

# Programación cuda



nvidia TESLA  
448 cores GPU's

# Vistazo rápido al modelo CUDA (cont.)

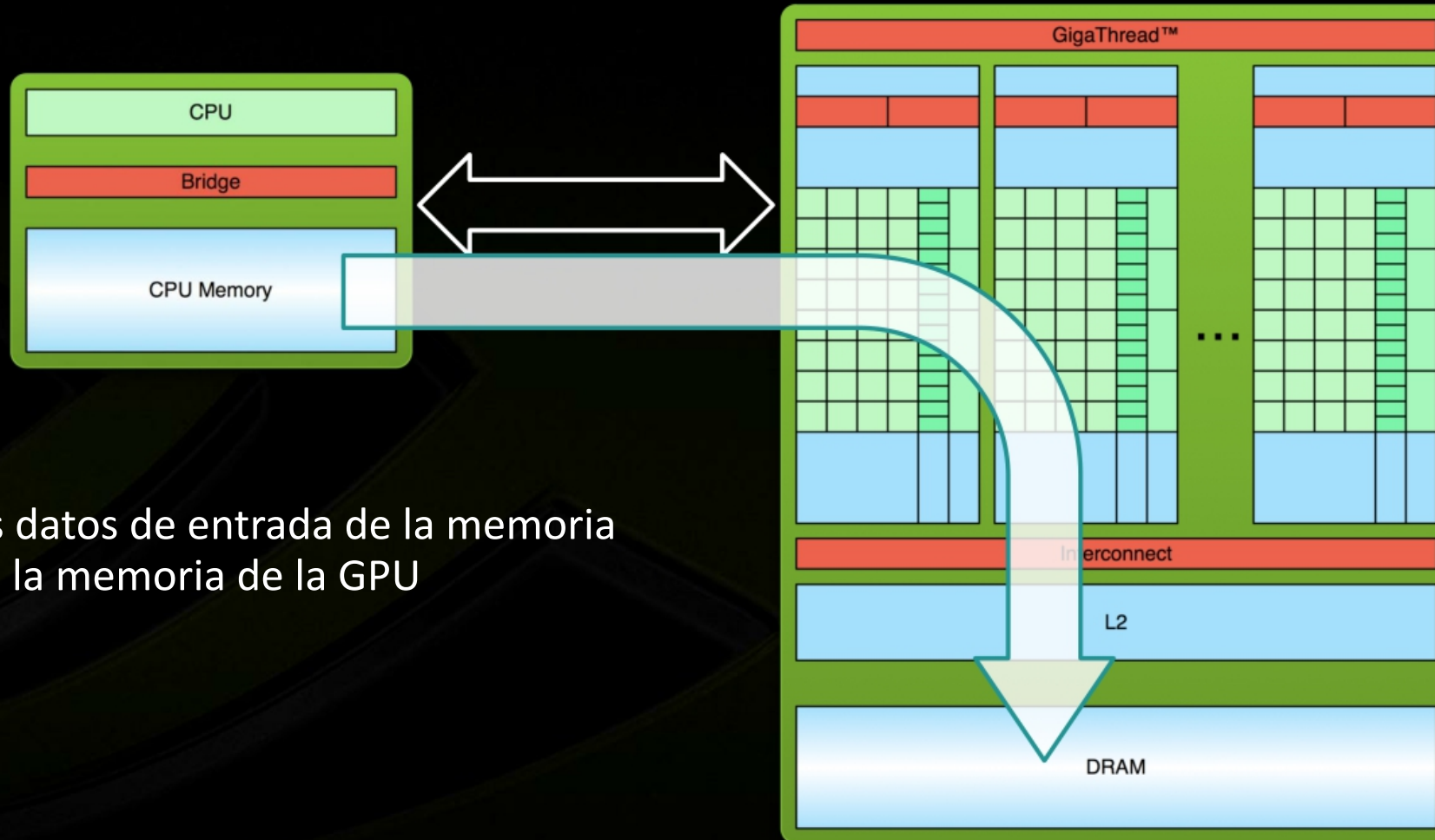
- La API es una extensión al ANSI C: **¡Fácil de aprender!**
- El hardware está diseñado para ejecución y control ligero: **¡Altas prestaciones!**

Introduce un nuevo  
modo operativo - interfaz de *hardware*  
para los cálculos

# C para CUDA

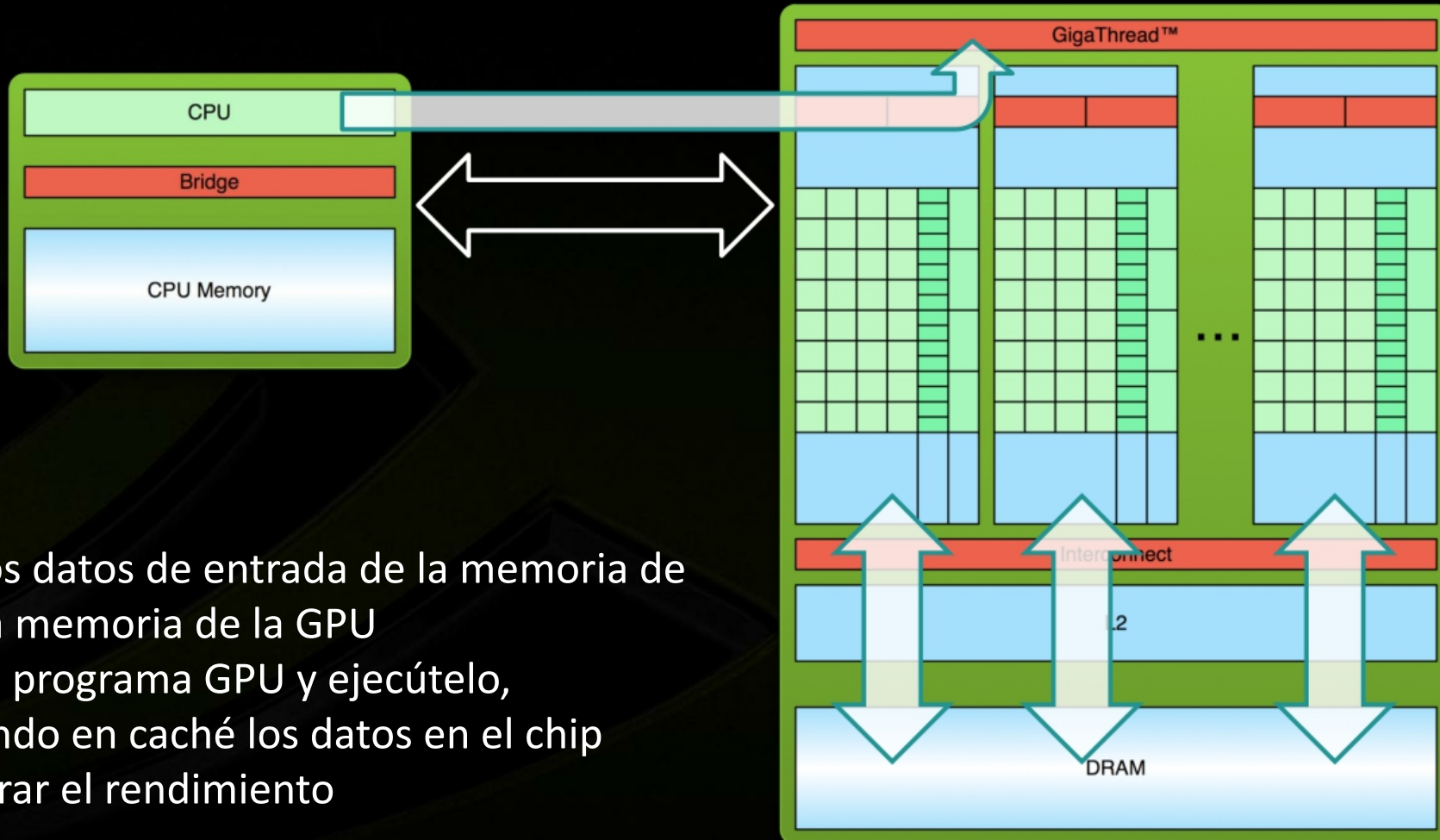
<b>Declspecs</b> global, device, shared, local, constant	<code>__device__ float filter[N];</code> <code>__global__ void convolve (float *image) {</code> <code>__shared__ float region[M];</code>
Keywords threadIdx, blockIdx	<code>region[threadIdx] = image[i];</code> ...
Intrinsics __syncthreads	<code>__syncthreads()</code> ... <code>image[j] = result; }</code>
Runtime API Memory, symbol, execution management	<code>// Allocate GPU memory</code> <code>void *myimage = cudaMalloc(bytes)</code>
Function launch	<code>// 100 blocks, 10 threads per block convolve&lt;&lt;&lt;100, 10&gt;&gt;&gt;</code> <code>(myimage);</code>

# Proceso de ejecución del código CUDA



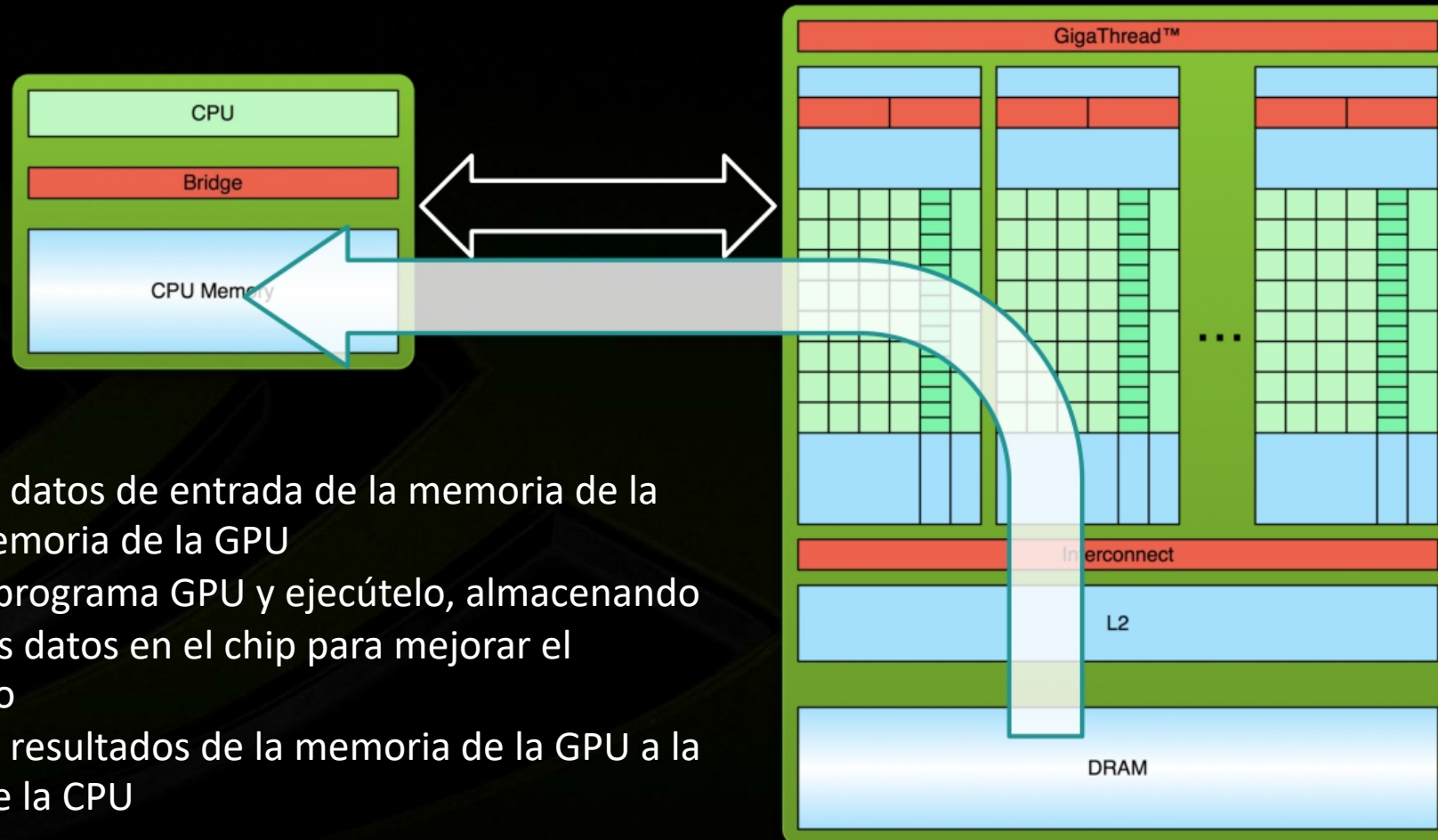
1. Copia los datos de entrada de la memoria de la CPU a la memoria de la GPU

# Proceso de ejecución del código CUDA



1. Copia los datos de entrada de la memoria de la CPU a la memoria de la GPU
2. Carga el programa GPU y ejecútelo, almacenando en caché los datos en el chip para mejorar el rendimiento

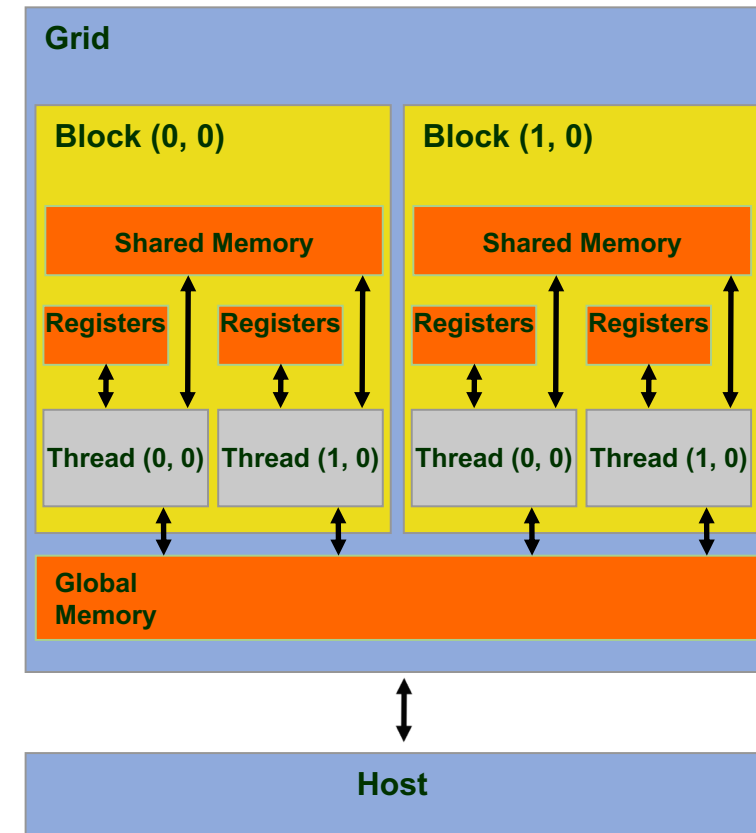
# Proceso de ejecución del código CUDA



1. Copia los datos de entrada de la memoria de la CPU a la memoria de la GPU
2. Carga el programa GPU y ejecútelo, almacenando en caché los datos en el chip para mejorar el rendimiento
3. Copia los resultados de la memoria de la GPU a la memoria de la CPU

