

COMPUTE UNIFIED DEVICE ARCHITECTURE



Basado en <https://docs.nvidia.com/cuda>

CUDA PRESENTACIÓN

- En el Q4 de 2006 NVIDIA presenta CUDA (*Compute Unified Device Architecture*).
- La GPGPU (*General-Purpose Computing on Graphics Processing Units*) introducida con CUDA busca la *escalabilidad automática*: el software debe escalar transparentemente al aumentar el número de *cores*.
- La estrategia:
 - Dividir el problema en sub-problemas “grandes” que pueden ejecutarse en paralelo de forma independiente (**bloques de hilos**).
 - Dividir cada sub-problema en piezas más finas que se resuelven concurrente y cooperativamente (**hilos**).
 - Cada bloque de hilos puede ejecutarse en cualquiera de los procesadores disponibles, en cualquier orden, simultánea o secuencialmente.
- La *Compute Capability* de una GPU viene definida por su revisión *major.minor*.
 - Igual *major* misma arquitectura. *minor* denotan mejoras incrementales sobre el núcleo de la arquitectura. Los *major* son: ~~1 Tesla~~, ~~2 Fermi~~, 3 Kepler, 5 Maxwell, 6 Pascal, 7 Volta y 8 Ampere.
 - Con el *major.minor* se sabe qué “cosas” CUDA soporta/puede hacer la GPU.

CUDA PRESENTACIÓN

CUDA Occupancy Calculator - Nvidia

(<https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html>)

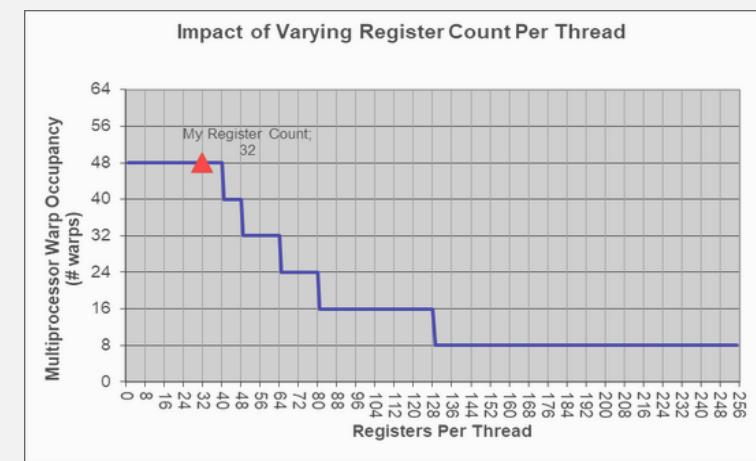
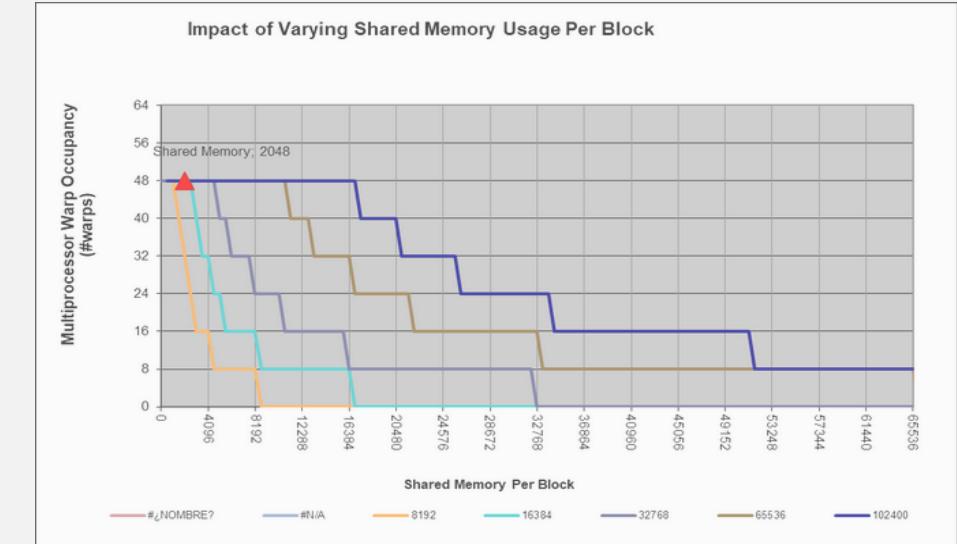
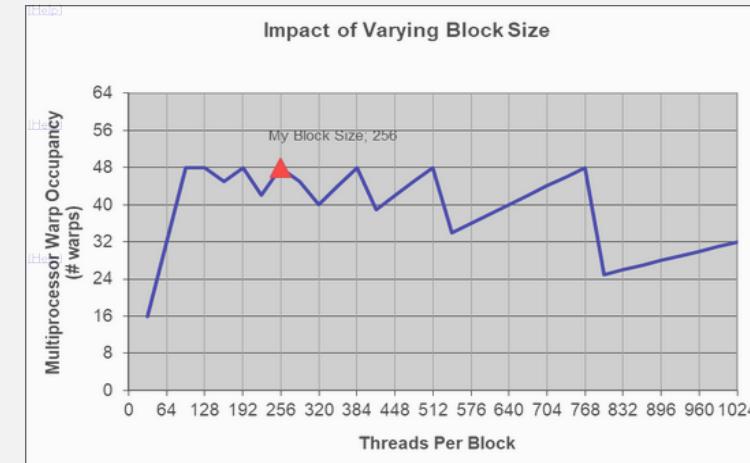
Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click): 8,6
 1.b) Select Shared Memory Size Config (bytes): 65536
 1.c) Select CUDA version: 11,1

2.) Enter your resource usage:
 Threads Per Block: 256
 Registers Per Thread: 32
 User Shared Memory Per Block (bytes): 2048
 (Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:
 Active Threads per Multiprocessor: 1536
 Active Warps per Multiprocessor: 48
 Active Thread Blocks per Multiprocessor: 6
 Occupancy of each Multiprocessor: 100%
 Physical Limits for GPU Compute Capability: 8,6
 Threads per Warp: 32
 Max Warps per Multiprocessor: 48
 Max Thread Blocks per Multiprocessor: 16
 Max Threads per Multiprocessor: 1536
 Maximum Thread Block Size: 1024
 Registers per Multiprocessor: 65536
 Max Registers per Thread Block: 65536
 Max Registers per Thread: 255
 Shared Memory per Multiprocessor (bytes): 65536
 Max Shared Memory per Block: 65536
 Register allocation unit size: 256
 Register allocation granularity: warp
 Shared Memory allocation unit size: 128
 Warp allocation granularity: 4
 Shared Memory Per Block (bytes) (CUDA runtime use): 1024
 Allocated Resources = Allocatable
 Per Block Limit Per SM Blocks Per SM
 Warps (Threads Per Block / Threads Per Warp): 8 48 8
 Registers (Warp limit per SM due to per-warp reg count): 8 64 8
 Shared Memory (Bytes): 2048 65536 32
 Note: SM is an abbreviation for (Streaming) Multiprocessor
 Maximum Thread Blocks Per Multiprocessor: Blocks/SM * Warps/Block = Warps/SM
 Limited by Max Warps or Max Blocks per Multiprocessor: 6 8 48
 Limited by Registers per Multiprocessor: 8
 Limited by Shared Memory per Multiprocessor: 32
 Note: Occupancy limiter is shown in orange
 Physical Max Warps/SM = 48
 Occupancy = 48 / 48 = 100%

CUDA Occupancy Calculator
 Version: 11,1
 Copyright and License



CUDA ECOSISTEMA

- NVIDIA CUDA-X:
 - Math Libraries:
 - cuBLAS, cuFFT, cuRAND, cuSOLVER, cuSPARSE, AmgX y CUDA Math Library.
 - cuTENSOR. GPU-accelerated tensor linear algebra library
 - Parallel Algorithms
 - Image and Video Libraries:
 - nvJPEG, NVIDIA Performance Primitives, NVIDIA Video Codec SDK y NVIDIA Optical Flow SDK.
 - Communication Libraries:
 - NVSHMEM y NCCL
 - Deep Learning:
 - NVIDIA cuDNN, NVIDIA TensorRT™, NVIDIA Jarvis, NVIDIA DeepStream SDK y NVIDIA DALI
- NVIDIA HPC SDK. Incluye compiladores, librerías y herramientas software para maximizar productividad, rendimiento y portabilidad de aplicaciones HPC.
- NVIDIA GPU Cloud™ (NGC). Contenedores para *frameworks* AI y aplicaciones HPC.
- ...

PROGRAMANDO ¿QUÉ ES CUDA?

- CUDA es C/C++ más un conjunto extensiones, tales como:
 - Calificadores de declaración que especifican lugares o ámbitos:

global void xxx(...)	<i>kernel</i> , se ejecuta en la GPU
device int xxx	variable en la memoria de la GPU
shared int xxx	variable en memoria compartida dentro del bloque
 - Sintaxis ampliada para la llamada al *kernel*:

kernel<<<a, b,c,d,>>>(...)	lanza la ejecución del <i>kernel</i>
---	--------------------------------------
 - Nuevos tipos / warpers:

dim3	vector de 3 enteros
-------------	---------------------
 - Variables especiales (intrínsecas) para identificar los hilos en el *kernel*:

threadIdx / blockIdx	ID de los hilos y bloques
blockDim / gridDim	dimensiones de los bloques y las <i>grid</i>
 - Operaciones específicas (intrínsecas) dentro del *kernel*:

_syncthreads()	barreda de sincronización dentro del <i>kernel</i>
-----------------------	--

PROGRAMANDO ¿QUÉ ES CUDA?

- Un *kernel* se ejecuta en paralelo por diferentes hilos CUDA.

