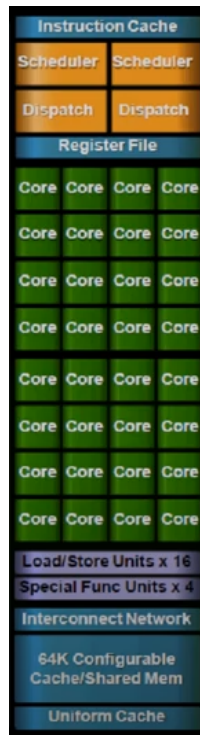


CUDA C

▼ Versión de CUDA

La elección de la versión del CUDA Toolkit depende de la GPU NVIDIA que tengamos instalada. El compilador incluido en cada versión del CUDA Toolkit (**nvcc**) genera código para una determinada arquitectura de GPU.

la arquitectura de cada GPU se clasifica en función de su **Capacidad de Cómputo**. Esta *capacidad de cómputo* determina ciertas características de hardware.



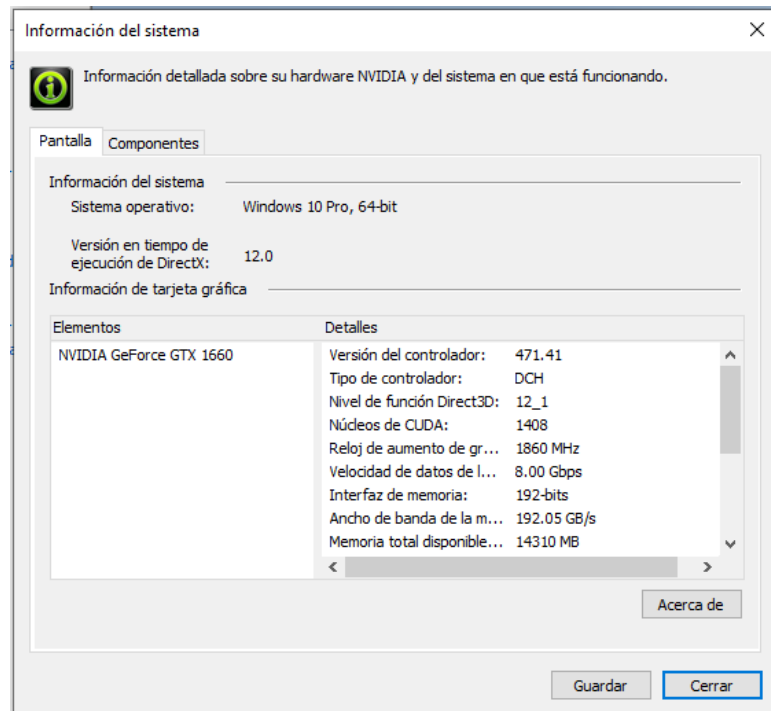
Capacidad de Cómputo	Nombre de la Arquitectura	Núcleos por multiprocesador (SP)
1.x	Tesla	8
2.0	Fermi	32
2.1	Fermi	48
3.x	Kepler	192
5.x	Maxwell	128
6.x	Pascal	64
7.0	Volta	64
7.5	Turing	64
8.0	Ampere	64

No toda las versiones del CUDA Toolkit soportan todas las Capacidades de Cómputo

Versión de CUDA	Capacidad de Cómputo Soportada
-----------------	--------------------------------

Versión de CUDA	Capacidad de Cómputo Soportada
6.5	1.0 - 5.3
7.5	2.0 - 5.3
8.0	2.0 - 6.2
9.2	3.0 - 7.2
10.2	3.0 - 7.5
11.0	5.0 - 8.0

Para conocer nuestra GPU hay que abrir el panel de control de NVIDIA




Nos fijamos en la capacidad de Memoria

Después, buscamos la versión compatible en la siguiente página:

CUDA GPUs

Your GPU Compute Capability Are you looking for the compute capability for your GPU, then check the tables below. You can learn more about Compute Capability here. NVIDIA GPUs power millions of desktops, notebooks, workstations and supercomputers around the world, accelerating computationally-

<https://developer.nvidia.com/cuda-gpus>



En mi caso (una GTX 1660), no se encuentra listada en la página de NVIDIA. Buscando en foros encontré que soporta 7.5 SP, por lo que la versión más nueva de CUDA también me funciona.