# Gestión de la memoria

- `cudaMalloc()`
  - Asigna objetos en la memoria global
  - Requiere dos parámetros
    - Puntero del objeto a asignar
    - Tamaño del objeto a asignar
- `cudaFree()`
  - Libera un objeto de la memoria global
    - Puntero al objeto a liberar





# Gestión de la memoria (cont.)

Código de ejemplo:

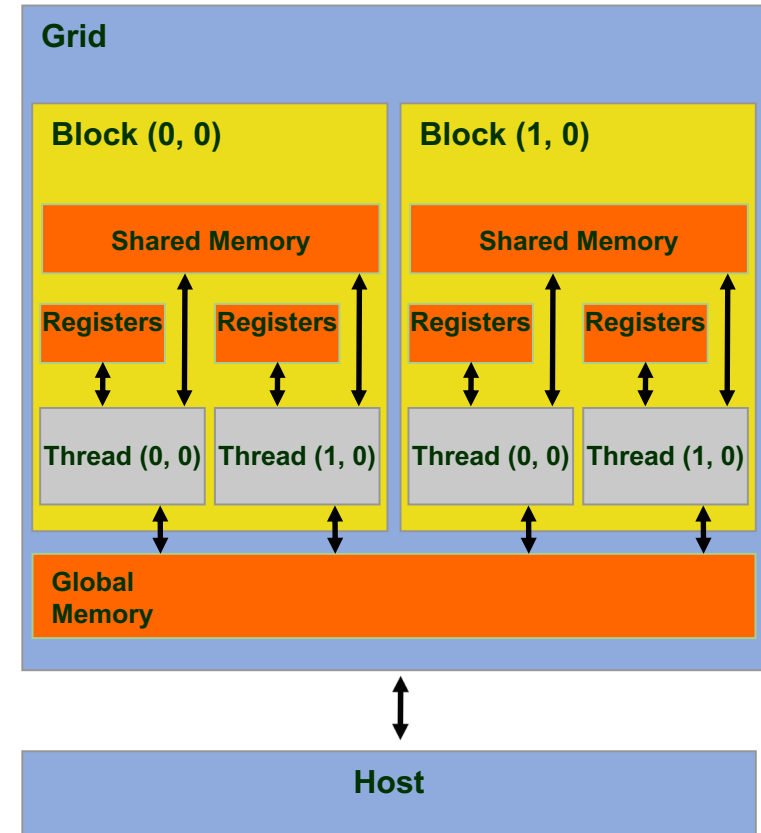
- Asigna 64 \* 64 floats
- Adscribir al espacio asignado a Md
- “d” se emplea normalmente para indicar la estructura de datos en el dispositivo

```
T_WIDTH = 64;
Float* Md;
int size = T_WIDTH * T_WIDTH * sizeof(float);

cudaMalloc((void**)&Md, size);
cudaFree(Md);
```

# Transferencia de datos

- `cudaMemcpy()`
  - Transfiere datos
  - Require 4 parámetros
    - Puntero al destino
    - Puntero a la origen
    - Nº bytes a copiar
    - Tipo de la transferencia
      - Host a Host
      - Host a Device
      - Device a Host
      - Device a Device
- Transferencia asíncrona



# Transferencia de datos (cont.)

- Código ejemplo:
  - Transfiere  $64 * 64$  floats
  - M está en el *host* y Md en el *device*
  - Constantes simbólicas
    - cudaMemcpyHostToDevice
    - cudaMemcpyDeviceToHost

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

# Declaración de funciones

	Se ejecuta en	Ejecutable desde
<code>__device__ float DeviceFunc()</code>	device	device
<code>__global__ void KernelFunc()</code>	device	host
<code>__host__ float HostFunc()</code>	host	host

- La directiva `__global__` define un kernel y debe devolver `void`
- Las directivas `__device__` y `__host__` pueden usarse simultáneamente

# Declaración de Funciones (cont.)

- No se puede obtener la dirección de las funciones  
\_\_device\_\_
- Las funciones que se ejecutan en el device:
  - No pueden ser recursivas
  - No pueden declarar variables estáticas
  - No pueden tener un número variable de argumentos

# Estructura típica de código CUDA

```
#include <stdlib .h>  #include <cuda.h>  #include <cuda runtime.h>
```

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

```
int main (void){  
    // Declare all variables .  
    ...  
    // Allocate host memory.  
    ...  
    // Dynamically allocate device memory for GPU results.  
    ...  
    // Write to host memory .  
    ...  
    // Copy host memory to device memory.  
    ...  
    // Execute kernel on the device .  
    some_kernel<<< num blocks , num theads per block >>>(...) ;  
    // Write GPU results in device memory back to host memory.  
    ...  
    // Free dynamically-allocated host memory ...  
    // Free dynamically-allocated device memory ...  
}
```

# Estructura típica de código CUDA

```
// Declaración variables globales
__host__, __device__, __global__, __constant__, __texture__

// Prototipos de funciones
__global__ void kernelOne(...)
float handy Function(...)

Main ()
    // Asignar memoria en el dispositivo
    cudaMalloc(&d_GlblVarPtr, bytes )
    // Transferir datos al dispositivo
    cudaMemcpy(d_GlblVarPtr, h_Gl...)
    // Configurar la ejecución y llamar al kernel
    kernelOne<<<execution configuration>>>( args... );
    //Transferir resultados al anfitrión
    cudaMemcpy(h_GlblVarPtr,...)
    // Opcional: comparar contra una solución
    // calculada en el anfitrión

void kernelOne(type args,...)

// Declaración de variables
    __local__, __shared__
    // Variables automáticas asignadas a registros
    // o memoria local
    syncthreads()...

// Otras funciones
float handyFunction(int inVar);
```

# Ejecución de kernels/Creación de Hilos

- Las llamadas a kernels deben tener una configuración de ejecución.
- Las llamadas a los kernels son asíncronas, se necesita sincronización explícita para bloquear.

```
__global__ void KernelFunc(...);  
dim3    DimGrid(100, 50); // 5000 bloques de hilos  
dim3    DimBlock(4, 8, 8); // 256 hilos por bloque  
size_t  SharedMemBytes = 64; // 64 bytes de memoria comp.  
KernelFunc<<< DimGrid, DimBlock, SharedMemBytes >>>(...);
```



# Ejemplo kernel

## Indicación kernel

```
__global__ void sharedABMultiply(float *a, float* b, float *c, int N) {  
    __shared__ float aTile[TILE_DIM][TILE_DIM], Mem compartida  
    bTile[TILE_DIM][TILE_DIM];  
registros int row = blockIdx.y * blockDim.y + threadIdx.y;  
    int col = blockIdx.x * blockDim.x + threadIdx.x; Identificadores hilo  
    float sum = 0.0f;  
    aTile[threadIdx.y][threadIdx.x] = a[row*TILE_DIM+threadIdx.x];  
    bTile[threadIdx.y][threadIdx.x] = b[threadIdx.y*N+col];  
    __syncthreads(); Barrera para todos los hilos del mismo bloque  
    for (int i = 0; i < TILE_DIM; i++) {  
        sum += aTile[threadIdx.y][i] * bTile[i][threadIdx.x];  
    }  
    c[row*N+col] = sum;  
}
```

# Ejemplos de codigo Cuda

## Standard C Code

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

// Perform SAXPY on 1M elements
saxpy_serial(4096*256, 2.0, x, y);
```

## Parallel C Code

```
__global__
void saxpy_parallel(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];
}

// Perform SAXPY on 1M elements
saxpy_parallel<<<4096,256>>>(n,2.0,x,y);
```

# Hello world!

```
#include <stdio .h>

int main(){
printf ("Hello , World!\n") ;
return 0;
}
```

```
#include<stdio.h>
int main (int argc, char**argv){
printf("Hola mundo\n");
return0;
}
```

```
#include<stdio.h>
#include<stdio.h>
int main (void){
    printf("Hola mundo\n");
    System("pause");
    Return EXIT_SUCCESS;
}
```

```
#include <stdio .h>

__global__ void myKernel(){
}

int main(){
myKernel<<<1, 1>>>();
printf ("Hello , World!\n") ;
return 0;
}
```

```
#include <iostream>
#include "common/book.h"

__global__ void kernel(void){
}

int main( void ) {
    kernel<<<1,1>>>();
    printf("Hello, world!\n");
    return 0;
}
```

# Incrementar un valor “b” a los N elementos de un vector

Programa C en CPU (compilado con gcc)

```
void incremento_en_cpu(float *a, float b, int N)
{
    for (int idx = 0; idx < N; idx++) a[idx] = a[idx] + b;
}

void main() {
    .....
    incremento_en_cpu(a, b, N);
}
```

El kernel CUDA que se ejecuta en la GPU (parte superior), seguido del código host en CPU.

Este archivo se compila con nvcc (ver más adelante).

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

void main() {
    .....
    dim3 dimBlock (blocksize);
    dim3 dimGrid( ceil( N / (float)blocksize ) );
    incremento_en_gpu<<<dimGrid, dimBlock>>>(a, b, N);
}
```

# Múltiple Kernel

```
#include<stdio.h>

__global__ void kernel(void){
}

__global__ void foo(void){
}

int main (int argc, char** argv){
    kernel<<<1,1>>>();
    foo<<<1,1>>>();
    printf("Hola mundo\n");
    return 0;
}
```

# Suma de dos números

```
#include<stdio.h>

__device__ floatx(floatx,floaty){
    returnx+y;
}

__global__ voidkernel(void){
    funcion(1.0,2.0);
}

Int main(int argc, char**argv){
    kernel<<<1,1>>>();
    printf("Llamada a función desde kernel\n");
    return0;
}
```

# Suma cambiar el valor en el dispositivo

```
#include <stdio.h> #include <stdlib .h>
```

```
#include <cuda.h> #include <cuda runtime.h>
```

```
__global__ void colonel(int *a_d){  
    *a_d = 2;}
```

```
int main(){  
    int a = 0, *a_d;
```

```
    cudaMalloc((void**) &a_d, sizeof(int));  
    cudaMemcpy(a_d, &a, sizeof(int), cudaMemcpyHostToDevice);
```

```
    colonel<<<1,1>>>(a_d);
```

```
    cudaMemcpy(&a , a_d , sizeof (int) , cudaMemcpyDeviceToHost) ;
```

```
    printf("a=%d\n", a);
```

```
    cudaFree(a_d);
```

```
}
```

```
> nvcc simple.cu -o simple  
> ./ simple  
a=2
```



# Búsqueda de un carácter en una cadena

```
#include<stdio.h>
```

```
#include<iostream>
```

```
// Incluye utilidades de CUDA
```

```
#include<cutil.h>
```

```
__global__ void buscaCadena (char *cadena, int longitud, char caracter, bool*encontrado){
```

```
    int i=0;
```

```
    while((cadena[i]!=caracter)&& i<longitud)
```

```
        i++;
```

```
    *encontrado=(i!=longitud)? True : false;
```

```
}
```

```
int main(int argc, char**argv){
```

```
}
```

```
    char HOSTcad[34]={"Volverán las oscuras golondrinas"};
```

```
    bool HOSTencontrado;
```

```
    char *GPUcad;
```

```
    bool *GPUencontrado;
```

# Búsqueda de un carácter en una cadena

```
CUDA_SAFE_CALL(cudaMalloc((void**)&GPUencontrado,sizeof(bool)));
CUDA_SAFE_CALL(cudaMalloc((void**)&GPUcad,sizeof(char)*32));
CUDA_SAFE_CALL(cudaMemcpy(GPUcad,HOSTcad,sizeof(char)*32,cudaMemcpyHostToDevice));

buscaCadena<<<1,1>>>(GPUcad,32,'i',GPUencontrado);

CUDA_SAFE_CALL(cudaMemcpy(&HOSTencontrado,GPUencontrado,sizeof(bool),cudaMemcpyDeviceToHost));

std::cout<<"Carácter encontrado: "<<((HOSTencontrado)?"si": "no")<<std::endl;

cudaFree(GPUencontrado);
cudaFree(GPUcad);
return 0;
}
```

Nota: CUDA\_SAFE\_CALL es una macro de atención al errores definidas para versiones anteriores al Cuda 4.

# Suma de dos enteros

```
#include <iostream>
#include "common/book.h"
__global__ void add( int a, int b, int *c ) {
    *c = a + b;
}

int main ( void ) {
    int c; int *dev_c;
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c, sizeof(int) ) );
    add<<<1,1>>>( 2, 7, dev_c );

    HANDLE_ERROR( cudaMemcpy( &c, dev_c, sizeof(int), cudaMemcpyDeviceToHost ) );

    printf( "2 + 7 = %d\n", c );
    cudaFree( dev_c );
    return 0;
}
```

Nota: HANDLE\_ERROR es una función para el tratamiento de errores definida en el libro de Cuda by example

# Características de la tarjeta

# Características de la tarjeta

```
#include "common/book.h"
```

```
int main( void ) {  
    cudaDeviceProp prop;  
    int count;  
    HANDLE_ERROR( cudaGetDeviceCount( &count ) );  
    for( int i = 0; i < count; i++ ) {  
        HANDLE_ERROR( cudaGetDeviceProperties( &prop, i ) );  
        printf( "Name: %s\n", prop.name);  
        printf("MAX Threads per block: %d\n",deviceProp.maxThreadsPerBlock); printf("MAX BLOCK SIZE\n");  
        printf(" [x -> %d]\n [y -> %d]\n [z -> %d]\n",deviceProp.maxThreadsDim[0],  
                deviceProp.maxThreadsDim[1], deviceProp.maxThreadsDim[2]);  
        printf("MAX GRID SIZE\n");  
        printf(" [x -> %d]\n [y -> %d]\n [z -> %d]\n",deviceProp.maxGridSize[0],  
                deviceProp.maxGridSize[1], deviceProp.maxGridSize[2]);  
    }  
}
```

# Características de la tarjeta

- Querying Devices, pag 27 de “Cuda by Example”

[http://www.mat.unimi.it/users/sansotte/cuda/CUDA\\_by\\_Example.pdf](http://www.mat.unimi.it/users/sansotte/cuda/CUDA_by_Example.pdf)

- `cudaError_t cudaGetDeviceProperties` (struct `cudaDeviceProp` \* prop, int device)

