Modelo de programación CUDA

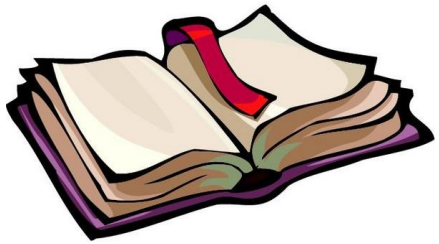
Universidad Nacional de La Plata

Facultad de Informática

Prof. Dr. Adrian Pousa

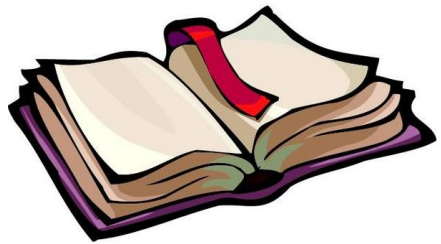
Agenda

I. Introducción al modelo de programación Nvidia CUDA
II. Estructura de programa CUDA
    i. Declaración de variables
    ii. Gestión de la memoria
    iii. Gestión de hilos
    iv. Kernel
    v. Modularidad
    vi. Variables Built-in y Thread ID
    vii. Planificación
    viii. Manejo de errores
    ix. Limitaciones
III. Tensor cores
IV. Métricas y depuración



Agenda

I. Introducción al modelo de programación Nvidia CUDA
II. Estructura de programa CUDA
    i. Declaración de variables
    ii. Gestión de la memoria
    iii. Gestión de hilos
    iv. Kernel
    v. Modularidad
    vi. Variables Built-in y Thread ID
    vii. Planificación
    viii. Manejo de errores
    ix. Limitaciones
III. Tensor cores
IV. Métricas y depuración



CUDA

- En 2006 Nvidia comercializa la serie 8 (**G80**) y provee **CUDA** para facilitar su programación.
- **Compute Unified Device Architecture (CUDA):** plataforma para cómputo paralelo que incluye un compilador y un conjunto de herramientas de desarrollo que permiten a los programadores usar una extensión del lenguaje de programación C para implementar algoritmos sobre GPUs de NVidia.
  - **Extensión al lenguaje C** con constructores y palabras claves.
  - Considera a la GPU como una arquitectura paralela para la resolución de problemas de propósito general (**GPGPU**).
  - Ve la GPU como un conjunto de **multiprocesadores**. Cada multiprocesador posee **procesadores simples**.
  - Sigue un Modelo Flynn **SIMD**.
- El código **CUDA** se almacena con extensión **.cu** que **compilamos en Linux**:

```
nvcc -o ejecutable fuente.cu
```

CUDA

- El programador debe tener en cuenta dos aspectos importantes:

La jerarquía de memoria

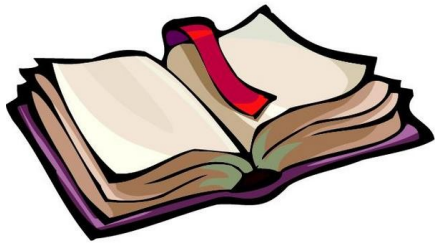
- **Memoria Global:** Lectura y Escritura por CPU y GPU
- **Memoria de Constantes:** Escritura por CPU y solo Lectura en GPU
- **Memoria de Texturas:** Escritura por CPU y solo Lectura en GPU
- **Memoria Compartida (shared):** Lectura y Escritura solo en GPU
- **Memoria Local:** Lectura y Escritura solo en GPU
- **Registros:** Lectura y Escritura solo en GPU

La organización de los hilos

- Grids
- Bloques
- Threads

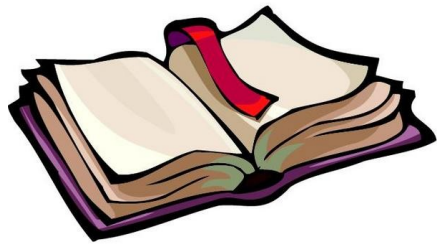
Agenda

I. Introducción al modelo de programación Nvidia CUDA
II. Estructura de programa CUDA
    i. Declaración de variables
    ii. Gestión de la memoria
    iii. Gestión de hilos
    iv. Kernel
    v. Modularidad
    vi. Variables Built-in y Thread ID
    vii. Planificación
    viii. Manejo de errores
    ix. Limitaciones
III. Tensor cores
IV. Métricas y depuración



Agenda

I. Introducción al modelo de programación Nvidia CUDA
II. Estructura de programa CUDA
    i. Declaración de variables
    ii. Gestión de la memoria
    iii. Gestión de hilos
    iv. Kernel
    v. Modularidad
    vi. Variables Built-in y Thread ID
    vii. Planificación
    viii. Manejo de errores
    ix. Limitaciones
III. Tensor cores
IV. Métricas y depuración



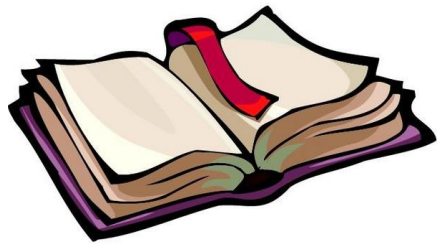
Estructura de programa CUDA

- Un programa CUDA sigue la siguiente estructura:

```
#include <cuda.h>
 //Declaración de variables de CPU y GPU
 //Definición de la función kernel que ejecutara cada hilo en la GPU
…
 int main(int argc, char** argv){
          //Declaración de variables de CPU y GPU
          //Alocación de memoria (si es necesario) en CPU y GPU
          //Copia de datos de memoria CPU a memoria GPU
          //Definir la organización y cantidad de hilos
          //Invocación a la función Kernel que se ejecutará en GPU
          //Copia de los resultados de memoria de GPU a memoria de CPU
          //Liberar memoria en CPU y GPU
 }
```

Agenda

I. Introducción al modelo de programación Nvidia CUDA
II. Estructura de programa CUDA
    i. Declaración de variables
    ii. Gestión de la memoria
    iii. Gestión de hilos
    iv. Kernel
    v. Modularidad
    vi. Variables Built-in y Thread ID
    vii. Planificación
    viii. Manejo de errores
    ix. Limitaciones
III. Tensor cores
IV. Métricas y depuración



Estructura de programa CUDA
Declaración de variables

```
#include <cuda.h>
 //Declaración de variables de CPU y GPU
 //Definición de la función kernel que ejecutara cada hilo en la GPU
…
 int main(int argc, char** argv){
          //Declaración de variables de CPU y GPU
          //Alocación de memoria (si es necesario) en CPU y GPU
          //Copia de datos de memoria CPU a memoria GPU
          //Definir la organización y cantidad de hilos
          //Invocación a la función Kernel que se ejecutará en GPU
          //Copia de los resultados de memoria de GPU a memoria de CPU
          //Liberar memoria en CPU y GPU
 }
```

Estructura de programa CUDA
Declaración de variables

- CUDA utiliza los tipos de datos del lenguaje C y sólo crea algunos nuevos (Por ejemplo: `Dim3` y tipos vectoriales `int2,int3,float2,float3, etc`)
- Declaración de variables en la memoria de la GPU (por convención se utiliza el prefijo `d_` delante, no obligatorio):
  - En memoria Global se agrega el identificador `__device__`:

```
__device__ int d_N = 10;
```

  - En memoria de Constantes se agrega el identificador `__constant__`:

```
__constant__ int d_N=1000;

__constant__ int d_arregloConstante[4]={2,4,6,8}
```

Estructura de programa CUDA
Declaración de variables – Tipos vectoriales

- Los tipos vectoriales son tuplas de hasta 4 dimensiones de tipos simples:
  - int2, int3, int4
  - float2, float3, float4
  - char2, char3, char4
  - etc.
- Por ejemplo, un tipo vectorial flotante de 4 dimensiones:

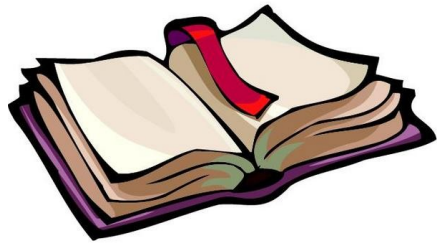
```
float4 miVariable = make_float4( 1.0 , 2.0, 3.0, 4.0 );
```

- Se acceden:

```
miVariable.x
miVariable.y
miVariable.z
miVariable.w
```

Agenda

I. Introducción al modelo de programación Nvidia CUDA
II. Estructura de programa CUDA
    i. Declaración de variables
    ii. Gestión de la memoria
    iii. Gestión de hilos
    iv. Kernel
    v. Modularidad
    vi. Variables Built-in y Thread ID
    vii. Planificación
    viii. Manejo de errores
    ix. Limitaciones
III. Tensor cores
IV. Métricas y depuración



Estructura de programa CUDA
Gestión de la memoria

```
#include <cuda.h>
 //Declaración de variables de CPU y GPU
 //Definición de la función kernel que ejecutara cada hilo en la GPU
…
 int main(int argc, char** argv){
          //Declaración de variables de CPU y GPU
          //Alocación de memoria (si es necesario) en CPU y GPU
          //Copia de datos de memoria CPU a memoria GPU
          //Definir la organización y cantidad de hilos
          //Invocación a la función Kernel que se ejecutará en GPU
          //Copia de los resultados de memoria de GPU a memoria de CPU
          //Liberar memoria en CPU y GPU
 }
```

Estructura de programa CUDA
Gestión de la memoria – Copia explícita

- Copia explícita de memoria:

  - Los datos a ser utilizados en la GPU deben enviarse **explícitamente** desde la memoria RAM de la **CPU (Host)** a la memoria de la **GPU (Device)**. Esto se conoce como **transferencias H2D**.
  - Los resultados obtenidos en la GPU deben recuperarse **explícitamente** desde la memoria de la **GPU (Device)** a la memoria RAM de la **CPU (Host)**. Esto se conoce como **transferencias D2H**.

Estructura de programa CUDA
Gestión de la memoria - Alocación

- Previo a transferir los datos, el espacio de direcciones a usar en memoria global de la GPU debe alocarse y al terminar su uso debe liberarse.

- La función **cudaMalloc** aloca espacio en memoria global de la GPU:

```
cudaMalloc(void ** devPtr,  size_t nbytes)
```

  - `(void**)` puntero a una dirección de memoria.

- La función **cudaFree** libera espacio en memoria global de la GPU:

```
cudaFree(void * devPtr)
```

Estructura de programa CUDA
Gestión de la memoria - Copia

- Para hacer la copia explícita entre CPU y GPU, se utilizan las funciones:
  - **cudaMemcpy**: Para transferencias hacia y desde **memoria global**

```
cudaMemcpy(void* dst, void* src, size_t nbytes, enum cudaMemcpyKind direction)
```

    - `direction` puede ser:
      - `cudaMemcpyHostToDevice` (Para transferencias **H2D**)
      - `cudaMemcpyDeviceToHost` (Para transferencias **D2H**)
      - `cudaMemcpyDeviceToDevice` (Para transferencias **D2D** entre múltiples GPUs)

  - **cudaMemcpyToSymbol:** Para transferencia de CPU a **memoria de constantes**

```
cudaMemcpyToSymbol (const char *symbol, const void * src, size_t nbytes,
    size_t offset = 0, enum cudaMemcpyKind kind = cudaMemcpyHostToDevice)
```

    - `symbol` es el identificador del dato declarado como __constant__

Estructura de programa CUDA
Gestión de la memoria – Ejemplo memoria global

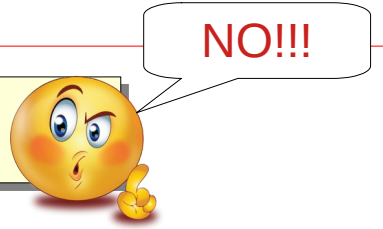
```
int main(int argc, char** argv){
    int n=atoi(argv[1]); //Recibe la longitud del vector como primer parámetro de programa
    int *h_array, *d_array, *h_array_result, *d_array_result;
     …
     h_array = (int*)malloc(n*sizeof(int));
     "Inicializar h_array con datos de entrada"
     h_array_result = (int*)malloc(n*sizeof(int));
     cudaMalloc(&d_array, n*sizeof(int));
     cudaMalloc(&d_array_result, n*sizeof(int));
     cudaMemcpy(d_array, h_array, n*sizeof(int), cudaMemcpyHostToDevice);
     "Ejecución en GPU"
     cudaMemcpy(h_array_result, d_array_result, n*sizeof(int), cudaMemcpyDeviceToHost);
     …
     free(h_array);
     free(h_array_result);
     cudaFree(d_array);
     cudaFree(d_array_result);
}
```

Estructura de programa CUDA
Gestión de la memoria – Ejemplo memoria constantes

```
   unsigned long h_N=16;
   __constant__ unsigned long d_N;

   unsigned long h_Nv[3]={1,2,3};
   __constant__ unsigned long d_Nv[3];
   ...
int main(int argc, char** argv){
      …
      cudaMemcpyToSymbol(d_N,&h_N,sizeof(unsigned long));
      cudaMemcpyToSymbol(d_Nv,&h_Nv,3*sizeof(unsigned long));
      …
}
```

Importante: en el caso de vectores de constantes, las dimensiones deben ser conocidas en tiempo de compilación

```
unsigned long* h_Nv;
__constant__ unsigned long* d_Nv;
```



Estructura de programa CUDA
Gestión de la memoria - memset

- Si necesitamos en GPU un vector con valores idénticos (por ejemplo: inicializado en 0) podemos evitar la transferencia alocando y haciendo que la GPU lo inicialice.
- Para esto utilizamos la función:

```
cudaMemset(void* devPtr, int value, size_t count)
```

  - **`devPtr`:** puntero al vector en device que se va a inicializar.
  - **`value`:** valor que se desea escribir en el vector.
  - **`count`:** cantidad de bytes a inicializar.

- Ejemplo que inicializa un vector de floats con valores en 0.0:

```
cudaMemset(miVector, 0.0, N*sizeof(float));
```

Estructura de programa CUDA
Gestión de la memoria – Memoria de texturas

- Uso de memoria de texturas: memoria de sólo lectura optimizada para accesos multidimensionales (*1D*, *2D*, *3D*).

  Una textura se define mediante el prototipo:

```
texture (tipo, dim, <readmode>) varTextura;
```

  - **`tipo`:** tipo de datos (`int`, `float`, `char` etc.)
  - **`dim`:** dimensiones de la textura (1, 2 o 3) por defecto 1
  - **`readmode`:** (opcional) Control de conversión.
    - **`cudaReadModeElementType`** (por defecto): sin conversión
    - **`cudaReadModeNormalizedFloat`**: con conversión. Los valores se normalizan a [-1.0,1.0] con signo y [0.0, 1.0] sin signo

- Ejemplo *1D*:
```
texture(int) miTextura1D;
```
- Ejemplo *2D*:
```
texture(int,2) miTextura2D;
```

Estructura de programa CUDA
Gestión de la memoria – Memoria de texturas

- Antes de utilizar la memoria de texturas es necesario hacer un **Binding**, es decir asociar un buffer del host a la memoria de texturas:

```
cudaBindTexture(size *t offset, const struct texture<T, dim, readMode> &tex , const void * devptr, size_t
                                                    size=UINT_MAX) ;
```

  - **`offset`:** offset en bytes
  - **`tex`:** textura a asociar.
  - **`devptr`:** área de memoria en el device.
  - **`size`:** tamaño en bytes del área de memoria apuntado por devptr.

