INTRODUCCIÓN A LAS ARQUITECTURAS MASIVAMENTE PARALELAS: GPUS

Sergio Orts Escolano sorts@dtic.ua.es

José García Rodríguez jgarcia@dtic.ua.es

Universidad de Alicante

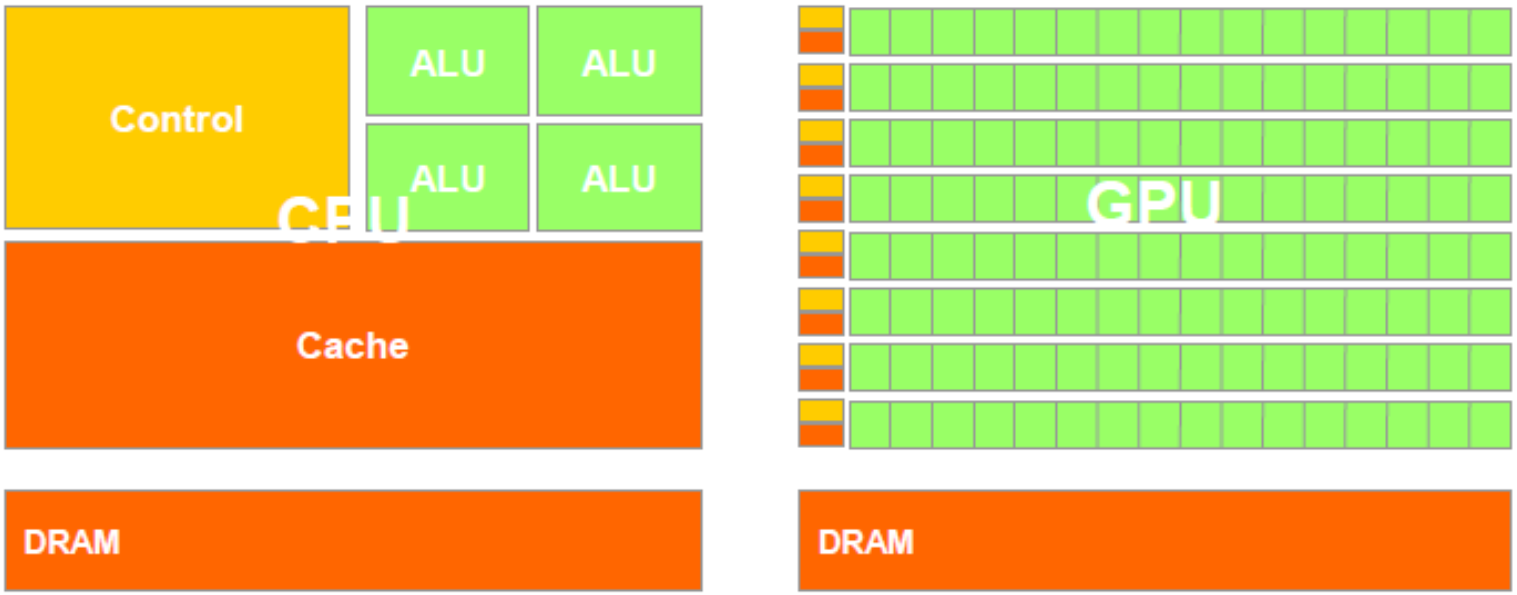
Departamento de tecnología informática y computación

Procesadores masivamente paralelos

- ¿Dónde encontramos procesadores masivamente paralelos?
  - *GPUs*
- ¿Diferencias respecto a la CPU tradicional?
- ¿Cómo aprovechar esa capacidad de cómputo que nos ofrece?

