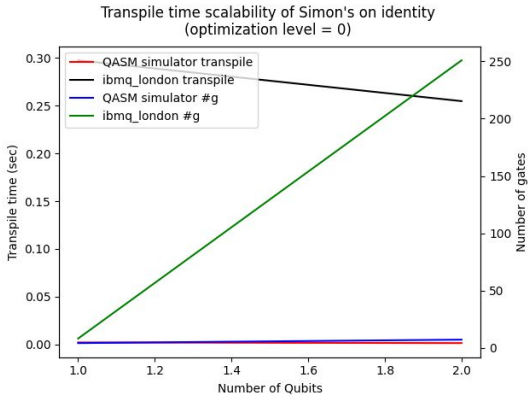
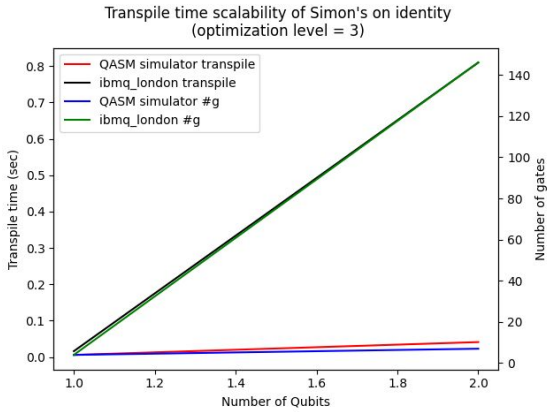
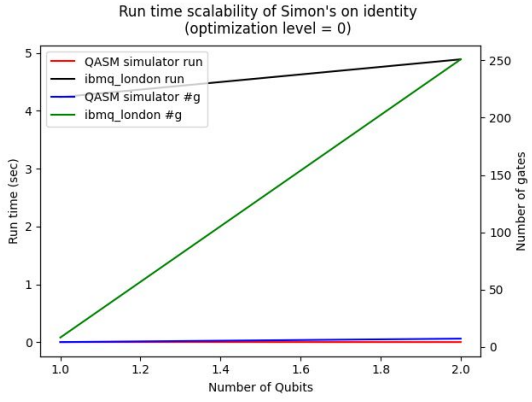
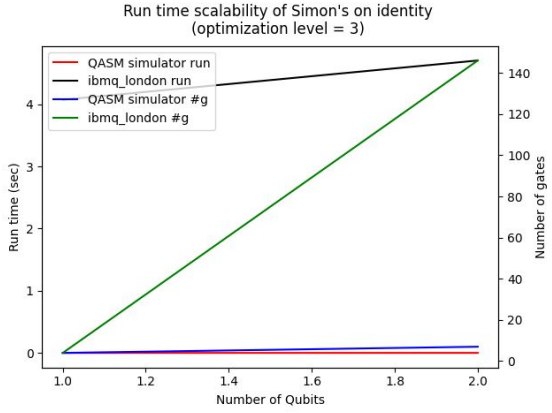
Transpile Time and Run Time Scalability

function: identity

Comparison of transpile time and run time for n = 2:



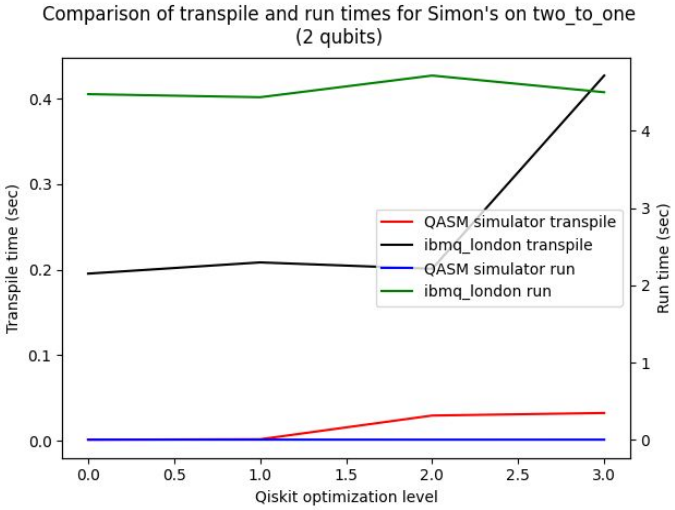
Note: the number of qubits n does not include the n helper qubit

