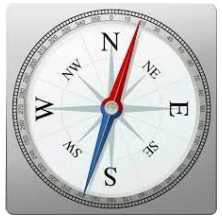
Seminario de CUDA C

Nicolás Calvo Cruz

UNIVERSIDAD DE GRANADA

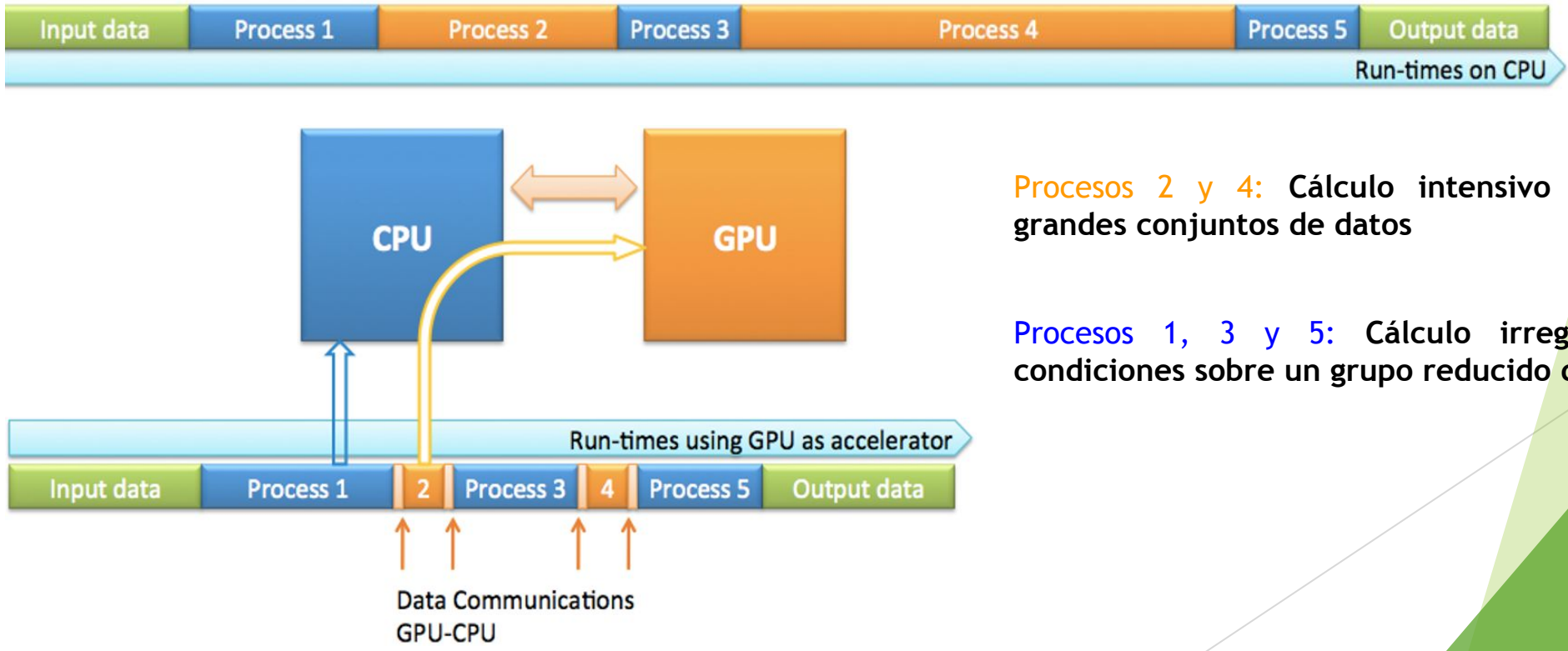
Contenidos



1. CPU y GPU
2. Contexto de programación
3. Conceptos de CUDA C
4. Primeros kernels
5. Configurando un kernel en detalle
6. Identificación de hilos
7. Jerarquía de memoria
8. Sincronización y atomicidad
9. Ejemplo de reducción
10. Ejemplo de atomicidad
11. Funciones de gestión del dispositivo
12. Gestión de errores
13. Medición de tiempos
14. Compilador
15. Herramientas disponibles
16. Enlaces de interés

CPU y GPU (I)

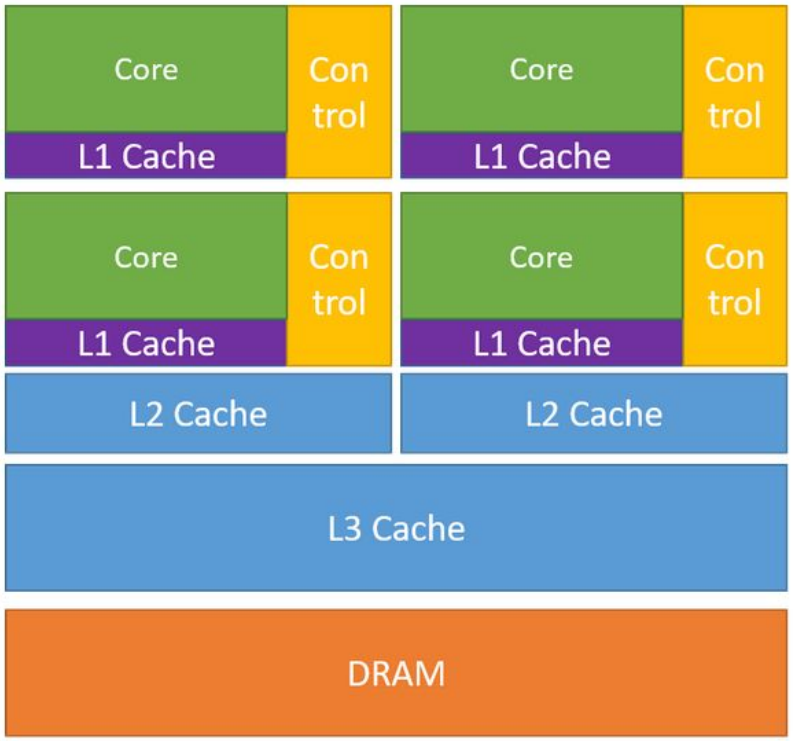
La GPU como acelerador:



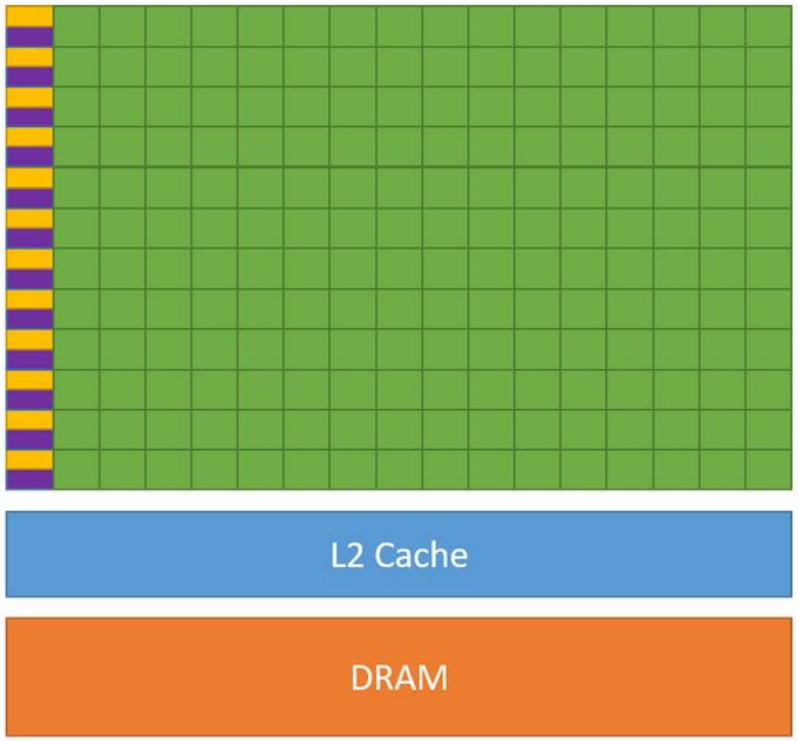
Procesos 2 y 4: **Cálculo intensivo y regular sobre grandes conjuntos de datos**

Procesos 1, 3 y 5: **Cálculo irregular basado en condiciones sobre un grupo reducido de datos**

CPU y GPU (II)



