

Introducción a CUDA

Chips de Interconexión del Futuro – Curso 2007/2008

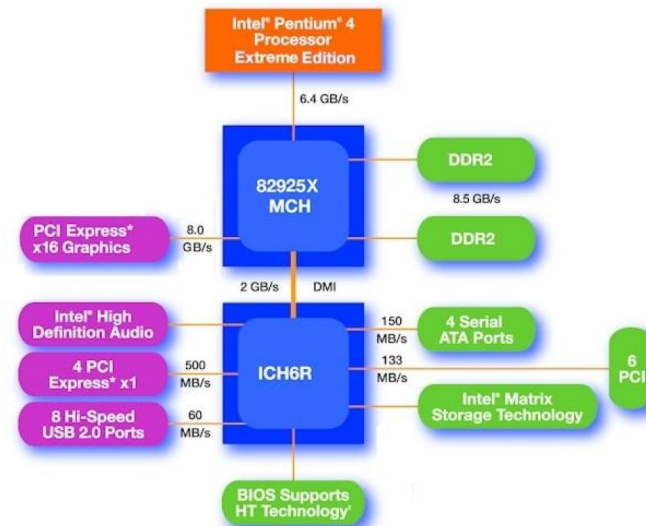
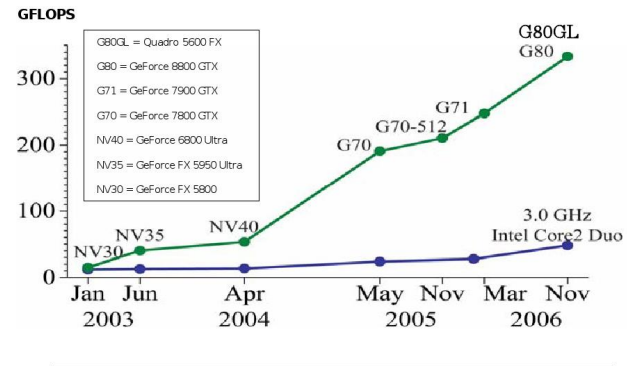
Gaspar Mora

Basado en las Transparencias de:

David Kirk (NVIDIA), Wen-mei Hwu (Univ. of Illinois)

Introducción

- GPUs: Una revolución en la sombra.
 - Potencia de cálculo: 367 GFLOPS vs 32 GFLOPS.
 - Ancho de banda de la memoria: 86.4 GB/s vs 8.4 GB/s.
 - Disponibles en casi todo PC.



CUDA

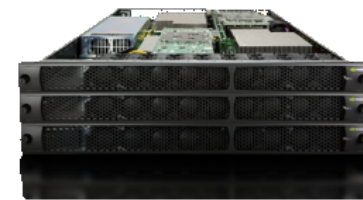
- CUDA (*Compute Unified Device Architecture*)
 - Programación de propósito general para la GPU (no más OpenGL).
 - GPU como coprocesador masivamente paralelo multi-hilo.
 - Arquitectura SPMD.



