Farnaz Nouraei | Advanced Digital Design

Project Report

January 7, 2020

1 Problem Statement

In May 2016, IBM made its superconducting quantum computer available to members of the public via the cloud. People could access and run experiments on IBM's quantum processor. IBM called the service the IBM Quantum Experience. The quantum processor is composed of five superconducting qubits and is housed at IBM's Thomas J. Watson Research Center.

In early 2018, IBM started challenges to encourage people to take advantage of the IBM Q Experience and the IBM QISKit development platform [1]. One of them was the IBM QISKit Developer Challenge, which asked for a compiler that maps circuits composed of random operations from SU(4) to a certain quantum hardware topology. But why did they need a mapping in the first place?

In every physical realization of quantum computers, certain operations can only be conducted on certain pairs of physical qubits. A coupling map, an example of which is shown in Figure 1, defines what pairs of physical qubits can actually interact in operations realized on a quantum processor architecture.

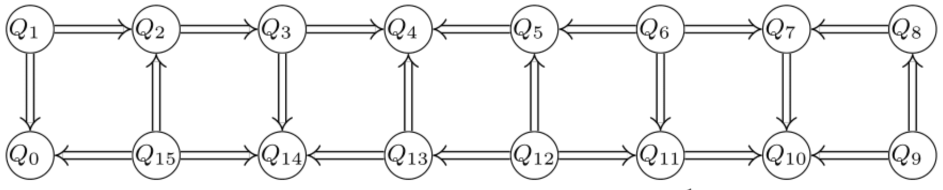


Figure 1: Coupling Map for QX5

In order to respect these constraints, usually a number of SWAP operations are applied in certain points of a quantum circuit. SWAP gates exchange the state of two physical qubits. This way the information is swapped between two logical qubits, so the desired operations can be realized. Figure 2 illustrates the process: On the left, the desired quantum circuit is shown which includes several operations over pairs of qubits which are not able to interact according to the coupling map in Figure 1. But adding SWAP operations as shown on the right, moves all the qubits to positions so that they eventually are in line with the restrictions imposed by the coupling map.

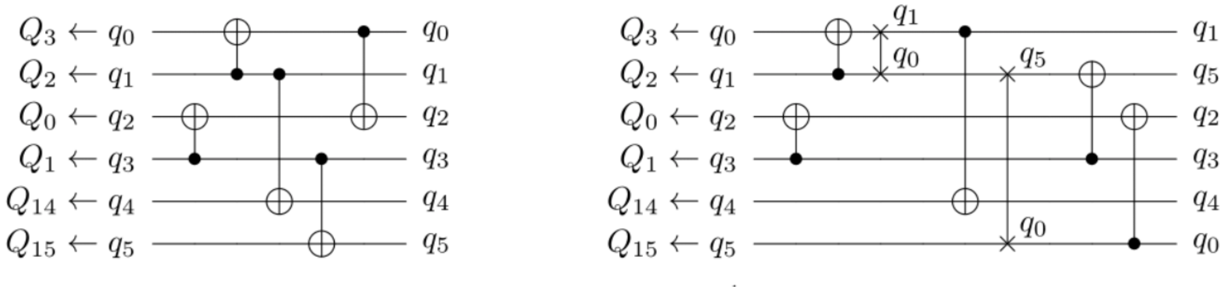


Figure 2: Applying SWAPs to the original circuit (left) to make it a circuit realizable on QX5 (right)

However, inserting the additional gates in order to satisfy the constraints drastically increases the number of operations. Each gate has a certain error rate hence this huge number of additional gates will introduce

too much error to the overall functioning of the circuit. The reason for this drastic increase is given in [2] as follows:
"Since the SWAP operation is not part of the gate library of IBM's QX architectures, it has to be decomposed into single-qubit gates and CNOTs as shown in Fig. 3. Assume that logical qubits q0 and q1 are initially mapped to the physical qubits Q0 and Q1 of QX2, and that their values are to be swapped. As a first decomposition step, we realize the SWAP operation with three CNOTs. If we additionally consider the CNOT-constraints, we have to flip the direction of the CNOT in the middle. To this end, we apply Hadamard operations before and after this CNOT. These Hadamard operations then have to be realized by the gate $U(\pi/2, 0, \pi) = H$."

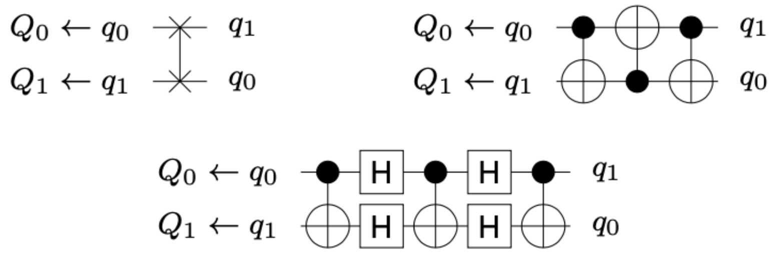


Figure 3: The decomposition of SWAP gates into elementary quantum gates

Hence we know that each SWAP operation is composed of 7 elementary gates (Figure 3), so we want to keep their number as small as possible. Moreover, the circuit depth should also be small since it determines the runtime of the quantum circuit. A SWAP operation alone has a depth of 5, hence the motivation for finding an efficient optimized circuit becomes even more prominent.
Given all the above, it is crucial for quantum circuit designers to have a compiler which would do both jobs of breaking down the desired functionality to elementary operations, and find a mapping of the logical qubits from the circuit to the physical qubits available in the actual quantum computer. And the latter is what we are focusing on in this project.

2 Related Work

One of the teams who have been working intensely to bridge the two areas of EDA and quantum computing, are Alwin Zulehner, Alexandru Paler and Robert Wille from Johannes Kepler University Linz, Austria. In several papers such as [2], they have addressed the aforementioned optimization problem in two steps: *"First, the given circuit is partitioned into layers which can be realized in a CNOT-constraint-compliant fashion. Afterwards, for each of these layers, a respectively compliant mapping is determined which requires as few additional gates as possible."* For each layer, an A^* search algorithm is run to find the optimal swap insertions; the sequential mapping of layers to the optimal solution will result in a globally-optimal solution for the whole circuit. Their method is much faster and better than IBM's approach and only requires up to several minutes on 16-qubit circuits; however, searching all possible combinations of concurrent SWAP gates still requires exponential time. Their initial mapping was determined by only those two-qubit gates at the beginning of the circuit without global consideration.
In another work [3], Li et. al. at UCSB have proposed another mapping method which uses a SWAP-based BidiREctional heuristic search algorithm (SABRE), which is applicable to NISQ devices with arbitrary connections between qubits.Experiment results show that SABRE can generate hardware-compliant circuit among different objectives with less or comparable overhead consuming much shorter execution time. Ac-

cording to their paper, SABRE works for IBM chips with arbitrary symmetric CNOT coupling, the hardware model, which differs among vendors and may change over time, is also simplified and single-qubit gates are not yet considered.

3 IBM Quantum Experience

IBM QX is a nice playground and research tool for those interested but mostly inexperienced in quantum computing. It allows the public to access actual quantum computers at IBM, develop and run their programs on them and see the results shortly afterwards.
There are various ways to describe a quantum circuit in QX, including *circuit diagrams* and *OpenQASM* - the quantum assembly language developed by IBM. Below we can see the two options in examples:

3.1 Circuit Diagrams

In QX we can use a simple toolbox which is provided on the top of the webpage, and which includes all different quantum gates (operations) and measurement tools:

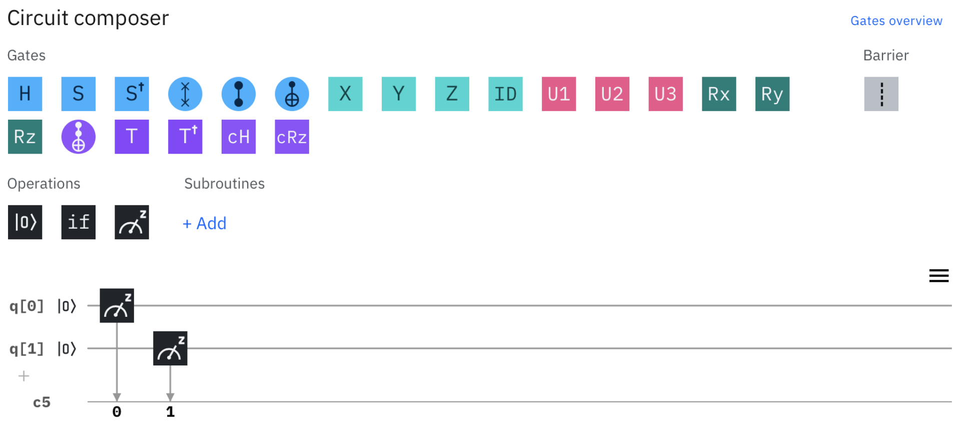
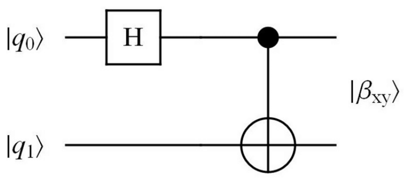
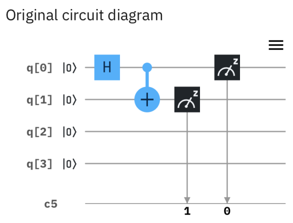


