TP2

June 20, 2020

1 TP2 : Interação e Concorrência

- João Bastos (A47419)
- André Sá (A76361)

Cada grupo de estudantes tem um número atribuído e pretendemos utilizar um algoritmo quântico para encontrar o nosso número. O nosso grupo de trabalho é o 1 e usaremos o algoritmo de Grover para o encontrar numa lista de 8 possíveis. Assim, para representação dos números possíveis (0 a 7) teremos 3 qubits ($2^3 = 8$) e pretendemos encontrar a representação binária de 1 (001).

[87]:
```
from qiskit import *
%matplotlib inline
from qiskit.tools.visualization import *
import math as m
import qiskit.tools.jupyter
from qiskit.tools.monitor import backend_overview, backend_monitor
from qiskit.compiler import transpile
from qiskit.providers.aer.noise import NoiseModel
from qiskit.visualization import plot_circuit_layout
from qiskit.ignis.mitigation.measurement import (complete_meas_cal,␣
↪tensored_meas_cal,
                                                 CompleteMeasFitter,␣
↪TensoredMeasFitter)
```

[88]:
```
backend_state = Aer.get_backend("statevector_simulator")
backend_unitary = Aer.get_backend('unitary_simulator')
backend = Aer.get_backend("qasm_simulator")
```

[89]:
```
def intersperse(iterable, delimiter):
    it = iter(iterable)
    yield next(it)
    for x in it:
        yield delimiter
        yield x

def concat_circuits(circuits, barrier=None):
    if barrier:
        circuits = intersperse(circuits, barrier)
```

```
    ret = QuantumCircuit()
    for circuit in circuits:
        ret += circuit
    return ret

def run(circuit, backend, **kwargs):
    if type(backend) is str:
        backend = Aer.get_backend(backend)
    return execute(circuit, backend, **kwargs).result()

def bloch_sphere(circuit, result):
    return plot_bloch_multivector(result.get_statevector(circuit))

def state_city(circuit, result):
    return plot_state_city(result.get_statevector(circuit))

def state_hinton(circuit, result):
    return plot_state_hinton(result.get_statevector(circuit))

def state_qsphere(circuit, result):
    return plot_state_qsphere(result.get_statevector(circuit))

def histogram(circuit, result):
    return plot_histogram(result.get_counts(circuit))
```

2 Algoritmo de Grover

2.1 Inicialização

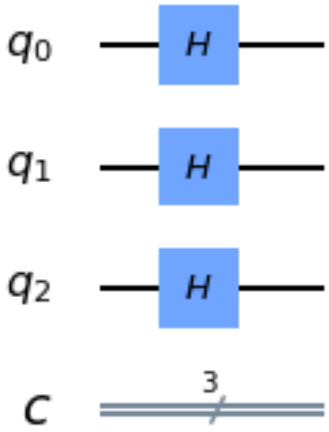
Pretendemos que todos os estados sejam verificados portanto criamos uma sobreposição uniforme de todas as possibilidades aplicando um Hadamard a cada qubit. Sendo $N = 2^3 = 8$,

$$\frac{1}{\sqrt{N}} \sum_{x_i} |x_i\rangle$$

[90]:
```
n = 3 # nº de bits
N = 2 ** n

cr = ClassicalRegister(n, 'c')
qr = QuantumRegister(n, 'q')

barrier = QuantumCircuit(qr, cr)
barrier.barrier()

init = QuantumCircuit(qr, cr)
```

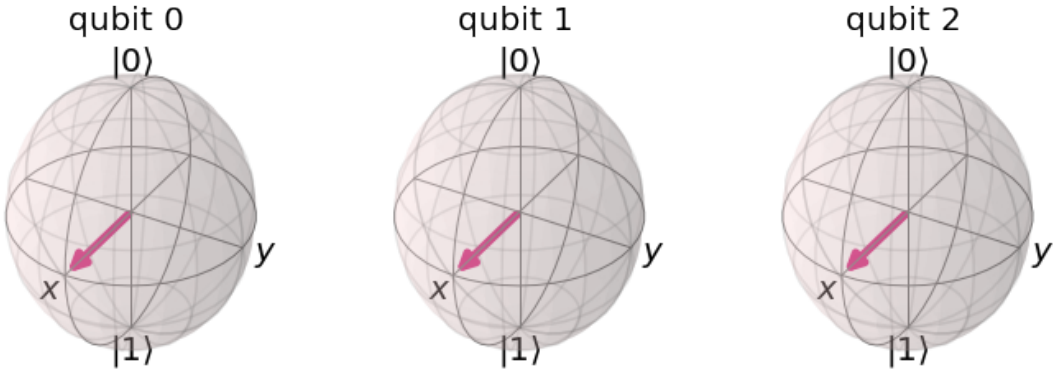
```
init.h(qr)
init.draw('mpl')
```

[90]:



[5]:
```
result = run(init, backend_state)
bloch_sphere(init, result)
```

[5]:



Como podemos verificar, cada qubit está no estado de sobreposição $|+\rangle$ tendo igual probabilidade de ocorrer $|0\rangle$ como $|1\rangle$. Assim, todos os resultados têm igual probabilidade de ocorrer como podemos verificar na matriz seguinte.

[6]:
```
result.get_statevector().real
```

[6]:
```
array([0.35355339, 0.35355339, 0.35355339, 0.35355339, 0.35355339,
       0.35355339, 0.35355339, 0.35355339])
```

2.2 Oráculo

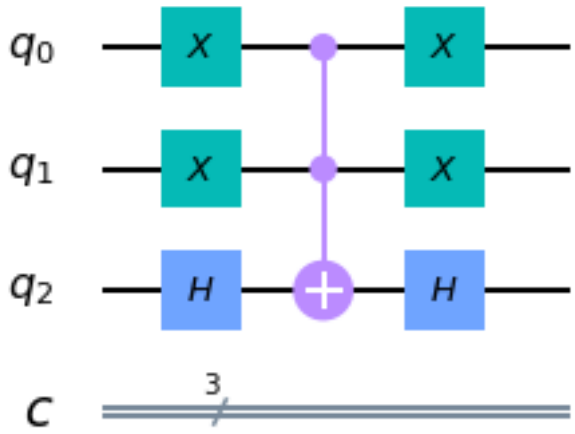
Pretendemos assinalar o nosso número de grupo mudando a fase da componente $|001\rangle$. Para tal optámos por um oráculo de fase que evita usar um qubit auxiliar diminuindo os erros na execução.
O oráculo de fase consiste em selecionar a componente pretendida ($|001\rangle$), aplicar um CCZ e reverter a seleção aplicada anteriormente.
Uma vez que o operador CCZ não existe em qiskit decompusemo-lo em $(I \otimes I \otimes H) \cdot CNOT \cdot (I \otimes I \otimes H)$.

$$-\alpha_{001}|001\rangle + \beta \sum_{x_i \neq 001} |x_i\rangle$$

[7]:
```
def ccZ(circuit, c1, c2, t):
    circuit.h(t)
    circuit.ccx(c1, c2, t)
    circuit.h(t)

def phase_oracle(circuit, qr):
    circuit.x(qr[0])
    circuit.x(qr[1])
    ccZ(circuit, qr[0], qr[1], qr[2])
    circuit.x(qr[0])
    circuit.x(qr[1])
```

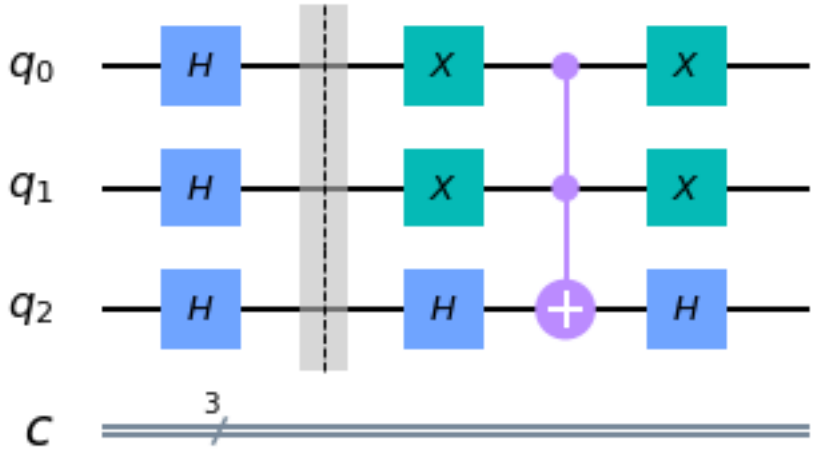
[8]:
```
oracle = QuantumCircuit(qr, cr)
phase_oracle(oracle, qr)
oracle.draw('mpl')
```

[8]:



[9]:
```
circuit = concat_circuits([ init, oracle ], barrier=barrier)
circuit.draw('mpl')
```

