Computación de Altas Prestaciones
Seminario sobre GPGPUs

Contenidos

Sesión 1 Teoría (12-3): Introducción a Computación en GPUs con CUDA, Compilación, conceptos básicos. Presencial

Sesión 2(31-3): Programación de algoritmos “trivialmente paralelos”. Online

Sesión 3(2-4): Uso de la memoria “Shared”. “Reducciones” en GPUs. Presencial

Sesión 4(7-4): Optimización, temas avanzados, librerías. Online

Sesión 5(9-4): Evaluación. Presencial

Evaluación GPU

- Se hará en los ordenadores de los laboratorios 0 y 6. Habrá que usar los ordenadores del aula obligatoriamente. No se podrá tener encima ningún dispositivo con conexión a Internet (móviles, portátiles, smartwatch,...)
- Los ordenadores no tendrán acceso a Internet. Se os facilitará un zip con las transparencias y con los ejercicios resueltos en clase.
- La evaluación durará hora y media. En caso de no poder acabar el ejercicio en hora y media, se permitirá entregar a lo largo del día siguiente, con una nota máxima del 35%.
- No se aceptarán soluciones a los ejercicios propuestos con técnicas no explicadas en clase.

Tarjeta gráfica y su GPU

El objetivo fundamental de la tarjeta gráfica es encargarse de la visualización. Toman mucha fuerza al surgir los sistemas operativos gráficos (Windows, Mac)

-Visualización 2D
-Ejecución de video
-Gráficos 3D

Tarjeta gráfica y su GPU

-A finales de los 90 y comienzos de los 2000, las GPUs se van haciendo cada vez más potentes.
-Operaciones principales : rendering, shading,... operaciones sencillas que hay que realizar sobre muchos pixeles.
-Se desarrollan GPUs con muchos elementos de computación, programables, relativamente sencillos (sin ejecución out of order, sin ejecución especulativa, sin caches).
-Hasta ese momento, la CPU manda ordenes a GPU, que ejecuta ordenes gráficas. GPGPU surge cuando aparece la posibilidad de mandar resultados de vuelta a CPU

Tarjeta gráfica y su GPU

-La potencia de cálculo acumulada de todos esos cores empieza a ser mayor que la de las CPUs

Surge la idea de usarla para computación de carácter general.
Inicialmente:
-difíciles de programar (OpenGl, DirectX), poca flexibilidad (programar otros cálculos como si fueran operaciones de gráficos)

-poca precisión (no había números reales)

-Sin medios de depurar

CUDA

CUDA (*Compute Unified Device Architecture*) es (en el momento de presentación, 2006) una nueva arquitectura para tarjetas gráficas con elementos diseñados para cálculo general con GPUs.

-Para facilitar el acceso a esas tarjetas, Nvidia creó CUDA-C: Una serie de “añadidos” a C estándar para permitir el acceso

