Grovers_algorithm

June 5, 2021

1 Quantum Pratical Assignment

1.1 Iteração e Concorrência

Realizado por Grupo 19:

```
* José Pedro Ribeiro Alves A78178
* Ricardo Nuno Alves Teixeira A85688
```

1.2 0. Introdução

Este trabalho consiste em implementar um algorítmo em Quiskit de maneira a encontrar um determinado numero dentro de uma lista não ordenada de números. O número a encontrar é determinado por: $14 \ mod \ 8$ 6 que corresponde a $|110\rangle$.

Para resolver o problema que nos foi proposto, optamos por usar o algorítmo de Grover que tem como principal objetivo solucionar uma procura numa determinada base de dados independentemente da sua estrutura com uma otimização quadrática do tempo de execução comparado à computação clássica.

Num caso de computação clássica a complexidade média do problema era dada por $\frac{N}{2}$. Através da computação quantica ao usar o algoritmo de Grover a complexidade média do mesmo problema é $\sqrt{N}$.

1.3 1. Algoritmo de Grover

O algoritmo de Grover está dividido em três etapas:

```
a. Inicialização
b. Oráculo
c. Amplificação
```

1.3.1 a. Inicialização

A **sobreposição quântica** é um dos princípios fundamentais da mecânica e computação quântica. O algoritmo de Grover necessita que esta sobreposição seja uniforme. Para obter a sobreposição uniforme vamos usar o gate de Hadamard.

Ao aplicar o gate de Hadamard a n qubits (Transformada de Hadamard) mapeamos todos n bits inicializados a $|0\rangle$ para uma superposição de todos os estados ortogonais de 2^n na base de $|0\rangle$ e $|1\rangle$, onde todos esses estados ortogonais têm a mesma amplitude (a probabilidade de esse estado colapsar para o respetivo estado clássico é o quadrado da amplitude).

Geralmente, a sobreposição é criada da seguinte forma:

$$|\psi\rangle = H^{\bigotimes n}|0\rangle^{\bigotimes n} = \frac{1}{\sqrt{2}^n}\sum_{x=0}^{2^n-1}|x\rangle$$

Como estamos a usar 3 qubits, temos:

$$H^3|000\rangle = \frac{1}{2\sqrt{2}}|000\rangle + \frac{1}{2\sqrt{2}}|001\rangle + \frac{1}{2\sqrt{2}}|010\rangle + \frac{1}{2\sqrt{2}}|011\rangle + \frac{1}{2\sqrt{2}}|100\rangle + \frac{1}{2\sqrt{2}}|101\rangle + \frac{1}{2\sqrt{2}}|111\rangle$$

Graficamente:

[2]:
```
##Sendo o qr o registo quântico do circuito

def init(circuit,qr):
    circuit.h(qr)
```

1.3.2 b. Oráculo

A ideia do oráculo é identificar e marcar a resposta pretendida. No nosso caso será o estado $|011\rangle$. Essa mesma marcação consiste em inverter a fase do estado solução. Para tal temos de identificar o estado solução e utilizar o gate Z para inverter a fase desse estado.

Para o fazermos, a ideia é transformar o estado solução de tal maneira o gate CCZ produza o efeito desejado (inversão de fase). Por outras palavras, para ativar o CCZ, temos que transformar o nosso estado solução no estado $|111\rangle$. Após fazer isto, aplicamos o CCZ e revertemos o resultado para retornar ao estado solução, mas com a fase invertida.

Graficamente, a ideia é a seguinte:

NOTA: Visto que o gate CCZ não está definido no qiskit, implementamos a função CCZ através da composição de gates (H após CCX após H)

[3]:
```
#Definicao do CCZ
def ccz(circuit, qr):
    circuit.h(qr[2])
    circuit.ccx(qr[0],qr[1],qr[2])
    circuit.h(qr[2])

def phase_oracle(circuit, qr):
    #muda a fase do estado |011>  -> |qr[2] qr[1] qr[0]>
    circuit.x(qr[2])
    ccz(circuit,qr)
    circuit.x(qr[2])
```

1.3.3 c. Amplificação

Esta etapa consiste aplicar um difusor $U_D = WRW$, onde W é a transformada de Hadamard e R é a matriz rotação. O objetivo deste difusor é voltar a inverter a fase do estado solução e aumentar a sua amplitude, fazendo assim com que quando o estado colapsar, a probabilidade de retornar o estado solução seja consideravelmente maior.

[4]:
```
def diffuser(circuit, qr):
    circuit.h(qr)
    circuit.x(qr)
    ccz(circuit, qr)
    circuit.x(qr)
    circuit.h(qr)
```

Para terminar, e obter os melhores resultados possiveis, o ideal é repetir os passos **b.Oráculo** e **c.Amplificador** $\sqrt{N}$ vezes, sendo N o número de elementos da lista.

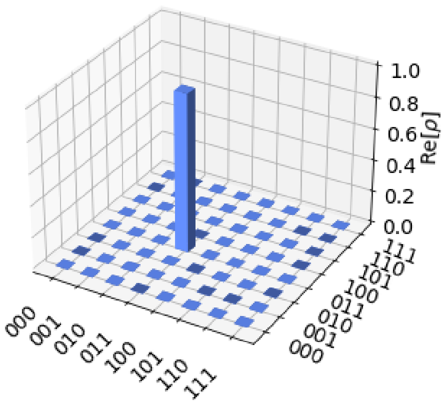
A seguinte função implementa este circuito f vezes.

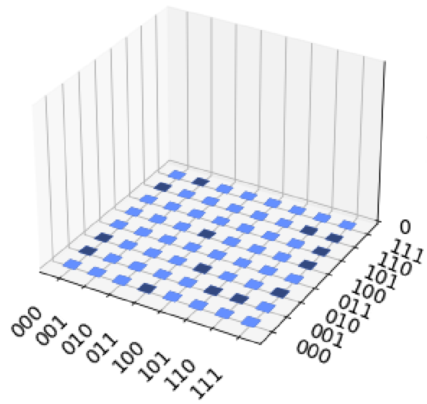
[5]:
```
def buildCircuitGrover(f):
    qr = QuantumRegister(x, 'q')
    cr = ClassicalRegister(x,'c')
    circuit = QuantumCircuit(qr,cr)

    init(circuit,qr)

    for t in range(f):
        # phase oracle
        phase_oracle(circuit,qr)

        # diffuser
        diffuser(circuit,qr)

    circuit.measure(qr,cr)

    return circuit
```

Matemáticamente, numa situação perfeita onde não houvesse ruído, o resultado obtido seria o seguinte:

[6]:
```
qc_Grover = buildCircuitGrover(2)

#STATE VECTOR
backend_state = Aer.get_backend('statevector_simulator')
result = execute(qc_Grover, backend_state).result()
psi  = result.get_statevector(qc_Grover)
plot_state_city(psi)
```

[6]:

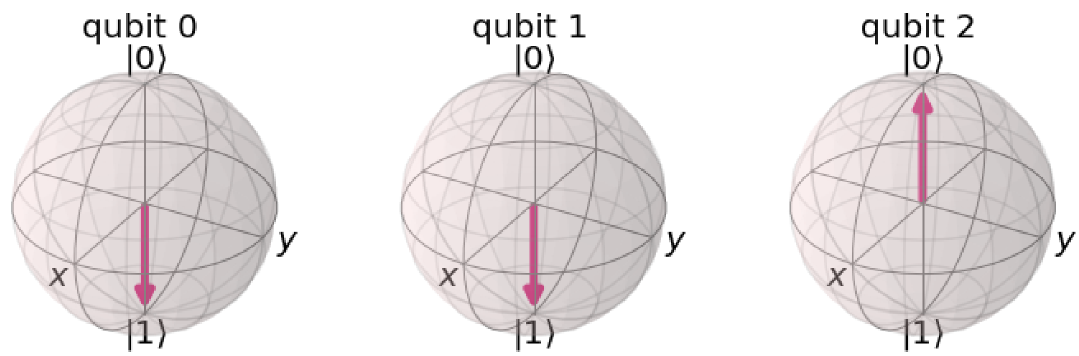




[7]:
```
plot_bloch_multivector(psi)
```

```
/home/ricardo/anaconda3/envs/IC/lib/python3.9/site-
packages/qiskit/visualization/bloch.py:69: MatplotlibDeprecationWarning:
The M attribute was deprecated in Matplotlib 3.4 and will be removed two minor
releases later. Use self.axes.M instead.
  x_s, y_s, _ = proj3d.proj_transform(xs3d, ys3d, zs3d, renderer.M)
```

[7]:



1.4 Simulação do ruído para prever uma melhor otimização

Atualmente as máquinas quânticas ainda têm o chamado ruído quântico que resulta da incerteza da física quântica assim como erros provenientes da limitação de hardware (quanto maior o circuito, maior a probabilide de obter erros). Como este ruído existe, os resultados obtidos não são perfeitos, e por isso temos que fazer uma previsão de maneira a conseguir obter os melhores resultados possíveis.

