



UNIVERSITAT POLITÈCNICA DE CATALUNYA  
BARCELONATECH

Departament d'Arquitectura de Computadors

# Tarjetas Gráficas y Aceleradores

## CUDA

Agustín Fernández

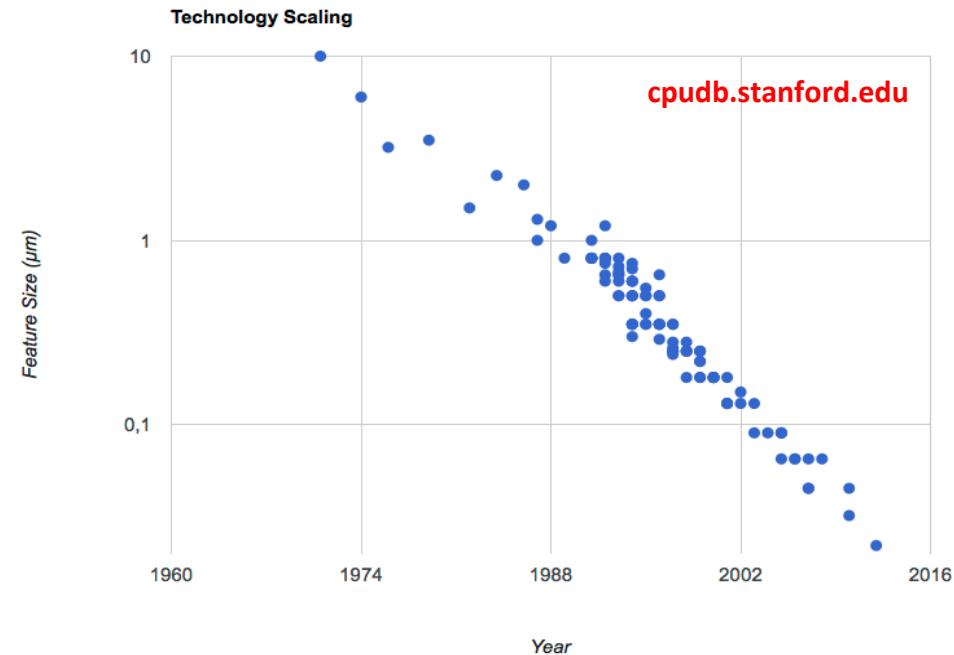
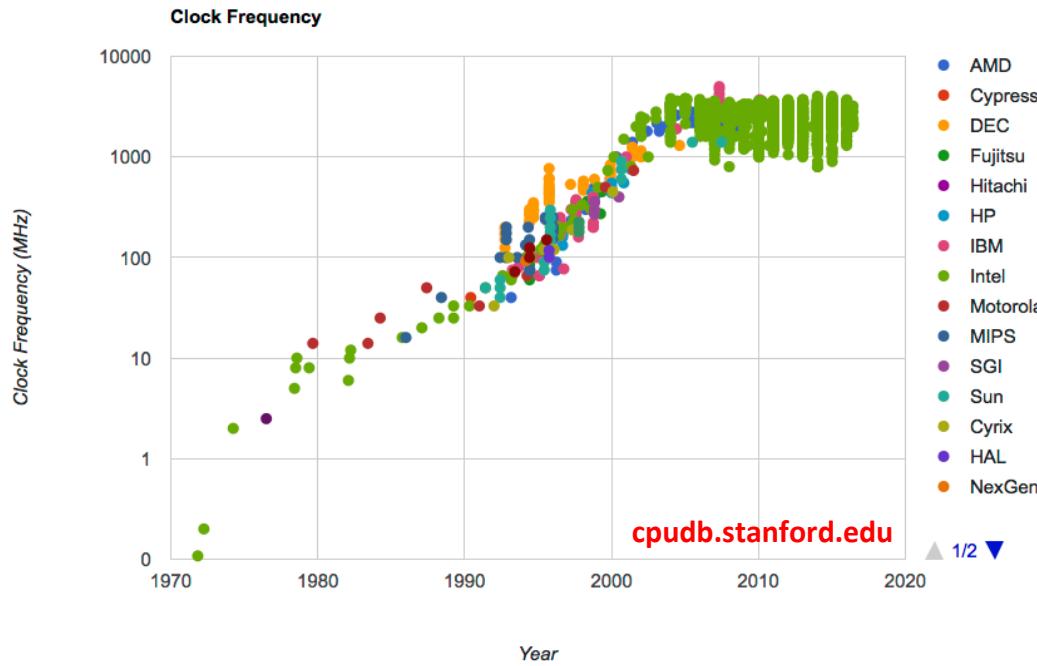
Departament d'Arquitectura de Computadors

Facultat d'Informàtica de Barcelona

Universitat Politècnica de Catalunya



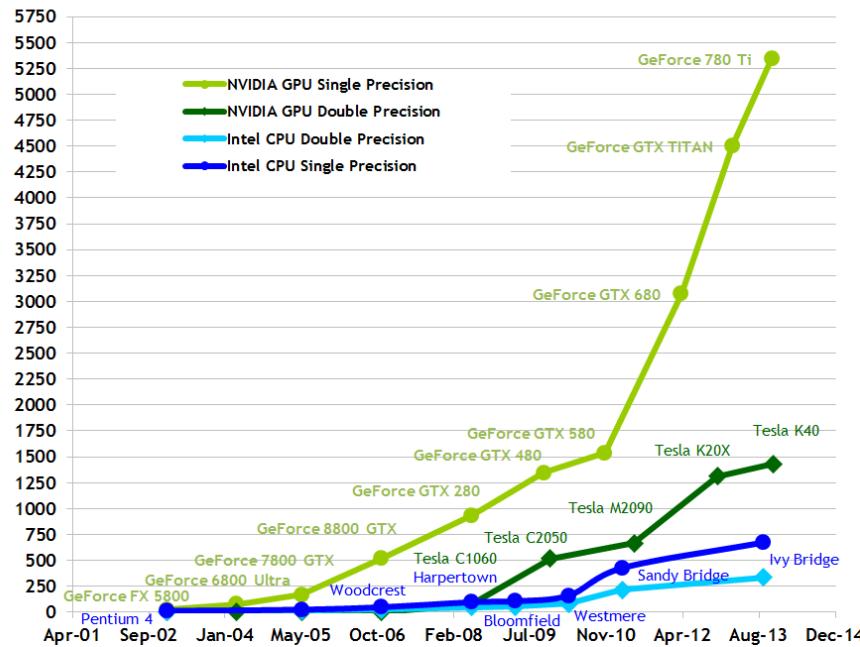
# Tendencias tecnológicas en el diseño de CPUs



- **Buenas noticias: Los transistores son cada vez más pequeños**
- **El aumento de rendimiento no se puede conseguir subiendo la frecuencia**
- **El consumo y la disipación de calor influyen mucho en el diseño de las nuevas CPUs**

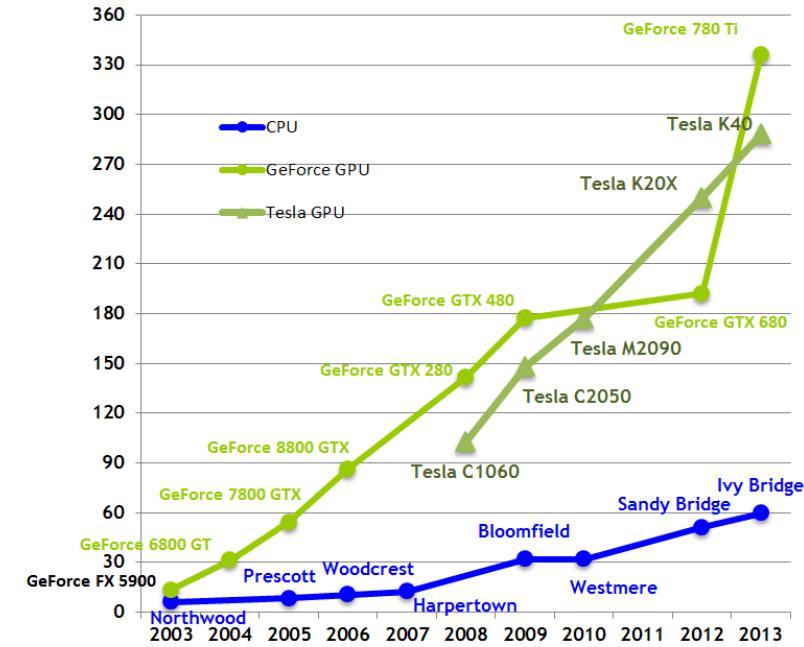
# Comparativa GPU - CPU

Theoretical GFLOP/s



Operaciones en coma flotante por segundo

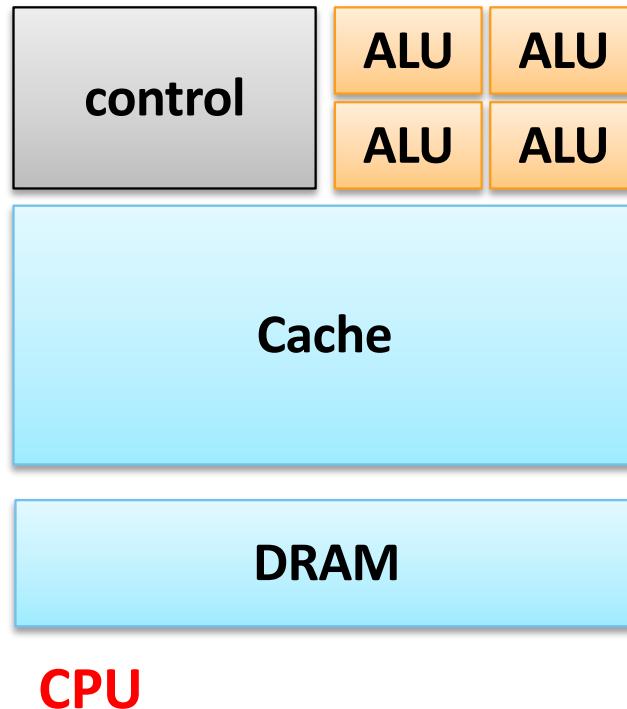
Theoretical GB/s



Ancho de banda con Memoria

Hay una GPU en casi todos los PCs y/o workstations

# CPU: Diseño Orientado a Reducir la Latencia



## Caches muy grandes

- Reducir la latencia con memoria

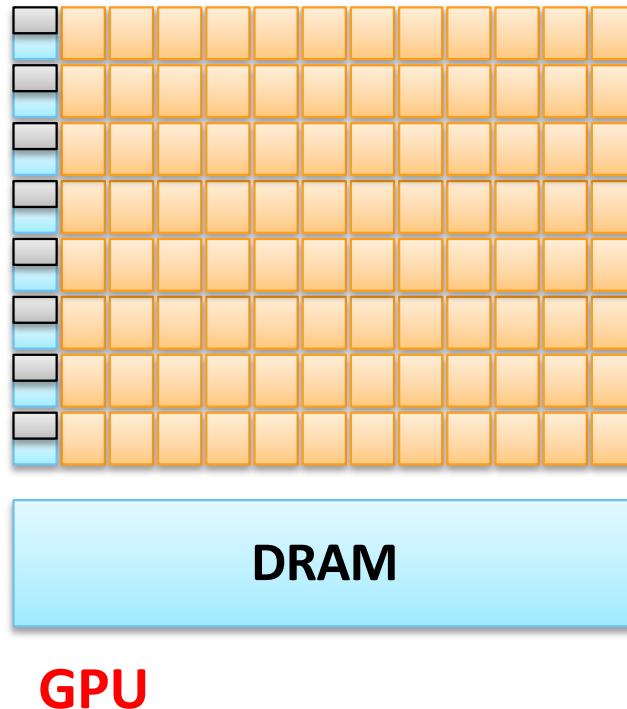
## Control muy sofisticado

- Predictores de saltos para ocultar la latencia de los saltos
- Cortocircuitos para reducir la latencia de los riesgos de datos

## ALUs muy potentes

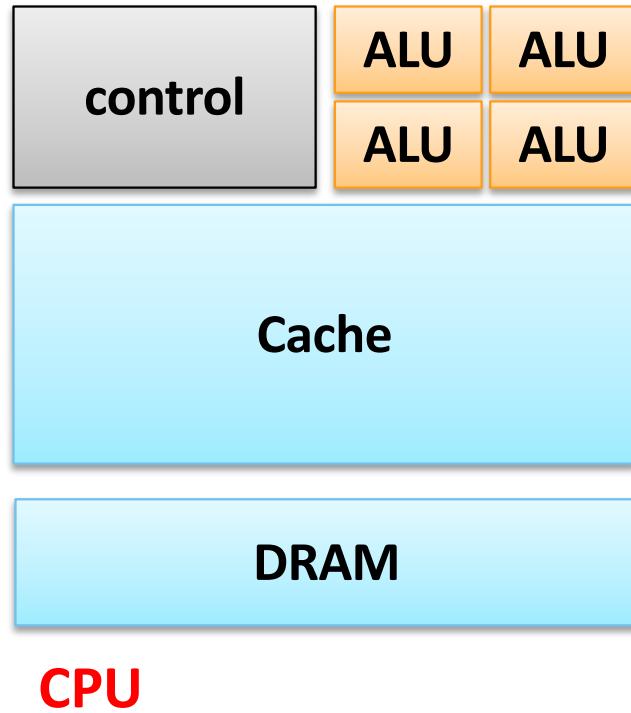
- Reducir la latencia de las operaciones de cálculo

# GPU: Diseño Orientado a Aumentar el Throughput

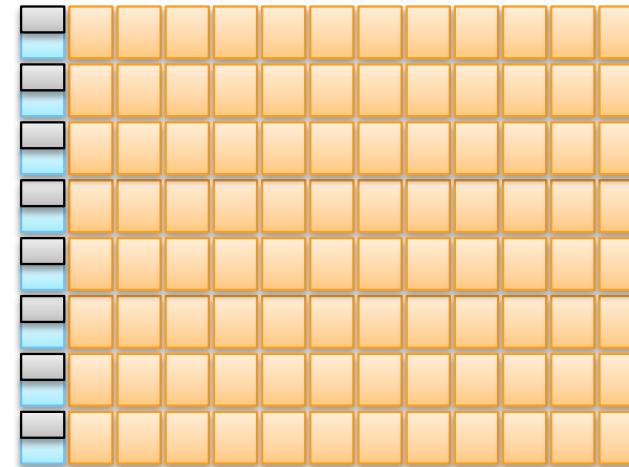


- Caches pequeñas
- Control sencillo
  - Sin predicción de saltos
  - Sin cortocircuitos
- ALUs con alta eficiencia energética
  - Muchas y simples
  - Elevadas latencias
  - Muy segmentadas para aumentar el throughput
- Requiere un número masivo de threads para ocultar la latencia

# Comparativa GPU - CPU



CPU



GPU

La filosofía de diseño es completamente diferente.

# CUDA (Compute Unified Device Architecture)

- CUDA es un conjunto de herramientas (creadas por nVIDIA) que permite codificar programas para las GPUs de nVIDIA.
- Funciona en todas las GPUs de nVIDIA a partir de la serie G8x.
- No es necesario utilizar ninguna API gráfica. CUDA se encarga de gestionar el uso de la GPU.
- No todas las GPUs de nVIDIA soportan cualquier versión de CUDA.
  - En “[developer.nvidia.com/cuda-gpus](http://developer.nvidia.com/cuda-gpus)” hay una lista exhaustiva de las todas las GPUs que soportan CUDA, indicando la versión soportada.



## Tesla K40c

- Familia: Kepler, GPU GK110B
- 2880 CUDA cores
- 4.29 TFLOPS (32b)
- 1.43 TFLOPS (64b)
- Computing Capability: 3.5

# CUDA

- nVIDIA está dedicando muchos recursos a promocionar CUDA
- Centenares de aplicaciones portadas a CUDA en múltiples ámbitos:
  - ✓ Dinámica de Fluidos
  - ✓ Diseño asistido por computador
  - ✓ Creación de contenidos digitales
  - ✓ Diseño electrónico
  - ✓ Finanzas
  - ✓ Generación imágenes médicas
  - ✓ Cálculo numérico
  - ✓ Ciencias de la vida
  - ✓ Industrias de Gas y Petróleo
  - ✓ Procesado de señal
  - ✓ Audio y Vídeo
  - ✓ ...

**GPU Gems [1][2][3]: Programming Techniques,  
Tips and Tricks for Real-Time Graphics.**

- Hay aceleraciones que superan 1000x, pero cuando hay resultados, 10x-20x es lo más común.

# ¿Qué es CUDA?

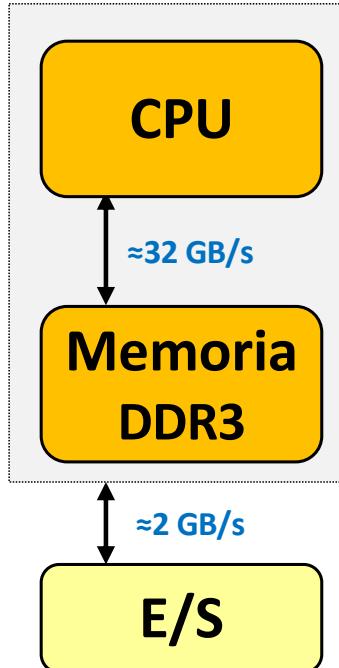
- Plataforma diseñada de forma conjunta a nivel de software y hardware para hacer uso de la potencia de cálculo de las GPUs en aplicaciones de uso general en tres niveles:
  - **Software:** Permite programar la GPU con pequeñas extensiones, pero muy potentes, que permiten la **programación heterogénea** y logran una ejecución eficiente y escalable.
  - **Firmware:** Ofrece un driver orientado a la programación GPGPU, que es compatible con la utilizada en la renderización de gráficos 3D. Es una API sencilla para gestionar dispositivos, memoria, etc.
  - **Hardware:** Expone el paralelismo masivo de la GPU para cálculo de propósito general a través de una serie de multiprocesadores con múltiples de núcleos y una jerarquía de memoria.

# CUDA C de un vistazo

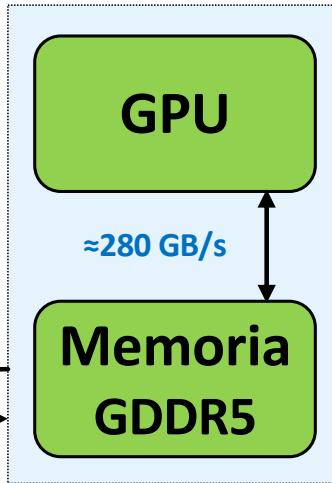
- En esencia, es C, con algunas extensiones:
  - El programador describe el **código de 1 thread**, y el código se instancia automáticamente para miles de threads.
- CUDA define:
  - Un modelo de arquitectura:
    - ✓ Con muchos núcleos de cálculo agrupados en multiprocesadores que usan una unidad de control SIMD.
  - Un modelo de programación:
    - ✓ Basado en paralelismo masivo de datos y paralelismo de grano fino.
    - ✓ Escalable: El código se puede ejecutar en un número diferente de núcleos sin necesidad recompilar.
  - Un modelo de gestión de la memoria:
    - ✓ Explícita al programador, las caches no son transparentes.
- Objetivos:
  - Construir códigos que escalan a cientos de núcleos de manera simple. Permite declarar miles de threads.
  - Permitir la computación heterogénea (CPUs y GPUs).

# Host and Device (GPU)

## HOST



## DEVICE



## HOST

- ❑ Gestión de los datos, E/S
- ❑ Inicializaciones
- ❑ Código secuencial
- ❑ Interacción con el usuario

## DEVICE

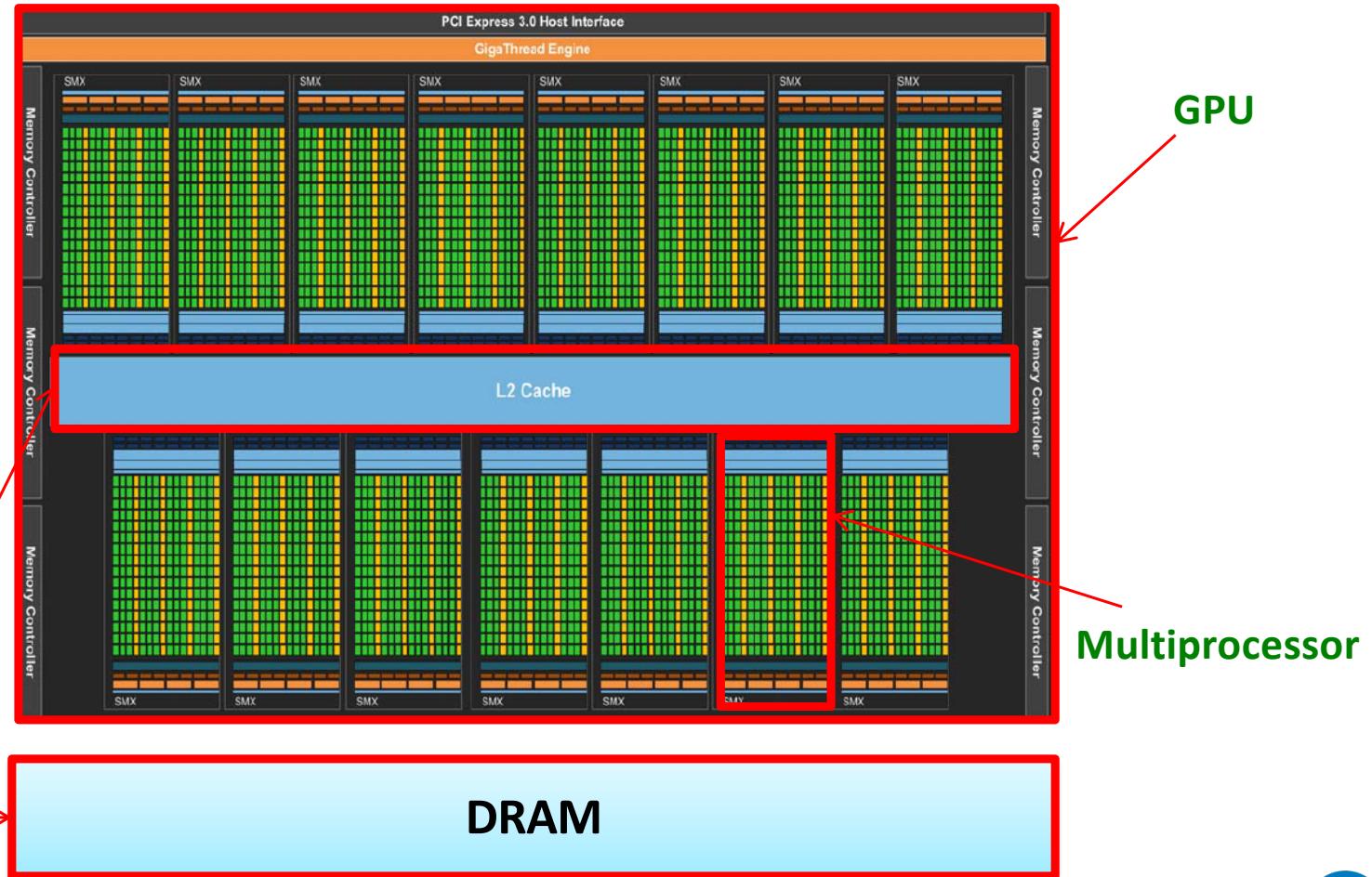
- ❑ Código paralelo
- ❑ Cálculo intensivo

¡Atención! Los anchos de banda están poco equilibrados: **CUELLO de BOTELLA**.

# Terminología CUDA

## Terminología CUDA

Host	✓
GPU	✓
Multiprocessor	✓
Scalar or CUDA core	
Global or Device Memory	✓
Shared Memory (per block)	
Local Memory (registers)	
Thread Block	
Thread	
Warp	
Grid	



# Terminología CUDA

## Terminología CUDA

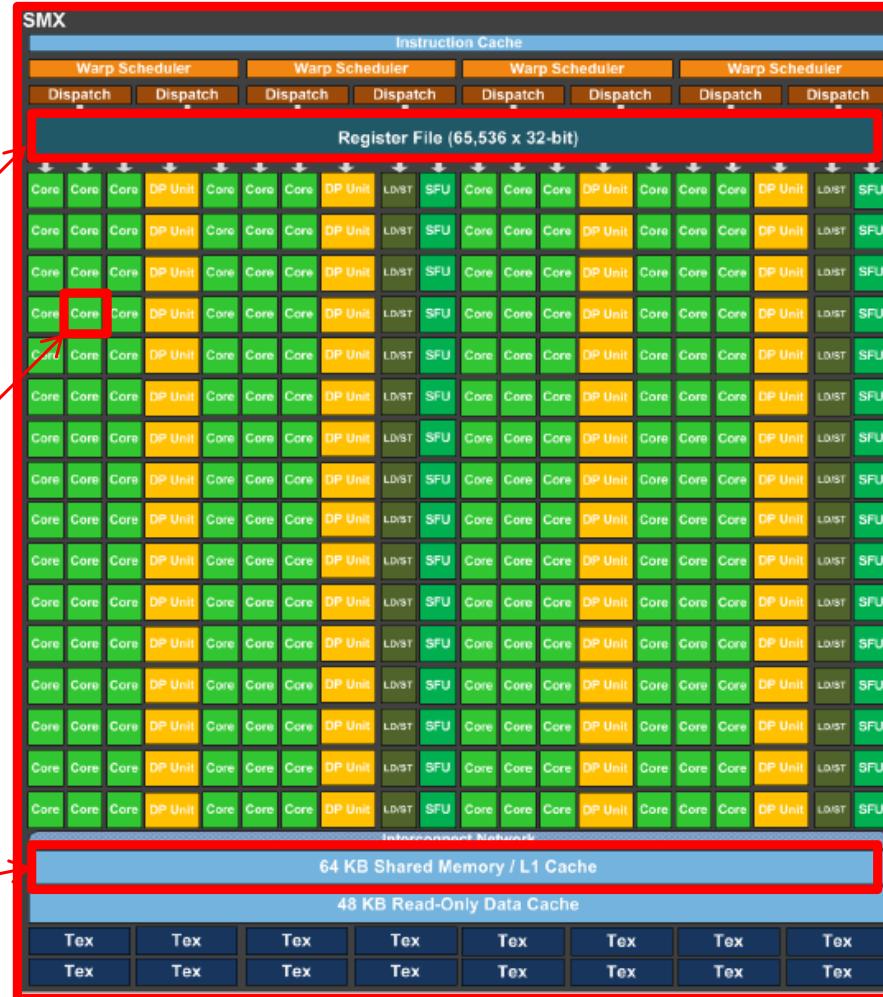
Host	✓
GPU	✓
Multiprocessor	✓
Scalar or CUDA core	✓
Global or Device Memory	✓
Shared Memory (per block)	✓
Local Memory (registers)	✓
Thread Block	
Thread	
Warp	
Grid	

Shared Memory

Local Memory

CUDA core

Multiprocessor



# Terminología CUDA

## Terminología CUDA

Host	✓
GPU	✓
Multiprocessor	✓
Scalar or CUDA core	✓
Global or Device Memory	✓
Shared Memory (per block)	✓
Local Memory (registers)	✓
Thread Block	✓
Thread	✓
Warp	✓
Grid	✓

**Thread**  
 (CUDA core)  
  
**Warp**  
 (32 CUDA cores)

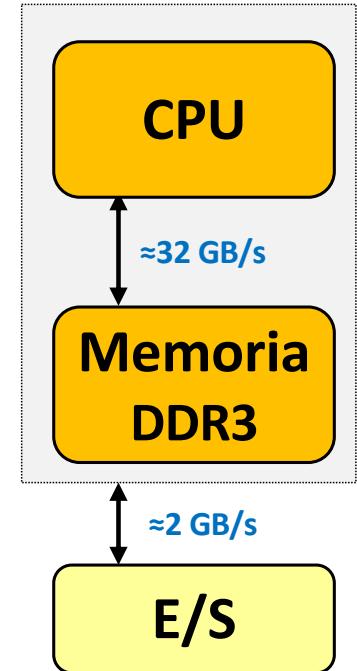


**Thread Block**  
 (Multiprocessor)

# Saxpy Secuencial

```
void saxpyS (int N, float a, float *x, float *y) {  
    for (int i=0; i<N; i++)  
        y[i] = a * x[i] + y[i];  
}  
  
.  
.  
.  
// Obtener Memoria  
float* Vx = (float*) malloc(N * sizeof(float));  
float* Vy = (float*) malloc(N * sizeof(float));  
// Inicializar vectores  
// Invocación  
saxpyS(N, 3.5, Vx, Vy);  
// Guardar/Mostrar/Procesar Resultados  
// Liberar Memoria  
free(Vx);  
free(Vy);
```

HOST



iGestión de errores!

iExpectativas de Rendimiento!

# Saxpy Secuencial

## Gestión de errores

```
// Obtener Memoria
float* Vx = (float*) malloc(N * sizeof(float));
if (Vx == NULL) {
    /* Error: tomar medidas necesarias */
    exit(0);
}

. . .

// Liberar Memoria
free(Vx);
Vx = NULL;
```

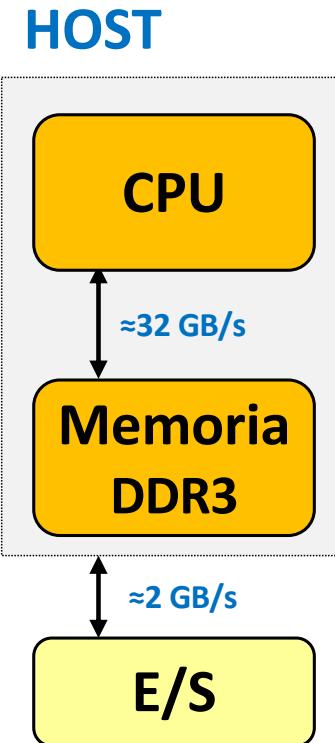
□ No controlar errores es una gran fuente de problemas

# Saxpy Secuencial

## Expectativas de Rendimiento

```
void saxpyS (int N, float a, float *x, float *y) {  
    for (int i=0; i<N; i++)  
        y[i] = a * x[i] + y[i];  
}
```

- Si sólo tenemos en cuenta el cálculo en coma flotante, la CPU del servidor de prácticas tiene 6 cores con capacidad vectorial AVX a 2.1 GHz, tiene una potencia de pico aproximada de **200 GFLOPs**.
- Si sólo contamos las lecturas de **x[i]** e **y[i]**, rendimiento máximo: **8 GFLOPs** (suponiendo 32 GB/s ancho de banda, 4 bytes por float, necesitamos 4 bytes para cada operación en CF)
- Si hay que incluir el coste de la E/S el rendimiento todavía bajará más.



# La gran idea que hay detrás de CUDA

```
void saxpyS (int N, float a, float *x, float *y) {  
    for (int i=0; i<N; i++)  
        y[i] = a * x[i] + y[i];  
}  
  
// Invocación  
saxpyS(N, 3.5, x, y);
```

Código Secuencial

Describe **TODO** lo que hay que hacer.

```
_global_  
void saxpyP (int N, float a, float *x, float *y) {  
    int i = blockIdx.x*blockDim.x + threadIdx.x;  
    if (i<N) y[i] = a * x[i] + y[i];  
}  
  
// Invocación (con 64 threads por bloque)  
int nblocks = (N+63)/64;  
saxpyP<<<nblocks,64>>>(N, 3.5, x, y);
```

Código CUDA

Indica **quién** soy.

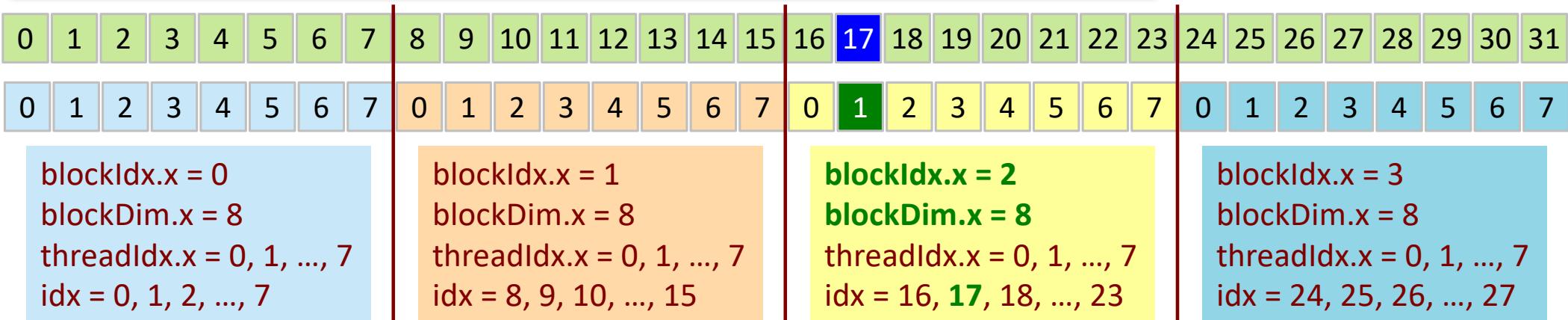
Describe cómo se calcula **1 elemento** del vector:  $y[i]$ . Es lo que se ejecuta en un thread.

# La gran idea que hay detrás de CUDA

```
__global__  
void saxpyP (int N, float a, float *x, float *y) {  
    int idx = blockIdx.x*blockDim.x + threadIdx.x;  
    if (idx<N) y[idx] = a * x[idx] + y[idx]; }  
saxpyP<<<4, 8>>>(32, 3.5, x, y);
```

Indica **quién soy**.