[https://www.cs.cmu.edu/afs/cs/academic/class/15668-s11/www/cuda-doc/html/group\\_\\_CUDART\\_\\_DEVICE\\_g5aa4f47938af8276f08074d09b7d520c.html](https://www.cs.cmu.edu/afs/cs/academic/class/15668-s11/www/cuda-doc/html/group__CUDART__DEVICE_g5aa4f47938af8276f08074d09b7d520c.html)

- How to Query Device Properties and Handle Errors in CUDA C/C++

<https://developer.nvidia.com/blog/how-query-device-properties-and-handle-errors-cuda-cc/>

# Suma de vectores

# Suma de vectores

```
#include "cuda_runtime.h" #include "device_launch_parameters.h" #include <stdio.h>
cudaError_t addWithCuda(int *c, const int *a, const int *b, unsigned int size);
__global__ void addKernel(int *c, const int *a, const int *b)
{ int i = threadIdx.x;    c[i] = a[i] + b[i]; }
int main()
{
    const int arraySize = 5;    const int a[arraySize] = { 1, 2, 3, 4, 5 }; const int b[arraySize] = { 10, 20, 30, 40, 50 };
    int c[arraySize] = { 0 };
    // Add vectors in parallel.

    cudaError_t cudaStatus = addWithCuda(c, a, b, arraySize);
    if (cudaStatus != cudaSuccess) { fprintf(stderr, "addWithCuda failed!"); return 1; }

    printf("{1,2,3,4,5} + {10,20,30,40,50} = {%d,%d,%d,%d,%d}\n", c[0], c[1], c[2], c[3], c[4]);

    // cudaDeviceReset must be called before exiting in order for profiling and
    // tracing tools such as Nsight and Visual Profiler to show complete traces.
    cudaStatus = cudaDeviceReset();
    if (cudaStatus != cudaSuccess) { fprintf(stderr, "cudaDeviceReset failed!"); return 1; }
    return 0;
}
```



# Suma de vectores

```
// Helper function for using CUDA to add vectors in parallel.
cudaError_t addWithCuda(int *c, const int *a, const int *b, unsigned int size)
{
    int *dev_a = 0; int *dev_b = 0; int *dev_c = 0;
    cudaStatus = cudaSetDevice(0);
    cudaMalloc((void**)&dev_c, size * sizeof(int));
    cudaMalloc((void**)&dev_a, size * sizeof(int));
    cudaMalloc((void**)&dev_b, size * sizeof(int));
    // Copy input vectors from host memory to GPU buffers.
    cudaMemcpy(dev_a, a, size * sizeof(int), cudaMemcpyHostToDevice);
    cudaStatus = cudaMemcpy(dev_b, b, size * sizeof(int), cudaMemcpyHostToDevice);
    // Launch a kernel on the GPU with one thread for each element.
    addKernel<<<1, size>>>>(dev_c, dev_a, dev_b);
    // Copy output vector from GPU buffer to host memory.
    cudaMemcpy(c, dev_c, size * sizeof(int), cudaMemcpyDeviceToHost);
    cudaFree(dev_c); cudaFree(dev_a); cudaFree(dev_b);
    return cudaStatus;
}
```

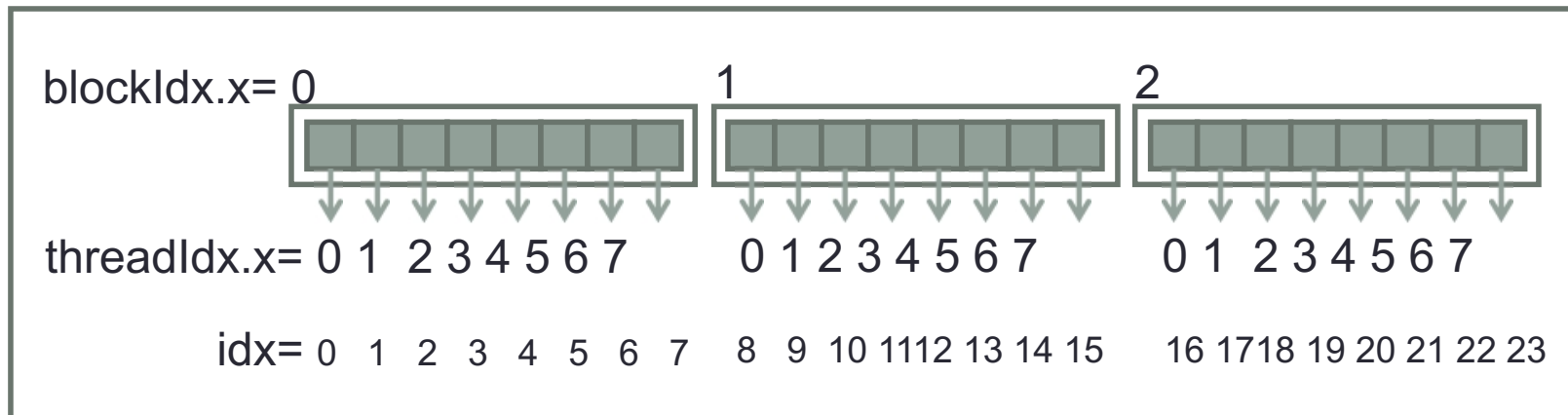
# Suma de vectores (varios bloques)

# Suma de Vectores por bloques

$\text{idx} = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}$

$N=24$  y  $\text{blockDim.x}=8$

Grid



# Suma de vectores (varios bloques)

```
// Device code
__global__ void VecAdd(float* A, float* B, float* C, int N)
{
    Int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}

// Host code
int main()
{
    int N = ...;
    size_t size = N * sizeof(float);
    // Allocate input vectors h_A and h_B in host memory
    float* h_A = (float*) malloc(size);
    float* h_B = (float*) malloc(size);
    // Initialize input vectors
```

# Suma de vectores (varios bloques)

```
// Allocate vectors in device memory
float* d_A; cudaMalloc(&d_A, size);
float* d_B; cudaMalloc(&d_B, size);
float* d_C; cudaMalloc(&d_C, size);

// Copy vectors from host memory to device memory
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

// Invoke kernel
Int threadsPerBlock = 256;
Int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);

// Copy result from device memory to host memory
// h_C contains the result in host memory
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

// Free device memory
cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);

// Free host memory
```

# Multiplicación de matrices

# Ejemplo: Multiplicación de matrices

La multiplicación de matrices que muestra las características básicas de la memoria y el manejo de hilos en los programas de CUDA

- Dejar el uso de la memoria compartida hasta más tarde.
- Uso de registro, local.
- Uso del ID del hilo.
- API para la transferencia de datos en memoria entre host y device.
- Asumimos una matriz cuadrada para simplificar.

# Multiplicaciones de Matrices

$$\begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$



# Representación de Matrices en una dimensión

```
matriz = [15; 24; 57; 72; 74; 23; 20; 25;  
          46; 36; 98; 8; 42; 55; 10; 21;  
          44; 47; 98; 10; 82; 94; 89; 7;  
          92; 90; 52; 50; 77; 59; 44; 42;  
          94; 3; 25; 90; 98; 30; 43; 12;  
          50; 49; 77; 93; 97; 85; 80; 52;  
          60; 74; 47; 17; 58; 47; 82; 71;  
          29; 53; 46; 82; 23; 88; 80; 43];;
```

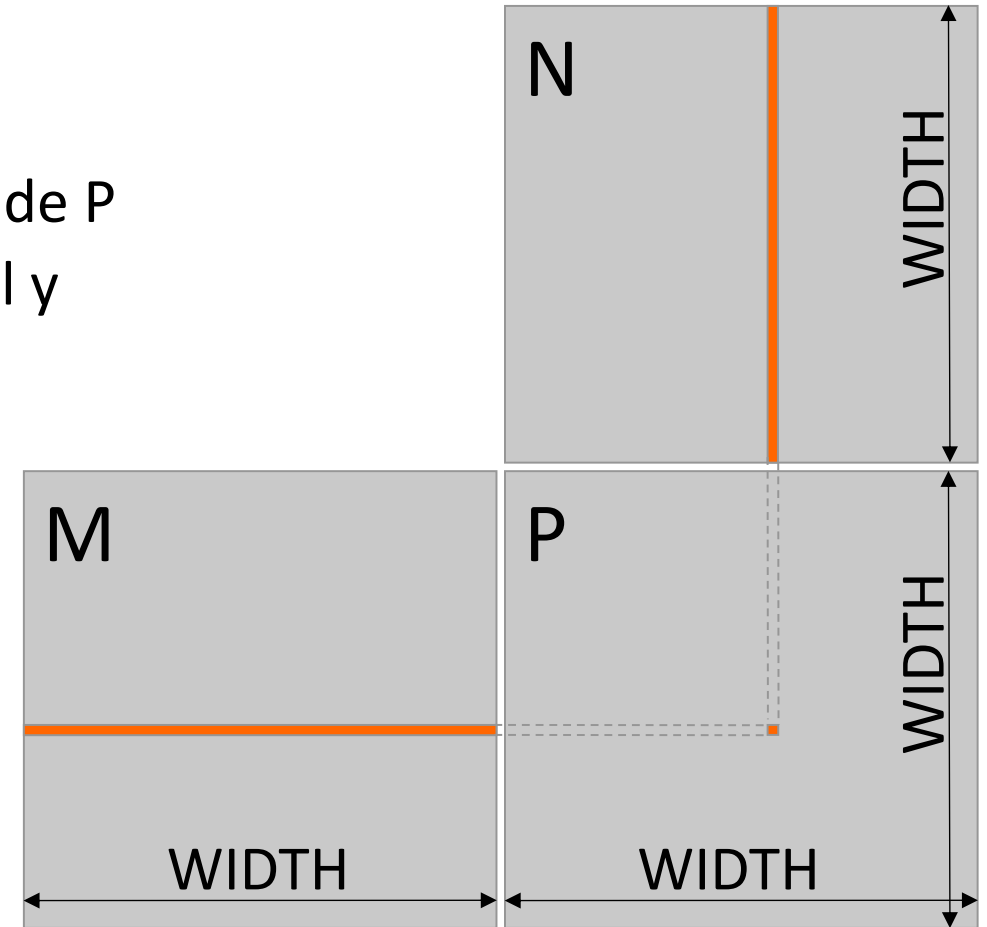
```
matriz = [1,1; 1,2; 1,3; 1,4; 1,5; 1,6; 1,7;1,8;  
          2,1; 2,2; 2,3; 2,4; 2,5; 2,6; 2,7;2,8;  
          3,1; 3,2; 3,3; 3,4; 3,5; 3,6; 3,7;3,8;  
          4,1; 4,2; 4,3; 4,4; 4,5; 4,6; 4,7;4,8;  
          5,1; 5,2; 5,3; 5,4; 5,5; 5,6; 5,7;5,8;  
          6,1; 6,2; 6,3; 6,4; 6,5; 6,6; 6,7;6,8;  
          7,1; 7,2; 7,3; 7,4; 7,5; 7,6; 7,7;7,8;  
          8,1; 8,2; 8,3; 8,4; 8,5; 8,6; 8,7;8,8];;
```

**elemento (i,j) = i\_esimo (j+(i-1)\*8)**

**elemento (1,3) matriz;;  
- : int = 57**

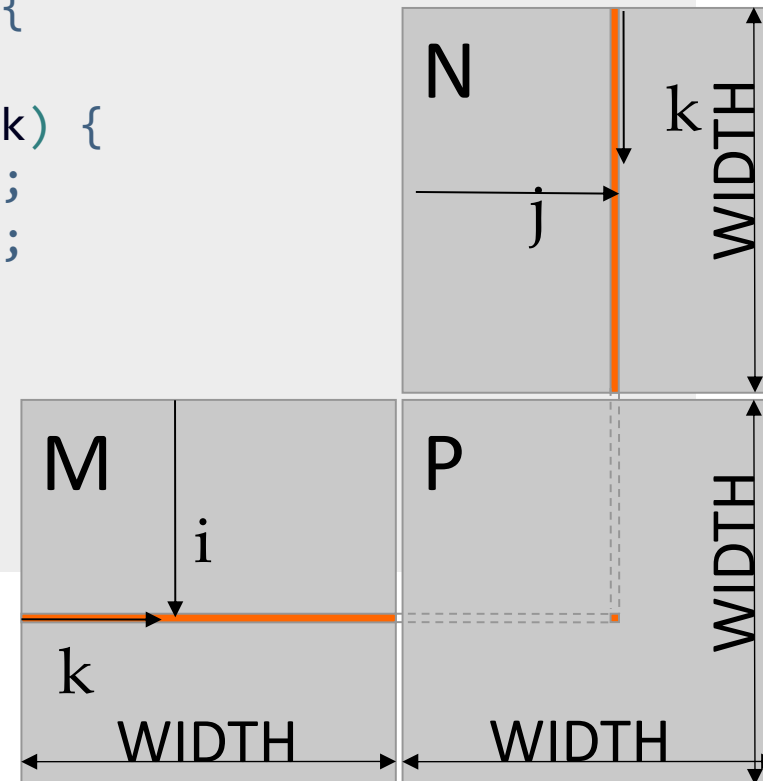
# Ejemplo: Multiplicación matrices $n \times n$

- $P = M \times N$
- Sin teselado
  - Un único hilo calcula un único elemento de  $P$
  - $M$  y  $N$  se cargan desde la memoria global y tienen tamaño  $WIDTH \times WIDTH$



# Ejemplo: Código (sólo host)

```
// Multiplicación de matrices en (CPU) host
// en doble precisión
void MatrixMulOnHost(float* M, float* N, float* P, int Width) {
    for (int i = 0; i < Width; ++i)
        for (int j = 0; j < Width; ++j){
            double sum = 0;
            for (int k = 0; k < Width; ++k) {
                double a = M[i * width + k];
                double b = N[k * width + j];
                sum += a * b;
            }
            P[i * Width + j] = sum;
        }
}
```



# Esquema de uso de la memoria de una matriz en C

$M_{0,0}$	$M_{1,0}$	$M_{2,0}$	$M_{3,0}$
$M_{0,1}$	$M_{1,1}$	$M_{2,1}$	$M_{3,1}$
$M_{0,2}$	$M_{1,2}$	$M_{2,2}$	$M_{3,2}$
$M_{0,3}$	$M_{1,3}$	$M_{2,3}$	$M_{3,3}$

- En **C** se almacenan los elementos de una fila de manera contigua
- En **Fortran** se almacenan los de una columna
- En **Java** se almacena un array de referencias a los arrays de las columnas

