

Arquitectura y Programación de Altas Prestaciones

Práctica 4 y 5: CUDA Inicio



Especialidad Ingeniería de Computadores
Departamento de Arquitectura y Tecnología
de los Computadores
Maribel García Arenas
mgarenas@ugr.es





Objetivos

- Conocer el entorno de ejecución heterogéneo para computación paralela
- Conocer arquitecturas de altas prestaciones con soporte para programación SIMD
- Conocer las características básicas de las arquitecturas Graphics Processors Units
- Programar con CUDA



Indice

1. Introducción a la computación heterogénea
2. Nociones de Cuda
3. Nsight
4. Ejemplo 1
5. Depurar
6. Profiling
7. Práctica a Realizar





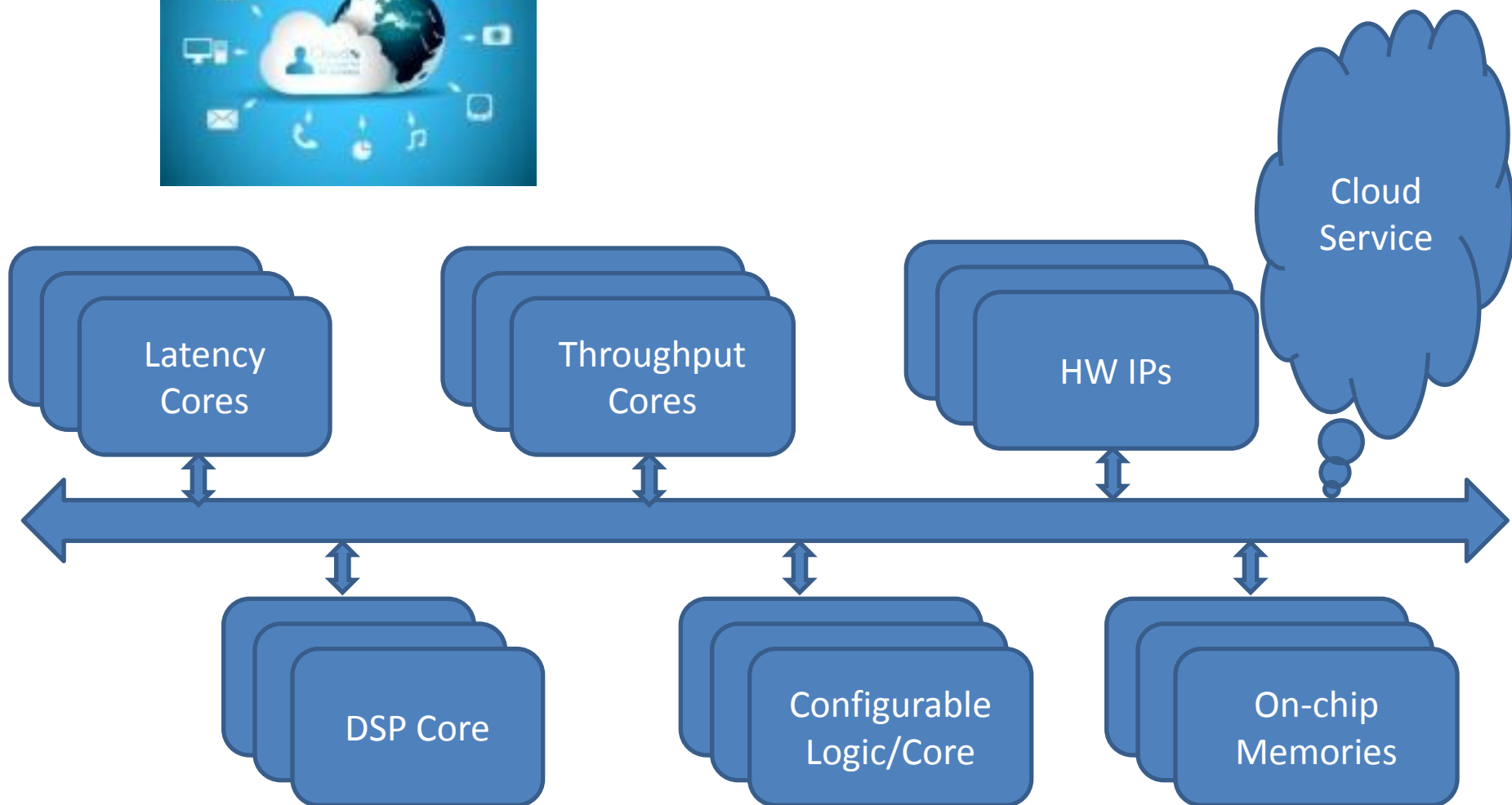
Entorno de trabajo

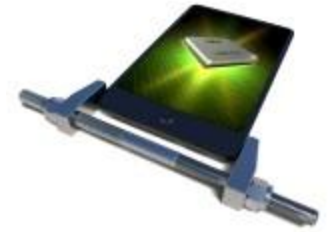
- Máquinas con GPU, concretamente con la GPU XXXXX
- Compilador con soporte para generar código que utilice la GPU
- Versión x de CUDA
- NVIDIA Nsight Eclipse Edition
 - Nsight source code Editor
 - Nsight debugger
 - Nsight profiler





Computación Heterogénea

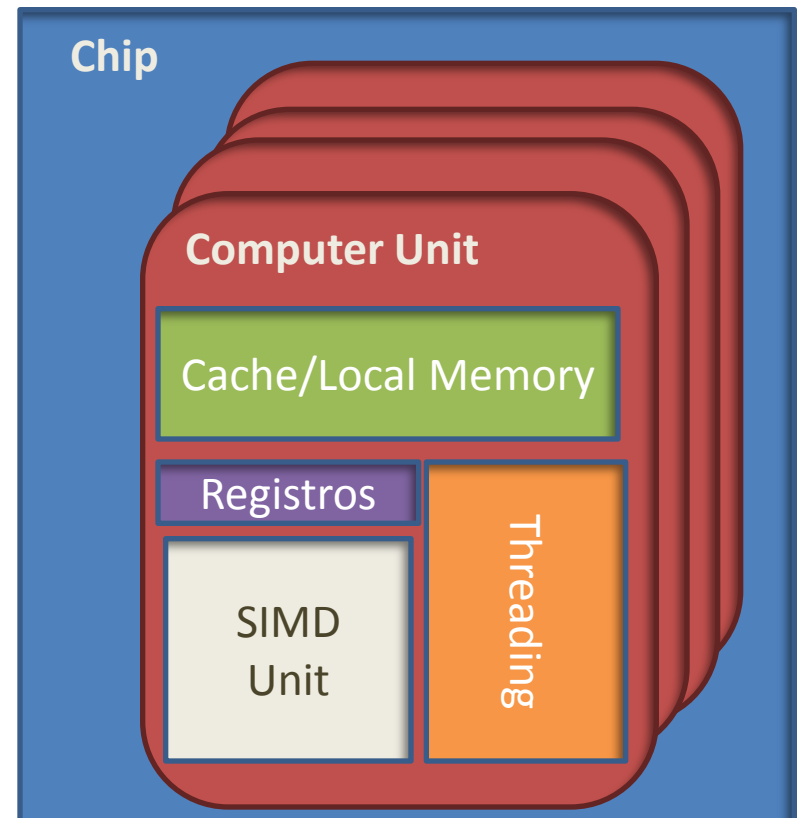
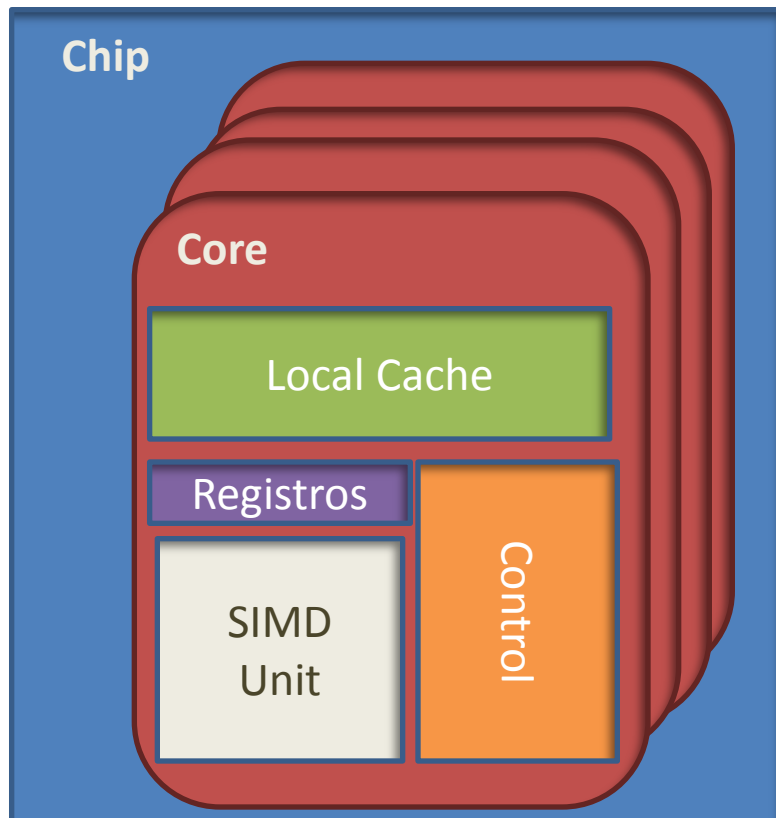




CPU vs GPU

Latency oriented cores:
Diseñados para optimizar la
latencia de las instrucciones

