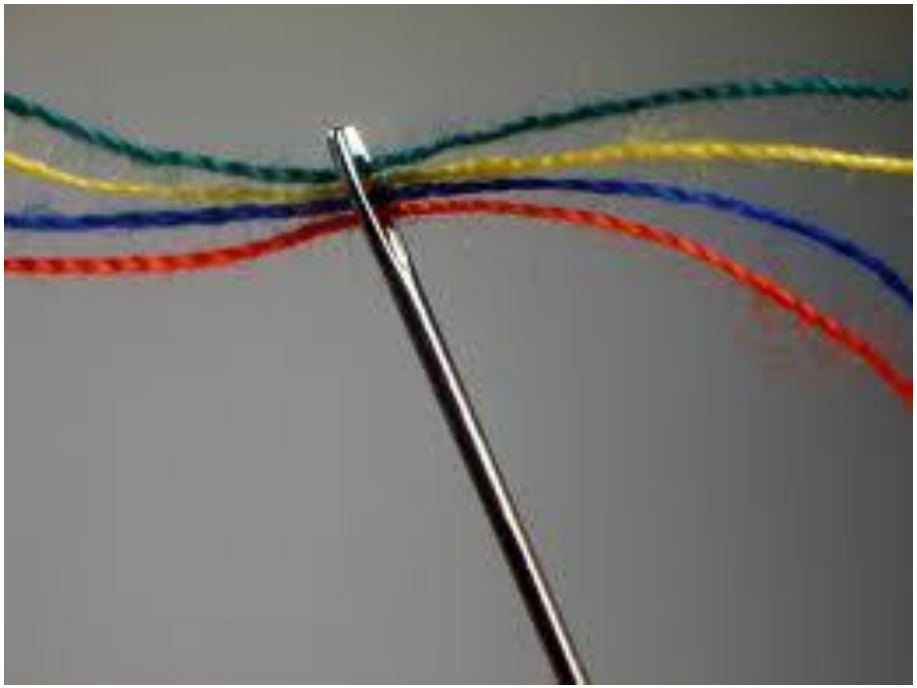
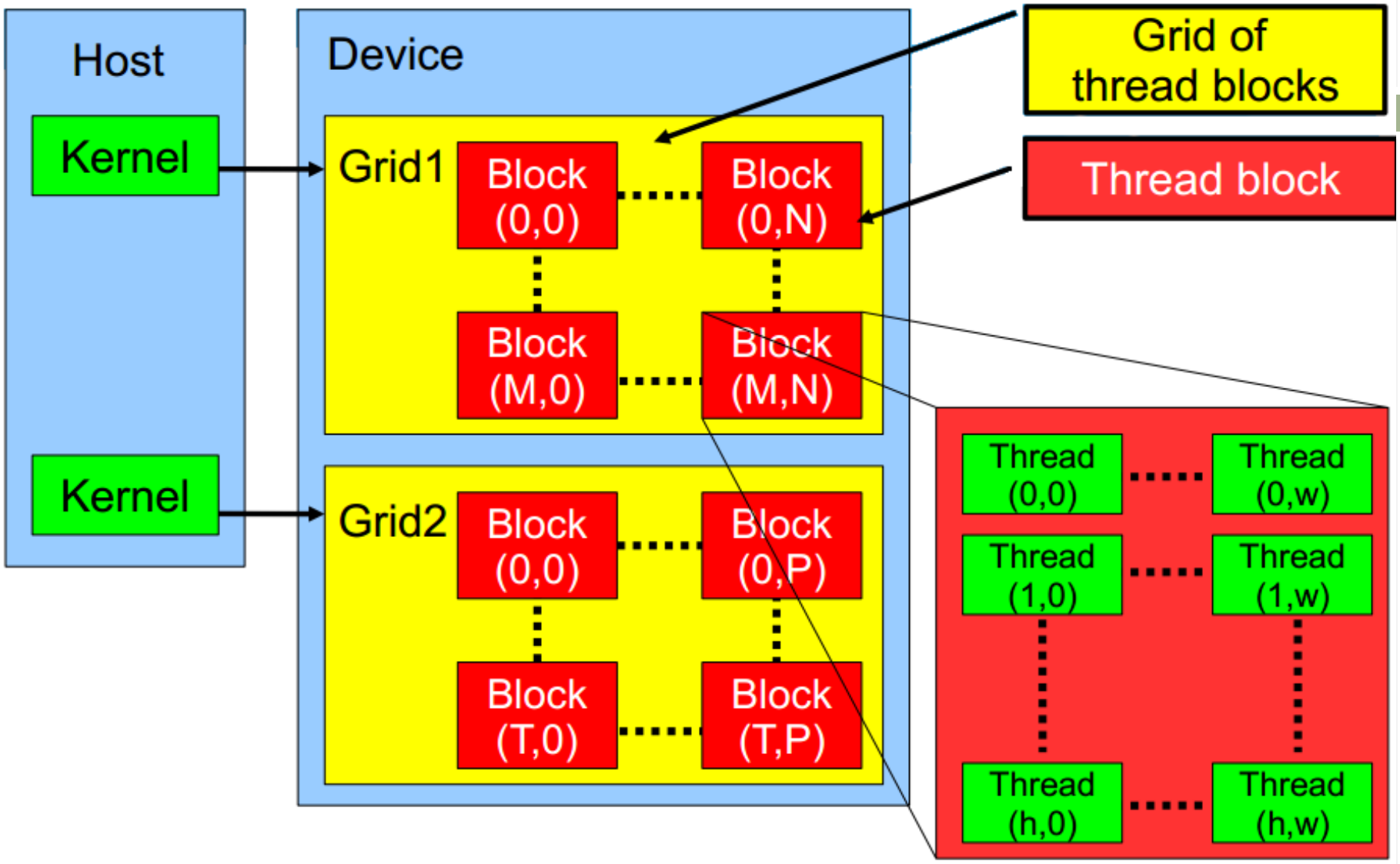


