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

COM SCI 239 Algorithms in Qiskit Project Report

Design and Evaluation

Deutsch-Jozsa (deutsch_jozsa.py)

Design and Implementation of U_f

For Deutsch-Jozsa, implementing the black-box U_f proved to be tricky. Since U_f |x> |b> = |x> |b + f(x)>, where x represents the n input qubits and b is the helper qubit, we initially thought that the matrix representing U_f could be constructed as the simple tensor product of n identity gates and either the identity gate, if f(x) == 0, to preserve b, or the Pauli X gate, if f(x) == 1, to invert b. However, we quickly realized this was not the case since whether the helper qubit is preserved or flipped varies based on x.

Hence, to figure out a general form for U_f, we worked through the case of f(x) = XOR(x) for 2 input qubits and 1 helper qubit. By doing so, we devised an algorithm for constructing U_f. U_f is a 2^(n + 1) by 2^(n + 1) NumPy array (n + 1 because n input qubits and 1 helper qubit), and is initially all zeros. Using a loop, we iterate through each possible n-bit input x, and if f(x) == 0, then we preserve the state of the helper qubit b, otherwise, we invert the state of the helper qubit. In essence, the “diagonal” of U_f consists of 2 by 2 blocks that are either the identity or Pauli X; the i-th diagonal block corresponds to the i-th possible n-bit input x_i to f (going from {0}^n to {1}^n), and if f(x_i) == 0 we insert the identity into the i-th diagonal block, else if f(x_i) == 1, we insert Pauli X into the i-th diagonal block.

The implementation of U_f is pretty neat and easy to read; it clearly follows the logic described above. If we opened up the black-box U_f, it would also be neat and easy to read since all the non-zero entries in the matrix representing the unitary gate are either along or just off the diagonal. In our program, we call a function get_U_f, which takes as input the function f and n and executes the algorithm discussed to return the matrix representing U_f. We then pass this matrix into dj_program (the function that builds the Deutsch-Jozsa quantum circuit), which actually defines the gate U_f using the matrix and specifies it to be applied to the input and helper qubits, as shown below.

```
U_f_gate = Operator(U_f)
circuit.unitary(U_f_gate, range(n, -1, -1), label='U_f')
```

At first, we used `circuit.unitary(U_f_gate, range(n), label='U_f')`, which produced incorrect and variable results, despite Deutsch-Jozsa being deterministic on an ideal simulator. We decided to pretty print the circuit diagram to diagnose the issue and quickly realized that the qubit ordering convention in Qiskit was the reverse of the one we learned in

class (the helpfulness of visualizing the circuit led us to add the --draw option described in the **README**). In class, we used the qubit ordering convention that the leftmost qubit in Dirac notation (i.e. the most significant qubit) is the top qubit in the circuit and the rightmost qubit in Dirac notation (i.e. the least significant qubit) is the bottom qubit in the circuit. The implication of Qiskit's unconventional qubit ordering was that U_f, as we had constructed it, was no longer valid, as it had been for the PyQuil assignment. We initially considered re-designing how U_f is constructed but instead decided upon simply reversing the mapping of the qubits in the circuits to the inputs of U_f. This decision also necessitated the reversal of the mapping of qubits to classical registers during measurement: `circuit.measure(range(n), range(n - 1, -1, -1))`.

Preventing Access to U_f

With regards to preventing the user accessing the implementation of U_f, the user writes the classical implementation of f in func.py and inputs the name of f as a command line argument when running deutsch_jozsa.py. Our program checks that f's definition exists in func.py and that f only takes 1 argument (a bit string passed in as a Python list). We achieved this type-checking by using the built-in getattr function in Python, which returns a function pointer to the function with the specified name if it exists and otherwise throws an exception, and the Python function inspect.signature to count the number of parameters the function requires.

U_f is, then, constructed dynamically during runtime using get_U_f, which produces the matrix for the gate U_f corresponding to f. We chose to do this so that the user is forced to only interact with the classical parts of our program and because U_f is constructed dynamically during runtime, there is no way to access the implementation or entries of the matrix for U_f. Moreover, while our program provides the option to draw the circuit, we do NOT draw the transpiled circuit; the circuit visualization we print still represents U_f as a black box. However, because get_U_f is not encapsulated in a class, and furthermore, there is no private access modifier on get_U_f, the user could create their own script to call get_U_f and inspect the internals of the matrix returned.

Parametrization of Solution in n

The parameter n is passed into the program as a command line argument. During runtime, n is passed to get_U_f to dynamically construct the matrix for U_f with the appropriate dimensions (i.e. 2^(n + 1) by 2^(n + 1)). n is also passed to dj_program, where it governs the number of qubits to which Hadamard is applied at the beginning of the circuit and at the end and on which qubits U_f operates.

Testing Methodology and Presentation of Results

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

We tested our Deutsch-Jozsa implementation on the functions constant 0, constant 1, XOR, and XNOR (where the former two are constant and the latter two are balanced). Since the QASM simulator is a noisy (i.e. not ideal) quantum simulator by default, that is, it emulates the execution of quantum circuits on real quantum computers, 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 `['u1', 'u2', 'u3', 'cx']`, which, as we discussed in class, form the instruction set of the IBMQX5 quantum device on which we will likely run our quantum programs in the next couple of weeks. 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; we use nice formatting to clearly label the test to which the trials correspond and print, separately, the transpilation time 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 actual measurements of the n input qubits and 3) the interpretation of the measurements (i.e. “Constant” if all the qubits are measured to be zero and “Balanced” otherwise).

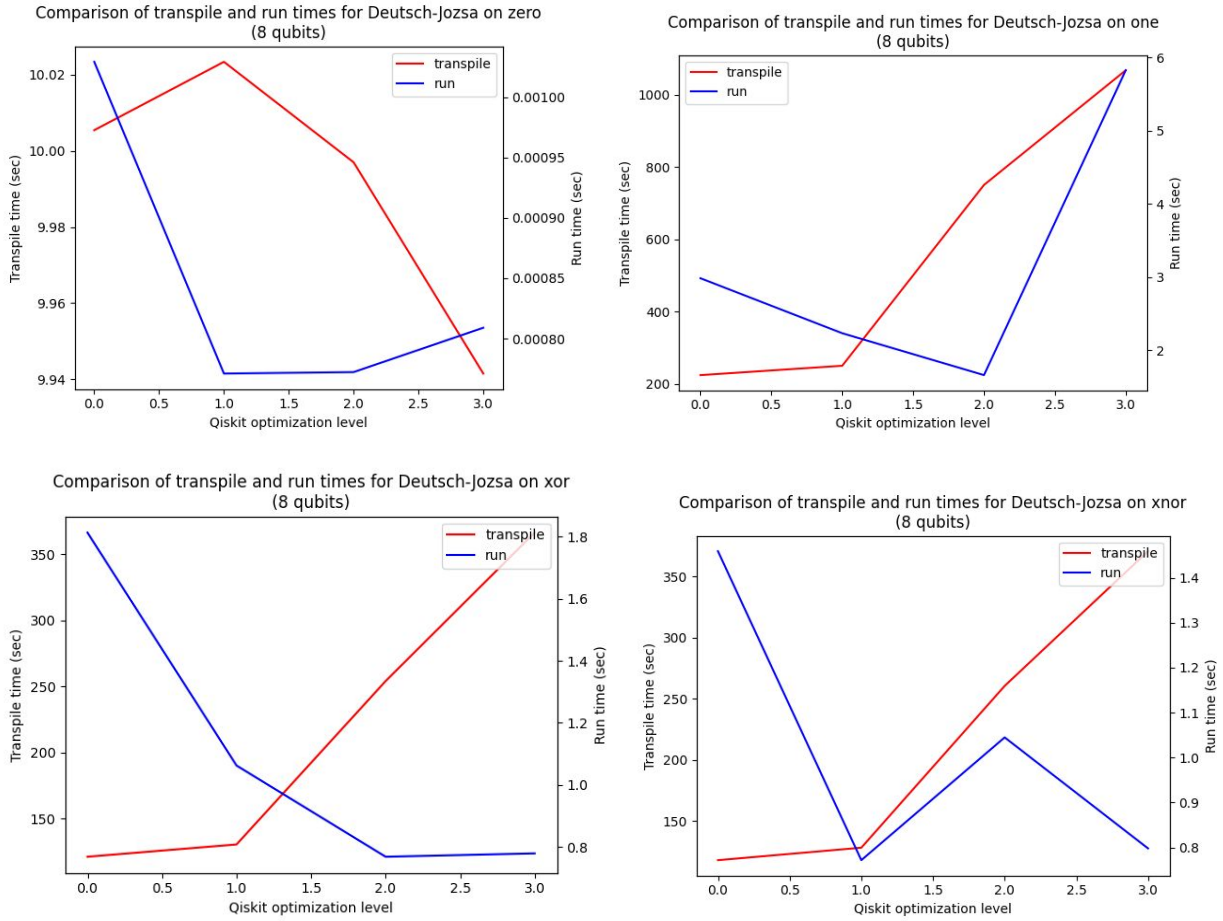
Execution Times

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 us 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 for running deutsch_jozsa.py with 8 qubits 1-shot with all 4 optimization levels, on 4 different functions (constant 0, constant 1, XOR, and XNOR).

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

Transpile Time and Run Time Comparisons



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.

We noticed that with 8 qubits, constant 0 led to a much faster transpilation time than XOR and XNOR, and constant 1 was noticeably slower than both. This is noticeably different than with PyQuil; for 4 qubits with PyQuil, constant 0 led to a much faster execution time than constant 1, and XOR and XNOR were noticeably slower than both. 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. We are unsure why the transpile time for constant 1 is so large, relatively.

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

One would expect, for each function, as the optimization level increases, the transpiling time should increase, 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). While this is generally the case with XOR and XNOR, it is not the case with constant 0 or constant 1. For constant 0, because the identity is extremely easy for the transpiler to decompose regardless of dimension and $I^{\otimes(n+1)}$ translates to no basis gates being applied, it makes sense that the transpile times and run times for all optimization levels are relatively similar. For constant 1, the transpile time clearly increases as the optimization level increases, as expected. However, we are unsure why the run time decreases and then shoots up with level-3 optimization; again, level-3 optimization heavily optimizes the circuit to use as few basis gates as possible in transpilation, so the run time should be relatively low compared to lower levels of optimization.

Scalability an n Grows

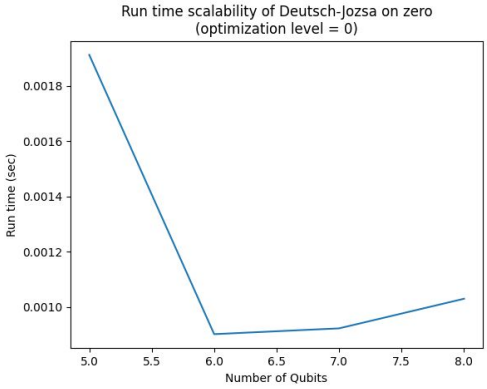
The plots below show for each level of optimization, the transpile times and run times vs. the number of qubits for the constant 0, constant 1, XOR, and XNOR functions. Unfortunately, we could not successfully generate results for a larger number of qubits because we ran into a potential bug in Qiskit when we attempted to transpile a circuit with 10 or more qubits: **ValueError: too many subscripts in einsum**. We found this odd given that the QASM simulator is intended to dynamically support as many qubits as your personal device has memory for, unlike the PyQuil simulator which only supports a fixed number of qubits depending on the QPU device you choose.

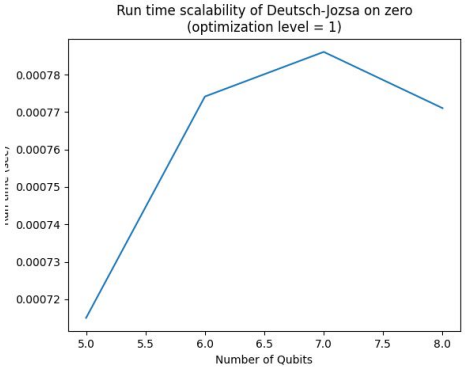
With regards to the scalability of n, for each test U_f, regardless of the optimization level, the transpile time appears to be exponential in number of qubits; this makes sense since, as we discussed in class, transpiling is NP-complete and hence the compiler will have an exponentially harder time transpiling 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. (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.)

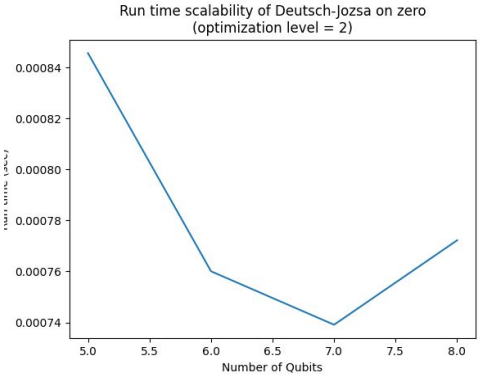
Lastly, regardless of number of qubits, with the exceptions of constant 0 and constant 1 for previously-specified reasons, clearly, higher optimization levels generate circuits with a shorter run time, at the expense of longer transpilation time, which substantiates Qiskit's documentation. Again, we hope that, with more research into optimized transpilation, we can reduce run times for large numbers of qubits and reduce errors in quantum computation.

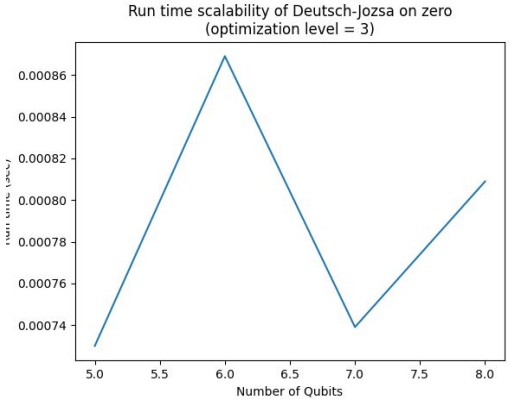
Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

Constant 0 Run Times



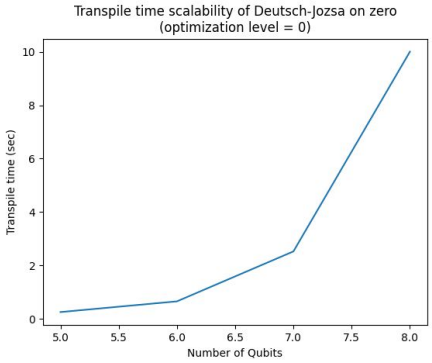


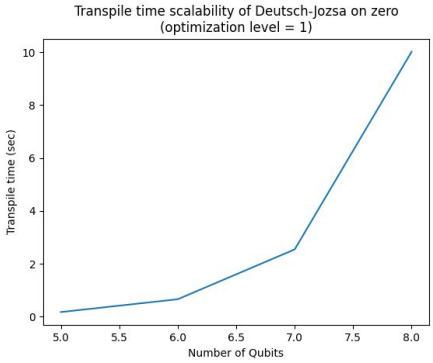


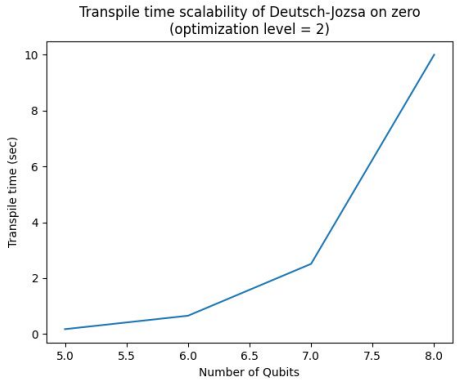


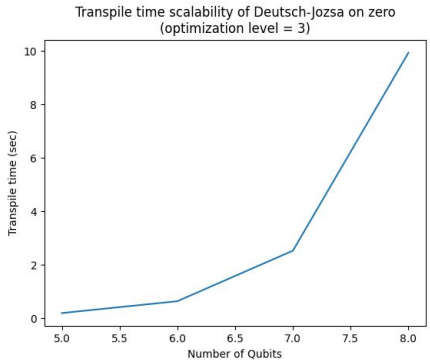
Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

Constant 0 Transpile Times



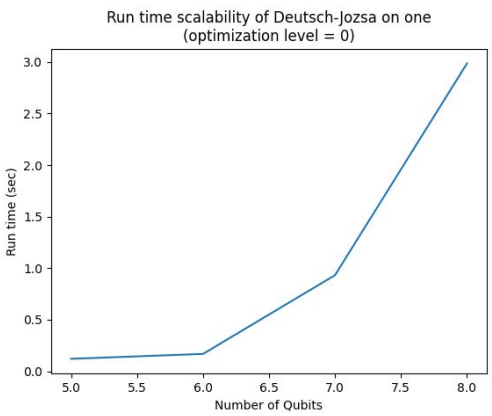


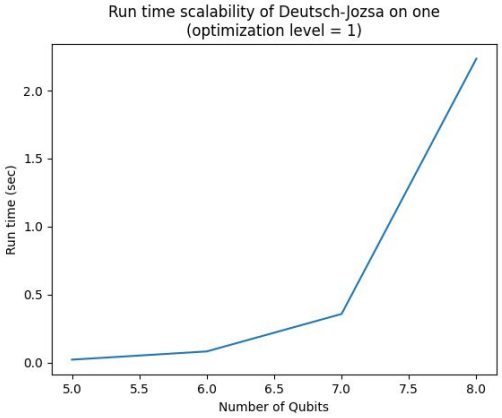


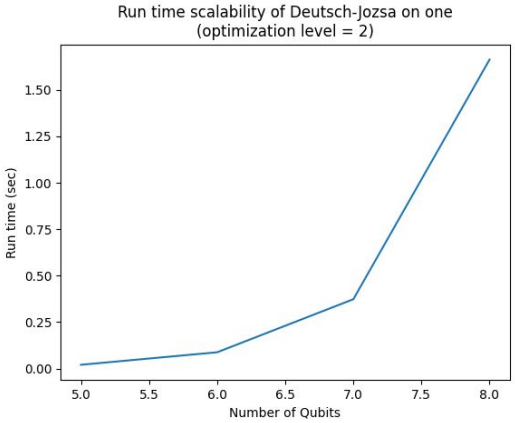


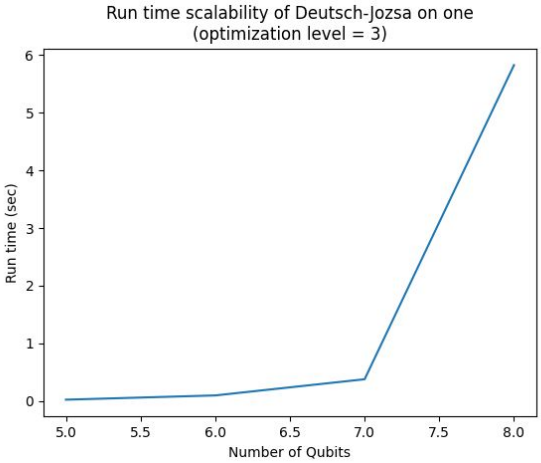
Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