[9]:

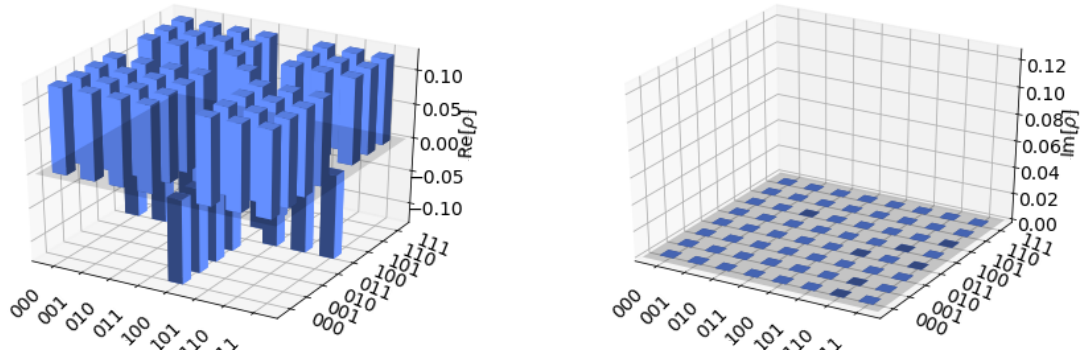


[10]:
```
result = run(circuit, backend_state)
result.get_statevector(circuit).real
```

[10]:
```
array([ 0.35355339,  0.35355339,  0.35355339,  0.35355339, -0.35355339,
        0.35355339,  0.35355339,  0.35355339])
```

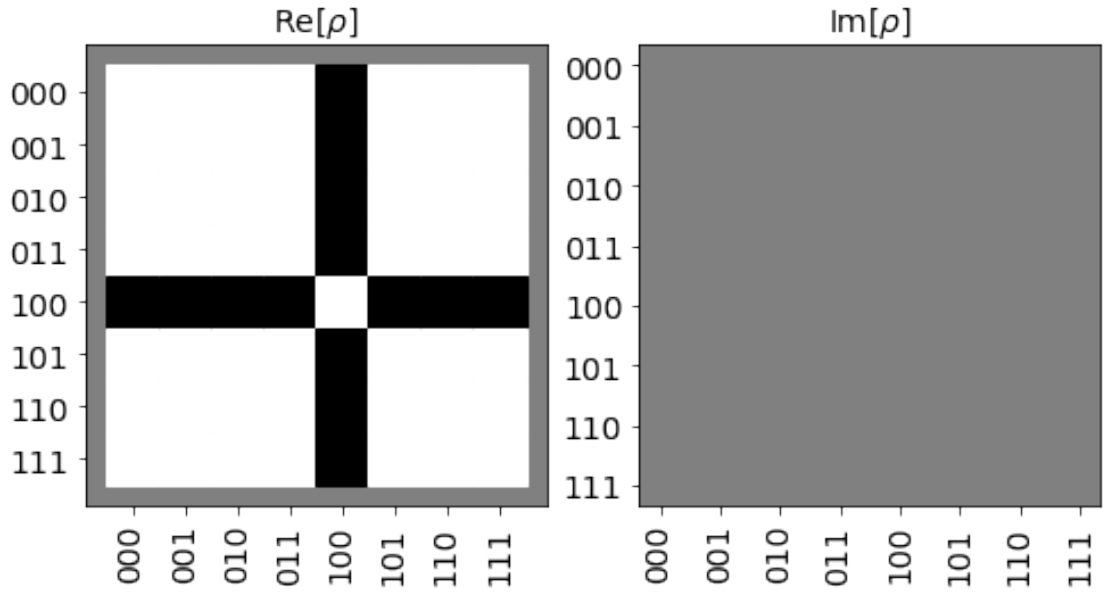
[11]:
```
state_city(circuit, result)
```

[11]:



[12]:
```
state_hinton(circuit, result)
```

[12]:



Como se pode verificar a componente $|100\rangle$ (que representa o $|001\rangle$ pretendido) manteve a amplitude mas alterou a fase, tal como pretendíamos.

2.3 Amplificação

Apesar da mudança de fase, neste momento mantemos as probabilidades iguais para todos casos, o que não nos ajuda muito para o que pretendemos. O objetivo desta secção é aumentar a probabilidade de obtermos o resultado pretendido. Para isso, invertemos e aumentamos a amplitude da componente pretendida em torno da

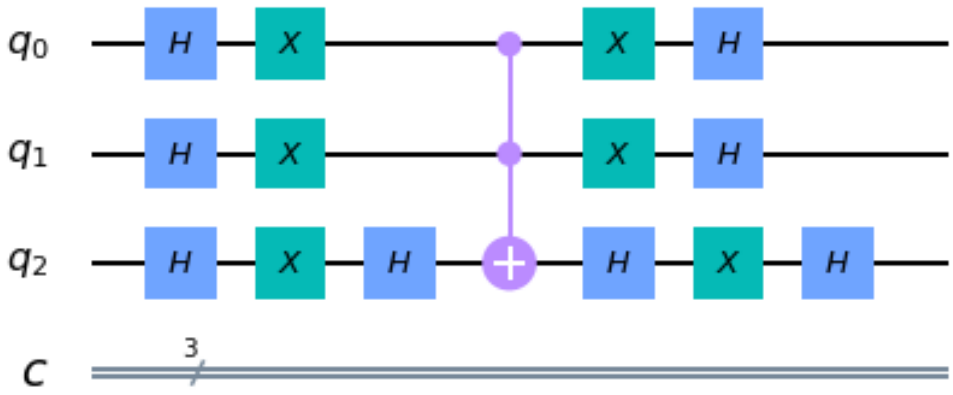
média das amplitudes (A) e simultaneamente reduzimos a amplitude das restantes componentes de forma uniforme.

$$(2A + \alpha_{001})|001\rangle + (2A - \beta) \sum_{x_i \neq 001} |x_i\rangle$$

[13]:
```
def diffuser(circuit, qr):
    circuit.h(qr)
    circuit.x(qr)
    ccZ(circuit, qr[0], qr[1], qr[2])
    circuit.x(qr)
    circuit.h(qr)
```

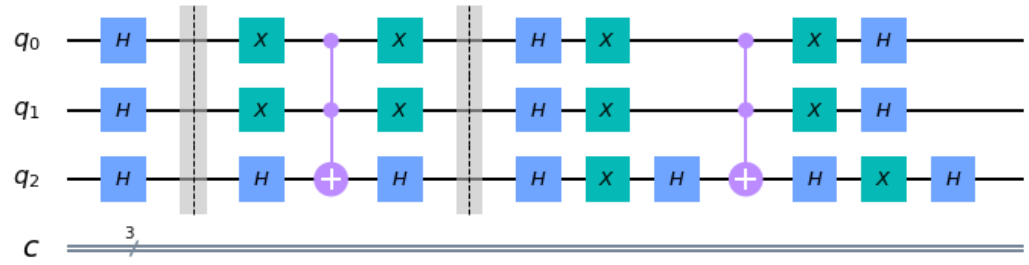
[14]:
```
amplifier = QuantumCircuit(qr, cr)
diffuser(amplifier, qr)

amplifier.draw('mpl')
```

[14]:



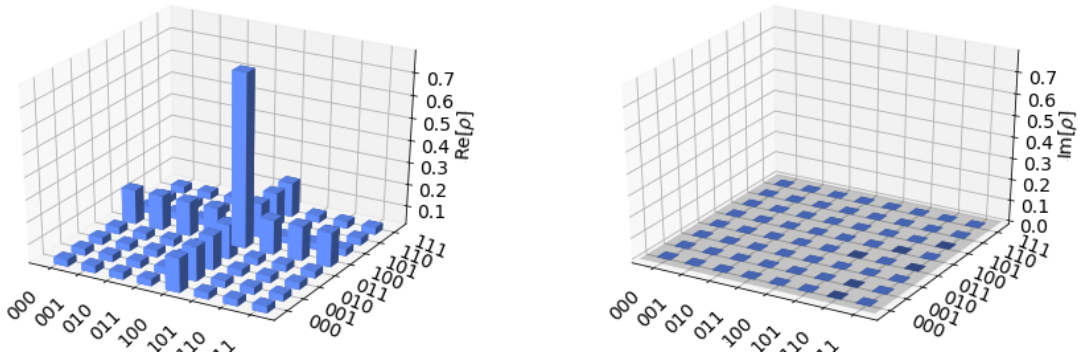
[15]:
```
circuit = concat_circuits([ init, oracle, amplifier ], barrier=barrier)
circuit.draw('mpl')
```

[15]:



[16]:
```
result = run(circuit, backend_state)
state_city(circuit, result)
```

[16]:



Verificamos que a magnitude da componente $|001\rangle$ foi aumentada.

