EURO^{2}

Introduction to NVIDIA CUDA Quantum

Jiří Tomčala

IT4Innovations,
VŠB - Technical University of Ostrava

7 November 2024

Part I

Request to gain access to IonQ hardware

Request to gain access to IonQ hardware

- Go to https://cloud.ionq.com

Request to gain access to IonQ hardware

- Go to https://cloud.ionq.com
- Click on "Get started for free"

Request to gain access to IonQ hardware

- Go to https://cloud.ionq.com
- Click on "Get started for free"
- Fill out the form

Request to gain access to IonQ hardware

- Go to https://cloud.ionq.com
- Click on "Get started for free"
- Fill out the form
- Verify your email address

Request to gain access to IonQ hardware

- Go to https://cloud.ionq.com
- Click on "Get started for free"
- Fill out the form
- Verify your email address
- Wait for the welcome email (this may take some time)

Request to gain access to IonQ hardware

- Go to https://cloud.ionq.com
- Click on "Get started for free"
- Fill out the form
- Verify your email address
- Wait for the welcome email (this may take some time)
- Log in to https://cloud.ionq.com

Request to gain access to IonQ hardware

- Go to https://cloud.ionq.com
- Click on "Get started for free"
- Fill out the form
- Verify your email address
- Wait for the welcome email (this may take some time)
- Log in to https://cloud.ionq.com
- Generate API key

Request to gain access to IonQ hardware

- Go to https://cloud.ionq.com
- Click on "Get started for free"
- Fill out the form
- Verify your email address
- Wait for the welcome email (this may take some time)
- Log in to https://cloud.ionq.com
- Generate API key
- Insert your generated API key into your system:

```
export IONQ_API_KEY="ionq_generated_api_key"
```

Part II

Installation

Local installation

Python 3.10 is recommended for the current version of NVIDIA CUDA Quantum simulator. It is also recommended to use Conda to create a separate environment for this simulator. If you are not already using Conda, you can install a minimal version (miniconda) following the instructions here: https://docs.anaconda.com/miniconda/

Once you have Conda installed, you can create an environment with the required version of Python and other software using the command:

```
conda create -y -n cuda-quantum python=3.10 pip notebook
```

You can then install the simulator itself using the command:

```
conda run -n cuda-quantum pip install cuda-quantum
```

You can then activate the created environment as follows:

```
conda activate cuda-quantum
```

Now you are ready to program the simulator using Python scripts or Jupyter notebooks. This procedure is for installation on your PC for testing smaller examples on local CPUs.

Installation on the Cluster

Load the Conda module with the command:

```
ml Anaconda3
```

Create an environment with the required version of Python and other software using the command:

```
conda create -y -n cuda-quantum python=3.10 pip
```

You can then install the simulator itself using the command:

```
conda run -n cuda-quantum pip install cuda-quantum
```

You can then activate the created environment as follows:

```
conda activate cuda-quantum
```

Now you are ready to program the simulator on the Cluster using Python scripts. This procedure can be used to install NVIDIA CUDA Quantum to test examples on CPUs or GPUs that reside on a single compute node, which will be sufficient for the purposes of this course.

Part III

Visualization

Qubit visualization

What are the possible states a qubit can be in and how can we build up a visual cue to help us make sense of quantum states and their evolution?

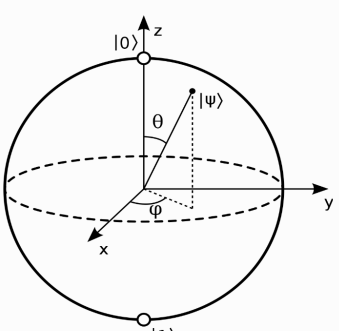
We know our qubit can have two distinct states: $|0\rangle$ and $|1\rangle$. If these were the only two states, we could represent them as two vectors on a one-dimensional line (i.e., the z-axis in the image below). We also know that qubits can be in an equal superposition of states: $|+\rangle$ and $|-\rangle$. In order to capture all of the states in equal superposition, we will need a 2D plane (i.e., the xy-plane in the image below). If you dive deeper you will learn about the existence of other states that will call for a 3D extension.

In general, a quantum state can be written in the form $|\psi\rangle = \cos(\frac{\theta}{2})|0\rangle + e^{i\varphi}\sin(\frac{\theta}{2})|1\rangle$ where θ is a real number between 0 and π and φ is a real value between 0 and 2π. For example, the minus state, $|-\rangle = \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle$, can be rewritten as

$$|-\rangle = \cos(\frac{\theta}{2})|0\rangle + e^{i\varphi}\sin(\frac{\theta}{2})|1\rangle \text{ with } \theta = \frac{\pi}{2} \text{ and } \varphi = \pi.$$

This can be visualized in the image below as a unit vector pointing in the direction of the negative x-axis.

Using spherical coordinates, it is possible to depict all the possible states of a single qubit on a sphere. This is called a Bloch sphere.



Qubit visualization

Let us try to showcase the functionality to render such a 3D representation with CUDA-Q. First, let us define a single-qubit kernel that returns a different state each time. This kernel uses random rotations.

Note: CUDA-Q uses the QuTiP library to render Bloch spheres. The following code will throw an error if QuTiP is not installed.

