TALLER DE PROGRAMACIÓN SOBRE GPUS

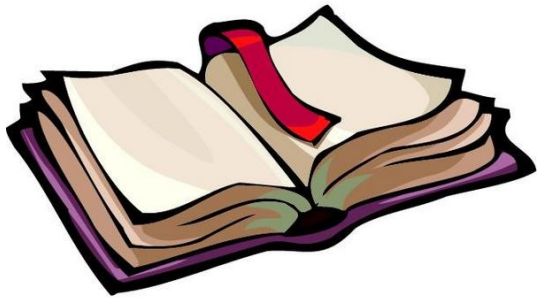
Facultad de Informática – Universidad Nacional de La Plata

Dr. Adrián Pousa

Modelo de Programación CUDA

Agenda

I. *Introducción al modelo de programación Nvidia CUDA*
II. *Estructura de programa CUDA*
    I. *Declaración de variables*
    II. *Gestión de la memoria global*
    III. *Gestión de hilos*
    IV. *Kernel*
    V. *Modularidad*
    VI. *Variables Built-in y Thread ID*
    VII. *Planificación*
    VIII. *Manejo de errores*
    IX. *Limitaciones*
III. *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 global

   III. Gestión de hilos

   IV. Kernel

   V. Modularidad

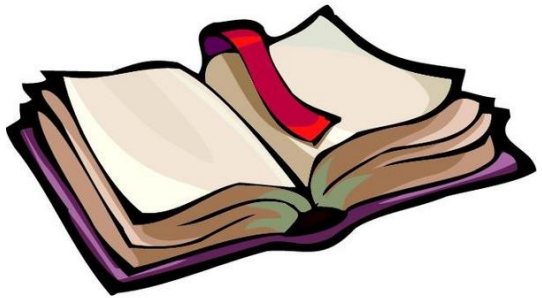
   VI. Variables Built-in y Thread ID

   VII. Planificación

   VIII. Manejo de errores

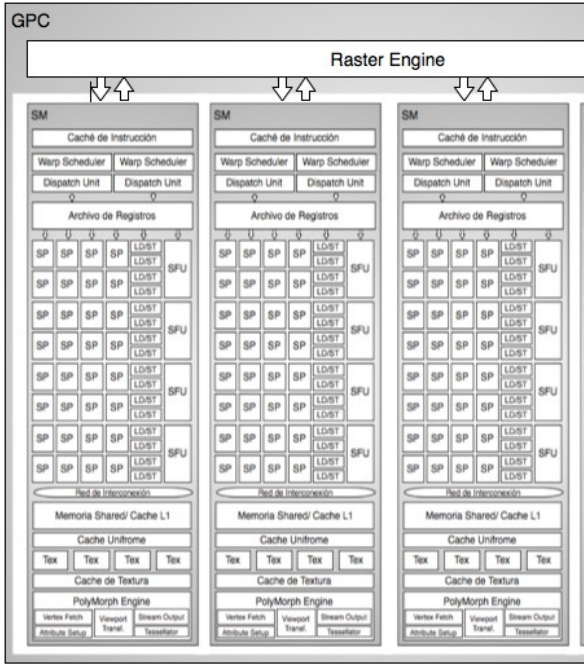
   IX. Limitaciones

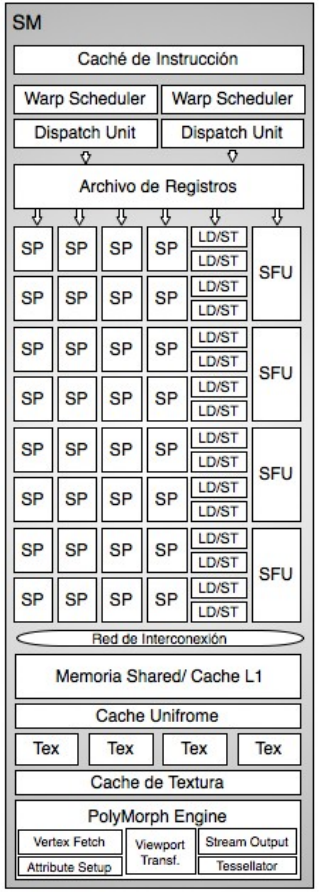
III. Métricas y depuración



Arquitectura Nvidia

- Arquitectura Nvidia:
  - Multiprocesadores SM.
    - SPs.
    - SFU.
    - Load/Store.
  - Jerarquía de Memoria.
    - Global.
    - Textura.
    - Constante.
    - Shared.





Nvidia CUDA

- *GPUs implementan en hardware el pipeline gráfico (por naturaleza paralelo intensivo en cómputo).*
- GPUs evolucionan mejorando el rendimiento del pipeline grafico.
- Modelos de arquitectura fija evolucionan a modelos unificados programables pero de programación compleja.
- En 2006 Nvidia comercializa la serie 8 (G80) y CUDA para facilitar su programación.