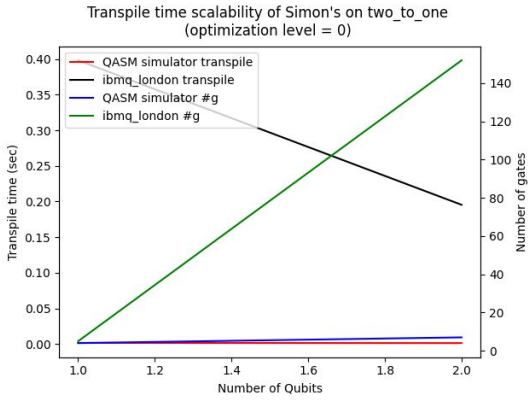
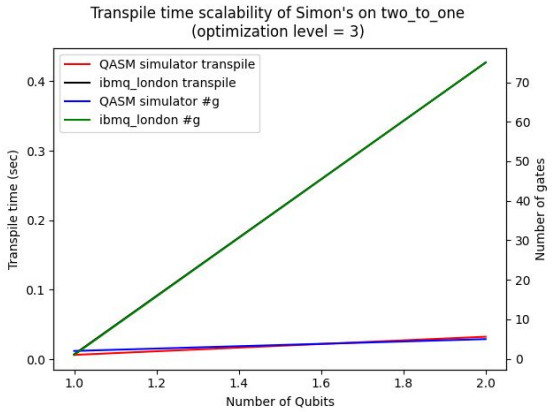
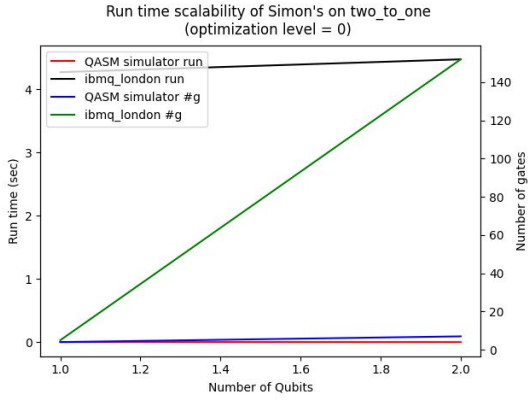
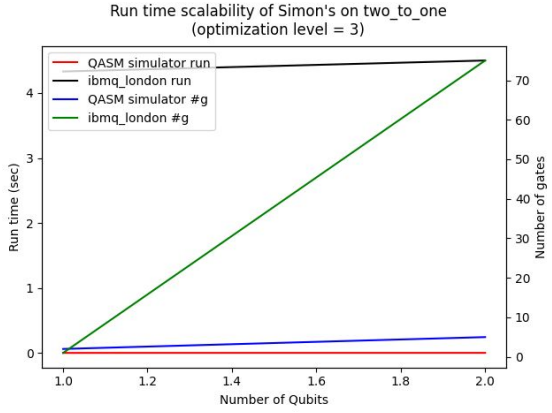
	Optimization Level = 0	Optimization Level = 3
Transpile Time Scalability		
Run Time Scalability		

Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

function: two_to_one

Comparison of transpile tim and run time for n = 2:



	Optimization Level = 0	Optimization Level = 3
Transpile Time Scalability		
Run Time Scalability		

Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

Analysis of execution times

We can see that the distributions for the qubit states when running the ‘identity’ function are more or less uniform. This is to be expected, as the identity function has an s value of all 0s, so the inner product of any 2-bit string with s will always be 0. Since we expect the bit states that satisfy this property to have an equal probability of being present, it is therefore completely reasonable that the distribution is uniform. Furthermore, we note that the error mitigation acts to boost the frequency of the ‘11’ state while suppressing that of the ‘00’ state. This is probably to counteract the natural bias towards the ground state that is present in IBM quantum computers.