CPU

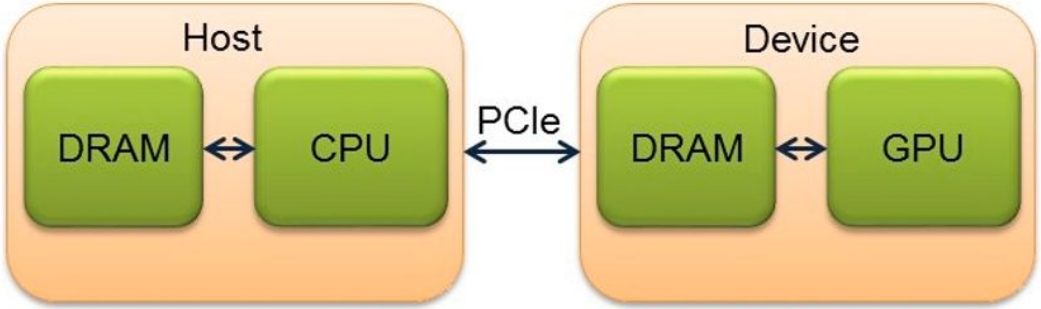


GPU

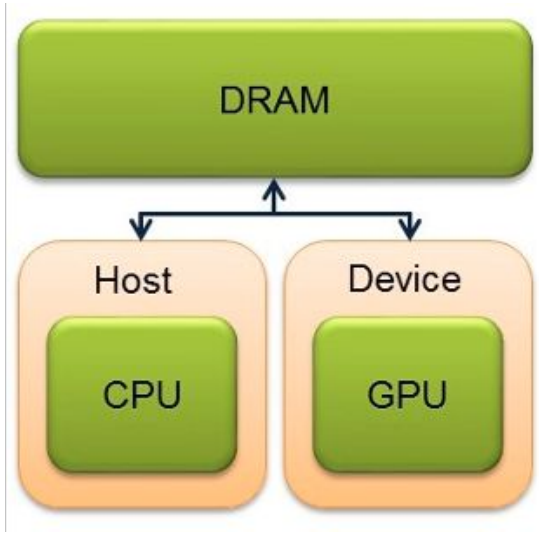
La **GPU** dedica **más recursos al cómputo** pero **sacrifica** capacidad de **control** (predicción de saltos, reordenamiento de instrucciones…)

CPU y GPU (III)

Posibles **conexiones CPU-GPU**:

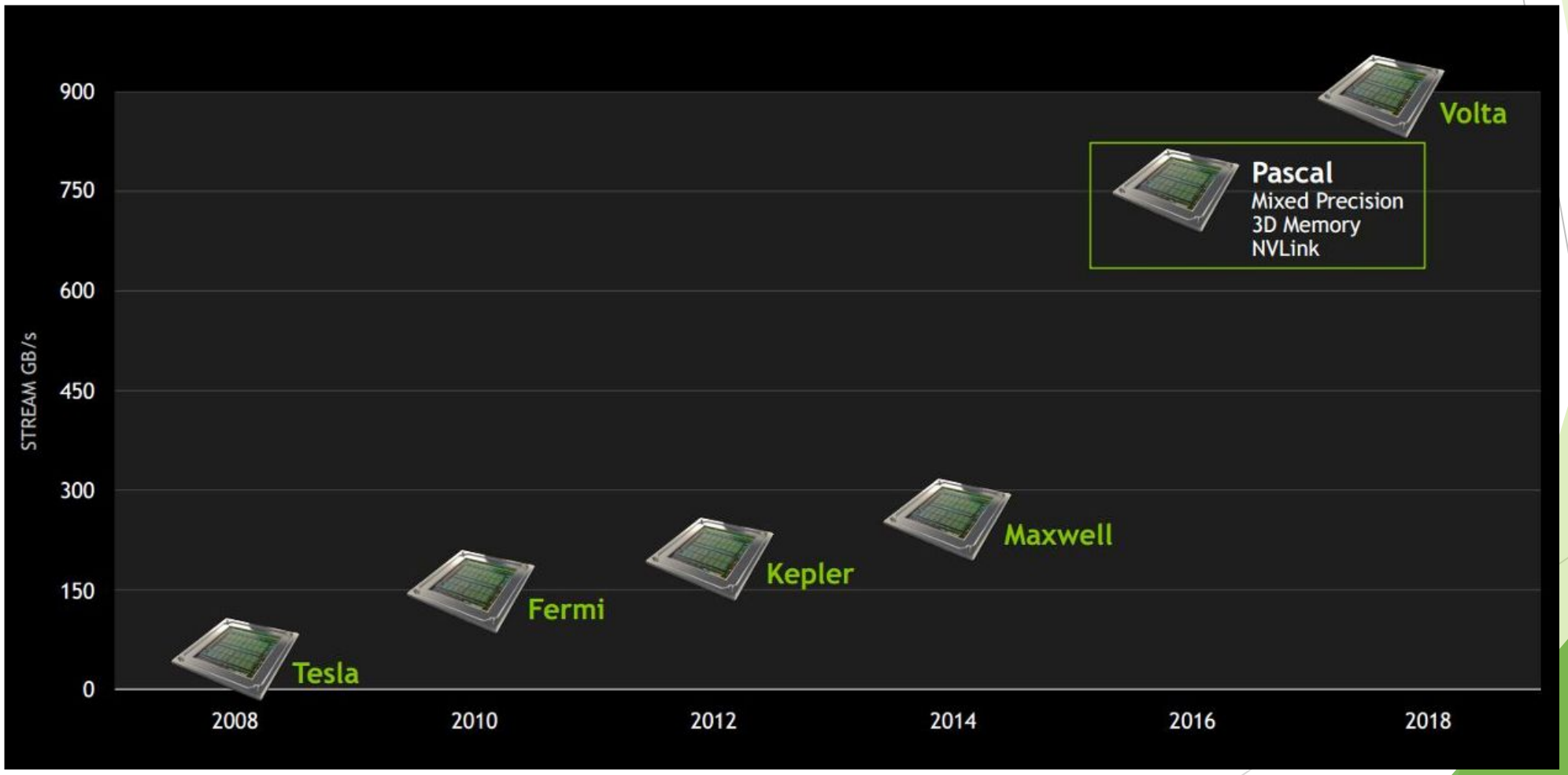


GPU externa



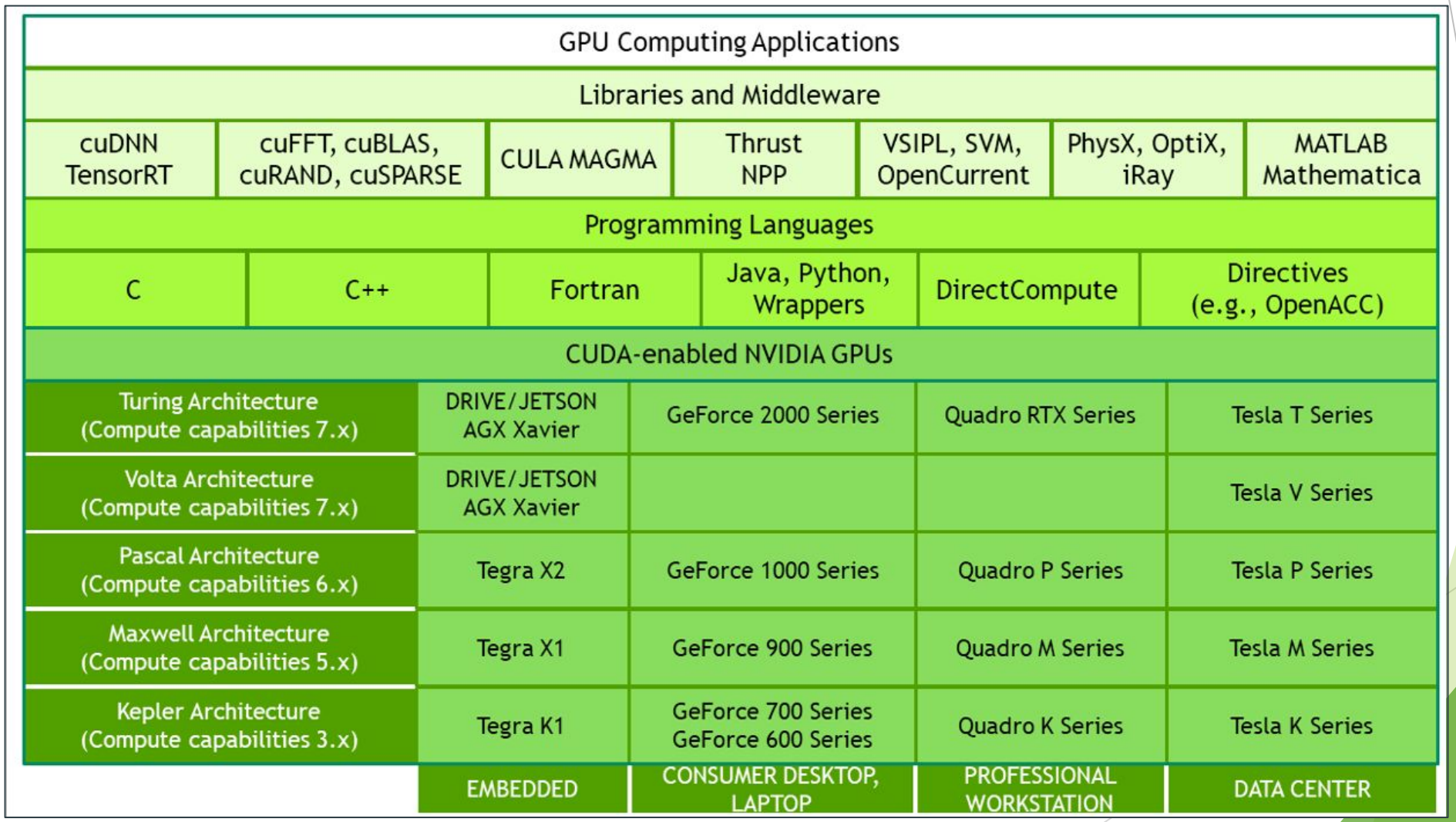
GPU integrada

CPU y GPU (IV)