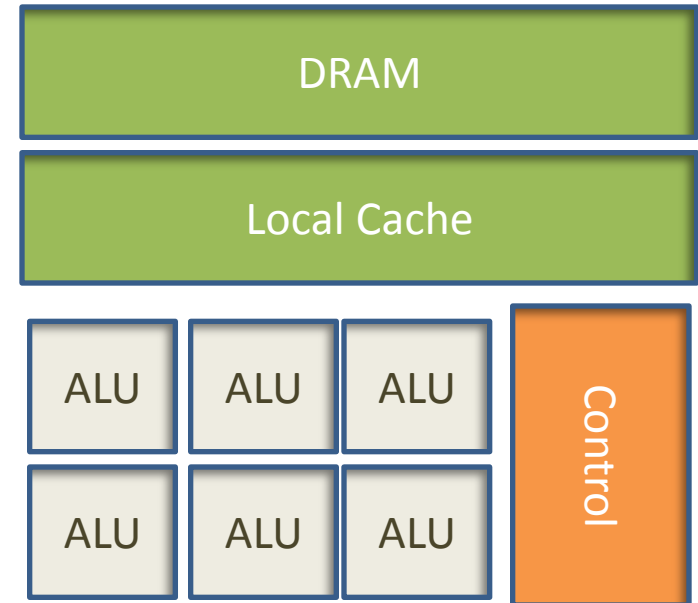
Throughput Oriented cores:
Diseñados para ejecutar muchas
instrucciones por unidad de tiempo

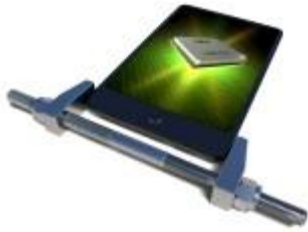




CPU: Diseño orientado a suplir la latencia de la memoria

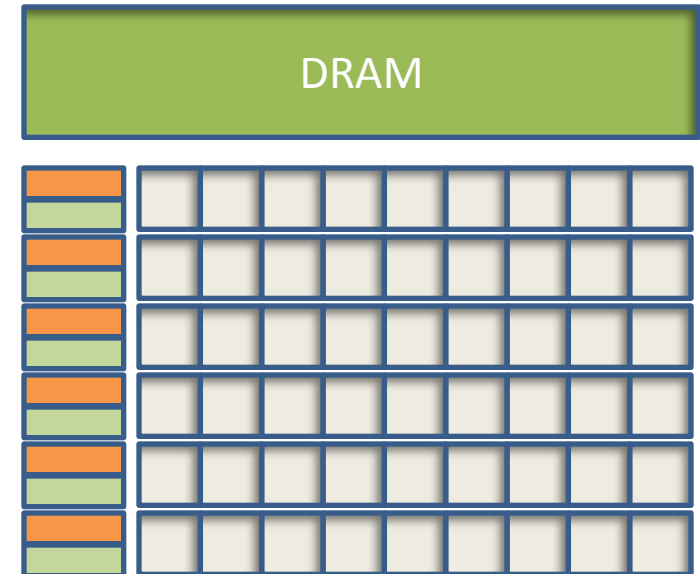
- **Caches Grandes**
 - Para intentar minimizar la latencia del acceso a memoria principal
- **Unidades de Control complejas:**
 - Branch Prediction: para minimizar la latencia de los saltos
 - Data Forwarding: Para reducir la latencia de las dependencias de datos (hazards/dependencias/riesgos)
- **ALUs muy potentes**
 - Para reducir la latencia de ejecución en cada tipo de instrucción





GPUs: Diseño orientado a la productividad

- Caches pequeñas
 - Para incrementar la productividad
- Unidades de Control simples:
 - Sin Branch Prediction
 - Sin Data Forwarding
- ALUs energéticamente eficientes
 - Hay muchas, más lentas pero altamente segmentadas para una mejor productividad
- Necesitan ejecutar muchas hebras simultáneas para enmascarar las latencias



Ventajas de usar CPUs y GPUs



- CPUs para las partes secuenciales
- La CPU puede ser 10 veces más rápidas que las GPUs para ejecutar código secuencial
- Poseen APIs de funciones extensas para código secuencial.
- GPUs para las partes paralelas donde la productividad debe prevalecer sobre el resto
- Utiliza el modelo de paralelización de SIMD.
- Poseen una API sencilla y especializada para un soporte específico.



Qué pasa con el Coste del software?

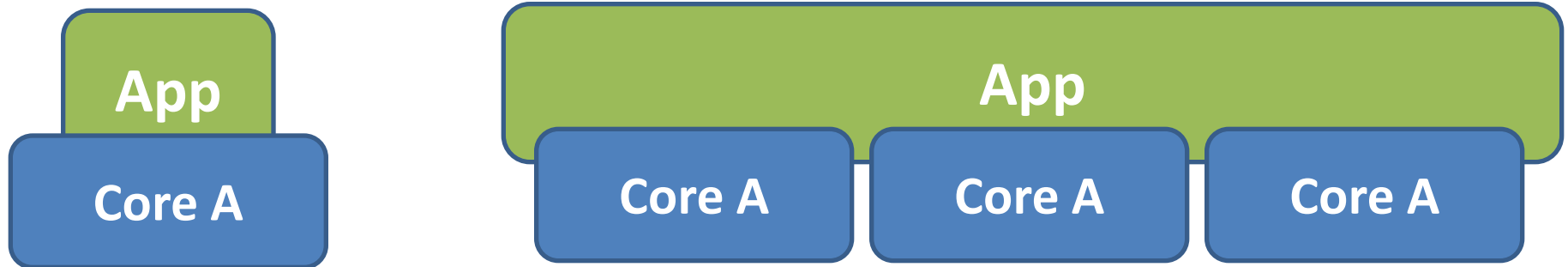
- Cada día el coste software se incrementa por:
 - Nuevas plataformas
 - Nuevas técnicas de optimización de hardware
- Además:
 - Las líneas software ejecutadas en cada chip se duplican cada 10 meses.
 - La ley de Moore con respecto a la densidad de integración que se duplica cada 18 meses.
- Por lo que debemos minimizar el proceso de “Redesarrollo” del software en todos sus niveles, puesto que actualmente cuesta más adaptar el software al nuevo hardware que la construcción de nuevo hardware.
- Podemos controlar el coste del software preservando:
 - Escalabilidad del software desarrollado
 - Portabilidad del software que se desarrolle

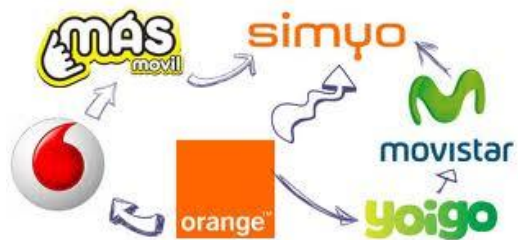




Escalabilidad

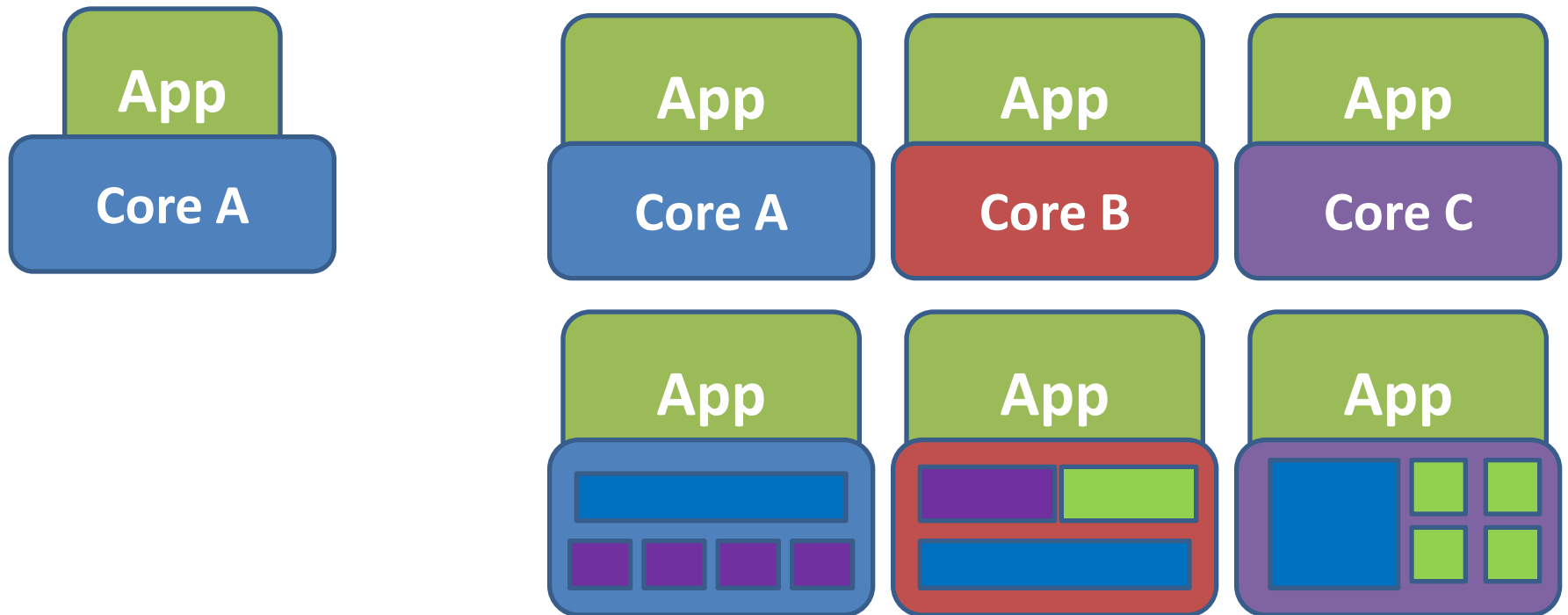
- La misma aplicación debe ejecutarse eficientemente en nuevas generaciones de plataformas
- La misma aplicación se ejecuta en más cores del mismo tipo