[13]:
```
# install `qutip` in the current Python kernel. Skip this if `qutip` is already installed.
# `matplotlib` is required for all visualization tasks.
# Make sure to restart your kernel if you execute this!
# In a Jupyter notebook, go to the menu bar > Kernel > Restart Kernel.
# In VSCode, click on the Restart button in the Jupyter toolbar.

# The '\' before the '>' operator is so that the shell does not misunderstand
# the '>' qualifier for the bash pipe operation.

import sys

try:
    import matplotlib.pyplot as plt
    import qutip

except ImportError:
    print("Tools not found, installing. Please restart your kernel after this is done.")
    !{sys.executable} -m pip install qutip\>=5 matplotlib\>=3.5
    print("\nNew libraries have been installed. Please restart your kernel!")
```

[14]:
```
import cudaq
import numpy as np

## Retry the subsequent cells by setting the target to density matrix simulator.
# cudaq.set_target("density-matrix-cpu")


@cudaq.kernel
def kernel(angles: np.ndarray):
    qubit = cudaq.qubit()
    rz(angles[0], qubit)
    rx(angles[1], qubit)
    rz(angles[2], qubit)
```

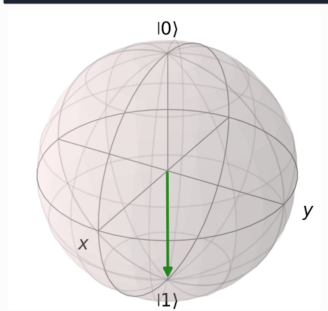
Qubit visualization

Next, we instantiate a random number generator, so we can get random outputs. We then create 4 random single-qubit states by using `cudaq.add_to_bloch_sphere()` on the output state obtained from the random kernel.

[15]:
```
rng = np.random.default_rng(seed=11)
blochSphereList = []
for _ in range(4):
    angleList = rng.random(3) * 2 * np.pi
    sph = cudaq.add_to_bloch_sphere(cudaq.get_state(kernel, angleList))
    blochSphereList.append(sph)
```

We can display the spheres with `cudaq.show()`. Show the first sphere:

[16]:
```
cudaq.show(blochSphereList[0])
```



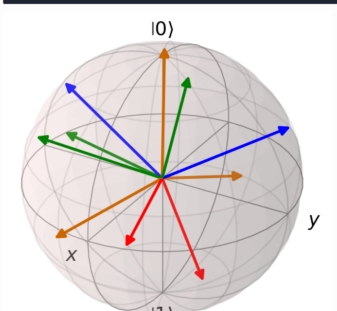
Qubit visualization

What if we had to add multiple vectors to a single Bloch sphere? CUDA-Q uses the QuTiP toolbox to construct Bloch spheres. We can then add multiple states to the same Bloch sphere by passing the sphere object as an argument to `cudaq.add_to_bloch_sphere()`.

[20]:
```
import qutip

rng = np.random.default_rng(seed=47)
blochSphere = qutip.Bloch()
for _ in range(10):
    angleList = rng.random(3) * 2 * np.pi
    sph = cudaq.add_to_bloch_sphere(cudaq.get_state(kernel, angleList), blochSphere)
```

This created a single Bloch sphere with 10 random vectors. Let us see how it looks.

[21]:
```
blochSphere.show()
```

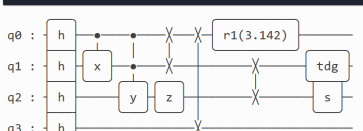


Kernel visualization

A CUDA-Q kernel can be visualized using the `cudaq.draw` API which returns a string representing the drawing of the execution path, in the specified format. ASCII (default) and LaTeX formats are supported.

[22]:
```
@cudaq.kernel
def kernel_to_draw():
    q = cudaq.qvector(4)
    h(q)
    x.ctrl(q[0], q[1])
    y.ctrl([q[0], q[1]], q[2])
    z(q[2])

    swap(q[0], q[1])
    swap(q[0], q[3])
    swap(q[1], q[2])

    r1(3.14159, q[0])
    tdg(q[1])
    s(q[2])
```

[23]:
```
print(cudaq.draw(kernel_to_draw))
```



[24]:
```
print(cudaq.draw('latex', kernel_to_draw))
```

```
\documentclass{minimal}
\usepackage{quantikz}
\begin{document}
\begin{quantikz}
  \lstick{$q_0$} & \gate{H} & \ctrl{1} & \ctrl{2} & \swap{1} & \swap{3} & \gate{R_1(3.142)} & \qw & \qw \\
  \lstick{$q_1$} & \gate{H} & \gate{X} & \ctrl{1} & \targX{} & \qw & \swap{1} & \gate{T^\dag} & \qw \\
  \lstick{$q_2$} & \gate{H} & \qw & \gate{Y} & \gate{Z} & \qw & \targX{} & \gate{S} & \qw \\
  \lstick{$q_3$} & \gate{H} & \qw & \qw & \qw & \targX{} & \qw & \qw & \qw \\
\end{quantikz}
\end{document}
```

