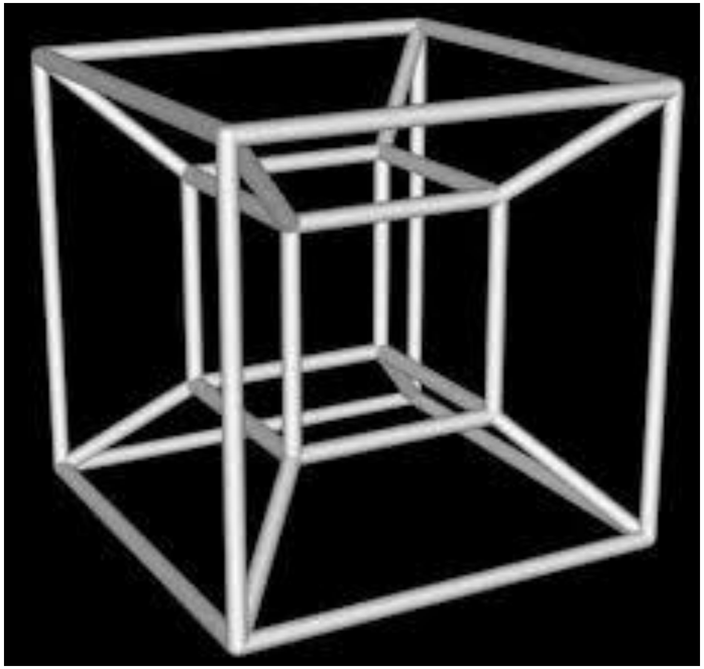


DIMENSIONES DE THREADS Y BLOCKS

Netz Romero

`gridDim`   - indica el número de bloques en un grid.

`blockDim` - indica el número de threads en un bloque.

`blockIdx` - índice del bloque dentro del grid.

`threadIdx` - índice del thread dentro del bloque.

□ Los bloques puede trabajar en 1D ó 2D

□ Los threads pueden trabajar en 1D, 2D ó 3D

dim3

- **`dim3`** - es un tipo de vector entero usado en cuda para indicar las dimensiones en el grid o bloque cuando se invoca el kernel.
- dim3 puede tomar 1, 2 ó 3 argumentos

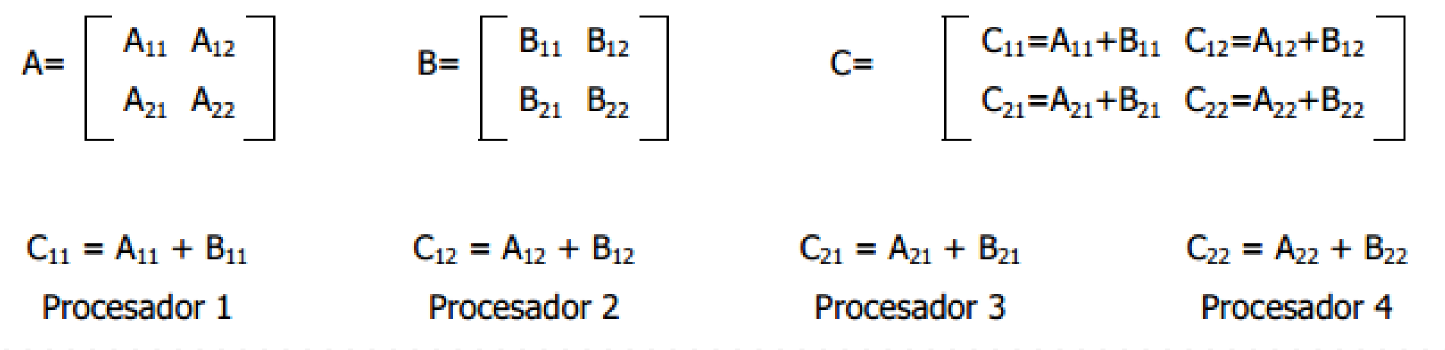
```
dim3 varDim1D(x);
dim3 varDim2D(x, y);
dim3 varDim3D(x, y, z);
```

Actividad 4.1

- Realizar una función que sume dos matrices, pero los argumentos deben ser apuntadores sencillos.
- Construir el main para comprobar la función.
- Corroborar el casting adecuado.

