TP2 Interacao e Concorrencia

June 5, 2021

1 TP2: Interação e Concorrência

Realizado por:
André Araújo A87987
Paulo Costa A87986
Grupo 5

1.1 Enunciado:

Each group of students has a number assigned, N. Now, you have to use a quantum algorithm to find s

$$s = N mod 8$$

in an unsorted list.

Implement the correct algorithm in a Jupyter Notebook file. Each work should contain (and will be evaluated on) the following steps: 1. Division of the algorithm into sections; Utilisation of the state vector simulator to explain each step (special attention to the oracle); 2. Application of noise simulator to predict the best optimisation; 3. Execution in an IBM Q backend. 4. Mitigation of Error with Ignis.

1.2 Resolução:

[1]:
```
# importing Qiskit
from qiskit import Aer, IBMQ
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister
from qiskit import execute, transpile

from qiskit.tools.visualization import plot_histogram, plot_state_city,␣
↪plot_state_hinton

import matplotlib.pyplot as plt
%matplotlib inline
```

[2]:
```
backend_vector = Aer.get_backend("statevector_simulator")
backend = Aer.get_backend("qasm_simulator")
```

1.2.1 1)

Para encontar o $s = 5$ numa lista não ordenada, vamos implementar o algoritmo de Grover apresentado como um algoritmo rápido para resolver problemas de procura em bases de dados ou listas não ordenadas, pois este consegue resolver estes problemas em apenas avaliações da função $\mathcal{O}(\sqrt{D})$, em que D é o tamanho do domínio da função, já o problema análogo na computação clássica não pode ser resolvido em menos de $\mathcal{O}(D)$ avaliações.

A implementação deste algoritmo tem 3 passos: >1. Inicializar o sistema; >2. Repetir $\sqrt{D}$ vezes:
a) O operador quantico oraculo;
b) A transformação de difusão; >3. Medir o valor dos qubits.

[3]:
```
N=5
s = N % 8
print(N,'mod 8 =',s)
sb = bin(s)[2:]
print(N,'em binario =',sb)
```

```
5 mod 8 = 5
5 em binario = 101
```

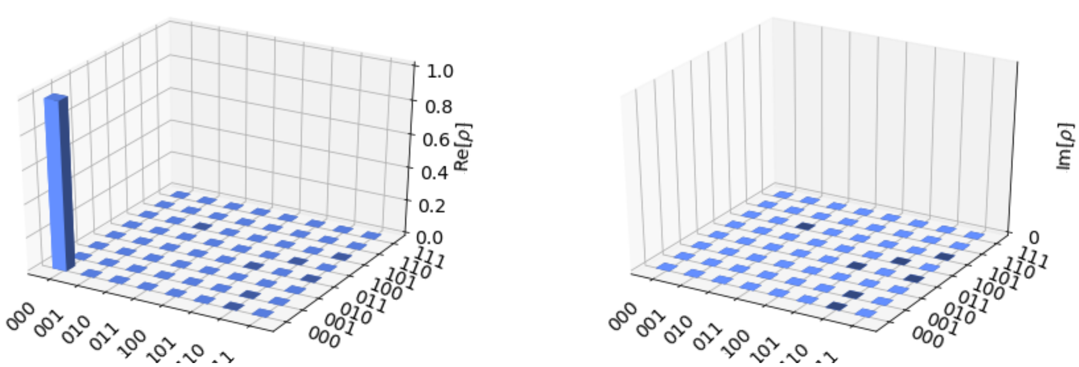
[4]:
```
X=3
```

[5]:
```
qr = QuantumRegister(X,'q')
cr = ClassicalRegister(X,'c')
qc = QuantumCircuit(qr,cr)
```

[6]:
```
result = execute(qc, backend_vector).result()
qstate= result.get_statevector(qc)
print(qstate)
plot_state_city(qstate)
```

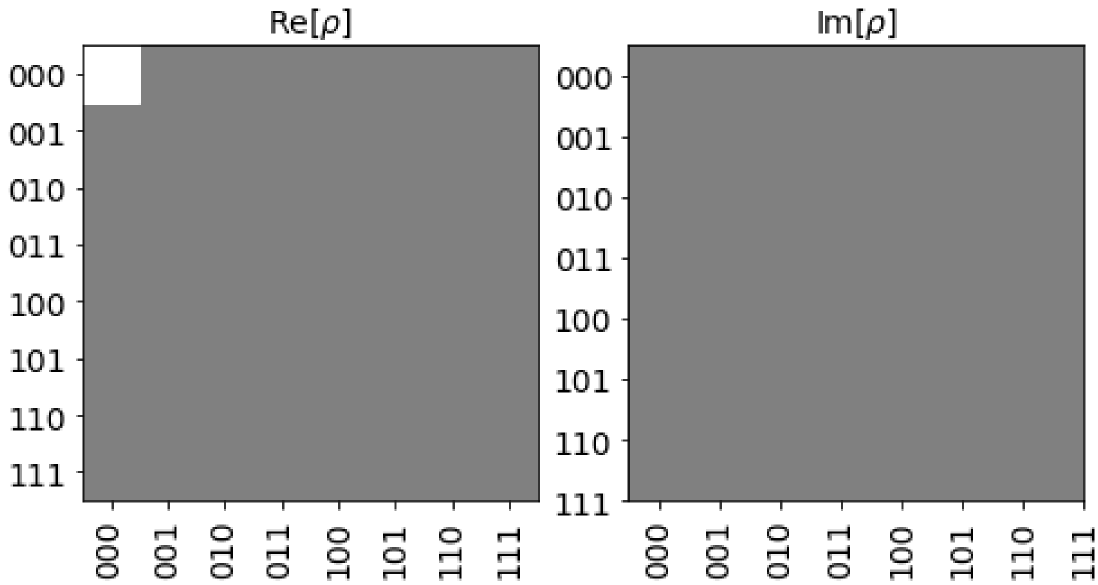
```
[1.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
```

[6]:



[7]:
```
plot_state_hinton(qstate)
```

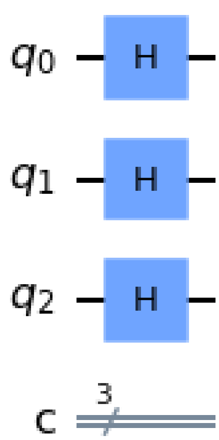
[7]:



1. Começamos por inicializar o sistema em todos os estados possiveis com a mesma amplitude

[8]:
```
# init
qc.h(qr)

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

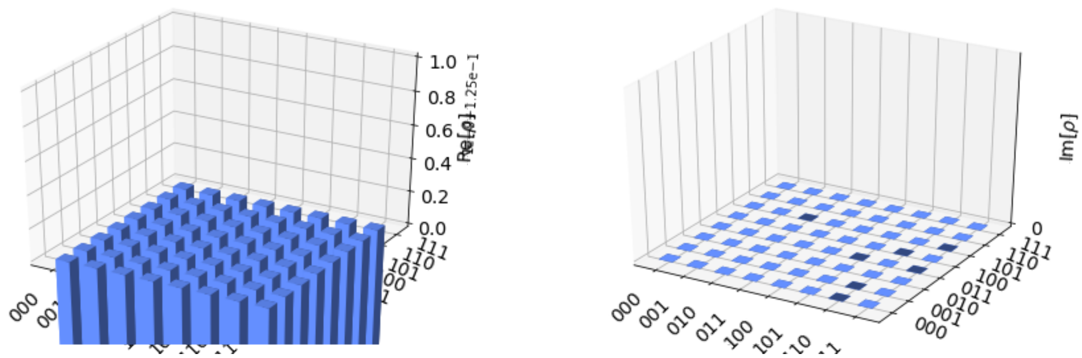
[8]:



[9]:
```
result = execute(qc, backend_vector).result()
qstate= result.get_statevector(qc)
print(qstate)
plot_state_city(qstate)
```

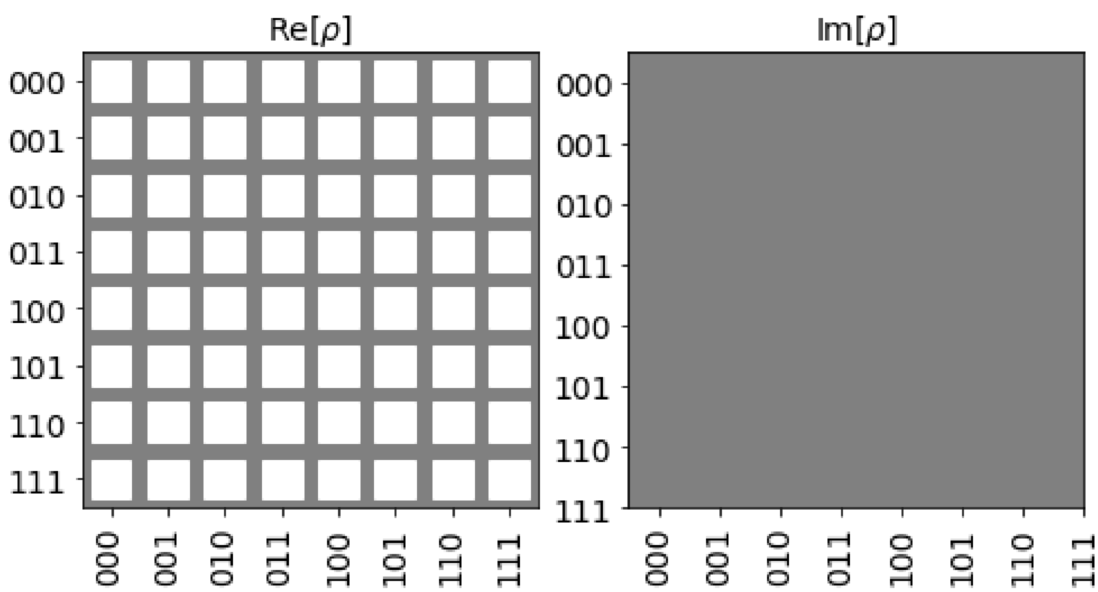
```
[0.35355339+0.j 0.35355339+0.j 0.35355339+0.j 0.35355339+0.j
 0.35355339+0.j 0.35355339+0.j 0.35355339+0.j 0.35355339+0.j]
```

[9]:



[10]:
```
plot_state_hinton(qstate)
```

[10]:



2. a) O operador quantico oraculo U_w é o responsavel por identificar as soluções para o problema e indicar o alvo da solução.

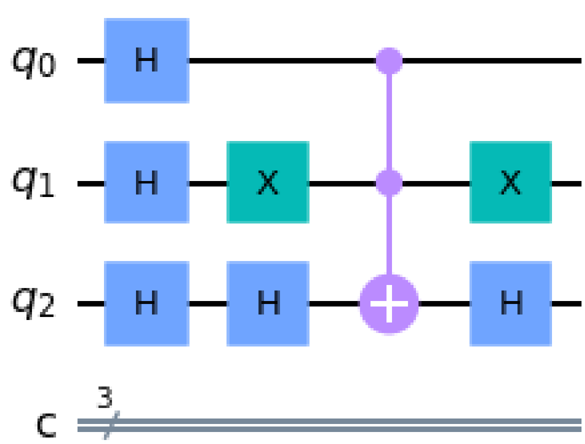
$$U_w|x\rangle = (1)^f(x)|x\rangle$$

,onde $f(|101\rangle) = 1$ e $\forall_{xi \neq |101\rangle} f(x) = 0$.

Assim, desta forma o estado marcado roda π radians marcando o valor simetrico e todos os outros estado mantem o sistema inalterado.
Para isso aplicamos a gate de Pauli-X no qubit 1 deforma a marcar o estado e de seguida aplicamos a gate CCZ como esta não existe compomos esta gate utilizando Hadamard e CCX e voltamos a aplicar gate de Pauli-X no qubit 1.

[11]:
```python
# Oraculo
def oraculo():
    qc.x(qr[1])
    qc.h(qr[2])
    qc.ccx(qr[0], qr[1], qr[2])
    qc.h(qr[2])
    qc.x(qr[1])