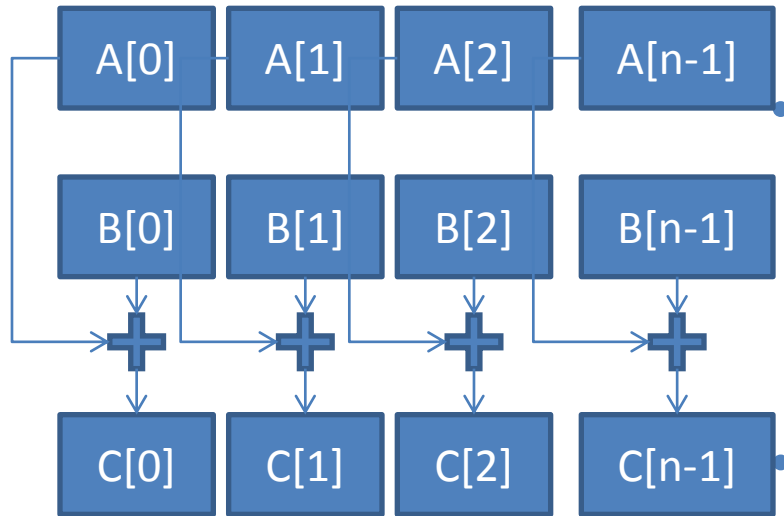


Portabilidad

- La misma aplicación se debe ejecutar eficientemente en diferentes tipos de cores
- La misma aplicación se debe ejecutar eficientemente en sistemas con organizaciones e interfaces hardware diferentes



Paralelismo de Datos: Suma de Vectores



Aplicación en C

- Partes paralelas en la GPU (kernels) (device code)
- Partes Serie en la CPU (host code)

Esquema

Parte Serie 1

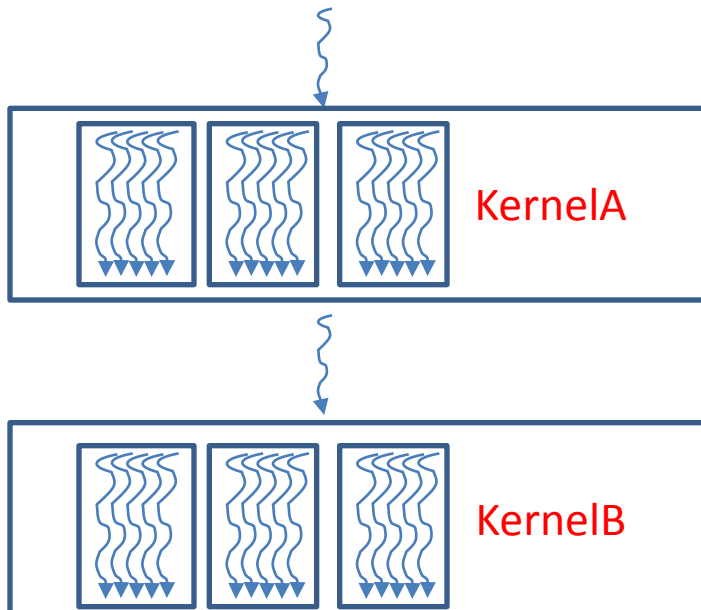
Kernel paralelo(device)

```
kernelA<<<nBlk,nTid>>>(args);
```

Parte Serie 2

Kernelparalelo(device)

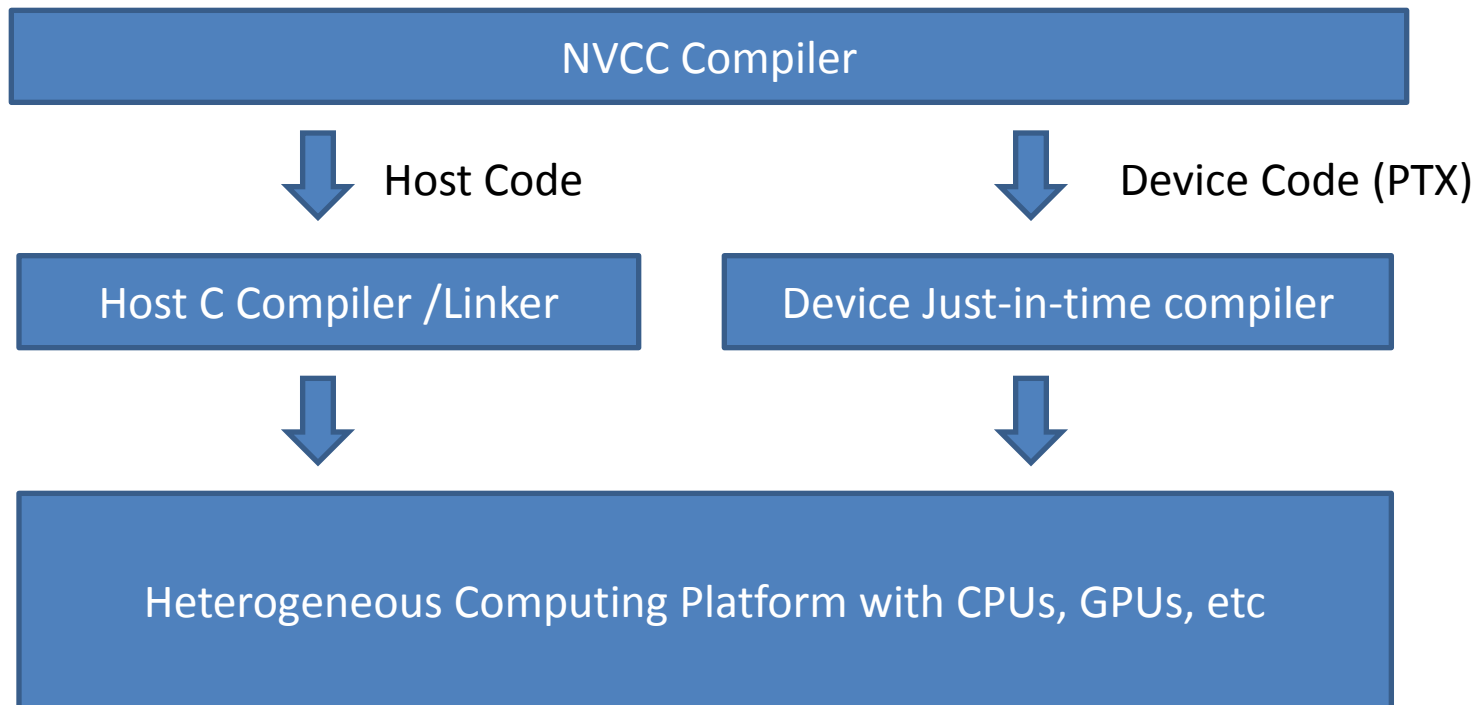
```
kernelB<<<nBlk,nTid>>>(args);
```





Compiler CUDA

Integrated C programs with CUDA extensions



Compilar

- `/usr/local/cuda-5.0/bin/nvcc -m64 -I/usr/local/cuda-5.0/include -o vectorAdd vectorAdd.cu`
- Opciones básicas:
 - `-m64`: generación de código para 64 bits
 - `-I` paths de librerías a incluir
- Opciones:
 - `/usr/local/cuda-5.0/bin/nvcc --help`



Ejemplos

- Buscar directorio `NVIDIA_CUDA-5.0_Samples`
- Categorías
 - 0_Simple
 - 1_utilities
 - 2_Graphics (mandelbrot)
 - 3_Imaging (histogram)
 - 4_Finance (quasi randomGenerator)
 - 5_Simulations
 - 6_Advanced
 - 7_CUDALibraries (Pi)
- Compilar y Ejecutar los ejemplos anteriores





Arrays de Hebras paralelas

- Un kernel en cuda se ejecuta en un “grid” o “array” de hebras
 - Todas las hebras **en un grid ejecutan el mismo código** (SPMD)
 - Cada hebra tiene una identidad única mediante las que se identifican y se puede guiar la ejecución
- Las hebras están organizadas en “blocks”
 - Todas las hebras del mismo bloque comparten memoria, son capaces de ejecutar operaciones atómicas y se pueden sincronizar.
 - Las hebras en bloques diferentes no interactúan





Bloques de Hebras

Trhead Block 0

0 1 255

```
i = blockIdx.x*blockDim.x  
+threadIdx.x;  
C[i] = A[i]+B[i];
```

Trhead Block 1

0 1 255

```
i = blockIdx.x*blockDim.x  
+threadIdx.x;  
C[i] = A[i]+B[i];
```

Trhead Block N-1

0 1 255

```
i = blockIdx.x*blockDim.x  
+threadIdx.x;  
C[i] = A[i]+B[i];
```

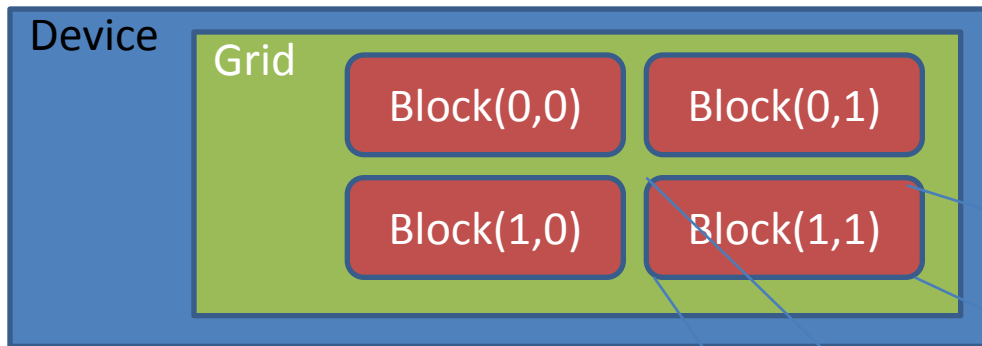
La sincronización y la
comunicación es a
nivel de bloque

No hay interacción entre
bloques!!!