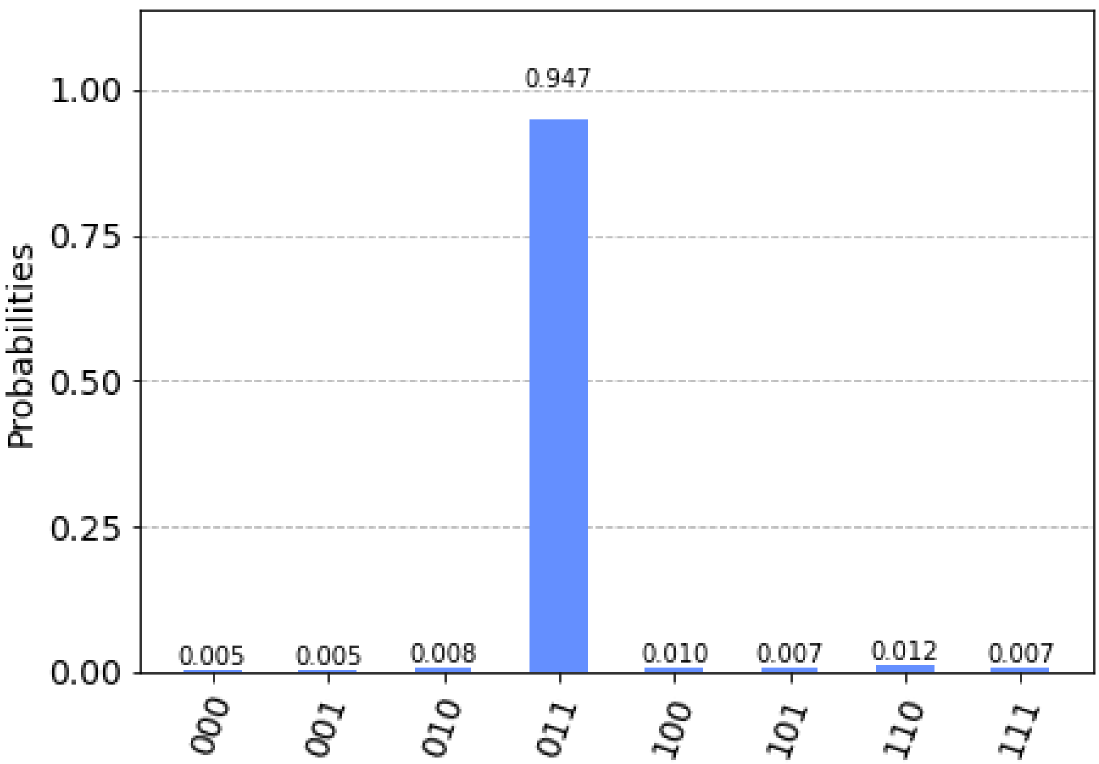
Idealmente, o número de vezes que repetimos os passos **b. Oráculo** e **c. Amplificador** $\sqrt{N}$ é um inteiro. Dado que neste caso $N = 8$, e $\sqrt{8}$ nao é um número inteiro tentamos repetir os passos para os dois números mais proximos (*floor* e *ceiling*), sendo eles 2 e 3.

[8]:
```
import math as m
res = m.sqrt (8)
print (res)
```

2.8284271247461903

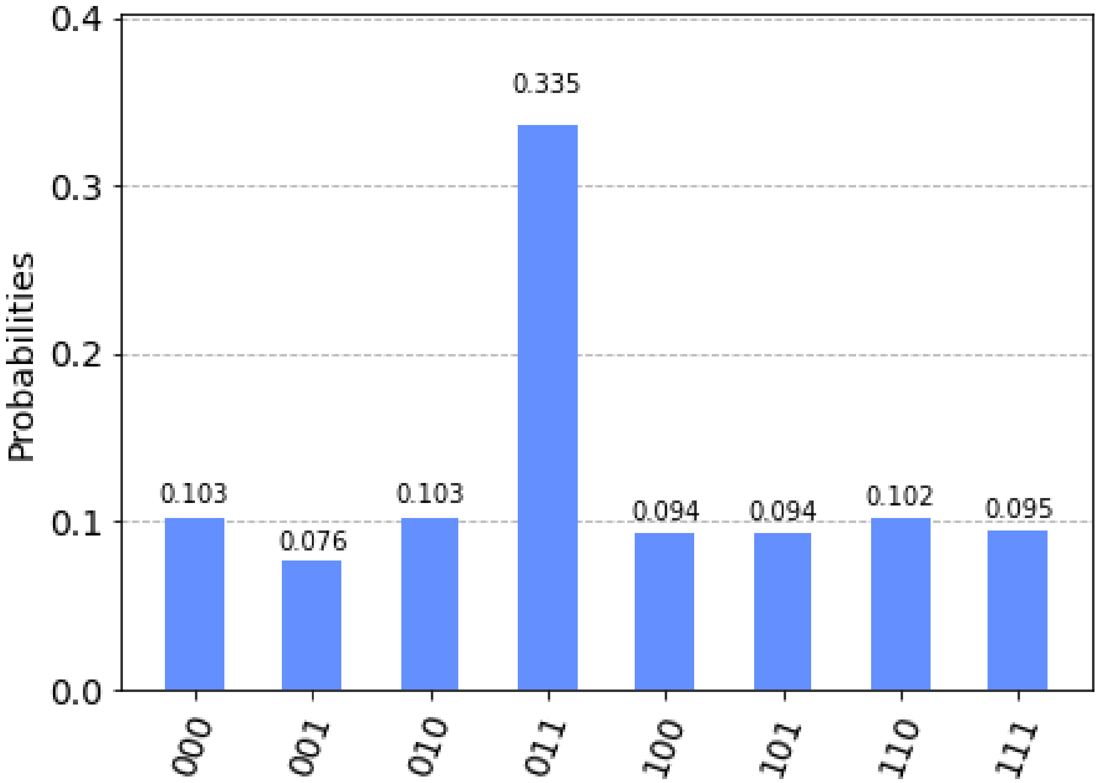
[9]:
```
backend_qasm = Aer.get_backend('qasm_simulator')
shots = 1024

#SIM RUIDO for = 2
qc_Grover = buildCircuitGrover(2)
result2 = execute(qc_Grover, backend_qasm, shots=shots).result()
count = result2.get_counts(qc_Grover)
plot_histogram(count)
```

[9]:



[10]:
```
#SIM RUIDO for = 3
qc_Grover2 = buildCircuitGrover(3)
result3 = execute(qc_Grover2, backend_qasm, shots=shots).result()
count2 = result3.get_counts(qc_Grover2)
plot_histogram(count2)
```

[10]:



Ao observar os resultados, concluímos que há melhorias bastante significativas de um para o outro (de 33% para 94%). Por isso, no nosso caso, daqui para a frente, optamos por repetir os passos **b.** e **c.** apenas duas vezes

1.5 3. Execução numa Máquina Quântica real

[11]:
```
provider = IBMQ.load_account()
provider.backends()
```

[11]:
```
[<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_armonk') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_athens') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_santiago') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_lima') from IBMQ(hub='ibm-q', group='open',
project='main')>,
```

```
 <IBMQBackend('ibmq_belem') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_quito') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQSimulator('simulator_statevector') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQSimulator('simulator_mps') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQSimulator('simulator_extended_stabilizer') from IBMQ(hub='ibm-q',
group='open', project='main')>,
 <IBMQSimulator('simulator_stabilizer') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_manila') from IBMQ(hub='ibm-q', group='open',
project='main')>]
```

[12]:
```
%qiskit_backend_overview
```

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

[13]:
```
backend_overview()
```

```
ibmq_manila                 ibmq_quito                  ibmq_belem
-----------                 ----------                  ----------
Num. Qubits:  5             Num. Qubits:  5             Num. Qubits:  5
Pending Jobs: 2             Pending Jobs: 8             Pending Jobs: 5
Least busy:   True          Least busy:   False         Least busy:   False
Operational:  True          Operational:  True          Operational:  True
Avg. T1:      151.0         Avg. T1:      75.2          Avg. T1:      79.3
Avg. T2:      67.0          Avg. T2:      73.2          Avg. T2:      91.6



ibmq_lima                   ibmq_santiago               ibmq_athens
---------                   -------------               -----------
Num. Qubits:  5             Num. Qubits:  5             Num. Qubits:  5
Pending Jobs: 10            Pending Jobs: 2             Pending Jobs: 5
Least busy:   False         Least busy:   False         Least busy:   False
Operational:  True          Operational:  True          Operational:  True
Avg. T1:      69.2          Avg. T1:      136.2         Avg. T1:      95.9
Avg. T2:      64.9          Avg. T2:      136.4         Avg. T2:      120.6



ibmq_armonk                 ibmq_16_melbourne           ibmqx2
-----------                 -----------------           ------
Num. Qubits:  1             Num. Qubits:  15            Num. Qubits:  5
Pending Jobs: 16            Pending Jobs: 3             Pending Jobs: 2
```

```
Least busy:    False           Least busy:    False           Least busy:    False
Operational:   True            Operational:   True            Operational:   True
Avg. T1:       124.6           Avg. T1:       57.5            Avg. T1:       54.1
Avg. T2:       217.3           Avg. T2:       56.2            Avg. T2:       40.5
```