Entorno de Desarrollo

Se recomienda utilizar **Microsoft Visual Studio** en entornos Windows pues el SDK de CUDA cuenta con proyectos ya configurados para Microsoft Visual Studio. Es importante revisar la versión de Visual Studio que soporta la versión de CUDA elegida.

De igual manera, primero se debe instalar Visual Studio y posteriormente CUDA Toolkit.


Si queremos programar en CUDA pero no contamos con una tarjeta gráfica NVIDIA, necesitamos compilar con la opción de **emulador**. Sin embargo, esta opción sólo es posible hasta la versión **2.3** de CUDA.


Descargar CUDA Toolkit

Para descargar CUDA Toolkit, entramos en la siguiente página web.

CUDA Toolkit

Free Tools and Trainings for Developers Get exclusive access to hundreds of SDKs, technical trainings, and opportunities to connect with millions of like-minded developers, researchers, and students. Learn More CUDA Toolkit Develop, Optimize and Deploy GPU-Accelerated Apps The NVIDIA® CUDA® Toolkit


 <https://developer.nvidia.com/cuda-toolkit>



Nvidia cuenta con una serie de ejemplos para empezar a trabajar con CUDA.


GitHub - NVIDIA/cuda-samples: Samples for CUDA Developers which demonstrates features in CUDA Toolkit

Samples for CUDA Developers which demonstrates features in CUDA Toolkit. This version supports CUDA Toolkit 11.6. This section describes the release notes for the CUDA Samples on GitHub only. Added new folder structure for samples Added support of Visual Studio 2022 to all samples supported on Windows.

 <https://github.com/nvidia/cuda-samples>

NVIDIA/cuda-samples

Samples for CUDA Developers which demonstrate features in CUDA Toolkit

 3 Contributors 56 Issues 2k Stars 885 Forks

▼ Acceso a la Memoria

CUDA es la arquitectura de las tarjetas gráficas de la empresa NVIDIA. El objetivo es hacer accesible las tarjetas gráficas para realizar cómputo de propósito general.

La programación en CUDA asume un modelo de computación **heterogéneo**:

- Distintos recursos de hardware
- Distinto repertorio de instrucciones

EL modelo estará formado por una CPU y una GPU.

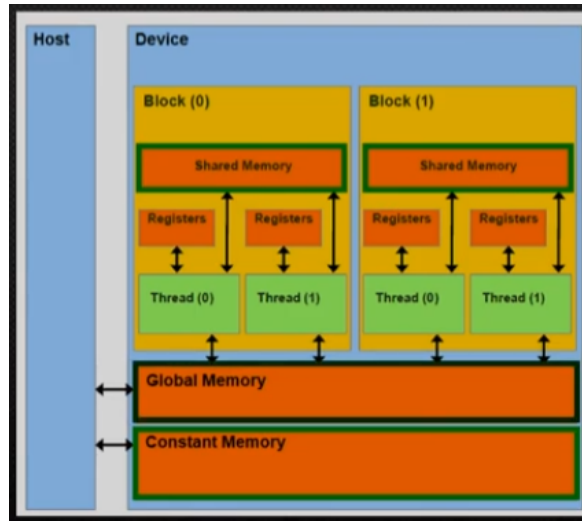
El programa principal se ejecutará en CPU. Sin embargo, un trozo del código se ejecutará dentro del *GPU*. Este trozo de código se llama **kernel**.

Terminología

- **Host**: CPU y su memoria.
- **Device**: GPU y su memoria.
- **Kernel**: Función principal que se ejecuta en el *device*.
- **Hilo**: Cada una de las copias del *kernel* que se ejecuta en el *device*.