- Luego de usar la memoria de texturas se debe hacer un **Unbind**:

```
cudaUnbindTexture (const struct texture<T, dim, readMode> &tex) ;
```

Estructura de programa CUDA
Gestión de la memoria – Memoria de texturas

- La lectura de la textura dentro del código de la GPU (**kernel**) se hace:

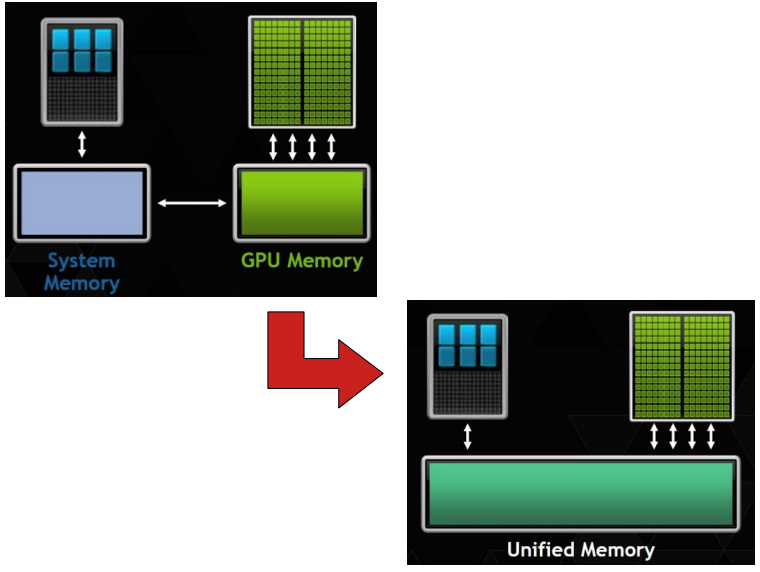
```
tex1Dfetch(textura, posX)
tex2Dfetch(textura, posX, posY)
tex3Dfetch(textura, posX, posY, posZ)
```

Estructura de programa CUDA
Gestión de la memoria – Ejemplo memoria texturas

```
texture<int> miTextura;

__global__ void miKernel {
...
        int a = tex1Dfetch(miTextura,2));
...
}

int main(int argc, char* argv[]){
int h_datos[3]={1,2,3};
int *d_datos;

        cudaMalloc( (void**)&d_datos, N*sizeof(int));
        cudaMemcpy(d_datos,h_datos,N*sizeof(int),cudaMemcpyHostToDevice);
        cudaBindTexture(NULL,miTextura,d_datos,N*sizeof(int));
        ...
        miKernel<<<dimGrid, dimBlock>>>(); "Ejecución en GPU"
        cudaDeviceSynchronize();
        cudaUnbindTexture(miTextura);
        cudaFree(d_datos);
}
```

Estructura de programa CUDA
Gestión de la memoria – Memoria unificada

- **Memoria unificada:** idea de gestionar la memoria de forma implícita (transparente al programador)
- **Nuevo símbolo:** **`__managed__`** combinado con **`__device__`** indica que una variable se puede acceder tanto de CPU como GPU.



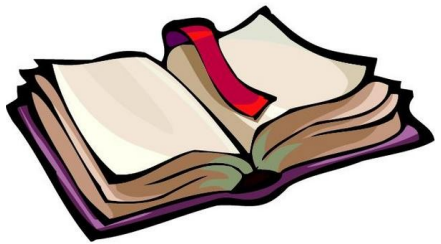

```
__device__ __managed__ float *x ; // Variable accesible desde CPU y GPU


int main(int argc, char** argv){
      // Aloca Memoria unificada accesible desde CPU o GPU
      cudaMallocManaged(&x, N*sizeof(float));
      "Ejecución en GPU"
      //Espera que termine la ejecución para acceder a los resultados
     CudaDeviceSynchronize();
      //Liberar memoria en CPU y GPU
      cudaFree(x);
}
```

Estructura de programa CUDA
Gestión de la memoria – Memoria unificada

- Limitaciones de la memoria unificada:
  - La cantidad de memoria unificada disponible está determinada por el tamaño de la memoria de la GPU.
  - Las páginas de memoria de asignaciones unificadas modificadas por la CPU deben migrarse nuevamente a la GPU antes de iniciar cualquier kernel.
  - La CPU y la GPU no pueden acceder simultáneamente a la memoria unificada (uso de cudaDeviceSynchronize para sincronizar).
  - Mientras se ejecuta en la GPU, esta tiene acceso exclusivo a la memoria unificada.

Agenda

I. Introducción al modelo de programación Nvidia CUDA
II. Estructura de programa CUDA
    i. Declaración de variables
    ii. Gestión de la memoria
    iii. Gestión de hilos
    iv. Kernel
    v. Modularidad
    vi. Variables Built-in y Thread ID
    vii. Planificación
    viii. Manejo de errores
    ix. Limitaciones
III. Tensor cores
IV. Métricas y depuración



Estructura de programa CUDA
Gestión de hilos

```
#include <cuda.h>
 //Declaración de variables de CPU y GPU
 //Definición de la función kernel que ejecutara cada hilo en la GPU
…
 int main(int argc, char** argv){
          //Declaración de variables de CPU y GPU
          //Alocación de memoria (si es necesario) en CPU y GPU
          //Copia de datos de memoria CPU a memoria GPU
          //Definir la organización y cantidad de hilos
          //Invocación a la función Kernel que se ejecutará en GPU
          //Copia de los resultados de memoria de GPU a memoria de CPU
          //Liberar memoria en CPU y GPU
 }
```

Estructura de programa CUDA
Gestión de hilos

- Los hilos se organizan en tres niveles cuyo tamaño define el programador:
  - **Grids:** Conjunto de bloques.
  - **Bloques:** Conjunto de threads.
  - **Hilos (Threads):** hilos propiamente dicho.
- Antes de ejecutar sobre la GPU deben definirse:
  - La **Dimensión de grid** (1 dimensión o 2 dimensiones de bloques) que determina implícitamente la cantidad de bloques.
  - La **Dimensión de bloque** (1 dimensión, 2 dimensiones o 3 dimensiones de hilos) que determina implícitamente la cantidad de hilos del bloque.
- Estas dimensiones se definen en variables tipo `dim3`

Estructura de programa CUDA
Gestión de hilos – Definición de grid y bloques

Grid

Unidimensional de 3 bloques de hilos:

```
dim3 miGrid1D(3,1,1)
```

Bidimensional de 3x2 bloques de hilos:

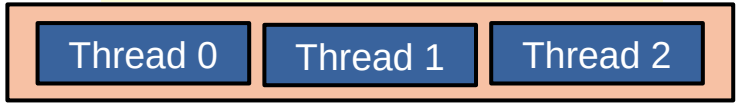
```
dim3 miGrid2D(3,2,1)
```

*El último parámetro no se utiliza.