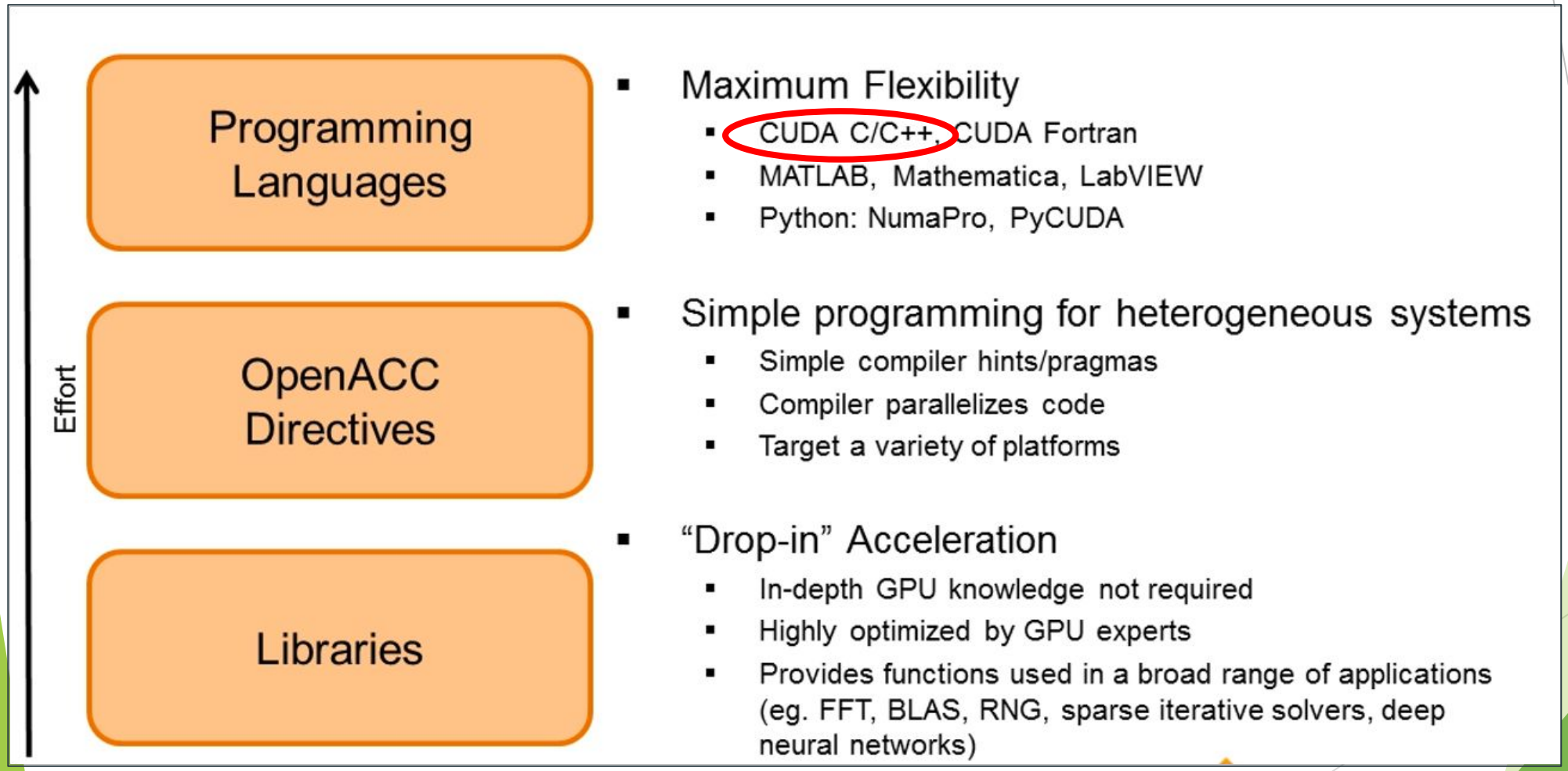

900
750
600
450
300
150
0
STREAM GB/s
2008
2010
2012
2014
2016
2018
Tesla
Fermi
Kepler
Maxwell
Pascal
Mixed Precision
3D Memory
NVLink
Volta

CPU y GPU (V)

CUDA:

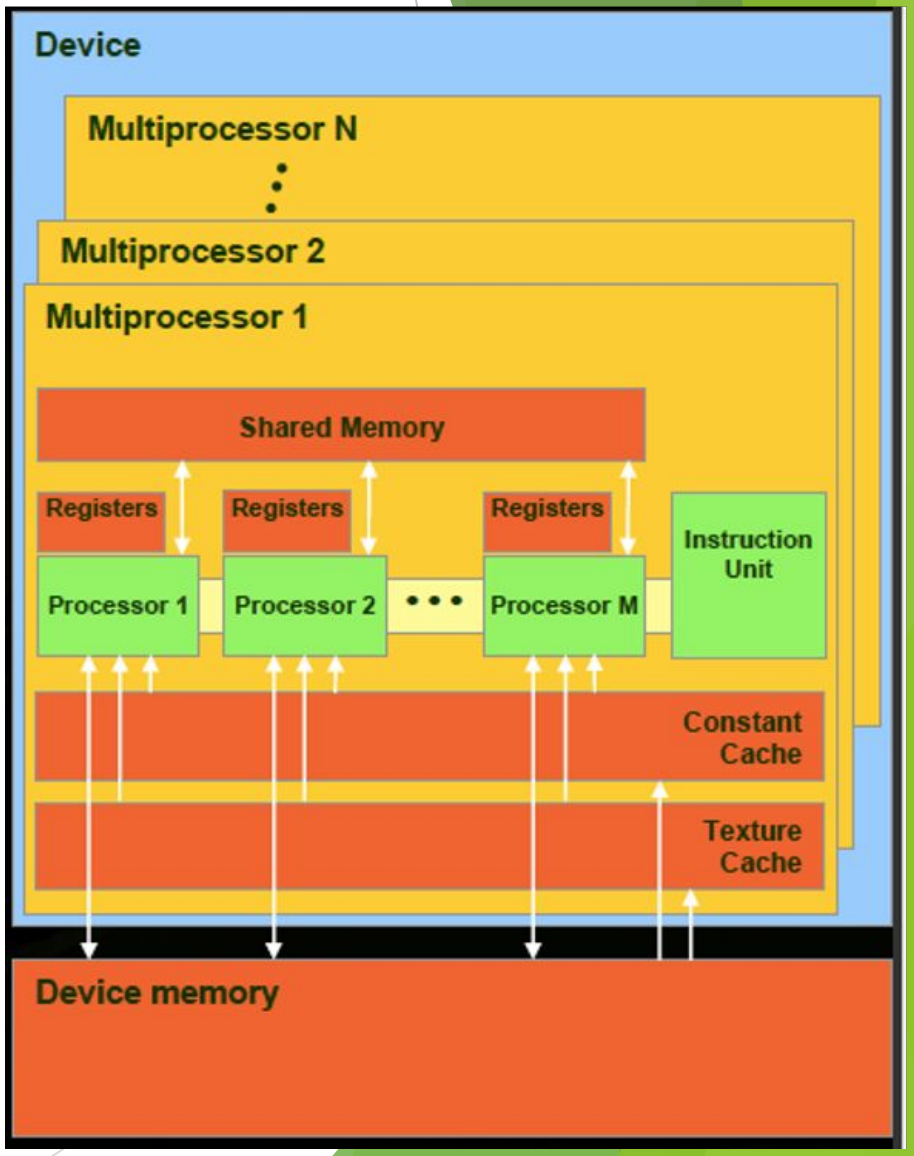


CPU y GPU (VI)


Effort
Programming Languages
OpenACC Directives
Libraries
Maximum Flexibility
CUDA C/C++, CUDA Fortran
MATLAB, Mathematica, LabVIEW
Python: NumaPro, PyCUDA
Simple programming for heterogeneous systems
Simple compiler hints/pragmas
Compiler parallelizes code
Target a variety of platforms
"Drop-in" Acceleration
In-depth GPU knowledge not required
Highly optimized by GPU experts
Provides functions used in a broad range of applications (eg. FFT, BLAS, RNG, sparse iterative solvers, deep neural networks)

Contexto de programación

- La **GPU contiene N multiprocesadores. Cada uno tiene:**
  - **M procesadores**
  - Banco de registros
  - **Memoria compartida: muy rápida, pequeña**
  - Cachés de constantes y de texturas (sólo lectura)
  - **La memoria global o (memoria del dispositivo): grande pero** unas 500 veces más **lenta** que la memoria compartida (según el dispositivo).