oraculo()
qc.draw(output='mpl')
```

[11]:

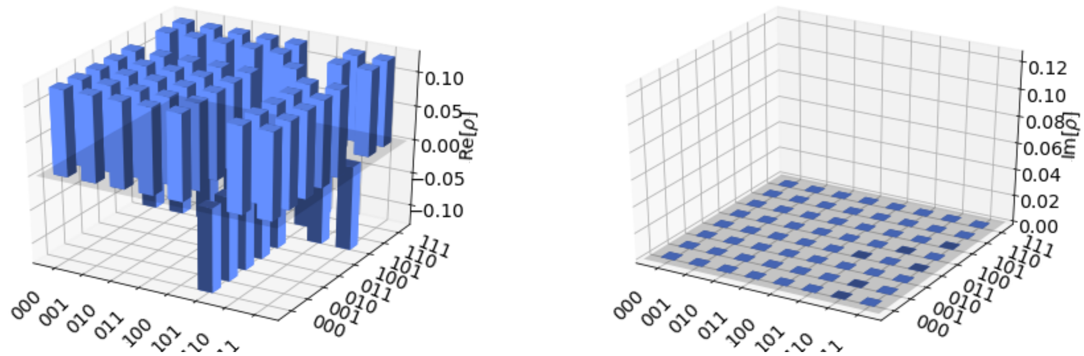


[12]:
```python
result = execute(qc, backend_vector).result()
qstate= result.get_statevector(qc)

print(qstate)
plot_state_city(qstate)
```

```
[ 0.35355339+0.00000000e+00j  0.35355339-4.32978028e-17j
  0.35355339+0.00000000e+00j  0.35355339+0.00000000e+00j
```

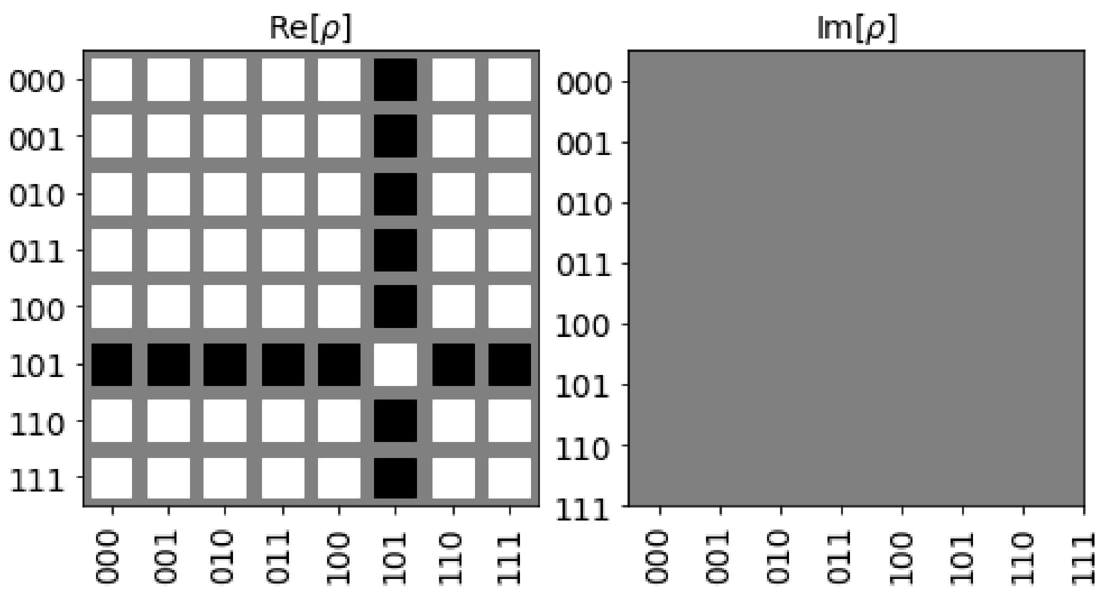
```
0.35355339+0.00000000e+00j -0.35355339+4.32978028e-17j
0.35355339+0.00000000e+00j  0.35355339+0.00000000e+00j]
```

[12]:



[13]:
```
plot_state_hinton(qstate)
```

[13]:



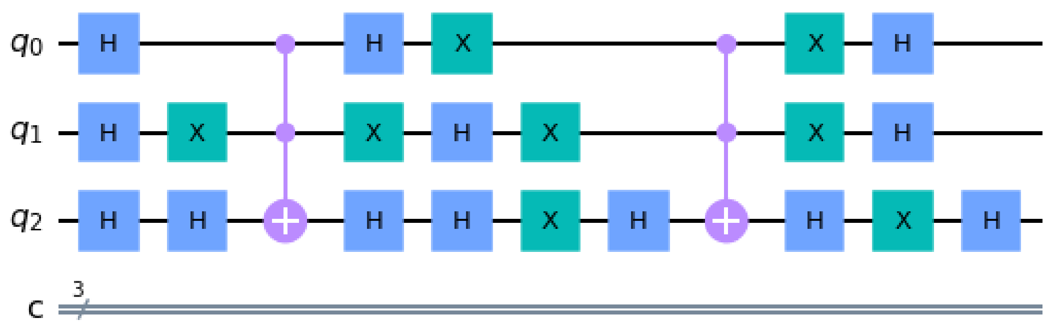
2. b) A transformação de difusão é obtida ao aplicar a gate de Hadamard, seguida da gate Pauli-X a todos os qubits, depois aplicámos a gate CCZ através da composição das gates de Hadamard e CCX e, por fim, voltamos a aplicar as gates de Pauli-X e de Hadamard a todos os qubits, respetivamente.
Ou seja, ao aplicar a transformação dada por:

$$H^n(2|0\rangle\langle 0|I)H^n$$

Esta transformação roda o input desejado e aumenta a sua amplitude.

[14]:
```
# difusor
def difusor():
    qc.h(qr)
    qc.x(qr)
    qc.h(qr[2])
    qc.ccx(qr[0], qr[1], qr[2])
    qc.h(qr[2])
    qc.x(qr)
    qc.h(qr)

difusor()
qc.draw(output='mpl')
```

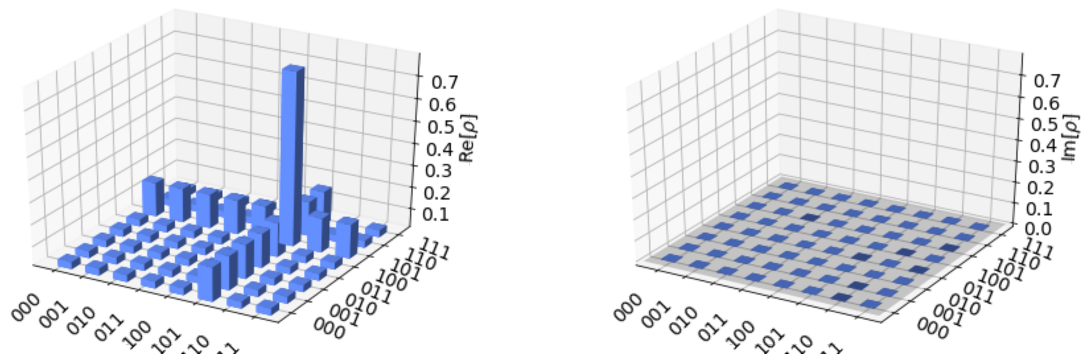
[14]:



[15]:
```
result = execute(qc, backend_vector).result()
qstate= result.get_statevector(qc)
print(qstate)
plot_state_city(qstate)
```

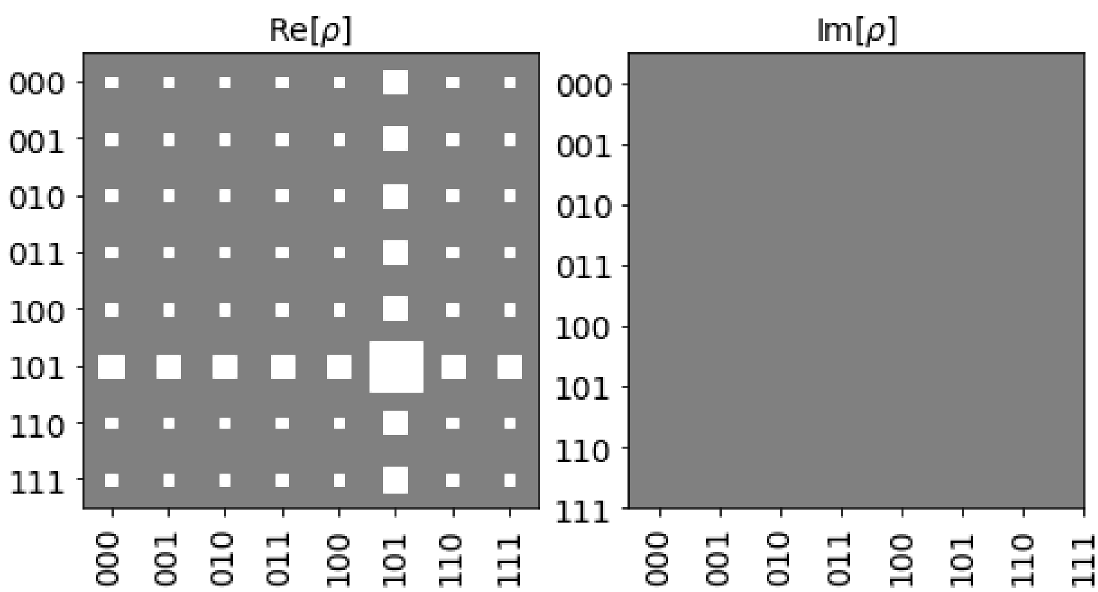
```
[-0.1767767 -2.16489014e-17j -0.1767767 +4.32978028e-17j
 -0.1767767 -3.07319371e-33j -0.1767767 +2.16489014e-17j
 -0.1767767 -1.65100229e-16j -0.88388348-2.40980805e-16j
 -0.1767767 -1.00153524e-16j -0.1767767 -1.65100229e-16j]
```

[15]:




```
[16]: plot_state_hinton(qstate)
```

[16]:

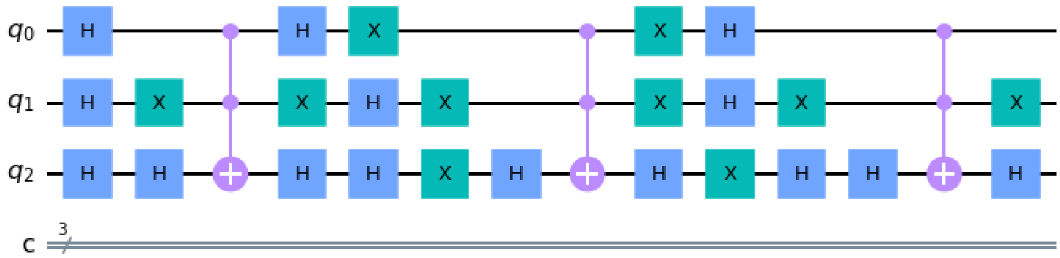


2. Voltamos a reperir o oraculo e o difusor, isto porque no algoritmo de Grover devemos repetir o oraculo e o difusor $\sqrt{D}$, onde D é tamanho do domínio, como neste caso a lista tem 8 elementos e $\sqrt{8} = 2$, repetimos o oraculo e o difusor mais uma vez.

   De forma a conseguirmos melhorar, ainda mais, o resultado.

