Introducción a la Programación Paralela (CUDA)

Práctica 02 - Introducción a CUDA

Segundo Semestre, 2025

¡STOP!

Estas son las ¡SOLUCIONES! a los problemas **¿estás seguro que vas a revisar esto?** La idea de estas soluciones es que estén aquí para que puedas revisarlas cuando ya agotaste todas las posibilidades de poder resolver el problema por tu cuenta. De todas formas te recomendamos algunas cosas:

- **NO las leas sin intentarlo**: Porque si no lo intentaste y lees la solución, muchas veces vas a intentar llegar a la solución escrita y no intentar solucionar el problema. Puede parecer lo mismo, pero no lo es. La idea de la práctica es llegar a estas soluciones por tu cuenta.
- **¡Pero no me salen!**: El momento de real aprendizaje ocurre cuando intentás suficiente y, de repente, te sale. Es un momento de **¡AHA!** donde entendiste cómo resolver el problema. Siempre intentá un poquito más antes de leer las soluciones.
- **¡Ya lo pensé de varias formas diferentes!... ¡no me sale!**: Parte del aprendizaje es intentarlo. Si creés que lo intentaste suficiente y aún así no te salió, es el momento de leer la solución. Cada una de estas soluciones tiene alguna pista antes de la respuesta, intentá leer las pistas y ver si con eso te sale. Las pistas van de más difícil a más fácil, para que tengas un camino de aprendizaje.
- **¡Leí todas las pistas y no entiendo!**: No te preocupes, a veces pasa. Es momento de leer la solución, intentar pensar cómo llegar a esa solución y sino preguntar al docente.

Contents

2.1 Ejercicio: Suma de dos vectores

Si quisiéramos utilizar un thread para calcular la suma de los dos vectores. ¿Cómo se calcularía el índice del thread?

1. **PISTA 1**: El índice del thread se calcula teniendo en cuenta el tamaño del bloque, el número de bloque y el número de thread.
2. **PISTA 2**: El índice del thread se mueve entre 0 y el tamaño del bloque menos uno. El número de bloque se mueve entre 0 y el número de bloques menos uno.

SOLUCIÓN: El índice del thread se calcula como:

```
idx = blockIdx.x * blockDim.x + threadIdx.x;
```
(1)

donde `blockIdx.x` es el número de bloque, `blockDim.x` es el tamaño del bloque y `threadIdx.x` es el número de thread dentro del bloque.

2.2 Ejercicio: Sumar elementos adyacentes

Supongamos ahora que queremos usar un thread para sumar dos elementos contiguos de un vector (por simplicidad podemos pensar que el vector y el de bloque tiene una cantidad par de elementos). ¿Cómo se calcularía el índice del thread?

1. **PISTA 1**: El índice del thread tiene que ir de dos en dos.
2. **PISTA 2**: Usá una hoja y un lápiz / lapicera para calcular el índice del thread a mano a ver cuál podría ser.

SOLUCIÓN: El índice del thread se calcula como: `idx = (blockIdx.x * blockDim.x + threadIdx.x) * 2;`.

2.3 Ejercicio: Calcular el tamaño de los threads

1. **PISTA 1**: Primero hay que calcular cuántos bloques de threads serían necesarios.
2. **PISTA 1**: Dado que 1024 no es divisor de 8000 siempre vamos a tener más threads que elementos a procesar. ¿Cómo se calcularía la cantidad de bloques sabiendo esa información?

SOLUCIÓN: El total de threads es **8192**, que se calcula de la siguiente manera:

$$numBlocks = \left\lceil \frac{8000}{1024} \right\rceil * 1024 = 8 * 1024 = 8192 \quad (2)$$

2.4 Ejercicio: cudaMalloc (parte 1)

1. **PISTA 1**: El primer argumento de `cudaMalloc` es el puntero a la memoria que queremos reservar.

2. **PISTA 2**: El segundo argumento de `cudaMalloc` es el tamaño de la memoria que queremos reservar.

SOLUCIÓN: El segundo argumento de `cudaMalloc` es el tamaño de la memoria que queremos reservar, que se calcula como [1]:

```
sizeof(int) * v
```
(3)

2.5 Ejercicio: cudaMalloc (parte 2)

1. **PISTA 1**: El primer argumento de `cudaMalloc` es el puntero a la memoria que queremos reservar.

2. **PISTA 2**: El segundo argumento de `cudaMalloc` es el tamaño de la memoria que queremos reservar.

SOLUCIÓN: El primer argumento de `cudaMalloc` es el puntero a un puntero a la memoria que queremos reservar, ya que es la única forma en C para poder modificar el puntero (hacer una referencia). [2]. La expresión sería entonces:

```
(void**)&d_A
```
(4)

2.6 Ejercicio: Copia de memoria desde el *host* al *device*

1. **PISTA 1**: La función `cudaMemcpy` tiene cuatro argumentos (pensá qué argumentos necesitarías para poder copiar esa información)

2. **PISTA 2**: Los 3 primeros argumentos son fáciles de descubrir, porque son los punteros de la memoria de origen, la memoria de destino y el tamaño de la memoria a copiar. Siendo que `cudaMemcpy` es una función que funciona en ambos sentidos ¿cuál sería el cuarto argumento?.

SOLUCIÓN: La llamada a la API apropiada para esta copia de datos en CUDA es:

```
cudaMemcpy((void*)d_A, (void*)h_A, 3000, cudaMemcpyHostToDevice)
```
(5)

2.7 Ejercicio: Manejo de errores

1. **PISTA 1**: Las funciones de manejo de memoria de cuda devuelven siempre un tipo de error, por convención es un tipo de datos especial que termina con el sufijo `_t` ¿te acordás cuál es?

[1] CUDA documentation
[2] CUDA Documentation

SOLUCIÓN: El tipo de error que devuelven las funciones de manejo de memoria de CUDA es `cudaError_t`. Con lo cual la correcta definición de una variable (`err`) para recibir el error de las funciones de manejo de memoria de CUDA sería:

```
cudaError_t err;
```
(6)

2.8 Ejercicio: funciones en CUDA

1. **PISTA 1**: Las funciones de CUDA tienen modificadores para el host como para el device. ¿Te acordás cómo se llaman?

2. **PISTA 2**: ¿Hay realmente que declararlas dos veces?

SOLUCIÓN: Las funciones de CUDA tienen modificadores para el host como para el device, y estas son `__host__` y `__device__`. En realidad no es necesario declarar las funciones dos veces, ya que el compilador se encarga de generar el código para ambos dispositivos. Por ejemplo:

```
__host__ __device__ void foo(int *a) { ...  }
```
(7)