GeForce 8800



Tesla D870

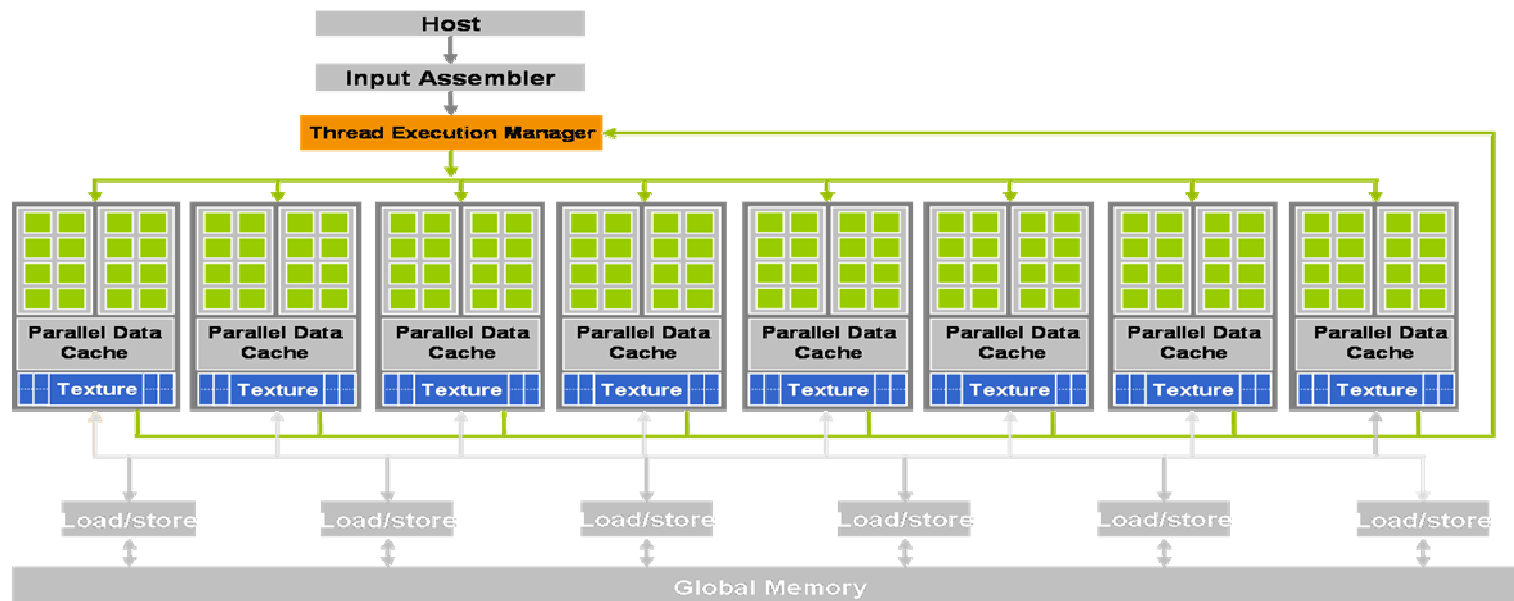


Tesla S870



NVIDIA GeForce 8800

- 16 multiprocesadores (90W consumo)
- 8 procesadores (FPUs) por multiprocesador.
- Total 128 cores (378 GFLOPs)
- 768MB RAM
- 86.4GB/s ancho de banda de la memoria.

