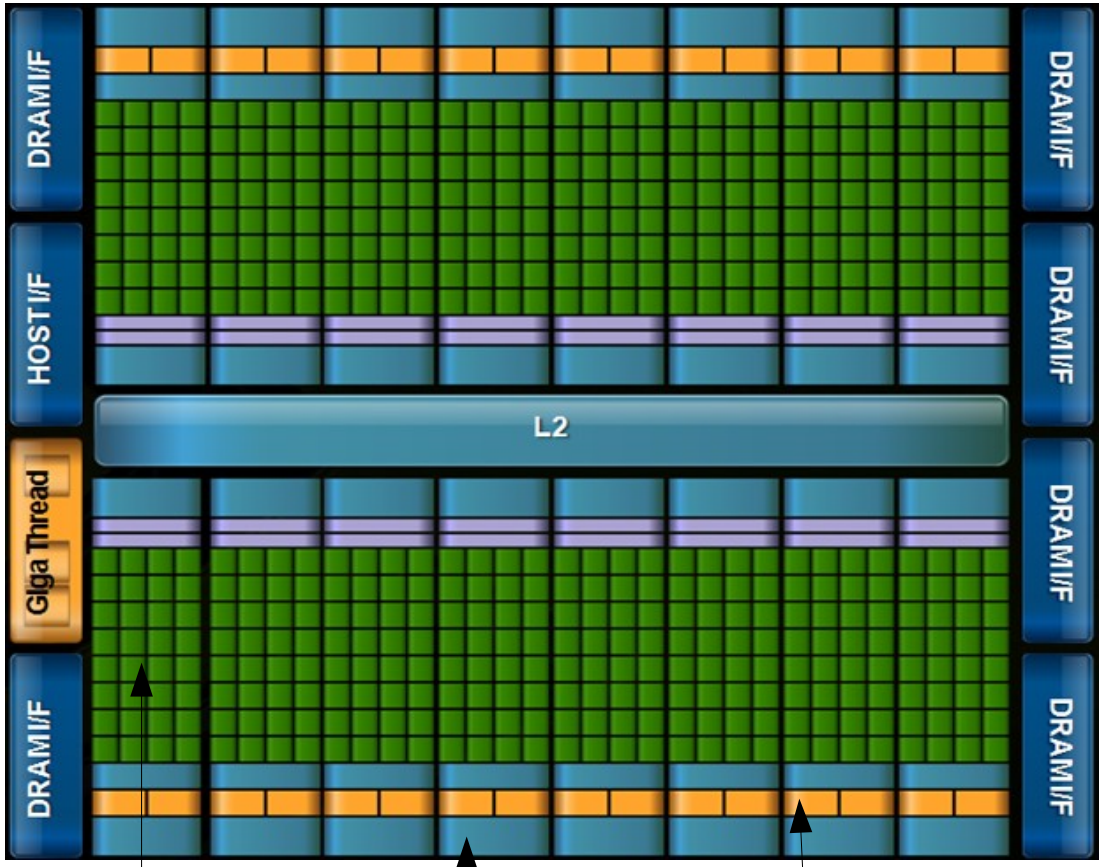
ICNPG 2023

Clase 2: CUDA C

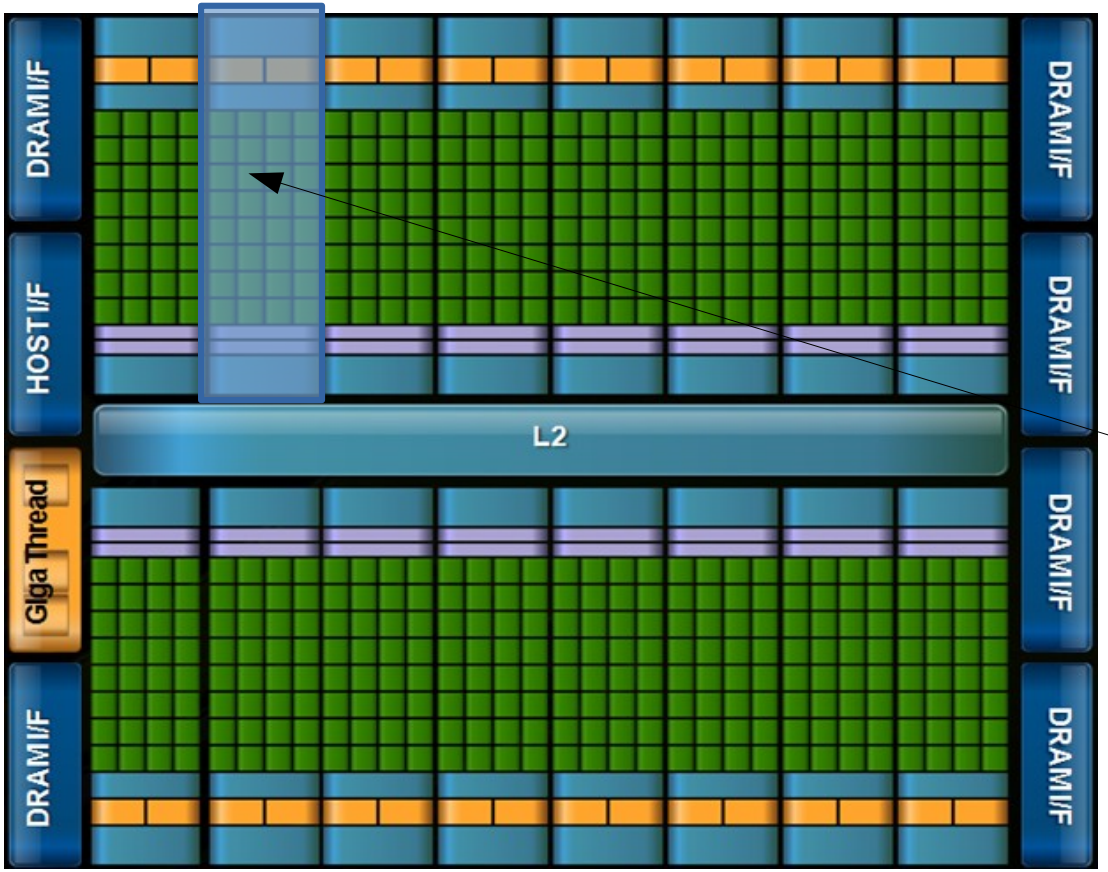
El Hardware

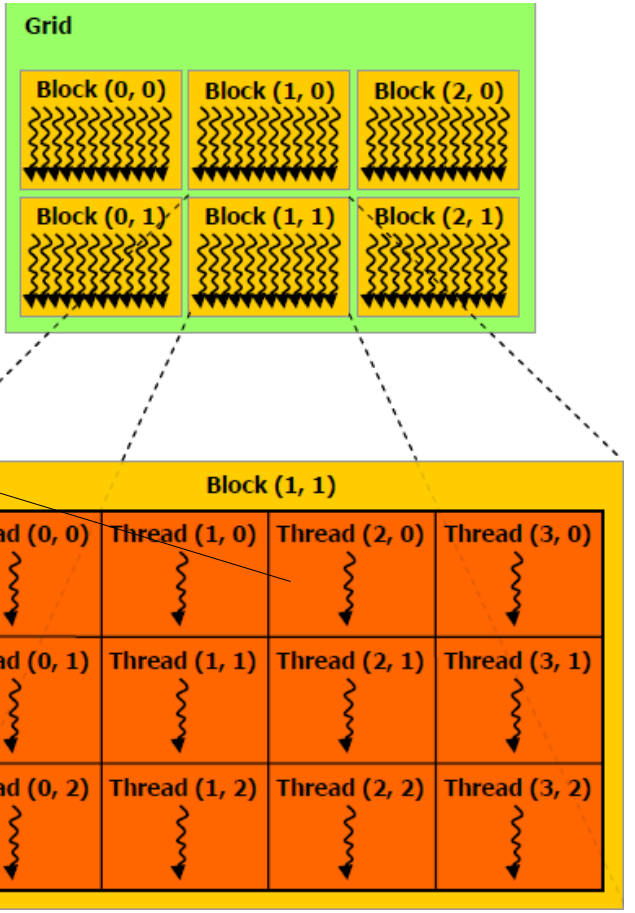


- Unidades de procesamiento.
- Lógica de control.
- Módulos de memoria.