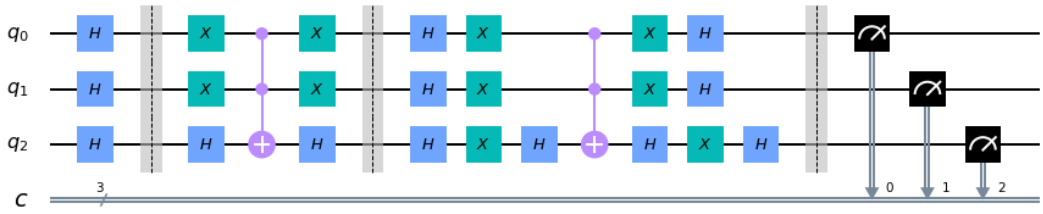
2.4 Medição

A medição é a projeção do estado quântico de cada um dos qubits para um bit clássico. Ao ser medido, cada qubit colapsa para um dos estados possíveis, não sendo mais possível obter o estado quântico inicial.

Apresentamos de seguida o circuito com as 4 secções referidas separadas por barreiras:

[17]:
```
measurements = QuantumCircuit(qr, cr)
measurements.measure(qr, cr)

circuit = concat_circuits([ init, oracle, amplifier, measurements ],␣
 ↪barrier=barrier)
circuit.draw('mpl')
```

[17]:



2.4.1 O iterador de Grover

Antes de fazermos as medições podemos melhorar a probabilidade de medirmos o valor pretendido. Para isso, iremos repetir o iterador de Grover que consiste em aplicar o oráculo e a amplificação. Lov Grover definiu que o número ideal de iterações é aproximadamente $\frac{\pi}{4}\sqrt{N}$, algo que podemos verificar de seguida.

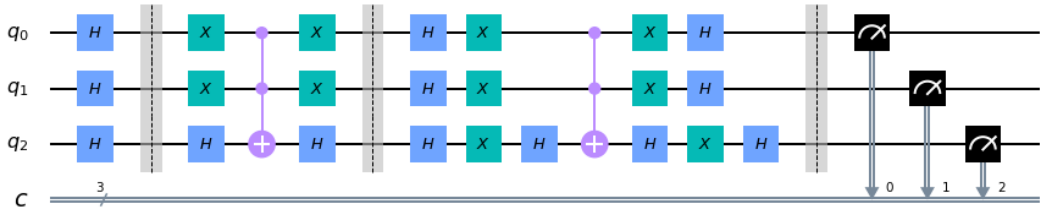
[18]:
```
times = round((m.pi/4) * m.sqrt(N))
times
```

[18]: 2

2.4.2 Uma iteração

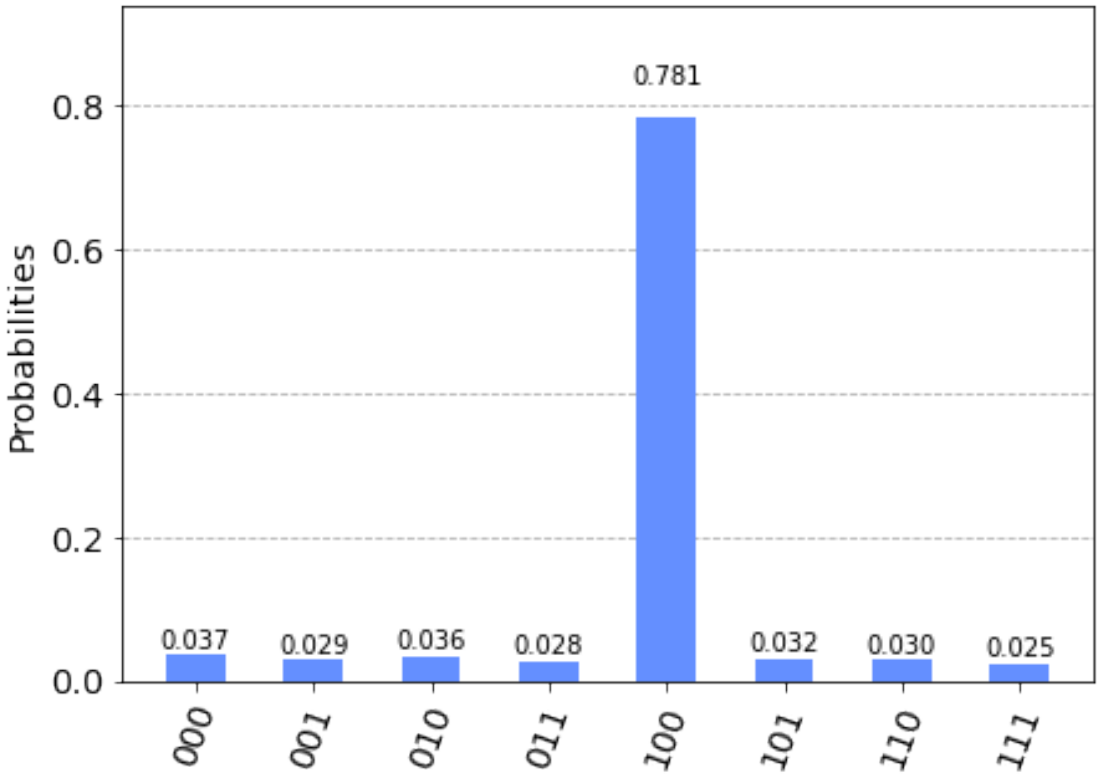
[19]:
```
circuit_1it = concat_circuits([ init ] + ([ oracle, amplifier ] * 1) + [␣
↪measurements ])
circuit.draw('mpl')
```

[19]:



[20]:
```
result = run(circuit_1it, backend)
fst_loop_counts = result.get_counts(circuit_1it)
plot_histogram(fst_loop_counts)
```

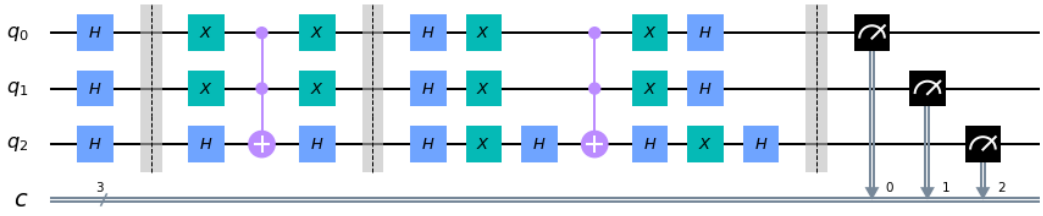
[20]:



2.4.3 Duas iterações

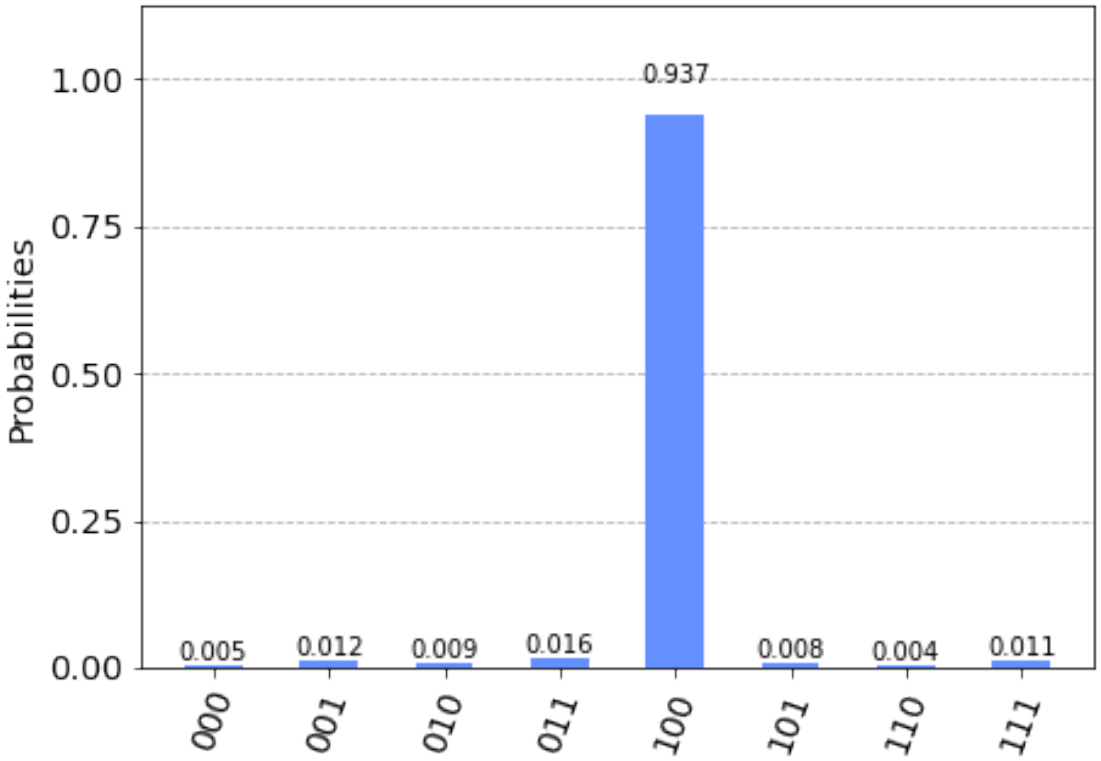
[21]:
```
circuit_2it = concat_circuits([ init ] + ([ oracle, amplifier ] * 2) + [␣
 ↪measurements ])
circuit.draw('mpl')
```

