Debugging Quantum Circuits with Breakpoints

Palvit Garg
Department of Computer Science
North Carolina State University
Raleigh, NC, USA
pgarg5@ncsu.edu

Sirajus Salekin
Department of ECE
North Carolina State University
Raleigh, NC, USA
ssaleki@ncsu.edu

Harshwardhan Joshi
Department of Computer Science
North Carolina State University
Raleigh, NC, USA
hjoshi2@ncsu.edu

***Abstract*—We investigated the feasibility of implementing quantum circuit debugger with breakpoints that can run on quantum hardware instead of simulators. We found several challenges over the course of our investigation. In this paper, we discuss the various theoretical approaches, relevant experiments, and our findings. We selected the two most feasible approaches and have implemented one of the approaches successfully. To implement the other approach, a few open problems need to be solved that are beyond the scope of this study; however, they can be part of future work.**

I. Introduction

The aim of this project is to introduce breakpoints in quantum circuits that runs on quantum hardware. We hoped be able to give the user the ability debug a quantum circuit similar to how one debugs a classical circuits with breakpoints.

In classical computing, debugging with breakpoints means halting the program execution at any given point and freezing the program state. This allows programmers to examine the program internals mid-execution (as in, what variables contain which values, analyze control flow etc.). However, doing so in a quantum circuit brings the first challenge: measuring a quantum circuit collapses the superposition of the qubits and all entanglement is lost. Thus, it will be impossible to resume the execution of the same circuit after any kind of measurement is performed. Due to the no-cloning principle, it is also not possible to make a copy a qubit, consequently a quantum circuit. The only metric we have after measuring the circuit is the probability distribution of all possible states the qubits can be in. However, this probability distribution is not enough to reverse engineer the qubits into the states they were in before the measurement was performed. Unlike a classical bit, a quantum bit has a phase in addition to a value. While the superposition of a qubit can be reconstructed with some errors from the probability distribution, the phase data is lost after a measurement. Therefore, the primary challenge boiled down to working out a way to preserve that phase and reinitialize the qubits with appropriate value to resume execution in an equivalent circuit.

II. Related Work

Earlier methods to introduce breakpoints were successfully implemented on quantum circuit simulators. The simulators do not introduce any noise into the circuit. However, running the same circuits on real hardware and measuring it proves to be very challenging. The major problems are the introduction of noise, loss of superposition and entanglement, loss of phase, and the inability to recreate the quantum states with the limited amount of information gathered while measuring the qubits. As such, we didn't find any work directly associated with the problem of debugging on real hardware.

III. Design

We came up with two viable approaches to the problem.

A. First Approach: Simulators and Hardware

The first approach is a combination of simulator and actual quantum hardware. In this approach, we run the quantum circuit on real hardware until a breakpoint i.e. circuit part A, and show the measurement results to the user with a disclaimer of the possibility of noise in the results. At the same time, we also run the same circuit until the breakpoint i.e. part A on a unitary simulator. The unitary matrix received is then combined with the circuit after the breakpoint i.e. Unitary + B. This way a new circuit is created but instead of part A we replaced it with the unitary matrix. Further, the new circuit can be run on hardware until the next breakpoint. This process can be extended to any number of breakpoints.

This approach is almost noise-free, in terms of intermediate measurements i.e. no additional noise is carried over because of the breakpoints. Of course, the noise will be there in measurement results because of the hardware, but the unitary matrix from the simulator makes sure that this noise does not transfer to the next measurement.

The biggest limitation of this approach is it cannot be extended to any number of qubits because the unitary matrix grows exponentially with the number of qubits in the circuit. Thus the approach is good for small circuits with fewer number of qubits.

The following diagram represents the flow of our circuit.

Run 1 (Hardware) : Run the circuit upto the breakpoint

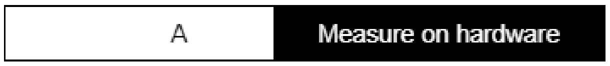


Fig. 1: Run 1

Run 2 (Unitary simulator) : Run the circuit parallelly upto the breakpoint and create a unitary matrix.

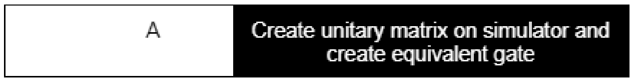

A
Create unitary matrix on simulator and create equivalent gate

Fig. 2: Run 2

Run 3 (Hardware) : Convert the unitary matrix that we got from run 2 to an equivalent gate. Then append this gate to the hardware device, and then run circuit part B after the gate.