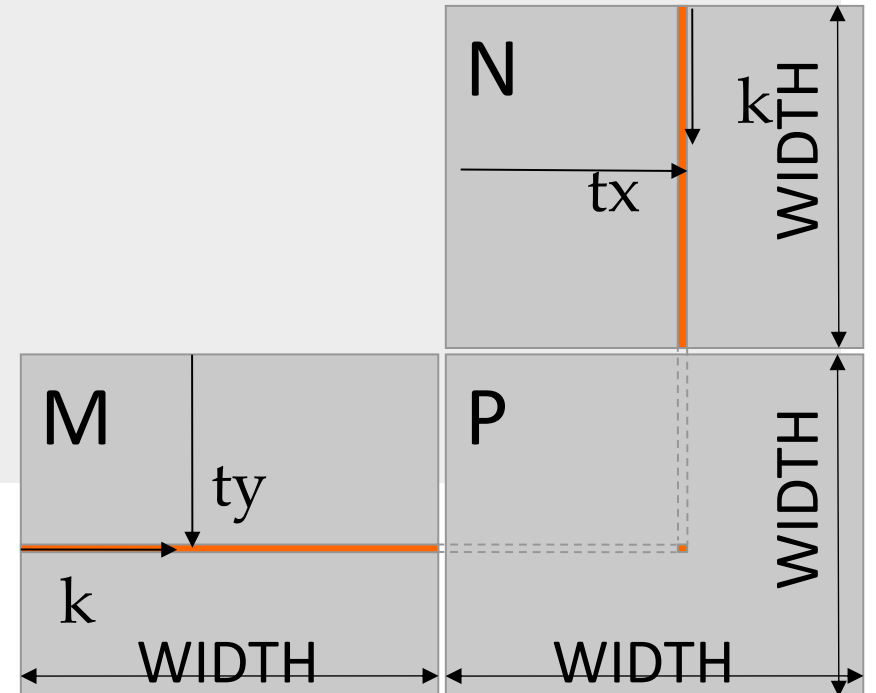


# Ejemplo: Código (parte host)

```
void MatMulOnDevice(float* M, float* N, float* P, int Width)
{
    int size = Width * Width * sizeof(float);
    float* Md, Nd, Pd;
    // 1.- Asignamos y cargamos M & N en la memoria del device
    cudaMalloc(&Md, size);
    cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
    cudaMalloc(&Nd, size);
    cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);
    // 2.- Asignamos P en el device
    cudaMalloc(&Pd, size);
    // 3.- Invocamos el kernel - detalles después -
    . . . . .
    // 4.- Leemos P del device
    cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
    // Liberamos la memoria del dispositivo
    cudaFree(Md); cudaFree(Nd); cudaFree(Pd);
}
```

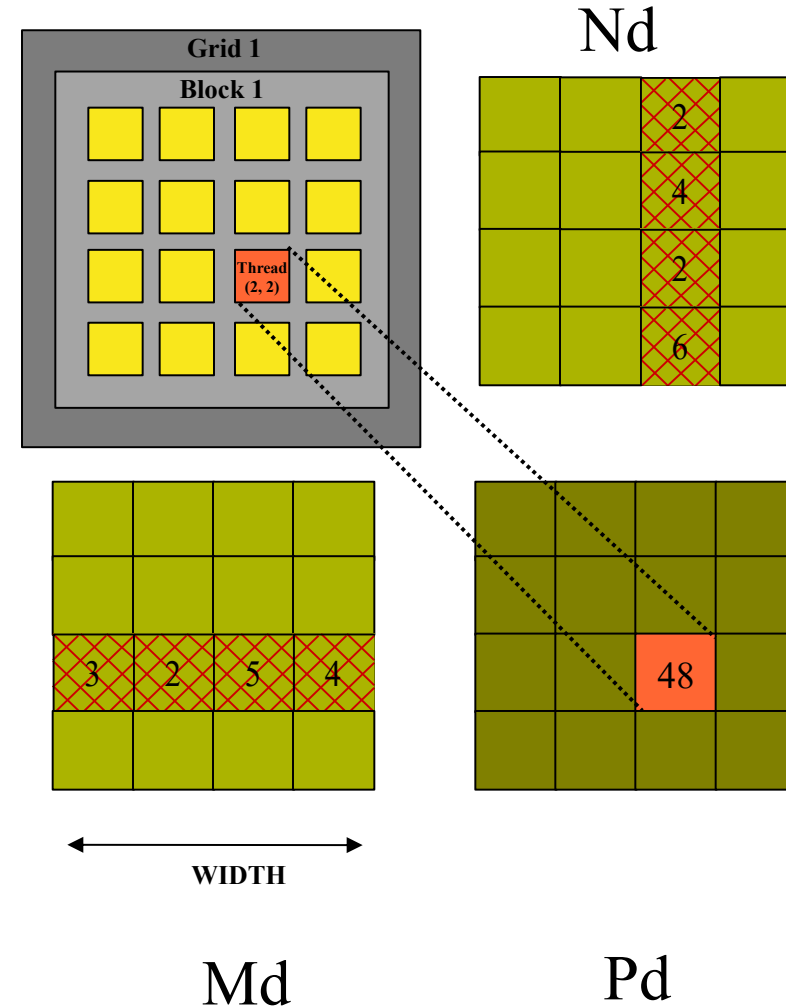
# Ejemplo: Paso 4 (código del kernel)

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
    // Usamos Pvalue para almacenar el valor de la
    // matriz calculado por el thread
    float Pvalue = 0;
    for (int k = 0; k < Width; ++k){
        float Melement = Md[threadIdx.y*Width+k];
        float Nelement = Nd[k*Width+threadIdx.x];
        Pvalue += Melement * Nelement;
    }
    Pd[threadIdx.y*Width+threadIdx.x] = Pvalue;
}
```



# Características e inconvenientes

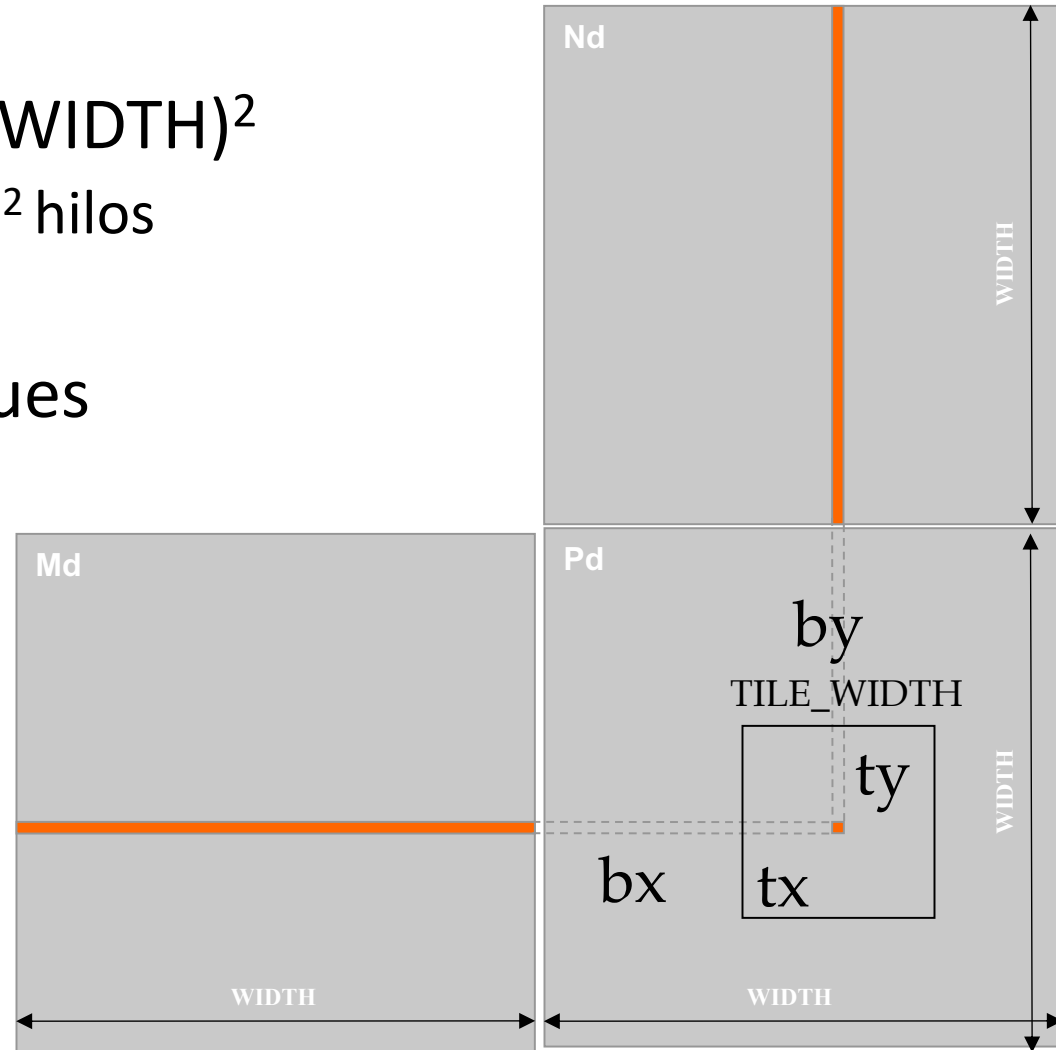
- Un bloque de hilos calcula la matriz Pd.
  - Cada hilo calcula un elemento de Pd
- Cada hilo
  - Carga una fila de la matriz Md
  - Carga una columna de la matriz Nd
  - Realiza una multiplicación y una suma para cada par de elementos de Md y Nd
  - Cálculo de la tasa de acceso a la memoria off-chip cercana a 1:1 (no muy alta)
- **El tamaño de la matriz está limitado por el número de hilos permitidos en un bloque de hilos**



# Manejo de matrices arbitrarias

- Cada bloque 2d calcula una submatriz de tamaño  $(\text{TILE\_WIDTH})^2$ 
  - Cada uno tiene  $(\text{TILE\_WIDTH})^2$  hilos
- Genera una cuadrícula de  $(\text{WIDTH}/\text{TILE\_WIDTH})^2$  bloques

Todavía se necesita un bucle si el ratio  $\text{WIDTH}/\text{TILE\_WIDTH}$  es mayor que 64Kb!

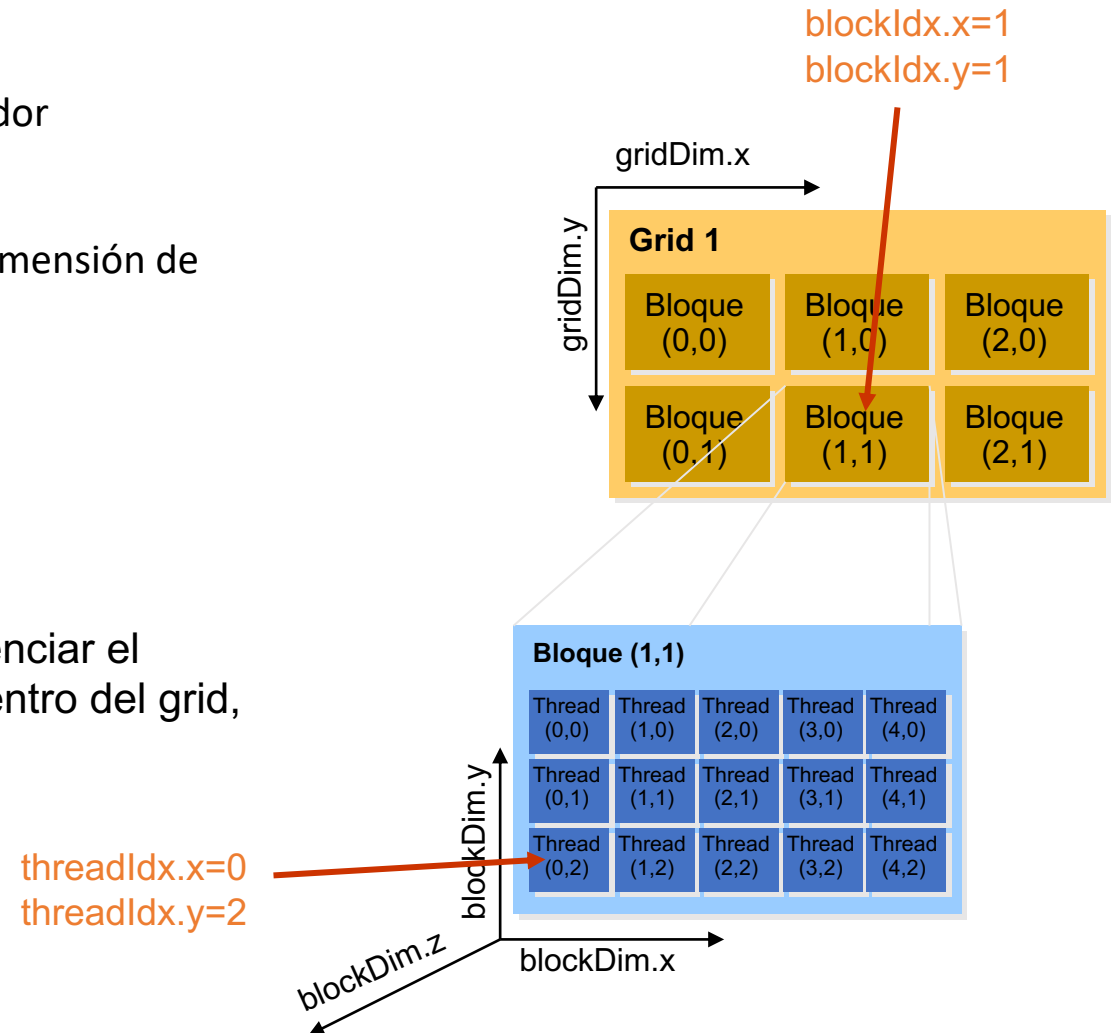




# Manejo de matrices arbitrarias

- Identificadores y dimensiones

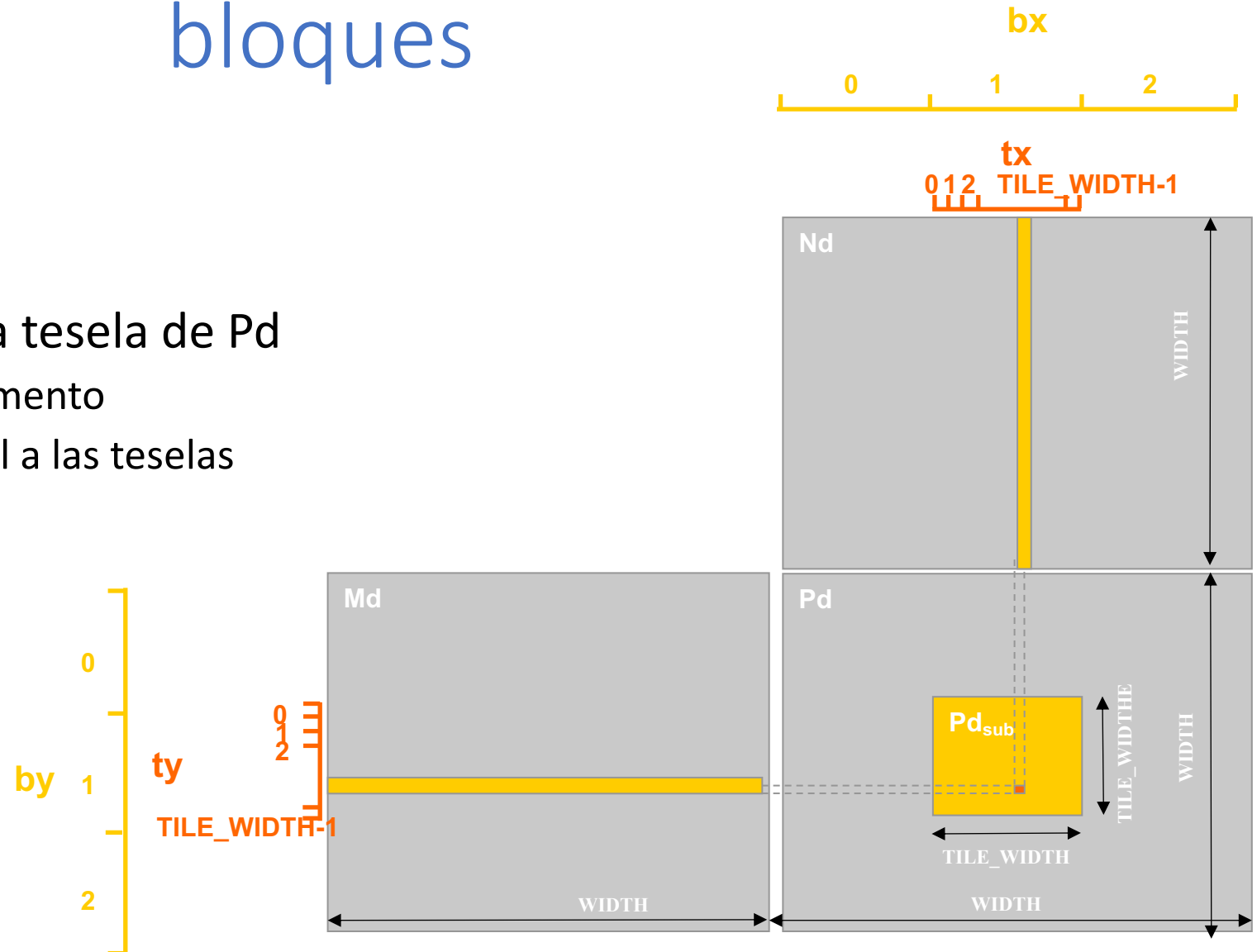
- El tamaño del grid y de los bloques los determina el programador
- Se usan las variables **gridDim** y **blockDim** para referenciar la dimensión de grid y bloque, respectivamente
- Un thread queda indentificado por:
  - Un identificador propio dentro del bloque al que pertenece
  - El identificador del bloque al que pertenece
- Se usan las variables **threadIdx** y **blockIdx** para referenciar el identificador del thread dentro del bloque y al bloque dentro del grid, respectivamente



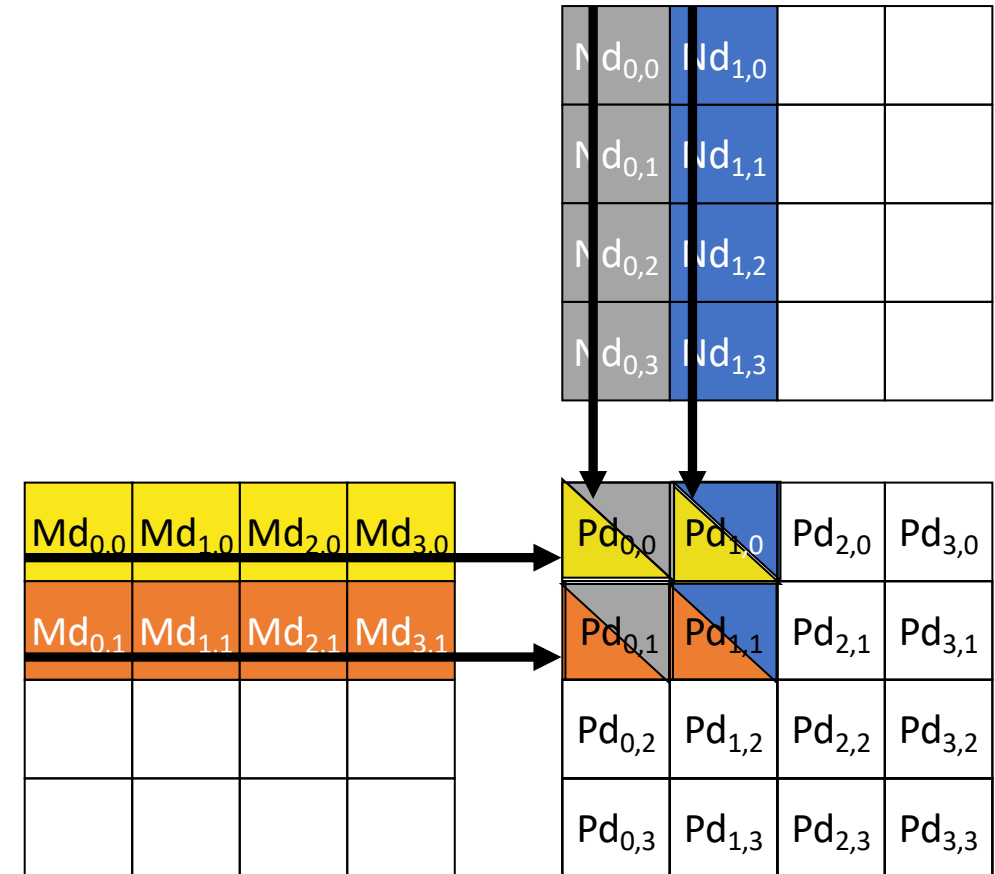
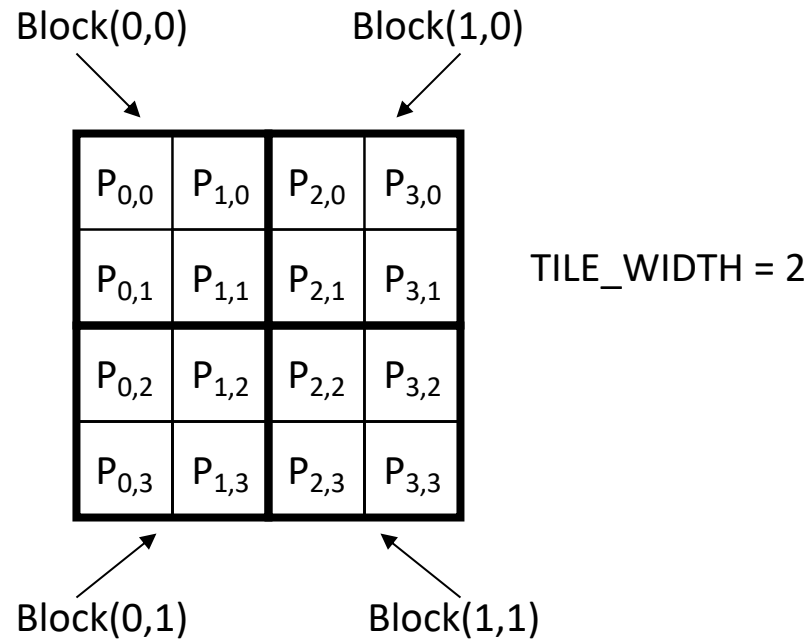
# Multiplicación de matrices por bloques

# Multiplicación de matrices usando múltiples bloques

- Partimos  $P_d$  en teselas
- Cada bloque calcula una tesela de  $P_d$ 
  - Cada hilo calcula un elemento
  - Bloques de tamaño igual a las teselas



# Multiplicación de matrices usando múltiples bloques



# Multiplicación de matrices usando múltiples bloques

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
    // Calculate the row index of Pd element and M
    int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
    // Calculate the column idenx of Pd and N
    int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;
    float Pvalue = 0;
    // each thread computes one element of the block sub-matrix
    for (int k = 0; k < Width; ++k)
        Pvalue += Md[Row*Width+k] * Nd[k*Width+Col];

    Pd[Row*Width+Col] = Pvalue;
}
```