Reserva dinámica de memoria

La memoria del *device* accesible desde el *host* para leer/escribir se la denomina **Memoria Global**. Dentro del *device* hay otros espacios de memoria que también se pueden utilizar para mejorar las presentaciones de nuestra aplicación.



Desde el `main()` ejecutado en el *host* se puede acceder a los datos del *device* si se conocen sus direcciones en la memoria global. Esto se hace reservando memoria del *device* de forma **dinámica**. La función en CUDA C que nos permite reservar espacio es `cudaMalloc()`.

Esta función nos permite reservar una determinada cantidad de espacio y obtener la dirección de comienzo de dicho espacio. Las direcciones se almacenan en **apuntadores**.

La sintaxis de la función es:

```
cudaMalloc(void **devPtr, size_t size);
```

Se puede mover la memoria entre *host* y el *device* conociendo las direcciones de origen y destino.

Las direcciones de memoria del *host* y *device* son incompatibles entre sí. Se recomienda utilizar un “prefijo” para identificar las variables correspondientes al *host* y al *device*.

Una vez teniendo el espacio reservado y conocemos las direcciones de origen y destino de los datos, se pueden mover utilizando la función `cudaMemcpy()`.

```
/**
 * void *dst Dirección de destino de los datos.
 * void *src Dirección de origen de los datos.
 * size_t count Número de bytes a transferir.
 * ... tipo Tipo de transferencia
 */
cudaMemcpy(void *dst, void *src, size_t count, cudaMemcpyKind tipo);
```

El tipo de transferencia se especifica con una palabra clave que corresponde al sentido de transferencia de datos:

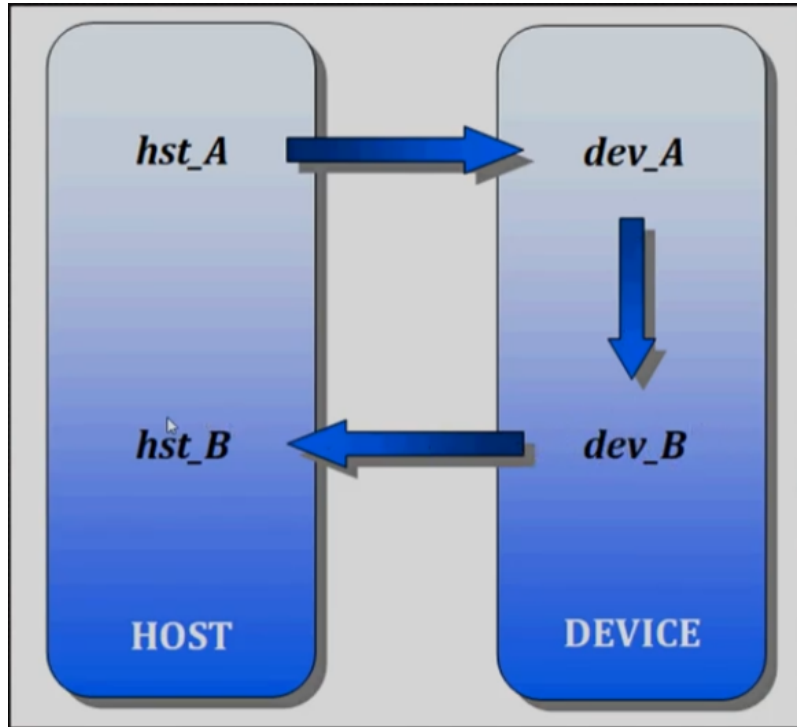
Tipo de transferencia	Sentido de la transferencia
<code>cudaMemcpyHostToHost</code>	host → host
<code>cudaMemcpyHostToDevice</code>	host → device
<code>cudaMemcpyDeviceToHost</code>	device → host
<code>cudaMemcpyDeviceToDevice</code>	device → device

Para liberar la memoria reservada en el *device* utilizamos la función `cudaFree()`.

```
// void *devPtr Dirección de memoria que queremos liberar  
cudaFree(void *devPtr);
```