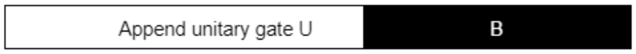


Fig. 3: Run 3

Overall circuit flow :

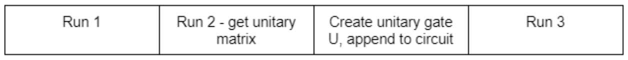
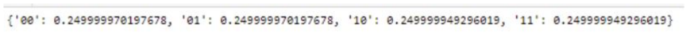

Run 1
Run 2 - get unitary matrix
Create unitary gate U, append to circuit
Run 3

Fig. 4: Overall run

B. Second Approach: Hardware

The second approach is only based on hardware. The approach is based on assumption that we can recreate the state vector of qubits from hardware measurement. This approach follows below steps: (Note: The example results are shown from this notebook [1])

1. Run the quantum circuit until the breakpoint on the hardware and get the probability distribution of the output state.



```
{'00': 0.249999970197678, '01': 0.249999970197678, '10': 0.249999949296019, '11': 0.249999949296019}
```

Fig. 5: Probability distribution

2. Using this, compute the amplitude of the state vector i.e. {00: 1/2, 01: 1/2, 10: 1/2, 11: 1/2}.

3. Further, to compute the final state vector, we need to preserve the exact relative phase for each output. An embedded QPE (Quantum Phase Estimation) circuit seems like a feasible approach to achieve this step. The phase will look something like this:

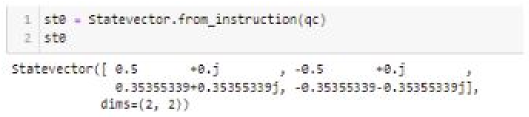
{00: 0 deg, 01: 180 deg, 10: 45 deg, 11: -135 deg (225 deg)}

4. This relative phase must be used to compute the state vector as below:

00	1/2(cos(0 deg) + sin (0 deg) * i)	0.5 + 0i
01	1/2(cos(180 deg) + sin (180 deg) * i)	-0.5 +0i
10	1/2(cos(45 deg) + sin (45 deg) * i)	0.35 + 0.35i
11	1/2(cos(-135 deg) + sin (-135 deg) * i)	-0.35 -0.35i

TABLE I

5. The final state vector will look like the following figure:



```
st0 = Statevector.from_instruction(qc)
st0
```

```
Statevector([ 0.5       +0.j        , -0.5       +0.j        ,
              0.35355339+0.35355339j, -0.35355339-0.35355339j],
            dims=(2, 2))
```

Fig. 6: State Vector

6. Now this state vector can be used to prepare the state for this first part run until the breakpoint i.e. part A as shown in fig 1. Thus, we don't need to rerun this part of the circuit, because the previous state collapsed. We can just initialize the qubits with this state vector and then start with part B directly.

The biggest challenge with this approach is to find the relative phase. In theoriy, QPE seems like a useful accessory to this end. However, it has its own limitations, the primary one being the number of qubits required to achieve high precision. Thus more optimization options need to be explored in this direction.

Another issue with this problem is the noise introduced at each step. Since the phase estimation is not accurate and real hardware can also introduce noise, so the additional noise will be introduced at each breakpoint. We will discuss these issues in-depth next.

Theoretical Analysis of Noise Introduced in Circuit via QPE

At each breakpoint, QPE will estimate the phase. Let's assume we use 4 qubits for precision. Thus precision of the measured phase will be 1/16. We cannot measure any value between e.g. 0 to 1/16. Thus for any intermediate phase, it will estimate to be 0 or 1/16. The worst-case error introduced is 1/32 at the first breakpoint. Once this erroneous phase is used to compute the state vector and applied, the phase error will transmit up to the next breakpoint. Again next breakpoint will have the same precision for the phase. So phase error can either cancel out, in the best case, or add up in the worst case. i.e. the error will be in between 0 (1/32 -1/32) to 1/ 16 (1/32 + 1/32). Using the mathematical induction, we can conclude that phase error after n breakpoints using 4 ancilla qubits will range from 0 to $1/32 * 2^{n-1}$

Generalizing it a bit more, for m ancilla qubits the precision will be $1/2^m$. The error at the first breakpoint will be between 0 to $1/2^{m+1}$.