Part IV

First circuit

First circuit

We can define our quantum kernel as a typical Python function, with the additional use of the `@cudaq.kernel` decorator. Let's begin with a simple GHZ-state example, producing a state of maximal entanglement amongst an allocated set of qubits.

```
import cudaq


# Define our kernel.
@cudaq.kernel
def kernel(qubit_count: int):
    # Allocate our qubits.
    qvector = cudaq.qvector(qubit_count)
    # Place the first qubit in the superposition state.
    h(qvector[0])
    # Loop through the allocated qubits and apply controlled-X,
    # or CNOT, operations between them.
    for qubit in range(qubit_count - 1):
        x.ctrl(qvector[qubit], qvector[qubit + 1])
    # Measure the qubits.
    mz(qvector)
```

This kernel function can accept any number of arguments, allowing for flexibility in the construction of the quantum program. In this case, the `qubit_count` argument allows us to dynamically control the number of qubits allocated to the kernel. As we will see in further examples, we could also use these arguments to control various parameters of the gates themselves, such as rotation angles.

First circuit

Now that you have defined your first quantum kernel, let's look at different options for how to execute it. In CUDA-Q, quantum circuits are stored as quantum kernels. For estimating the probability distribution of a measured quantum state in a circuit, we use the `sample` function call, and for computing the expectation value of a quantum state with a given observable, we use the `observe` function call.

Sample

Quantum states collapse upon measurement and hence need to be sampled many times to gather statistics. The CUDA-Q `sample` call enables this.

The `cudaq.sample()` method takes a kernel and its arguments as inputs, and returns a `cudaq.SampleResult`.

This result dictionary contains the distribution of measured states for the system.

Continuing with the GHZ kernel defined in Building Your First CUDA-Q Program, we will set the concrete value of our `qubit_count` to be two.

```
qubit_count = 2
print(cudaq.draw(kernel, qubit_count))
results = cudaq.sample(kernel, qubit_count)
# Should see a roughly 50/50 distribution between the |00> and
# |11> states. Example: {00: 505  11: 495}
print("Measurement distribution:" + str(results))
```

Part V

Observe

Observe

The observe function allows us to calculate expectation values for a defined quantum operator, that is the value of $\langle\psi|H|\psi\rangle$, where H is the desired operator and $|\psi\rangle$ is the quantum state after executing a given kernel.

The `cudaq.observe()` method takes a kernel and its arguments as inputs, along with a `cudaq.SpinOperator`.

Using the `cudaq.spin` module, operators may be defined as a linear combination of Pauli strings. Functions, such as `cudaq.spin.i()`, `cudaq.spin.x()`, `cudaq.spin.y()`, `cudaq.spin.z()` may be used to construct more complex spin Hamiltonians on multiple qubits.

Below is an example of a spin operator object consisting of a `z(0)` operator, or a Pauli Z-operator on the qubit zero. This is followed by the construction of a kernel with a single qubit in an equal superposition. The Hamiltonian is printed to confirm it has been constructed properly.

```
import cudaq
from cudaq import spin

operator = spin.z(0)
print(operator)  # prints: [1+0j] Z


@cudaq.kernel
def kernel():
    qubit = cudaq.qubit()
    h(qubit)
```

The `observe` function takes a kernel, any kernel arguments, and a spin operator as inputs and produces an `ObserveResult` object. The expectation value can be printed using the `expectation` method.

Observe

Note

It is important to exclude a measurement in the kernel, otherwise the expectation value will be determined from a collapsed classical state. For this example, the expected result of 0.0 is produced.

```
result = cudaq.observe(kernel, operator)
print(result.expectation())  # prints: 0.0
```

Unlike `sample`, the default `shots_count` for `observe` is 1. This result is deterministic and equivalent to the expectation value in the limit of infinite shots. To produce an approximate expectation value from sampling, `shots_count` can be specified to any integer.

```
result = cudaq.observe(kernel, operator, shots_count=1000)
print(result.expectation())  # prints non-zero value
```

Part VI

Running on a GPU

Running on a GPU

GPU node allocation: `salloc -A dd-24-88 -p qgpu_exp`

Using `cudaq.set_target()`, different targets can be specified for kernel execution.

If a local GPU is detected, the target will default to `nvidia`. Otherwise, the CPU-based simulation target, `qpp-cpu`, will be selected.

We will demonstrate the benefits of using a GPU by sampling our GHZ kernel with 25 qubits and a `shots_count` of 1 million. Using a GPU accelerates this task by more than 35x. To learn about all of the available targets and ways to accelerate kernel execution, visit the Backends page.

```
import sys
import timeit

# Will time the execution of our sample call.
code_to_time = 'cudaq.sample(kernel, qubit_count, shots_count=1000000)'
qubit_count = int(sys.argv[1]) if 1 < len(sys.argv) else 25

# Execute on CPU backend.
cudaq.set_target('qpp-cpu')
print('CPU time')  # Example: 27.57462 s.
print(timeit.timeit(stmt=code_to_time, globals=globals(), number=1))

if cudaq.num_available_gpus() > 0:
    # Execute on GPU backend.
    cudaq.set_target('nvidia')
    print('GPU time')  # Example: 0.773286 s.
    print(timeit.timeit(stmt=code_to_time, globals=globals(), number=1))
```

