A close up of a document

Description automatically generated with low confidence**Integrantes:**

Alejandro Díaz Pereira

Bertha Brenes Brenes

Ignacio Vargas Campos

Carlos Adrián Araya Ramírez

Shakime Richards Sparks

A picture containing text, screenshot, font, algebra

Description automatically generated

Significa Compute Unified Device Architecture, la cual fue desarrollada por NVIDIA Compute Unified Device Architecture para trabajar tareas de computación paralela las cuales utilizan GPU (Unidades de procesamiento gráfico) en tareas de procesamiento intensivo en paralelo, como el procesamiento de imágenes, la simulación científica, el aprendizaje profundo y el análisis de datos [1].

A picture containing text, screenshot, font, algebra

Description automatically generated

Un kernel CUDA es una función especializada que está diseñada para ser ejecutada en paralelo por un gran número de hilos en la GPU utilizando la plataforma de cómputo paralelo CUDA. En la arquitectura CUDA, la GPU se utiliza para realizar cálculos altamente paralelos, lo que puede acelerar significativamente ciertos tipos de tareas. La función del kernel se define utilizando un lenguaje similar a C con extensiones específicas proporcionadas por CUDA y se invoca con un número específico de bloques y hilos por bloque, y es responsabilidad del programador definir esta organización en función del problema que se está resolviendo. Cada hilo ejecuta el mismo código, pero opera sobre datos diferentes.

Los kernels CUDA se utilizan con frecuencia para tareas intensivas en cómputo que pueden ser paralelizadas, como operaciones de matriz, procesamiento de imágenes, simulaciones y algoritmos de aprendizaje automático. Al aprovechar la arquitectura masivamente paralela de la GPU, los kernels CUDA pueden lograr aceleraciones significativas en comparación con la ejecución del mismo código en una CPU [1].

A picture containing text, screenshot, font, algebra

Description automatically generated

Cuando se lanza un programa CUDA, se crean múltiples hilos y se organizan en bloques de hilos. Estos bloques de hilos se agrupan en una malla (grid) y a cada hilo se le asigna una identificación única de hilo, lo que le permite acceder y procesar una porción diferente de los datos de entrada y cada bloque es atendido por un multiprocesador. El sistema de tiempo de ejecución CUDA administra la ejecución del kernel en la GPU, distribuyendo la carga de trabajo entre los recursos de cómputo disponibles.  Esto permite que los hilos de un mismo bloque puedan sincronizar su ejecución y compartir datos de forma eficiente usando memoria local de baja latencia [2].

A picture containing text, screenshot, font, algebra

Description automatically generated

La plataforma Jetson Nano es un sistema en módulo desarrollado por NVIDIA para aplicaciones de inteligencia artificial y computación de borde. Está compuesto por un procesador ARM Cortex-A57 de cuatro núcleos, una GPU NVIDIA Maxwell de 128 núcleos y 4 GB de memoria LPDDR4.

La arquitectura de la plataforma Jetson Nano se divide en dos partes principales: la CPU y la GPU. La CPU es responsable de las tareas de procesamiento general, mientras que la GPU se encarga de las tareas de cómputo intensivo, como el procesamiento de imágenes y la detección de objetos.

Además, la plataforma Jetson Nano cuenta con una amplia variedad de puertos y conectores, incluyendo USB 3.0, HDMI y GPIO, lo que la hace ideal para una amplia variedad de aplicaciones de inteligencia artificial y robótica.

A picture containing text, screenshot, font, algebra

Description automatically generated

Una vez instalado el CUDA Toolkit, se puede utilizar el compilador nvcc para compilar el código CUDA. El proceso de compilación implica especificar la ubicación de los archivos de inclusión y bibliotecas necesarios, así como las opciones de optimización, arquitectura y enlazado. También es importante asegurarse de que cualquier biblioteca externa utilizada en el código esté instalada y configurada correctamente para su uso con CUDA. Para compilar un código se utiliza el siguiente comando:

Esto compilará el archivo my\_code.cu y generará un ejecutable llamado my\_executable. Una vez compilado el código CUDA y generado el archivo ejecutable, para ejecutar el programa en la GPU compatible se debe abrir una terminal y navegar hasta el directorio donde se encuentra el archivo ejecutable. Luego, el programa puede ser ejecutado en la GPU utilizando el siguiente comando:

Como se observa, el nombre del archivo ejecutable debe ser el que se le haya dado durante la generación con el compilador nvcc. Es importante verificar que la GPU esté correctamente instalada y configurada en el sistema antes de ejecutar el programa [3].