Invoca el kernel con  
4 Blocks y 8  
Threads por Block



int idx = (blockIdx.x \* blockDim.x) + threadIdx.x  
Obtiene en índice global (**idx**) a partir del índice local (**threadIdx.x**)

Mismo patrón de acceso  
para todos los threads.

# La gran idea que hay detrás de CUDA

```
__global__  
void saxpyP (int N, float a, float *x, float *y) {  
    int i = blockIdx.x*blockDim.x + threadIdx.x;  
    if (i<N) y[i] = a * x[i] + y[i];  
}
```

// Invocación (con 64 threads por bloque)

```
int nblocks = (N+63)/64;
```

```
saxpyP<<<nblocks, 64>>>(N, 3.5, x, y);
```

Código CUDA

Código de 1 thread

Threads de 1 Block

Descripción del Grid: nblocks  
Blocks de 64 threads cada uno.

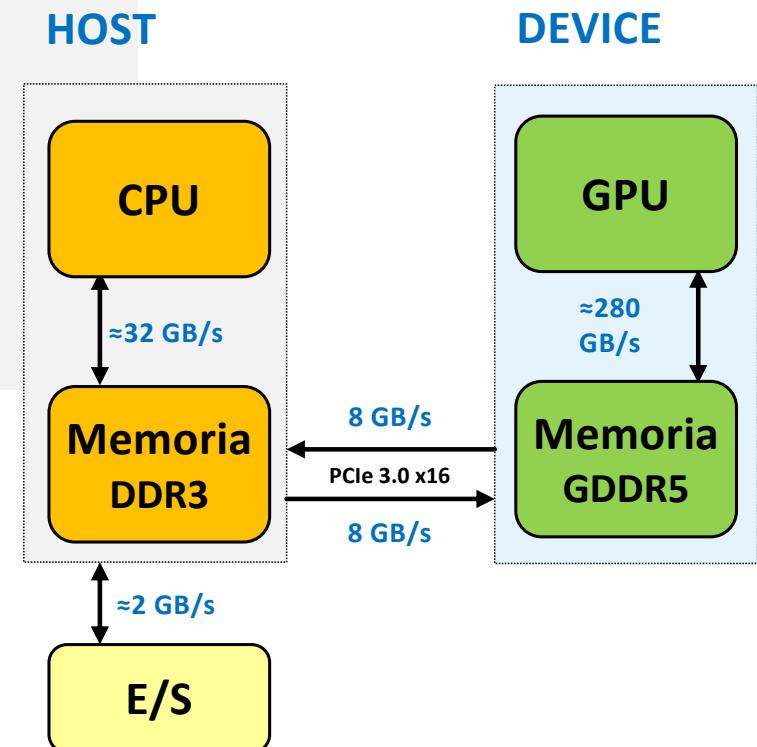
# Saxpy Paralela en CUDA

```
__global__
void saxpyP (int N, float a, float *x, float *y) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i<N) y[i] = a * x[i] + y[i];
}

// Invocación (con 64 threads por bloque)
int nblocks = (N+63)/64;
saxpyP<<<nblocks,64>>>(N, 3.5, x, y);
```

¡El Kernel se ejecuta en el DEVICE!

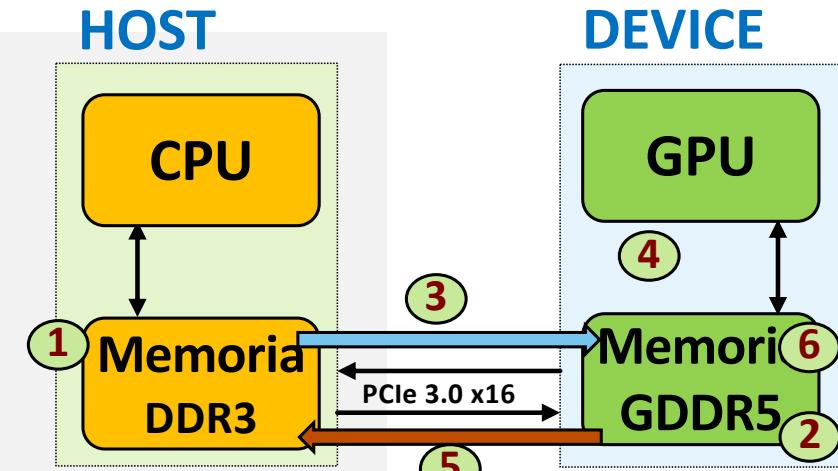
¡Los Datos están en HOST!



# ¿Cómo es el código en el Host?

```
1 // Obtener Memoria en el host  
2 // Obtener Memoria en el device  
3 // Copiar datos desde el host en el device  
4 // Ejecutar el kernel  
5 // Obtener el resultado desde el host  
6 // Liberar Memoria del device
```

```
unsigned int numBytes = N * sizeof(float);  
float* h_x = (float*) malloc(numBytes);  
float* h_y = (float*) malloc(numBytes);  
  
float* d_x, d_y;  
cudaMalloc((void**)&d_x, numBytes);  
cudaMalloc((void**)&d_y, numBytes);  
  
cudaMemcpy(d_x, h_x, numBytes, cudaMemcpyHostToDevice);  
cudaMemcpy(d_y, h_y, numBytes, cudaMemcpyHostToDevice);  
  
saxpyP<<<(N+63)/64, 64>>>(N, 3.5, d_x, d_y);  
  
cudaMemcpy(h_y, d_y, numBytes, cudaMemcpyDeviceToHost);  
  
cudaFree(d_x); cudaFree(d_y);
```



**h\_x VS d\_x**

**¡Gestión de errores!**

# Gestión de Errores en CUDA

- Todas las llamadas CUDA devuelven un código de error:

- Excepto la ejecución de kernels
  - Tipo: `cudaError_t`

- Siempre se puede utilizar la rutina:

`cudaError_t cudaGetLastError(void)`

- Devuelve el código del último error

- `char* cudaGetStringError(cudaError_t)`

- A partir del código de error devuelve un string con la descripción del error

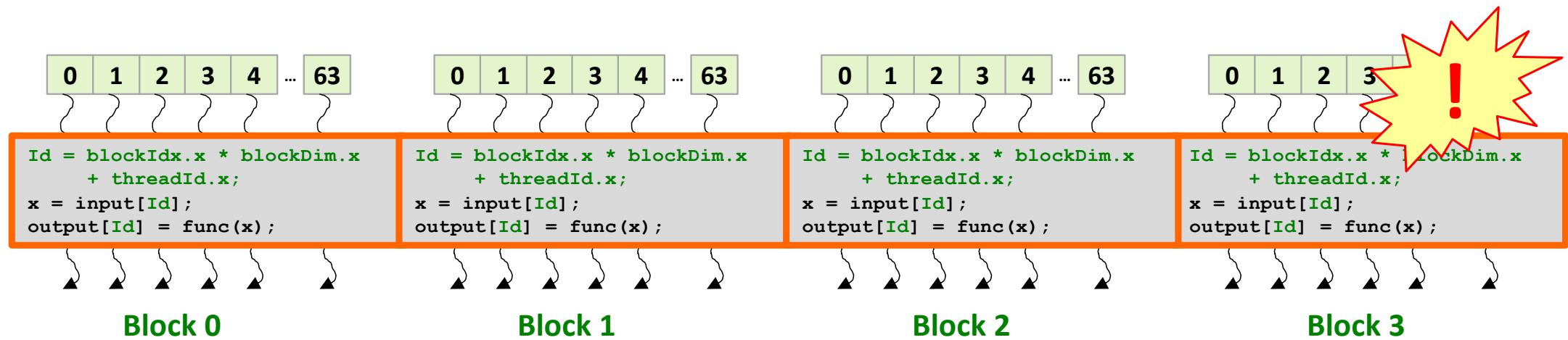
`printf("%s\n", cudaGetStringError(cudaGetLastError()));`

- ✓ NO SIEMPRE DA INFORMACIÓN ÚTIL

# Dimensiones, múltiplos y restos

## ☐ El tamaño del problema, no tiene porque adaptarse al tamaño de bloque.

- Por ejemplo: N = 250 con 4 blocks de 64 threads (blockDim.x = 64)
- El último block tendrá 6 threads sin datos de entrada (lo más fácil para un **!segmentation fault!**)



- Hay que asegurarse que operar sin datos no dará problemas:
  - ✓ Aumentando el tamaño de los datos hasta N = 256 [**No siempre es posible**]
  - ✓ Usando un condicional: `if (Id < N) ...` [**COSTOSO**]

# Dimensiones, múltiplos y restos

```
__global__
void saxpyP (int N, float a, float *x, float *y {
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i<N) y[i] = a * x[i] + y[i];
}
//Invocación (con 64 threads por bloque)
int nblocks = (N+63)/64;
saxpyP<<<nblocks,64>>>(N, 3.5, x, y);
```

Código para cualquier valor de N

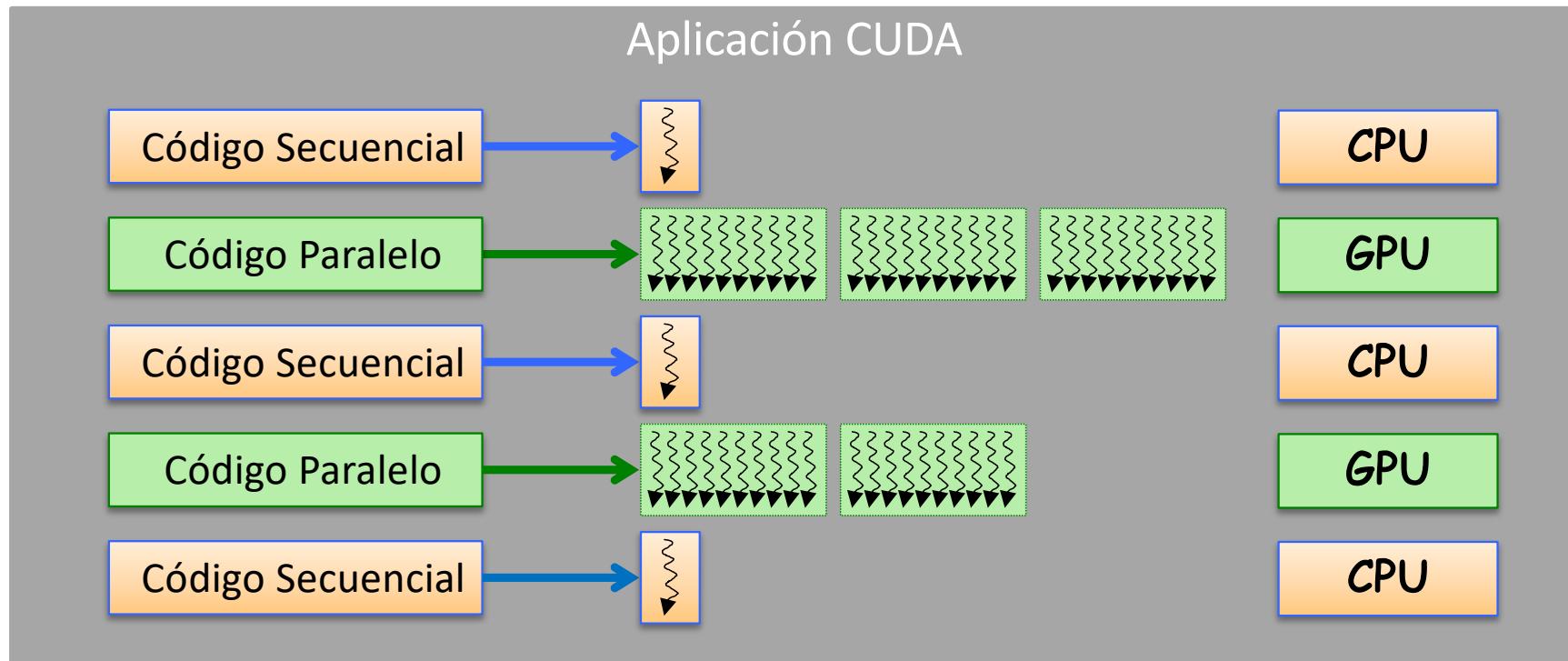
El condicional se ejecuta en TODOS los threads

```
__global__
void saxpyP (int N, float a, float *x, float *y) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    y[i] = a * x[i] + y[i];
}
//Invocación (con 64 threads por bloque)
int nblocks = N/64;
saxpyP<<<nblocks,64>>>(N, 3.5, x, y);
```

Código para un valor de N múltiplo de 64

# Modelo de Programación CUDA

- Programas secuenciales con kernels paralelos (todos ellos en C, C++):
  - Código secuencial en un host (CPU)
  - Kernels paralelos ejecutados en múltiples elementos de proceso (GPU)



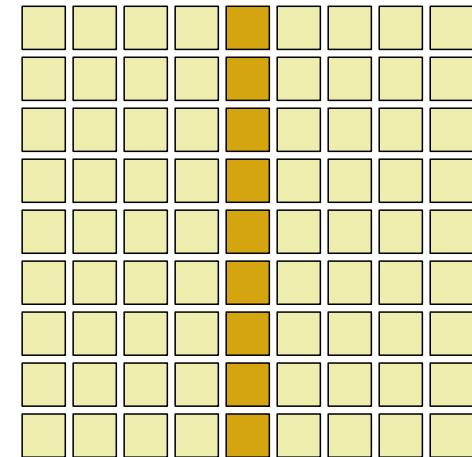
# Producto de Matrices

$$C = A \cdot B \text{ (tamaño } N \times N\text{)}$$

- Algoritmo fundamental.
- Es la base de muchos algoritmos numéricos.
- Para calcular el elemento  $C[i][j]$  se necesita la **fila i** de la matriz A y la **columna j** de la matriz B.
- Por simplicidad usamos matrices cuadradas ( $N \times N$ )
- El algoritmo secuencial es muy conocido

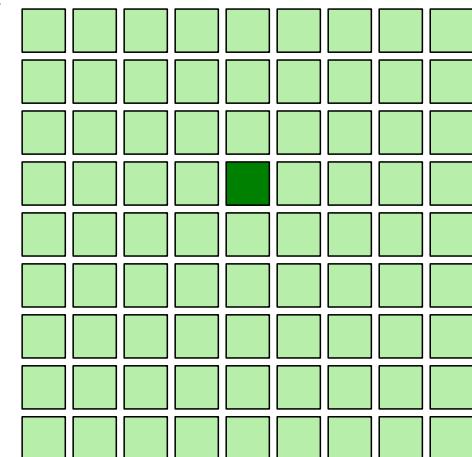
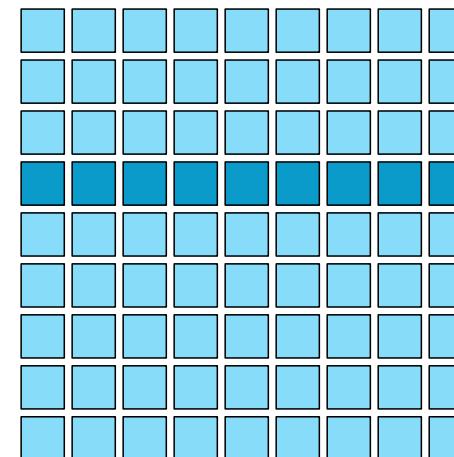
```
for (i=0; i<N; i++)  
    for (j=0; j<N; j++) {  
        tmp = 0.0;  
        for (k=0; k<N; k++)  
            tmp += A[i][k] * B[k][j];  
        C[i][j] = tmp;  
    }
```

$$B (N \times N)$$



$$C = A \cdot B$$

$$A (N \times N)$$



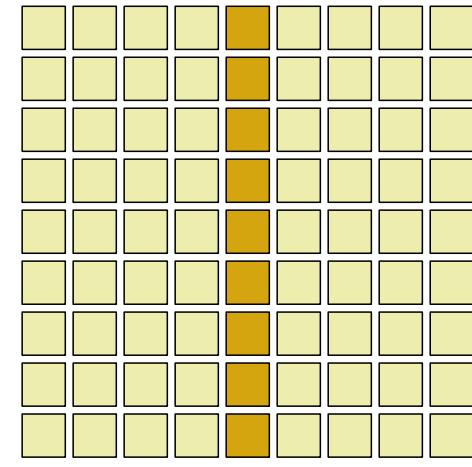
# Producto de Matrices

$$C = A \cdot B \text{ (tamaño } N \times N\text{)}$$

- Algoritmo fundamental.
- $2 \cdot N^3$  ops en CF,  $2 \cdot N^3 + N^2$  accesos a memoria
- Es la base de muchos algoritmos numéricos.
- El primer paso es convertir los accesos a las matrices, en accesos a vectores, para poder trabajar con punteros.

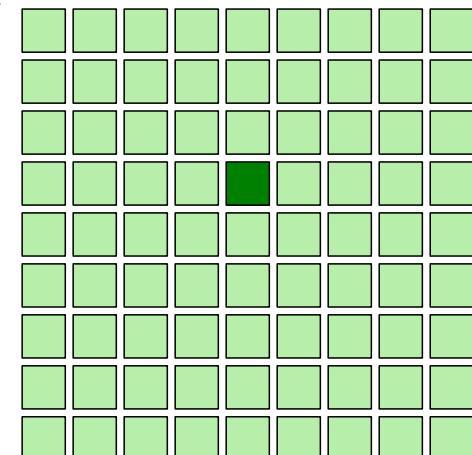
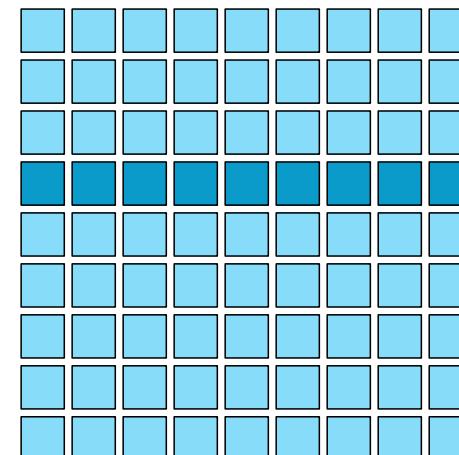
```
for (i=0; i<N; i++)  
    for (j=0; j<N; j++) {  
        tmp = 0.0;  
        for (k=0; k<N; k++)  
            tmp += A[i*N+k] * B[k*N+j];  
        C[i*N+j] = tmp;  
    }
```

$$B (N \times N)$$



$$C = A \cdot B$$

$$A (N \times N)$$



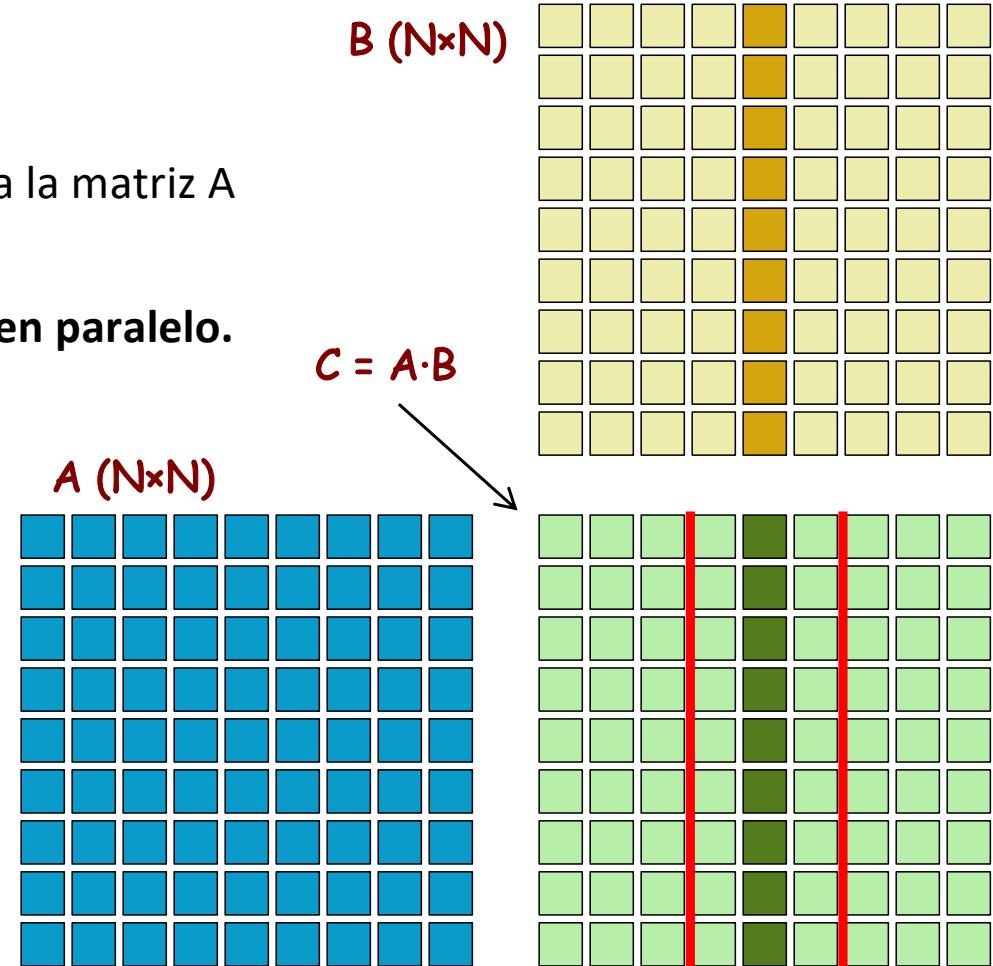
# Producto de Matrices por columnas

## Algoritmo Paralelo por columnas

- 1 Thread calcula la columna  $C[-] [j]$ .
- Para calcular la columna  $C[-] [j]$  se necesita toda la matriz A y la columna  $B[-] [j]$ .
- **Todas las columnas  $C[-] [j]$  se pueden calcular en paralelo.**
- Descomponemos el problema en N/S blocks.
- En cada block utilizaremos S threads.

```
dim3 dimGrid(N/S, 1, 1); // nBlocks  
dim3 dimBlock(S, 1, 1); // nThreads  
  
// Invocación  
MMCKernel<<<dimGrid, dimBlock>>>(...);
```

- Supondremos N múltiplo de S.



# Producto de Matrices por columnas

```
for (i=0; i<N; i++)  
    for (j=0; j<N; j++) {  
        tmp = 0.0;  
        for (k=0; k<N; k++)  
            tmp += A[i*N+k] * B[k*N+j];  
        C[i*N+j] = tmp;  
    }
```

```
dim3 dimGrid(N/S, 1, 1); // nBlocks  
dim3 dimBlock(S, 1, 1); // nThreads  
  
// Invocación  
MMCKernel<<<dimGrid, dimBlock>>>(...);
```

```
__global__ void MMCKernel(float *dA, float *dB, float *dC, int N) {  
  
    int col = blockIdx.x * blockDim.x + threadIdx.x;  
  
    for (int i=0; i<N; i++) {  
        float tmp = 0.0;  
        for (int k=0; k<N; k++)  
            tmp += dA[i*N+k] * dB[k*N+col];  
        dC[i*N+col] = tmp;  
    }  
}
```

Quién soy yo

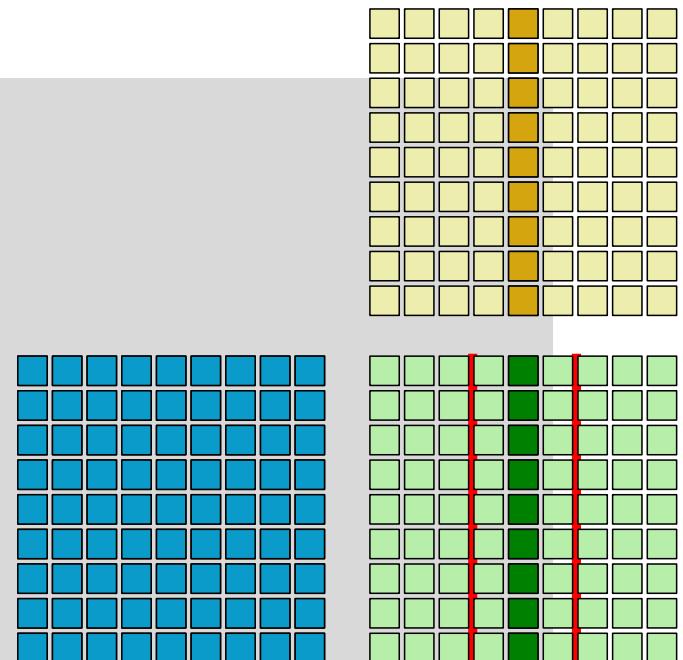
Qué hace cada thread

# Producto de Matrices por columnas

```
for (i=0; i<N; i++)  
    for (j=0; j<N; j++) {  
        tmp = 0.0;  
        for (k=0; k<N; k++)  
            tmp += A[i*N+k] * B[k*N+j];  
        C[i*N+j] = tmp;  
    }
```

```
dim3 dimGrid(N/S, 1, 1); // nBlocks  
dim3 dimBlock(S, 1, 1); // nThreads  
  
// Invocación  
MMCKernel<<<dimGrid, dimBlock>>>(...);
```

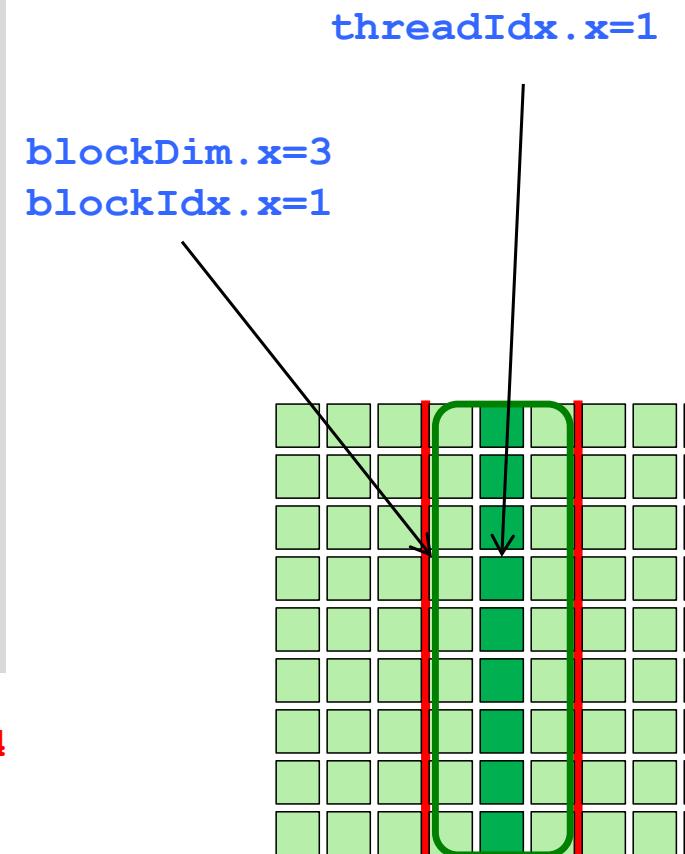
```
__global__ void MMCKernel(. . .) {  
  
    int col = blockIdx.x * blockDim.x + threadIdx.x;  
  
    for (int i=0; i<N; i++) {  
        float tmp = 0.0;  
        for (int k=0; k<N; k++)  
            tmp += dA[i*N+k] * dB[k*N+col];  
        dC[i*N+col] = tmp;  
    }  
}
```



# Producto de Matrices por columnas

```
__global__ void MMCKernel( . . . ) {  
  
    int col = blockIdx.x * blockDim.x + threadIdx.x;  
    for (int i=0; i<N; i++)  
        float tmp = 0.0;  
        for (int k=0; k<N; k++)  
            tmp += dA[i*N+k] * dB[k*N+col];  
        dC[i*N+col] = tmp;  
    }  
}  
  
int N = 9;  
dim3 dimGrid(N/3, 1, 1); // nBlocks  
dim3 dimBlock(3, 1, 1); // nThreads  
// Invocación  
MMCKernel<<<dimGrid, dimBlock>>>(...);
```

$$\text{col} = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x} = 1 * 3 + 1 = 4$$



# Lenguaje CUDA: C con extensiones mínimas

- Especificadores que indican dónde están las cosas

```
__global__ void MMK(...); // kernel que se ejecuta en el device
```

- Extensión que permite invocar el kernel paralelo **MMK**

```
MMK<<<500, 128>>>(...); // lanza 500 blocks con 128 threads cada uno
```

- Variables especiales PREDEFINIDAS para identificar los threads

```
dim3 threadIdx; // número de thread dentro del block
```

```
dim3 blockIdx; // número de block dentro del grid
```

```
dim3 blockDim; // Dimensiones del block
```

```
dim3 gridDim; // Dimensiones del grid
```

- **dim3**, tipo predefinido, equivale a 1 vector de 3 elementos,

- Uso de estas variables: **threadIdx.x**, **blockIdx.y**, **blockDim.z**, **threadIdx[0]**

# blockIdx & threadIdx

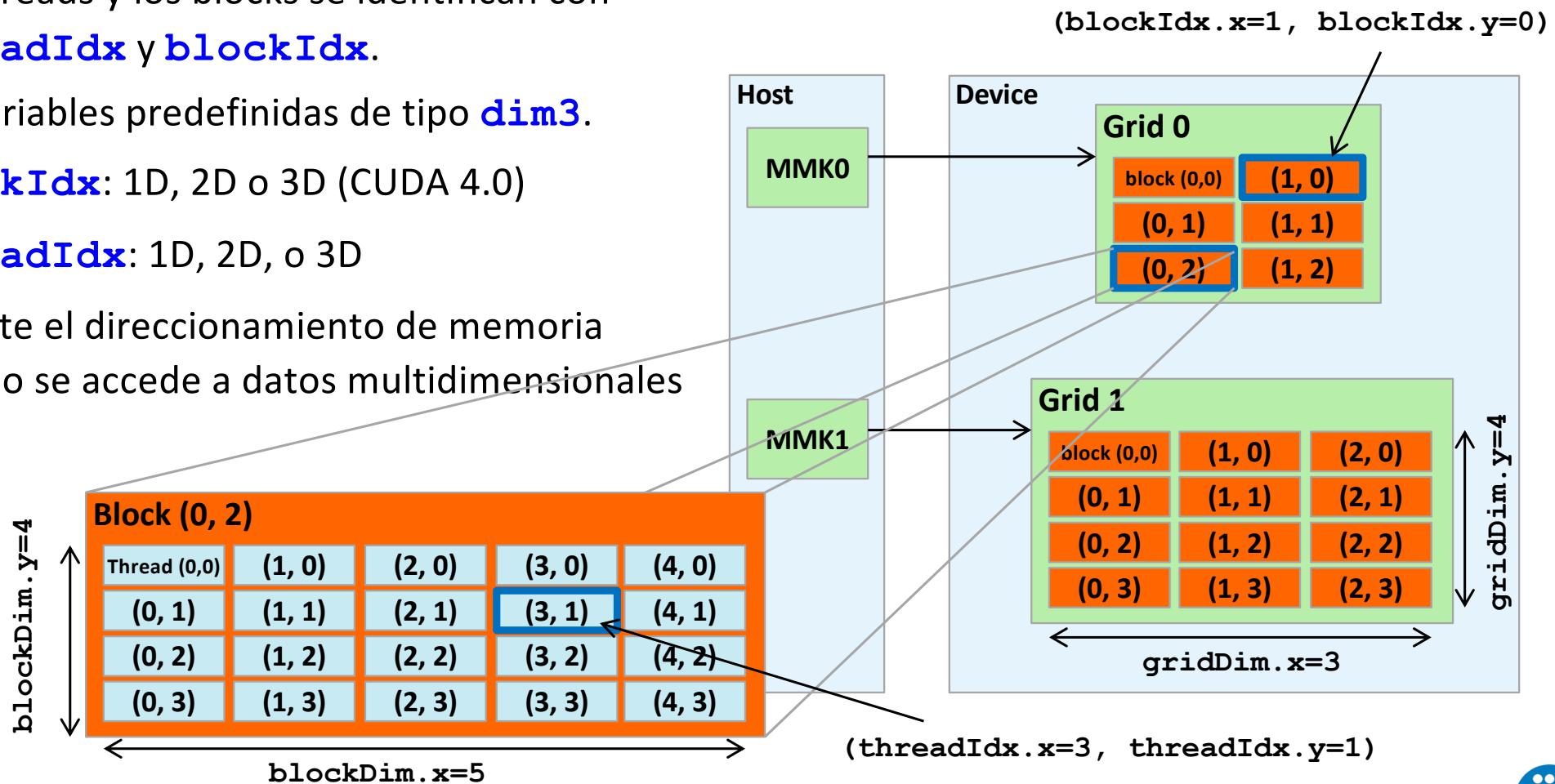
- Los threads y los blocks se identifican con **threadIdx** y **blockIdx**.

- Variables predefinidas de tipo **dim3**.