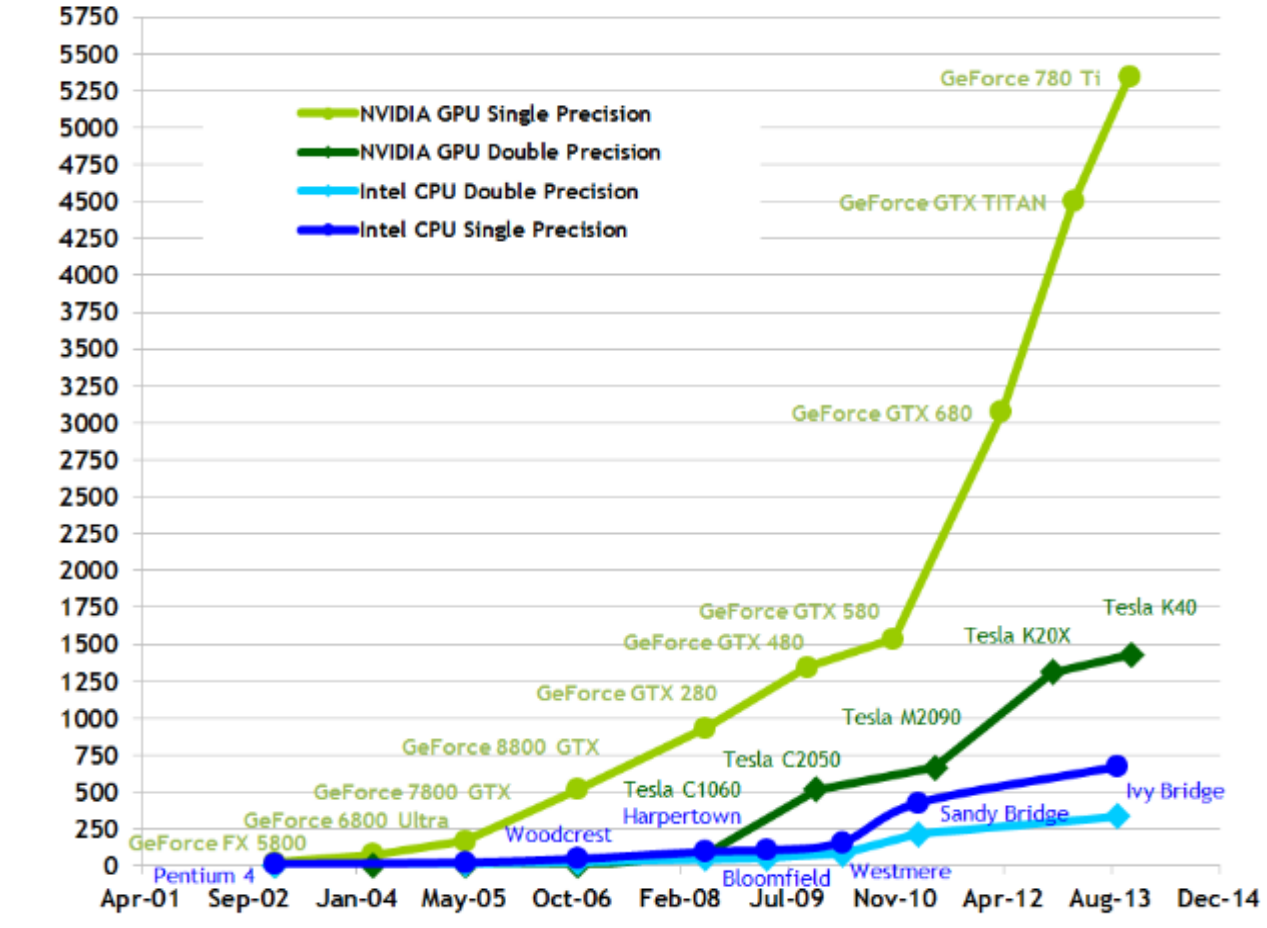
CUDA:

Definición actual de CUDA:
“a general purpose parallel computing platform and programming model that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU.”

-La principal forma de programar GPUs de Nvidia es mediante CUDA-C. Sin embargo, hay otros lenguajes que lo permiten: FORTRAN, DirectCompute, OpenACC, o desde Matlab.

Porque CUDA?

Figure 1. Floating-Point Operations per Second for the CPU and GPU



http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#introduction

Aplicaciones de CUDA

Se han desarrollado aplicaciones de CUDA prácticamente en cualquier campo en el que sea necesaria computación masiva:

-Medicina
-Biología
-Astrofísica
-Dinámica de fluidos computacional
-Estudio del clima
-Geofísica
-Economía
-Big Data
-Estadística
-...

La lista no acaba, porque crece cada día.

Librerías aceleradas con CUDA

Hay muchos casos de librerías aceleradas con CUDA, para cuyo uso no es necesario escribir código CUDA.
Lista actualizada en:
https://developer.nvidia.com/gpu-accelerated-libraries#signal