Ejemplo definición función en C

```
void VecAdd(float* A, float* B, float* C) {  
    ...  
}
```

Ejemplo definición de kernel

```
_global_ void VecAdd(float* A, float* B, float* C) {  
    ...  
}
```

Ejemplo llamada a función en C

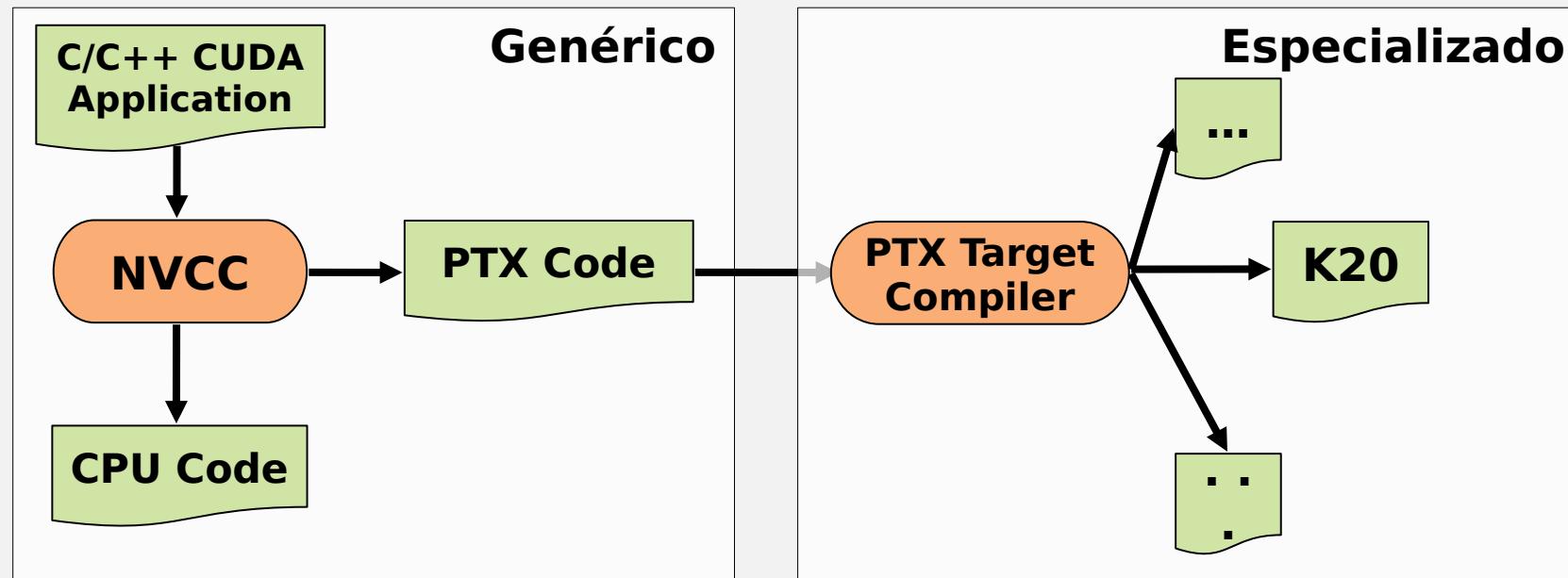
```
VecAdd(A, B, C);
```

Ejemplo llamada a kernel

```
VecAdd<<<1, n>>>(A, B, C);
```

PROGRAMANDO ¿QUÉ ES CUDA?

- El compilador **nvcc** separa los códigos CPU y GPU. Compilación en dos etapas:
 - **Virtual** genera código PTX (*Parallel Thread eXecution*).
 - **Física** genera binarios para GPU y CPU.



CUDA MODELO ORGANIZATIVO

- Los kernels se ejecutan en el dispositivo. Distintos kernels se pueden ejecutar simultáneamente.
- El kernel es ejecutado por hilos. Los hilos ejecutan el mismo código sobre diferentes datos basándose en su **Id**.
- Los hilos se agrupan en bloques de hilos.
- Los hilos del mismo bloque pueden **sincronizarse** (con `_syncthreads()`). Los hilos del mismo bloque pueden **compartir** datos (con *shared memory*).
- Los bloques no pueden sincronizarse: se ejecutan en cualquier orden, secuencial o paralelo.
- **Los kernels son asíncronos respecto a la CPU (retorno inmediato del control)**.
- Los kernels (del mismo *stream*), no comienzan su ejecución hasta que no hayan finalizado todas las llamadas CUDA anteriores ,p. ej. otro kernel (del mismo stream).
- Los kernels no finalizan hasta que finalicen todos sus hilos.

CUDA MODELO PROGRAMACIÓN

- GPU → GPC → SM → Core CUDA
- Bloque → Warp → hilo
- Dato → hilo

{}

¿ Cómo ?

- Cada hilo que ejecuta el kernel recibe un identificador (**Id**).
- Existe, puede existir, una relación entre el dato y el Id.
- A este Id se accede dentro del kernel a través de **variables intrínsecas**.
- Intrínseca **threadIdx** {threadIdx.x, threadIdx.y, threadIdx.z}
 - Identificador del hilo dentro del bloque (rango [0, blockSize-1]).
 - Los hilos están organizados dentro del bloques en 1, 2 ó 3 dimensiones.
 - Vector de 3 componentes. Hay un nuevo tipo de dato (**dim3**) a tal efecto.

CUDA MODELO PROGRAMACIÓN

- Hacer un kernel que sume los elementos de 2 vectores de tamaño n=1000000

Ejemplo V.1

```
__global__ void VecAdd(float* A, float* B,
float* C) {
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
VecAdd<<<1, n>>>(A, B, C);
```

¿ Resultado ejecución ?
*cudaCheckError() failed:
invalid configuration argument*

Ver 1

Ejemplo V.2

```
__global__ void VecAdd(float* A, float* B,
float* C) {
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
VecAdd<<<n/10, 10>>>(A, B, C);
```

¿ Resultado ejecución ?
*Error similar al de la V.1:
algo del número de bloques*

Ver 1

CUDA MODELO PROGRAMACIÓN

- Hacer un kernel que sume los elementos de 2 vectores de tamaño

Ejemplo V.3

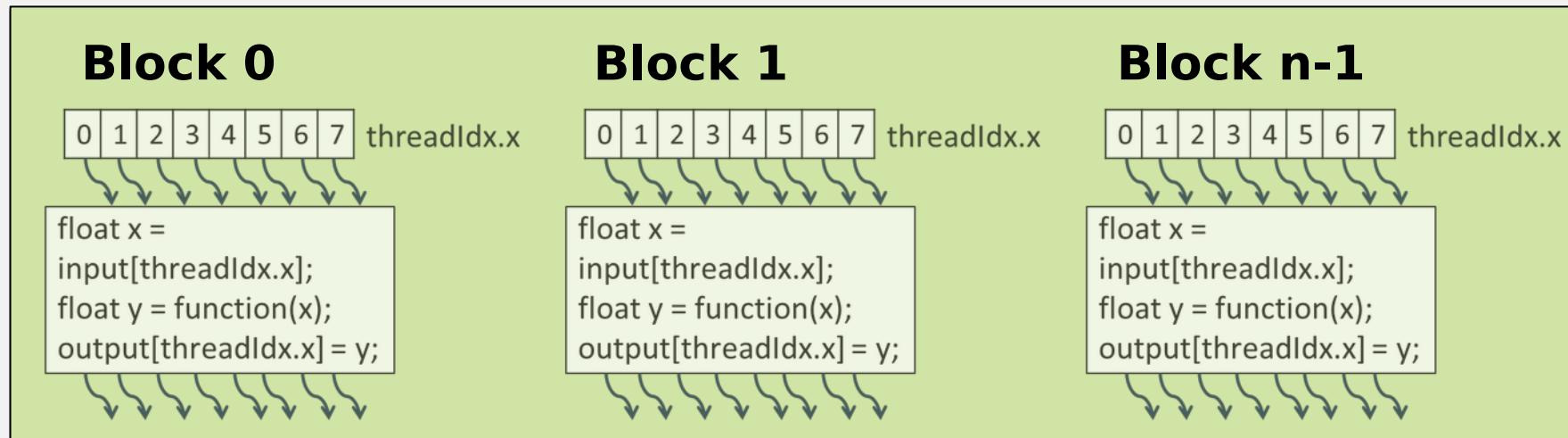
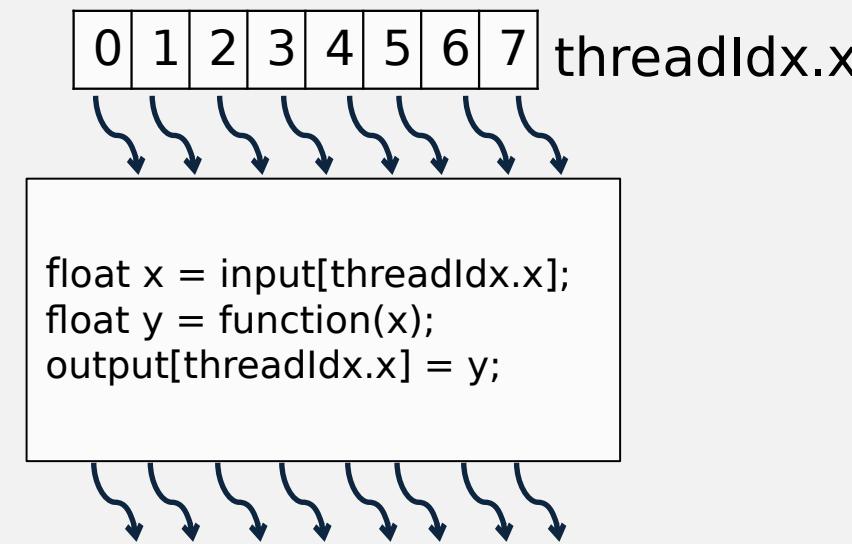
```
__global__ void VecAdd(float* A, float* B,  
float* C) {  
    int i = threadIdx.x;  
    C[i] = A[i] + B[i];  
}  
  
VecAdd<<<1000000/101, 101>>>(A, B, C);
```

¿ Resultado ejecución ?

(Suponiendo que no da los errores de V.1 y V.2)

Suma mal: suma (1000000/101) veces las 101 primeras posiciones.

CUDA MODELO ORGANIZATIVO



CUDA MODELO PROGRAMACIÓN

- Hacer un kernel que sume los elementos de 2 vectores de tamaño n=1000000

Ejemplo V.3

```
__global__ void VecAdd(float* A, float* B,  
float* C) {  
    int i = threadIdx.x;  
    C[i] = A[i] + B[i];  
}  
  
VecAdd<<<1000000/101, 101>>>(A, B, C);
```

Intrínseca **blockIdx** {blockIdx.x, blockIdx.y, blockIdx.z}

- Identificador del bloque dentro de la *grid*.
- Los bloques están organizados en *grids* de bloques de 1, 2 o 3 dimensiones.
- blockDim** es otra intrínseca *dim3* con las dimensiones del bloque.