# Invocación del núcleo (Host-side Code)

```
// Setup the execution configuration

dim3 dimGrid(Width/TILE_WIDTH, Width/TILE_WIDTH);
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);

// Launch the device computation threads!

MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```

# Multiplicación de matrices vía memoria compartida

# Ejemplo: Multiplicación de matrices usando múltiples bloques

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
    // Calculate the row index of the Pd element and M
    int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
    // Calculate the column idenx of Pd and N
    int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;

    float Pvalue = 0;
    // each thread computes one element of the block sub-matrix
    for (int k = 0; k < Width; ++k)
        Pvalue += Md[Row*Width+k] * Nd[k*Width+Col];

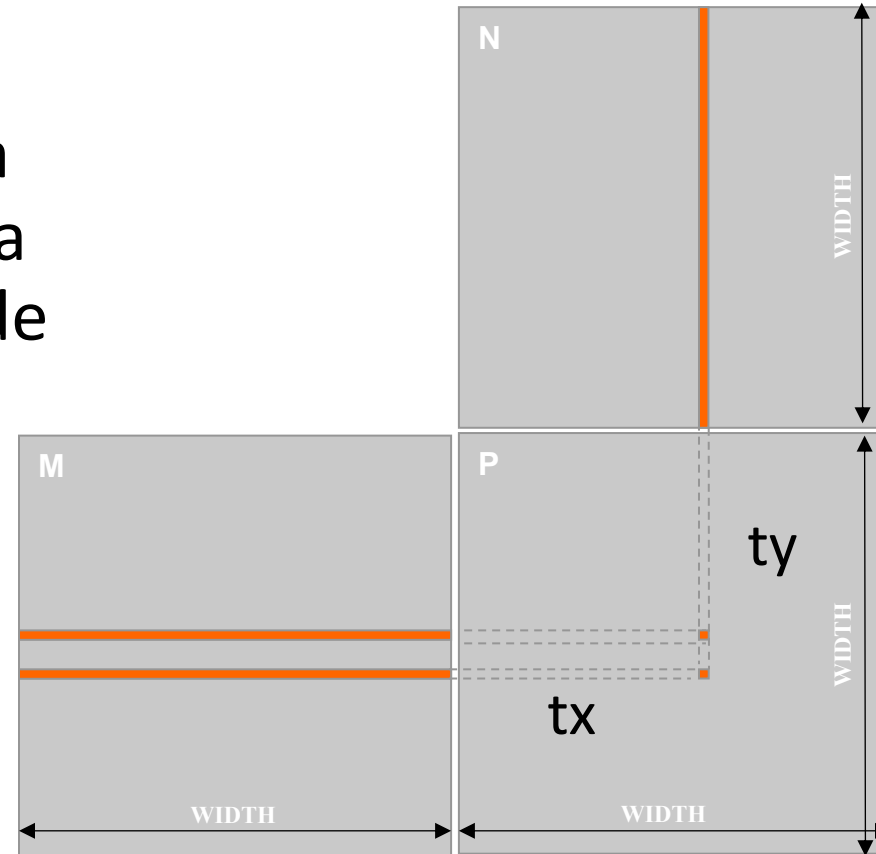
    Pd[Row*Width+Col] = Pvalue;
}
```



# Mejora: Empleo de memoria compartida para reutilizar datos de la memoria global

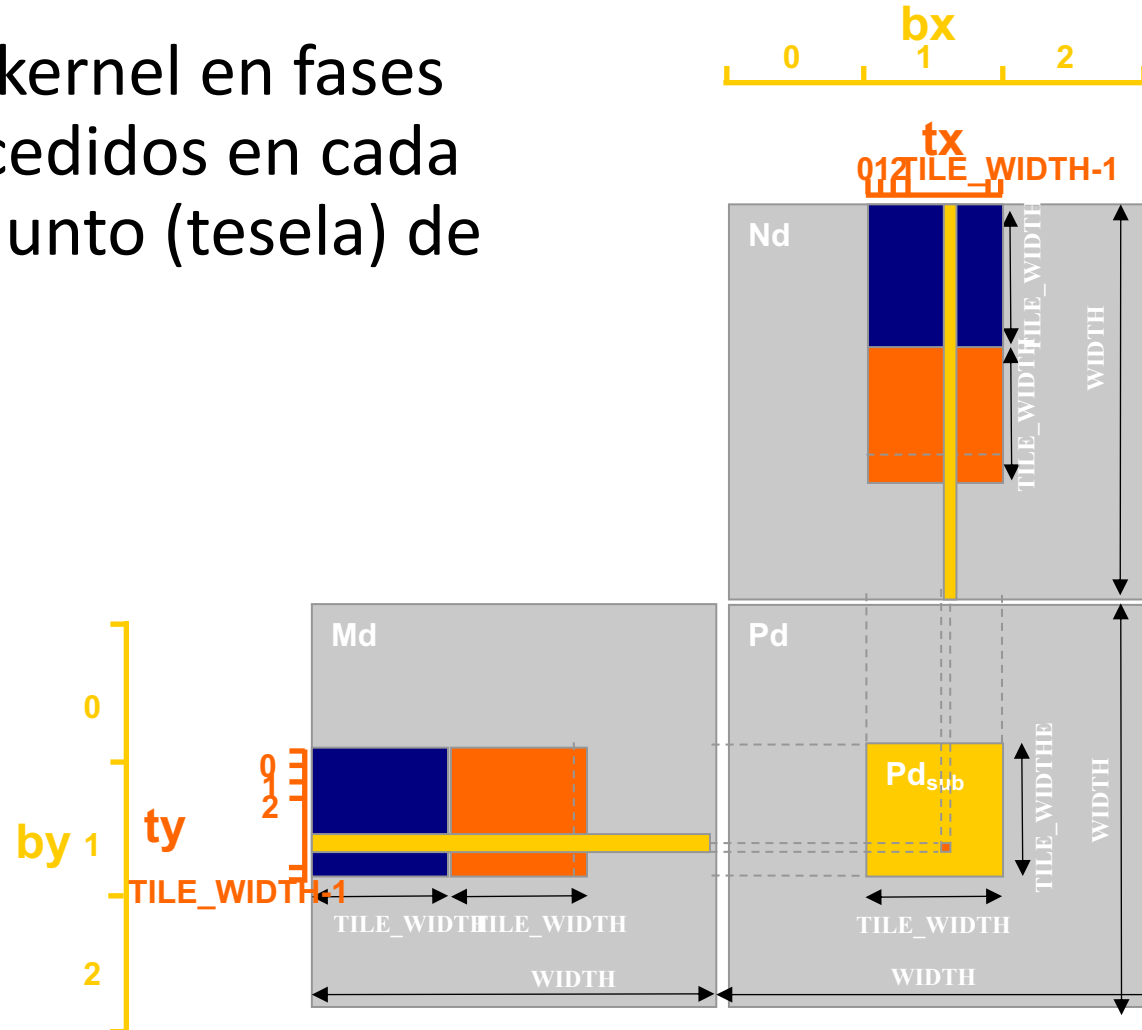
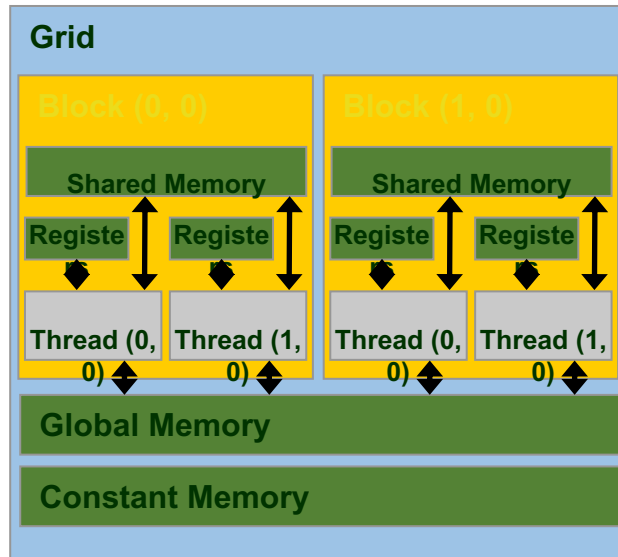
- Cada elemento es leído por WIDTH hilos
- Carga cada elemento en la memoria compartida y varios hilos emplean la versión local para reducir el ancho de banda

