

Práctica 5. Introducción a CUDA.

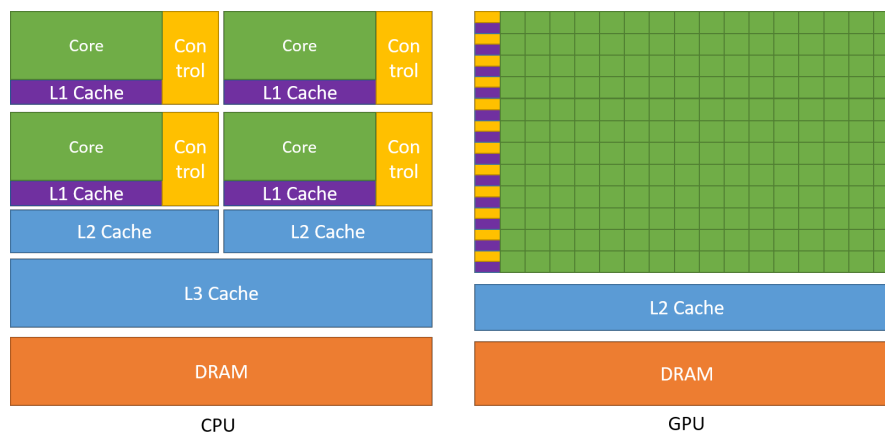
Objetivos de la práctica:

- Adquirir la capacidad de programar en lenguaje C con CUDA.
- Adquirir la capacidad de resolución de problemas paralelos en GPU.
- Comprender los conceptos básicos de CUDA: hilos, bloques, grids.
- Comprender los tipos de funciones de CUDA.
- Adquirir la capacidad de paralelizar un programa mediante GPU.

1 Introducción

Las *Graphics Processing Units* (GPU) tienen una capacidad de instrucciones mucho mayor y un ancho de banda de memoria más alto que una *Central Processing Unit* (CPU), dentro de un rango de precio y consumo de energía similares. Muchas aplicaciones aprovechan estas capacidades para obtener un funcionamiento más rápido en la GPU que en la CPU. Otros dispositivos, como las FPGA, también son muy eficientes energéticamente, pero tienen menos flexibilidad de programación que las GPUs. Esta diferencia de capacidad entre la GPU y la CPU existe porque están diseñadas con objetivos diferentes en mente. Mientras que las CPUs están diseñadas para sobresalir en la ejecución de una secuencia de operaciones, llamada hilo, y puede ejecutar unas pocas decenas de estos hilos en paralelo; la GPU está diseñada para sobresalir en la ejecución de miles de hilos en paralelo.

De esta manera, la GPU está especializada en cálculos altamente paralelos y, por lo tanto, está diseñada de manera que más transistores se dedican al procesamiento de datos en lugar de almacenamiento en caché y control de flujo de datos. La siguiente figura muestra una distribución de recursos comparando una CPU con una GPU. Dedicar más transistores al procesamiento de datos, por ejemplo, cálculos de coma flotante, es beneficioso para cálculos altamente paralelos; la GPU puede ocultar las latencias de acceso a memoria con cálculos, en lugar de depender de grandes cachés de datos y un control de flujo complejo para evitar largas latencias de acceso a memoria, ambas costosas en términos de transistores. En general, una aplicación tiene una combinación de partes paralelas y partes secuenciales, por lo que los sistemas se diseñan con una mezcla de GPUs y CPUs para maximizar el rendimiento general. Las aplicaciones con un alto grado de paralelismo pueden aprovechar esta naturaleza masivamente paralela de la GPU para lograr un rendimiento superior al de la CPU.



2 CUDA

La programación de CUDA en C es un enfoque para aprovechar las capacidades computacionales de las GPUs de NVIDIA para tareas de procesamiento de propósito general. Permite a los desarrolladores trasladar porciones paralelizables de su código a la GPU, aprovechando su arquitectura masivamente paralela para lograr mejoras significativas en la velocidad de procesamiento en comparación con la ejecución en CPU.