Running in parallel on a GPU using _async

```python
import cudaq
from cudaq import spin

qubit_count = 2

@cudaq.kernel
def kernel(qubit_count: int):
    qvector = cudaq.qvector(qubit_count)

    # 2-qubit GHZ state.
    h(qvector[0])
    for i in range(1, qubit_count):
        x.ctrl(qvector[0], qvector[i])

# Set the simulation target to a multi-QPU platform
cudaq.set_target("nvidia", option = 'mqpu')

# Measuring the expectation value of 2 different hamiltonians in parallel
hamiltonian_0 = spin.x(0) + spin.y(1) + spin.z(0)*spin.y(1)
hamiltonian_1 = spin.z(1) + spin.y(0) + spin.x(1)*spin.x(0)
hamiltonian_2 = spin.x(1) + spin.z(0) + spin.y(1)*spin.y(0)
hamiltonian_3 = spin.y(1) + spin.x(0) + spin.z(1)*spin.y(0)

# Asynchronous execution on multiple qpus via nvidia gpus.
result_0 = cudaq.observe_async(kernel, hamiltonian_0, qubit_count, qpu_id=0)
result_1 = cudaq.observe_async(kernel, hamiltonian_1, qubit_count, qpu_id=1)
result_2 = cudaq.observe_async(kernel, hamiltonian_2, qubit_count, qpu_id=2)
result_3 = cudaq.observe_async(kernel, hamiltonian_3, qubit_count, qpu_id=3)

# Retrieve results
print(result_0.get().expectation())
print(result_1.get().expectation())
print(result_2.get().expectation())
print(result_3.get().expectation())
```

Running in parallel on a GPU using cudaq.parallel.thread

```python
import cudaq
from cudaq import spin

cudaq.set_target("nvidia", option="mqpu")
target = cudaq.get_target()
num_qpus = target.num_qpus()
print("Number of QPUs:", num_qpus)

# Define spin ansatz.
@cudaq.kernel
def kernel(angle: float):
    qvector = cudaq.qvector(2)
    x(qvector[0])
    ry(angle, qvector[1])
    x.ctrl(qvector[1], qvector[0])

# Define spin Hamiltonian.
hamiltonian = 5.907 - 2.1433 * spin.x(0) * spin.x(1) - 2.1433 * spin.y(
    0) * spin.y(1) + .21829 * spin.z(0) - 6.125 * spin.z(1)

exp_val = cudaq.observe(kernel,
                        hamiltonian,
                        0.59,
                        execution=cudaq.parallel.thread).expectation()
print("Expectation value: ", exp_val)
```

Part VII

Quantum operations

Quantum operations

We can manipulate the state of a qubit via quantum gates. For example, the Pauli X gate allows us to flip the state of the qubit:

$$X|0\rangle = |1\rangle$$

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

```python
import cudaq


@cudaq.kernel
def kernel():
    # A single qubit initialized to the ground / zero state.
    qubit = cudaq.qubit()

    # Apply the Pauli x gate to the qubit.
    x(qubit)

    # Measurement operator.
    mz(qubit)


# Sample the qubit for 1000 shots to gather statistics.
result = cudaq.sample(kernel)
print(result.most_probable())
```

Quantum operations

The Hadamard gate allows us to put the qubit in an equal superposition state:

$$H|0\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle \equiv |+\rangle$$

$$\frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}\begin{bmatrix}1\\ 0\end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\ 0\end{bmatrix} + \frac{1}{\sqrt{2}}\begin{bmatrix}0\\ 1\end{bmatrix}.$$

The probability of finding the qubit in the $|0\rangle$ or $|1\rangle$ state is hence $|\frac{1}{\sqrt{2}}|^2 = \frac{1}{2}$. Lets verify this with some code:

```
import cudaq


@cudaq.kernel
def kernel():
    # A single qubit initialized to the ground/ zero state.
    qubit = cudaq.qubit()

    # Apply Hadamard gate to single qubit to put it in equal superposition.
    h(qubit)

    # Measurement operator.
    mz(qubit)


result = cudaq.sample(kernel)
print("Measured |0> with probability " +
      str(result["0"] / sum(result.values())))
print("Measured |1> with probability " +
      str(result["1"] / sum(result.values())))
```

Quantum operations

Just like the single-qubit gates above, we can define multi-qubit gates to act on multiple-qubits. The controlled-NOT or CNOT gate, for example, acts on 2 qubits: the control qubit and the target qubit. Its effect is to flip the target if the control is in the excited $|1\rangle$ state.

```
import cudaq


@cudaq.kernel
def kernel():
    # 2 qubits both initialized to the ground/ zero state.
    qvector = cudaq.qvector(2)

    x(qvector[0])

    # Controlled-not gate operation.
    x.ctrl(qvector[0], qvector[1])

    mz(qvector[0])
    mz(qvector[1])


result = cudaq.sample(kernel)
print(result)
```

Measurements

Quantum theory is probabilistic and hence requires statistical inference to derive observations. Prior to measurement, the state of a qubit is all possible combinations of α and β and upon measurement, wavefunction collapse yields either a classical 0 or 1.