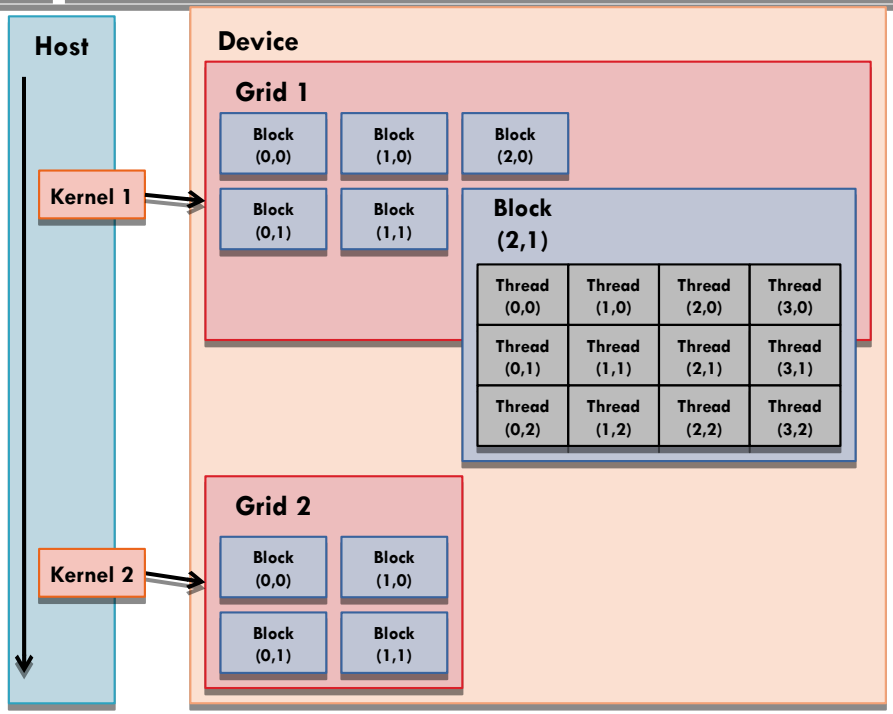
Nvidia CUDA

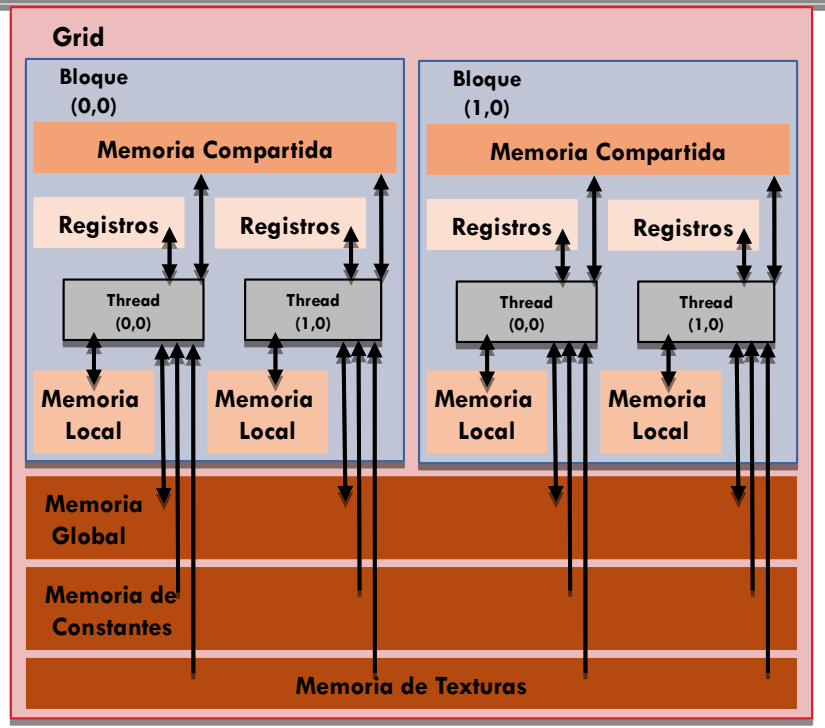
- ***C**ompute **U**nified **D**evice **A**rchitecture* (CUDA): plataforma para cómputo paralelo que incluye un compilador y un conjunto de herramientas de desarrollo creadas por Nvidia que permiten a los programadores usar una extensión del lenguaje de programación C para implementar algoritmos sobre 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.

Nvidia CUDA

- La extensión al lenguaje C se basa en dos características:
  - La jerarquía de memoria y sus costos de acceso:
    - Memoria Global (Lectura y Escritura por CPU y GPU).
    - Memoria Compartida (shared) (Lectura y Escritura solo en GPU).
    - Memoria de Constantes (Escritura por CPU y solo Lectura en GPU).
    - Memoria de Texturas (Escritura por CPU y solo Lectura en GPU).
    - Memoria Local (Lectura y Escritura solo en GPU).
    - Registros (Lectura y Escritura solo en GPU).
  - La organización de trabajo entre los Threads:
    - Grids
    - Bloques.
    - Threads.

Nvidia CUDA


Host
Kernel 1
Kernel 2
Device
Grid 1
Block (0,0)
Block (1,0)
Block (2,0)
Block (0,1)
Block (1,1)
Block (2,1)
Thread (0,0)
Thread (1,0)
Thread (2,0)
Thread (3,0)
Thread (0,1)
Thread (1,1)
Thread (2,1)
Thread (3,1)
Thread (0,2)
Thread (1,2)
Thread (2,2)
Thread (3,2)
Grid 2
Block (0,0)
Block (1,0)
Block (0,1)
Block (1,1)


Grid
Bloque (0,0)
Bloque (1,0)
Memoria Compartida
Registros
Thread (0,0)
Thread (1,0)
Memoria Local
Memoria Global
Memoria de Constantes
Memoria de Texturas

Agenda

I. *Introducción al modelo de programación Nvidia CUDA*

II. ***Estructura de programa CUDA***

   I. ***Declaración de variables***

   II. ***Gestión de la memoria global***

   III. ***Gestión de hilos***

   IV. ***Kernel***

   V. ***Modularidad***

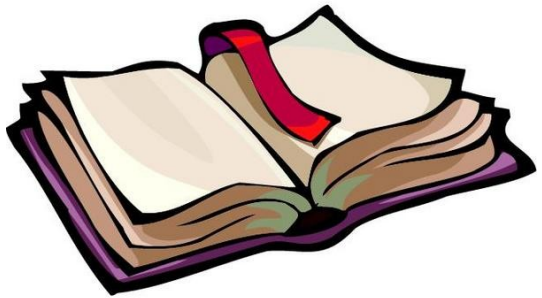
   VI. ***Variables Built-in y Thread ID***

   VII. ***Planificación***

   VIII. ***Manejo de errores***

   IX. ***Limitaciones***

III. *Métricas y depuración*



Nvidia CUDA – Programa CUDA

- El código CUDA se almacena con extensión .cu

- Para compilar en Linux:

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

- Para imprimir salida de texto en pantalla (printf):

```
nvcc -arch=sm_20 -o ejecutable fuente.cu
```

Nvidia CUDA – Estructura de programa


```
#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 global*

   III. *Gestión de hilos*

   IV. *Kernel*

   V. *Modularidad*

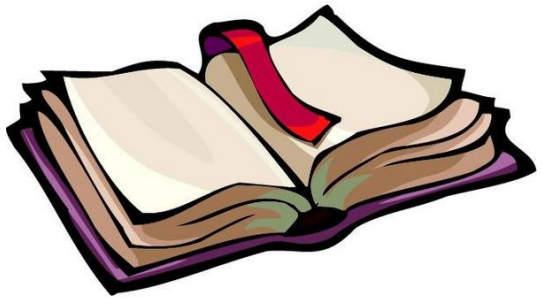
   VI. *Variables Built-in y Thread ID*

   VII. *Planificación*

   VIII. *Manejo de errores*

   IX. *Limitaciones*

III. *Métricas y depuración*



Nvidia CUDA – variables y constantes