[21]:



[22]:
```
result = run(circuit_2it, backend)
snd_loop_counts = result.get_counts(circuit_2it)
plot_histogram(snd_loop_counts)
```

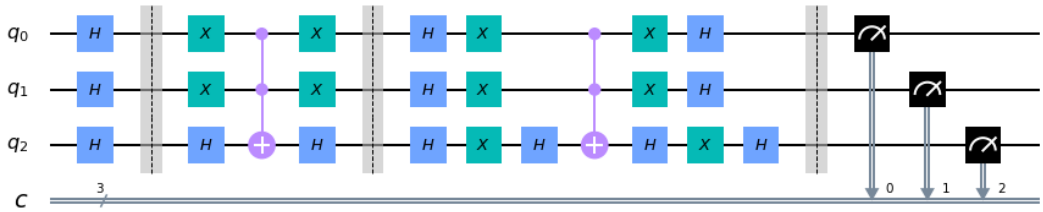
[22]:



2.4.4 Três iterações

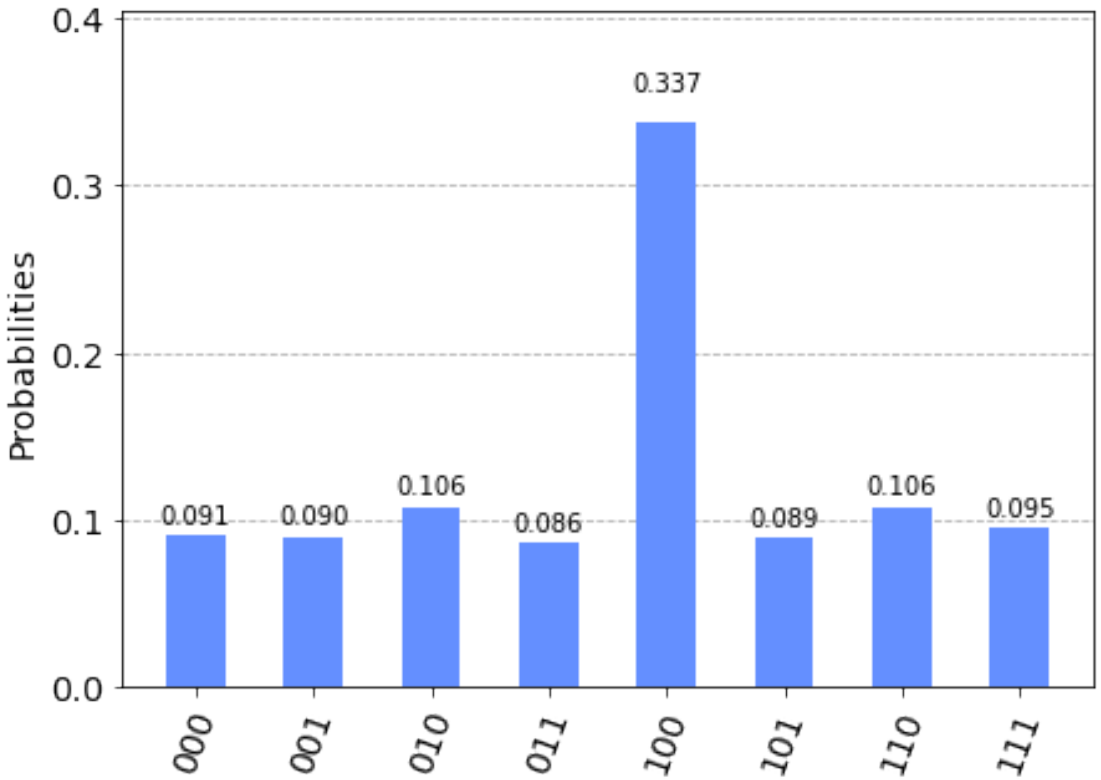
[23]:
```
circuit_3it = concat_circuits([ init ] + ([ oracle, amplifier ] * 3) + [␣
 ↪measurements ])
circuit.draw('mpl')
```

[23]:



[24]:
```
result = run(circuit_3it, backend)
trd_loop_counts = result.get_counts(circuit_3it)
plot_histogram(trd_loop_counts)
```

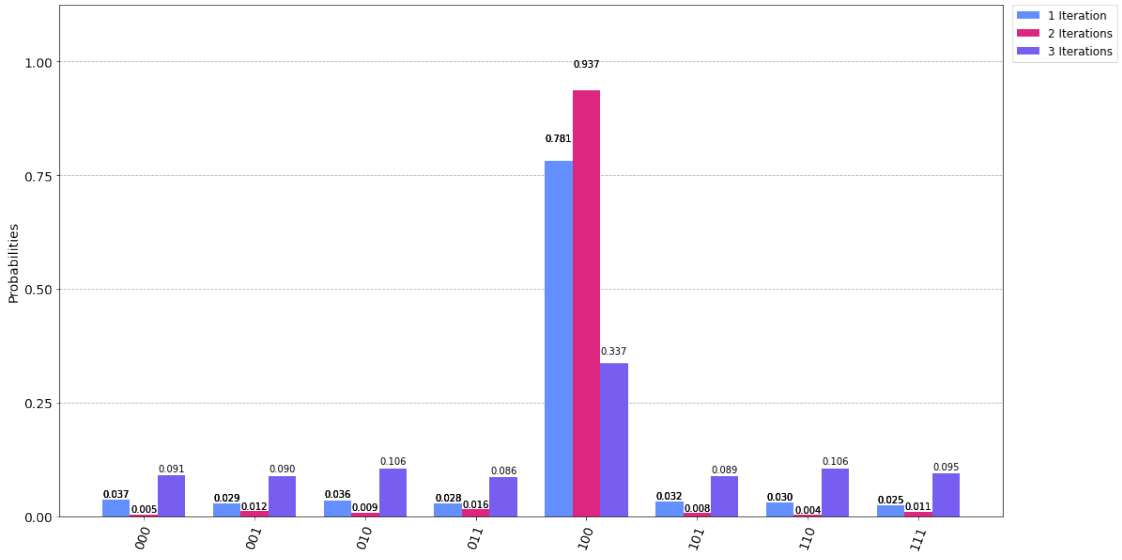
[24]:



2.4.5 Análise de resultados

[25]:
```
plot_histogram([ fst_loop_counts, snd_loop_counts, trd_loop_counts ], legend=[␣
↪"1 Iteration", "2 Iterations", "3 Iterations" ], figsize=(18, 10))
```

[25]:



Como se pode verificar, houve um aumento significativo entre aplicar uma e duas iterações, no entanto ao aplicar uma terceira há uma diminuição substancial.

3 Simulação de ruído

[27]:
```
provider = IBMQ.load_account()
provider.backends(simulator=False, open_pulse=False)
```

[27]:
```
[<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_rome') from IBMQ(hub='ibm-q', group='open',
project='main')>]
```

[105]:
```
%qiskit_backend_overview
```

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

Tendo em conta o valor elevado do T1 e T2 - que se refere à média do tempo de vida dos qubits até chegar à decoerência - à reduzida taxa de erros de medida e cx, e ainda ao facto de ter uma boa disponibilidade optámos pelo device backend ibmq_vigo.