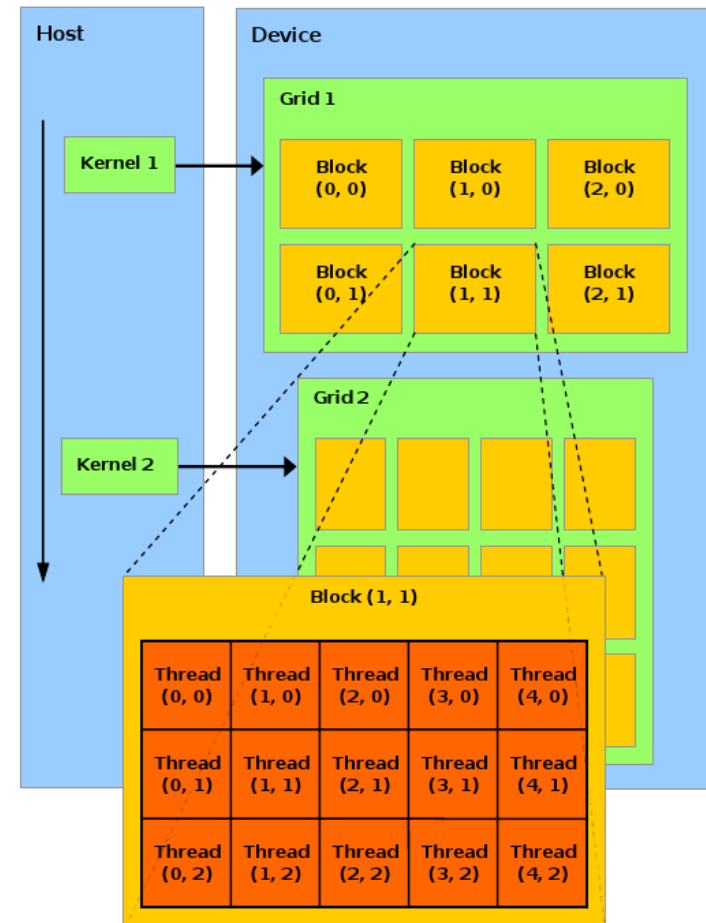


Programación en CUDA

- La GPU es un dispositivo (*device*):
 - Que funciona como coprocesador de la CPU.
 - Tiene su propia RAM.
 - Ejecuta multitud de hilos en paralelo.
- Paradigma SPMD:
 - Porciones de una aplicación actuando en paralelo sobre diferentes datos se ejecuta en la GPU como “*kernels*”:
- Hilos en la CPU vs Hilos en la GPU:
 - Los hilos en la GPU son extremadamente ligeros (poco coste de creación)
 - En la GPU se trabaja con miles de hilos simultáneamente frente a unos pocos en la CPU.

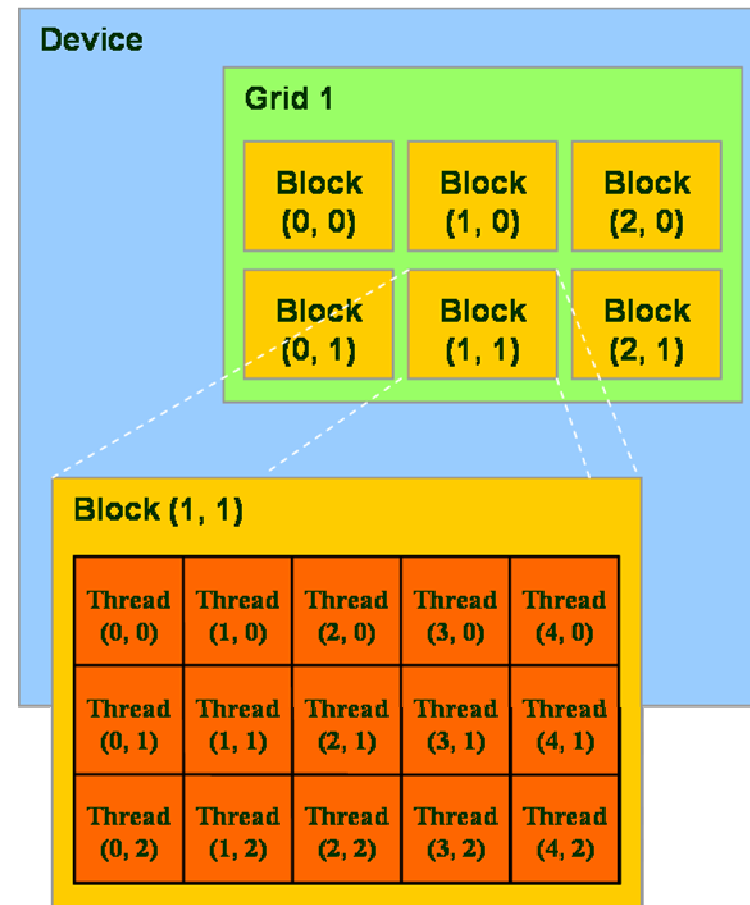
Organización de Hilos en CUDA

- Un *kernel* se ejecuta como **mall** (*grid*) de **bloques** de hilos.
 - Todos los hilos comparten la memoria de datos.
- Un **bloque** de hilos es un lote de hilos que pueden cooperar entre ellos:
 - Sincronizándose.
 - Compartiendo datos mediante una memoria compartida de muy baja latencia.