The error range after n breakpoints while using m ancilla qubits is:

$$0 \ to \ 1/2^{m+1} * 2^{n-1}$$

The mathematics derived is just theoretical, it has not been experimentally verified.

IV. Implementation

We implemented the hybrid approach that uses both the simulator and hardware with IBM's Qiskit API. In light of our discussion of the earlier section, the primary motivation for choosing this approach was keeping error propagated due to noise at a constant level.

The core of the debugger consists of two classes, namely, `QuantumDebugCircuit` and `QCDebugger`. `QuantumDebugCircuit` is a simple wrapper around Qiskit's `QuantumCircuit` class, with an implementation of a `bp()` method (short for breakpoint). Behind the scenes, whenever this method is called, it keeps track of those points in the circuit. It does so in the form of indices in the list of qiskit `Instruction` objects. One can add as many breakpoints as one desires to this circuit. One key limitation is the circuit cannot contain any mid-circuit or final measurements since we cannot generate a unitary matrix from a such a circuit.

While implementing this debugger, we focused on usability and compatibility with existing code. As such, any existing quantum circuit can be converted into a "debuggable" circuit by simply changing the initialization line from `QuantumCircuit` to `QuantumDebugCircuit`

The other class `QCDebugger` acts similar to how one expects GDB-like debuggers to behave. The `QCDebugger` is initiated by passing the `QuantumDebugCircuit` as a parameter. Then the user can run the circuit between a pair of breakpoints and watch intermediate results until the circuit finishes running. We implemented this functionality within the `QCDebugger.c()`(gdb's continue equivalent) method. The `QCDebugger` accomplishes this by generating sub-circuits from the original circuits based on the breakpoints set earlier within the circuit object.

In the following code snippet, we will illustrate the usage of debugging an example circuit with `QCDebugger` and `QuantumDebugCircuit`.

```
# example_breakpoint_debug_circuit.py
from qiskit_debugger import
    QuantumDebugCircuit
from qiskit_debugger import QCDebugger

# Initiate a debuggable circuit
qc = QuantumDebugCircuit(2)

qc.x(0)
qc.h(range(2))
qc.cx(0, 1)
qc.h(range(2))
qc.bp()   # <-- Add a breakpoint here
qc.h(range(2))
qc.x(range(2))
qc.bp()   # <-- Add a breakpoint here
qc.cx(1, 0)
qc.h(range(2))

# Initiate a debugger with the circuit above
qdb = QCDebugger(qc)
qdb.c() # step through the breakpoints
# This will print the probability
    distributions
qdb.c() # continue to next breakpoint
qdb.c() # continue to next breakpoint
qdb.c() # continue next breakpoint
# Once all the breakpoints are done,
# the user gets a message: circuit finished
    running
```

V. Experimental Setup

For the first experiment, we tried to get a unitary matrix from the hardware run. It was intended that the said unitary matrix would assist in resuming the circuit. We appended the unitary matrix at the breakpoint in order to make the qubits go back to the state in which they were immediately before measurement. However, we were unable to retrieve a unitary matrix from the hardware run. Then researching through various sources we found it is not possible.

In the second experiment, we tried to verify that if qubits are initialized with the correct state vector then, they can be used to resume the circuit. This approach seems feasible. But the next challenge was to prepare the state vector. The probability distributions were obtained by measuring the circuit. [1] From the probability distribution, we were able to derive the amplitude of the state vectors. But to compute phase we need some algorithm. We experimented with QPE for 2 qubits [1] and 1 qubit [8]. Both experiment results were not as per expectations. Thus they are still open problems for this project and once the phase is estimated correctly it can be used as per design approach 2.

In our final experiment, we tried to run the same quantum circuit in-parallel on a simulator. The idea was to let the circuit run until the breakpoint both on the hardware and simulator. On the simulator run, we would be able to get the unitary matrix. We would then use this unitary matrix and use it to re-create the qubit states on the hardware device. From this point, the process would continue until the next breakpoint.The intuition behind this idea was that we would get the same results from the simulator, however, due to noise on real devices, the final outputs were a little bit deviated. [7]

In order to verify that our debug circuit works as intended, we used a list that contained simple quantum circuits and well-known ones. Doing so allowed us to trivially calculate or figure out the intermediate steps within those circuits, set up breakpoints, and confirm whether the implementation of the debugger matches our expectation.

