Cuda-Q

Jaime Giné

May 2024

Resumen

La simulación de circuitos en sistemas de computación de alto rendimiento tiene un papel clave en la implementación de algoritmos. En este trabajo vamos a ver una especificación resumida de CUDA-Q, las principales características que hacen destacar a dicho modelo de programación, y el por qué puede ayudar a mejorar el rendimiento de simulaciones de circuitos cuánticos.

1 Introducción

CUDA-Q es un modelo de programación, librería cuántica y cadena de herramientas para sistemas híbridos cuántico-clásicos. Esta basado en programación de kernels y se puede utilizar en C++ y Python. Permite el uso de aceleración cuántica mediante GPUs de NVIDIA, además de diversos backends que ofrece, como NVIDIA cuQuantum, y está diseñado para aprovechar sistemas heterogéneos, con CPUs, GPUs, y QPUs. Ofrece una cadena de herramientas de compilación que utiliza NVQ++ para los kernels cuánticos, y utilizando MLIR y QIR para permitir la aceleración.

Programación basada en kernels

Los kernels, en computación paralela, son fragmentos de código que se ejecutan en dispositivos hardware programables, como pueden ser GPUs. Este estándar de programación ha sido adoptado por muchos lenguajes de paralelización como OpenCL, SYCL y CUDA. En la figura 1 podemos ver un ejemplo de cómo es un kernel en CUDA que implementa una multiplicación de matrices. Podemos observar que un kernel se trata de una función, en la que se indica que es un kernel mediante palabras clave. En el caso de CUDA, se indica con *__global__*.

2 Especificación de CUDA-Q

En este capítulo vamos a explicar la especificación en Python de CUDA-Q.

```
__global__ void addMatrixGPU(float *a, float *b, float *c, int N )
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<N) c[i] = a[i] + b[i];
}
```

Figure 1: Kernel de CUDA que se utiliza para la suma de matrices o vectores.

Kernels en CUDA-Q

Los kernels en CUDA-Q, al igual que en otros modelos de programación, son funciones que representan datos a ejecutar en dispositivos hardware, siendo en este caso un QPU o un simulador de QPU. Suelen contener descripciones de circuitos cuánticos y son una nueva forma de abstracción de programación cuántica, permitiendo la mezcla con funciones clásicas para crear aplicaciones cuantico-clásicas reales que se pueden ejecutar en un sistema heterogéneo.

Una de las ventajas que supone la programación de kernels cuánticos es la adición de sentencias de flujo de control clásico, como *if-else, for*, etc., y de operaciones matemáticas clásicas. Dichas sentencias condicionales se pueden incluir en medición de qubit y demás operaciones cuánticas que permiten la creación de circuitos cuánticos dinámicos.

Como hemos podido ver, los circuitos cuánticos se representan con kernels cuánticos, pero estos últimos no están limitados a solo representar circuitos.

En la figura 2 tenemos un ejemplo de un kernel cuántico en CUDA-Q. Para crear un kernel tenemos que definir una función con los parámetros que deseemos, y añadir previamente las palabras clave *@cudaq.kernel* para indicar que la siguiente función es un kernel. A partir de ahí podemos programar lo que necesitemos. En caso del ejemplo, representa un circuito cuántico GHZ con número de qubits pasado por parámetro.

Existe otra manera de crear kernels cuánticos, y es haciendo uso del método *cudaq.make_kernel()*. De esta manera alojamos en memoria un kernel cuántico dinámico, y podemos utilizar los mismos métodos que se usan a la hora de crear kernels dentro de una función, con la diferencia de que hay que indicar el kernel (resultado de utilizar *cudaq.make_kernel()*) al que queremos aplicar la operación.

Como crear qubits

Aunque CUDA-Q trabaja con qudits de n-dimensiones nos vamos a centrar en el uso de qubits, que son qudits de 2 dimensiones.

Para crear qubits dentro de una función kernel, basta con utilizar *cuda.qubit* si queremos utilizar uno solo, o *cuda.qvector(n)* si queremos utilizar n qubits.

```python
import cudaq

# Definición del kernel GHZ
@cudaq.kernel
def kernel(numero_qubits: int):
    # Creamos qubits
    qubits = cudaq.qvector(numero_qubits)

    # Ponemos el primer qubit en superposición
    h(qubits[0])

    # Aplicamos un CNOT a los demás qubits
    for qubit in range(numero_qubits - 1):
        x.ctrl(qubits[qubit], qubits[qubit + 1])

    # Medimos los qubits
    mz(qubits)
```

Figure 2: Kernel cuántico en CUDA-Q que representa el circuito cuántico GHZ.