2.1 Gestionar la transferencia de datos

En el núcleo de la programación CUDA se encuentran los conceptos del host y el dispositivo. El **host** se refiere a la CPU, donde se ejecuta el programa principal, y el **dispositivo** se refiere a la GPU, que ejecuta los "kernels" de CUDA. Estos kernels son funciones que se ejecutan en paralelo por muchos hilos en la GPU. La gestión de la memoria es crucial en la programación CUDA. Los desarrolladores deben administrar la memoria entre el host y el dispositivo de manera eficiente. CUDA provee funciones como `cudaMemcpy` para transferir datos entre los espacios de memoria de la CPU y la GPU. Esta transferencia implica copiar datos desde la memoria del host al dispositivo (y viceversa) para asegurar que la GPU pueda acceder a los datos necesarios para el procesamiento. La optimización del código en CUDA implica consideraciones como minimizar las transferencias de datos entre el host y el dispositivo, optimizar los patrones de acceso a memoria para maximizar el rendimiento y utilizar de manera efectiva las capacidades de procesamiento paralelo de la GPU. Comprender las complejidades de la jerarquía de memoria, la organización de hilos y el diseño de kernels es crucial para lograr un rendimiento óptimo en aplicaciones CUDA.

2.2 Bloques y hilos

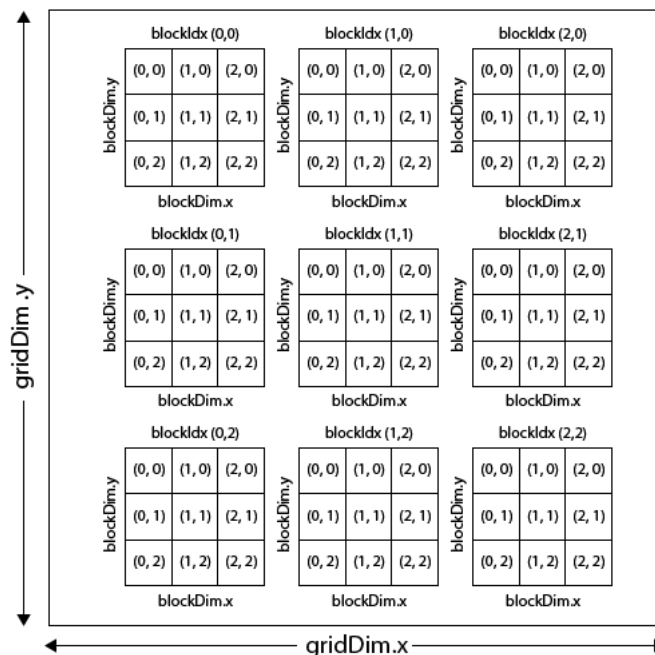
Los bloques y los hilos son elementos fundamentales en CUDA. Los **bloques** son grupos de hilos que se ejecutan de manera independiente y pueden ser programados para correr en multiprocesadores dentro de la GPU. Los hilos de un bloque pueden comunicarse y sincronizarse usando memoria compartida, lo que

permite la cooperación entre hilos que trabajan en los mismos datos. Los **hilos** son la unidad más pequeña de ejecución en CUDA. Ejecutan código de manera concurrente y realizan tareas específicas. En CUDA, miles o incluso millones de hilos pueden ejecutarse simultáneamente. Los hilos dentro del mismo bloque pueden cooperar y sincronizarse utilizando memoria compartida.

CUDA utiliza una estructura jerárquica para los hilos dentro de los bloques. Los hilos se organizan en una cuadrícula 1D, 2D o 3D dentro de un bloque, lo que permite la indexación multidimensional para acceder a datos y realizar cálculos. Por ejemplo, los bloques de hilos organizados en 1D tienen hilos organizados de manera lineal; mientras que los bloques de hilos organizados en 2D y 3D proporcionan una estructura de matriz o cubo para los hilos. Esta jerarquía ayuda a organizar eficientemente los hilos para procesar datos multidimensionales o realizar cálculos complejos en matrices, volúmenes u otras estructuras de datos indexadas. De esta forma, **blockDim.x**, **blockDim.y** y **blockDim.z** dan el número de hilos en un bloque para una dirección en particular. Cada hilo tiene un índice único dentro de su bloque. Para acceder a estos índices dentro de la jerarquía de hilos se tienen las variables **threadIdx.x**, **threadIdx.y** y **threadIdx.z**. Estos índices ayudan a los hilos a identificar su posición dentro de un bloque y se utilizan para determinar qué elementos de datos procesar.