CUDA MODELO PROGRAMACIÓN

- Hacer un kernel que sume los elementos de 2 vectores de tamaño n=1000000

Ejemplo V.4

```
__global__ void VecAdd(float* A, float* B, float* C) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    C[i] = A[i] + B[i];
}

VecAdd<<<1000000/101, 101>>>(A, B, C);
```

¿ Resultado ejecución ?

(Suponiendo que no da los errores de V.1, V.2 y V.3)

Suma mal: solo suma 999.900 posiciones.

Ejemplo V.5

```
__global__ void VecAdd(float* A, float* B, float* C) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    C[i] = A[i] + B[i];
}
```

¿ Resultado ?

```
VecAdd<<< (n + threadsPerBlock - 1) /  

threadsPerBlock, threadsPerBlock>>>(A, B, C);
```

CUDA MODELO PROGRAMACIÓN

- Hacer un kernel que sume los elementos de 2 vectores de tamaño

Ejemplo V.6

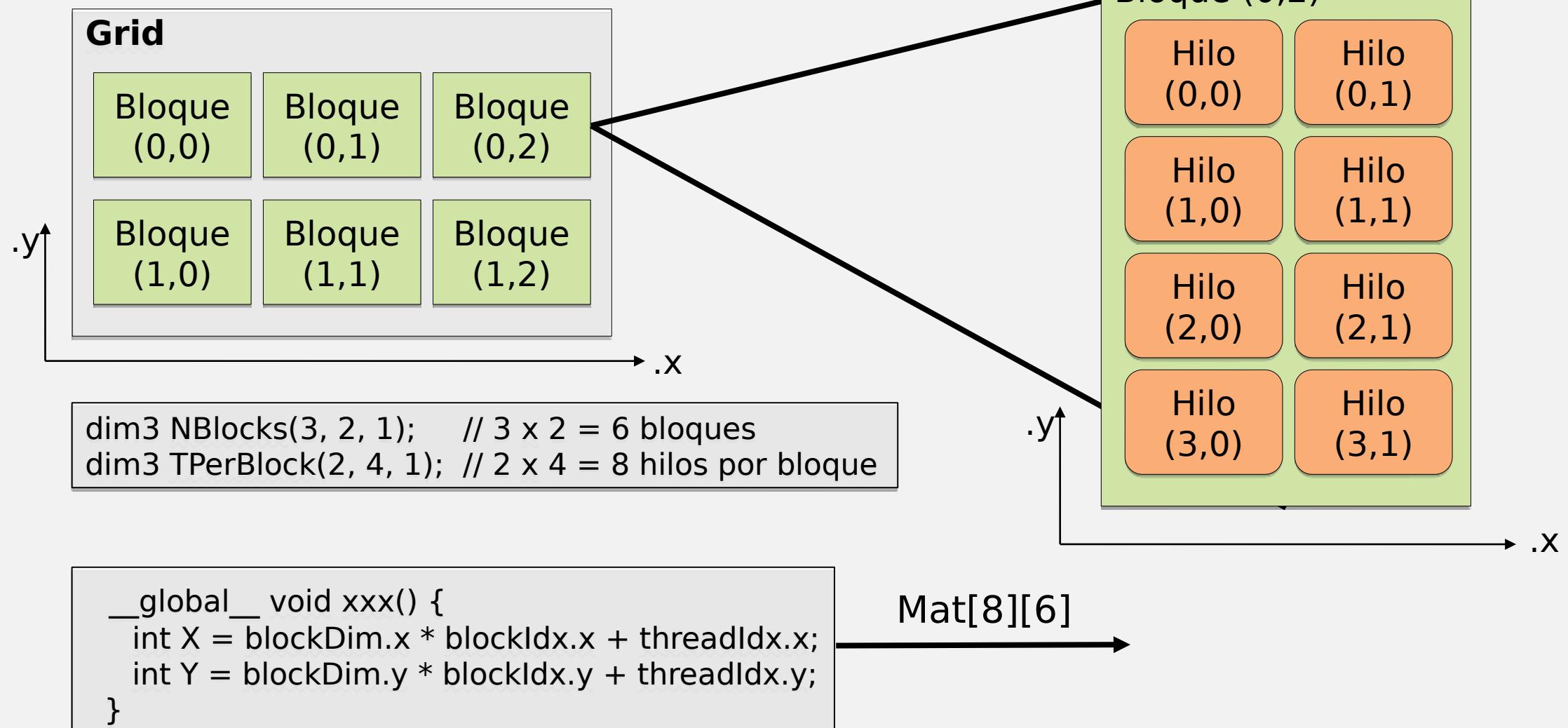
```
__global__ void VecAdd(float* A, float* B, float* C, int n) {  
    int i = blockIdx.x * blockDim.x + threadIdx.x;  
    if (i < n)  
        C[i] = A[i] + B[i];  
}  
VecAdd<<<(n + threadsPerBlock - 1)/threadsPerBlock,  
threadsPerBlock>>>(A, B, C, n);
```

¿ Resultado para **n** muy grande ?

*Podemos volver a los errores de la V.1 ó V.2
según sean los valores de threadsPerBlock y de n.*

CUDA MODELO ORGANIZATIVO

- La Grid: cómo se organizan los bloques



CUDA MODELO PROGRAMACIÓN

- Hacer un *kernel* que sume los elementos de 2 vectores de tamaño 4000x4000

Ejemplo V.7

```
__global__ void VecAdd(float* A, float* B, float* C, int rows, int cols) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    if ((y < rows) && (x < cols))
        C[y*cols+x] = A[y*cols+x] + B[y*cols+x];
}
dim3 TPB2d(threadsPerBlock, threadsPerBlock);
dim3 NB2d( (sqrt(n) + TPB2d.x - 1) / TPB2d.x, (sqrt(n) + TPB2d.y - 1) / TPB2d.y );
VecAdd<<< NB2d, TPB2d >>>(A, B, C, sqrt(n), sqrt(n));
```