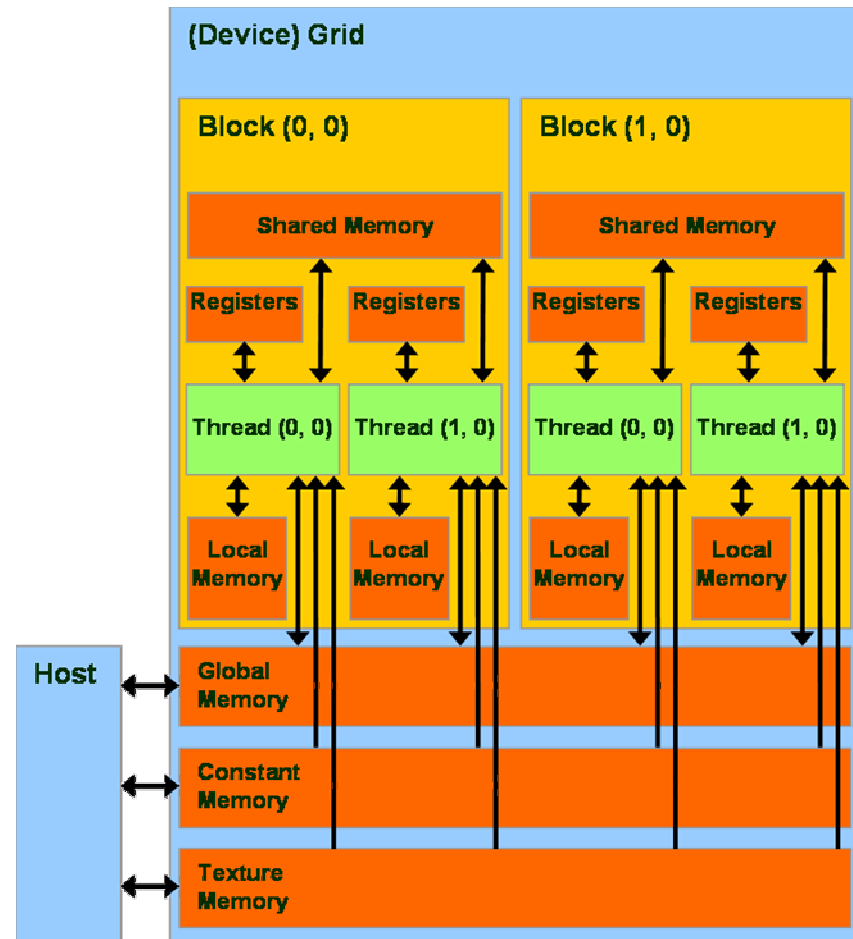
Identificadores de Bloques e Hilos

- Los bloques e hilos tienen identificadores.
 - Así cada hilo sabe sobre qué datos actuar.
 - BlockID: 1D ó 2D
 - ThreadID: 1D, 2D ó 3D
- Simplifica el direccionamiento al procesar datos multidimensionales:
 - Imágenes, matrices, PDEs en volúmenes, etc.