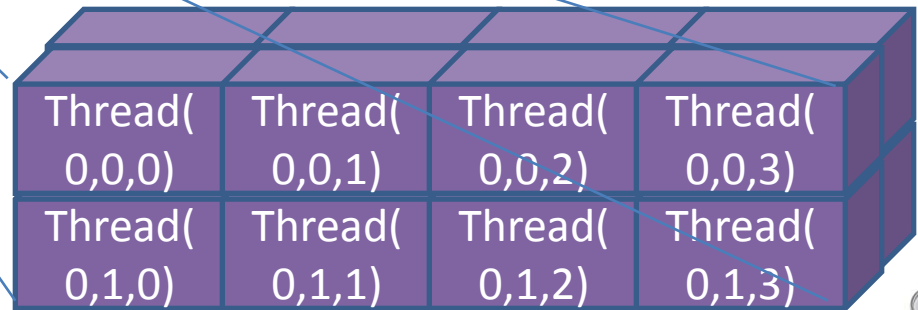


blockIdx y threadIdx

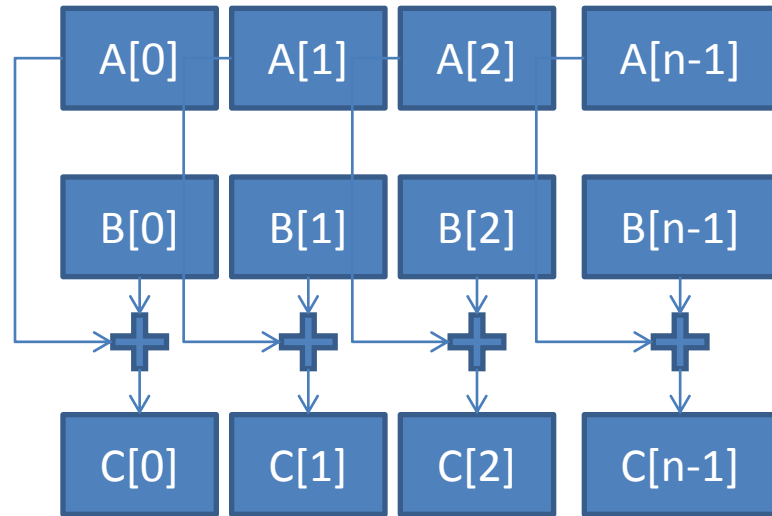
- blockIdx: 1D, 2D o 3D (cuda 4.0): blockIdx.x, blockIdx.y...
- threadIdx: 1D, 2D o 3D; threadIdx.x, threadIdx.y, threadIdx.z
- blockDim: 1D, 2D o 3D (cuda 4.0); blockDim.x, blockDim.y...



Cada hebra posee sus índices y los utiliza para acceder a direcciones de memoria o llevar el control de la ejecución

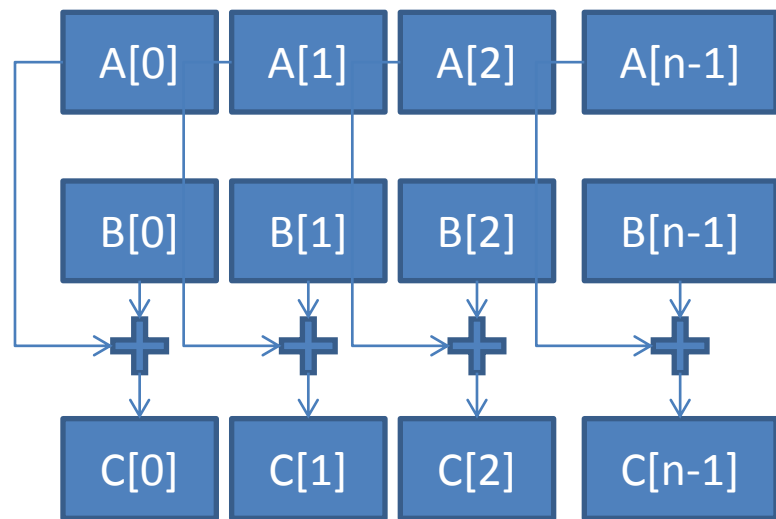


Ejemplo: Suma de dos vectores

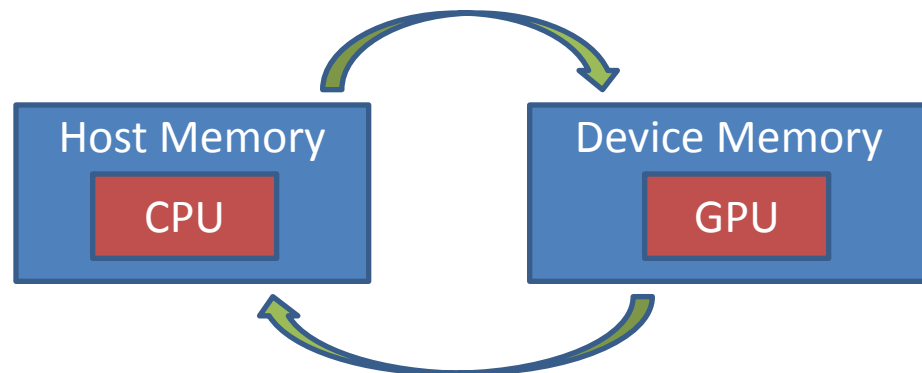


```
Void vecAdd(float* h_A, float* h_B, float* h_C, int n){  
    int i;  
    for(i=0;i<n;i++) h_C[i]=h_A[i]+h_B[i];  
}  
  
int main (){  
    // almacenamiento de n elementos h_A,h_B y h_C  
    vecAdd(h_A, h_B, h_C, n);  
}
```





Ejemplo: Suma de dos vectores

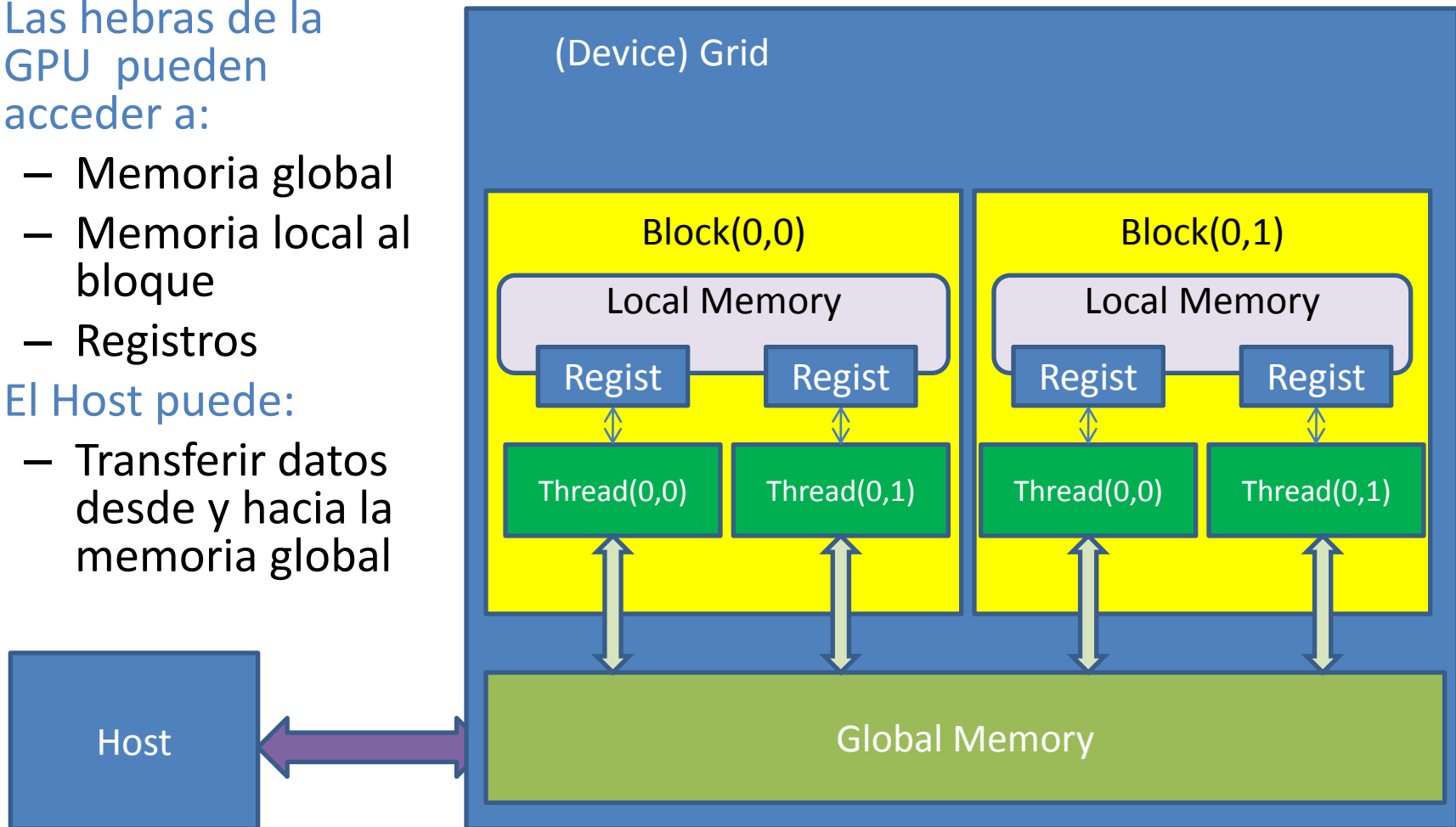


```
#include <cuda.h>
Void vecAdd(float * h_A, float h_B, float h_C, int
n) {
    int size = n*sizeof(float);
    float * d_A, d_B, d_C;
    // Alocate memory para A, B, C
    // Copiar A y B a la memoria del dispositivo
    // lanzar el kernel
    // copiar C desde el dispositivo
}
```