*NOTA:* *Es necesario tener en cuenta que el código CUDA tiene que ser diseñado para ser paralelizable para que el programa se ejecute en la GPU. Si el código no se puede paralelizar, entonces el programa se ejecutará en la CPU en lugar de la GPU, lo que puede ser menos eficiente.*

A picture containing text, screenshot, font, document

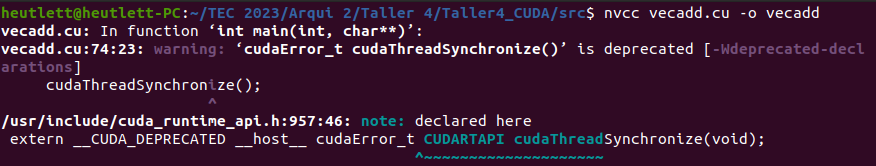
Description automatically generated

La operación que se realiza con los vectores de entrada es una suma vectorial, se suman los elementos de los vectores a y b de forma paralela y el resultado se almacena en el vector c. Esta operación se realiza tanto en la CPU como en la GPU, en la función vecAdd\_h y en la función del kernel vecAdd, respectivamente. La diferencia es que la CPU realiza esta operación de manera secuencial, mientras que la GPU lo hace en paralelo mediante hilos.

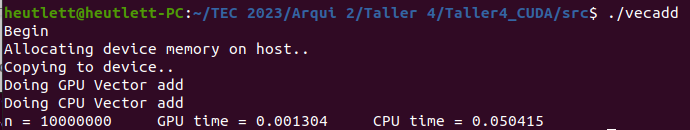
Los vectores de entrada se identifican por la forma en que se almacenan en memoria. Los punteros que apuntan a la memoria del CPU se inicializan utilizando la función malloc, mientras que los punteros que apuntan a la memoria del GPU se inicializan utilizando la función cudaMalloc. Para inicializar los datos, el CPU utiliza un bucle for que recorre todos los elementos de los vectores y los inicializa con valores aleatorios. Para copiar los datos de los vectores de entrada desde la memoria del CPU a la memoria del GPU, se utiliza la función cudaMemcpy, que copia los valores de a y b y los almacena en a\_d y b\_d, respectivamente.

Respecto a la forma en que se realiza el procesamiento en paralelo, en la CPU, la suma vectorial se realiza en la función vecAdd\_h, que recibe los vectores a y b como argumentos, realiza la suma vectorial elemento a elemento y guarda el resultado en el vector c2. Esto se hace en un ciclo for que recorre todos los elementos de los vectores. En la GPU, la suma vectorial se realiza en la función del kernel vecAdd, que también recibe los vectores a y b, así como el vector c donde se guardará el resultado de la suma. Para realizar el procesamiento paralelo en el GPU, se utilizan los conceptos de bloques y rejillas (grids). Cada bloque es un grupo de hilos que ejecutan la función del kernel de manera paralela, lo cual implica que la función del kernel se ejecuta en paralelo en varios hilos en la GPU, esto permite acelerar la operación de suma vectorial en comparación con la CPU.

A picture containing text, screenshot, font, document

Description automatically generated

A picture containing text, screenshot, font, document

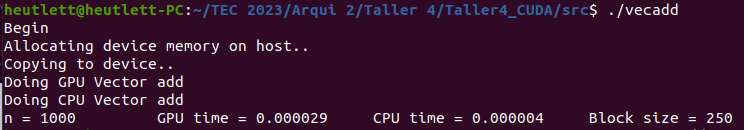
Description automatically generated

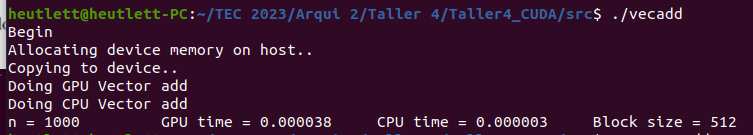
Este código realiza la suma de dos vectores, "A" y "B", almacenados en el host (CPU), y almacena el resultado en un tercer vector "C". Para acelerar la operación, utiliza el paralelismo de la GPU a través de la tecnología CUDA de NVIDIA. Para ello, el programa define una función kernel, "vecAdd()", que se ejecuta en la GPU y suma los vectores "A" y "B", almacenándolos en "C".