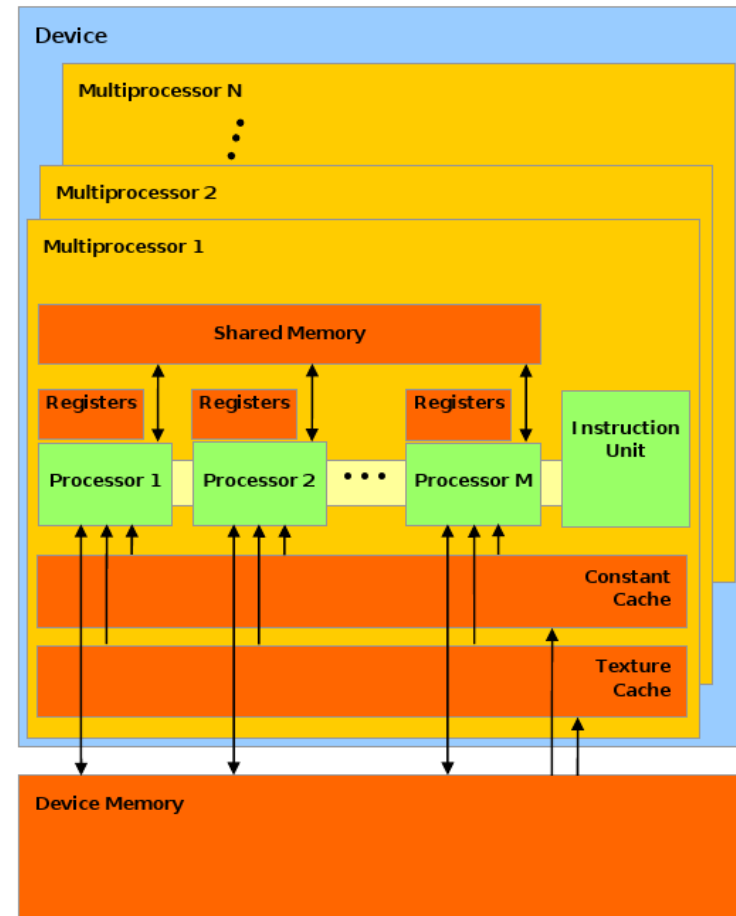
Espacio de memoria en CUDA

- Cada hilo puede:
 - R/W registros locales al hilo.
 - R/W memoria local al hilo.
 - R/W memoria compartida por todo el bloque.
 - R/W memoria global compartida por toda la malla (grid).
 - Sólo lectura de las memorias constante y de textura.
- El host (CPU) puede R/W las memorias globales, constante y de textura.



Hardware CUDA (Versión 1.x)

- Cada **bloque** es asignado a un multiprocesador.
- 1 bloque son como máximo 512 hilos.
- Los hilos de un bloque se agrupan en **warps** para su ejecución.
- 1 warp son 32 hilos.
- Los hilos de un warp se ejecutan en un procesador.
- 8 procesadores con registros de 32 bits por multiprocesador.
- Por tanto, 4 ciclos de reloj para completar un warp.