Sumar dos matrices en CUDA

- Se van a sumar cada uno de los elementos de las matrices a sumar en forma simultánea

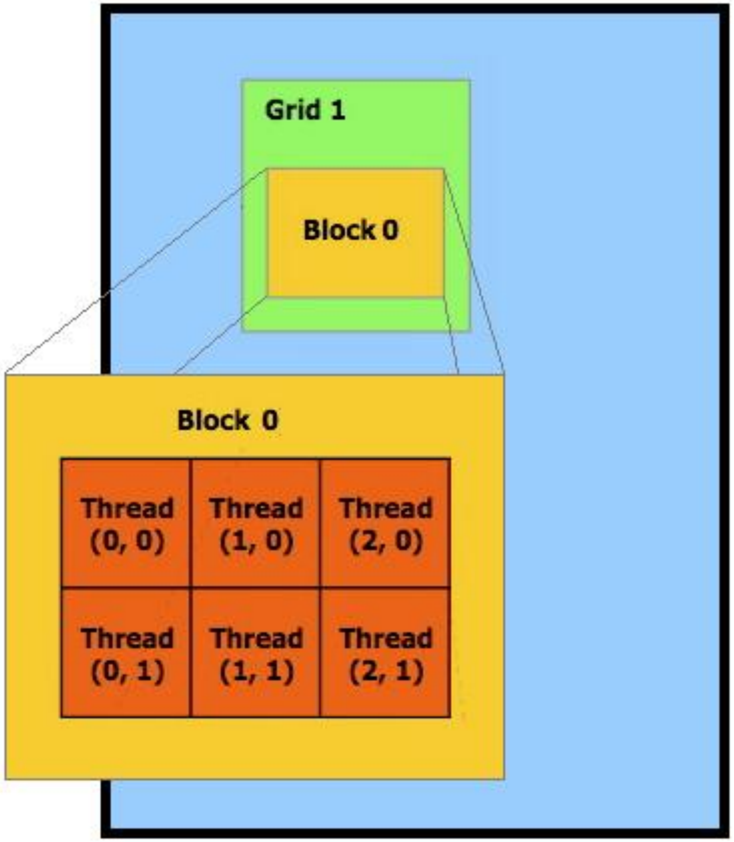


$$A= \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \qquad B= \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \qquad C= \begin{bmatrix} C_{11}=A_{11}+B_{11} & C_{12}=A_{12}+B_{12} \\ C_{21}=A_{21}+B_{21} & C_{22}=A_{22}+B_{22} \end{bmatrix}$$

$C_{11} = A_{11} + B_{11}$	$C_{12} = A_{12} + B_{12}$	$C_{21} = A_{21} + B_{21}$	$C_{22} = A_{22} + B_{22}$
Procesador 1	Procesador 2	Procesador 3	Procesador 4

```
#include<stdio.h>
#define F 2
#define C 3
__global__ void sumaMat(int a[], int b[], int c[]) {
  int x = threadIdx.x; //Filas
  int y = threadIdx.y; //Columnas
  int tid = y + (C*x);
  c[tid] = a[tid] + b[tid];
}
int main() {
  int numBlocks = 1;
  int i, j;
  dim3 threadsPerBlock(F, C);
  int a_h[F][C]={{1,1,1},{3,3,3}};
  int b_h[F][C]={{2,2,2},{4,4,4}};
  int c_h[F][C];
  int *a_d, *b_d, *c_d;
  int size = F*C*sizeof(int);
  cudaMalloc((void **)&a_d, size);
  cudaMalloc((void **)&b_d, size);
  cudaMalloc((void **)&c_d, size);
  cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);
  cudaMemcpy(b_d, b_h, size, cudaMemcpyHostToDevice);
  sumaMat<<<numBlocks, threadsPerBlock>>>(a_d, b_d, c_d);
  cudaMemcpy(c_h, c_d, size, cudaMemcpyDeviceToHost);

  for(i = 0; i < F; i++)
    for(j = 0; j < C; j++)
       printf("c_h[%d][%d]=%d\n",i, j, c_h[i][j]);

  cudaFree(a_d);
  cudaFree(b_d);
  cudaFree(c_d);
  return 0;
}
```

```
#define F 2

#define C 3

...

int numBlocks = 1;

dim3 threadsPerBlock(F, C);

...

sumaMat<<<numBlocks,threadsPerBlock
   >>>(a_d, b_d, c_d);

...
```



```
int numBlocks = 1;
dim3 threadsPerBlock(2, 3);
sumaMat<<<numBlocks,threadsPerBlock>>>(...);
gridDim.x   = 1
blockDim.x  = 2
blockDim.y  = 3
threadIdx.x = 0,1
threadIdx.y = 0,1,2
```