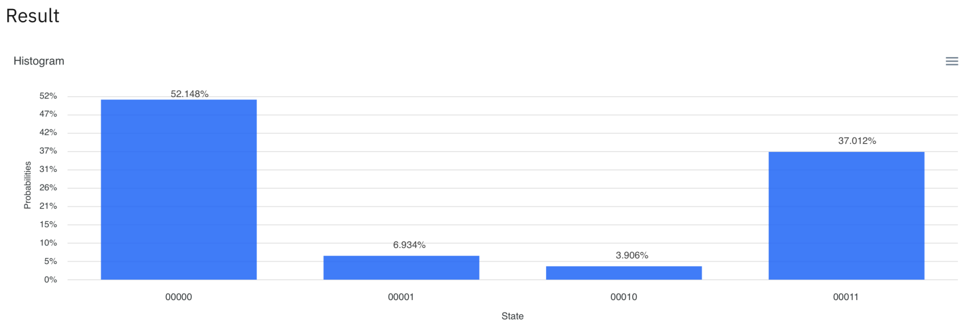
Figure 4: The QX Environment

A good example of a simple quantum circuit with two input qubits, is a Hadamard gate applied on one qubit, followed by a CNOT gate applied on the two qubits. Such a construction is called a “Bell state generator” and is used to build the entangled pairs used in applications such as the famous “teleportation” problem.



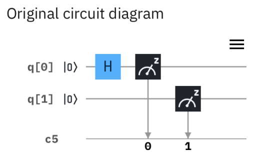
In a Bell state generator, the inputs can be classical bits 00, while the output should have a %50 chance for being either 00 or 11 (hence the 'entanglement' term). However, if we run this on an actual quantum computer (using QX), even with a high iteration of 1024 shots, the percentages we see are different from what we expect:

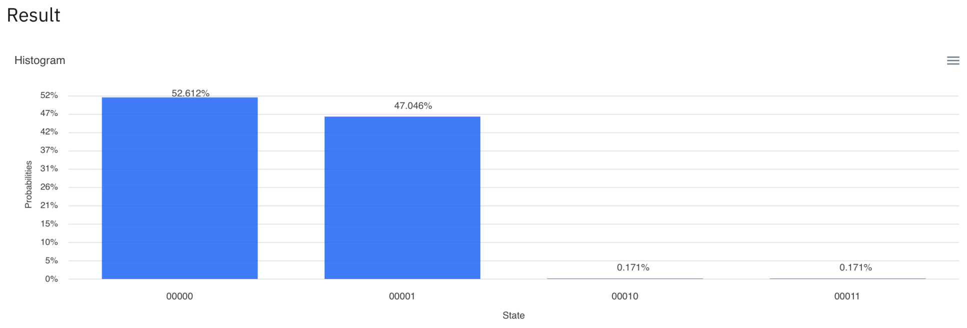




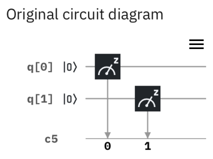
Note that the results indicate a slight chance for the states 01 and 10 to occur, which is not supposed to happen in an ideal entanglement circuit.

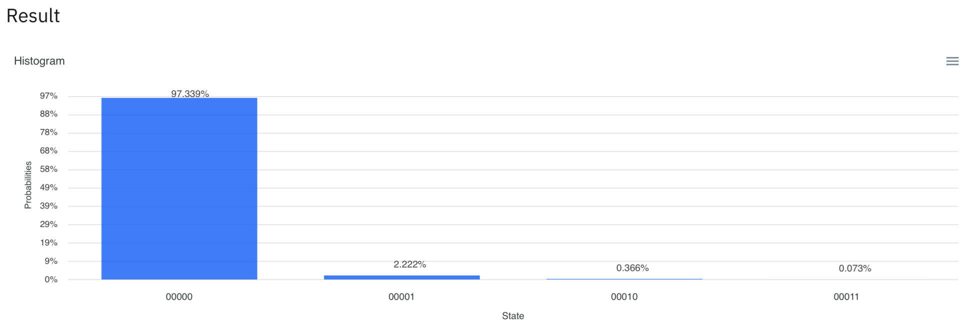
The erroneous result from the previous circuit led me to experiment more on physical computers like IBM Q 16 Melbourne; with only one Hadamard gate and two measurement:





with only measurements:

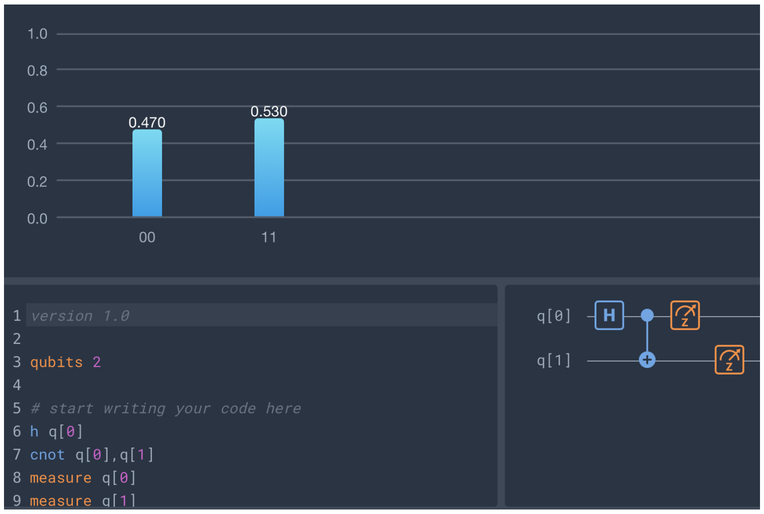




seems like even a sole measurement can get things wrong! There is a chance for the measurement gate to flip the *classical bit* input to it!

4 IBM Quantum Inspire: Simulators for Quantum Computation

Now to see how things are “supposed” to be, we can use another tool called the Quantum Inspire, which consists of a number of classical computers that *simulate* the behaviour of an actual quantum computer. This will help us have a ground truth for any operation we are testing.
We test the circuits we had in the previous section, and compare the results:
The Bell state generator:

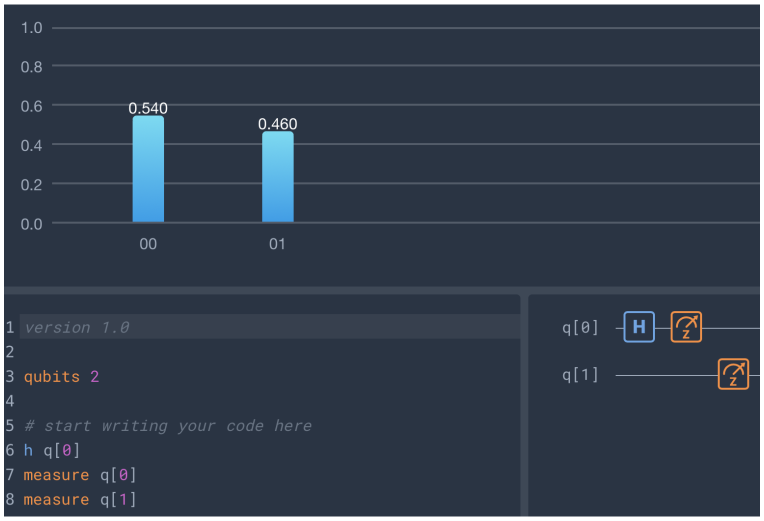


```
version 1.0

qubits 2

# start writing your code here
h q[0]
cnot q[0],q[1]
measure q[0]
measure q[1]
```

Hadamard gate:

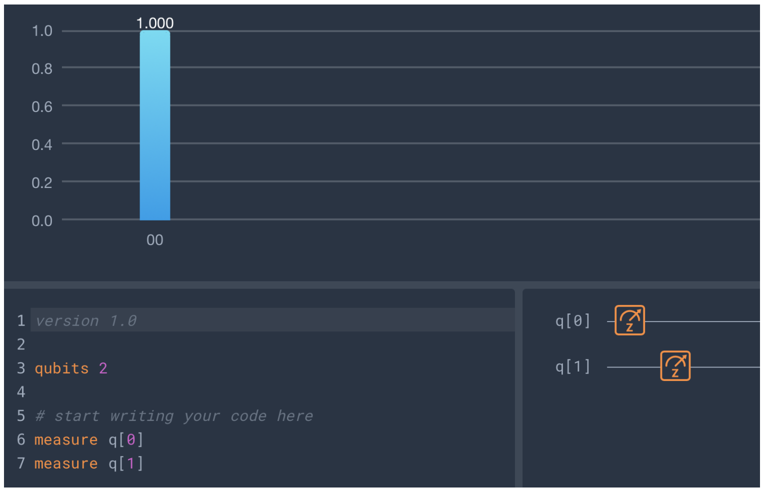


```
version 1.0

qubits 2

# start writing your code here
h q[0]
measure q[0]
measure q[1]
```

Measurement only:



```
version 1.0

qubits 2

# start writing your code here
measure q[0]
measure q[1]
```

Thus the errors we observed in the results from the physical computer, are not present here. In the simulation, there is no probability assigned to states that should not happen! This confirms that the physical computer has a noise/error problem, which shows partly why we need to always minimize the number of quantum gates used in a circuit!