However, it is not so expected that the distributions of the qubit states when running the ‘two_to_one’ function should also be uniform. This is because the value of s in a 2-bit string is ‘1 0’, so the qubit states we expect to observe are ‘0 0’ and ‘0 1’. However, we notice two interesting things under the results of the two_to_one function. First, the distributions when the optimization level is 1 seem more or less uniform, which suggests a very high error rate in the quantum computer (much higher than that of the simulator). Second, and possibly more interesting, the increase in optimization level from 1 to 3 led to a less uniform distribution, but the most frequent results weren’t necessarily the correct ones. Specifically, we would expect to primarily find qubits in the ‘0 0’ and ‘0 1’ states, with relatively few in the ‘1 0’ and ‘1 1’ states which could be chalked up to error. However, we see that the actual distribution we observe has two clear frequent states, ‘0 1’ and ‘1 1’, with ‘0 0’ and ‘1 0’ being relatively less common. The presence of ‘0 1’ and the relative absence of ‘1 0’ is expected, but it is quite odd that ‘1 1’ is observed so frequently and ‘0 0’ so infrequently. We also felt uncomfortable attributing this phenomenon to the natural bias of the quantum computers because that should result in bias toward ‘0 0’ and away from ‘1 1’, not the other way around.

Looking at the scalability data for both of the functions, we see that there is a clear upward trend in the run time as the number of qubits increases. This is probably due to the increase in the number of gates in the transpiled circuit, as the graphs almost exactly overlap, suggesting a strong correlation. However, we are unable to discern the nature of this trend (linear vs polynomial vs exponential) because we only had 2 data points. An interesting phenomenon we observed was that the transpilation time actually decreased for both functions when increasing the number of qubits when the optimization level was 0. However, for all other optimization levels, we observed the expected increase in transpilation time.

We also noted that both the transpilation and run times were drastically lower on the simulator than on the quantum computer, which was expected. Furthermore, we noted that, although the transpile time seemed to increase exponentially as the optimization level increased, there was no significant change in the run time. The run times for both of the functions were more or less the same, whereas the transpilation time for the identity function was around double that of the two_to_one function.

Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

README

Open the file “func.py,” which is in the same directory as the quantum program files; it already comes with a lot of test functions. In this file, define the function(s) you want to input into any of the quantum algorithms. The function must take only one parameter, a list of bits, and it should return a value that aligns with the assumptions made by the algorithm you plan on using (e.g. for Grover, you should only return a single bit). Furthemore, the function may NOT make any assumptions about the length of the list of bits. This is an example:

```
# constant 1 function
def one(x):
        return 1
```

All quantum programs must be run from the command line, and they all take the same arguments. Ensure to appropriately replace ‘API_KEY’ at the top of each quantum program (i.e. in the line MY_API_KEY = 'API_KEY') prior to running. There are three options:

1) python3 quantum_program.py **fn_name** **n** **shots** **opt_level**

   where **fn_name** is the name of the function you want to input (exactly the same as in the definition in func.py), **n** is the number of bits in the bit string passed to the function, **shots** is the number of times you want to run the quantum circuit and measure the qubits, and **opt_level** is the Qiskit optimization level you want to use when transpiling the circuit into the basis gates available on an IBMQX5 (**opt_level = 1** by default). The Qiskit optimization levels are as follows:
   * 0: no optimization
   * 1: light optimization
   * 2: heavy optimization
   * 3: even heavier optimization

   This is an example for Deutsch-Jozsa:

   python3 deutsch_jozsa.py one 4 10 2

   Each script computes the measurement error mitigation calibration matrix, transpiles the quantum circuit for the specified function and number of qubits with the appropriate optimization level, and runs the circuit and measures the qubits for the specified number of trials. It then prints out the results; it clearly labels the test to which the trials correspond, the transpile time, the total number of gates in the transpiled circuit, and the average run time for each trial, and subsequently, in order of decreasing frequency of results, for each result, presents 1) the frequency of the result, 2) the error-mitigated

frequency of the result, 3) the actual measurements of the n input qubits, and 4) the interpretation of the measurements.