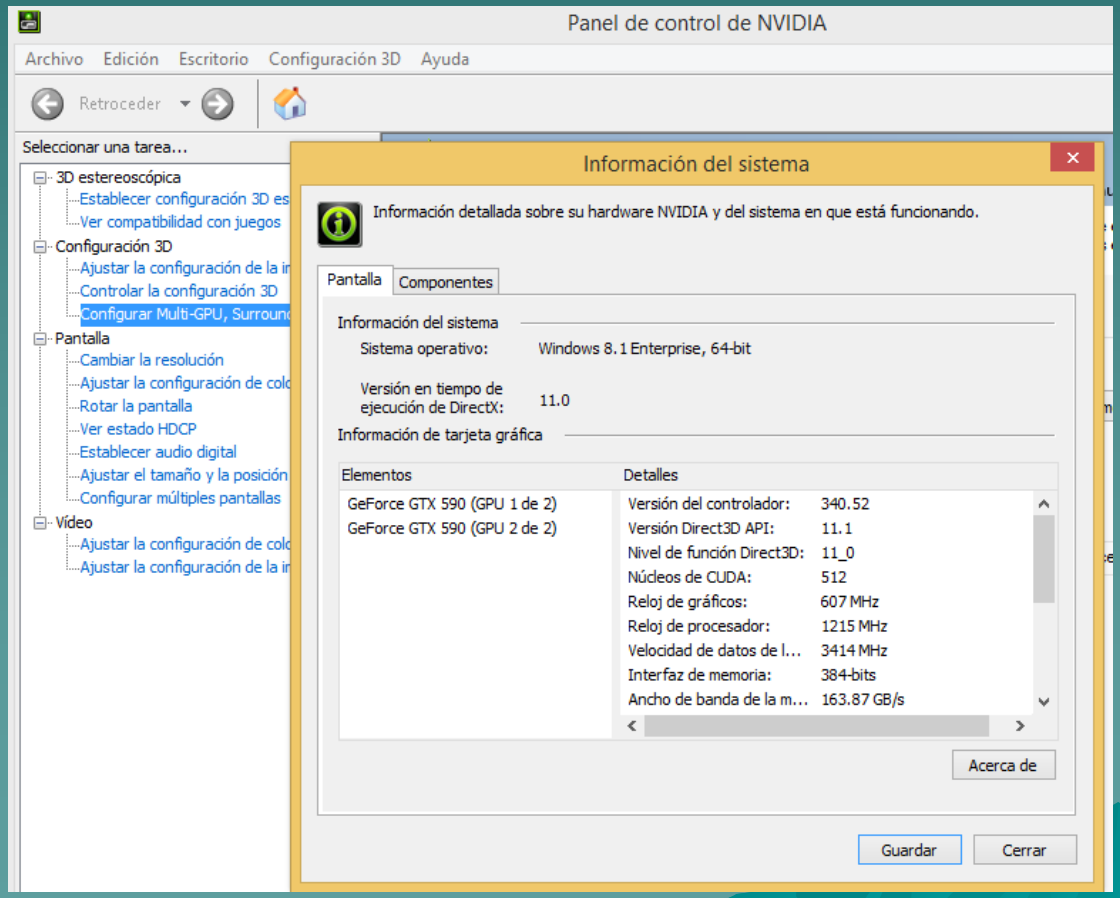
Alternativas a CUDA

OpenCL es un lenguaje bastante similar a CUDA. OpenCL es Open Source (apoyado sobre todo por AMD), mientras que CUDA es propiedad de Nvidia.

En general se considera que CUDA es una tecnología mas madura (y eficiente) que OpenCL. Por ejemplo, Matlab (a través del Parallel Computing Toolbox) soporta tarjetas gráficas que soporten CUDA. Las tarjetas Nvidia soportan también OpenCL

Como podemos usar CUDA en nuestro ordenador?

1) Necesitamos una GPU NVIDIA compatible con CUDA (Prácticamente todas las tarjetas Nvidia son ya compatibles con CUDA: GeForce, Quadro, Tesla...)



Como podemos usar CUDA en nuestro ordenador?

2) Tenemos que instalar un “device driver” apropiado (el último disponible para la tarjeta servirá).
Lo ideal es tener un sistema con una GPU para visualización y otra para cálculo, pero puede ser difícil de configurar.

3) Tenemos que descargar e instalar el “CUDA Toolkit” de la web de Nvidia.

4) Compilador de C compatible:
-Windows: necesitamos tener instalada alguna versión apropiada de Visual Studio
-En Linux no se necesita software “propietario”; hay que instalarlo en modo “root”

Podéis consultar “Cuda Quick Start Guide” en poliformat
NO podemos depurar programas CUDA con una GPU que se esté usando para controlar una interfaz gráfica de usuario.