Puede dar el mismo error que V.1, según el valor *threadPerBlock*

V.6 vs V.7

n = 4000x4000

Tiempo V.6 con: 1.0213000E-02 segundos

Tiempo V.7 con: 2.6340000E-03 segundos

CUDA MODELO PROGRAMACIÓN

Ejemplo V.8 (también válido para sumar matrices)

```
#define BLOCKSIZE 32

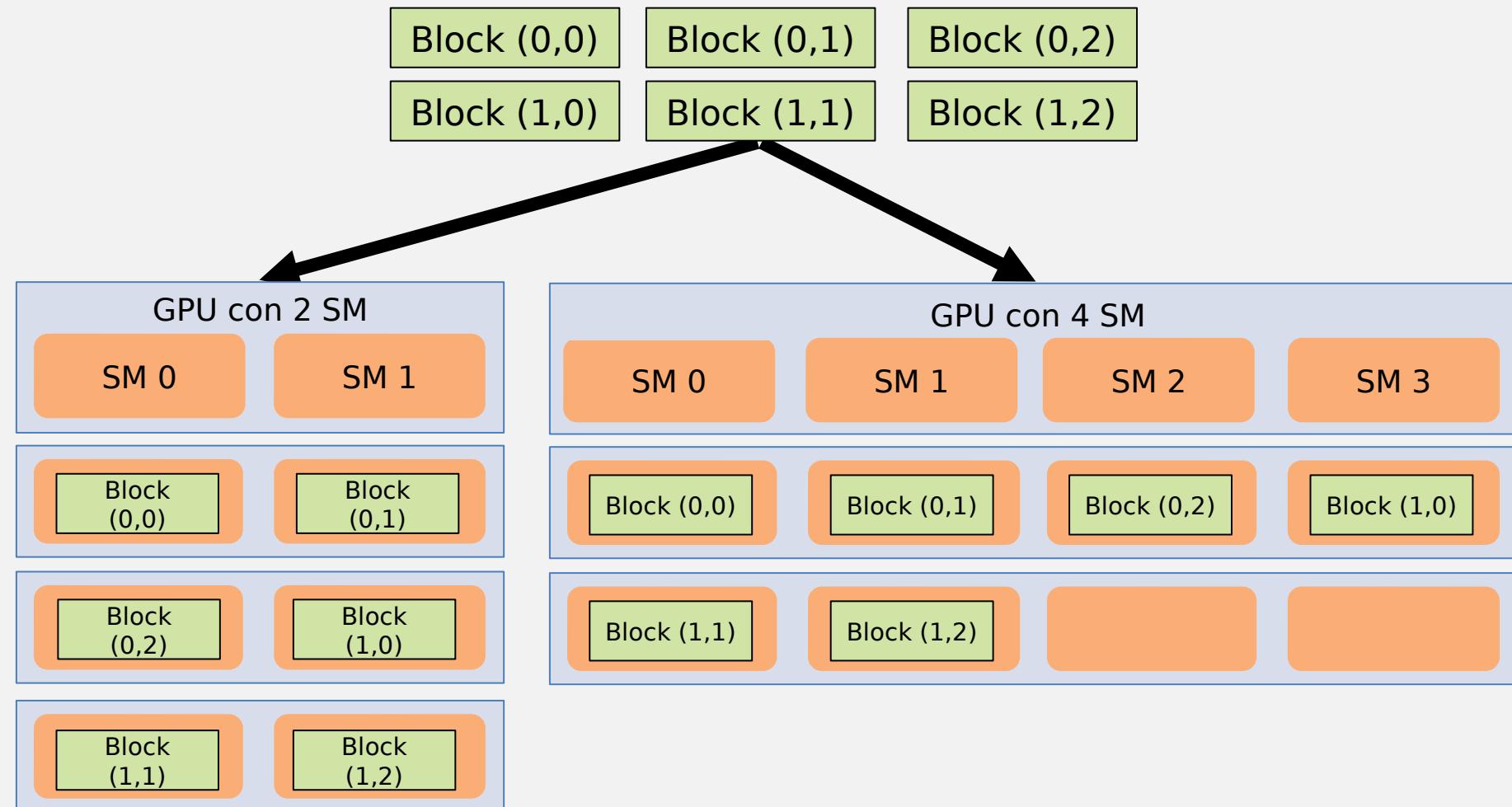
__global__ void MatSum(double *A, double *B, double *C, int rows, int cols) {
    int X= blockIdx.x * blockDim.x + threadIdx.x;
    int Y= blockIdx.y * blockDim.y + threadIdx.y;
    int index = Y * cols + X;

    if (Y < rows && X < cols) {
        C[index] = A[index] + B[index];
    }
}

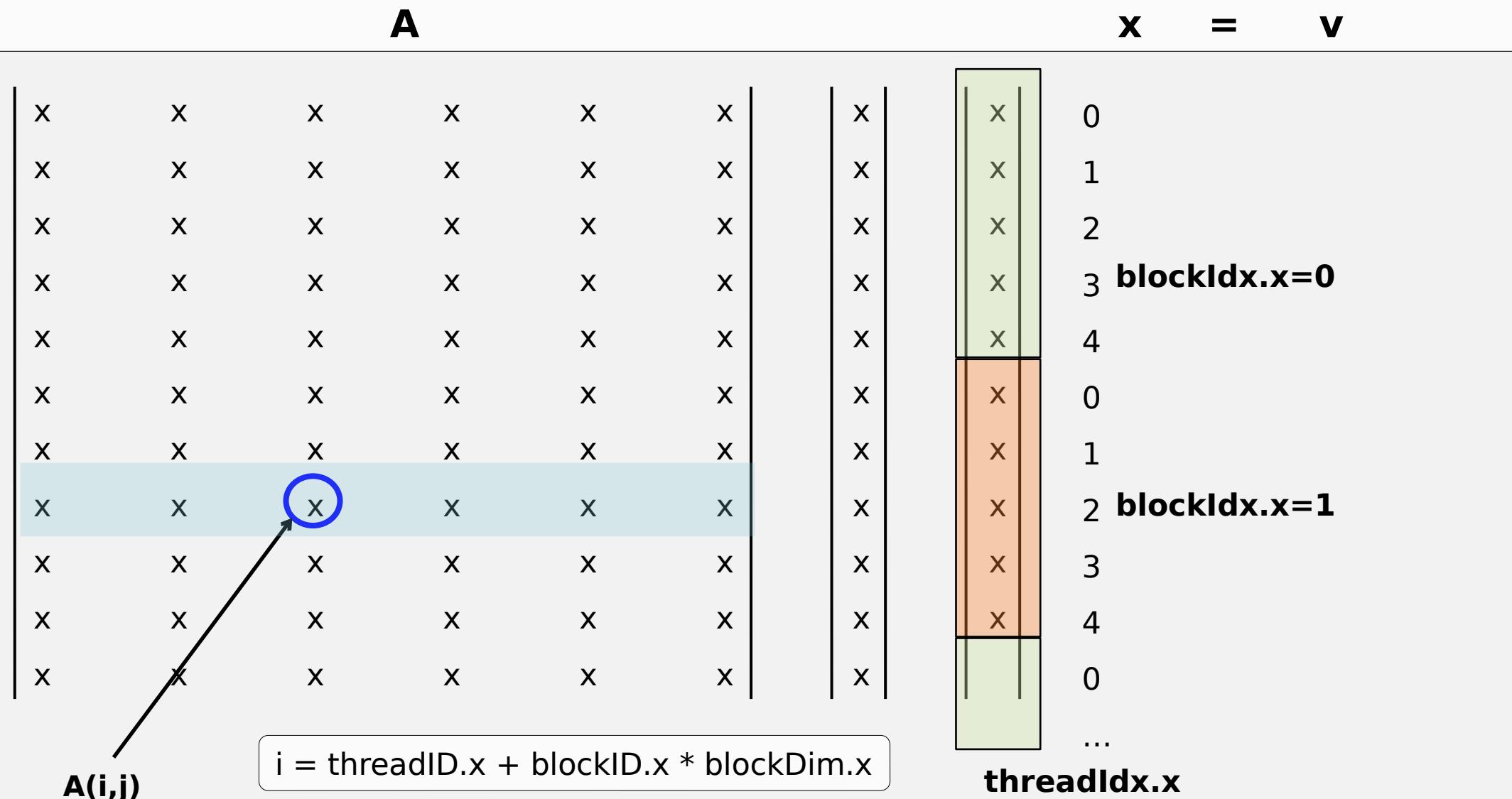
int main() {
    ...
    X = (int) ceil( (float)cols / BLOCKSIZE);
    Y = (int) ceil( (float)rows / BLOCKSIZE);
    dim3 TPBLCK(BLOCKSIZE, BLOCKSIZE);
    dim3 NBLCK(X, Y);
    MatSum<<<NBLCK, TPBLCK>>>(A, B, C, rows, cols);
    ...
}
```

CUDA MODELO ORGANIZATIVO

- Escalabilidad automática



EJEMPLO



EJEMPLO

Ejemplo $v = A * x$

```
__global__ void MatdotVec(double *A, double *x, double *v, int rows, int cols) {
    int i = blockIdx.x * blockDim.x + threadIdx.x; // global matrix row index
    int j;
    double tmp=0.0;

    if (i < rows)
        for (j=0; j < cols; j++)
            tmp += A[i*cols+j] * x[j];
    v[i] = tmp;
}

int main() {
    ...
    dim3 TPBLCK(XXX, 1, 1); dim3 NBLCK(ZZZ, 1, 1);
    MatdotVec<<<NBLCK, TPBLCK>>>(A, x, v, rows, cols);
    ...
}
```

CUDA MODELO ORGANIZATIVO

¿Por qué, en general, almacenar matrices como vectores?

- Por todo lo visto hasta ahora: localidad, librerías, etc.

Y entonces ¿por qué en CUDA usan *grids/blocks* 2D si lo *recomendado* es almacenar en 1D? ¿No es complicarlo?

- Por flexibilidad, limitaciones, rendimiento potencialmente, etc.
- La localidad en GPU es igual de importante que en CPU, pero se *entiende* de otra manera (segunda parte).

Ejemplo binarizado

binariza 1000 2000 16

El tiempo en CPU: 3.8437440E-03 segundos.

El tiempo con CuBLAS 1D: vecbin.cu(114): getLastCudaError() CUDA error...

El tiempo con CuBLAS 2D: 2.1923199E-04 segundos.

binariza 1000 2000 32

El tiempo en CPU: 3.8437440E-03 segundos.

El tiempo con CuBLAS 1D: 4.7323201E-04 segundos.

El tiempo con CuBLAS 2D: 2.7491199E-04 segundos.

CUDA CONTROL ERRORES

- Todas las funciones de la API de CUDA (cuBLAS, cuFFT, etc.) retornan un valor. Se debe comprobar.

Ejemplo

```
int main() {  
    ...  
    if (cudaMemcpy(...) != cudaSuccess)  
}
```

Mejor así

```
#define CUDAERR(x) do { if((x)!=cudaSuccess) { \  
    printf("CUDA error: %s : %s, line %d\n", cudaGetErrorString(x), __FILE__, __LINE__);\  
    return EXIT_FAILURE; } } while(0)  
int main() {  
    ...  
    CUDAERR(cudaMemcpy(...));  
}
```

CUDA CONTROL ERRORES

- Los kernel son *void* (no devuelven nada) y asíncronos con la CPU. Sus errores se gestionan con *cudaGetLastError*.

Ejemplo

```
kernel<<<...>>>(...);  
if (cudaSuccess != cudaGetLastError()) ...
```

O también

```
#define CUDACHECK() __getLastCudaError(__FILE__, __LINE__)  
inline void __getLastCudaError(const char *file, const int line) {  
    cudaError_t err = cudaGetLastError();  
    if (cudaSuccess != err) {  
        fprintf(stderr, "%s(%i): getLastCudaError() error: (%d) %s.\n", file, line, (int)err, cudaGetString(err));  
        exit(-1);  
    }  
}
```

CUDA EVENTOS

- Muchos usos, uno es medir tiempos.

Aquí ¿ Cuántos segundos tarda el kernel ?

```
Ctimer(&elapsed, &ucpu, &scpu, 0);  
    kernel<<<...>>>(...)  
Ctimer(&elapsed, &ucpu, &scpu, 1);
```

En todo caso

```
Ctimer(&elapsed, &ucpu, &scpu, 0);  
    kernel<<<...>>>(...);  
    cudaDeviceSynchronize()  
Ctimer(&elapsed, &ucpu, &scpu, 0);
```

O si procede y tiene sentido

```
Ctimer(&elapsed, &ucpu, &scpu, 0);  
    kernel<<<...>>>(...);  
    cudaMemcpy(...)  
Ctimer(&elapsed, &ucpu, &scpu, 0);
```