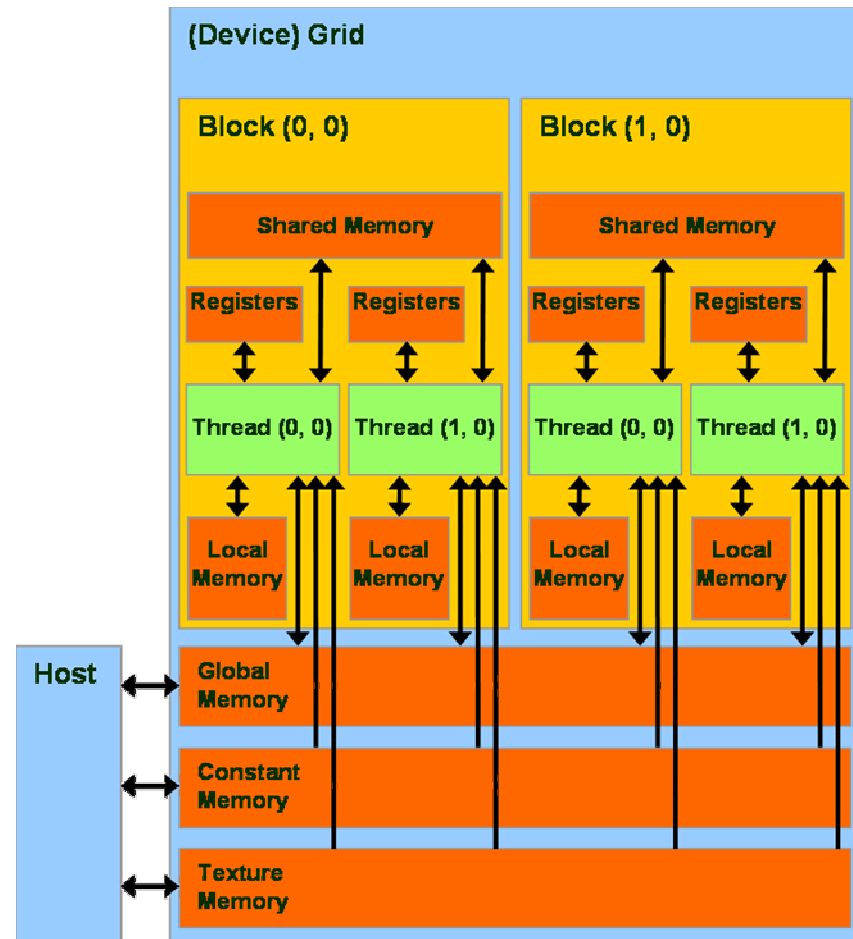
API de CUDA

- La API de CUDA es una extensión al lenguaje **ANSI C**.
 - Curva de aprendizaje suave.
- Ejemplo de multiplicación de matrices.
 - Usaremos la siguiente estructura para las matrices:
 - Matriz 2D
 - Elementos `float`.
 - La memoria para la matriz se reserva con `malloc()` y se enlaza con `elements`.

```
typedef struct {  
    int width;  
    int height;  
    int pitch;  
    float * elements;  
} Matrix;
```

Gestión de memoria con CUDA

- **cudaMalloc()**
 - Obtiene espacio en la memoria global.
 - Requiere dos parámetros (dirección del puntero y el tamaño a reservar)
- **cudaFree()**
 - Libera el objeto.