Ejemplo

Ejemplo para ver cómo utilizar la memoria global para leer y escribir datos. Inicializamos un arreglo de tipo `float` y transferimos los datos entre la memoria del host y la memoria del device como en el siguiente esquema.



```
#include<stdio.h>  
#include<stdlib.h>  
#include<cuda_runtime.h>  
  
#define N 16  
  
int main(int argc, char** argv) {  
    float *hst_A, *hst_B;  
    float *dev_A, *dev_B;  
  
    //reservamos host  
    hst_A = (float*)malloc(N*sizeof(float));  
    hst_B = (float*)malloc(N*sizeof(float));  
  
    //reservamos device  
    cudaMalloc((void*)&dev_A, N * sizeof(float));  
    cudaMalloc((void*)&dev_B, N * sizeof(float));  
  
    //inicialización  
    for(int i= 0; i< N; i++) {  
        hst_A[i] = (float)rand() / RAND_MAX;  
        hst_B[i] = 0;  
    }  
  
    //copia de datos  
    cudaMemcpy(dev_A, hst_A, N*sizeof(float), cudaMemcpyHostToDevice);  
    cudaMemcpy(dev_B, dev_A, N*sizeof(float), cudaMemcpyDeviceToDevice);  
    cudaMemcpy(hst_B, dev_B, N*sizeof(float), cudaMemcpyDeviceToHost);  
}
```

```

    cudaFree(dev_A);
    cudaFree(dev_B);

    return 0;
}

```

▼ Lanzar un Kernel

CUDA es adecuado para algoritmos que tengan un gran paralelismo a nivel de datos, es decir, problemas donde realizamos la misma operación sobre conjuntos de datos distintos. Por ejemplo, operaciones sobre vectores y matrices.

Una aplicación en CUDA tendrá una parte que se ejecute en el Host (CPU) de manera serial, y una parte que se ejecute en el device (GPU) de manera paralela.

Las aplicaciones CUDA se escriben en archivos con terminación **.cu**.

La estructura de una aplicación CUDA puede tener la siguiente estructura:

- Definición de Funciones
 - Funciones ejecutadas en el HOST
 - Funciones llamadas desde el Host y ejecutadas en el Device.
 - Funciones llamadas desde el Device y ejecutadas en el Device.
- Reserva de recursos.
- Inicialización de datos.
- Copia de Datos CPU → GPU (Los datos pueden inicializarse igual en el GPU)
- Llamada al Kernel
- Copia de datos GPU → CPU
- Presentación de Resultados
- Liberación de recursos
- Salida

El archivo **.cu** contiene tanto el código de la función **kernel** como el de cualquier otro tipo de función.

La función **main()** incluye el código secuencial junto con todas las llamadas a las funciones CUDA para la reserva de recursos, copia de datos, lanzamiento de **kernel**, etc.

El fichero ejecutable final contiene instrucciones que se van a ejecutar en la CPU e instrucciones que se van a ejecutar en la GPU.

El **nvcc** es el encargado de separa ambos tipos.

Para que el compilador distinga entre las funciones ejecutables en la CPU de las ejecutables en la GPU se utilizan **etiquetas**.

Especificador	Llamada desde	Ejecutada en	Ejemplo de sintaxis
<code>__host__</code>	host	host	<code>__host__ float HostFunc()</code>
<code>__global__</code>	host	device	<code>__global__ void KernelFunc()</code>
<code>__device__</code>	device	device	<code>__device__ float DeviceFunc()</code>

El **kernel** es entonces una función del tipo `__global__`. El dato devuelto es de tipo vacío (`void`), por lo que el retorno de datos debe realizarse siempre por referencia a través de apuntadores.

```
__global__ void myKernel(arg_1,arg_2,...,arg_n) {
// . . . Código para ejecutar en la GPU
}
```

El código debe escribirse **pensando en paralelo**, es decir, sabiendo que se pueden ejecutar miles de copias simultáneas.