CUDA EVENTOS

Pero tal vez sea más apropiado

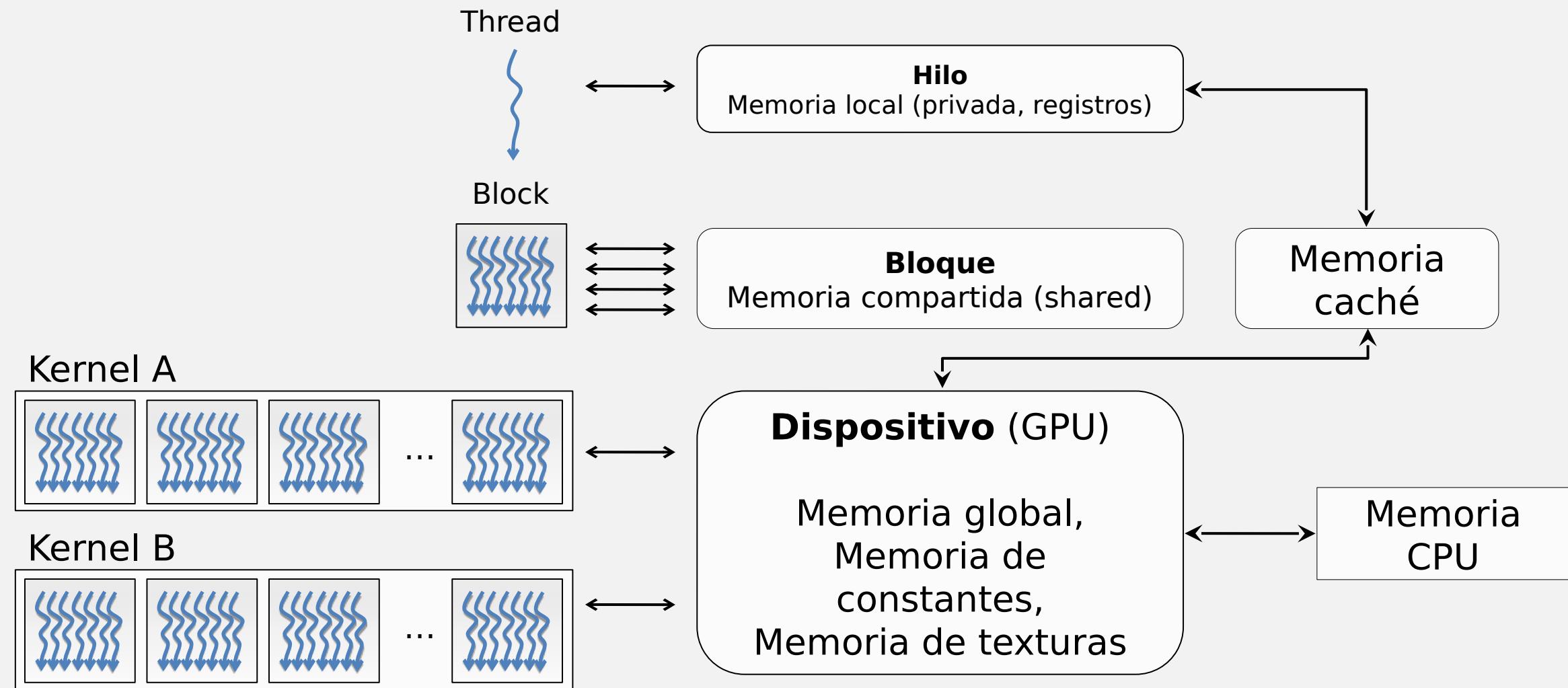
```
int main() {
    cudaEvent_t INICIO, FIN;
    float time;
    cudaEventCreate(&INICIO); cudaEventCreate(&FIN);
    ...
    // Record event INICIO starts on stream 0
    cudaEventRecord(INICIO, 0);
    kernel<<<...>>>(...);
    // Record event FIN starts on stream 0
    cudaEventRecord(FIN, 0);

    // Wait until all device work before its call on stream 0 ends.
    cudaEventSynchronize(FIN);
    cudaEventElapsedTime(&time, INICIO, FIN);
    printf("%f milisegundos\n", time);
    cudaEventDestroy(INICIO); cudaEventDestroy(FIN);
}
```

GPU / CUDA JERARQUÍA DE MEMORIA

- Los hilos CUDA pueden acceder a los datos desde múltiples espacios de memoria durante su ejecución.
- Cada hilo tiene su memoria local privada. Todos los hilos tienen acceso a la memoria global.
- Cada bloque de hilos tiene su espacio de memoria compartida (shared) accesible/visible a todos sus hilos y con el mismo ciclo de vida que el bloque.
- Las GPUs tienen cachés L2 y L1, que se comportan de forma similar a las cachés de la CPU. La L1 comparte espacio (recursos físicos) con la shared.
- Las memorias de texturas y constantes son espacios de memoria adicionales accesibles a todos los hilos.
- Los espacios de memoria global, texturas y constantes tiene el mismo ciclo de vida que la aplicación (persistente durante su ejecución).

GPU / CUDA JERARQUÍA DE MEMORIA



CUDA API RESERVANDO MEMORIA

- Para reservar:

cudaMalloc(), **cudaMallocPitch()**, **cudaMalloc3D()**, **cudaMallocHost()**, etc.

Permiten reservar memoria lineal (CUDA arrays, *layouts* de memoria opacas optimizadas).

- Para liberar:

cudaFree(), **cudaFreeHost()**, etc.

- Para inicializar:

cudaMemset(), etc.

- Para copiar:

cudaMemcpy(), **cudaMemcpyAsync()**, etc. Hay que indicar el tipo: cudaMemcpyDeviceToDevice, cudaMemcpyDeviceToHost, cudaMemcpyHostToDevice, cudaMemcpyHostToHost, cudaMemcpyDefault. En cudaMemcpyDefault el tipo de transferencia se infiere del valor del puntero y se permite en sistemas con soporte UVA (Unified Virtual Addressing).

- Para prebúsquedas:

cudaMemPrefetchAsync(), etc.

CUDA MODELO CLÁSICO (HANDMADE) RESERVA MEMORIA

Ejemplo

```
int main() {
    double *Host_Mat, *Device_Mat;
    int size=m*n*sizeof(double);

    Host_Mat=(double *)calloc(m*n, sizeof(double));
    cudaMalloc((void **) &Device_Mat, size);

    cudaMemcpy(Device_Mat, Host_Mat, size, cudaMemcpyHostToDevice);

    // hacer cosas en CPU con Host_Mat y en GPU con Device_Mat

    cudaMemcpy(Host_Mat, Device_Mat, size, cudaMemcpyDeviceToHost);

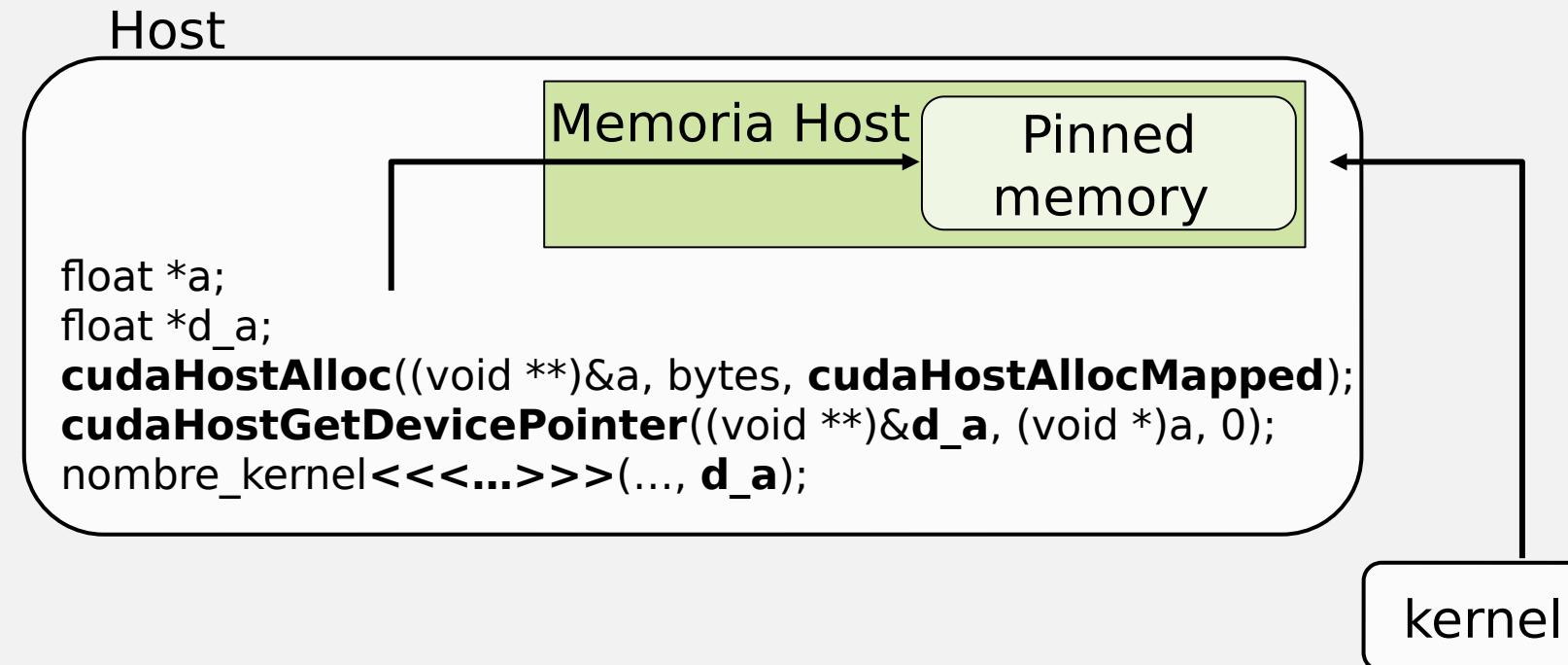
    // hacer más cosas

    cudaFree(Device_Mat);
    free(Host_Mat);
}
```

CUDA PINNED MEMORY

cudaHostAlloc(void ptr, size_t size, unsigned int flag)**

- Reserva (como **cudaMallocHost**) memoria en la RAM del Host (CPU), potencialmente accesible a la GPU.
- La reserva es de tipo *page-locked (pinned memory)*, que acelera las transferencias de información.
- Su uso excesivo puede degradar el rendimiento.



CUDA PINNED MEMORY

cudaHostAlloc(void ptr, size_t size, unsigned int flag)**

- Valores de **flag**
 - *cudaHostAllocDefault*. Equivalente a `cudaMallocHost()`.
 - *cudaHostAllocPortable*. *Pinned para todos los contextos CUDA.*
 - *cudaHostAllocMapped*. *Mapea el espacio de direcciones de la CPU en la GPU. El puntero se obtiene con `cudaHostGetDevicePointer()`. Usado en el ejemplo anterior.*
 - *cudaHostAllocWriteCombined*. Es *write-combined (WC)*. Según Nvidia: “*WC memory can be transferred across the PCI Express bus more quickly on some system configurations, but cannot be read efficiently by most CPUs. WC memory is a good option for buffers that will be written by the CPU and read by the device via mapped pinned memory or host→device transfers*”.

