TP2

June 21, 2020

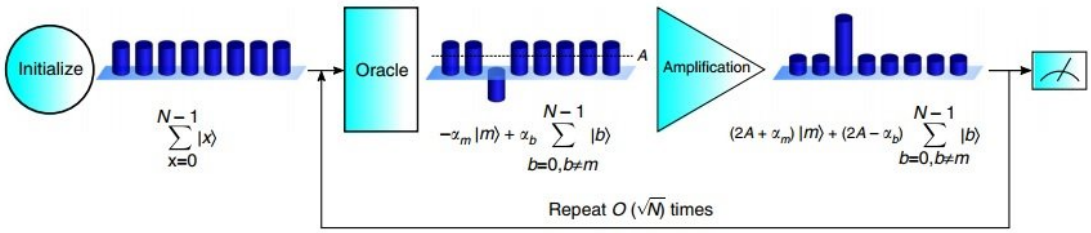
1 Trabalho Prático 2

Trabalho realizado no âmbito da unidade curricular de Interação e Concorrência pelas alunas: * Maria Barbosa - a85390 * Constança Elias - a83543

Neste trabalho, pretende-se encontrar um número numa lista de 8 elementos. Neste caso, será o número do nosso grupo módulo 8. Como somos o grupo 10, o número que procuramos é o 2 (10 mod 8).

Se o algoritmo que pretendemos desenvolver fosse feito em computadores clássicos, teríamos de percorrer, no caso médio, metade dos elementos. Com computadores quânticos, seguindo o algoritmo de Grover, basta apenas "percorrer 1 elemento".

Este algoritmo pode ser resumido na imagem seguinte.



The three stages of the 3-qubit Grover search algorithm: initialization, oracle, and amplification. Credit: C. Figgatt et al. Published in Nature Communications

No nosso trabalho, seguiremos os passos descritos na imagem anterior.

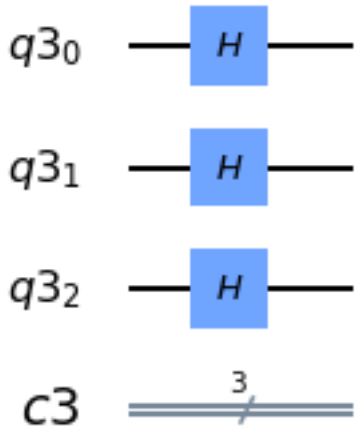
[40]:
```
from qiskit import *
%matplotlib inline
from qiskit.tools.visualization import *
```

1.1 Questão 1

Precisaremos de 3 qubits uma vez que 8 é representado em binário por 3 bits. Neste caso, procuramos 2 = 010. Vamos criar um circuito com 3 qubits e iniciá-lo com Hadamard gates para criar sobreposição uniforme.

[41]:
```
qr = QuantumRegister(3)
cr = ClassicalRegister(3)
qc_1 = QuantumCircuit(qr,cr) #circuito para implementar 1 vez o algoritmo␣
 ↪desenvolvido

for i in range(3):
    qc_1.h(i)

qc_1.draw(output='mpl')
```

[41]:



Vamos agora criar o oráculo de mudança de fase. Pretende-se mudar a fase do elemento que estamos à procura, ou seja, $U_f|w\rangle = -|w\rangle$.

$U_f|x\rangle = (-1)^{f(x)}|x\rangle$.

$$U_f|s\rangle = U_f \frac{1}{\sqrt{8}} (|000\rangle + |001\rangle + |010\rangle + |011\rangle + |100\rangle + |101\rangle + |110\rangle + |111\rangle) =$$

$$\frac{1}{\sqrt{8}} (|000\rangle + |001\rangle - |010\rangle + |011\rangle + |100\rangle + |101\rangle + |110\rangle + |111\rangle).$$

Para isso, temos de aplicar as portas X aos qubits que queremos a zero, para contrariar o comportamento da porta Z, ou seja, para que esta mude a fase do

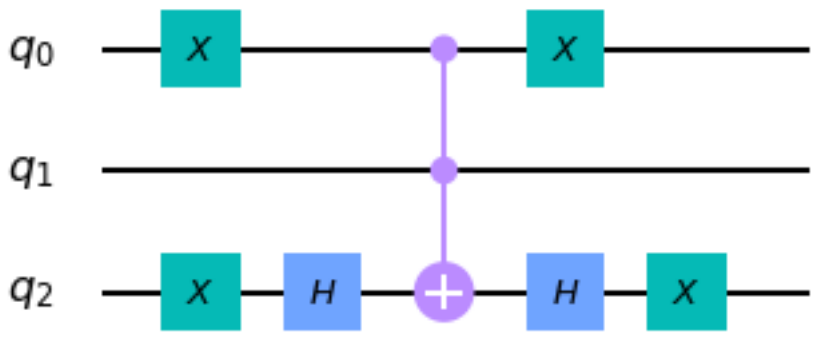
estado que pretendemos. E para aplicar a porta CCZ, uma vez que esta não existe em Qiskit, temos de aplicar uma combinação equivalente. H CCX H.

[42]:
```
# oracle to change phase  |010>
def phase_oracle(qc):

    #passar os Zeros a 1
    qc.x(0)
    qc.x(2)

    ## HXH - Controlled Z
    qc.h(2)
    qc.ccx(0,1,2)
    qc.h(2)

    ##voltar aos qubits iniciais
    qc.x(2)
    qc.x(0)
```

Vamos testar o oráculo num circuito de teste, para verificar que está bem implementado.

[43]:
```
qc_test = QuantumCircuit(3)

phase_oracle(qc_test)


qc_test.draw(output='mpl')
```

[43]:



[44]:
```
backend_unitary = Aer.get_backend('unitary_simulator')
```

[45]:
```
job = execute(qc_test, backend_unitary)
result = job.result()
matrix = result.get_unitary(qc_test, decimals=3)

print(matrix.real)
```

```
[[ 1.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  1.  0.  0.  0.  0.  0.  0.]
 [ 0.  0. -1.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  1.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  1.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  1.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  1.]]
```

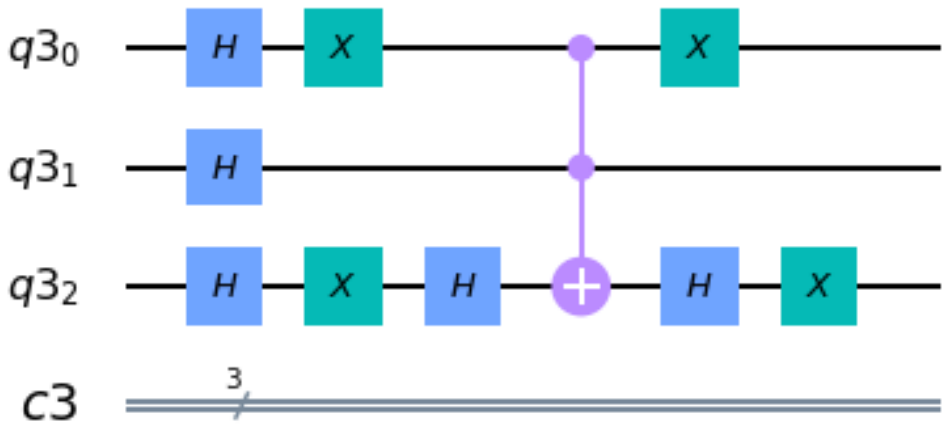
Com esta matriz, podemos confirmar que o oráculo da mudança de fase está bem definido pois obtemos o resultado correto (houve mudança de fase em 010).

Como o nosso número é uma capicua, é indiferente o target do CNOT estar no qubit 2 ou no qubit 0. Caso o nosso número fosse outro, teríamos de ter isso em atenção, uma vez que o Qiskit coloca o bit menos significativo no topo do circuito. Para obtermos uma matriz a apresentar os resultados de forma semelhante à anterior, teríamos de colocar o circuito invertido, ou seja, o target no início.

Podemos então aplicar este oráculo ao nosso circuito principal.

[46]:
```
phase_oracle(qc_1)

qc_1.draw(output='mpl')
```

[46]:



De seguida é necessário fazer a amplificação, o próximo passo do algoritmo.

Este passo serve para o computador quântico realçar a probabilidade do elemento desejado, amplificando a sua amplitude. Para isso, vai inverter todos os estados menos o que pretendemos e vai amplificar o pretendido.

Como podemos ler em Nielsen and Chuang[1], o operador de difusão do algoritmo de Grover pode ser expresso por

$$2|\psi\rangle\langle\psi| = H^{\otimes n}(2|0\rangle\langle 0| - I_N)H^{\otimes n}$$

sendo que $|\psi\rangle$ é o estado que procuramos. Assim sendo, é fácil verificar que o difusor a seguir definido implementa esta mesma fórmula. $2|0\rangle\langle 0| - I_N$ opera uma reflexão sobre o $|0\rangle$ por isso temos de aplicar as portas X a todos os qubits para inverter o efeito da porta CCZ (inverter a fase em 000 e não em 111), como implementado anteriormente no oráculo.