Hasta ahora solo vimos cómo cargar datos sobre la memoria más grande L2. No es la más rápida. Si uno quisiera hacer más rápido debería cargar los datos desde L2 a L1

El Hardware

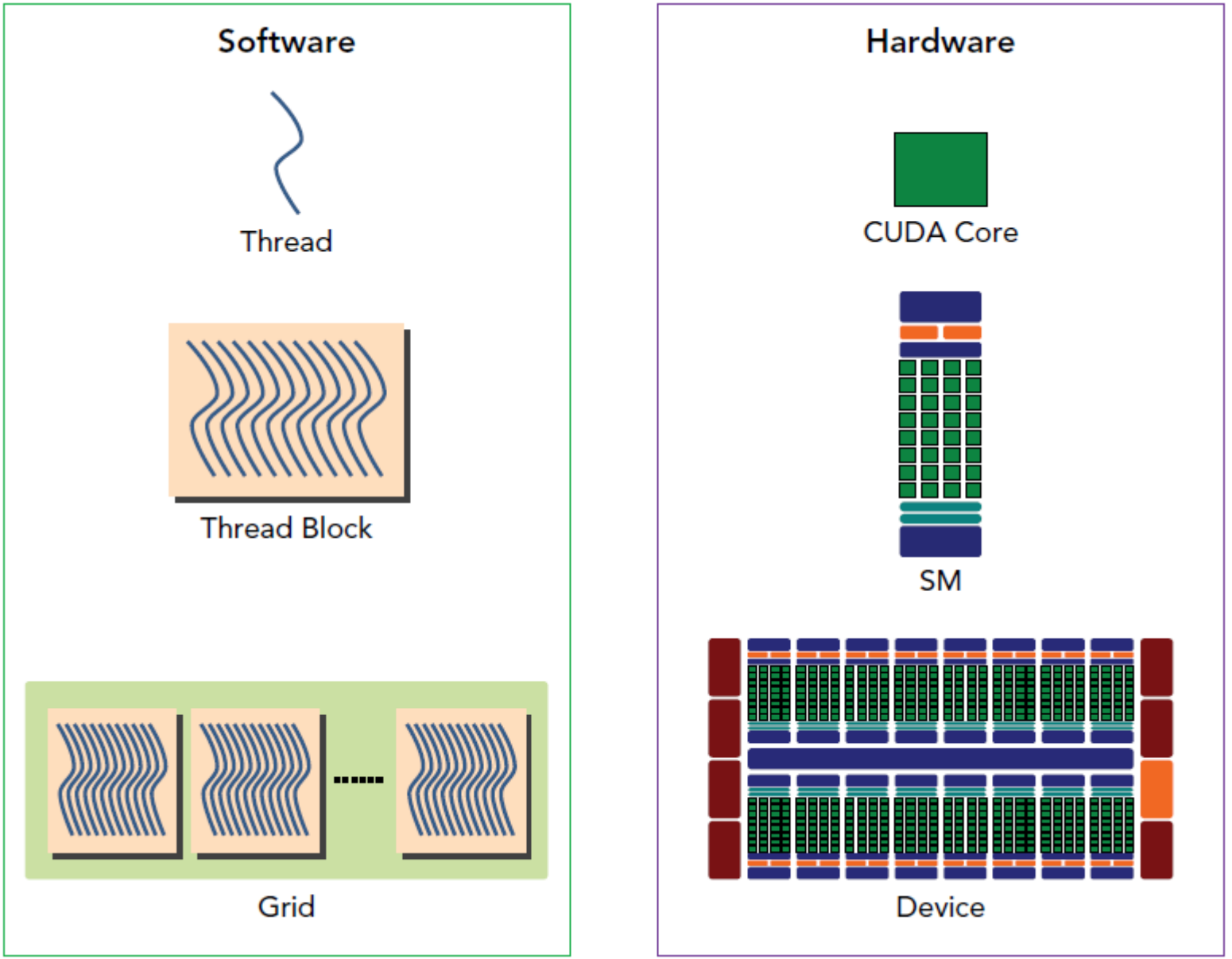




kernel<<<BLOCK_DIM, GRID_DIM>>>(...);

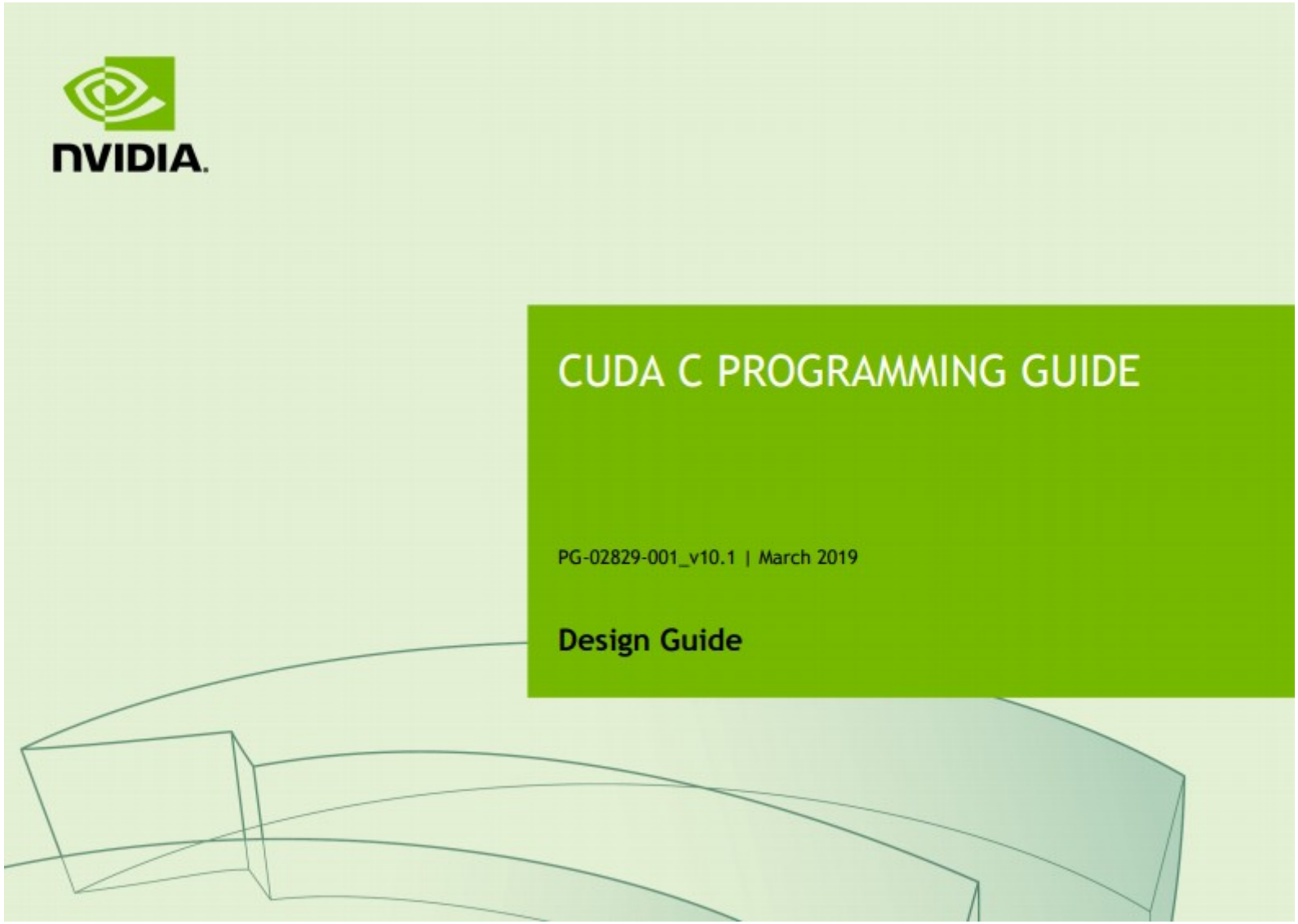
Los bloques se ejecutan en cualquier orden -----> Escalabilidad

- Una NVIDIA GPU consiste de un número de Multiprocesadores (SMs).
- Cada SM contiene un número de CUDA cores.
- Cada SM recibe un bloque y lo ejecuta.
- Cada núcleo ejecuta instrucciones de los threads en un modo SIMD.