CUDA PINNED MEMORY

Ejemplo Clásica

```
int main() {
    double *Host_Mat, *Device_Mat;
    int size=m*n*sizeof(double);

    cudaHostAlloc((void **) &Host_Mat, size, cudaHostAllocDefault);
    cudaMalloc ((void **) &Device_Mat, size);

    cudaMemcpy(Device_Mat, Host_Mat, size,
    cudaMemcpyHostToDevice);

    // hacer cosas en CPU con Host_Mat y en GPU con Device_Mat
    cudaMemcpy(Host_Mat, size, Device_Mat,
    cudaMemcpyDeviceToHost);
    // hacer más cosas

    cudaFree(Device_Mat);
    cudaFreeHost(Host_Mat);
}
```

CUDA PINNED MEMORY

Ejemplo

```
int main() {
    double *Host_Mat, *Device_Mat;
    int size=m*n*sizeof(double);

    cudaHostAlloc((void **) &Host_Mat, size, cudaHostAllocMapped);

    cudaHostGetDevicePointer((void **)&Device_Mat, (void *)Host_Mat, 0);

    // hacer cosas en CPU con Host_Mat y en GPU con Device_Mat

    cudaFreeHost(Host_Mat);
}
```

Hay ciertas limitaciones (quién, cuándo y cómo se accede a los datos) en este caso (*cudaHostAllocMapped*).

CUDA UNIFIED MEMORY

- *Pinned* tiene limitaciones y problemas de estabilidad si no se usa correctamente.
- Memoria Unificada (*Unified Memory*) hace *lo mismo* de forma más eficiente y con un mayor rango de uso.
- La reserva se hace en “alguna memoria” (¿CPU? ¿GPU? Depende...). El *mismo* puntero es accesible por la CPU y la GPU.
- No todos los sistemas la soportan y en arquitecturas previas a *Pascal* su comportamiento no es bueno.

cudaMallocManaged(void devPtr, size_t size, unsigned int flag)**

Flag especifica el *stream* asociado, que puede ser:

- *cudaMemAttachGlobal*: Accesible por cualquier *stream* en cualquier dispositivo.
- *cudaMemAttachHost*: No accesible para dispositivos con atributo *cudaDevAttrConcurrentManagedAccess* con valor 0.
- Se libera memoria de forma normal: *cudaFree*.

CUDA UNIFIED MEMORY

Ejemplo

```
_global_ void add(int n, float *x, float *y) {
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;
    for (int i = index; i < n; i += stride) y[i] = x[i] + y[i];
}
int main() {
    int N = 1<<20, blockSize = 256, numBlocks = (N + blockSize - 1) / blockSize;
    float *x, *y;
    cudaMallocManaged(&x, N*sizeof(float));
    cudaMallocManaged(&y, N*sizeof(float));
    for (int i = 0; i < N; i++) { x[i] = 1.0f; y[i] = 2.0f; }
    add<<<numBlocks, blockSize>>>(N, x, y);
    cudaDeviceSynchronize();
    cudaFree(x);  cudaFree(y);
    return 0;
}
```

CUDA MEMORIA DE CONSTANTES

En CPU

```
__constant__ double GPUCostes[5];  
...  
int main() {  
    double Costes[5] = {1.0, 4.0, 6.0, 4.0, 1.0};  
    cudaMemcpyToSymbol(GPUCostes, Costes,  
    sizeof(Costes));  
    ...  
    kernel_ejemplo<<<...,...>>(devMat);  
    ...  
}
```

En GPU

```
__global__ void kernel_ejemplo(double *Mat) {  
    ...  
    Mat[i] = Mat[i] * GPUCostes[0] + ...;
```

CUDA SHARED MEMORY

- Se reserva usando el calificador `_shared_`
- Es más rápida (se espera que sea) que la memoria *global*.
- Se divide en módulos de igual tamaño (*banks*) a los que se puede acceder simultáneamente.
- El número de bancos es 32 (¿casualidad?) y ancho de banda de 32 bits por 2 ciclos de reloj (puede cambiar).
- Los accesos a n direcciones de n bancos distintos se pueden hacer simultáneamente: ancho de banda n veces mayor que el ancho de banda de un solo módulo.
- 2 accesos simultáneos al mismo banco generan *conflicto de acceso* y se produce *serialización* en el acceso.
- Los hilos de un *warp* no generan conflicto de banco si acceden dentro de la misma palabra de 32 bits.

CUDA SHARED MEMORY

En tiempo de compilación

```
_global_ void kernel(...) {  
    _shared_ float sData[256];  
    ...  
}  
void main() {  
    kernel<<<nB, nT>>>(...);  
}
```

En tiempo de ejecución

```
_global_ void kernel(...) {  
    extern _shared_ float sData[ ];  
    ...  
}  
void main() {  
    kernel<<<nB, nT, nT*sizeof(float)>>>(...);  
}
```

CUDA SHARED MEMORY

Cache

```
__global__ void EjemploCache(float* A, float *F, ...) {  
    int i = blockDim.x * blockIdx.x + threadIdx.x;  
    int j = blockDim.y * blockIdx.y;  
    ...  
    temp += A[t]*F[0];  
    ...  
}
```

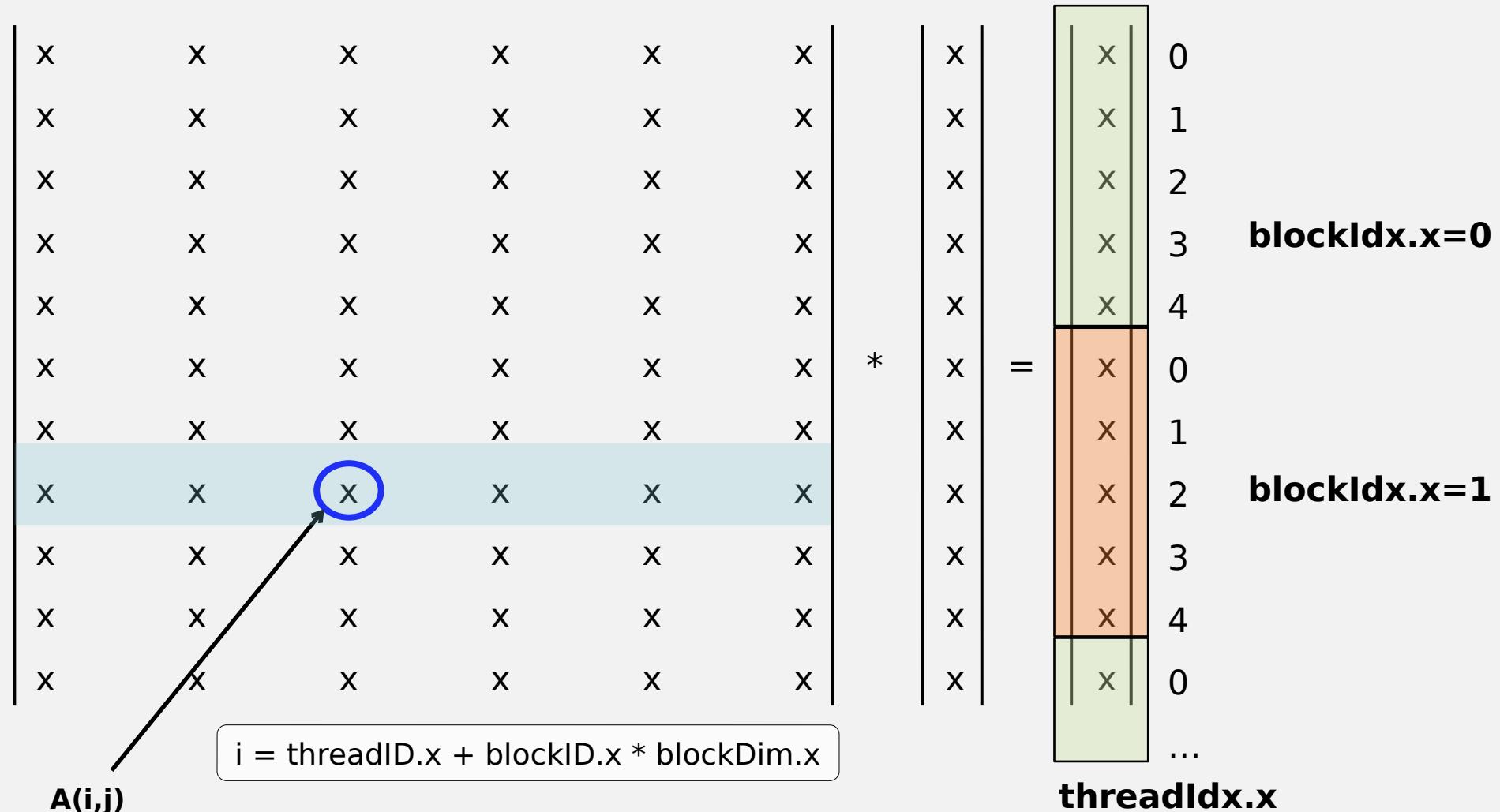
Shared

```
__global__ void EjemploShared(float* A, float *F, ...) {  
    __shared__ float copia[(256+2)*3];  
    int i = blockDim.x * blockIdx.x + threadIdx.x;  
    int j = blockDim.y * blockIdx.y;  
    ...  
    // write data to shared memory  
    copia[threadIdx.x + 2] = A[t];  
    t += N;  
    copia[dim + threadIdx.x + 2] = A[t];  
    ...  
    __syncthreads();  
    ...  
    temp += copia[t]*F[0];  
    ...  
}
```

CUDA SHARED MEMORY

Recordando

$$\mathbf{v} = \mathbf{A} * \mathbf{x}$$



CUDA SHARED MEMORY

Ejemplo $v = A * x$

```
__global__ void MatdotVec(double *A, double *x, double *v, int rows, int cols) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j;
    double tmp=0.0;

    if (i < rows)
        for (j=0; j < cols; j++)
            tmp += A[i*cols+j] * x[j];
    v[i] = tmp;
}

int main() {
    ...
    dim3 TPBLCK(XXX, 1, 1);  dim3 NBLCK(ZZZ, 1, 1);
    MatdotVec<<<NBLCK, TPBLCK>>>(A, x, v, rows, cols);
    ...
}
```

CUDA SHARED MEMORY

$$v = A * x$$

Shared memory



CUDA SHARED MEMORY

Ejemplo $v = A * x$ con Shared

```
_global_ void MatdotVecSh(double *A, double *x, double *v, int rows, int cols) {  
    int j, i=blockIdx.x * blockDim.x + threadIdx.x;  
    extern __shared__ double sh_x[];  
    double tmp=0.0;  
    if (i < rows){  
        if (threadIdx.x == 0)  
            for (j=0; j < cols; j++) { sh_x[j] = x[j]; }  
        _syncthreads();  
        for (j=0; j < cols; j++) { tmp += A[i*cols+j] * sh_x[j]; }  
        v[i] = tmp;  
    }  
}  
int main() {  
    dim3 TpBlock (XXX, 1, 1);  
    dim3 Nblocks (ZZZ, 1, 1);  
    MatdotVec<<<Nblocks, TpBlock, n*sizeof(double)>>>(A, x, v, rows, cols);  
}
```

CUDA SHARED MEMORY

```
$ matdotvec
```

Uso: matdotvec <rows> <cols> <seed> <threadsPerBlock>

```
$ matdotvec 10000 4000 13 32
```

INFO: Hay 1 GPUs con capacidades CUDA, seguimos

El tiempo en CPU es 4.7871487E-02 segundos.