The mathematical theory devised to explain quantum phenomena tells us that the probability of observing the qubit in the state $|0\rangle$ or $|1\rangle$, yielding a classical 0 or 1, is $|\alpha|^2$ or $|\beta|^2$, respectively.

As we see in the example of the Hadamard gate above, the result 0 or 1 each is yielded roughly 50% of the times as predicted by the postulate stated above thus proving the theory.

Classically, we cannot encode information within states such as 00 + 11 but quantum mechanics allows us to write linear superpositions

$$|\psi\rangle = \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle$$

where the probability of measuring $x = 00, 01, 10, 11$ occurs with probability $|\alpha_x|^2$ with the normalization condition that $\sum_{x\in\{0,1\}^2}|\alpha_x|^2 = 1$.

Part VIII

User-defined operations

User-defined operations

Users can define a custom quantum operation by its unitary matrix. First use the API to register a custom operation, outside of a CUDA-Q kernel. Then the operation can be used within a CUDA-Q kernel like any of the built-in operations defined above. Custom operations are supported on qubits only (`qudit` with `level = 2`).

The `cudaq.register_operation` API accepts an identifier string for the custom operation and its unitary matrix. The matrix can be a `list` or `numpy` array of complex numbers. A 1D matrix is interpreted as row-major.

```
import cudaq
import numpy as np

cudaq.register_operation("custom_h", 1. / np.sqrt(2.) * np.array([1, 1, 1, -1]))

cudaq.register_operation("custom_x", np.array([0, 1, 1, 0]))

@cudaq.kernel
def bell():
    qubits = cudaq.qvector(2)
    custom_h(qubits[0])
    custom_x.ctrl(qubits[0], qubits[1])

cudaq.sample(bell).dump()
```

User-defined operations

For multi-qubit operations, the matrix is interpreted with MSB qubit ordering, i.e. big-endian convention. The following example shows two different custom operations, each operating on 2 qubits.

```
import cudaq
import numpy as np

# Create and test a custom CNOT operation.
cudaq.register_operation("my_cnot", np.array([1, 0, 0, 0,
                                              0, 1, 0, 0,
                                              0, 0, 0, 1,
                                              0, 0, 1, 0]))

@cudaq.kernel
def bell_pair():
    qubits = cudaq.qvector(2)
    h(qubits[0])
    my_cnot(qubits[0], qubits[1]) # `my_cnot(control, target)`

cudaq.sample(bell_pair).dump() # prints { 11:500 00:500 } (exact numbers will be random)

# Construct a custom unitary matrix for X on the first qubit and Y
# on the second qubit.
X = np.array([[0,  1 ], [1 , 0]])
Y = np.array([[0, -1j], [1j, 0]])
XY = np.kron(X, Y)

# Register the custom operation
cudaq.register_operation("my_XY", XY)

@cudaq.kernel
def custom_xy_test():
    qubits = cudaq.qvector(2)
    my_XY(qubits[0], qubits[1])
    y(qubits[1]) # undo the prior Y gate on qubit 1

cudaq.sample(custom_xy_test).dump() # prints { 10:1000 }
```

Part IX

Deutsch's algorithm

Deutsch's algorithm

We have a function which takes in a bit and outputs a bit. This can be represented as $f : \{0, 1\} \longrightarrow \{0, 1\}$.

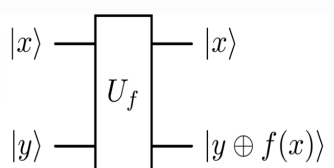
The function f has a property; either it is constant or balanced.

If constant, the outputs are the same regardless of the inputs, i.e., $f(0) = f(1) = 0$ or $f(0) = f(1) = 1$.

If balanced, the ouputs are balanced across their possibilities, i.e, if $f(0) = 0$ then $f(1) = 1$ or if $f(0) = 1$ then $f(1) = 0$.

The question we would like to answer is if the function is constant or balanced.

Quantum oracles



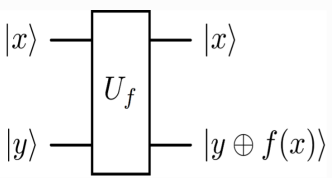
Suppose we have $f(x) : \{0, 1\} \longrightarrow \{0, 1\}$. We can compute this function on a quantum computer using oracles which we treat as black box functions that yield the output with an appropriate sequence of logic gates.
Above you see an oracle represented as U_f which allows us to transform the state $|x\rangle|y\rangle$ into:

$$U_f|x\rangle|y\rangle = |x\rangle|y \oplus f(x)\rangle$$

If $y = 0$, then $U_f|x\rangle|y\rangle = U_f|x\rangle|0\rangle = |x\rangle|0 \oplus f(x)\rangle = |x\rangle|f(x)\rangle$ since $f(x)$ can either be $0/1$ and $0 \oplus 0 = 0$ and $0 \oplus 1 = 1$.
This is remarkable because by setting $|y\rangle = |0\rangle$, we can extract the value of $f(x)$ by measuring the value of the second qubit.