Bloques

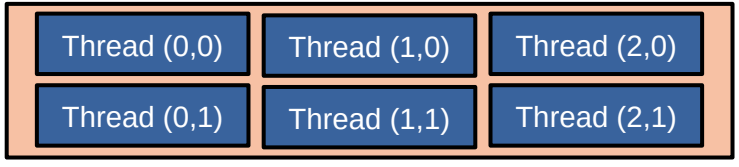
Unidimensional de 3 de hilos:

```
dim3 miBloque1D(3,1,1)
```



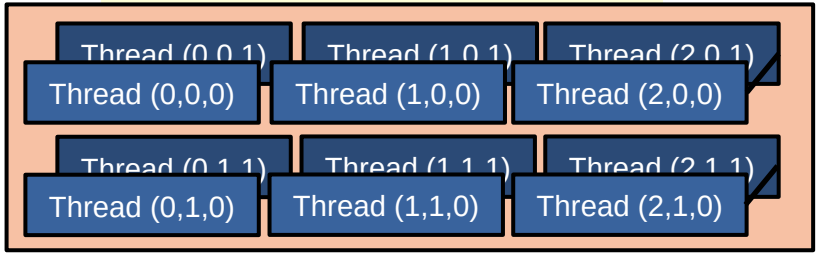
Bidimensional de 3x2 hilos:

```
dim3 miBloque2D(3,2,1)
```



Tridimensional de 3x2x2 hilos:

```
dim3 miBloque3D(3,2,2)
```



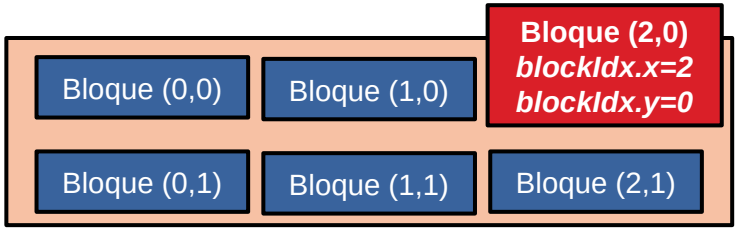
Estructura de programa CUDA
Gestión de hilos – Identificación – Variables built-in

- Durante la ejecución los hilos pueden conocer a que bloque pertenecen y su identificación dentro del bloque a partir de **variables built-in.**

Identificación del hilo en el Grid

Variables built-in:

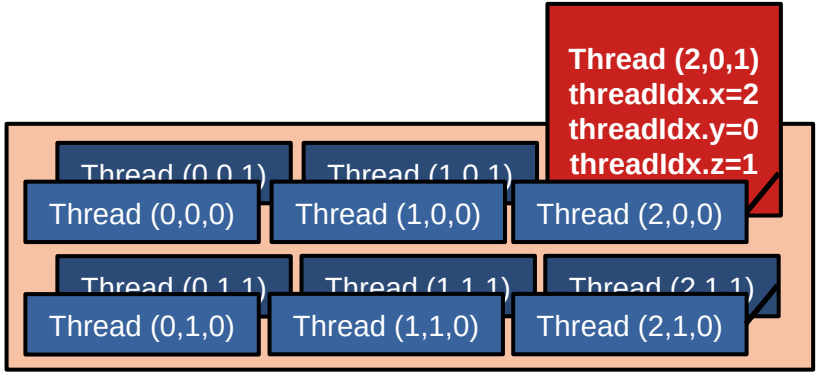
- **blockIdx.x:** coordenada x del bloque de hilos dentro del grid.
- **blockIdx.y:** coordenada y del bloque de hilos dentro del grid.



Identificación del hilo en el Bloque

Variables built-in:

- **threadIdx.x:** coordenada x del hilo dentro del bloque.
- **threadIdx.y:** coordenada y del hilo dentro del bloque.
- **threadIdx.z:** coordenada z del hilo dentro del bloque.



Estructura de programa CUDA
Gestión de hilos – A tener en cuenta

- Todos los hilos ejecutan la misma función (el mismo código).
- Todos los hilos de un mismo bloque se ejecutan en un mismo SM.
- Todos los hilos de un mismo bloque comparten la memoria shared y por medio de esta pueden comunicarse.
- Los hilos de un mismo bloque pueden sincronizar, por ejemplo, por barreras.
- Los hilos de diferentes bloques no se relacionan entre si.
- Existen limitaciones en la cantidad de hilos que puede contener un bloque. Estas limitaciones dependen de la arquitectura.

Estructura de programa CUDA
Gestión de hilos

- Cualquier variable de tipo `dim3` puede modificarse en tiempo de ejecución sobre el **código del host**:

```
…
dim3 miVariableDim3(3,1,2);
…
miVariableDim3.x = 1;
miVariableDim3.y = 1;
miVariableDim3.z = 1;
…
```

Estructura de programa CUDA
Gestión de hilos

Pero... ¿Cuantos bloques o hilos creamos?

- Generalmente, en cada arquitectura hay un tamaño de hilos por bloque adecuado (**threadsPerBlock**) que puede ser: 128, 256, 512 etc.

- Aunque depende de las características del problema, podemos resolver la organización de hilos respondiendo (y resolviendo) la siguiente pregunta:

Si tengo un problema de tamaño N y una cantidad de hilos por bloque threadsPerBlock ¿Cuantos bloques necesito para mi grid?

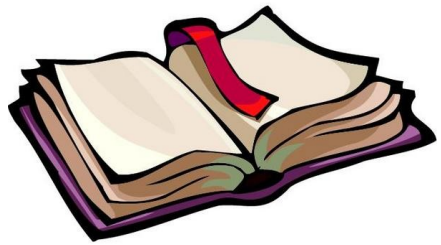
- Luego, ejecutamos nuestro programa parametrizado de la siguiente forma:

```
./miPrograma N threadsPerBlock
```

- En ejecución calculamos la cantidad de bloques que necesitamos.

Agenda

I. Introducción al modelo de programación Nvidia CUDA
II. Estructura de programa CUDA
    i. Declaración de variables
    ii. Gestión de la memoria
    iii. Gestión de hilos
    iv. Kernel
    v. Modularidad
    vi. Variables Built-in y Thread ID
    vii. Planificación
    viii. Manejo de errores
    ix. Limitaciones
III. Tensor cores
IV. Métricas y depuración



Estructura de programa CUDA
Kernel


```
#include <cuda.h>
 //Declaración de variables de CPU y GPU
 //Definición de la función kernel que ejecutara cada hilo en la GPU
…
 int main(int argc, char** argv){
          //Declaración de variables de CPU y GPU
          //Alocación de memoria (si es necesario) en CPU y GPU
          //Copia de datos de memoria CPU a memoria GPU
          //Definir la organización y cantidad de hilos
          //Invocación a la función Kernel que se ejecutará en GPU
          //Copia de los resultados de memoria de GPU a memoria de CPU
          //Liberar memoria en CPU y GPU
 }
```

Estructura de programa CUDA
Kernel – Definición e invocación

- Se llama **kernel** a la función que ejecutarán todos los hilos del grid.
- Se la define de la siguiente forma:

```
__global__ void miFuncion( parámetros ) {
            …
}
```

  - El identificador **`__global__`** identifica la función como **kernel.**
  - La función no tiene valor de retorno.

- Se invoca:

```
…
dim3 bloque(N,N); //Bloque bidimensional de N*N hilos
dim3 grid(M,M);  //Grid bidimensional de M*M bloques
miFuncion<<<grid, bloque>>>(parametros);
…
```

El llamado ejecuta la función kernel en la GPU con $NxNxMxM$ hilos.