El tiempo en GPU es 1.2955520E-02 segundos.

El tiempo con SH es 3.0112000E-05 segundos.

“los tiempos de GPU incluyen la copia al Host del resultado”

```
$ matdotvec 4000 10000 13 32 (si compila...)
```

INFO: Hay 1 GPUs con capacidades CUDA, seguimos

El tiempo en CPU es 4.8071648E-02 segundos.

El tiempo en GPU es 1.0892768E-02 segundos.

El tiempo con SH es 2.7676960E-02 segundos.

¿ La memoria compartida es infinita ?

CUDA SHARED MEMORY

Ejemplo $v = A * x$ con Shared V2: “n debe ser divisible por 4”

```
__global__ void MatdotVecSH2(double *A, double *x, double *v, int rows, int cols) {
    int k, j, i=blockIdx.x * blockDim.x + threadIdx.x;
    __shared__ double sh_x[4];           // o cualquier otro valor razonable
    double tmp=0.0;

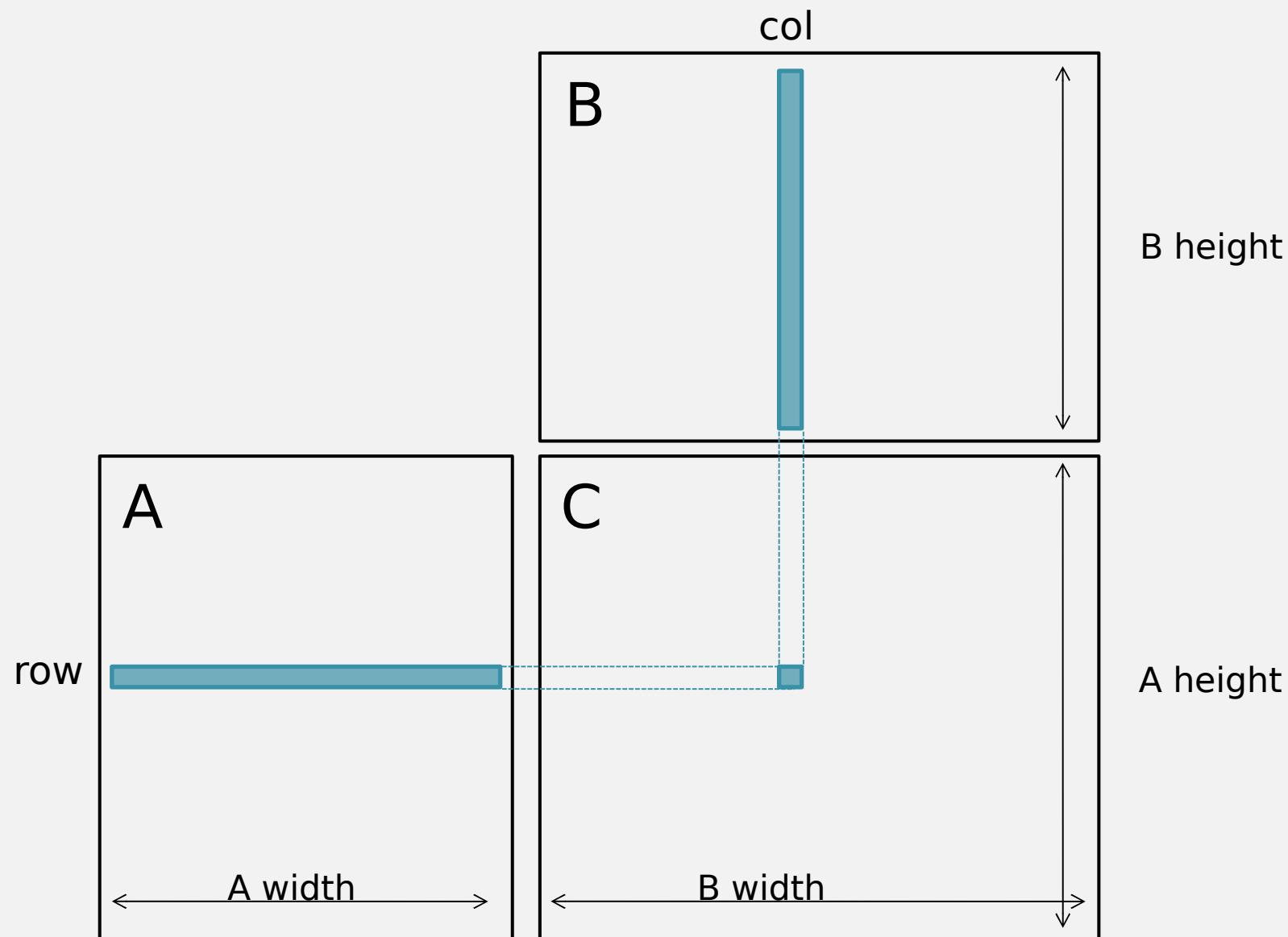
    if (i < rows)
        for (k=0; k < cols; k+=4)
    {
        if (threadIdx.x == 0)
            for (j=0; j<4; j++) { sh_x[j] = x[k+j]; }
        _syncthreads();

        for (j=0; j<4; j++) { tmp += A[i*cols+k+j] * sh_x[j]; }
        _syncthreads();           // evita que si el 0 avanza antes cambie sh_x antes de ...
    }
    v[i] = tmp;
```

Puede ser más lento o rápido dependiendo de *rows*, hilos por bloque, etc.

CUDA SHARED MEMORY

$$C = A * B$$



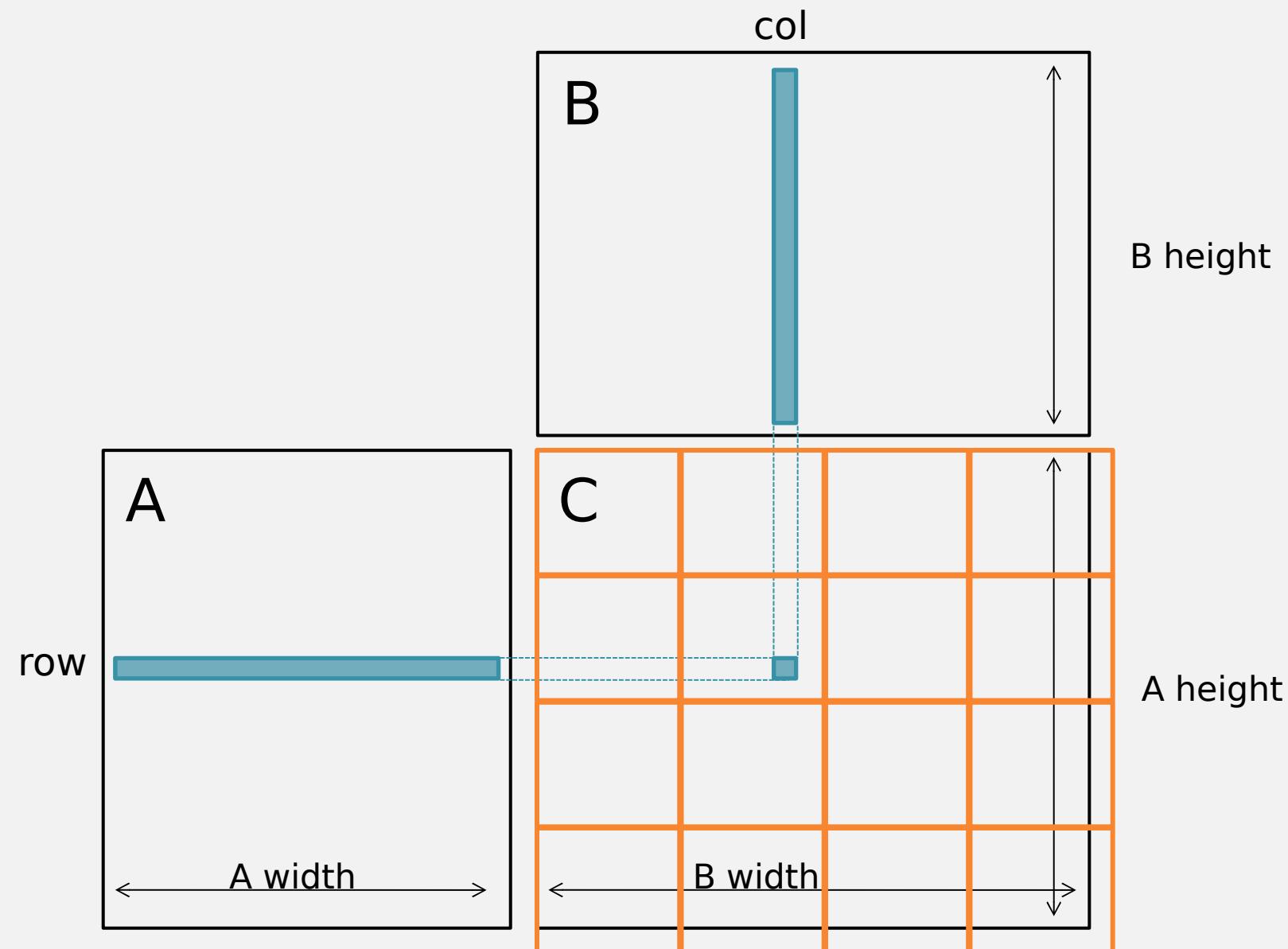
CUDA SHARED MEMORY

C = A * B

```
_global_ void kernel_MatMul(int m, int n, int k, double *C, int IdC, double *A, int IdA, double *B, int IdB) {  
    int i = blockIdx.y * blockDim.y + threadIdx.y; //rows  
    int j = blockIdx.x * blockDim.x + threadIdx.x; //cols  
    int r;  
    double dtmp=.0;  
  
    if (i<m && j<n)  
    {  
        for (r = 0; r < k; r++)  
            dtmp += A[i * IdA + r] * B[r * IdB + j];  
        C[i * IdC + j] = dtmp;  
    }  
}  
...  
dim3 NBlk((int)ceil((float)n/threadsPerBlock), (int)ceil((float)m/threadsPerBlock), 1);  
dim3 TpBlk(threadsPerBlock, threadsPerBlock, 1);  
kernel_MatMult<<<NBlk, TpBlk>>>(m, n, k, Devi_MatC, n, Devi_MatA, n, Devi_MatB, k);
```