[14]:
```
backend_device = provider.get_backend('ibmq_manila')
print("Running on: ", backend_device)
```

```
Running on:  ibmq_manila
```

[15]:
```
# See backend information
backend_device
```

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

[15]:
```
<IBMQBackend('ibmq_manila') from IBMQ(hub='ibm-q', group='open',
project='main')>
```

[16]:
```
backend_monitor(backend_device)
```

```
ibmq_manila
===========
Configuration
-------------
    n_qubits: 5
    operational: True
    status_msg: active
    pending_jobs: 2
    backend_version: 1.0.1
    basis_gates: ['id', 'rz', 'sx', 'x', 'cx', 'reset']
    local: False
    simulator: False
    max_experiments: 75
    conditional: False
    conditional_latency: []
    n_uchannels: 8
    meas_lo_range: [[6.663214088e+18, 7.663214088e+18], [6.783322155e+18,
7.783322155e+18], [6.718928102e+18, 7.718928102e+18], [6.610142342e+18,
7.610142342e+18], [6.846997692e+18, 7.846997692e+18]]
    memory: True
    allow_q_object: True
    description: 5 qubit device
    qubit_lo_range: [[4.462786122624366e+18, 5.462786122624367e+18],
[4.338382031192193e+18, 5.338382031192193e+18], [4.536925198971235e+18,
```

5.536925198971235e+18], [4.451289298021841e+18, 5.451289298021841e+18],
[4.566342494178997e+18, 5.566342494178997e+18]]
hamiltonian: {'description': 'Qubits are modeled as Duffing oscillators. In
this case, the system includes higher energy states, i.e. not just |0> and |1>.
The Pauli operators are generalized via the following set of
transformations:\n\n$(\\mathbb{I}-\\sigma_{i}^z)/2 \\rightarrow O_i \\equiv
b^\\dagger_{i} b_{i}$,\n\n$\\sigma_{+} \\rightarrow b^\\dagger$,\n\n$\\sigma_{-}
\\rightarrow b$,\n\n$\\sigma_{i}^X \\rightarrow b^\\dagger_{i} +
b_{i}$.\n\nQubits are coupled through resonator buses. The provided Hamiltonian
has been projected into the zero excitation subspace of the resonator buses
leading to an effective qubit-qubit flip-flop interaction. The qubit resonance
frequencies in the Hamiltonian are the cavity dressed frequencies and not
exactly what is returned by the backend defaults, which also includes the
dressing due to the qubit-qubit interactions.\n\nQuantities are returned in
angular frequencies, with units 2*pi*GHz.\n\nWARNING: Currently not all system
Hamiltonian information is available to the public, missing values have been
replaced with 0.\n', 'h_latex': '\\begin{align} \\mathcal{H}/\\hbar = & \\sum_{i
=0}^{4}\\left(\\frac{\\omega_{q,i}}{2}(\\mathbb{I}-\\sigma_i^{z})+\\frac{\\Delta
_{i}}{2}(O_i^2-O_i)+\\Omega_{d,i}D_i(t)\\sigma_i^{X}\\right) \\\\ & +
J_{0,1}(\\sigma_{0}^{+}\\sigma_{1}^{-}+\\sigma_{0}^{-}\\sigma_{1}^{+}) +
J_{1,2}(\\sigma_{1}^{+}\\sigma_{2}^{-}+\\sigma_{1}^{-}\\sigma_{2}^{+}) +
J_{2,3}(\\sigma_{2}^{+}\\sigma_{3}^{-}+\\sigma_{2}^{-}\\sigma_{3}^{+}) +
J_{3,4}(\\sigma_{3}^{+}\\sigma_{4}^{-}+\\sigma_{3}^{-}\\sigma_{4}^{+}) \\\\ & +
\\Omega_{d,0}(U_{0}^{(0,1)}(t))\\sigma_{0}^{X} +
\\Omega_{d,1}(U_{1}^{(1,0)}(t)+U_{2}^{(1,2)}(t))\\sigma_{1}^{X} \\\\ & +
\\Omega_{d,2}(U_{3}^{(2,1)}(t)+U_{4}^{(2,3)}(t))\\sigma_{2}^{X} +
\\Omega_{d,3}(U_{6}^{(3,4)}(t)+U_{5}^{(3,2)}(t))\\sigma_{3}^{X} \\\\ & +
\\Omega_{d,4}(U_{7}^{(4,3)}(t))\\sigma_{4}^{X} \\\\ \\end{align}', 'h_str':
['_SUM[i,0,4,wq{i}/2*(I{i}-Z{i})]', '_SUM[i,0,4,delta{i}/2*O{i}*O{i}]',
'_SUM[i,0,4,-delta{i}/2*O{i}]', '_SUM[i,0,4,omegad{i}*X{i}||D{i}]',
'jq0q1*Sp0*Sm1', 'jq0q1*Sm0*Sp1', 'jq1q2*Sp1*Sm2', 'jq1q2*Sm1*Sp2',
'jq2q3*Sp2*Sm3', 'jq2q3*Sm2*Sp3', 'jq3q4*Sp3*Sm4', 'jq3q4*Sm3*Sp4',
'omegad1*X0||U0', 'omegad0*X1||U1', 'omegad2*X1||U2', 'omegad1*X2||U3',
'omegad3*X2||U4', 'omegad4*X3||U6', 'omegad2*X3||U5', 'omegad3*X4||U7'], 'osc':
{}, 'qub': {'0': 3, '1': 3, '2': 3, '3': 3, '4': 3}, 'vars': {'delta0':
-2.1573187977651487, 'delta1': -2.1753119475601674, 'delta2':
-2.159281266514359, 'delta3': -2.158603148482815, 'delta4': -2.1495256907311115,
'jq0q1': 0.011836082919670043, 'jq1q2': 0.01196783968906386, 'jq2q3':
0.012402113956012368, 'jq3q4': 0.012186910370408229, 'omegad0':
0.9509675935130749, 'omegad1': 0.9790127567442053, 'omegad2':
0.9464853316063281, 'omegad3': 0.9623594258007115, 'omegad4':
0.9744079339937477, 'wq0': 31.182104848348175, 'wq1': 30.40045088890851, 'wq2':
31.647934403538684, 'wq3': 31.10986816892636, 'wq4': 31.832768720565053}}
open_pulse: False
multi_meas_enabled: True
rep_times: [0.001]
backend_name: ibmq_manila
url: None