Software
Thread
Thread Block
Grid
Hardware
CUDA Core
SM
Device


PCI Express 3.0 Host Interface
GigaThread Engine
Memory Controller
Memory Controller
Memory Controller
Memory Controller
Memory Controller
Memory Controller
Memory Controller
Memory Controller
GPC
Raster Engine
TPC
PolyMorph Engine
SM
Olvídense de este quilombo

https://docs.nvidia.com/cuda/cuda-c-programming-guide/



Paralelización, en general

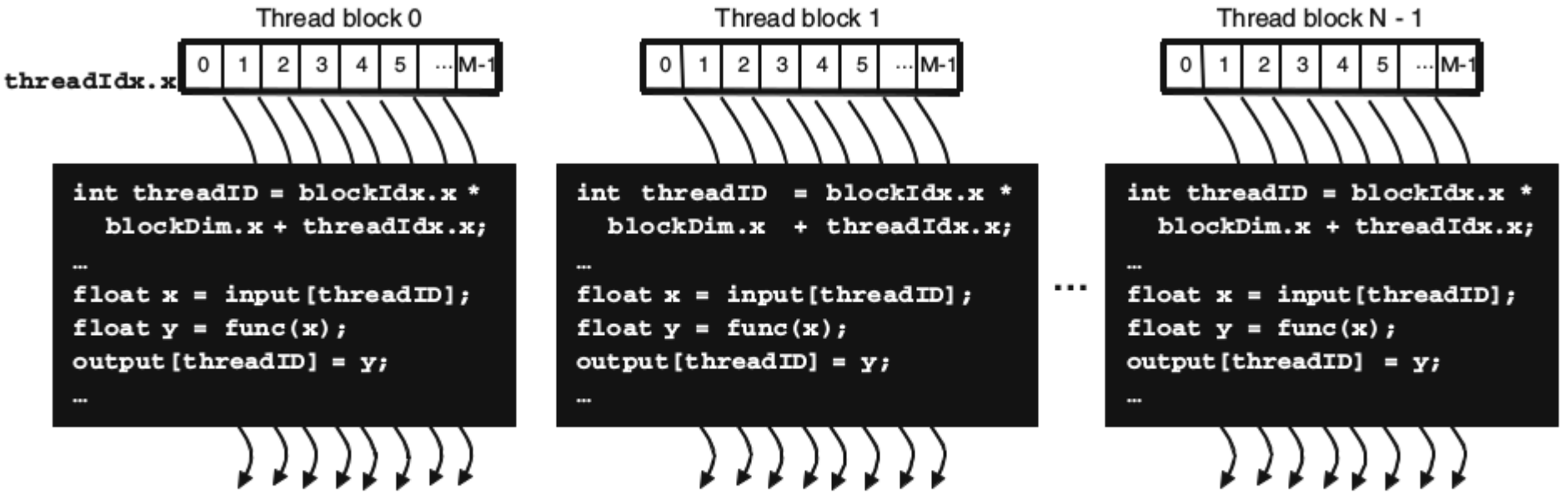


FIGURE 4.1

Overview of CUDA thread organization.

Threads del mismo bloque pueden cooperar: sincronizar, compartir “shared memory”, etc

Threads, Blocks, Grid

- There is a limit to the number of threads per block, since all threads of a block are expected to reside on the same processor core and must share the limited memory resources of that core. On current GPUs, **a thread block may contain up to 1024 threads.**
- A kernel can be executed by multiple equally-shaped thread blocks, so that the total number of threads is equal to **the number of threads per block times the number of blocks**.
- Blocks are organized into a *one-dimensional, two-dimensional, or three-dimensional grid* of thread blocks as illustrated by Figure 6. **The number of thread blocks in a grid is usually dictated by the size of the data being processed** or the number of processors in the system, which it can greatly exceed.

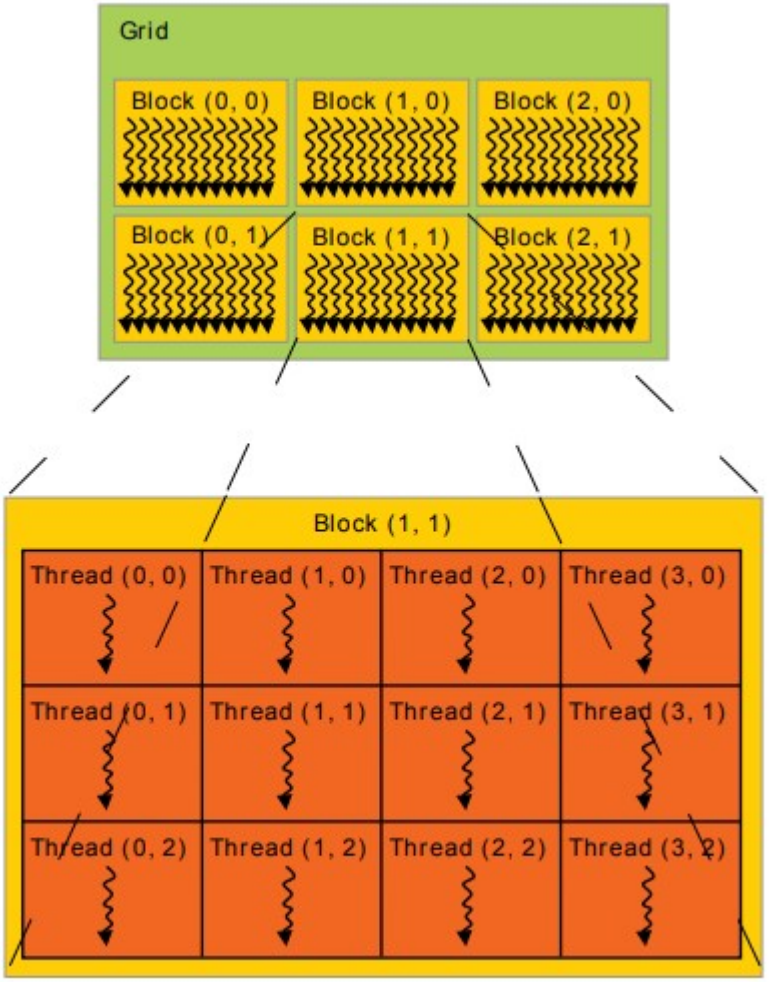


Figure 6 Grid of Thread Blocks

Threads y los blocks: elegí 1d, 2d o 3d para indexarlos

- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution
    - For hazard-free shared memory accesses
  - Efficiently sharing data through a low-latency shared memory
- Two threads from two different blocks cannot cooperate