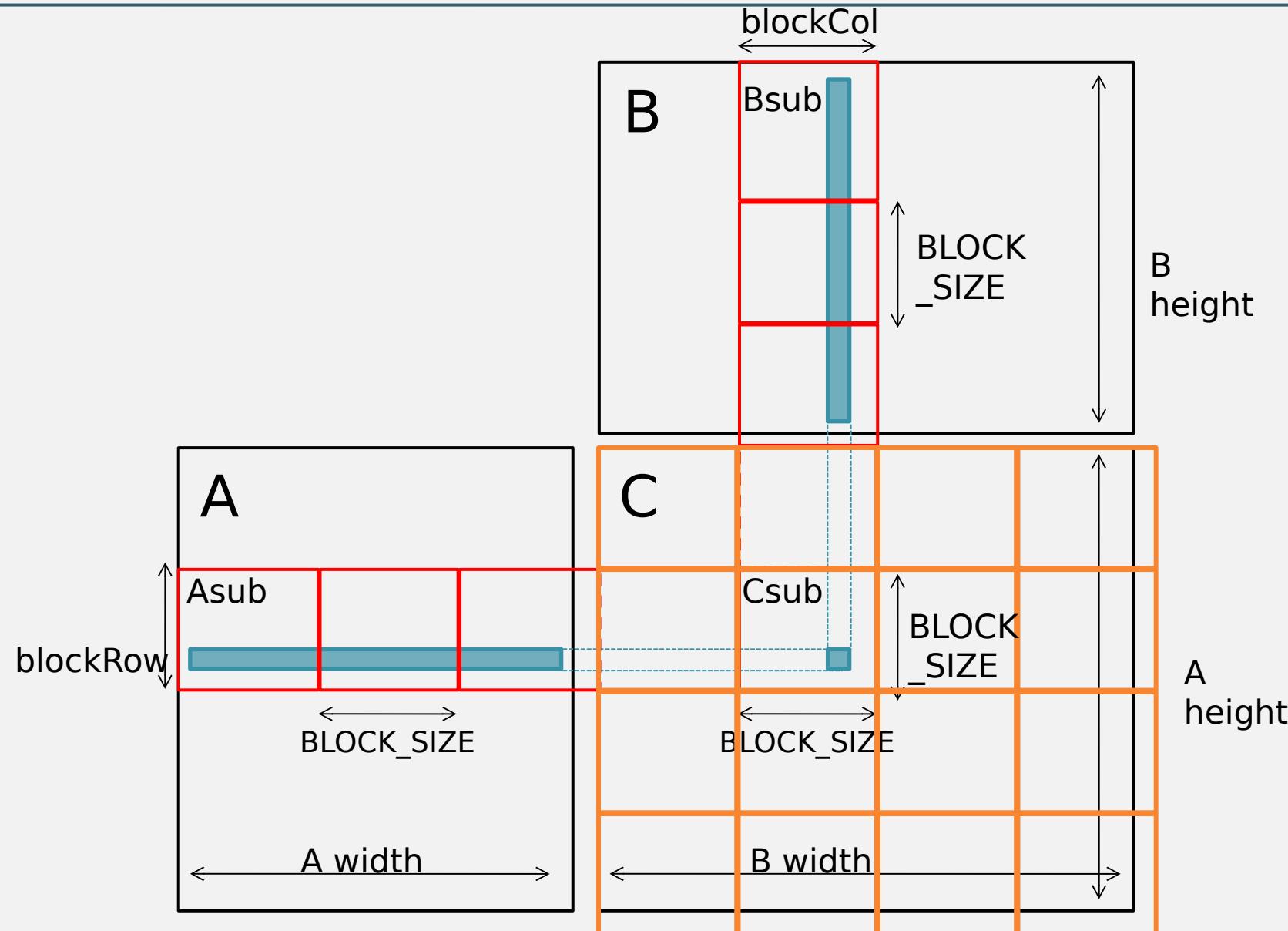
CUDA SHARED MEMORY

$$C = A * B$$



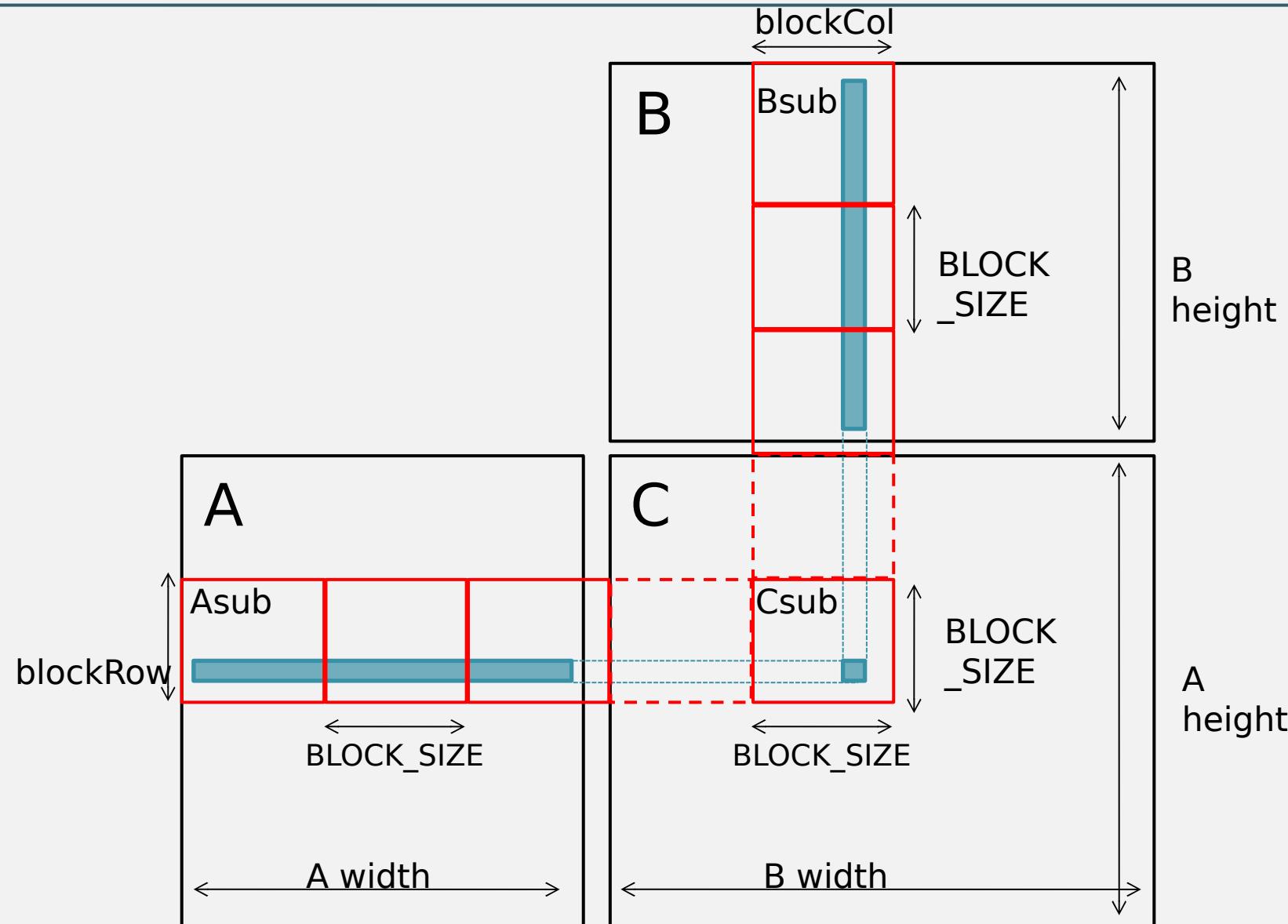
CUDA SHARED MEMORY

$$C = A * B$$



CUDA SHARED MEMORY

$$C = A * B$$



CUDA SHARED MEMORY

C = A * B

```
$ matmul_cuda  
Uso: matmul_cuda <m> <n> <k> <seed> <threadsPerBlock>
```

```
$ matmul_cuda 4000 4000 4000 11 16  
INFO: Hay 1 GPUs con capacidades CUDA, seguimos  
El tiempo en CPU con MKL es 3.2251323E+00 segundos.  
El tiempo en GPU es 2.1978037E+00 segundos.  
El tiempo en GPU Shared es 1.0863396E+00 segundos.
```

```
$ matmul_cuda 4000 4000 4000 11 32  
INFO: Hay 1 GPUs con capacidades CUDA, seguimos  
El tiempo en CPU con MKL es 3.8699146E+00 segundos.  
El tiempo en GPU es 2.9133772E+00 segundos.  
El tiempo en GPU Shared es 1.0914811E+00 segundos.
```

CUDA NIVELES DE CONCURRENCIA

- Las siguientes operaciones se pueden ejecutar simultáneamente entre sí:
 - Computación en el Host (CPU).
 - Ejecución de kernels en el *device* (GPU).
 - Transferencia de información del Host a la GPU.
 - Transferencia de información de la GPU al Host.
 - En la copia de datos entre direcciones de memoria del mismo subsistema.
 - Transferencia de información entre memorias de varias GPUs.
- El nivel de concurrencia alcanzado depende de las características y de la CC del dispositivo.
- En las transferencias de datos con *pinned* y la ejecución de los *kernels*.
- En la copia de datos entre la CPU y las GPUs cuando son bloques de tamaño ≤ 64 KB.
- En los movimientos de datos con funciones *Async*.
- ...