```
#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
 }
```

Nvidia CUDA – variables y constantes

- Declaración de variables en memoria de la GPU (por convención d_ delante, no obligatorio):
  - En memoria Global:

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

  - En memoria de Constantes:

    ```
    __constant__ int d_N=1000;
    __constant__ int d_arregloConstante[4]={2,4,6,8}
    ```

  - En caso de arreglos puede no utilizarse el identificador y queda explícito al momento de alocar memoria en GPU:

    ```
    Tipo *d_miArreglo;
    cudaMalloc(&d_miArreglo, N*sizeof(Tipo));
    ```

Nvidia CUDA – variables y constantes

- CUDA no redefine tipos de datos primitivos, solo agrega algunos tipos nuevos como Dim3 y tipos vectoriales (int2, int3, int4, float2, float3, float4, char2 … etc).

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

  Se acceden: miVariable.x, miVariable.y, miVariable.z, miVariable.w.

- Los tipos de datos básicos son los primitivos de C/C++:
  - int
  - float
  - char
  - double
  - signed / unsigned.

Agenda

I. *Introducción al modelo de programación Nvidia CUDA*

II. ***Estructura de programa CUDA***

   I. *Declaración de variables*

   II. ***Gestión de la memoria global***

   III. *Gestión de hilos*

   IV. *Kernel*

   V. *Modularidad*

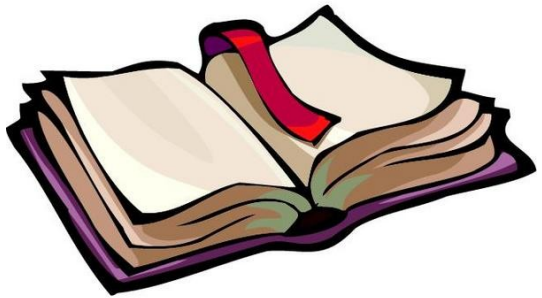
   VI. *Variables Built-in y Thread ID*

   VII. *Planificación*

   VIII. *Manejo de errores*

   IX. *Limitaciones*

III. *Métricas y depuración*



Nvidia CUDA – gestión de 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
 }
```

Nvidia CUDA – gestión de memoria

- 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**.

Nvidia CUDA – gestión de memoria

- 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)
```

Nvidia CUDA – gestión de memoria

- Hay varias formas de hacer la copia explícita de los datos desde la CPU a la GPU, dos de ellas son:

Función **cudaMemCpy** (Para transferencias hacia y desde memoria Global):

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

Donde direction puede ser:

- CudaMemCpyHostToDevice (Para transferencias H2D)
- CudaMemCpyDeviceToHost (Para transferencias D2H)
- CudaMemCpyDeviceToDevice (Para transferencias D2D entre múltiples GPUs)

Función **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)

En la invocación, symbol es el identificador del dato declarada como __constant__

Nvidia CUDA – gestión de memoria

- Las funciones anteriores son **Sincrónas** (el programa espera que la transferencia termine).
- Hay formas de hacerlo **Asíncrono**:

cudaMemcpyAsync

cudaMemcpyToSymbolAsync

  - Útil en caso de más de una GPU.
  - El llamado puede retornar antes que la copia se complete.
  - No puede hacerse directamente, sólo en conjunto con CUDA Streams.

Nvidia CUDA – gestión de memoria


```
int n=512
      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);
       "Ejecucion 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);
```

Nvidia CUDA – gestión de memoria

- Ejemplo de copia a memoria de 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(){
    …
    cudaMemcpyToSymbol(d_N,&h_N,sizeof(unsigned long));
    cudaMemcpyToSymbol(d_Nv,&h_Nv,3*sizeof(unsigned long));
    …
}
```

Nvidia CUDA – gestión de memoria

- **Copia a memoria de constantes – Limitaciones:** en el caso de vectores, las dimensiones deben ser conocidas en tiempo de compilación.

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

Nvidia CUDA – gestión de memoria

- En ocasiones necesitamos pasar un vector con valores idénticos (ejemplo: un vector inicializado en 0).
- Para evitar la transferencia de este vector, alocamos y luego hacer que la GPU lo inicialice.
- Para esto se utiliza la función:

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

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

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

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

Nvidia CUDA – gestión de memoria

- **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;

Nvidia CUDA – gestión de memoria

- **Binding en memoria de texturas:** antes de utilizar la memoria de texturas es necesario hacer un bind, 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 usado se debe hacer un unbind:

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

Nvidia CUDA – gestión de memoria

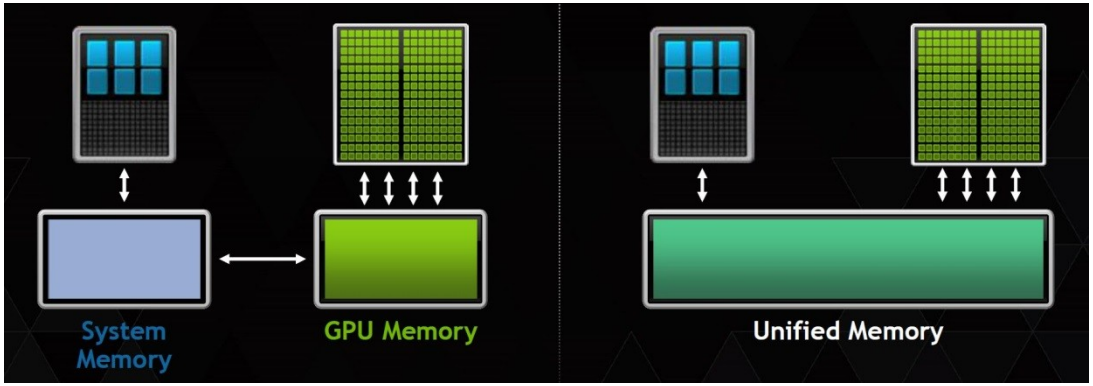
- **Lectura de la memoria de textura dentro del kernel:** la lectura se hace mediante las funciones
  - tex1Dfetch(textura, posX)
  - tex2Dfetch(textura, posX, posY)
  - tex3Dfetch(textura, posX, posY, posZ)