Estructura de programa CUDA
Kernel – Llamado asíncrono

- La invocación a un kernel es **asíncrona**.
- De esta forma, mientras la función del kernel se ejecuta, se puede continuar la ejecución en la CPU o en otras GPUs .
- Para que el proceso que llama al kernel se demore luego de la invocación es necesario hacer lo siguiente:

```
…
dim3 bloque(N,N); //Bloque bidimensional de N*N hilos
dim3 grid(M,M);  //Grid bidimensional de M*M bloques
miFuncion<<<grid, bloque>>>(parámetros);
cudaDeviceSynchronize();
…
```

Estructura de programa CUDA
Kernel – Tiempo de vida de las variables

- En ocasiones, necesitamos invocar varias veces al mismo o diferentes kernels y trabajar sobre los mismos datos.
- Los datos en memoria global, de constantes y de texturas permanecen en la GPU hasta que la aplicación termine.
- No es necesario hacer copias H2D o D2H cada vez que se invoque a un kernel, a menos que sea necesario.

```
…
cudaMemcpy(d_array, h_array, n*sizeof(int), cudaMemcpyHostToDevice); //Copia H2D
kernel1<<<grid, bloque>>>(parámetros); //Trabaja sobre la GPU
cudaDeviceSynchronize();
kernel2<<<grid, bloque>>>(parámetros); //kernel2 trabaja sobre los resultados que kernel1 deja GPU
cudaDeviceSynchronize();
cudaMemcpy(h_array_result, d_array_result, n*sizeof(int), cudaMemcpyDeviceToHost); //Copia D2H
…
```

Estructura de programa CUDA
Kernel – Punteros en memoria global

- Existen limitaciones respecto al uso de punteros a la memoria global:
  - No es posible alocar punteros dentro del kernel.
  - Los punteros pueden usarse de dos formas:
    - Un puntero alocado desde el host con la función `cudaMalloc()` y pasado como parámetro en la invocación del kernel:

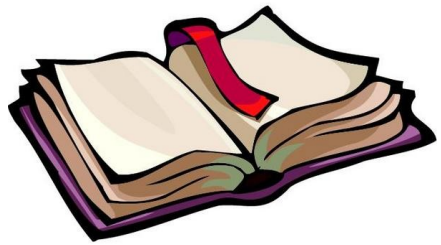
```
__global__ void miFuncion( int* misDatos ) {
            …
}
```

    - Crear un puntero en el kernel y asignarle la dirección de una variable:

```
__global__ void miFuncion( ... ) {
 int miDato;
 int* pmiDato = &miDato;
...
}
```

Agenda

I. Introducción al modelo de programación Nvidia CUDA
II. Estructura de programa CUDA
    i. Declaración de variables
    ii. Gestión de la memoria
    iii. Gestión de hilos
    iv. Kernel
    v. Modularidad
    vi. Variables Built-in y Thread ID
    vii. Planificación
    viii. Manejo de errores
    ix. Limitaciones
III. Tensor cores
IV. Métricas y depuración



Estructura de programa CUDA
Modularidad

- El kernel puede invocar funciones para permitir un código modular.
- Se definen de forma diferente dependiendo desde donde se invocan:

Sólo desde el Kernel

Deberá llevar delante el identificador `__device__`
Podra invocarse en GPU.
No podrá invocarse en CPU

```
__device__ fGPU(){
      …
}
```

Sólo desde CPU

No deberá llevar delante ningún identificador o el identificador `__host__`
Podrá invocar en CPU.
No podrá invocarse en GPU.

```
fCPU(){
      …
}
```

```
__host__ fCPU(){
      …
}
```

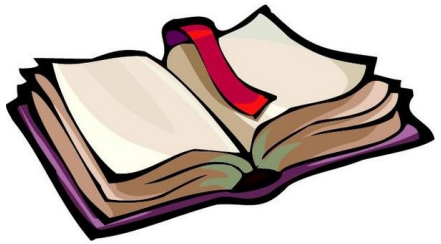
Desde CPU y GPU

Deberá llevar delante ambos identificadores `__host__` y `__device__`
Podrá invocarse en CPU y GPU

```
__host__ __device__ fCPU_GPU(){
      …
}
```

Agenda

I. Introducción al modelo de programación Nvidia CUDA
II. Estructura de programa CUDA
    i. Declaración de variables
    ii. Gestión de la memoria
    iii. Gestión de hilos
    iv. Kernel
    v. Modularidad
    vi. Variables Built-in y Thread ID
    vii. Planificación
    viii. Manejo de errores
    ix. Limitaciones
III. Tensor cores
IV. Métricas y depuración



Estructura de programa CUDA
Variables built-in

- Dentro del kernel podemos usar **variables built-in** que se utilizan para obtener distintos niveles de información:

Identificar hilos en el bloque

- threadidx.x
- threadidx.y
- threadidx.z

Identificar bloques en un grid

- blockIdx.x
- blockIdx.y

Obtener dimensiones de un bloque

- blockDim.x
- blockDim.y
- blockDim.z

Obtener dimensiones de un grid

- gridDim.x
- gridDim.y

Estructura de programa CUDA
Variables built-in – Thread id

- CUDA no tiene un identificador único para cada hilo.
- En ocasiones un identificador único permite determinar que posición de memoria lee cada hilo para su posterior procesamiento.
- Es común utilizar las variables built-in para generar éste identificador único por hilo.
- Un identificador único por hilo, en el caso de bloques unidimensionales, se obtiene mediante la fórmula:

```
int idx = blockDim.x*blockIdx.x + threadIdx.x
```

Estructura de programa CUDA
Variables built-in – Thread id

- Ejemplo: 1 grid de 2 bloques unidimensionales de 3 hilos cada uno.

blockDim.x	blockIdx.x	threadIdx.x	Idx=blockDimx*blockIdx.x + threadidx.x
3	0	0	0
3	0	1	1
3	0	2	2
3	1	0	3
3	1	1	4
3	1	2	5

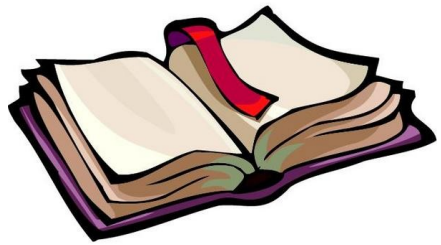
Estructura de programa CUDA
Variables built-in – Thread id

- Las fórmulas para los distintos tamaños de Grid y Bloques son:

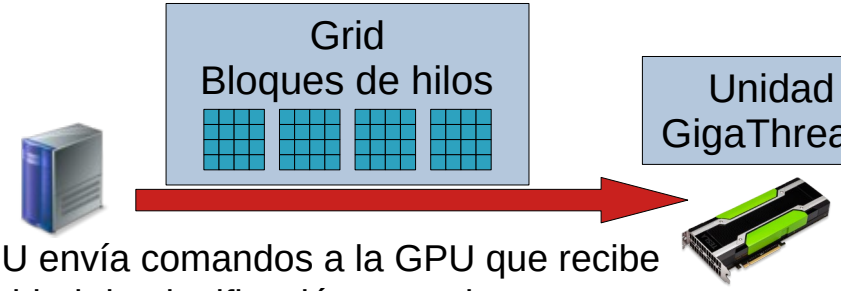
Grid	Bloque	threadId
1D	*1D*	*threadId = blockIdx.x *blockDim.x + threadIdx.x;*
	2D	*threadId = blockIdx.x * blockDim.x * blockDim.y + threadIdx.y * blockDim.x + threadIdx.x*
	3D	*threadId = blockIdx.x * blockDim.x * blockDim.y * blockDim.z + threadIdx.z * blockDim.y * blockDim.x + threadIdx.y * blockDim.x + threadIdx.x*
2D	*1D*	*blockId = blockIdx.y * gridDim.x + blockIdx.x;* *threadId = blockId * blockDim.x + threadIdx.x;*
	2D	*blockId = blockIdx.x + blockIdx.y * gridDim.x;* *threadId = blockId * (blockDim.x * blockDim.y) + (threadIdx.y * blockDim.x) + threadIdx.x;*
	3D	*blockId = blockIdx.x + blockIdx.y * gridDim.x;* *threadId = blockId * (blockDim.x * blockDim.y * blockDim.z) + (threadIdx.z * (blockDim.x * blockDim.y)) + (threadIdx.y * blockDim.x) + threadIdx.x;*