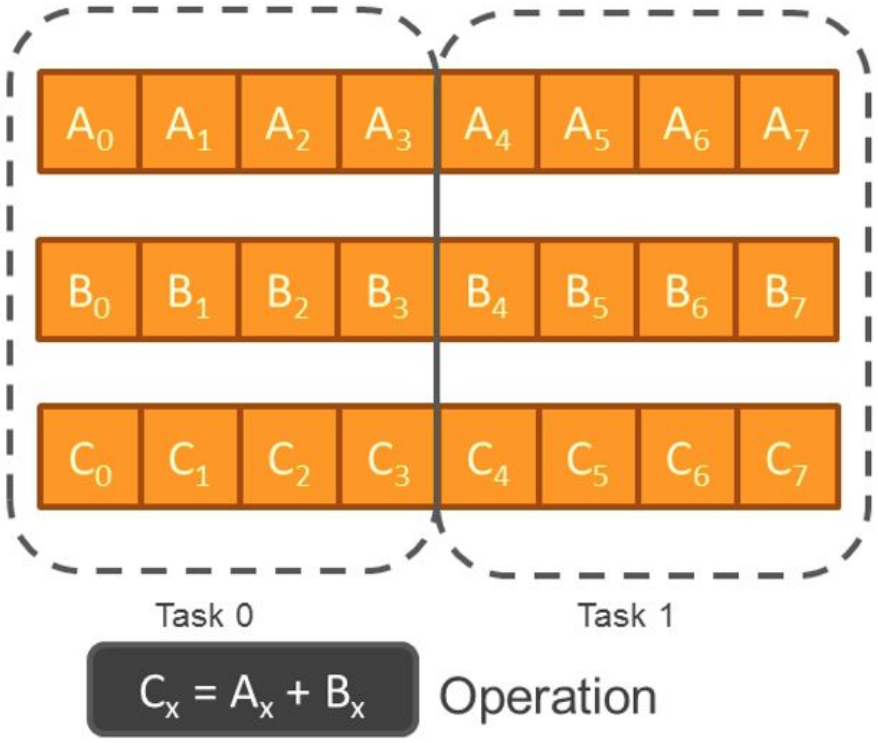


Conceptos de CUDA C (I)

- **CUDA C es una extensión de C++** convencional **que permite programar código que interacciona con GPU's de NVidia compatibles**, pudiendo explotar así el poder de cómputo de la CPU y la GPU.
- La **programación** en CUDA **es de tipo heterogéneo** y está orientada a un modelo de paralelismo de datos, Single Program Multiple Data (**SPMD**).
- Se opera sobre datos de una misma estructura (p.ej., arrays y matrices).
- Un **conjunto de hilos** actúa **en paralelo sobre la misma estructura**, pero **cada hilo procesa sobre su** propia **porción.**
- Todos los hilos hacen **las** mismas **operaciones** sobre sus porciones de estructura, por lo que éstas **han de ser independientes de los datos.**

Conceptos de CUDA C (II)

- Ejemplo de cálculo con paralelismo de datos:



- Conjunto de datos definido por los arrays A, B y C.
- Se hace la misma operación en cada elemento.
- Cada tarea funciona en un subconjunto independiente de los datos.

Conceptos de CUDA C (III)

- **Una función** pensada **para** ejecutarse en la **GPU se denomina** **Kernel**, y se espera que tenga un gran paralelismo de datos.

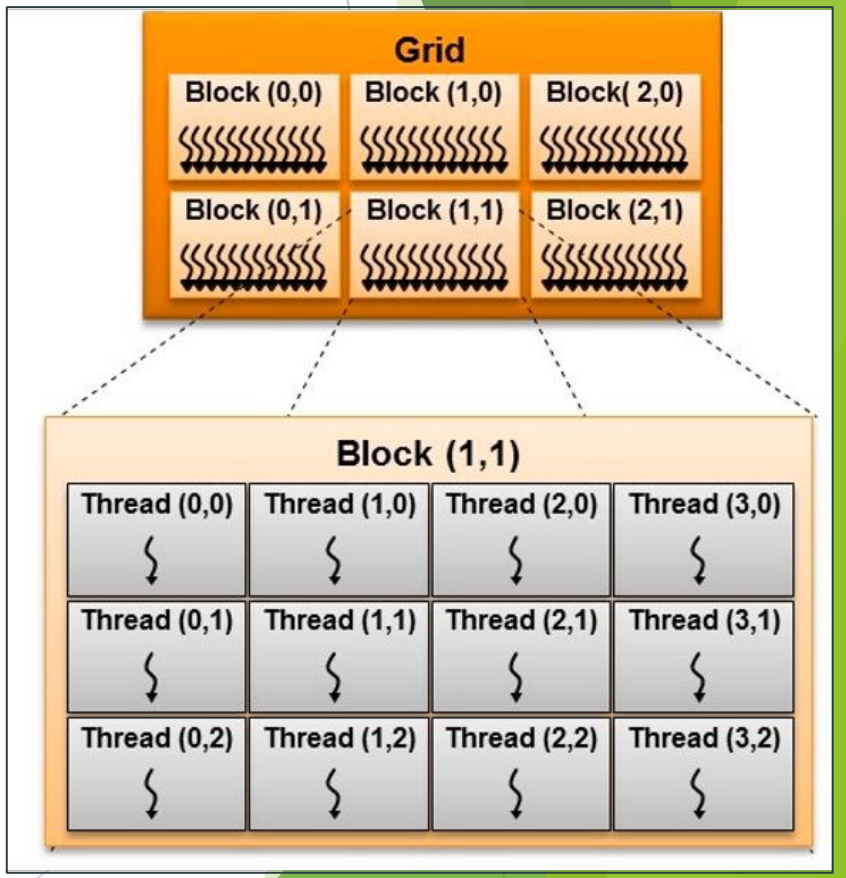
- Un **kernel se ejecutará en paralelo por N hilos diferentes en una GPU**.

- En código, una función kernel lleva delante uno (o varios) de estos **identificadores**:

  - __host__ para código **llamado y ejecutado en la CPU**
  - __device__ para kernels **llamados y ejecutados en la GPU**
  - __global__ para kernels **llamados en la GPU o la CPU y ejecutados en la GPU**

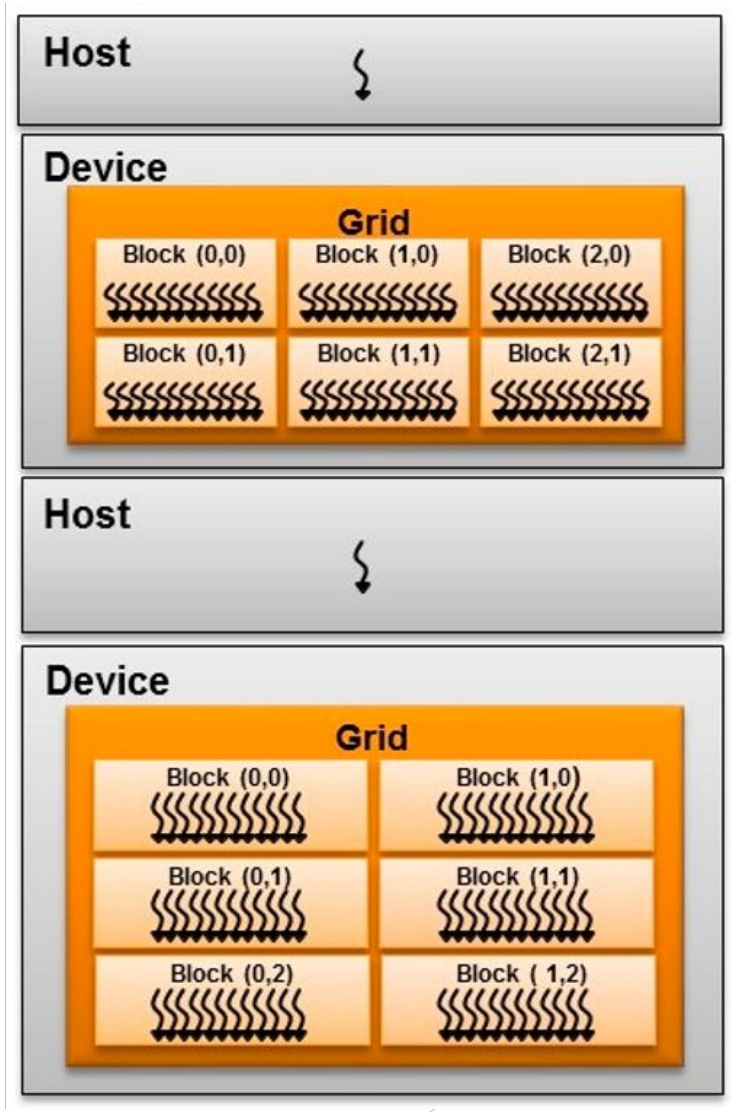
Conceptos de CUDA C (IV)

- **Los** hilos de CUDA o **CUDA Threads** no son como los de CPU, **son** más **ligeros** de activar, desactivar y cambiar de contexto. **La idea es** poder **ejecutar miles de threads.**
- Los **CUDA Threads tienen que ejecutar el mismo kernel**, por lo que se espera que hagan la mismas operaciones sobre parte de una estructura de datos de forma independiente.
- **Los hilos se agrupan en bloques** de **hasta 3 dimensiones**.
- Los **bloques de threads se agrupan en un grid**.
- **Las dimensiones** del grid y el bloque **se definen en la llamada** al kernel con **los caracteres <<< ... >>>**.



Conceptos de CUDA C (V)

- El equipo anfitrión o ***host***, se asocia a la idea de "**CPU**", y se encarga de:
  - **Lanzar** los **kernels**
  - **Gestionar** la **memoria** de la GPU (***o*** *device*) y del propio host
  - Intercambiar datos entre *device* y *host*
  - El host, cuando concluye el control del *device*, puede ejecutar código propio.