[91]:
```
my_provider_ibmq = IBMQ.get_provider(hub='ibm-q', group='open', project='main')
backend_device = my_provider_ibmq.get_backend('ibmq_vigo')
backend_device
```

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

[91]:
```
<IBMQBackend('ibmq_vigo') from IBMQ(hub='ibm-q', group='open', project='main')>
```

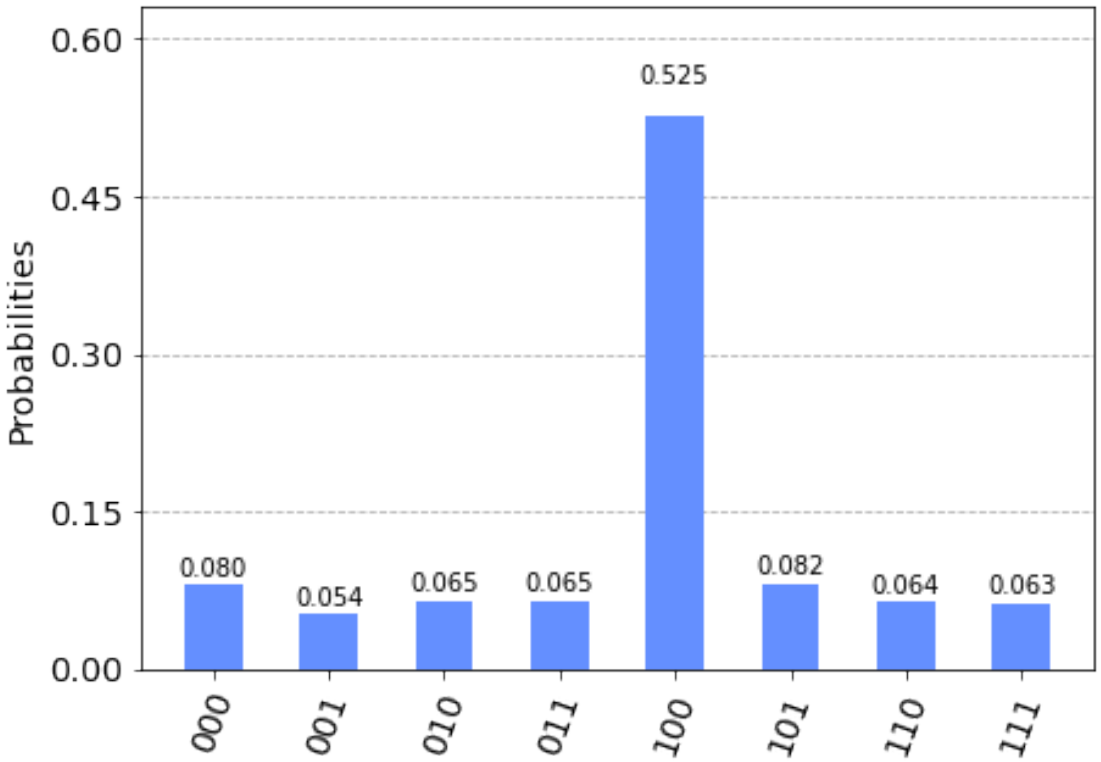
[34]:
```
coupling_map = backend_device.configuration().coupling_map
noise_model = NoiseModel.from_backend(backend_device)
```

```
basis_gates = noise_model.basis_gates

print(basis_gates)
```

```
['cx', 'id', 'u2', 'u3']
```

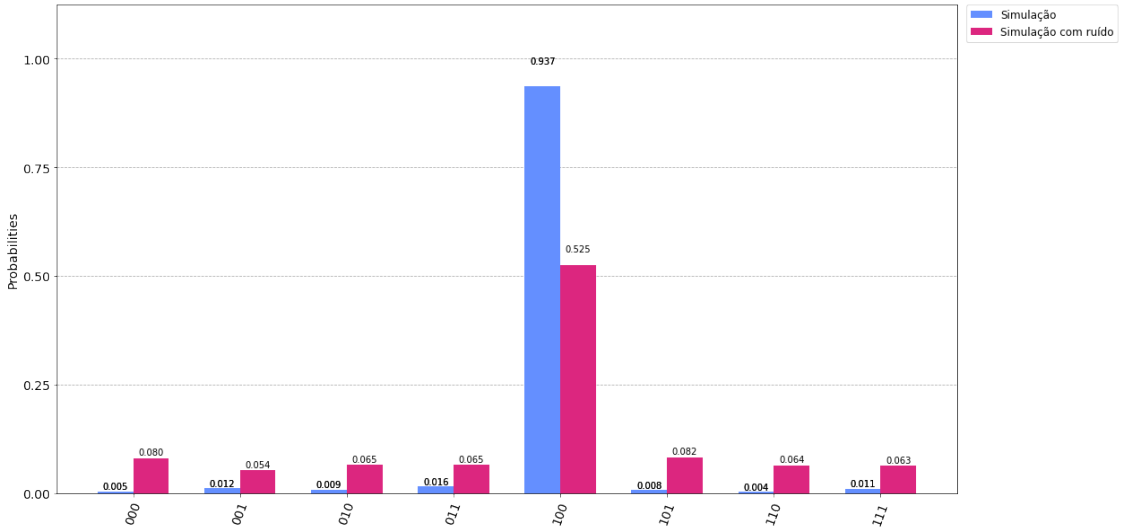
[37]:
```
result_noise = run(circuit_2it,
                        backend,
                        noise_model=noise_model,
                        coupling_map=coupling_map,
                        basis_gates=basis_gates)

counts_noise = result_noise.get_counts(circuit_2it)
plot_histogram(counts_noise)
```

[37]:



[38]:
```
plot_histogram([snd_loop_counts, counts_noise], legend=[ "Simulação",␣
↪"Simulação com ruído" ], figsize=(18, 10))
```

[38]:



Podemos ver que os erros produzidos pelo ruído no backend device serão significativos, pelo que devemos optimizar o circuito de forma a reduzir a taxa de erro.

4 Optimização do circuito

Após simulação com ruído do backend device selecionado vamos executar no mesmo para podermos fazer uma análise mais correcta do ganho efectivo na optimização.

[41]:
```
shots=1024

#executar o circuito na máquina real
#job_G_r = execute(circuit_2it, backend_device, shots=shots)

#jobID_G_r = job_G_r.job_id()

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

```
JOB ID: 5eee298334af18001bf1e7b9
```

[94]:
```
%qiskit_job_watcher ## widget para visualizar o estado da job
#%qiskit_disable_job_watcher  ## para fechar o widget
```

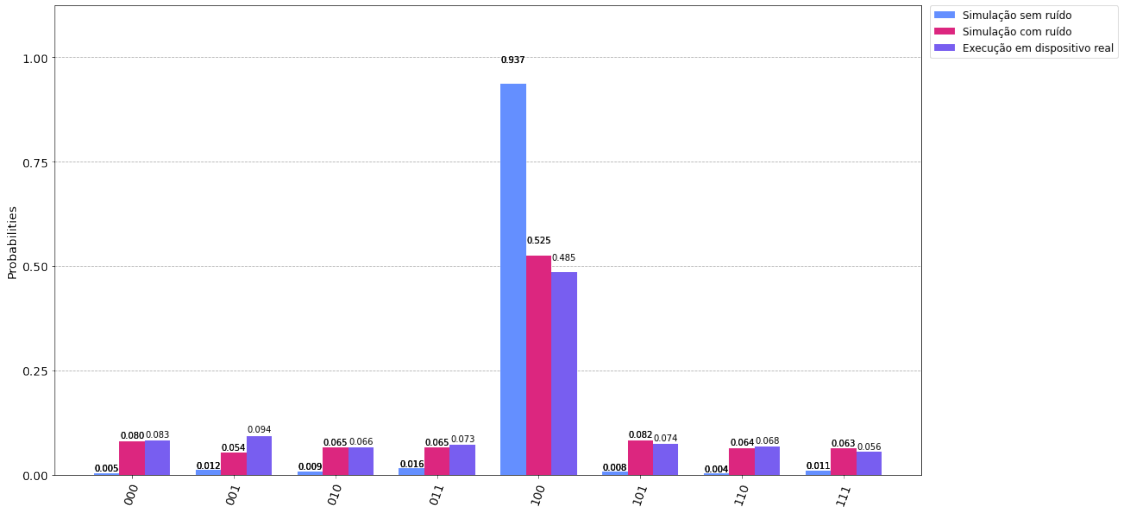
```
Accordion(children=(VBox(layout=Layout(max_width='710px', min_width='710px')),), layout=Layout
```

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

[44]:
```
# Recolha de resultados do job executado no backend device
job_get=backend_device.retrieve_job("5eee298334af18001bf1e7b9")