Agenda

I. Introducción al modelo de programación Nvidia CUDA
II. Estructura de programa CUDA
    i. Declaración de variables
    ii. Gestión de la memoria
    iii. Gestión de hilos
    iv. Kernel
    v. Modularidad
    vi. Variables Built-in y Thread ID
    vii. Planificación
    viii. Manejo de errores
    ix. Limitaciones
III. Tensor cores
IV. Métricas y depuración

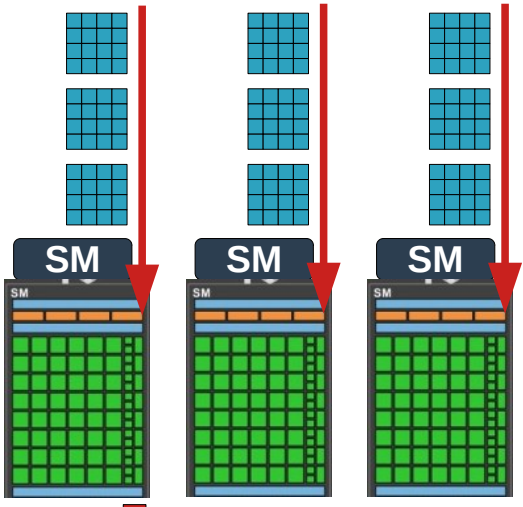


Estructura de programa CUDA
Planificación

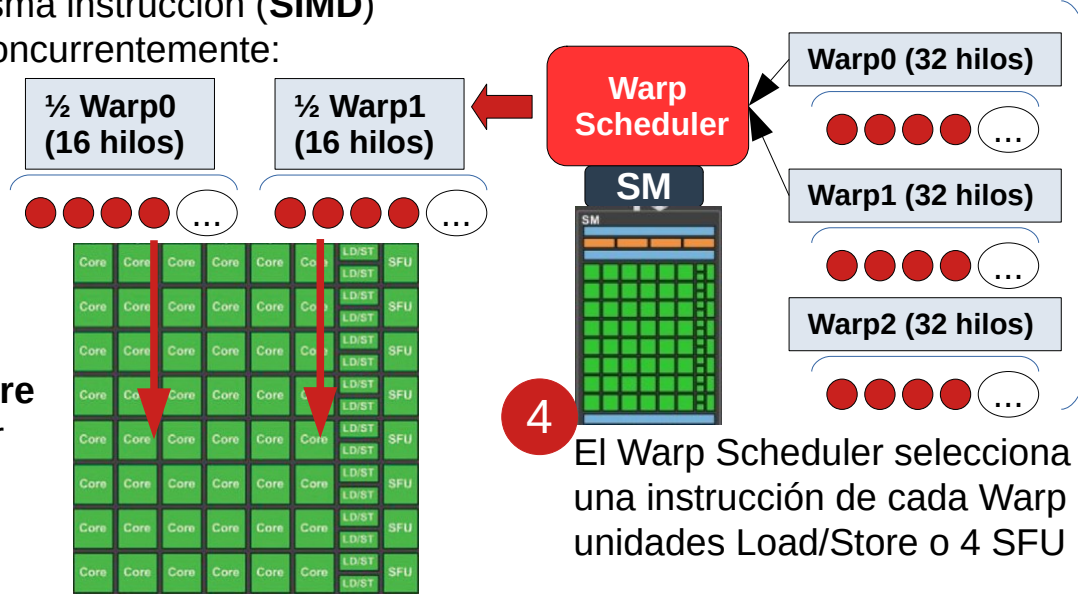


1 La CPU envía comandos a la GPU que recibe una unidad de planificación central

2 La unidad de planificación central envía bloques de hilos a varios SMs



3 Cada SM divide los bloques de hilos en Warps (32 hilos)



4 El Warp Scheduler selecciona 2 Warps y da una instrucción de cada Warp a 16SPs, 16 unidades Load/Store o 4 SFU

5 Cada Warp ejecuta la misma instrucción (**SIMD**)
Instrucciones lanzadas concurrentemente:

- 2 operaciones enteras
- Load/Store
- SFU

NO doble precisión

6 **Multithreading por hardware**
Cambios de contextos por hardware nativo.

Estructura de programa CUDA
Planificación

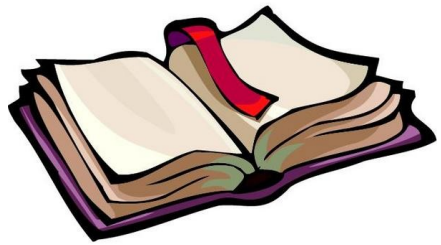
- Las GPUs **no** soportan **afinidad**, es decir el programador sólo define la organización de los hilos pero no puede decidir sobre la planificación (por ejemplo: que bloques van a que SM)

- El estado de los warps se lleva en una tabla en hardware llamada **scoreboarding** (una por warp scheduling) que permite decidir que warp será el próximo en ejecutar.

Warp	Instrucción actual	Estado
warp1	*42*	Computando
warp2	*95*	Leyendo operandos
warp3	*11*	Listo para escribir resultados
warp4	*7*	Operandos listos

Agenda

I. Introducción al modelo de programación Nvidia CUDA
II. Estructura de programa CUDA
    i. Declaración de variables
    ii. Gestión de la memoria
    iii. Gestión de hilos
    iv. Kernel
    v. Modularidad
    vi. Variables Built-in y Thread ID
    vii. Planificación
    viii. Manejo de errores
    ix. Limitaciones
III. Tensor cores
IV. Métricas y depuración



Estructura de programa CUDA
Manejo de errores

- La mayoría de las funciones CUDA tienen como valor de retorno un código de error:

```
cudaError_t cudaMalloc(void** devPtr, size_t size)
cudaError_t cudaFree(void* devPtr)
cudaError_t cudaMemcpy(void* dst, const void* src, size_t count, enum cudaMemcpyKind kind)
```

- Todas retornan un valor del tipo `cudaError_t`.

- El valor de retorno es **`cudaSuccess`** si la función se ejecutó con **éxito** o un **código de error** en caso contrario.

Estructura de programa CUDA
Manejo de errores

- Un llamado al kernel no tiene valor de retorno.
- Hay dos formas de conocer si el kernel se ejecutó con éxito:
  - Mediante la función `cudaDeviceSynchronize`:

```
cudaError_t cudaDeviceSynchronize(void)
```

```
miKernel<<<GridSize,BlockSize>>>(Parámetros);
cudaError_t error = cudaDeviceSynchronize();
if(error != cudaSuccess) printf("Error %d",error);
```

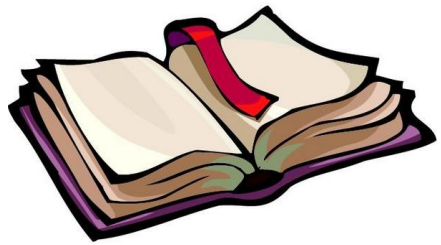
  - Mediante la función función `cudaGetLastError`:

```
cudaError_t cudaGetLastError(void)
```

```
miKernel<<<GridSize,BlockSize>>>(Parámetros);
cudaError_t error = cudaGetLastError();
if(error != cudaSuccess) printf("Error
%d",error);
```

Agenda

I. Introducción al modelo de programación Nvidia CUDA
II. Estructura de programa CUDA
    i. Declaración de variables
    ii. Gestión de la memoria
    iii. Gestión de hilos
    iv. Kernel
    v. Modularidad
    vi. Variables Built-in y Thread ID
    vii. Planificación
    viii. Manejo de errores
    ix. Limitaciones
III. Tensor cores
IV. Métricas y depuración