credits_required: True
meas_kernels: ['hw_qmfk']
rep_delay_range: [0.0, 500.0]
dt: 0.2222222222222222
uchannels_enabled: True
meas_levels: [1, 2]
online_date: 2021-04-28 04:00:00+00:00
u_channel_lo: [[{'q': 1, 'scale': (1+0j)}], [{'q': 0, 'scale': (1+0j)}], [{'q': 2, 'scale': (1+0j)}], [{'q': 1, 'scale': (1+0j)}], [{'q': 3, 'scale': (1+0j)}], [{'q': 2, 'scale': (1+0j)}], [{'q': 4, 'scale': (1+0j)}], [{'q': 3, 'scale': (1+0j)}]]
dtm: 0.2222222222222222
input_allowed: ['job']
acquisition_latency: []
coupling_map: [[0, 1], [1, 0], [1, 2], [2, 1], [2, 3], [3, 2], [3, 4], [4, 3]]
meas_map: [[0, 1, 2, 3, 4]]
allow_object_storage: True
quantum_volume: 32
parametric_pulses: ['gaussian', 'gaussian_square', 'drag', 'constant']
max_shots: 8192
default_rep_delay: 250.0
channels: {'acquire0': {'operates': {'qubits': [0]}, 'purpose': 'acquire', 'type': 'acquire'}, 'acquire1': {'operates': {'qubits': [1]}, 'purpose': 'acquire', 'type': 'acquire'}, 'acquire2': {'operates': {'qubits': [2]}, 'purpose': 'acquire', 'type': 'acquire'}, 'acquire3': {'operates': {'qubits': [3]}, 'purpose': 'acquire', 'type': 'acquire'}, 'acquire4': {'operates': {'qubits': [4]}, 'purpose': 'acquire', 'type': 'acquire'}, 'd0': {'operates': {'qubits': [0]}, 'purpose': 'drive', 'type': 'drive'}, 'd1': {'operates': {'qubits': [1]}, 'purpose': 'drive', 'type': 'drive'}, 'd2': {'operates': {'qubits': [2]}, 'purpose': 'drive', 'type': 'drive'}, 'd3': {'operates': {'qubits': [3]}, 'purpose': 'drive', 'type': 'drive'}, 'd4': {'operates': {'qubits': [4]}, 'purpose': 'drive', 'type': 'drive'}, 'm0': {'operates': {'qubits': [0]}, 'purpose': 'measure', 'type': 'measure'}, 'm1': {'operates': {'qubits': [1]}, 'purpose': 'measure', 'type': 'measure'}, 'm2': {'operates': {'qubits': [2]}, 'purpose': 'measure', 'type': 'measure'}, 'm3': {'operates': {'qubits': [3]}, 'purpose': 'measure', 'type': 'measure'}, 'm4': {'operates': {'qubits': [4]}, 'purpose': 'measure', 'type': 'measure'}, 'u0': {'operates': {'qubits': [0, 1]}, 'purpose': 'cross-resonance', 'type': 'control'}, 'u1': {'operates': {'qubits': [1, 0]}, 'purpose': 'cross-resonance', 'type': 'control'}, 'u2': {'operates': {'qubits': [1, 2]}, 'purpose': 'cross-resonance', 'type': 'control'}, 'u3': {'operates': {'qubits': [2, 1]}, 'purpose': 'cross-resonance', 'type': 'control'}, 'u4': {'operates': {'qubits': [2, 3]}, 'purpose': 'cross-resonance', 'type': 'control'}, 'u5': {'operates': {'qubits': [3, 2]}, 'purpose': 'cross-resonance', 'type': 'control'}, 'u6': {'operates': {'qubits': [3, 4]}, 'purpose': 'cross-resonance', 'type': 'control'}, 'u7': {'operates': {'qubits': [4, 3]}, 'purpose': 'cross-resonance', 'type': 'control'}}

```
    pulse_num_channels: 9
    dynamic_reprate_enabled: True
    n_registers: 1
    processor_type: {'family': 'Falcon', 'revision': '5.11', 'segment': 'L'}
    supported_instructions: ['acquire', 'play', 'u1', 'u3', 'id', 'measure',
'delay', 'cx', 'u2', 'x', 'rz', 'sx', 'reset', 'setf', 'shiftf']
    qubit_channel_mapping: [['m0', 'd0', 'u0', 'u1'], ['u1', 'u3', 'u0', 'u2',
'd1', 'm1'], ['u4', 'u3', 'u5', 'u2', 'm2', 'd2'], ['d3', 'm3', 'u4', 'u7',
'u5', 'u6'], ['m4', 'd4', 'u6', 'u7']]
    pulse_num_qubits: 3
    discriminators: ['hw_qmfk', 'quadratic_discriminator',
'linear_discriminator']
    sample_name: family: Falcon, revision: 5.11, segment: L

Qubits [Name / Freq / T1 / T2 / RZ err / SX err / X err / Readout err]
----------------------------------------------------------------------
    Q0 / 4.96279 GHz / 129.07081 us / 93.73012 us / 0.00000 / 0.00024 / 0.00024
/ 0.02620
    Q1 / 4.83838 GHz / 154.33295 us / 105.31840 us / 0.00000 / 0.00023 / 0.00023
/ 0.03050
    Q2 / 5.03693 GHz / 132.49478 us / 24.04182 us / 0.00000 / 0.00023 / 0.00023
/ 0.01840
    Q3 / 4.95129 GHz / 210.38901 us / 68.63889 us / 0.00000 / 0.00024 / 0.00024
/ 0.02350
    Q4 / 5.06634 GHz / 128.53225 us / 43.04893 us / 0.00000 / 0.00036 / 0.00036
/ 0.01880

Multi-Qubit Gates [Name / Type / Gate Error]
--------------------------------------------
    cx4_3 / cx / 0.00519
    cx3_4 / cx / 0.00519
    cx2_3 / cx / 0.00619
    cx3_2 / cx / 0.00619
    cx1_2 / cx / 0.00932
    cx2_1 / cx / 0.00932
    cx0_1 / cx / 0.00704
    cx1_0 / cx / 0.00704
```

[20]:

```
%qiskit_job_watcher

job_r = execute(qc_Grover, backend_device, shots=shots)

jobID_r = job_r.job_id()

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

```
Accordion(children=(VBox(layout=Layout(max_width='710px', min_width='710px')),),␣
↪layout=Layout(max_height='500…
```

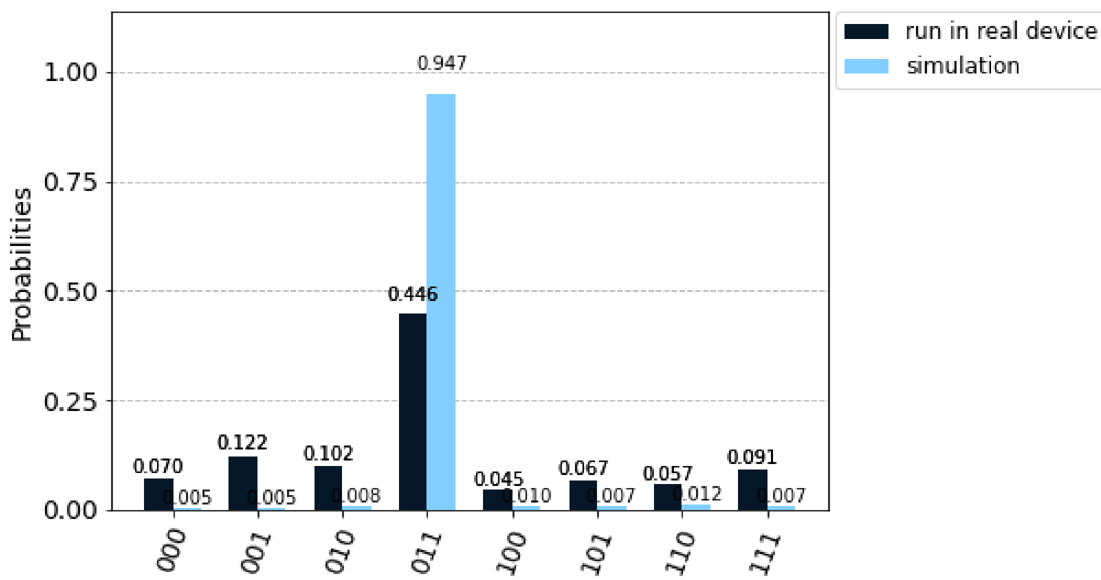
```
<IPython.core.display.Javascript object>
```

```
JOB ID: 60bbcc84b454d035e6aa68f2
```

[21]:
```
%qiskit_disable_job_watcher
job_get=backend_device.retrieve_job("60bbcc84b454d035e6aa68f2")