Quantum parallelism

Consider:



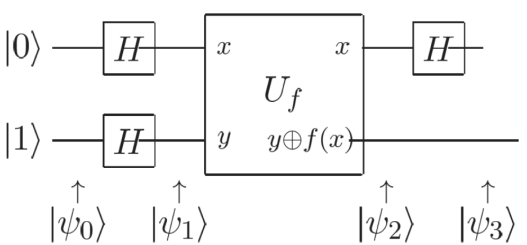
$$
\begin{aligned}
U_f|+\rangle|0\rangle &= U_f\tfrac{1}{\sqrt{2}}(|0\rangle + |1\rangle)|0\rangle \\
&= U_f\tfrac{1}{\sqrt{2}}(|0\rangle|0\rangle + |1\rangle|0\rangle) \\
&= \tfrac{1}{\sqrt{2}}(U_f|0\rangle|0\rangle + U_f|1\rangle|0\rangle) \\
&= \tfrac{1}{\sqrt{2}}(U_f|0\rangle|0\rangle + U_f|1\rangle|0\rangle) \\
&= \tfrac{1}{\sqrt{2}}(|0\rangle|0 \oplus f(0)\rangle + |1\rangle|0 \oplus f(1)\rangle) \\
&= \tfrac{1}{\sqrt{2}}(|0\rangle|f(0)\rangle + |1\rangle|f(1)\rangle)
\end{aligned}
$$

We have calculated information about both $f(0)$ and $f(1)$ simultaneously. Quantum mechanics allows for parallelism by exploiting the ability of superposition states.

Deutsch's algorithm

Our aim is to find out if $f : \{0, 1\} \longrightarrow \{0, 1\}$ is a constant or a balanced function? If constant, $f(0) = f(1)$, and if balanced, $f(0) \neq f(1)$.

We step through the circuit diagram below and follow the math after the application of each gate.



Deutsch's algorithm

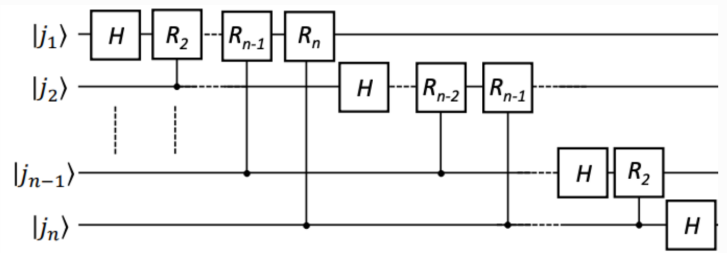
```python
@cudaq.kernel
def kernel(fx: List[int]):
    qubit_0 = cudaq.qubit()
    qubit_1 = cudaq.qubit()

    # Psi 0
    x(qubit_1)

    # Psi 1
    h(qubit_0)
    h(qubit_1)

    # Psi 2 - oracle
    if fx[0] == 1:
        x.ctrl(qubit_0, qubit_1)
        x(qubit_1)

    if fx[1] == 1:
        x.ctrl(qubit_0, qubit_1)

    # Psi 3
    h(qubit_0)

    # Measure the qubit to yield if the function is constant or balanced.
    mz(qubit_0)

print(cudaq.draw(kernel, fx))

result = cudaq.sample(kernel, fx, shots_count=1)

if np.array(result)[0] == '0':
    print('f(x) is a constant function')
elif np.array(result)[0] == '1':
    print('f(x) is a balanced function')
```

Part X

Quantum Fourier transform

Quantum Fourier transform

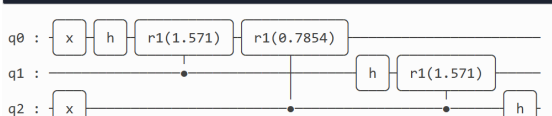
Here is a generalized ciruit for a n-qubit quantum Fourier transform:



[1]:
```
import cudaq
import numpy as np
from typing import List

@cudaq.kernel
def quantum_fourier_transform(input_state: List[int]):
    '''Args:
    input_state (list[int]): specifies the input state to be Fourier transformed.  '''
    qubit_count = len(input_state)

    # Initialize qubits.
    qubits = cudaq.qvector(qubit_count)

    # Initialize the quantum circuit to the initial state.
    for i in range(qubit_count):
        if input_state[i] == 1:
            x(qubits[i])

    # Apply Hadamard gates and controlled rotation gates.
    for i in range(qubit_count):
        h(qubits[i])
        for j in range(i + 1, qubit_count):
            angle = (2 * np.pi) / (2**(j - i + 1))
            cr1(angle, [qubits[j]], qubits[i])
```

Quantum Fourier transform

```
#Can be changed to 'nvidia' for single gpu, 'nvidia-mgpu' for multi-GPU or quantum hardware.
cudaq.set_target("qpp-cpu")

# The state to which the QFT operation is applied to. The zeroth element in the list is the zeroth qubit.
input_state = [1, 0, 1]

# Number of decimal points to round up the statevector to.
precision = 2

# Draw the quantum circuit.
print(cudaq.draw(quantum_fourier_transform, input_state))

# Print the statevector to the specified precision
statevector = np.array(cudaq.get_state(quantum_fourier_transform, input_state))
print(np.round(statevector, precision))
```




```
[ 0.35+0.j   -0.25-0.25j  0.  +0.35j  0.25-0.25j -0.35+0.j    0.25+0.25j
 -0.  -0.35j -0.25+0.25j]
```

Part XI

Cost minimization

Cost minimization

Below we start with a basic example of a hybrid variational algorithm which involves flipping the Bloch vector of a qubit from the $|0\rangle$ to the $|1\rangle$ state. First we import the relevant packages and set our backend to simulate our workflow on NVIDIA GPUs.