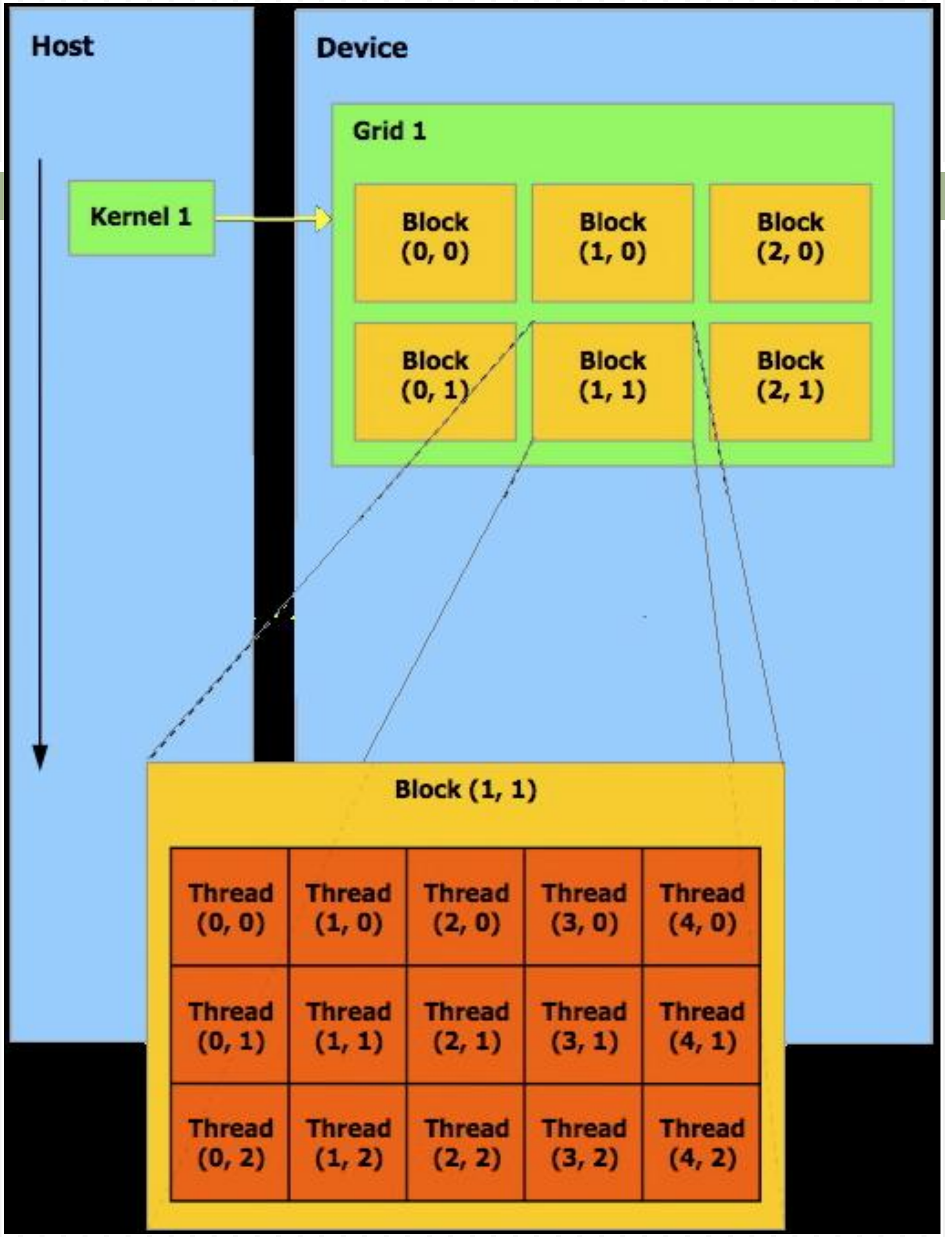
Actividad 4.2

- **Imprimir los valores anteriores utilizando la biblioteca `cuPrintf.cu`, la salida seria:**

```
$./imprimeDatosMat
[0, 0]: gd=1 bdx=2 bdy=3 tx=0  ty=0
[0, 1]: gd=1 bdx=2 bdy=3 tx=1  ty=0
[0, 2]: gd=1 bdx=2 bdy=3 tx=0  ty=1
[0, 3]: gd=1 bdx=2 bdy=3 tx=1  ty=1
[0, 4]: gd=1 bdx=2 bdy=3 tx=0  ty=2
[0, 5]: gd=1 bdx=2 bdy=3 tx=1  ty=2
```

Ejercicio

- □ Calcular la suma de dos matrices de 6x15.
- □ Con un grid de 2x3 bloques.
- □ Con bloques de 3x5 threads.
- □ Se maneja la sintaxis Filas x Columnas.



Código del kernel

```
__global__ void sumaMat(int a[], int b[], int c[])
{
  int x = threadIdx.x + blockDim.x*blockIdx.x; //Fil
  int y = threadIdx.y + blockDim.y*blockIdx.y; //Col
  ...
}
```

Llamada al kernel

```
...
dim3 numBlocks(2, 3);         //Grid structure
dim3 threadsPerBlock(3, 5);   //Block structure
...
//Invocando al kernel
sumaMat<<<numBlocks, threadsPerBlock>>>(a_d, b_d, c_d);
...
```

```
dim3 numBlocks(2, 3);
dim3 threadsPerBlock(3, 5);