```
[17]: oraculo()
      qc.draw(output='mpl')
```

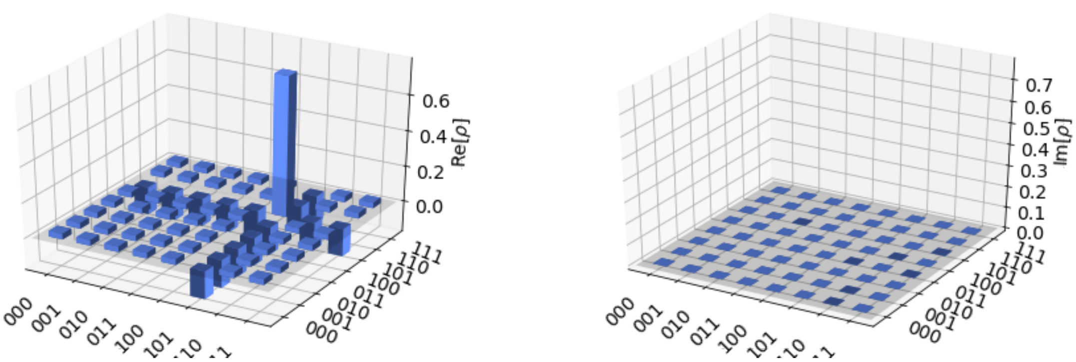
[17]:



[18]:
```
result = execute(qc, backend_vector).result()
qstate= result.get_statevector(qc)
print(qstate)
plot_state_city(qstate)
```

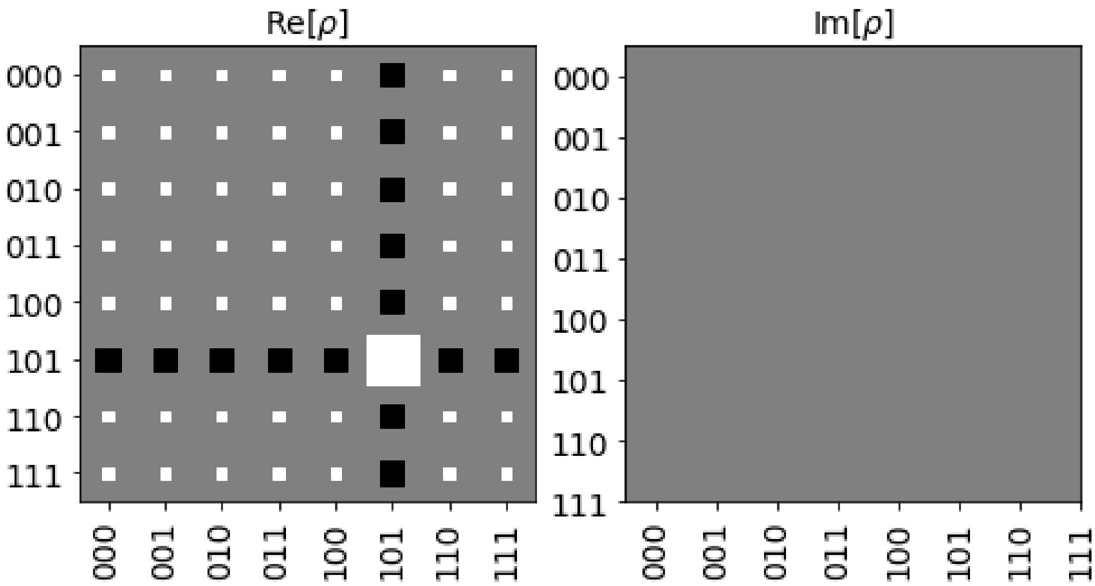
```
[-0.1767767 -4.32978028e-17j -0.1767767 +8.65956056e-17j
 -0.1767767 -2.16489014e-17j -0.1767767 -1.18630041e-32j
 -0.1767767 -1.29893408e-16j  0.88388348+4.32978028e-17j
 -0.1767767 -6.49467042e-17j -0.1767767 -8.65956056e-17j]
```

[18]:



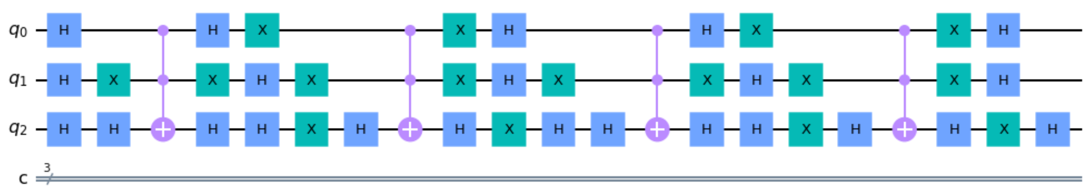
[19]:
```
plot_state_hinton(qstate)
```

[19]:



[20]:
```
difusor()
qc.draw(output='mpl')
```

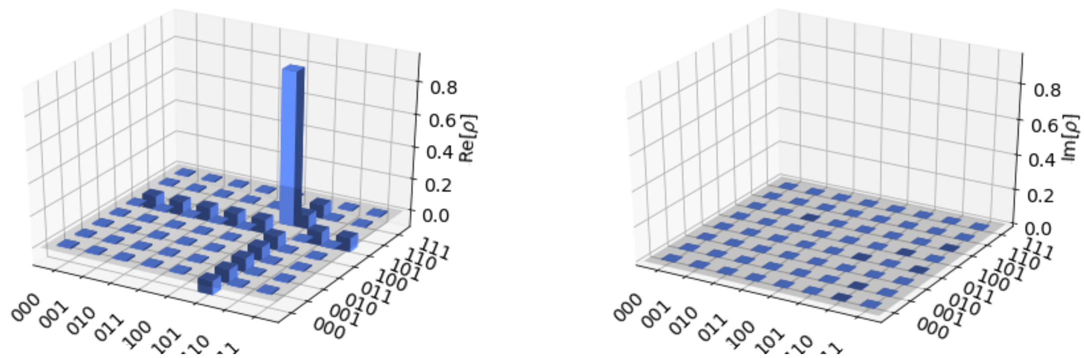
[20]:



[21]:
```
result = execute(qc, backend_vector).result()
qstate= result.get_statevector(qc)
print(qstate)
plot_state_city(qstate)
```

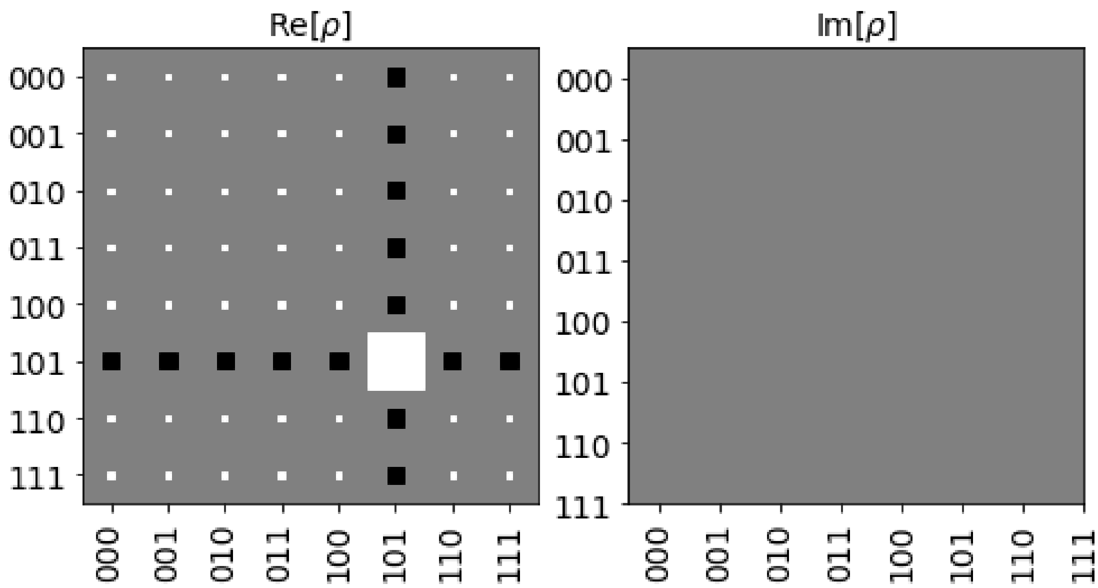
```
[-0.08838835-7.73659589e-33j -0.08838835+1.08244507e-17j
 -0.08838835+3.24733521e-17j -0.08838835+6.49467042e-17j
 -0.08838835-2.84278608e-17j  0.97227182+3.66828722e-16j
 -0.08838835-1.76034101e-17j -0.08838835+1.01465548e-16j]
```

[21]:



[22]:
```
plot_state_hinton(qstate)
```

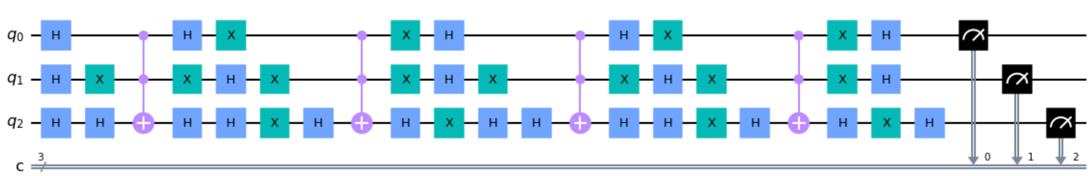
[22]:



3. Medimos os valores dos qubits.

[23]:
```
qc.measure(qr,cr)

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

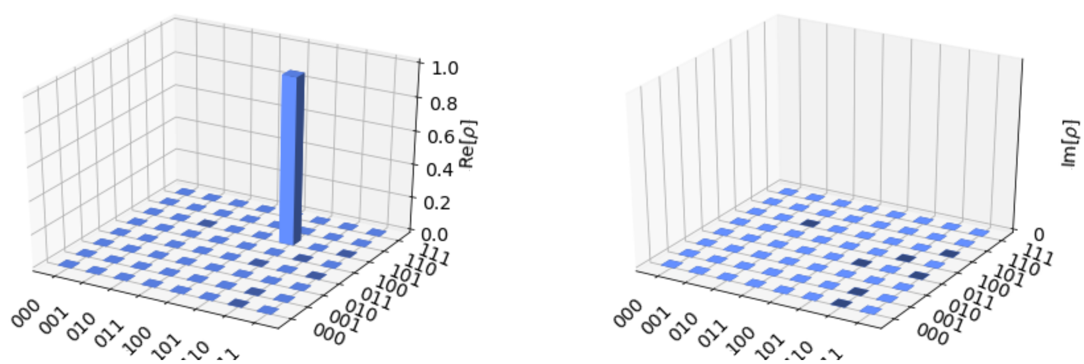
[23]:



[24]:
```
result = execute(qc, backend_vector).result()
qstate= result.get_statevector(qc)
print(qstate)
plot_state_city(qstate)
```

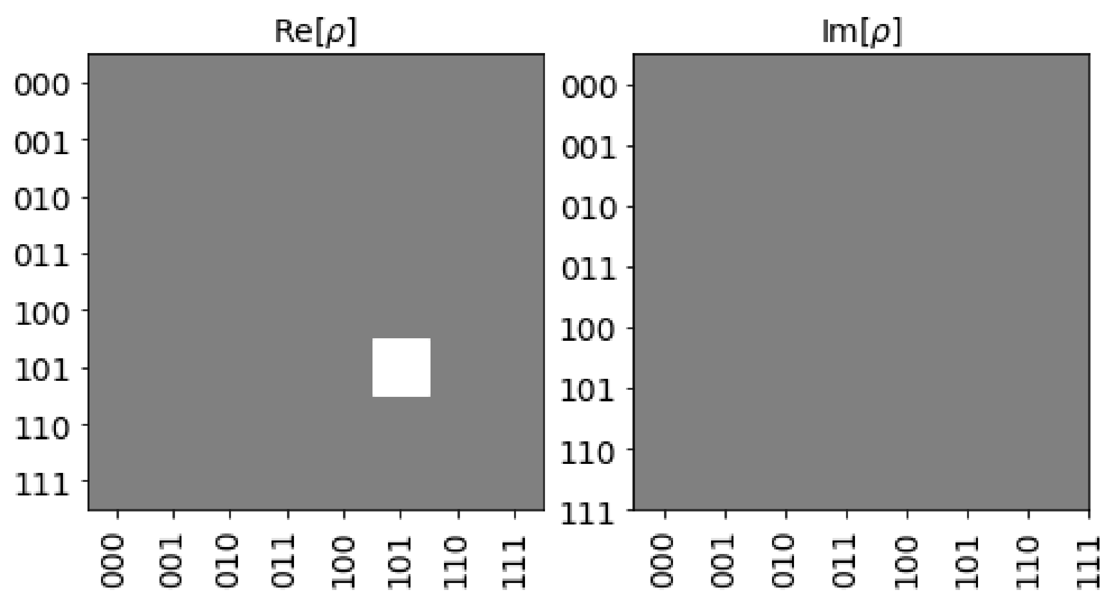
```
[ 0.+0.00000000e+00j -0.+0.00000000e+00j -0.+0.00000000e+00j
 -0.+0.00000000e+00j  0.+0.00000000e+00j  1.+3.77290294e-16j
  0.+0.00000000e+00j -0.+0.00000000e+00j]
```

[24]:



[25]:
```
plot_state_hinton(qstate)
```

[25]:



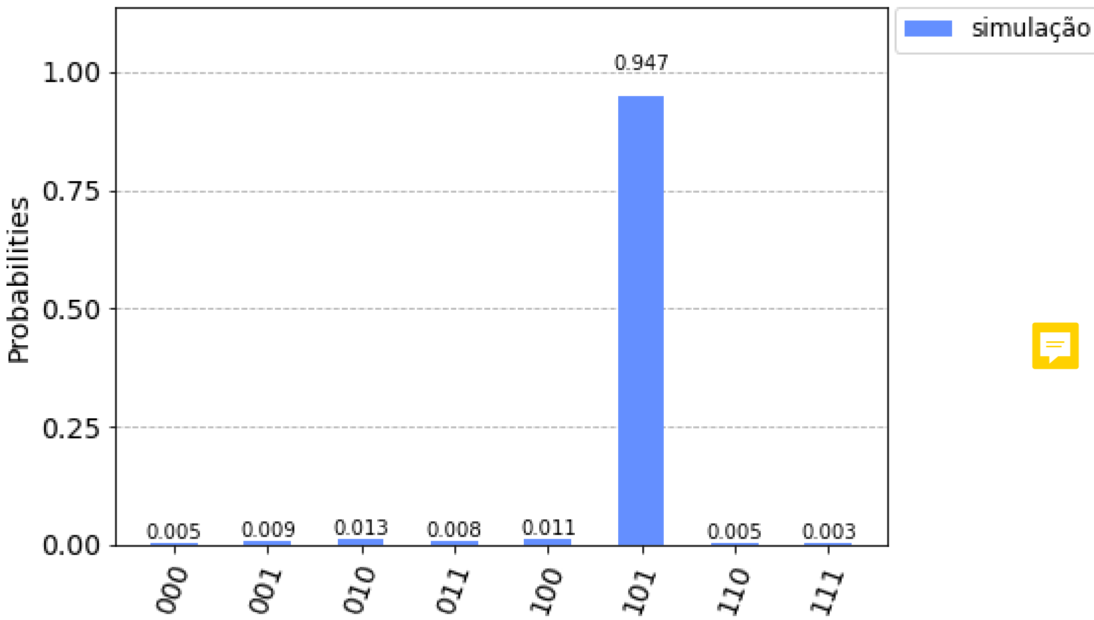
[26]:
```
qc.depth()
```

[26]: 22

Simulamos agora o resultado, obtendo o resultado ideal.

[27]:
```
shots=1024
result = execute(qc, backend, shots=shots).result()
counts_sim = result.get_counts(qc)
plot_histogram(counts_sim , legend=['simulação'])
```

[27]:



1.2.2 2) Aplicação do simulador de ruido para prever a melhor otimisação

[28]:
```
provider = IBMQ.load_account()
provider.backends()
```

[28]:
```
[<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')>]
```

Escolhemos um servidor que continha a média de erros mais baixas para as operações que pretendemos realizar.

[29]:
```
import qiskit.tools.jupyter
%qiskit_backend_overview
```

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

[30]:
```
from qiskit.tools.monitor import backend_overview, backend_monitor
backend_overview()
```

```
ibmq_manila                 ibmq_quito                  ibmq_belem
-----------                 ----------                  ----------
Num. Qubits:  5             Num. Qubits:  5             Num. Qubits:  5
Pending Jobs: 3             Pending Jobs: 5             Pending Jobs: 6
Least busy:   False         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: 12            Pending Jobs: 5             Pending Jobs: 0
Least busy:   False         Least busy:   False         Least busy:   True
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: 19            Pending Jobs: 3             Pending Jobs: 1
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
```

[31]:
```python
backend_device = provider.get_backend('ibmq_santiago')
print("Running on: ", backend_device)
```

```
Running on:  ibmq_santiago
```

[32]:
```python
backend_monitor(backend_device)
```

```
ibmq_santiago
=============
Configuration
-------------
    n_qubits: 5
    operational: True
    status_msg: active
    pending_jobs: 5
    backend_version: 1.3.22
    basis_gates: ['id', 'rz', 'sx', 'x', 'cx', 'reset']
    local: False
    simulator: False
    input_allowed: ['job']
    n_registers: 1
    allow_q_object: True
    parametric_pulses: ['gaussian', 'gaussian_square', 'drag', 'constant']
    conditional_latency: []
    online_date: 2020-06-03 04:00:00+00:00
    meas_lo_range: [[6.952624018e+18, 7.952624018e+18], [6.701014434e+18,
7.701014434e+18], [6.837332258e+18, 7.837332258e+18], [6.901770712e+18,
7.901770712e+18], [6.775814414e+18, 7.775814414e+18]]
    multi_meas_enabled: True
    meas_map: [[0, 1, 2, 3, 4]]
    max_shots: 8192
    meas_levels: [1, 2]
    pulse_num_channels: 9
    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'}}
    n_uchannels: 8
    qubit_channel_mapping: [['u1', 'd0', 'u0', 'm0'], ['d1', 'u3', 'm1', 'u2',
'u1', 'u0'], ['u4', 'd2', 'u3', 'u2', 'm2', 'u5'], ['u4', 'd3', 'u7', 'm3',
'u5', 'u6'], ['d4', 'm4', 'u6', 'u7']]
    rep_delay_range: [0.0, 500.0]
    meas_kernels: ['hw_qmfk']
    pulse_num_qubits: 3
    memory: True
    allow_object_storage: True
    backend_name: ibmq_santiago
    acquisition_latency: []
    uchannels_enabled: True
    qubit_lo_range: [[4.33342839657397e+18, 5.333428396573969e+18],
[4.1236103027229476e+18, 5.123610302722947e+18], [4.3205309850357484e+18,
5.320530985035748e+18], [4.242308922763805e+18, 5.242308922763806e+18],
[4.316322038954131e+18, 5.316322038954131e+18]]
    rep_times: [0.001]
    discriminators: ['quadratic_discriminator', 'hw_qmfk',
'linear_discriminator']
    dynamic_reprate_enabled: True
    credits_required: True
    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_{3,4}(\\sigma_{3}^{+}\\sigma_{4}^{-}+\\sigma_{3}^{-}\\sigma_{4}^{+}) +
J_{2,3}(\\sigma_{2}^{+}\\sigma_{3}^{-}+\\sigma_{2}^{-}\\sigma_{3}^{+}) +
J_{1,2}(\\sigma_{1}^{+}\\sigma_{2}^{-}+\\sigma_{1}^{-}\\sigma_{2}^{+}) \\\\ & +
\\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', 'jq3q4*Sp3*Sm4', 'jq3q4*Sm3*Sp4',
'jq2q3*Sp2*Sm3', 'jq2q3*Sm2*Sp3', 'jq1q2*Sp1*Sm2', 'jq1q2*Sm1*Sp2',
'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':
-2.1481278490714906, 'delta1': -2.0623435150768743, 'delta2':
-2.1429828509850863, 'delta3': -2.137118237032298, 'delta4': -2.154596484455155,
'jq0q1': 0.007378105608801839, 'jq1q2': 0.007268700678758498, 'jq2q3':
0.007255936195908655, 'jq3q4': 0.006881064755295536, 'omegad0':
1.011137872642343, 'omegad1': 0.9860187056541215, 'omegad2': 1.0018026333654275,
'omegad3': 1.0073346201781475, 'omegad4': 1.0008689448135097, 'wq0':
30.369326284658154, 'wq1': 29.051000320192983, 'wq2': 30.288289457980554, 'wq3':
29.796805745616194, 'wq4': 30.261843869801826}}
    quantum_volume: 32
    supported_instructions: ['cx', 'setf', 'sx', 'delay', 'u3', 'play',
'measure', 'id', 'u2', 'reset', 'rz', 'acquire', 'shiftf', 'u1', 'x']
    description: 5 qubit device
    coupling_map: [[0, 1], [1, 0], [1, 2], [2, 1], [2, 3], [3, 2], [3, 4], [4,
3]]
    max_experiments: 75
    url: None
    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)}]]
```

```
    open_pulse: False
    dt: 0.2222222222222222
    sample_name: family: Falcon, revision: 4, segment: L
    dtm: 0.2222222222222222
    processor_type: {'family': 'Falcon', 'revision': 4, 'segment': 'L'}
    conditional: False
    default_rep_delay: 250.0

Qubits [Name / Freq / T1 / T2 / RZ err / SX err / X err / Readout err]
----------------------------------------------------------------------
    Q0 / 4.83343 GHz / 122.26194 us / 240.32302 us / 0.00000 / 0.00027 / 0.00027
/ 0.01770
    Q1 / 4.62361 GHz / 123.08792 us / 108.48683 us / 0.00000 / 0.00016 / 0.00016
/ 0.00970
    Q2 / 4.82053 GHz / 140.37440 us / 97.48492 us / 0.00000 / 0.00022 / 0.00022
/ 0.01010
    Q3 / 4.74231 GHz / 179.30549 us / 98.64605 us / 0.00000 / 0.00018 / 0.00018
/ 0.00480
    Q4 / 4.81632 GHz / 115.98140 us / 136.96134 us / 0.00000 / 0.00044 / 0.00044
/ 0.01720

Multi-Qubit Gates [Name / Type / Gate Error]
--------------------------------------------
    cx4_3 / cx / 0.00610
    cx3_4 / cx / 0.00610
    cx2_3 / cx / 0.00567
    cx3_2 / cx / 0.00567
    cx2_1 / cx / 0.00592
    cx1_2 / cx / 0.00592
    cx0_1 / cx / 0.00610
    cx1_0 / cx / 0.00610
```