Memorias en la GPU

- Las hebras de la GPU pueden acceder a:
 - Memoria global
 - Memoria local al bloque
 - Registros
- El Host puede:
 - Transferir datos desde y hacia la memoria global



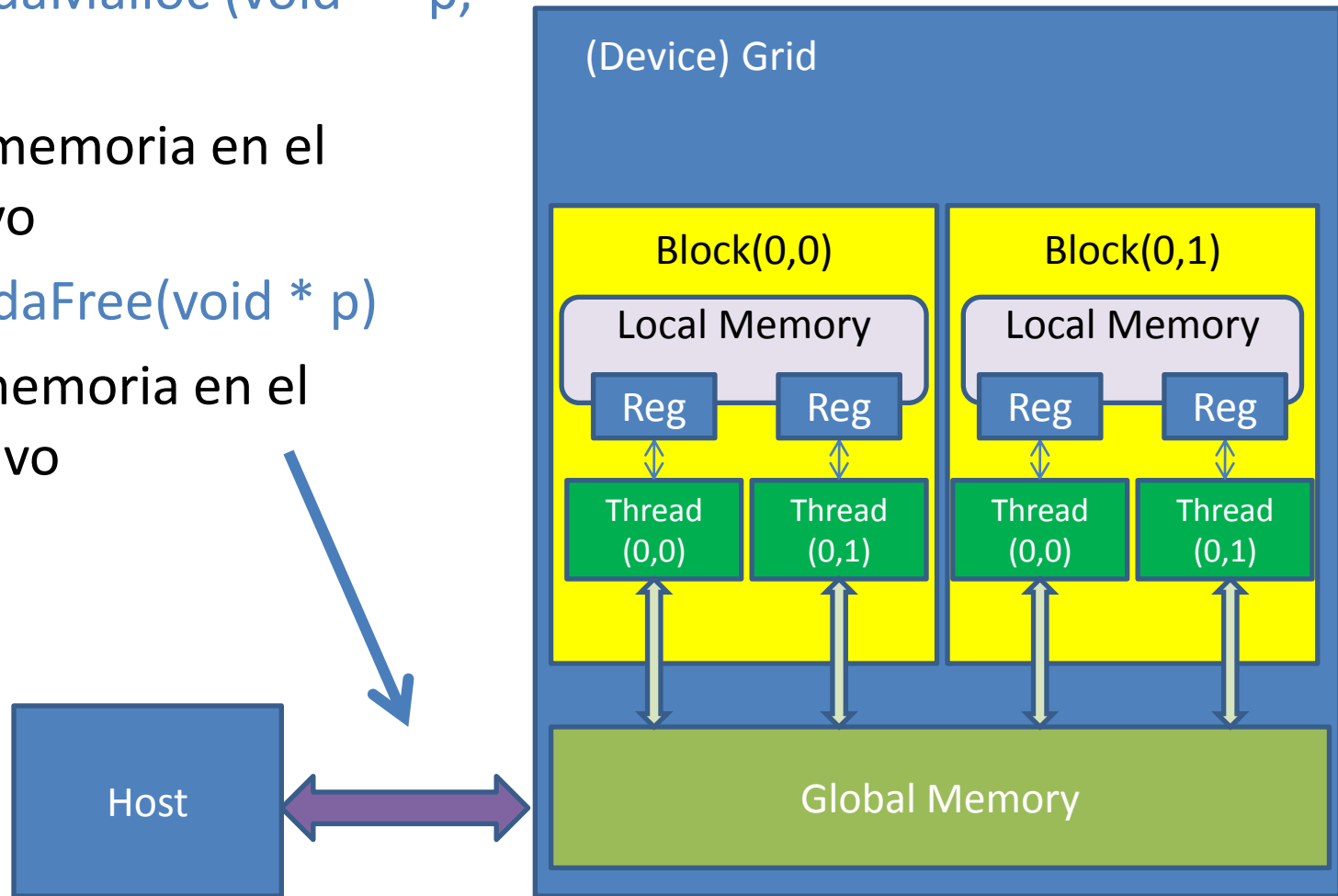
Memorias en la GPU

`cudaError_t cudaMalloc (void ** p,
int n)`

- Reserva memoria en el dispositivo

`cudaError_t cudaFree(void * p)`

- Libera memoria en el dispositivo



Memorias en la GPU

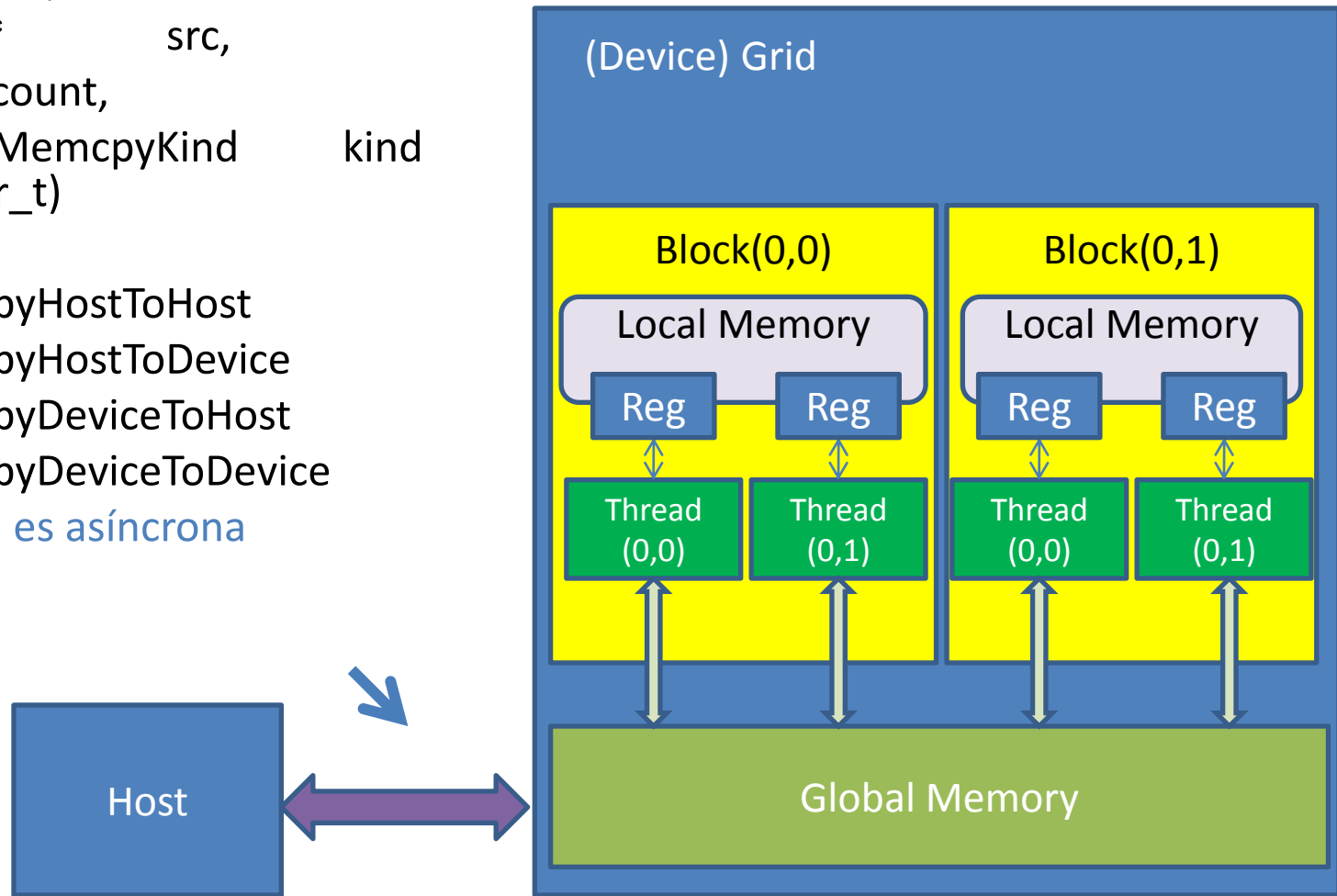
`cudaMemcpy(`

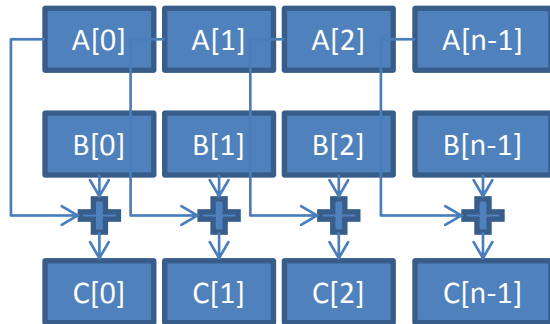
- `void * dst,`
- `const void * src,`
- `size_t count,`
- `enum cudaMemcpyKind kind`
`cudaError_t)`