result_G = job_get.result()
counts_G = result_G.get_counts(circuit_2it)
```

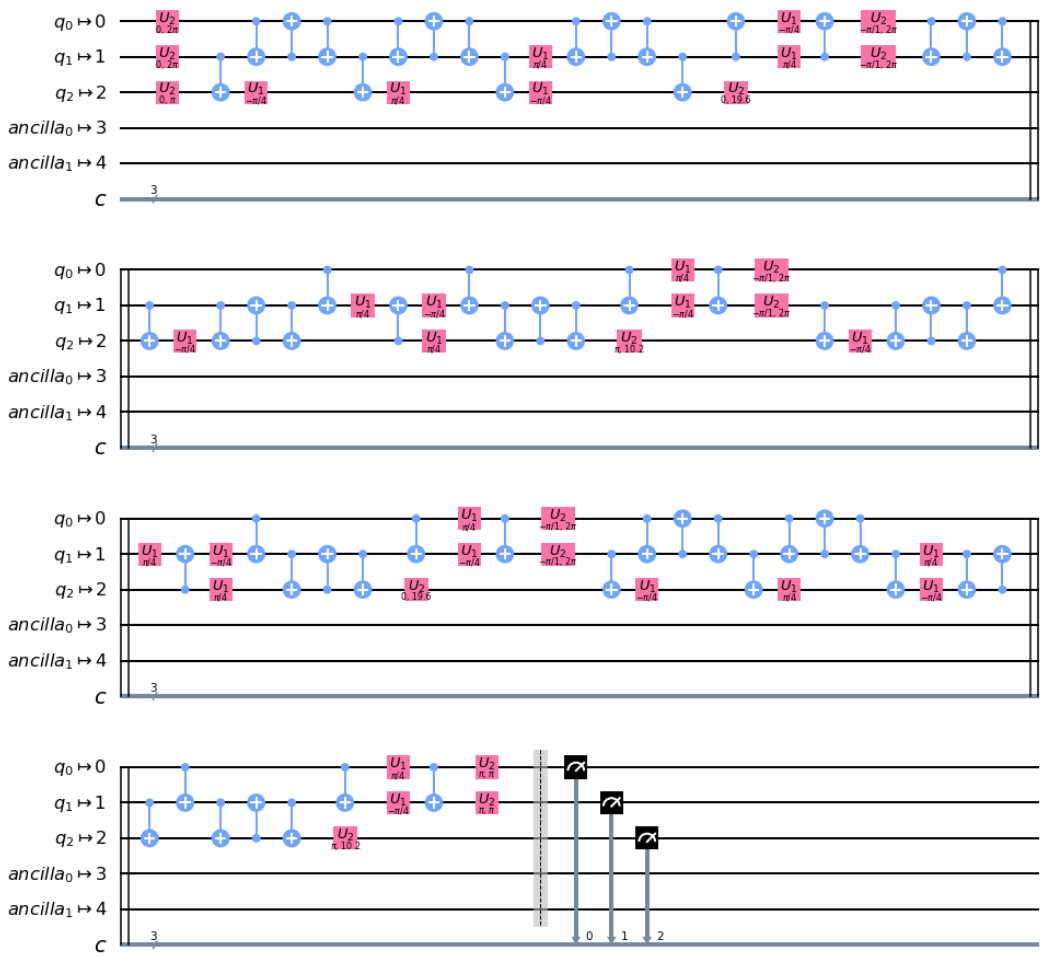
[45]:
```
plot_histogram([snd_loop_counts, counts_noise, counts_G], legend=[ "Simulação␣
↪sem ruído", "Simulação com ruído", "Execução em dispositivo real"],␣
↪figsize=(18, 10))
```

[45]:



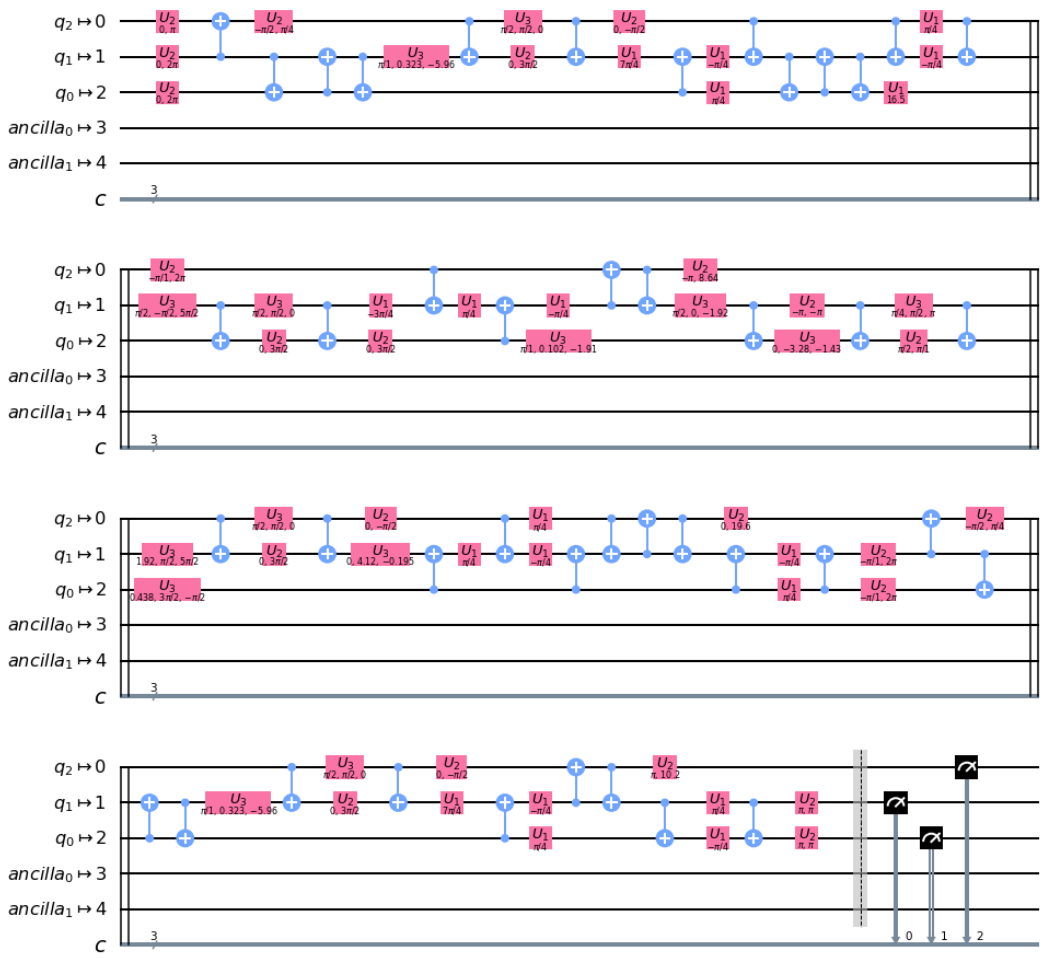
[70]:
```
circuit_real = transpile(circuit_2it, backend=backend_device)
circuit_real.draw(output='mpl', scale=0.5)
```

[70]:



[71]:
```
circuit_opt = transpile(circuit_2it, backend=backend_device,␣
↪optimization_level=3)
circuit_opt.draw(output='mpl', scale=0.5)
```

[71]:

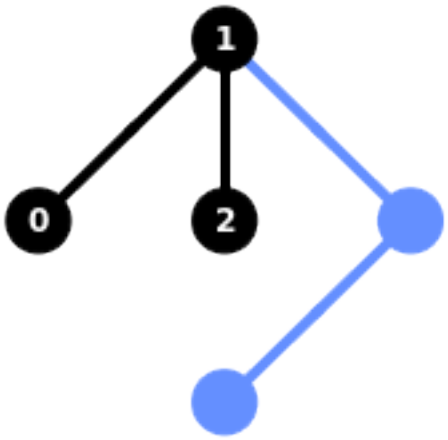


[72]:
```
circuit_real.depth()
```

[72]: 78

[50]:
```
plot_circuit_layout(circuit_real, backend_device)
```

[50]:

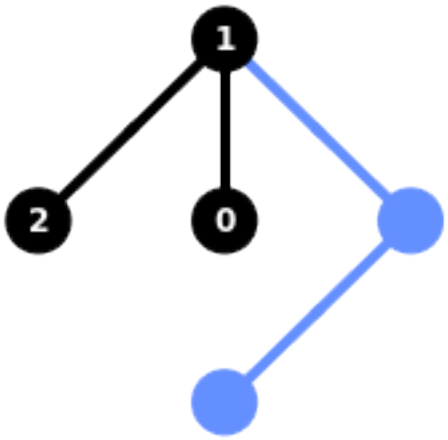


[73]:
```
circuit_opt.depth()
```

[73]: 71

[54]:
```
plot_circuit_layout(circuit_opt, backend_device)
```

[54]:



Após ser aplicada uma optimização de nível 3 à tradução do circuito para a backend device notamos uma ligeira melhoria quanto à profundidade do circuito.

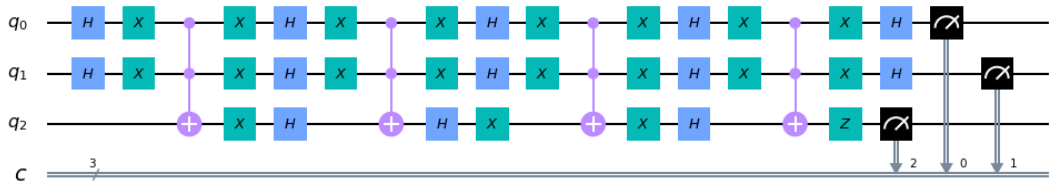
4.1 Optimização do número de gates

Na tentativa de diminuição do número de erros testamos a diminuição do número de gates no circuito fazendo a substituição de algumas combinações de gates por outras.

[74]:
```
cr = ClassicalRegister(n, 'c')
qr = QuantumRegister(n, 'q')

qc_short = QuantumCircuit(qr, cr)

qc_short.h([ qr[0], qr[1] ])
qc_short.x([ qr[0], qr[1] ])