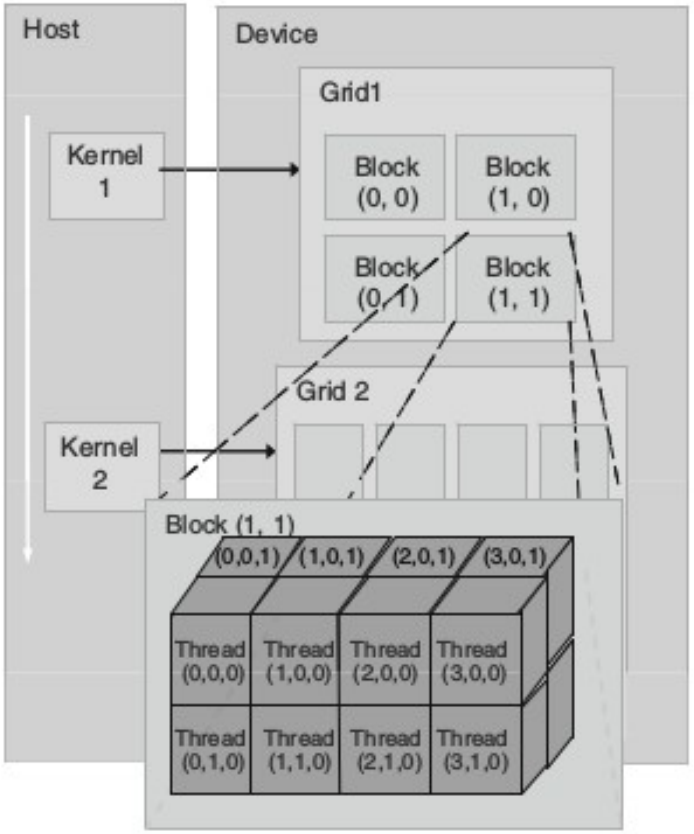


FIGURE 3.13

CUDA thread organization.

Dimensiones e Indexado multidimensional

- Indice del bloque **blockIdx** e índice de thread **threadIdx** en el bloque.
- Dimensión del bloque **blockDim** y de la grilla **gridDim**.
- ¿ Que dimensiones son permitidas ? → deviceQuery

/share/apps/icnpg/clases/Cuda_Basico/grillas_simple/

```
// kernel
__global__ void Quiensoy()
{
        printf("Soy el thread (%d,%d,%d) del bloque (%d,%d,%d) [blockDim=(%d,%d,%d),gridDim=(%d,%d,%d)]
\n",threadIdx.x,threadIdx.y,threadIdx.z,blockIdx.x,blockIdx.y,blockIdx.z,
blockDim.x,blockDim.y,blockDim.z,gridDim.x,gridDim.y,gridDim.z);

}
```

Indices de hilo en el bloque Indices de bloque en la grilla

Variables Reservadas para los kernels → threadIdx, blockIdx
blockDim, gridDim

Dimensiones de bloque Dimensiones de grilla

Indexado de hilos

- **$ less grillas.cu**
- **$ make submit** (cluster)
- **$ make run** (su maquina)

Indices de hilo en el bloque — Indices de bloque en la grilla

threadIdx, blockIdx
blockDim, gridDim

Dimensiones de bloque — Dimensiones de grilla

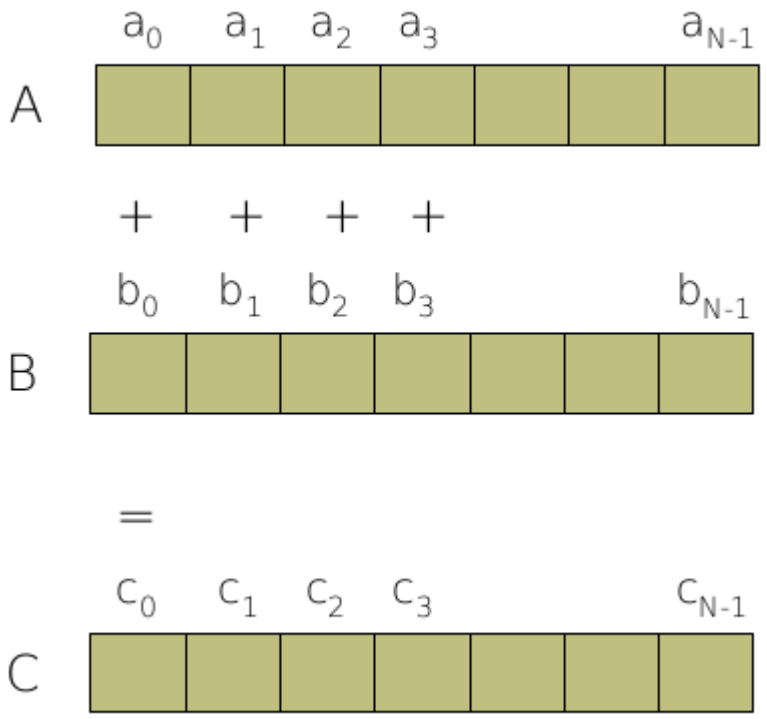
ejemplo 1: Quiensoy<<< 3, 2>>>
Soy el thread (0,0,0) del bloque (0,0,0) [blockDim=(2,1,1),gridDim=(3,1,1)]
Soy el thread (1,0,0) del bloque (0,0,0) [blockDim=(2,1,1),gridDim=(3,1,1)]
Soy el thread (0,0,0) del bloque (1,0,0) [blockDim=(2,1,1),gridDim=(3,1,1)]
Soy el thread (1,0,0) del bloque (1,0,0) [blockDim=(2,1,1),gridDim=(3,1,1)]
Soy el thread (0,0,0) del bloque (2,0,0) [blockDim=(2,1,1),gridDim=(3,1,1)]
Soy el thread (1,0,0) del bloque (2,0,0) [blockDim=(2,1,1),gridDim=(3,1,1)]

ejemplo 2: Quiensoy<<< dim3(2,2), dim3(2,1) >>>();
Soy el thread (0,0,0) del bloque (0,0,0) [blockDim=(2,1,1),gridDim=(2,2,1)]
Soy el thread (1,0,0) del bloque (0,0,0) [blockDim=(2,1,1),gridDim=(2,2,1)]
Soy el thread (0,0,0) del bloque (1,0,0) [blockDim=(2,1,1),gridDim=(2,2,1)]
Soy el thread (1,0,0) del bloque (1,0,0) [blockDim=(2,1,1),gridDim=(2,2,1)]
Soy el thread (0,0,0) del bloque (1,1,0) [blockDim=(2,1,1),gridDim=(2,2,1)]
Soy el thread (1,0,0) del bloque (1,1,0) [blockDim=(2,1,1),gridDim=(2,2,1)]
Soy el thread (0,0,0) del bloque (0,1,0) [blockDim=(2,1,1),gridDim=(2,2,1)]
Soy el thread (1,0,0) del bloque (0,1,0) [blockDim=(2,1,1),gridDim=(2,2,1)]

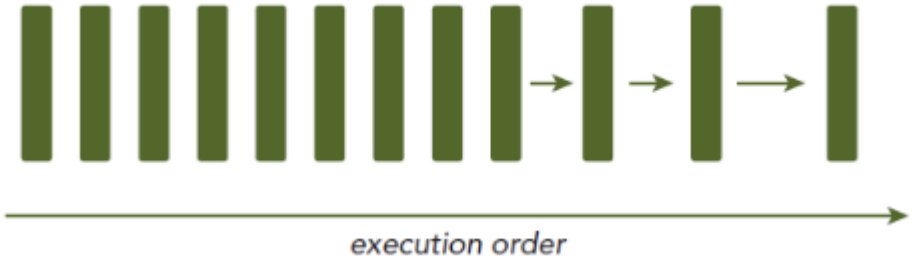
Grillas

- **dim3** es una variable tipo vector tridimensional.
- **Inicialización sobrecargada:**
  - dim3 n; n.x=7; nx=8; nz=1; // 1) declaración, 2) inicialización
  - dim3 n(7,8,1); //declaración+inicialización
  - dim3 n(7,8); //sobreentiende que n.z=1
- **Todos estos lanzamientos son equivalentes:**
  - dim3 nb(4,1,1); dim3 nt(3,1,1); Quiensoy<<<nb, nt>>>();
  - Quiensoy<<< dim3(4,1,1), dim3(3,1,1) >>>();
  - Quiensoy<<< 4,3 >>>();
  - Quiensoy<<< dim3(4,1), dim3(3,1) >>>();