[47]:
```
def diffuser(qc):
    for i in range(3):
        qc.h(i)

    for i in range(3):
        qc.x(i)

    ## HXH = Controlled - Z
    qc.h(2)
    qc.ccx(0,1,2)
    qc.h(2)


    for i in range(3):
        qc.x(i)

    for i in range(3):
        qc.h(i)
```

[48]:
```
diffuser(qc_1)
```

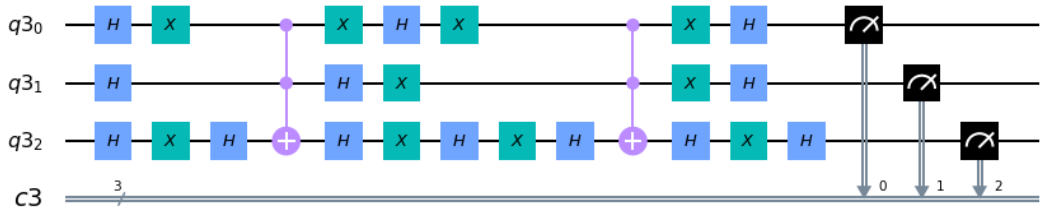
[49]:
```
job = execute(qc_1, backend_unitary)
result = job.result()
matrix = result.get_unitary(qc_1, decimals=3)

print(matrix.real)
```

```
[[-0.177  0.53   0.177  0.177  0.53   0.53   0.177  0.177]
 [-0.177 -0.177  0.177 -0.53   0.53  -0.177  0.177 -0.53 ]
 [-0.884 -0.177  0.177  0.177 -0.177 -0.177  0.177  0.177]
 [-0.177 -0.177 -0.53   0.177  0.53  -0.177 -0.53   0.177]
 [-0.177  0.53   0.177  0.177 -0.177 -0.177 -0.53  -0.53 ]
 [-0.177 -0.177  0.177 -0.53  -0.177  0.53  -0.53   0.177]
 [-0.177  0.53  -0.53  -0.53  -0.177 -0.177  0.177  0.177]
 [-0.177 -0.177 -0.53   0.177 -0.177  0.53   0.177 -0.53 ]]
```

Vamos introduzir as portas de medição, para poder medir o resultado final.

[50]:
```
for x in range(3):
    qc_1.measure(qr[x], cr[x])

qc_1.draw(output='mpl')
```

[50]:

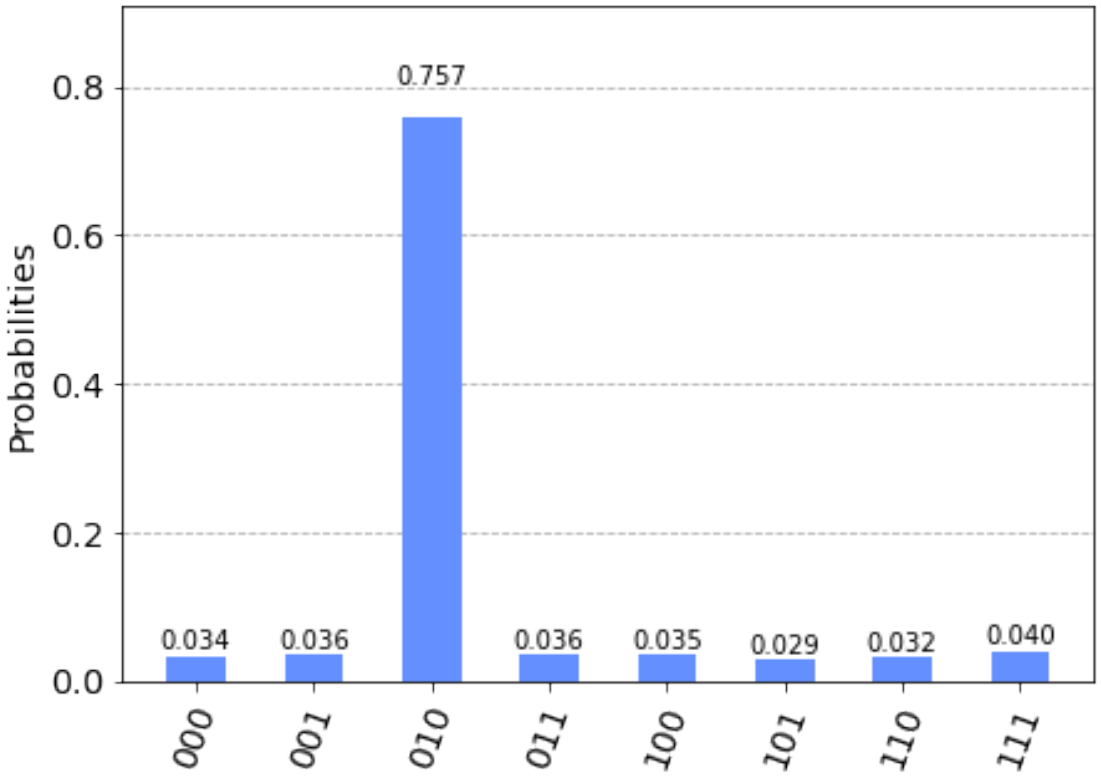


Vamos agora fazer a simulação do algoritmo.

[51]:
```
backend = Aer.get_backend("qasm_simulator")
```

[52]:
```
shots=1024
result = execute(qc_1, backend, shots=shots).result()
counts_sim_1 = result.get_counts(qc_1)
plot_histogram(counts_sim_1)
```

[52]:



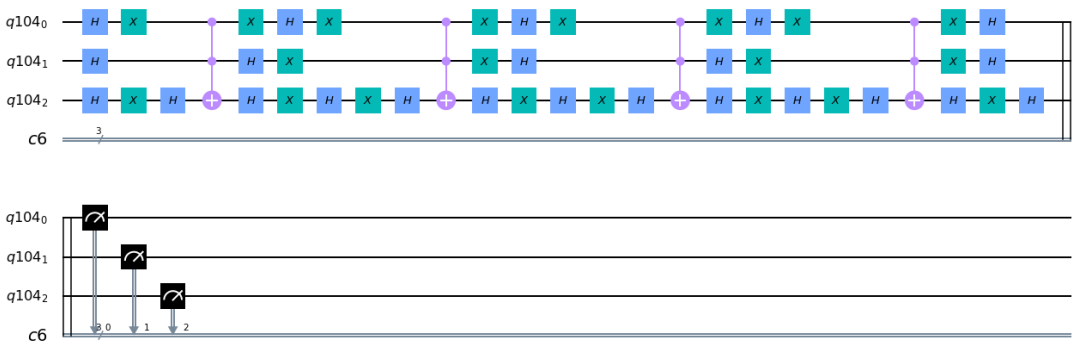
Para tentar obter resultados mais corretos, vamos criar um novo circuito e desta vez aplicar o algoritmo duas vezes, que é o valor aproximado, por defeito, de $\sqrt{N}$.

[146]:
```
qr = QuantumRegister(3)
cr = ClassicalRegister(3)
qc_2 = QuantumCircuit(qr, cr)

for i in range(3):
    qc_2.h(i)

for i in range(2):
    phase_oracle(qc_2)
    diffuser(qc_2)
```

É necessário aplicar as gates de medida para depois podermos fazer a medição. Estas gates vão introduzir mais erros no nosso circuito.

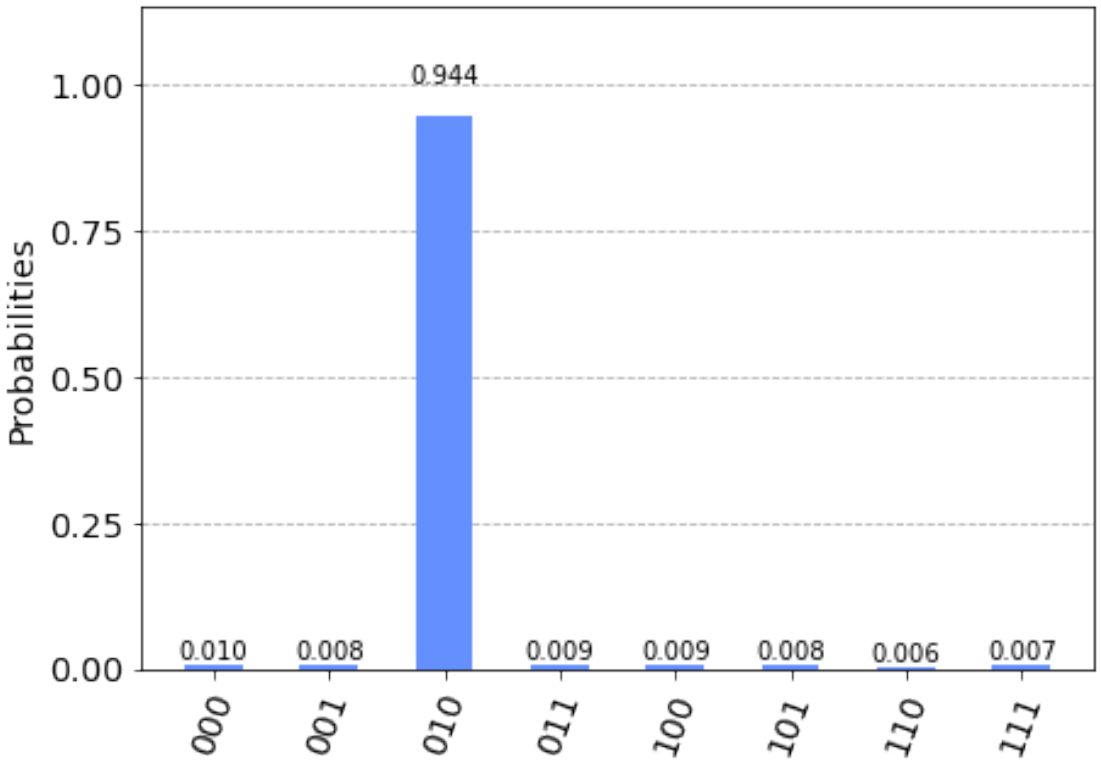
[147]:
```
for x in range(3):
    qc_2.measure(qr[x], cr[x])

qc_2.draw(output='mpl')
```

[147]:



[148]:
```
shots=1024
result = execute(qc_2, backend, shots=shots).result()
counts_sim_2 = result.get_counts(qc_2)
plot_histogram(counts_sim_2)
```

[148]:



Pelo gráfico anterior, podemos observar que os resultados melhoraram.