############

qc_short.ccx(qr[0], qr[1], qr[2])
qc_short.x([ qr[0], qr[1] ])
qc_short.h([ qr[0], qr[1] ])
qc_short.x([ qr[0], qr[1] ])
```

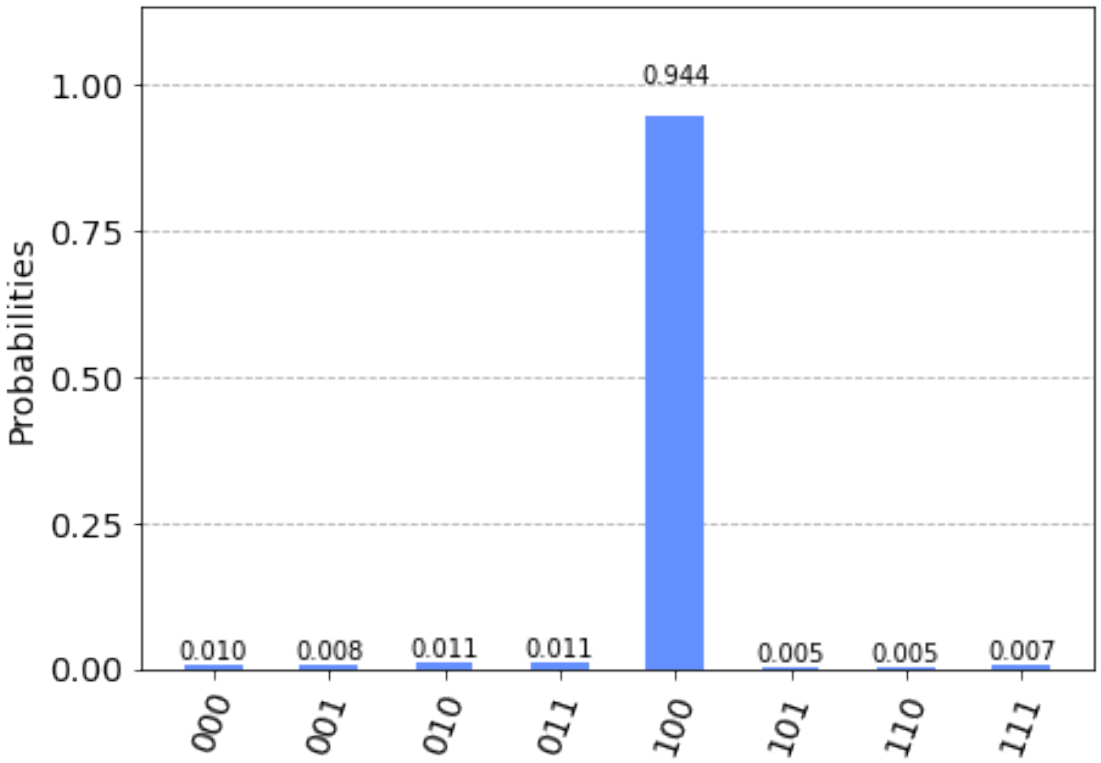
```
if False:
    qc_short.h(qr[2])
    qc_short.z(qr[2])
else:
    qc_short.x(qr[2])
    qc_short.h(qr[2])

############

qc_short.ccx(qr[0], qr[1], qr[2])
qc_short.x([ qr[0], qr[1] ])
qc_short.h([ qr[0], qr[1] ])
qc_short.x([ qr[0], qr[1] ])

if False:
    qc_short.z(qr[2])
    qc_short.h(qr[2])
else:
    qc_short.h(qr[2])
    qc_short.x(qr[2])

############

qc_short.ccx(qr[0], qr[1], qr[2])
qc_short.x([ qr[0], qr[1] ])
qc_short.h([ qr[0], qr[1] ])
qc_short.x([ qr[0], qr[1] ])

if False:
    qc_short.h(qr[2])
    qc_short.z(qr[2])
else:
    qc_short.x(qr[2])
    qc_short.h(qr[2])

############

qc_short.ccx(qr[0], qr[1], qr[2])
qc_short.x([ qr[0], qr[1] ])
qc_short.h([ qr[0], qr[1] ])
qc_short.z(qr[2])

qc_short.measure(qr, cr)

qc_short.draw('mpl')
```

[74]:



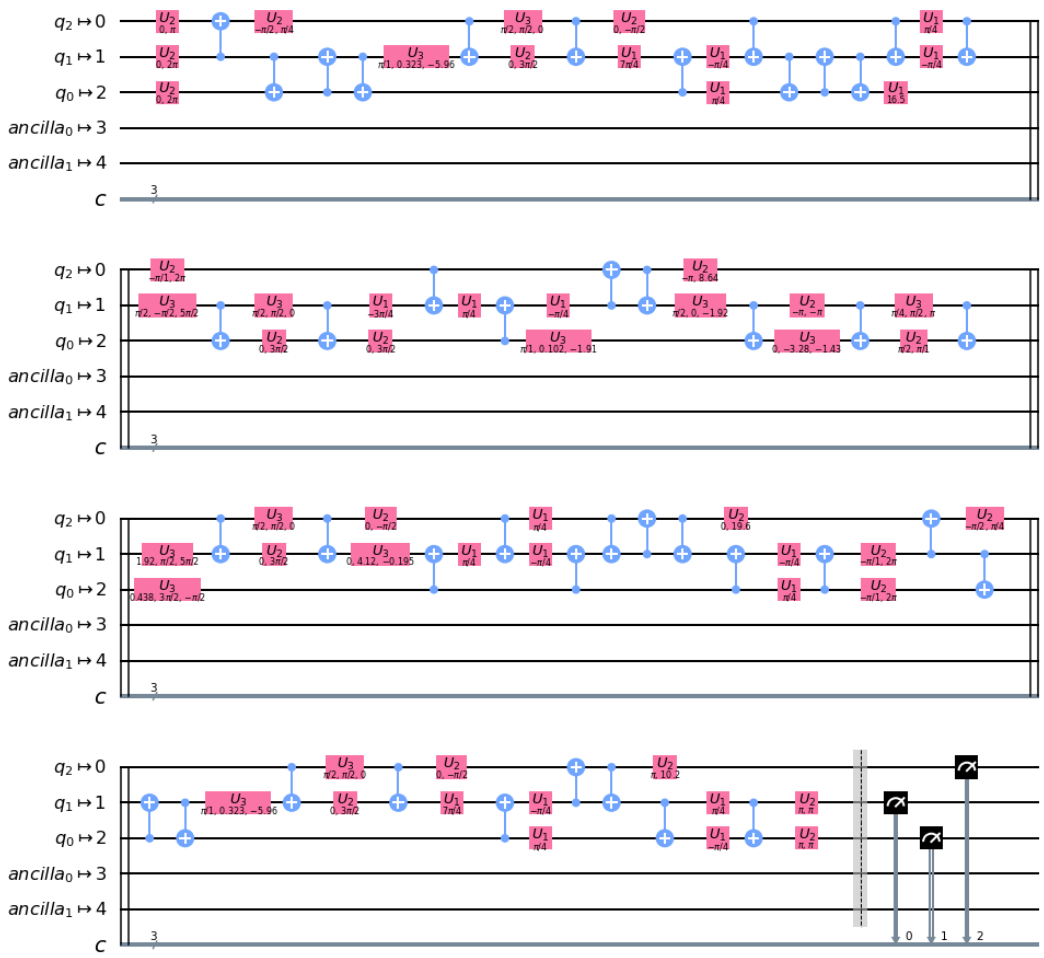
[75]:
```
result = run(qc_short, backend)
histogram(qc_short, result)
```

[75]:



[77]:
```
qc_short_real = transpile(qc_short, backend=backend_device)
qc_short_opt = transpile(qc_short, backend=backend_device, optimization_level=3)
circuit_opt.draw(output='mpl', scale=0.5)
```

[77]:



[78]:
```
qc_short_real.depth()
```

[78]: 76

[79]:
```
qc_short_opt.depth()
```

[79]: 64

[80]:
```
# execução do circuito com optimização nivel 3 #
#job_G_r2 = execute(circuit_opt, backend_device, shots=shots)

#jobID_G_r2 = job_G_r2.job_id()