Suma secuencial de dos arrays




```
void VectorAdd(int *a, int *b, int *c, int n)
{
        for(int i=0;i<n;i++)
        {
                c[i] = a[i] + b[i];
        }
}
```

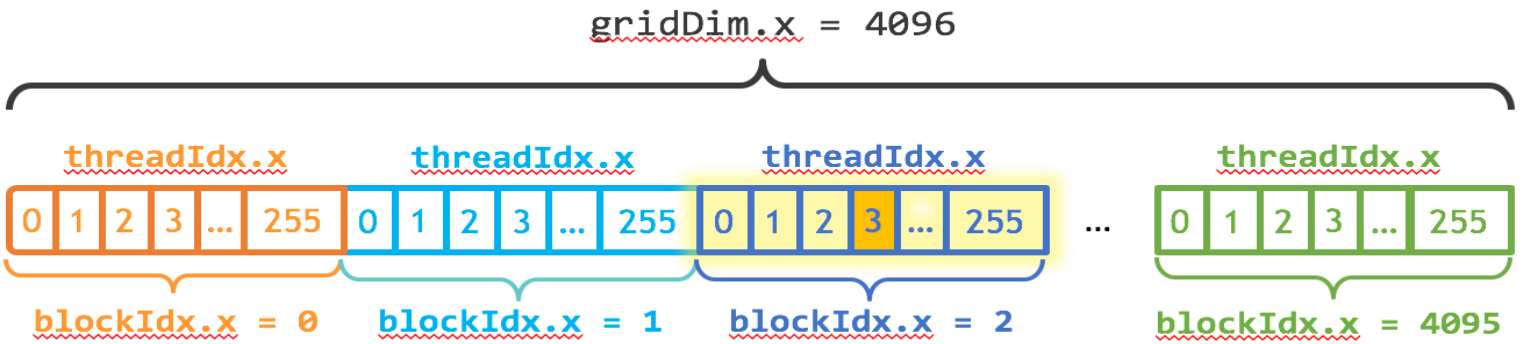


Ejercicio

- /share/apps/icnpg/clases/Cuda_Basico/suma_vectores/
  - 0_suma_vectores_cpu.cpp
  - 4_suma_vectores_gpu_general.cu

N	Tiempo CPU	Tiempo GPU
10		
100		
1000		
10000		
100000		
1000000		
10000000		

Suma paralela: “binary transform”



index = blockIdx.x * blockDim.x + threadIdx.x

index = (2) * (256) + (3) = 515

```
// grilla de threads suficientemente grande...
dim3 nThreads(256);
dim3 nBlocks((N + nThreads.x - 1) / nThreads.x);
// suma paralela en el device
VectorAdd<<< nBlocks, nThreads >>>(d_a, d_b, d_c, N);
```

Lanzamiento del Kernel

```
// kernel
__global__ void VectorAdd(int *a, int *b, int *c, int n)
{
        // indice de thread mapeado a indice de array
        int i = blockIdx.x * blockDim.x + threadIdx.x;
        if (i < n)
                c[i] = a[i] + b[i];
}
```

Función Kernel

Funciones “Kernel”

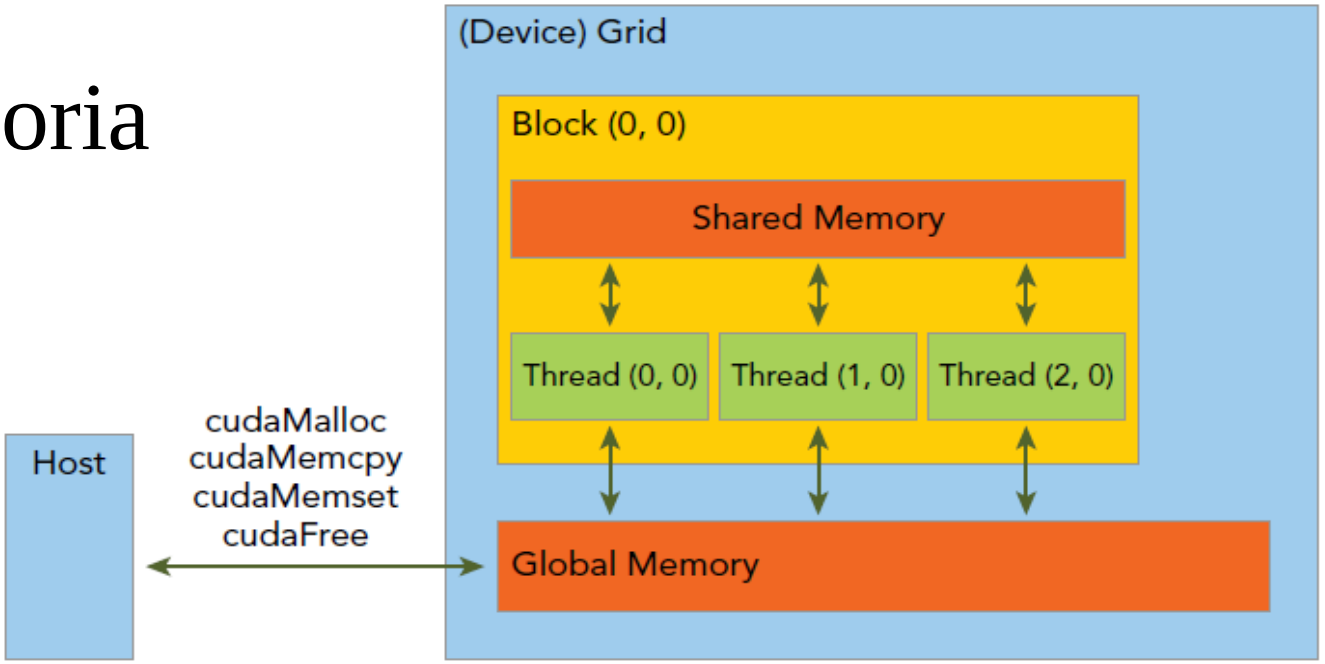
CUDA KERNELS ARE FUNCTIONS WITH RESTRICTIONS

The following restrictions apply for all kernels:

- Access to device memory only
- Must have `void` return type
- No support for a variable number of arguments
- No support for static variables
- No support for function pointers
- Exhibit an asynchronous behavior

```
// kernel
__global__ void VectorAdd(int *a, int *b, int *c, int n)
{
        // indice de thread mapeado a indice de array
        int i = blockIdx.x * blockDim.x + threadIdx.x;
        if (i < n)
                c[i] = a[i] + b[i];
}
```

Manejo de memoria




```
// alocacion memoria de device
cudaMalloc( &d_a, N*sizeof(int));
cudaMalloc( &d_b, N*sizeof(int));
cudaMalloc( &d_c, N*sizeof(int));
```

```
// copia de host a device
cudaMemcpy( d_a, a, N*sizeof(int), cudaMemcpyHostToDevice );
cudaMemcpy( d_b, b, N*sizeof(int), cudaMemcpyHostToDevice );
cudaMemcpy( d_c, c, N*sizeof(int), cudaMemcpyHostToDevice );
```

Lanzamiento Kernel

```
// copia (solo del resultado) del device a host
cudaMemcpy( c, d_c, N*sizeof(int), cudaMemcpyDeviceToHost );
```

```
// liberacion memoria de device
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);
```

Muchas formas de repartir datos

- Un solo hilo
- M<N hilos en un solo bloque.
- M<N hilos en varios bloques.
- M=N hilos en un solo bloque.
- M=N bloques, un hilo por bloque.
- (N+B-1)/B bloques B hilos por bloque.