Nvidia CUDA – gestión de memoria


```
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>>>(); "Ejecucion en GPU"
        cudaDeviceSynchronize();
        cudaUnbindTexture(miTextura);
        cudaFree(d_datos);
}
```

Nvidia CUDA – gestión de memoria

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



Nvidia CUDA – gestión de memoria


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


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

Nvidia CUDA – gestión de memoria

- La cantidad de memoria unificada disponible está dada por el tamaño de la memoria de la GPU.
- Las páginas de memoria de asignaciones unificadas modificadas por la CPU deben ser migradas 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 global*

   III. ***Gestión de hilos***

   IV. *Kernel*

   V. *Modularidad*

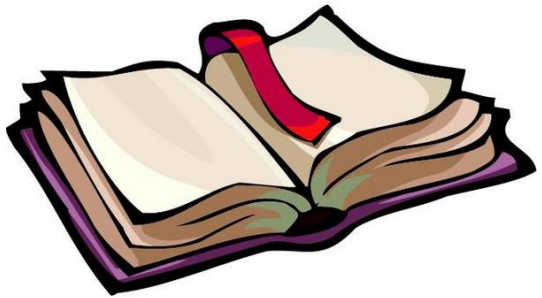
   VI. *Variables Built-in y Thread ID*

   VII. *Planificación*

   VIII. *Manejo de errores*

   IX. *Limitaciones*

III. *Métricas y depuración*



Nvidia 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
 }
```

Nvidia CUDA – gestión de hilos

- Los hilos se organizan en tres niveles cuyo tamaño define el usuario (en CPU):
  - **Grids:** Conjunto de bloques.
  - **Bloques:** Conjunto de threads.
  - **Threads:** hilos propiamente dicho.

- Antes de ejecutar sobre la GPU hay que indicar:
  - Dimensión de grid (1 dimensión o 2 dimensiones de bloques) que determina implícitamente la cantidad de bloques.
  - Dimensión de bloque (1 dimensión, 2 dimensiones o 3 dimensiones de hilos) que determina implícitamente la cantidad de hilos del bloque.

- Esto se define en variables tipo dim3

Nvidia CUDA – gestión de hilos - grid

grid unidimensional de 3 bloques de hilos

dim3 miGrid1D(3,1,1)

grid bidimensional de 3x2 bloques de hilos

dim3 miGrid2D(3,2,1)

El último parámetro no se utiliza.

Nvidia CUDA – gestión de hilos - grid

Si se tiene un hilo ejecutando en la GPU y se necesita saber a que bloque pertenece, se puede identificar mediante las variables built-in:

blockIdx.x : da la coordenada x del bloque de hilos dentro del grid.

blockIdx.y : da la coordenada y del bloque de hilos dentro del grid, para grid unidimensional este valor es siempre 0.

Nvidia CUDA – gestión de hilos - bloques

Bloque unidimensional de 3 de hilos

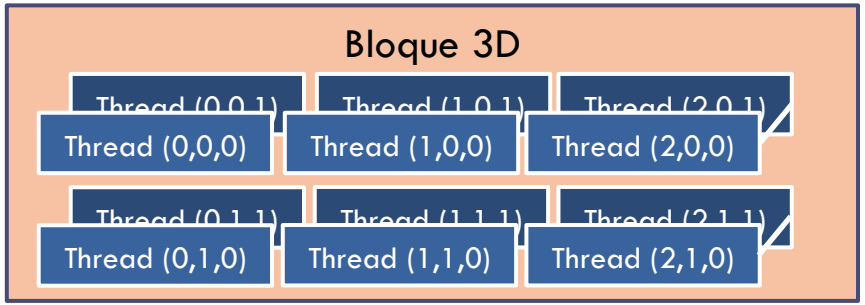
dim3 miBloque1D(3,1,1)

Bloque bidimensional de 3x2 hilos

dim3 miBloque2D(3,2,1)

Bloque tridimensional de 3x2x2 hilos

dim3 miBloque3D(3,2,2)



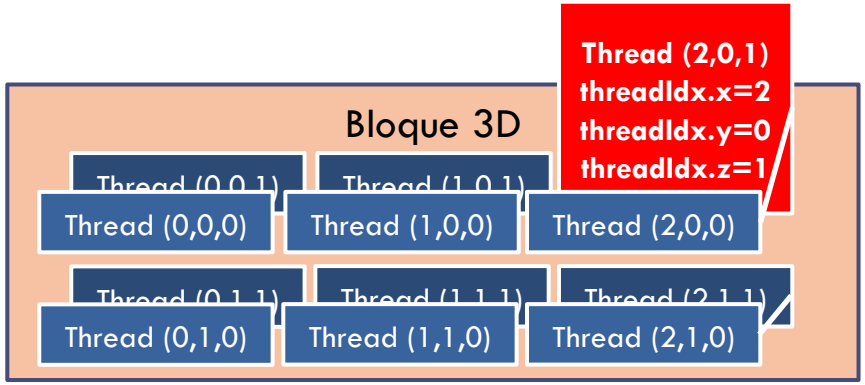
Nvidia CUDA – gestión de hilos - bloques

Si un hilo quiere saber cual es su identificación dentro de su bloque se utilizan las variables built-in:

threadIdx.x : da la coordenada x del hilo dentro del bloque.

threadIdx.y : da la coordenada y del hilo dentro del bloque, para bloques 1D este valor es siempre 0.

threadIdx.z : da la coordenada z del hilo dentro del bloque, para 1D y 2D este valor es siempre 0.



Nvidia CUDA – gestión de hilos - threads

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

Nvidia CUDA – Variables dim3

- 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;
…
```

Nvidia CUDA – gestión de hilos - threads

Pero... ¿Cuantos bloques o hilos creo?

- Generalmente, en cada arquitectura hay un tamaño de hilos por bloque (**threadsPerBlock**) adecuado (128, 256, 512 etc).
- Además, podemos partir del tamaño del problema (**N**)
- Aunque todo 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?