Además, no podemos usar funciones de la biblioteca estándar de C, ya que el código se ejecuta en el GPU, no el CPU.

Dentro de la función `main()` hay que llamar a la función **kernel**, lo que añade una modificación en la sintaxis necesaria para ser invocada. Esta modificación se debe a que vamos a lanzar múltiples copias del kernel y hay que especificar cuántas. La sintaxis para lanzar un kernel es:

```
myKernel<<<blocks,threads>>>(arg_1,arg_2,...,arg_n);
/**
 blocks: número de bloques
 threads: número de copias en cada bloque
 **/
```

A la hora de pensar en paralelo se debe decidir sobre qué conjunto de datos actúa cada hilo o copia. CUDA proporciona a cada hilo su propio **identificador** y lo almacena en la variable `threadIdx`. Con este identificador podremos decidir sobre qué datos actúa cada hilo e incluso decidir si algún hilo o conjunto de hilos ejecuta instrucciones diferentes (*bifurcación de hilos*).

Ejemplo: Suma de 2 vectores

Versión Secuencial

```
__host__ int suma(int *vector_1, int *vector_2, int *vector_suma, int n) {
    for(int i = 0; i < n; i++){
        vector_suma[i] = vector_1[i] + vector_2[i];
    }

    return 0;
}
```

Versión Paralela

```
__global__ void suma(int *vector_1, int *vector_2, int *vector_suma) {
    int myID = threadIdx.x;

    vector_suma[myID] = vector_1[myID] + vector_2[myID];
}
```

¿Cómo leer el código en paralelo?

Parte del problema es que no podemos ver lo que está sucediendo ya que en CUDA no se puede usar la función `printf()` para mostrar cosas en pantalla. No obstante, la impresión en pantalla nos podría mostrar cosas inesperadas ya que los hilos no se ejecutan en orden.

▼ Hilos y Bloques

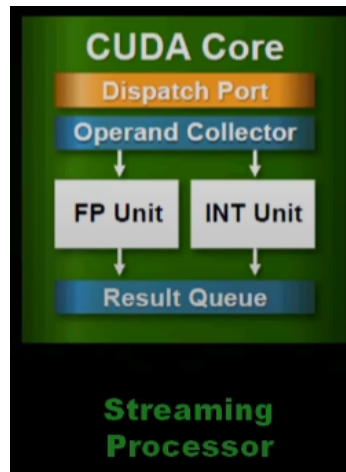
La función **kernel** se va a ejecutar en la GPU en forma de múltiples hilos o copias. Estos hilos se deben agrupar desde el punto de vista "lógico" en **bloques**. El **bloque** es+a la unidad mínima de ejecución de una aplicación **CUDA**.

En la llamada al kernel hay que especificar cuántas copias o hilos queremos. Para ello debemos indicar el número de bloques y el número de hilos por bloque.

```
myKernel<<<num_blocks, num_threads>>>(params)
```

Arquitectura CUDA

Una GPU CUDA está formada por varios **multiprocesadores**. Cada uno de ellos dispone de distintos **espacios de almacenamiento** y cientos de **núcleos de cómputo**. Los núcleos de cómputo (SP) son los encargados de ejecutar las instrucciones.



Los recursos de hardware mínimos disponibles en cada GPU dependen de su capacidad de cómputo. Por otro lado, cada GPU puede disponer de mejores recursos que otras.

Para consultar estos datos se debe acceder a la estructura de propiedades `cudaDeviceProp` generada tras una llamada a la función `cudaGetDeviceProperties()`.