Operadores

En las tablas 1 y 2 podemos encontrar los diferentes operadores unitarios que podemos aplicar a los qubits, donde podemos encontrar los operadores predefinidos {x, y, z, h, t, s, swap} y operadores con cantidad de rotación por parámetro {r1, rx, ry, rz, u3}, y sus respectivas matrices que representan.

Para aplicar operaciones controladas, basta con utilizar el método *ctrl* de cualquiera de las puertas. Este método tiene como parámetros una primera lista conteniendo los qubits de control, y una segunda lista conteniendo los qubits sobre los que se aplica la operación.

Para aplicar la conjugada traspuesta de cualquiera de las puertas, basta con utilizar el método *adj*.

Mediciones de qubits

Para añadir mediciones a los qubits podemos utilizar las funciones {mz, mx, my} las cuales miden el qubit respecto sus bases de Pauli-(Z, X, Y).

Nombre	Operador
x	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$
y	$\begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$
z	$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$
h	$(1/\sqrt{2})\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$
t	$\begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}$
s	$\begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$
swap	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

Table 1: Operadores unitarios predeterminados

Nombre	Operador
rx(θ)	$\begin{bmatrix} \cos\theta/2 & -i\sin\theta/2 \\ -i\sin\theta/2 & \cos\theta/2 \end{bmatrix}$
ry(θ)	$\begin{bmatrix} \cos\theta/2 & -\sin\theta/2 \\ \sin\theta/2 & \cos\theta/2 \end{bmatrix}$
rz(λ)	$\begin{bmatrix} e^{-i\lambda/2} & 0 \\ 0 & e^{i\lambda/2} \end{bmatrix}$
r1(λ)	$\begin{bmatrix} 1 & 0 \\ 0 & e^{i\lambda} \end{bmatrix}$
u3(θ, ϕ, λ)	$\begin{bmatrix} \cos\theta/2 & -e^{i\lambda} * \sin\theta/2 \\ e^{i\phi} * \sin\theta/2 & e^{i(\lambda+\phi)} * \cos\theta/2 \end{bmatrix}$

Table 2: Operadores unitarios con cantidad de rotación por parámetro

Operadores de spin

CUDA-Q ofrece un tipo que es *spin_op*, que provee una abstracción de un producto tensorial de operadores de Pauli. En otras palabras, podemos crear Hamiltonianos, que luego usaremos para computar los valores esperados de un kernel.

Un hamiltoniano, en mecánica cuántica, es una descripción matemática de la energía de un sistema. En caso de trabajar con CUDA-Q, los sistemas serán los circuitos cuánticos que creemos con los kernels.

```python
import cudaq
from cudaq import spin

@cudaq.kernel
def kernel(theta: float):
    qubits = cudaq.qvector(2)
    x(qubits[0])
    ry(theta, qubits[1])
    x.ctrl(qubits[1], qubits[0])

spin_operator = 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)

# ángulo que le pasamos por parametro
angulo = 0.59

energia = cudaq.observe(kernel, spin_operator, angulo).expectation()
print(f"La energía es {energia}")
```

Figure 3: Ejecución de los valores esperados y medición de energía de un kernel utilizando el método Observe

Ejecutando los kernels

Una vez hemos visto todos los bloques anteriores, podemos definir gran cantidad de circuitos cuánticos, pero ¿como los ejecutamos? Existen varios métodos para ejecutar un kernel, y cada uno ofrece unas características distintas.

Sample

Empezando por el primero de ellos, *cudaq.sample* equivale a la medida de los valores de la ejecución de un circuito. En este caso, en la llamada de la función *sample*, le indicamos el kernel que queremos ejecutar, el número de veces (o shots) que queremos medir el circuito, y opcionalmente un modelo de ruido. Esta función devuelve los resultados que luego podemos mostrar por pantalla e interpretarlos.

Más adelante veremos los modelos de ruido, y ejemplos de ejecución con el método *sample*.

Observe

Habiendo visto ya el operador *spin_op*, este siguiente método, *cudaq.observe*, computa los valores esperados de un kernel respecto a un *spin_operator* (hamiltoniano). En la figura 3 podemos observar un ejemplo de uso del método observe, donde medimos la energía del kernel previamente descrito, al que le pasamos el ángulo por parámetro, respecto al Hamiltoniano definido.

Get_state

El último, y menos utilizado (en mi caso), es *cudaq.get_state*. Este método ejecuta un kernel y nos devuelve el estado, generalmente como vector o matriz, de dicho kernel al finalizar la ejecución.