- Ejecutamos nuestro programa parametrizado de la siguiente forma:

./miPrograma N threadsPerBlock

- Luego, calculamos en ejecución 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 global*

   III. *Gestión de hilos*

   IV. ***Kernel***

   V. *Modularidad*

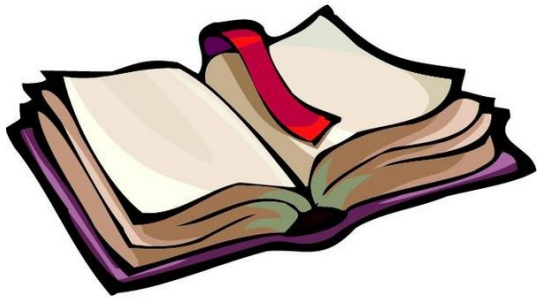
   VI. *Variables Built-in y Thread ID*

   VII. *Planificación*

   VIII. *Manejo de errores*

   IX. *Limitaciones*

III. *Métricas y depuración*



Nvidia 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
 }
```

Nvidia CUDA – kernel

- Se llama **kernel** a la función que ejecutarán todos los threads del grid.

- Se la define de la siguiente forma:

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

- La función tiene el identificador **__global__** que la identifica como kernel y no tiene valor de retorno.

Nvidia CUDA – kernel

- La invocación a esta función se hace de la siguiente forma:

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

En la GPU se ejecutarán: N*N*M*M threads.

- Los parámetros no necesitan ningún tratamiento especial ni tampoco se definen con palabras claves ni identificadores.

Nvidia CUDA – kernel

- 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();
…
```

Nvidia CUDA – kernel

- La función **cudaDeviceSynchronize** demora la ejecución en la CPU hasta que termine la invocación del kernel.
- Esta función también es útil para saber si el kernel terminó o no de forma exitosa.
- La función cudaDeviceSynchronize retorna un error de tipo cudaError_t:
  - Si el kernel terminó con éxito el valor es CUDASUCCESS.
  - Si terminó con error retorna un código que se encuentra detallado en el sitio de Nvidia y permite identificar el problema. https://docs.nvidia.com/cuda/index.html
- Ejemplo:

```
…
cudaError_t error;
miFuncion<<<grid, bloque>>>(parámetros);
error = cudaDeviceSynchronize();
…
```

Nvidia CUDA – kernel

- A veces, necesitamos llamar varias veces a un mismo kernel o hacer invocaciones con dos kernels diferentes 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.
- Ejemplo:

```
…
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
…
```

Agenda

I. *Introducción al modelo de programación Nvidia CUDA*

II. ***Estructura de programa CUDA***

   I. *Declaración de variables*

   II. *Gestión de la memoria global*

   III. *Gestión de hilos*

   IV. *Kernel*

   V. ***Modularidad***

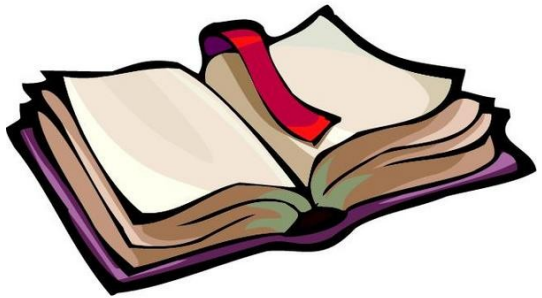
   VI. *Variables Built-in y Thread ID*

   VII. *Planificación*

   VIII. *Manejo de errores*

   IX. *Limitaciones*

III. *Métricas y depuración*



Nvidia CUDA – kernel - Modularidad

- El kernel puede invocar funciones para permitir un código modular.
- Toda función invocada por el kernel deberá llevar delante el identificador __device__.
- Si una función no tiene ningún identificador o tiene el identificador __host__ solo podrá ser invocada desde la CPU y no de la GPU.
- Si una función puede ser invocada tanto en la CPU como en la GPU deberá llevar delante tanto el identificador __device__ como el identificador __host__.

Nvidia CUDA – kernel

- **Función invocada sólo por la GPU:**

```
__device__ int funcionAuxGPU(int x){
…
}
```

- **Función invocada en CPU y GPU:**

```
__host__ __device__ int funcionAuxCPUGPU(int x){
…
}
```

- **Función invocada sólo por la CPU:**

```
__host__ int funcionAuxCPU(int x)
{
…
}
```

O bien:

```
int funcionAuxCPU(int x){
…
}
```

Agenda

I. *Introducción al modelo de programación Nvidia CUDA*

II. ***Estructura de programa CUDA***

   I. *Declaración de variables*

   II. *Gestión de la memoria global*

   III. *Gestión de hilos*

   IV. *Kernel*

   V. *Modularidad*

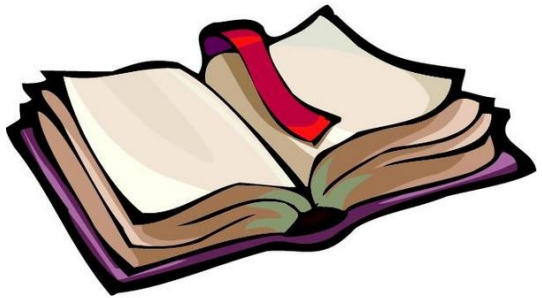
   VI. ***Variables Built-in y Thread ID***

   VII. *Planificación*

   VIII. *Manejo de errores*

   IX. *Limitaciones*

III. *Métricas y depuración*



Nvidia CUDA – kernel – Variables Built-in

- Dentro del kernel se pueden utilizar las variables built-in que permiten ayudar a la identificación de hilos y bloques:

  - Identifican al hilo dentro del bloque:
    - threadidx.x
    - threadidx.y
    - threadidx.z

  - Identifican al bloque dentro del grid.
    - blockIdx.x
    - blockIdx.y:

Nvidia CUDA – kernel – Built-in

- Otras variables built-in se utilizan para obtener dimensiones:
  - Obtienen las dimensiones del grid:
    - dimGrid.x
    - dimGrid.y
  - Obtienen las dimensiones de los bloques:
    - blockDim.x
    - blockDim.y
    - blockDim.z