Máquinas disponibles en el DSIC

1) PCS de laboratorios 0,6,7,8

2) Máquinas knights.dsic.upv.es y gpu.dsic.upv.es, a la que podeis acceder con ssh desde el dominio dsic.upv.es

-knights.dsic.upv.es: Intel Xeon con 24 cores + 1 Teslas K20c
Tesla K20c: 5Gb memoria, 13 Multiprocesadores con 192 cores cada uno: 2496 cores

-gpu.dsic.upv.es : Core i9 (12 cores) + 2 Quadro RTX 5000 con 16 Gb y 3000 cores cada una

Para averiguar cuantas gpus tenemos en nuestro sistema, podemos ejecutar la aplicación deviceQuery(parte del conjunto de “Samples” que viene con el “CUDA Toolkit”). Tenéis el directorio devicequery comprimido en Poliformat

-Descomprimir paquete en algún directorio donde podáis escribir y ejecutar el makefile

Averiguando características de nuestra(s) GPU(s)

Devicequery muestra por pantalla información sobre la(s) GPUs disponibles

CUDA DeviceQuery (Runtime API) versión (CUDART static linking)
Detected 1 CUDA Capabledevice(s)
Device0: "Tesla K20c"
CUDA Driver Version/ RuntimeVersion7.0 / 7.0
CUDA Capability Major/Minor versión number: 3.5
Total amount of global memory: 4800 MBytes(5032706048 bytes)
(13) Multiprocessors, (192) CUDA Cores/MP: 2496 CUDA Cores
GPU Max Clock rate: 706 MHz (0.71 GHz)
Memory Clock rate: 2600 Mhz
Memory Bus Width: 320-bit
L2 Cache Size: 1310720 bytes
Maximum Texture Dimension Size(x,y,z) 1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
Maximum Layered1D Texture Size, (num) layers1D=(16384), 2048 layers
Maximum Layered2D Texture Size, (num) layers2D=(16384, 16384), 2048 layers
Total amount of constant memory: 65536 bytes Total amount of shared memory per block: 49152 bytes
...

Programas CUDA

Aunque pueden hacerse programas en C, C++ o Fortran que usan las GPUs, lo normal es hacer los programas con extensión .cu

Se compila con el compilador nvcc, que hace uso del compilador gcc (en Linux) o del compilador de Microsoft cl (en Windows).

Hay un buen número de opciones de compilación específicas de CUDA, aunque las típicas suelen hacer la misma tarea (-g para depurar, -O3 para optimizar, -o para darle nombre al ejecutable)

Primer programa en CUDA: Hola mundo

```
#include <stdio.h>
__global__   void kernel()
{
}

int main()
{
kernel<<<1,1>>>();
printf("Hello world");
return 0;
}
```

__global__ indica código que se ejecuta en el “device” (GPU)

El main se ejecuta en la CPU; el kernel se ejecuta en la GPU

En este caso, en la GPU no se hace nada. Se compila con el compilador de CUDA nvcc (que llama a gcc)

Estructura general (normal) de programa en CUDA:

Los espacios de memoria de la CPU (HOST) y de la GPU (DEVICE) son diferentes; para poder usar la GPU necesitamos funciones para reservar memoria (CudaMalloc), liberar memoria (CudaFree), Copiar memoria de CPU a GPU y de GPU a CPU (CudaMemcpy)

Programa CPU
(main)
Inicializar memoria CPU
(malloc, leer ficheros,...)
Reservar memoria GPU
(CudaMalloc)
Enviar datos de CPU a GPU
(CudaMemcpy)
Llamar a un kernel(se ejecuta en la GPU)
(nombre_de_kernel<<<x,y>>>(...parametros))
Copiar resultados de la GPU a CPU
(CudaMemcpy)
Liberar memoria
(CudaFree)

Sumar dos enteros en CUDA

```
#include<stdio.h>
__global__ void suma(int a, int b, int *c)
{
  *c=a+b;
}
int main()
{
  int c;
  int *dev_c;

  cudaMalloc ( (void**)&dev_c, sizeof(int) );

  suma<<<1,1>>>(2,7,dev_c);

  cudaMemcpy( &c ,dev_c,  sizeof(int), cudaMemcpyDeviceToHost);

  printf("2+7 = %d\n", c);

  cudaFree(dev_c);
  return 0;
}
```

Sumar dos enteros en CUDA

Observaciones:
1) CudaMalloc y CudaFree equivalen a malloc y free, pero para memoria de la GPU.

2) Los punteros a los que se le ha dado memoria con CudaMalloc no se pueden usar “correctamente“ en el código de la CPU (solo a través de llamadas a kernels y cudamemcpy).

3) De forma similar, un puntero al que se le da memoria con malloc no se debe usar en la GPU.

4) Podemos copiar memoria (datos) entre CPU y GPU usando CudaMemcpy con las opciones cudaMemcpyDeviceToHost o cudaMemcpyHostToDevice.