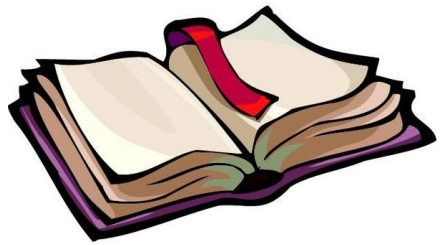


Estructura de programa CUDA
Limitaciones

- No puede hacerse Entrada-Salida a disco desde la GPU.
- Dificultad para utilizar bibliotecas de CPU en GPU: deben modificarse las funciones para indicar que código se ejecuta en CPU (`__host__`) y que en GPU (`__device__`).
- No pueden hacerse llamadas recursivas dentro de la GPU.
- No pueden declararse variables estáticas.
- El kernel no puede recibir un número variable de argumentos.
- No se permite afinidad.
- Una función que se utiliza en el host y en el device pero define variables en un sólo contexto (host o device), debe ser duplicada con una variable para cada contexto.

Agenda

I. Introducción al modelo de programación Nvidia CUDA
II. Estructura de programa CUDA
    i. Declaración de variables
    ii. Gestión de la memoria
    iii. Gestión de hilos
    iv. Kernel
    v. Modularidad
    vi. Variables Built-in y Thread ID
    vii. Planificación
    viii. Manejo de errores
    ix. Limitaciones
III. Tensor cores
IV. Métricas y depuración



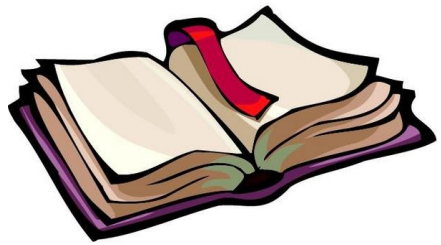
Tensor cores

- Los **tensor cores** son unidades **específicas** para acelerar el cálculo de multiplicación y sumas de matrices.
- Cada tensor core provee de una matriz de procesamiento de *4x4x4* (o más según la arquitectura) que realiza la operación de matrices $D = AB + C$

- Múltiples tensor cores se ejecutan simultáneamente proporcionando gran capacidad de cómputo
- No se ejecutan directamente mediante funciones CUDA sino mediante bibliotecas:
  - Deep Learning Frameworks (DL Frameworks: TensorFlow, PyTorch, MXNet)
  - CuBlas (HGEMM, GEMMEX)
  - WMMA API
  - CuTlas

Agenda

I. Introducción al modelo de programación Nvidia CUDA
II. Estructura de programa CUDA
    i. Declaración de variables
    ii. Gestión de la memoria
    iii. Gestión de hilos
    iv. Kernel
    v. Modularidad
    vi. Variables Built-in y Thread ID
    vii. Planificación
    viii. Manejo de errores
    ix. Limitaciones
III. Tensor cores
IV. Métricas y depuración



Métricas y depuración
Speedup

- Supongamos un programa al cual se le hizo alguna mejora. Si queremos conocer cual fue el beneficio alcanzado podemos calcular el ***Speedup***:

$$Speedup = \frac{t_{antes}}{t_{después}}$$

  - t_{antes}: tiempo de ejecución antes de la mejora.
  - $t_{después}$: tiempo de ejecución después de la mejora.

- El ***Speedup*** es una medida de rendimiento relativa que indica cuanto mejora (o empeora) un algoritmo.
- No podemos implementar un algoritmo secuencial en GPU y no podemos calcular un ***Speedup*** absoluto, pero podemos calcular el ***Speedup*** **relativo** para medir cuanto mejora un algoritmo en la GPU respecto a ejecutarlo en una CPU (secuencialmente o en otra arquitectura paralela):

$$Speedup = \frac{t_{CPU}}{t_{GPU}}$$

Métricas y depuración
Tiempo de ejecución

¿Cómo obtener el tiempo de ejecución?

- Para el cálculo del ***Speedup***, lo correcto es tomar el tiempo de la ejecución del kernel y las transferencias H2D y D2H.

```
double tiempo_inicial;
double tiempo_final;
...
tiempo_inicial = now();
 cudaMemcpy(d_array, h_array, n*sizeof(int), cudaMemcpyHostToDevice);
 miKernel<<<GridSize,BlockSize>>>("Parámetros");
 cudaDeviceSynchronize();
 cudaMemcpy(h_array_result, d_array_result, n*sizeof(int), cudaMemcpyDeviceToHost);
tiempo_final = now() - tiempo_inicial;
```

Métricas y depuración
Speedup

- A tener en cuenta:
  - No está claro cuál es el Speedup ideal y tampoco la idea de eficiencia.
  - Aunque podemos tener una idea de cuántos SMs estamos usando no tenemos una idea clara de cuándo estamos usando mas o menos SMs. Por eso, la ley de Amdahl no puede evaluarse fácilmente. Eventualmente, podemos evaluarla considerando una GPU como una única unidad de procesamiento y luego analizar que ocurre al utilizar mas de una GPU.
  - En paralelismo “clásico” un proceso o hilo se ejecuta sobre una única unidad de procesamiento y, en el mejor caso, se crean tantos hilos o procesos como unidades de procesamiento disponibles, pero no mas. En GPU cada hilo realiza muy poco trabajo (granularidad fina) y se crea un número de hilos muy superior (Multithreading por hardware) al número de unidades de procesamiento (SMs).
  - Las GPUs tienen una capacidad de memoria limitada, al incrementar la carga de trabajo se debe particionar en porciones que quepan en la memoria de la GPU, la ejecución de cada porción tiene el mismo Speedup y el mismo Overhead. Por eso, la ley de Gustafson no puede evaluarse fácilmente. Tiene sentido si la carga de trabajo cabe en la memoria de GPU pero no en caso contrario, el Speedup es proporcional al número de veces que se ejecuta.
- Por estas razones, el Speedup no representa una métrica útil y se utilizan otro tipo de métricas.
- Esto nos permite evaluar la escalabilidad, utilizando como base esas nuevas métricas, al incrementar el número de hilos, la carga de trabajo y/o el número de GPUs.

Métricas y depuración
Ancho de banda teórico

- El **Ancho de banda teórico** se obtiene a partir de las especificaciones de la placa.
  - Se mide en GB/s.
  - Ejemplo: GPU NVIDIA Tesla M2050:
    - RAM DDR (tasa de datos doble) con una velocidad de reloj de 1546 MHz.
    - Ancho de la interfaz de memoria 384 bits.
    - El ancho de banda máximo de la memoria teórica es:

$$BW_{teórico} = 1546\,Mhz \cdot 10^6 \cdot \frac{384\,bits}{8} \cdot \frac{2}{10^9} = 148\,GB/seg$$

Métricas y depuración
Ancho de banda efectivo

- El **Ancho de banda efectivo** se calcula a partir de la aplicación.
  - Se mide en GB/s.
  - Se calcula como:

$$BW_{efectivo} = \frac{(R_B + W_B)}{t \cdot 10^9}$$

R_B: bytes leídos por el Kernel. (cudaMemcpy -> cudaMemcpyHostToDevice)
W_B: bytes escritos por el Kernel. (cudaMemcpy -> cudaMemcpyDeviceToHost)
t: tiempo transcurrido en segundos.

Métricas y depuración
Rendimiento teórico

- El **Rendimiento teórico** se obtiene a partir de las especificaciones de la placa.
  - Se mide en GFLOP/s.
  - Ejemplo: especificaciones GPU NVIDIA Tesla M2050
    - Rendimiento de punto flotante de máxima precisión teórico 1030 GFLOP/s.
    - Rendimiento de doble precisión teórico máximo de 515 GFLOP/s.