- blockIdx**: 1D, 2D o 3D (CUDA 4.0)

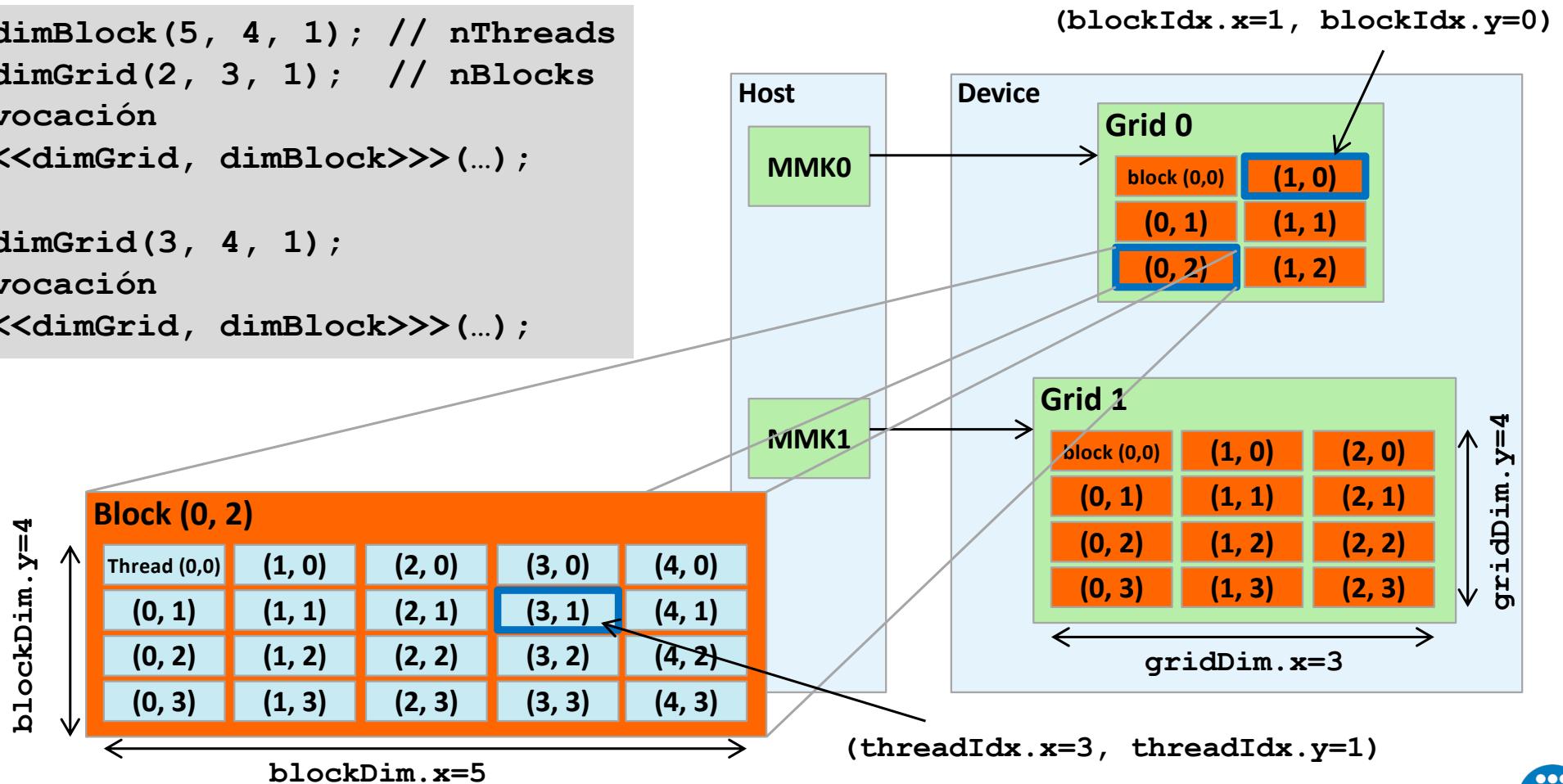
- threadIdx**: 1D, 2D, o 3D

- Permite el direccionamiento de memoria cuando se accede a datos multidimensionales



# blockIdx & threadIdx

```
dim3 dimBlock(5, 4, 1); // nThreads  
dim3 dimGrid(2, 3, 1); // nBlocks  
// Invocación  
MMK0<<<dimGrid, dimBlock>>>(...);  
  
dim3 dimGrid(3, 4, 1);  
// Invocación  
MMK1<<<dimGrid, dimBlock>>>(...);
```



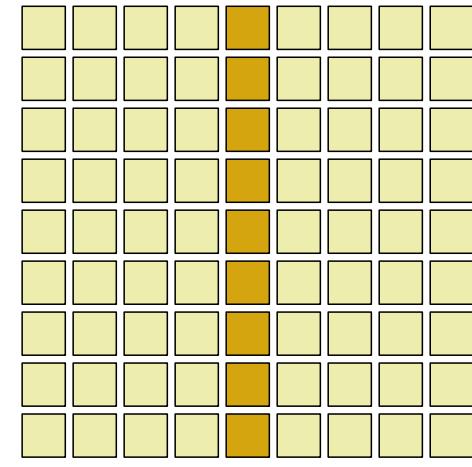
# Producto de Matrices

$$C = A \cdot B \text{ (tamaño } N \times N\text{)}$$

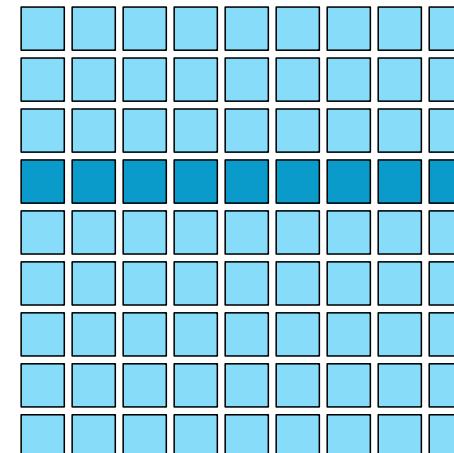
- Algoritmo fundamental.
- $2 \cdot N^3$  ops en CF,  $2 \cdot N^3 + N^2$  accesos a memoria
- Es la base de muchos algoritmos numéricos.
- El primer paso es convertir los accesos a las matrices, en accesos a vectores, para poder trabajar con punteros.

```
for (i=0; i<N; i++)  
    for (j=0; j<N; j++) {  
        tmp = 0.0;  
        for (k=0; k<N; k++)  
            tmp += A[i*N+k] * B[k*N+j];  
        C[i*N+j] = tmp;  
    }
```

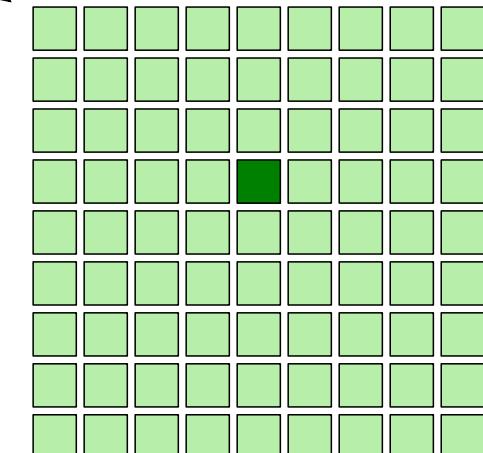
$$B (N \times N)$$



$$A (N \times N)$$



$$C = A \cdot B$$



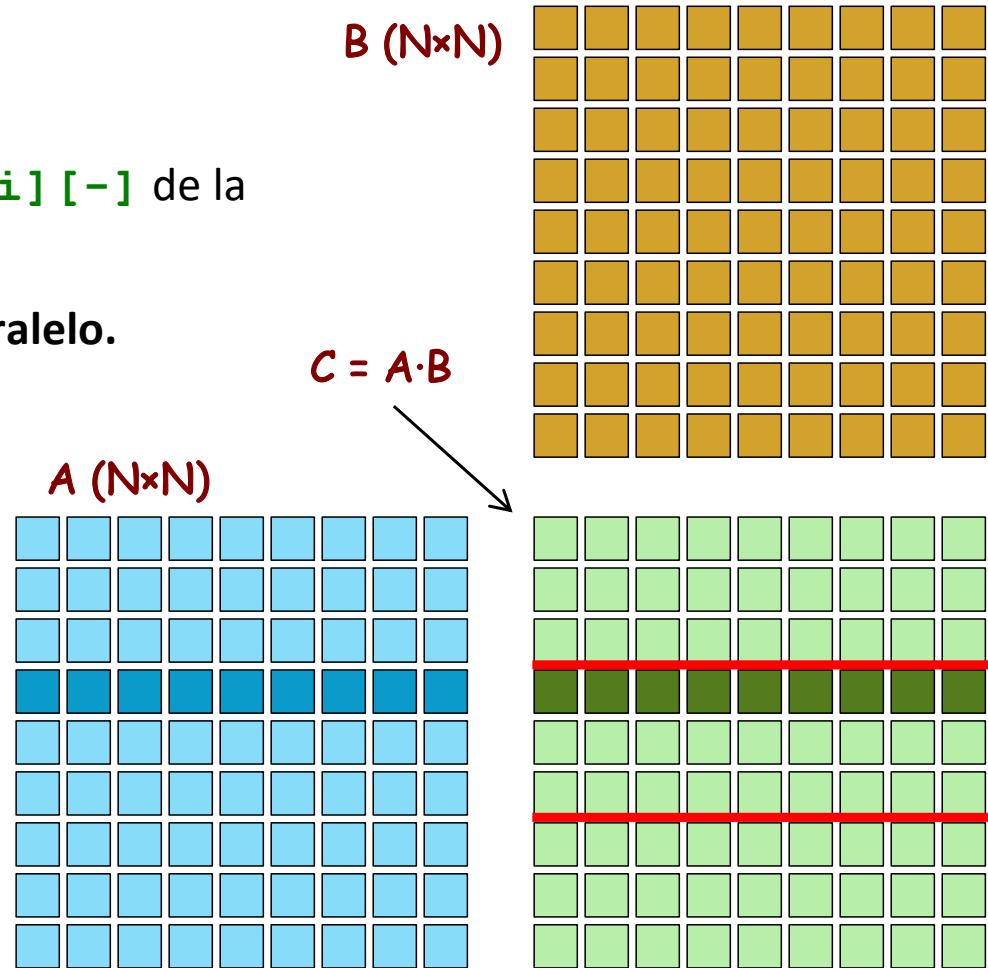
# Producto de Matrices por filas

## Algoritmo Paralelo por filas

- 1 Thread calcula la fila  $C[i] [-]$ .
- Para calcular la fila  $C[i] [-]$  se necesita la fila  $A[i] [-]$  de la matriz A y toda la matriz B.
- Todas las filas  $C[i] [-]$  se pueden calcular en paralelo.**
- Descomponemos el problema en N/S blocks.
- En cada block utilizaremos S threads.

```
dim3 dimGrid(1, N/S, 1); // nBlocks  
dim3 dimBlock(1, S, 1); // nThreads  
  
// Invocación  
MMFKernel<<<dimGrid, dimBlock>>>(...);
```

- Supondremos N múltiplo de S.



# Producto de Matrices por filas

```
for (i=0; i<N; i++)  
    for (j=0; j<N; j++) {  
        tmp = 0.0;  
        for (k=0; k<N; k++)  
            tmp += A[i*N+k] * B[k*N+j];  
        C[i*N+j] = tmp;  
    }
```

```
dim3 dimGrid(1, N/S, 1); // nBlocks  
dim3 dimBlock(1, S, 1); // nThreads  
  
// Invocación  
MMFkernel<<<dimGrid, dimBlock>>>(...);
```

```
__global__ void MMFkernel(float *dA, float *dB, float *dC, int N) {  
  
    int fil = blockIdx.y * blockDim.y + threadIdx.y;  
  
    for (int j=0; j<N; j++) {  
        float tmp = 0.0;  
        for (int k=0; k<N; k++)  
            tmp += dA[fil*N+k] * dB[k*N+j];  
        dC[fil*N+j] = tmp;  
    }  
}
```

Quién soy yo

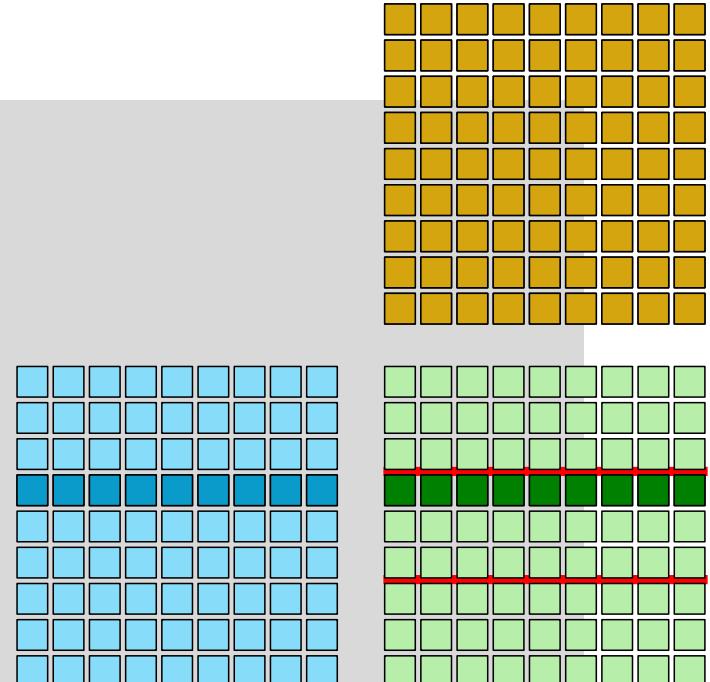
Qué hace cada thread

# Producto de Matrices por filas

```
for (i=0; i<N; i++)  
    for (j=0; j<N; j++) {  
        tmp = 0.0;  
        for (k=0; k<N; k++)  
            tmp += A[i*N+k] * B[k*N+j];  
        C[i*N+j] = tmp;  
    }
```

```
dim3 dimGrid(1, N/S, 1); // nBlocks  
dim3 dimBlock(1, S, 1); // nThreads  
  
// Invocación  
MMFkernel<<<dimGrid, dimBlock>>>(...);
```

```
__global__ void MMFkernel(. . .) {  
  
    int fil = blockIdx.y * blockDim.y + threadIdx.y;  
  
    for (int j=0; j<N; j++) {  
        float tmp = 0.0;  
        for (int k=0; k<N; k++)  
            tmp += dA[fil*N+k] * dB[k*N+j];  
        dC[fil*N+j] = tmp;  
    }  
}
```



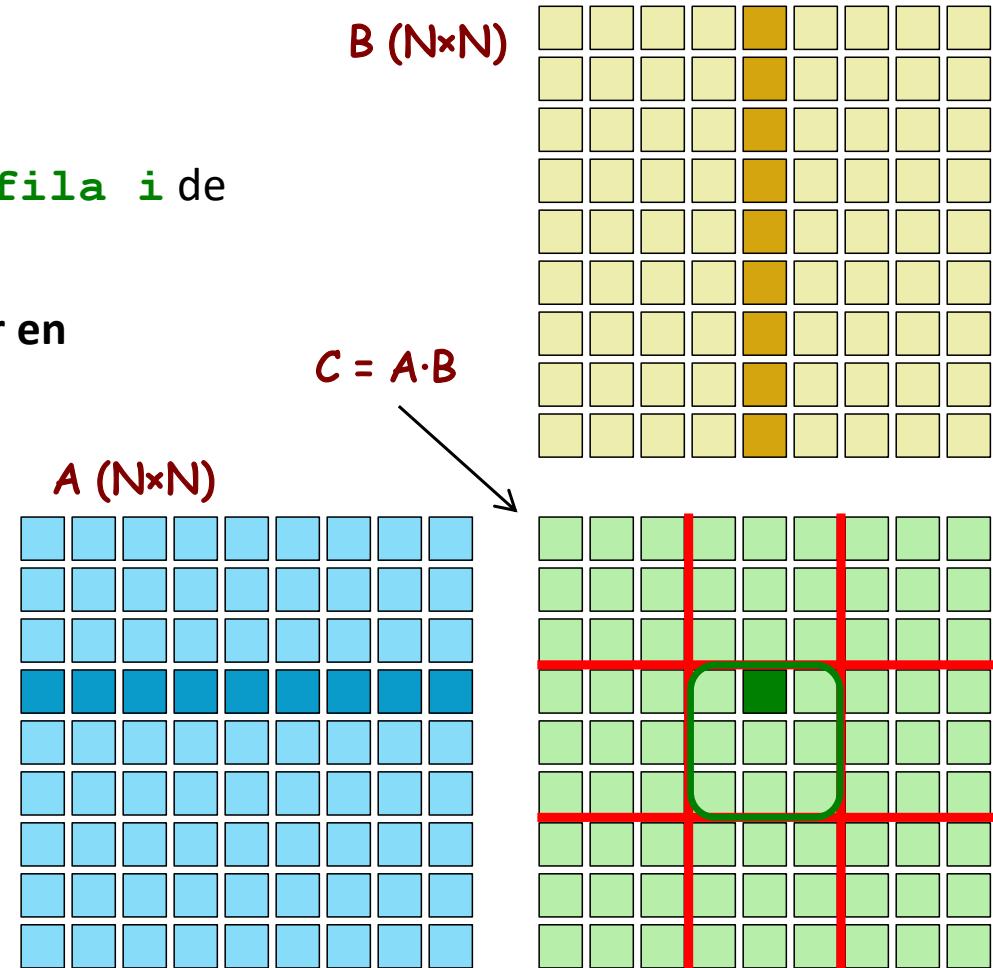
# Producto de Matrices 2D

## Algoritmo Paralelo

- 1 Thread calcula el elemento  $C[i][j]$ .
- Para calcular el elemento  $C[i][j]$  se necesita la **fila i** de la matriz A y la **columna j** de la matriz B.
- **Todos los elementos  $C[i][j]$  se pueden calcular en paralelo.**
- Descomponemos el problema en  $B \times B$  blocks.
- En cada block utilizaremos  $S \times S$  threads.

```
dim3 dimGrid(N/S, N/S, 1); // nBlocks  
dim3 dimBlock(S, S, 1);    // nThreads  
  
// Invocación  
MMkernel<<<dimGrid, dimBlock>>>(...);
```

- Supondremos N múltiplo de S.



# Producto de Matrices 2D

```
for (i=0; i<N; i++)  
    for (j=0; j<N; j++) {  
        tmp = 0.0;  
        for (k=0; k<N; k++)  
            tmp += A[i*N+k] * B[k*N+j];  
        C[i*N+j] = tmp;  
    }
```

```
dim3 dimGrid(N/S, N/S, 1); // nBlocks  
dim3 dimBlock(S, S, 1); // nThreads  
  
// Invocación  
MMkernel<<<dimGrid, dimBlock>>>(...);
```

```
__global__ void MMkernel(float *dA, float *dB, float *dC, int N) {
```

```
    int row = blockIdx.y * blockDim.y + threadIdx.y;  
    int col = blockIdx.x * blockDim.x + threadIdx.x;
```

```
    float tmp = 0.0;  
    for (int k=0; k<N; k++)  
        tmp += dA[row*N+k] * dB[k*N+col];  
    dC[row*N+col] = tmp;  
}
```

Quién soy yo

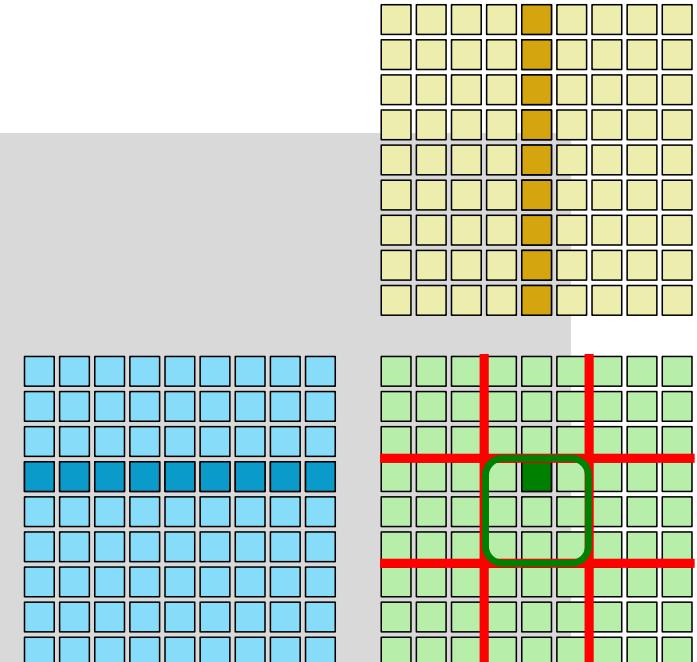
Qué hace cada thread

# Producto de Matrices 2D

```
for (i=0; i<N; i++)  
    for (j=0; j<N; j++) {  
        tmp = 0.0;  
        for (k=0; k<N; k++)  
            tmp += A[i*N+k] * B[k*N+j];  
        C[i*N+j] = tmp;  
    }
```

```
__global__ void MMkernel(. . .) {  
  
    int row = blockIdx.y * blockDim.y + threadIdx.y;  
    int col = blockIdx.x * blockDim.x + threadIdx.x;  
  
    float tmp = 0.0;  
    for (int k=0; k<N; k++)  
        tmp += dA[row*N+k] * dB[k*N+col];  
    dC[row*N+col] = tmp;  
}
```

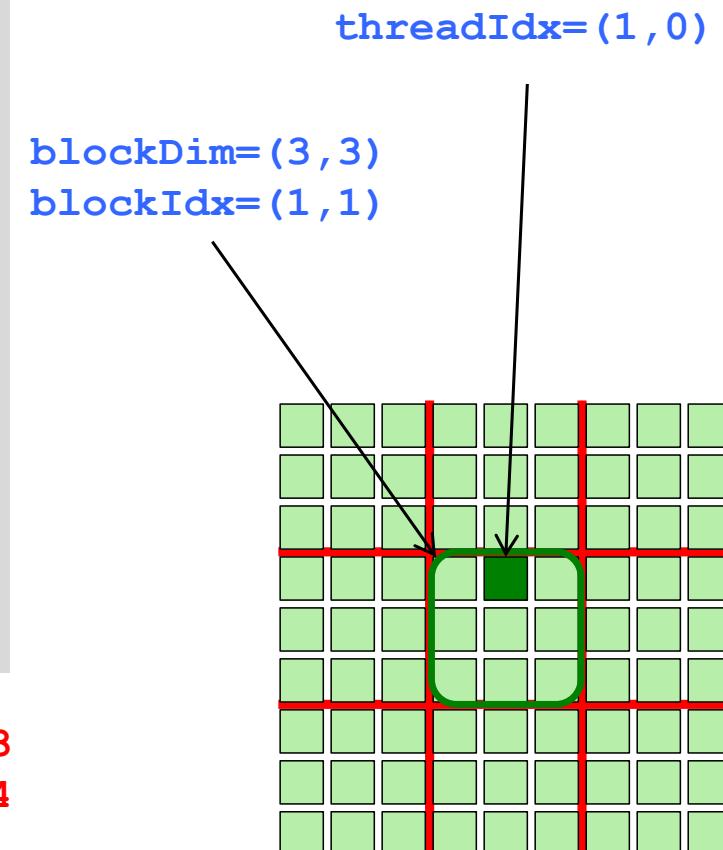
```
dim3 dimGrid(N/S, N/S, 1); // nBlocks  
dim3 dimBlock(S, S, 1); // nThreads  
  
// Invocación  
MMkernel<<<dimGrid, dimBlock>>>(...);
```



# Producto de Matrices 2D

```
__global__ void MMkernel(. . .) {  
  
    int row = blockIdx.y * blockDim.y + threadIdx.y;  
    int col = blockIdx.x * blockDim.x + threadIdx.x;  
  
    float tmp = 0.0;  
    for (int k=0; k<N; k++)  
        tmp += dA[row*N+k] * dB[k*N+col];  
    dC[row*N+col] = tmp;  
}  
  
int N = 9;  
dim3 dimGrid(N/3, N/3, 1); // nBlocks  
dim3 dimBlock(3, 3, 1); // nThreads  
// Invocación  
MMkernel<<<dimGrid, dimBlock>>>(...);
```

row = blockIdx.y\*blockDim.y+threadIdx.y = 1\*3+0 = 3  
col = blockIdx.x\*blockDim.x+threadIdx.x = 1\*3+1 = 4



# Evaluación del rendimiento

## □ Medimos el rendimiento en el código del Host

```
// Obtener Memoria en el host  
// Inicializa las matrices  
// Obtener Memoria en el device
```

```
// Copiar datos desde el host en el device  
cudaMemcpy(d_A, h_A, numBytes, cudaMemcpyHostToDevice);  
cudaMemcpy(d_B, h_B, numBytes, cudaMemcpyHostToDevice);
```

```
// Ejecutar el kernel  
MMxKernel<<<dimGrid, dimBlock>>>(...);
```

```
// Obtener el resultado desde el host  
cudaMemcpy(h_C, d_C, numBytes, cudaMemcpyDeviceToHost);
```

```
dim3 dimGrid(2048/128, 1, 1);  
dim3 dimBlock(128, 1, 1);  
...  
MMCkernel<<<...>>>(...);
```

```
dim3 dimGrid(1, 2048/128, 1);  
dim3 dimBlock(1, 128, 1);  
...  
MMFkernel<<<...>>>(...);
```

```
dim3 dimGrid(2048/32, 2048/32, 1);  
dim3 dimBlock(32, 32, 1);  
...  
MMkernel<<<...>>>(...);
```

Tiempo Kernel.  
**Medimos tiempo de ejecución del kernel.**

Tiempo Global.  
**Medimos tiempo de ejecución del kernel y las transferencias de datos.**

# Evaluación del rendimiento

- Hemos ejecutado los 3 kernels en una tarjeta GeForce GTX1080ti.

Kernel	Por Columnas	Por Filas	2D
Dimensión problema	2048×2048	2048×2048	2048×2048
Número de Blocks	16	16	64×64 (4096)
Número de Threads por Block	128	128	32×32 (1024)
Tiempo Global	366,32 ms	512,72 ms	56.77 ms
Tiempo Kernel	356,42 ms	502,82 ms	45,19 ms
Rendimiento Global	46,90 GFLOPS	33,51 GFLOPS	302,62 GFLOPS
Rendimiento Kernel	48,20 GFLOPS	34,17 GFLOPS	380,13 GFLOPS

- ¿Cómo explicamos la diferencia de rendimientos?
- ¿Porqué baja tanto el rendimiento global en el caso 2D?
- ¿Estamos cerca del rendimiento óptimo?

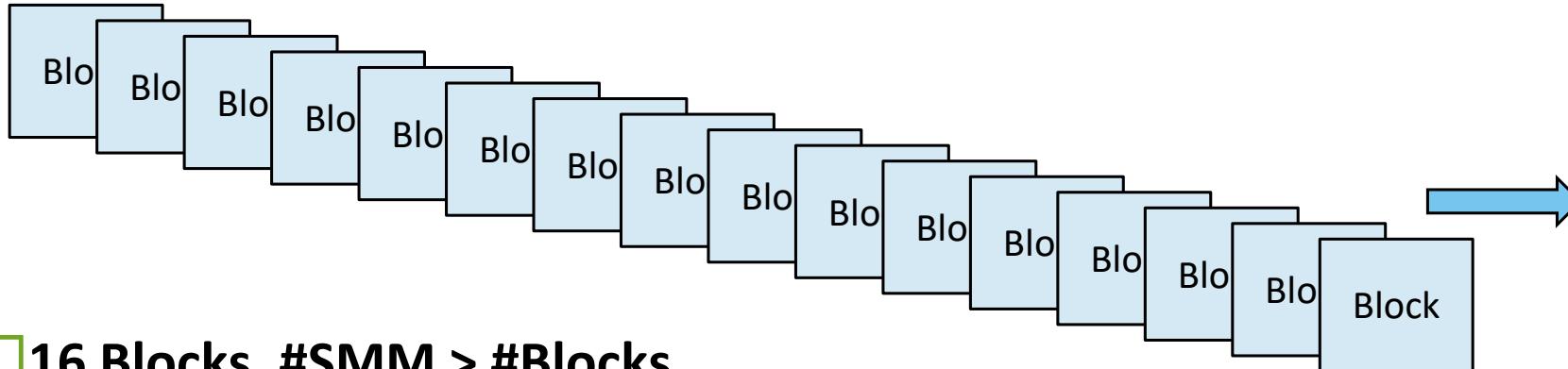
Con el código de la sesión  
4-5 de laboratorio se  
obtienen 1.475 GFLOPs

# Recursos hardware de una GPU

- Rutina `cudaGetDeviceProperties`, código disponible en la distribución de CUDA.

```
Device 0: "GeForce GTX 1080 Ti"
  CUDA Driver Version / Runtime Version      8.0 / 8.0
  CUDA Capability Major/Minor version number: 6.1
  Total amount of global memory:             11158 MBytes (11700207616 bytes)
  (28) Multiprocessors, (128) CUDA Cores/MP: 3584 CUDA Cores
  GPU Max Clock rate:                      1582 MHz (1.58 GHz)
  Memory Clock rate:                       5505 Mhz
  Memory Bus Width:                        352-bit
  .
  .
  Total amount of constant memory:          65536 bytes
  Total amount of shared memory per block:   49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                32
  Maximum number of threads per multiprocessor: 2048
  Maximum number of threads per block:        1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                     2147483647 bytes
  Concurrent copy and kernel execution:     Yes with 2 copy engine(s)
  Run time limit on kernels:                Yes
  .
  .
```

# Evaluación del rendimiento (por filas/columnas)



GTX 1080Ti

□ 16 Blocks, #SMM > #Blocks

□ 128 threads por Block

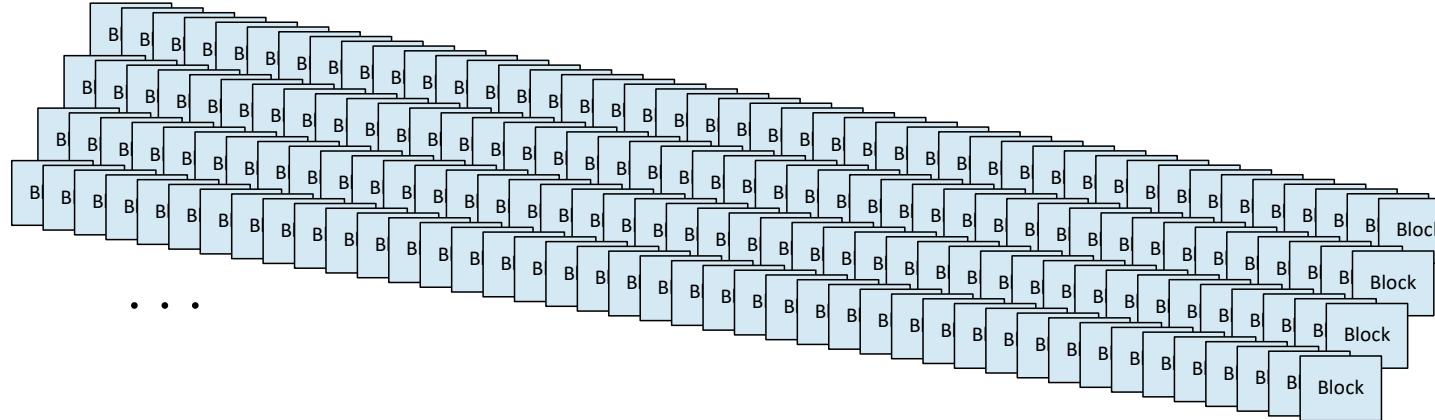
□ Trabajo de 1 Thread:

- $2 \cdot 2048^2$  ops de coma flotante
- $2 \cdot 2048^2 + 2048$  accesos a memoria
- Como máximo: 2.048 threads activos en total

Una parte importante de la GPU queda inactiva.  
Poco paralelismo, NO podremos ocultar la latencia de los accesos a memoria

28 SMM  
128 cores/SMM  
3584 cores

# Evaluación del rendimiento (2D)



GTX 1080Ti

28 SMM  
128 cores/SMM  
3584 cores

□ 4.096 Blocks, #Blocks >> #SMM

□ 1.024 threads por Block

□ Trabajo de 1 Thread:

- 2·2048 ops de coma flotante
- 2·2048+1 accesos a memoria

Podemos saturar todos los recursos de la GPU.  
Mucho paralelismo, SI podremos ocultar la latencia de los accesos a memoria

# Evaluación del rendimiento

- Los 3 kernels en una tarjeta NVIDIA K40c.

Kernel	Por Columnas	Por Filas	2D
Dimensión problema	2048×2048	2048×2048	2048×2048
Número de Blocks	16	16	64×64 (4096)
Número de Threads por Block	128	128	32×32 (1024)
Tiempo Global	1,24 s	3,79 s	150,21 ms
Tiempo Kernel	1,23 s	3,78 s	140,00 ms
Rendimiento Global	13,85 GFLOPS	4,53 GFLOPS	114,37 GFLOPS
Rendimiento Kernel	13,96 GFLOPS	4,54 GFLOPS	122,71 GFLOPS

- ¿Cómo explicamos la diferencia de rendimientos?
- ¿Porqué baja tanto el rendimiento global en el caso 2D?
- ¿Estamos cerca del rendimiento óptimo?

Con el código de la sesión  
4-5 de laboratorio se  
obtienen 376,5 GFLOPs

# Recursos hardware de una GPU

- Rutina `cudaGetDeviceProperties`, código disponible en la distribución de CUDA.