- ¿Que pasa si no entran los arrays en la memoria de la GPU?
- Se puede conocer la GPU en el runtime? → **devicequery**

Device query

- A veces es útil que el programa, al empezar a correr conozca las propiedades y limitaciones de la placa (memoria, compute capability, etc), y decidir sobre la marcha como realizar el trabajo.
- El cuda runtime provee funciones y estructuras para entrevistar a la gpu...

```
#include <stdio.h>

int main(int argc, char **argv)
{
        cudaDeviceProp deviceProp;

        int deviceCount = 0;
        cudaError_t error_id = cudaGetDeviceCount(&deviceCount);


        printf("En este nodo hay %d placas\n\n",deviceCount);
        for(int dev=0;dev<deviceCount;dev++){
                cudaSetDevice(dev);
                cudaGetDeviceProperties(&deviceProp, dev);
                printf("Hola!, yo soy [Device %d: \"%s\"], tu acelerador grafico personal\n", dev, deviceProp.name);
        }

   int dev; cudaGetDevice(&dev);
        printf("\nle asigno la device %d, que esta desocupada\n", dev);

        return 0;
}
```

Ejemplo completo
https://github.com/NVIDIA/cuda-samples/tree/master/Samples/1_Utilities/deviceQuery

Número de bloques

- El número de hilos por bloque M no puede superar 1024 (recursos limitados del SM).
- En número de hilos por bloque M conviene que sea múltiplo de 32 (SIMD warp).
- Para tener al menos N hilos podemos fijar el número de bloques Nb:

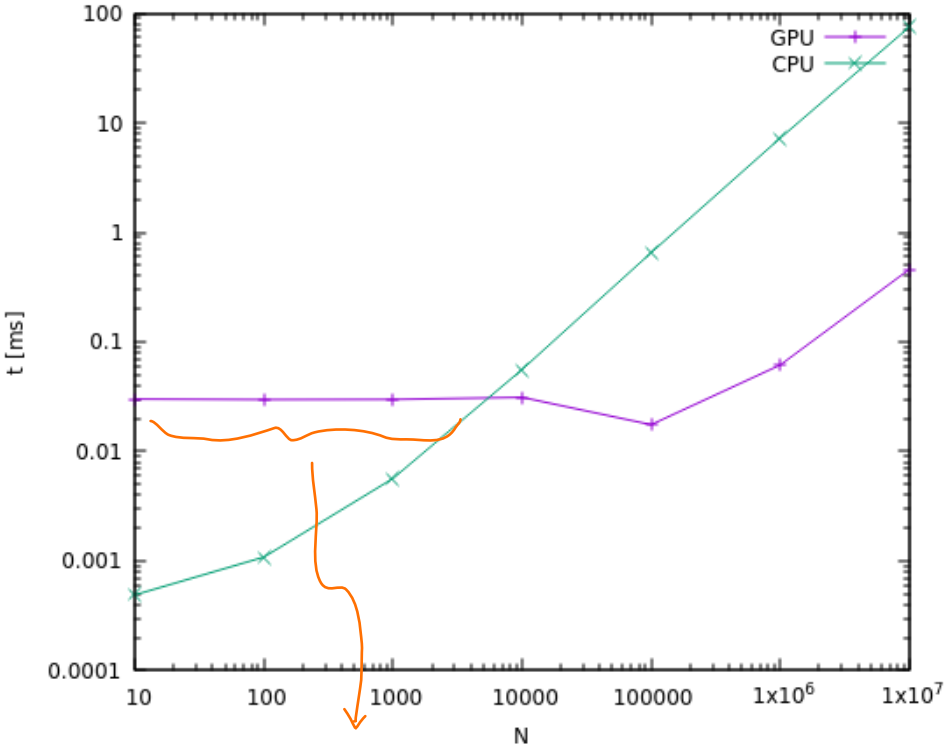
$$N_b = int[(N + M - 1)/M]$$

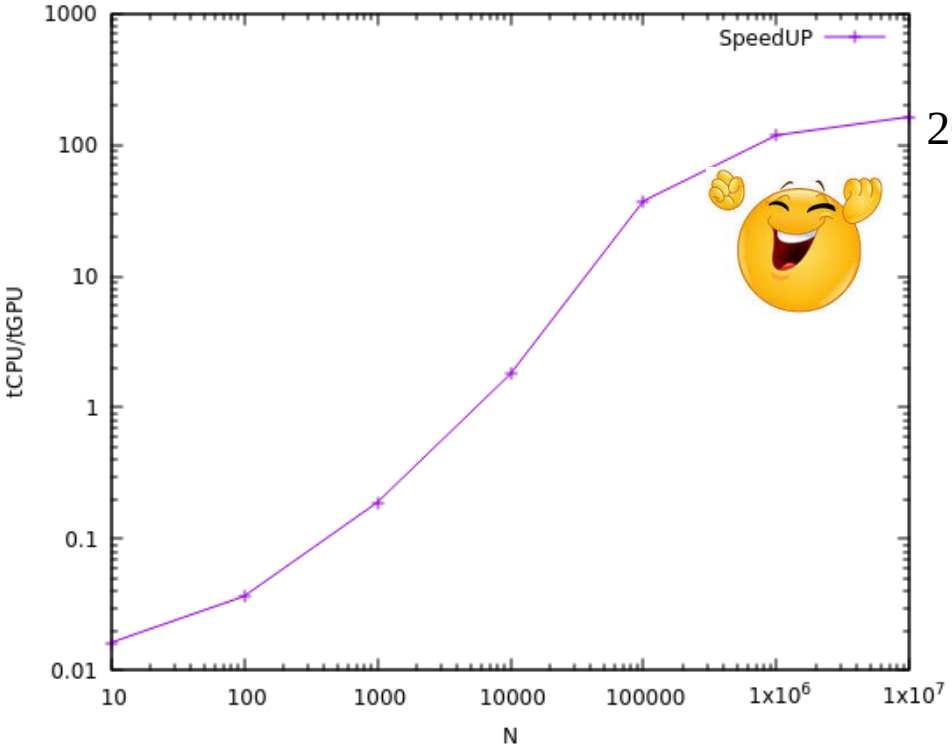
$$0 \leq N_b M - N < M$$

¿Que pasa si N es múltiplo de M?

Aceleración

Estos tiempos no están bien calculados porque no se están considerando los tiempo de copia de los arrays