Constant 1 Run Times



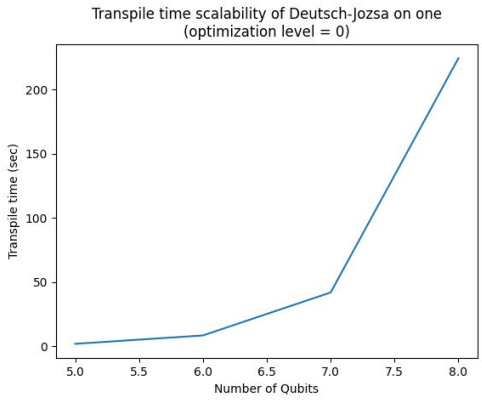


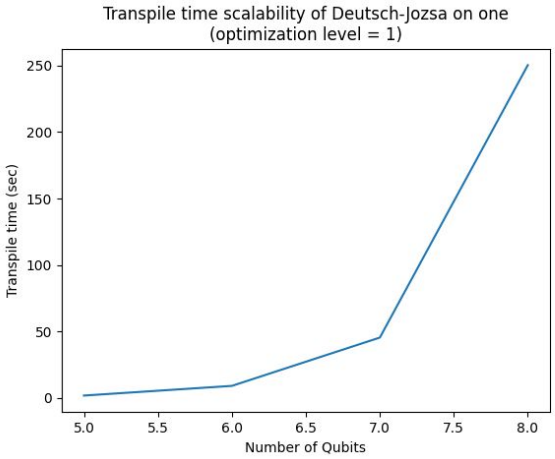


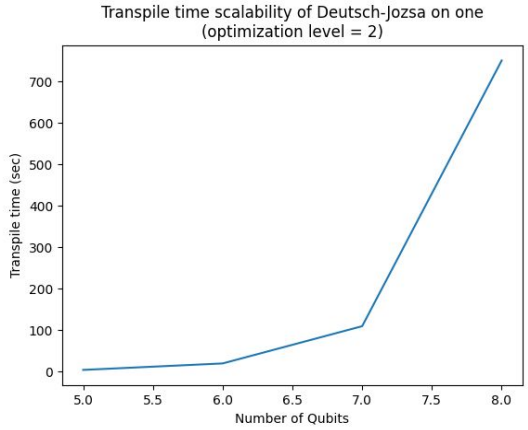


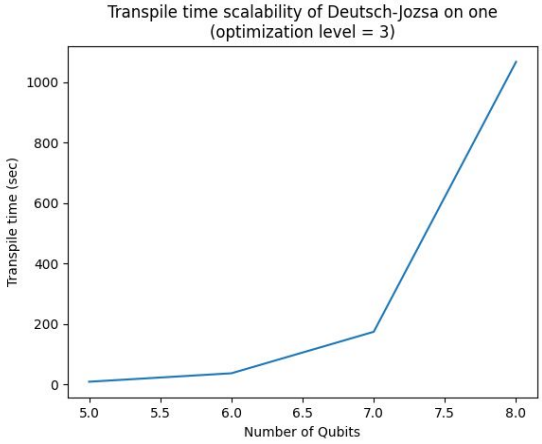
Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

Constant 1 Transpile Times



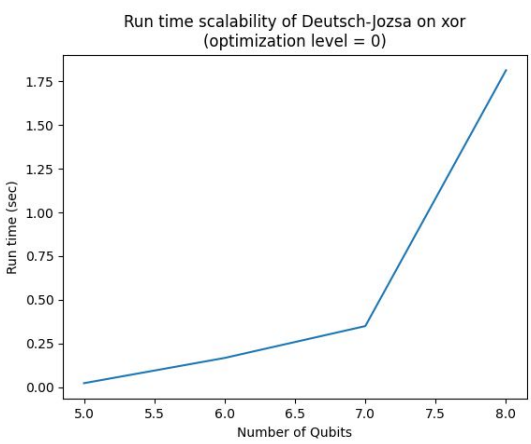


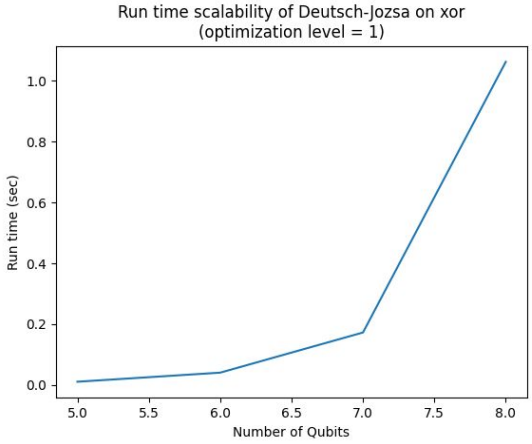


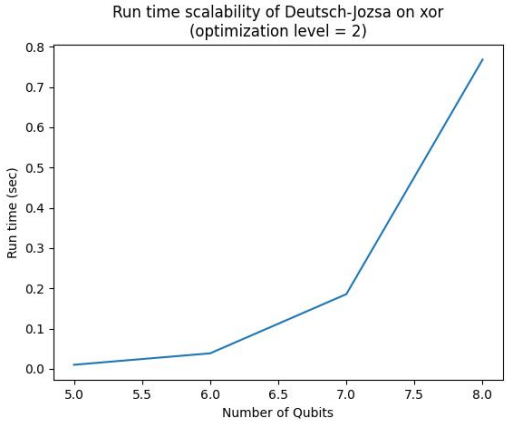


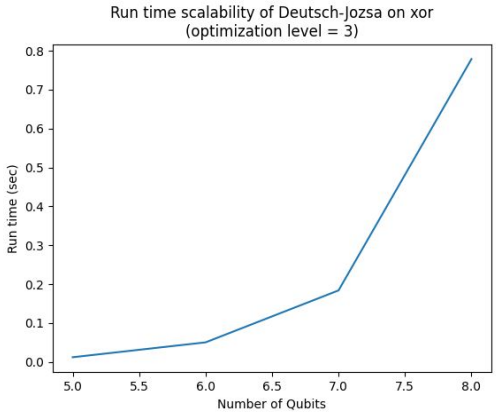
Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

XOR Run Times



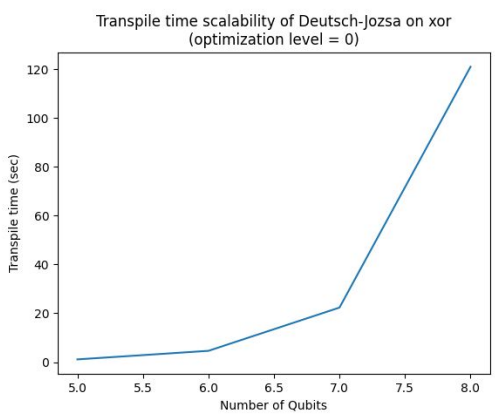


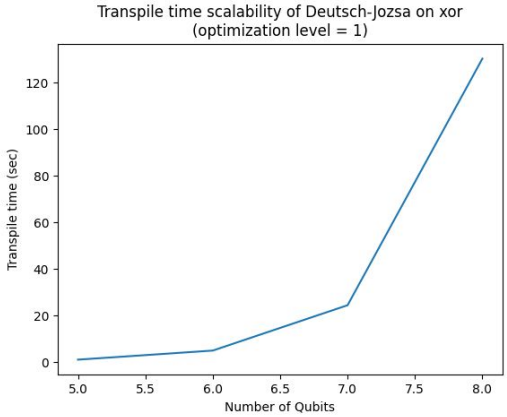


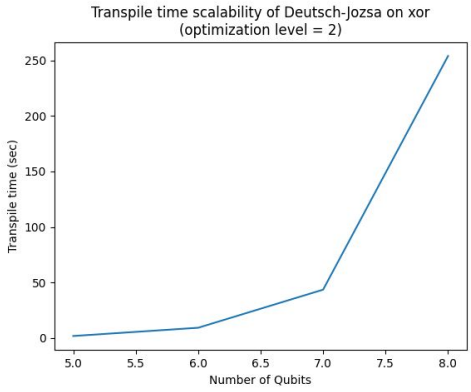


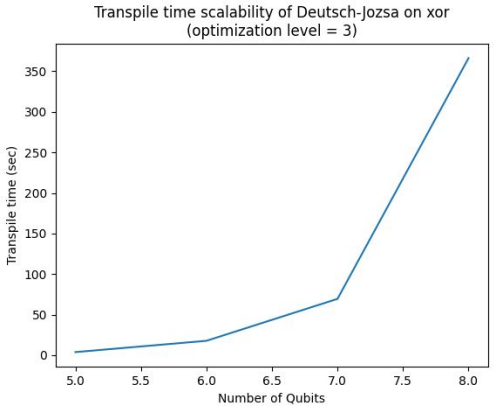
Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

XOR Transpile Times



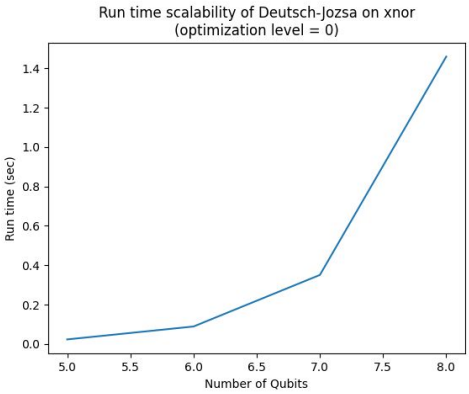


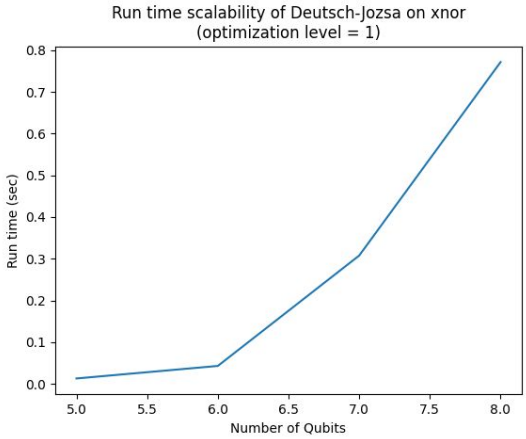


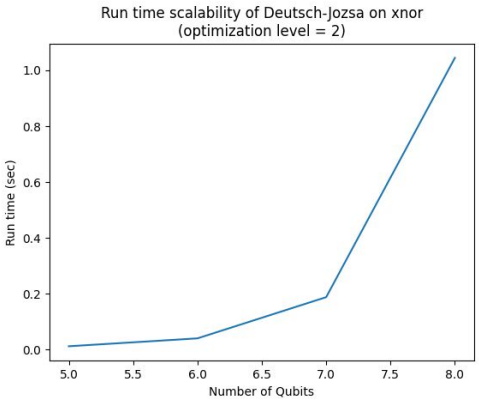


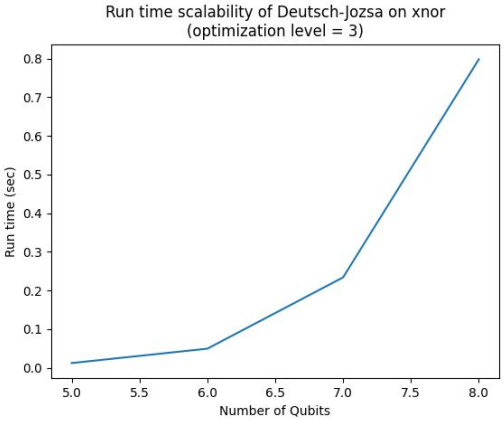
Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

XNOR Run Times



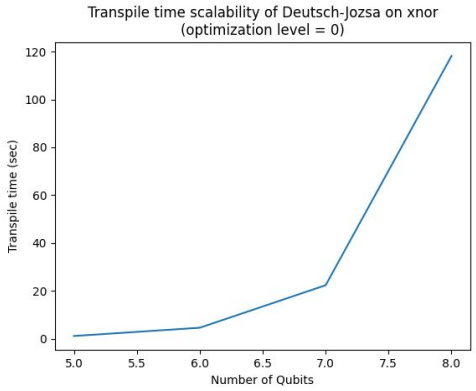


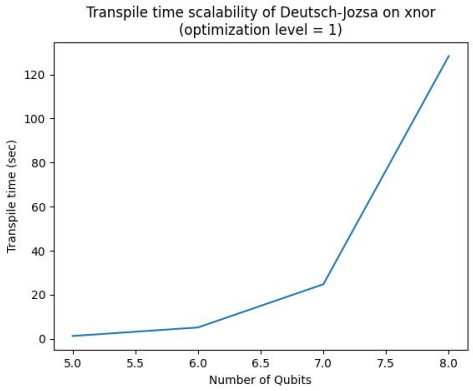


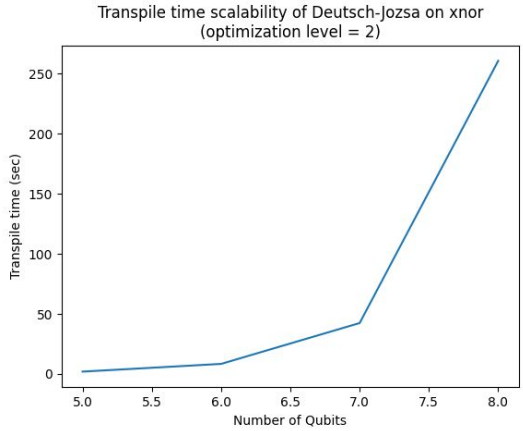


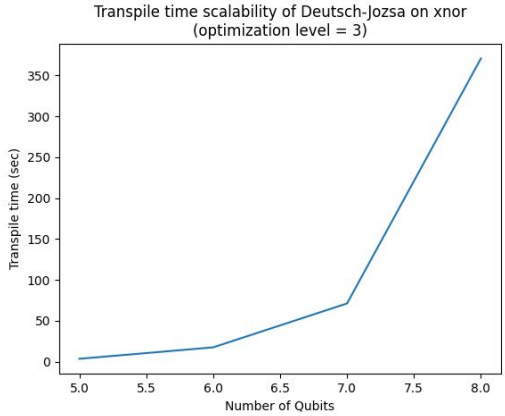
Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

XNOR Transpile Times









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

Bernstein-Vazirani (bernstein_vazirani.py)

Design and Implementation of U_f

For Bernstein-Vazirani, the logic and implementation of U_f is identical to that for Deutsch-Jozsa, i.e., U_f is a $2^{n+1}x2^{n+1}$ complex unitary matrix that can be divided into a $2^n x 2^n$ grid, where each box in the grid is a 2x2 matrix, such that the boxes are the zero matrix if they are not on the diagonal, and the k^{th} box on the diagonal is $X^{f(k)}$. The implementation of U_f is neat and easy to read as it is for Deutsch-Jozsa and all the challenges faced adding U_f to the circuit were identical to those for Deutsch-Jozsa above.

Preventing Access to U_f, Parametrization of Solution in n

We used the same approaches as for Deutsch-Jozsa to prevent the user from accessing U_f and parametrize the solution in n.

Testing Methodology and Presentation of Results

For each test, we print out the results; we use nice formatting to clearly label the test to which the trials correspond and print, separately, the transpilation time 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 actual measurements of the n input qubits and 3) 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).

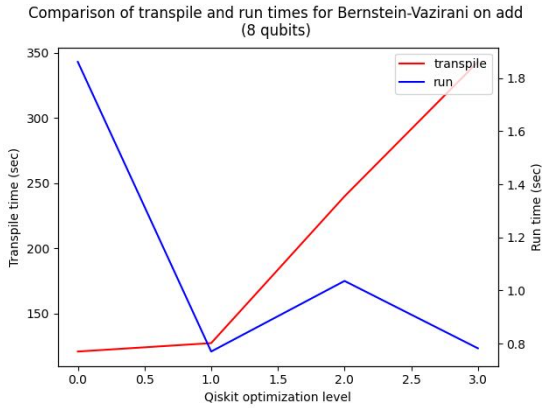
We tested our Bernstein-Vazirani implementation on the following functions:

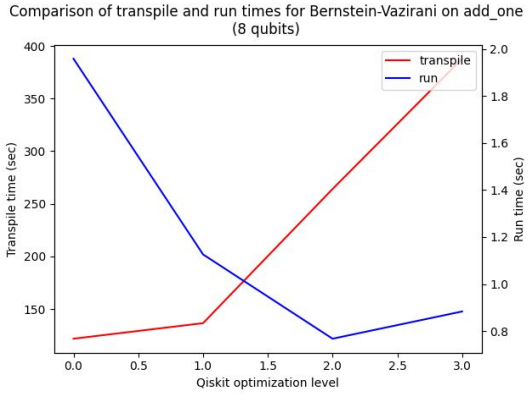
- add: returns the sum of all bits in the bit string (mod 2)
- add_one: returns the sum of all bits in the bit string plus 1 (mod 2)
    - This test allows us to additionally validate that the classical computation of b is correct
- top_half: returns the sum of the first half, rounded down, of the bit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

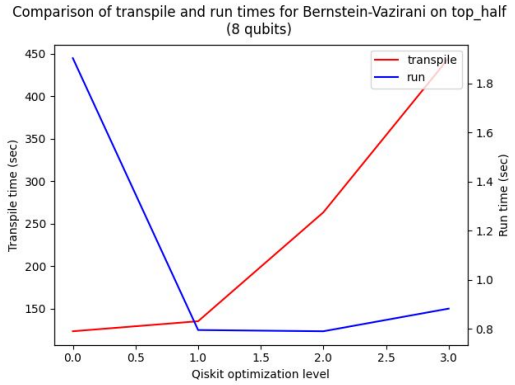
Execution Times

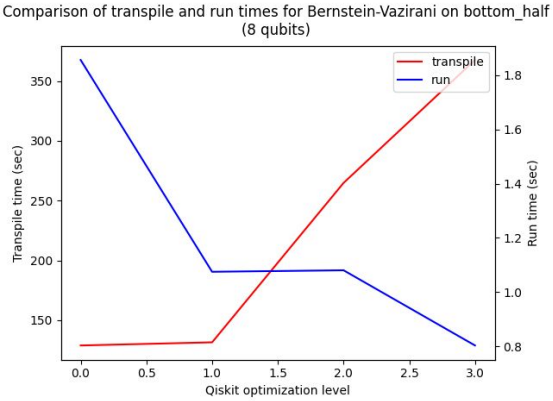
We use the same methodology as for Deutsch-Jozsa to compute transpilation and execution times. Presented below are plots comparing the transpilation and execution times for running deutsch_jozsa.py with 8 qubits 1-shot with all 4 optimization levels, on the 6 aforementioned test functions.