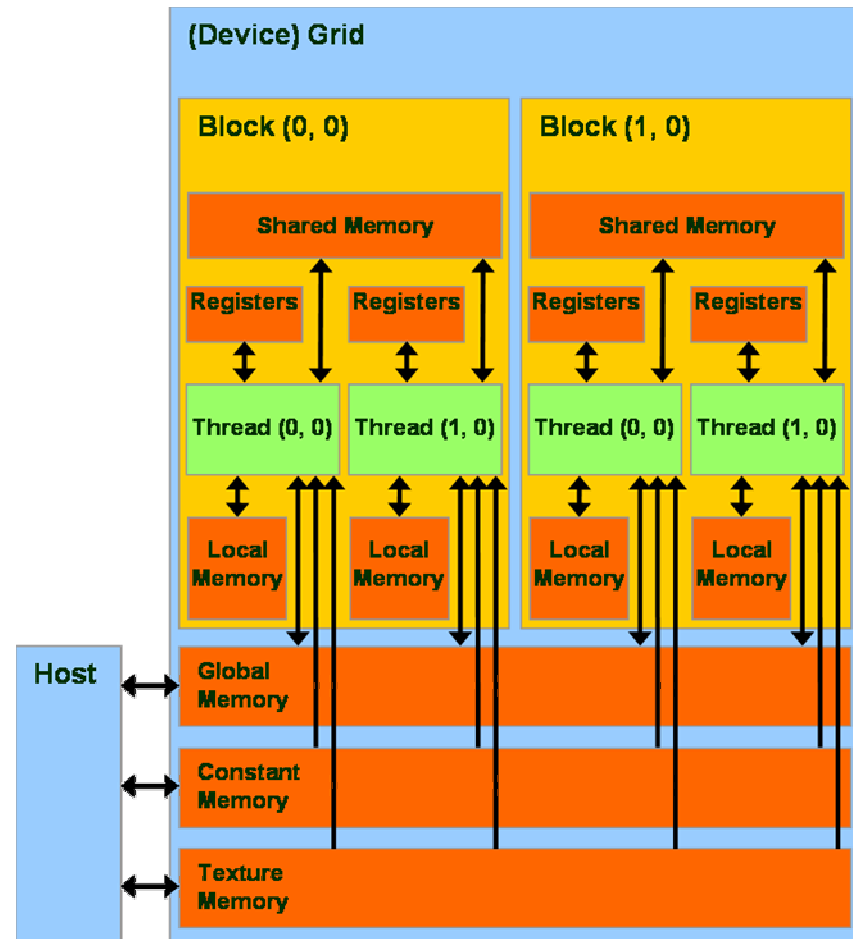
Gestión de memoria con CUDA – Ejemplo 1

- Pretendemos:
 - Reservar espacio para una matriz 64x64 de elementos `float`.
 - Enlazar ese espacio a `Md.elements`.

```
#define BLOCK_SIZE 64
Matrix Md;
int size = BLOCK_SIZE*BLOCK_SIZE*sizeof(float);
cudaMalloc((void**) &Md.elements, size);
cudaFree(Md.elements);
```

Gestión de memoria con CUDA (2)

- **cudaMemcpy ()**
 - Transfiere datos entre memorias.
 - Requiere 4 parámetros:
 - Puntero al origen.
 - Puntero al destino.
 - Bytes a copiar.
 - Tipo de transferencia:
 - Host a host.
 - Host a dispositivo.
 - Dispositivo a host.
 - Dispositivo a dispositivo.



Gestión de memoria con CUDA – Ejemplo 2

- Pretendemos:
 - Transferir un array de 64x64 `float` de la memoria de la GPU a la del sistema y viceversa.
 - Matriz **M** está en el *host* y la matriz **Md** en el dispositivo.

```
cudaMemcpy(Md.elements, M.elements, size, cudaMemcpyDeviceToHost) ;  
cudaMemcpy(M.elements, Md.elements, size, cudaMemcpyHostToDevice) ;
```

Modificadores de Función

	Executed on the:	Only callable from the:
<code>__device__ float DeviceFunc()</code>	device	device
<code>__global__ void KernelFunc()</code>	device	host
<code>__host__ float HostFunc()</code>	host	host

- `__global__` define un **kernel**.
 - Devuelve siempre un `void`.
- Restricciones en funciones ejecutadas en el dispositivo:
 - Recursividad **no** permitida.
 - Número variable de argumentos **no** permitido.

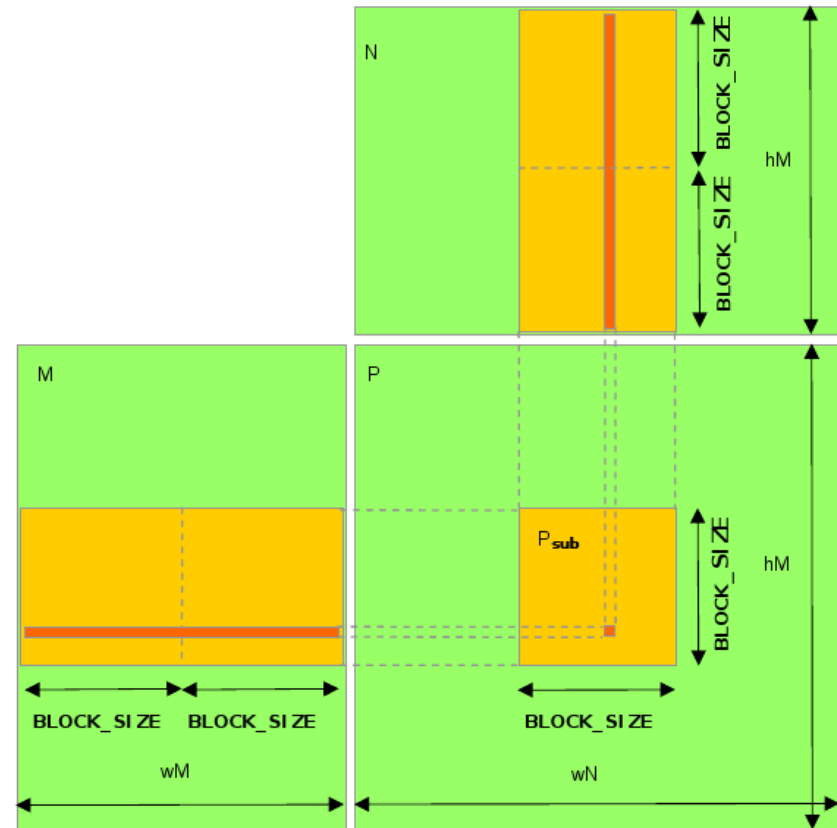
Creando Hilos (llamadas a *kernel*)