Example:

```
==================================

Test: first_bit
Transpile time: 0.8180689811706543 sec
Number of gates in transpiled circuit: 831
Run time: 0.007341574907302856 sec

==================================
==================================

==================================

Result 1
Frequency: 93
Mitigated frequency: 81.33267106025248

a is 0001
b is 0

==================================

==================================

Result 2
Frequency: 89
Mitigated frequency: 67.52753794991152

a is 0000
b is 0

==================================
```

Furthermore, a histogram of the raw and error-mitigated frequencies of the results is plotted and saved to the same directory in which the quantum program files are. The histogram contains a horizontal, black dashed line indicating the probability of random guessing a result given *n* (i.e. 1/2^*n*), to provide reference for the distribution relative to the uniform distribution.

Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

For simon.py, the trials option is used as m, which determines the number of iterations of the circuit to increase confidence in the results.

2) python3 quantum_program.py **fn_name --graph n_1 n_2 n_3…**

Alternatively, each script also runs the quantum circuit for the provided function specified by **fn_name** for the specified numbers of qubits **n_1, n_2, n_3…** and saves the following plots to the same directory in which the quantum program files are:

1) For each optimization level (0-3):
    a) A plot of transpile time and number of gates vs. number of qubits (for both a quantum device and the QASM simulator)
    b) A plot of run time and number of gates vs. number of qubits (for both a quantum device and the QASM simulator)
2) A plot comparing transpile time and run time vs. optimization level (on twin y-axes)

This is an example for Deutsch-Jozsa:

python3 deutsch_jozsa.py one --graph 1 2 4

For simon.py, we left the values for the number of qubits at (1, 2). This is because since Simon's algorithm needs 2*n qubits to work, it causes massive slowdown after 4 qubits. To compensate for the lack of a trials option, we set m=5, to ensure that the probability of failure is less than 1% (using the rule that $P(\text{failure}) < e^{-5}$)

Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

Experience

We found that the transition from using a simulator to using a real IBM quantum device was smooth and easy! Constructing U_f, building the circuit, and timing transpilation and execution (using result.time_taken) remained the same for all the algorithms. Below, we outline the major changes we had to make to our code.

1. **Accessing, Transpiling for, and Executing on IBMQ Provider Backend**
   Although the circuit building process did not change, we had to specify a real quantum device backend for which to transpile the circuit (i.e. decompose the circuit into the gates supported by the device: [id, u1, u2, u3, cx]) and on which to run the transpiled circuit. Fortunately, this simply meant passing in an IBMQ provider backend object instead of the QASM simulator backend to the qiskit.transpile and qiskit.execute functions.

   To set up the QASM simulator backend object, we simply had to include `simulator = Aer.get_backend('qasm_simulator')`. However, setting up an IBMQ provider backend object was more complicated. Most of the difficulty stemmed from being unable to easily find a tutorial online. Fortunately, a student linked this relevant notebook in a Piazza post. The set-up process involved:
   1. Saving the IBMQ account using our API key
   2. Loading the IBMQ provider
   3. Selecting a real quantum backend from the provider
      a. Thanks to another Piazza post, we used the filter functionality to programatically select feasible backends and then the qiskit.providers.ibmq.least_busy function to choose the least busy backend
      b. This saved us time searching for a backend with a small queue through the IBM QX portal
      c. However, we had to be careful to use the same backend for the scalability tests in the **Evaluation** section. This is important to minimize external factors beyond the variables we care about affecting the results of our experiments. For instance, different quantum devices could have different topologies, which could change the execution time, as that would affect how our program is compiled. Furthermore, there are many differences on the hardware level, e.g. measurement errors will be different on different backends. Hence, measurement error mitigation (which we discuss below) depends on the specific hardware being used.