Nvidia CUDA – kernel – 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
```

Nvidia CUDA – kernel – Thread id

- Ej: 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

Nvidia CUDA – kernel – Thread id

- Grid 1D:
  - Bloques 1D:

```
blockIdx.x *blockDim.x + threadIdx.x;
```

  - Bloques 2D:

```
blockIdx.x * blockDim.x * blockDim.y + threadIdx.y * blockDim.x +
threadIdx.x
```

  - Bloques 3D:

```
blockIdx.x * blockDim.x * blockDim.y * blockDim.z +
    threadIdx.z * blockDim.y * blockDim.x + threadIdx.y * blockDim.x +
threadIdx.x
```

Nvidia CUDA – kernel – Thread id

- Grid 2D:
  - Bloques 1D:

```
blockId   = blockIdx.y * gridDim.x + blockIdx.x;
threadId = blockId * blockDim.x + threadIdx.x;
```

  - Bloques 2D:

```
int blockId = blockIdx.x + blockIdx.y * gridDim.x;
int threadId = blockId * (blockDim.x * blockDim.y) + (threadIdx.y *     blockDim.x) +
threadIdx.x;
```

  - Bloques 3D:

```
int blockId = blockIdx.x  + blockIdx.y * gridDim.x;
int 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 global*

   III. *Gestión de hilos*

   IV. *Kernel*

   V. *Modularidad*

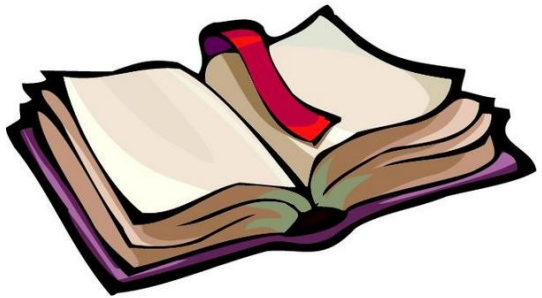
   VI. *Variables Built-in y Thread ID*

   VII. ***Planificación***

   VIII. *Manejo de errores*

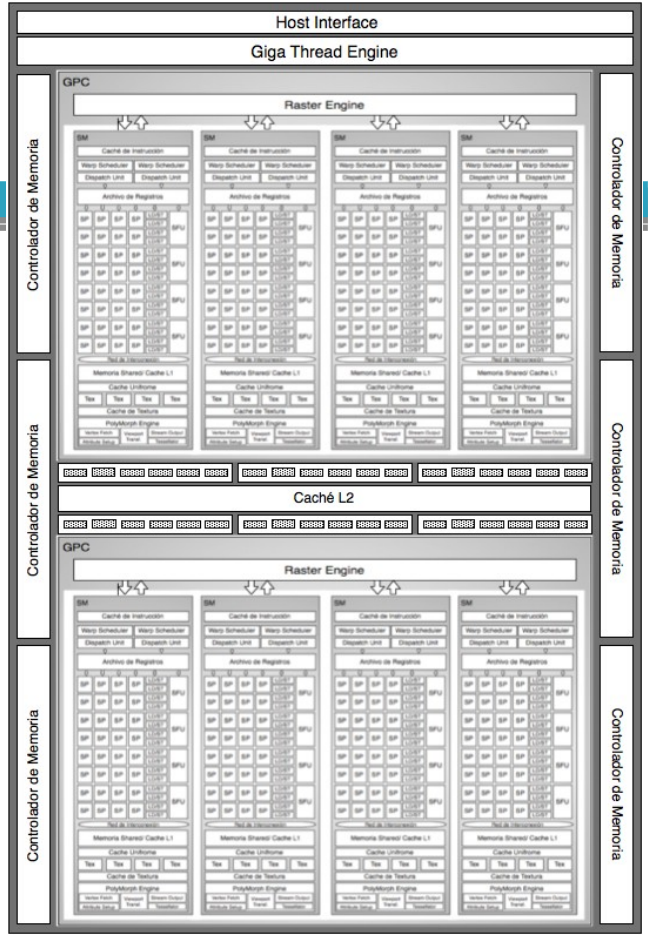
   IX. *Limitaciones*

III. *Métricas y depuración*



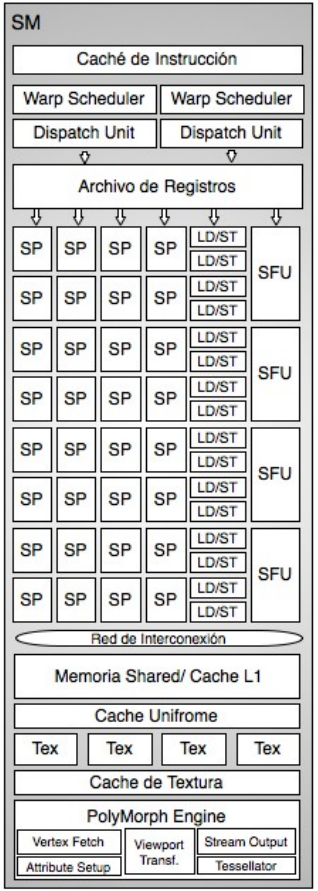
CUDA – Planificación

- Una vez invocado el kernel, se envía el grid a la unidad GigaThread de la GPU y esta se encarga de la planificación.
- La unidad Gigathread envía bloques a los SM. Cada SM puede planificar varios bloques.
- El usuario solo define la organización de los hilos pero no puede decidir sobre la planificación (por ej: que bloques va a que SM).



CUDA – Planificación

- Un SM recibe un bloque y lo divide en warps (32 threads) , un warp es la unidad de planificación.
- Los hilos de un WARP ejecutan la misma instrucción (SIMD).
- El planificador dual de warps selecciona 2 warps y da una instrucción de cada warp a 16SPs, 16 unidades Load/Store o 4 SFU.
- La mayoría de las instrucciones pueden ser lanzadas dual:
  - Se pueden lanzar 2 operaciones enteras, Load/Store y SFU concurrentemente.
  - Las instrucciones de doble precisión no pueden ser lanzadas concurrentemente.