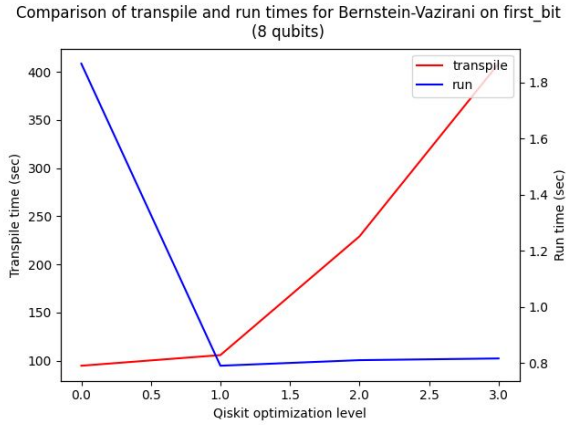
Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

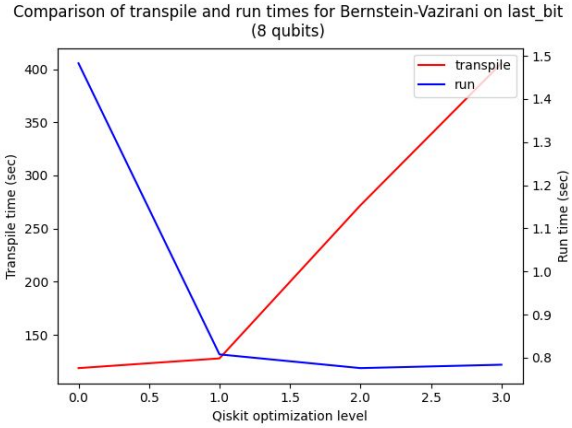












We noticed that with 8 qubits, all test functions exhibited relatively similar transpile and run times for each optimization level, with add having the smallest transpile time and top_half having the largest transpile times when optimization_level = 3. Interestingly, with PyQuil, top_half and first_bit seemed to have significantly lower execution times than the rest of the functions, which doesn't appear to be the case here.

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

As one would expect, for each function, as the optimization level increases, the transpile 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 decreases (obviously, a more optimized circuit with fewer gates will run more quickly).

Scalability an n Grows

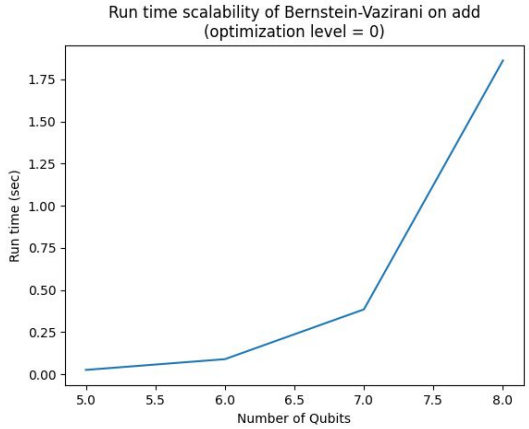
The plots below show for each level of optimization, the transpile times and run times vs. the number of qubits for the 6 test functions.

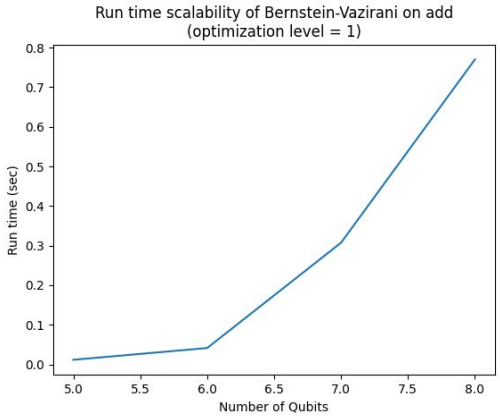
With regards to the scalability of n, for each test U_f, regardless of the optimization level, the transpile time appears to be exponential in number of qubits; this makes sense since, as we discussed in class, transpiling is NP-complete and hence the compiler will have an exponentially harder time transpiling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

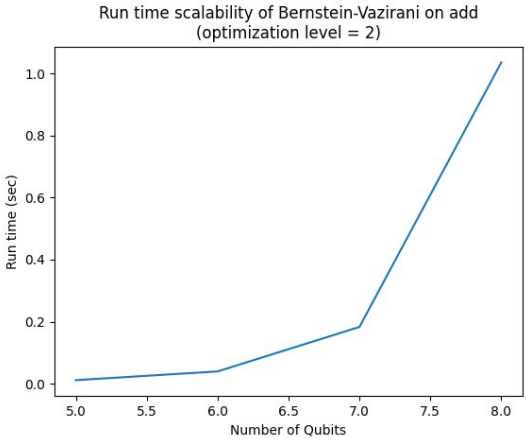
Lastly, regardless of number of qubits, clearly, higher optimization levels generate circuits with a shorter run time, at the expense of longer transpilation time, which substantiates Qiskit's documentation. We hope that, with more research into optimized transpilation, we can reduce run times for large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

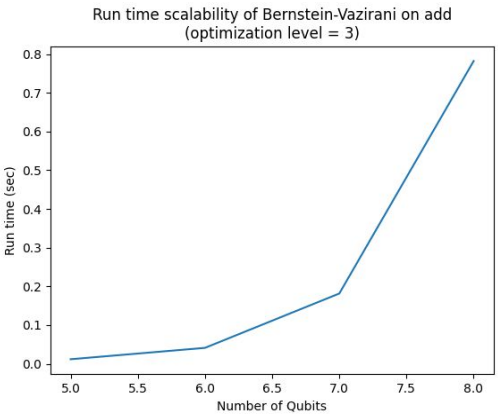
Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

add Run Times



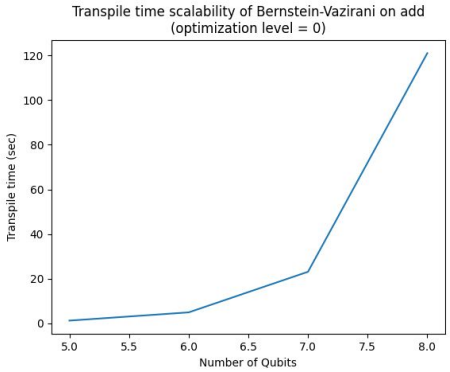


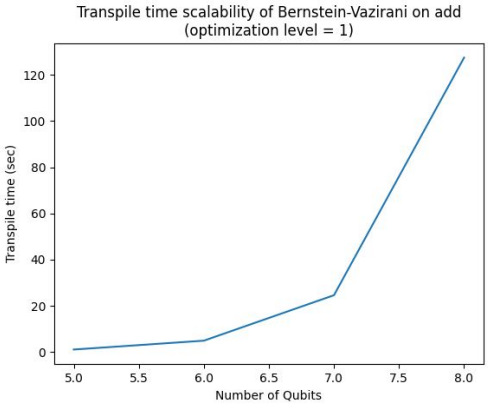


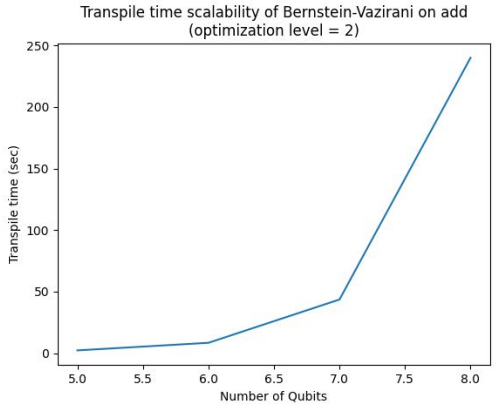


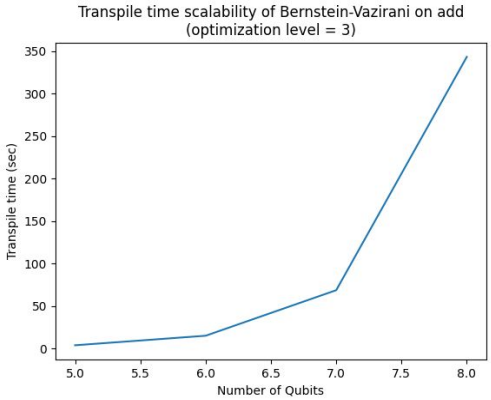
Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

add Transpile Times



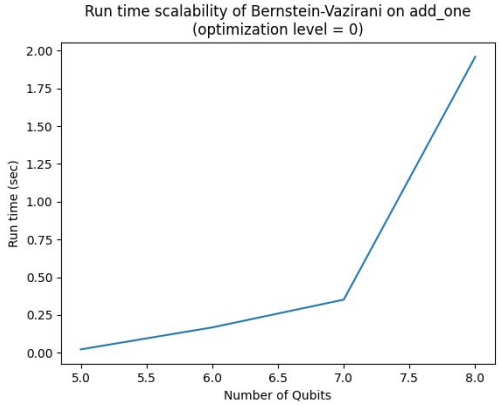


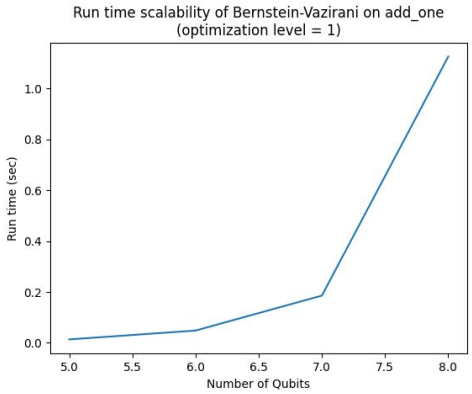


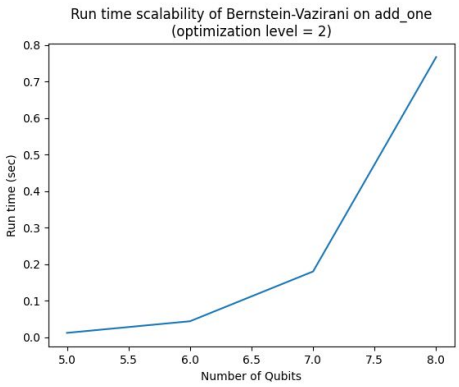


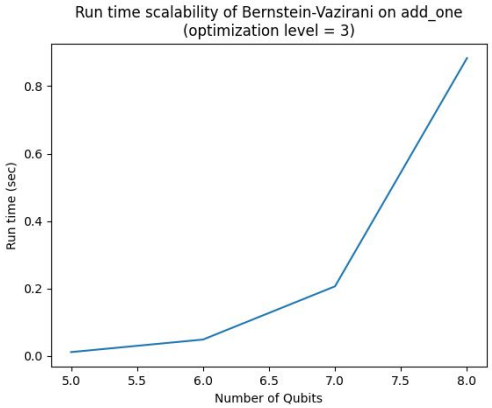
Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

add_one Run Times



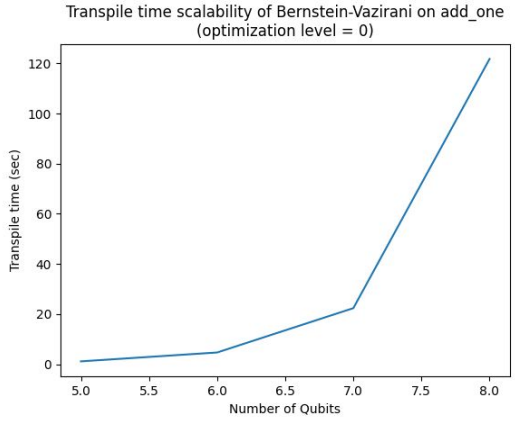


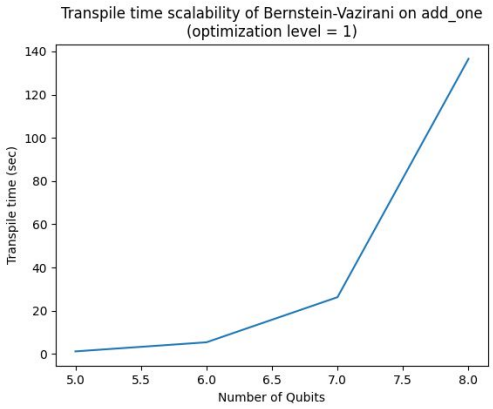


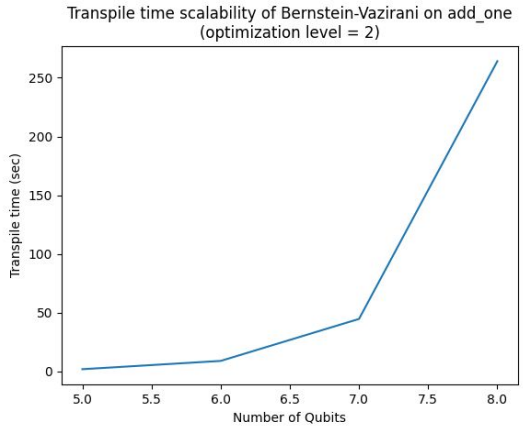


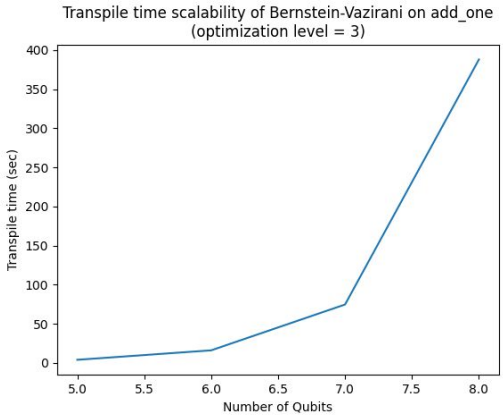
Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

add_one Transpile Times



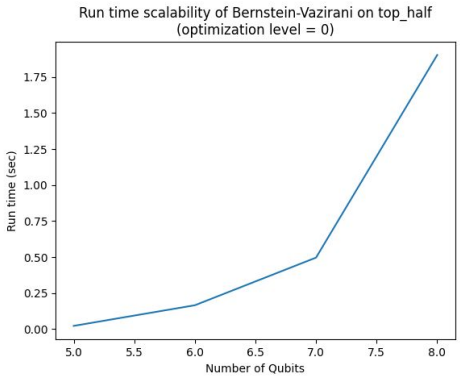


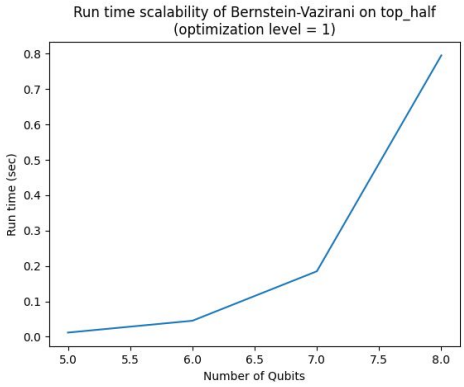


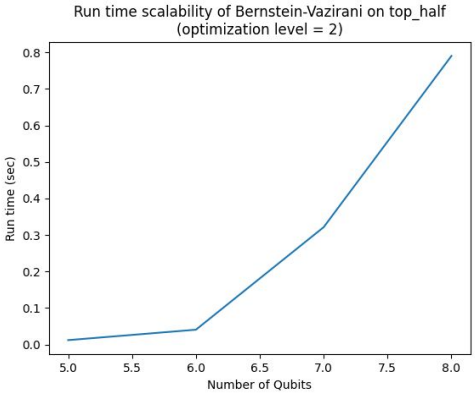


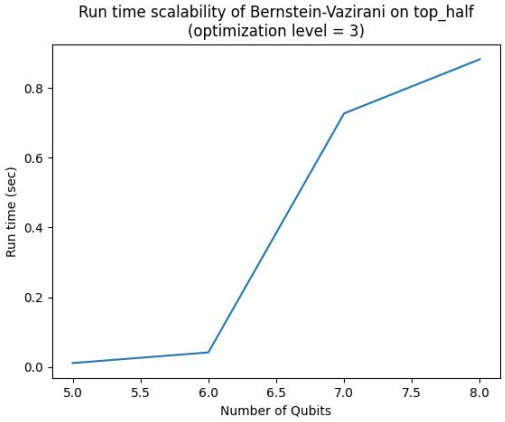
Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

top_half Run Times



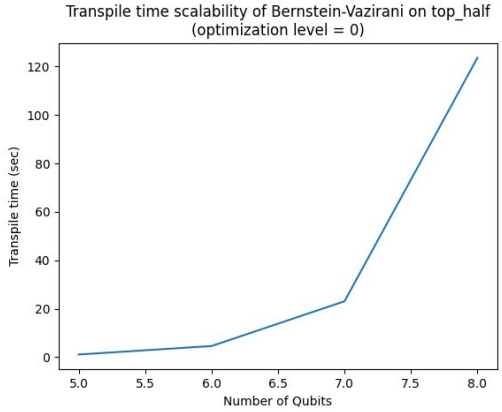


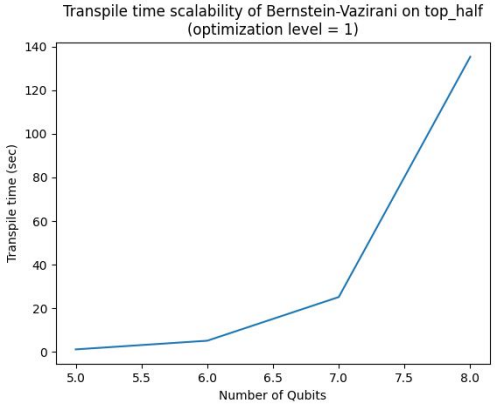


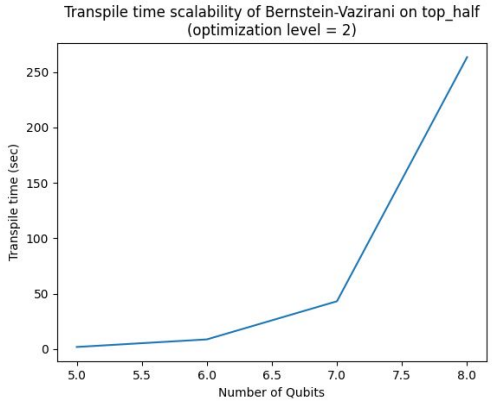


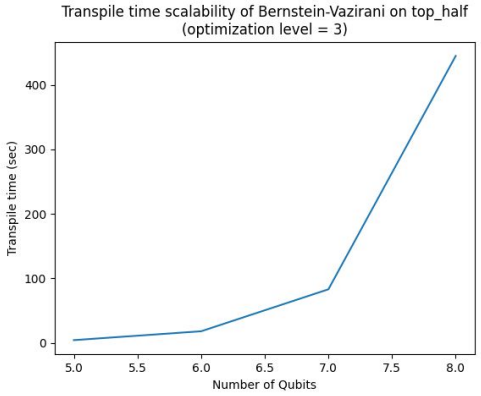
Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