COOPERACIÓN DE THREADS

Netz Romero

Conceptos

- **Thread**.- código concurrente y asociado a un estado de ejecución en el device.
- **Thread block**.- es un grupo de threads que son ejecutados juntos y pueden compartir memoria.
- **Warp**.- es un grupo de threads dentro de un thread block que son ejecutados físicamente en paralelo.
- **Grid**.- es un grupo de thread blocks que se ejecutan lógicamente en un kernel de forma paralela en una GPU.


Host
Kernel
Kernel
Device
Grid1
Block (0,0)
Block (0,N)
Block (M,0)
Block (M,N)
Grid2
Block (0,0)
Block (0,P)
Block (T,0)
Block (T,P)
Grid of thread blocks
Thread block
Thread (0,0)
Thread (0,w)
Thread (1,0)
Thread (1,w)
Thread (h,0)
Thread (h,w)

Kernel

- El kernel corre en el device.
- La sintaxis es:

```
__global__ void correKernel(...) { ... }
```

- Se ejecuta como:

```
correKernel <<<nBlocks, nThreads>>>(...);
```

donde:

`nBlocks` es el número de bloques en un grid

`nThreads` es el número de hilos en un bloque

- Manejando los nBlocks

```
correKernel <<<N, 1>>>(...);
```

Para acceder a los números de bloques

```
blockIdx.x
```

- Manejando los nThreads

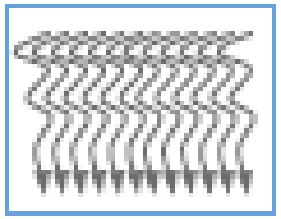
```
correKernel <<<1, N>>>(...);
```

Para acceder a los números de hilos:

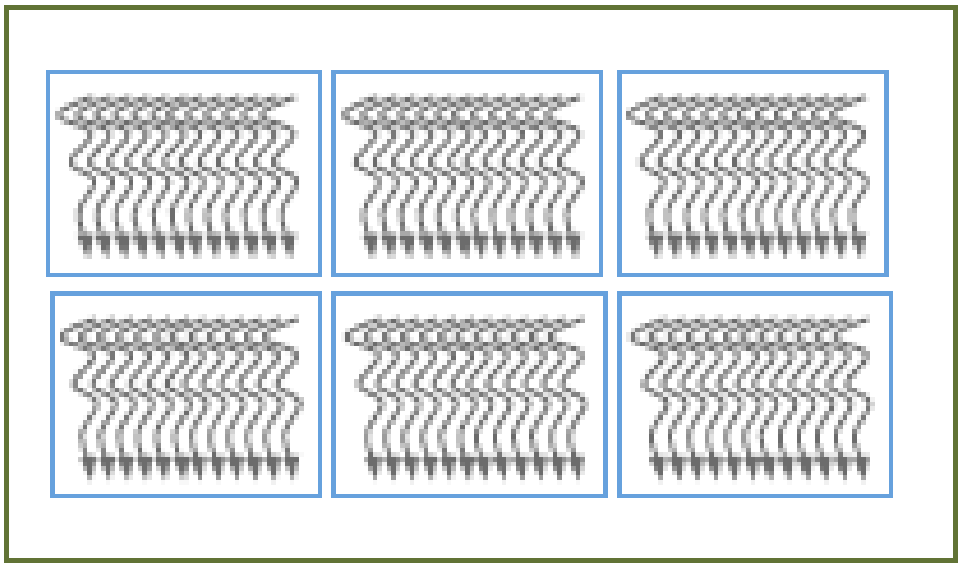
```
threadIdx.x
```

Thread

Block



Grid



Actividad

- Modificar el programa anterior sobre la suma de vectores en cuda, para que funcione por nThreads.

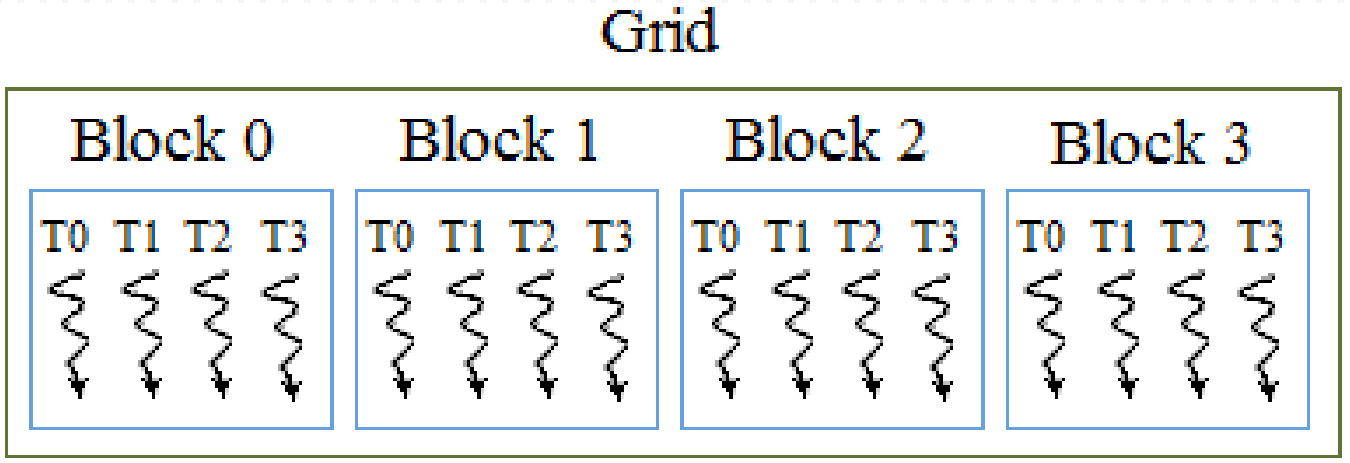
Problema

- Si quiero utilizar un arreglo de tamaño 3000 y la capacidad de cada bloque es de 1024 threads. ¿Cuántos bloques por grid y threads por bloque necesito, para hacer operaciones con el tamaño especificado?

Ejemplo

- Supongamos que tenemos un vector de tamaño 16, entonces con cuatro bloques en un grid y cuatro bloques en un hilo resolvemos el problema.

  Se lanzan los cuatro bloques al mismo tiempo y cada bloque lanza los cuatro hilos al mismo tiempo.