DEVICE PROPERTY	DESCRIPTION
<code>char name[256];</code>	An ASCII string identifying the device (e.g., "GeForce GTX 280")
<code>size_t totalGlobalMem</code>	The amount of global memory on the device in bytes
<code>size_t sharedMemPerBlock</code>	The maximum amount of shared memory a single block may use in bytes
<code>int regsPerBlock</code>	The number of 32-bit registers available per block
<code>int warpSize</code>	The number of threads in a warp
<code>size_t memPitch</code>	The maximum pitch allowed for memory copies in bytes
<code>int maxThreadsPerBlock</code>	The maximum number of threads that a block may contain
<code>int maxThreadsDim[3]</code>	The maximum number of threads allowed along each dimension of a block
<code>int maxGridSize[3]</code>	The number of blocks allowed along each dimension of a grid
<code>size_t totalConstMem</code>	The amount of available constant memory
<code>int major</code>	The major revision of the device's compute capability
<code>int minor</code>	The minor revision of the device's compute capability

Uno de los parámetros más relevantes que debemos conocer es el **máximo número de hilos por bloque** que podemos lanzar.

Terminología Básica en CUDA

- **Hilo (thread):** Cada una de las copias del kernel que se ejecuta en el device.
- **Bloque (block):** Conjunto de hilos.
- **Malla (grid):** Conjunto de bloques.

Cada término tiene su equivalencia en la arquitectura CUDA.

- CUDA thread → CUDA core
- CUDA thread block → CUDA Streaming Multiprocessor
- CUDA kernel grid → CUDA-enabled GPU

Dentro de la función kernel, CUDA proporciona palabras clave que sirven para conocer las dimensiones con las que ha sido lanzado. Esto nos sirve para programar funciones más genéricas que puedan ser válidas sin conocer previamente el número de hilos y/o bloques lanzados. Las palabras clave son:

`blockDim` → Número de hilos lanzados en cada bloque

`gridDim` → Número de bloques lanzados

¿Cómo identificamos los hilos?

La clave en cualquier programa está en identificar a los hilos. Con ese identificador podemos decidir sobre qué dato actúa cada hilo o decidir si algún hilo o conjunto de hilos ejecutan instrucciones distintas.

El identificador `threadIdx` permite identificar a cada hilo pero solo **dentro de un mismo bloque**. Entonces, ¿Cómo identificamos hilos en un kernel multi-bloque?

En CUDA también existe el identificador de bloque: `blockIdx`

Solo con los identificadores `threadIdx` y `blockIdx` no podemos identificar de forma unívoca todos los hilos. Es necesario combinar ambos identificadores:

$$index = threadIdx + blockDim * blockIdx$$

▼ Kernel Bidimensional

Topologías Virtuales

Una característica de los lenguajes de programación paralelos suele ser la posibilidad de crear **topologías virtuales**, es decir, podemos ordenar los hilos de ejecución de acuerdo a cierta geometría espacial.

Esta ordenación suele emplearse si queremos que:

- Los hilos se acomoden al hardware subyacente, optimizando así los recursos disponibles.
- La distribución de los hilos se asemeje a la geometría del problema que queremos resolver, facilitando la programación.

En los problemas donde se trabaja con matrices o con imágenes puede ser conveniente ordenar los hilos siguiendo una topología cartesiana.

A la hora de identificar los hilos y los bloques de un kernel multidimensional usaremos las mismas variables de identificación ya conocidas, especificando el eje **x**, **y** o **z** según corresponda.

Para identificar a cada hilo de manera única debemos combinar varios identificadores:

```
idx = threadIdx.x + blockDim.x * blockIdx.x
idy = threadIdx.y + blockDim.y * blockIdx.y

index = idx + blockDim.x * gridDim.x * idy
```

Con estos valores, podemos identificar de forma unívoca cada hilo de toda la malla (grid)

Lanzando el Kernel

A la hora de invocar o lanzar el kernel desde el programa principal `main()`, la variable correspondiente al número de bloques y al número de hilos por bloque debe ser una variable tridimensional que indique el valor en cada eje del espacio.

Para indicar en **CUDA C/C++** que una variable es tridimensional y que tiene tres valores, se debe declarar usando la etiqueta `dim3`.

Por ejemplo, si queremos lanzar un kernel tridimensional con bloques e hilos en los 3 ejes del espacio, la declaración e inicialización de las variables `blocks` y `threads` sería así:

```
dim3 blocks(Bx,By,Bz);
dim3 threads(hx,hy,hz);
```