top_half Transpile Times



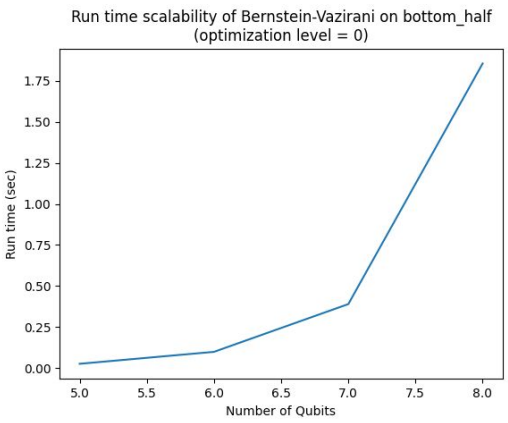


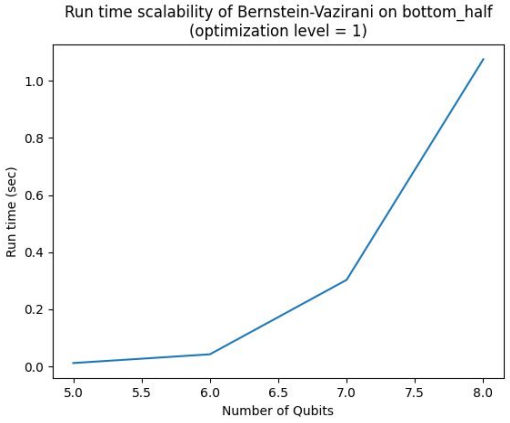


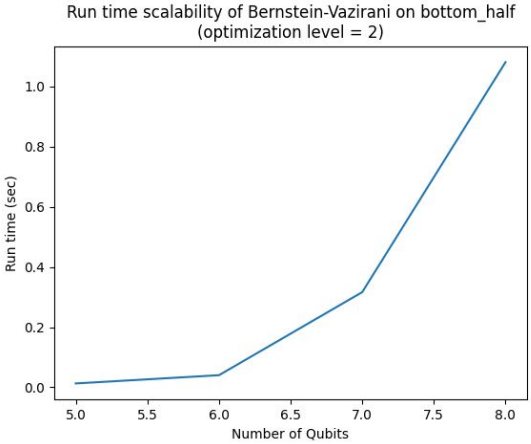


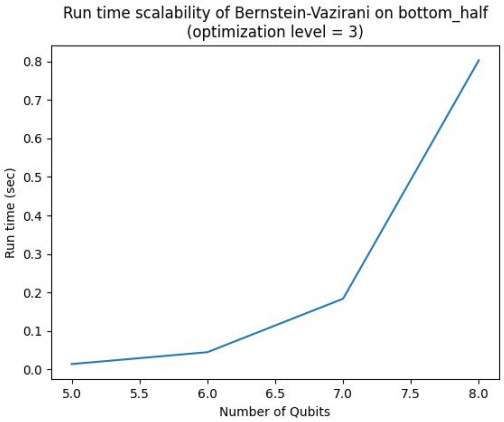
Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

bottom_half Run Times



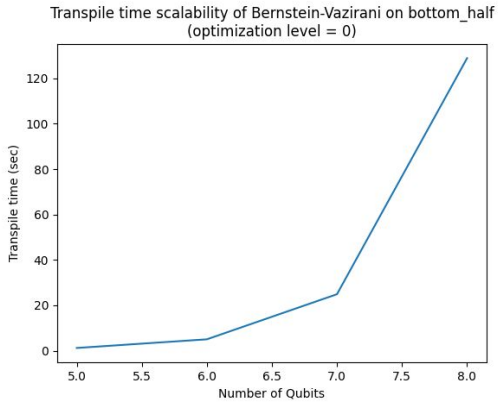


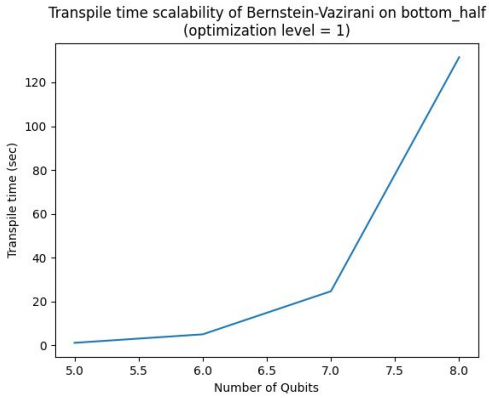


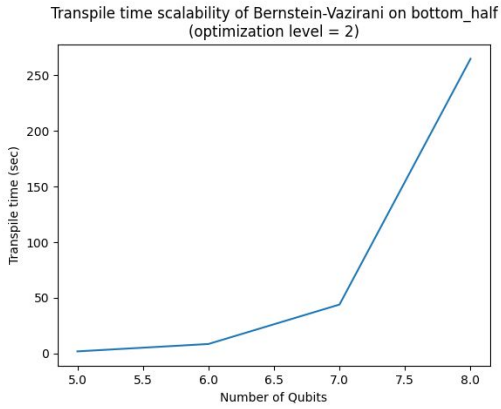


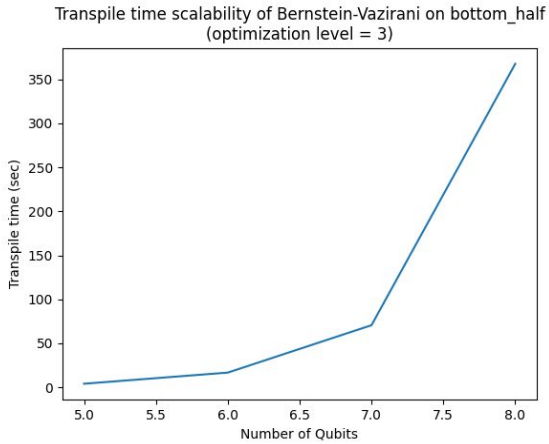
Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

bottom_half Transpile Times



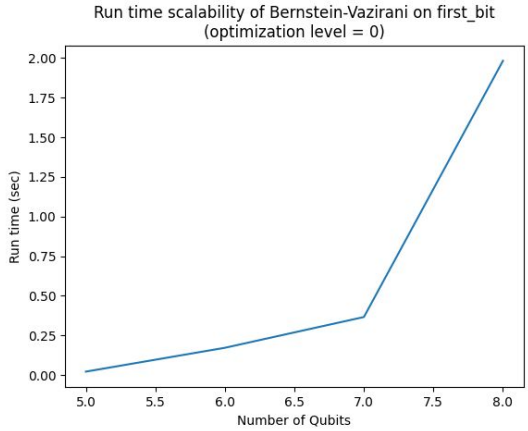


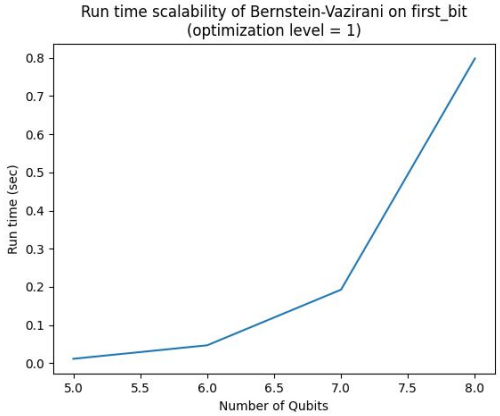


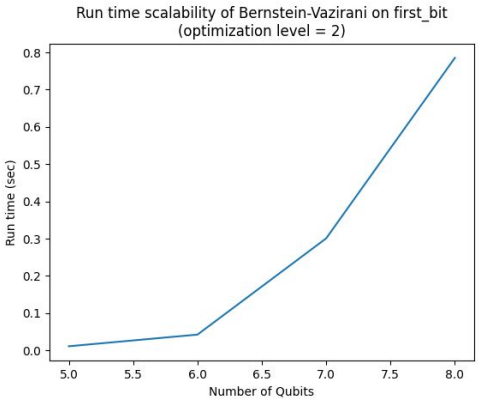


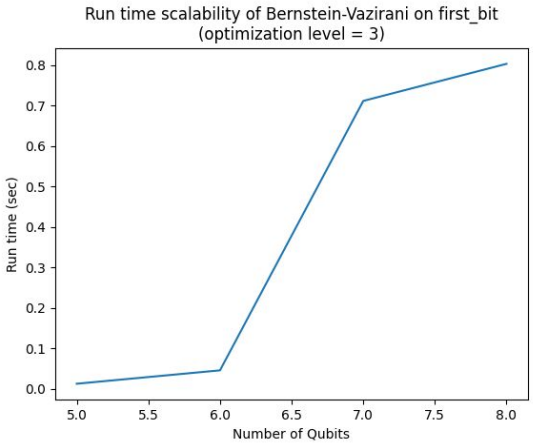
Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

first_bit Run Times



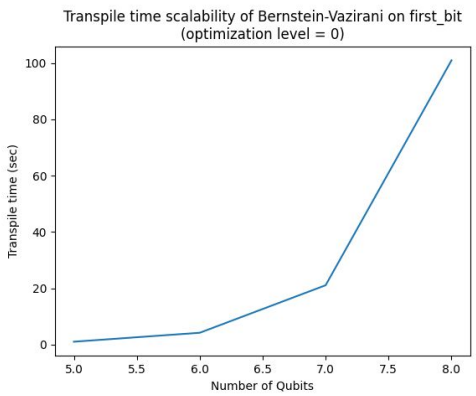


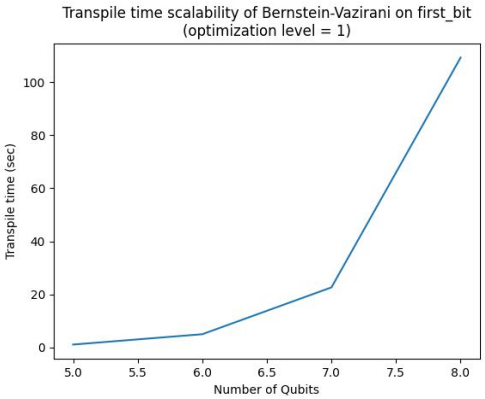


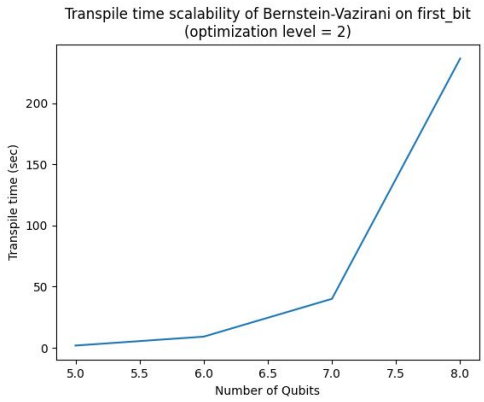


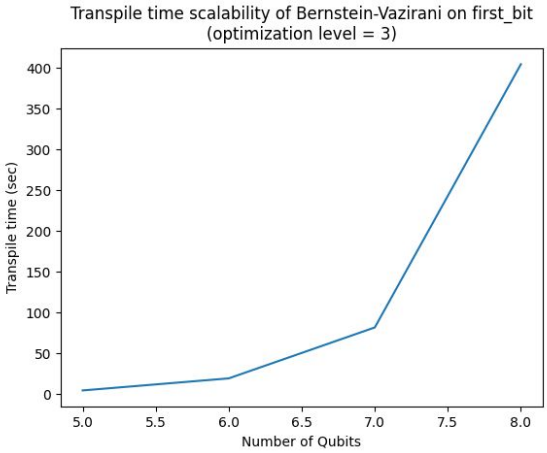
Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

first_bit Transpile Times



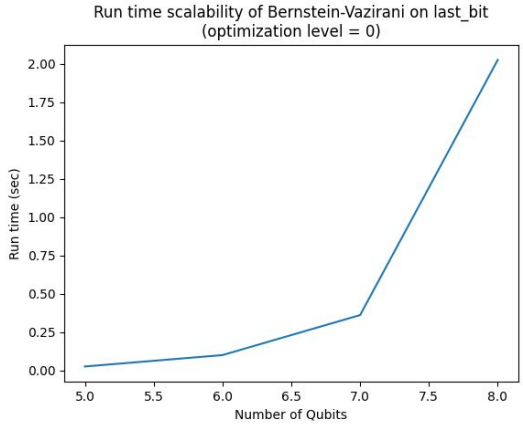


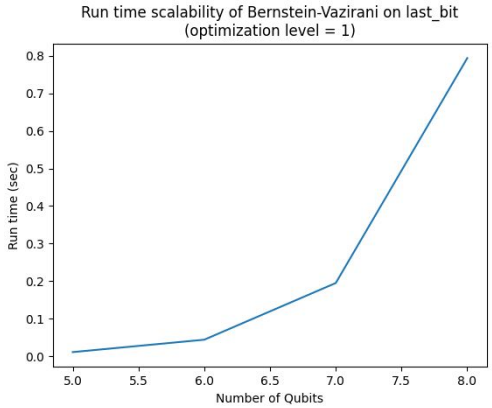


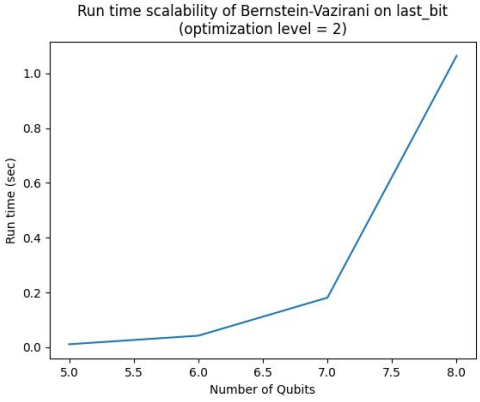


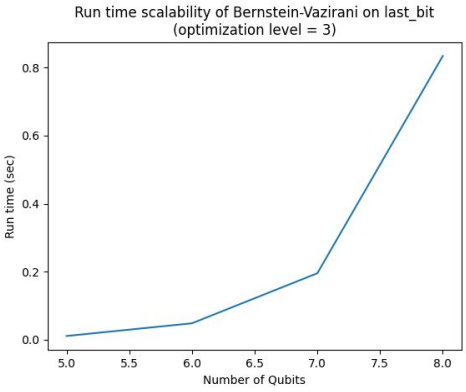
Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

last_bit Run Times



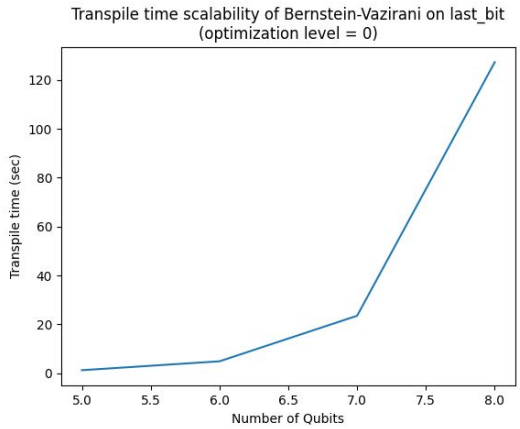


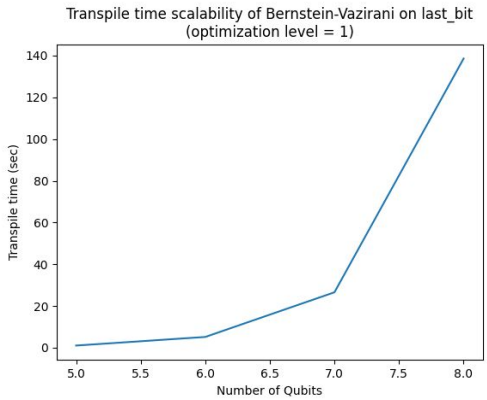


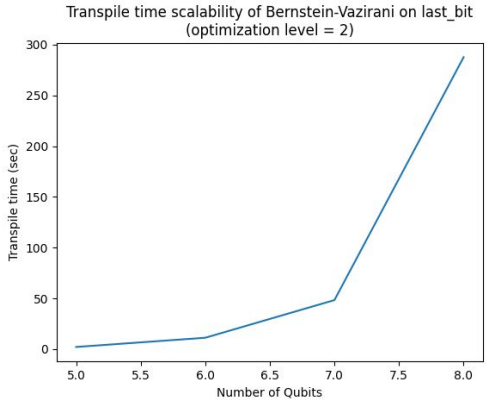


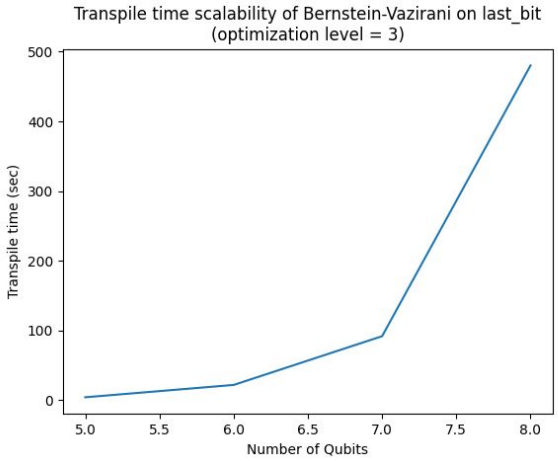
Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

last_bit Transpile Times









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

Grover (grover.py)

Implementing Z_f and Z_0 was relatively straightforward. Z_f is just a 2^n by 2^n diagonal matrix, where the i-th entry in the diagonal corresponds to the 2^i-th possible n-bit input x and is equal to (-1)^f(x). In this way, |x> gets mapped to (-1)^f(x) |x>. We just passed a list of all 2^n diagonal entries to np.diag to obtain the appropriate matrix for Z_f as a NumPy array. Similarly, Z_0 is just a 2^n by 2^n diagonal matrix, where the i-th entry in the diagonal corresponds to the 2^i-th possible n-bit input x and is equal to -1 if x == {0}^n and 1 otherwise. In this way, |x> gets mapped to -|x> if x == {0}^n and |x> otherwise. To efficiently implement this, we used np.eye to create a 2^n by 2^n identity NumPy array and set the top-leftmost entry to -1.