## Algoritmos Teselados



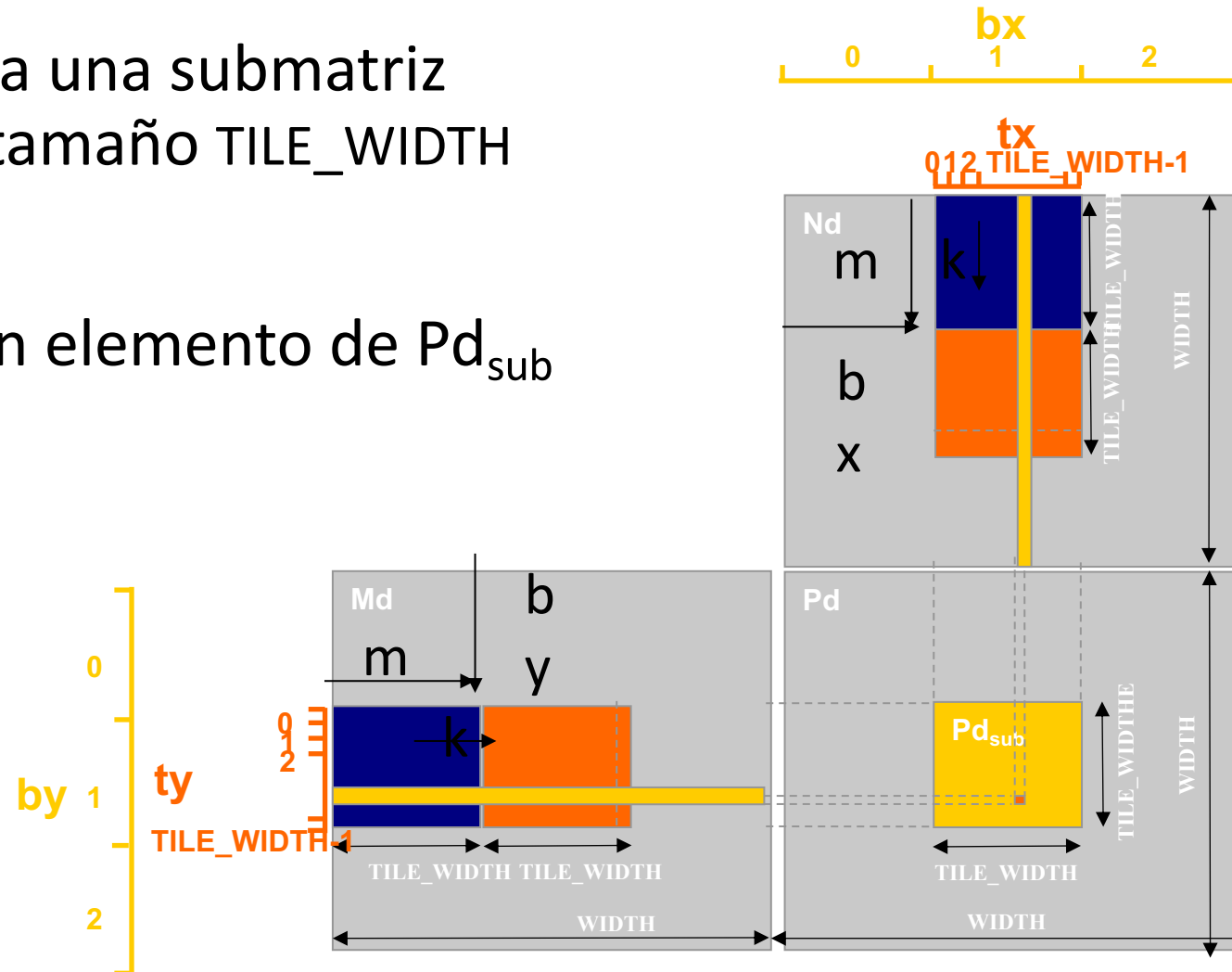
# Multiplicación teselada

Rompemos la ejecución del kernel en fases de manera que los datos accedidos en cada fase se centra en un subconjunto (tesela) de  $M_d$  y  $N_d$

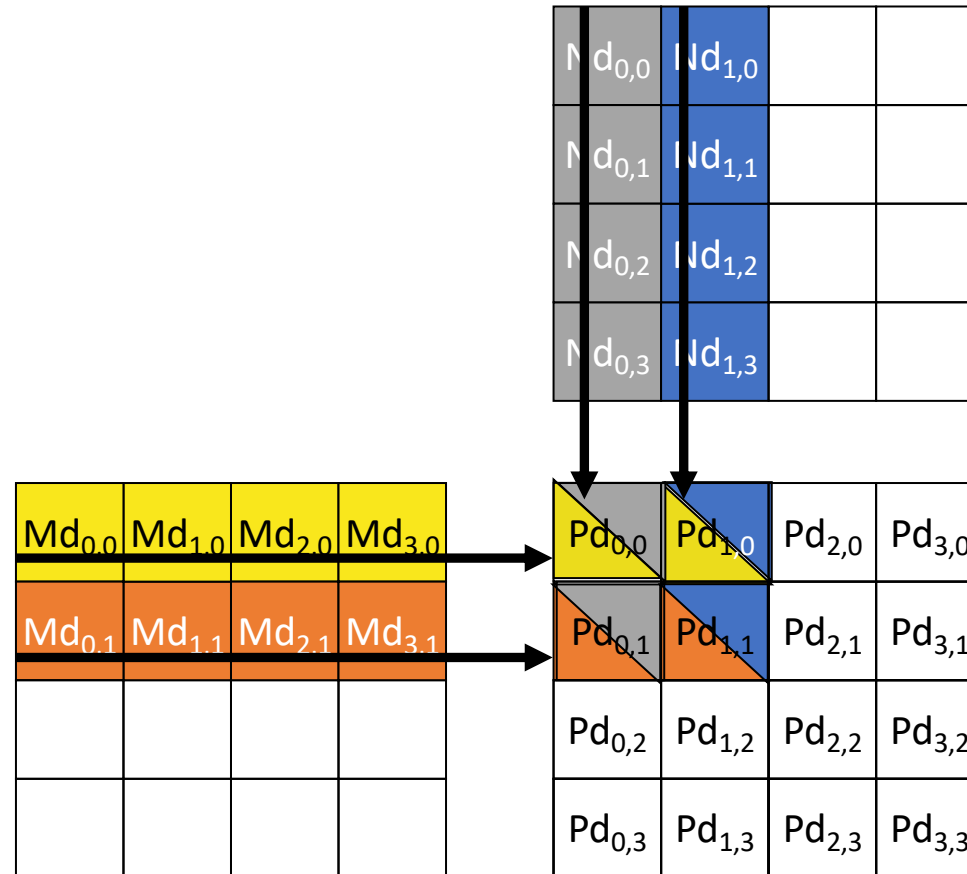


# Multiplicación teselada

- Cada bloque calcula una submatriz cuadrada  $Pd_{sub}$  de tamaño  $TILE\_WIDTH$
- Cada hilo calcula un elemento de  $Pd_{sub}$



# Ejemplo: Producto de Matrices 4X4



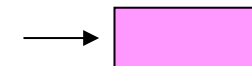
# Ejemplo: Producto de Matrices 4X4

Cada thread calcula un punto de la matriz resultado

orden accesos ↓

$Pd_{0,0}$ thread <sub>0,0</sub>	$Pd_{1,0}$ thread <sub>1,0</sub>	$Pd_{0,1}$ thread <sub>0,1</sub>	$Pd_{1,1}$ thread <sub>1,1</sub>
$Md_{0,0} \times Nd_{0,0}$	$Md_{0,0} \times Nd_{1,0}$	$Md_{0,1} \times Nd_{0,0}$	$Md_{0,1} \times Nd_{1,0}$
$Md_{1,0} \times Nd_{0,1}$	$Md_{1,0} \times Nd_{1,1}$	$Md_{1,1} \times Nd_{0,1}$	$Md_{1,1} \times Nd_{1,1}$
$Md_{2,0} \times Nd_{0,2}$	$Md_{2,0} \times Nd_{1,2}$	$Md_{2,1} \times Nd_{0,2}$	$Md_{2,1} \times Nd_{1,2}$
$Md_{3,0} \times Nd_{0,3}$	$Md_{3,0} \times Nd_{1,3}$	$Md_{3,1} \times Nd_{0,3}$	$Md_{3,1} \times Nd_{1,3}$

En una primera fase se llevan a memoria compartida

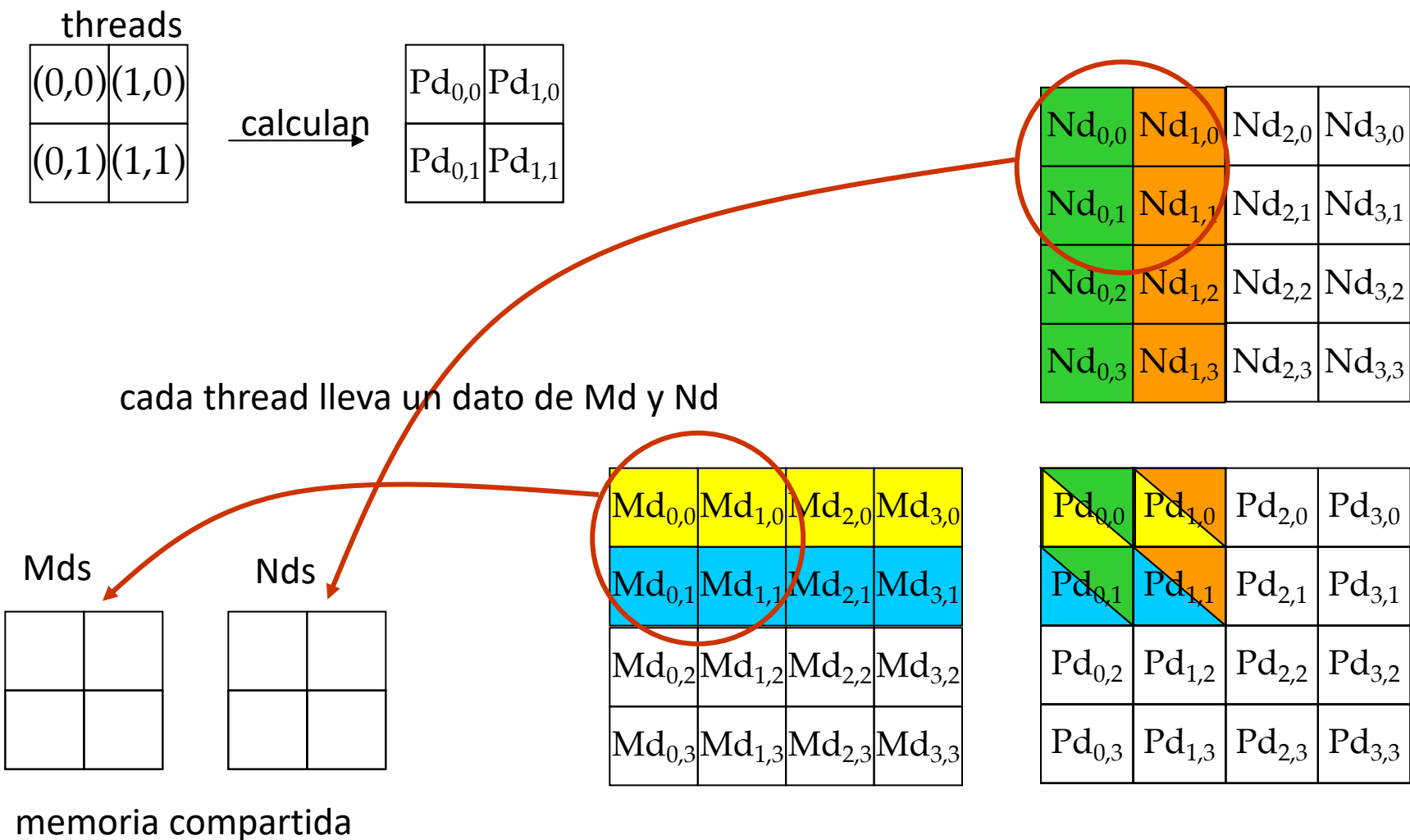


En una segunda fase se llevan a memoria compartida

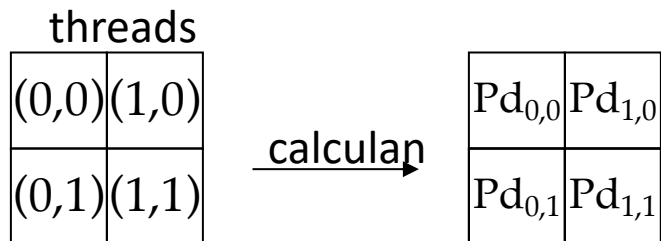


Cada  $Md_{i,j}$  y  $Nd_{i,j}$  se lee dos veces

# Ejemplo: Producto de Matrices 4X4



# Ejemplo: Producto de Matrices 4X4



$$Pd_{0,0} = Md_{0,0} \times Nd_{0,0} + Md_{1,0} \times Nd_{0,1}$$

$$Pd_{1,0} = Md_{0,0} \times Nd_{1,0} + Md_{1,0} \times Nd_{1,1}$$

$$Pd_{0,1} = Md_{0,1} \times Nd_{0,0} + Md_{1,1} \times Nd_{0,1}$$

$$Pd_{1,1} = Md_{0,1} \times Nd_{1,0} + Md_{1,1} \times Nd_{1,1}$$

Mds

$Md_{0,0}$	$Md_{1,0}$
$Md_{0,1}$	$Md_{1,1}$

Nds

$Nd_{0,0}$	$Nd_{1,0}$
$Nd_{0,1}$	$Nd_{1,1}$

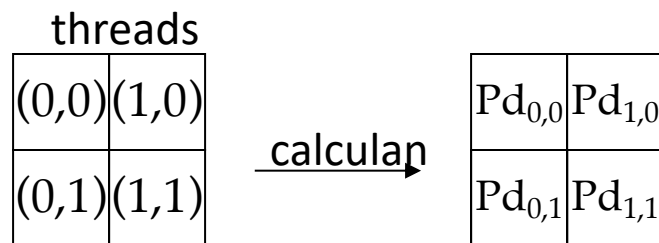
memoria compartida

$Nd_{0,0}$	$Nd_{1,0}$	$Nd_{2,0}$	$Nd_{3,0}$
$Nd_{0,1}$	$Nd_{1,1}$	$Nd_{2,1}$	$Nd_{3,1}$
$Nd_{0,2}$	$Nd_{1,2}$	$Nd_{2,2}$	$Nd_{3,2}$
$Nd_{0,3}$	$Nd_{1,3}$	$Nd_{2,3}$	$Nd_{3,3}$