Primeros kernels (I)

Este es el kernel más sencillo que podemos hacer:

```
#include <cstdio>

__global__ void kernel(void){

}

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

Atención: Un kernel __global__ no puede devolver resultados (directamente): **ha de ser de tipo void**

No tiene acceso a funciones del host

Sí se espera que reciba **punteros a memoria de GPU** y **parámetros** pasados por valor

```
[ncalvo@atcgrid SeminarioCUDA]$ srun -p ac4 nvcc holaMundo.cu
[ncalvo@atcgrid SeminarioCUDA]$ srun -p ac4 a.out
Hola Mundo!
```

Primeros kernels (II)

Vamos a hacer algo de provecho:

```
#include <cstdio>

__global__ void sumarNumero(int a, int b, int* c){
        *c = a + b;
}

int main(){
        int a = 4, b = 2;
        int c, *dev_C;
        cudaMalloc((void**) &dev_C, sizeof(int));
        sumarNumero<<<1,1>>>(a, b, dev_C);
        cudaMemcpy(&c, dev_C, sizeof(int), cudaMemcpyDeviceToHost);
        cudaFree(dev_C);
        printf("%d + %d = %d\n", a, b, c);
        return 0;
}
```

Los principales tipos que se esperan en un kernel son:
char, short, int, long, long long, float y double
(**y sus punteros**)

```
[ncalvo@atcgrid SeminarioCUDA]$ srun -p ac4 nvcc sumarNumero.cu
[ncalvo@atcgrid SeminarioCUDA]$ srun -p ac4 ./a.out
4 + 2 = 6
```

Primeros kernels (III)

Vamos a hacer algo de provecho:

```
[ncalvo@atcgrid SeminarioCUDA]$ srun -p ac4 nvcc sumarNumeroB.cu
[ncalvo@atcgrid SeminarioCUDA]$ srun -p ac4 a.out
4 + 2 = 6
```

```
#include <cstdio>

__global__ void sumarNumero(int a, int* b, int* c){
        *c = a + (*b);
}

int main(){
        int a = 4, b = 2;
        int c, *dev_C, *dev_B;
        cudaMalloc((void**) &dev_B, sizeof(int));
        cudaMemcpy(dev_B, &b, sizeof(int), cudaMemcpyHostToDevice);
        cudaMalloc((void**) &dev_C, sizeof(int));
        sumarNumero<<<1,1>>>(a, dev_B, dev_C);
        cudaMemcpy(&c, dev_C, sizeof(int), cudaMemcpyDeviceToHost);
        cudaFree(dev_B);
        cudaFree(dev_C);
        printf("%d + %d = %d\n", a, b, c);
        return 0;
}
```

Primeros kernels (IV)

Esto no es un kernel... pero es interesante:

```
#include <cstdio>

int main(void){
        cudaDeviceProp prop;
        int numDevices = 0;
        cudaGetDeviceCount(&numDevices);
        for(int i = 0; i<numDevices; i++){
                cudaGetDeviceProperties( &prop, i );
                printf("[%d] Nombre: %s\n", i, prop.name);
        }
        return 0;
}
```

```
[ncalvo@atcgrid SeminarioCUDA]$ srun -p ac4 nvcc infoGPU.cu
[ncalvo@atcgrid SeminarioCUDA]$ srun -p ac4 a.out
[0] Nombre: Quadro RTX 5000
```

Configurando un kernel en detalle (I)

- Formalmente, la sintaxis para definir cómo se ejecuta un kernel en GPU es:

  myKernel<<<**dim3** tamGrid, **dim3** tamBlock>>>(…)

- Ya hemos dicho que se trabaja en **hasta 3 dimensiones, y dim3 es el tipo de datos que nos proporciona CUDA** para definir esta información. Internamente es una **tupla 3D (x, y, z).**

- Cuando omitimos información, **CUDA asume que el resto de dimensiones son 1** (aunque esto **sólo podemos hacerlo en una dimensión**, como en los ejemplos previos).

```
#include <cstdio>

int main(void){
        dim3 a(1);
        printf("%d, %d, %d\n", a.x, a.y, a.z);
        return 0;
}
```

```
[ncalvo@atcgrid SeminarioCUDA]$
1, 1, 1
```

Configurando un kernel en detalle (I)

- Teniendo en cuenta lo anterior, vamos a mirar en detalle esto:
  myKernel<<<**dim3** tamGrid, **dim3** tamBlock>>>(…)

- El **tamaño del grid**, es decir, el **número de bloques,** será:
  *tamGrid.x * tamGrid.y * tamGrid.z*

- El **tamaño de bloque**, es decir, el **número de hilos por bloque**, será:
  *tamBlock.x * tamBlock.y * tamBlock.z*

- **Importante**: Las **cantidades** máximas vendrán **limitadas por nuestra GPU**.

Identificación de hilos (I)

- En última instancia, como pasa con MPI, necesitamos **ajustar el trabajo según números de identificación**.

- Para hacerlo, **CUDA integra en los kernels estas variables (sólo lectura)**:

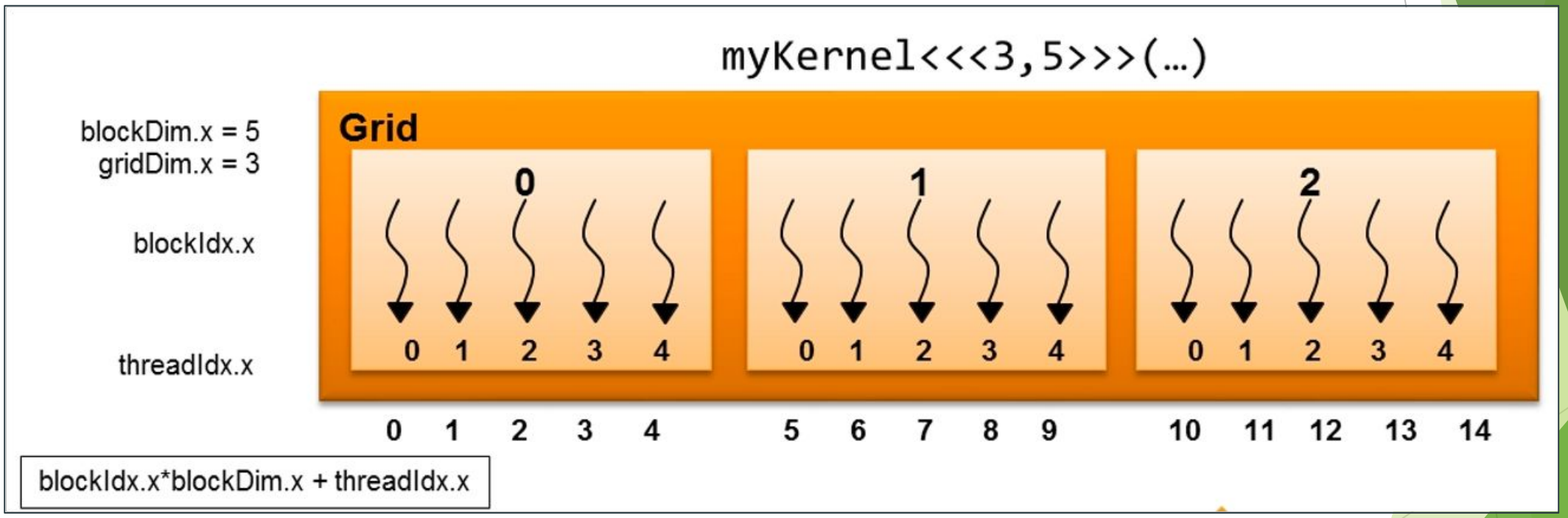
Nombre	Tipo	Significado
gridDim	dim3	Dimensiones del grid
blockIdx	uint3	Identificador de bloque en el grid
blockDim	dim3	Dimensiones del bloque
threadIdx	uint3	Identificador del hilo en el bloque

Identificación de hilos (II)

- **gridDim**:
  - gridDim.x: Número de bloques en la dimensión X del grid
  - gridDim.y: Número de bloques en la dimensión Y del grid
  - gridDim.z: Número de bloques en la dimensión Z del grid
- **blockIdx**:
  - blockIdx.x: Posición del bloque en la dimensión X del grid
  - blockIdx.y: Posición del bloque en la dimensión Y del grid
  - blockIdx.z: Posición del bloque en la dimensión Z del grid
- **blockDim**:
  - blockDim.x: Número de hilos en la dimensión X del bloque
  - blockDim.y: Número de hilos en la dimensión Y del bloque
  - blockDim.z: Número de hilos en la dimensión Z del bloque
- **threadIdx**:
  - threadIdx.x: Posición del hilo en la dimensión X del bloque
  - threadIdx.y: Posición del hilo en la dimensión Y del bloque
  - threadIdx.z: Posición del hilo en la dimensión Z del bloque

Identificación de hilos (III)

Vamos a ver un ejemplo gráfico:



Identificador global del hilo

Los índices de los threads se mapean en los índices del array

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Identificación de hilos (IV)

Otro ejemplo más:

```
MyKernel<<<3,4>>>(a);
```

```
__global__ void MyKernel(int* a)
{
    int idx = blockIdx.x*blockDim.x+threadIdx.x;
    a[idx] = 7;
}

__global__ void MyKernel(int* a)
{
    int idx = blockIdx.x*blockDim.x+threadIdx.x;
    a[idx] = blockIdx.x;
}

__global__ void MyKernel(int* a)
{
    int idx = blockIdx.x*blockDim.x+threadIdx.x;
    a[idx] = threadIdx.x;
}
```

```
a: 7 7 7 7 7 7 7 7 7 7 7 7

a: 0 0 0 0 1 1 1 1 2 2 2 2

a: 0 1 2 3 0 1 2 3 0 1 2 3
```

Identificación de hilos (V)

Formas de sumar dos vectores:

```
#define N ...