5 Programming for Quantum Design

To describe quantum circuits, high level quantum languages (e.g. Scaffold or Quipper) or quantum assembly languages (e.g. OpenQASM 2.0 developed by IBM) can be used. However, there are more straightforward SDK's for researcher's convenience, such as Qiskit python library.

5.1 The QisKit Toolset

Qiskit is an end-to-end open-source software library for quantum computing, covering the full stack from the actual interaction with the IBM Q hardware, through simulation and emulation, and up to application-level algorithms. The tool itself is arranged in four libraries named after the four classical elements terra, aqua, aer, and ignis. The most useful tool for us here is Terra.

Terra: The Terra library covers all low-level sections of Qiskit. These include tools for specifying and manipulating quantum circuits through the OpenQASM language , or at the pulse levels through Open-Pulse. It provides transpilers to make quantum circuits more optimized for running on real hardware e.g. by minimizing occurrences of CNOT gates. This way, the user can write a circuit which captures the required functionality without investing much effort in optimizing for the specific hardware, and then letting the transpiler find a more optimized circuit while maintaining the exact functionality prescribed by the user. Terra also includes infrastructure for specifying and modeling physical noise processes. These are especially important in order to analyze behavior of a quantum algorithm when run on a noisy quantum computer, as is the case with present-day hardware. Finally, Terra provides the suitable data structures and interfaces to define the various software constructs relevant to quantum computing, and pass those constructs among the different Qiskit libraries, and to the hardware.

6 Working with Qiskit

Written in Python, the Qiskit library is available online. One can download and install Qiskit and Qiskit Visualization Package using the commands: `pip install qiskit` and `pip install qiskit-terra[visualization]` Since Terra is the low-level part of qiskit where we can deal with qubits and quantum gates directly, it is the only part of qiskit that we need to deal with, in order to program a sample circuit:

```
from qiskit import *
from qiskit import QuantumCircuit
from qiskit.tools.visualization import *

#always load the IBM token from this account:
IBMQ.save_account('d8b73c934840b9ece3a5e4bd24e38d5278c72b7f1c24d46fed0f4721f30927d90644b
e3232eb2ac71f4f66090e6d2496dabde3807250bc5be84355a87fc9b089');

# Create a Quantum Circuit
qr = QuantumRegister(2, 'q')
cr = ClassicalRegister(2, 'c')
qc = QuantumCircuit(qr, cr)

# Add a H gate on qubit 0, putting this qubit in superposition.
qc.h(0)
# Add a CX (CNOT) gate, to complete the Bell-state generator
qc.cx(0,1)

# Add a Measure gate to see the state:
qc.measure([0,1], [0,1])
#The two arguments show quantum & classical regs respectively

#print('Circuit Depth = ',qc.depth(), ',', qc_basis.depth())


#show the circuit
qc.draw(output = 'mpl', interactive = True)
```

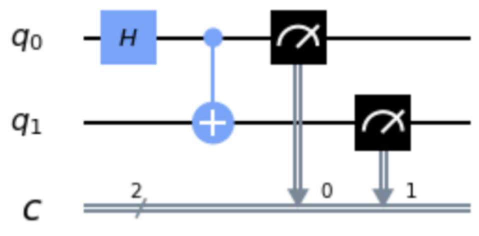


Figure 5: Example circuit to be implemented using Terra

6.1 Backend: Conventional Simulator

Executing the circuit in figure 5 on a simulator:

```
from qiskit import execute, BasicAer

backend_sim = BasicAer.get_backend('qasm_simulator')

#running the job
job_exp = execute(qc, backend_sim, shots=1024, max_credits=10)
result_exp = job_exp.result()

# Show the results
print('Counts: ', result_exp.get_counts(qc));
plot_histogram(result_exp.get_counts(),legend=['1'])
```

```
Counts:  {'11': 496, '00': 528}
```

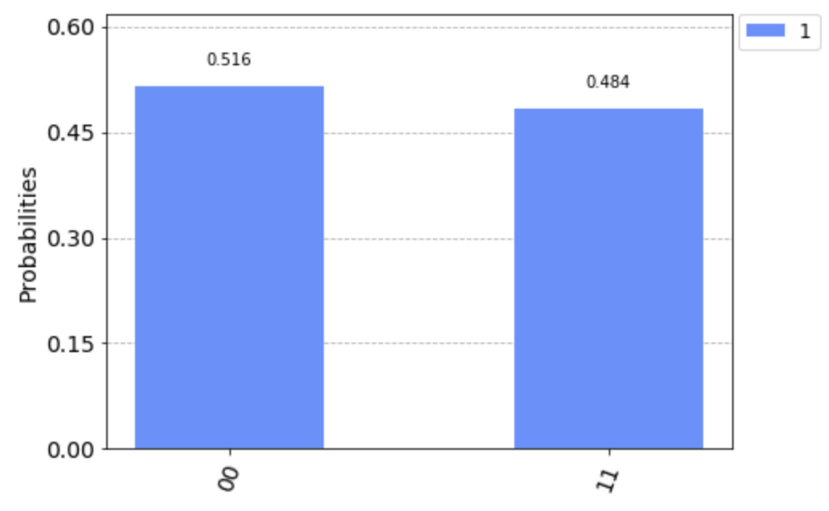


Figure 6: Results from running the circuit of figure 5 on IBM quantum simulator ibmq_qasm_simulator. On the top the counts of each state is shown in numbers and the visualization is provided on the bottom

6.2 Backend: Real Quantum Computer

Executing the circuit in figure 5 on a real quantum computer, namely the 5-qubit IBMQX2:

```
from qiskit import execute, BasicAer

backend= BasicAer.get_backend('ibmqx2')

#running the job
job_exp = execute(qc, backend, shots=1024, max_credits=10)
result_exp = job_exp.result()

# Show the results
print('Counts: ', result_exp.get_counts(qc));
plot_histogram(result_exp.get_counts(),legend=['1'])
```

```
Counts:  {'10': 20, '00': 444, '11': 536, '01': 24}
```

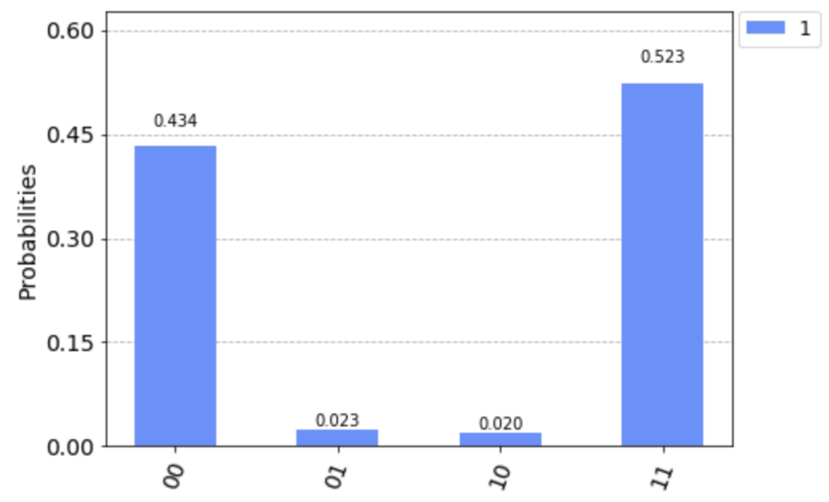


Figure 7: Results from running the circuit of figure 5 on quantum computer IBMQX2. On the top the counts of each state is shown in numbers and the visualization is provided on the bottom

Hence the noisy nature of the real backend is observed in the results in figure 7 as well; as the two irrelevant states `01` and `10` turn out to have a role in the results which also depreciates the results for entangled states.

7 Quantum Circuit Transpilation

A builtin part of IBM Quantum Experience and Qiskit Terra is a transpiler, which takes a quantum circuit defined by user and converts it to one that is suitable for the specified backend, i.e. complies with the CNOT constraints in the backend. We call this *transpiler* because it receives a circuit and generates another circuit that needs to be implemented.
Hence IBM itself is using a set of methods for transpilation and technology mapping.

7.1 IBM Qiskit Transpilation

As utilized in section **6.1**, Qiskit contains a helper function, namely `execute` which does 3 jobs:
1) rewriting and optimizing the given circuit to match the constraints of a given backend.
2) packaging the rewritten circuit for submission.
3) submitting the packaged circuits to the device.