$Md_{0,0}$	$Md_{1,0}$	$Md_{2,0}$	$Md_{3,0}$
$Md_{0,1}$	$Md_{1,1}$	$Md_{2,1}$	$Md_{3,1}$
$Md_{0,2}$	$Md_{1,2}$	$Md_{2,2}$	$Md_{3,2}$
$Md_{0,3}$	$Md_{1,3}$	$Md_{2,3}$	$Md_{3,3}$

$Pd_{0,0}$	$Pd_{1,0}$	$Pd_{2,0}$	$Pd_{3,0}$
$Pd_{0,1}$	$Pd_{1,1}$	$Pd_{2,1}$	$Pd_{3,1}$
$Pd_{0,2}$	$Pd_{1,2}$	$Pd_{2,2}$	$Pd_{3,2}$
$Pd_{0,3}$	$Pd_{1,3}$	$Pd_{2,3}$	$Pd_{3,3}$

# Ejemplo: Producto de Matrices 4X4



$$Pd_{0,0} = Md_{0,0} \times Nd_{0,0} + Md_{1,0} \times Nd_{0,1} + Md_{2,0} \times Nd_{0,2} + Md_{3,0} \times Nd_{0,3}$$

$$Pd_{1,0} = Md_{0,0} \times Nd_{1,0} + Md_{1,0} \times Nd_{1,1} + Md_{2,0} \times Nd_{1,2} + Md_{3,0} \times Nd_{1,3}$$

$$Pd_{0,1} = Md_{0,1} \times Nd_{0,0} + Md_{1,1} \times Nd_{0,1} + Md_{2,1} \times Nd_{0,2} + Md_{3,1} \times Nd_{0,3}$$

$$Pd_{1,1} = Md_{0,1} \times Nd_{1,0} + Md_{1,1} \times Nd_{1,1} + Md_{2,1} \times Nd_{1,2} + Md_{3,1} \times Nd_{1,3}$$

Nd <sub>0,0</sub>	Nd <sub>1,0</sub>	Nd <sub>2,0</sub>	Nd <sub>3,0</sub>
Nd <sub>0,1</sub>	Nd <sub>1,1</sub>	Nd <sub>2,1</sub>	Nd <sub>3,1</sub>
Nd <sub>0,2</sub>	Nd <sub>1,2</sub>	Nd <sub>2,2</sub>	Nd <sub>3,2</sub>
Nd <sub>0,3</sub>	Nd <sub>1,3</sub>	Nd <sub>2,3</sub>	Nd <sub>3,3</sub>

Mds		Nds	
Md <sub>2,0</sub>	Md <sub>3,0</sub>	Nd <sub>0,2</sub>	Nd <sub>1,2</sub>
Md <sub>2,1</sub>	Md <sub>3,1</sub>	Nd <sub>0,3</sub>	Nd <sub>1,3</sub>

memoria compartida

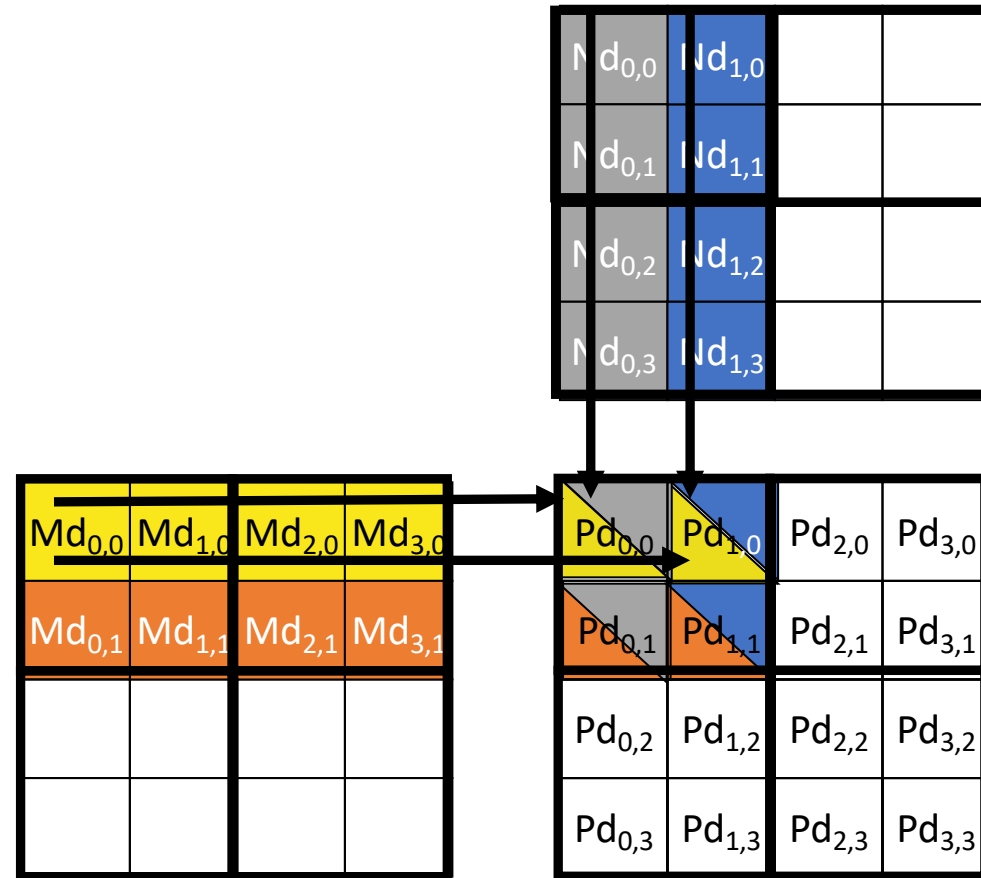
Md <sub>0,0</sub>	Md <sub>1,0</sub>	Md <sub>2,0</sub>	Md <sub>3,0</sub>
Md <sub>0,1</sub>	Md <sub>1,1</sub>	Md <sub>2,1</sub>	Md <sub>3,1</sub>
Md <sub>0,2</sub>	Md <sub>1,2</sub>	Md <sub>2,2</sub>	Md <sub>3,2</sub>
Md <sub>0,3</sub>	Md <sub>1,3</sub>	Md <sub>2,3</sub>	Md <sub>3,3</sub>

Pd <sub>0,0</sub>	Pd <sub>1,0</sub>	Pd <sub>2,0</sub>	Pd <sub>3,0</sub>
Pd <sub>0,1</sub>	Pd <sub>1,1</sub>	Pd <sub>2,1</sub>	Pd <sub>3,1</sub>
Pd <sub>0,2</sub>	Pd <sub>1,2</sub>	Pd <sub>2,2</sub>	Pd <sub>3,2</sub>
Pd <sub>0,3</sub>	Pd <sub>1,3</sub>	Pd <sub>2,3</sub>	Pd <sub>3,3</sub>




# Teselación de Md y Nd: Caso 4X4

- Rompemos los productos escalares en fases
- Al comienzo de cada fase, cargamos los elementos de Md y Nd que se necesitan en la memoria compartida
- Todos acceden a los elementos de Md y Nd vía la memoria compartida



Cada elemento Md y Nd se usa 2 veces en cada tesela 2X2 de P

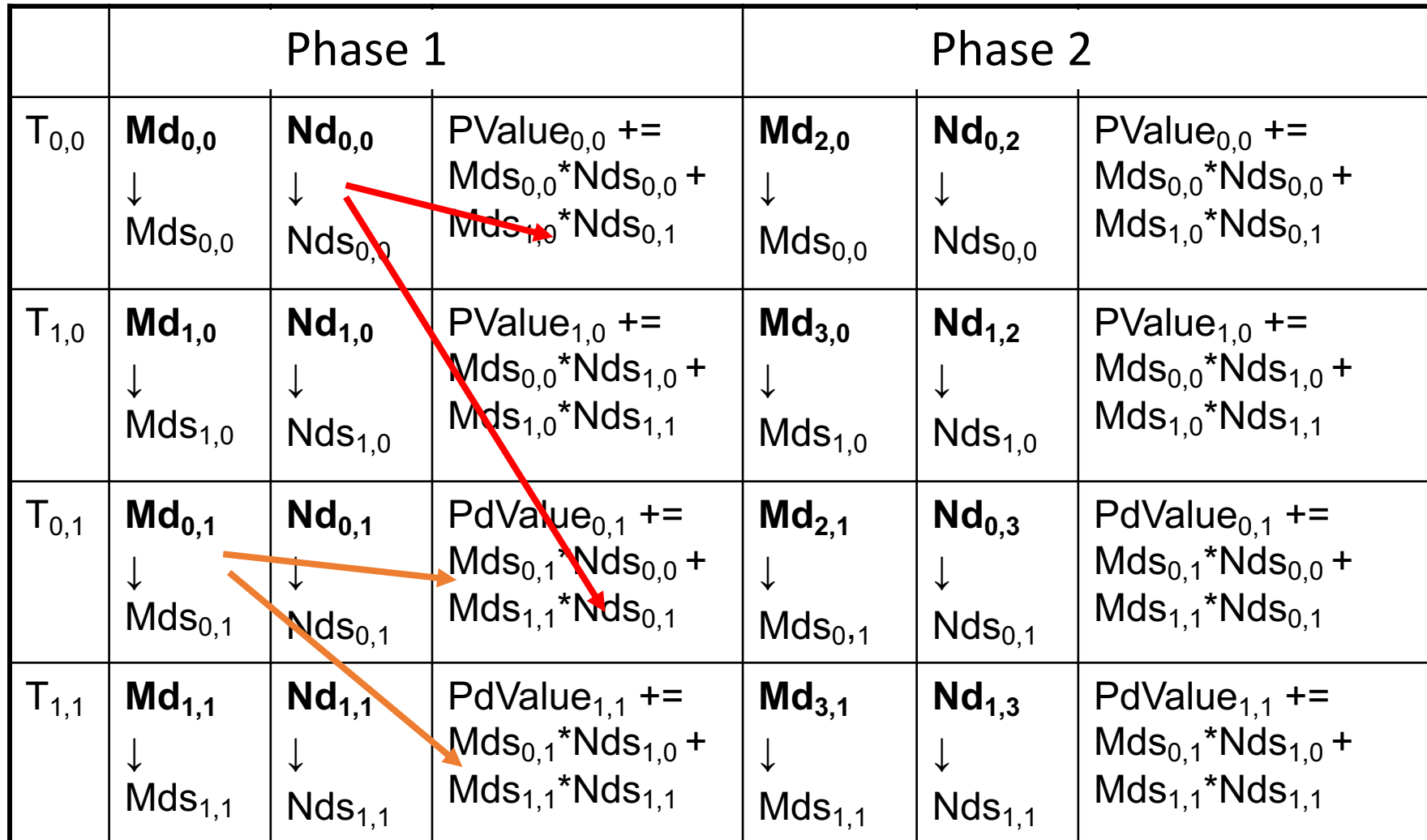
Orden  
Acceso



$P_{0,0}$ thread <sub>0,0</sub>	$P_{1,0}$ thread <sub>1,0</sub>	$P_{0,1}$ thread <sub>0,1</sub>	$P_{1,1}$ thread <sub>1,1</sub>
$M_{0,0} * N_{0,0}$	$M_{0,0} * N_{1,0}$	$M_{0,1} * N_{0,0}$	$M_{0,1} * N_{1,0}$
$M_{1,0} * N_{0,1}$	$M_{1,0} * N_{1,1}$	$M_{1,1} * N_{0,1}$	$M_{1,1} * N_{1,1}$
$M_{2,0} * N_{0,2}$	$M_{2,0} * N_{1,2}$	$M_{2,1} * N_{0,2}$	$M_{2,1} * N_{1,2}$
$M_{3,0} * N_{0,3}$	$M_{3,0} * N_{1,3}$	$M_{3,1} * N_{0,3}$	$M_{3,1} * N_{1,3}$

# Cada fase de un bloque de hilos usa una tesela de $Md$ y una de $Nd$

Tiempo



	Phase 1			Phase 2		
$T_{0,0}$	<b><math>Md_{0,0}</math></b> ↓ $Mds_{0,0}$	<b><math>Nd_{0,0}</math></b> ↓ $Nds_{0,0}$	$PValue_{0,0} +=$ $Mds_{0,0} * Nds_{0,0} +$ $Mds_{1,0} * Nds_{0,1}$	<b><math>Md_{2,0}</math></b> ↓ $Mds_{0,0}$	<b><math>Nd_{0,2}</math></b> ↓ $Nds_{0,0}$	$PValue_{0,0} +=$ $Mds_{0,0} * Nds_{0,0} +$ $Mds_{1,0} * Nds_{0,1}$
$T_{1,0}$	<b><math>Md_{1,0}</math></b> ↓ $Mds_{1,0}$	<b><math>Nd_{1,0}</math></b> ↓ $Nds_{1,0}$	$PValue_{1,0} +=$ $Mds_{0,0} * Nds_{1,0} +$ $Mds_{1,0} * Nds_{1,1}$	<b><math>Md_{3,0}</math></b> ↓ $Mds_{1,0}$	<b><math>Nd_{1,2}</math></b> ↓ $Nds_{1,0}$	$PValue_{1,0} +=$ $Mds_{0,0} * Nds_{1,0} +$ $Mds_{1,0} * Nds_{1,1}$
$T_{0,1}$	<b><math>Md_{0,1}</math></b> ↓ $Mds_{0,1}$	<b><math>Nd_{0,1}</math></b> ↓ $Nds_{0,1}$	$PdValue_{0,1} +=$ $Mds_{0,1} * Nds_{0,0} +$ $Mds_{1,1} * Nds_{0,1}$	<b><math>Md_{2,1}</math></b> ↓ $Mds_{0,1}$	<b><math>Nd_{0,3}</math></b> ↓ $Nds_{0,1}$	$PdValue_{0,1} +=$ $Mds_{0,1} * Nds_{0,0} +$ $Mds_{1,1} * Nds_{0,1}$
$T_{1,1}$	<b><math>Md_{1,1}</math></b> ↓ $Mds_{1,1}$	<b><math>Nd_{1,1}</math></b> ↓ $Nds_{1,1}$	$PdValue_{1,1} +=$ $Mds_{0,1} * Nds_{1,0} +$ $Mds_{1,1} * Nds_{1,1}$	<b><math>Md_{3,1}</math></b> ↓ $Mds_{1,1}$	<b><math>Nd_{1,3}</math></b> ↓ $Nds_{1,1}$	$PdValue_{1,1} +=$ $Mds_{0,1} * Nds_{1,0} +$ $Mds_{1,1} * Nds_{1,1}$

En general, si una matriz de entrada es de dimensión  $N$  y el tamaño de mosaico es  $TILE\_WIDTH$ , el producto escalar se realiza en  $N / TILE\_WIDTH$  fases.