- Los tipos son:

- `cudaMemcpyHostToHost`
- `cudaMemcpyHostToDevice`
- `cudaMemcpyDeviceToHost`
- `cudaMemcpyDeviceToDevice`

- La transferencia es asíncrona





Ejemplo: Suma de dos vectores



```
void vecAdd(float *hA, float *hB, float *hC, int n){  
    int size = n*sizeof(float);  
    float * dA, dB, dC;  
    cudaMalloc((void **) &dA, size);  
    cudaMemcpy(dA, hA, size, cudaMemcpyHostToDevice);  
    cudaMalloc((void **) &dB, size);  
    cudaMemcpy(dB, hB, size, cudaMemcpyHostToDevice);  
    cudaMalloc((void **) &dC, size);  
    // llamada al kernel  
    cudaMemcpy(hC, dC, size, cudaMemcpyDeviceToHost);  
    cudaFree(dA); cudaFree(dB); cudaFree(dC);  
}
```

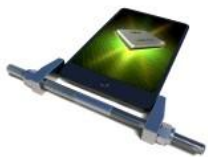


Consultar errores



```
cudaError_t err = cudaMalloc((void **) &dA,size);  
if(err != cudaSuccess){  
    printf("%s en %s en línea %d\n",  
        cudaGetErrorString(err),__FILE__, __LINE__);  
    exit(EXIT_FAILURE);  
}
```





Kernel de ejecución: Código en la GPU y en el host

GPU

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

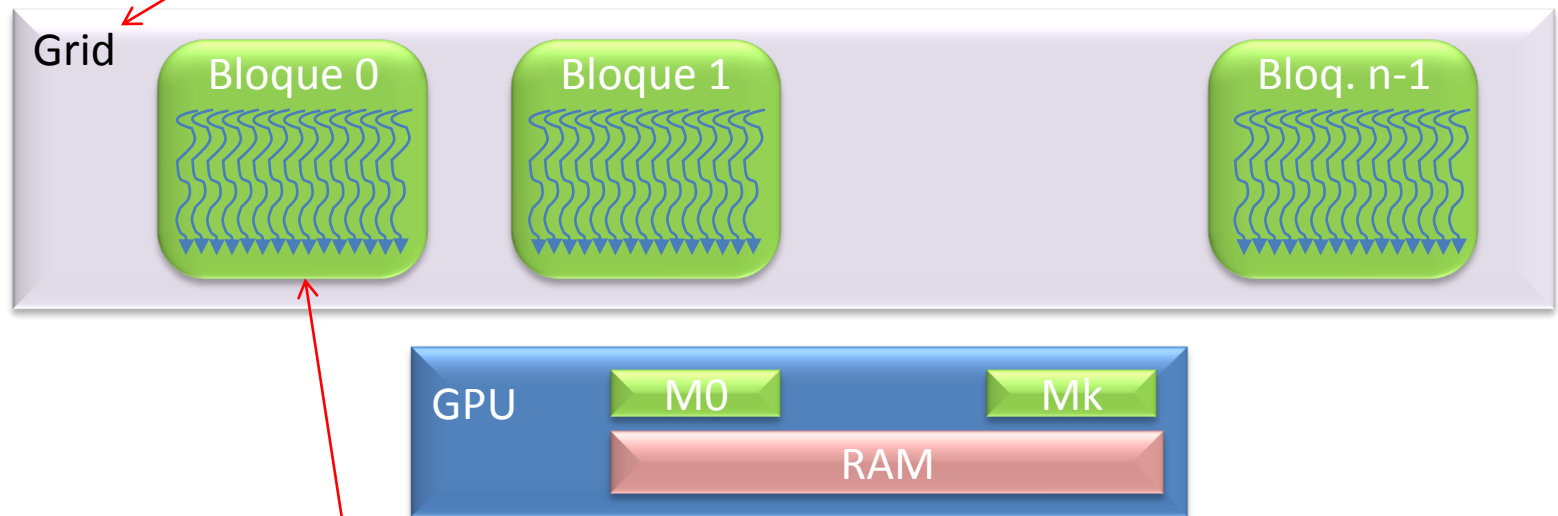
CPU

```
int vecAdd(float *hA, float *hB, float *hC, int n){  
    // dA, dB, dC tal y como lo hemos visto antes  
    // Para bloques de hebras de 256  
    // Llamada al kernel  
    dim3 DimGrid(((n-1)/256)+1,1,1);  
    dim3 DimBlock(256,1,1);  
    vecAddKernel<<<DimGrid,DimBlock>>>(dA,dB,dC,n);  
}
```



Kernel de ejecución: Código en la GPU y en el host

```
int vecAdd(float *hA, float *hB, float *hC, int n){  
    dim 3 DimGrid(((n-1)/256)+1,1,1);  
    dim3 DimBlock(256,1,1);  
    vecAddKernel<<<DimGrid,DimBlock>>>(dA,dB,dC,n);}
```



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

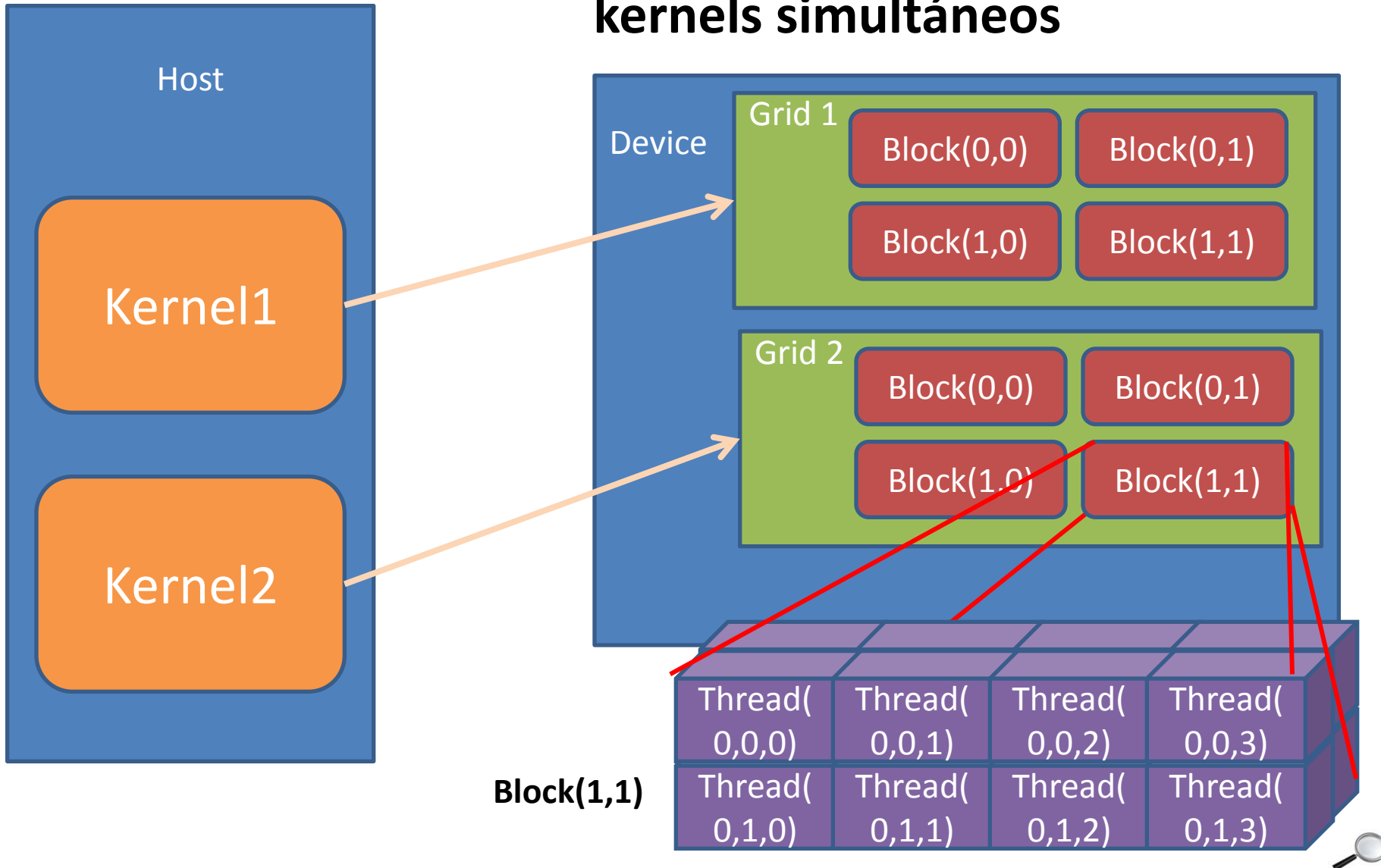


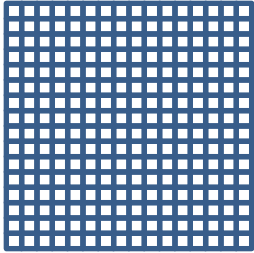
Declaración de funciones

	Se Ejecuta en	Se llama desde
<code>__device__ float deviceFunction()</code>	GPU	GPU
<code>__global__ void kernelFunction()</code>	GPU	Host
<code>__host__ float HostFunction()</code>	Host	Host

- `__global__` sirve para definir funciones kernel y debe devolver void
- `__device__` y `__host__` puede ser utilizada juntas