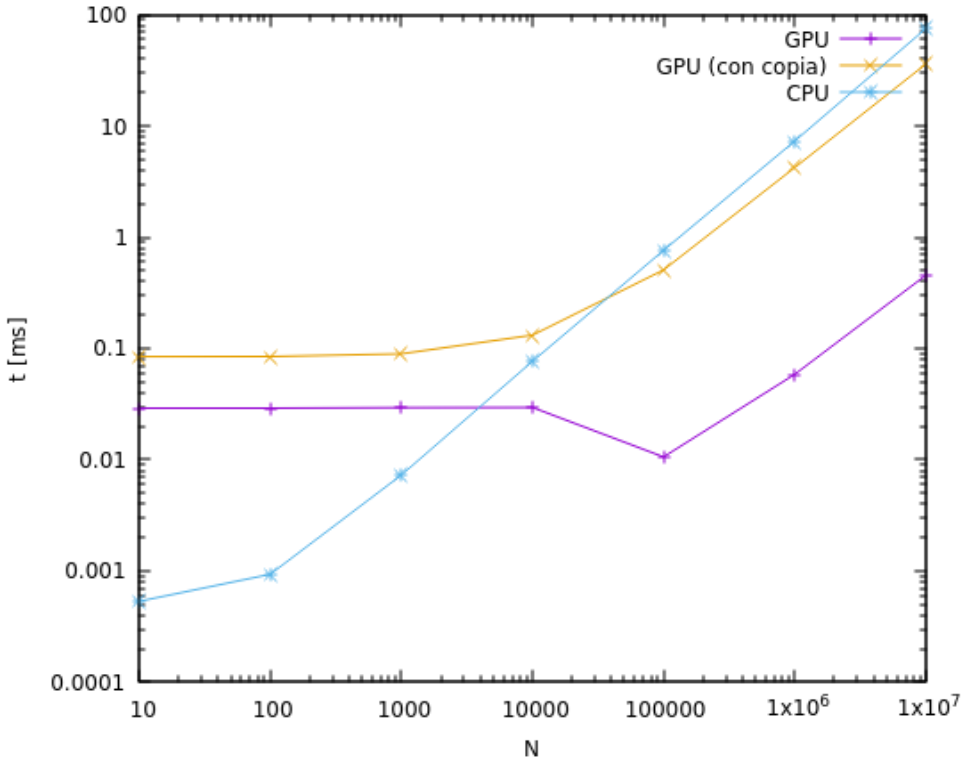
200x

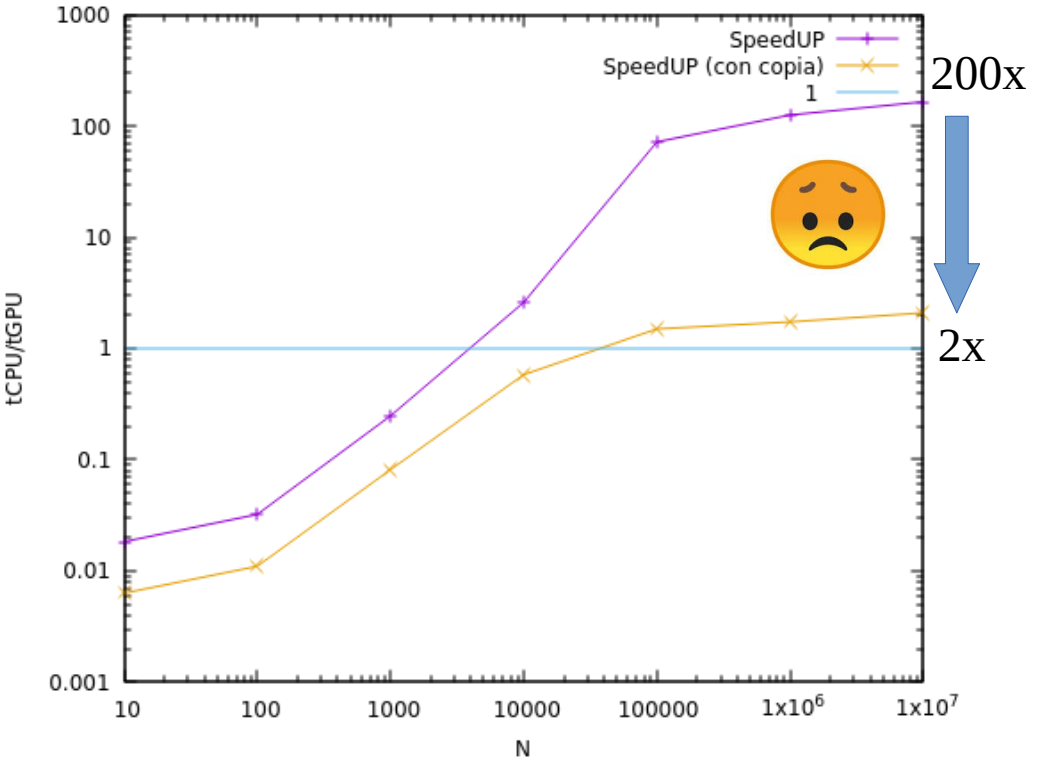
Inicialmente la GPU tarda más porque es el costo de crear el kernel y demás inicializaciones

Aceleración:

Hay que minimizar las copias todo lo posible

Ahora contando copias H2D y D2H





CUDA PROGRAM STRUCTURE

A typical CUDA program structure consists of five main steps:

1. Allocate GPU memories.
2. Copy data from CPU memory to GPU memory.
3. Invoke the CUDA kernel to perform program-specific computation.
4. Copy data back from GPU memory to CPU memory.
5. Destroy GPU memories.

THREE RULES OF GPGPU PROGRAMMING

Observation has shown that there are three general rules to creating high-performance GPGPU programs:

1. Get the data on the GPGPU and keep it there.
2. Give the GPGPU enough work to do.
3. Focus on data reuse within the GPGPU to avoid memory bandwidth limitations.

para amortizar el gran costo de las copias

These rules make sense, given the bandwidth and latency limitations of the PCIe bus and GPGPU memory system as discussed in the following subsections.

¿Preguntas?

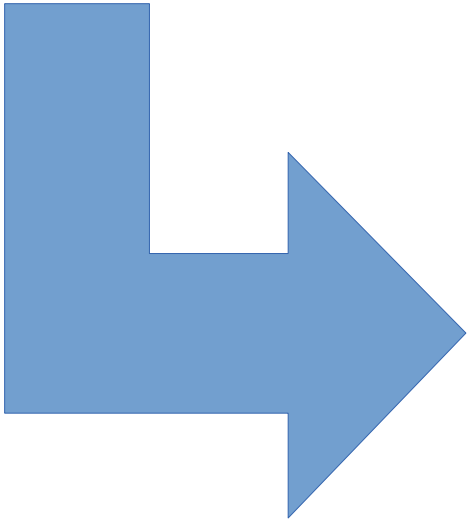
Otro ejemplo: “tabulate”

/share/apps/icnpg/clases/Cuda_Basico/llenar_vector/

- Llenar un vector en la GPU

```
// grilla de threads suficientemente grande...
dim3 nThreads(256);
dim3 nBlocks((N + nThreads.x - 1) / nThreads.x);
```

```
// llena en el device
Llenar<<< nBlocks, nThreads >>>(d_a);
```



```
// funcion para rellenar
__host__ __device__ float Mifuncion(int i)
{
        return tanh(cos(exp(-i*0.01)+0.02));
}


// kernel para tabular
__global__ void Llenar(float *a)
{
        // indice de thread mapeado a indice de array
        int i = blockIdx.x * blockDim.x + threadIdx.x;
        a[i]=Mifuncion(i);
}
```

__device__

__global__

Otro ejemplo: “tabulate”

/share/apps/icnpg/clases/Cuda_Basico/llenar_vector/

- Kernel y una función de device

```
// funcion para rellenar
__host__ __device__ float Mifuncion(int i)
{
        return tanh(cos(exp(-i*0.01)+0.02));
}


// kernel para tabular
__global__ void Llenar(float *a)
{
        // indice de thread mapeado a indice de array
        int i = blockIdx.x * blockDim.x + threadIdx.x;
        a[i]=Mifuncion(i);
}
```

__device__

__global__

QUALIFIERS	EXECUTION	CALLABLE	NOTES
`__global__`	Executed on the device	Callable from the host Callable from the device for devices of compute capability 3	Must have a `void` return type
`__device__`	Executed on the device	Callable from the device only	
`__host__`	Executed on the host	Callable from the host only	Can be omitted