#print('JOB ID: {}'.format(jobID_G_r2))
```

JOB ID: 5eee32dc34af18001bf1e844

[83]:
```
job_get2=backend_device.retrieve_job("5eee32dc34af18001bf1e844")

result_G2 = job_get2.result()
counts_G2 = result_G2.get_counts(circuit_opt)
```

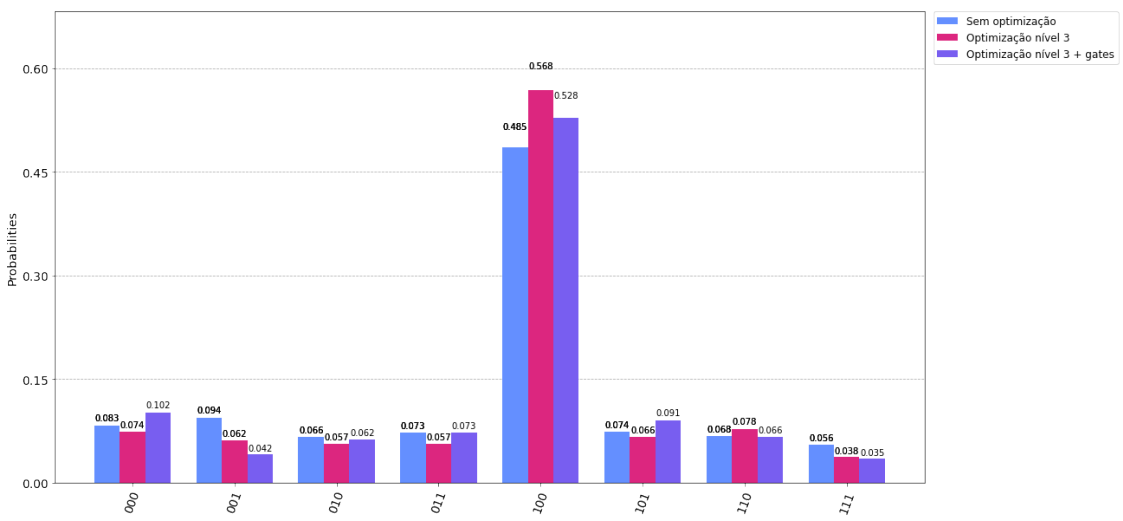
[81]:
```
# execução do circuito com optimização nivel 3 + optimização nº de gates #
#job_G_r3 = execute(qc_short_opt, backend_device, shots=shots)

#jobID_G_r3 = job_G_r3.job_id()

#print('JOB ID: {}'.format(jobID_G_r3))
```

JOB ID: 5eee32e06bb14a001cca1b2e

[84]:
```
job_get3=backend_device.retrieve_job("5eee32e06bb14a001cca1b2e")

result_G3 = job_get3.result()
counts_G3 = result_G3.get_counts(qc_short_opt)
```

[86]:
```
plot_histogram([counts_G, counts_G2, counts_G3], legend=[ "Sem optimização",␣
↪"Optimização nível 3", "Optimização nível 3 + gates"], figsize=(18, 10))
```

[86]:



Apesar da profundidade do circuito com optimização do número de gates ter diminuido, contraintuitivamente teve um pior resultado que o circuito com apenas a optimização nível 3. Assim, teremos de abdicar desta última optimização.

5 Mitigação de erros com o Ignis

Por fim, após termos executado o nosso circuito vamos usar o Ignis para, sobre os resultados obtidos, mitigar os erros de medição do backend device escolhido.

Primeiro geramos a matriz de calibração a partir das propriedades do backend device.

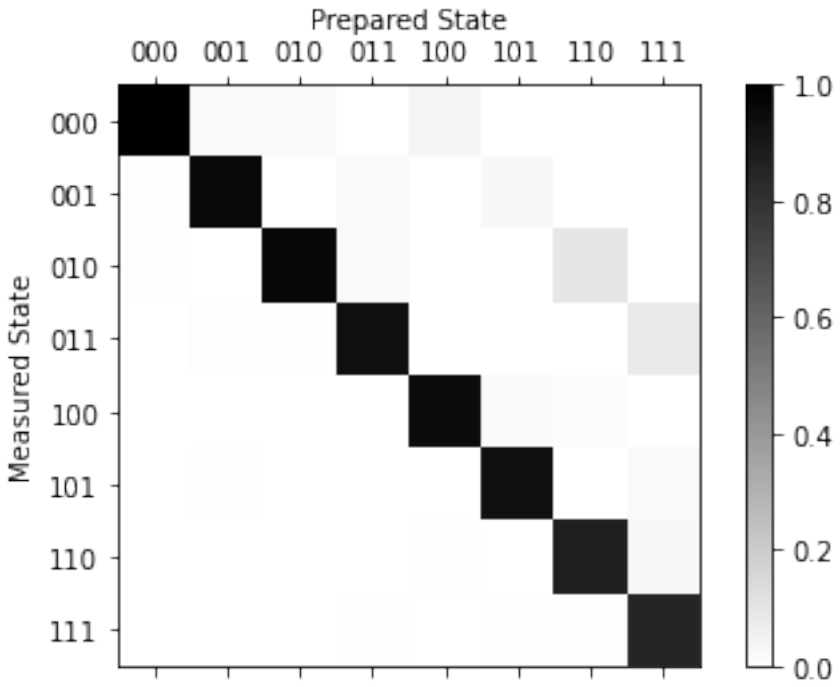
[107]:
```
meas_calibs, state_labels = complete_meas_cal(qubit_list=[0,1,2], qr=qr,␣
↪circlabel='mcal')
```

[93]:
```
#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: 5eee477b61ad6a00190bd14f

[95]:
```
job_get=backend_device.retrieve_job("5eee477b61ad6a00190bd14f")

cal_results = job_get.result()
```

[96]:
```
meas_fitter = CompleteMeasFitter(cal_results, state_labels, circlabel='mcal')

meas_fitter.plot_calibration()
```



[106]:
```
print("Fidelidade média de Medição: %f" % meas_fitter.readout_fidelity())
```

```
Fidelidade média de Medição: 0.934570
```

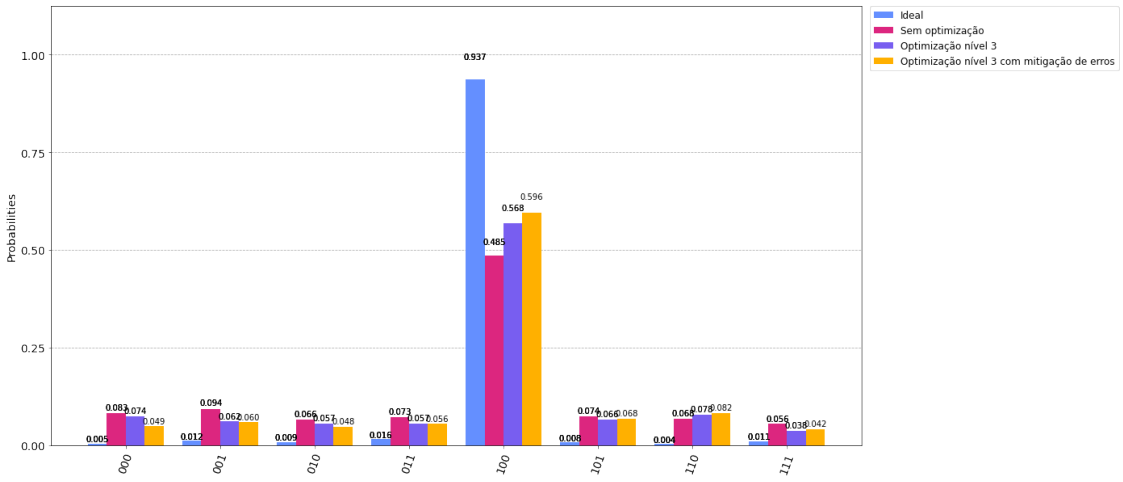
Podemos verificar que a fidelidade média de medição é bastante elevada pelo que a melhoria não deverá ser significativa.

Por fim, aplicamos a calibração aos resultados com um filtro baseado na matriz de calibração.

[100]:
```
meas_filter = meas_fitter.filter

# Resultados com mitigação
mitigated_results = meas_filter.apply(result_G2)
mitigated_counts = mitigated_results.get_counts()
```

[102]:
```
plot_histogram([snd_loop_counts, counts_G, counts_G2, mitigated_counts],␣
↪legend=[ "Ideal", "Sem optimização", "Optimização nível 3", "Optimização␣
↪nível 3 com mitigação de erros"], figsize=(18, 10))
```

[102]:



Verificamos uma melhoria após a mitigação dos erros de medição no entanto os resultados ficam muito aquém do ideal.

Referências:

- *Complete 3-Qubit Grover Search on a Programmable Quantum Computer*
- *Qiskit Documentation*