result_r = job_get.result()
counts_run = result_r.get_counts(qc_Grover)
```

[22]:
```
plot_histogram([counts_run, count ], legend=[ 'run in real device',
↪'simulation'], color=['#061727','#82cfff'])
```

[22]:



Ao correr o código numa máquina real, podemos concluir que os problemas de ruído de facto existem e que atualmente as máquinas não são perfeitas. Por isso, em computação quântica, é necessário uma atenção especial para evitar que esses mesmos erros existam.

1.5.1 3.1 Otimização

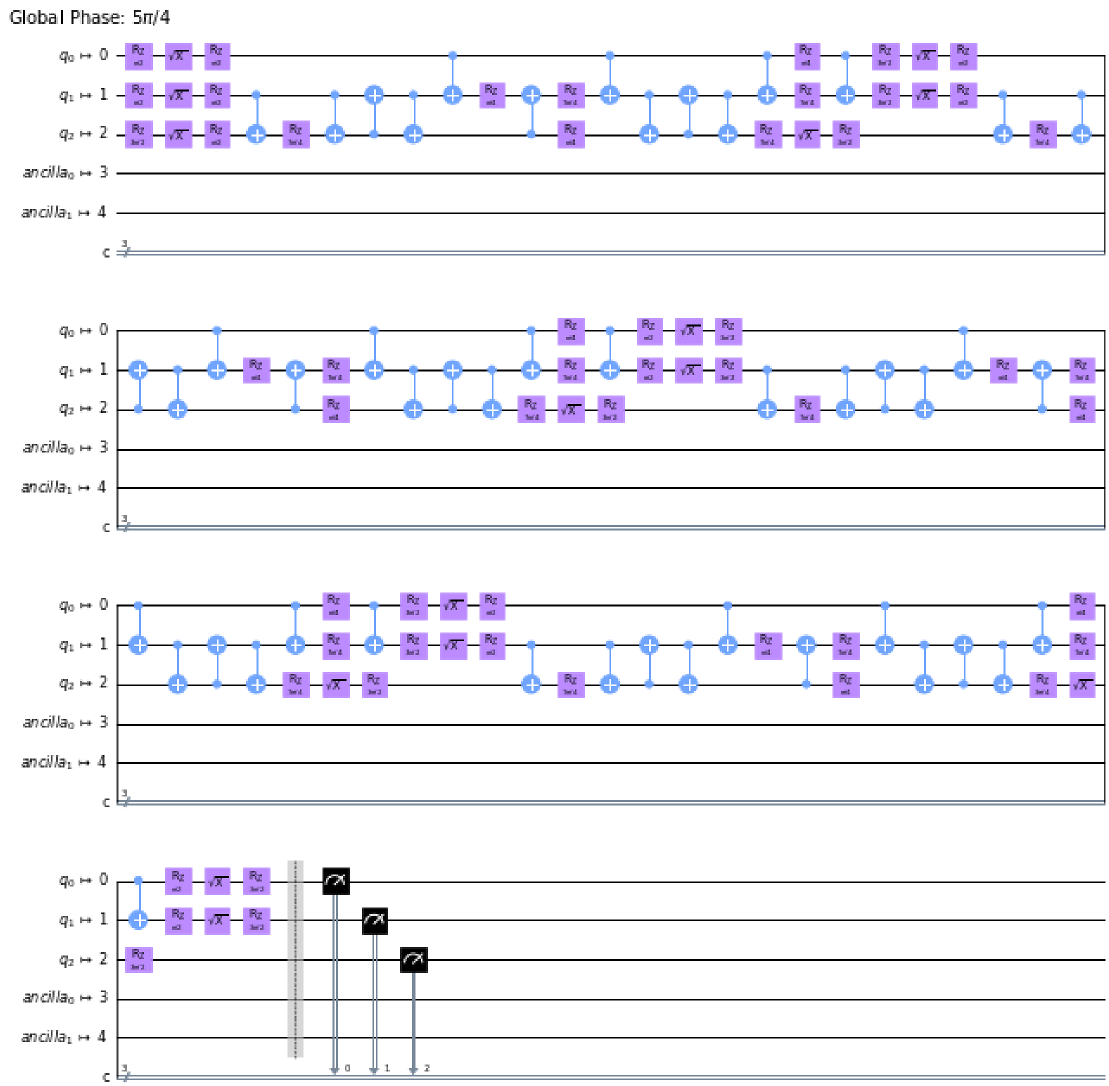
Para otimizar o nosso circuito vamos usar transpilers. Resumidamente o transpiler recebe um circuito e reformula-o de maneira a que o mesmo se alinhe melhor com a tipologia da máquina em questão tornando-o mais efeciente. Ao alinhar o circuito com a máquina, o tempo de execução baixa, e consequentemente, como o circuito é executado mais rápido, a probabilidade de haver erros devido a ruído diminui.

Neste trabalho, vamos testar os três diferentes niveis de otimização com transpilers.

Otimização nivel 1

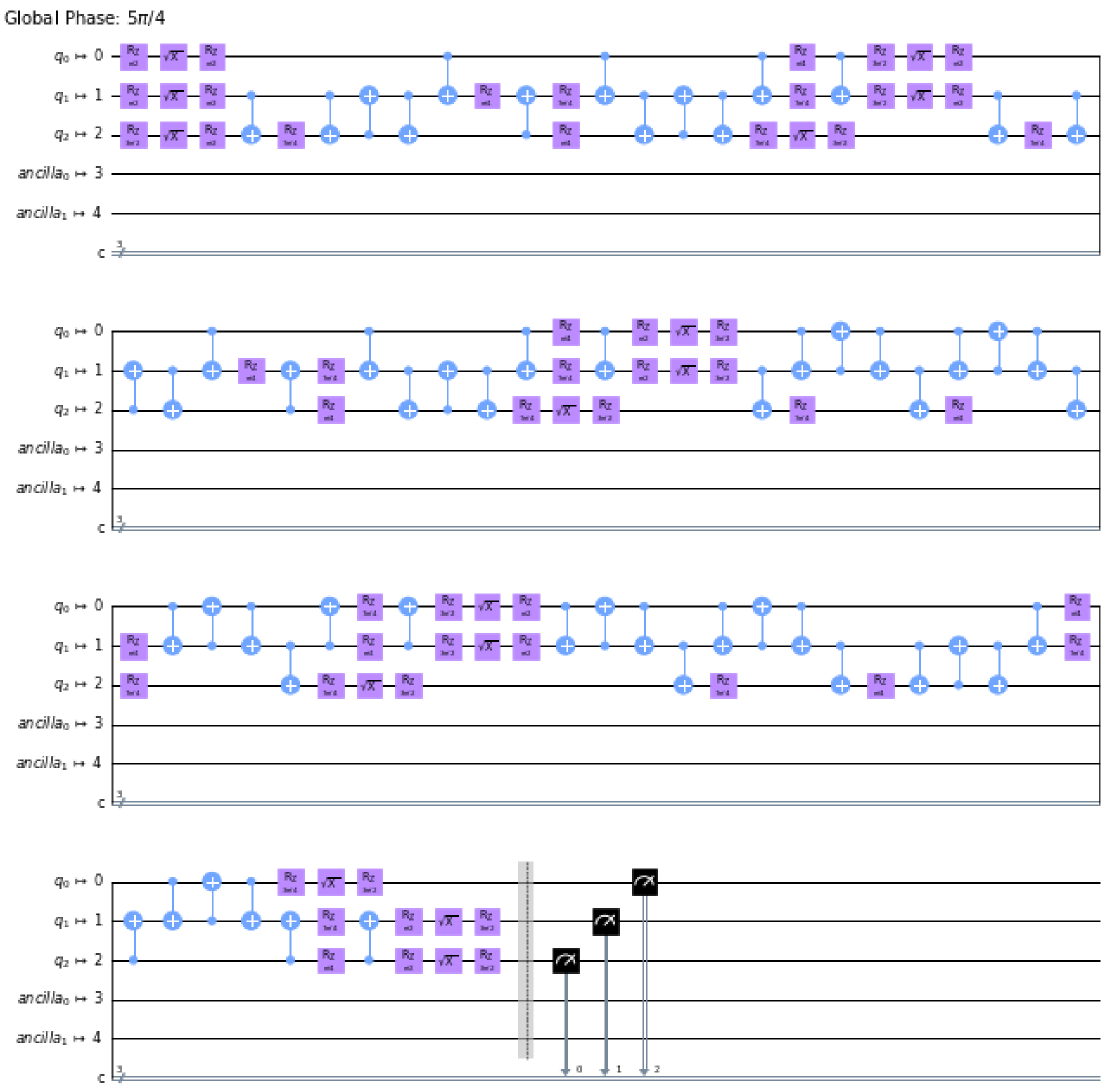
[23]:
```
qc_t_real = transpile(qc_Grover, backend=backend_device)

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

[23]:



[24]:
```
qc_optimized1 = transpile(qc_Grover, backend=backend_device,␣
↪optimization_level=1)
qc_optimized1.draw(output='mpl', scale=0.5)
```

[24]:



[25]:
```
qc_Grover.depth()
```

[25]: 26

[26]:
```
qc_t_real.depth()
```

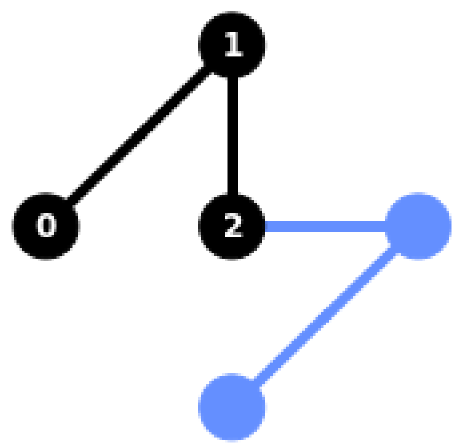
[26]: 80

[27]:
```
plot_circuit_layout(qc_t_real, backend_device)

qc_optimized1.depth()
```

[27]: 86

[28]:
```
plot_circuit_layout(qc_optimized1, backend_device)
```

[28]:



[29]:
```
%qiskit_job_watcher
job_exp1 = execute(qc_optimized1, backend_device, shots = shots)

# job_id allows you to retrive old jobs
jobID = job_exp1.job_id()

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

job_exp1.result().get_counts(qc_optimized1)
```

```
Accordion(children=(VBox(layout=Layout(max_width='710px', min_width='710px')),),␣
↪layout=Layout(max_height='500…
```

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

```
JOB ID: 60bbccf743987e3932f10627
```

