

# Computación de Altas Prestaciones Seminario sobre GPGPUs



# Contenidos

Sesión 1 Teoría: Introducción a Computación en GPUs con CUDA.

Sesión 2: Compilación, conceptos básicos

Sesión 3: Programación de algoritmos “trivialmente paralelos”

Sesión 4: Uso de la memoria “Shared”.  
“Reducciones” en GPUs

Sesión 5: Optimización, temas avanzados, librerías

# 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 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: MULTI-INF-CAP

```
#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 línea `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;  
...
```

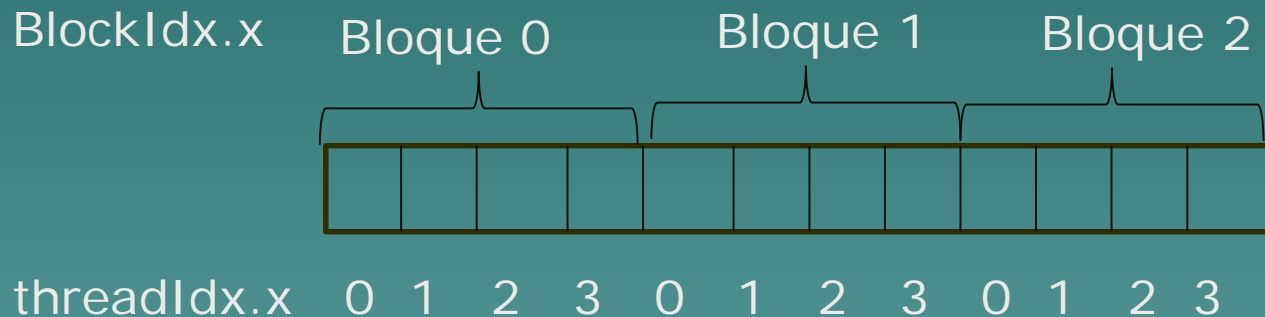
BlockDim es una variable que vale lo mismo para todos los bloques, da el número de threads en un bloque.

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:



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

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

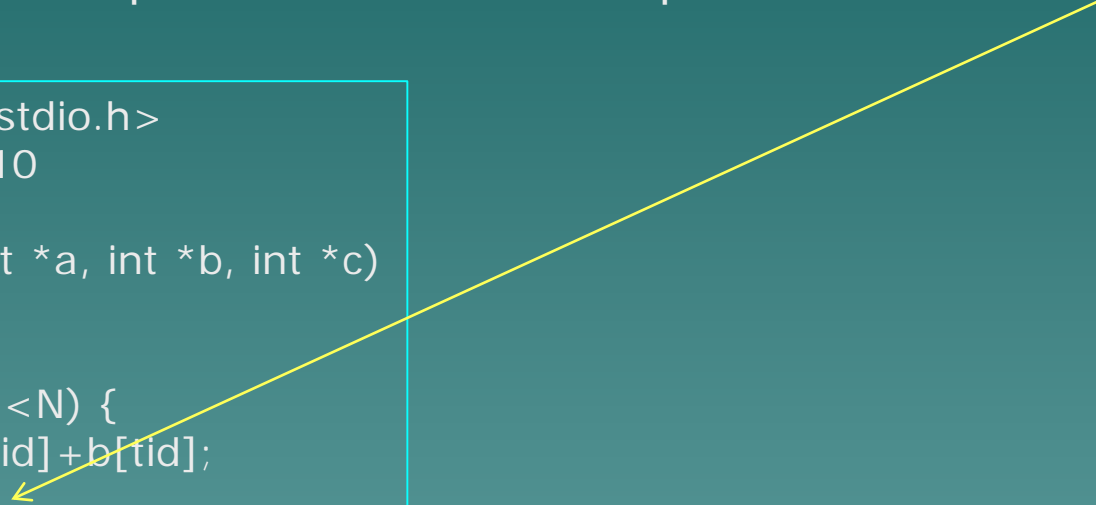
# Programación Paralela en CUDA: Suma de dos vectores; threads + bloques