5) No hace falta reservar memoria para parámetros de tipo simple (no vectores) que se pasen por valor (a,b en el ejemplo anterior). Si los argumentos son vectores, sí que hay que reservar memoria

Programación Paralela en CUDA

-Hemos visto que las GPUs tienen cientos o miles de cores capaces de ejecutar cientos o miles de **threads** simultáneamente.

-Los threads se organizan en "**Bloques**" de Threads; se deben visualizar como "equipos" de Threads que trabajan en paralelo.

Cuando se invoca a un kernel desde el código del host:

kernel<<<x,y>>> (...)

El valor x es el número de bloques de threads que vamos a usar para ejecutar el kernel.

El valor y es el número de threads por bloque que vamos a usar para ejecutar el kernel.

Cada thread dispone de una cantidad (limitada) de memoria local; la memoria a la que se accede a través de los parámetros/punteros es memoria "GLOBAL" y todos los threads pueden acceder a ella.

Programación Paralela en CUDA: Suma de dos vectores: Código C, sin CUDA:

```
#include <stdio.h>
#define N 10

void add(int *a, int *b, int *c)
{
  int tid=0;
  while (tid <N) {
  c[tid]=a[tid]+b[tid];
  tid +=1;
  }
 }

 int main() {
 int a[N], b[N], c[N],i;
 //llenar arrays en cpu
 for (i=0;i<N;i++)
   {
     a[i]=-i;
     b[i]=i*i;
    }
 add(a,b,c);
 for (i=0;i<N;i++)
  printf(" %d + %d = %d\n", a[i],b[i], c[i]);
  }
```

Programación Paralela en CUDA: Suma de dos vectores

Y si tuviéramos dos cores?

```
#include <stdio.h>
#define N 10

void add(int *a, int *b, int *c)
{
  int tid=0;
  while (tid <N) {
  c[tid]=a[tid]+b[tid];
  tid +=2;
  }
 }
```

```
#include <stdio.h>
#define N 10

void add(int *a, int *b, int *c)
{
  int tid=1;
  while (tid <N) {
  c[tid]=a[tid]+b[tid];
  tid +=2;
  }
 }
```

Y si tuviéramos N cores ?

```
#include <stdio.h>
#define N 10

void add(int *a, int *b, int *c, int tid)
{
if (tid <N)
  c[tid]=a[tid]+b[tid];
 }
```

Programación Paralela en CUDA: Suma de dos vectores

Versión CUDA, usando N bloques cada uno con un solo thread: main(1/2)

```
 int main()
{
 int a[N], b[N], c[N],i;
 int *dev_a, *dev_b, *dev_c;
 //reservar memoria en GPU
 cudaMalloc((void **) &dev_a, N*sizeof(int) );
 cudaMalloc((void **) &dev_b, N*sizeof(int) );
 cudaMalloc((void **) &dev_c, N*sizeof(int) );
 //rellenar vectores en CPU
  for (i=0;i<N;i++)
   {
     a[i]=-i;
     b[i]=i*i;
    }
//enviar vectores a GPU
cudaMemcpy( dev_a, a, N*sizeof(int) , cudaMemcpyHostToDevice );
cudaMemcpy( dev_b, b, N*sizeof(int) , cudaMemcpyHostToDevice );
cudaMemcpy( dev_c, c, N*sizeof(int) , cudaMemcpyHostToDevice );
```

Programación Paralela en CUDA: Suma de dos vectores

Versión CUDA, usando N bloques cada uno con un solo thread: main(2/2)

```
 //llamar al Kernel
 add<<<N,1>>>(dev_a,dev_b,dev_c);

 //obtener el resultado de vuelta en la CPU
 cudaMemcpy( c, dev_c, N*sizeof(int), cudaMemcpyDeviceToHost );

//imprimir resultado
 for (i=0;i<N;i++)
  printf(" %d + %d = %d\n", a[i],b[i], c[i]);

//liberar memoria en la GPU
  cudaFree(dev_a) ;
  cudaFree(dev_b) ;
  cudaFree(dev_c) ;
```

Programación Paralela en CUDA: Suma de dos vectores

Kernel:

```
__global__void add(int *a, int *b, int *c)
{
  int tid=blockIdx.x;
  if (tid <N) {
  c[tid]=a[tid]+b[tid];
   }
 }
```

La linea add<<<N,1>>>(dev_a, dev_b, dev_c) lanza N bloques de Threads, cada uno con un solo thread. Cada thread ejecuta el mismo código, pero con datos diferentes.