1.2 Questão 2

Vamos usar modelos de simulacão de ruído do Aer para prever qual é a melhor optimização a fazer ao algoritmo.

Começamos por fazer import dos packages necessários para construir os modelos de ruido para efetuar as simulações.

[58]:
```
from qiskit.providers.aer.noise import NoiseModel
import qiskit.providers.aer.noise as noise
```

Podemos construir modelos de ruído que podem ser usados adicionando QuantumErrors ao circuito, redefinindo ou medindo instruções e ReadoutError para medir instruções.

Vamos começar por escolher uma máquina, para obter um modelo de ruído de acordo com as características da mesma. Para isso, vamos primeiro ver as informações de cada uma.

[59]:
```
provider = IBMQ.load_account()
provider.backends()
```

[59]:
```
[<IBMQSimulator('ibmq_qasm_simulator') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmqx2') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibmq_16_melbourne') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_vigo') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_ourense') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_london') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_burlington') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_essex') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_armonk') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_rome') from IBMQ(hub='ibm-q', group='open',
project='main')>]
```

[60]:
```
# Backend overview
import qiskit.tools.jupyter

%qiskit_backend_overview
```

```
VBox(children=(HTML(value="<h2 style ='color:#ffffff; background-color:#000000;padding-top: 1%;
```

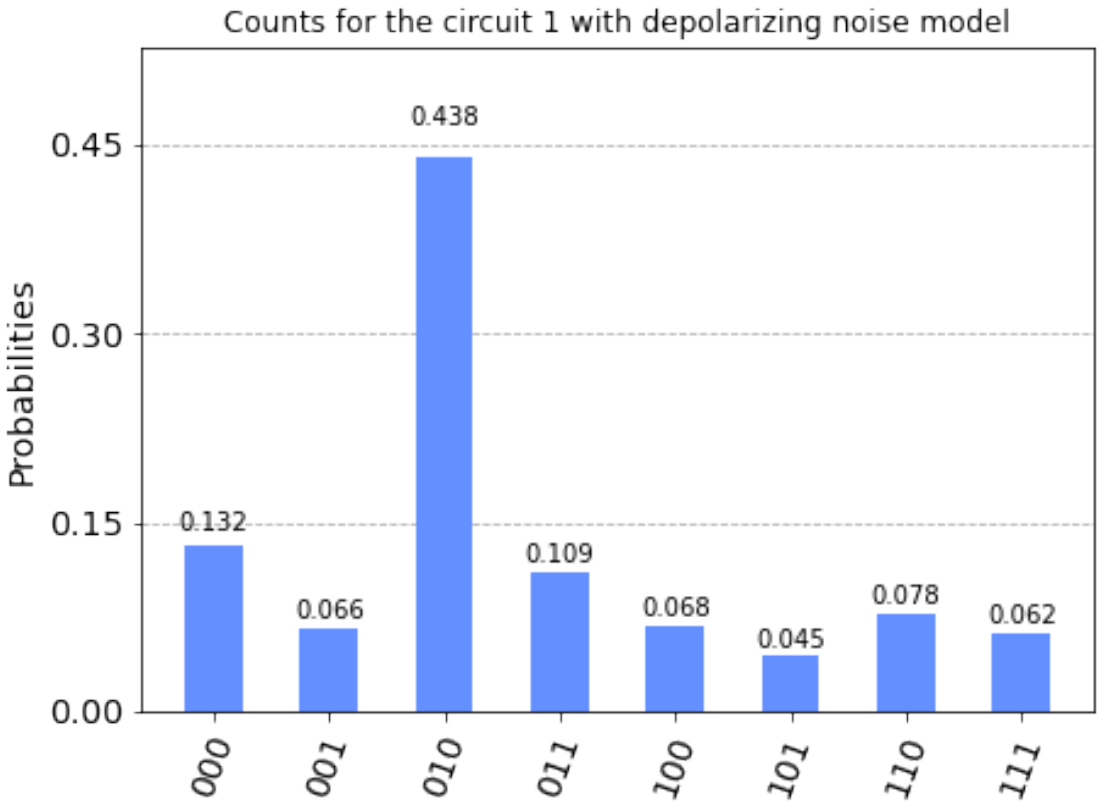
Optámos por obter a simulação com a máquina de Melbourne, uma vez que o erro médio de medição é maior. Assim, podemos analisar os resultados no pior cenário.

[61]:
```
my_provider_ibmq = IBMQ.get_provider(hub='ibm-q', group='open', project='main')

backend_device = my_provider_ibmq.get_backend('ibmq_16_melbourne')

coupling_map = backend_device.configuration().coupling_map

# Construct the noise model from backend properties
noise_model = NoiseModel.from_backend(backend_device)

basis_gates = noise_model.basis_gates
```

Começamos por executar o modelo de ruído no primeiro circuito.

[66]:
```
# Execute noisy simulation and get counts
result_noise = execute(qc_1, backend,
                       noise_model=noise_model,
                       coupling_map=coupling_map,
                       basis_gates=basis_gates).result()

counts_noise_1 = result_noise.get_counts(qc_1)
plot_histogram(counts_noise_1, title="Counts for the circuit 1 with depolarizing␣
 ↪noise model")
```

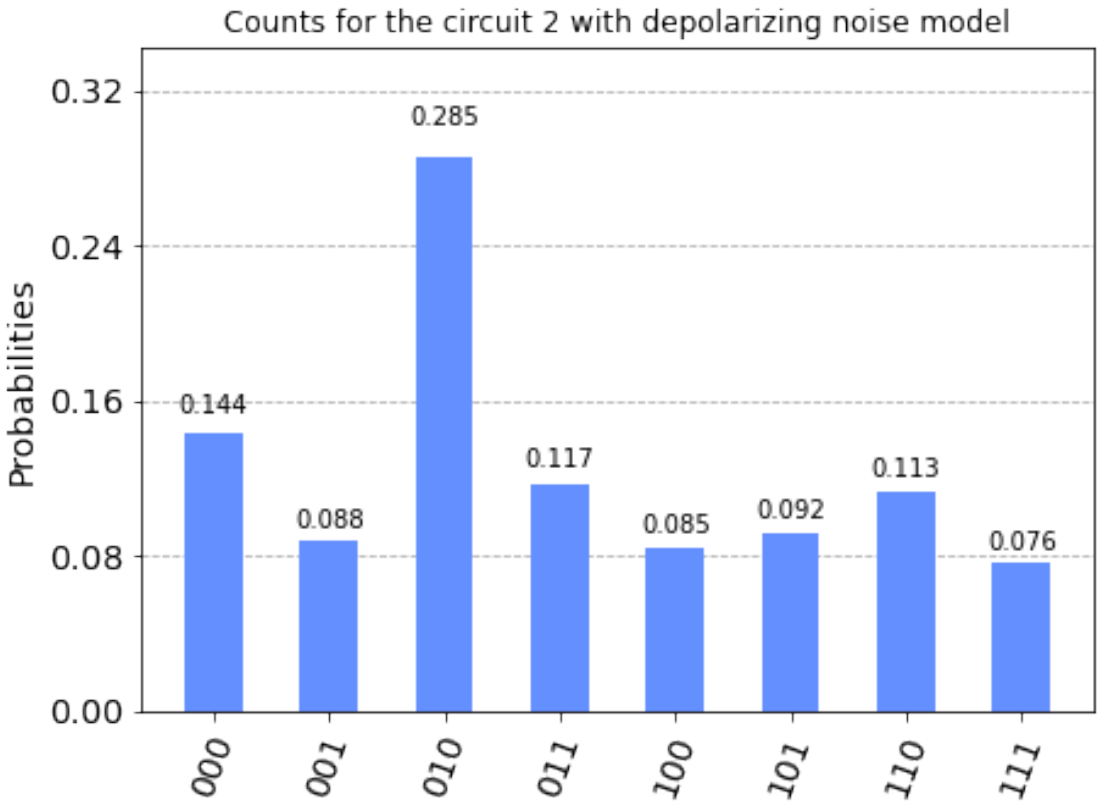
[66]:



Executamos agora o modelo de ruído no segundo circuito para poder comparar resultados.