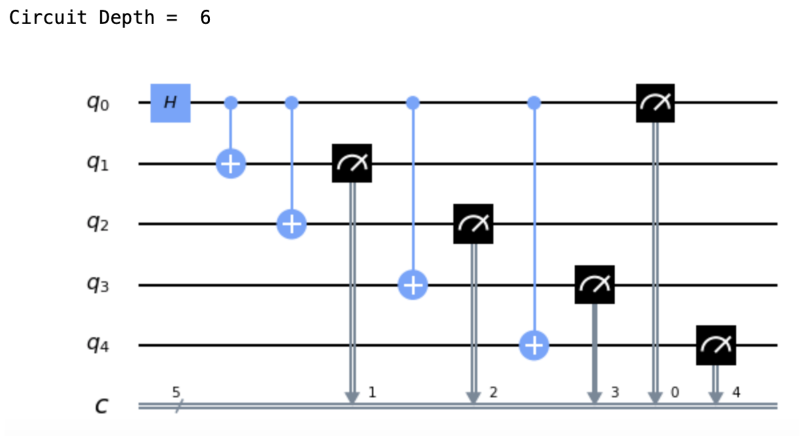
The first step is equivalent to *transpilation*, and is the fundamental step required for running circuits on real quantum devices. The qiskit function that does this is `transpile`, which takes a single or list of input circuits, as well as a collection of parameters, and returns a modified list of circuits. `transpile` is built in the function `execute` as the agent that does the first out of the three steps described above. However, it is also possible for a user to utilize `transpile` by itself, and visualize the transpiled circuit without submitting it for backend implementation. The latter is what we do here.
The metric we use for optimization, is the `depth` of a circuit after running a certain sequence of transpilation passes. The depth of a circuit shows the number of levels of operations that the circuit contains, which will determine the runtime (delay) and the noise margin of the circuit. In qiskit code suite, this is given by an attribute of the class QuantumCircuit, namely `depth`. Hence in order to define a circuit, print the depth of it and then draw it, we write:

```
#Circuit Definition:
qr = QuantumRegister(5, 'q');#5-qubit quantum register
cr = ClassicalRegister(5, 'c');#5-bit classical register for holding measurement results
qc = QuantumCircuit(qr, cr); #get an instance of object QuantumCircuit

#The operations the circuit contains:
# Add a H gate on qubit 0, putting this qubit in superposition.
qc.h(0)
# Add CX (CNOT) gates
qc.cx(0,1)
qc.cx(0,2)
qc.cx(0,3)
qc.cx(0,4)
# Add Measure gates to see the states of qubits:
qc.measure([0,1,2,3,4], [0,1,2,3,4])

#Get the Depth:
print('Circuit Depth = ',qc.depth())

#Show the circuit:
qc.draw(output = 'mpl', interactive = True)
```



```
Circuit Depth =  6
```

Figure 8: An example 5-qubit quantum circuit. On the top the circuit depth is printed. All measurements together are considered as one level, since they can be done at the same time.

Any circuit in Qiskit can also be described using a graph-based method, where a *directed acyclic graph* shows the circuit, the edges denoting qubits and nodes denoting operations. The example circuit of figure 8 can therefore be visualized as shown in figure 9, using the code below:

```
from qiskit.converters import circuit_to_dag

dag = circuit_to_dag(qc)
dag_drawer(dag)
```

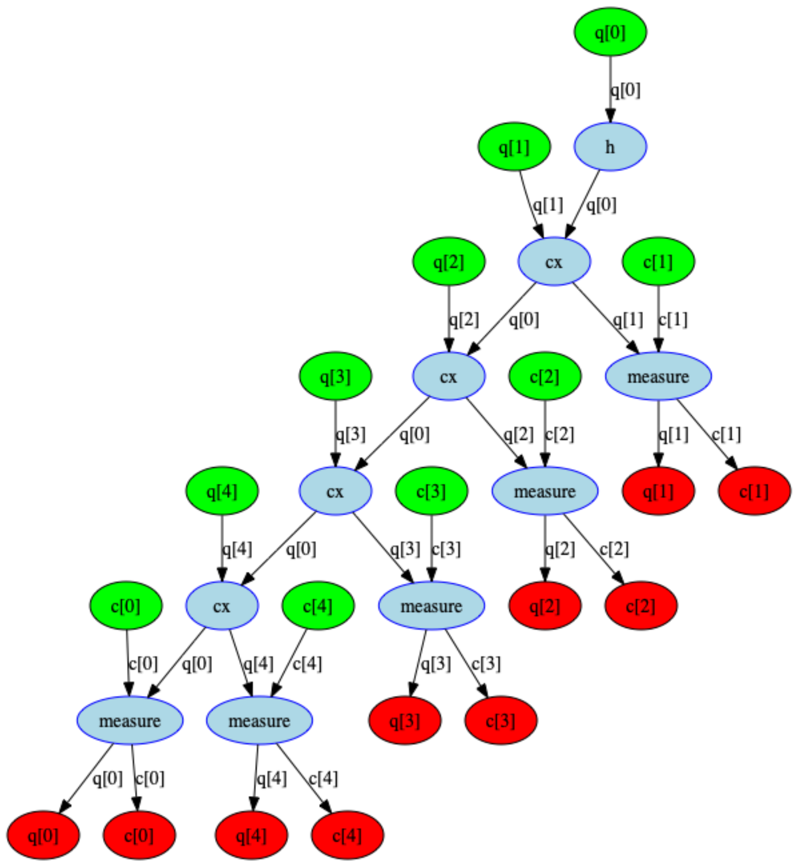


Figure 9: The DAG denoting example circuit of figure 8. The primary input nodes and also the edges all over the graph, denote the (physical) qubits, and other nodes in blue indicate the operations on those inputs. The number of operation levels in this dag indicates the depth of the circuit.

Now let us consider the circuit in figure 8 for implementation using IBMQX2. Firstly, the circuit needs to be decomposed into only unitary single-qubit gates, along with CNOT as the only two-qubit gates. Our example circuit is already in the simplest form, as it only contains a Hadamard and several CNOT gates. However for the general case we apply:

```
qc_basis = qc.decompose()
```

Then, we need to check the coupling map of our desired backend device. In addition to IBM QX website as a good reference, one can see the coupling map of any of the IBM backends using the following code snippet:

```
from qiskit import execute, IBMQ

my_provider = IBMQ.get_provider()
backend  = my_provider.get_backend('ibmqx2') #gets the info of the given backend
plot_gate_map(backend, plot_directed=True) #shows the coupling map
```

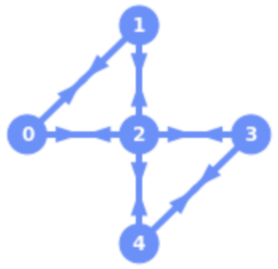


Figure 10: Coupling Map of IBMQX2

After that, we may start out our experiments by transpiling the circuit of figure 8 using some Qiskit preset transpilation passes. In Qiskit, we can define or use entities called **PassManager** which is a sequence of different transpilation steps. Qiskit community has four preset pass managers, which differ in their levels of optimization, zero for no optimization, and three for the highest degree of optimization. The first preset pass manager we can try has optimization level zero, which means the only key thing done for the actual transpilation is throwing in a number of `StochasticSwap` operations to map the given circuit into one that is compliant with the given coupling map.

```
transpiled_circ_lv0 = transpile(qc
,backend=backend
,optimization_level=0)
#use the preset level-0 pass manager

print('Depth of transpiled circuit with preset pass manager with optimization level 0: '
,transpiled_circ_lv0.depth())

transpiled_circ_lv0.draw(output = 'mpl', interactive = True)
#show the transpiled circuit
```

Figures 11 and 12 show the circuit of figure 8 after transpilation with optimization level 0, before and after applying the final decomposition of added swap gates, respectively:

In the preset pass manager with optimization level 0, the initial layout is either given by the user, or if not given it is simply set as a default dictionary of what qiskit calls `TrivialLayout`. A trivial layout is one where each virtual qubit with index i is simply mapped onto the physical qubit of the same index, namely: {qr[0]:0, qr[1]:1, ...}.