The implementations of Z_f and Z_0 are pretty neat and easy to read; they clearly follow the logic described above. If we opened up the black-boxes Z_f and Z_0, it would also be neat and easy to read since all the non-zero entries in the matrix representing the unitary gates are along the diagonal. In our program, we call the function get_Z_f, which takes as input the function f and n and returns the matrix representing Z_f, and get_Z_0, which takes as input n and returns the matrix representing Z_0,. We then pass this matrix into grover_program (the function that builds the Grover quantum circuit), which actually defines the gates Z_f and Z_0 using the matrices and specifies them to be applied to the input qubits, as shown below.

```
# define the Z_f gate based on the unitary matrix returned by get_Z_f
Z_f_GATE = Operator(Z_f)
circuit.unitary(Z_f_GATE, range(n), label = 'Z_f')

# define the Z_0 gate based on the unitary matrix returned by get_Z_0
Z_0_GATE = Operator(Z_0)
circuit.unitary(Z_f_GATE, range(n), label = 'Z_0')
```

Preventing Access to U_f

With regards to preventing the user accessing the implementations of Z_f and Z_0, the user writes the classical implementation of f in func.py and inputs the name of f as a command line argument when running grover.py.

Z_f and Z_0 are constructed dynamically during runtime using get_Z_f and get_Z_0, so there is no way to access the entries of the matrix for Z_f or Z_0. However, because get_Z_f and get_Z_0 are not encapsulated in a class, and furthermore, there is no private access modifier on either, the user could create their own script to call get_Z_f or get_Z_0 and inspect the internals of the matrices returned.

Parametrization of Solution in n

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

The parameter n is passed into the program as a command line argument. Then, during runtime, n is passed to get_Z_f and get_Z_0 to dynamically construct the matrices for Z_f and Z_0 with the appropriate dimensions (i.e. 2^n by 2^n). n is also passed to grover_program, where it governs the number of qubits to which Hadamard is applied at the beginning of the circuit and on which qubits and the number of times G operates.

Testing Methodology and Presentation 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 the QASM simulator is a noisy (i.e. not ideal) quantum simulator, we set up our test driver to accept the number of trials to run as a command line argument. For each test, we print out the results; we use nice formatting to clearly label the test to which the trials correspond and the total execution time for the entire test (including all the trials), and subsequently, for each trial, present 1) the actual measurements of the n input qubits and 2) what f evaluates to for the measurements (0 or 1).

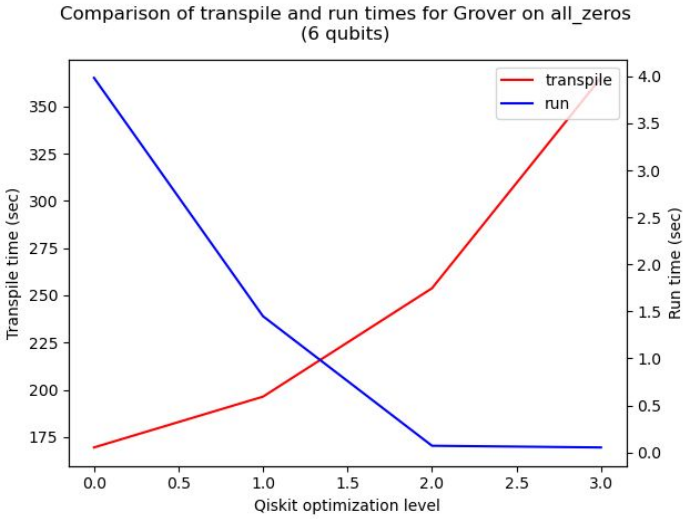
Execution Times

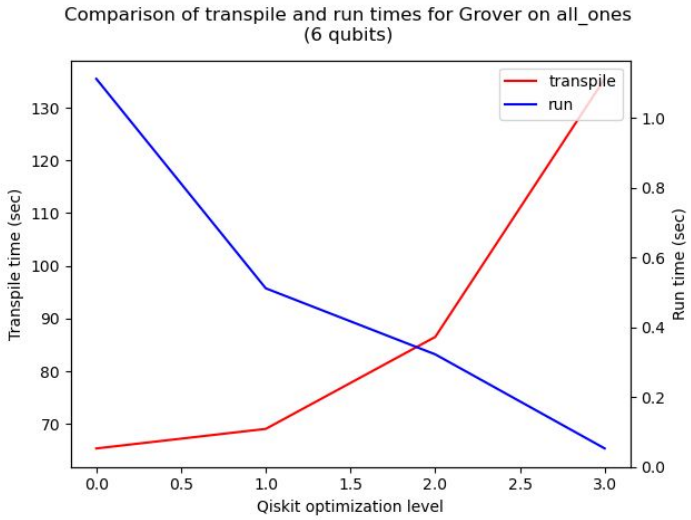
Different cases of Z_f definitely lead to different execution times. As stated earlier, for each test, we compute the execution time including all the trials. This execution time does not include the construction of the black-boxes Z_f and Z_0 (since Grover assumes Z_f and Z_0 are available), but it does include the compilation and running of the quantum circuit, as well as the measurements of the qubits and their interpretation (since the classical logic for interpreting the measurements is part of Grover's algorithm). We noticed that with 4 qubits, the constant zero functionand 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 compiler to decompose than matrices with off-diagonal non-zero entries.

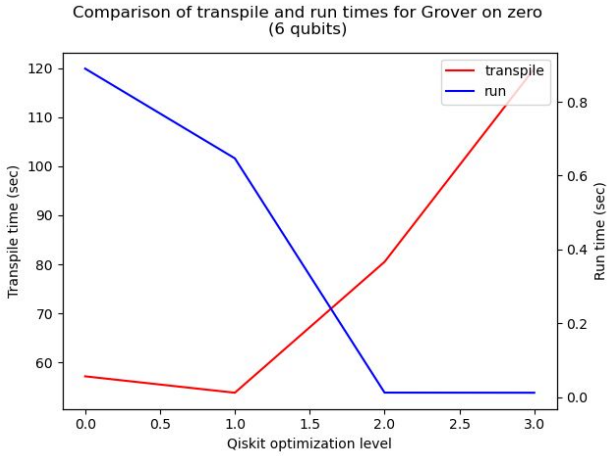
Neither the transpilation time nor the run time includes the construction of the black-box Z_f (since Grover assumes Z_f is available). Presented below are plots comparing the transpilation and execution times for running grover.py with 6 qubits 1-shot with all 4 optimization levels, on 4 different functions (all_ones, all_zeros, constant 0, and XNOR).

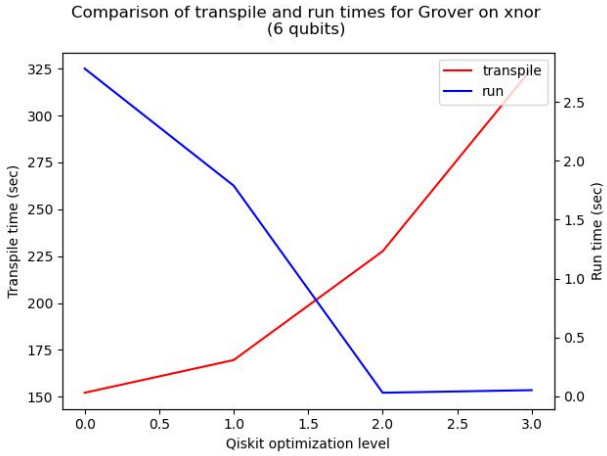
Transpile Time and Run Time Comparisons

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









We noticed that with 6 qubits, constant 0 led to a much faster execution time than all_zeros and XNOR. We thought this was due to the fact that constant functions make the composition of Z_f much easier.

Scalability an n Grows

With regards to the scalability of n, for each test function, the execution time appears to be exponential in number of qubits; this makes sense as the compiler will have a harder time decomposing larger matrices. The plots below show the execution time for all the test functions mentioned above vs. the number of qubits.

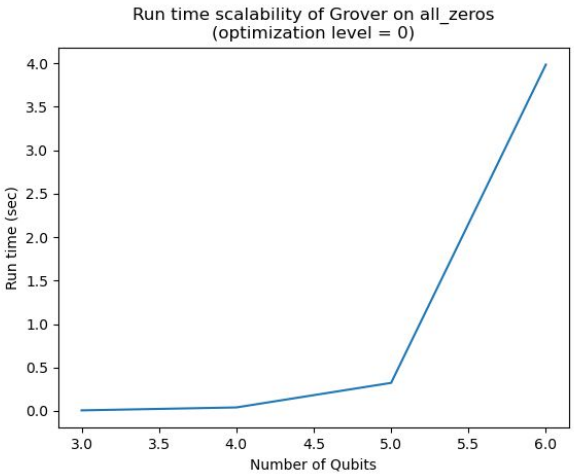
Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

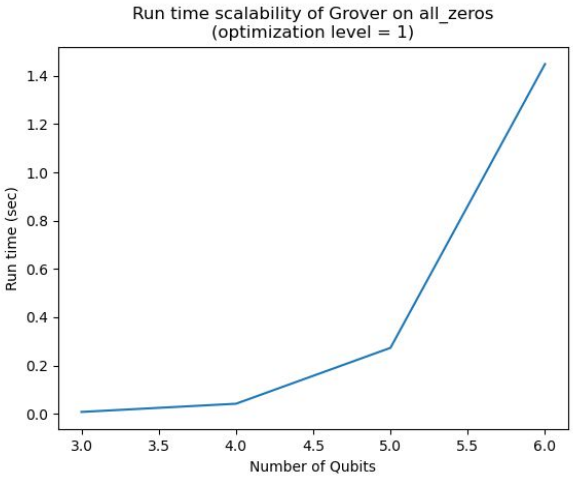
Regardless of the optimization level, the transpile time appears to be exponential in number of qubits; this makes sense since, as we discussed in class, transpiling is NP-complete and hence the compiler will have an exponentially harder time transpiling larger matrices whose dimension grows exponentially in the number of qubits. Additionally, for each test Z_f, 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.

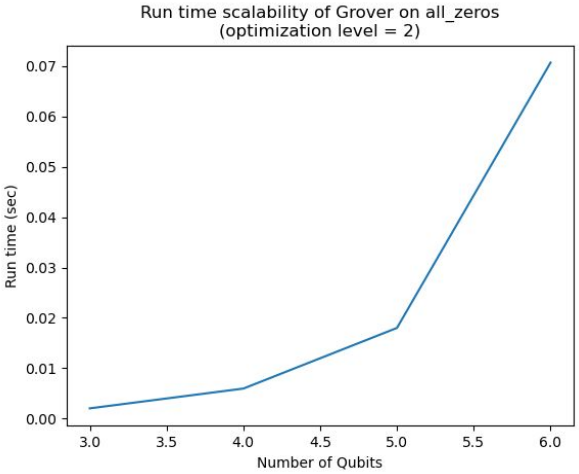
Lastly, regardless of number of qubits, higher optimization levels generate circuits with a shorter run time, at the expense of longer transpilation time, which substantiates Qiskit's documentation.

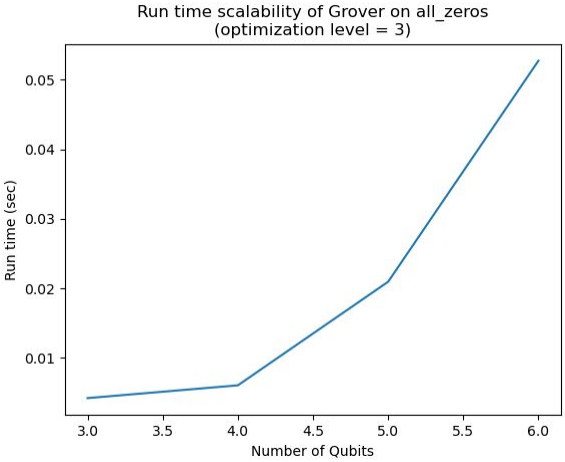
Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

All Zeros Run Times



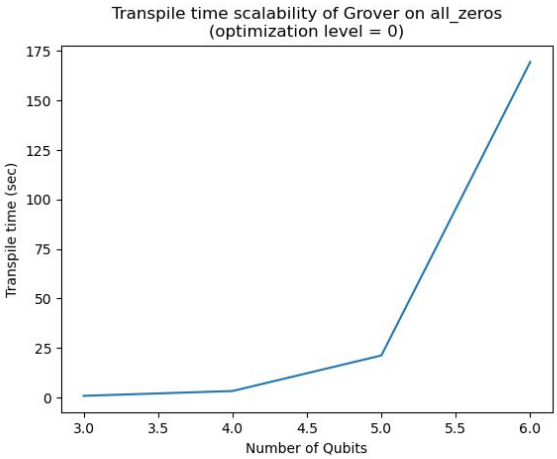


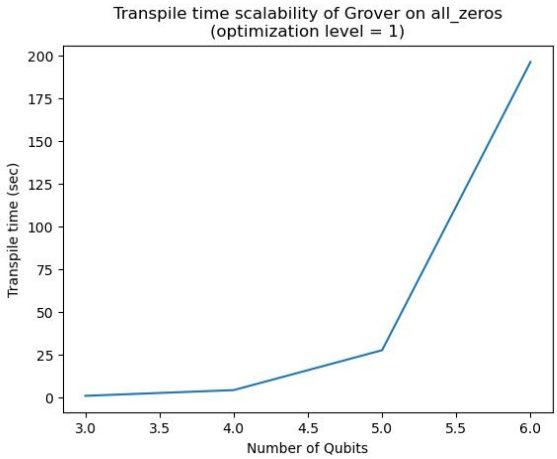


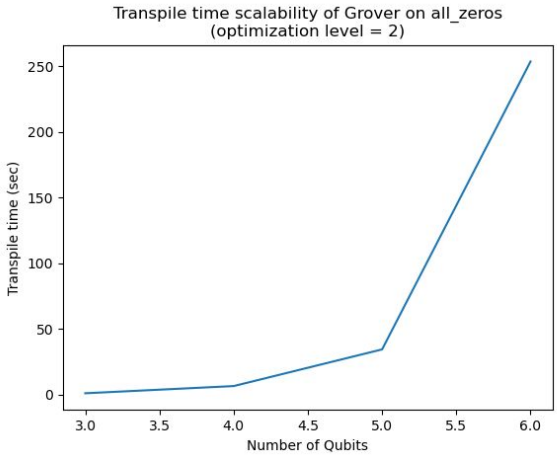


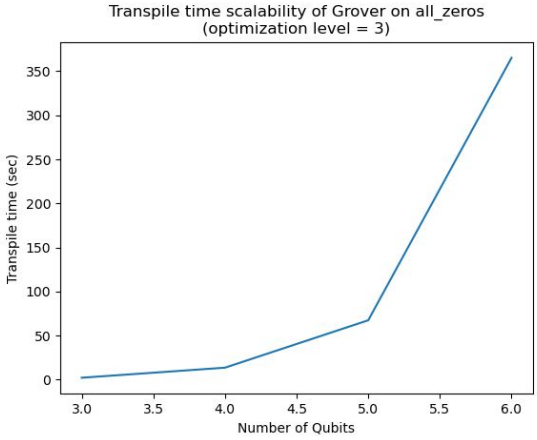
Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

All Zeros Transpile Times



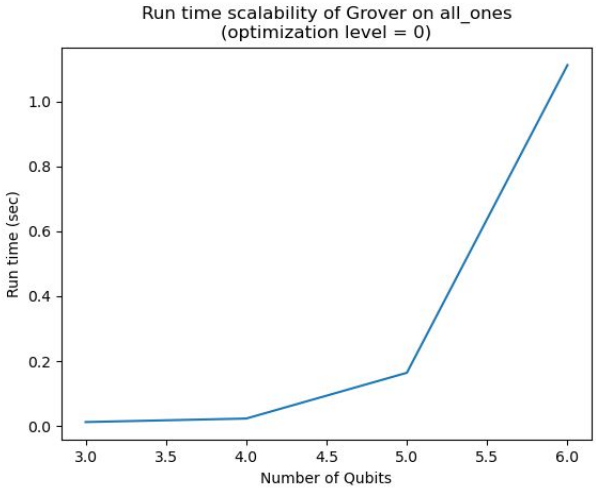


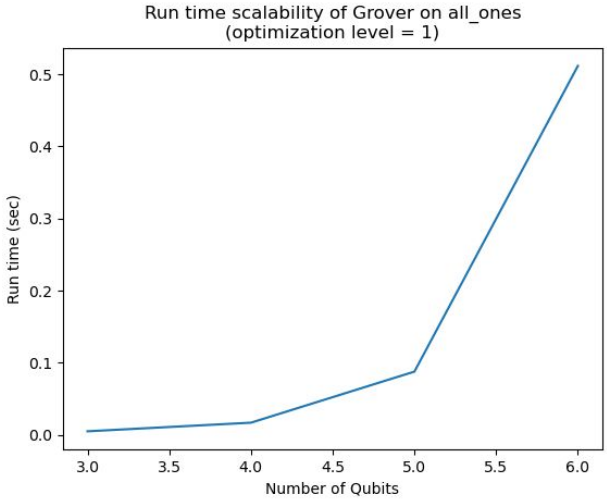


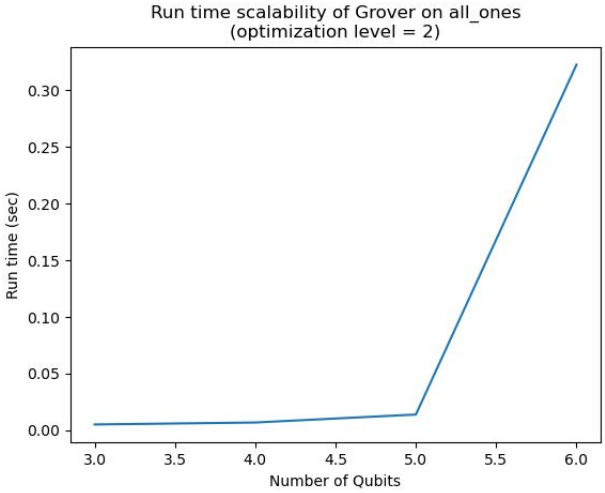


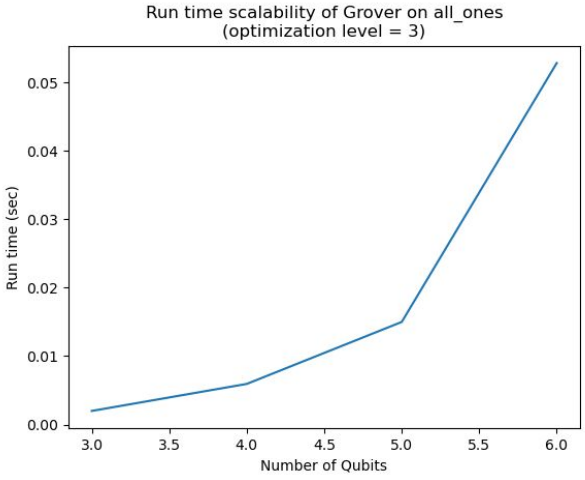
Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