[149]:
```
# Execute noisy simulation and get counts
result_noise = execute(qc_2, backend,
                       noise_model=noise_model,
                       coupling_map=coupling_map,
                       basis_gates=basis_gates).result()

counts_noise_2 = result_noise.get_counts(qc_2)
plot_histogram(counts_noise_2, title="Counts for the circuit 2 with depolarizing␣
 ↪noise model")
```

[149]:



Comparar resultados Com base na função definida no guião 4, podemos obter as probabilidades de obter cada estado.

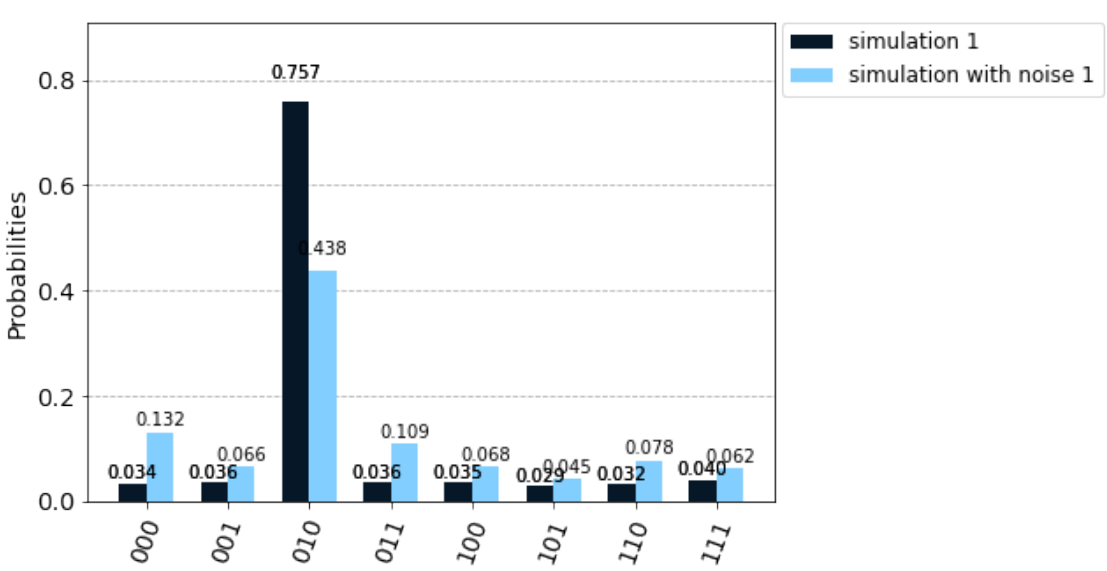
[70]:
```
def resume(counts_raw):
    s000 = s001 = s010 = s011 = s100 = s101 = s110 = s111 = 0
    k = counts_raw.keys()
    lk=list(k)
    for c in lk:
        if c == '000':
            s000 = s000 + counts_raw.get(c)
        elif c == '001':
            s001 = s001 + counts_raw.get(c)
        elif c == '010':
            s010 = s010 + counts_raw.get(c)
        elif c == '011':
            s011 = s011 + counts_raw.get(c)
        elif c == '100':
            s100 = s100 + counts_raw.get(c)
        elif c == '101':
            s101 = s101 + counts_raw.get(c)
        elif c == '110':
```

```
            s110 = s110 + counts_raw.get(c)
        else:
            s111 = s111 + counts_raw.get(c)

    return({'000': s000, '001': s001, '010': s010, '011': s011, '100': s100,␣
 ↪'101': s101, '110': s110, '111': s111})
```

[71]:
```
cn_1 = resume(counts_noise_1)
c_1 = resume(counts_sim_1)
```

[72]:
```
plot_histogram([c_1,cn_1], legend= ['simulation 1','simulation with noise 1'],␣
 ↪color=['#061727','#82cfff'])
```

[72]:

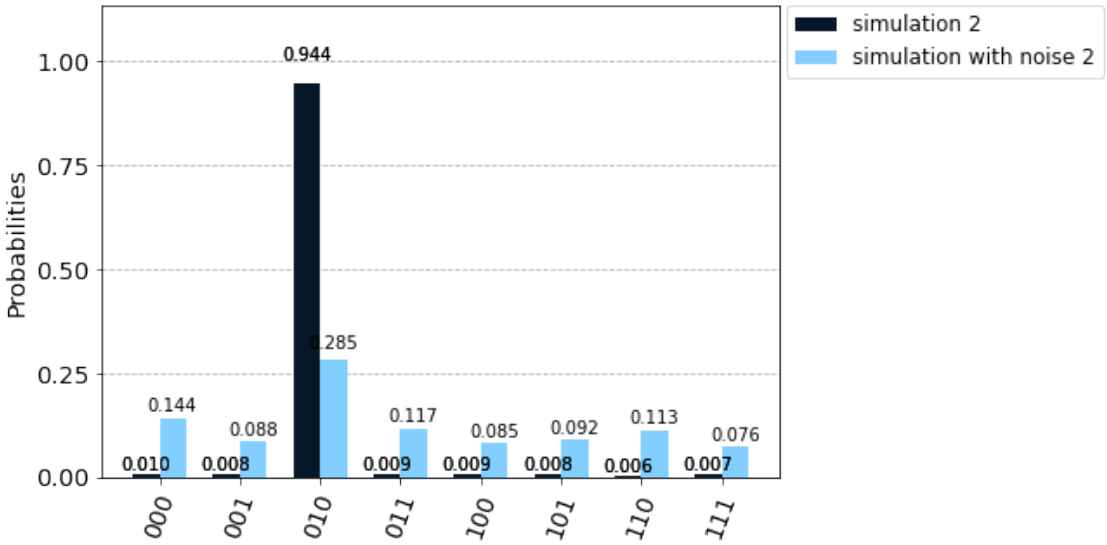


[150]:
```
cn = resume(counts_noise_2)
c = resume(counts_sim_2)
print(cn)
print(c)
```

```
{'000': 147, '001': 90, '010': 292, '011': 120, '100': 87, '101': 94, '110':
116, '111': 78}
{'000': 10, '001': 8, '010': 967, '011': 9, '100': 9, '101': 8, '110': 6, '111':
7}
```

[151]:
```
plot_histogram([c,cn], legend= ['simulation 2','simulation with noise 2'],␣
 ↪color=['#061727','#82cfff'])
```

[151]:



Com esta simulação de ruído, já estamos a ver que o primeiro algoritmo dá melhores resultados. Vamos confirmar isto e explicar o porquê na próxima questão do trabalho.

1.3 Questão 3

Nesta questão pretende-se executar o algoritmo no IBM Q. O primeiro passo, consite em carregar o nosso IBM Q Experience.

[131]:
```
IBMQ.
 ↪save_account("bd78901eadc1f47deac47e3d7b9942fbeed93c17747ee105327cf26f48a772bc20c07f0f2d0ec9e
provider = IBMQ.load_account()
provider.backends()
```

```
configrc.store_credentials:WARNING:2020-06-21 18:17:59,181: Credentials already
present. Set overwrite=True to overwrite.
ibmqfactory.load_account:WARNING:2020-06-21 18:17:59,804: Credentials are
already in use. The existing account in the session will be replaced.
```

[131]:
```
[<IBMQSimulator('ibmq_qasm_simulator') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmqx2') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibmq_16_melbourne') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_vigo') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_ourense') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_london') from IBMQ(hub='ibm-q', group='open',
```

```
project='main')>,
 <IBMQBackend('ibmq_burlington') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_essex') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_armonk') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_rome') from IBMQ(hub='ibm-q', group='open',
project='main')>]
```

[78]:
```
backends_list =provider.backends( simulator=False, open_pulse=False)
```

[79]:
```
# Backend overview
import qiskit.tools.jupyter

%qiskit_backend_overview
```

```
VBox(children=(HTML(value="<h2 style ='color:#ffffff; background-color:#000000;padding-top: 1%;
```

Vamos em seguida especificar que pretendemos usar a máquina quântica de Essex.