El programa también realiza la misma operación de suma en el host (CPU) utilizando la función "vecAdd\_h()". Finalmente, el programa mide el tiempo de ejecución de ambas operaciones, tanto en la GPU como en el host, y los imprime junto con el tamaño del bloque utilizado en la ejecución del kernel.

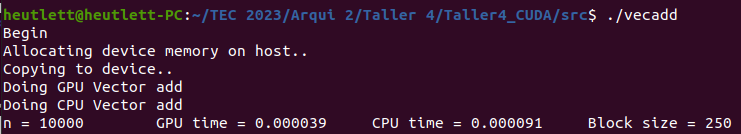
A picture containing text, screenshot, font, document

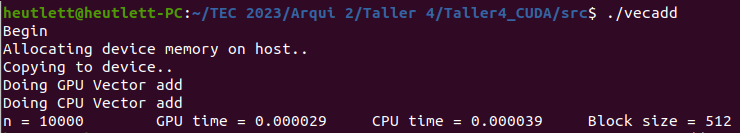
Description automatically generatedN = 1000



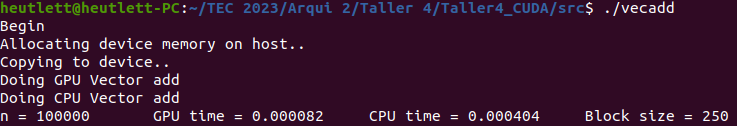


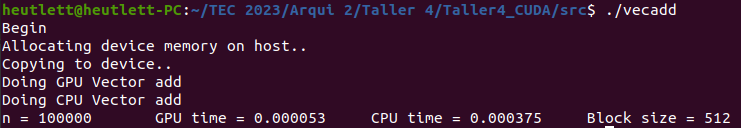
N = 10000

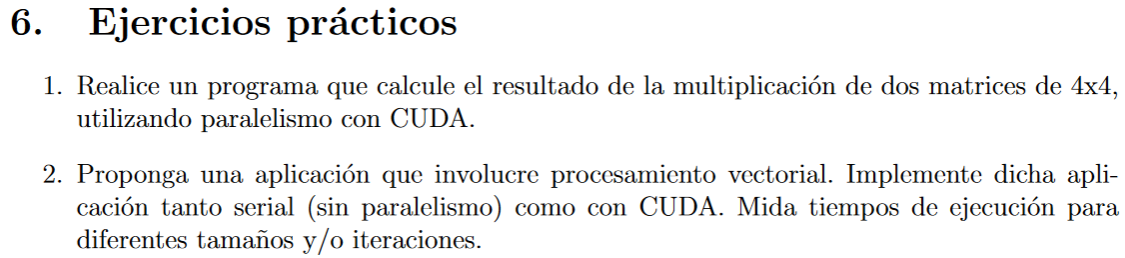


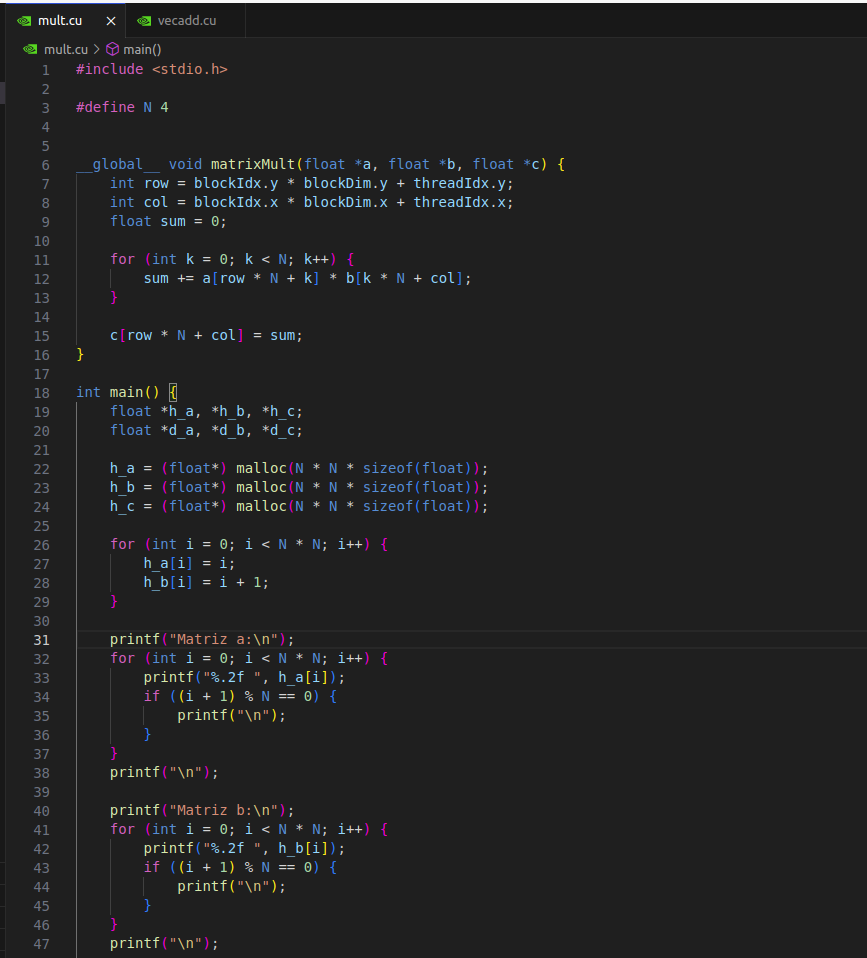


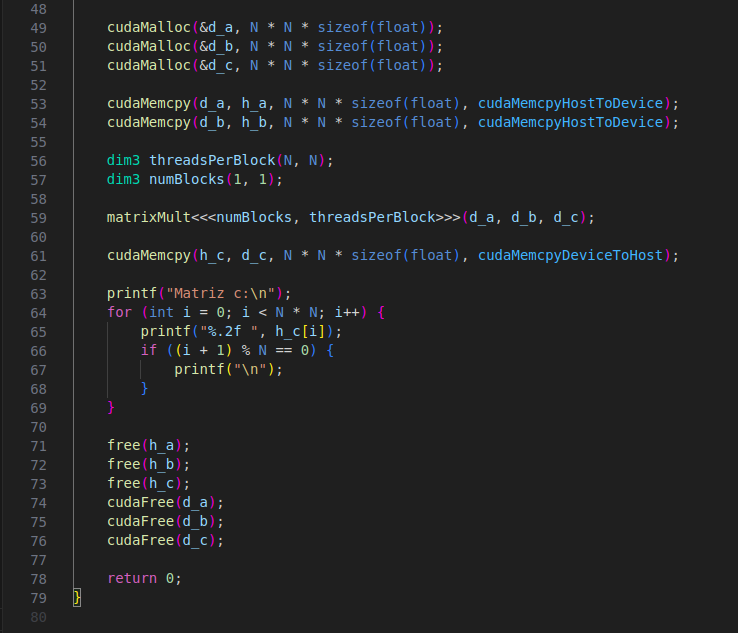
N = 100000



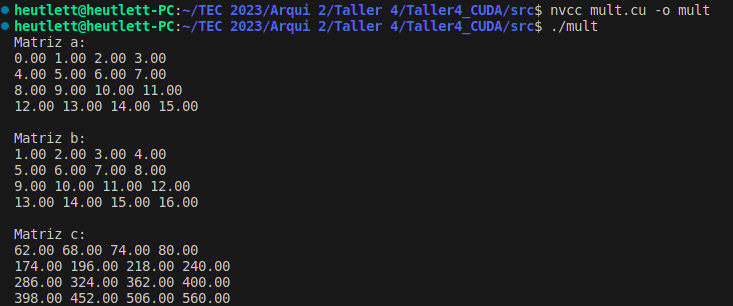


