cuda-Q Demo — A Kernel of Probable Truth

Getting Started:

If installing locally/natively, follow the *Install* section below. If running the Docker container, follow the *Docker* section, which immediately follows.

Installation:

First, make sure you have a modern gcc toolchain installed. Clang should be fine, too, if that is your preference. *If you do not have a CUDA Compute 8.0-capable GPU in your system, do not run the gpu simulations. They will fail.*

Download the appropriate install file for your Debian system:
[x86 Installer]
[ARM Installer]

If working from the CLI, download the installer via the following command. This demo is intended for x86 machines with a CUDA-compatible device installed. CPU-only users can still execute the portions of this demo which don't require a GPU.

x86:
```
Wget https://github.com/NVIDIA/cuda-quantum/releases/download/0.7.0/install_cuda_quantum.x86_64
```
ARM:
```
Wget https://github.com/NVIDIA/cuda-quantum/releases/download/0.7.0/install_cuda_quantum.aarch64
```

`LINK_ADDRESS` should be the url of the download file.

To complete the installation, `cd` to the directory where the installer downloaded, and enter the commands:
```
sudo -E bash install_cuda_quantum.$(uname -m) --accept
. /etc/profile
```

Because of some weirdness in the install scripts released by NVidia, you may need to run the following, if you find that your shell is either broken or acting strangely:
```
source ~/.bashrc
```

Docker:

To make the demo quite a bit neater, you can pull a Docker image for it from DockerHub.

```
docker pull connorpds/cuda-q-cpp-demo:1.0
```

As long as your system has an x86 CPU and some CUDA-compatible device, this should be a plug-and-play solution.
To run the Docker image:

```
docker run -it connorpds/cuda-q-cpp-demo:1.0
```

Demonstration:

Simple Quantum Circuit and Compilation:

To test our installation/container, and to highlight the basic compilation flow, we start with the following simple GHZ state preparation kernel, defined in *ghz_single_10.cpp*, trialled once on 10 qubits.

```cpp
#include <cudaq.h>

// cudaq kernel that is fully specified
// at compile time via templates.
template <std::size_t N>
struct ghz {
  auto operator()() __qpu__ {

    // Compile-time sized array like std::array
    cudaq::qarray<N> q;
    h(q[0]);
    for (int i = 0; i < N - 1; i++) {
      x<cudaq::ctrl>(q[i], q[i + 1]);     ■ Use of function template
    }
    mz(q);

};

int main() {

  auto kernel = ghz<10>{};
  auto counts = cudaq::sample(kernel);

  if (!cudaq::mpi::is_initialized() || cudaq::mpi::rank() == 0) {
    counts.dump();

    // Fine grain access to the bits and counts
    for (auto &[bits, count] : counts) {
      printf("Observed: %s, %lu\n", bits.data(), count);
    }
  }

  return 0;
}
```

Your LSP won't like It, either.

To compile:

```
nvq++ ghz_single_10.cpp -O ghz_single_10.x  --target qpp-cpu
```

To execute the quantum kernel, run the binary as you would any other:

```
./ghz_single_10.x
```

Scaling Up:

Here is a slightly more complicated example—but the only difference we care about is on line 30: the change in input to `cudaq::sample.` In increasing the number of qubits to 28, we are instantiating a quantum kernel which is considerably harder to simulate.

```
13 #include <cudaq.h>
14
15 // Define a quantum kernel with a runtime parameter
16 struct ghz {
17   auto operator()(const int N) __qpu__ {
18
19     // Dynamically sized vector of qubits
20     cudaq::qvector q(N);
21     h(q[0]);
22     for (int i = 0; i < N - 1; i++) {
23       x<cudaq::ctrl>(q[i], q[i + 1]);      ■ Use of function template
24     }
25     mz(q);
26   }
27 };
28
29 int main() {
30   auto counts = cudaq::sample(/*shots=*/100, ghz{}, 28);
31
32   if (!cudaq::mpi::is_initialized() || cudaq::mpi::rank() == 0) {
33     counts.dump();
34
35     // Fine grain access to the bits and counts
36     for (auto &[bits, count] : counts) {
37       printf("Observed: %s, %lu\n", bits.data(), count);
38     }
39   }
40
41   return 0;
42 }
```

Depending on the size of our model, we may have trouble simulating locally with just a CPU. For example, when prepping GHZ states, we find that simulating more than 27 or so qubits hangs. Thus, we can instead opt to use the cuQuantum simulation backend, which uses the local CUDA device to accelerate quantum simulation. To use the cuQuantum simulation backend instead of the conventional CPU/MPI-based backend, change the compilation target to "nvidia".

```
nvq++ --target nvidia ghz_28.cpp -o ghz_28-gpu-sim.x
```

To execute:

```
./ghz_28-gpu-sim.x
```

To compile the same source, targeting the conventional CPU/MPI backend.

```
nvq++ --target qpp-cpu ghz_28.cpp -o ghz_28-cpu-sim.x
```

Don't execute that on its own. It will appear to hang simply because of the complexity of simulating many qubits. Instead, to see how the runtimes compare between the gpu and cpu, run the following, which invokes both binaries (`ghz_28-cpu-sim.x, ghz_28-gpu-sim.x`, times execution out after 10 seconds, and reports the runtimes of completed executions:

```
bash compare-with-timeout.sh ghz_28-cpu-sim.x ghz_28-gpu-sim.x
```

Targeting A Hardware Backend:

Among those from a few other providers, cudaQ supports IonQ hardware backends, both physical and emulated. Access to real hardware is slow due to queue times, expensive due to physics, and restricted (pick 3!). IonQ provides free and instant access to noiseless and noisy *emulations* of their hardware. No modification of the source files is necessary to change to IonQ backends. Indeed, to target this noise-aware emulated backend, merely alter the compilation as follows:

```
nvq++ ghz_28.cpp -o ghz_28-aria_1_noisy-sim.x --target ionq --ionq-machine
simulator --ionq-noise-model aria-1
```

Simply Lovely Syntax and Semantics:

Classical architects, and C and c++ programmers in general, will find themselves right at home with clean, expressive syntax that follows familiar patterns. The quantum kernel API is modeled directly off the standard CUDA kernel API—more familiarity and consistency. Naturally, cuda-Q natively supports writing programs which integrate traditional CPU execution with GPU *and* QPU operations in tandem.

```
#include <cudaq.h>

struct basic_gates_example{

    __qpu__ void operator()() {

      cudaq::qvector q(2);
      h(q[0]);  // Hadamard gate on the first qubit
      x(q[0]);  // Pauli-X gate on the first qubit
      y(q[1]);  // Pauli-Y gate on the second qubit
      z(q[1]);  // Pauli-Z gate on the second qubit
      x<cudaq::ctrl>(q[0], q[1]);     ■ Use of function template name
      mz(q);
    }
};

int main() {
    auto counts = cudaq::sample(100, basic_gates_example{});

    if (!cudaq::mpi::is_initialized() || cudaq::mpi::rank() == 0) {
      counts.dump();

    // Fine grain access to the bits and counts
    for (auto &[bits, count] : counts) {
      printf("Observed: %s, %lu\n", bits.data(), count);
    }
  }
    return 0;
}
```