Métricas y depuración
Rendimiento efectivo (Throughput)

- El **Rendimiento efectivo (Throughput)** se calcula a partir de la aplicación.
  - Se mide en GFLOP/s.
  - Se calcula como:

$$\left(GFLOPS/S\right)_{efectivo} = \frac{Ops}{t \cdot 10^9}$$

Ops : Operaciones de punto flotante
t : tiempo transcurrido en segundos.

Métricas y depuración
Herramientas de depuración

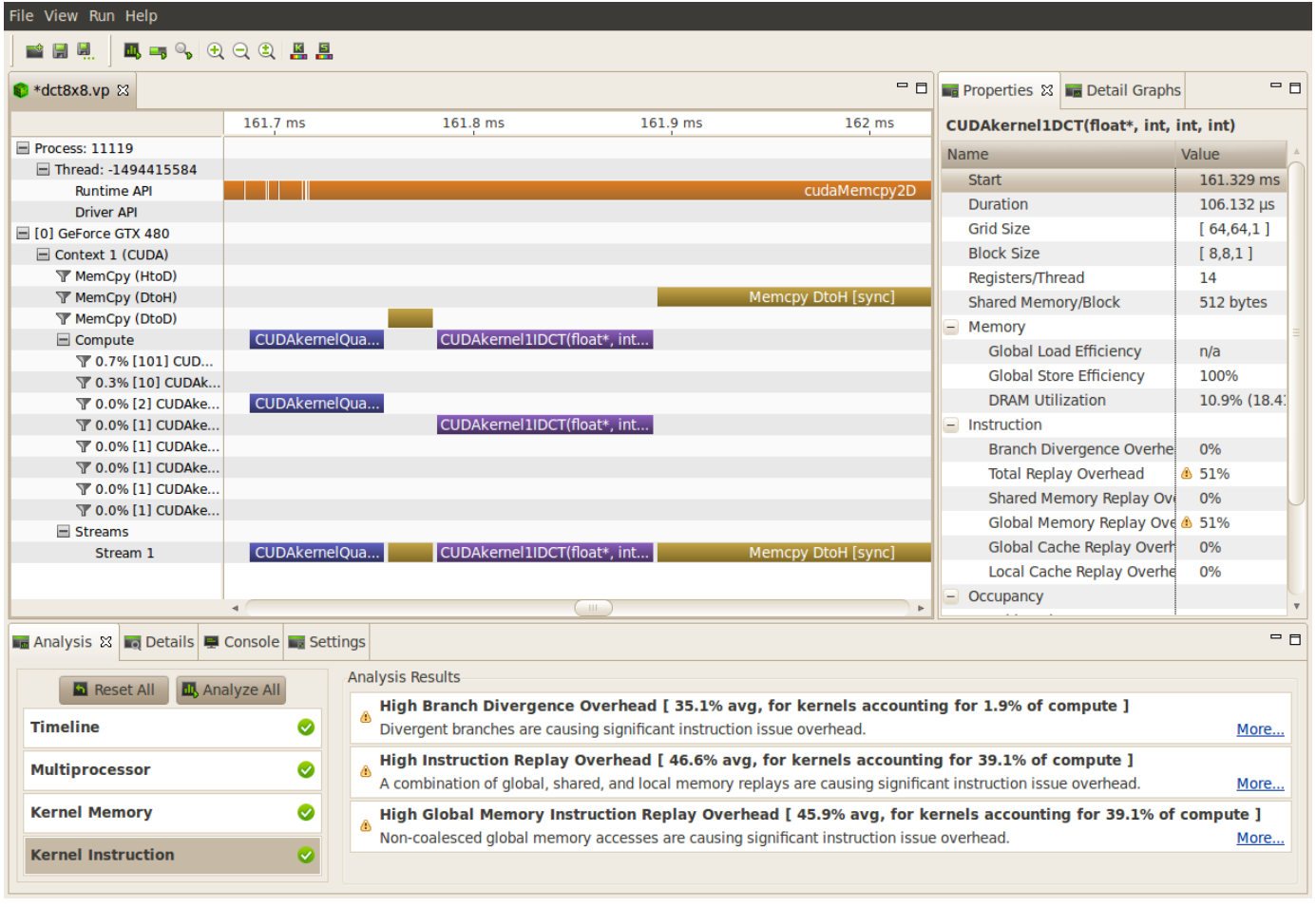
- Podemos obtener información sobre la ejecución de un programa CUDA a partir de herramientas proporcionadas por Nvidia:
  - Comando `nvprof`: es una herramienta de profiling que permite recopilar información y ver datos de perfiles desde la línea de comandos. Recopila actividades de la CPU y la GPU, ejecución del kernel y transferencias de memoria, entre otras.
  - Visualización mediante **Nvidia Visual Profiler** (**nvvp**)
  - Comando `cuda-gdb`: ambiente de depuración similar a gdb.

Métricas y depuración
Herramientas de depuración - nvprof

```
user# nvprof ./miProgCUDA 1000 7
==14234== NVPROF is profiling process 14234, command: ./miProgCUDA 1000 7
==14234== Profiling application: ./miProgCUDA 1000 7
==14234== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 40.98%  2.6880us         1  2.6880us  2.6880us  2.6880us  moduloP(int*, int*, int, int)
 36.10%  2.3680us         1  2.3680us  2.3680us  2.3680us  [CUDA memcpy DtoH]
 22.93%  1.5040us         1  1.5040us  1.5040us  1.5040us  [CUDA memcpy HtoD]

==14234== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 99.44%  86.121ms         2  43.060ms  6.0210us  86.115ms  cudaMalloc
  0.24%  205.57us        91  2.2580us     123ns  86.690us  cuDeviceGetAttribute
  0.14%  120.38us         1  120.38us  120.38us  120.38us  cuDeviceTotalMem
  0.08%  70.715us         2  35.357us  7.4060us  63.309us  cudaFree
  0.03%  27.976us         2  13.988us  12.147us  15.829us  cudaMemcpy
  0.03%  24.587us         1  24.587us  24.587us  24.587us  cudaLaunch
  0.03%  23.475us         1  23.475us  23.475us  23.475us  cuDeviceGetName
  0.00%  4.2210us         1  4.2210us  4.2210us  4.2210us  cudaDeviceSynchronize
  0.00%  2.6790us         4     669ns     188ns  1.8770us  cudaSetupArgument
  0.00%  1.4740us         3     491ns     120ns  1.1350us  cuDeviceGetCount
  0.00%     977ns         3     325ns     144ns     507ns  cuDeviceGet
  0.00%     974ns         1     974ns     974ns     974ns  cudaConfigureCall
```

Métricas y depuración
Herramientas de depuración – NVidia Visual Profiler

```
user# nvprof nvvp ./miProgCUDA 1000 7
```



Métricas y depuración
Herramientas de depuración – cuda-gdb

- Se recomienda compilar con los símbolos de depuración:
  - -g (para Host)
  - -G (para Device)

```
user# nvcc -g -G miPrograma.cu
```

- Al ejecutar cuda-gdb se ingresa al modo comando del depurador y pueden realizarse distintas tareas:
  - Obtener información de GPU (info cuda devices)
  - Ejecutar (run)
  - Crear breakpoints
  - Etc.

```
user# cuda-gdb miPrograma
NVIDIA (R) CUDA Debugger
8.0 release
Portions Copyright (C) 2007-2016 NVIDIA Corporation
GNU gdb (GDB) 7.6.2
…
(cuda-gdb)
```