VI. Results

The above experiment was conducted on multiple circuits. We will use a circuit that implements the Grover's search algorithm with two qubits to discuss the results. [9] The target string for the search algorithm is '11'. The circuit given below was divided into 3 parts using breakpoints. First we manually calculated the probability distributions of the circuit at each specific breakpoint, and then the circuit as a whole. We ran the circuit through the debugger and stopped at each breakpoint, and cross checked the intermediate measurements against our manual calculations. Given below are the circuit results. [10]

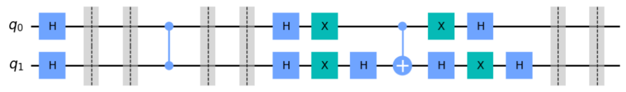


Fig. 7: the complete circuit

1. At the first breakpoint, the circuit will look as follows:

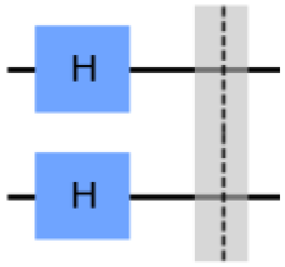
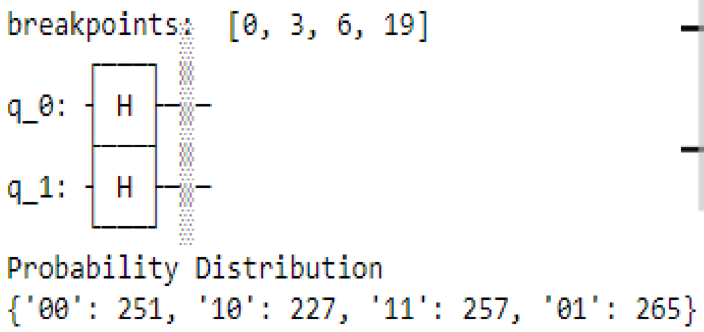


Fig. 8: From start to breakpoint 1

The expected result upto this breakpoint is an equal superposition of the states 00, 01, 10 and 11. Following is the result shown by the debugger:



```
breakpoints:  [0, 3, 6, 19]
     ┌───┐ ░
q_0: ┤ H ├─░─
     ├───┤ ░
q_1: ┤ H ├─░─
     └───┘ ░
Probability Distribution
{'00': 251, '10': 227, '11': 257, '01': 265}
```

Fig. 9: result for part 1

2. At the second breakpoint, the circuit will look as follows:

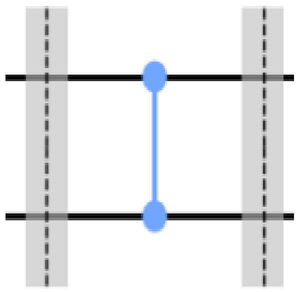
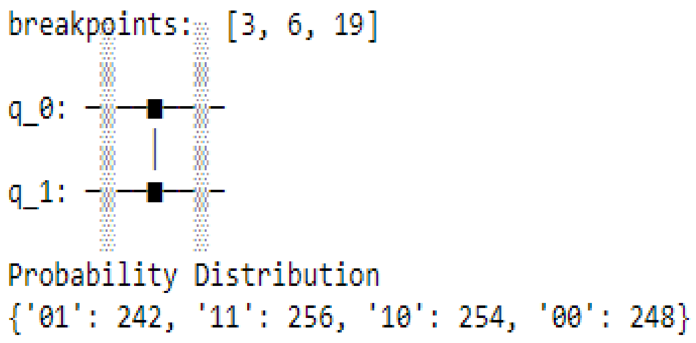


Fig. 10: From breakpoint 1 to breakpoint 2

The expected result upto this breakpoint (which is a swap gate) is the same as above. Following is the result shown by the debugger:



```
breakpoints:  [3, 6, 19]
      ░     ░
q_0: ─░──■──░─
      ░  │  ░
q_1: ─░──■──░─
      ░     ░
Probability Distribution
{'01': 242, '11': 256, '10': 254, '00': 248}
```

Fig. 11: result for part 2

3. The third part of the circuit is the diffuser, which should return the target string '11'. At this stage, the circuit will look as follows:

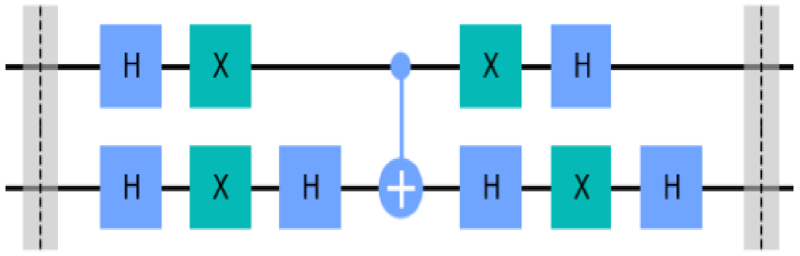
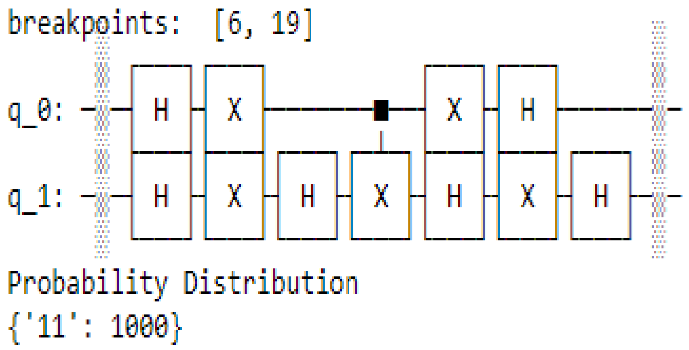


Fig. 12: From breakpoint 2 to breakpoint 3

Hence, the expected value is '11'. Below is the value calculated by the debugger:



```
breakpoints:  [6, 19]
      ░ ┌───┐┌───┐                ┌───┐┌───┐      ░
q_0: ─░─┤ H ├┤ X ├──────────■─────┤ X ├┤ H ├──────░─
      ░ ├───┤├───┤┌───┐┌───┐│┌───┐├───┤├───┤┌───┐ ░
q_1: ─░─┤ H ├┤ X ├┤ H ├┤ X ├┤ H ├┤ X ├┤ H ├─░─
      ░ └───┘└───┘└───┘└───┘└───┘└───┘└───┘ ░
Probability Distribution
{'11': 1000}
```

Fig. 13: result for part 3

4. Now, we run the full circuit without the debugger, and see that the entire circuit gives us the same value that we obtained at the end of breakpoint 3.

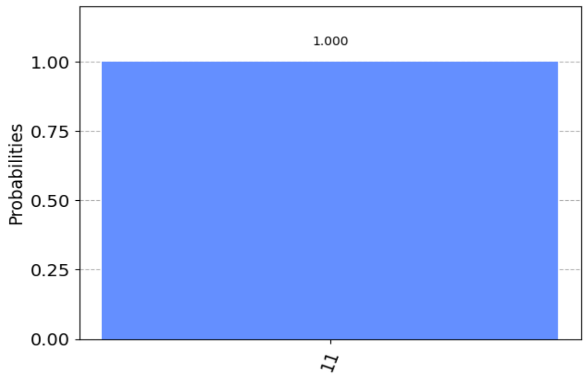


Fig. 14: result for part 3

A. Error Propagation

We have compared results from both the simulator and hardware runs to see how much the hardware run deviates from the ideal situation. Introducing breakpoints in a quantum circuit introduces a lot of errors, and resuming a circuit can throw off the final intended result. Below is a comparison of the probability distributions that we get from our hardware and simulator runs:

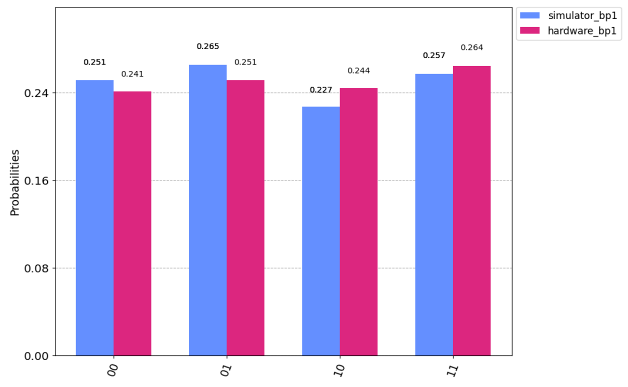


Fig. 15: Comparison between ideal(simulator) and hardware till the first breakpoint

The figure above compares the probability distribution at the first breakpoint. There are slight differences between these two runs, which can be attributed the noise seen on any given experiments ran on the hardware. As we have discussed earlier, a purely hardware-based approach we would accumulate errors linearly with the number of breakpoints. In contrast, our expectation with the hybrid approach keeping error propagation bounded within the same sub-circuit is verified with the experimental data.