Por otro lado, los **grids** de bloques constituyen una estructura fundamental para organizar y ejecutar tareas en paralelo en la GPU. Un grid se compone de múltiples bloques de hilos. Así, **gridDim.x**, **gridDim.y** y **gridDim.z** representan el número de bloques en el grid para una dirección en particular. También se tienen los valores de **blockIdx.x**, **blockIdx.y** y **blockIdx.z** para acceder a los bloques de esos índices dentro del grid.

CUDA Grid



Otro concepto que se tiene es el de **stride**, el incremento o tamaño de paso utilizado al acceder a elementos en una secuencia, matriz o región de memoria. Define el espacio entre elementos consecutivos a los que accede. Con esto se asegura que cada hilo procese un subconjunto único de los datos sin redundancia o conflicto con otros hilos. Por ejemplo, considera una situación donde tienes un vector de 1024 elementos y 256 hilos. Sin stride, si cada hilo procesara elementos consecutivos del 0 al 1023, solo se tratarían los primeros 256 elementos, dejando una parte significativa del vector sin procesar. Al introducir un enfoque basado en stride, los hilos pueden cubrir colectivamente todo el vector trabajando en subconjuntos de elementos no superpuestos, asegurando que todos los elementos estén incluidos en el cálculo de la media.

Por último, sincronizar hilos en CUDA es fundamental para garantizar una ejecución coordinada dentro de un bloque. Para ello se tienen mecanismos como la función `__syncthreads()`, que permite a los hilos dentro del mismo bloque sincronizar su ejecución. Cuando un hilo alcanza la llamada a `__syncthreads()`, espera hasta que todos los demás hilos en el bloque alcancen el mismo punto antes de continuar la ejecución. Este punto de sincronización asegura que los datos compartidos se actualicen correctamente y que los hilos no procedan con cálculos que dependan de los resultados intermedios de otros hilos hasta que esos resultados estén disponibles. El uso cuidadoso de `__syncthreads()` es esencial para evitar posibles condiciones de carrera o bloqueos, garantizando un comportamiento coherente y predecible entre los hilos dentro de un bloque durante la ejecución en paralelo. Otro caso es la función `atomicAdd()`. Es una operación atómica especial disponible en CUDA que realiza una suma atómica en una ubicación de memoria. Las operaciones atómicas son esenciales en la programación paralela para garantizar un comportamiento correcto cuando múltiples hilos acceden y modifican la misma ubicación de memoria de forma simultánea. En CUDA, `atomicAdd()` se utiliza para incrementar de manera atómica un valor en la memoria global (o memoria compartida) en una cantidad específica. Su sintaxis básica es: `int atomicAdd(int* address, int val);`

2.3 Tipos de funciones

Las funciones en CUDA desempeñan un papel fundamental para llevar a cabo el procesamiento paralelo en las GPUs. Existen diferentes tipos de funciones utilizadas en la programación CUDA. Los desarrolladores necesitan optimizar su código aprovechando el tipo correcto de funciones para tareas específicas, gestionando de manera eficiente el movimiento de datos entre el host y el dispositivo, y orquestando la ejecución en paralelo de kernels para maximizar la utilización y el rendimiento de la GPU. Dominar estas funciones y sus roles permite a los desarrolladores aprovechar todo el poder computacional de las GPUs para diversas aplicaciones, desde simulaciones científicas hasta aprendizaje automático y más allá.