[29]:
```
{'000': 102,
 '001': 93,
 '010': 150,
 '011': 366,
 '100': 78,
```

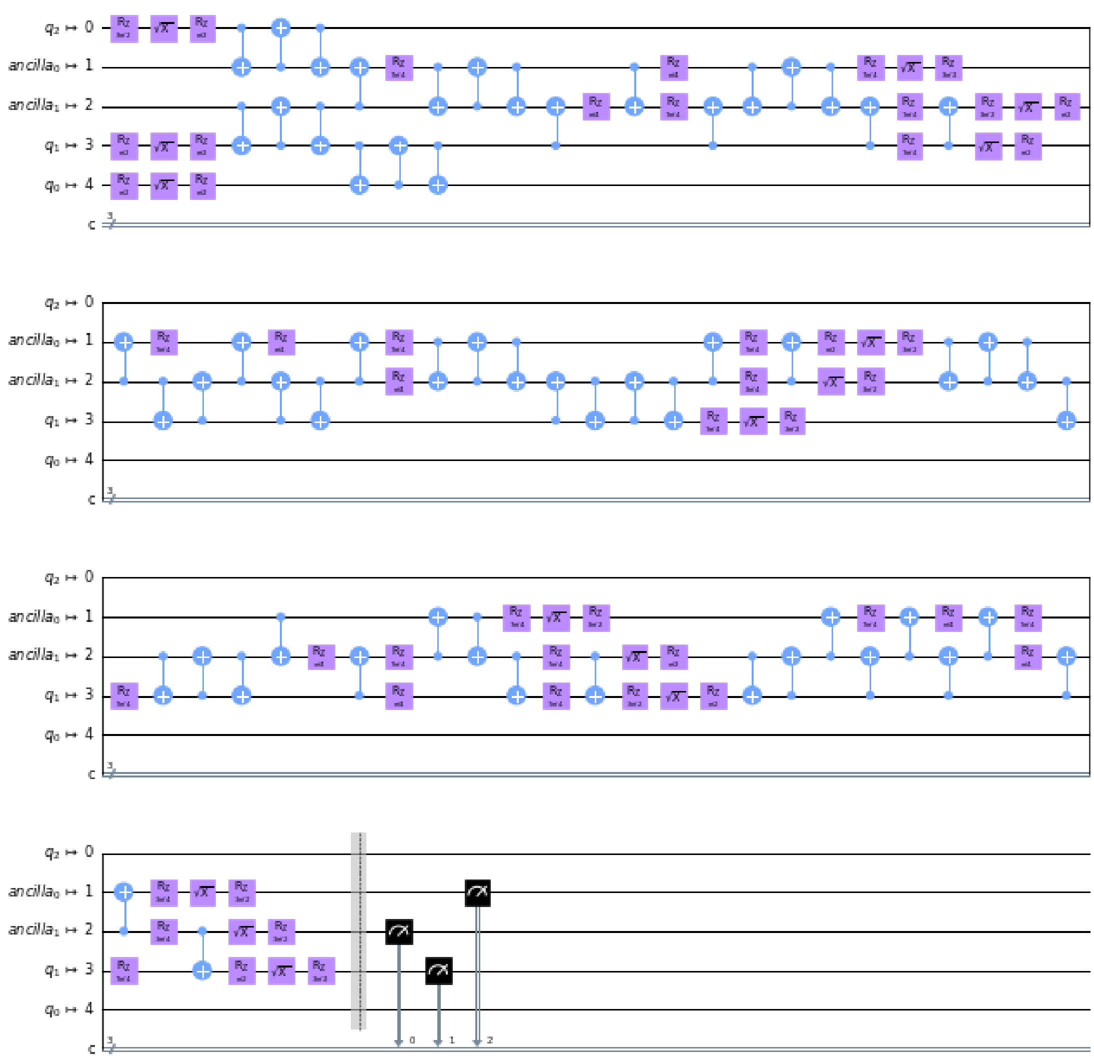
```
 '101': 93,
 '110': 62,
 '111': 80}
```

[30]:
```
#with optimization 1
job_get_o1 = backend_device.retrieve_job("60bbccf743987e3932f10627")

result_real_o1 = job_get_o1.result(timeout=3600, wait=5)

counts_opt1 = result_real_o1.get_counts(qc_optimized1)
```

Otimização nivel 2

[31]:
```
qc_optimized2 = transpile(qc_Grover, backend=backend_device,␣
 ↪optimization_level=2)
qc_optimized2.draw(output='mpl', scale=0.5)
```

[31]:

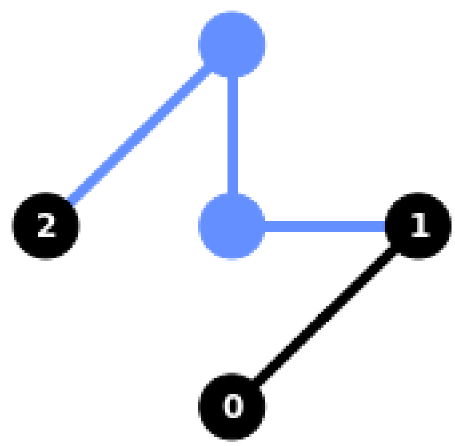


[32]:
```
qc_optimized2.depth()
```

[32]: 82

[33]:
```
plot_circuit_layout(qc_optimized2, backend_device)
```

[33]:



[34]:
```
job_exp2 = execute(qc_optimized2, backend_device, shots = shots)

# job_id allows you to retrive old jobs
jobID = job_exp2.job_id()

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

job_exp2.result().get_counts(qc_optimized2)
```

```
JOB ID: 60bbcd2225cc6ea71d65c813
```

[34]:
```
{'000': 102,
 '001': 153,
 '010': 116,
 '011': 251,
```

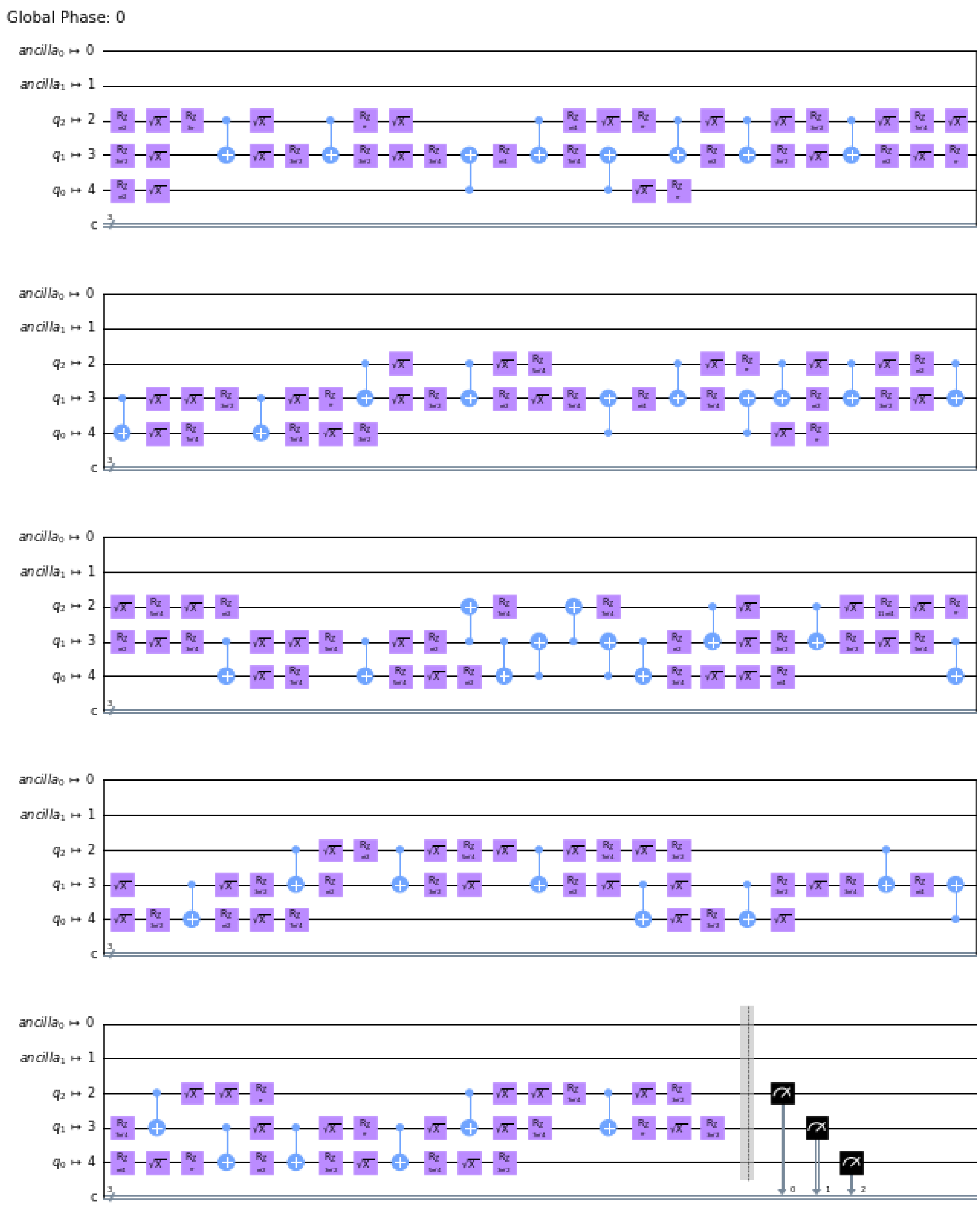
```
 '100': 83,
 '101': 102,
 '110': 134,
 '111': 83}
```

[35]:
```
#with optimization 2
job_get_o2 = backend_device.retrieve_job("60bbcd2225cc6ea71d65c813")

result_real_o2 = job_get_o2.result(timeout=3600, wait=5)

counts_opt2 = result_real_o2.get_counts(qc_optimized2)
```