[81]:
```
backend_device = provider.get_backend('ibmq_essex')

# See backend information
backend_device
```

```
VBox(children=(HTML(value="<h1 style='color:#ffffff;background-color:#000000;padding-top: 1%;pad
```

[81]:
```
<IBMQBackend('ibmq_essex') from IBMQ(hub='ibm-q', group='open', project='main')>
```

[82]:
```
from qiskit.tools.monitor import backend_overview, backend_monitor

backend_monitor(backend_device)
```

```
ibmq_essex
==========
Configuration
-------------
    n_qubits: 5
    operational: True
    status_msg: active
    pending_jobs: 1
    backend_version: 1.0.1
    basis_gates: ['u1', 'u2', 'u3', 'cx', 'id']
    local: False
    simulator: False
    sample_name: Giraffe
```

```
    max_experiments: 75
    open_pulse: False
    quantum_volume: 8
    credits_required: True
    online_date: 2019-09-13T04:00:00Z
    n_registers: 1
    allow_object_storage: True
    backend_name: ibmq_essex
    description: 5 qubit device Essex
    meas_map: [[0, 1, 2, 3, 4]]
    url: None
    coupling_map: [[0, 1], [1, 0], [1, 2], [1, 3], [2, 1], [3, 1], [3, 4], [4,
3]]
    allow_q_object: True
    conditional: False
    max_shots: 8192
    memory: True

Qubits [Name / Freq / T1 / T2 / U1 err / U2 err / U3 err / Readout err]
----------------------------------------------------------------------
    Q0 / 4.4994 GHz / 93.37511 µs / 136.43708 µs / 0 / 0.00039 / 0.00077 / 0.045
    Q1 / 4.69464 GHz / 67.95974 µs / 124.87987 µs / 0 / 0.0005 / 0.00101 /
0.03333
    Q2 / 4.66332 GHz / 109.09747 µs / 154.85374 µs / 0 / 0.00093 / 0.00185 /
0.025
    Q3 / 4.65377 GHz / 88.29824 µs / 123.26329 µs / 0 / 0.0005 / 0.001 / 0.03
    Q4 / 4.61093 GHz / 55.15717 µs / 79.84517 µs / 0 / 0.00181 / 0.00361 /
0.02167

Multi-Qubit Gates [Name / Type / Gate Error]
--------------------------------------------
    cx0_1 / cx / 0.01167
    cx1_0 / cx / 0.01167
    cx1_2 / cx / 0.02066
    cx1_3 / cx / 0.01154
    cx2_1 / cx / 0.02066
    cx3_1 / cx / 0.01154
    cx3_4 / cx / 0.03133
    cx4_3 / cx / 0.03133
```

[83]:
```
%qiskit_job_watcher
```

```
Accordion(children=(VBox(layout=Layout(max_width='710px', min_width='710px')),), layout=Layout(m


<IPython.core.display.Javascript object>
```

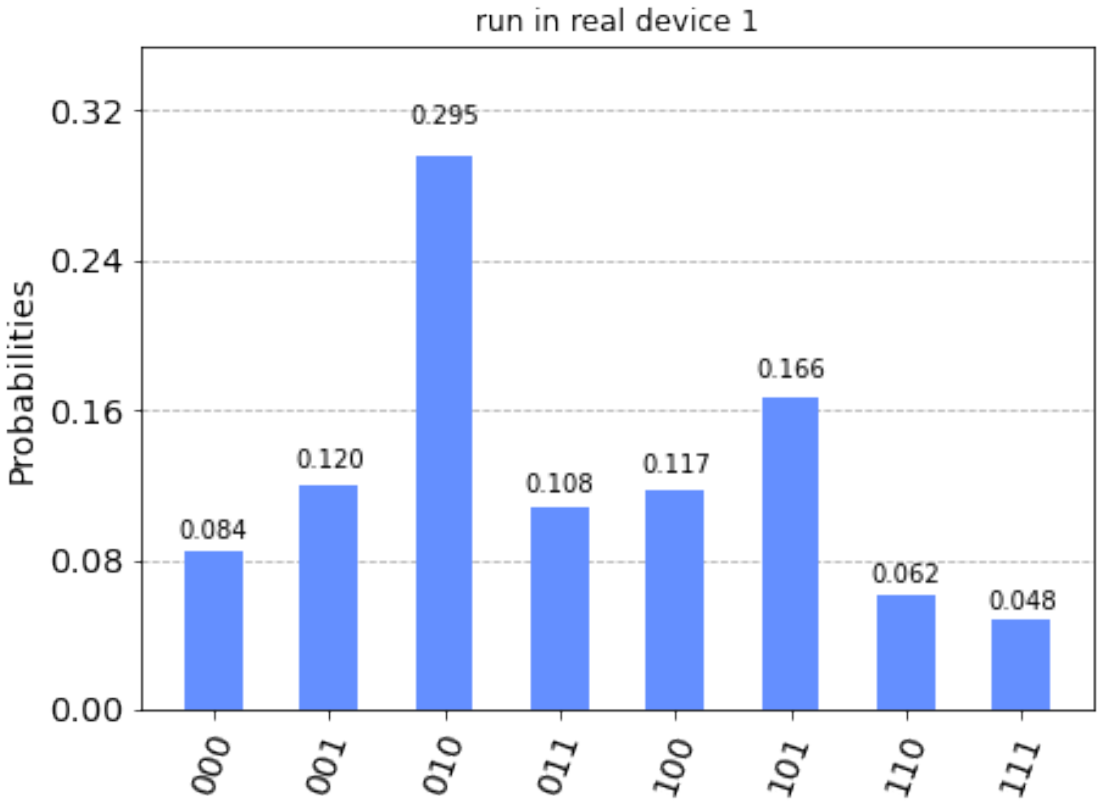
Vamos executar o primeiro circuito, em que o algoritmo é executado apenas uma vez.

[84]:
```
shots=1024
job = execute(qc_1, backend_device, shots=shots)

jobID = job.job_id()

print('JOB ID: {}'.format(jobID))
```

```
JOB ID: 5eef88526bb14a001cca2ba8
```

[86]:
```
job_get=backend_device.retrieve_job("5eef88526bb14a001cca2ba8")

job_get.error_message()

result = job_get.result()
counts_real_1 = result.get_counts(qc_1)

plot_histogram(counts_real_1, title='run in real device 1')
```

[86]:

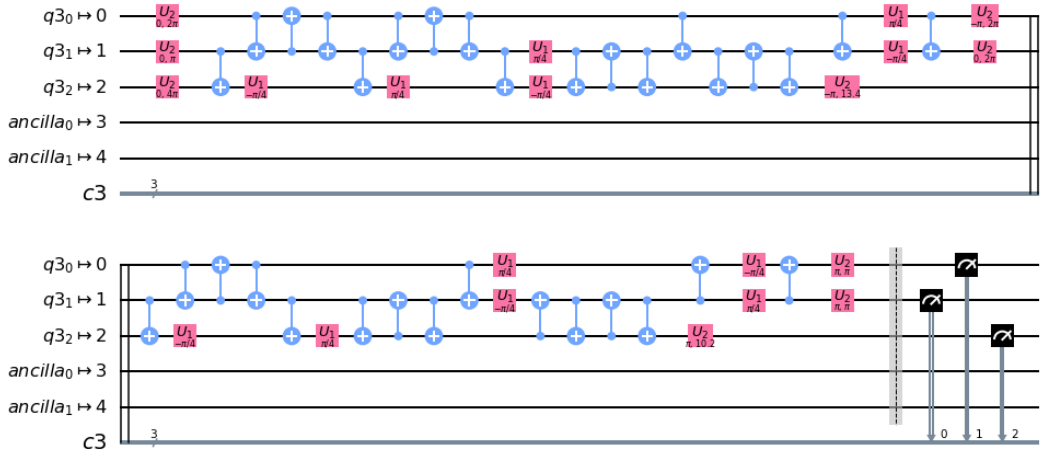


[87]:
```
from qiskit.compiler import transpile
```

Pretendemos agora obter uma otimização de ambos os circuitos (1 e 2). Decidimos optar pela otimização de nível 1.

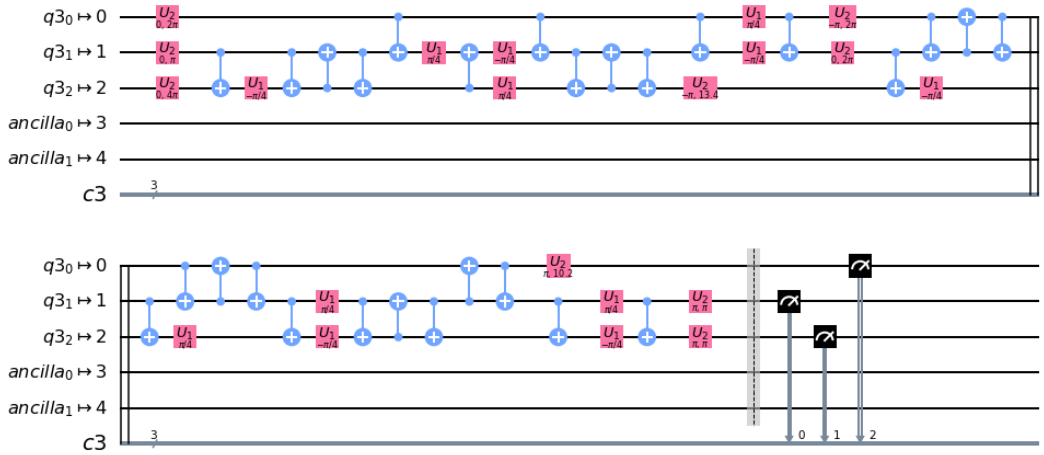
[92]:
```
qc_t_real_1 = transpile(qc_1, backend=backend_device)

qc_t_real_1.draw(output='mpl', scale=0.5)
```