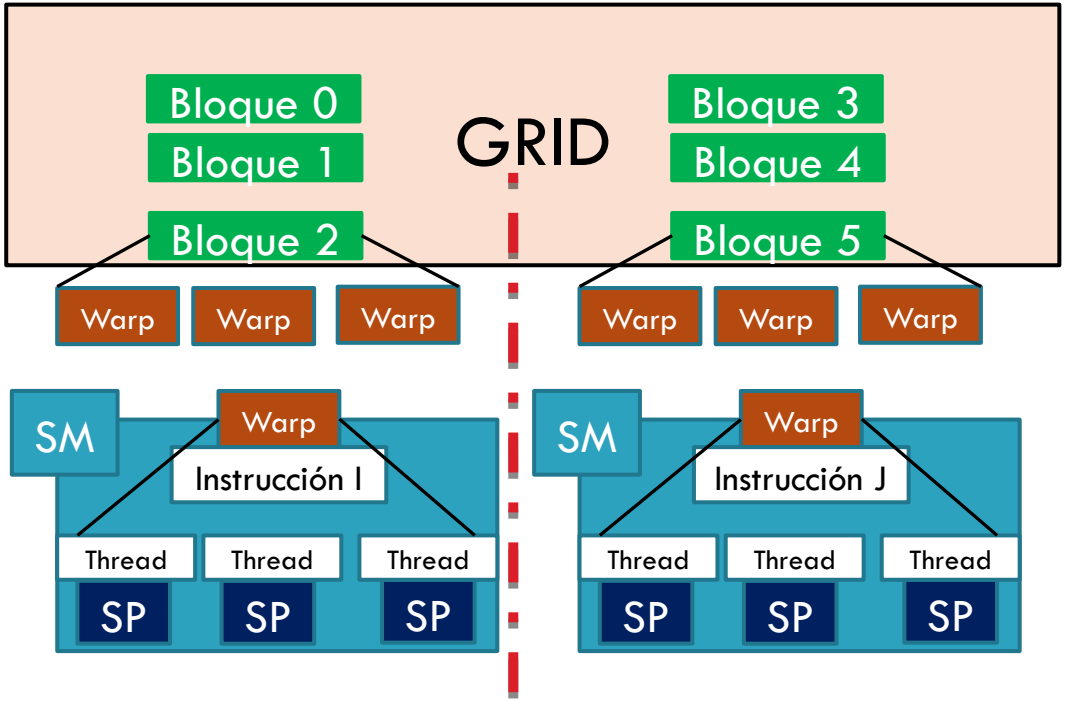


CUDA – Planificación

- 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

CUDA – Planificación



Agenda

I. *Introducción al modelo de programación Nvidia CUDA*

II. ***Estructura de programa CUDA***

   I. *Declaración de variables*

   II. *Gestión de la memoria global*

   III. *Gestión de hilos*

   IV. *Kernel*

   V. *Modularidad*

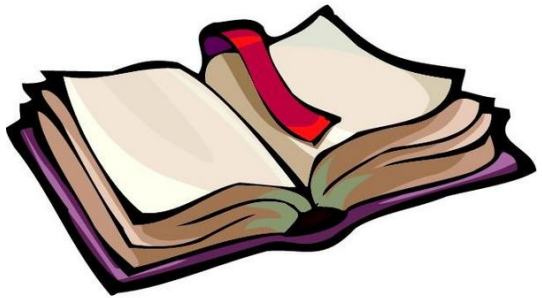
   VI. *Variables Built-in y Thread ID*

   VII. *Planificación*

   VIII. ***Manejo de errores***

   IX. *Limitaciones*

III. *Métricas y depuración*



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.

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 global*

   III. *Gestión de hilos*

   IV. *Kernel*

   V. *Modularidad*

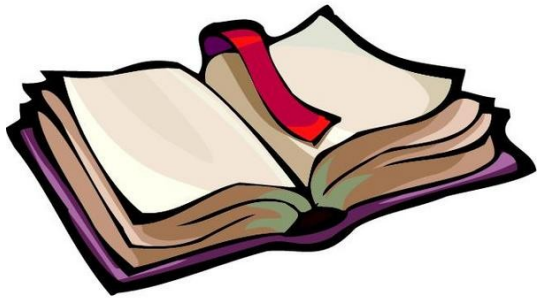
   VI. *Variables Built-in y Thread ID*

   VII. *Planificación*

   VIII. *Manejo de errores*

   IX. ***Limitaciones***

III. *Métricas y depuración*



CUDA – Limitaciones

- No se puede hacer E/S a disco desde la GPU.
- Dificultad para utilizar bibliotecas de CPU en GPU: en todo lo que se quiera reutilizar en GPU debe indicarse que se ejecuta en CPU y que en GPU, y luego recompilarse.
- No se pueden hacer llamadas recursivas dentro de la GPU.
- No se pueden declarar variables estáticas.
- No se permite pasar al kernel un número variable de argumentos.
- No existe el concepto de afinidad, la planificación es transparente, sólo se indica la cantidad de hilos y el código a ejecutar por cada uno.
- Una función que se utiliza en host y 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 global*

   III. *Gestión de hilos*

   IV. *Kernel*

   V. *Modularidad*

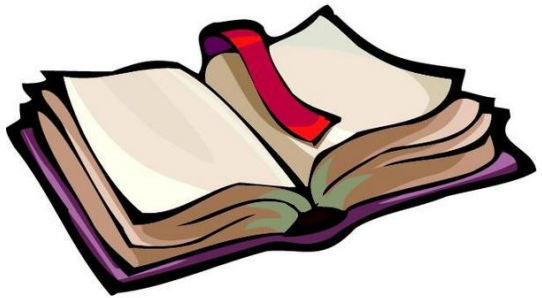
   VI. *Variables Built-in y Thread ID*

   VII. *Planificación*

   VIII. *Manejo de errores*

   IX. *Limitaciones*

III. ***Métricas y depuración***



CUDA – Métricas

- Suponer un programa al cual se le hizo alguna mejora.
- Si se quiere saber cual fue el beneficio alcanzado o speedup se calcula:

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

  - 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 ejecutado sobre distintas arquitecturas.