Otimização nivel 3

[36]:
```
qc_optimized3 = transpile(qc_Grover, backend=backend_device,␣
 ↪optimization_level=3)
qc_optimized3.draw(output='mpl', scale=0.5)
```

[36]:

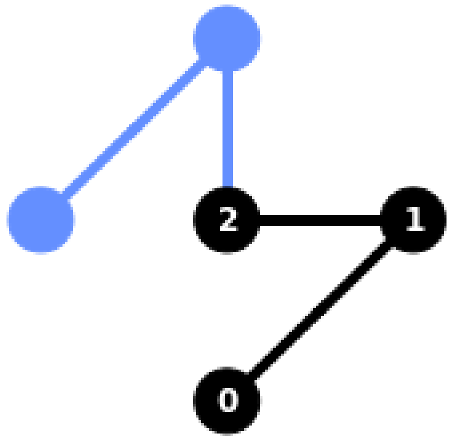


[37]:
```
qc_optimized3.depth()
```

[37]: 119

[38]:
```
plot_circuit_layout(qc_optimized3, backend_device)
```

[38]:



[39]:
```
job_exp3 = execute(qc_optimized3, backend_device, shots = shots)

# job_id allows you to retrive old jobs
jobID = job_exp3.job_id()

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

job_exp3.result().get_counts(qc_optimized3)
```

```
JOB ID: 60bbcd6100aded18c86a713d
```

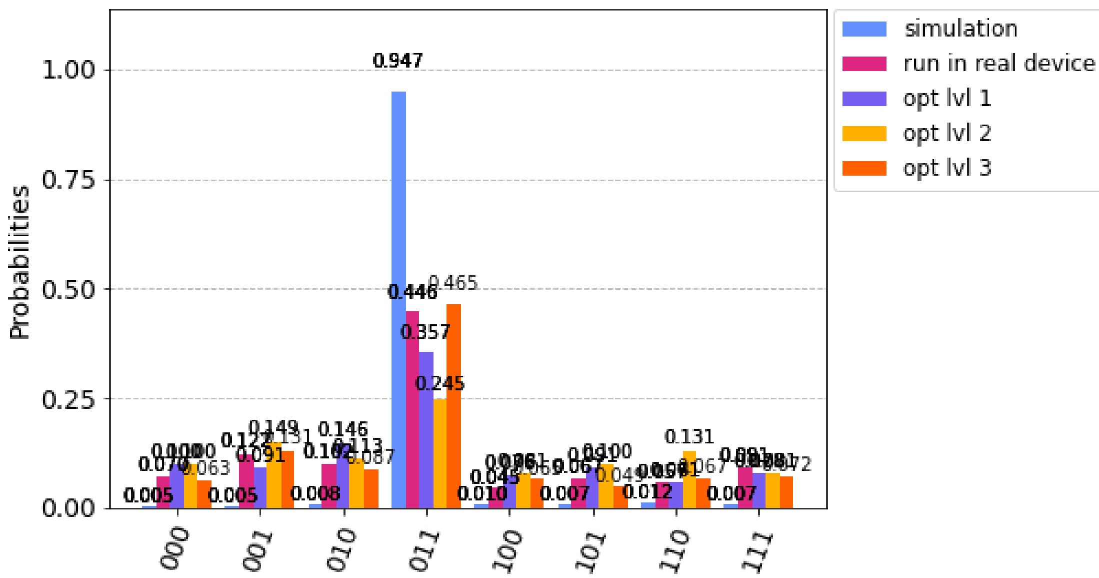
[39]:
```
{'000': 65,
 '001': 134,
 '010': 89,
 '011': 476,
 '100': 67,
 '101': 50,
 '110': 69,
 '111': 74}
```

[40]:
```
#with optimization 3
job_get_o3 = backend_device.retrieve_job("60bbcd6100aded18c86a713d")
```

```
result_real_o3 = job_get_o3.result(timeout=3600, wait=5)

counts_opt3 = result_real_o3.get_counts(qc_optimized3)
%qiskit_disable_job_watcher
```

[41]:
```
plot_histogram([count, counts_run, counts_opt1, counts_opt2, counts_opt3],␣
↪legend=[ 'simulation', 'run in real device', "opt lvl 1", "opt lvl 2", "opt␣
↪lvl 3"])
```

[41]:



Como podemos ver no gráfico em cima, obtemos melhores resultados quando usamos o transpiler com optmização de nível 3. Nos restantes pontos deste trabalho vamos usar estes resultados em vez dos resultados sem optimização.

1.6 4. Mitigação dos erros

Neste trabalhos vamos usar o **Qiskit Ignis**. O Qiskit Ignis é uma *framework* que tenta entender e mitigar o ruído de circuitos quânticos.

1.6.1 Calibration Matrices

a. Gerar uma lista de circuitos de calibração de medida Cada circuito cria um estado base. Dados que a número máximo de bits é 3, temos de criar $2^3 = 8$ circuitos.

[42]:
```
# Generate the calibration circuits
qr = QuantumRegister(x)
meas_calibs, state_labels = complete_meas_cal(qubit_list=[0,1,2], qr=qr,␣
↪circlabel='mcal')
state_labels
```

[42]: ['000', '001', '010', '011', '100', '101', '110', '111']

b. Computar a matrix de calibração Se não houvesse ruído na máquina, a matriz de calibração seria a matriz Id_{8x8}. Como esta matriz é computada numa máquina quântica, vai sempre existir algum ruído que a vai alterar.

[43]:
```
%qiskit_job_watcher
job_ignis = execute(meas_calibs, backend=backend_device, shots=shots)

jobID_run_ignis = job_ignis.job_id()

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

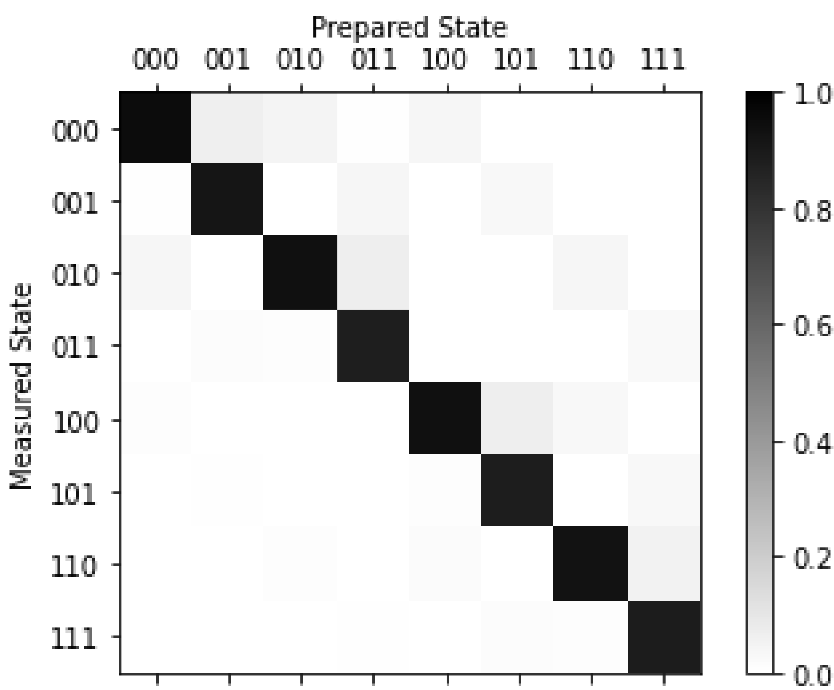
```
Accordion(children=(VBox(layout=Layout(max_width='710px', min_width='710px')),),␣
↪layout=Layout(max_height='500…
```

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

```
JOB ID: 60bbce581eb0244d01ceeb66
```

[44]:
```
job_get=backend_device.retrieve_job("60bbce581eb0244d01ceeb66")

cal_results = job_get.result()
```

[45]:
```
meas_fitter = CompleteMeasFitter(cal_results, state_labels, circlabel='mcal')