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;
    }
}
```

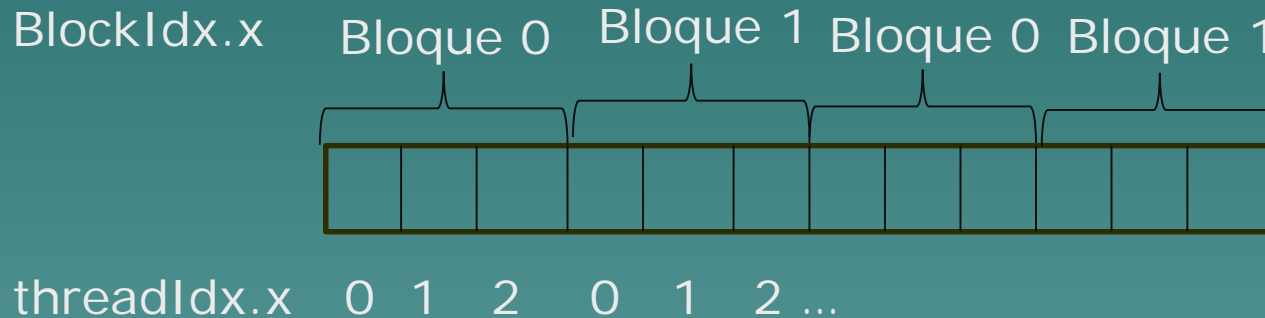


El número de bloques lanzados se encuentra en la variable gridDim.x

➔ El número total de threads lanzados es  $\text{gridDim.x} * \text{blockDim.x}$

# 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:

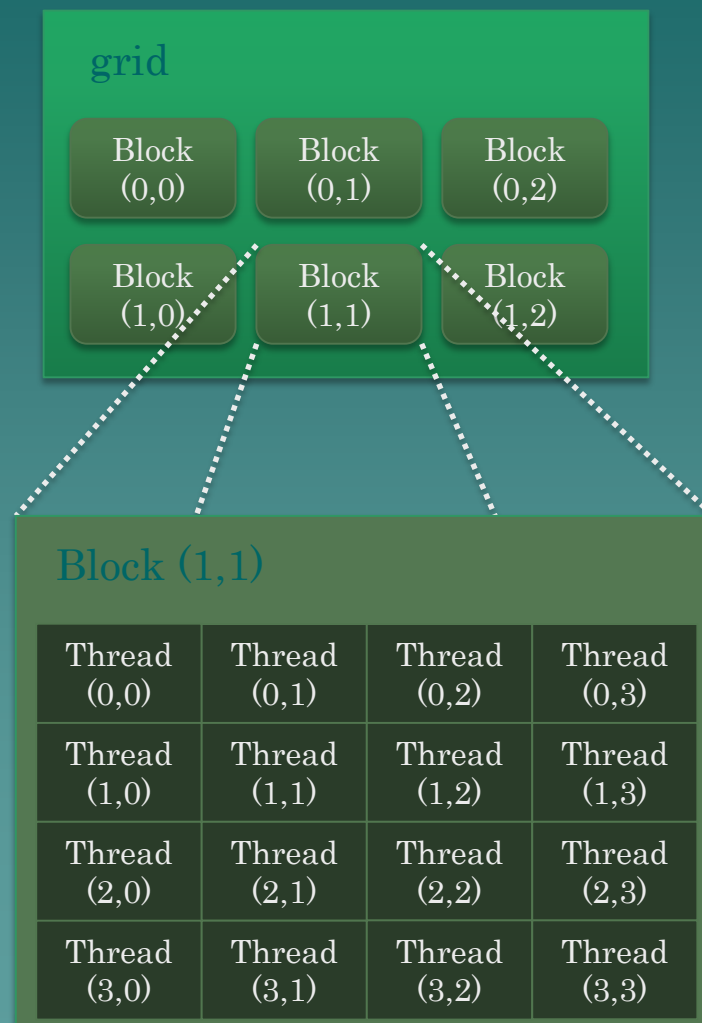
```
__global__ void add(int *a, int *b, int *c)
{
    int tid=threadIdx.x+blockIdx.x * blockDim.x;
    while(tid < N) {
        c[tid] = a[tid]+b[tid];
        tid += gridDim.x * blockDim.x;
    }
}
```

# 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 bloques 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.