QUANTIS Notebook 2 Circuit in IBM Quantum Computer

January 8, 2024

0.1 Quantum Circuits on both Simulators and IBM Quantum Computer

In this notebook, we are going to learn how to use Qiskit to define a simple circuit and to execute it on both simulators and the quantum computers of the IBM Quantum Experience..

We start by importing the necessary packages.

[2]:
```
%matplotlib inline

from qiskit import *
from qiskit.visualization import *
from qiskit.tools.monitor import *
from qiskit.quantum_info import Statevector
```

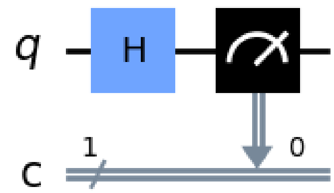
0.2 Defining the circuit

We are going to define a very simple circuit: we will use the H gate to put a qubit in superposition and then we will measure it

[3]:
```
# Let's create a circuit to put a state in superposition and measure it

circ = QuantumCircuit(1,1) # We use one qubit and also one classical bit for␣
↪the measure result

circ.h(0) #We apply the H gate

circ.measure(range(1),range(1)) # We measure

circ.draw(output='mpl') #We draw the circuit
```

```
C:\Users\miche\anaconda3\envs\malis\lib\site-
packages\qiskit\visualization\circuit\matplotlib.py:266: FutureWarning: The
default matplotlib drawer scheme will be changed to "iqp" in a following
release. To silence this warning, specify the current default explicitly as
style="clifford", or the new default as style="iqp".
  self._style, def_font_ratio = load_style(self._style)
```

[3]:



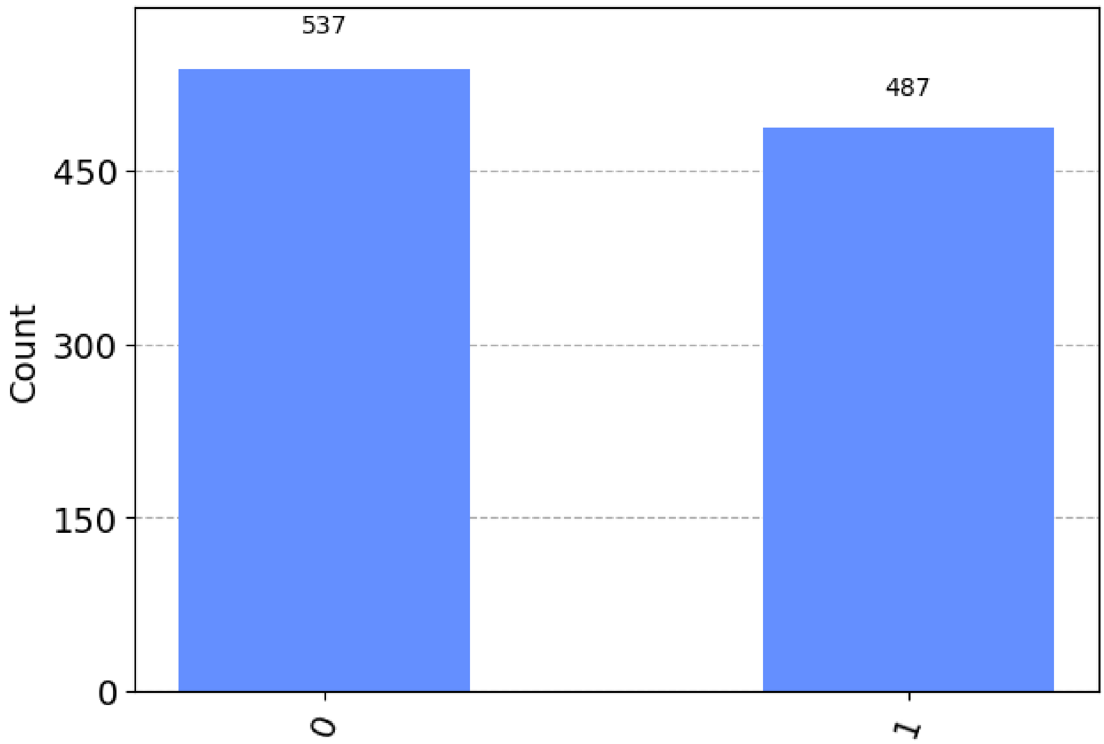
0.3 Running the circuit on simulators

Once that we have defined the circuit, we can execute it on a simulator.