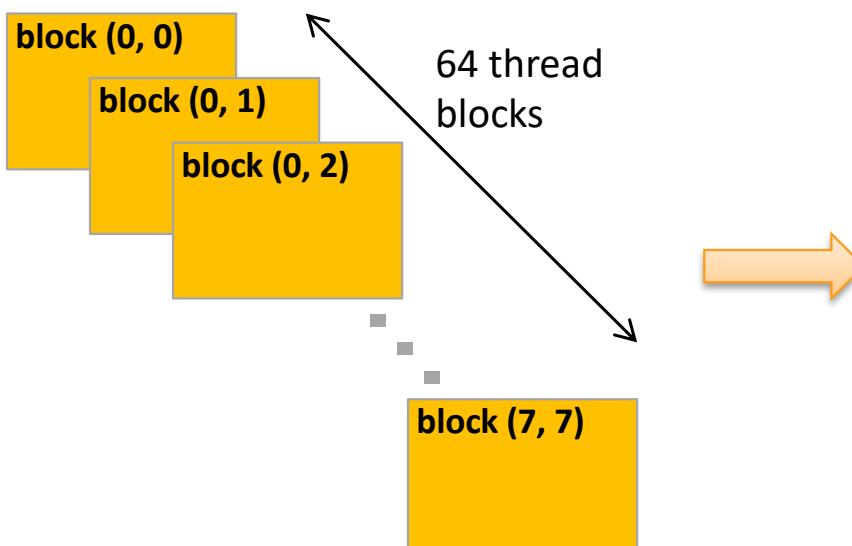
```
Device 1: "Tesla K40c"
  CUDA Driver Version / Runtime Version      8.0 / 8.0
  CUDA Capability Major/Minor version number: 3.5
  Total amount of global memory:             11440 MBytes (11995578368 bytes)
  (15) Multiprocessors, (192) CUDA Cores/MP: 2880 CUDA Cores
  GPU Max Clock rate:                      745 MHz (0.75 GHz)
  Memory Clock rate:                       3004 Mhz
  Memory Bus Width:                        384-bit
  .
  .
  Total amount of constant memory:          65536 bytes
  Total amount of shared memory per block:   49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                32
  Maximum number of threads per multiprocessor: 2048
  Maximum number of threads per block:        1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                     2147483647 bytes
  Texture alignment:                        512 bytes
  Concurrent copy and kernel execution:     Yes with 2 copy engine(s)
  Run time limit on kernels:                No
  .
  .
```

# Producto de Matrices (2048x2048)

<pre>__global__ void MMCKernel(float *dA, float *dB, float *dC, int N) { int col = blockIdx.x * blockDim.x + threadIdx.x; for (int i=0; i&lt;N; i++) {     for (int k=0, float tmp = 0.0; k&lt;N; k++)         tmp += dA[i*N+k] * dB[k*N+col];     dC[i*N+col] = tmp; }}</pre>	<p>GTX1080ti K40c</p> <pre>dim3 dimG(16, 1, 1); dim3 dimB(128, 1, 1); // Invocación MMCKernel&lt;&lt;&lt;dimG, dimB&gt;&gt;&gt;(..., 2048);</pre>	48,20 GFLOPs 13,85 GFLOPs
<pre>__global__ void MMFKernel(float *dA, float *dB, float *dC, int N) { int fil = blockIdx.y * blockDim.y + threadIdx.y; for (int j=0; j&lt;N; j++) {     for (int k=0, float tmp = 0.0; k&lt;N; k++)         tmp += dA[fil*N+k] * dB[k*N+j];     dC[fil*N+j] = tmp; }}</pre>	<pre>dim3 dimG(1, 16, 1); dim3 dimB(1, 128, 1); // Invocación MMFKernel&lt;&lt;&lt;dimG, dimB&gt;&gt;&gt;(..., 2048);</pre>	34,17 GFLOPs 4,53 GFLOPs
<pre>__global__ void MMKernel(float *dA, float *dB, float *dC, int N) { int row = blockIdx.y * blockDim.y + threadIdx.y; int col = blockIdx.x * blockDim.x + threadIdx.x; for (int k=0, float tmp = 0.0; k&lt;N; k++)     tmp += dA[row*N+k] * dB[k*N+col]; dC[row*N+col] = tmp; }</pre>	<pre>dim3 dimG(64, 64, 1); dim3 dimB(32, 32, 1); // Invocación MMKernel&lt;&lt;&lt;dimG, dimB&gt;&gt;&gt;(..., 2048);</pre>	380,1 GFLOPs 114,4 GFLOPs

# Scheduling

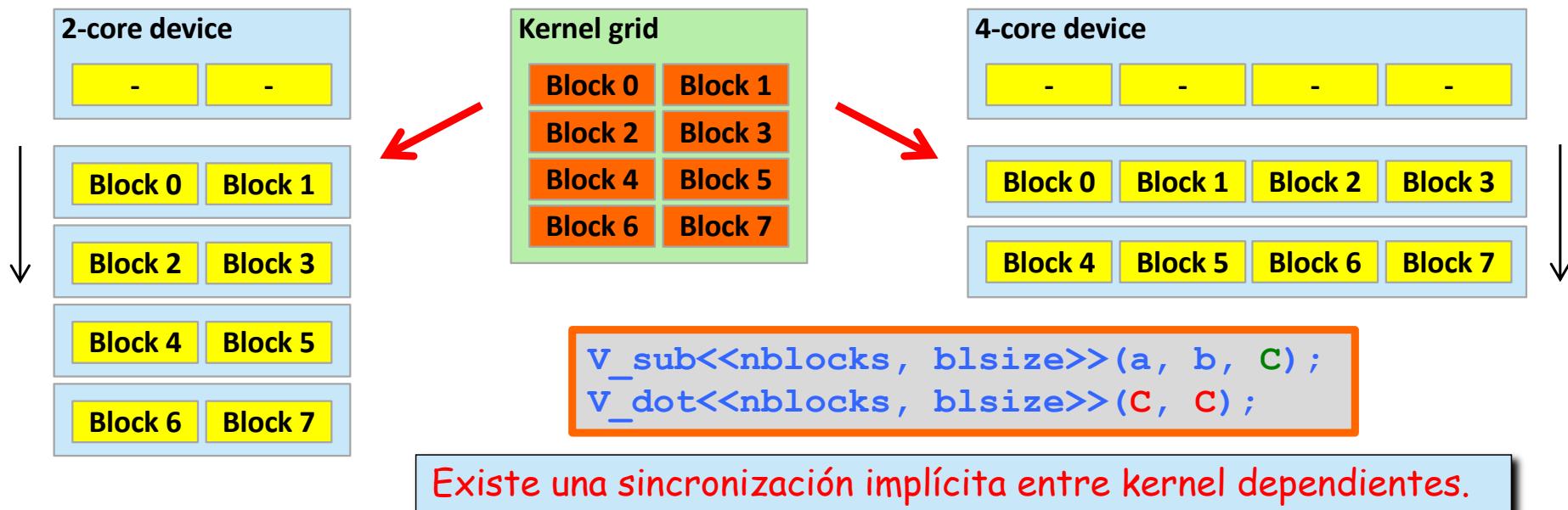
```
N = 128; size = 16;  
dim3 Grid(8, 8, 1); // 64 blocks  
dim3 Block(16, 16, 1); // 256 threads  
  
// Invocación 256 threads por block  
MMkernel<<<Grid, Block>>>(dA, dB, dC, N);
```



- Los threads se asignan a los **Streaming Multiprocessors (SM)** por blocks. En Fermi:
  - Hasta 8 blocks por **SM** si los recursos lo permiten
  - Cada **SM** permite hasta 1.536 threads: 6 blocks de 256 threads
- Los threads se ejecutan de forma concurrente
  - El **SM** mantiene los identificadores de thread y block
  - El **SM** gestiona la ejecución de los threads
- Los threads de un **MISMO BLOCK PUEDEN** compartir información y sincronizarse entre si.
- Los threads de **BLOCKS DIFERENTES NO PUEDEN** compartir información, ni sincronizarse entre si.
  - No existe ningún orden prefijado entre ellos.

# Escalabilidad

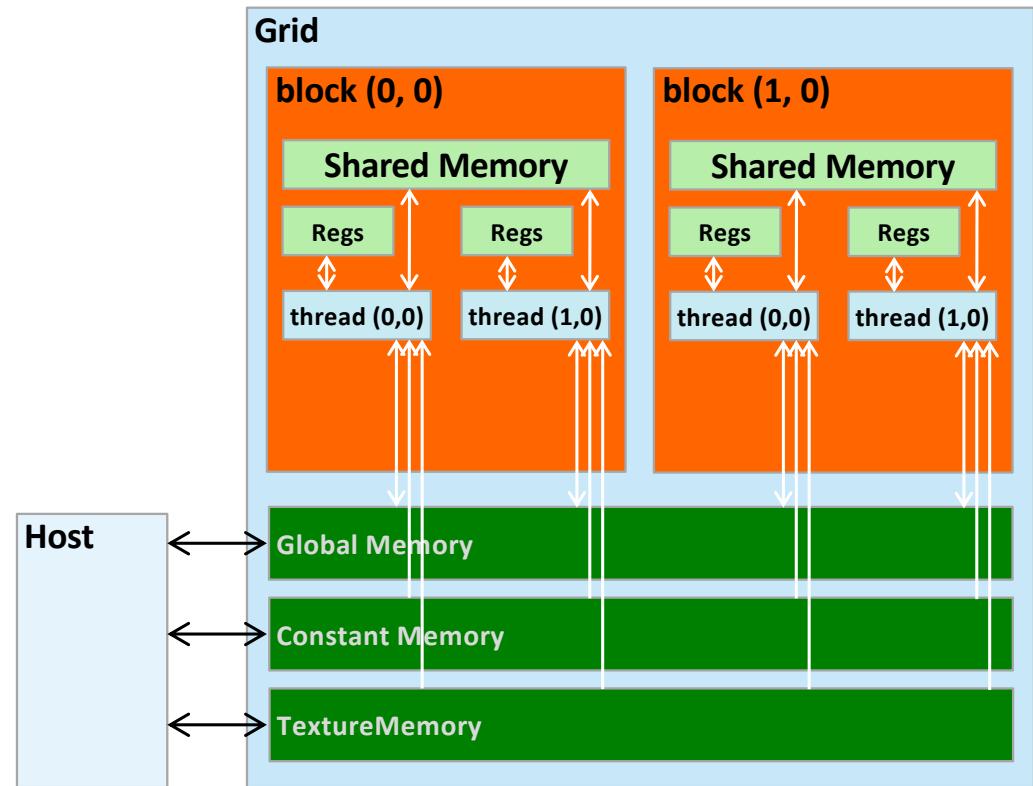
- Los thread blocks no se pueden sincronizar entre si
  - Los threads blocks han de ser independientes
  - En consecuencia, pueden ejecutarse en cualquier orden, secuencial o concurrente.
- Esta independencia nos da escalabilidad
  - Un kernel dividido en thread blocks escala perfectamente en cualquier número de cores.



# Visión del programador CUDA de la Memoria

Cada thread puede:

- READ/WRITE per-thread **registers**
  - ≈ 1 ciclo
- READ/WRITE per-block **shared memory**.
  - ≈ 5 ciclos
- READ/WRITE per-grid **global memory**.
  - ≈ 100's ciclos
- READ ONLY per-grid **constant memory**.
  - ≈ 5 ciclos (si está en cache)
- READ/WRITE per-grid **Texture memory**.
  - ≈ 100's ciclos
  - pocos ciclos (si está en cache)



# Espacios de Memoria

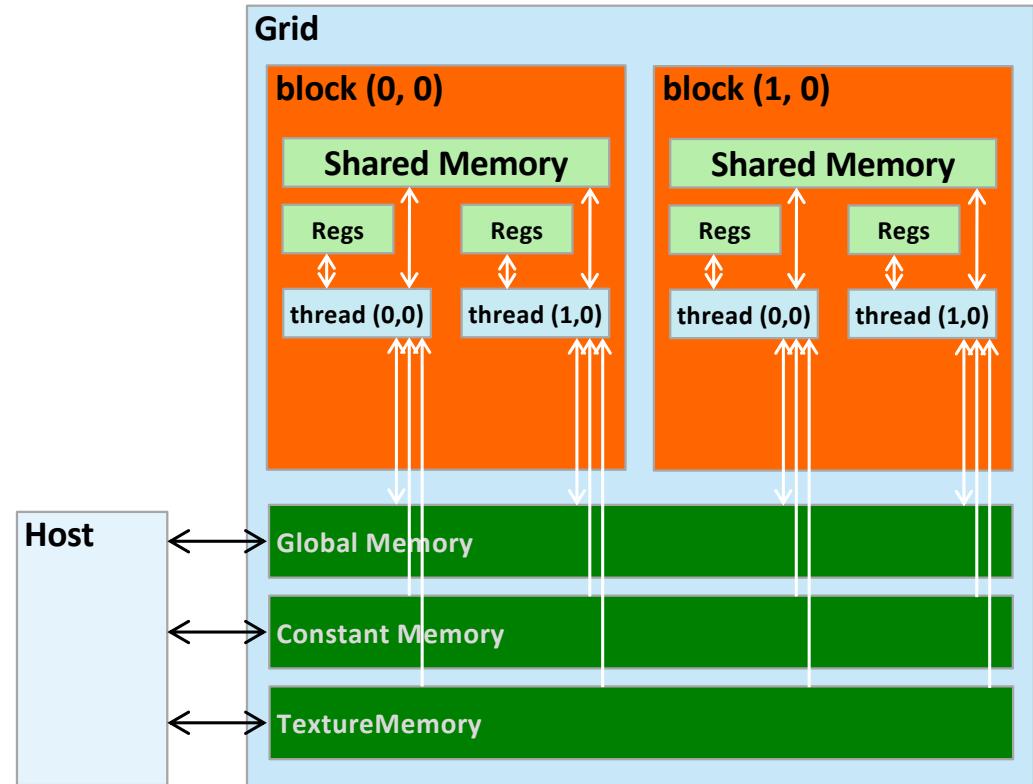
Cada thread puede:

- READ/WRITE per-thread **registers**.
- READ/WRITE per-block **shared memory**.
- READ/WRITE per-grid **global memory**.
- Los más importantes y comúnmente usados.**

Además, cada thread puede:

- READ ONLY per-grid **constant memory**
- READ ONLY per-grid **texture memory**
- Usados por conveniencia o rendimiento**

El host puede READ/WRITE global, constant y texture memory.



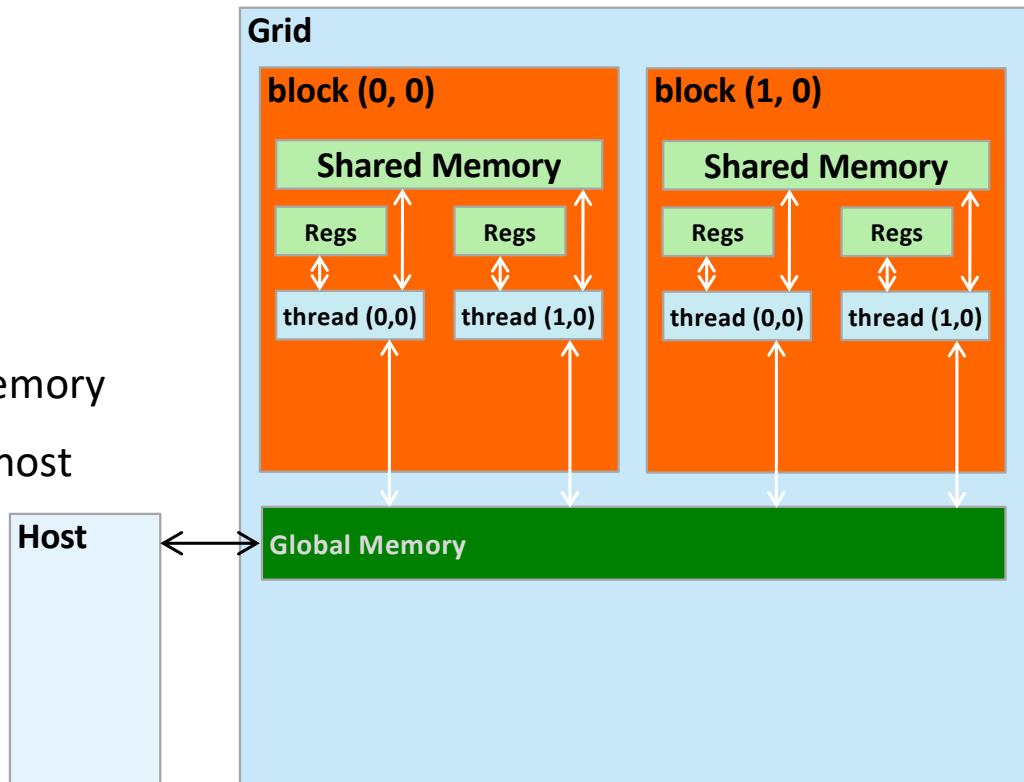
# Espacios de Memoria

El código del device puede

- READ/WRITE per-thread **registers**.
- READ/WRITE per-block **shared memory**.
- READ/WRITE per-grid **global memory**.

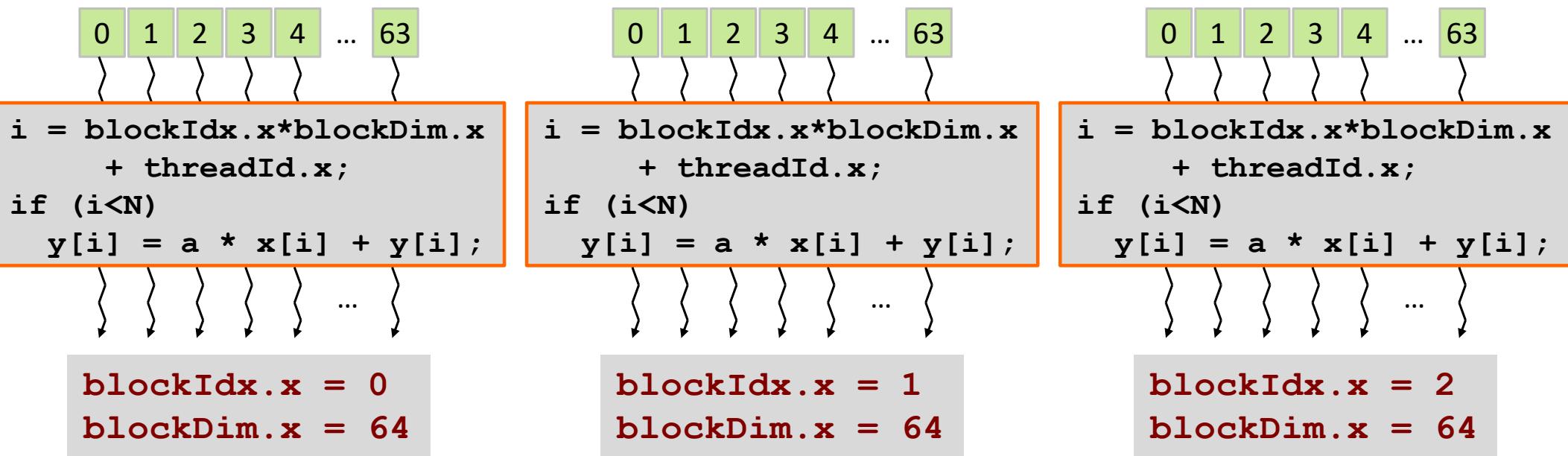
El código del host puede

- Transferir datos per-grid desde el host a la global memory
- Transferir datos per-grid desde la global memory al host
- cudaMalloc ()**, **cudaFree ()**
- cudaMemcpy ()**
  - La transferencia hacia el device es síncrona.



# La gran idea que hay detrás de CUDA

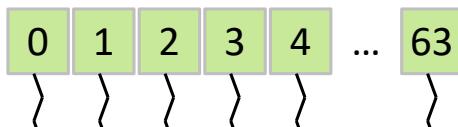
```
// Invocación (3 bloques con 64 threads por bloque)
int nBlocks = 3, nThreads = 64;
saxpyP<<<nBlocks, nThreads>>>(192, 3.5, x, y);
```



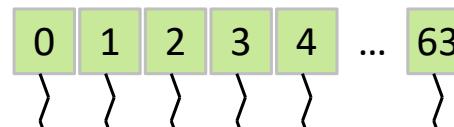
# Kernel ≈ Muchos Threads Concurrentes

## ☐ Los threads se ejecutan en un grid de thread blocks

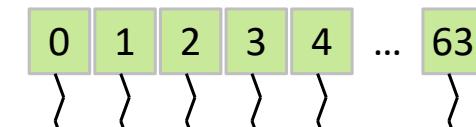
- Todos los threads de un grid ejecutan el mismo kernel (SPMD, *single program multiple data*).
- Los threads de un mismo thread block pueden cooperar a través de memoria compartida, usar operaciones atómicas y herramientas de sincronización.
- Los threads de diferentes thread blocks no pueden cooperar directamente entre ellos.



```
i = blockIdx.x*blockDim.x  
      + threadIdx.x;  
if (i<N)  
    y[i] = a * x[i] + y[i];
```



```
i = blockIdx.x*blockDim.x  
      + threadIdx.x;  
if (i<N)  
    y[i] = a * x[i] + y[i];
```



```
i = blockIdx.x*blockDim.x  
      + threadIdx.x;  
if (i<N)  
    y[i] = a * x[i] + y[i];
```

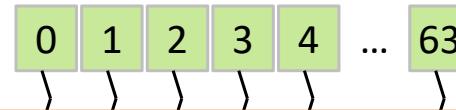
Thread Block 0

Thread Block 1

Thread Block 2

# Sincronización entre Threads

- La sincronización entre los threads de un mismo thread block se realiza con la función `_syncthreads()`. Funciona como un BARRIER.



```
idx = blockIdx.x*blockDim.x + threadIdx.x;  
scratch[idx] = f(input[idx]);  
_syncthreads();  
left = scratch[idx-1];  
right = scratch[idx+1];
```

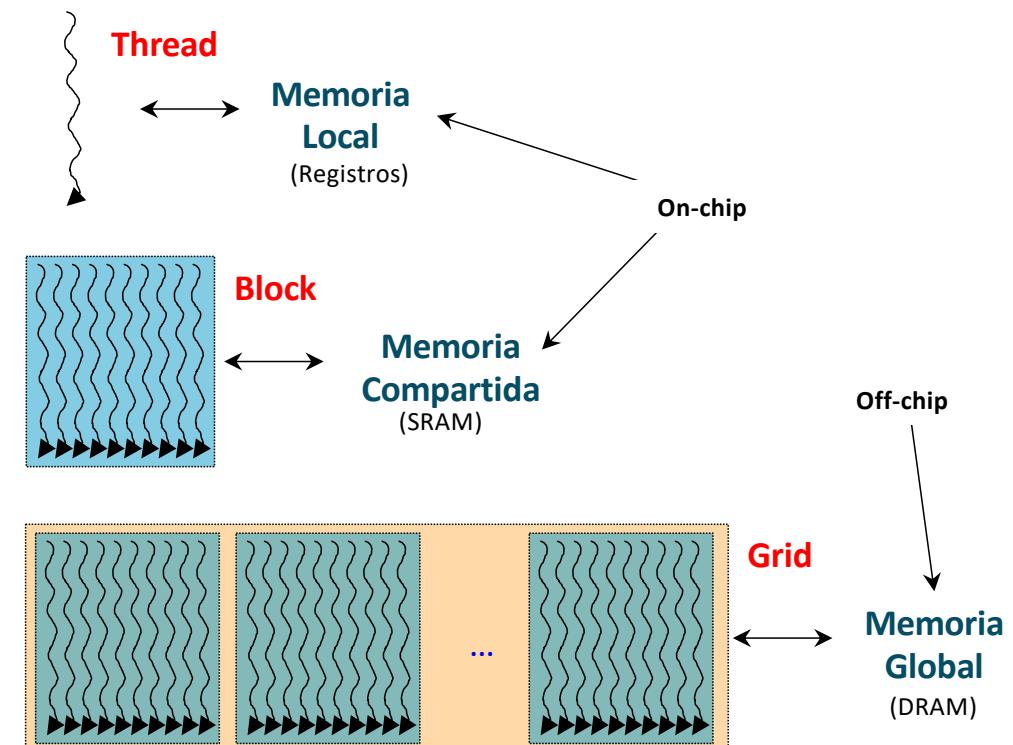


Los threads se esperan en el barrier hasta que **TODOS** los threads del thread block llegan a este punto.

Dentro de un kernel, no se puede sincronizar los thread blocks.

# Jerarquía de Memoria vs Jerarquía de Cálculo

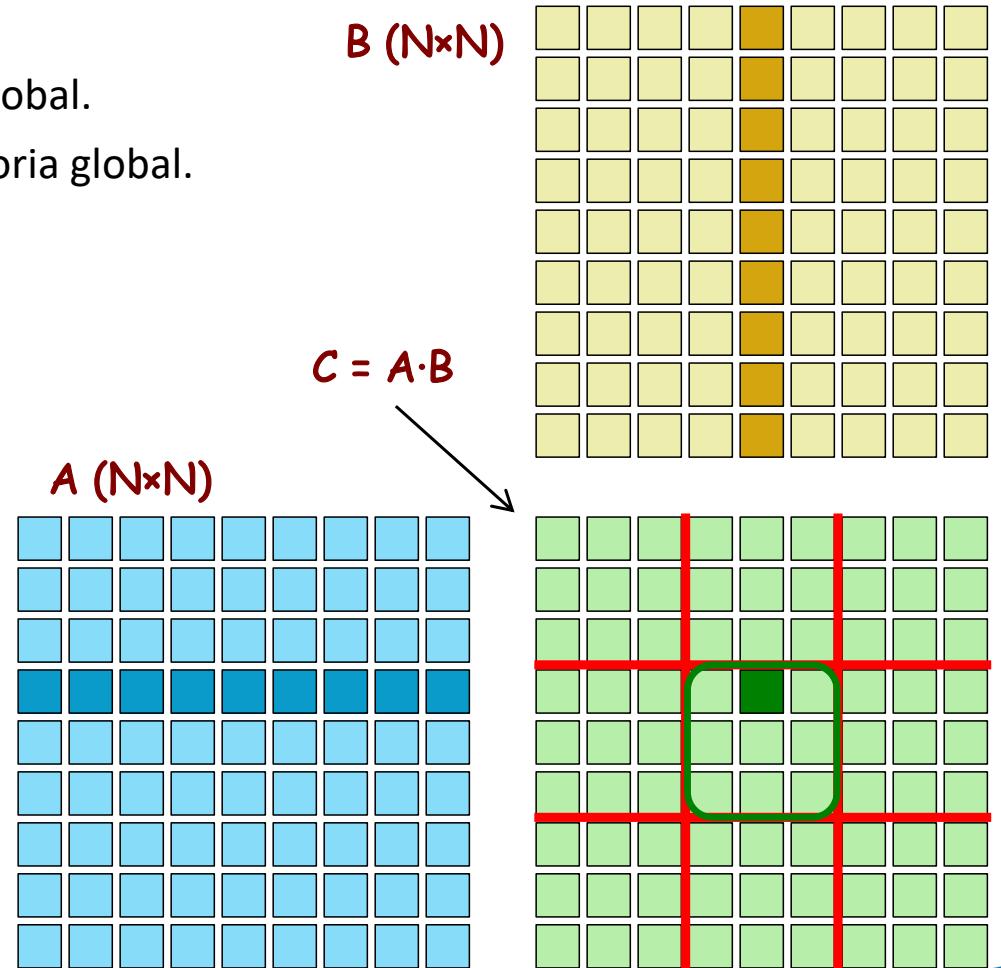
- **Thread**, en un thread se ejecuta una instancia concreta de un kernel.
  - Utiliza Memoria Local (registros)
- **Thread Block**, está compuesto por threads.
  - El número de threads del bloque se ha de indicar explícitamente.
  - Los threads pueden cooperar compartiendo datos en la shared memory (memoria interna de la GPU).
- **Grid**, está compuesto por bloques.
  - El número de bloques se ha indicar explícitamente.
  - Utilizan la memoria global (memoria de la tarjeta).
- La jerarquía se ha de manipular de forma explícita.
- El buen uso de esta jerarquía es fundamental para obtener buenos rendimientos.



Esta jerarquía está directamente relacionada con la estructura interna de las GPUs de NVIDIA.

# Producto de Matrices

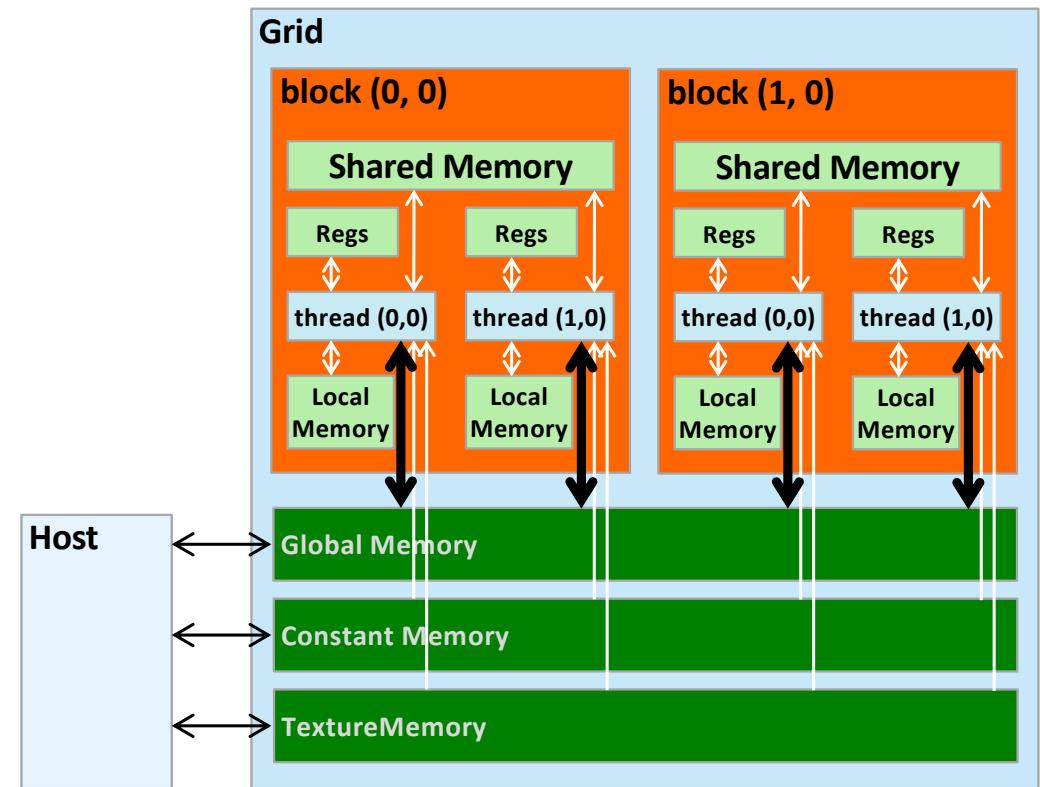
- Cada thread calcula un elemento de la matriz C.
  - Cada fila de A es leída N veces desde la memoria global.
  - Cada columna de B es leída N veces desde la memoria global.
- Se desperdicia mucho ancho de banda.
- Mal equilibrio entre cálculo y ancho de banda
- Cada thread realiza:
  - $2N$  operaciones en CF
  - $2N$  accesos a memoria ( $2N \times 4B$ )
- **Objetivo: 1 TFLOP**
  - Rendimiento pico K40: 4,29 TFLOPS (✓)
- Ancho de banda necesario: 4 TB/s
  - Ancho de banda K40: 288 GB/s (X)



# Revisando el Producto de Matrices

El rendimiento está limitado por el ancho de banda con memoria.

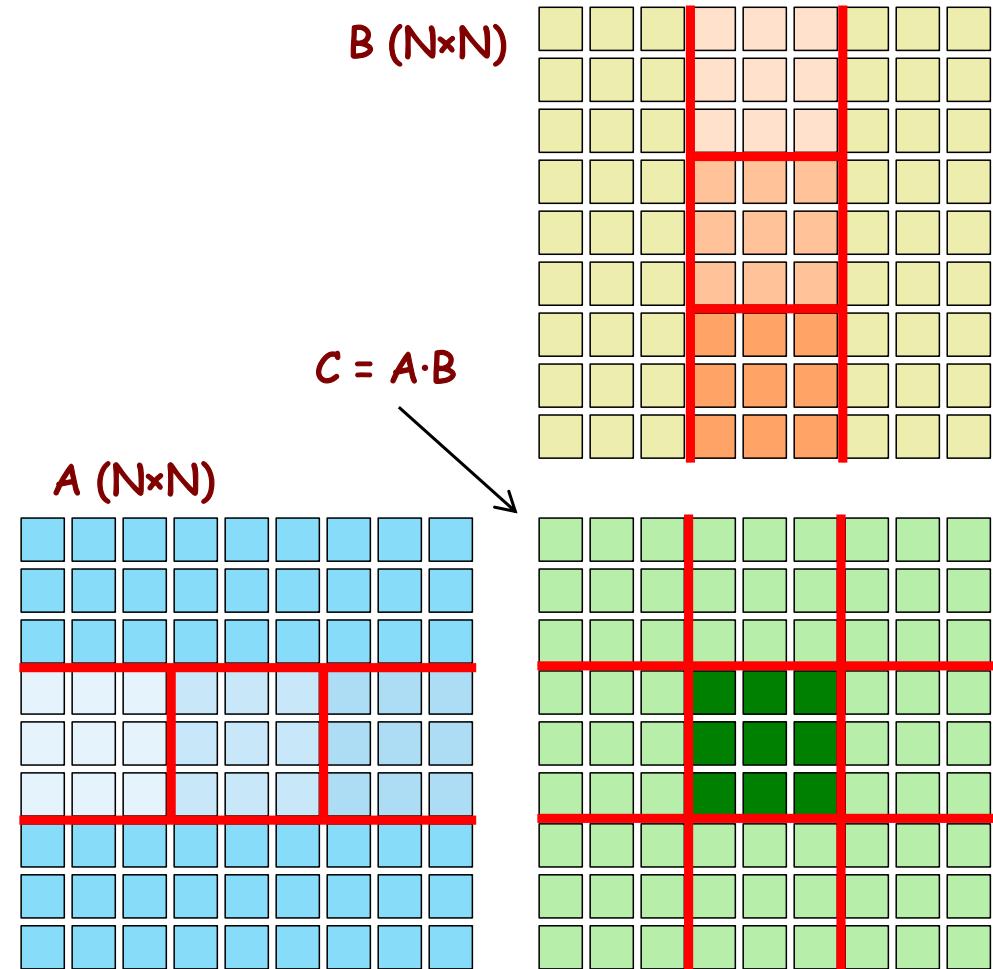
- En una tarjeta Fermi llegamos a 25 GFLOPs, pero:
  - Potencia de cálculo: 1TFLOP
  - Ancho de Banda: 150 GB/s
- En cada iteración de k hacemos 2 accesos a memoria (8B) y 2 operaciones en coma flotante.
- Por cada 4B leídos hacemos 1 operación CF.
- Para alcanzar un TFLOP con este algoritmo necesitamos: 4.000 GB/s de ancho de banda.
- Con 150 GB/s, el máximo que podríamos alcanzar es: 37,5 GFLOPs.
- **Hacen falta cambios sustanciales para acercarnos a la potencia de pico.**
  - **HAY QUE USAR LA SHARED MEMORY.**



# Producto de Matrices a bloques

$$C = A \cdot B \text{ (tamaño } N \times N\text{)}$$

- Cada block thread calcula un bloque de datos de  $M \times M$  elementos de la matriz  $C$ .
- El algoritmo itera  $N$ /veces. En cada iteración
  - Traemos un bloque  $M \times M$  de  $A$  a la memoria compartida. La matriz  $A$  se lee  $N/M$  veces.
  - Traemos un bloque  $M \times M$  de  $B$  a la memoria compartida. La matriz  $B$  se lee  $N/M$  veces.
- El valor de  $M$  es importante:
  - Las submatrices han de caber en la memoria compartida.
  - Necesitamos  $M \times M$  threads por bloque.
- Utilizamos menos ancho de banda. El equilibrio entre cálculo y ancho de banda mejorará.



