Running your first CUDA Quantum Program

Now that you have defined your first quantum kernel, let's look at different options for how to execute it.

Sample

Python C++

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 Quantum Program, we will set the concrete value of our `qubit_count` to be two. The following will assume this code exists in a file named `sample.py`.

```
qubit_count = 2
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(results)
```

By default, `sample` produces an ensemble of 1000 shots. This can be changed by specifying an integer argument for the `shots_count`.

```
# With an increased shots count, we will still see the same 50/50 distribution,
# but now with 10,000 total measurements instead of the default 1000.
# Example: {00: 5005  11: 4995}
results = cudaq.sample(kernel, qubit_count, shots_count=10000)
print(results)
```

A variety of methods can be used to extract useful information from a `cudaq.SampleResult`. For example, to return the most probable measurement and its respective probability:

```
print(results.most_probable())  # prints: `00`
print(results.probability(results.most_probable()))  # prints: `0.5005`
```

We can execute this program as we do any Python file.

```
python3 sample.py
```

See the API specification for further information.

Observe

Python C++

The `cudaq.observe()` method takes a kernel and its arguments as inputs, along with a `cudaq.SpinOperator`. As opposed to `cudaq.sample()`, `observe` is primarily used to produce expectation values of a kernel with respect to a provider operator.

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 zeroth qubit. 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)
```

`cudaq::observe` 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.

> **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 `cudaq::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
```

Running on a GPU

Python C++

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))
```