Let us analyze what is happening in the transpilation process, using the example of figure 11. In the analysis below, by the term "qubit" we always denote a *virtual qubit* which is one that holds the logical information; a physical qubit will be explicitly specified:

As a first step in the transpilation procedure, the circuit is partitioned into different layers (or equivalently, levels). A level is a set of gates that can be done at the same time, simploy because they are affecting qubit pairs that are independent from each other.
First the trivial layout (initial mapping of the qubits) is applied to the circuit. The first operation that the circuit then has to perform is a single-qubit operation which is already unrolled to basis set of gates as part of the `transpile` procedure. Because single-qubit gates do not involve dependencies between qubits, this

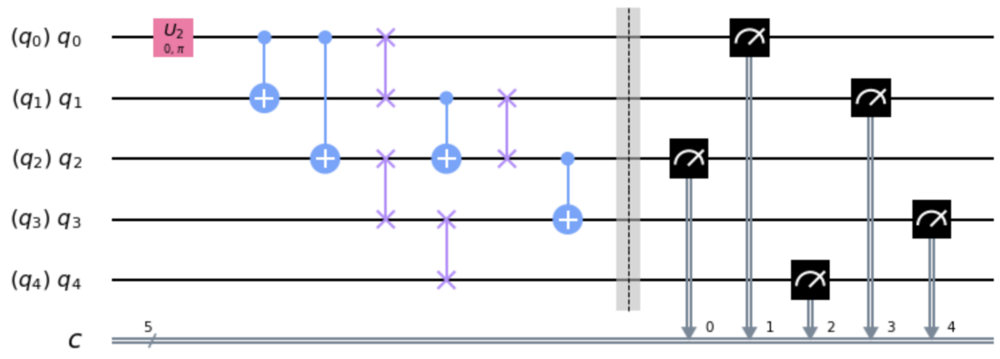


Figure 11: Transpiled circuit with preset pass manager with optimization level 0, before decomposing the additional swap gates into basis (CNOT) gates.In the visualized circuit, the q_i's on the left (in parantheses) show the virtual indices of the qubits (logical qubits) and the indexing on the right is used on the real device (physical qubits).

first gate will not need transformation to be made in the qubit allocation. Hence the qubits hold their prior values in the second level, which is after the U_2 gate.
In the second and third levels, there are CNOT gates with control qubit 0 and target qubits 1 and 2 respectively. These two dependencies exist in the coupling map of our desired backend (figure 10), and therefore no changes need to be made in the allocation.
When moving to level 4 of the circuit, we encounter a CNOT gate controlled by qubit 0 while the target is qubit 3. This is not possible according to the coupling map of figure 10, which necessitates that we move the two qubits closer together in the physical device. Hence the pass manager uses two swap gates at the same time, which swaps the information held by physical qubits (0,1) and (2,3). This technically means the allocation we had before (the trivial layout) will now change at this point of time, to the mapping: $\{q[0] : 1, q[1] : 0, q[2] : 3, q[3] : 2, q[4] : 4\}$. This way, as we want to perform the desired CNOT operation between q[0] and q[3], we will need only to perform it between physical qubits 1 and 2, which is an allowed dependency according to the coupling map. The same step is performed on all following layers until we reach the end (right-most hand) of the circuit, having had met all the CNOT constraints of our coupling map, while also having performed the operations given in the circuit.

```
Depth of transpiled circuit with preset pass manager with optimization level 0:  12
```

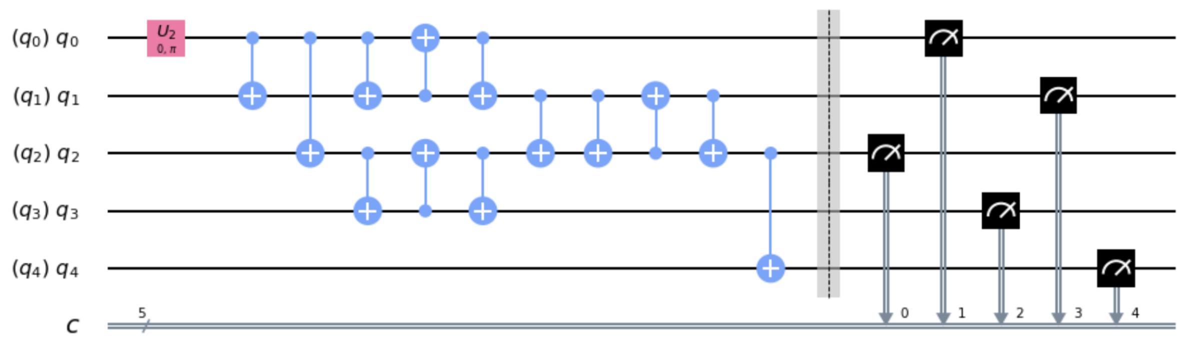


Figure 12: Transpiled circuit with preset pass manager with optimization level 0, after decomposing the additional swap gates into basis (CNOT) gates. The depth of the circuit is printed on the top. In the visualized circuit, the q_i's on the left (in parantheses) show the virtual indices of the qubits (logical qubits) and the indexing on the right is used on the real device (physical qubits).

We can visualize this initial mapping using the code snippet below. The resulting mapping in python is shown in figure 13. In a coupling map, the numbers shown in black are the virtual qubits, and the blue circles are the physical qubits that the virtual ones are mapped onto, all in the exact locations as the physical qubits of the coupling map in figure 10. In the case of a 5-qubit circuit, all the physical qubits are used up and hence no blue circles are left.

```
plot_circuit_layout(transpiled_circ_lv0,backend,view = 'virtual')
```

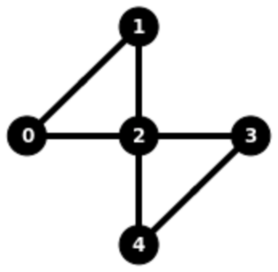


Figure 13: Default initial (Trivial) Mapping of Qubits with Optimization Level = 0 for IBMQX2. User can override this by giving the transpiler a customized dictionary showing the desired mapping.

To see what transpilation passes run in this preset pass manager, let us visualize the pass manager flow:

```
from qiskit.compiler.transpile import _parse_transpile_args

transpile_config = _parse_transpile_args(circuits=[qc]
,backend=backend
,basis_gates=None
,coupling_map=None
,backend_properties=None
,initial_layout=None
,seed_transpiler=None
,optimization_level=0
,pass_manager=None
,callback=None
,output_name=None)[0] #extract the overall configs for the circuit

pass_manager_drawer(level_0_pass_manager(transpile_config),filename='level0')
#visualize the pass manager flow
```

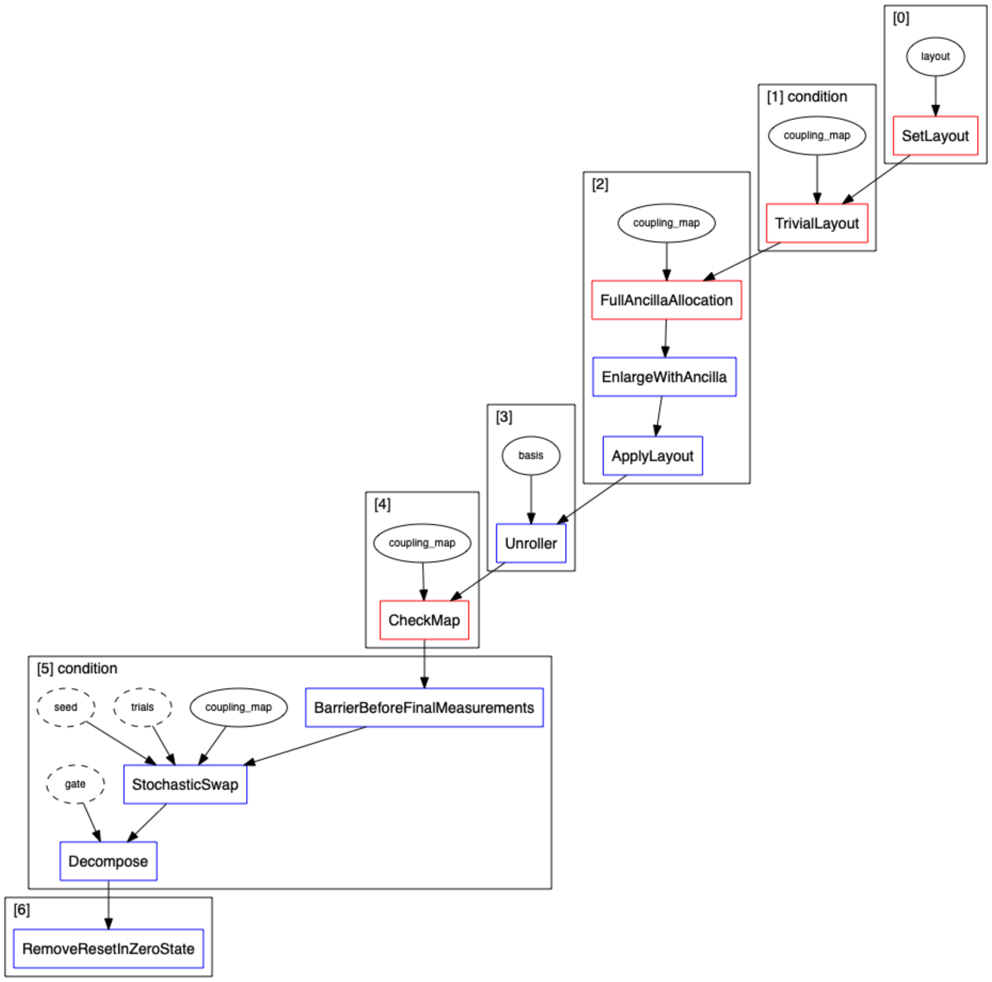


Figure 14: The flow of preset pass manager with optimization level = 0

As observed in figure 14, this process includes several steps, some of which are not really necessary for a circuit as small as our example circuit. By going through qiskit source code and checking the functionality of each step ([4]), we conclude that the most critical ones are as follows:

- `SetLayout`: Takes a custom initial layout from the user and applies it to the circuit at the beginning of the pass. If this is not given, then the TrivialLayout is used as a default.