All Ones Run Times



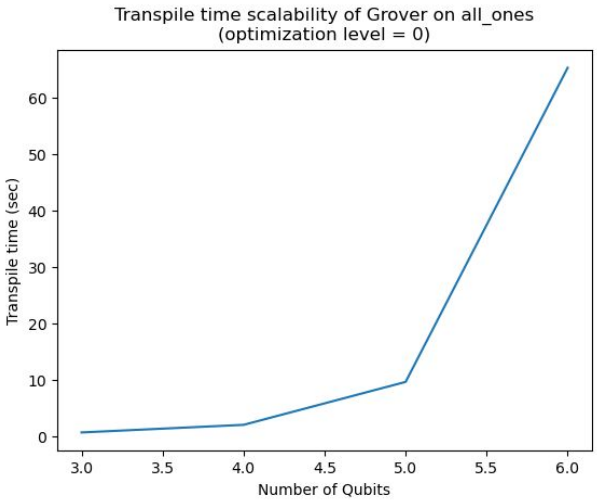


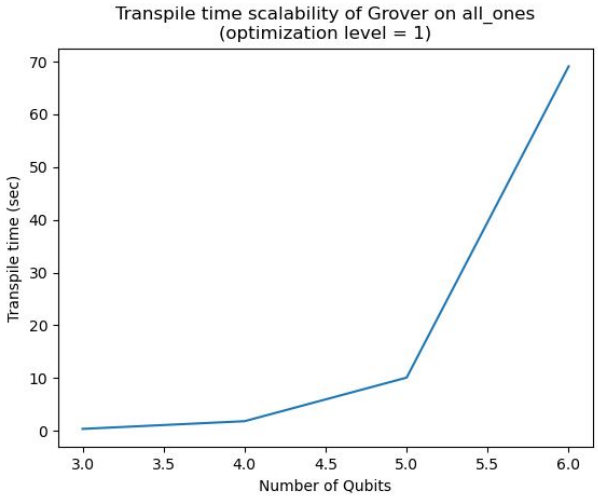


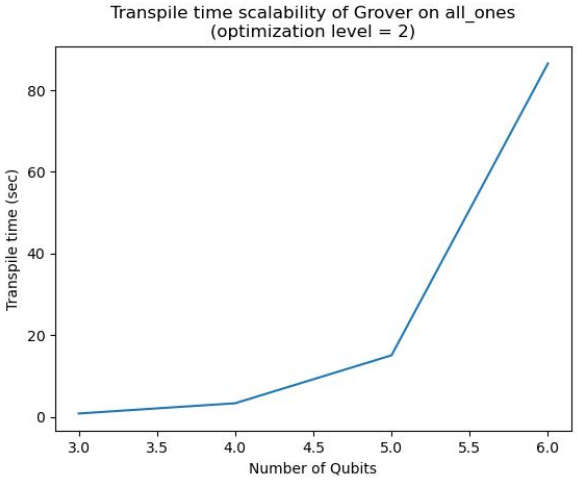


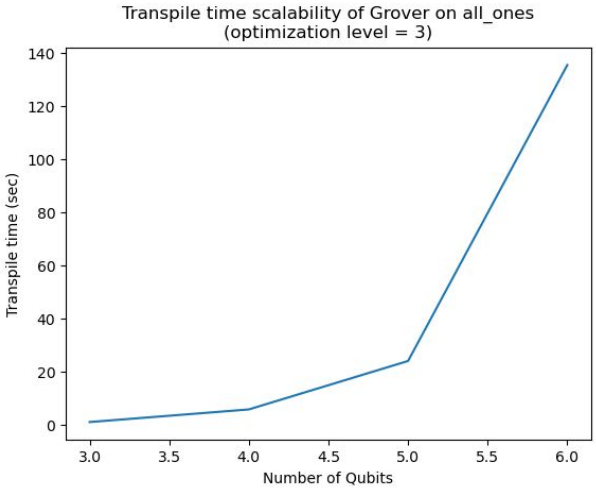
Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

All Ones Transpile Times



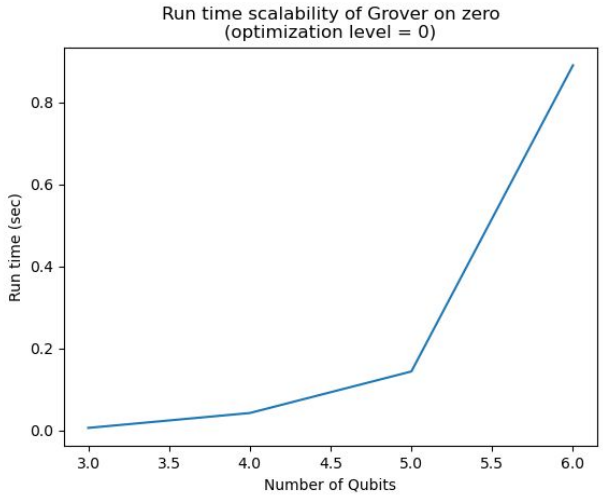


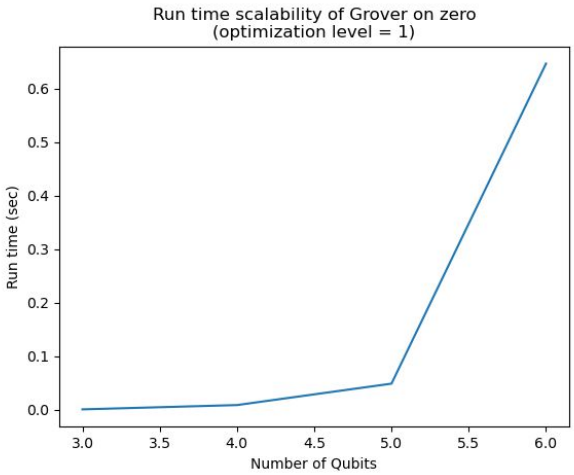


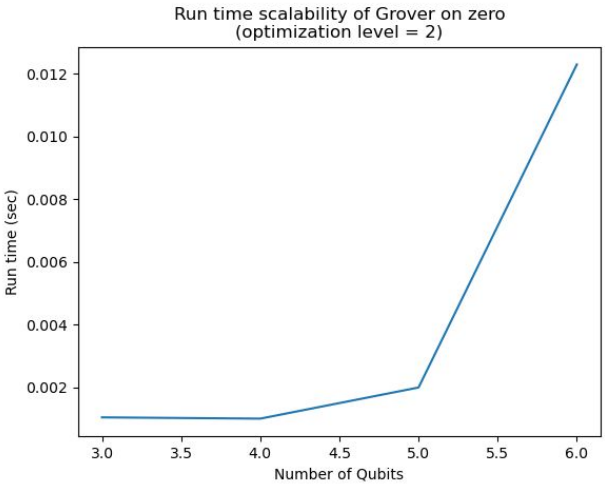


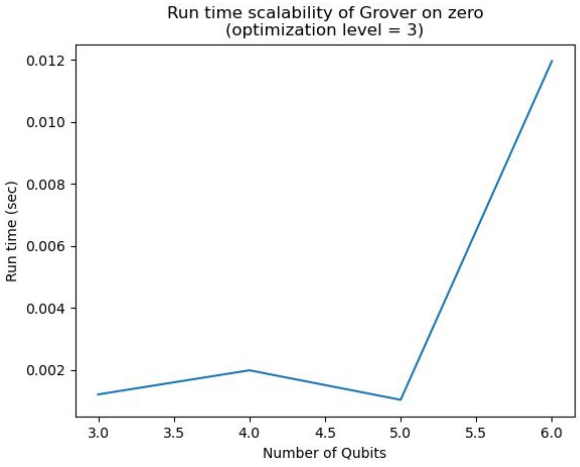
Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

Constant Zero Run Times



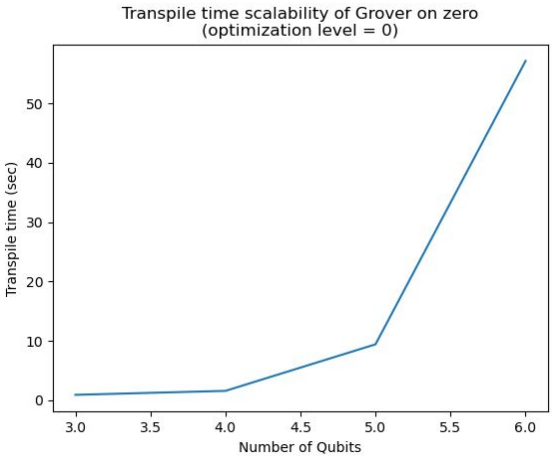


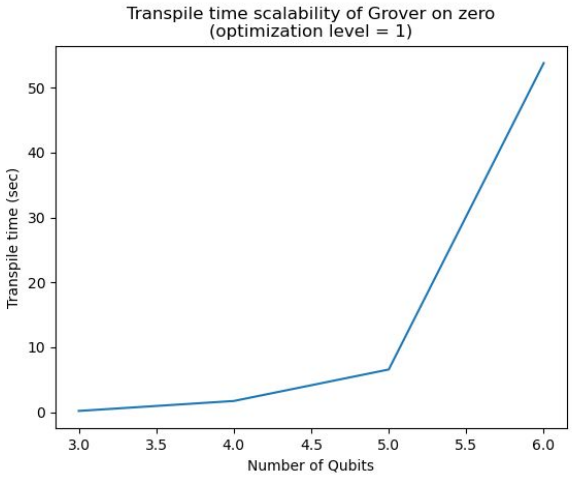


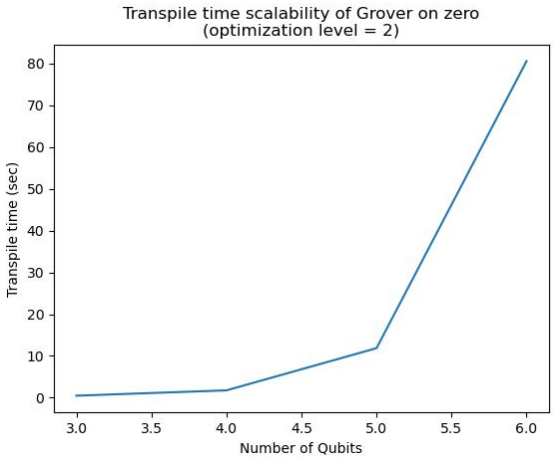


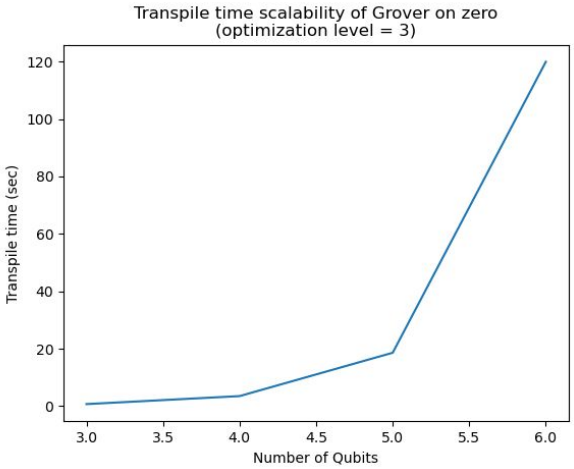
Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

Constant Zero Transpile Times



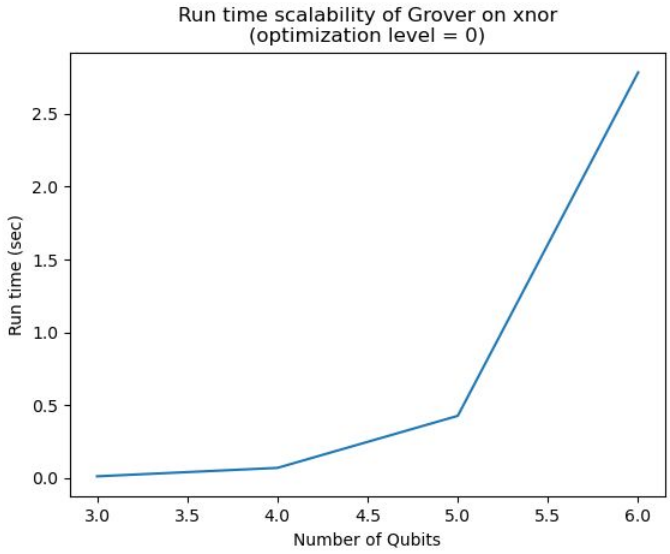


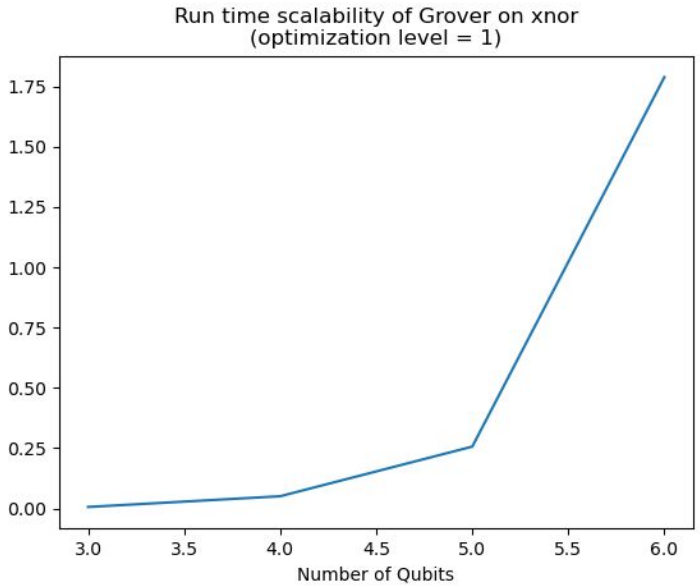


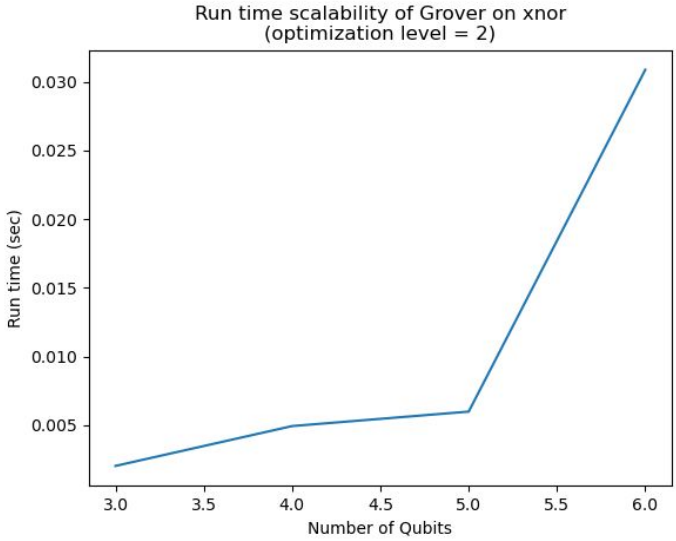


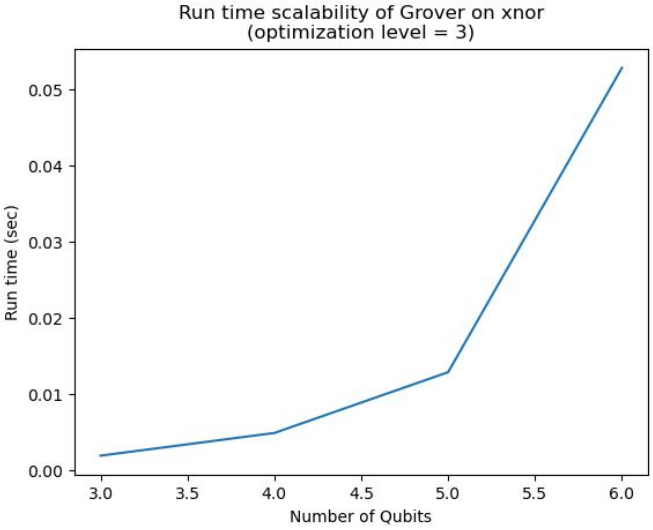
Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

XNOR Run Times



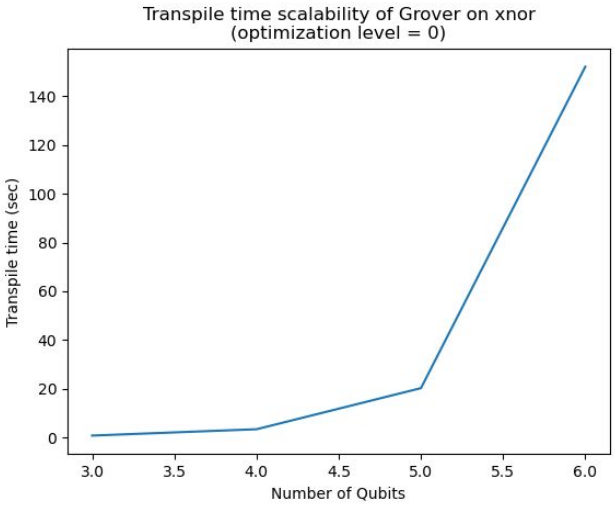


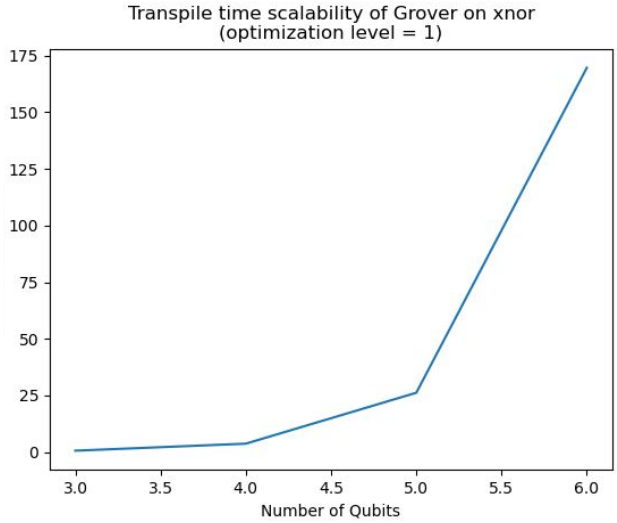


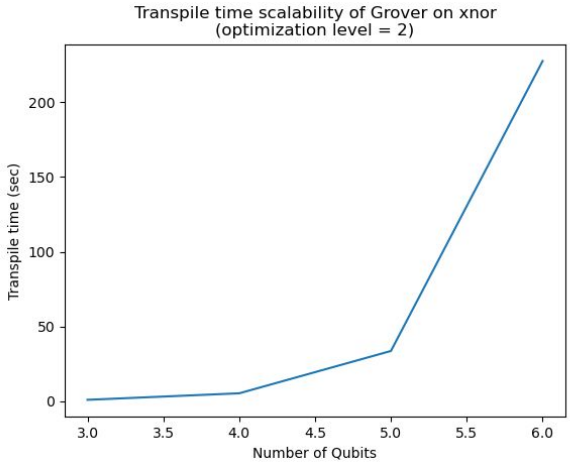


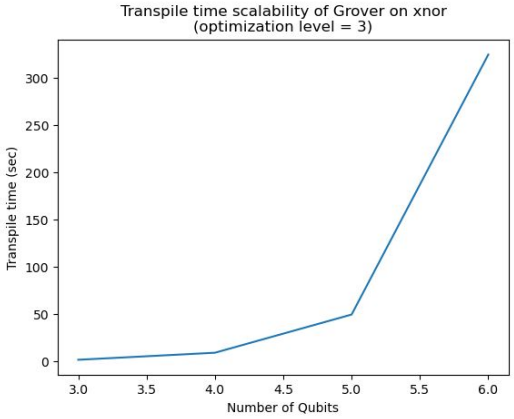
Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

XNOR Transpile Times









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

Simon (simon.py)

Design and Implementation of U_f

Determining U_f proved tricky initially, as we tried to follow a method similar to that of Deutsch-Jozsa and Bernstein-Vazarani, except with a U_f of size 2*n by 2*n rather than n+1 by n+1. However, we soon realized we had to generate 2^2*n permutations of possible input strings and not 2^n. We determined U_f by first computing f(x_a) + b for every possible input x, where x_a is the first n bits and b is the second n bits of the input x. Note that the first n bits are the same for the input and output, and the second n bits are determined by the mod2 sum of f(x_a) and b. Iterating through the list of all possible input values, we mapped each input and output to the index value of that iteration using two dictionaries. We initialized every element in U_f to 0, and then used the input and output pairs to determine which elements were one. The indices mapped to the input values represented the rows of U_f, and the indices mapped to the output values represented the columns. If a given input mapped to index ‘a’ is the output mapped to index ‘b’, then the element U_f[a][b] is 1. It was initially confusing to determine whether the inputs represented the column or row indices, but we worked out the logic by starting with n = 1 where U_f is only a 4 by 4 matrix.