Ejecución con ruido

En esta última sección vamos a ver como ejecutar kernels con modelos de ruido. En CUDA-Q es requerido la creación de un modelo de ruido personalizado, por lo que se necesita crear un modelo de ruido vacío e ir añadiendo los tipos de ruido a sus respectivas puertas manualmente.

En la figura 4 vamos a ver un ejemplo de código que muestra unos cuantos de los tipos de ruidos que hay, cómo generarlos y como aplicarlos a un kernel para una posterior simulación con ruido. En dicha figura podemos ver como creamos un circuito cuántico que genera el estado $|11>$, y cuando probamos a simularlo sin ruido efectivamente nos da un 100% de casos con $|11>$. Posteriormente pasamos a crear un canal de ruido de despolarización con probabilidad de error de 0.1, y a crear ruido de *bit_flip* representado por operadores de Kraus (se puede crear también con *cudaq.BitFlipChannel(err)*). Una vez creados ambos canales de ruido los añadimos a un modelo vacío creado con *cudaq.NoiseModel()*, indicando que el canal de ruido de despolarización va a hacer efecto a la puerta NOT del primer qubit y el segundo canal de ruido va a hacer efecto a la puerta NOT del segundo qubit. Finalmente ejecutamos el kernel indicando el modelo de ruido y podemos ver como afecta en los resultados de ejecución.

Además de los ruidos de despolarización y de *bitflip* también se pueden implementar los tipos de ruido de *phase-flip* y *amplitude damping*, cuyas explicaciones se pueden encontrar en esta pagina.

3 Puntos clave de CUDA-Q

Hasta ahora hemos visto una explicación de lo básico y necesario para crear programas y circuitos cuánticos, pero en este capítulo vamos a ver las características que hacen a CUDA-Q destacar.

3.1 Aceleración de kernels

La característica principal de CUDA-Q es la posibilidad de acelerar ejecuciones de kernels mediante la simulación de QPUs a partir de GPUs de NVIDIA. Esto se suele utilizar para aplicaciones que hacen uso de gran cantidad de qubits y, como veremos, produce speedups notables.

Cómo acelerar con GPUs localmente

En CUDA-Q podemos utilizar los propios dispositivos hardware (tanto CPU como GPUs) para ejecutar kernels. Antes de ejecutar kernels en nuestra GPU,

hace falta ver si CUDA-Q la detecta. En caso positivo podemos pasar a ejecutar kernels en la GPU usando la función *cudaq.set_target("nvidia")*.

En la tabla 3 tenemos los diferentes tipos de objetivos que podemos utilizar y, como veremos en secciones posteriores, dichos objetivos no están limitados localmente, sino que permiten también acceder a backends reales de diferentes proveedores.

Objetivo	Descripción
qpp-cpu	Backend de CPU que utiliza multi-threading
nvidia	Aceleración en NVIDIA GPUs mediante cuQuantum
nvidia-mqpu	Utilización de múltiples procesadores cuánticos con emulación de GPUs
nvidia-mgpu	Permite el escalado de circuitos no computables por una QPU normal
density-matrix-cpu	Para simulación con ruido (por ahora solo con CPU)

Table 3: Tipos de objetivos locales

3.2 Aceleración simultánea y paralela con múltiples GPUs

Otra de las características que incluye CUDA-Q es la posibilidad de ejecutar kernels asíncronamente en diversos dispositivos o backends. Esto es gracias a las variaciones *_async()* de los diferentes métodos *sample* y *observe*, que permiten enviar la ejecución de kernels a dispositivos y poder seguir ejecutando el programa, y posteriormente poder acceder a los resultados una vez haya terminado el kernel.

A su vez, es posible el encolamiento y ejecución de kernels en diferentes dispositivos simultáneamente, por lo que podríamos tener kernels ejecutándose en paralelo en nuestra CPU, GPU y QPU.

3.3 Utilización de CUDA-Q en aplicaciones híbridas

Como hemos hablado anteriormente, CUDA-Q destaca a la hora de utilización de entornos heterogéneos. En esta página de la documentacion de CUDA-Q podemos ver un ejemplo extenso de implementación de una aplicación híbrida que sigue el modelo de la figura 5. En este caso ambas secciones, tanto las partes clásicas como las cuánticas pueden ser aceleradas mediante el uso de kernels.