- `Unroller`: In this process a set of standard gates are defined by the user (e.g. $['u1','u2','u3','cx']$) and the Unroller decomposes the given circuit into one with only these predefined gates. In our example circuit of figure 8, a Hadamard gate must be rewritten as a unitary gate of either 'u1','u2' or 'u3'. These three operations are defined in figure 15. The simplest one for a Hadamard will be a 'u2' with $\phi = 0$ and $\lambda = \pi$. The other gates in the circuit include only CNOTs ($'cx'$) which is already in the defined set and hence unrolling will not affect it.

$$U(\theta,\phi,\lambda) = \begin{pmatrix} \cos\left(\frac{\theta}{2}\right) & -e^{i\lambda}\sin\left(\frac{\theta}{2}\right) \\ e^{i\phi}\sin\left(\frac{\theta}{2}\right) & e^{i(\lambda+\phi)}\cos\left(\frac{\theta}{2}\right) \end{pmatrix}$$

$$U_1(\lambda) = U(0,0,\lambda) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\lambda} \end{pmatrix}$$
$$U_2(\phi,\lambda) = U\left(\frac{\pi}{2},\psi,\lambda\right) = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & -e^{i\lambda} \\ e^{i\phi} & e^{i(\lambda+\phi)} \end{pmatrix}$$
$$U_3(\theta,\phi,\lambda) = U(\theta,\phi,\lambda) = \text{see above}$$

Figure 15: Definitions of the three single-qubit unitary gates used as a basis gate set

- `StochasticSwap`: This is the key operation in this process. We add swap gates at each layer (level) of the circuit and check if the overall circuit can then be mapped to the coupling map of the backend. In StochasticSwap, swap insertions between all permutations of pairs of qubits at each layer are tried out, and the one that leads to minimum depth is selected. In other cases this can be replaced by other kinds of swap insertion procedures, such as `BasicSwap` which is the most basic type of swap insertion.

- `Decompose(SwapGate)`: This last step is essential as it will change the additional swap gate(s) resulting from the transpilation process (in the case of BasicSwap, the resulting swap gate is not in our standard set of gates ({'u1','u2','u3','cx'}), hence we decompose it into an equivalent subcircuit using only three CNOTs. This swap decomposition technique has been shown in figure 3).

After decomposition, it is observed that all the additional swap gates are decomposed into 3 CNOT gates, hence adding 6 gates to the overall depth of the original circuit. Therefore it becomes clear that we need optimization techniques to perform the mapping more efficiently and minimize the gate overhead of the circuit.

To sum up, we can use the aforementioned simple steps and build our own pass manager:

```
import qiskit
from qiskit import *
from qiskit.transpiler import PassManager
from qiskit.transpiler.passes import BasicSwap
from qiskit.extensions.standard import SwapGate
from qiskit.transpiler.passes import Unroller
from qiskit.transpiler.passes import Decompose
from qiskit.transpiler.passes import SetLayout
```

```
def my_pass_manager(coupling_map, initial_layout):

    _given_layout = SetLayout(initial_layout);
    pm = PassManager([Unroller(['u1','u2','u3','cx'])
    ,_given_layout
    ,BasicSwap(coupling_map)
    ,Decompose(SwapGate)]);
    #sequence of operations are given as a list

    return pm
```

This code will set the initial layout for our mapping, according to the argument `initial_layout` that the main program feeds the function. The coupling map will also be input by the program. Then the object `PassManager` is built using the three aforementioned pass steps: initial layout set, basic swap and finally decomposition of the non-standard gates, listed sequentially.
We save the above code in a file named `my_pass_manager.py` , which we later import in our main code for use:

```
from qiskit.transpiler.my_pass_manager import my_pass_manager
#adding this to the main code
```

We can then start experimenting with different initial layouts:

```
from qiskit.transpiler.coupling import CouplingMap

coupling_map = CouplingMap(coupling_list)

#specify the initial layout using a {virtual:physical} qubit dictionary:
initial_layout = Layout({qr[0]: 1, qr[1]: 4, qr[2]: 3, qr[3]: 2, qr[4]: 0})

#transpile the circuit using the desired parameters:
qc_transpiled = transpile(qc
,backend=backend
,pass_manager = my_pass_manager(coupling_map, initial_layout))

#print the resulting depth:
print('transpiled circuit with custom pass manager:'
,qc_transpiled.depth())

#show the transpiled circuit:
qc_transpiled.draw(output = 'mpl', interactive = True)
```

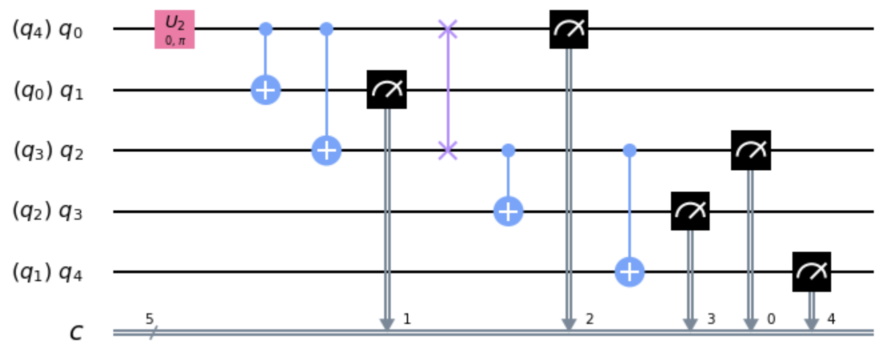


Figure 16: Result of transpilation of the circuit in figure 8 using the customized pass manager, excluding the decomposition of swap gates. In the visualized circuit, the q_i's on the left (in parantheses) show the virtual indexing of the qubits (logical qubits) and the indexing on the right is used on the real device (physical qubits).

```
transpiled circuit with custom pass manager:  9
```

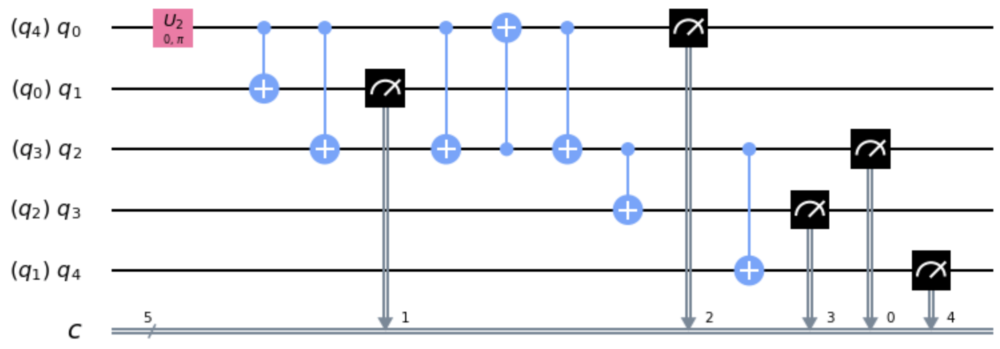


Figure 17: Result of transpilation of the circuit in figure 8 using the customized pass manager, including the swap decomposition. In the visualized circuit, the q_i's on the left (in parantheses) show the virtual indices of the qubits (logical qubits) and the indexing on the right is used on the real device (physical qubits).

Comparing the above resulting circuit in figure 17 the result from the level-zero pass manager (figure 12), in which the initial layout is a trivial layout, shows the importance of an appropriate initial layout in having a minimal transpiled circuit, as the number of gates can drastically decrease when using a smart initial qubit allocation.

We can visualize the given initial mapping using the code snippet below. The resulting mapping in python is shown in figure 18. Again, the physical qubits of the coupling map in figure 10 are all covered by the black virtual qubits, hence no blue circles are left in the coupling map.

```
plot_circuit_layout(qc_transpiled,backend,view = 'virtual')
```

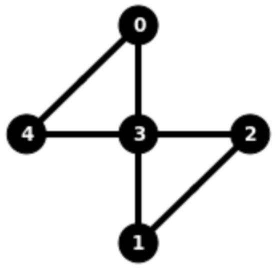


Figure 18: User-defined initial Mapping of Qubits for IBMQX2. This mapping is then fed to the customized pass manager for further transpilation steps.