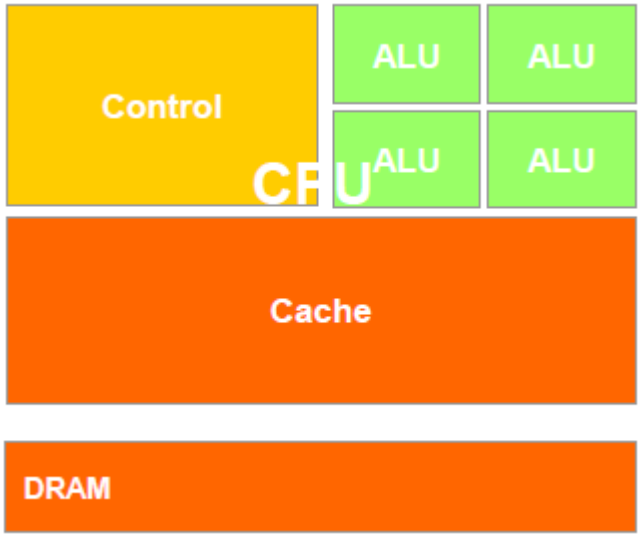
Comparación CPU y GPU

- **Las CPUs y las GPUs tienen fundamentos de diseño diferentes:**



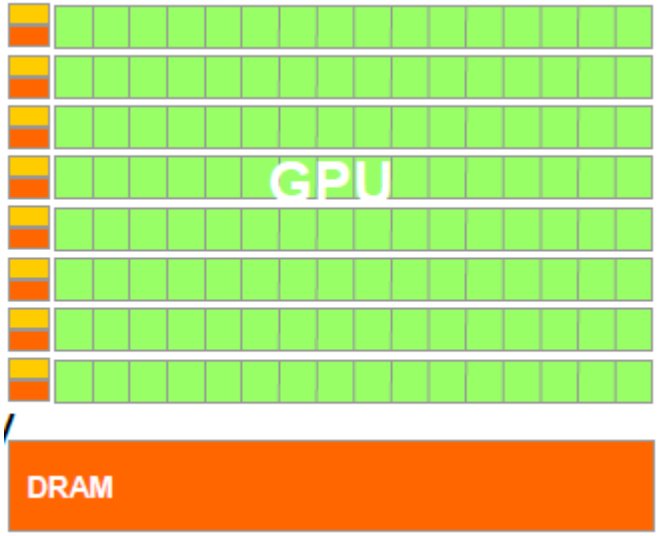
CPUs: Diseño orientado a latencia

- **CPU**
  - Caches grandes
    - Permiten bajar la latencia en los accesos a memoria
  - Unidad de control compleja
    - Branch prediction
    - Data forwarding
  - ALUs complejas
    - Reducen latencia de las operaciones



CPUs: Diseño orientado a latencia

- **GPU**
  - Caches pequeñas
    - Potenciar ancho banda memoria
  - Unidad de control simple
    - No Branch prediction
    - No Data forwarding
  - Muchas ALUs simples
    - Alta segmentadas para aumentar el ancho de banda de la memoria
  - Requiere un número muy elevado de hilos para ocultar las latencias



Diferencias CPU y GPU

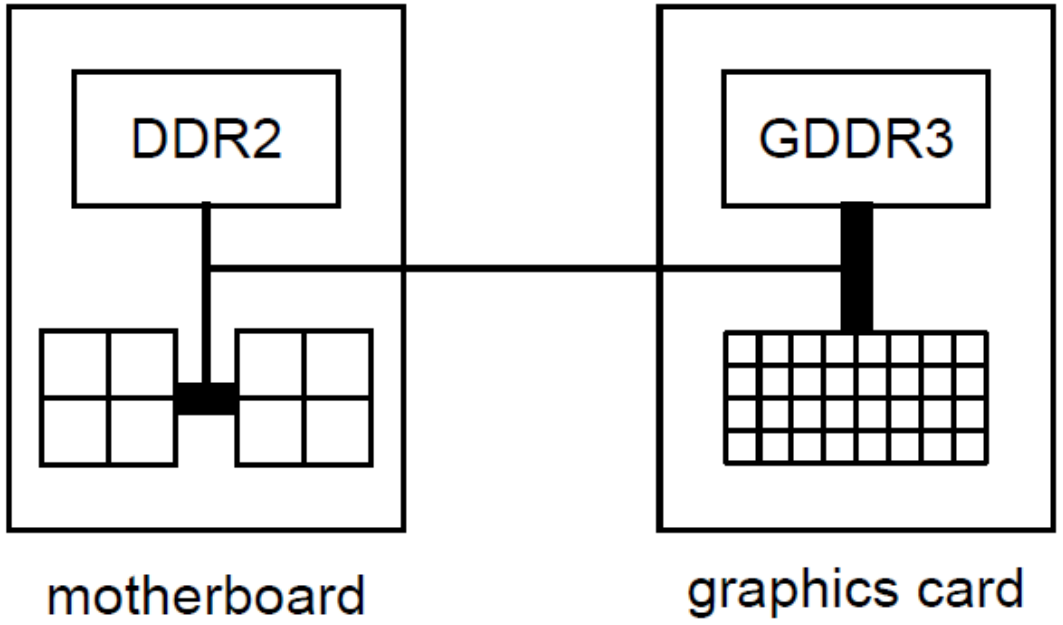
- **Los hilos manejados por una GPU son muy ligeros**
  - Se necesita muy poco tiempo para crear/destruir hilos
- **La GPU necesita más de 1000 hilos para ser eficiente**
  - Una CPU multi-núcleo solo necesita unos pocos hilos.