# Producto de Matrices

```
__global__ void MMkernel2(float *dA, float *dB, float *dC, int N) {
    __shared__ float sA[M][M]; __shared__ float sB[M][M];
    int thx = threadIdx.x; int thy = threadIdx.y;
    int row = blockIdx.y * M + thy;
    int col = blockIdx.x * M + thx;
    float tmp = 0.0;
    for (s=0; s<(N/M); s++) {
        sA[thy][thx] = dA[row*N + s*M + thx];
        sB[thy][thx] = dB[(s*M + thy)*N + col];
        __syncthreads();
        for (int k=0; k<M; k++)
            tmp += sA[thy][k] * sB[k][thx];
        __syncthreads();
    }
    dC[row*N+col] = tmp;
}
```

Submatrices en la Memoria Compartida

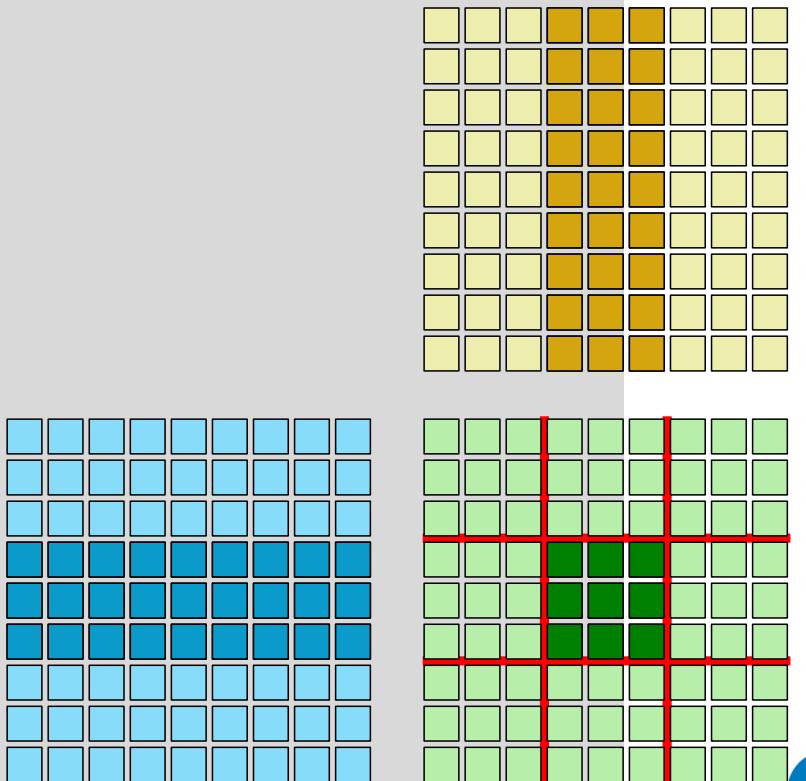
Todos los threads del bloque cooperan para cargar las submatrices A y B en la memoria compartida

Antes de empezar a calcular hay que asegurarse que todos los threads hayan acabado de cargar las submatrices A y B.

Antes de continuar hay que asegurarse que todos los threads hayan acabado los cálculos que le corresponden.

# Producto de Matrices

```
__global__ void MMkernel2(float *dA, float *dB, float *dC, int N) {
    __shared__ float sA[M][M]; __shared__ float sB[M][M];
    int thx = threadIdx.x; int thy = threadIdx.y;
    int row = blockIdx.y * M + thy;
    int col = blockIdx.x * M + thx;
    float tmp = 0.0;
    for (s=0; s<(N/M); s++) {
        sA[thy][thx] = dA[row*N + s*M + thx];
        sB[thy][thx] = dB[(s*M + thy)*N + col];
        __syncthreads();
        for (int k=0; k<M; k++)
            tmp += sA[thy][k] * sB[k][thx];
        __syncthreads();
    }
    dC[row*N+col] = tmp;
}
```



# Producto de Matrices (granularidad)

```
__global__ void MMkernel2(float *dA, float *dB, float *dC, int N) {
    __shared__ float sA[M][M]; __shared__ float sB[M][M];
    int thx = threadIdx.x; int thy = threadIdx.y;
    int row = blockIdx.y * M + thy;
    int col = blockIdx.x * M + thx;
    float tmp = 0.0;
    for (s=0; s<(N/M); s++) {
        sA[thy][thx] = dA[row*N + s*M + thx];
        sB[thy][thx] = dB[(s*M + thy)*N + col];
        __syncthreads();
        for (int k=0; k<M; k++)
            tmp += sA[thy][k] * sB[k][thx];
        __syncthreads();
    }
    dC[row*N+col] = tmp;
}
```

dim3 dimGrid(N/M, N/M, 1);  
dim3 dimBlock(M, M, 1);

// Invocación, **N es múltiplo de M**  
MMkernel2<<<dimGrid, dimBlock>>>(...);

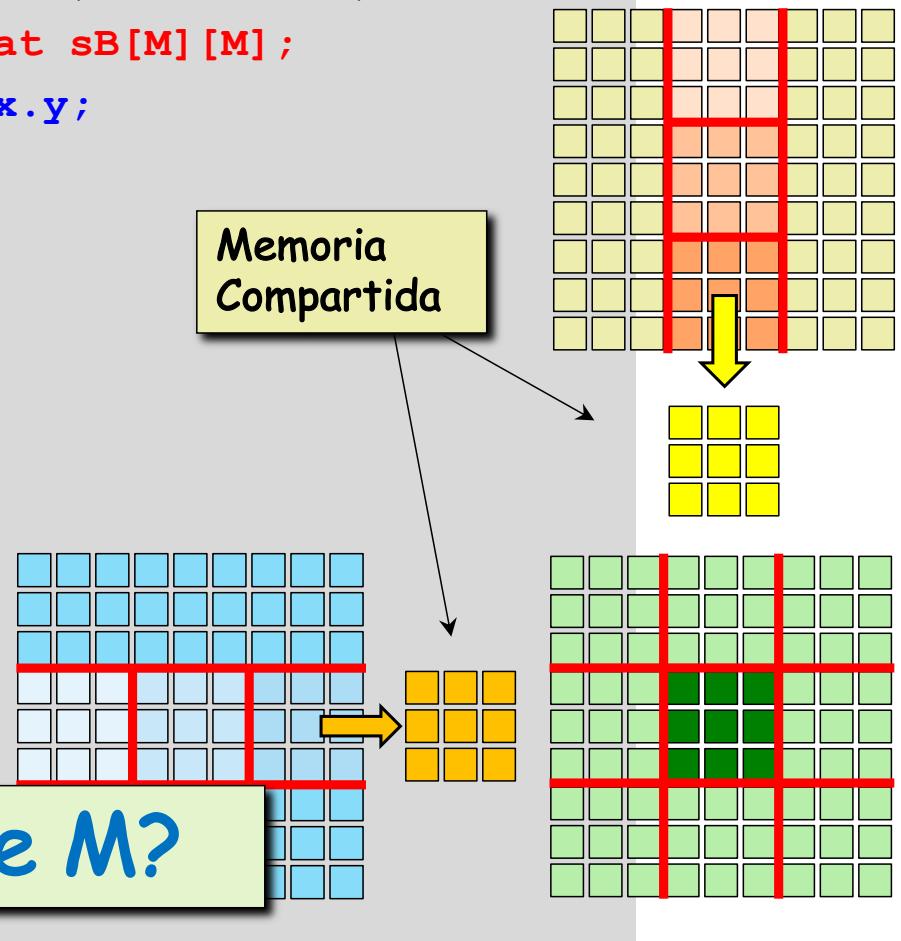
**Memoria Compartida**

Todos los threads del bloque cooperan para cargar las submatrices A y B en la memoria compartida

# Producto de Matrices (granularidad)

```
__global__ void MMkernel2(float *dA, float *dB, float *dC, int N) {
    __shared__ float sA[M][M]; __shared__ float sB[M][M];
    int thx = threadIdx.x; int thy = threadIdx.y;
    int row = blockIdx.y * M + thy;
    int col = blockIdx.x * M + thx;
    float tmp = 0.0;
    for (s=0; s<(N/M); s++) {
        sA[thy][thx] = dA[row*N + s*M + thx];
        sB[thy][thx] = dB[(s*M + thy)*N + col];
        __syncthreads();
        for (int k=0; k<M; k++)
            tmp += sA[thy][k] * sB[k][thx];
        __syncthreads();
    }
    dC[row*N+col]
}
```

Memoria Compartida



# Producto de Matrices (granularidad) en GPUs Fermi

Los **SM** de Fermi permiten hasta **1.536 threads** y un máximo de **8 blocks**.

**$8 \times 8 = 64 \text{ threads por block}$ :**

- Con un máximo de 8 blocks sólo se ejecutarán 512 threads.

**$16 \times 16 = 256 \text{ threads por block}$ :**

- Con un máximo de 1.536 threads sólo pueden ejecutarse 6 blocks

**$32 \times 32 = 1.024 \text{ threads por block}$ :**

- Con un máximo de 1.536 threads sólo pueden ejecutarse 1 block

6 blocks  
16 × 16 threads por block  
8 bytes por thread

Cada **SM** en Fermi tiene 16 KB o 48 KB de memoria compartida (configurable).

- No estamos limitados por la memoria compartida
- Con el máximo número de bloques necesitamos:
  - ✓  $(8 \times 8)$ : 4KB;  $(16 \times 16)$ : 12KB;  $(32 \times 32)$ : 8KB;

# Producto de Matrices (granularidad) en GPUs Kepler

Los **SM** de Kepler permiten hasta **2.048 threads** y un máximo de **16 blocks**.

**$8 \times 8 = 64 \text{ threads por block}$** :

- Con un máximo de 16 blocks sólo se ejecutarán 1024 threads.

**$16 \times 16 = 256 \text{ threads por block}$** :

- Con un máximo de 2.048 threads sólo pueden ejecutarse 8 blocks

**$32 \times 32 = 1.024 \text{ threads por block}$** :

- Con un máximo de 2.048 threads sólo pueden ejecutarse 2 blocks

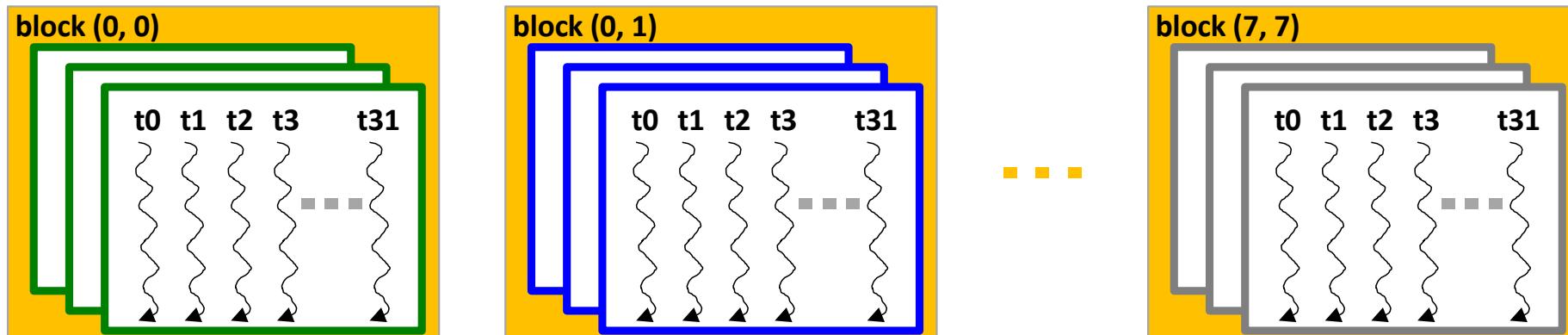
Cada **SM** en Kepler tiene 16 KB o 48 KB de memoria compartida (configurable).

- No estamos limitados por la memoria compartida
- Con el máximo número de bloques necesitamos:
  - ✓  $(8 \times 8)$ : 8KB;  $(16 \times 16)$ : 16KB;  $(32 \times 32)$ : 16KB;

# Producto de Matrices (granularidad)

- **size = 16 →  $16 \times 16 = 256$  threads por block**, para cada bloque se realizan:
  - $256 \times 16 \times 2 = 8.192$  operaciones en coma flotante
  - $2 \times 256 = 512$  loads de 1 float (2 KB)
  - Por cada acceso a memoria (4B) se pueden hacer 16 operaciones CF (4 op CF por 1B leido).
  - Si el ancho de banda es 150 GB/s, podríamos llegar como **máximo a 600 GFLOPs**.
- **size = 32 →  $32 \times 32 = 1.024$  threads por block**, para cada bloque se realizan:
  - $1.024 \times 32 \times 2 = 65.536$  operaciones en coma flotante
  - $2 \times 1.024 = 2.048$  loads de 1 float (8 KB)
  - Por cada acceso a memoria (4B) se pueden hacer 32 operaciones CF (8 op CF por 1B leido).
  - Si el ancho de banda es 150 GB/s, podríamos llegar como **máximo a 1.2 TFLOPs**.
- **Para alcanzar estos rendimientos hay que saturar los SMs de la GPU**

# Thread Scheduling



## Cada block se ejecuta en grupos de 32 threads (**WARP**)

- Es una decisión de diseño, no es parte del modelo de programación CUDA
- Los **WARPS** son la unidad de ejecución en un **SM**
- Un warp ejecuta la misma instrucción para todos los threads
- Si hay saltos condicionales, sólo se ejecuta uno de los caminos.
- El particionado en **WARPS** siempre es el mismo.
- Se puede usar este conocimiento en el control de flujo.
- El tamaño exacto de los **WARPS** podría cambiar en el futuro.

## El patrón de acceso a memoria de los warps es fundamental para obtener un buen rendimiento.

### MUY IMPORTANTE!

Es fundamental que el número de THREADS de un BLOCK sea **múltiplo del tamaño del WARP** si queremos obtener buenos rendimientos.

# Comportamiento de los saltos

¿Qué hacemos con los saltos en un warp?

- Si hay saltos condicionales, sólo se ejecuta uno de los caminos. ¡PÉRDIDA de RENDIMIENTO!

TIEMPO ↓

0	1	2	3	4	5	...	31
C	C	C	C	C	C	...	C
C	C	C	C	C	C	...	C
T	T	F	T	T	T	...	F
I1	I1	X	I1	I1	I1	...	X
I2	I2	X	I2	I2	I2	...	X
I3	I3	X	I3	I3	I3	...	X
J	J	X	J	J	J	...	X
X	X	I4	X	X	X	...	I4
X	X	I5	X	X	X	...	I5
I6	I6	I6	I6	I6	I6	...	I6

```
if (cond) {INST1; INST2; INST3}  
else {INST4; INST5}  
INST6
```

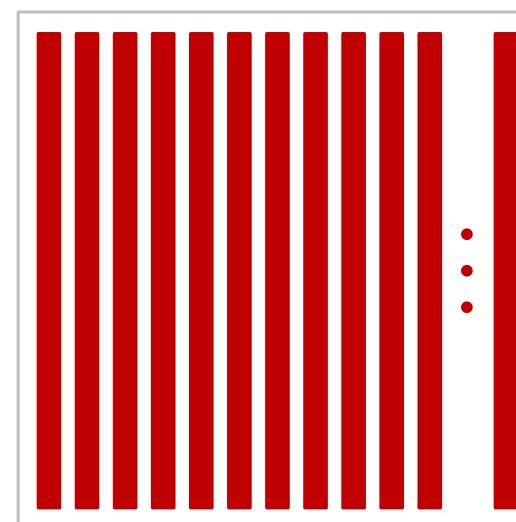
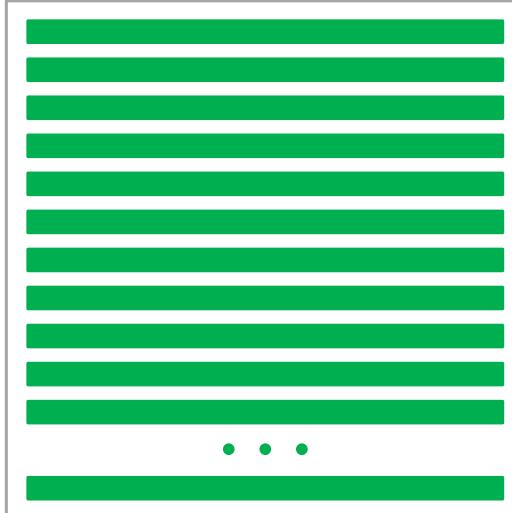
La pérdida de rendimiento es evidente.  
Podría ser mucho peor.

```
if (cond1) {  
    if (cond2) INST1 else INST2  
} else {  
    if (cond3) INST3 else INST4  
}
```

# Transponer una matriz

## □ Ejemplo simple, pero muy ilustrativo

```
void TransSerie(int N, float In[N][N], float Out[N][N]) {  
    for (int i=0; i<N; i++)  
        for (int j=0; j<N; j++)  
            Out[j][i] = In[i][j];  
}
```



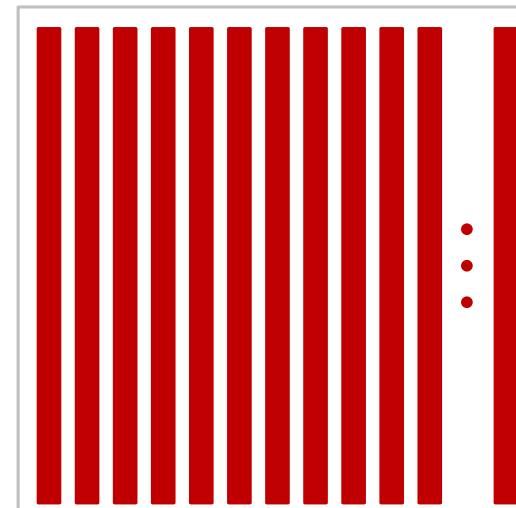
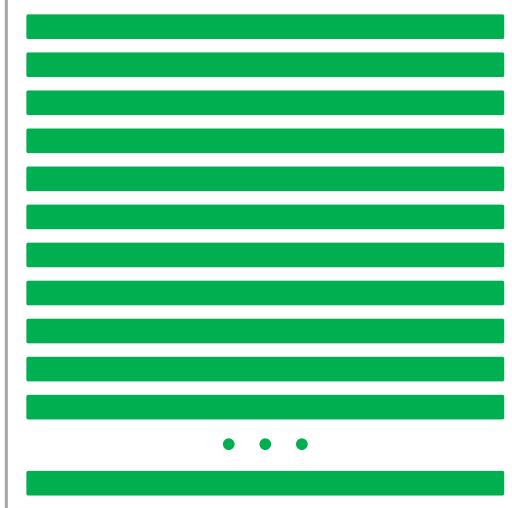
- Intercambiar filas por columnas
- Por simplicidad, matrices cuadradas y  $N = 2^m$

# Transponer una matriz

## □ Ejemplo simple, pero muy ilustrativo

```
void TransSerie(int N, float *In, float *Out) {  
    for (int i=0; i<N; i++)  
        for (int j=0; j<N; j++)  
            Out[j*N + i] = In[i*N + j];  
}
```

Calcular explícitamente los índices para trabajar con punteros.



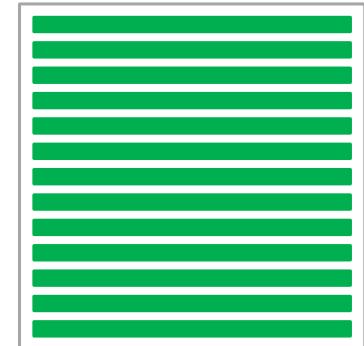
- Intercambiar filas por columnas
- Por simplicidad, matrices cuadradas y  $N = 2^m$

# Transponer una matriz

- Cada Thread se encarga de 1 fila

```
__global__ void Trans1row(int N, float *In, float *Out) {  
    int row = blockIdx.y * blockDim.y + threadIdx.y;  
    for (int col=0; col<N; col++)  
        Out[col * N + row] = In[row * N + col];  
}
```

```
nThreads = 1024;           // número de Threads  
nBlocks = N/nThreads; // número de Blocks  
  
dim3 dimGrid(1, nBlocks, 1);  
dim3 dimBlock(1, nThreads, 1);  
  
Trans1row<<<dimGrid, dimBlock>>>(N, d_A, d_B);
```



# Transponer una matriz

## □ Código del Host

```
// Obtener Memoria en el host  
h_A = (float*) malloc(numBytes);  
h_B = (float*) malloc(numBytes);  
  
// Inicializa las matrices  
  
// Obtener Memoria en el device  
cudaMalloc((float**)&d_A, numBytes);  
cudaMalloc((float**)&d_B, numBytes);  
  
// Copiar datos desde el host en el device  
cudaMemcpy(d_A, h_A, numBytes, cudaMemcpyHostToDevice);  
  
// Ejecutar el kernel  
Translrow<<<dimGrid, dimBlock>>>(N, d_A, d_B);  
  
// Obtener el resultado desde el host  
cudaMemcpy(h_B, d_B, numBytes, cudaMemcpyDeviceToHost);
```

Medimos tiempos de ejecución.

Con N = 2048  
1024 threads  
2 Blocks

Texe = 6,1 ms

¿Es un tiempo aceptable?

# Transponer una matriz

- Cada Thread se encarga de 1 fila
- Calculamos el ancho de banda con memoria:
  - Lectura matriz 2048 × 2048 floats
  - Escritura matriz 2048 × 2048 floats
  - En total movemos 8.388.608 floats (33.554.432 bytes) en 6,1 ms
  - **ANCHO de BANDA: 5,5 GB/s**
- El ancho de banda de la K40 es 288 GB/s (usamos el 1,9% de Ancho de banda de la DRAM)
- Muchos códigos para GPU están limitados por memoria

**SIEMPRE HAY QUE EMPEZAR  
MIDIENDO EL ANCHO de  
BANDA.**

**AB $\leq$ 35%, ¡POBRE!  
35 $<$ AB $\leq$ 60%, ¡ACEPTABLE!  
60% $<$ AB, ¡EXCELENTE!**

# Transponer una matriz

- Cada Thread se encarga de 1 fila

```
__global__ void Translrow(int N, float *In, float *Out) {  
    int row = blockIdx.y * blockDim.y + threadIdx.y;  
    for (int col = 0; col < N; col++)  
        Out[col * N + row] = In[row * N + col];  
}
```

Acceso a posiciones con stride  $4 \cdot N$ .  
¡Rendimiento muy pobre!

Acesso a posiciones consecutivas  
de memoria. ¡Perfecto!

¿Seguro que es así? ¿Cómo influyen los warps?

# Transponer una matriz

- Cada Thread se encarga de 1 elemento

```
__global__ void Trans1x1(int N, float *In, float *Out) {  
    int row = blockIdx.y * blockDim.y + threadIdx.y;  
    int col = blockIdx.x * blockDim.x + threadIdx.x;  
    Out[col * N + row] = In[row * N + col];  
}
```

```
nThreads = 32;           // número de Threads  
nBlocks = N/nThreads;   // número de Blocks  
  
dim3 dimGrid(nBlocks, nBlocks, 1);  
dim3 dimBlock(nThreads, nThreads, 1);  
  
Trans1x1<<<dimGrid, dimBlock>>>(N, d_A, d_B);
```

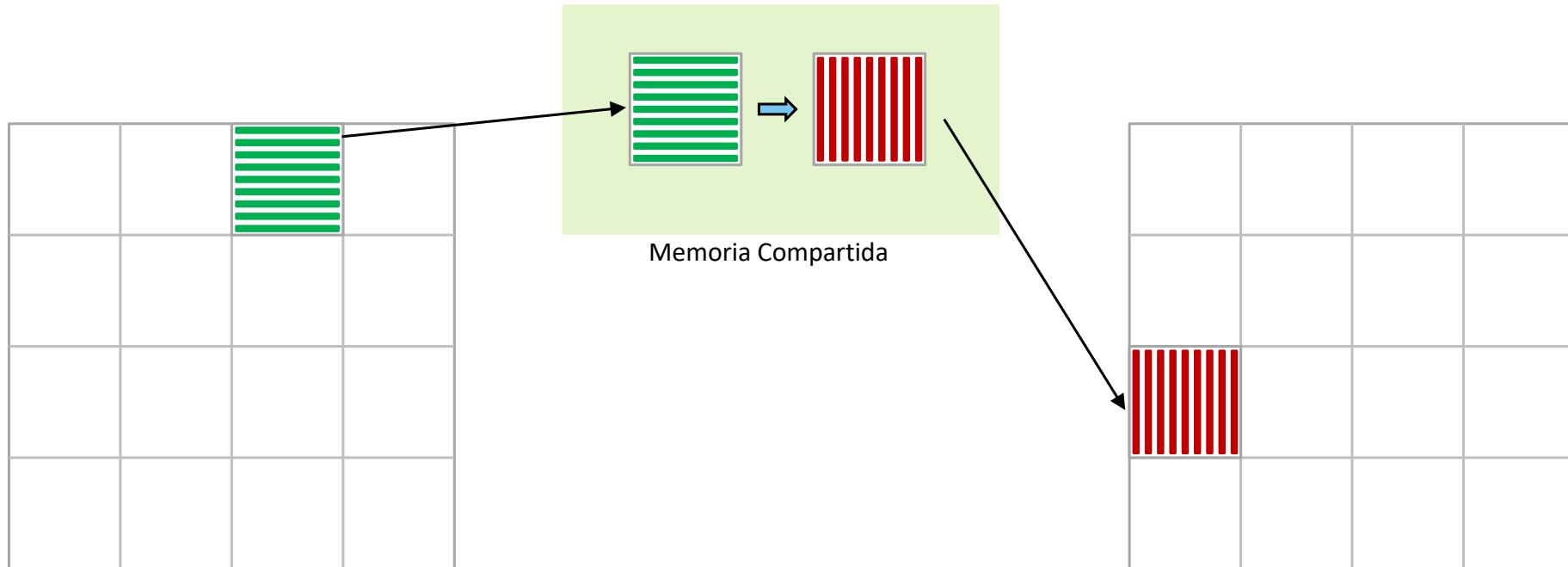
Con  $N = 2048$   
 $32 \times 32$  threads  
 $64 \times 64$  Blocks

$T_{exe} = 0,58$  ms  
 $AB = 57,85$  GB/s  
(20% uso DRAM)

Se mantienen los patrones de acceso. ¿Porqué mejoramos?

# Transponer una matriz

- Cada Thread se encarga de 1 elemento, pero usaremos la memoria compartida



**Los accesos a Memoria Global siempre serán a posiciones consecutivas de memoria.**

# Transponer una matriz

- Cada Thread se encarga de 1 elemento, pero usaremos la memoria compartida

```
__global__ void TranSxS(int N, float *In, float *Out) {  
    __shared__ float sA[SIZE][SIZE];  
    int Icorner_c = blockIdx.x * blockDim.x;  
    int Icorner_r = blockIdx.y * blockDim.y;  
    int Ocorner_c = blockIdx.y * blockDim.y;  
    int Ocorner_r = blockIdx.x * blockDim.x;  
    int y = threadIdx.y;  
    int x = threadIdx.x;  
  
    sA[y][x] = In[(Icorner_r + y) * N + (Icorner_c + x)];  
    __syncthreads();  
    Out[(Ocorner_r + y) * N + (Ocorner_c + x)] = sA[x][y];  
}
```

¿SINCRONIZACIÓN?

Con  $N = 2048$   
 $32 \times 32$  threads  
 $64 \times 64$  Blocks

$T_{exe} = 0,49$  ms  
 $AB = 68,46$  GB/s

(24% uso DRAM)

¿Qué está pasando?

# Transponer una matriz

- Cada Thread se encarga de 1 elemento, pero usaremos la memoria compartida

Con N = 2048  
 $32 \times 32$  threads  
 $64 \times 64$  Blocks

$T_{exe} = 0,49$  ms  
 $AB = 68,46$  GB/s  
(24% uso DRAM)

Con N = 2048  
 $16 \times 16$  threads  
 $128 \times 128$  Blocks

$T_{exe} = 0,34$  ms  
 $AB = 99,16$  GB/s  
(34% uso DRAM)

Con N = 2048  
 $8 \times 8$  threads  
 $256 \times 256$  Blocks

$T_{exe} = 0,47$  ms  
 $AB = 70,19$  GB/s  
(24% uso DRAM)

Múltiples factores.  
Sobre todo, tiene que ver en cómo estamos utilizando los recursos de la GPU

¿Cómo averiguar los recursos de la GPU?

# Recursos hardware de una GPU

- ☐ Rutina `cudaGetDeviceProperties`, en un ejemplo disponible en todas las distribuciones de CUDA.

```
Device 0: "Tesla K40c"
Major revision number: 3
Minor revision number: 5
Total amount of global memory: 11995054080 bytes
Number of multiprocessors: 15
Number of cores: 2880
Total amount of constant memory: 65536 bytes
Total amount of shared memory per block: 49152 bytes
Total number of registers available per block: 65536
Warp size: 32
Maximum number of threads per block: 1024
Maximum sizes of each dimension of a block: 1024 x 1024 x 64
Maximum sizes of each dimension of a grid: 2147483647 x 65535 x 65535
Maximum memory pitch: 2147483647 bytes
Clock rate: 0.75 GHz
Memory Clock rate: 3.00 GHz
Memory Bus Width: 384 bits
Number of asynchronous engines: 2
It can execute multiple kernels concurrently: Yes
Concurrent copy and execution: Yes
```

3  
5

Computing Capability: 3.5

# Computing Capability

□ Indica las características que soporta el hardware de la GPU.