[4]:
```
# Executing on the local simulator

backend_sim = Aer.get_backend('qasm_simulator') # We choose the backend

job_sim = execute(circ, backend_sim, shots=1024) # We execute the circuit,␣
↪selecting the number of repetitions or 'shots'

result_sim = job_sim.result() # We collect the results

counts = result_sim.get_counts(circ) # We obtain the frequency of each result␣
↪and we show them
print(counts)
plot_histogram(counts)
```

```
{'1': 487, '0': 537}
```

[4]:



We can also run the circuit run the circuit with a simulator that computes the final state. For that, we need to create a circuit with no measures

[5]:
```
# Execution to the get the statevector

circ2 = QuantumCircuit(1,1)

circ2.h(0)

backend = Aer.get_backend('statevector_simulator') # We change the backend

job = execute(circ2, backend) # We execute the circuit with the new simulator.␣
 ↪Now, we do not need repetitions

result = job.result() # We collect the results and access the stavector
outputstate = result.get_statevector(circ2)
print(outputstate)
```

```
Statevector([0.70710678+0.j, 0.70710678+0.j],
            dims=(2,))
```

Finally, we can also obtain the unitary matrix that represents the action of the circuit

[6]:
```
import numpy as np
backend = Aer.get_backend('unitary_simulator') # We change the backend again

job = execute(circ2, backend) # We execute the circuit

result = job.result() # We collect the results and obtain the matrix
unitary = result.get_unitary()
print(np.round(unitary,10))
```

```
[[ 0.70710678+0.j  0.70710678-0.j]
 [ 0.70710678+0.j -0.70710678+0.j]]
```

0.4 Running the circuit on Quantum Computer

Now, we are going to use the quantum computers at the IBM Quantum Experience to use our circuit

One you have created an IBMid account here: https://quantum-computing.ibm.com/

…in the below code, you will need to replace MY API TOKEN with the API number you have save into your clipboard. Alternatively, you can load the account (if you have saved the Token in a file).

For more details, you can read here: https://github.com/Qiskit/qiskit-ibmq-provider

[11]:
```
# Connecting to the real quantum computers
provider = IBMQ.
 ↪enable_account('c481f37255158ac85e720f723445f8bc981ed106d903b47e27e8754e0399c23f0e7dfe43d27
provider.backends() # We retrieve the backends to check their status

for b in provider.backends():
    print(b.status().to_dict())
```

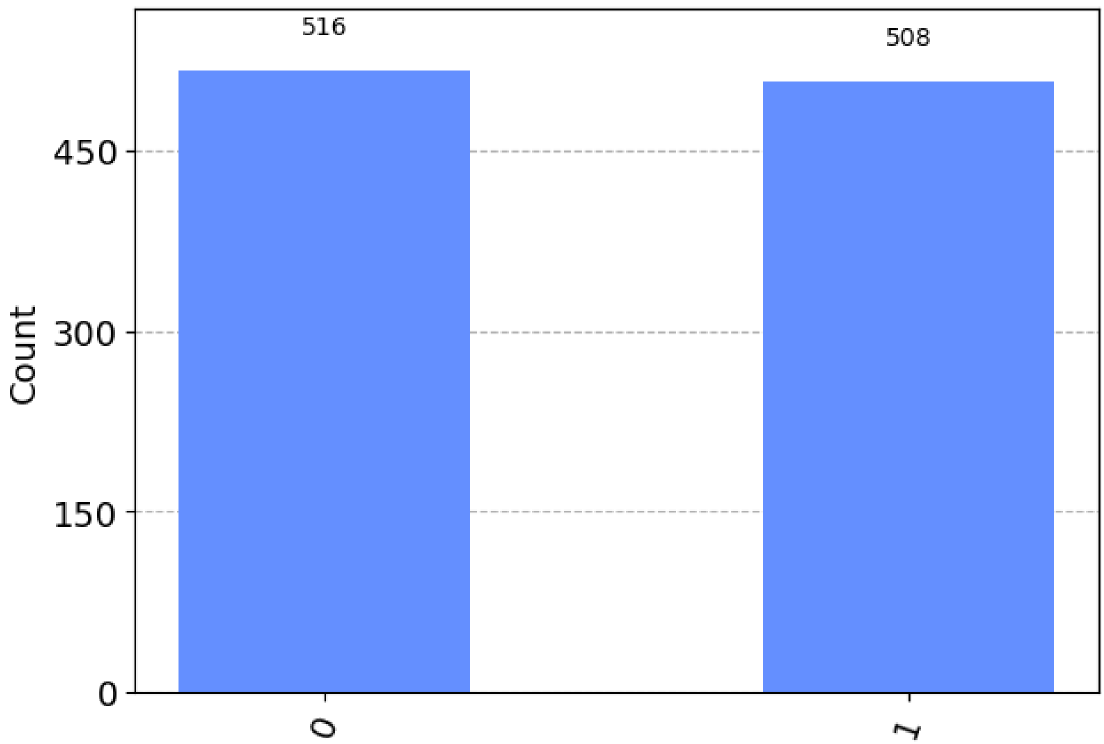
```
{'backend_name': 'ibmq_qasm_simulator', 'backend_version': '0.1.547',
'operational': True, 'pending_jobs': 0, 'status_msg': 'active'}
{'backend_name': 'simulator_statevector', 'backend_version': '0.1.547',
'operational': True, 'pending_jobs': 0, 'status_msg': 'active'}
{'backend_name': 'simulator_mps', 'backend_version': '0.1.547', 'operational':
True, 'pending_jobs': 0, 'status_msg': 'active'}
{'backend_name': 'simulator_extended_stabilizer', 'backend_version': '0.1.547',
'operational': True, 'pending_jobs': 0, 'status_msg': 'active'}
{'backend_name': 'simulator_stabilizer', 'backend_version': '0.1.547',
'operational': True, 'pending_jobs': 0, 'status_msg': 'active'}
{'backend_name': 'ibm_brisbane', 'backend_version': '1.1.13', 'operational':
True, 'pending_jobs': 1414, 'status_msg': 'active'}
{'backend_name': 'ibm_kyoto', 'backend_version': '1.1.0', 'operational': True,
'pending_jobs': 1206, 'status_msg': 'active'}
```