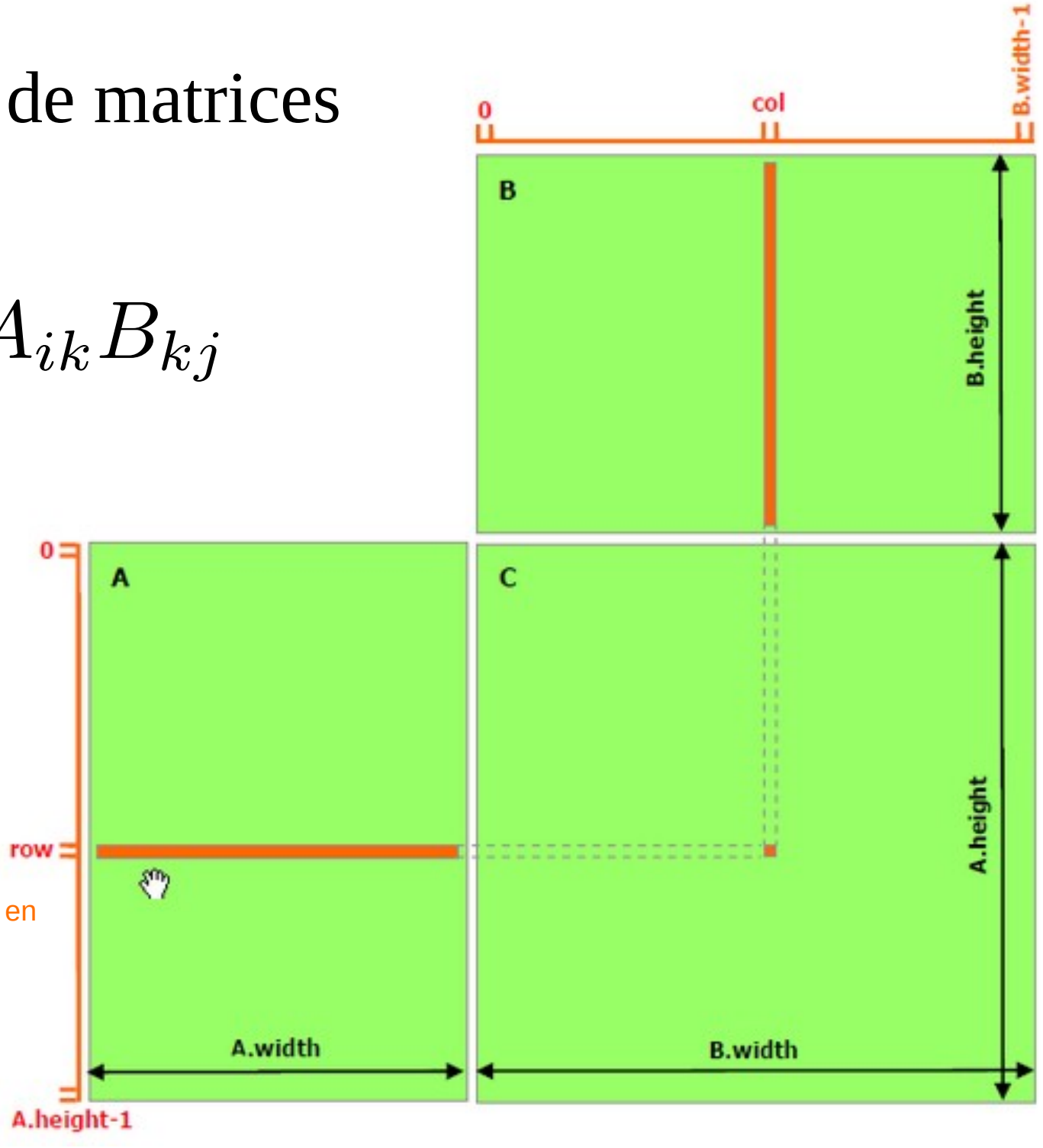
Multiplicación de matrices

$O(N^3)$

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

¿Como paralelizo este calculo?

A diferencia del caso de suma vectorial, en cada producto se reusan datos



Multiplicación de matrices

$O(N^3)$

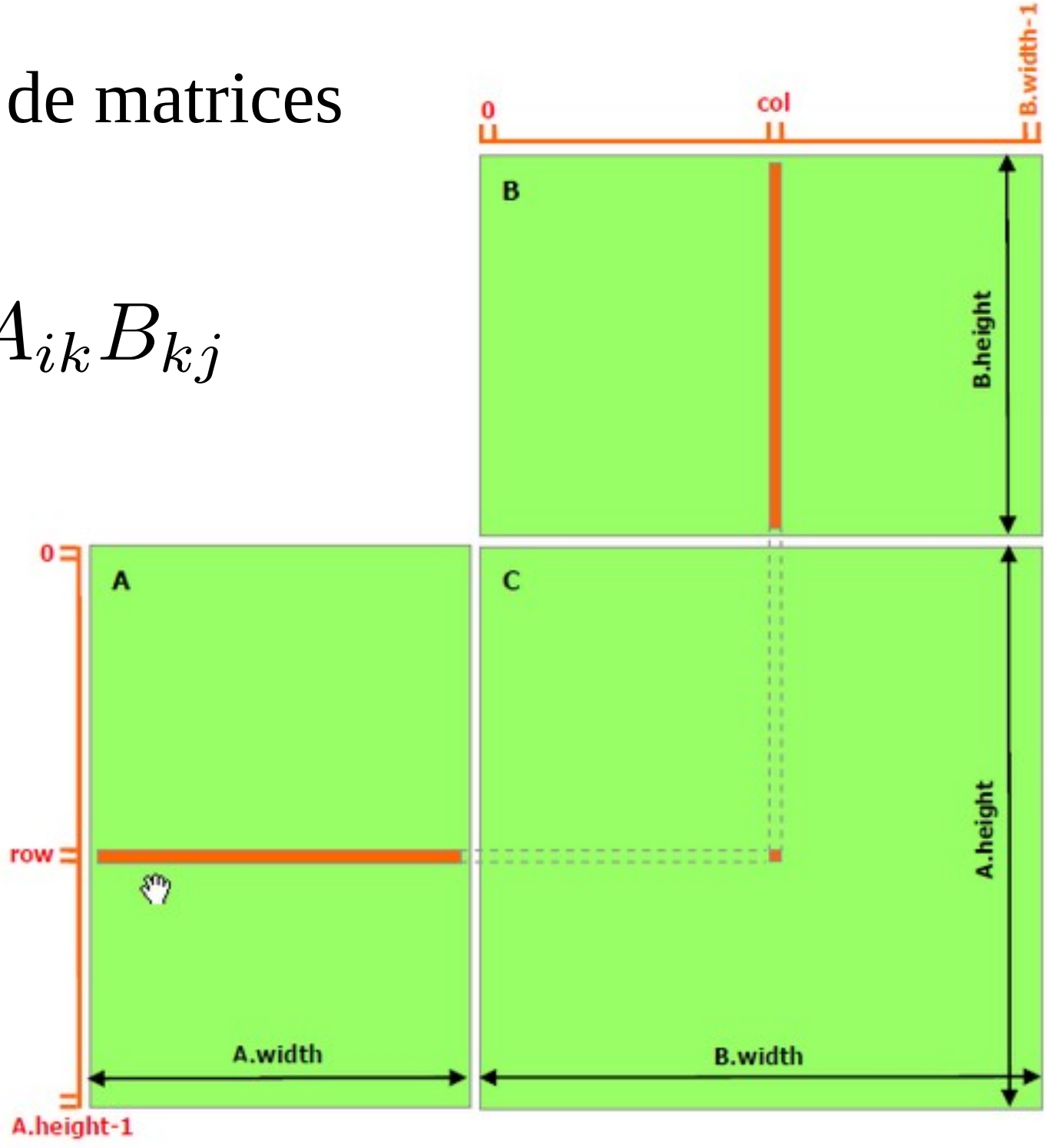
$$C_{ij} = \sum_k A_{ik} B_{kj}$$

Un hilo ← *Paralelización*

threadIdx
blockIdx

Varias formas de mapear a:

ij



Tarea 1

- Completar el programa de Multiplicacion de Matrices

```
// Matrix multiplication kernel called by MatMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
    // Each thread computes one element of C
    // by accumulating results into Cvalue
    float Cvalue = 0;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    for (int e = 0; e < A.width; ++e)
        Cvalue += A.elements[row * A.width + e]
                * B.elements[e * B.width + col];
    C.elements[row * C.width + col] = Cvalue;
}
```

Tarea 2

- Dada una señal discreta x[n] y un flitro h[n], la convolución $y = x * h$ se define como:

$$y[n] = [x * h][n] = \sum_k x[k+n]h[k]$$

  - Considerar un array x de números reales de tamaño N que representa la señal en un dominio discreto y un array h de tamaño M que describe el filtro h, M << N.

- ¿Como paralelizamos esto?

```
/* convolucion en la cpu: requiere dos loops */
void conv_sec(FLOAT* input, FLOAT* output, FLOAT * filter)
{
        FLOAT temp;
        for(int j=0;j<N;j++){
                temp=0.0;
                for(int i=0;i<M;i++){
                        temp += filter[i]*input[i+j];
                }
                output[j] = temp;
        }
}
```