[92]:



[93]:
```
qc_optimized_1 = transpile(qc_1, backend=backend_device, optimization_level=1)
qc_optimized_1.draw(output='mpl', scale=0.5)
```

[93]:



[94]:
```
qc_t_real_1.depth()
```

[94]: 42

[95]:
```
qc_optimized_1.depth()
```

[95]: 38

[97]:
```
job_exp = execute(qc_optimized_1, backend_device, shots = shots)

# job_id allows you to retrive old jobs
jobID = job_exp.job_id()

print('JOB ID: {}'.format(jobID))

job_exp.result().get_counts(qc_optimized_1)
```

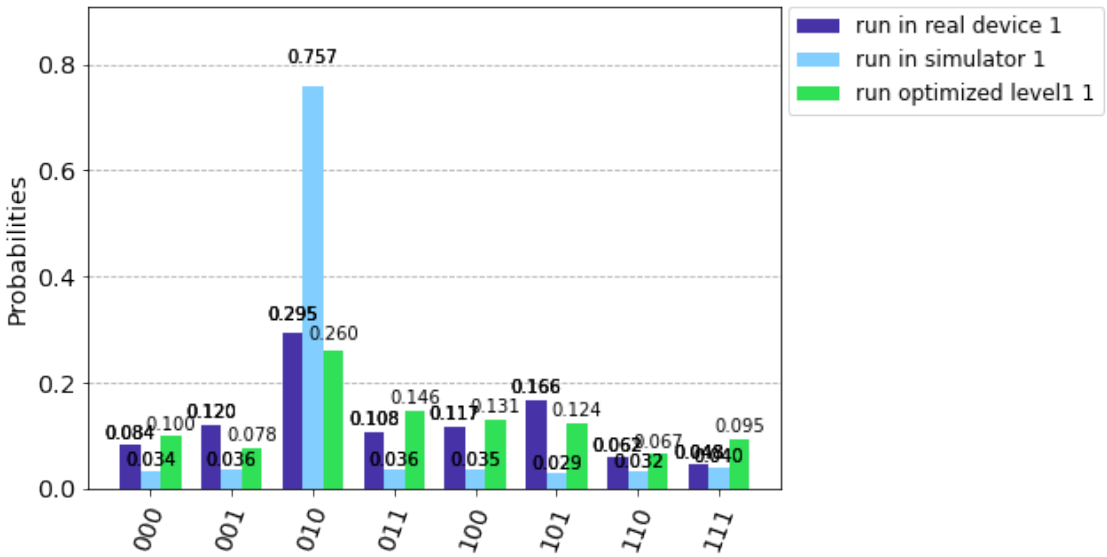
```
JOB ID: 5eef8910571601001 98ff387
```

[97]:
```
{'100': 134,
 '110': 69,
 '101': 127,
 '001': 80,
 '011': 149,
 '111': 97,
 '010': 266,
 '000': 102}
```

[99]:
```
job_get_o=backend_device.retrieve_job("5eef8910571601001 98ff387")

result_real_o = job_get_o.result(timeout=3600, wait=5)

counts_opt_1 = result_real_o.get_counts(qc_optimized_1)
```

[100]:
```
plot_histogram([counts_real_1, counts_sim_1, counts_opt_1], legend=[ 'run in␣
↪real device 1', 'run in simulator 1', 'run optimized level1 1'],␣
↪color=['#4732a7','#82cfff','#30e157'])
```

[100]:



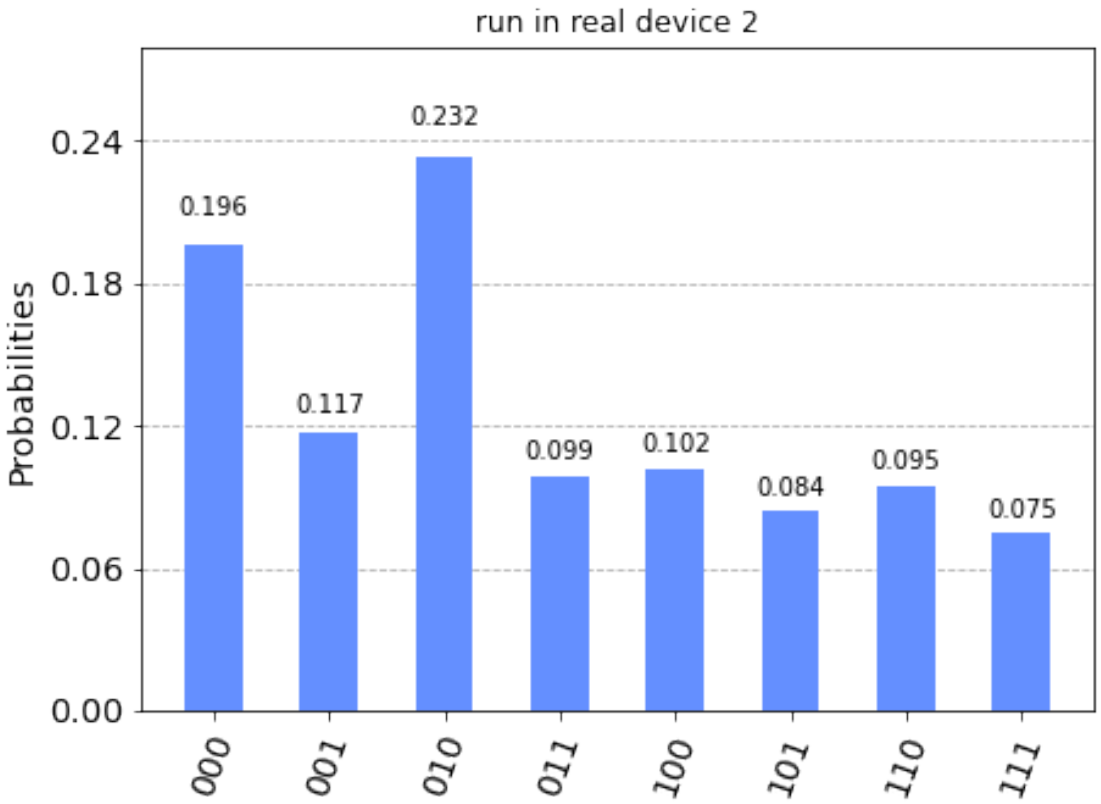
Vamos agora executar o segundo circuito.

[152]:
```
job = execute(qc_2, backend_device, shots=shots)

jobID = job.job_id()

print('JOB ID: {}'.format(jobID))
```

```
JOB ID: 5eef9cd91ef17a0019563f41
```

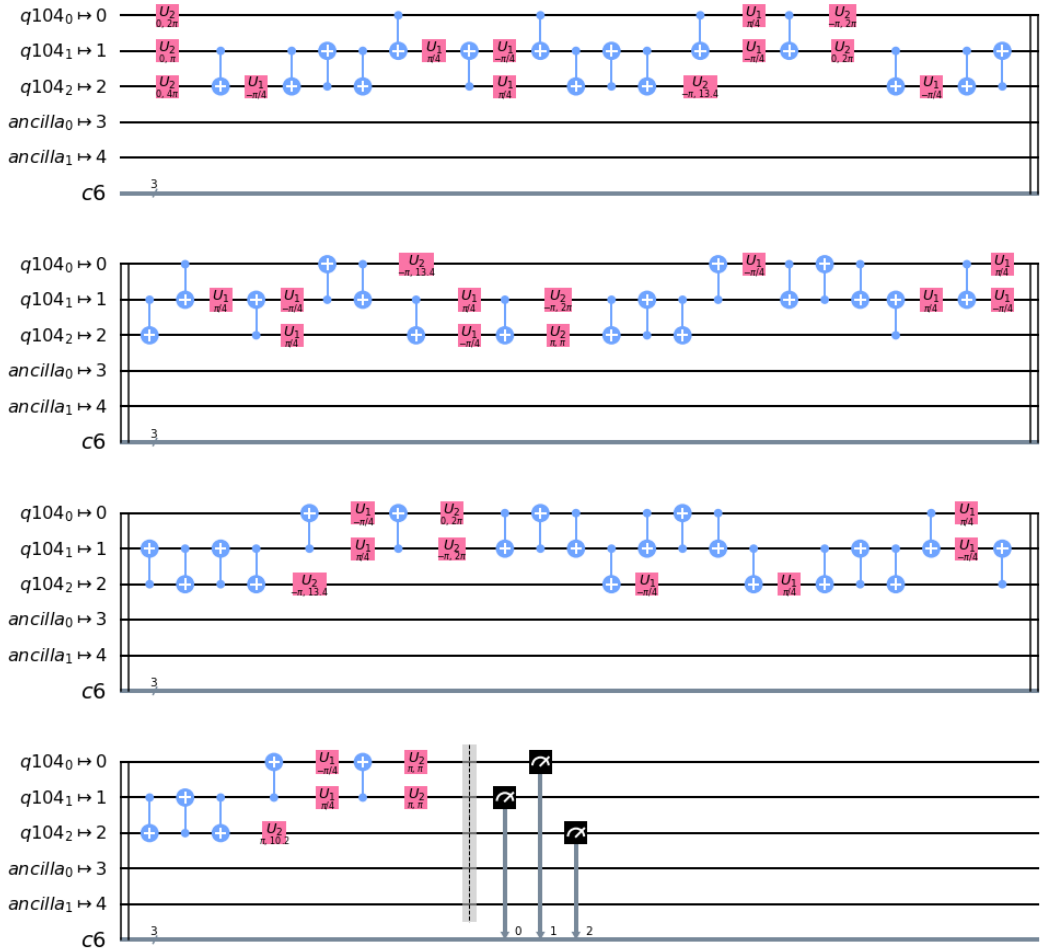
[153]:
```
job_get=backend_device.retrieve_job("5eef9cd91ef17a0019563f41")

job_get.error_message()

result = job_get.result()
counts_real_2 = result.get_counts(qc_2)

plot_histogram(counts_real_2, title='run in real device 2')
```