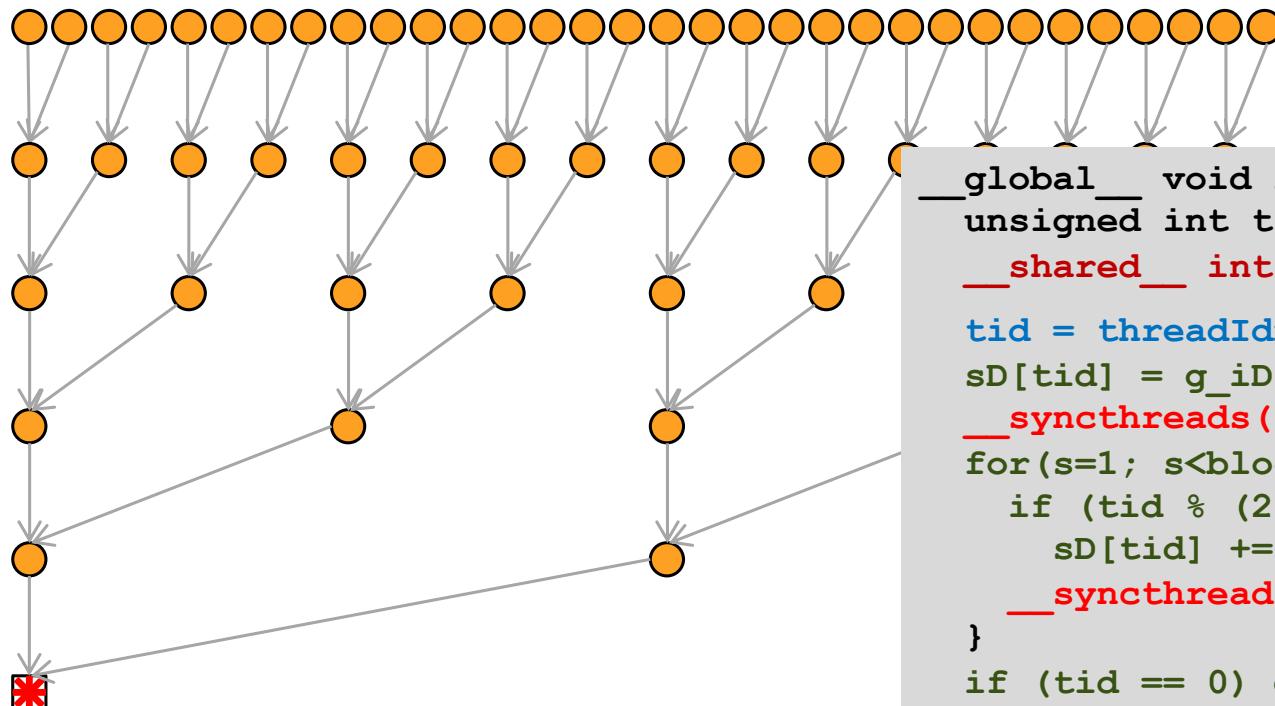
- El primer número identifica la familia: 2 (Fermi), 3 (Kepler), 5 (Maxwell)
- No confundir con la versión de CUDA

□ Algunos datos de la versión 3.5

- Maximum number of resident grids per device: 32
- Maximum number of resident blocks per SM: 16
- Maximum number of resident warps per SM: 64
- Maximum number of resident threads per SM: 2048
- Maximum number 32-bit registers per block: 64K
- Maximum amount of memory per SM: 48 KB

# Una reducción

- Aplicación típica
  - Reducción a nivel de block, #threads =  $2^n$  = N.



```
sum = 0;  
for (i=0; i<N; i++)  
    sum = sum + V[i];
```

```
__global__ void reduce(int *g_id, int *g_oD) {
    unsigned int tid, s;
    __shared__ int sD[];
    tid = threadIdx.x;
    sD[tid] = g_id[tid];
    __syncthreads();
    for(s=1; s<blockDim.x; s *= 2) {
        if (tid % (2*s) == 0)
            sD[tid] += sD[tid + s];
        __syncthreads();
    }
    if (tid == 0) g_oD[blockIdx.x] = sD[0];
}
```

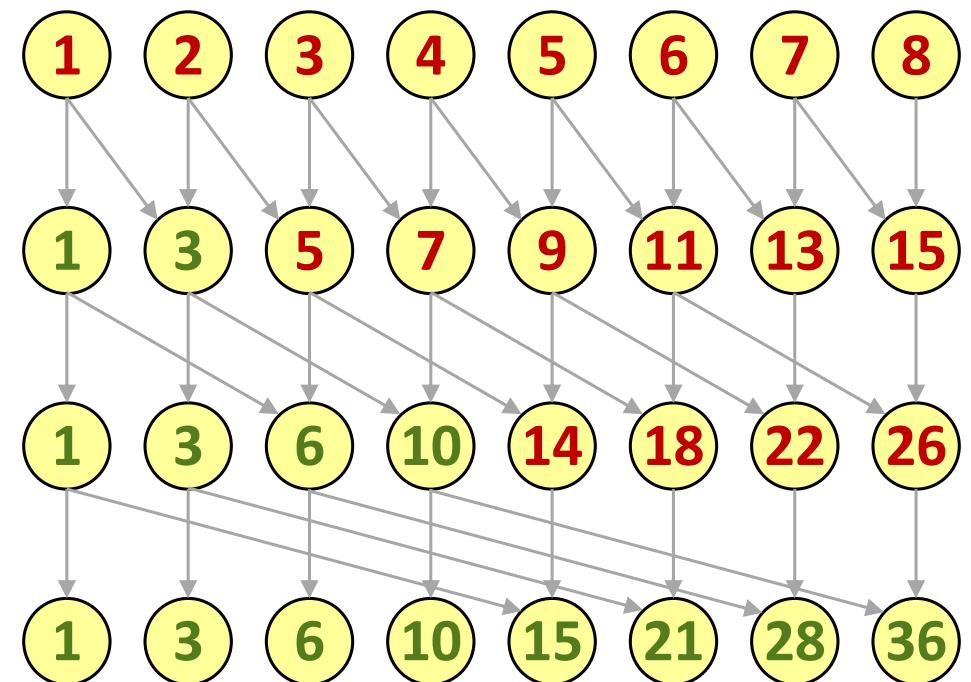
# Inclusive Scan

- Esquema típico que aparece en múltiples situaciones.
- Scan a nivel de 1 block, #threads =  $2^n = N = N_{th}$ .
- Algoritmo Sencillo

```
Out[0] = In[0];
for (i=1; i<N; i++)
    Out[i] = In[i] + Out[i-1];
```

In 1 2 3 4 5 6 7 8

Out 1 3 6 10 15 21 28 36

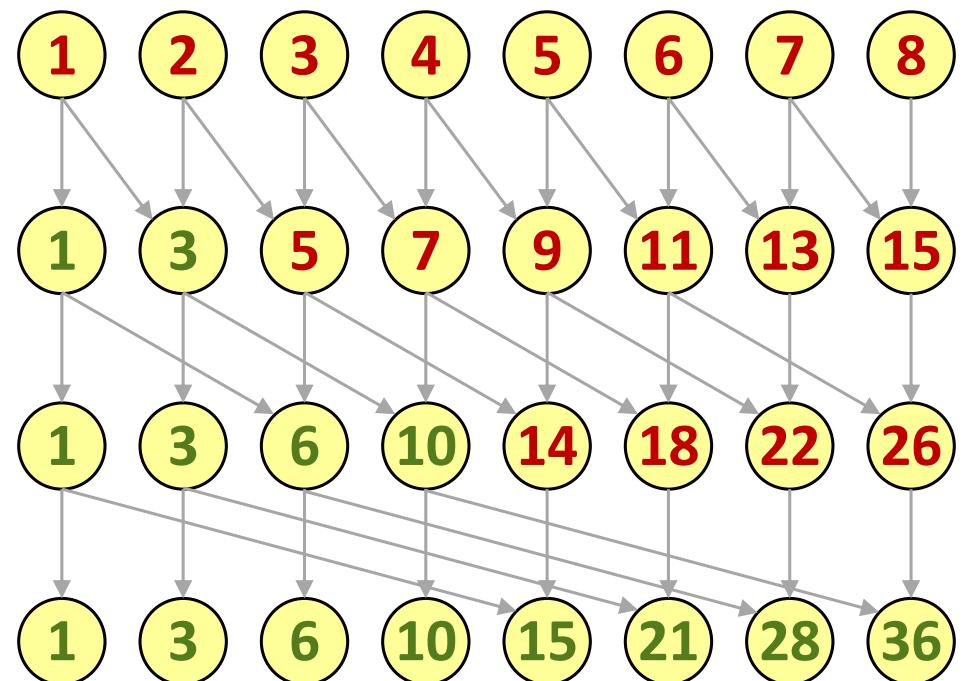


# Inclusive Scan

- Esquema típico que aparece en múltiples situaciones.
- Scan a nivel de 1 block, #threads =  $2^n = N = N_{th}$ .
- Algoritmo Sencillo

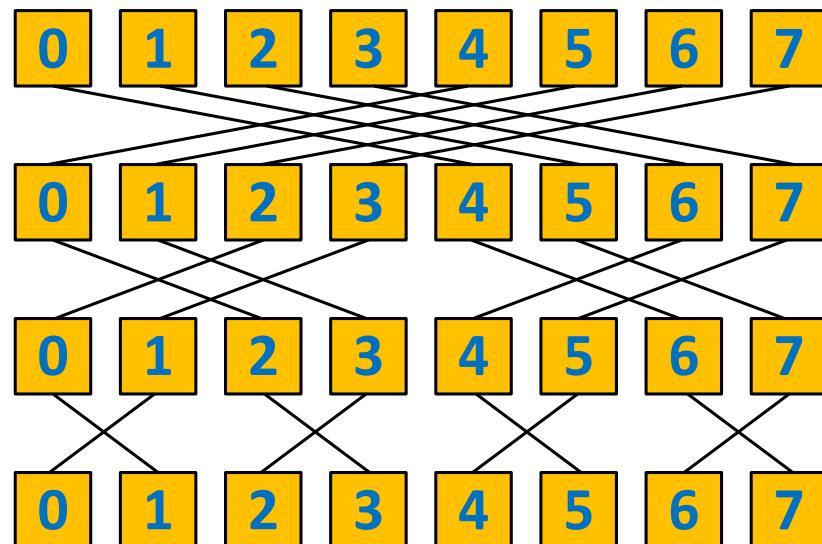
```
__global__ void Scan(float *In, float *Out) {
    __shared__ float T1[Nth];
    float tmp;
    int i = threadIdx.x;

    T1[i] = In[i]; __syncthreads();
    for (int s=1; s<Nth; s=s*2){
        if (i>=s)
            tmp = T1[i] + T1[i-s];
        else
            tmp = T1[i];
        __syncthreads();
        T1[i] = tmp; __syncthreads();
    }
    Out[i] = T1[i];
}
```



# Otro Ejemplo de Reducción

- A veces, es necesario que el resultado de la reducción llegue a todos los elementos de proceso.
- Esquema típico. A veces, es más barato hacer el cálculo que enviar el resultado.
- Reducción a nivel de block, #threads =  $2^n = N = N_{th}$ .
- Aplicaremos un esquema tipo butterfly



```
sum = 0;  
for (i=0; i<N; i++)  
    sum = sum + v[i];
```

# Otro Ejemplo de Reducción

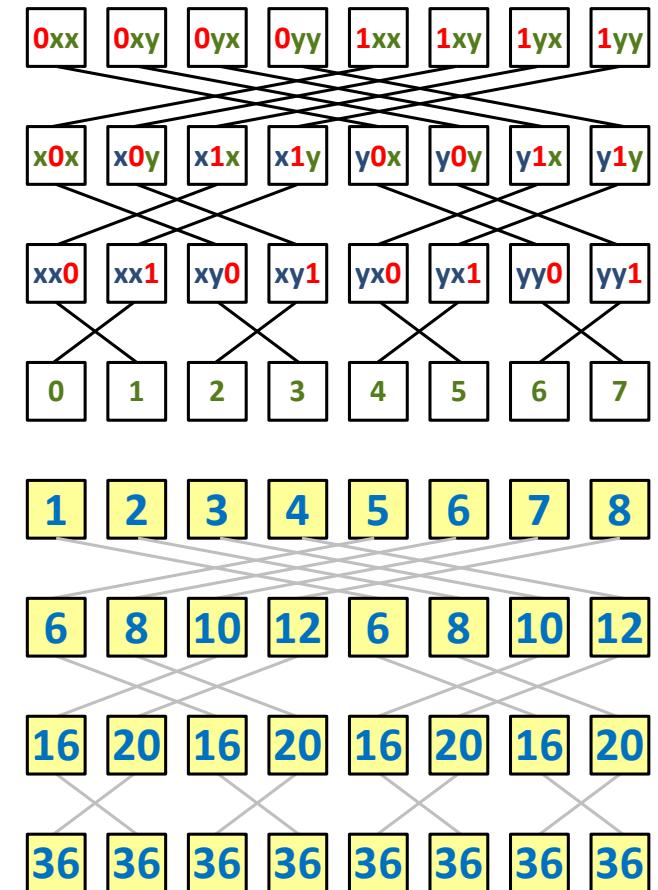
```
__global__ void MultiR (float *In, float *Out) {
    int i = threadIdx.x;

    __shared__ float sum[Nth];

    // 1 dato por thread
    sum[i] = In[i]; __syncthreads();
    for (int bit=NTh>>1; bit>0; bit >>= bit) {
        int t = sum[i] + sum[i^bit];
        __syncthreads();

        sum[i] = t; __syncthreads();
    }

    // OUTPUT: Todos los threads tienen el mismo valor
    Out[i] = sum[i];
}
```



# Histograma

- Calcular cuántas veces aparece cada carácter en un string/texto

```
void Histo(unsigned char *s, long N, unsigned long *h) {  
    for (int i=0; i<N; i++)  
        h[s[i]]++; }  
/*Invocación */ Histo(string, N, histograma);
```

```
_global_ void HistoK(unsigned char *s, long N, unsigned long *h) {  
    int i = blockIdx.x*blockDim.x + threadIdx.x;  
    int stride = blockDim.x * gridDim.x;  
    while (i<N) {  
        h[s[i]]++;  
        i = i + stride;  
    }  
}  
  
//Invocación con 256 threads por bloque y 128 bloques  
// N >> 128*256  
histoK<<<128,256>>>(string, N, histograma);
```

# Histograma

## Resultado

```
File Size: 49175956
nThreads: 256
nBlocks: 1024
Tiempo Total 57.121567 ms
Ancho de Banda 0.861 GB/s
 10 - <CR>   : 3126  1.715%
 99 - "c"     : 3086  1.693%
112 - "p"     : 3085  1.693%
101 - "e"     : 3084  1.692%
110 - "n"     : 3077  1.688%
111 - "o"     : 3077  1.688%
 50 - "2"     : 3077  1.688%
109 - "m"     : 3069  1.684%
 97 - "a"     : 3066  1.682%
108 - "l"     : 3065  1.682%
114 - "r"     : 3065  1.682%
116 - "t"     : 3065  1.682%
```

¿Qué está pasando?

## Debería dar

.	.	.		
32	- " "	:	7495904	15.243%
114	- "r"	:	4192696	8.526%
45	- "-"	:	4122644	8.383%
111	- "o"	:	2811676	5.718%
49	- "1"	:	2369971	4.819%
116	- "t"	:	2011600	4.091%
48	- "0"	:	1784628	3.629%
97	- "a"	:	1710675	3.479%
			1533544	3.118%
			1424083	2.896%
112	- "p"	:	969509	1.972%
99	- "c"	:	964174	1.961%
115	- "s"	:	917104	1.865%
10	- <CR>	:	898198	1.826%
108	- "l"	:	879686	1.789%

# Histograma

- Calcular cuántas veces aparece cada carácter en un string/texto

```
void Histo(unsigned char *s, long N, unsigned long *h) {  
    for (int i=0; i<N; i++)  
        h[s[i]]++; }  
  
/*Invocación */ Histo(string, N, histograma);
```

```
__global__ void HistoK(unsigned char *s, long N  
{ int i = blockIdx.x*blockDim.x + threadIdx.x;  
    int stride = blockDim.x * gridDim.x;  
    while (i<N) {  
        h[s[i]]++;  
        i = i + stride;  
    }  
  
    //Invocación con 256 threads por bloque y 128 bloques  
    // N >> 128*256  
histoK<<<128,256>>>(string, N, histograma);
```

El problema está aquí:

$x_1 \leftarrow s[i]$   
 $x_2 \leftarrow h[x_1]$   
 $x_2 \leftarrow x_2 + 1$   
 $h[x_1] \leftarrow x_2$

Estas 3 operaciones se han de ejecutar de forma ATÓMICA.

# Operaciones Atómicas

- Operaciones atómicas sobre enteros (32b, 64b, con signo y sin signo):
  - `atomicAdd()`, `atomicSub()`
  - `atomicExch()`
  - `atomicMin()`, `atomicMax()`
  - `atomicInc()`, `atomicDec()`
  - `atomicCAS() // compare and swap`
  - `atomicAnd()`, `atomicOr()`, `atomicXor()`
- Operaciones en memoria global y compartida (desde CUDA 2.0)
- Operación atómica: “**LEER VALOR, MODIFICAR VALOR, ESCRIBIR VALOR**”.
- El hardware se asegura de que no se accede al valor hasta que la operación atómica termine. Los accesos de los threads a la misma dirección son serializados.
- Operación atómica en memoria compartida:
  - Latencia baja, pero privada de cada thread block
- Operación atómica en memoria global:
  - Latencia muy alta (100's de ciclos), pero accesible por todos los thread blocks.

# Histograma

- Calcular cuántas veces aparece cada carácter en un string/texto

```
void Histo(unsigned char *s, long N, unsigned long *h) {  
    for (int i=0; i<N; i++)  
        h[s[i]]++; }  
  
/*Invocación */ Histo(string, N, histograma);
```

```
__global__ void HistoK(unsigned char *s, long N, unsigned long *h) {  
    int i = blockIdx.x*blockDim.x + threadIdx.x;  
    int stride = blockDim.x * gridDim.x;  
    while (i<N) {  
        atomicAdd(&h[s[i]], 1);  
        i = i + stride;  
    }  
}  
  
//Invocación con 256 threads por bloque y 128 bloques  
// N >> 128*256  
histoK<<<128,256>>>(string, N, histograma);
```

- Operación muy costosa (100's de ciclos).
- El vector h tiene 256 elementos.
- En el mejor caso, sólo puede haber 256 threads haciendo esta operación.

# Histograma

## □ Posible optimización: PRIVATIZACIÓN

```
__global__ void HistoK(unsigned char *s, long N, unsigned long *h) {  
    __shared__ unsigned long h_private[256];  
    int i = blockIdx.x*blockDim.x + threadIdx.x;  
    int stride = blockDim.x * gridDim.x;  
    h_private[threadIdx.x] = 0;  
    __syncthreads();  
    while (i < N) {  
        atomicAdd(&h_private[s[i]], 1);  
        i = i + stride;  
    }  
    __syncthreads();  
    i = threadIdx.x;  
    atomicAdd(&h[i], h_private[i]);  
}  
// Invocación con 256 threads por bloque y 128 bloques  
// N >> 128*256  
histoK<<<128,256>>>(string, N, histograma);
```

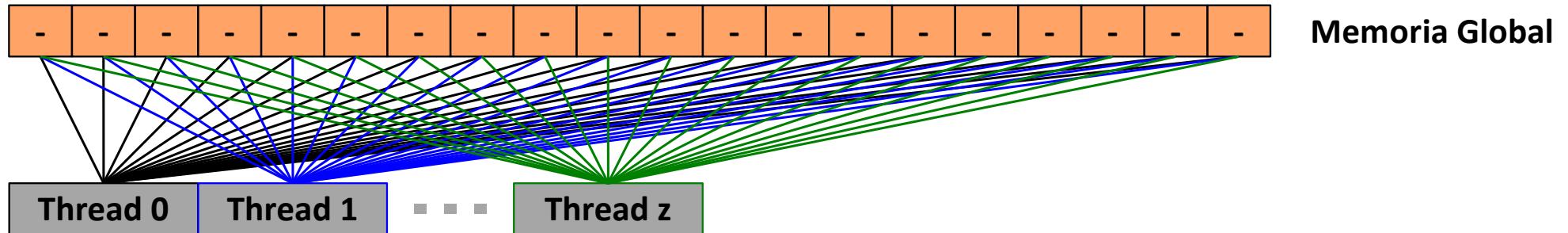
- Copia privada de cada thread block en la Memoria Compartida.
- Operación más barata (10's de ciclos).
- Cada thread block tiene su copia privada.
- Los thread blocks corren en paralelo

- Al final hay que acceder a la memoria global

# Privatización

- La privatización es una herramienta muy potente.
- Se usa muy frecuentemente.
- Necesitamos que la operación a realizar sea asociativa y conmutativa
- Necesitamos que los datos a privatizar no sean muy grandes
  - Han de caber en la memoria compartida.

# Estrategia de programación relacionada

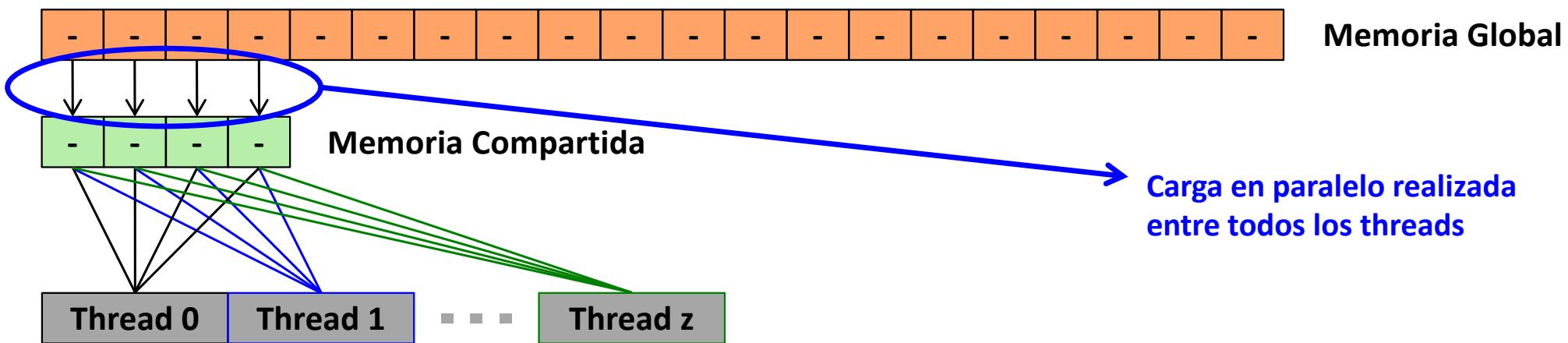


- La memoria global reside en la DRAM de la tarjeta:
  - Latencia elevada
  - Ancho de banda muy grande
- Estrategia de programación: **particionar los datos de entrada para aprovechar la velocidad de la memoria compartida**

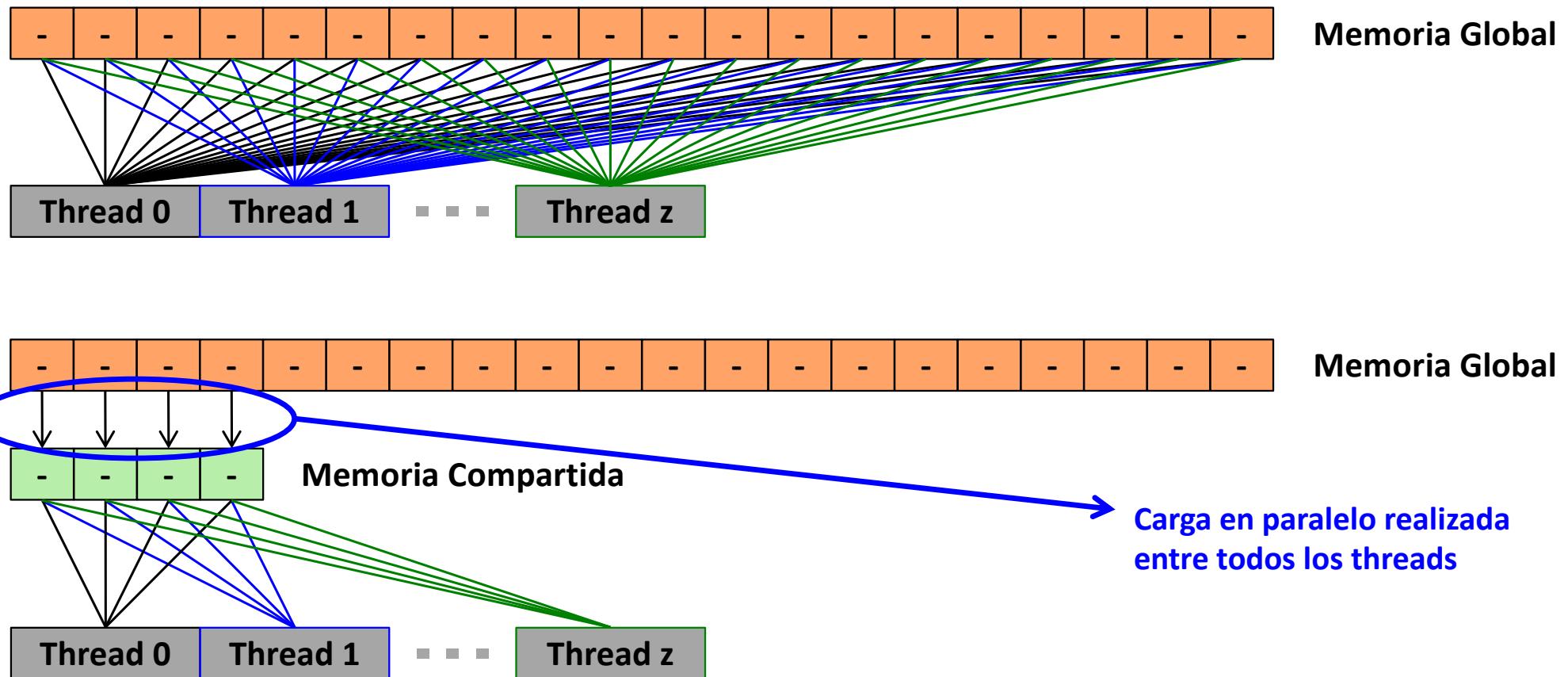
# Estrategia de programación relacionada

## Partitionar los datos de entrada para aprovechar la velocidad de la memoria compartida

- Partir los datos en subconjuntos que quepan en la memoria compartida
- Gestionar cada subconjunto de datos desde un thread block:
  - ✓ Cargar los datos utilizando múltiples threads para aprovechar el paralelismo
  - ✓ Realizar todos los cálculos que se puedan sobre los datos almacenados en la memoria compartida
  - ✓ Copiar los resultados en la memoria global

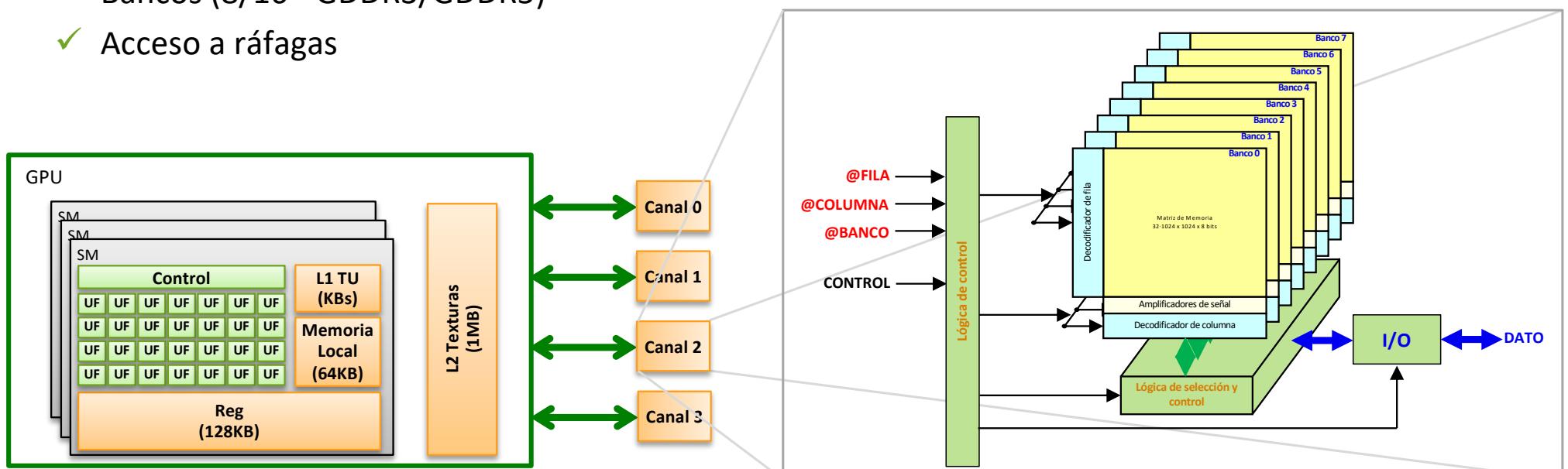


# Estrategia de programación relacionada



# Patrones de Acceso a Memoria

- ❑ La forma de acceder a memoria tiene una gran influencia en el rendimiento de una aplicación.
- ❑ Relacionado con la jerarquía de memoria de una GPU
  - Varios canales con Memoria
  - Estructura interna de una DRAM
    - ✓ Bancos (8/16 - GDDR3/GDDR5)
    - ✓ Acceso a ráfagas

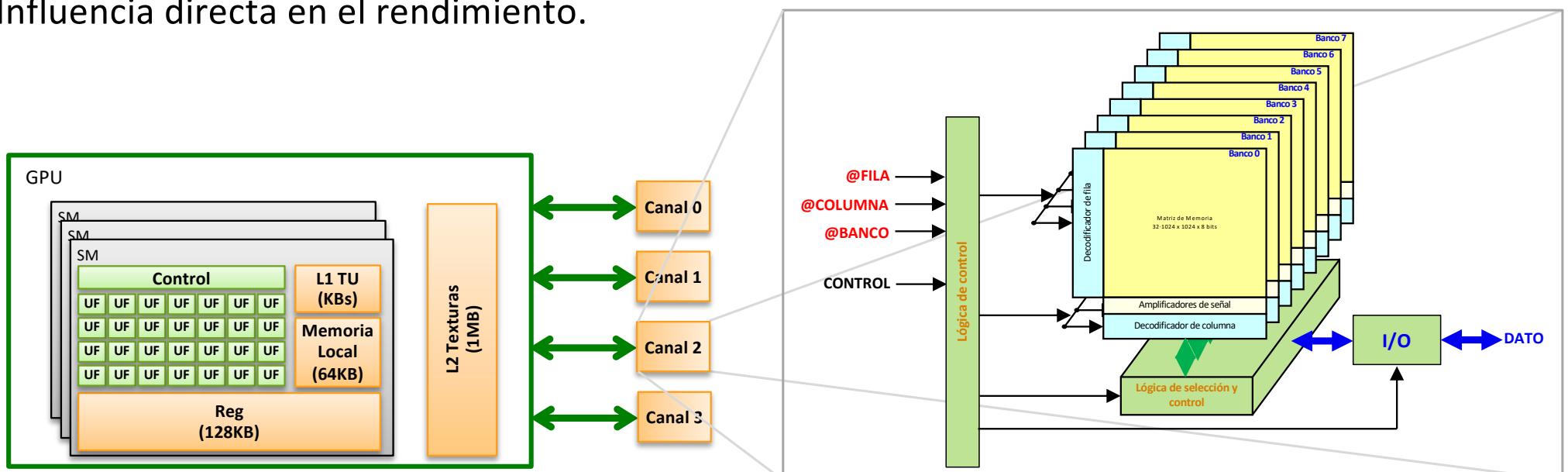


# Patrones de Acceso a Memoria

- ¿Qué bits de la dirección utilizamos para seleccionar canal, banco, ...?



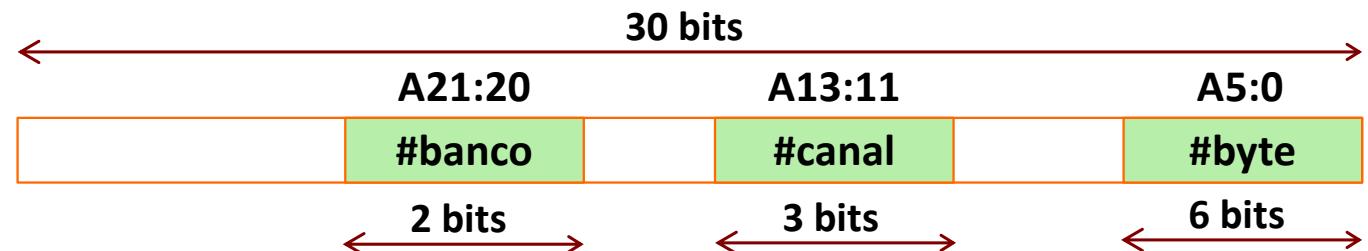
- Influencia directa en el rendimiento.



# Patrones de Acceso a Memoria

## ☐ nVIDIA GTX 280

- 30 Stream Multiprocessors (SM)
- 8 canales de memoria DDR3
- 32 bits por canal
- 4 bancos por chip
- Ráfagas de 64B
- 1 GB memoria



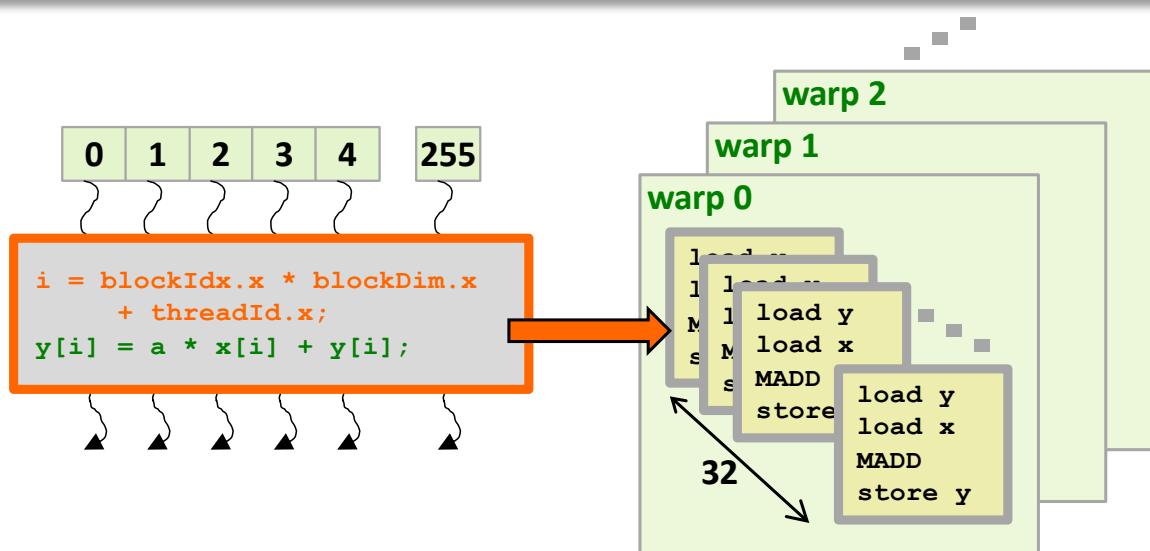
- Acceso a 1 ráfaga: 64B consecutivos en el mismo canal y banco
- Cada 2KB cambiamos de canal

**Los patrones de acceso se han de aprovechar del acceso a ráfaga y distribuirse equitativamente entre los canales para maximizar el ancho de banda.**

# Patrones de Acceso a Memoria

```
__global__
void saxpyP (int N, float a, float *x, *y) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    y[i] = a * x[i] + y[i];
}

saxpyP<<<100,256>>>(256*100, 3.5, x, y);
```



- “load y”: acceso a 128B en posiciones consecutivas de memoria:
  - 2 ráfagas 64B, 2 accesos a posiciones contiguas.
  - IMPORTANTE: la dirección de inicio de y ha de estar alineada a 128B  
[TAREA del COMPILADOR – DRIVER]
- Para conseguir buenos rendimientos es muy importante forzar las direcciones de inicio de x e y para que x[0] e y[0] estén en diferentes canales de memoria  
[TAREA del COMPILADOR – DRIVER]
- Hay que utilizar el tamaño de bloque para forzar que blocks diferentes accedan a canales diferentes  
[TAREA del PROGRAMADOR]

# Patrones de Acceso a Memoria

```
__global__ void MMkernel(float *dA, float *dB, float *dC, int N) {
    int row = . . .; int col = . . .;
    float tmp = 0.0;
    for (int k=0; k<N; k++)
        tmp += dA[row*N+k] * dB[k*N+col];
    dC[row*N+col] = tmp;
}

// Invocación
MMkernel<<<Grid, Block>>>(dA, dB, dC, 1024);
```

¡ Acceso a posiciones NO consecutivas !