- Un *kernel* debe ser ejecutado del siguiente modo:

```
__global__ void KernelFunction(...);  
dim3 DimGrid(100,50); // 5000 bloques  
dim3 DimBlock(4,8,8); // 256 hilos/bloque  
KernelFunction<<< Dimgrid,Dimblock >>> (...);
```

Ejemplo multiplicación de matrices

- Producto de $M \times N$.
- Cada hilo computa un elemento de P .



Paso 1: Copiar a memoria.

```
// Allocate the device memory where we will copy M to
Matrix Md;
Md.width  = WIDTH;
Md.height = WIDTH;
Md.pitch  = WIDTH;
int size = WIDTH * WIDTH * sizeof(float);
cudaMalloc((void**)&Md.elements, size);

// Copy M from the host to the device
cudaMemcpy(Md.elements, M.elements, size,
           cudaMemcpyHostToDevice);

// Read M from the device to the host into P
cudaMemcpy(P.elements, Md.elements, size,
           cudaMemcpyDeviceToHost);

...
// Free device memory
cudaFree(Md.elements);
```

Paso 2: Estructura general

```
int main(void) {  
    // Allocate and initialize the matrices (at host)  
    Matrix M = AllocateMatrix(WIDTH, WIDTH, 1);  
    Matrix N = AllocateMatrix(WIDTH, WIDTH, 1);  
    Matrix P = AllocateMatrix(WIDTH, WIDTH, 0);  
  
    // M * N on the device  
    MatrixMulOnDevice(M, N, P);  
  
    // Free matrices  
    FreeMatrix(M);  
    FreeMatrix(N);  
    FreeMatrix(P);  
    return 0;  
}
```

Paso 3a: Multiplicación en la GPU

```
// Matrix multiplication on the device
void MatrixMulOnDevice(const Matrix M, const Matrix N, Matrix P)
{
    // Load M and N to the device
    Matrix Md = AllocateDeviceMatrix(M);
    CopyToDeviceMatrix(Md, M);
    Matrix Nd = AllocateDeviceMatrix(N);
    CopyToDeviceMatrix(Nd, N);

    // Allocate P on the device
    Matrix Pd = AllocateDeviceMatrix(P);
    CopyToDeviceMatrix(Pd, P); // Clear memory
```

Paso 3b: Multiplicación en la GPU

```
// Setup the execution configuration
dim3 dimBlock(WIDTH, WIDTH);
dim3 dimGrid(1, 1);

// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd);

// Read P from the device
CopyFromDeviceMatrix(P, Pd);

// Free device matrices
FreeDeviceMatrix(Md);
FreeDeviceMatrix(Nd);
FreeDeviceMatrix(Pd);
}
```

Paso 4a: *Kernel*

```
// Matrix multiplication kernel - thread specification
__global__ void MatrixMulKernel(Matrix M, Matrix N, Matrix P)
{
    // 2D Thread ID
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Pvalue is used to store the element of the matrix
    // that is computed by the thread
    float Pvalue = 0;
```

Paso 4b: *Kernel*

```
for (int k = 0; k < M.width; ++k)
{
    float Melement = M.elements[ty * M.pitch + k];
    float Nelement = Nd.elements[k * N.pitch + tx];
    Pvalue += Melement * Nelement;
}
// Write the matrix to device memory;
// each thread writes one element
P.elements[ty * P.pitch + tx] = Pvalue;
}
```