3.4 Ejecución en backends reales

La ejecución de kernels no está limitada a nuestro dispositivo local. Actualmente hay 4 proveedores hardware externos a los que es posible enviar kernels (con

acceso y configuración previa), los cuales son {Quantinuum, IonQ, IQM, OQC (Oxford Quantum Circuits)}.

NVIDIA también ofrece acceso a NVIDIA Quantum Cloud (NVQC), el cual es una plataforma que ofrece grandes capacidades de computación para usuarios cuánticos. Permite seleccionar múltiples backends, en los que se incluye simulación con múltiples GPUs y con ejecución asíncrona en multiples QPUs.

Ejemplo de minimización de coste con optimizadores de CUDA-Q

En esta última sección vamos a mostrar la ejecución de un ejemplo de minimización de coste haciendo uso de optimizadores proveídos por CUDA-Q.

En la figura 6 podemos ver el código de un programa entero que realiza la optmización de un algoritmo híbrido variacional. El cual implica darle la vuelta a un qubit desde el estado $|0>$ al estado $|1>$. Podemos observar que definimos una función de coste, que nos va a indicar la rotación del qubit (siendo 1 el estado $|0>$ y -1 el estado $|1>$) ya que utilizamos el método *observe* respecto a un operador de spin en el eje Z. Despues de definir la función de coste, basta con utilizar un optimizador CUDA-Q, en este caso se utiliza el optimizador de gradiente COBYLA, e indicarle los parámetros iniciales al igual que su función de cose. Una vez indicado, con el resultado obtenido por el método *optimizer.optimize(...)* podemos luego mostrarlo por pantalla, mostrándolo en una gráfica haciendo uso de matplotlib, en la figura 7.

Todas las operaciones hacen uso de aceleración con GPU ya que hemos indicado al principio del programa que haga uso de ello.

4 Conclusiones

Hemos logrado ver que CUDA-Q es una librería muy completa que permite diseñar circuitos cuánticos y kernels de una nueva manera, creando una nueva forma de programar para computación cuántica.

Hemos visto también que CUDA-Q permite acelerar kernels (o circuitos cuánticos) mediante GPUs de NVIDIA localmente, lo que genera grandes speedups para ejecuciones con gran cantidad de qubits, además de poder utilizar backends de proveedores de hardware real.

Respecto a el estado de CUDA-Q en la actualidad, no existen (a fecha de hoy) estudios o artículos que hagan uso de este debido a que se trata de una tecnología muy reciente (los únicos articulos que hay es sobre cuQuantum, que también podria haber merecido un trabajo propio).

```python
import numpy as np

import cudaq
from cudaq import spin

# Para poder utilizar ruido necesitamos utilizar "density-matrix-cpu"
cudaq.set_target("density-matrix-cpu")

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

numero_qubits = 2
print(cudaq.draw(kernel, numero_qubits))
#      ╭───╮
# q0 : ┤ x ├
#      ├───┤
# q1 : ┤ x ├
#      ╰───╯
counts_ideales = cudaq.sample(kernel, numero_qubits, shots_count=1000)
counts_ideales.dump()
# { 11:1000 }

# Creamos un modelo de ruido vacío
modelo_ruido = cudaq.NoiseModel()

# Creamos el canal de ruido de despolarización con probabiliad de error de 0.1
probabilidad_error = 0.1
canal_despolarizacion = cudaq.DepolarizationChannel(probabilidad_error)

# Creamos los operadores de kraus como matrices que indican la probabilidad de que sea |0> o |1>
kraus_0 = np.sqrt(1 - probabilidad_error) * np.array([[1.0, 0.0], [0.0, 1.0]], dtype=np.complex128)
kraus_1 = np.sqrt(probabilidad_error) * np.array([[0.0, 1.0], [1.0, 0.0]], dtype=np.complex128)
# Creamos el canal de ruido bitflip a partir de los operadores de Kraus
canal_bitflip = cudaq.KrausChannel([kraus_0, kraus_1])

#Aplicamos el canal de ruido de despolarización a la puerta NOT del primer qubit (0)
modelo_ruido.add_channel("x", [0], canal_despolarizacion)
#Aplicamos el canal de ruido de bitflip a la puerta NOT del segundo qubit (1)
modelo_ruido.add_channel("x", [1], canal_bitflip)

counts_ruidosos = cudaq.sample(kernel, numero_qubits, noise_model=modelo_ruido, shots_count=1000)
counts_ruidosos.dump()
# Ejemplo de ejecución:
# { 11:847 10:90 01:57 00:6 }
```