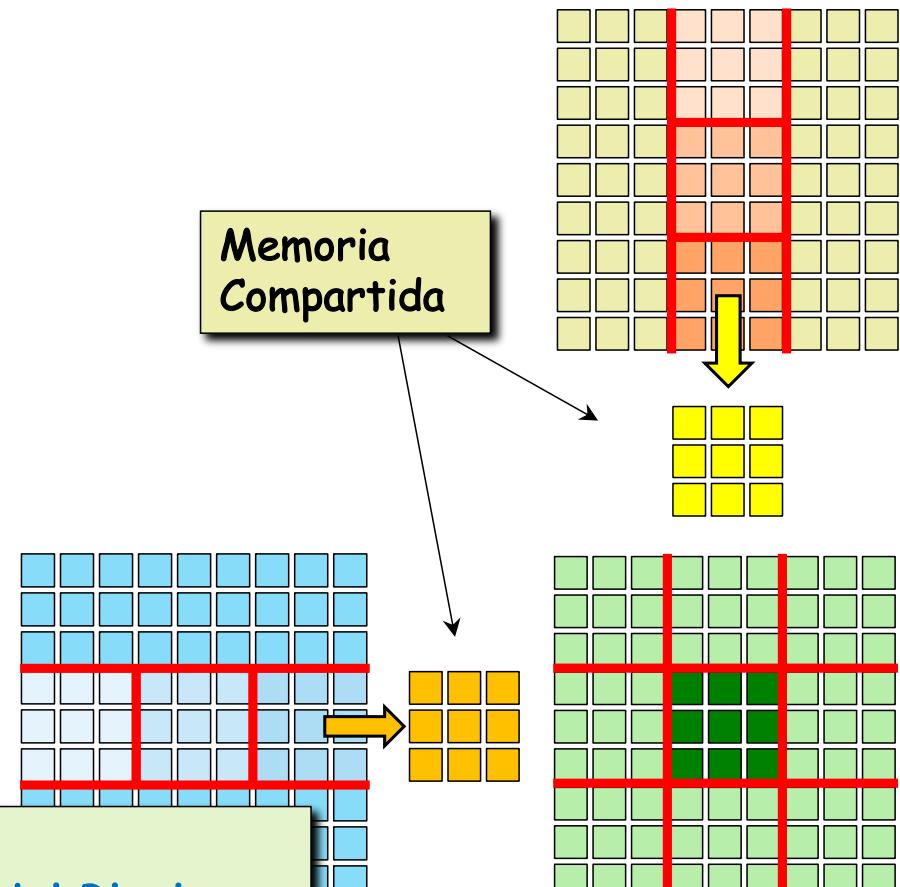
¡ Acceso a posiciones consecutivas !

- Acceso a posiciones consecutivas: aprovechamos el eficiente acceso a ráfagas de las DRAM [acceso *coalesced*, terminología nVIDIA]
- Acceso a posiciones NO consecutivas
  - @x, @x+4·N, @x+8·N, ...
  - Accesos muy ineficientes, sólo aprovechamos un dato por ráfaga.
  - Se puede evitar este problema haciendo una copia de la matriz **traspuesta**.

# Patrones de Acceso a Memoria

```
...
int tx = threadIdx.x; int ty = threadIdx.y;
int row = blockIdx.y * M + ty;
int col = blockIdx.x * M + tx;
float tmp = 0.0;
for (m=0; m < (N/size); m++) {
    sA[ty][tx] = dA[row*N + m*size + tx];
    sB[ty][tx] = dB[col + (m*size + ty)*N];
    __syncthreads();
    for (int k=0; k<size; k++)
        tmp += sA[ty][k] * sB[k][tx];
    __syncthreads();
}
dC[row*N+col] = tmp;
...
```

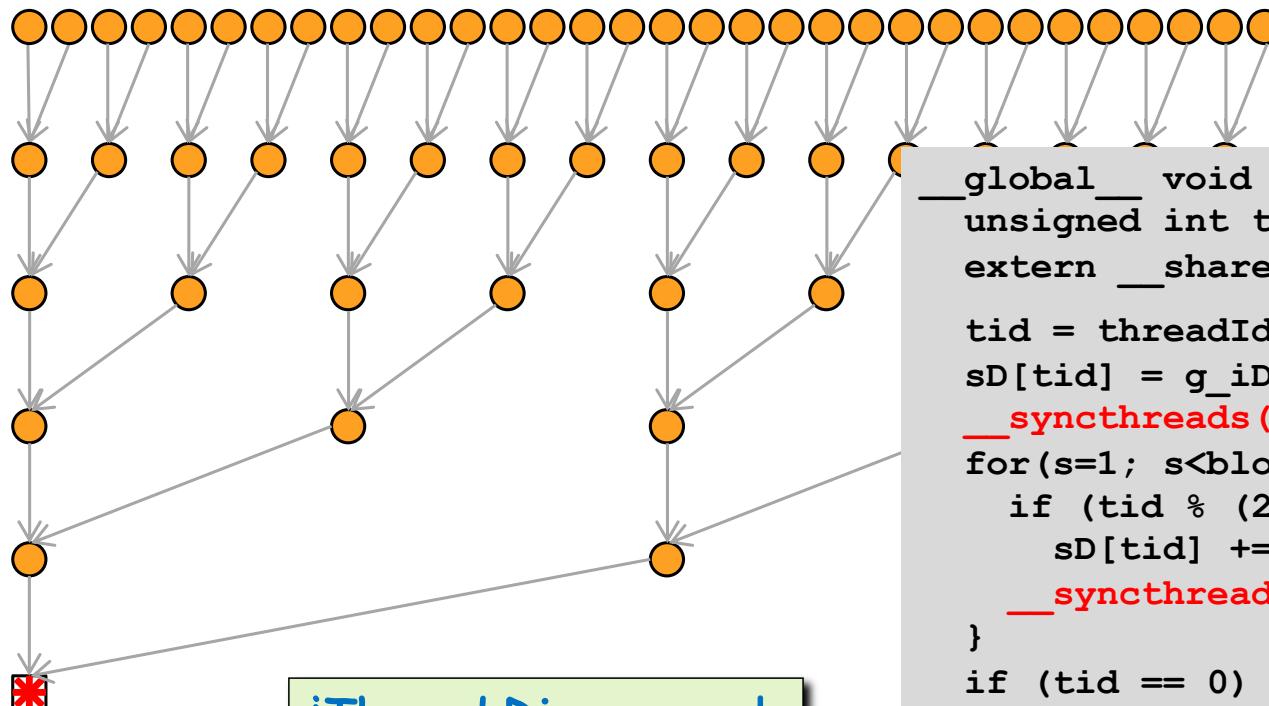
¡ Acceso a posiciones consecutivas !  
La copia se realiza entre todos los threads del Block



# Patrones de Acceso a Memoria

## Revisitando la reducción paralela

- El patrón de acceso a memoria no se ajusta bien a los warps

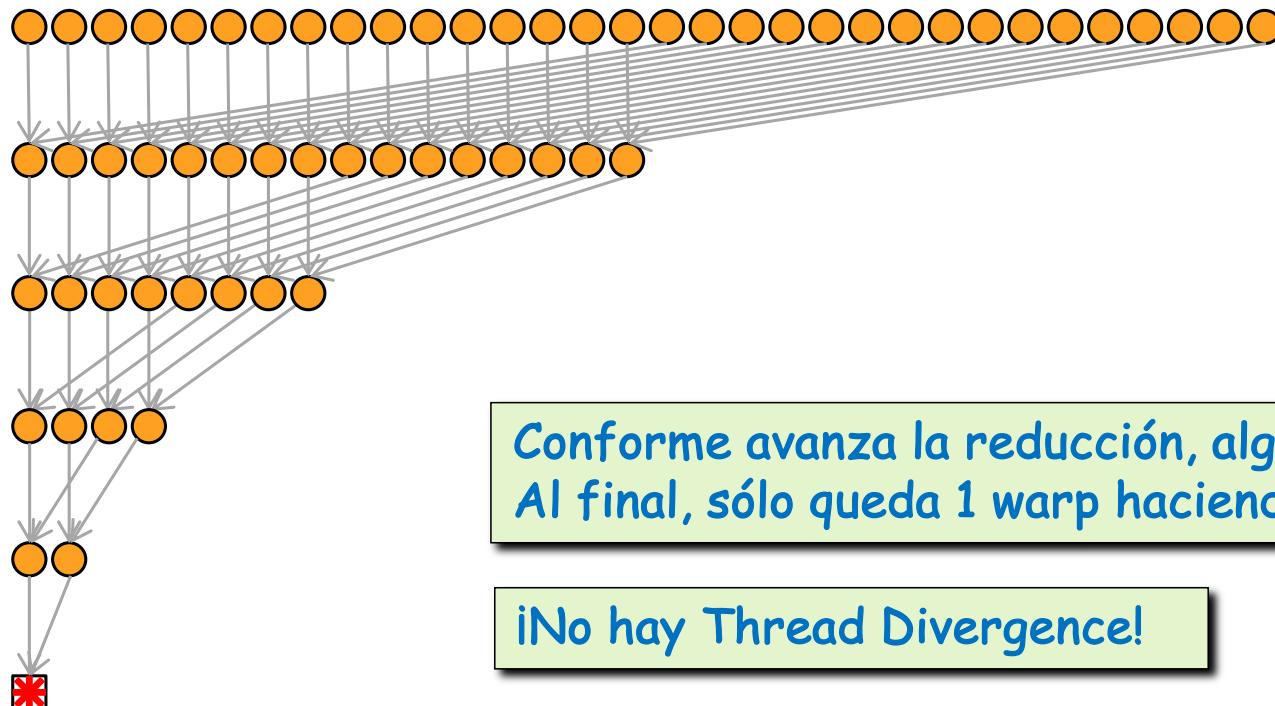


```
__global__ void reduce(int *g_iD, int *g_oD) {  
    unsigned int tid, s;  
    extern __shared__ int sD[];  
  
    tid = threadIdx.x;  
    sD[tid] = g_iD[tid];  
    __syncthreads();  
    for(s=1; s<blockDim.x; s *= 2) {  
        if (tid % (2*s) == 0)  
            sD[tid] += sD[tid + s];  
        __syncthreads();  
    }  
    if (tid == 0) g_oD[blockIdx.x] = sD[0];  
}
```

# Patrones de Acceso a Memoria

## Revisitando la reducción paralela

- Este patrón de acceso a memoria se ajusta mucho mejor a los warps



Conforme avanza la reducción, algunos warps dejan de trabajar.  
Al final, sólo queda 1 warp haciendo la reducción.

¡No hay Thread Divergence!

# Compilando un programa CUDA

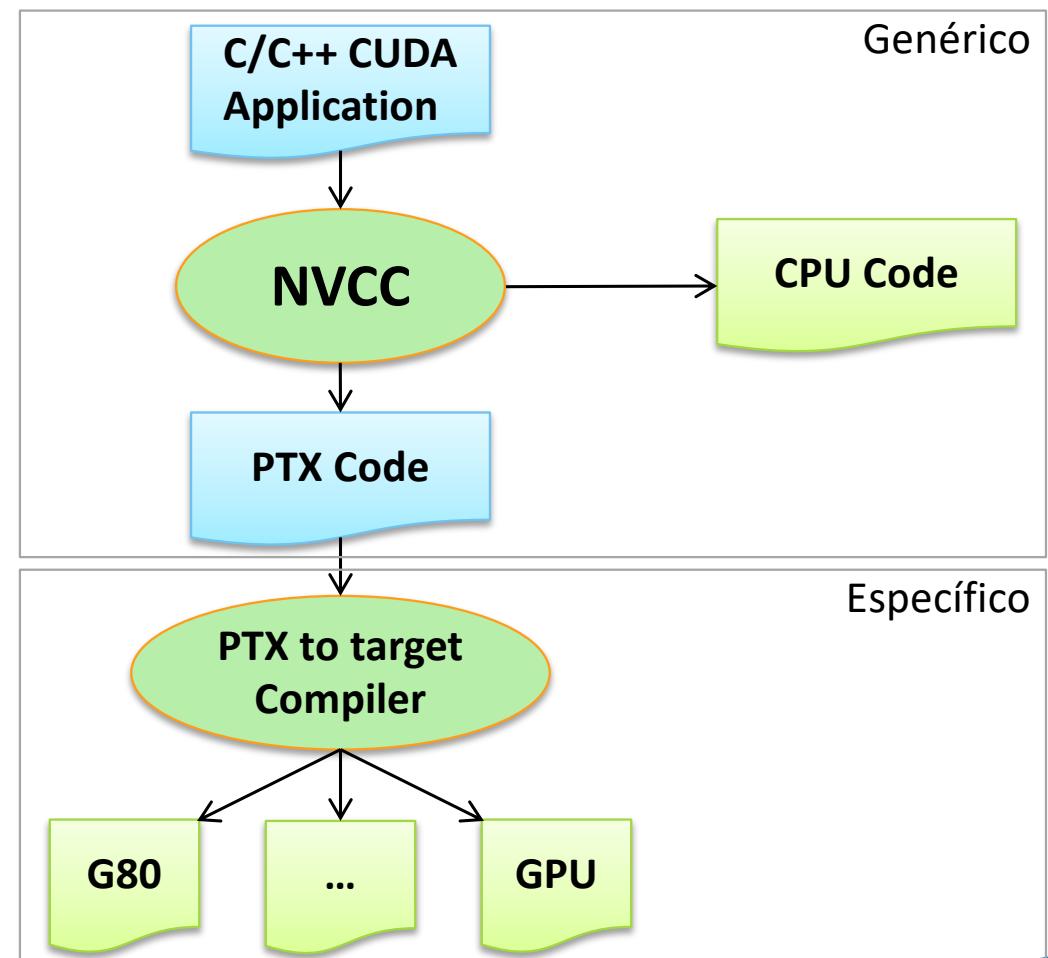
□ Cualquier programa con extensiones CUDA ha de ser compilado con NVCC

- NVCC separa el código que se ejecuta en el host del código que se ejecuta en el device.

> `nvcc MiPrograma.cu`

□ El código del device se obtiene en 2 pasos:

- NVCC genera código intermedio PTX (Parallel Thread eXecution)
- El código PTX se compila en tiempo de ejecución (Just in time) para obtener el binario específico de cada device.



# Lenguaje CUDA: C con extensiones mínimas

- Especificadores que indican dónde están las cosas

```
_global_ void Kernel(. . .); // función, se ejecuta en el device  
_device_ int GlobalVar; // variable en la memoria del device  
_shared_ int SharedVar; // variable en la memoria compartida
```

Declaración	Ejecutado en	Llamado desde
<b>_device_</b> float DeviceFunc()	DEVICE	DEVICE
<b>_global_</b> void KernelFunc()	DEVICE	HOST
<b>_host_</b> float HostFunc()	HOST	HOST

- Extensión que permite invocar kernel paralelo

```
Kernel<<<500, 128>>>(...); // lanza 500 blocks con 128 threads cada uno
```

- Variables especiales PREDEFINIDAS para identificar los threads

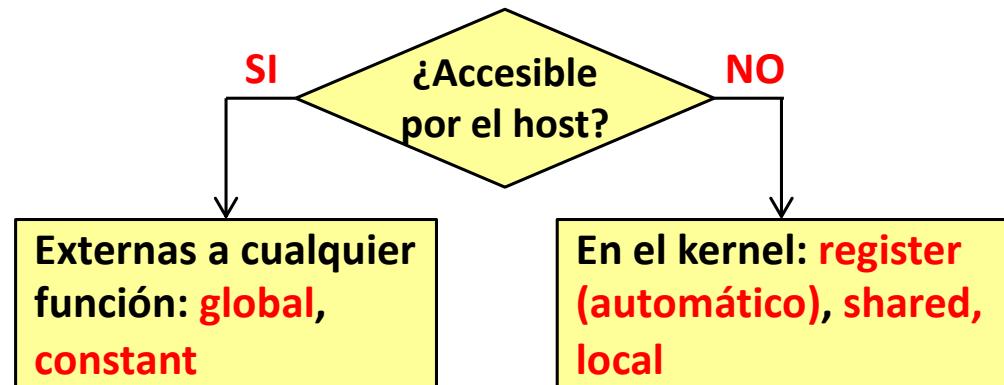
```
dim3 threadIdx; // número de thread dentro del block  
dim3 blockIdx; // número de block dentro del grid  
dim3 blockDim; // Dimensiones del block  
dim3 gridDim; // Dimensiones del grid
```

# Calificadores de variable

Declaración	Memoria	Ámbito	Duración
int LocalVar;	registros	thread	thread
<code>__device__ __shared__</code> int SharedVar;	shared	block	block
<code>__device__</code> int GlobalVar;	global	grid	aplicación
<code>__device__ __constant__</code> int ConstVar;	constant	grid	aplicación

- `__device__` es opcional cuando se usa con `__shared__` o `__constant__`
- Si no se utiliza calificador, las variables residen en registros.
  - Excepto los arrays per-thread que residen en memoria global.

¿dónde declaramos las variables?



# API de CUDA

## □ La API (Application Programming Interface) de CUDA es una extensión del Lenguaje de Programación C.

- Extensiones del lenguaje para indicar qué partes del código se ejecutan en el device
- Una librería dinámica que ofrece:
  - ✓ Una parte común que proporciona nuevos tipos de datos (vectores de tipos simples) y un conjunto de rutinas que corren en el host y en el device.
  - ✓ Una parte que corre en el host para controlar y acceder al device (o devices).
  - ✓ Una parte que corre en el device con funciones específicas.

## □ Funciones matemáticas (lista no exhaustiva):

- pow, sqrt, exp, exp2, log, log2, sin, cos, sinh, cosh, ceil, floor, ...
  - ✓ Si se ejecutan en el host, se utiliza la librería de C (si está disponible).
  - ✓ Estas funciones sólo están disponibles para escalares (no para vectores)
- Algunas de estas funciones tienen una versión para el device más rápida, pero con menos precisión:
  - ✓ \_\_pow, \_\_log, \_\_log2, \_\_exp, \_\_sin, \_\_cos

# Consideraciones sobre coma flotante

□ Usa coma flotante definida por IEEE-754:

- (S signo, E exponente [exceso], M mantisa)
- Valor =  $(-1)^S \cdot M \cdot 2^E$ , donde  $1.0 \leq M < 10.0$  (todas las mantisas son 1.xxx, bit escondido)
- Exponente = 1111...111 (reservado)
- Valores especiales

Exponente	Mantisa	Significado	Notas
111...11	$\neq 0$	NaN	Operar con val. no inicializados; $0/0$ ; $0 \cdot \infty$ ; $\infty/\infty$ ; $\infty - \infty$
111...11	$= 0$	$(-1)^S \cdot \infty$	$x/0 = \infty$ ; $x/\infty = 0$ ; $\infty \cdot \infty = \infty$ ; $\infty + \infty = \infty$
000...00	$\neq 0$	denormal	
000...00	$= 0$	0	

# Consideraciones sobre coma flotante

## IEEE-754

### **float** (precisión simple, 32b, 4B)

- 1 bit de signo, 8 bits de exponente (exceso 127) y 23 bits de mantisa
- Rango:
  - ✓  $\pm 1,18 \cdot 10^{-38} \leq x \leq \pm 3,4 \cdot 10^{38}$
  - ✓ aproximadamente 7 dígitos decimales de precisión

### **double** (precisión doble, 64b, 8B)

- 1 bit de signo, 11 bits de exponente (exceso 1023) y 52 bits de mantisa
- Rango:
  - ✓  $\pm 2,23 \cdot 10^{-308} \leq x \leq \pm 1,8 \cdot 10^{308}$
  - ✓ aproximadamente 15 dígitos decimales de precisión

# Consideraciones sobre coma flotante

## IEEE-754 Números denormales

- En muchos modelos físicos se trabaja con valores muy cercanos a cero.
- Sin la denormalización se crearían “artefactos” que comprometen la integridad numérica de los modelos físicos.
- La denormalización consiste en no aplicar la normalización de la mantisa (1.xxx) cuando trabajemos con números cercanos a cero (exponente = 000...00).
- Los números denormales son:
  - **float**:  $(S)^{-1} \cdot 0.M \cdot 2^{-127}$
  - **double**:  $(S)^{-1} \cdot 0.M \cdot 2^{-1023}$

# Consideraciones sobre coma flotante

## Precisión y redondeo

- La precisión de una operación en coma flotante se mide por el máximo error introducido por una operación.
- La mayoría de errores se producen porque el resultado de una operación no es exacto o porque no hay suficientes bits para representarlo.
- En muchos casos la fuente de errores es el propio hardware por utilizar menos bits de los necesarios, por razones de velocidad o de ahorro de área de chip.
- Una forma de medir el error es el ULP (Units in the Last Place).
  - En un sumador perfecto, el error es como máximo 0,5 ULP.
- Las operaciones en coma flotante no son estrictamente asociativas:
  - Ejemplo: 1 bit signo, 2 bits exponente, 2 bits mantisa
  - Suma secuencial:  $1.00 \cdot 2^0 + 1.00 \cdot 2^0 + 1.00 \cdot 2^{-2} + 1.00 \cdot 2^{-2} = 1.00 \cdot 2^1 + 1.00 \cdot 2^{-2} + 1.00 \cdot 2^{-2}$   
 $= 1.00 \cdot 2^1 + 1.00 \cdot 2^{-2} = \textcolor{green}{1.00 \cdot 2^1}$
  - Suma paralela:  $(1.00 \cdot 2^0 + 1.00 \cdot 2^0) + (1.00 \cdot 2^{-2} + 1.00 \cdot 2^{-2}) = 1.00 \cdot 2^1 + 1.00 \cdot 2^{-1} = \textcolor{red}{1.01 \cdot 2^1}$

# Consideraciones sobre coma flotante

## Runtime Math Library

- Dos tipos de funciones matemáticas:
  - `__func()`: directamente hardware de la tarjeta
    - ✓ Rápidas, pero poca precisión (`expf()`, 2 ulp; `tanf`, 4 ulp; ...)
  - `func()`: compilado en múltiples instrucciones
    - ✓ Lentas, pero con más precisión
  - Si usamos el flag `-use_fast_math` todas las funciones serán del tipo `__func()`

## Implementación CUDA IEEE-754

- Suma y multiplicación cumplen el estándar IEEE754 (Error máximo: 0.5 ULP)
- La división no cumple el estándar (2 ULP)
- No soporta todos los modos de redondeo
- No tiene mecanismos para detectar excepciones en coma flotante

# Jugando con la memoria

```
void function Traductor(unsigned char v[], int N) {  
    int i;  
    for (i=0; i<N; i++)  
        if (v[i] >= 'a' && v[i] <= 'z')  
            v[i] = v[i] - 'a' + 'A';  
}
```

Cambia minúsculas por mayúsculas.

iCódigo POCO EFICIENTE!

```
void function Traductor(unsigned char v[], unsigned char trad[256], int N) {  
    int i;  
    for (i=0; i<N; i++)  
        v[i] = trad[v[i]];  
    for (i=0; i<256; i++) trad[i] = i;  
    for (i='a'; i<= 'z'; i++)  
        trad[i] = i - 'a' + 'A';
```

No hay saltos, ni condicionales, ni operaciones aritméticas. Coste: 1 acceso a memoria.

# Jugando con la memoria

No hay saltos, ni condicionales, ni operaciones aritméticas. Coste: un acceso a memoria ....



NO HAY SALTOS, NI CONDICIONALES, NI OPERACIONES ARITMÉTICAS. COSTE: UN ACCESO A MEMORIA ....

```
__global__ void KernelGlobal(int N, unsigned char *data, unsigned char *trad) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;
    while (i < N) {
        data[i] = trad[data[i]];
        i = i + stride;
    }
}
```

Ancho de Banda 18.944 GB/s

# Jugando con la memoria

No hay saltos, ni condicionales, ni operaciones aritméticas. Coste: un acceso a memoria ....



Memoria Compartida



NO HAY SALTOS, NI CONDICIONALES, NI OPERACIONES ARITMÉTICAS. COSTE: UN ACCESO A MEMORIA ....

```
__global__ void KernelShared(int N, unsigned char *data, unsigned char *trad) {  
    __shared__ unsigned int strad[256];  
    int i = blockIdx.x * blockDim.x + threadIdx.x;  
    int stride = blockDim.x * gridDim.x;  
  
    strad[threadIdx.x] = trad[threadIdx.x];  
    __syncthreads();  
    while (i < N) {  
        data[i] = strad[data[i]];  
        i = i + stride;  
    }  
}
```

Ancho de Banda 81.486 GB/s

# Uso de la Constant Memory

## □ Datos que no se modifican en el kernel

- El host se encarga de la inicialización.
- Usa `cudaMemcpyToSymbol` para copiar la variable en la memoria del device
- Esta función indica al device que NO será modificada por el kernel.
- Se puede utilizar la cache sin problemas de coherencia

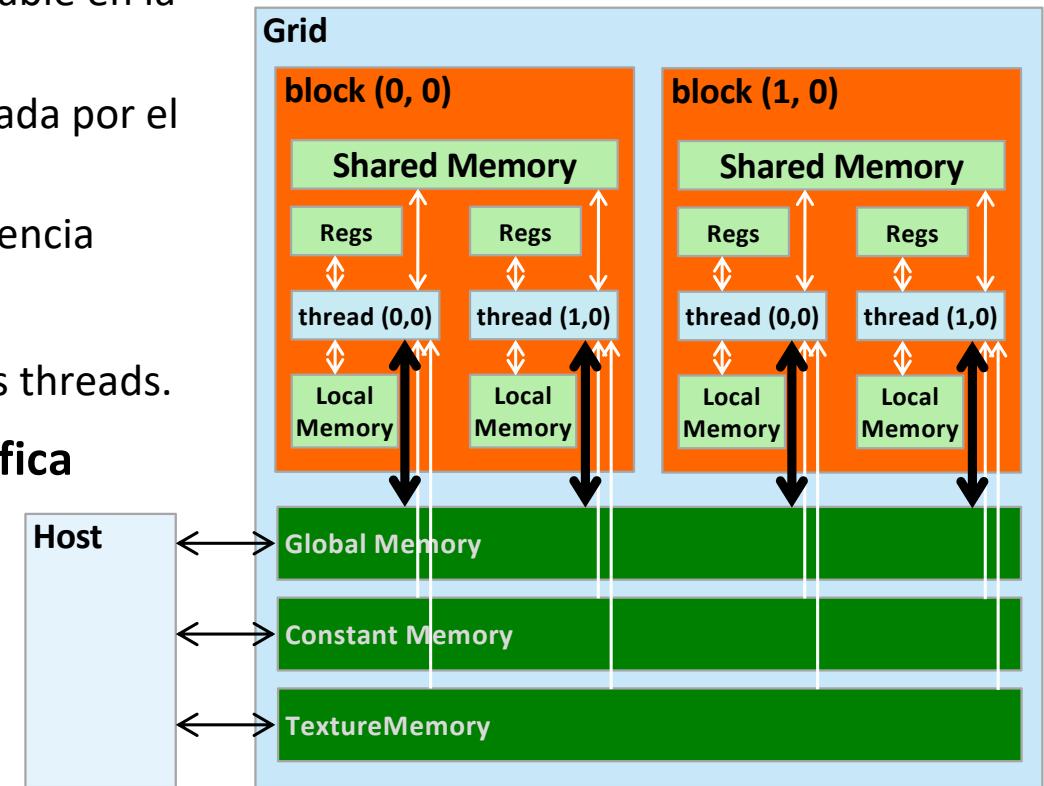
## □ Todos los SM tienen una L1 cache.

- Baja latencia y alto ancho de banda para todos los threads.

## □ Sin embargo, no se controla si un thread modifica estos datos en otro SM.

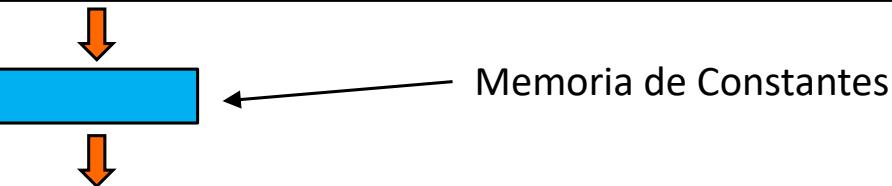
- No hay control de coherencia

**iNo es problema si los datos no son modificados por un kernel!**



# Jugando con la memoria

No hay saltos, ni condicionales, ni operaciones aritméticas. Coste: un acceso a memoria ....



```
__constant__ unsigned char dTrad[256];

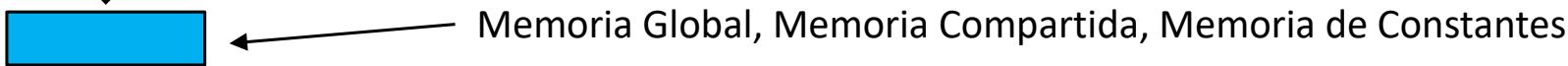
__global__ void KernelConst(int N, unsigned char *data) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;
    while (i < N) {
        data[i] = dTrad[data[i]];
        i = i + stride;
    }
}

cudaMemcpyToSymbol(dTrad, hTrad, 256);
```

Ancho de Banda 45.045 GB/s

# Jugando con la memoria, Rendimientos

No hay saltos, ni condicionales, ni operaciones aritméticas. Coste: un acceso a memoria ....



NO HAY SALTOS, NI CONDICIONALES, NI OPERACIONES ARITMÉTICAS. COSTE: UN ACCESO A MEMORIA ....

KERNEL Global

Vector Size: 49175956

nThreads: 256

nBlocks: 1024

Tiempo Total 7.787744 ms

Ancho de Banda 18.944 GB/s

TEST PASS

Time seq: 56.761963 ms

KERNEL Shared

Vector Size: 49175956

nThreads: 256

nBlocks: 1024

Tiempo Total 1.810464 ms

Ancho de Banda 81.486 GB/s

TEST PASS

Time seq: 57.536072 ms

KERNEL Constante

Vector Size: 49175956

nThreads: 256

nBlocks: 1024

Tiempo Total 3.275104 ms

Ancho de Banda 45.045 GB/s

TEST PASS

Time seq: 57.405029 ms

# Sincronización en el Host

## La ejecución de los kernels es ASÍNCRONA

- Una vez lanzado un kernel, el control es devuelto al host inmediatamente.
- El kernel se inicia cuando todas las llamadas CUDA previas se han acabado.

## La llamada `cudaMemcpy()` es SÍNCRONA.

- Se devuelve el control a la CPU cuando la copia termina.
- La copia se inicia cuando todas las llamadas CUDA previas se han acabado.

## Tenemos la posibilidad de realizar transferencias ASÍNCRONAS.

- La llamada es `cudaMemcpyAsync()` con los mismos parámetros que `cudaMemcpy()`.
- Una vez lanzada la transferencia, el control es devuelto al host inmediatamente.

## `cudaThreadSynchronize()`

- Bloquea la CPU hasta que todas las llamadas CUDA previas terminen.

# CUDA Streams

□ Objetivo: ejecutar varias operaciones CUDA simultáneamente:

- CUDA `kernel<<<>>`
- `cudaMemcpyAsync (HostToDevice)`
- `cudaMemcpyAsync (DeviceToHost)`
- Cálculos en la CPU

□ Stream

- Secuencia de operaciones que se ejecutan en la GPU en el orden en que se lanzan.

□ Las operaciones CUDA de diferentes streams pueden ejecutarse concurrentemente.

□ Las operaciones CUDA de diferentes streams pueden entrelazarse.