# Plot the calibration matrix
meas_fitter.plot_calibration()
```



c. Análise de resultados Como os elementos na diagonal da matriz de calibração obtida são as probabilidades de medir o estado x dada a preparação do estado x, então o traço dessa matriz é a fidelidade média da atribuição

[46]:
```
%qiskit_disable_job_watcher
print("Average Measurement Fidelity: %f" % meas_fitter.readout_fidelity())
```

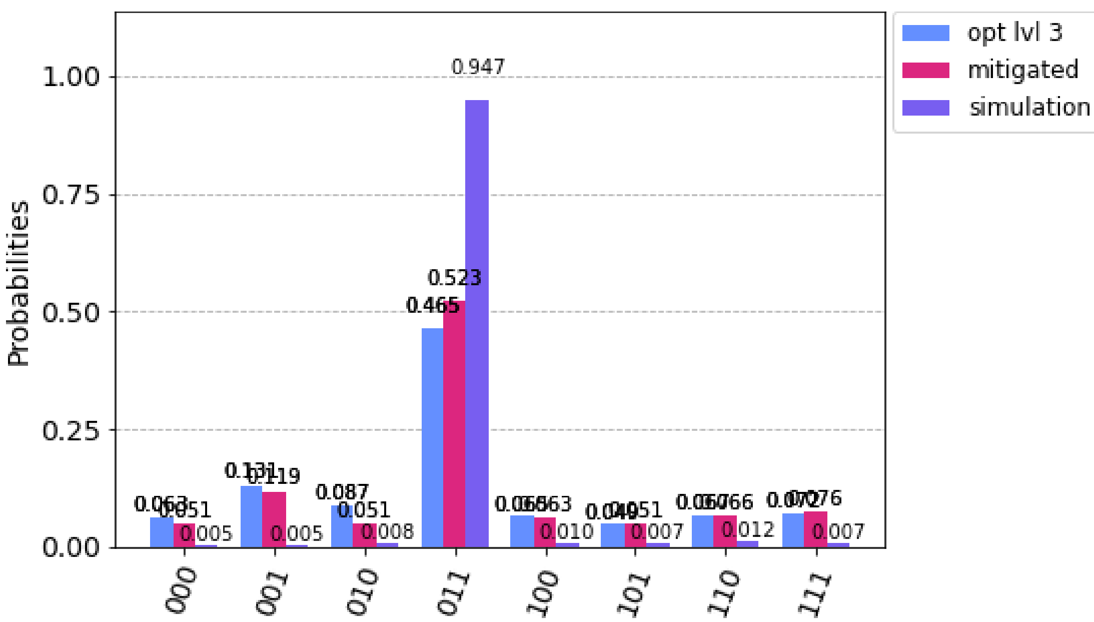
```
Average Measurement Fidelity: 0.914551
```

d. Aplicação da Calibração Aplicando então o filtro de calibração obtemos os resultados seguintes:

[47]:
```
# Get the filter object
meas_filter = meas_fitter.filter

# Results with mitigation
mitigated_results = meas_filter.apply(result_real_o3)
mitigated_counts = mitigated_results.get_counts()
```

[48]:
```
plot_histogram([counts_opt3, mitigated_counts, count], legend=['opt lvl 3',␣
↪'mitigated', 'simulation'])
```

[48]:



Ao comparar o circuito otimizado e com os erros mitigados ao circuito inicial, conseguimos perceber que existem várias técnicas para melhorar os problemas de ruído quântico.

1.7 5. Conclusão

Como ainda estamos numa fase inícial relativamente à computação quântica, a nível de hardware ainda existem imensos problemas. Esses problemas acabam por gerar ruído. Com isto, esse ruído gera bastantes problemas no resultado. Basta analizar que idealmente deveriamos ter 100% de hipótese de obter o estado $|110\rangle$ como estado solução, mas no entanto, mesmo após otimizar e mitigar os erros, as probabilidades são inferiores a 55% (embora estes resultados tenham sido muito melhores do que o normal, continua a ser demasiado longe do que é considerado ideal). Enquanto pessoa que gera o código para máquina, uma das maiores preocupações, é fazer o código de maneira a que os erros sejam o menor possível.

1.8 6. Extra

Como ponto extra, o nosso grupo decidiu implementar uma função que gera um oráculo que marca o elemento especificado no input (parâmetro *elem*), sendo este input uma string com o elemento a procurar (apenas elementos com 3 bits).

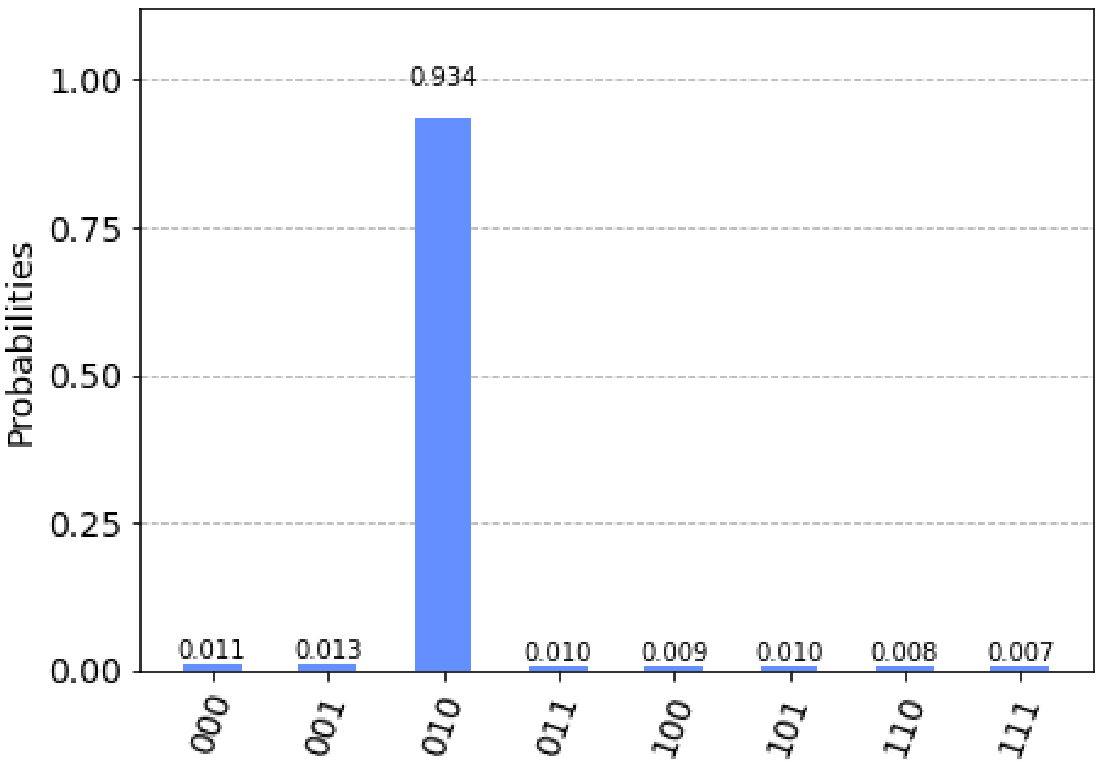
[55]:
```
def select(elem,circuit,qr):
    for i in range(len(elem)):
        if elem[i] == '0':
            circuit.x(qr[len(elem) - 1 - i])

def phase_oracle_gen(elem,circuit, qr):
    select(elem,circuit,qr)
    ccz(circuit,qr)
    select(elem,circuit,qr)
```

[56]:
```
def buildCircuitGroverGen(elem,f):
    qr = QuantumRegister(x, 'q')
    cr = ClassicalRegister(x,'c')
    circuit = QuantumCircuit(qr,cr)

    init(circuit,qr)

    for t in range(f):
        # phase oracle
        phase_oracle_gen(elem,circuit,qr)

        # diffuser
        diffuser(circuit,qr)

    circuit.measure(qr,cr)

    return circuit
```

[60]:
```
qc_GroverGen = buildCircuitGroverGen("010",2)
resultGen = execute(qc_GroverGen, backend_qasm, shots=shots).result()
countGen = resultGen.get_counts(qc_GroverGen)
```

```
plot_histogram(countGen)
```

[60]:



1.9 Anexo

[1]:
```
#IMPORTS
import qiskit
import qiskit.tools.jupyter
from qiskit import Aer, IBMQ
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister
from qiskit import execute, transpile
from qiskit.quantum_info import Statevector
from qiskit.tools.visualization import plot_histogram, plot_state_city,␣
 ↪plot_state_hinton, plot_bloch_multivector
import matplotlib.pyplot as plt
%matplotlib inline
from qiskit.tools.monitor import backend_overview, backend_monitor
from qiskit.visualization import plot_circuit_layout
from qiskit.compiler import transpile
from qiskit.ignis.mitigation.measurement import (complete_meas_cal,␣
 ↪tensored_meas_cal,
                                                 CompleteMeasFitter,␣
 ↪TensoredMeasFitter)
```

```
#s = NºGrupo Mod 8  // Numero a procurar
s = 19 % 8
#s to bin
wb = bin(s)[2:]

# 7 = {111} then we need 3 qubits
x = 3


def init(circuit,qr):
    circuit.h(qr)

def ccz(circuit, qr):
    circuit.h(qr[2])
    circuit.ccx(qr[0],qr[1],qr[2])
    circuit.h(qr[2])

def phase_oracle(circuit, qr):
    #muda a fase do estado |011>  -> |qr[2] qr[1] qr[0]>
    circuit.x(qr[2])
    ccz(circuit,qr)
    circuit.x(qr[2])

def diffuser(circuit, qr):
    circuit.h(qr)
    circuit.x(qr)
    ccz(circuit, qr)
    circuit.x(qr)
    circuit.h(qr)

def buildCircuitGrover(f):
    qr = QuantumRegister(x, 'q')
    cr = ClassicalRegister(x,'c')
    circuit = QuantumCircuit(qr,cr)

    init(circuit,qr)

    for t in range(f):
        # phase oracle
        phase_oracle(circuit,qr)

        # diffuser
        diffuser(circuit,qr)

    circuit.measure(qr,cr)
```

```
    return circuit
```