[1]:
```
import cudaq
from typing import List

cudaq.set_target("nvidia")
```

[2]:
```
# Initialize a kernel/ ansatz and variational parameters.
@cudaq.kernel
def kernel(angles: List[float]):
    # Allocate a qubit that is initialized to the |0> state.
    qubit = cudaq.qubit()
    # Define gates and the qubits they act upon.
    rx(angles[0], qubit)
    ry(angles[1], qubit)


# Our Hamiltonian will be the Z expectation value of our qubit.
hamiltonian = cudaq.spin.z(0)

# Initial gate parameters which intialize the qubit in the zero state
initial_parameters = [0, 0]

print(cudaq.draw(kernel, initial_parameters))
```

```
q0 : ─ rx(0) ─ ry(0) ─
```

Cost minimization

We build our cost function such that its minimal value corresponds to the qubit being in the $|1\rangle$ state. The observe call below allows us to simulate our statevector $|\psi\rangle$, and calculate $\langle\psi|Z|\psi\rangle$.

[3]:
```
cost_values = []


def cost(parameters):
    """Returns the expectation value as our cost."""
    expectation_value = cudaq.observe(kernel, hamiltonian,
                                      parameters).expectation()
    cost_values.append(expectation_value)
    return expectation_value
```

[4]:
```
# We see that the initial value of our cost function is one, demonstrating that our qubit is in the zero state
initial_cost_value = cost(initial_parameters)
print(initial_cost_value)
```

```
1.0
```

Cost minimization

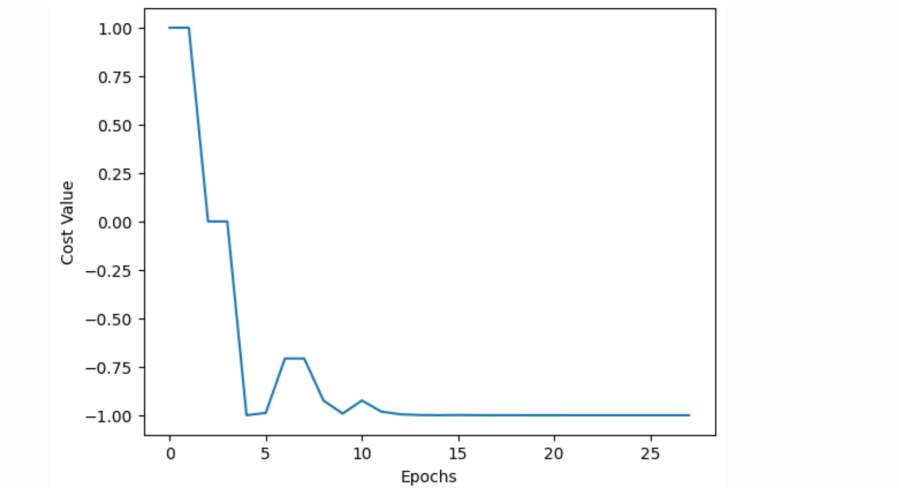
Below we use our built-in optimization suite to minimize the cost function. We will be using the gradient-free COBYLA alogrithm.

[5]:
```
# Define a CUDA-Q optimizer.
optimizer = cudaq.optimizers.COBYLA()
optimizer.initial_parameters = initial_parameters

result = optimizer.optimize(dimensions=2, function=cost)
```

[7]:
```
# Plotting how the value of the cost function decreases during the minimization procedure.
import matplotlib.pyplot as plt

x_values = list(range(len(cost_values)))
y_values = cost_values

plt.plot(x_values, y_values)

plt.xlabel("Epochs")
plt.ylabel("Cost Value")
```

[7]:
```
Text(0, 0.5, 'Cost Value')
```

Cost minimization

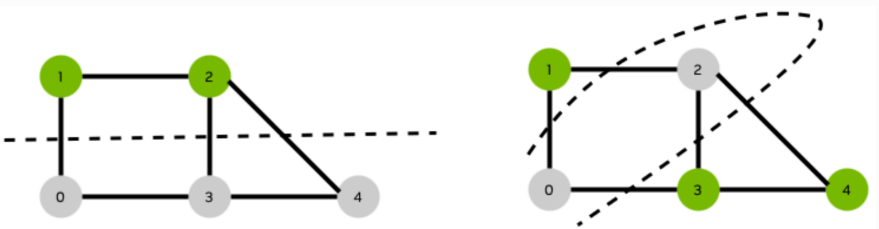


We see that the final value or our cost function, $\langle\psi|Z|\psi\rangle = -1$ demonstrating that the qubit is in the $|1\rangle$ state.