__global__ void sumVec(int* a, int* b, int* c){
    int tid = threadIdx.x;
    c[tid] = a[tid] + b[tid];
}

//...

    sumVec<<<1, N>>>(a, b, c);
```

Lanzamos 1 bloque de N hilos

o

```
#define N ...

__global__ void sumVec(int* a, int* b, int* c){
    int tid = blockIdx.x;
    c[tid] = a[tid] + b[tid];
}

//...

    sumVec<<<N, 1>>>(a, b, c);
```

Lanzamos N bloques de 1 hilo

Identificación de hilos (VI)

Realmente las construcciones tienen un interés meramente académico, **lo ideal es lanzar bloques concurrentes…**

```
#define N ...

__global__ void sumVec(int* a, int* b, int* c){
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if(tid < N){
        c[tid] = a[tid] + b[tid];
    }
}

//...
    sumVec<<< (N+127) / 128, 128 >>>(a, b, c);
```

Block 0	Thread 0	Thread 1	Thread 2	Thread 3
Block 1	Thread 0	Thread 1	Thread 2	Thread 3
Block 2	Thread 0	Thread 1	Thread 2	Thread 3
Block 3	Thread 0	Thread 1	Thread 2	Thread 3

La división es entera. Fijando 128 hilos por bloque… así garantizamos que se hace un redondeo al alza, de forma que si N no es múltimo exacto de 128, lanzaremos hilos de más. Por eso tenemos que incluir una condición que evite escribir si no se debe.

$$\frac{Size + BlkDim - 1}{BlkDim}$$

Identificación de hilos (VII)

- Incluso así, podemos quedarnos cortos, pues las dimensiones de los bloques tienen límite en cada componente…
- Hasta no hace mucho era 65 535. En ese caso, si nuestro vector fuera mayor que 65 535 * 128, nos encontraríamos con errores de ejecución. Esta es una solución general:

```
#define N ...

__global__ void sumVec(int* a, int* b, int* c){
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    while(tid < N){
        c[tid] = a[tid] + b[tid];
        tid += blockDim.x * gridDim.x;
    }
}

//...
    sumVec<<<128, 128>>>(a, b, c);
```

Nos aseguramos de no lanzar demasiados bloques ni hilos. Podemos quedarnos con 128 bloques de 128 hilos… Pero esto no deja de ser un valor arbitrario y sujeto a ajuste dentro de los rangos válidos.

Identificación de hilos (VIII)

- Otros ejemplos simplificados:

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}
```

dim3 dimBlock(width, height);

dim3 dimGrid(10);

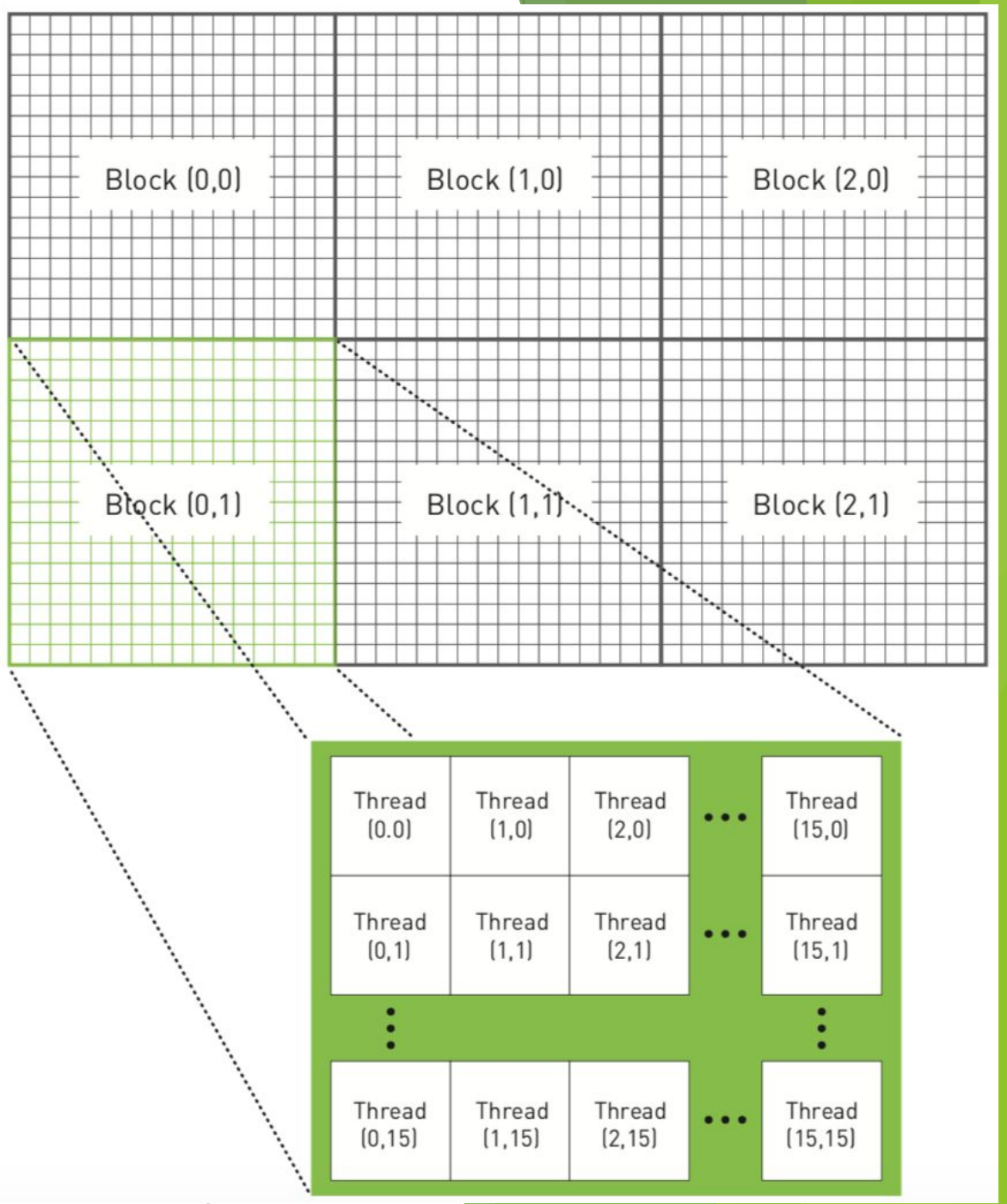
(y lo que no indicamos: 1)

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}
```

Identificación de hilos (IX)