We can execute the circuit on IBM's quantum simulator (supports up to 32 qubits). We only need to select the appropriate backend.

[12]:
```python
# Executing on the IBM Q Experience simulator

backend_sim = provider.get_backend('ibmq_qasm_simulator') # We choose the␣
↪backend

job_sim = execute(circ, backend_sim, shots=1024) # We execute the circuit,␣
↪selecting the number of repetitions or 'shots'

result_sim = job_sim.result() # We collect the results

counts = result_sim.get_counts(circ) # We obtain the frequency of each result␣
↪and we show them
print(counts)
plot_histogram(counts)
```

```
{'0': 516, '1': 508}
```

[12]:



To execute on one of the real quantum computers, we only need to select it as backend. We will use *job_monitor* to have live information on the job status

[13]:
```python
# Executing on the quantum computer

backend = provider.get_backend('ibm_kyoto')
```

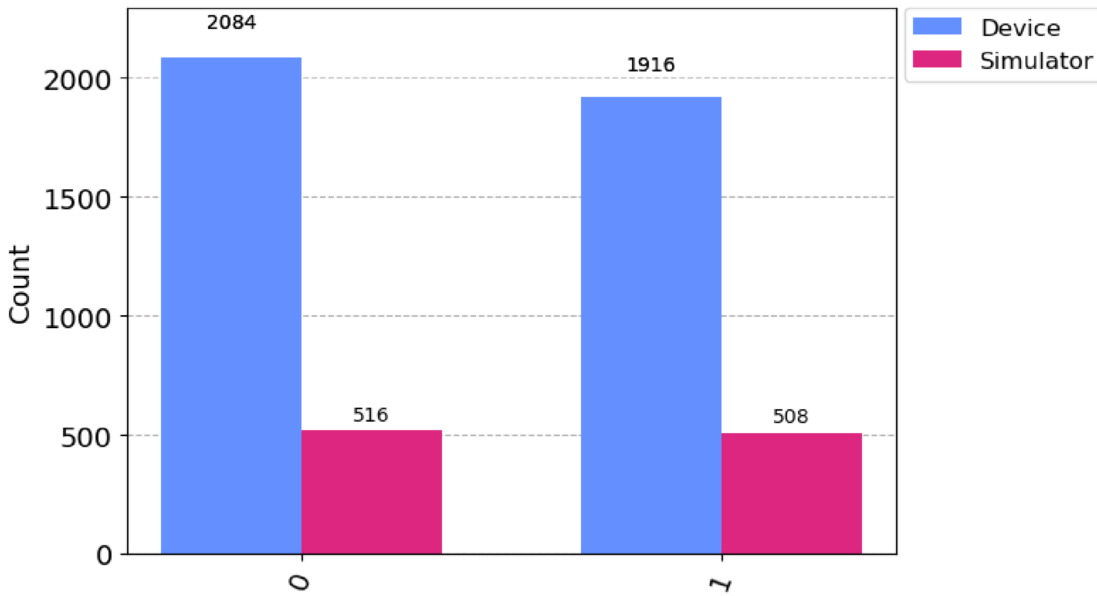
```
job_exp = execute(circ, backend=backend)
job_monitor(job_exp)
```

```
Job Status: job has successfully run
```