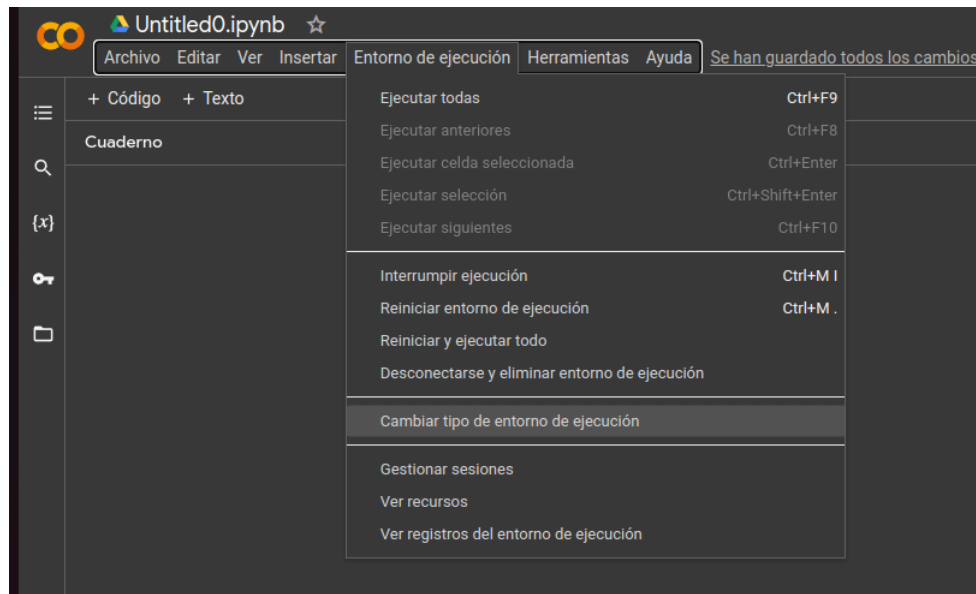
1. **Funciones Globales (Kernels):** Estas funciones están precedidas por `__global__` y son los puntos de entrada para la ejecución en paralelo en la GPU. Pueden ser invocadas desde la CPU, pero se ejecutan en la GPU. Estas funciones están diseñadas para realizar cálculos en paralelo y operan en un gran número de hilos organizados en bloques y rejillas (*grid*). Los desarrolladores escriben estas funciones para llevar a cabo

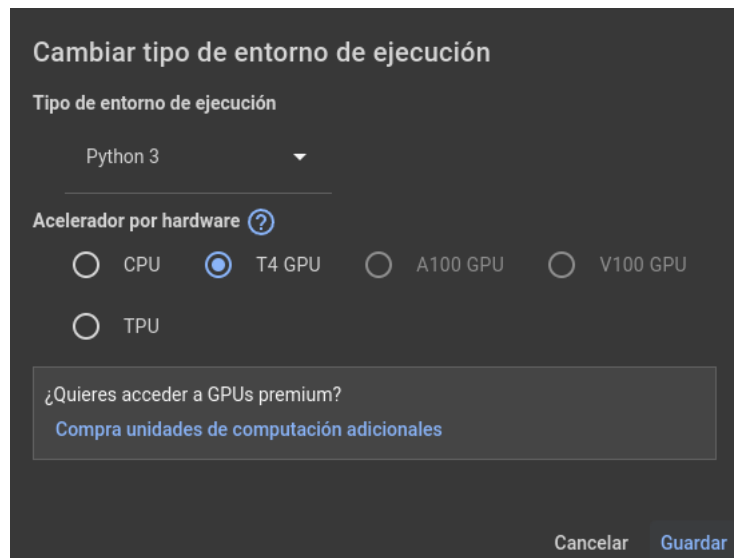
tareas específicas que pueden beneficiarse de una paralelización masiva, aprovechando la capacidad de la GPU para procesar datos simultáneamente. La sintaxis de configuración de ejecución de un kernel es la siguiente: $\lll M, T \ggg$, donde M es el número de bloques en el grid y T es el número de hilos por bloque. Para definir el número de bloques y de hilos se puede hacer con un entero o con **dim3**.

2. **Funciones de Dispositivo:** Estas son funciones que se ejecutan en la GPU pero solo pueden ser llamadas desde otras funciones en ejecución en la GPU. Se denotan con la palabra clave `__device__`. Las funciones de dispositivo se utilizan a menudo para encapsular cálculos repetitivos u operaciones compartidas entre diferentes funciones. Ayudan a modularizar el código y promueven la reutilización del código dentro de la GPU.
3. **Funciones de Host:** Estas funciones se ejecutan en la CPU (host) y pueden invocar funciones y gestionar transferencias de datos entre la CPU y la GPU. Las funciones de host son funciones típicas de C/C++ y se utilizan para orquestar la ejecución del código CUDA. Son responsables de configurar los datos, lanzar funciones y manejar los resultados devueltos desde la GPU.

3 Entorno de desarrollo

Para realizar esta práctica, se propo utilizar Google Colab. Una vez dentro hay que configurar el entorno para utilizar GPU. Para ello, se va *Entorno de ejecución* > *Cambiar tipo de entorno de ejecución* y se selecciona la opción de GPU.