```
dim3    blocks(DIM/16,DIM/16);
dim3    threads(16,16);
```

- Queremos procesar una imagen y que cada hilo se encargue de un píxel
- Imaginemos que es una imagen 48x32



Identificación de hilos (X)

Recuerda:

- **Cada bloque de hilos se ejecuta de forma independiente.**
- **Un bloque de hilos se gestiona mediante warps, que es un conjunto de *warpSize* (32) hilos dentro de un mismo bloque.**
- **Los accesos a memoria de holos de un warp se fusionan para minimizar costes.**

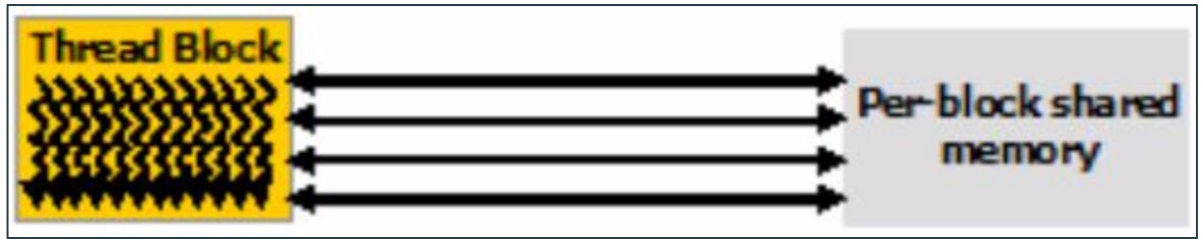
Jerarquía de memoria (I)

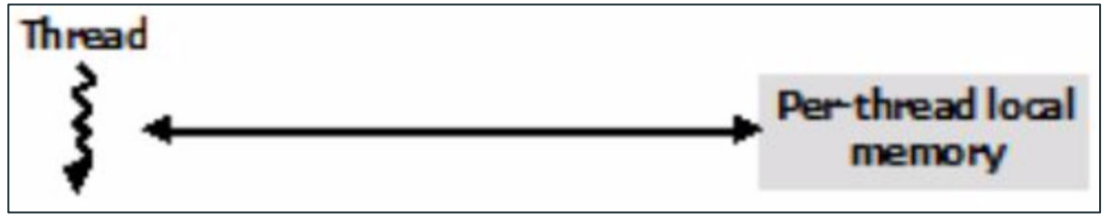
- **Memoria global**: accesible por todos los hilos de todos los bloques activos en la GPU (accesos de 32, 64 o 128 bytes).
- **Memoria shared**: accesible solo por los hilos de un mismo bloque y liberada cuando éste termina su ejecución. Es una **forma** ideal **de hacer cooperar a los hilos de un bloque**.

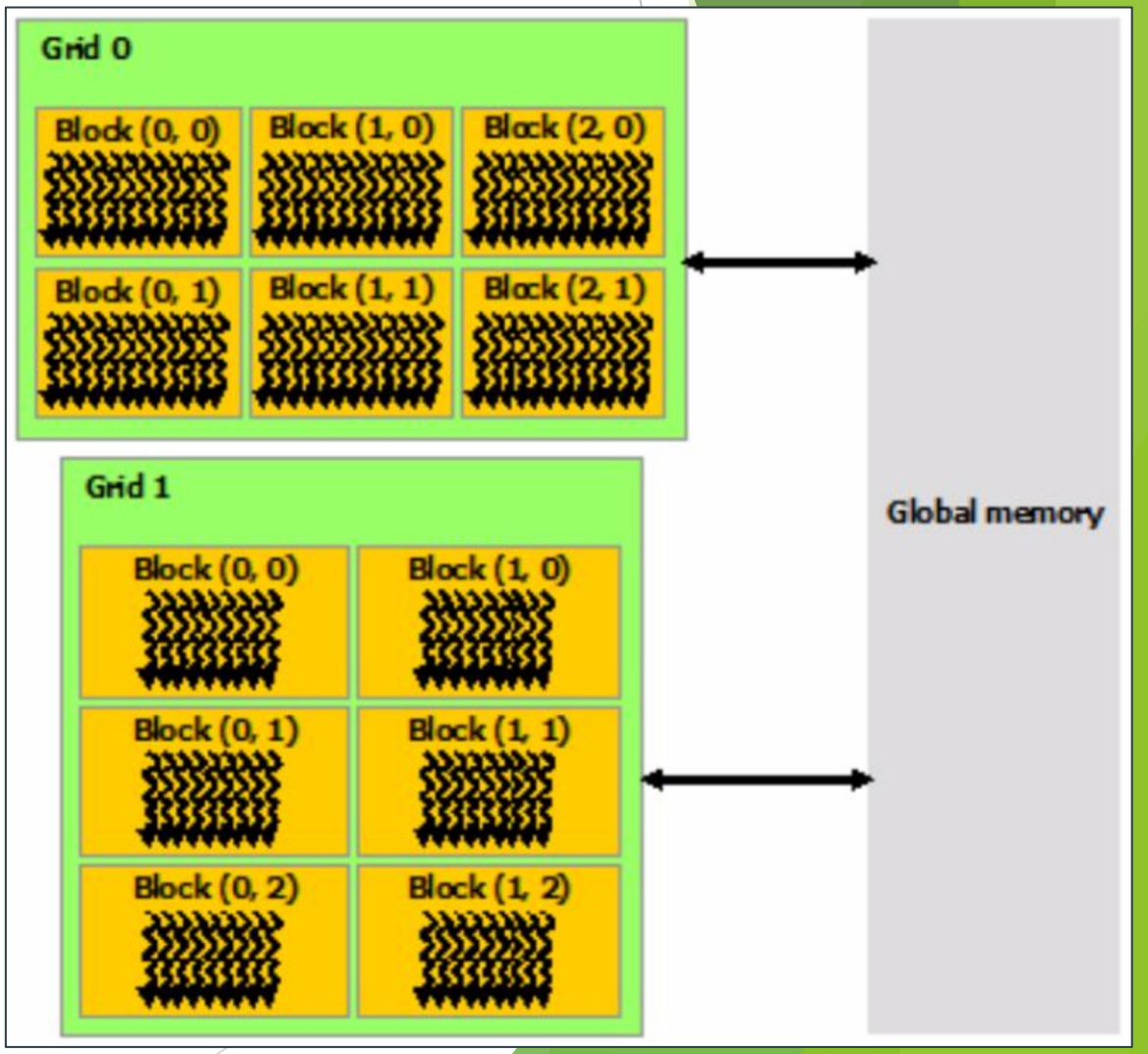
- **Memoria privada**: Propia e independiente de cada hilo.
- **Memorias adicionales**: Memorias constantes de sólo lectura y accesibles por todos los hilos:
  - Memoria de texturas: Organizada como una matriz. Los bloques de hilos organizados en 2D acceden más rápido a esta que a la global.
  - Memoria constante: Datos que se cargarán en la cache de constantes.

Jerarquía de memoria (II)

- Gráficamente:







Jerarquía de memoria (III)

- Al declarar variables:
  - Si no se pone nada, será una variable local de cada hilo que ejecute el kernel.
  - Si se precede de **__device__** será una variable almacenada en la tarjeta (device):
    - Si se pone sólo **__device__** ***variable*** se guardará en memoria global (lenta) y accesible por todos los hilos.
    - Si se combina con otro identificador se guardará en otro tipo de memoria de la GPU.
  - **__constant__** ***variable*** se guardará en la memoria de sólo lectura compartida por todos los hilos.
  - **__shared__** ***variable*** será una variable guardada en la memoria compartida de cada SM y se compartirá por todas las variables del mismo bloque.

```
__shared__ float shared[];
```

Jerarquía de memoria (IV)

O de forma más visual:

Muestra	Memoria	Alcance	Persistencia
__device__ int globalVar;	Global	Grid	Toda la ejecución
__device__ __shared__ int sharedVar;	Compartida	Bloque	Ejecución del bloque
__device__ __constant__ int constantVar;	Constante	Grid	Toda la ejecución

Tabla adaptada del tutorial de Volodymyr Kindratenko

Jerarquía de memoria (V)

- Para reservar memoria desde el host:
  - ***cudaMalloc****(void** puntero, size_t numBytes)*
  - ***cudaMallocManaged****(void** puntero, size_t numBytes)*: Reserva memoria para el host y la GPU con una sola llamada y podemos acceder al mismo puntero (aunque físicamente ambas memorias están separadas). Enlace de interés: https://developer.nvidia.com/blog/unified-memory-cuda-beginners/
- Para copiar en la memoria:
  - ***cudaMemcpy****(void* destino, void* origen, size_t nBytes, {cudaMemcpyDeviceToHost, cudaMemcpyHostToDevice})*
- Para inicializar memoria:
  - **cudaMemset***(void * puntero, int valor, size_t nBytes)*
- Para liberar memoria desde el host:
  - **cudaFree***(void* puntero)*

Sincronización y atomicidad (I)

- La función ***__syncthreads()*** sincroniza los hilos de un mismo bloque (es como una barrera).
- La función ***__syncwarp(unsigned mask==xFFFFFFFF)*** sincroniza los hilos del mismo warp que cumplen con la máscara.
- La función ***cudaDeviceSynchronize()*** sincroniza toda la tarjeta, asegurando transferencias de datos al completo.

Sincronización y atomicidad (II)

- CUDA ofrece también una serie de funciones para hacer o**peraciones atómicas de lectura-modificación-escritura:**

<funcion>(direccion, valor)

atomicAdd	nuevo = original + valor
atomicSub	nuevo = original - valor
atomicExch	nuevo = valor
atomicMin	nuevo = min(original, valor)
atomicMax	nuevo = max(original, valor)
atomicInc	nuevo = ((original>=valor)? 0 : (original+1))
atomicDec	nuevo = (((original==0) | (original > nuevo)) ? nuevo : (original-1))
atomicCAS	nuevo = (original == comparado ? nuevo : original)
atomic{And,Or,XOR}	nuevo = {(original & valor), (original | valor), (original^valor)}

Ejemplo de reducción (I)

Vamos a hacer un producto escalar:

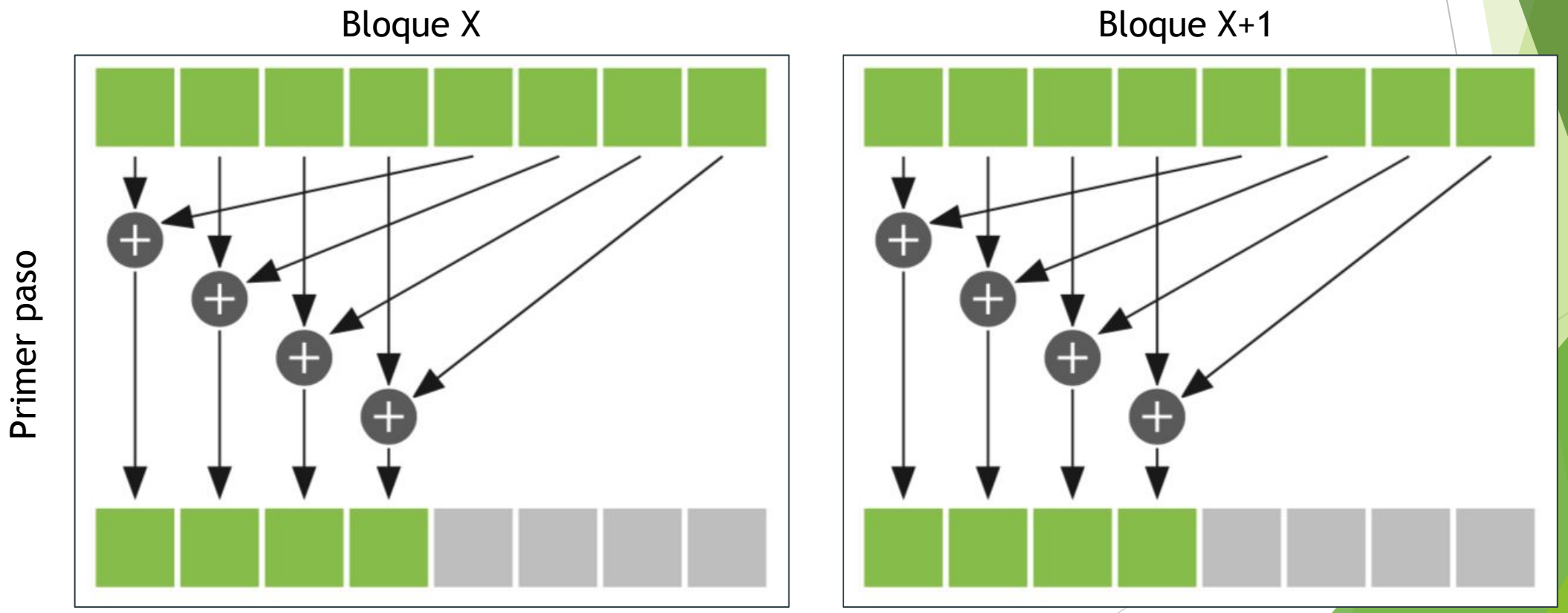
```
#include <cstdio>