[153]:



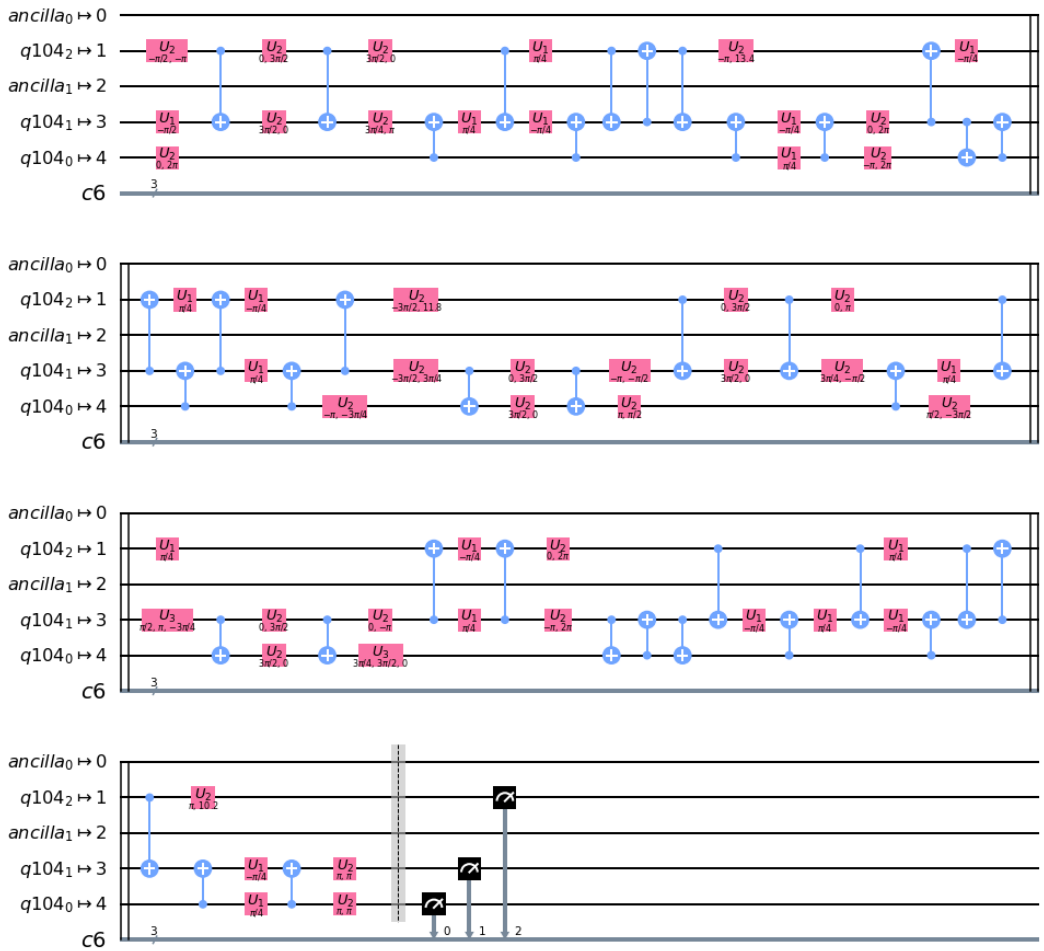
[155]:
```
qc_t_real_2 = transpile(qc_2, backend=backend_device)

qc_t_real_2.draw(output='mpl', scale=0.5)
```

[155]:



[156]:
```
qc_optimized_2 = transpile(qc_2, backend=backend_device, optimization_level = 3)
qc_optimized_2.draw(output='mpl', scale=0.5)
```

[156]:



[157]:
```
qc_t_real_2.depth()
```

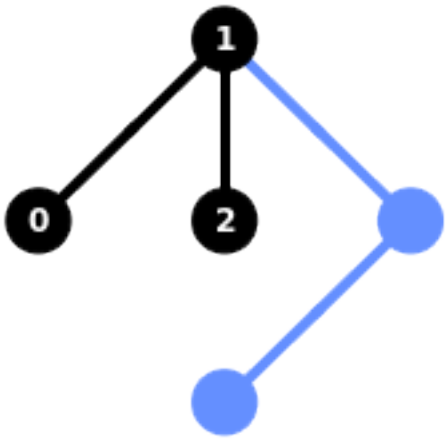
[157]: 76

[158]:
```
qc_optimized_2.depth()
```

[158]: 65

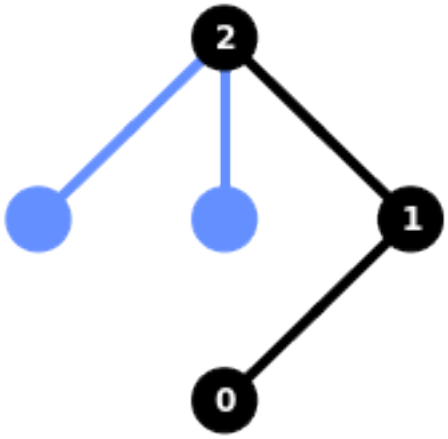
[159]:
```
from qiskit.visualization import plot_circuit_layout
plot_circuit_layout(qc_t_real_2, backend_device)
```

[159]:



[160]:
```
plot_circuit_layout(qc_optimized_2, backend_device)
```

[160]:



[161]:
```
job_exp = execute(qc_optimized_2, backend_device, shots = shots)

# job_id allows you to retrive old jobs
jobID = job_exp.job_id()

print('JOB ID: {}'.format(jobID))

job_exp.result().get_counts(qc_optimized_2)
```

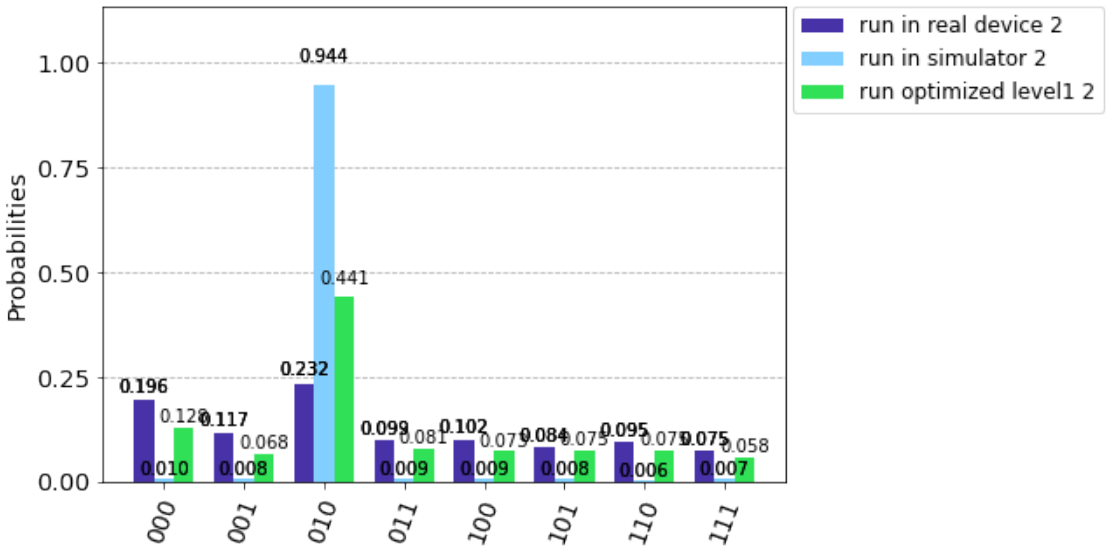
```
JOB ID: 5eef9d6dd8daae001a5f4324
```

[161]:
```
{'100': 75,
 '110': 77,
 '101': 77,
 '001': 70,
 '011': 83,
 '111': 59,
 '010': 452,
 '000': 131}
```

[162]:
```
job_get_o=backend_device.retrieve_job("5eef9d6dd8daae001a5f4324")

result_real_o = job_get_o.result(timeout=3600, wait=5)

counts_opt_2 = result_real_o.get_counts(qc_optimized_2)
```

[163]:
```
plot_histogram([counts_real_2, counts_sim_2, counts_opt_2], legend=[ 'run in␣
↪real device 2', 'run in simulator 2', 'run optimized level1 2'],␣
↪color=['#4732a7','#82cfff','#30e157'])
```

[163]:



Pelos resultados obtidos na execução dos dois circuitos podemos ver que, embora a simulação obtida para o segundo circuito seja melhor, os resultados obtidos na máquina real para o primeiro circuito são superiores. Isto deve-se ao facto a profundidade do primeiro circuito ser menor, uma vez que são usadas menos portas. Assim, embora teoricamente seja suposto repetir duas vezes o algoritmo (que é o valor arrondado por defeito de $\sqrt{8}$), o compromisso estabelecido entre o número de portas utilizadas e a repetição do circuito, mostra que é melhor optar por ter menos profundidade no circuito para se traduzir em menos erros.