Preventing Access to U_f

Similar to Deutsch-Jozsa, the end-user was prevented from directly accessing U_f because the matrix was dynamically constructed at runtime. Furthermore, when the circuit is drawn using the --draw option, it is the pre-transpilation version, preventing any of the internals of the construction of U_f from being revealed. Should the user want to construct a custom U_f, they will have to declare a function in func.py that meets the standards set in README. In particular, our program will check that the user-defined function only takes in one parameter (the bit string) as input.

Parametrization of Solution in n

The program is parametrized in n. The first step in ensuring this is to make sure that the black box functions we have defined in func.py do not depend on the input bit string being of a certain length. Then, our program takes in n as a parameter from the user at runtime, and passes this in to get_U_f to dynamically construct the matrix of the black box function, and to construct a QuantumCircuit with 2*n quantum registers (n of which are helper registers) and n classical registers.

Testing Methodology and Presentation of Results

We tested simon.py on four different functions. They are listed below, along with their corresponding s values:

1. two_to_one: s = 1010… (most significant bit is 1)

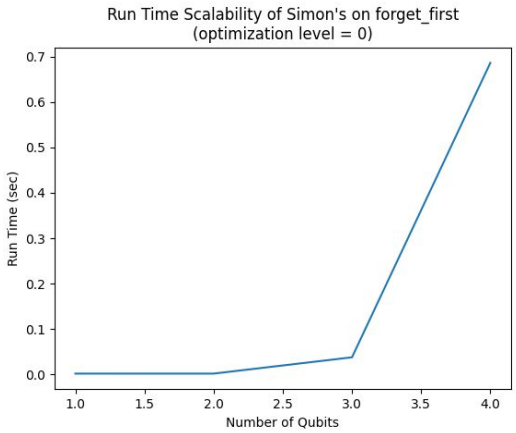
2. forget_first: s = 100...0
3. forget_last: s = 00...01
4. identity: s = 00...00

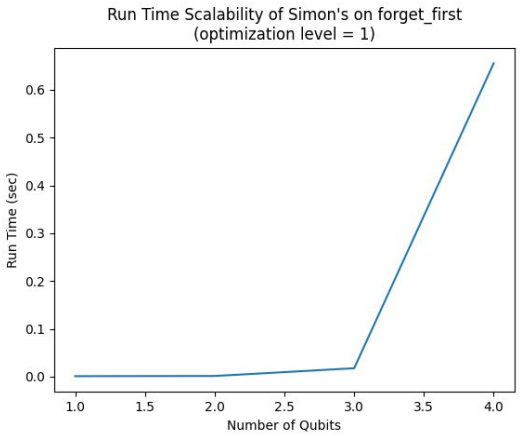
Since Simon's algorithm is not a deterministic algorithm (there is a chance that it fails to find s), we allowed a command line argument m, which determined how many times we generated new equations from the algorithm. Specifically, we generated 4*m*(n-1) equations, so that the probability of failure was: P(failure) < e^{-m}. 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 `['u1', 'u2', 'u3', 'cx']`, which, as we discussed in class, form the instruction set of the IBMQX5 quantum device on which we will likely run our quantum programs in the next couple of weeks. 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."

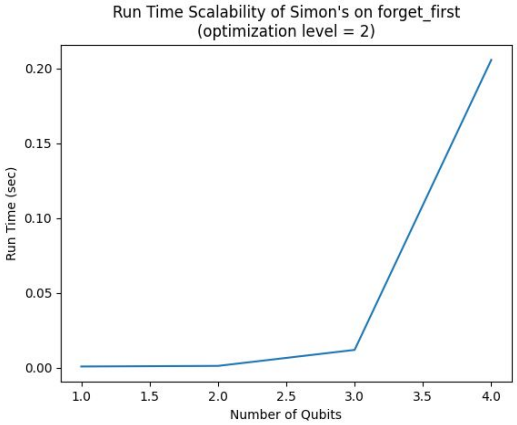
Scalability as n grows

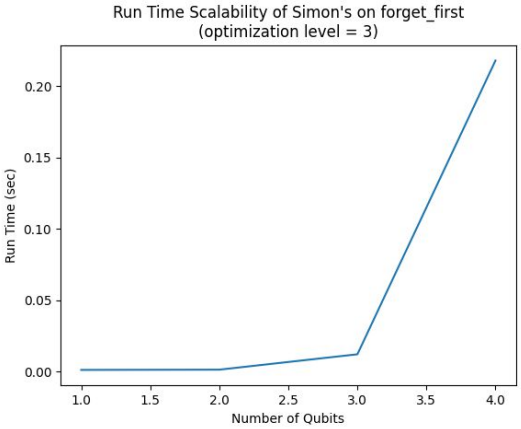
We noticed that, unlike for the other algorithms, the different functions all had similar transpile and run times to each other. Furthermore, it is clear that both transpile and run times increase exponentially as n grows. Also, as optimization level increases, transpile time increases and run time decreases, as expected.

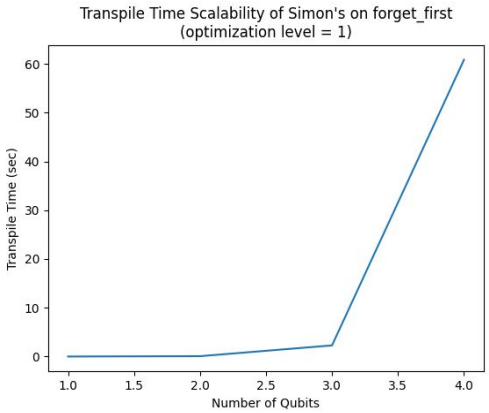
Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

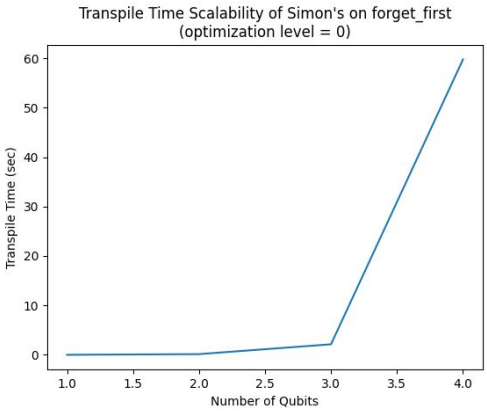




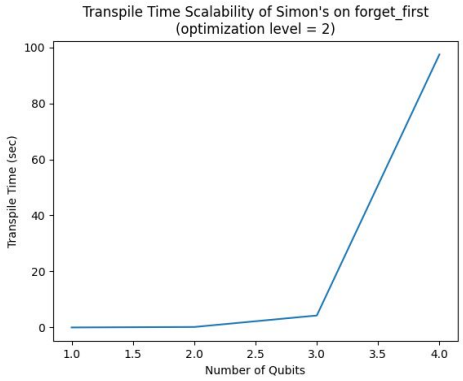


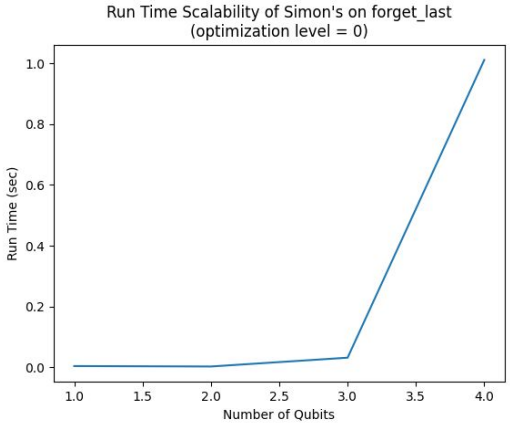


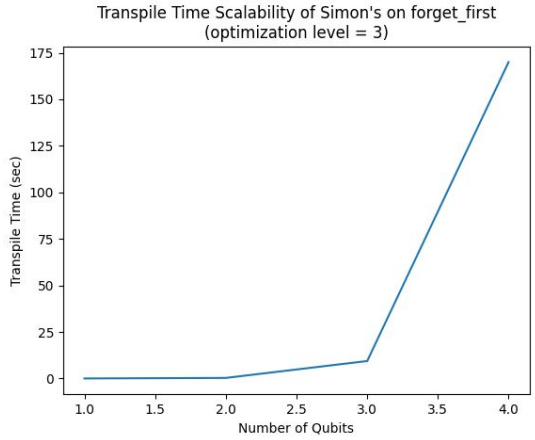


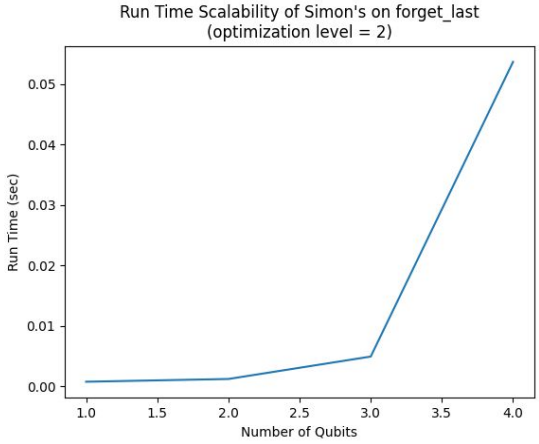


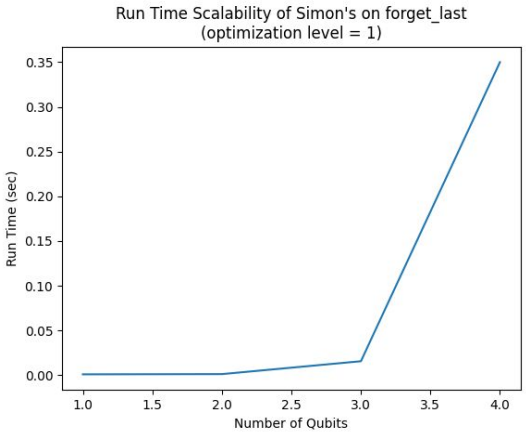
Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

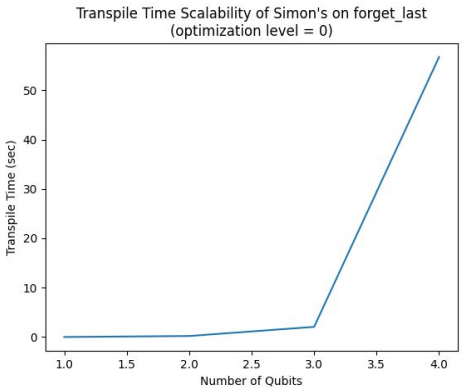




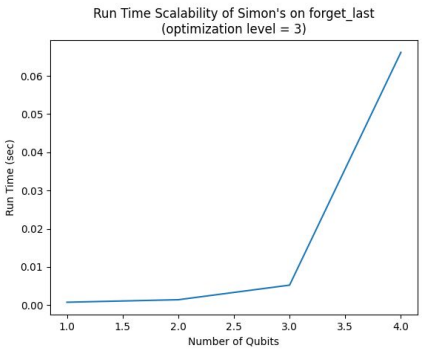


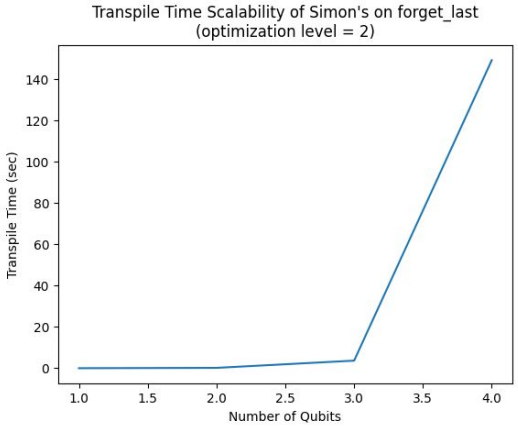


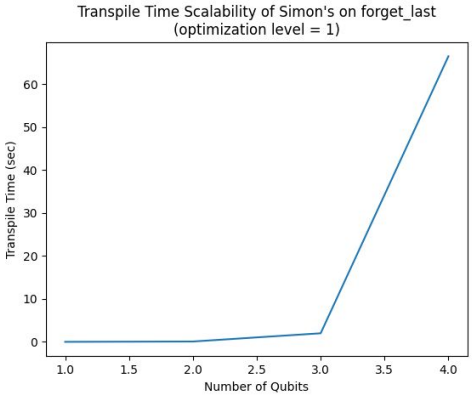


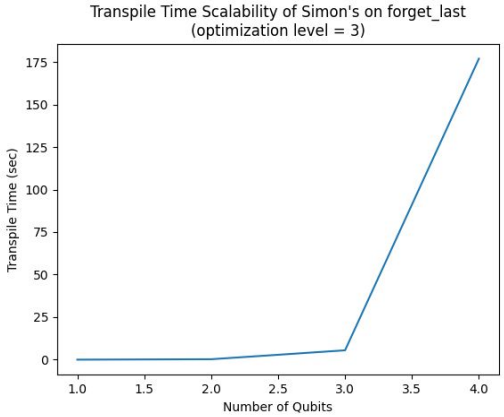


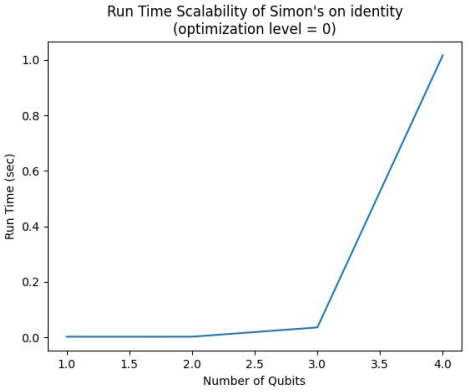
Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

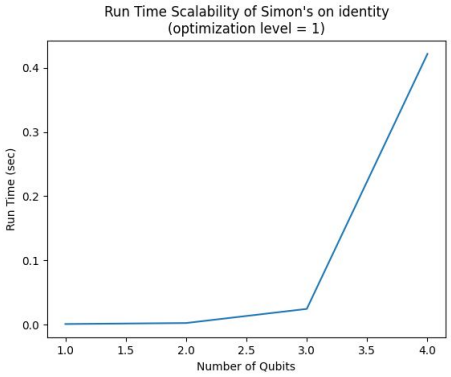




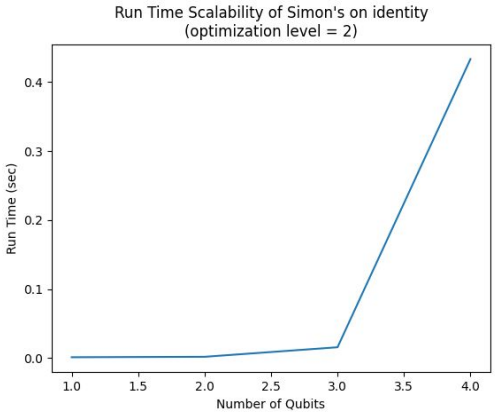


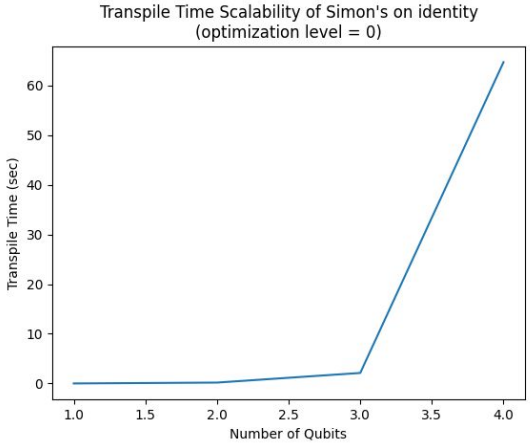


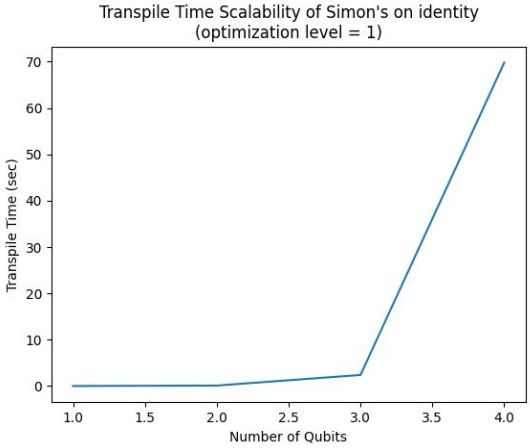


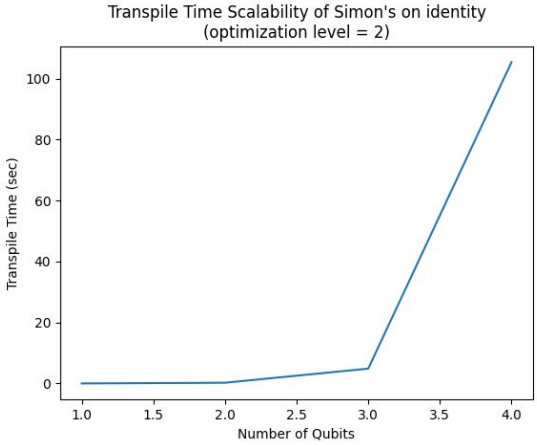


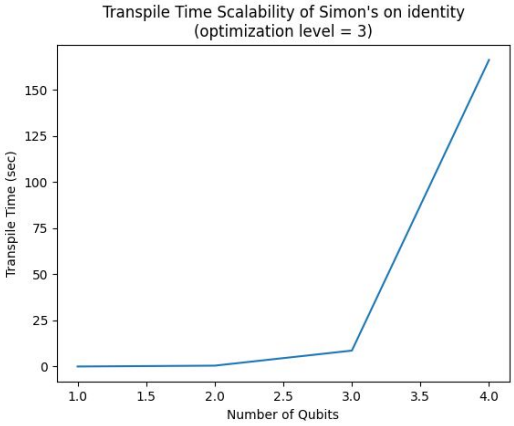
Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