La variable blockIdx.x me da el número de bloque que ejecuta esta instancia del kernel.

Como mucho podemos lanzar 65535 bloques simultáneamente (En realidad se pueden más, lo veremos otro día). Y si necesitamos más?

Programación Paralela en CUDA: Suma de dos vectores; threads en vez de bloques

Para obtener una versión que usa 1 bloque de N threads (en vez de N bloques de 1 thread) sólo tenemos que hacer estos cambios:

En el main:

```
...
//llamar al Kernel
 add<<<1,N>>>(dev_a,dev_b,dev_c);
...
```

En el kernel:

```
...
int tid=threadIdx.x;
...
```

Pero el número máximo de threads por bloque es 1024 (tampoco vale así) Necesitamos combinar threads y bloques

Programación Paralela en CUDA: Suma de dos vectores; threads en vez de bloques

Para obtener una versión que usa M bloques, cada uno con N threads tenemos que usar en el kernel una nueva variable de CUDA :blockDim.x

```
...
int tid=threadIdx.x+blockIdx.x * blockDim.x;
...
```

BlockDim es una variable que vale lo mismo para todos los bloques, da el número de threads en un bloque.

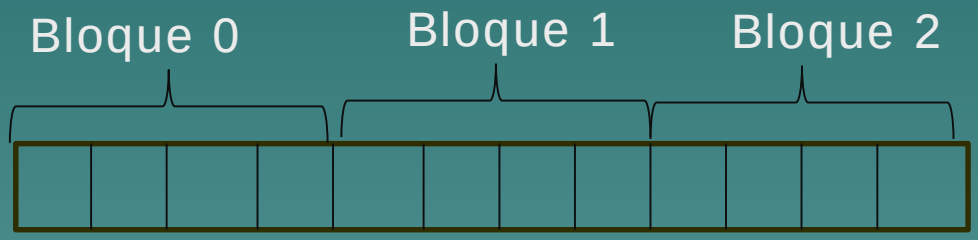
Seleccionamos un número de threads por bloque (256, por ejemplo) y cambiamos la llamada en el main

```
...
//llamar al Kernel
 add<<<(N+255)/256, 256>>>(dev_a,dev_b,dev_c);
...
```

Programación Paralela en CUDA: Suma de dos vectores; threads + bloques

Ejemplo: vector de tamaño 12:

BlockIdx.x



threadIdx.x 0 1 2 3 0 1 2 3 0 1 2 3

```
int tid=threadIdx.x+blockIdx.x * blockDim.x;
```

tid => 0 1 2 3 4 5 6 7 8 9 10 11

Programación Paralela en CUDA: Suma de dos vectores; threads + bloques

Y si todavía necesitamos un vector más grande?
Recordemos la implementación en CPU para el caso de dos threads/cores:

```
#include <stdio.h>
#define N 10

void add(int *a, int *b, int *c)
{
  int tid=0;
  while (tid <N) {
  c[tid]=a[tid]+b[tid];
  tid +=2;
  }
 }
```

El número de bloques lanzados se encuentra en la variable gridDim.x
➔ El número total de threads lanzados es gridDim.x * blockDim.x

Programación Paralela en CUDA: Suma de dos vectores; threads y bloques

Ejemplo: Imaginemos ahora que el trabajo se hace con dos bloques de tres threads cada uno:


```
int tid=threadIdx.x+blockIdx.x * blockDim.x;
..
tid += gridDim.x * blockDim.x; (gridDim*blockDim=numero total de
threads, 6 en el ejemplo )

El thread 0 del  bloque 0 hace el tid=0 y el tid=6
El thread 1 del bloque 0 hace el tid=1 y el tid=7
...
```

Programación Paralela en CUDA: Suma de dos vectores; threads + bloques

El kernel queda:

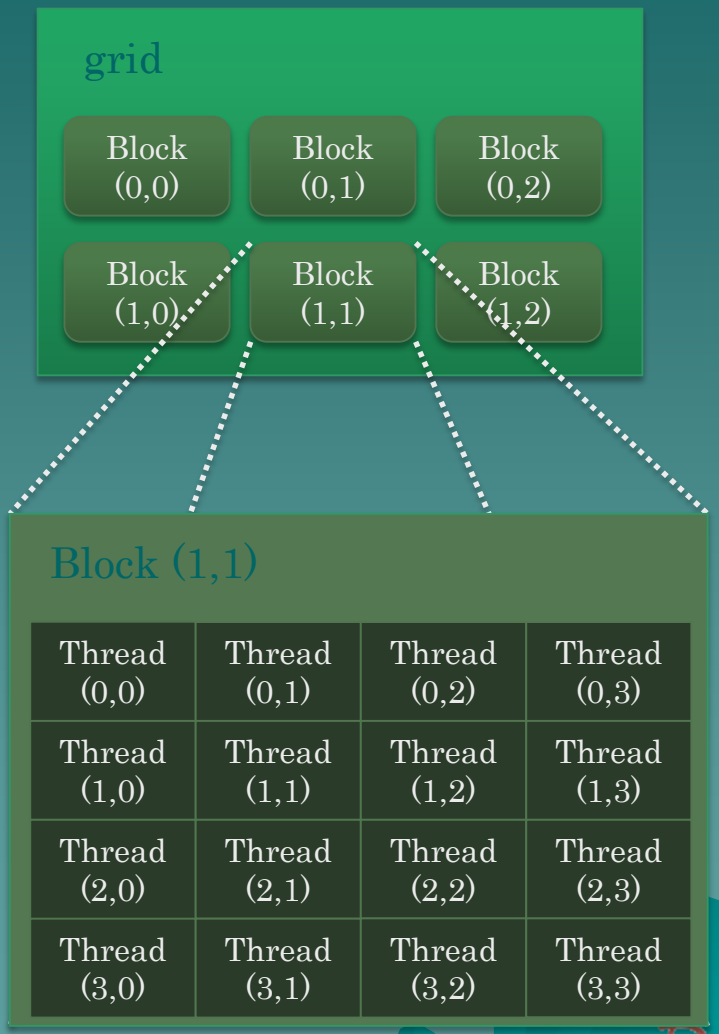
```
__global__void add(int *a, int *b, int *c)
{
  int tid=threadIdx.x+blockIdx.x * blockDim.x;
  while(tid <N) {
  c[tid] = a[tid]+b[tid];
  tid += gridDim.x * blockDim.x;
  }
 }
```

Programación Paralela en CUDA: threads, bloques, grids

Los bloques pueden ser unidimensionales (como en los ejemplos anteriores), bidimensionales (como una "matriz" de threads) o incluso tridimensionales (como un "cubo de threads").

Los bloque se organizan en "grids". Los "grids" también pueden ser unidimensionales, bidimensionales o tridimensionales.

Por ejemplo, si el bloque es bidimensional, el número de fila y columna del thread se obtienen como threadIdx.x, threadIdx.y



Programación Paralela en CUDA: Ejercicios

1) Escribe y ejecuta un programa en CUDA (N bloques de 1 thread) que lea un vector de N reales en la GPU y obtenga como resultado un vector de tamaño N-2 que contenga la media de tres elementos consecutivos del vector de entrada: ejemplo:

vector entrada=[0 1 2 3 4 5 6 7 8 9]

vector salida =[1 2 3 4 5 6 7 8]

2) Escribe y ejecuta un programa en CUDA que lea una matriz M*N en la gpu y devuelva un vector con N componentes que contenga las medias de las columnas de la matriz.
(Hacerlo con un thread por cada columna de la matriz, 1 bloque de N threads), Ejemplo, matriz $\begin{pmatrix} 0 & 1 & 2 & 3 \\ 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 5 \end{pmatrix}$ vector salida =[1,2,3,4]

Programación Paralela en CUDA: Ejercicios

Indicaciones para ejercicio 2

-Lo haremos adaptando el archivo mediamatriz_incompleto.cu (en poliformat), creando un kernel que haga el trabajo equivalente a la función mediasmatrizcpu.

-Hay que calcular medias ➔ divisiones ➔ conviene usar float o double.

-La entrada es una matriz M por N, la salida es un vector 1 por N.

-Lo haremos con un bloque de N threads

-Como en los ejemplos anteriores, hay que “quitar” un bucle, el kernel debe escribirse para que calcule la media de ***una*** columna

-Hay que usar el “tid” para referenciar correctamente los elementos de la matriz de entrada y del vector de salida.