[33]:
```
# See backend information
backend_device
```

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

[33]:
```
<IBMQBackend('ibmq_santiago') from IBMQ(hub='ibm-q', group='open',
project='main')>
```

Simulação com ruido

```
Com o NoiseModel, é possível construir um modelo de ruído aproximado
consistindo em:
* erros de gates de um qubit
* erros de gates de dois qubit
* erros de leitura de um qubit
```

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

[35]:
```
from qiskit.providers.aer.noise import NoiseModel
```

[36]:
```
noise_model = NoiseModel.from_backend(backend_device)
print(noise_model)
```

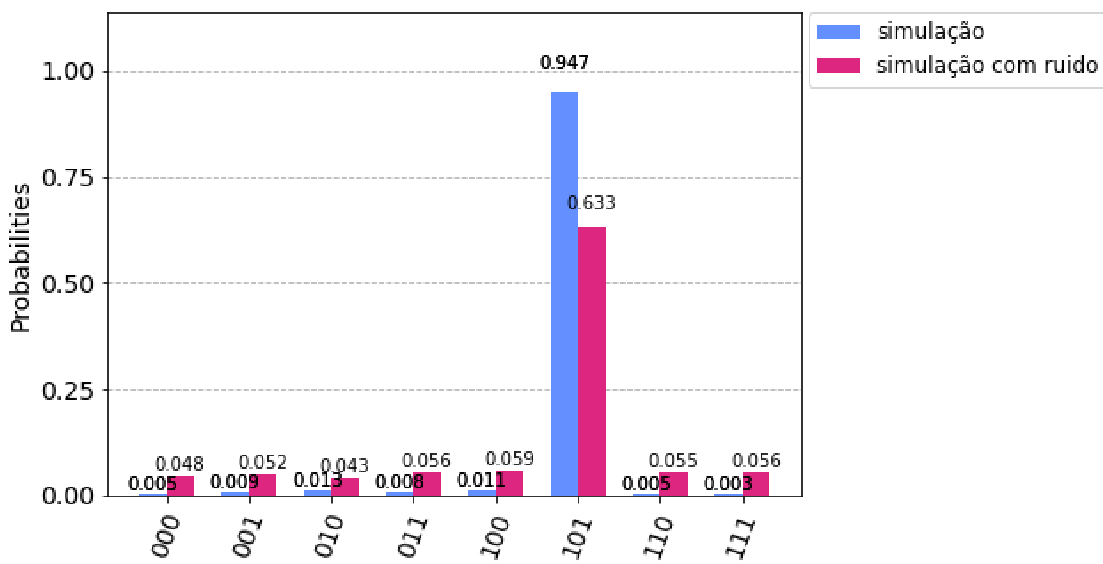
```
NoiseModel:
  Basis gates: ['cx', 'id', 'reset', 'rz', 'sx', 'x']
  Instructions with noise: ['id', 'measure', 'cx', 'reset', 'sx', 'x']
  Qubits with noise: [0, 1, 2, 3, 4]
  Specific qubit errors: [('id', [0]), ('id', [1]), ('id', [2]), ('id', [3]),
('id', [4]), ('sx', [0]), ('sx', [1]), ('sx', [2]), ('sx', [3]), ('sx', [4]),
('x', [0]), ('x', [1]), ('x', [2]), ('x', [3]), ('x', [4]), ('cx', [4, 3]),
('cx', [3, 4]), ('cx', [2, 3]), ('cx', [3, 2]), ('cx', [2, 1]), ('cx', [1, 2]),
('cx', [0, 1]), ('cx', [1, 0]), ('reset', [0]), ('reset', [1]), ('reset', [2]),
('reset', [3]), ('reset', [4]), ('measure', [0]), ('measure', [1]), ('measure',
[2]), ('measure', [3]), ('measure', [4])]
```

[37]:
```
basis_gates = noise_model.basis_gates
print(basis_gates)
```

```
['cx', 'id', 'reset', 'rz', 'sx', 'x']
```

[38]:
```
# Execute noisy simulation and get counts
result_noise = execute(qc, backend,
                       noise_model=noise_model,
                       coupling_map=coupling_map,
                       basis_gates=basis_gates).result()
counts_noise = result_noise.get_counts(qc)
plot_histogram([ counts_sim ,counts_noise], legend=[ 'simulação', 'simulação com␣
 ↪ruido'])
```

[38]:

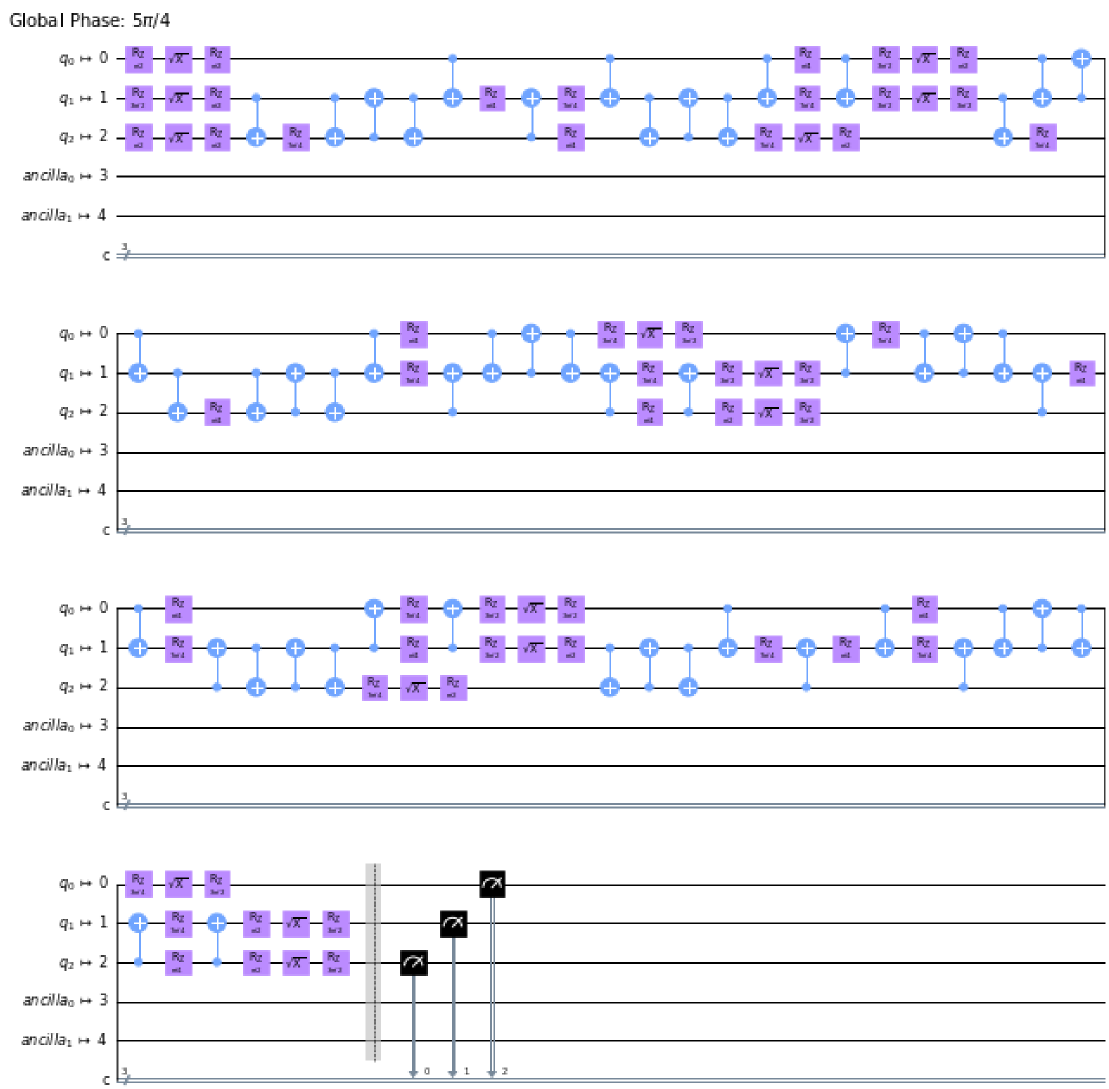


Otimização

[39]:
```
from qiskit.compiler import transpile

qc_t_real = transpile(qc, backend=backend_device)
print(qc_t_real.depth())
qc_t_real.draw(output='mpl', scale=0.5)
```

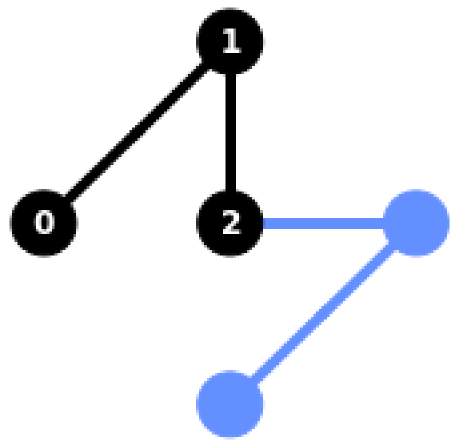
82

[39]:



[40]:
```
from qiskit.visualization import plot_circuit_layout
plot_circuit_layout(qc_t_real, backend_device)
```

[40]:

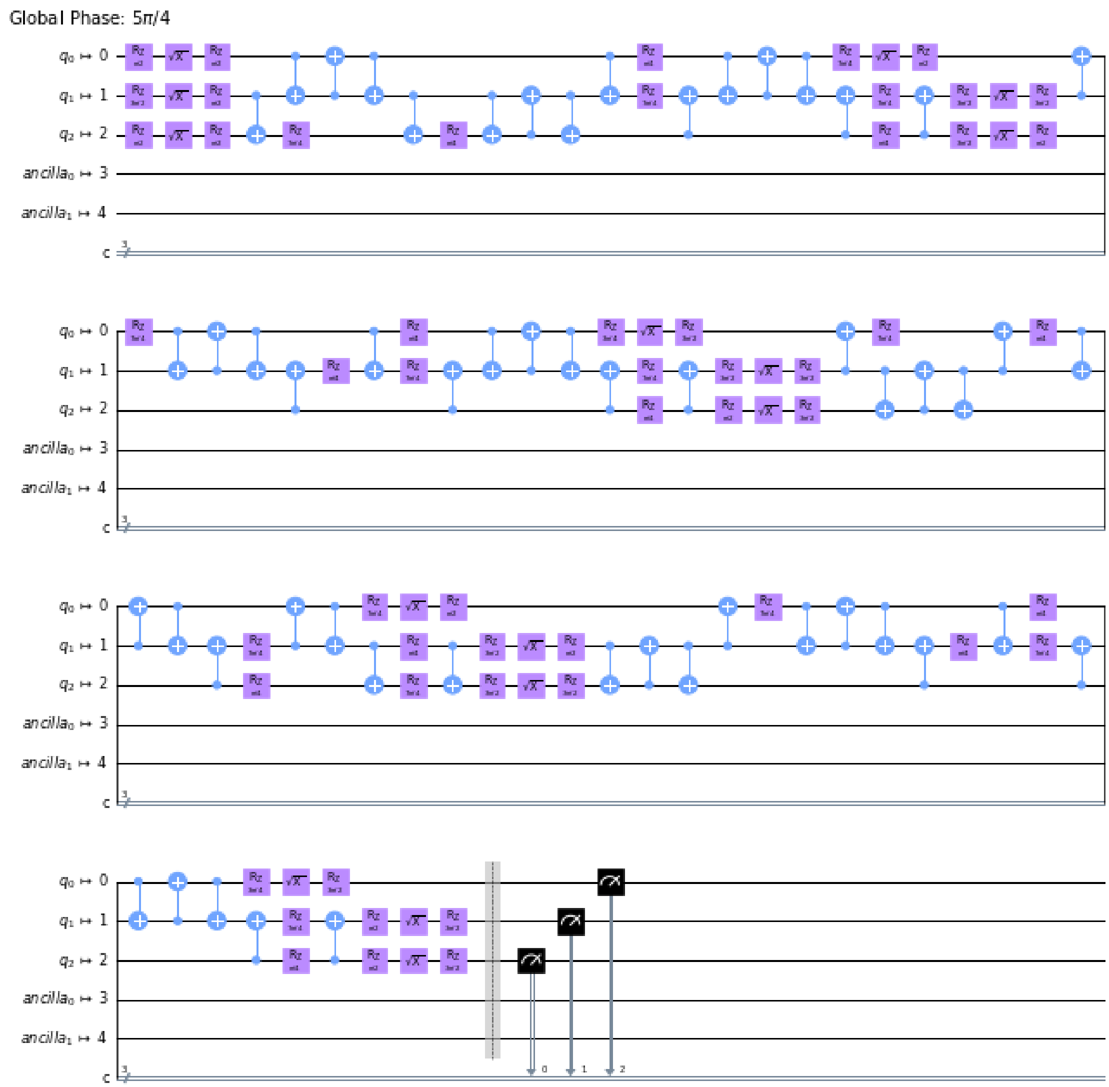


Otimização de nivel 1

[41]:
```
qc_optimized_1 = transpile(qc, backend=backend_device, optimization_level=1)
print(qc_optimized_1.depth())
qc_optimized_1.draw(output='mpl', scale=0.5)
```

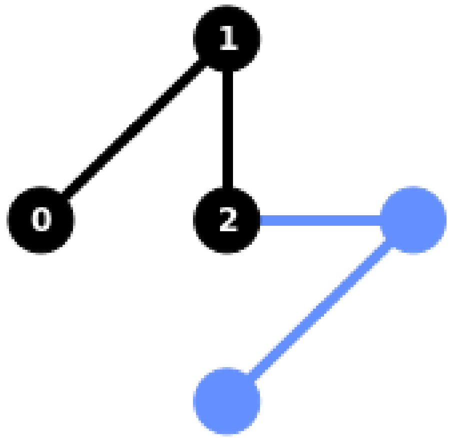
85

[41]:



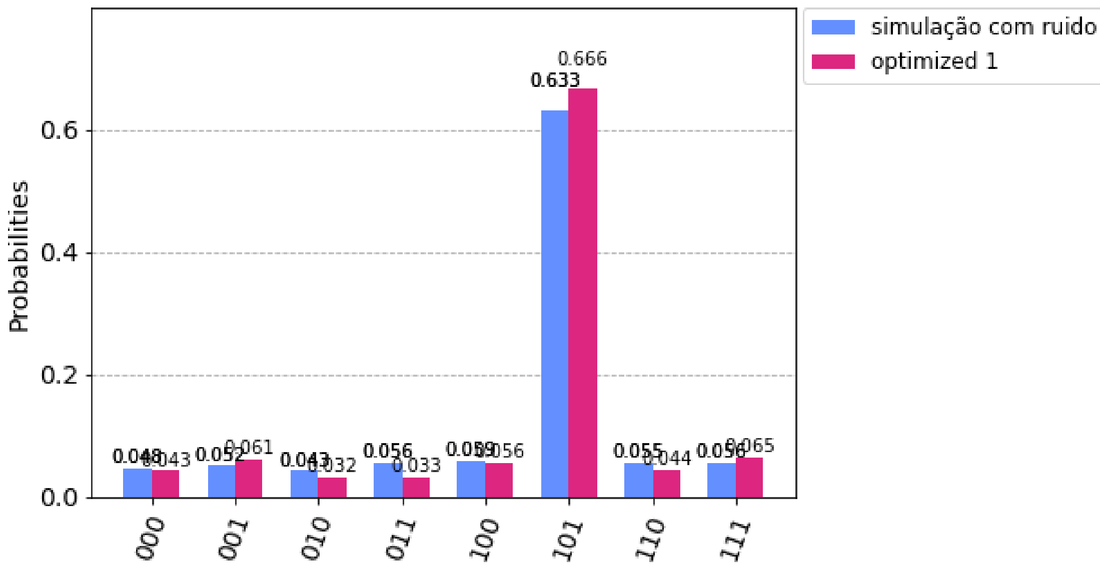
```
[42]: plot_circuit_layout(qc_optimized_1, backend_device)
```

```
[42]:
```



[43]:
```
result_optimized_1 = execute(qc_optimized_1, backend,
                        noise_model=noise_model,
                        coupling_map=coupling_map,
                        basis_gates=basis_gates).result()

counts_optimized_1 = result_optimized_1.get_counts(qc_optimized_1)

plot_histogram([counts_noise,counts_optimized_1], legend=['simulação com␣
 ↪ruido','optimized 1'])
```

[43]:

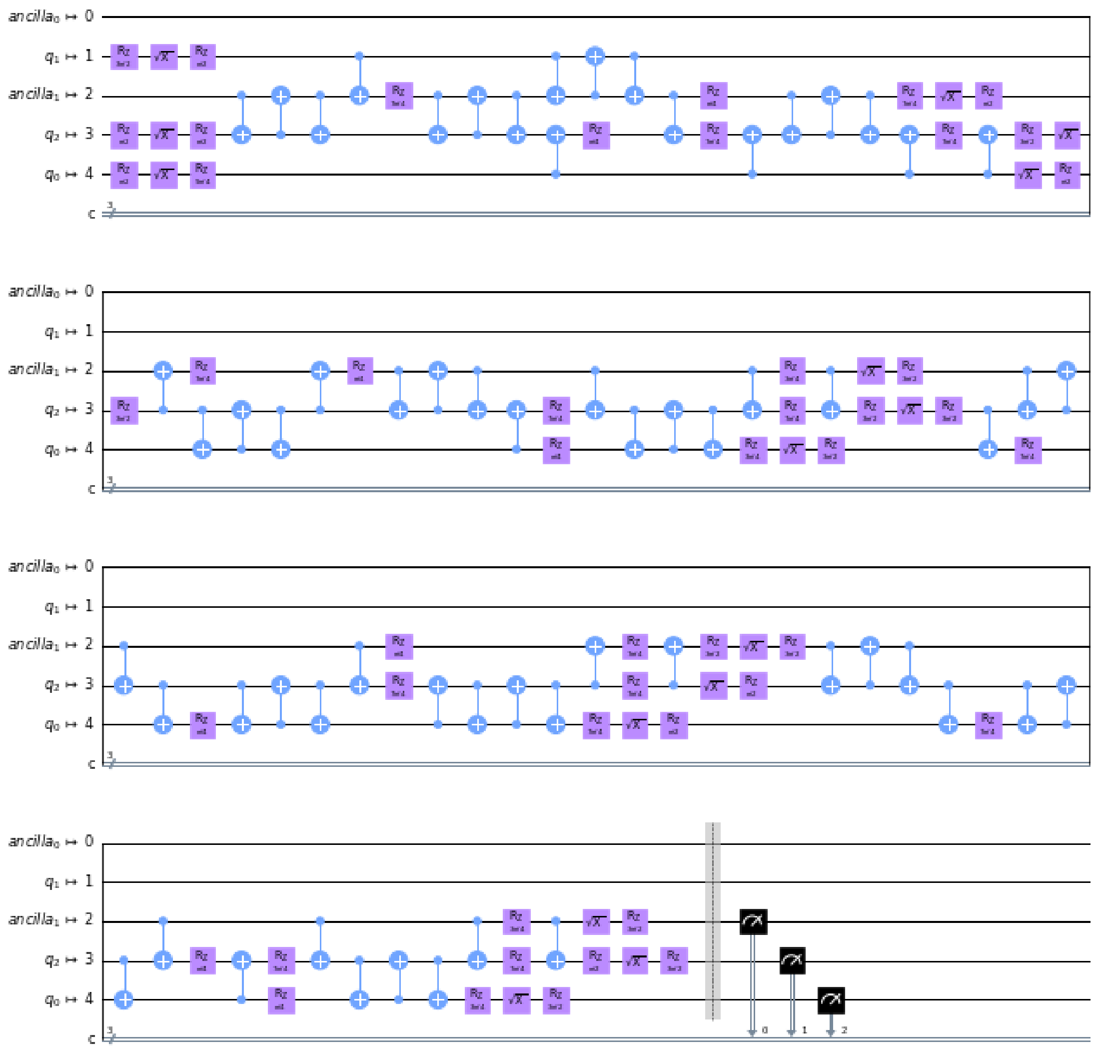


Otimização de nivel 2

[44]:
```
qc_optimized_2 = transpile(qc, backend=backend_device, optimization_level=2)
print(qc_optimized_2.depth())
qc_optimized_2.draw(output='mpl', scale=0.5)
```