Figure 4: Ejecución con ruido personalizado de un kernel

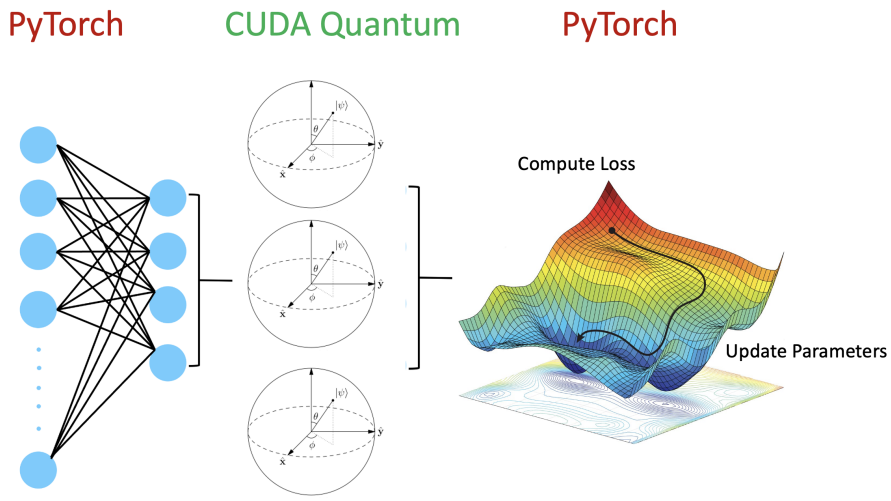

Figure 5: Enter Caption

Nota: Recuperado de *hybrid.png*, de https://nvidia.github.io/cuda-quantum/latest/examples/python/tutorials/hybrid_qnns.html, de NVIDIA, All rights reserved

```python
import cudaq
from typing import List

cudaq.set_target("nvidia")

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


# 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) ├
#      ╰───────╯ ╰───────╯

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

initial_cost_value = cost(initial_parameters)
print(initial_cost_value)
# Resultado de ejecución: 1.0

# Define a CUDA-Q optimizer.
optimizer = cudaq.optimizers.COBYLA()
optimizer.initial_parameters = initial_parameters

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

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")
```

Figure 6: Ejemplo de optimización del volteo de un qubit de 0 a 1 utilizando optimizadores de CUDA-Q

Nota: Adaptado del ejemplo de la documentación de CUDA-Q de NVIDIA

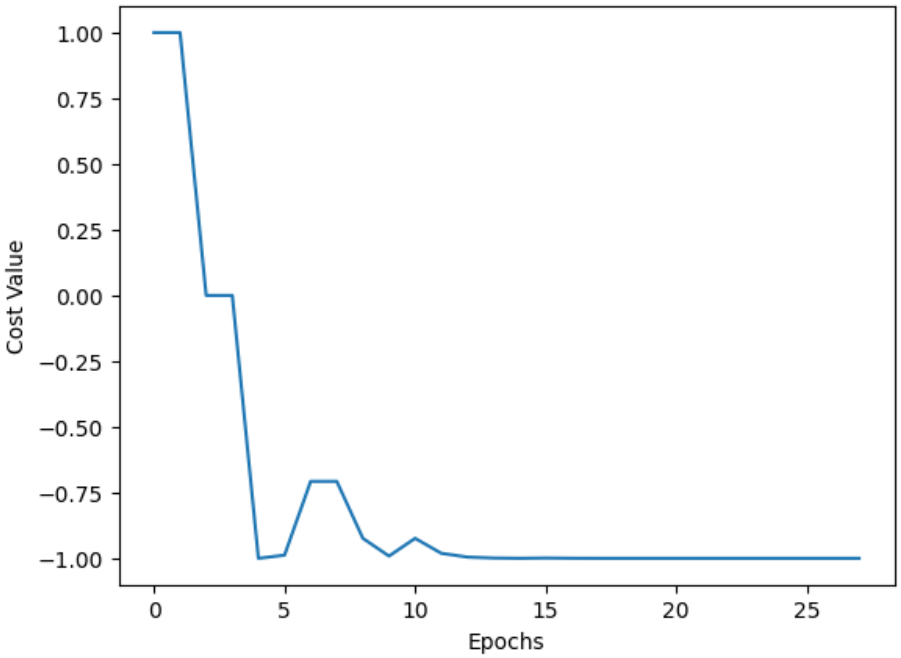

Figure 7: Gráfica que indica el resultado de la optimización producida en la figura 6.

Nota: Recuperado de https://nvidia.github.io/cuda-quantum/latest/examples/python/tutorials/cost_minimization.html, de NVIDIA, All rights reserved