2. **Jobs and Trials**
   An aspect of running on a real IBM quantum device that was hard to pick up was the asynchronous execution of jobs. Executing on the simulator was synchronous because the execution took place locally. However, executing on a real quantum backend

involved sending a message with the execution request to IBM servers and waiting for the job to run on a remote quantum device (sometimes on the other side of the world). Moreover, there were usually long job queues for the devices. The network latency and waiting queues necessitated asynchronous jobs.

To retrieve the result of a job, we had to use `backend.retrieve_job(job.job_id()).result()`, which polled the IBM servers for the completion of the job and shipped the results back to our program. One source of confusion was the difference between `result = job.result()` and `delayed_result = backend.retrieve_job(job.job_id()).result()`. We discovered via experimentation that `result` only contained an indeterminate number of preliminary results of the job while `delayed_result` contained all the results of the job (i.e. for all shots). We are unsure why the developers of Qiskit made a distinction between `result` and `delayed_result`.

A big issue we ran into was that real IBM quantum devices have a limit on the number of shots for which a circuit can be executed; for most of the devices available for free, 8192 was the cap. (Obviously, this was to ensure fair sharing of the device amongst multiple users.) Rather than constrain the users of our program to limit the number of shots desired to be less than or equal to 8192, we transparently overcame this problem by submitting multiple jobs! We divided any number of shots greater than 8192 into multiple jobs, each with at most 8192 shots. For instance, 15000 shots would be divided into one job with 8192 shots and another job with 6808 shots. This meant, however, that we had to retrieve and collect the delayed results of all jobs. Furthermore, we had to be careful not to submit too many jobs at once, as this would result in jobs being canceled by the IBM servers.

3. **Circuit Size**

   We decided to present the size (i.e. the number of gates) of the transpiled circuit as part of the results printed for the user. This is because the number of gates greatly affected the error in the results. Furthermore, a circuit that was too large could cause the job to terminate with the following error: “Circuit runtime is greater than the device repetition rate, exceeded maximum call stack size.” This error message was not helpful, and to make matters worse, a generic Qiskit error was returned to our program. After searching online, we diagnosed the problem to be too large of a circuit. The solutions are to increase the optimization level of transpilation to 3, reduce the number of qubits, and/or manually decompose U_f into the minimal number of basis gates available to the backend. Unfortunately, this error prevented us from doing the analysis in the **Evaluation** section with a larger number of qubits and a low level of optimization (e.g. 0 or 1). This also inspired us to include the number of gates on twin axes in the plots for transpile and run times vs. number of qubits generated using --graph.

Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

4. **Measurement Error Mitigation**

   We quickly discovered that the noise in real quantum devices is significantly higher than the noise in noisy quantum simulators. The decoherence in the real quantum devices produced a nearly uniform distribution of results over multiple shots for greater than 2 qubits, rather than a unimodal distribution with the mode at the correct result.

   To resolve this issue, we employed measurement calibration to mitigate measurement errors. The main idea is to prepare all 2^n basis input states and compute the probability of measuring counts in the other basis states. From these calibrations, it is possible to correct the average results of another experiment of interest. We found the measurement error mitigation tutorial extremely helpful! We followed the steps in the tutorial to compute the calibration matrix for the noisy quantum device once prior to executing all our shots and used least squares fitting via CompleteMeasFitter to calibrate the counts of the results returned by the job.

   Since we submitted multiple jobs in the case of more than 8192 shots, in our `print_results` functions we merged the dictionaries of counts and mitigated counts, summing up the counts and mitigated counts for each unique result across all dictionaries, and printed the results (with their corresponding frequency and mitigated frequency) in order of decreasing counts. Lastly, we took advantage of the qiskit.visualization.plot_histogram function to painlessly plot a histogram of the raw and mitigated counts of the results. (We really liked the functionality and ease of use of qiskit.visualization.plot_histogram.) We added a horizontal, black dashed line to the histogram indicating the probability of random guessing a result given *n* (i.e. 1/2^*n*), to provide reference for the distribution relative to the uniform distribution.