The figure below shows the final measurements between the circuits with debugger and the same circuit without using the debugger, and both of these experiments were run on IBM Quantum hardware. It is obvious from the figure below that the noise accumulated at the final measurement is at a similar level to the ones seen at earlier breakpoints. Therefore the final results do not contain accumulated noise from previous breakpoints. The graphs for each of the remaining breakpoints can be found here [12].

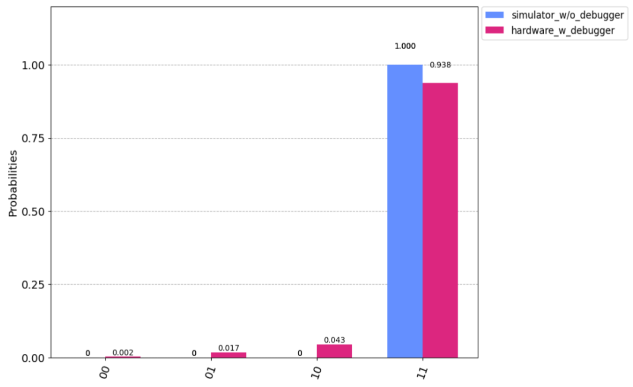


Fig. 16: Comparison between running the full circuit without the debugger and with the debugger

VII. Conclusion

We have successfully implemented the first approach which is a combination of simulation and hardware results. Using unitary matrices from unitary simulator runs and actual output from hardware runs, we were able to correctly re-create the qubit states pre-measurement, and were able to successfully resume the circuit after the breakpoint. Our implementation is readily usable as well. Our expectation with limited erorr propagation with this approach is also verified. We conclude that, this approach can aid the quantum computing programmers in interacting with and debugging their code in a more effective way, as long as the circuit width isn't too big.

On the other hand, we found that the pure hardware approach is cumbersome and expensive to implement, circuit-width wise. The key driver behind this approach is Quantum Phase Estimation, and while there are a number of algorithms uses this sub-routine, in practice it is a) not very precise, b) hard to implement, and c) limits the number of breakpoints to a handful due to error propagation.

VIII. Future Work

For the hardware approach, is worth investing how to extract phase information of a quantum circuit in a more effective way, with or without QPE. Generating an accurate statevector from noisy probability distribution and phase information is another avenue of further interesting research.

For the hybrid approach, it would be useful to investigate the limitation imposed by exponential growth of the unitary matrix based on the number of qubits. It is the more promising approach and thus worth the effort to make it scalable.

IX. Acknowledgements

We would like to thank Prof. Dr. Frank Mueller of NCSU Department of Computer Science for his constant help over the course of this project.

References

[1] https://github.ncsu.edu/ssaleki/qc_besteffort_breakpoint_debugging/blob/master/QPE%20experimentation.ipynb

[2] S. Metwalli, R. Meter, ”A Tool for Debugging Quantum Circuits”, https://arxiv.org/pdf/2205.01899.pdf

[3] https://qiskit.org/documentation/tutorials/circuits_advanced/02_operators_overview.html

[4] https://qiskit.org/documentation/stubs/qiskit.converters.circuit_to_instruction.html

[5] https://github.ncsu.edu/fmuelle/qc19/tree/master/hw/hw6/m3/csc591-quantum-debugging

[6] https://quantumcomputing.stackexchange.com/questions/27064/how-to-get-statevector-of-qubits-after-running-quantum-circuits-on-ibmq-real-har

[7] https://github.ncsu.edu/ssaleki/qc_besteffort_breakpoint_debugging/blob/master/Rerun%20circuit%20using%20Unitary%20matrix.ipynb

[8] https://github.ncsu.edu/ssaleki/qc_besteffort_breakpoint_debugging/blob/master/Approach%232%20phase%20estimation%20error.ipynb

[9] https://www.qmunity.tech/tutorials/grovers-algorithm-using-2-qubits

[10] https://github.ncsu.edu/ssaleki/qc_besteffort_breakpoint_debugging/blob/master/benchmark_grover_2-qubits.ipynb

[11] https://github.ncsu.edu/ssaleki/qc_besteffort_breakpoint_debugging/blob/master/benchmarks/benchamrk_grover_2-qubits_hw.ipynb

[12] https://github.ncsu.edu/ssaleki/qc_besteffort_breakpoint_debugging/blob/master/benchmarks/simulator_vs_hw_benchmar_comparison.ipynb