CUDA – Métricas

- Esta métrica puede ser usada para saber cual es el beneficio obtenido por la GPU con respecto a una CPU u otra arquitectura paralela (multicores, clusters etc).
- Por lo tanto el speedup en este caso se calcula como:

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

$t_{secuencial}$: tiempo de ejecución del algoritmo secuencial

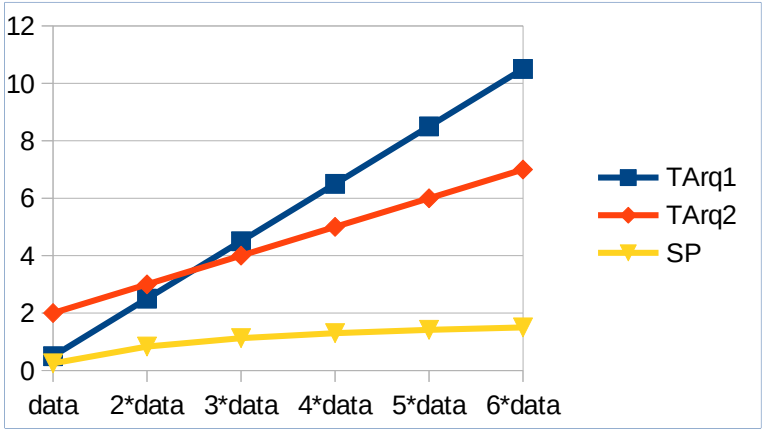
t_{GPU}: tiempo de ejecución del algoritmo sobre GPU

CUDA – Métricas

- Suponer dos arquitecturas Arq1 y Arq2.

$$SP = \frac{TArq1}{TArq2}$$

	TArq1	TArq2	SP
data	0,5	2	0,25
2*data	2,5	3	0,83
3*data	4,5	4	1,13
4*data	6,5	5	1,30
5*data	8,5	6	1,42
6*data	10,5	7	1,50



- Si $SP < 1$; (TArq1<TArq2): mejor rendimiento en Arq1 que en Arq2.
- Si $SP = 1$; (TArq1<TArq2): mismo rendimiento en ambas arquitecturas.
- Si $SP > 1$; (TArq1>TArq2): mejor rendimiento en Arq2 que en Arq1.

CUDA – Métricas - ¿Como tomar tiempos?

- Descargar el archivo de ejemplo *cuadrado.cu* que utiliza la función *dwalltime()*
- Con *dwalltime()* podemos tomar los tiempos de cada parte que nos interese.
- 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 = dwalltime();
 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 = dwalltime() - tiempo_inicial;
```

CUDA – Otras Métricas

- **Ancho de banda teórico:** se calcula a partir de las especificaciones de la placa.
- Medido 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:

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

CUDA – Otras Métricas

- **Ancho de banda efectivo:** se calcula a partir de la aplicación.
- Medido 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.

CUDA – Otras Métricas

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

CUDA – Otras Métricas

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

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

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

CUDA – Depuración

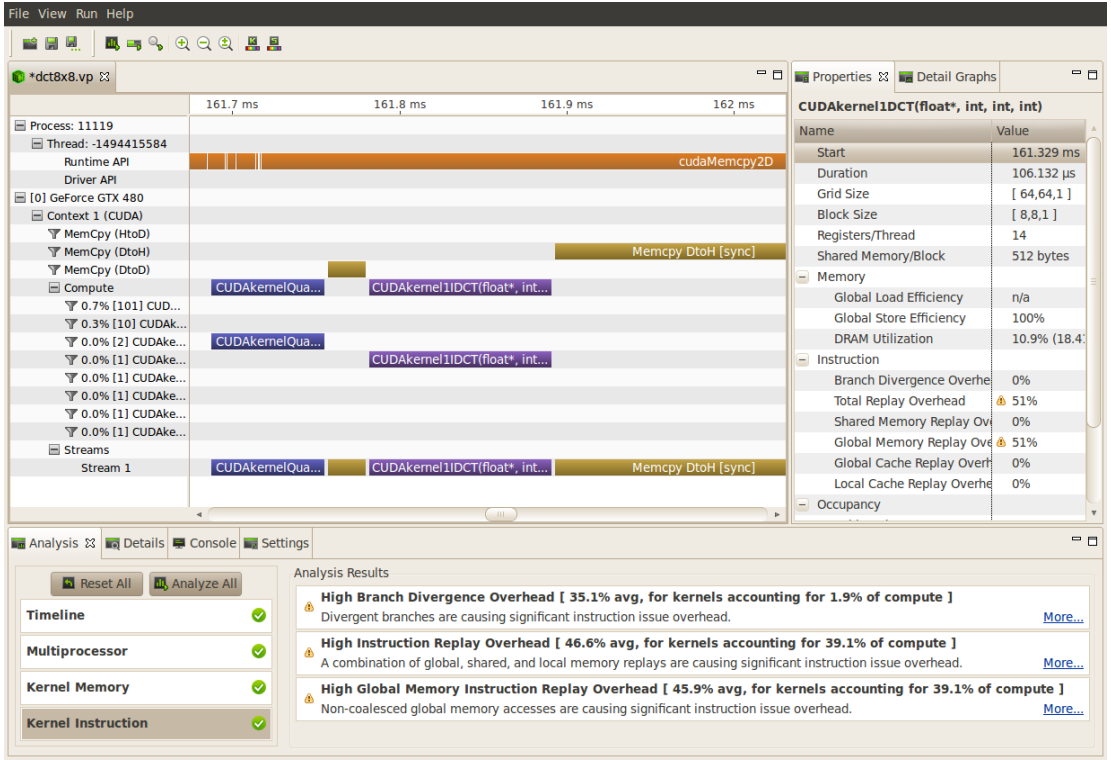
- Podemos obtener información sobre la ejecución de un programa CUDA a partir de herramientas proporcionadas por Nvidia:
  - Comando nvprof: es una herramienta de profiling que permite recopilar información y ver datos de perfiles desde la línea de comandos. Recopila actividades de la CPU y la GPU, ejecución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

CUDA – 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
```

CUDA – Depuración - nvvp


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



CUDA – Depuración - cuda-gdb


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

```
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)
```

- Se recomienda compilar con los símbolos de depuración -g (para Host) y -G (para Device).
- Al ejecutar cuda-gdb se ingresa al modo comando del depurador y se pueden realizar distintas tareas:
  - Ejecutar (run)
  - Crear breakpoints
  - Obtener información de los dispositivos (info cuda devices)
  - Etc.