# Núcleo de la multiplicación teselada

```
__global__ void MatrixMulKernel
(float* Md, float* Nd, float* Pd, int Width) {
    __shared__ __float Mds[TILE_WIDTH][TILE_WIDTH];
    __shared__ __float Nds[TILE_WIDTH][TILE_WIDTH];

    int bx = blockIdx.x; int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;

    // Identify the row and column of
    // the Pd element to work on
    int Row = by * TILE_WIDTH + ty;
    int Col = bx * TILE_WIDTH + tx;

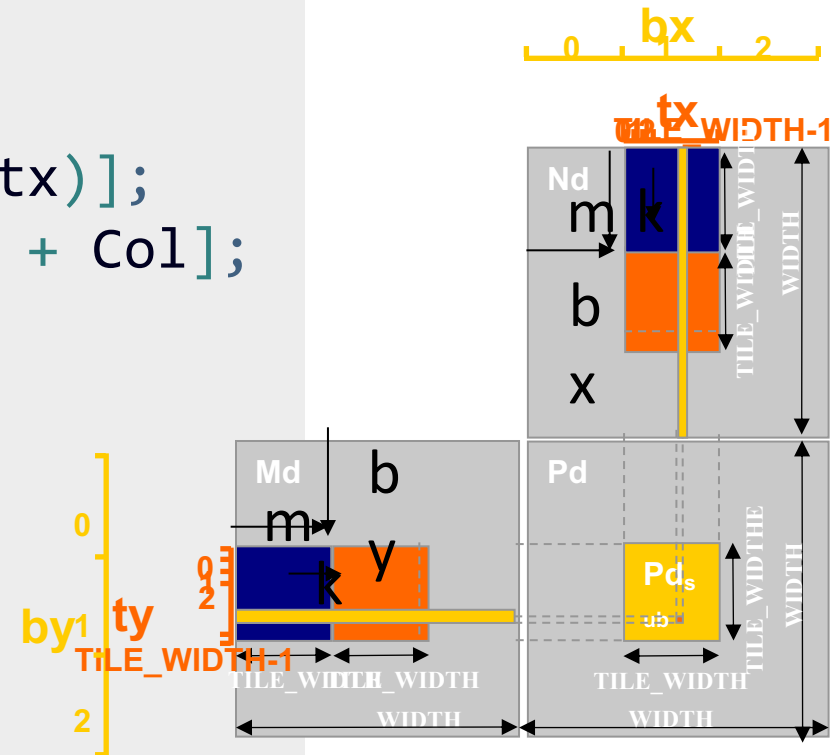
    float Pvalue = 0;
}
```

# Núcleo de la multiplicación teselada (y2)

```
// Loop over the Md and Nd tiles required
// to compute the Pd element

for (int m = 0; m < Width/TILE_WIDTH; ++m) {
    // Collaborative loading of Md and Nd
    // tiles into shared memory
    Mds[ty][tx]= Md[Row*Width+(m*TILE_WIDTH+tx)];
    Nds[ty][tx]=Nd[(m*TILE_WIDTH + ty)*Width + Col];
    __syncthreads();

    for (int k = 0; k < TILE_WIDTH; ++k)
        Pvalue += Mds[ty][k] * Nds[k][tx];
    __syncthreads();
}
Pd[Row*Width + Col] = Pvalue;
```



# Configuración de la ejecución del kernel

```
// Setup the execution configuration  
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);  
  
dim3 dimGrid(Width / TILE_WIDTH, Width / TILE_WIDTH);
```

# Tratamiento errores

# Manejo de Errores

- En CUDA todas las funciones (excepto los lanzamientos de un kernel) devuelven un código de error del tipo `cudaError_t`.
- Este código es simplemente un valor entero y toma distintos valores dependiendo del tipo de error encontrado.
- Cuando la llamada a una función finalizada con éxito, el código de error toma un valor definido como `cudaSuccess`.
- El principal inconveniente de este procedimiento es que puede resultar tedioso al tener que comprobar continuamente el código de error devuelto por cada función e imprimir el mensaje de error



# Manejo de Errores

```
// ...
cudaError_t error;
// ...
error = cudaMalloc( . . . );
if (error != cudaSuccess) {
    printf("\nERROR en cudaMalloc: %s \n",
        cudaGetErrorString(error) );
    printf("\npulsa INTRO para finalizar...");
    fflush(stdin);
    char tecla = getchar();
    exit(-1);
}
// ...
```

enum cudaError

CUDA error types

Values

cudaSuccess = 0

The API call returned with no errors. In the case of query calls, this also means that the operation being queried is complete (see [cudaEventQuery\(\)](#) and [cudaStreamQuery\(\)](#)).

cudaErrorInvalidValue = 1

This indicates that one or more of the parameters passed to the API call is not within an acceptable range of values.

cudaErrorMemoryAllocation = 2

The API call failed because it was unable to allocate enough memory to perform the requested operation.

cudaErrorInitializationError = 3

The API call failed because the CUDA driver and runtime could not be initialized.

cudaErrorCudartUnloading = 4

This indicates that a CUDA Runtime API call cannot be executed because it is being called during process shut down, at a point in time after CUDA driver has been unloaded.

cudaErrorProfilerDisabled = 5

This indicates profiler is not initialized for this run. This can happen when the application is running with external profiling tools like visual profiler.

cudaErrorProfilerNotInitialized = 6

**Deprecated**

This error return is deprecated as of CUDA 5.0. It is no longer an error to attempt to enable/disable the profiling via [cudaProfilerStart](#) or [cudaProfilerStop](#) without initialization.

# Manejo de Errores

La función `cudaGetErrorString()` se encarga de devolver un mensaje de texto explicando el tipo de error

Función dedicada al chequeo de errores, con un único parámetro de entrada que sea el mensaje de error que deseamos que aparezca por pantalla, y colocarla justo después de la llamada a la función cuyo código de error queramos saber

```
__host__ void check_CUDA_Error(const char *mensaje)
{
    cudaError_t error;
    cudaDeviceSynchronize();
    error = cudaGetLastError();
    if(error != cudaSuccess)
    {
        printf("ERROR %d: %s (%s)\n", error, cudaGetErrorString(error), mensaje);
        printf("\npulsa INTRO para finalizar...");
        fflush(stdin);
        char tecla = getchar();
        exit(-1);
    }
}
```

# Manejo de Errores

En la definición de esta función se han utilizado dos nuevas funciones de CUDA que son `cudaDeviceSynchronize()` y `cudaGetLastError()`.

Si queremos detectar posibles errores en alguna transferencia de datos o en el lanzamiento del kernel, podemos incluir en el código llamadas a la función `check_CUDA_Error()`, con un mensaje de error particular

```
// MAIN: rutina principal ejecutada en el host
int main(int argc, char** argv)
{
    // ...
    // copia de datos hacia el device
    cudaMemcpy( dev_A, hst_A, size, cudaMemcpyHostToDevice );
    check_CUDA_Error("ERROR EN cudaMemcpy");
    // ...
    // llamada al kernel multiplica <<< nBloques,hilosB>>> (dev_A);
    check_CUDA_Error("ERROR EN multiplica");
    // ...
    // salida
    printf("\npulsa INTRO para finalizar...");
    fflush(stdin);
    char tecla = getchar();
    return 0;
}
```

# Manejo de Errores

```
// cudaDeviceReset must be called before exiting in order for profiling and
// tracing tools such as Nsight and Visual Profiler to show complete traces.
    cudaStatus = cudaDeviceReset();
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaDeviceReset failed!");
        return 1;
    }

// Choose which GPU to run on, change this on a multi-GPU system.
    cudaStatus = cudaSetDevice(0);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaSetDevice failed! Do you have a CUDA-capable GPU installed?");
        goto Error;
    }

// Allocate GPU buffers for three vectors (two input, one output) .
    cudaStatus = cudaMalloc((void**)&dev_c, size * sizeof(int));
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMalloc failed!");
        goto Error;
    }
```

# Limites en los bloques

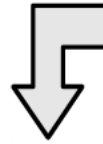
$$\text{bloquesX} = \left\lceil \frac{\text{columnas matriz}}{\text{número de threads por bloque}} \right\rceil$$

$$\text{bloquesY} = \left\lceil \frac{\text{filas matriz}}{\text{número de threads por bloque}} \right\rceil$$

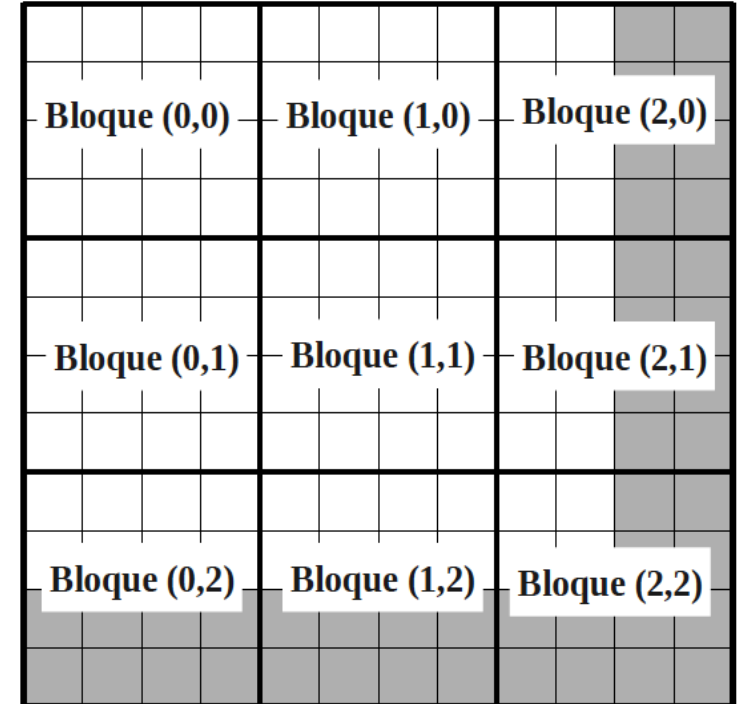
$\text{desplazamiento} = x + y * \text{número de columnas matriz}$

```
__global__ void sumaMatrices(int *x, int *y, int *z)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;

    int indice = i + j * N;
    if (i < N && j < M)
        z[indice] = x[indice] + y[indice];
}
```



Thread (0,0)	Thread (1,0)	Thread (2,0)	Thread (3,0)
Thread (0,1)	Thread (1,1)	Thread (2,1)	Thread (3,1)
Thread (0,2)	Thread (1,2)	Thread (2,2)	Thread (3,2)
Thread (0,3)	Thread (1,3)	Thread (2,3)	Thread (3,3)

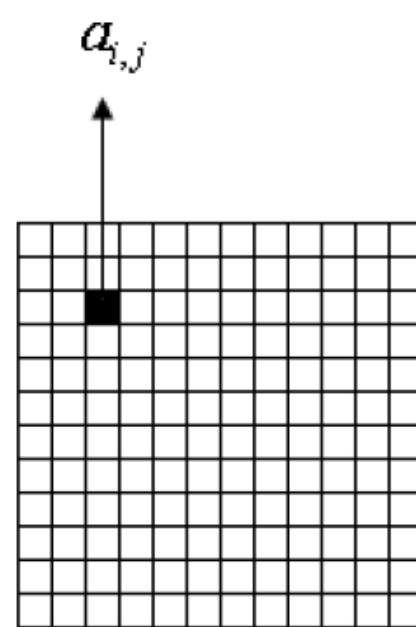


Threads utilizados



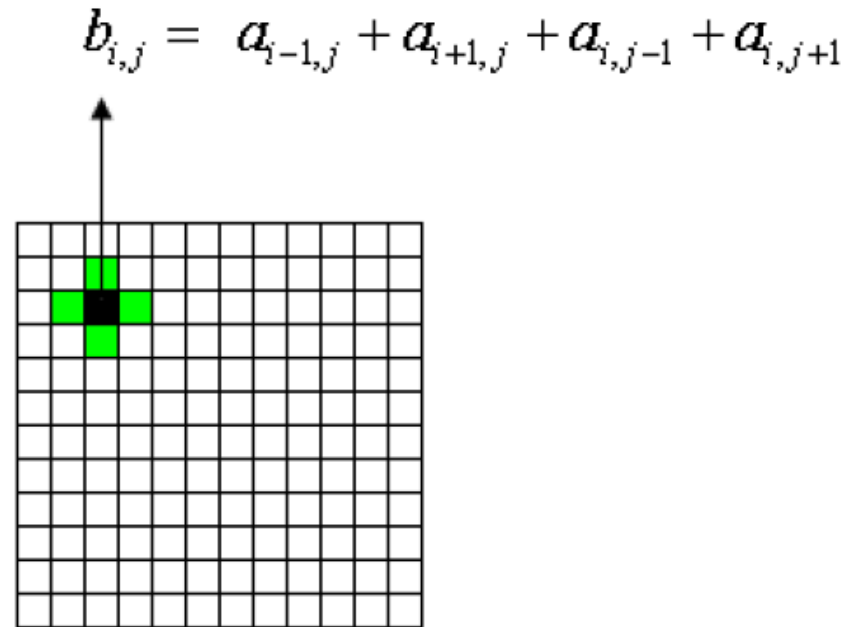
Threads no utilizados

# Ejercicio



Matriz original

**A**



Matriz resultado

**B**