We can also visualize the transpilation passes used in this customized pass manager, using the following script:

```
pass_manager_drawer(pass_manager = my_pass_manager(coupling_map, initial_layout)
,filename='my_passes')
```

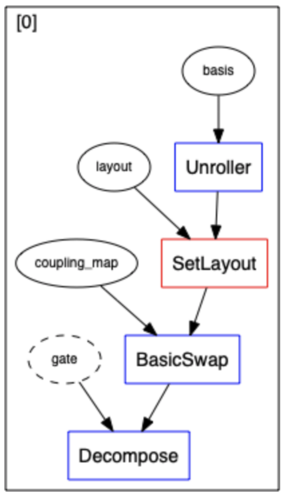


Figure 19: The flow of customized pass manager

The transpiled circuit of figure 17 can also be visualized using a directed acyclic graph as shown in figure 20, using the code below:

```
from qiskit.converters import circuit_to_dag

dag = circuit_to_dag(qc_transpiled)
dag_drawer(dag)
```

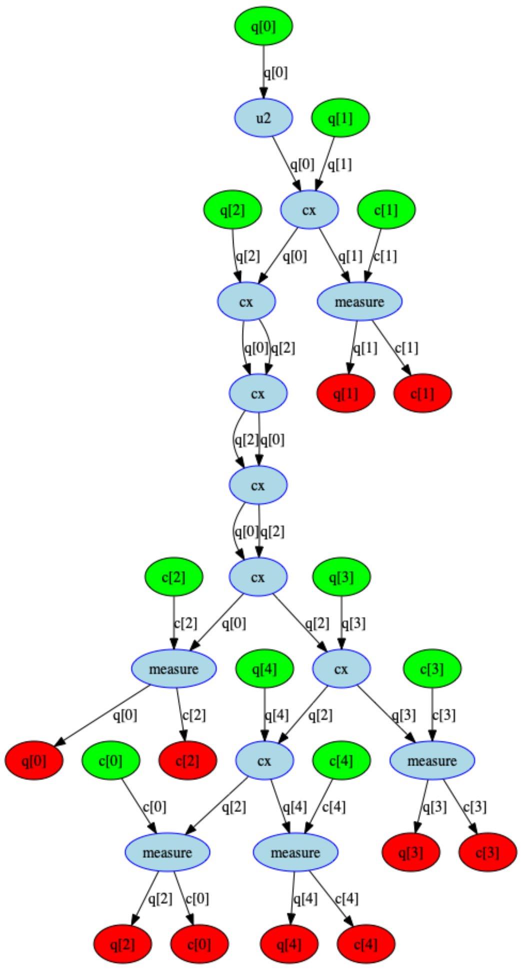


Figure 20: The DAG denoting transpiled circuit of figure 17. The primary input nodes and also the edges all over the graph, denote the (physical) qubits, and other nodes in blue indicate the operations on those inputs. The number of operation levels in this dag indicates the depth of the transpiled circuit (9), which is larger than the original depth before transpilation (6).

7.2 Techniques for Quantum Transpilation

Any given circuit can first be transformed into a standard form in which we have only single-qubit gates as well as CNOTs, as the only 2-qubit gate. We use this rule to help us only deal with CNOT gates as the sole cause of any qubit-qubit dependency throughout the circuit. Given that, transpilation means for any CNOT gate in a circuit, if it is not happening between qubits that are "allowed" to interact (or is happening in the wrong direction), we need to transform the qubit allocations of the circuit into one that ensures the satisfaction of the backend coupling map.
To that end, *Siraichi et. al.* came up with a graph representation and mapping solution from logical qubits (called "Pseduo-qubits") to physical ones [5]. Unlike many other works which only make use of SWAP gates to do transpilation, they use a set of three different gates throughout the transpiled circuit:

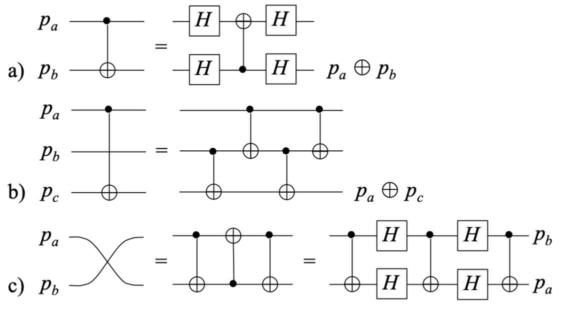


Figure 21: (a) Reversal. (b) Bridge. (c) Swap

The set of equivalences shown in figure 21, are used in the circuit anytime that we need a change in our qubit allocation, in order to make everything comply with the qubit coupling map (CNOT constraints). An example of a quantum circuit ready for transpilation is shown below:

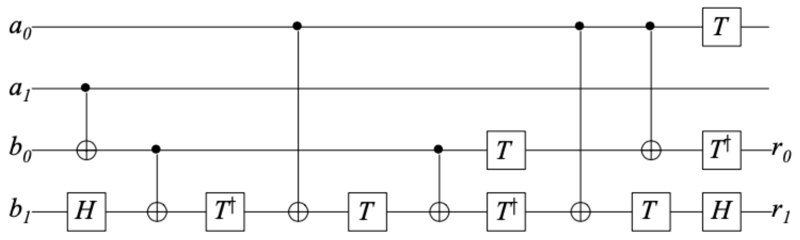


Figure 22: Example circuit to be transpiled

This circuit can be transpiled into the following, using two reversal gates that substitutes the non-compliant CNOTs with the coupling map shown with a directed graph in the same figure:

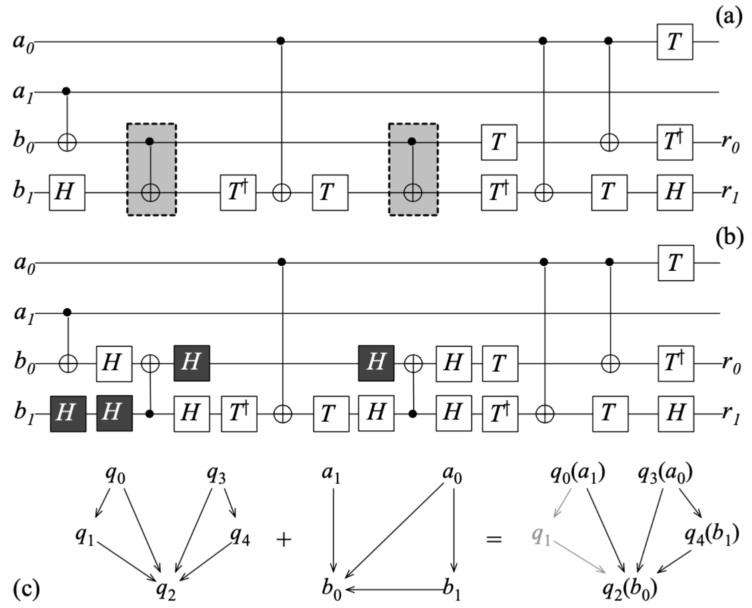


Figure 23: (a,b) Transpiling the circuit in figure 22, using two reversal gates, in a way that it follows the coupling map ((c) on the left). The gray boxes show the CNOTs in the original circuit that do not comply with the constraints and need a transformation in mapping. The successive Hadamard gates (in black) cancel out with each other. The second graph in (c) shows the pseudo-qubits and the way they need to be CNOT-connected according to the circuit design, and the last graph shows the initial mapping.

On the other hand, we can use one SWAP gate instead of two reversals, for the same purpose:

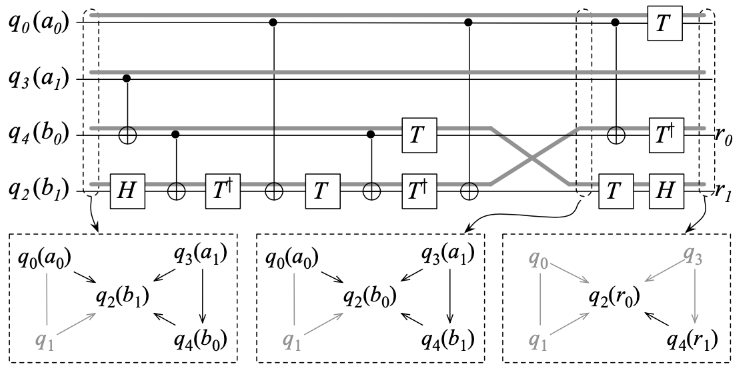


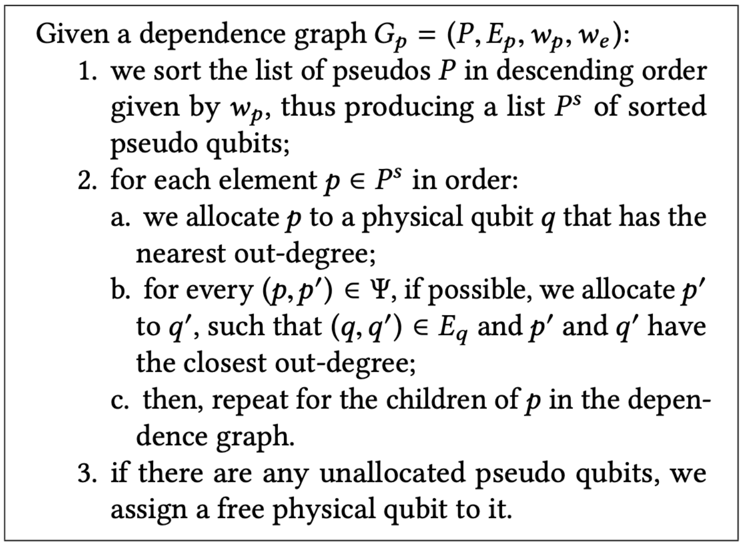
Figure 24: Transpiling the circuit in figure 22, in a way that it follows the coupling map shown with nodes $\{q_0 \dots q_4\}$. The set of nodes a_0, b_0, a_1 and b_1 are

Here we are using a SWAP between b_0 and b_1 so we will be able to make a connection between a_0 and b_0. Just like the analysis in section **7.1**, here the circuit is traversed in a chronological order (from left to right) and at each level, the required transforms are made.
Comparing these two transpilation methods shows that the latter will lead to an overhead of 2 in the number of gates, while the former leads to 5. This means the swapping technique is more optimal in terms of the

number of excessive operations. However, the depth of the circuit may be affected a bit differently, depending on the sequence of operations.