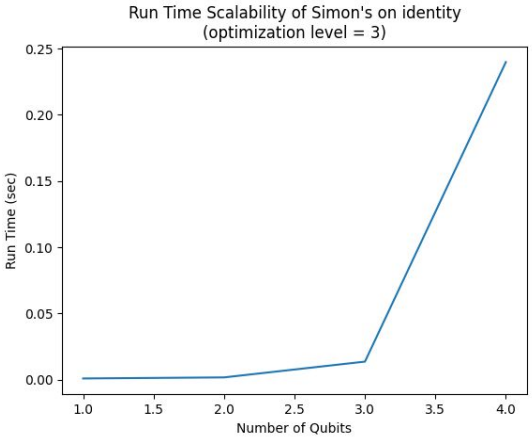




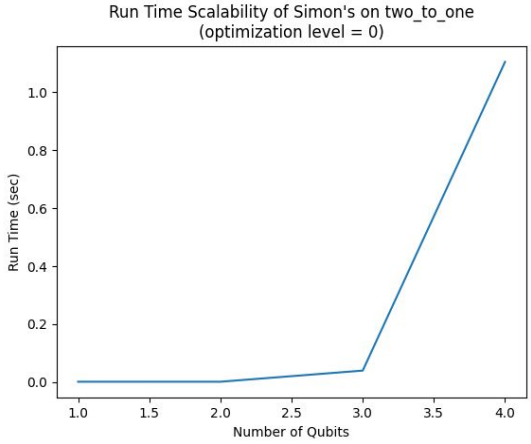


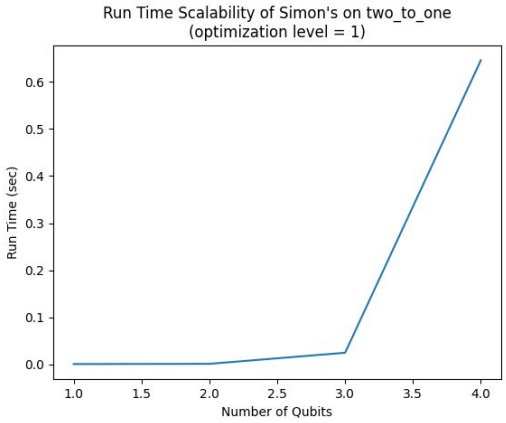


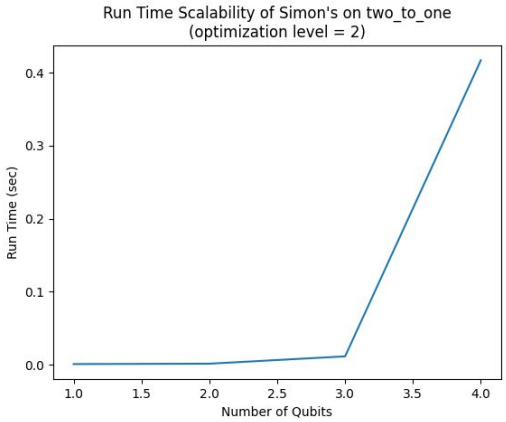


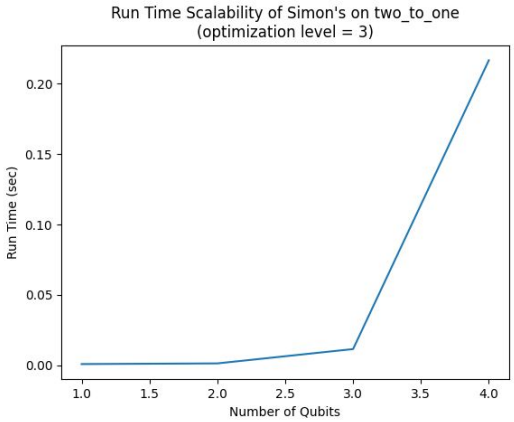


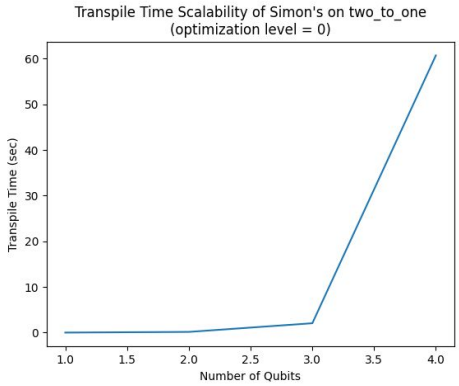
Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

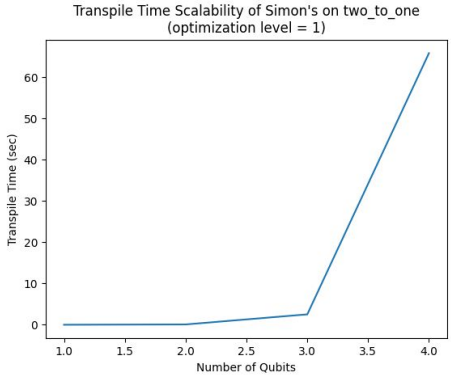




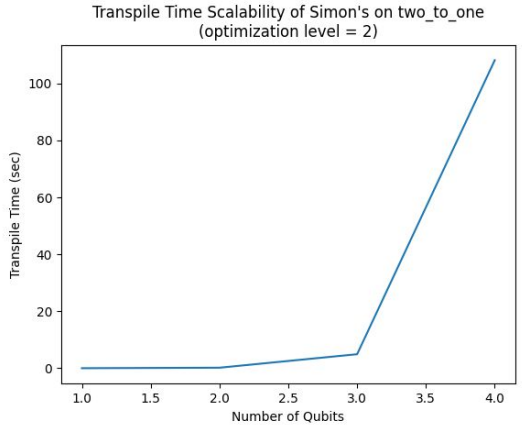


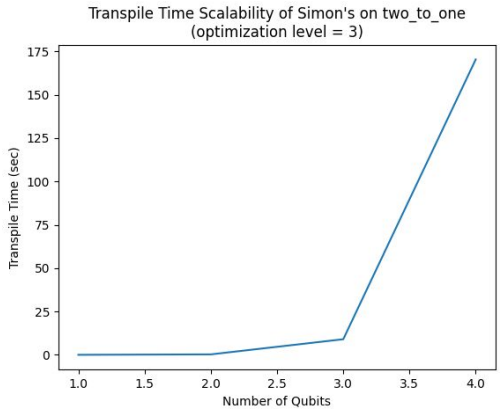


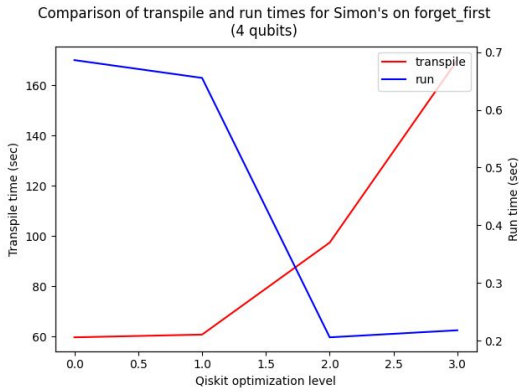


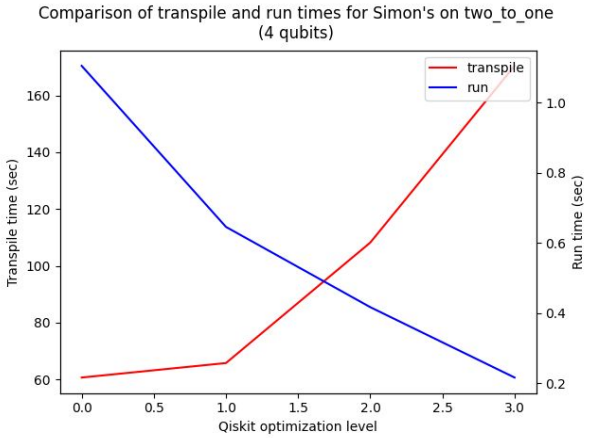


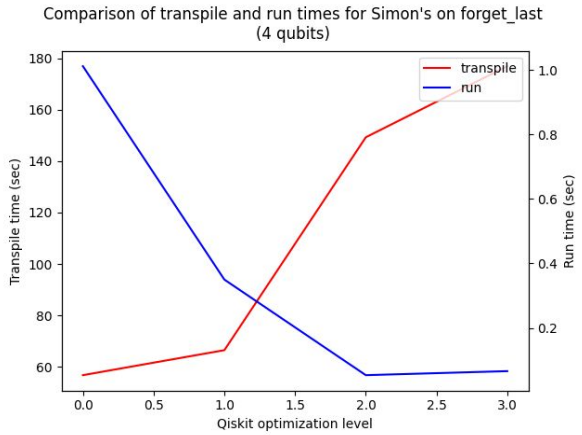
Arjun Subramonian, UID: 205105147
Vaishnavi Tipireddy, UID: 705143552
Siddarth Chalasani, UID: 705192236

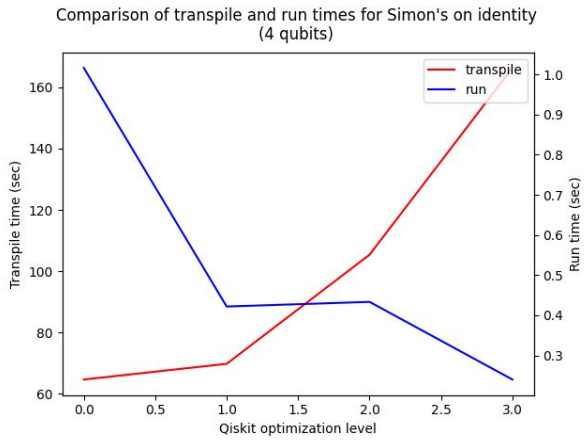












Overlap and Reuse of Code

There is significant overlap between the code for Deutsch-Jozsa and the code for Bernstein-Vazirani, since the structure of the quantum portion for both algorithms is identical.

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

Specifically, the shared code sections between deutsch_jozsa.py and bernstein_vazirani.py were the construction of U_f, given f(x), the entirety of the quantum circuit, as well as a majority of the test driver; the primary difference was the interpretation of the qubit measurements when printing results and the computation of b for Bernstein-Vazirani. In the ~200 lines of code in deutsch_jozsa.py and bernstein_vazirani.py, around 170 of them were shared, i.e., around 85% of the code for Deutsch-Jozsa and Bernstein-Vazirani was shared.

Despite the overlap, we did not modularize the shared quantum circuit for the reuse in both programs because these are the only two programs that share a significant amount of code, so it seemed excessive to do so. However, for the purposes of testing, we placed the test functions we used to evaluate the accuracy of the implementation of the algorithms in func.py, the same file in which the user defines custom functions to run the quantum programs on, since these test functions can be shared amongst the test drivers for the programs (e.g. the constant 0 function is a good test function for both Deutsch-Jozsa and Grover). Even though our test driver for each program made use of these shared test functions, the interpretation of the qubit measurement results being different for each driver made it hard to reuse code between the drivers. However, the code for the --graph option was almost completely reused between all the drivers, and its functionality could be better modularized.

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. 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 qubits, **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 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, and the run time for the entire test (including all the trials), and subsequently, for each result, presents 1) the frequency of the result, 2) the actual measurements of the n input qubits and 3) the interpretation of the measurements.

   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.

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

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 vs. number of qubits
      b) A plot of run time vs. number of qubits
   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 their default (1, 2, 3, 4). 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(failure) < e^{-5})

3) python3 quantum_program.py **fn_name** **--draw** **n**

   To take advantage of Qiskit's ability to print out the diagram of a quantum circuit, we added a **--draw** option that follows the above format, where **fn_name** is the function name, and **n** is the number of qubits. The program will then output a text diagram of the circuit, including the classical registers but *excluding the measure gates*. This is an example for Deutsch-Jozsa:

   python3 deutsch_jozsa.py one --draw 4

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

Qiskit Reflection

Aspects easy and difficult to learn

Some aspects of quantum programming in Qiskit that were easy to understand and use are the pre-implemented common gates (e.g. Hadamard, Pauli X), the two functions for measuring the qubits into classical registers and running the circuit, and defining a gate. As opposed to PyQuil, where we had to add a definition of the gate to the program before adding the actual gate instance, adding a gate in Qiskit was a one-step process. However, we did struggle in understanding how to add the gate to the circuit using the unitary operator. The documentation we found was very minimal, and there were few online forums (if any) which addressed the same question. Some other aspects of Qiskit that were difficult to learn are the classical registers that had to be explicitly specified when measuring. In addition, the qubit ordering convention in Qiskit was the reverse of the one we learned in class, and the one that is commonly used in textbooks, and was difficult to adjust to. In class, we used the qubit ordering convention that the leftmost qubit in Dirac notation (i.e. the most significant qubit) is the top qubit in the circuit and the rightmost qubit in Dirac notation (i.e. the least significant qubit) is the bottom qubit in the circuit. The implication of Qiskit's unconventional qubit ordering was that U_f, as we had constructed it, was no longer valid, as it had been for the PyQuil assignment. We initially considered re-designing how U_f is constructed but instead decided upon simply reversing the mapping of the qubits in the circuits to the inputs of U_f. This decision also necessitated the reversal of the mapping of qubits to classical registers during measurement: `circuit.measure(range(n), range(n - 1, -1, -1))`.

Aspects Qiskit does and doesn't support well

Qiskit supports circuit visualization and state vector printing really well. We found this feature especially useful when working with non-deterministic algorithms like Simon's and Grover's. Some other features Qiskit supported well are run time measurements which didn't have to be explicitly calculated, the ability to run individual qubit operations, and the histogram for frequencies of different results which was useful with error correction. We had no socket or timeout issues and also didn't have to key into individual qubit measurements per trial (we simply had to key into unique measurements for all qubits) when running multiple shots, both of which were an issue with PyQuil. Even without transpiling, the run times for the algorithms were really low which was very useful with initial testing and debugging. Since the QASM simulator dynamically supports as many qubits as we have memory for, we were able to run on more qubits for some of the algorithms in Qiskit than we were able to with PyQuil without having to specify a QPU with support for more qubits.

A feature Qiskit didn't support as well is accessing individual qubits, which makes the language less generalizable to a variety of quantum applications. We found comparing the run time and transpilation time to be a useful tool in assessing the performance of our implementations, but it

was difficult to graph the two variables together over various optimization levels since their time range is very different. It would be helpful to have a simpler way to graph the two performance indices on one chart. Just like with Pyquil, initializing qubits to be nonzero can also be made easier, as now it has to be done through applying the X gate. One of our team members experienced a kernel panic, because of which even their keyboard and trackpad processes were blocked, because of running an algorithm using 20 qubits, which was too many. When they tried to run with 15 qubits, zsh automatically killed the program. Qiskit doesn't notify the user when too many qubits are being requested given their available program memory, which would have been a time-saving feature to have.

Features of Qiskit we want to see

Some features we would like to see Qiskit support are access to the measurements of individual qubits and a way to measure transpile time separately from run time in qiskit.execute. The former would make presenting the results for individuals trials of the algorithms much easier and the latter would allow us to better investigate the power of transpiler optimization to reduce long run times for a large number of qubits with having to call qiskit.compiler.transpile separately. Furthermore, we were initially confused by the "reversed" qubit ordering convention Qiskit uses; a change in qubit order would make programming across the different quantum languages easier. We discovered a couple of forums online where quantum programmers are already requesting this change (e.g. https://github.com/Qiskit/qiskit-terra/issues/1148).

Thoughts on Qiskit Documentation

The installation instructions were clear and installing Qiskit was overall an easy process because we didn't need to download forest SDK zip and unzip like we had to for PyQuil. We found Qiskit tutorials on topics from introduction to circuits all the way to Grover optimization, which encourages people who are beginners in quantum programming to learn and use the language. The gate documentation was also more beginner-friendly (https://qiskit.org/documentation/tutorials/circuits/3_summary_of_quantum_operations.html), unlike for PyQuil where there was a comprehensive list of pre-implemented gates but not much documentation for defining gates. We really liked the documentation on QASM (noisy), StateVector (ideal), and unitary simulators because it was extremely thorough and full of examples.

On the other hand, there isn't much documentation for adding a custom unitary gate: https://qiskit.org/documentation/stubs/qiskit.circuit.QuantumCircuit.unitary.html. Qiskit has really nice textbook for quantum programming which we didn't discover until a while after we started this project. It would be very helpful if the textbook was referenced throughout the documentation. Furthermore, it could be difficult to locate the documentation pages detailing API interfaces instead of actual Qiskit source code when searching for how to use different Qiskit functions online. Lastly, the documentation for libraries like qiskit and qiskit.compiler could

be more organized, utilizing more subsections, and cleaned up. For example, optimization_level was not well-defined in the description of qiskit.execute, and we had to locate the documentation for qiskit.compiler.transpile to figure out what the levels 0-3 meant. Apart from these comments, we were very happy with the documentation provided for Qiskit.

Qiskit Dictionary

The only aspect of Qiskit we found to be really different was the qubit ordering convention, as we described in the *Aspects easy and difficult to learn* section. Otherwise, all the conventions used in Qiskit aligned well with the conventions we used in class and those in the Hidary textbook. For instance, unlike PyQuil which had the following dictionary:

{circuit : program,
simulator : quantum virtual machine (qvm),
9-qubit simulator : 9q-square-qvm,
shots : trials}

Qiskit actually used all terms on the left-hand side (the keys).

Type-checking

Our Qiskit implementation doesn't do a lot of type-checking because the library handles most of the type-checking for us. When passing data from the classical side to quantum, we ensure that the classical function f only takes in one parameter using the Python library function inspect.signature. Our implementation guarantees a unitary matrix U_f regardless of fby design of the algorithms. Moreover, trying to add a non-unitary gate to circuit results in: qiskit.extensions.exceptions.ExtensionError: 'Input matrix is not unitary.' Qiskit guarantees classical measurements are either 1 or 0, so no type-checking is required from the quantum to classical side.