- Para que el vector pueda hacer las 16 operaciones al mismo tiempo, el índice tiene que ser:

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

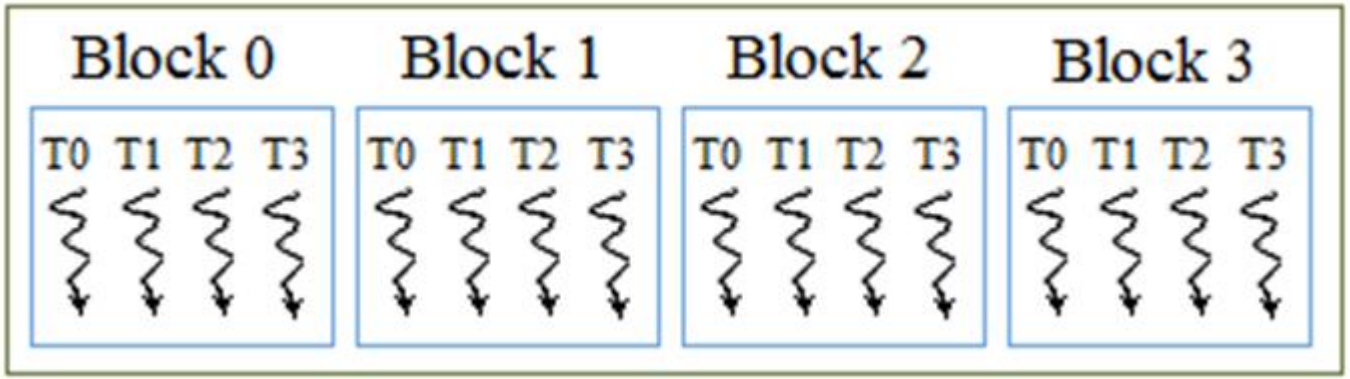
Recordemos que

`threadIdx.x` va desde 0 a 3 hilos

`blockIdx.x` va desde 0 a 3 bloques

`blockDim.x` su valor es 4 (número de bloques)

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

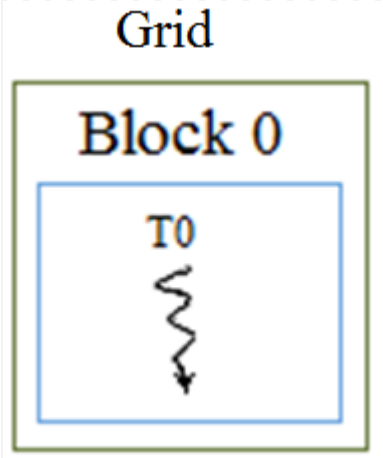


Bloque 0	Bloque 1	Bloque 2	Bloque 3
tid=0+0*4=0	tid=0+1*4=4	tid=0+2*4=8	tid=0+3*4=12
tid=1+0*4=1	tid=1+1*4=5	tid=1+2*4=9	tid=1+3*4=13
tid=2+0*4=2	tid=2+1*4=6	tid=2+2*4=10	tid=2+3*4=14
tid=3+0*4=3	tid=3+1*4=7	tid=3+2*4=11	tid=3+3*4=15

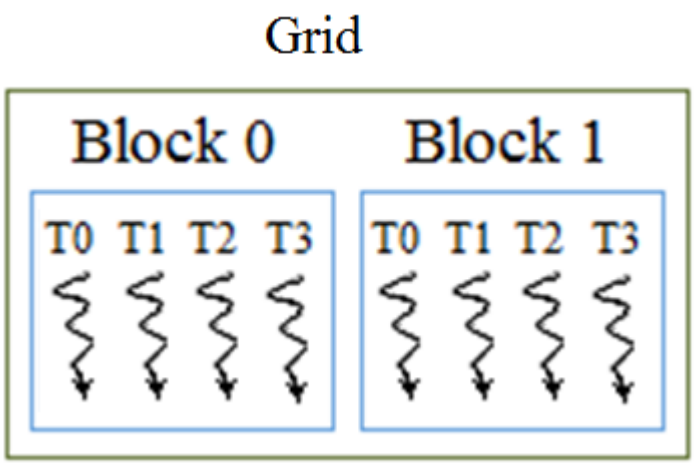
	Código del kernel
4	`__global__ void add( int a[], int b[], int c[] ) {`
5	`  int tid = threadIdx.x + blockIdx.x * blockDim.x;`
6	`  if (tid < N)`
7	`  c[tid] = a[tid] + b[tid];`
8	`}`