1.4 Questão 4

O objetivo desta questão é proceder ao tratamento de erros, recorrendo ao módulo Ignis.

[111]:
```
# Import measurement calibration functions
from qiskit.ignis.mitigation.measurement import (complete_meas_cal,␣
 ↪tensored_meas_cal,
                                                  CompleteMeasFitter,␣
 ↪TensoredMeasFitter)
```

Primeiro, vamos gerar uma lista de circuitos de calibração de medição para todo o espaço de Hilbert. Neste caso, como temos 3 qubits, então obtemos $2^3 = 8$ circuitos de calibração.

[112]:
```
# Generate the calibration circuits
qr = QuantumRegister(x)

# meas_calibs:
# list of quantum circuit objects containing the calibration circuits
```

```
# state_labels:
# calibration state labels
meas_calibs, state_labels = complete_meas_cal(qubit_list=None, qr=3, cr = 3,␣
 ↪circlabel='mcal')
```

[113]:
```
state_labels
```

[113]:
```
['000', '001', '010', '011', '100', '101', '110', '111']
```

Vamos em seguida calcular a matriz de calibração. No caso de não existir ruído na maquina escolhida a matriz de calibração é a matriz identidade 8 x 8.

Se executarmos os circuitos de calibração num simulador, então a matriz obtida não tem ruído (matriz identidade).

[114]:
```
# Execute the calibration circuits without noise
backend = qiskit.Aer.get_backend('qasm_simulator')
job = qiskit.execute(meas_calibs, backend=backend, shots=1000)
cal_results = job.result()

# The calibration matrix without noise is the identity matrix
meas_fitter = CompleteMeasFitter(cal_results, state_labels, circlabel='mcal')
print(meas_fitter.cal_matrix)
```

```
[[1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1.]]
```

Vamos agora executar no circuito de calibração recorrendo ao IBM_Q. Neste caso, por ser uma máquina real, vamos obter ruído e por isso a matriz não será a identidade.

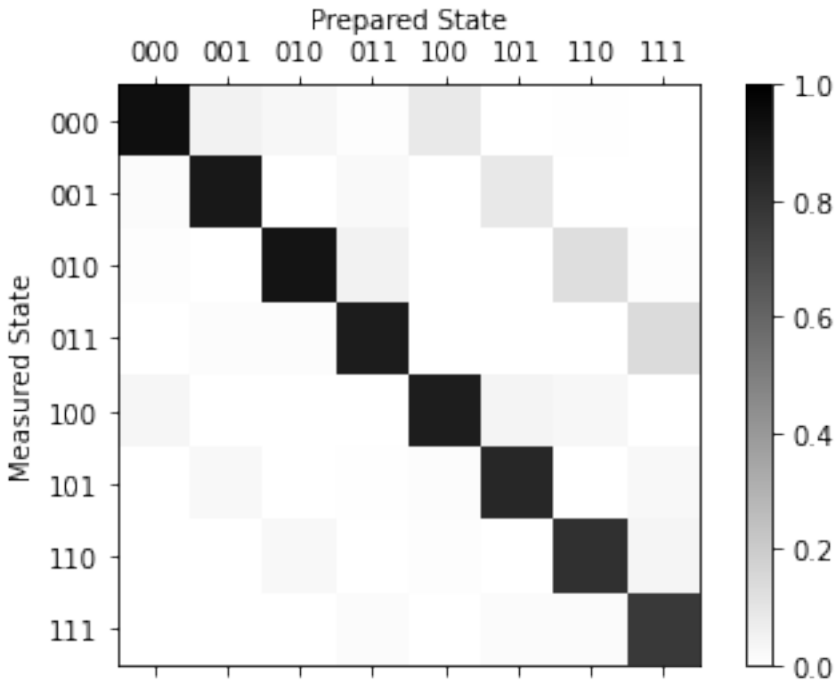
[115]:
```
job_ignis = execute(meas_calibs, backend=backend_device, shots=shots)

jobID_run_ignis = job_ignis.job_id()

print('JOB ID: {}'.format(jobID_run_ignis))
```

```
JOB ID: 5eef8a9c61ad6a00190be084
```

[116]:
```
job_get=backend_device.retrieve_job("5eef8a9c61ad6a00190be084")

cal_result = job_get.result()
```

[117]:
```
meas_fitter = CompleteMeasFitter(cal_result, state_labels, circlabel='mcal')
print(meas_fitter.cal_matrix)
# Plot the calibration matrix
meas_fitter.plot_calibration()
```

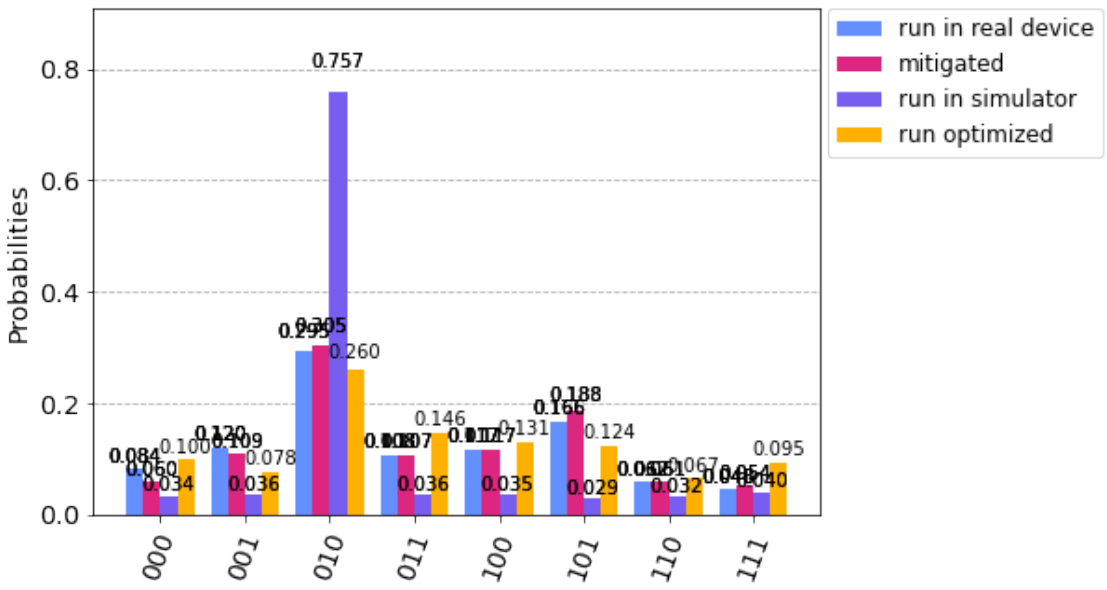
```
[[0.9375     0.05175781 0.03320312 0.0078125  0.08691406 0.00195312
  0.00585938 0.        ]
 [0.01855469 0.90136719 0.         0.02734375 0.00195312 0.09082031
  0.         0.00195312]
 [0.00878906 0.00097656 0.91894531 0.05371094 0.00195312 0.
  0.13183594 0.00976562]
 [0.         0.01269531 0.01464844 0.88671875 0.         0.
  0.00390625 0.14257812]
 [0.03515625 0.00292969 0.00390625 0.         0.88574219 0.04394531
  0.03125    0.00292969]
 [0.         0.03027344 0.         0.00488281 0.01269531 0.84667969
  0.         0.02929688]
 [0.         0.         0.02832031 0.00195312 0.01074219 0.00097656
  0.80957031 0.04199219]
 [0.         0.         0.00097656 0.01757812 0.         0.015625
  0.01757812 0.77148438]]
```



Podemos aplicar um filtro com base na matriz de calibração para obter contagens mitigadas.

[119]:
```
# Get the filter object
meas_filter = meas_fitter.filter

# Results with mitigation
mitigated_results_1 = meas_filter.apply(result)
mitigated_counts_1 = mitigated_results_1.get_counts()

plot_histogram([counts_real_1, mitigated_counts_1, counts_sim_1, counts_opt_1],␣
↪legend=['run in real device', 'mitigated', 'run in simulator', 'run␣
↪optimized'])
```

[119]:



1.4.1 Bibliografia

M. A. Nielsen and I. L. Chuang, Quantum Computation and Quantum Information: 10th Anniversary Edition. Cambridge University Press, New York, NY, USA, 2011