sumaMat<<<numBlocks,threadsPerBlock>>>(...);
gridDim.x   = 2
gridDim.y   = 3
blockDim.x  = 3
blockDim.y  = 5
blockIdx.x  = 0,1
blockIdx.y  = 0,1,2
threadIdx.x = 0,1,2
threadIdx.y = 0,1,2,3,4
```

Actividad 4.3

- Imprimir los valores para el ID del bloque y de los threads, utilizando la biblioteca `cuPrintf.cu`, la salida seria:

```
$./imprimeDatosMat_2
[0, 0]: bx =0 by =0 tx =0 ty =0
[0, 1]: bx =0 by =0 tx =1 ty =0
[0, 2]: bx =0 by =0 tx =2 ty =0
[0, 3]: bx =0 by =0 tx =0 ty =1
[0, 4]: bx =0 by =0 tx =1 ty =1
[0, 5]: bx =0 by =0 tx =2 ty =1
```

```
[0, 6]: bx =0 by =0 tx =0 ty =2
[0, 7]: bx =0 by =0 tx =1 ty =2
[0, 8]: bx =0 by =0 tx =2 ty =2
[0, 9]: bx =0 by =0 tx =0 ty =3
[0, 10]: bx =0 by =0 tx =1 ty =3
[0, 11]: bx =0 by =0 tx =2 ty =3
[0, 12]: bx =0 by =0 tx =0 ty =4
[0, 13]: bx =0 by =0 tx =1 ty =4
[0, 14]: bx =0 by =0 tx =2 ty =4
[1, 0]: bx =1 by =0 tx =0 ty =0
[1, 1]: bx =1 by =0 tx =1 ty =0
...
```

Referencias

- Sito de NVIDIA, https://developer.nvidia.com/
- CUDA C PROGRAMMING GUIDE, NVIDIA
- CUDA by Examples, NVIDIA