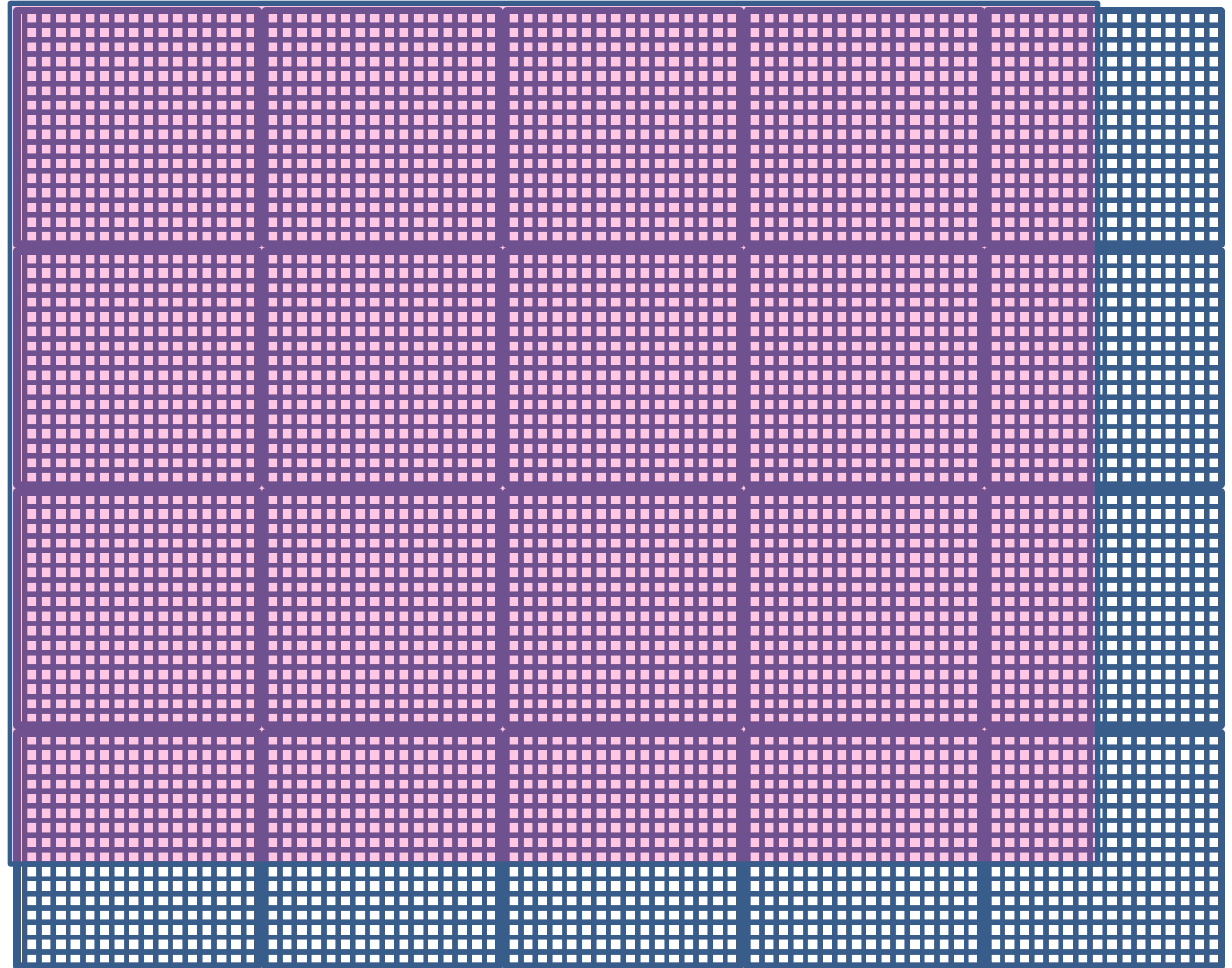
Un ejemplo multidimensional con varios kernels simultáneos





16x16 blocks

Procesamiento de una Imagen 2D



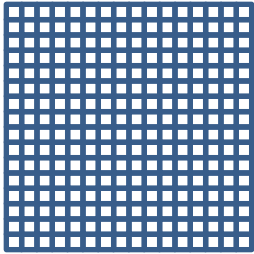
2D

Imagen Kernel

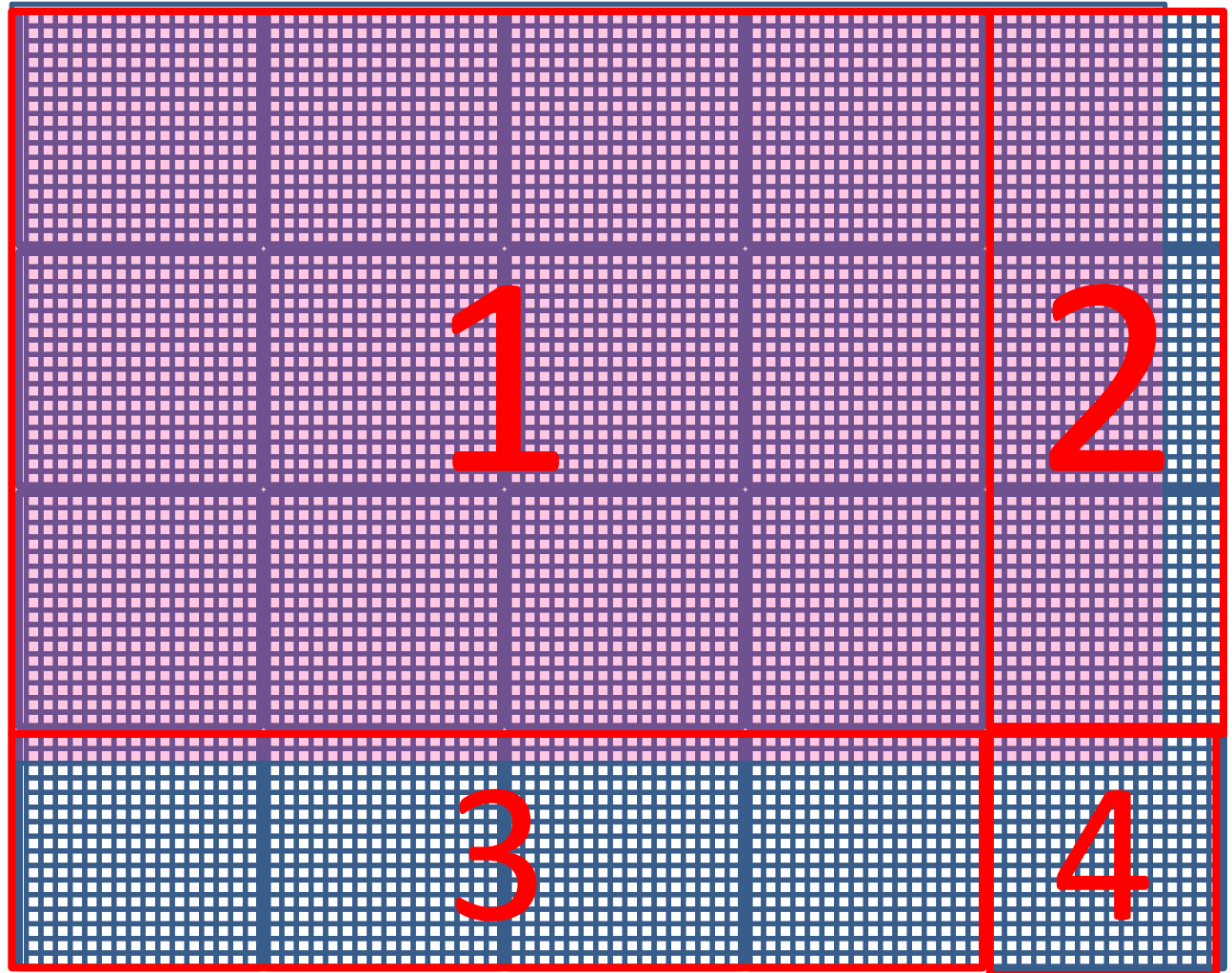
```
__global__ void ImagenKernel(float* dPin,  
    float* dPout, int n, int m){  
    int row = blockIdx.y*blockDim.y+threadIdx.y;  
    int col = blockIdx.x*blockDim.x+threadIdx.x;  
  
    // trabajo de cada hebra  
    if((row<m) && col<n){  
        dPout[row*n+col] = 2*dPin[row*n+col];  
    }  
}
```