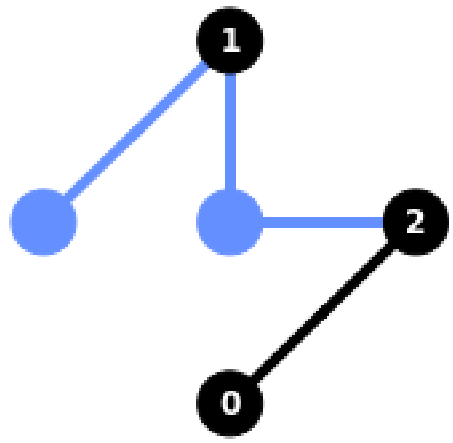
```
91
```

[44]:



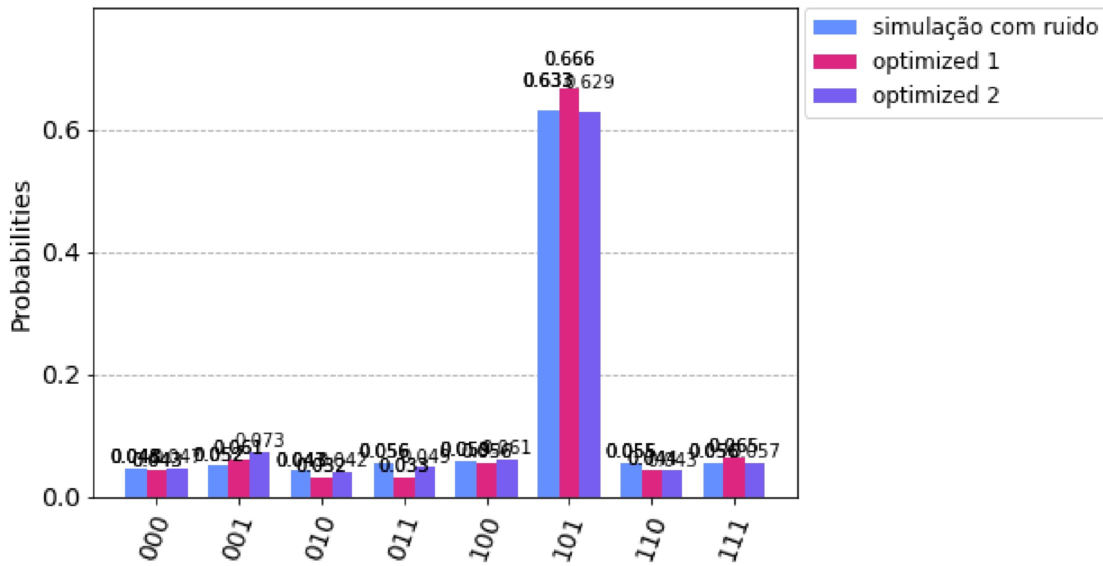
[45]:
```
plot_circuit_layout(qc_optimized_2, backend_device)
```

[45]:



[46]:
```
result_optimized_2 = execute(qc_optimized_2, backend,
                        noise_model=noise_model,
                        coupling_map=coupling_map,
                        basis_gates=basis_gates).result()

counts_optimized_2 = result_optimized_2.get_counts(qc_optimized_2)
plot_histogram([counts_noise,counts_optimized_1,counts_optimized_2], legend=[
 ↪'simulação com ruido','optimized 1','optimized 2'])
```

[46]:

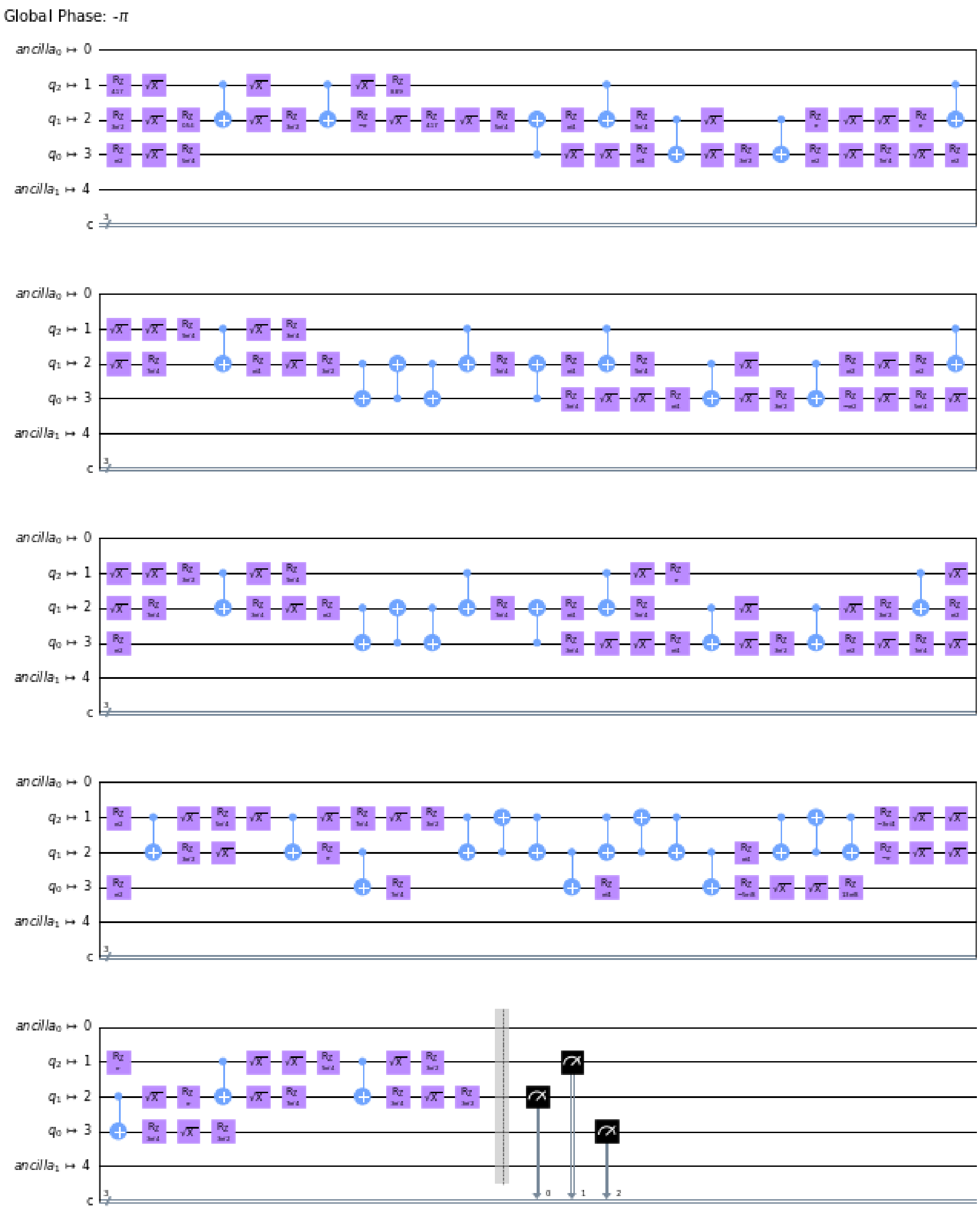


Otimização de nivel 3

[47]:
```
qc_optimized_3 = transpile(qc, backend=backend_device, optimization_level=3)
print(qc_optimized_3.depth())
qc_optimized_3.draw(output='mpl', scale=0.5)
```

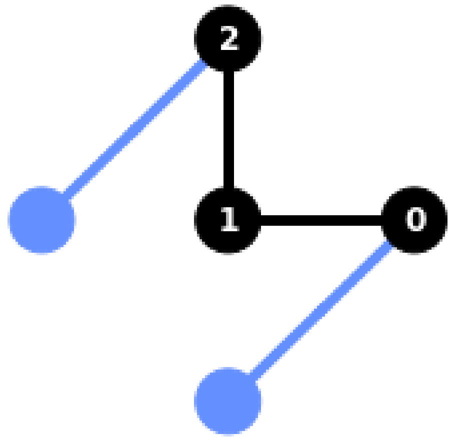
112

[47]:



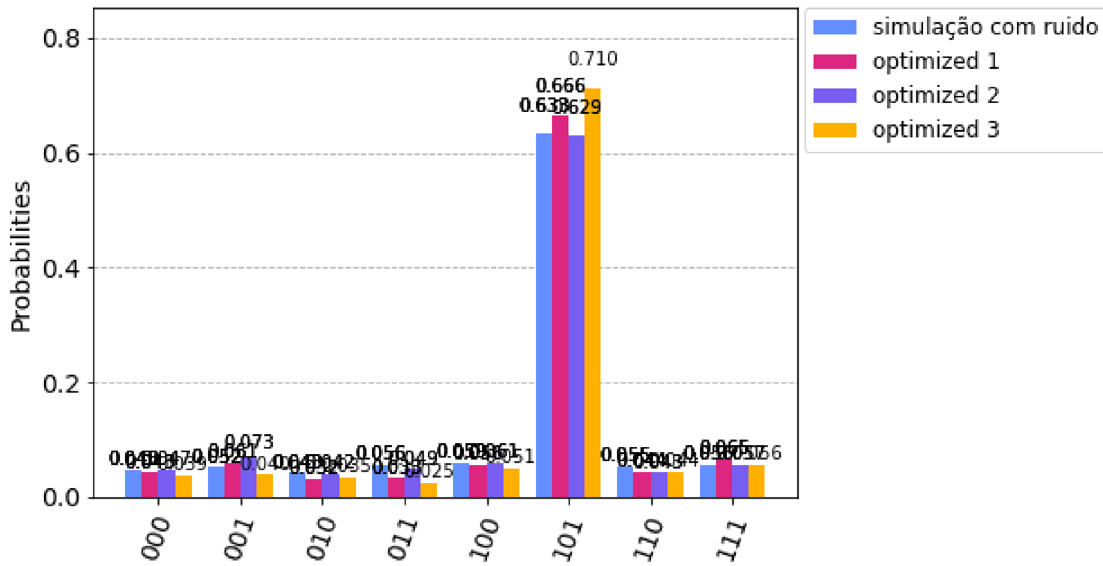
[48]:
```
plot_circuit_layout(qc_optimized_3, backend_device)
```

[48]:



[49]:
```
result_optimized_3 = execute(qc_optimized_3, backend,
                       noise_model=noise_model,
                       coupling_map=coupling_map,
                       basis_gates=basis_gates).result()

counts_optimized_3 = result_optimized_3.get_counts(qc_optimized_3)
plot_histogram([counts_noise,counts_optimized_1,counts_optimized_2,counts_optimized_3],␣
 ↪legend=['simulação com ruido','optimized 1','optimized 2','optimized 3'])
```

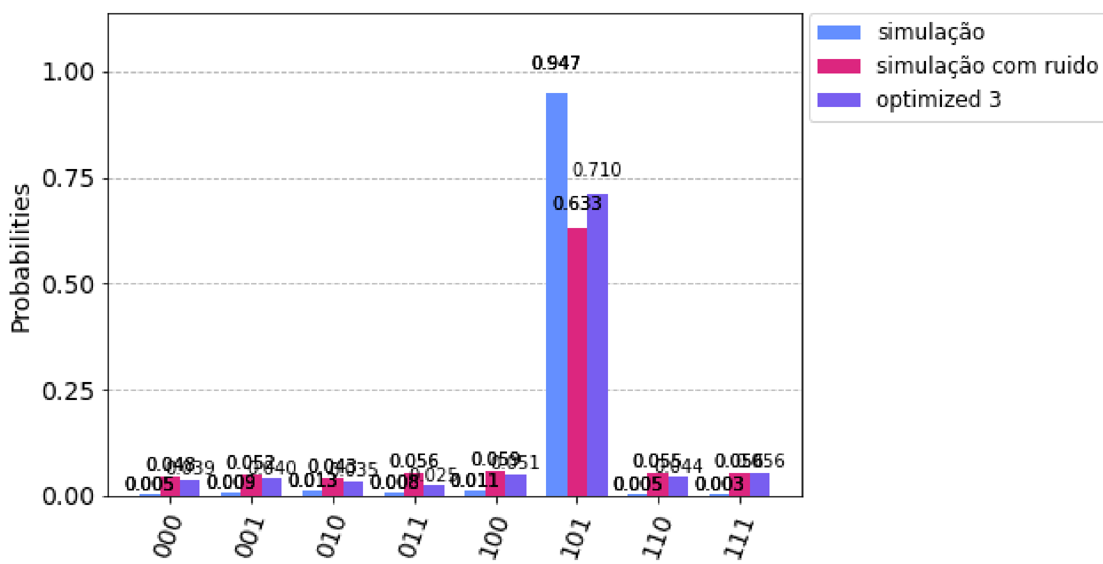
[49]:



Podemos ver que a otimização de nivel 3 apesar de realizar mais operações é a otimização com melhores resultados.