Ahora podemos comprobar las características de la GPU con el comando **nvidia-smi**:

```
[1] !nvidia-smi

Tue Nov 21 20:01:23 2023
+-----+
| NVIDIA-SMI 525.105.17   Driver Version: 525.105.17   CUDA Version: 12.0   |
+-----+-----+
| GPU  Name            Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf      Pwr:Usage/Cap|      Memory-Usage | GPU-Util  Compute M. |
|=====+=====+
| 0 Tesla T4             Off          | 00000000:00:04.0 Off  | 0%          Default  |
| N/A  42C    P8        9W / 70W     | 0MiB / 15360MiB |           | MIG M. |
+-----+-----+

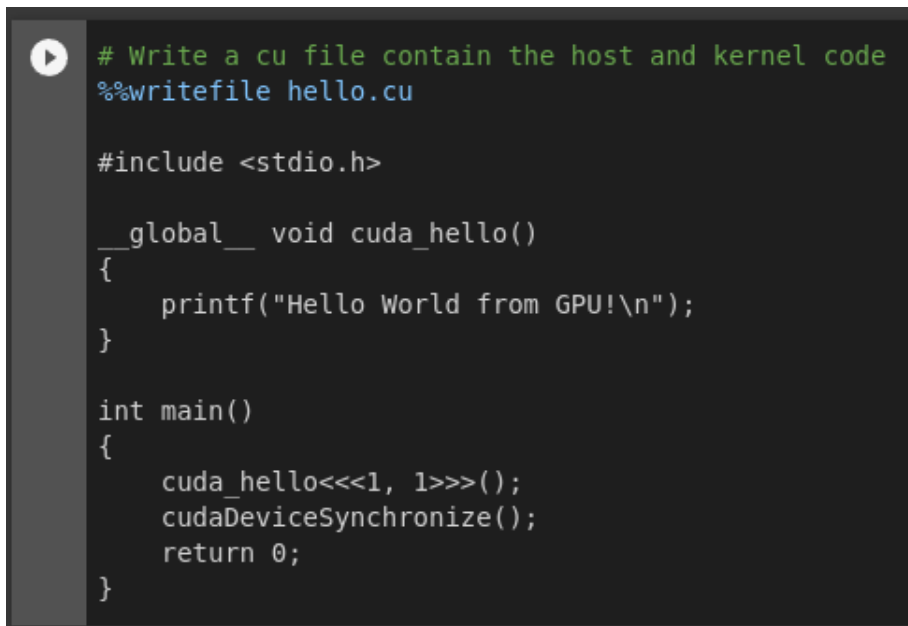
Processes:
+-----+
| GPU  GI    CI          PID  Type   Process name                      GPU Memory |
|   ID  ID                               Usage              |
+-----+
| No running processes found |
+-----+
```

Para compilar los programas vamos a utilizar **nvcc**, que es similar a gcc. Podéis comprobar vuestra versión con el comando `nvcc --version`.

```
!nvcc --version

nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2022 NVIDIA Corporation
Built on Wed Sep 21 10:33:58 PDT 2022
Cuda compilation tools, release 11.8, V11.8.89
Build cuda_11.8.r11.8/compiler.31833905_0
```

Por último, los programas se puede escribir en el Google Colab. Para guardarlos hay que escribir `%%writefile nombre_archivo.cu` al principio del código. Para compilarlo, se utiliza el comando `nvcc` de forma similar a `gcc`: `nvcc hello.cu -o hello`. Para ejecutarlo es igual que en las prácticas de C, es decir, `./hello`.



```
# Write a cu file contain the host and kernel code
%%writefile hello.cu

#include <stdio.h>

__global__ void cuda_hello()
{
    printf("Hello World from GPU!\n");
}

int main()
{
    cuda_hello<<<1, 1>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

4 Ejemplos

4.1 Hello World

```
#include <stdio.h>

__global__ void cuda_hello()
{
    printf(" Hello World from GPU!\n");
}

int main()
{
    cuda_hello<<<1, 1>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

cudaDeviceSynchronize() es una función de CUDA que actúa como un punto de sincronización para el hilo de CPU que la llama. Su propósito es asegurarse de que la GPU haya completado todas sus tareas antes de permitir que el hilo de CPU continúe con la ejecución adicional.