Procesamiento de una Imagen 2D de 76x66



16x16 blocks

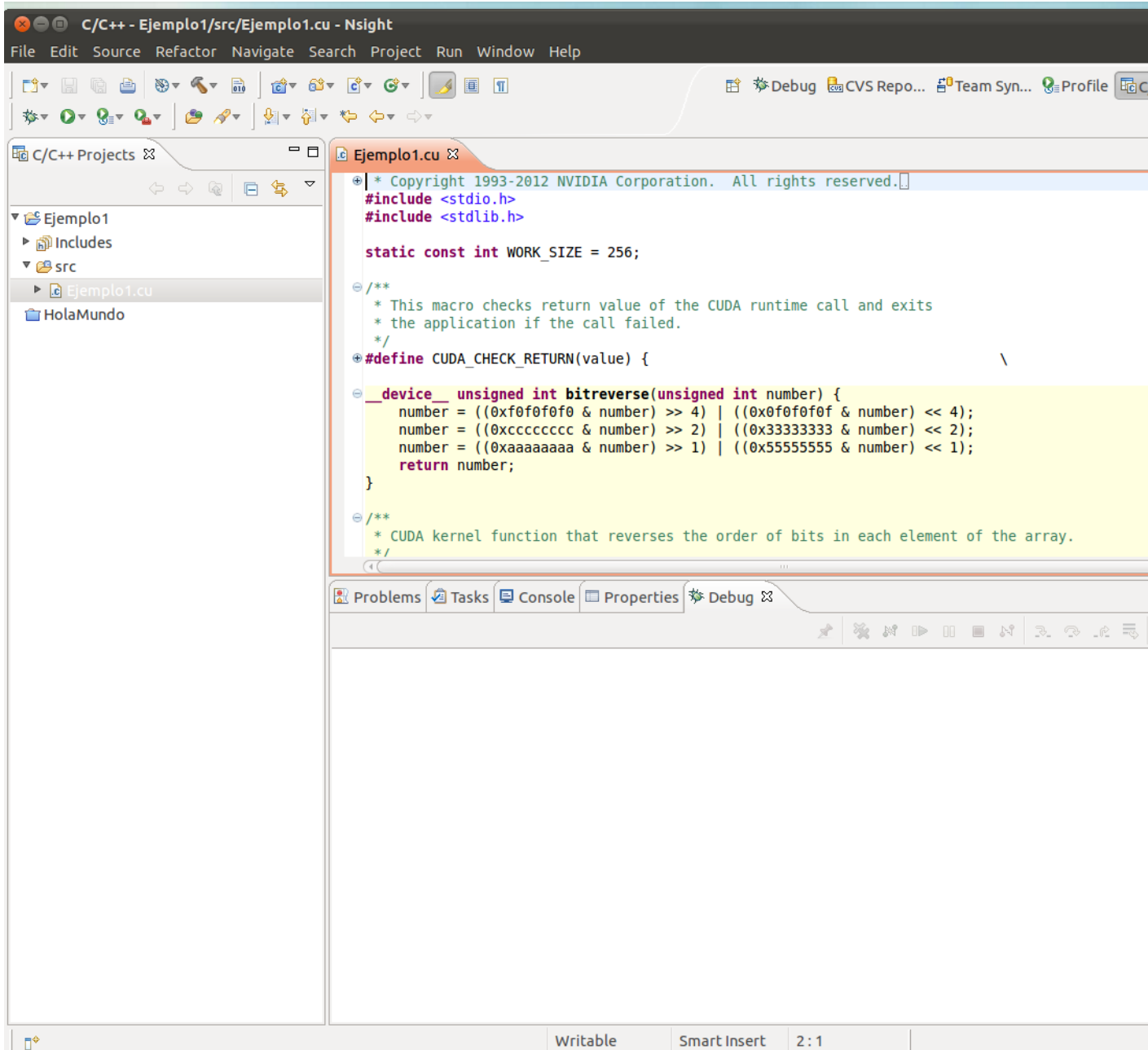


NSight

- Presentación Oficial Nsight
<https://developer.nvidia.com/nsight-eclipse-edition>
- Sigue las pautas y el formato que el entorno de programación Eclipse
- Permite:
 - Editar
 - Compilar
 - Depurar
 - Profiling
- Disponible para Linux y OSX



nsight





Ejemplo 1

- File->Nuevo->Project...
- Elegir la opción CUDA C/C++ Project
- Introducir el nombre “Ejemplo1”
- Seleccionar la ubicación para almacenar el proyecto dentro de vuestro directorio Home
- Seleccionar la opción por defecto “Cuda Runtime Project”
- En la pantalla de “Basic settings” dejar las opciones por defecto
- En la pantalla de “Select Configurations” dejar las opciones por defecto



Análisis de Ejemplo1

- ¿Qué hace Ejemplo1?
- Compilar igual que en eclipse 
- Ejecutar 
- Analizar la salida del método



Código comentado de reversebits

```
#include <stdio.h>
#include <stdlib.h>
// declaramos el tamaño de los vectores que vamos a utilizar como de 256 componentes
static const int WORK_SIZE = 256;
//Se trata de una macro que detecta si hay algún problema en el dispositivo y si no lo hay,
// ejecuta la llamada sin problema, y si hay algún problema, chequea el valor que devuelve el
// dispositivo usando máscaras.
#define CUDA_CHECK_RETURN(value) {
    cudaError_t _m_cudaStat = value;
    \
    if (_m_cudaStat != cudaSuccess) {
        \
        fprintf(stderr, "Error %s at line %d in file %s
        \
        n",
        \
        cudaGetErrorString(_m_cudaStat), __LINE__, __FILE__);
        \
        exit(1);
        \
    }
    //
    // Función de la GPU que le da la vuelta a los bits de un elemento entero
    __device__ unsigned int bitreverse(unsigned int number) {
        number = ((0xf0f0f0f0 & number) >> 4) | ((0x0f0f0f0f & number) << 4);
        number = ((0xcccccccc & number) >> 2) | ((0x33333333 & number) << 2);
        number = ((0xaaaaaaaa & number) >> 1) | ((0x55555555 & number) << 1);
        return number;
    }
    // Ejemplo de función __global__, acepta un vector de datos y llama a la función __device__
    // para invertir los bits almacenados en cada uno de los elementos del vector de entrada
    __global__ void bitreverse(void *data) {
        // se asigna el vector que se le pasa como parámetro al vector idata con un casting del
        mismo tipo.
        unsigned int *idata = (unsigned int*) data;
        // se almacena en la estructura local el resultado de la función para esta estructura
        pasándole como parámetro el identificador de la hebra.
        idata[threadIdx.x] = bitreverse(idata[threadIdx.x]);
    }
```

```
// Función main
int main(void) {
    // este es el puntero que va a apuntar a la zona de memoria de la GPU
    void *d = NULL;
    int i;
    // declaración del vector en la CPU. tendremos uno para la entrada idata, y otro para la
    // salida odata, ambos de tamaño WORK_SIZE
    unsigned int idata[WORK_SIZE], odata[WORK_SIZE];
    // Inicializo el vector, con los valores de 0 a 255
    for (i = 0; i < WORK_SIZE; i++)
        idata[i] = (unsigned int) i;
    // Se reserva memoria en la GPU
    CUDA_CHECK_RETURN(cudaMalloc((void**) &d, sizeof(int) * WORK_SIZE));
    // se copia el vector de la CPU a la GPU
    // parámetros:
    // d, puntero a la zona de memoria donde se van a copiar los datos
    // idata puntero a la zona de memoria desde donde se va a copiar datos
    // sizeof(int)*WORK_SIZE: Número de bytes a copiar
    // cudaMemcpyHostToDevice: Dirección de la copia
    CUDA_CHECK_RETURN(cudaMemcpy(d, idata, sizeof(int) * WORK_SIZE,
        cudaMemcpyHostToDevice));
    // Se llama a la función kernel
    // habrá WORK_SIZE bloques de hebras cada uno de los cuales tendrá
    // WORK_SIZE*sizeof(int) hebras independientes cada una
    // se le pasa a la función la zona de memoria de la GPU donde están dispuestos los datos.
    bitreverse<<<1, WORK_SIZE, WORK_SIZE * sizeof(int)>>>(d);
    // se sincronizan las hebras para que todas hayan terminado
    CUDA_CHECK_RETURN(cudaThreadSynchronize());
    // Se obtiene el error que cuda haya generado si es que existe
    CUDA_CHECK_RETURN(cudaGetLastError());
    // se copian los resultados de la GPU a la CPU utilizando el vector odata
    CUDA_CHECK_RETURN(cudaMemcpy(odata, d, sizeof(int) * WORK_SIZE,
        cudaMemcpyDeviceToHost));
    // se muestran los resultados por pantalla
    for (i = 0; i < WORK_SIZE; i++)
        printf("Input value: %u, device output: %u
        \
        n", idata[i], odata[i]);
    // se libera la zona de memoria de la GPU CUDA_CHECK_RETURN(cudaFree((void*) d));
    // se resetea el dispositivo para que quede con la configuración por defecto
    CUDA_CHECK_RETURN(cudaDeviceReset());
    return 0;
}
```

Depurar Ejemplo 1

- Hay que hacerlo con imaginación.
- A no ser que tengas más de una GPU y se utilice una para gestionar la salida estandar de la CPU y la otra para ejecutar cosas.
- No se puede usar la función `fprintf` con la salida `stderr`, puesto que el dispositivo GPU no conoce la salida estándar de error, el `stderr`.



Hacer profile de Ejemplo 1

- Abrir vista Profile (arriba a la derecha junto a la vista C/C++)
- Ejecutar el proyecto ya compilado
- Analizar la salida en el grafo de tiempos
- Opción de Analizar todo: Ejecuta el algoritmo varias veces y va tomando estadísticas de ejecución



Práctica a realizar

- Crear un proyecto que nos muestre por pantalla las características de la GPU donde vais a ejecutar vuestras prácticas utilizando la función `cudaGetDeviceProperties`
- <https://www.clear.rice.edu/comp422/resources/cuda/html/cuda-c-programming-guide/>
- Implementar una versión del problema que suma dos vectores que se ejecute sólo en la CPU.
- Implementar una versión del problema que suma dos vectores que se ejecute en la GPU la suma
- Realizar los tres pasos anteriores para los vectores que tenéis en swad en el fichero `datos20142015.tgz` tomando como entrada los ficheros `input0.raw` e `input1.raw` de cada tipo. Los ficheros incluyen en la primera línea el número de datos que tienen.
- Los tiempos de ejecución de la CPU y de la GPU debéis ponerlos en una tabla/gráfica donde se pueda apreciar la diferencia entre ambos e interpretar los resultados.