[50]:
```
plot_histogram([ counts_sim ,counts_noise,counts_optimized_3], legend=[␣
↪'simulação', 'simulação com ruido','optimized 3'])
```

[50]:



1.2.3 3) Execução em uma IBM Q backend.

[51]:
```
%qiskit_job_watcher
```

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

```
&lt;IPython.core.display.Javascript object&gt;
```

[52]:
```
job_r = execute(qc, backend_device, shots=shots)

jobID_r = job_r.job_id()

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

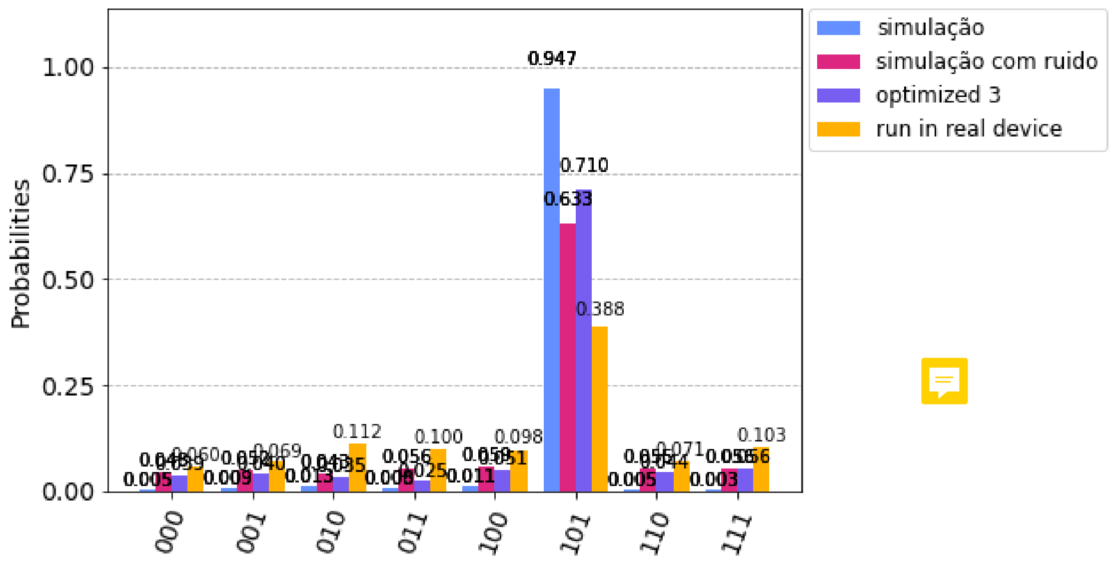
```
JOB ID: 60bbc406917aa0cbfc9b6f11
```

[53]:
```
job_get=backend_device.retrieve_job(jobID_r)

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

Comparando agora os resultados obtidos:

[54]:
```
plot_histogram([counts_sim, counts_noise,counts_optimized_3,counts_run  ],␣
↪legend=[ 'simulação', 'simulação com ruido','optimized 3','run in real device'␣
↪])
```

[54]:



1.2.4 4) Mitigação dos erros com Ignis.

[55]:
```
# Import measurement calibration functions
from qiskit.ignis.mitigation.measurement import (complete_meas_cal,␣
 ↪tensored_meas_cal,
                                                 CompleteMeasFitter,␣
 ↪TensoredMeasFitter)
```

Matrizes de calibração
>Geramos a lista de circuitos de calibração de medição.
>Cada circuito cria um estado básico.
>Uma vez que medimos 3 qubits, precisamos de $ 2^{3} = 8 $ circuitos de calibração.

[56]:
```
# 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=[0,1,2], qr=qr,␣
 ↪circlabel='mcal')
```

[57]:
```
state_labels
```

[57]: ['000', '001', '010', '011', '100', '101', '110', '111']

Calculamos a matriz de calibração
Se não houvesse ruído no dispositivo, a matriz de calibração seria a matriz identidade 8×8. Como calculamos essa matriz com um dispositivo quântico real, existe algum ruído.

[58]:
```
job_ignis = execute(meas_calibs, backend=backend_device, shots=shots)

jobID_run_ignis = job_ignis.job_id()

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

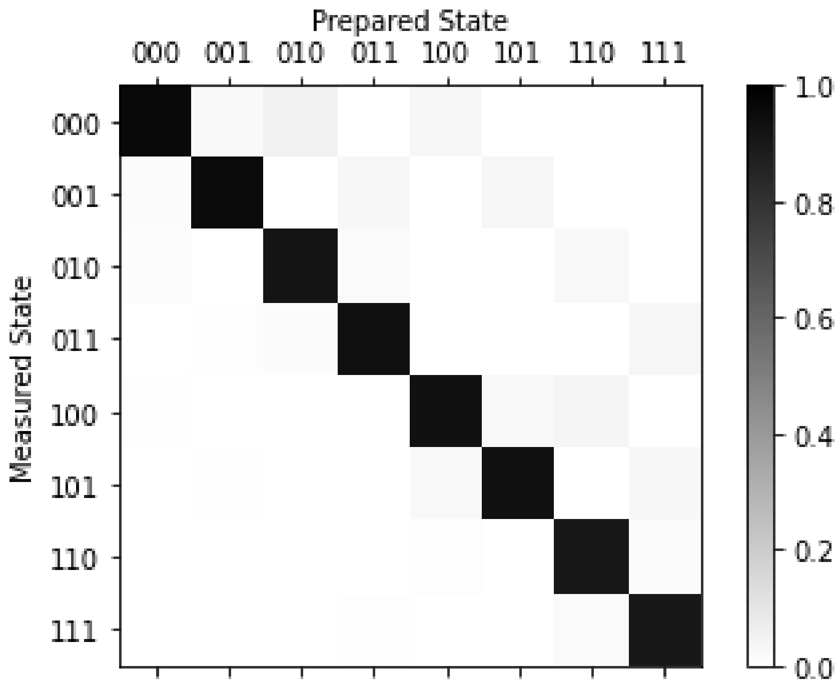
JOB ID: 60bbc44200adedebcd6a70c5

[59]:
```
job_get=backend_device.retrieve_job(jobID_run_ignis)

cal_results = job_get.result()
```

[60]:
```
%qiskit_disable_job_watcher
```

[61]:
```
meas_fitter = CompleteMeasFitter(cal_results, state_labels, circlabel='mcal')

# Plot the calibration matrix
```

```
meas_fitter.plot_calibration()
```



Análise dos resultados
A fidelidade de atribuição média é o traço da matriz anterior.

[62]:
```
print("Average Measurement Fidelity: %f" % meas_fitter.readout_fidelity())
```

Average Measurement Fidelity: 0.929077

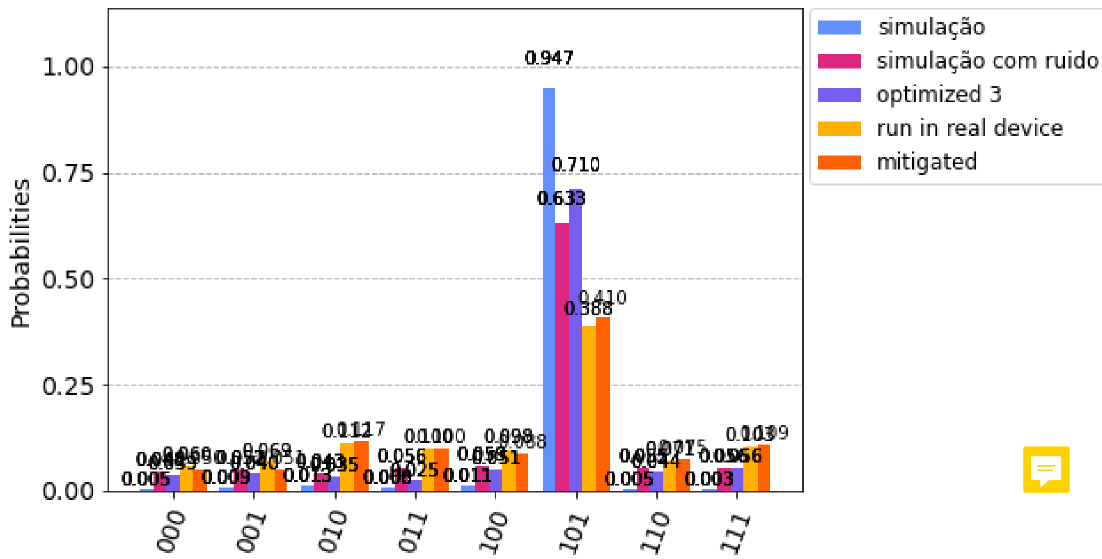
Aplicamos a calibração
Aplicamos um filtro baseado na matriz de calibração para obter a contagem mitigada.

[63]:
```
# Get the filter object
meas_filter = meas_fitter.filter

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

E, por fim, voltamos a comparar os resultados todos:

[64]:
```
plot_histogram([counts_sim ,counts_noise,counts_optimized_3, counts_run,␣
↪mitigated_counts ], legend=['simulação','simulação com ruido','optimized␣
↪3','run in real device', 'mitigated'])
```

[64]:



Podemos ver que a mitigação dos erros melhora ligeiramente o resultado obtido.

1.3 Conclusão:

Concluímos então este trabalho prático relativamente à UC Interação e Concorrência.

Achamos que os objetivos foram bem alcançados e em que foi, sem dúvida, uma boa forma de colocar os conceitos em prática, conceitos estes que ao início poderiam estar um pouco mais "enferrujados" e que acabaram por ser polidos e agora achamos que finalizamos da melhor forma!

```
[ ]:
```