# Ejemplo Revisitado: Nsaxpy

```
N = 1024 * 1024 * 16
numBytes = N * sizeof(float);

// Obtener Memoria en el host
// Obtener Memoria en el device

// Copiar datos desde el host en el device
cudaMemcpy(d_x, h_x, numBytes, cudaMemcpyHostToDevice);
cudaMemcpy(d_y, h_y, numBytes, cudaMemcpyHostToDevice);

// Ejecutar el kernel
NsaxpyP<<<N/256, 256>>>(N, 3.5, d_x, d_y);

// Obtener el resultado desde el host
cudaMemcpy(h_y, d_y, numBytes, cudaMemcpyDeviceToHost);

// Liberar Memoria del device
```

```
_global_ void NsaxpyP ( . . . )
{ int i = blockDim.x*blockDim.x + threadIdx.x;
  for (int j=0; j<100; j++)
    y[i] = a * x[i] + y[i];
}
```

# CUDA Streams

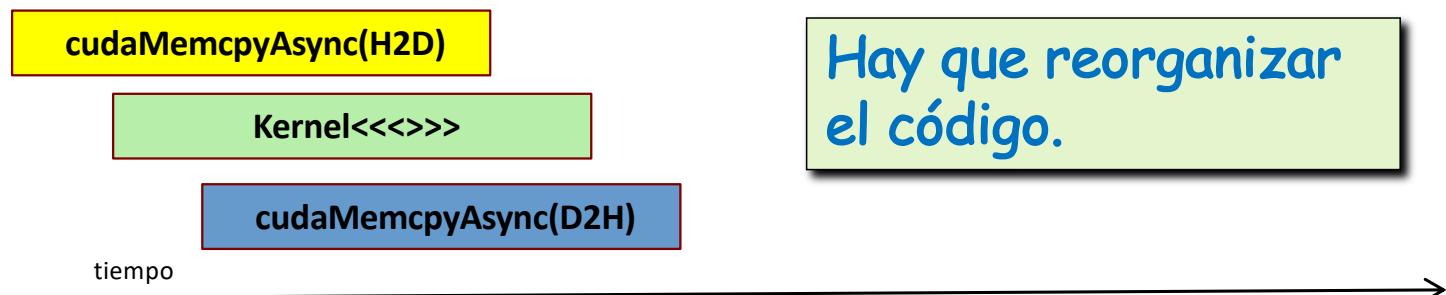
Ejecución secuencial



==16978== Profiling result:

Time (%)	Time	Calls	Avg	Min	Max	Name
45.65%	100.150ms	2	50.076ms	49.943ms	50.209ms	[CUDA memcpy HtoD]
33.40%	73.276ms	1	73.276ms	73.276ms	73.276ms	[CUDA memcpyDtoH]
20.95%	45.953ms	1	45.953ms	45.953ms	45.953ms	NsaxpyP()

Objetivo



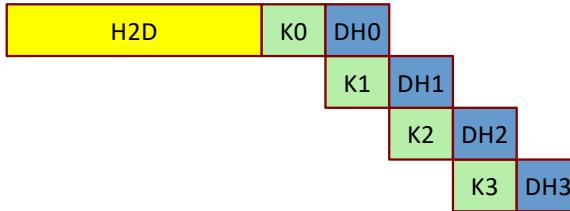
# CUDA Streams

## □ Múltiples grados de concurrencia

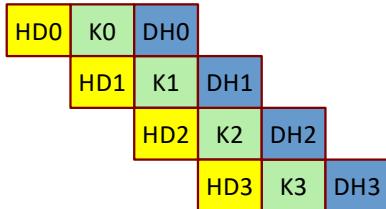
Serie (1x)



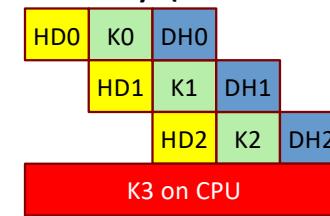
2-way (hasta 2x)



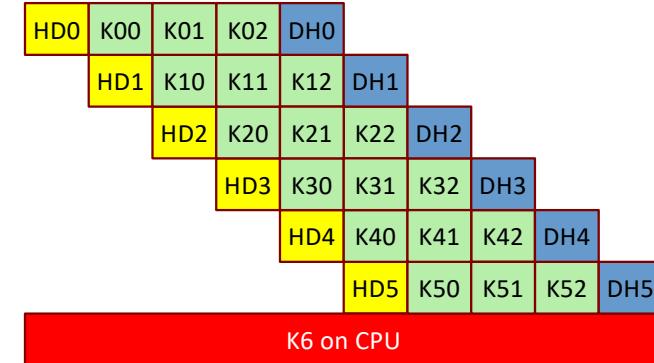
3-way (hasta 3x)



4-way (hasta 4x)



4+ way



# CUDA Streams

- Stream 0. Es el usado por defecto, si no se indica lo contrario.

- Completamente síncrono.

- Definición e inicialización de un stream:

```
cudaStream_t strm;  
cudaStreamCreate(&strm);
```

- Es necesario utilizar “**pinned memory**”

```
cudaMallocHost(&h_data, size);
```

- Se asigna memoria NO paginada
  - x2 más rápido que si usamos **malloc()** [asigna memoria paginada]

- Las transferencias han de ser asíncronas (en los 2 sentidos)

```
cudaMemcpyAsync(d1, h1, size, cudaMemcpyHostToDevice, strm);  
cudaMemcpyAsync(h2, d2, size, cudaMemcpyDeviceToHost, strm);
```

# CUDA Streams

- En la invocación del kernel, se ha indicar el stream utilizado:

```
KERNEL<<<grid, block, 0, strm>>>(parametros) ;
```

- Al acabar la aplicación hay que destruir el stream:

```
cudaStreamDestroy(strm) ;
```

- **OBJETIVO. Ejecutar concurrentemente:**

- 1 Kernel de CUDA
- Transferencia de información entre CPU y GPU
- Transferencia de información entre GPU y CPU
- Y si es posible, cálculos en la CPU

# Ejemplos

```
cudaMalloc(&dev1, size);
double* host1 = (double*) malloc(&host1, size);
...
cudaMemcpy(dev1, host1, size, H2D);
kernel2 <<<grid,block,0>>>( ..., dev2, ... );
kernel3 <<<grid,block,0>>>( ..., dev3, ... );
cudaMemcpy (host4, dev4, size, D2H) ;
...
```

**COMPLETAMENTE SÍNCRONO**  
Todas las operaciones en el stream0 son síncronas.

```
cudaMalloc(&dev1, size);
double* host1 = (double*) malloc(&host1, size);
...
cudaMemcpy (dev1, host1, size, H2D);
kernel2 <<<grid,block,0>>>( ..., dev2, ... );
CPU_code();
kernel3 <<<grid,block,0>>>( ..., dev3, ... );
cudaMemcpy (host4, dev4, size, D2H) ;
...
```

Los kernels son asíncronos con el host por defecto

Possible solapamiento

# CUDA Streams

```
cudaStream_t stream1, stream2, stream3, stream4;  
cudaStreamCreate (&stream1) ;  
...  
cudaMalloc(&dev1, size);  
cudaMallocHost(&host1, size);  
...  
cudaMemcpyAsync(dev1, host1, size, H2D, stream1);  
kernel2<<<grid, block, 0, stream2>>>();  
kernel3<<<grid, block, 0, stream3>>>();  
cudaMemcpyAsync(host4, dev4, size, D2H, stream4);  
some_CPU_method ();  
.  
COMPLETAMENTE ASÍNCRONO / CONCURRENTE
```

Pinned memory

Possible overlapping

## Mecanismos de Sincronización:

- `cudaDeviceSynchronize()`, bloquea el host hasta que todas las llamadas CUDA lanzadas terminen.
- `cudaStreamSynchronize(streamid)`, bloquea el host hasta que todas las llamadas CUDA lanzadas en el streamid terminen.

# CUDA Stream Scheduling

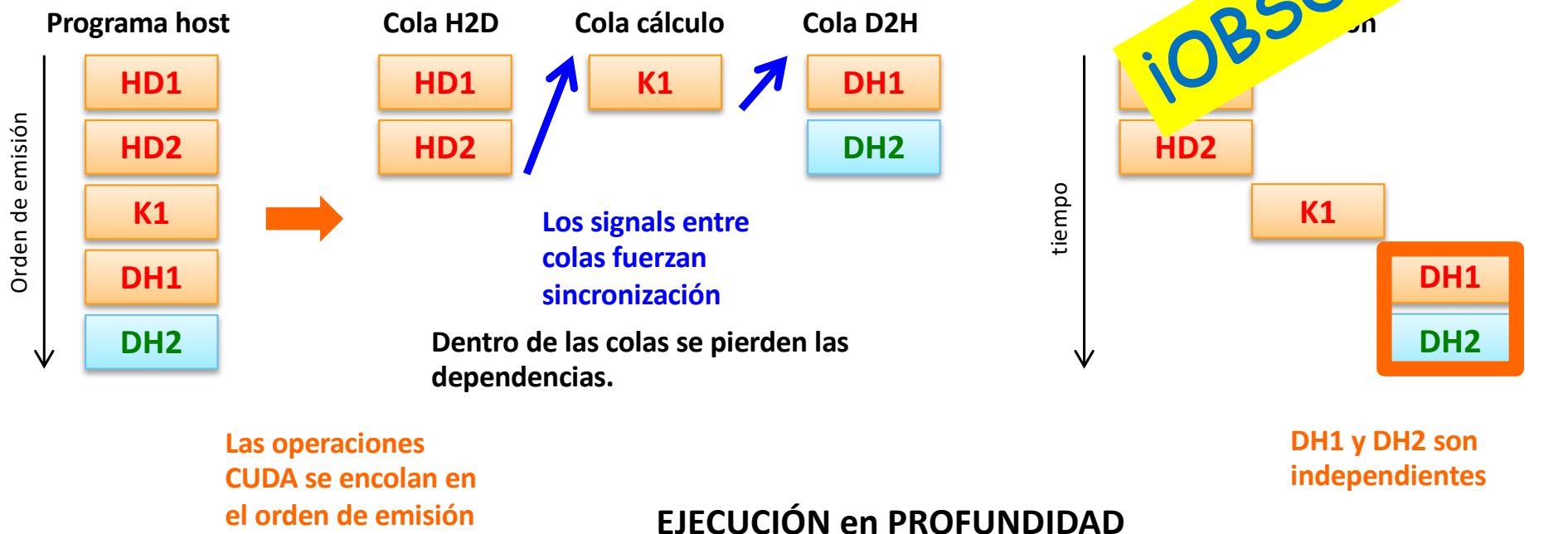
- La arquitectura Fermi tiene 3 colas:
  - 1 cola para cálculo
  - 2 colas para copiar datos: una en cada dirección (H2D, D2H)
  - Las 3 colas pueden correr tareas de forma concurrente
- Las operaciones CUDA se envían hacia el hardware en el orden en que se han lanzado
  - Se encolan
  - Las dependencias entre streams se mantienen entre colas.
- Una operación CUDA es enviada desde la cola si:
  - Las operaciones precedentes del mismo stream han terminado
  - Las operaciones precedentes de la cola se han enviado
  - Tiene los recursos necesarios
- Los kernels de CUDA pueden ejecutarse concurrentemente si son de streams diferentes.

iOBSOLETO!

# CUDA Stream Scheduling

## □ 2 streams (independientes):

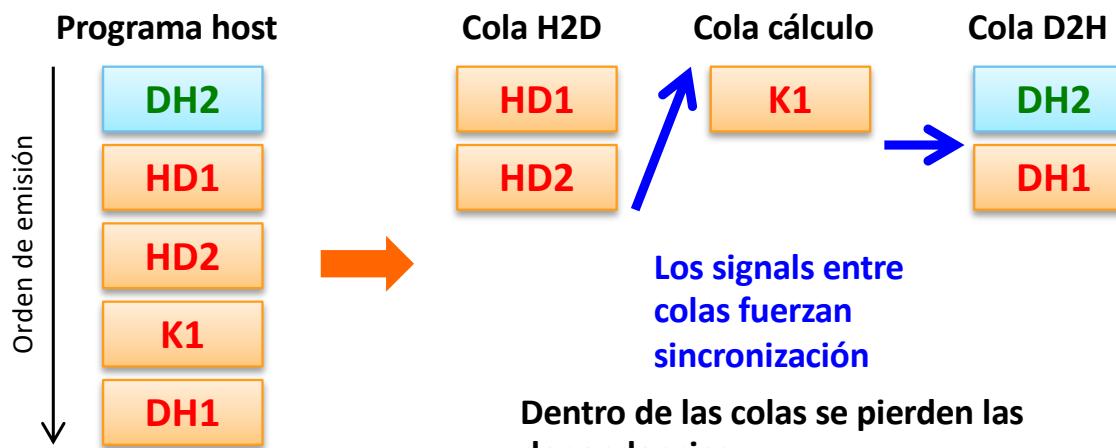
- Stream 1: HD1, HD2, K1, DH1 (emitido 1º)
- Stream2: DH2



# CUDA Stream Scheduling

## □ 2 streams (independientes):

- Stream 1: HD1, HD2, K1, DH1
- Stream2: DH2 (emitido 1º)

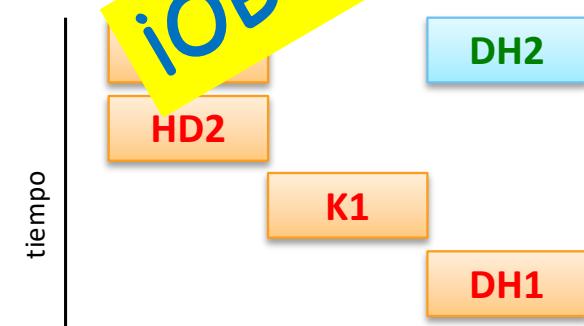


Las operaciones CUDA se encolan en el orden de emisión

EJECUCIÓN en ANCHURA

¡El orden de emisión es importante!

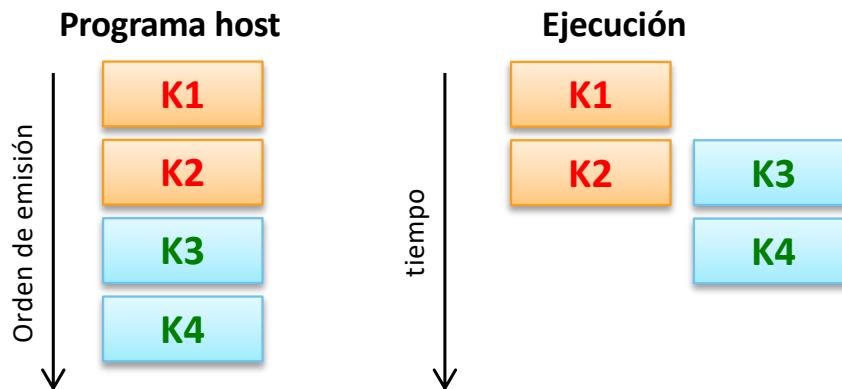
¡OBsoleto!



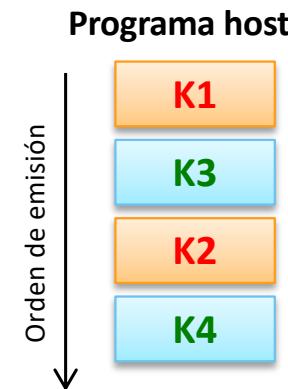
# CUDA Stream Scheduling

- 2 streams (cada kernel sólo utiliza la mitad de los SM):

- Stream 1: K1, K2
- Stream2: K3, K4



EJECUCIÓN en PROFUNDIDAD



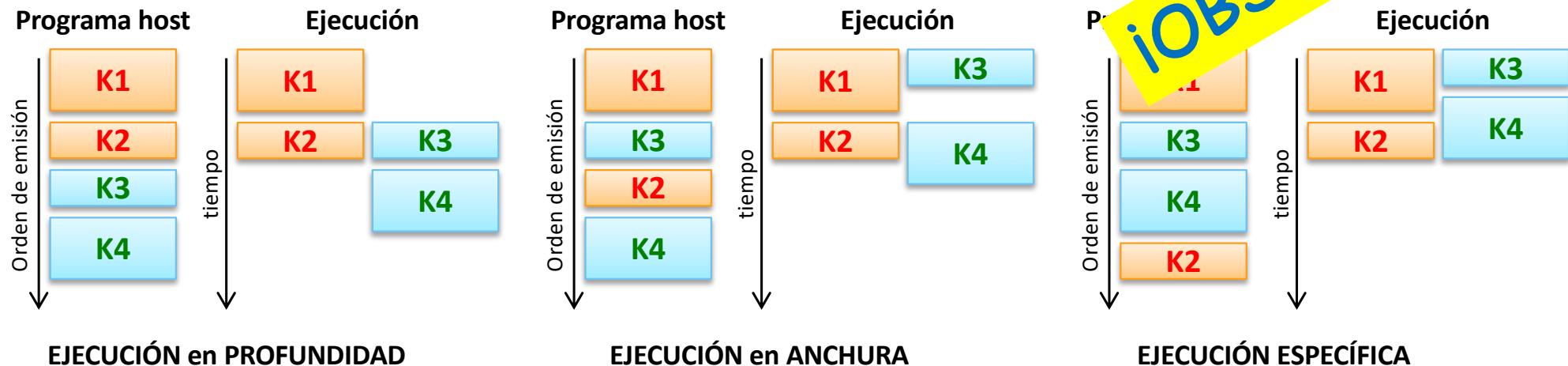
EJECUCIÓN en ANCHURA



# CUDA Stream Scheduling

□ 2 streams (cada kernel sólo utiliza la mitad de los SM, duración diferente):

- Stream 1: K1 (2), K2 (1)
- Stream2: K3 (1), K4 (2)



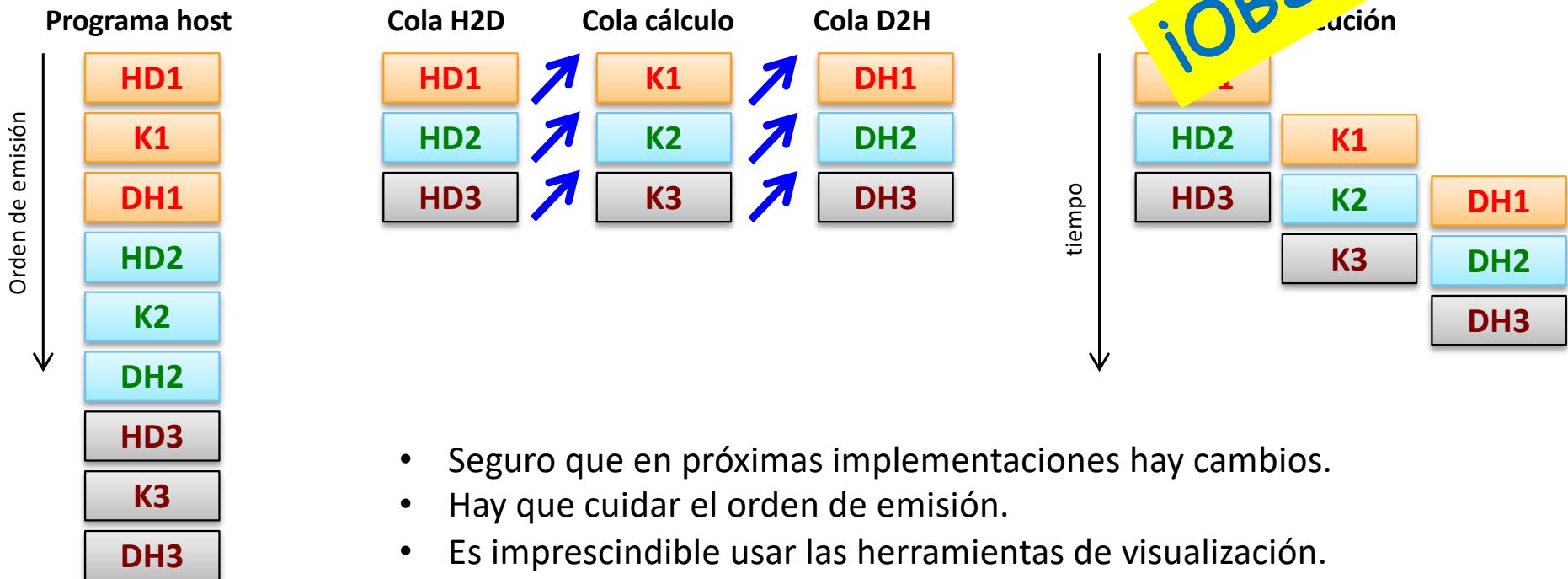
¡El orden de emisión es IMPORTANTE!

¡El tiempo de ejecución es IMPORTANTE!

# CUDA Stream Scheduling

## 3 streams (independientes):

- Stream 1: HD1, K1, DH1
- Stream2: HD2, K2, DH2
- Stream3: HD3, K3, DH3



# CUDA Streams

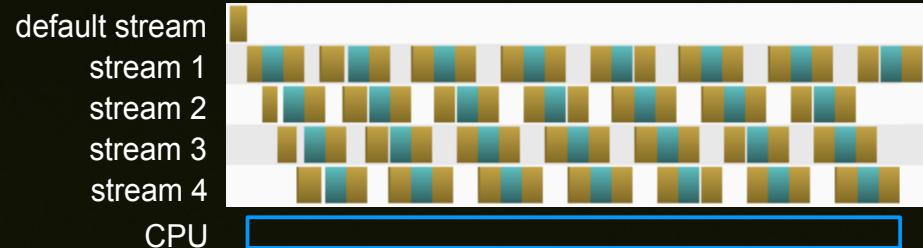
- CPU (4core Westmere x5670 @2.93 GHz, MKL)
  - **43 Gflops**

DGEMM: m=n=8192, k=288

- GPU (C2070)
  - Serial : 125 Gflops (2.9x)
  - 2-way : 177 Gflops (4.1x)
  - 3-way : 262 Gflops (6.1x)

- GPU + CPU
  - 4-way con.: **282 Gflops** (6.6x)
  - Up to **330 Gflops** for larger rank

- Obtain maximum performance by leveraging concurrency
- All communication hidden – effectively removes device memory size limitation



# Grid Management Unit

```
...
kernel_A <<< dimgridA, dimblockA, 0, stream_1 >>> (pars) ;
kernel_B <<< dimgridB, dimblockB, 0, stream_1 >>> (pars) ;
kernel_C <<< dimgridC, dimblockC, 0, stream_1 >>> (pars) ;
...
kernel_P <<< dimgridP, dimblockP, 0, stream_2 >>> (pars) ;
kernel_Q <<< dimgridQ, dimblockQ, 0, stream_2 >>> (pars) ;
kernel_R <<< dimgridR, dimblockR, 0, stream_2 >>> (pars) ;
...
kernel_X <<< dimgridX, dimblockX, 0, stream_3 >>> (pars) ;
kernel_Y <<< dimgridY, dimblockY, 0, stream_3 >>> (pars) ;
kernel_Z <<< dimgridZ, dimblockZ, 0, stream_3 >>> (pars) ;
...
```

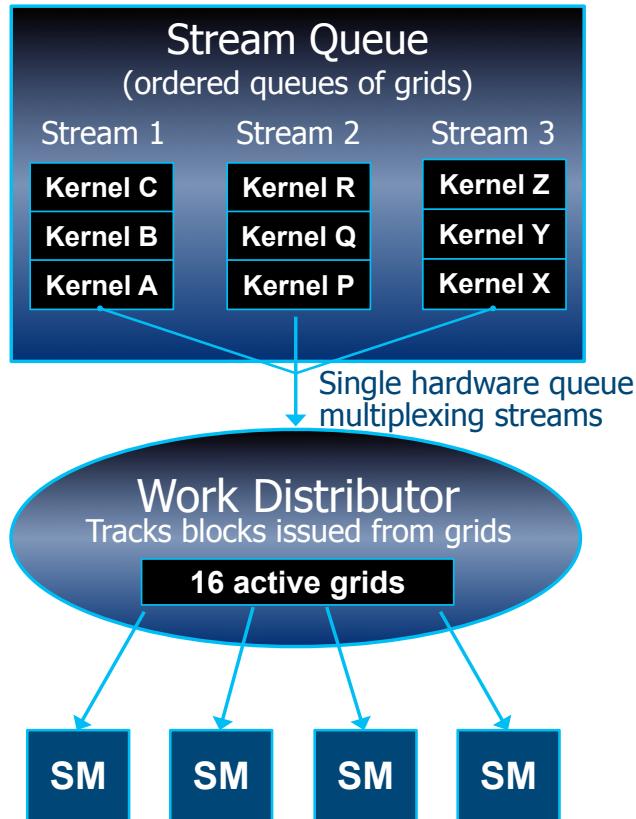
stream 1

stream 2

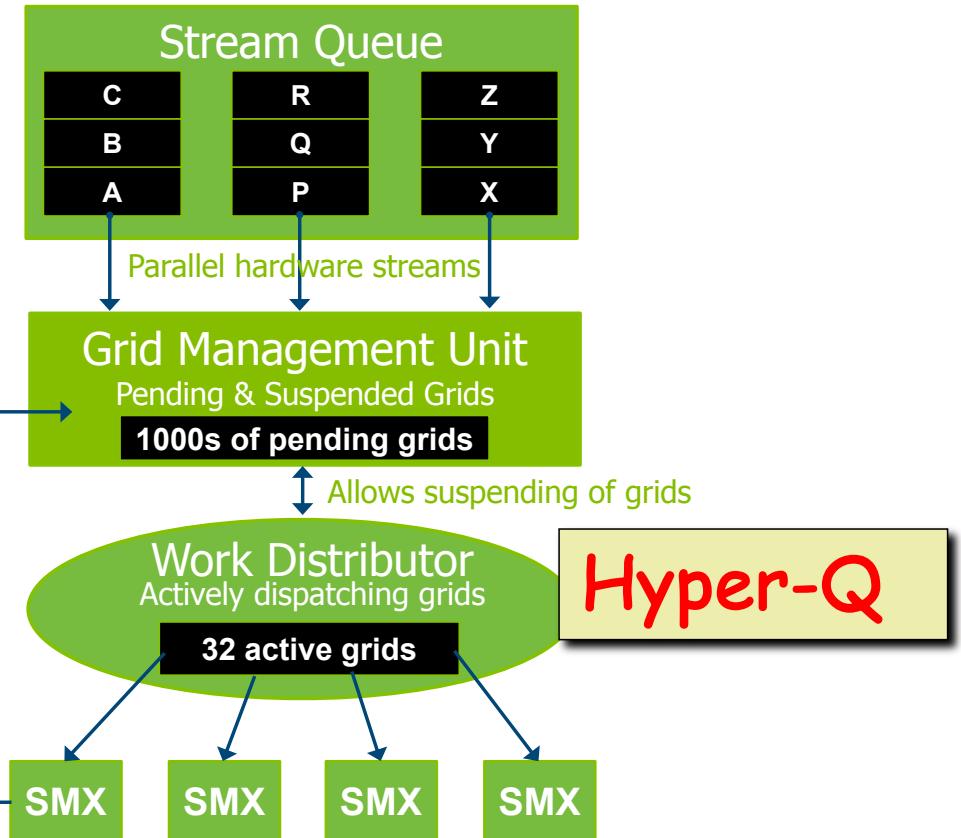
stream 3

# Grid Management Unit

Fermi



Kepler GK110



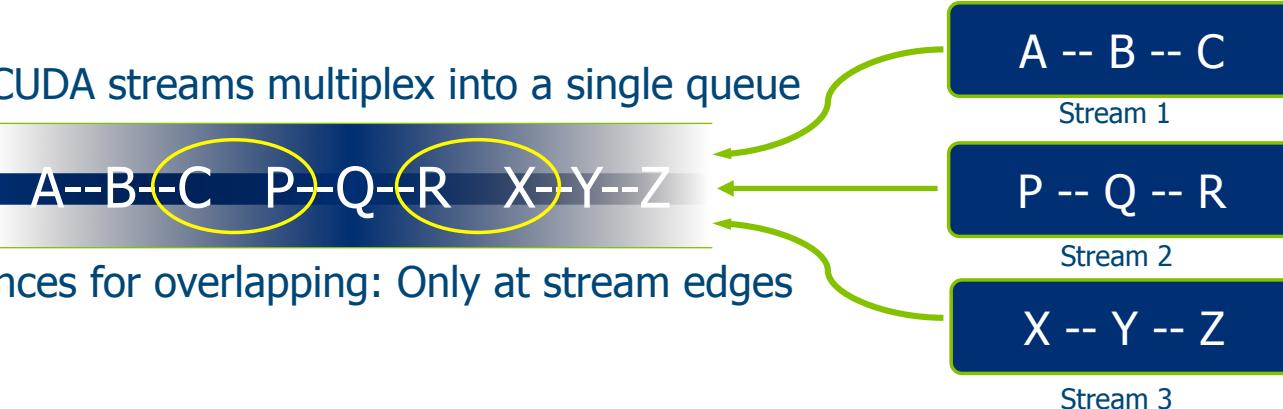
# Relación entre el software y las colas hardware

Fermi:

Up to 16 grids  
can run at once  
on GPU hardware

But CUDA streams multiplex into a single queue

Chances for overlapping: Only at stream edges



Kepler:

Up to 32 grids  
can run at once  
on GPU hardware

No inter-stream dependencies



# Resumiendo: Estructura típica programa CUDA

## □ Declaración de variables globales

- `__host__`, `__device__`
- `__global__`, `__constant__`

## □ Prototipos funciones

- `__global__ void kernel01(...)`

## □ `main ()`

- Asignar memoria en el device: `cudaMalloc (&d_GlblVarPtr, bytes)`
- Transferir datos del host al device: `cudaMemcpy (d_GlblVarPtr, h_Gl...)`
- Ejecutar setup configuración
- `kernel01<<<configuración>>>( args... ) ;`
- Transferir resultados del device al host: `cudaMemcpy (h_GlblVarPtr,...)`

## □ `void kernelOne(type args,...)`

- Declaración variables: `auto`, `__shared__`
  - ✓ Variables automáticas se asignan a registros
- `syncthreads ()`

Se repite las veces necesarias

# Optimización

## Exponer tanto paralelismo como sea posible

- Estructurad el algoritmo para maximizar el paralelismo.
- Si los threads de un block necesitan comunicarse, utilizad la memoria compartida y **`__syncthreads()`**
- Si threads de diferentes blocks necesitan comunicarse utilizad la memoria global y partid el cálculo en múltiples kernels
  - No existe mecanismo de sincronización entre los blocks
- Un paralelismo elevado es especialmente importante para ocultar la latencia de memoria.
- Aprovechad el uso de kernels asíncronos y la ejecución solapada con la CPU
- Aprovechad los streams y la posibilidad de solapar cálculos con transferencias (**`cudaMemcpyAsync()`**).

# Optimización

## Optimizar el uso de Memoria para maximizar el ancho de banda

- Procesar datos es más eficiente que moverlos, sobre todo entre GPU y CPU
- Y será peor en el futuro, cuanta menos limitado esté un kernel por el acceso a memoria, mejor funcionará en futuras GPUs.
- Hay que buscar:
  - Maximizar el uso de memoria de baja latencia y alto ancho de banda
  - Optimizar los patrones de acceso a memoria para maximizar el ancho de banda
  - Suficiente paralelismo como para ocultar la latencia de memoria
    - Ratio entre cálculo y memoria
  - A veces, es mejor recalcular los datos, que leerlos de memoria.
- Optimizad los patrones de acceso a memoria
  - El ancho de banda efectivo, puede variar un orden de magnitud dependiendo del patrón de acceso.

# Optimización

## Minimizar las transferencias de datos entre CPU y GPU

- El ancho de banda entre CPU y GPU es mucho menor que el ancho de banda de la GPU con la memoria global.
  - Usad memoria “pinned” (`cudaMallocHost()`) para maximizar el ancho de banda
- Minimizad las transferencias entre CPU y GPU, moviendo código de la CPU a la GPU
  - Incluso con poco paralelismo.
  - Estructuras de datos intermedias, se pueden crear, calcular y utilizar sin necesidad de pasar por la CPU
- Agrupad transferencias de datos
  - Una transferencia grande es mejor que varias pequeñas

# Heurísticas

- #threads por bloque: múltiplo del tamaño del warp
  - Evita el desperdicio de cálculo cuando hay warps incompletos.
- #blocks / #SM > 1
  - Así todos los SM tienen al menos un block para ejecutar.
- Mejor #blocks / #SM > 4
  - Para tener más de un block por SM.
  - Con múltiples blocks activos que no están esperando un `__syncthreads()`, el SM puede estar activo.
- Recursos utilizados por block (registros y memoria compartida): al menos la mitad del total disponible.
- #blocks > 1000, para que escale bien en futuros dispositivos
  - Uso de streams y ejecución segmentada
  - Tener 1000 blocks por grid asegura que la aplicación será escalable durante generaciones.

**En cualquier caso ¡EXPERIMENTAD!**

# Conclusiones finales

- NVIDIA está dedicando muchos esfuerzos a popularizar CUDA
- Programar en CUDA no es complejo
- Sin embargo, optimizar código CUDA es una tarea no trivial
  - Optimizar accesos a memoria global
  - Gestión explícita de la memoria compartida
  - Ajustar el tamaño de bloque
  - Gestionar correctamente los patrones de acceso a memoria
- Determinadas aplicaciones se resisten a este tipo de procesamiento. Por ejemplo, matrices dispersas:
  - Matrices con 100.000's de filas, pero sólo unos pocos elementos distintos de cero por fila
  - No se pueden almacenar completamente, se eliminan los ceros.

```
for (i=0; i<n; i++)
    for (j=IA[i]; j<IA[i+1]; j++)
        sum = sum + A[j] * v[JA[j]];
```

Matriz dispersa por vector. Operación básica en múltiples algoritmos para la resolución de sistemas de ecuaciones.

# Bibliografía & Documentación

- [www.nvidia.com](http://www.nvidia.com)
- NVIDIA.  
*CUDA C Programming Guide*,  
version 9, Nvidia 2018
- NVIDIA.  
*CUDA C Best Practices Guide*  
version 9, Nvidia 2018
- D. Kirk and W. Hwu,  
“Programming Massively Parallel Processors – A Hands-on Approach,” Third Edition  
Morgan Kaufman Publisher, 2017
  
- Manuales de Nvidia están **ACCESIBLES** en <https://docs.nvidia.com/cuda/>



UNIVERSITAT POLITÈCNICA DE CATALUNYA  
BARCELONATECH

Departament d'Arquitectura de Computadors

# Tarjetas Gráficas y Aceleradores

## CUDA

Agustín Fernández

Departament d'Arquitectura de Computadors

Facultat d'Informàtica de Barcelona

Universitat Politècnica de Catalunya