When the job is done, we can collect the results and compare them to the ones obtaine with the simulator

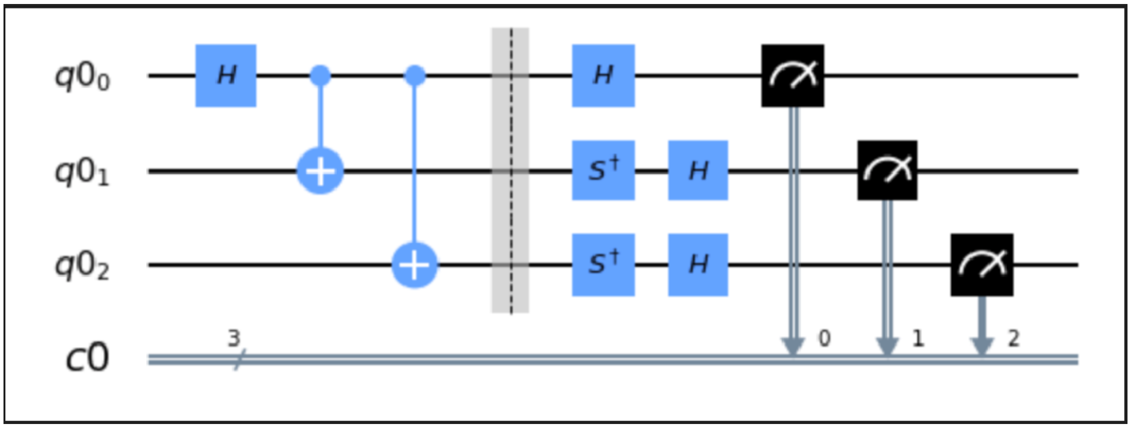
[14]:
```
result_exp = job_exp.result()
counts_exp = result_exp.get_counts(circ)
plot_histogram([counts_exp,counts], legend=['Device', 'Simulator'])
```

[14]:



0.5 EXERCISE TO DO

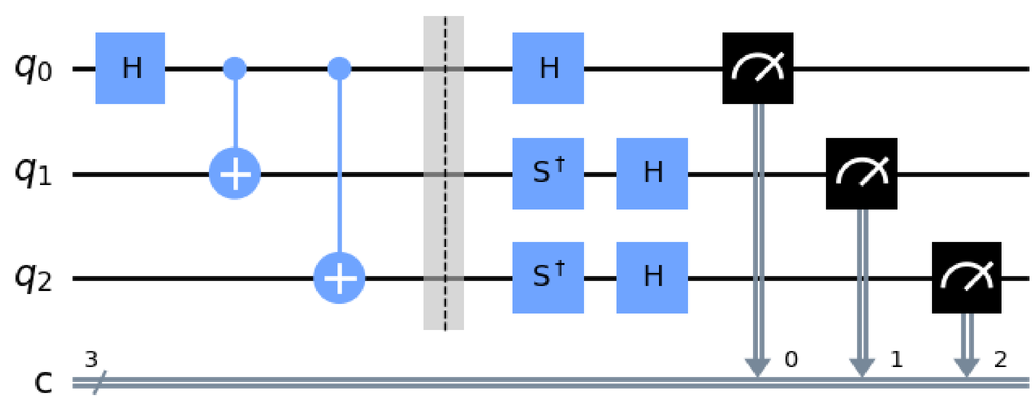
Based on the above notebook, execute both in a simulator and an IBM Quantum Computer the following circuit:



Comment on the final result (state) and provide your interpretation what this quantum circuit is doing.

[15]:
```
# creating the circuit
qc=QuantumCircuit(3,3)
qc.h(0)
qc.cx(0,1)
qc.cx(0,2)
qc.barrier()
qc.h(0)
qc.sdg(1)
qc.sdg(2)
qc.h(1)
qc.h(2)
for i in range(3):
    qc.measure(i,i)
qc.draw('mpl')
```

[15]:



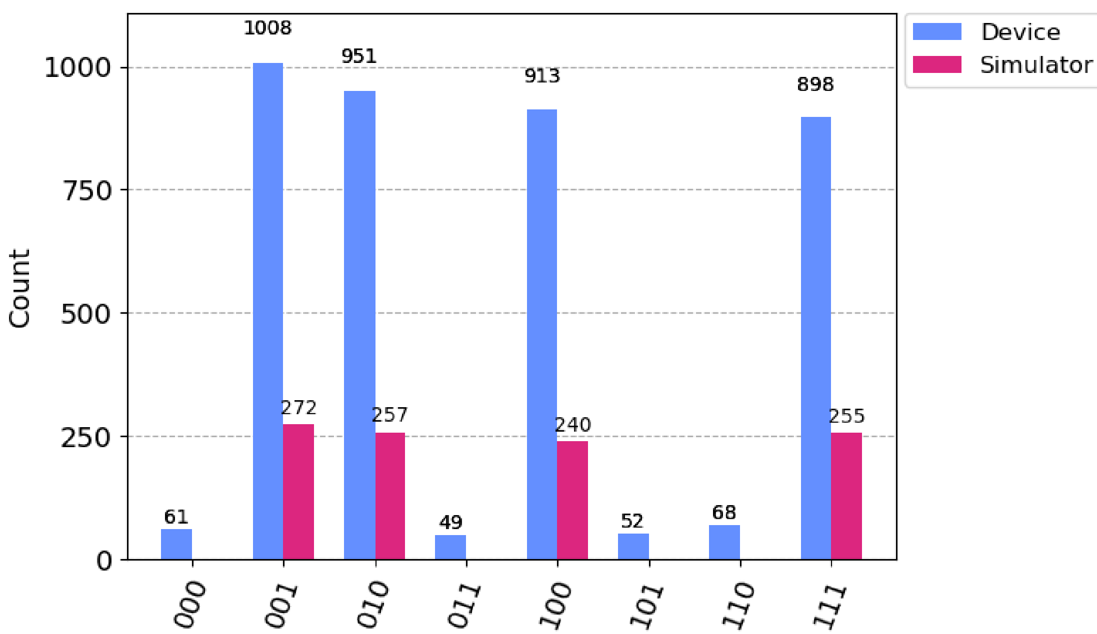
[16]:
```
# Executing on the ibm simulator
backend_sim = provider.get_backend('ibmq_qasm_simulator')

job_sim = execute(qc, backend_sim, shots=1024) # We execute the circuit,␣
↪selecting the number of repetitions or 'shots'

result_sim = job_sim.result() # We collect the results

counts = result_sim.get_counts(qc)
```

[17]:
```
# Executing on the real ibm quantum computer
backend = provider.get_backend('ibm_kyoto')

job_exp = execute(qc, backend=backend)
job_monitor(job_exp)
result_exp = job_exp.result()
counts_exp = result_exp.get_counts(qc)
```

```
Job Status: job has successfully run
```

[18]:
```
#display and compare them
plot_histogram([counts_exp,counts], legend=['Device', 'Simulator'])
```

[18]:



0.5.1 Considerations about the algorithm

The algorithm first part creates 3 qbits in entangle state, in fact the first part is the quantum circuit for building the GHZ state. The second part of the circuit is made for measuring the qbits in different basis, the *measure* function only measure on the Z base, so with the H and Sdg matrices and with some equivalences we can measure in X and Y basis too. Equivalences: - HZH = X - (SH)Z(HSdg) = Y

To conclude, the algorithm is computing an entangled state and measuring it in XYY basis. The difference between the simulator and the real execution is noise. In fact, as we can see in the previous plot, there is a low chance to get results such as 000 which shouldn't be there.

[1]:
```
#math explaining the results
#first part already seen in the first document
import numpy as np
H=np.matrix([[1,1],[1,-1]])/np.sqrt(2)
X=np.matrix([[0,1],[1,0]])
zero=np.matrix([[1],[0]])
one=np.matrix([[0],[1]])
I=np.matrix([[1,0],[0,1]])
CNOT=np.kron(I,np.kron(I,zero@zero.transpose()))+np.kron(I,np.kron(X,one@one.
 ↪transpose()))
CNOT3=np.kron(np.kron(I,I),zero@zero.transpose())+np.kron(np.kron(X,I),one@one.
 ↪transpose())
prebarrier=CNOT3@CNOT@np.kron(I,np.kron(I,H))
Sdg=np.matrix([[1,0],[0,-1j]])
postbarrier=np.kron(Sdg,np.kron(Sdg,I))
postbarrier=np.kron(H,np.kron(H,H))@postbarrier
three_bit_one=np.kron(np.kron(zero,zero),zero)
result=postbarrier@prebarrier@three_bit_one
result[result<0.00001]=0
print(result)
```

```
[[0. +0.j]
 [0.5+0.j]
 [0.5+0.j]
 [0. +0.j]
 [0.5+0.j]
 [0. +0.j]
 [0. +0.j]
 [0.5+0.j]]
```

[ ]:
```
```