4.2 Thread ID

```
#include <stdio.h>

__global__ void cuda_hello()
{
    printf("Hello World from thread %d %d\n", threadIdx.x, threadIdx.y);
}

int main()
{
    dim3 threadsPerBlock(3, 3);
    dim3 gridSize(3, 3);
    cuda_hello<<<gridSize, threadsPerBlock>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

Con **dim3 threadsPerBlock(3, 3);** definimos el tamaño de los bloques como matrices de 3x3 hilos.

4.3 Gestión de memoria

```
#include <stdio.h>

#define N 10

__global__ void sum_to_vector(float *A, float *B)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;

    if (index < N)
    {
        B[index] = A[index] + N;
    }
}

int main()
{
    float h_A[N];
    float h_B[N];

    for (int i = 0; i < N; ++i)
    {
        h_A[i] = (float)(i + 1);
    }

    float *d_A;
    float *d_B;
    cudaMalloc((void **)&d_A, N * sizeof(float));
    cudaMalloc((void **)&d_B, N * sizeof(float));
}
```



```

    cudaMemcpy(d_A, h_A, N * sizeof(float), cudaMemcpyHostToDevice);

    int blockSize = 32;
    int gridSize = (N + blockSize - 1) / blockSize;
    sum_to_vector<<<gridSize, blockSize>>>(d_A, d_B);

    cudaMemcpy(h_B, d_B, N * sizeof(float), cudaMemcpyDeviceToHost);

    printf("Resultant Vector:\n");
    for (int i = 0; i < N; ++i)
    {
        printf("%.2f\t", h_B[i]);
    }
    printf("\n");

    cudaFree(d_A);
    cudaFree(d_B);

    return 0;
}

```

- **cudaMemcpyHostToDevice**: transferimos memoria de CPU a GPU
- **cudaMemcpyDeviceToHost**: transferimos memoria de GPU a CPU

4.4 Suma de vectores

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N 10

__global__ void sum_vectors(float *A, float *B, float *C)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;

    if (index < N)
    {
        C[index] = A[index] + B[index];
    }
}

int main()
{
    float h_A[N], h_B[N], h_C[N];

    for (int i = 0; i < N; ++i)
    {

```

```
        h_A[i] = rand() % 10 + 1;
        h_B[i] = rand() % 10 + 1;
    }

    float *d_A, *d_B, *d_C;

    cudaMalloc((void **)&d_A, N * sizeof(float));
    cudaMalloc((void **)&d_B, N * sizeof(float));
    cudaMalloc((void **)&d_C, N * sizeof(float));

    cudaMemcpy(d_A, h_A, N * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, N * sizeof(float), cudaMemcpyHostToDevice);

    int blockSize = 256;
    int gridSize = (N + blockSize - 1) / blockSize;
    sum_vectors<<<gridSize, blockSize>>>(d_A, d_B, d_C);

    cudaMemcpy(h_C, d_C, N * sizeof(float), cudaMemcpyDeviceToHost);

    printf("Resultant Vector:\n");
    for (int i = 0; i < N; ++i)
    {
        printf("%.2f + %.2f = %.2f\t", h_A[i], h_B[i], h_C[i]);
    }
    printf("\n");

    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    return 0;
}
```

5 Ejercicios propuestos

5.1 Inicializa un vector

Escribe un programa en C que inicializa un vector de tamaño 128, utilizando un kernel CUDA, con los identificadores de los hilos. Utiliza un tamaño de bloque de 8. Repite el ejercicio utilizando un tamaño de bloque de 8 y tamaño de grid 8. **Nota:** tenéis que usar stride en la segunda parte.

5.2 Suma dos vectores

Escribe un programa en C que sume dos vectores de tamaño 128 utilizando un kernel CUDA. Los vectores se tienen que inicializar con números aleatorios entre 1 y 10. El resultado se tiene que guardar en un tercer vector.

5.3 Suma los elementos de un vector

Escribe un programa en C que suma los elementos de un vector de tamaño 1024 utilizando un kernel de CUDA. El vector se tiene que inicializar con números aleatorios entre 5 y 15. El tamaño del grid tiene que ser de 16 bloques y el tamaño de bloque de 16 hilos. **Nota:** tenéis que usar stride y podéis usar atomicAdd.

5.4 Suma dos matrices

Escribe un programa en C que sume dos matrices 100x100 utilizando un kernel de CUDA. Las matrices se tienen que inicializar con números aleatorios entre 10 y 20. El resultado se tiene que guardar en una tercera matriz. **Nota:** tenéis que usar dim3 para definir el tamaño del grid y de los bloques.