	Llamada al kernel
	`...`
24	`add<<<(N+3)/4,4>>>( dev_a, dev_b, dev_c );`
	`...`

```
correKernel<<<1 ,1>>>(...);
gridDim.x   = 1
blockDim.x  = 1
blockIdx.x  = 0
threadIdx.x = 0
```



```
correKernel<<<2 ,4>>>(...);
gridDim.x   = 2
blockDim.x  = 4
blockIdx.x  = 0,1
threadIdx.x = 0,1,2,3
```



Para grandes cantidades de datos

- Suponiendo que el tamaño del arreglo es mayor a la capacidad de los threads del GPU, entonces el código del kernel quedará de la siguiente manera:

	Código del kernel

```
__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 += blockDim.x * gridDim.x;
  }
}
```

- Supongamos que el tamaño máximo es:

  `gridDim.x = 4  blockDim.x = 4`

- Y nuestro vector es de tamaño 32, entonces:

Primer iteración, se cumple tid < 32

Bloque 0	Bloque 1	Bloque 2	Bloque 3
tid=0+0*4=0	tid=0+1*4=4	tid=0+2*4=8	tid=0+3*4=12
tid=1+0*4=1	tid=1+1*4=5	tid=1+2*4=9	tid=1+3*4=13
tid=2+0*4=2	tid=2+1*4=6	tid=2+2*4=10	tid=2+3*4=14
tid=3+0*4=3	tid=3+1*4=7	tid=3+2*4=11	tid=3+3*4=15

```
tid += blockDim.x * gridDim.x;
```

```
tid = tid + 4 * 4;
tid = tid + 16;
```

Segunda iteración

Bloque 0	Bloque 1	Bloque 2	Bloque 3
tid=0+16 =16	tid=4+16=20	tid= 8+16=24	tid=12+16=28
tid=1+16=17	tid=5+16=21	tid= 9+16=25	tid=13+16=29
tid=2+16=18	tid=6+16=22	tid=10+16=26	tid=14+16=30
tid=3+16=19	tid=7+16=23	tid=11+16=27	tid=15+16=31

Imprimir datos del GPU

- Para imprimir valores ubicados en el GPU, tenemos la biblioteca cuPrintf.
- Nos ayuda de tal manera que nos indica el bloque y thread al que pertenece la salida a impresión.

```
1   #include <cuda.h>
2   #include "cuPrintf.cu"
3   __global__ void cuPrintfExample()
4   {
5    int tid;
6    tid = blockIdx.x * blockDim.x + threadIdx.x;
7    cuPrintf("%d\n", tid);
8   }
9   int main()
10  {
11   cudaPrintfInit();
12   cuPrintfExample <<< 5, 2 >>> ();
13   cudaPrintfDisplay(stdout, true);
14   cudaPrintfEnd();
15   return 0;
16  }
```

```
[netzrod@tonatiuh programas]$ ./imprimeDatos
[0, 0]: 0
[0, 1]: 1
[1, 0]: 2
[1, 1]: 3
[2, 0]: 4
[2, 1]: 5
[3, 0]: 6
[3, 1]: 7
[4, 0]: 8
[4, 1]: 9
```

Actividad

- Realizar un programa que sume dos matrices.
- Realizar un módulo que sume dos matrices.

  Aquí es necesario conocer como pasar una matriz a un módulo.
- Elaborar el main para invocar al módulo que sume dos matrices.

Referencias

- Sito de NVIDIA, https://developer.nvidia.com/
- CUDA C PROGRAMMING GUIDE, NVIDIA
- CUDA by Examples, NVIDIA
- cuPrintf, http://www.jeremykemp.co.uk/08/02/2010/cuda-cuprintf/

http://www.cse.ohio-state.edu/~godwinj/gpgpu12-godwin/codes/cuPrintf.cu