Step-by-Step Transpilation

The first step in transpilation is to have an initial mapping. For that, the paper takes a graph-theory approach in which the circuit is visualized using a dependency graph. The nodes in a dependency graph denote (virtual) qubits and each edge shows a dependency, which is essentially a CNOT gate that we encounter in the circuit.



```
Given a dependence graph $G_p = (P, E_p, w_p, w_e)$:
  1. we sort the list of pseudos $P$ in descending order
     given by $w_p$, thus producing a list $P^s$ of sorted
     pseudo qubits;
  2. for each element $p \in P^s$ in order:
     a. we allocate $p$ to a physical qubit $q$ that has the
        nearest out-degree;
     b. for every $(p, p') \in \Psi$, if possible, we allocate $p'$
        to $q'$, such that $(q, q') \in E_q$ and $p'$ and $q'$ have
        the closest out-degree;
     c. then, repeat for the children of $p$ in the depen-
        dence graph.
  3. if there are any unallocated pseudo qubits, we
     assign a free physical qubit to it.
```

Figure 25: Algorithm for finding an optimal initial mapping for a circuit with dependency set ψ.

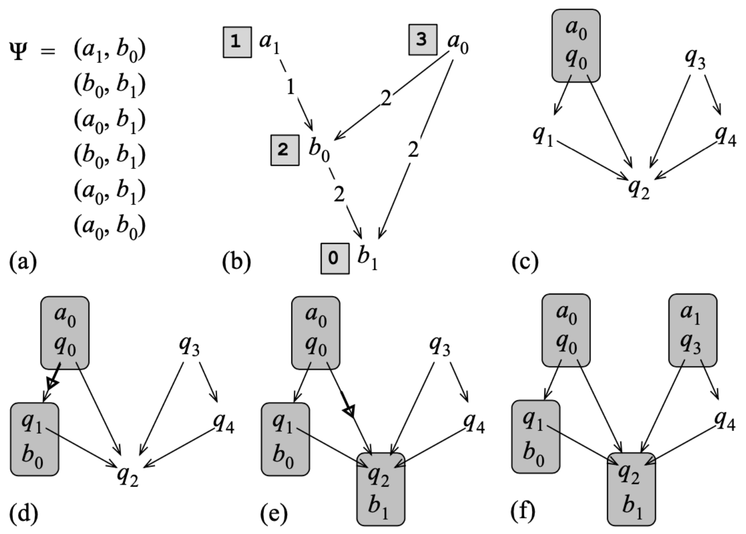
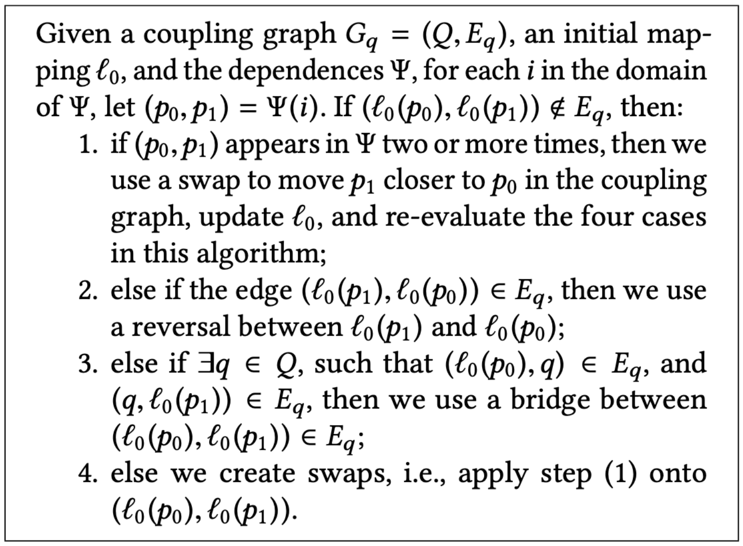


Figure 26: Finding the optimal initial mapping for a circuit with dependency set ψ. The algorithm for this is given in figure 25

After finding a good initial mapping, we should traverse the circuit in different time-steps and at each step, check if there are any dependencies in the circuit that do not comply with coupling graph. If that is the case, we apply one of the three gates defined in figure 21. The choice of the additional gate(s) depends on several things which are elaborated upon in the algorithm by *Siraichi et. al.*:


Given a coupling graph $G_q = (Q, E_q)$, an initial mapping ℓ_0, and the dependences Ψ, for each i in the domain of Ψ, let $(p_0, p_1) = \Psi(i)$. If $(\ell_0(p_0), \ell_0(p_1)) \notin E_q$, then:

1. if (p_0, p_1) appears in Ψ two or more times, then we use a swap to move p_1 closer to p_0 in the coupling graph, update ℓ_0, and re-evaluate the four cases in this algorithm;
2. else if the edge $(\ell_0(p_1), \ell_0(p_0)) \in E_q$, then we use a reversal between $\ell_0(p_1)$ and $\ell_0(p_0)$;
3. else if $\exists q \in Q$, such that $(\ell_0(p_0), q) \in E_q$, and $(q, \ell_0(p_1)) \in E_q$, then we use a bridge between $(\ell_0(p_0), \ell_0(p_1)) \in E_q$;
4. else we create swaps, i.e., apply step (1) onto $(\ell_0(p_0), \ell_0(p_1))$.

Figure 27: Algorithm to extend and update the qubit mapping for a circuit with dependency set ψ.

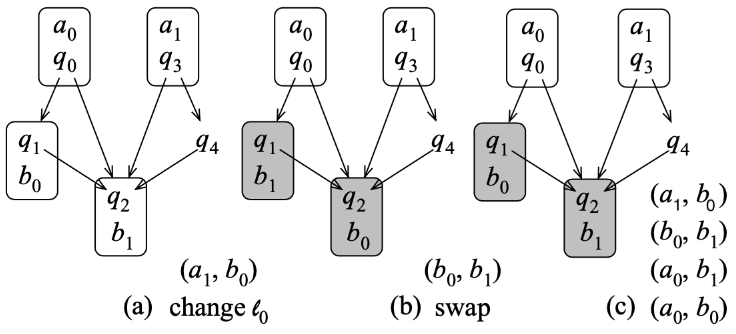


Figure 28: Extending the initial mapping to satisfy ψ. The algorithm for this is given in figure 27.(a) We change the original mapping to satisfy dependence $\psi(1) = (a_1, b_0)$. No transformation is created during this action. (b) We swap b1 and b0,to satisfy dependence $\psi(2) = (b_0, b_1)$.We use a swap because there are two occurrences of (b_0, b_1) in ψ. (c) The other dependencies are now satisfied.

The results in the paper show near-optimal performance in comparison with many well-known algorithms, such as IBM mapper itself.

8 Future Work

We have shown some important parts of Qiskit coding environment, which is the most widely used quantum programming tool. Qiskit has an enormous community who are contributing to its source codes as well as conceptual issues all the time.
An idea we can continue on is implementing Siraichi's work in a transpiler code from Qiskit. For that, we need to change the existing DAG-analysis mechanism in Qiskit, from one where nodes show operations and edges are qubits, to one where nodes are qubits and edges are CNOT operations. Then, using the algorithms in figures 25 and 27 we can analyze the graphs and find the allocation with minimal overhead, using the fact that Sirachi's algorithm has sort of a lookahead approch where it considers how many times a dependency is repeated, as well as which dependencies are already meeting the coupling map constraints that we can make use of.

References

[1] IBM, "Ibm qiskit development platform." https://qiskit.org.

[2] A. Zulehner, A. Paler, and R. Wille, "An efficient methodology for mapping quantum circuits to the ibm qx architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, pp. 1226–1236, July 2019.

[3] G. Li, Y. Ding, and Y. Xie, "Tackling the qubit mapping problem for nisq-era quantum devices," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, (New York, NY, USA), pp. 1001–1014, ACM, 2019.

[4] IBM, "Ibm qiskit-terra github." https://github.com/Qiskit/qiskit-terra/tree/master/qiskit.

[5] M. Y. Siraichi, V. F. dos Santos, S. Collange, and F. M. Q. Pereira, "Qubit allocation," in *CGO 2018*, 2018.