**Código:**



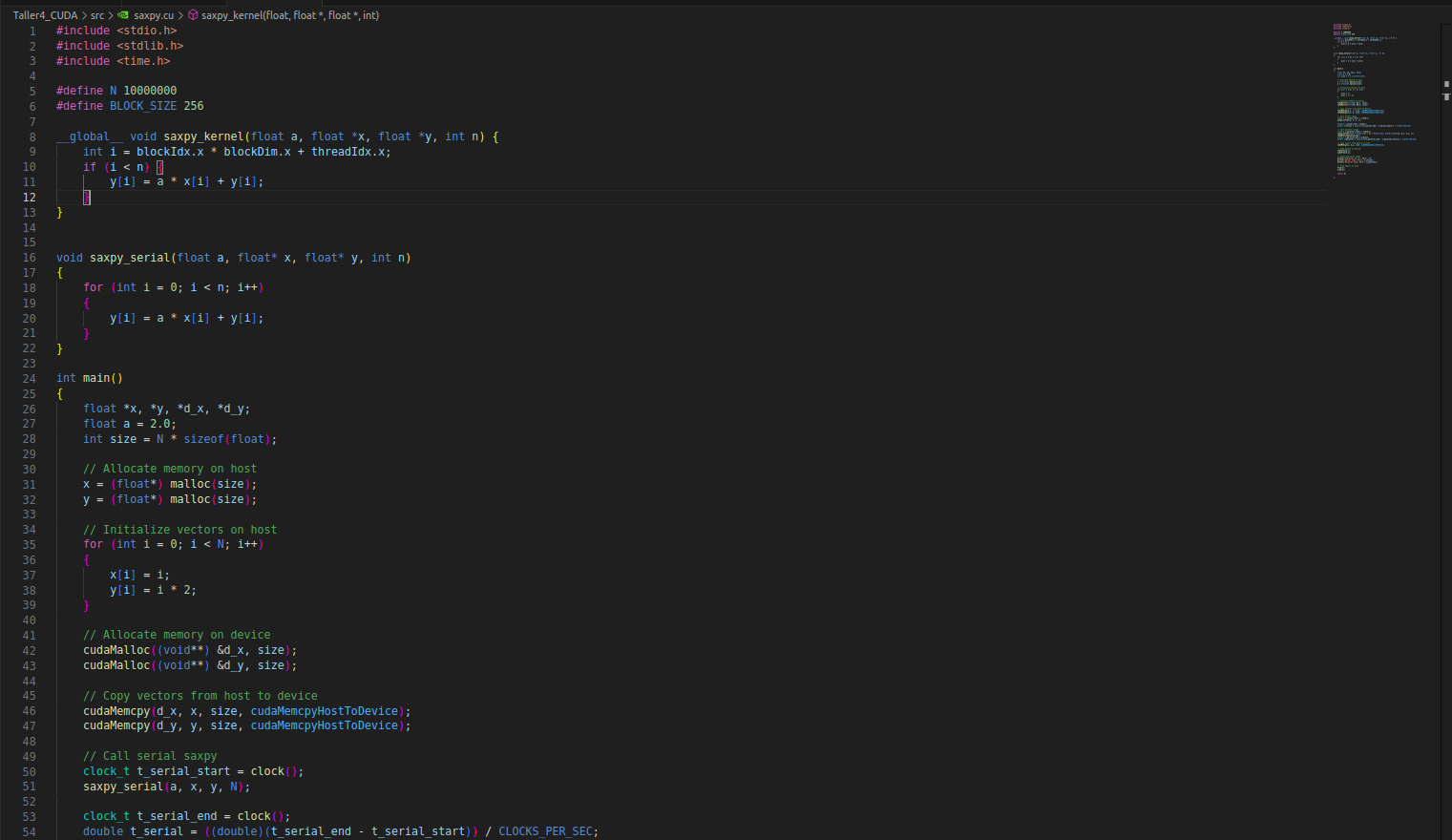


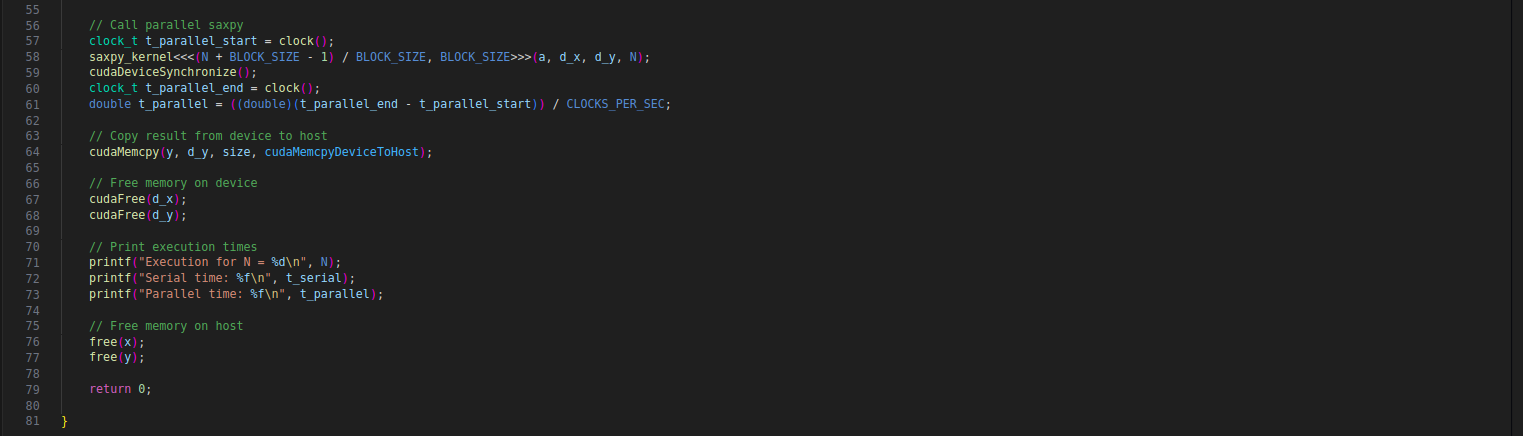
**Ejecución:**



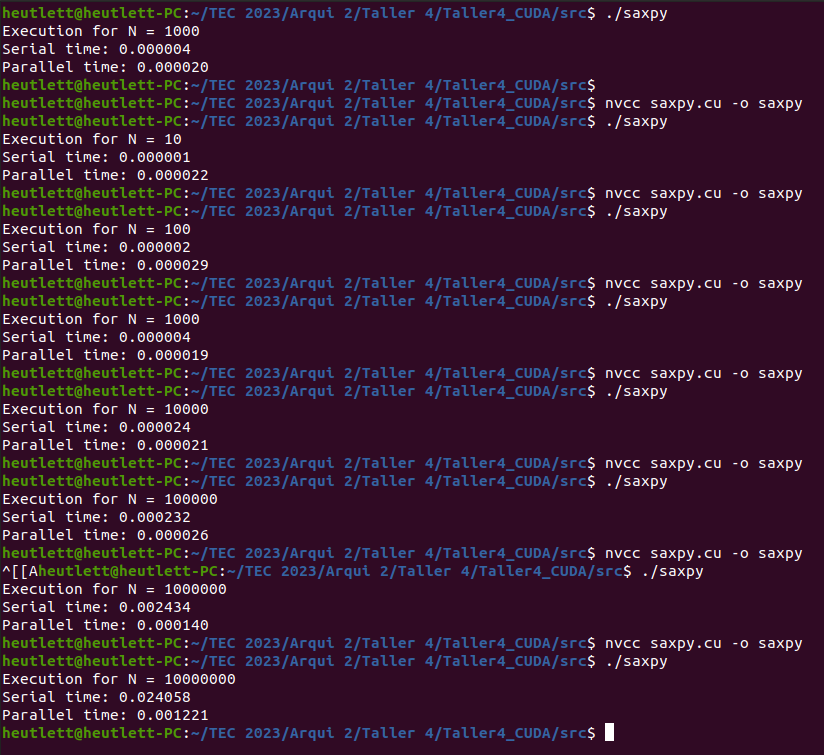
A picture containing text, font, screenshot, algebra

Description automatically generated**Código:**





**Ejecución:**



**References**

[1] NVIDIA. CUDA C++ Programming Guide. <https://docs.nvidia.com/cuda/cuda-c-programming-guide/>

[2] J. Roca, "Núcleos CUDA de NVIDIA: ¿qué son y para qué sirven?," HardZone, 21 de junio 2022. <https://hardzone.es/marcas/nvidia/nucleos-cuda/>

[3] J. Sanders and E. Kandrot, CUDA by Example : An Introduction to General-Purpose GPU Programming. Sydney: Pearson Education, Limited, 2010. <https://edoras.sdsu.edu/~mthomas/docs/cuda/cuda_by_example.book.pdf>

‌

‌