#define N 2048
#define blocks 32
#define threadsPerBlock 32

__global__ void productoEscalar(double* a, double* b, double* c){
        __shared__ double partialVec[threadsPerBlock];
        int tid = blockIdx.x * blockDim.x + threadIdx.x;
        double buffer = 0.0;
        while(tid < N){
                buffer += ( a[tid] * b[tid] );
                tid += ( gridDim.x * blockDim.x );
        }
        partialVec[threadIdx.x] = buffer;
        __syncthreads();
        int localIndex = threadIdx.x;
        int i = blockDim.x / 2;
        while(i != 0){
                if(localIndex < i){
                        partialVec[localIndex] += partialVec[localIndex + i];
                }
                __syncthreads();
                i = i / 2;
        }
        if(localIndex == 0){
                c[blockIdx.x] = partialVec[0];
        }
}
```

Ejemplo de reducción (II)

- Estamos haciendo esto:



Ejemplo de reducción (III)

Vamos a hacer un producto escalar:

```
[ncalvo@atcgrid SeminarioCUDA]$ srun -p ac
4 nvcc productoEscalar.cu
[ncalvo@atcgrid SeminarioCUDA]$ srun -p ac
4 ./a.out
Resultado: 2861214720
```

Command Window

```
>> a = [0:1:2047];
>> b = a;
>> dP = a*b'

dP =

   2.8612e+09

>> fprintf('%f\n', dP);
2861214720.000000
```

```
int main(void){
        double* a = (double*) malloc(sizeof(double)*N);
        double* b = 0;
        cudaMallocManaged((void**) &b, sizeof(double)*N);
        double* bufferC = 0;
        cudaMallocManaged((void**) &bufferC, sizeof(double)*blocks);

        for(int i = 0; i<N; i++){
                a[i] = (double) i;
                b[i] = (double) i;
        }

        double* dev_A = 0;
        cudaMalloc((void**) &dev_A, sizeof(double)*N);
        cudaMemcpy(dev_A, a, sizeof(double)*N, cudaMemcpyHostToDevice);

        productoEscalar <<<blocks, threadsPerBlock>>> (dev_A, b, bufferC);
        cudaDeviceSynchronize();

        double finalResult = 0.0;
        for(int i = 0; i<blocks; i++){
                finalResult += bufferC[i];
        }
        printf("Resultado: %.lf\n", finalResult);

        cudaFree(a);
        cudaFree(b);
        cudaFree(bufferC);
        return 0;
}
```

Ejemplo de atomicidad

```
#include <cstdio>

__global__ void inc(unsigned long long int *foo) {
        atomicAdd(foo, 1);
}

int main(){
        unsigned long long int count = 0, *cuda_count;
        cudaMalloc((void**)&cuda_count, sizeof(unsigned long long int));
        cudaMemcpy(cuda_count, &count, sizeof(unsigned long long int), cudaMemcpyHostToDevice);
        printf("Valor inicial de count: %lld\n", count);
        inc <<< 100, 25 >>> (cuda_count);
        cudaMemcpy(&count, cuda_count, sizeof(unsigned long long int), cudaMemcpyDeviceToHost);
        cudaFree(cuda_count);
        printf("Valor final de count: %lld\n", count);
        return 0;
}
```

```
[ncalvo@atcgrid SeminarioCUDA]$ srun -p ac4 nvcc atomicAdd.cu
[ncalvo@atcgrid SeminarioCUDA]$ srun -p ac4 ./a.out
Valor inicial de count: 0
Valor final de count: 2500
```

Ejemplo adaptado de: https://stackoverflow.com/questions/17302752/cuda-atomicadd-with-long-long-int

Funciones de gestión del dispositivo

```
__host__ __device__ cudaError_t cudaGetDeviceCount ( int* count )
```

- Devuelve el número de dispositivos CUDA compatibles

```
__host__ cudaError_t cudaGetDeviceProperties ( cudaDeviceProp* prop, int device )
```

- Obtiene información del dispositivo número *device*

```
__host__ cudaError_t cudaSetDevice ( int device )
```

- Escoge la GPU indicada para ejecutar los kernels

```
__host__ __device__ cudaError_t cudaGetDevice ( int* device )
```

- Devuelve el índice del dispositivo usado

```
__host__ cudaError_t cudaChooseDevice ( int* device, const cudaDeviceProp* prop )
```

- Busca el dispositivo que mejor se ajusta a las propiedades indicadas

Gestión de errores

- Las funciones de CUDA devuelven un código de error que se va sobreescribiendo cada vez que se da uno nuevo. Podemos consultarlo con:

```
__host__ __device__ cudaError_t cudaGetLastError ( void )
```

- Lee el estado de la variable de error y la resetea a cudaSuccess

```
__host__ __device__ const char* cudaGetErrorString ( cudaError_t error )
```

- Devuelve el mensaje asociado a un código de error

```
cudaError_t err = cudaGetLastError();
if (cudaSuccess != err){
    printf("CUDA error: %s.\n", cudaGetErrorString( err) );
}
```

Medición de tiempos

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);

cudaEventRecord(start);
saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);
cudaEventRecord(stop);

cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);

cudaEventSynchronize(stop);
float milliseconds = 0;
cudaEventElapsedTime(&milliseconds, start, stop);
```

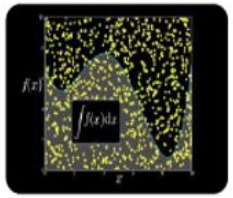
https://developer.nvidia.com/blog/how-implement-performance-metrics-cuda-cc/

Compilador

- Como habrás visto, el compilador usado es **nvcc**, un wrapper de NVidia sobre GCC.
- Éste genera código CUDA para los kernels con un repertorio de instrucciones llamado PTX y código para la CPU según el repertorio que le corresponda.
- Opciones disponibles: *nvcc —help*
- nvcc puede generar en el mismo código para diferentes dispositivos o GPUs indicándoselo con los flags -gencode, arch=compute_XX y code=sm_XX (Ver makefiles de los cuda-samples).
- Para consultar las capacidades de cada serie se puede usar el siguiente enlace: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities

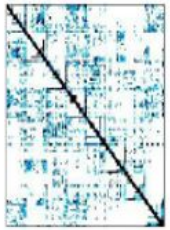
Herramientas disponibles (I)

NVIDIA cuBLAS



NVIDIA cuRAND

NVIDIA NPP



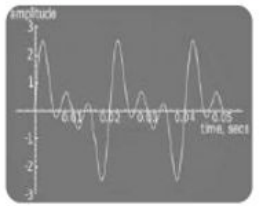
NVIDIA cuSPARSE

CUSP

Sparse Linear Algebra

Thrust

C++ STL Features for CUDA



NVIDIA cuFFT

Herramientas disponibles (II)

- profilers, como nvprof
- cuda-memcheck
- nsight es un entorno de desarrollo para cuda que incluye interacción con la CPU y con la GPU, pero es gráfico
- CUPTI es el interfaz para hacer profiling y tracing en programas cuda
- Utilidades binarias como cuobjdump que te proporciona información de ficheros binarios como para qué arquitectura están compilados por ejemplo o el código en ensamblador de las instrucciones que se ejecutan con cuobjdump -ptx

Enlaces de interés

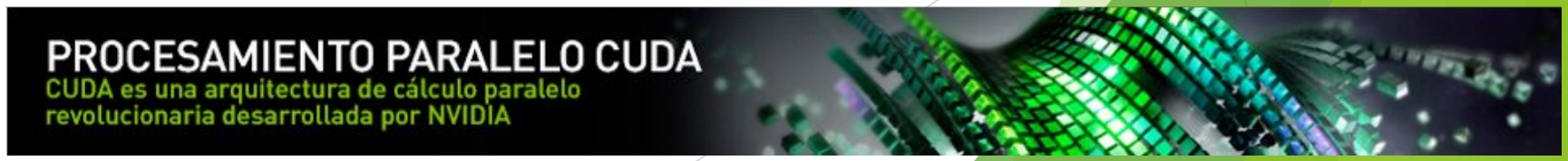
https://developer.nvidia.com/cuda-zone

https://developer.nvidia.com/cuda-education

https://developer.nvidia.com/accelerated-computing-training

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html



¡Gracias!

¿Dudas?