Part XII

Max-Cut with QAOA

Max-Cut with QAOA



[1]:
```
import numpy as np
import cudaq
from cudaq import spin
from typing import List

# We'll use the graph below to illustrate how QAOA can be used to
# solve a max cut problem

#       v1  0-------------0 v2
#           |             | \
#           |             |  \
#           |             |   \
#           |             |    \
#       v0  0-------------0 v3-- 0 v4
# The max cut solutions are 01011, 10100, 01010, 10101 .

# First we define the graph nodes (i.e., vertices) and edges as lists of integers so that they can be broadcast into
# a cudaq.kernel.
nodes: List[int] = [0, 1, 2, 3, 4]
edges = [[0, 1], [1, 2], [2, 3], [3, 0], [2, 4], [3, 4]]
edges_src: List[int] = [edges[i][0] for i in range(len(edges))]
edges_tgt: List[int] = [edges[i][1] for i in range(len(edges))]
```

Part XIII

Noisy simulations

Noisy simulations

Quantum noise can be characterized into coherent and incoherent sources of errors that arise during a computation. Coherent noise is commonly due to systematic errors originating from device miscalibrations, for example, gates implementing a rotation $\theta + \epsilon$ instead of θ.

Incoherent noise has its origins in quantum states being entangled with the environment due to decoherence. This leads to mixed states which are probability distributions over pure states and are described by employing the density matrix formalism.

We can model incoherent noise via quantum channels which are linear, completely positive, and trace preserving maps. These maps are called Kraus operators, $\{K_i\}$, which satisfy the condition $\sum_i K_i^\dagger K_i = \mathbb{I}$.

The bit-flip channel flips the qubit with probability p and leaves it unchanged with probability $1 - p$. This can be represented by employing Kraus operators:

$$K_0 = \sqrt{1-p}\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

$$K_1 = \sqrt{p}\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

Noisy simulations

Let's implement the bit-flip channel using CUDA-Q:

[1]:
```
import cudaq
from cudaq import spin

import numpy as np

# To model quantum noise, we need to utilize the density matrix simulator target.
cudaq.set_target("density-matrix-cpu")
```

[2]:
```
# Let's define a simple kernel that we will add noise to.
qubit_count = 2


@cudaq.kernel
def kernel(qubit_count: int):
    qvector = cudaq.qvector(qubit_count)
    x(qvector)


print(cudaq.draw(kernel, qubit_count))
```

```
q0 : ┤ x ├
q1 : ┤ x ├
```

Noisy simulations

```
# First, we will define an out of the box noise channel. In this case,
# we choose depolarization noise. This depolarization will result in
# the qubit state decaying into a mix of the basis states, |0> and |1>,
# with our provided probability.
error_probability = 0.1
depolarization_channel = cudaq.DepolarizationChannel(error_probability)

# Other built in noise models
bit_flip = cudaq.BitFlipChannel(error_probability)
phase_flip = cudaq.PhaseFlipChannel(error_probability)
amplitude_damping = cudaq.AmplitudeDampingChannel(error_probability)

# Define the Kraus Error Operator as a complex ndarray.
kraus_0 = np.sqrt(1 - error_probability) * np.array([[1.0, 0.0],
                                                     [0.0, 1.0]],
                                                    dtype=np.complex128)

kraus_1 = np.sqrt(error_probability) * np.array([[0.0, 1.0],
                                                 [1.0, 0.0]],
                                                dtype=np.complex128)

# Add the Kraus Operator to create a quantum channel.
bitflip_channel = cudaq.KrausChannel([kraus_0, kraus_1])

# Add noise channels to our noise model.
noise_model = cudaq.NoiseModel()

# Apply the depolarization channel to any X-gate on the 0th qubit.
noise_model.add_channel("x", [0], depolarization_channel)
# Apply the bitflip channel to any X-gate on the 1st qubit.
noise_model.add_channel("x", [1], bitflip_channel)

# Due to the impact of noise, our measurements will no longer be uniformly
# in the |11> state.
noisy_counts = cudaq.sample(kernel,
                            qubit_count,
                            noise_model=noise_model,
                            shots_count=1000)
noisy_counts.dump()
```

Part XIV

Hardware backends

Hardware backends

The following code illustrates how to run kernels on IonQ's backends:

```
import cudaq
import matplotlib.pyplot as plt

cudaq.set_target("ionq", target="simulator", noise="aria-1") # noise="aria-1" noise="ideal" noise="harmony" emulate=True
#cudaq.set_target('quantinuum', emulate=True) # ionq oqc quantinuum

# Create the kernel we'd like to execute on IonQ.
@cudaq.kernel
def kernel(qubit_count: int):
    qvector = cudaq.qvector(qubit_count)
    h(qvector[0])
    for qubit in range(qubit_count - 1):
        x.ctrl(qvector[qubit], qvector[qubit + 1])

results = cudaq.sample(kernel,5)

print(results)
```

Thanks

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 101101903. The JU receives support from the Digital Europe Programme and Germany, Bulgaria, Austria, Croatia, Cyprus, Czech Republic, Denmark, Estonia, Finland, Greece, Hungary, Ireland, Italy, Lithuania, Latvia, Poland, Portugal, Romania, Slovenia, Spain, Sweden, France, Netherlands, Belgium, Luxembourg, Slovakia, Norway, Türkiye, Republic of North Macedonia, Iceland, Montenegro, Serbia