Comparación CPU y GPU

- **Intel Core 2 / Xeon / i7**
  - 4-6 núcleos MIMD
  - Pocos registros, cache multi-nivel
  - **10-30 GB/s** ancho de banda hacia la memoria principal

- **NVIDIA GTX480**
  - 512 núcleos, organizados en 16 unidades SM cada una con 32 núcleos.
  - Muchos registros, inclusión cachés nivel 1 y 2.
  - 5 GB/s ancho de banda hacia el procesador HOST.
  - 180 GB/s ancho de banda memoria tarjeta gráfica.

Hardware: Ubicación GPU



La GPU se sitúa sobre una placa gráfica pci-e dentro de un computador con uno o varios núcleos.

Hardware: GPU compatible CUDA

SIMT: Una instrucción múltiples hilos

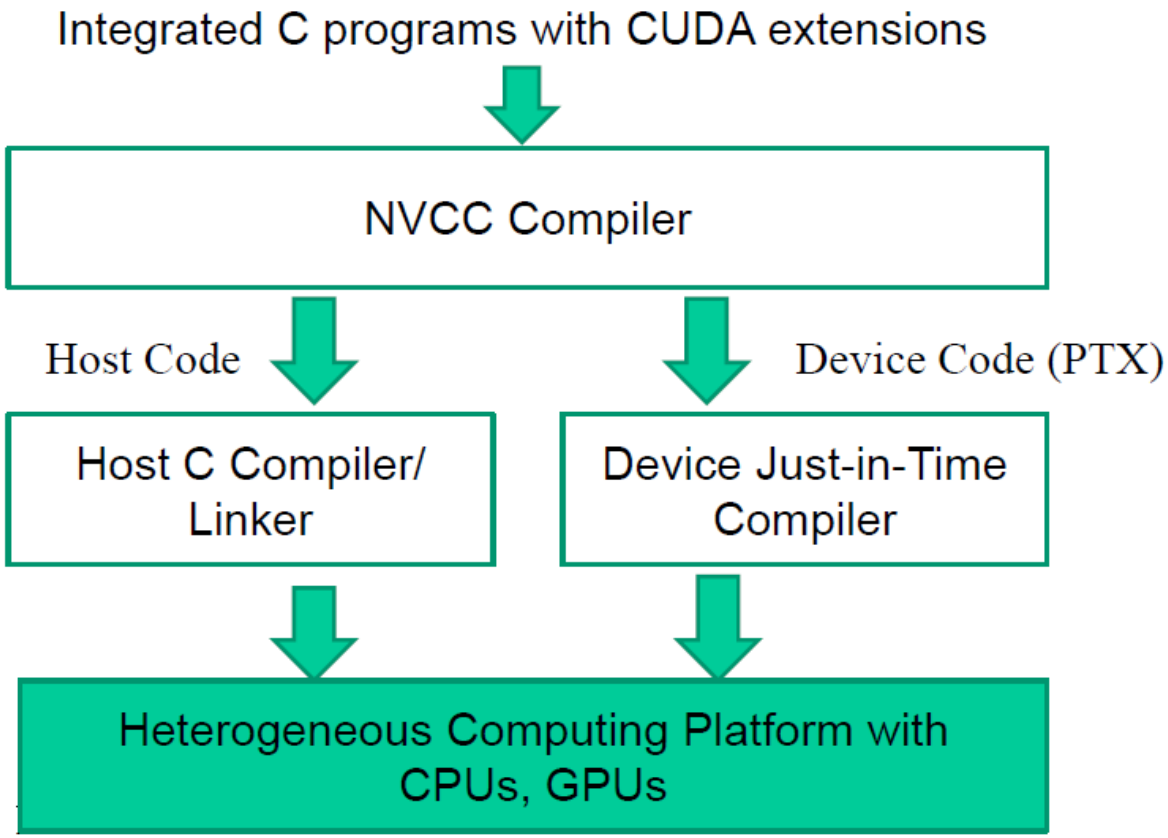
- Característica clave de los núcleos dentro de un SM.
  - Todos los núcleos ejecutan la misma instrucción simultáneamente pero con distintos datos
  - Similar a la computación en los supercomputadores CRAY
  - Mínimo de 32 hilos realizando la misma tarea (casi) al mismo tiempo
  - Técnica tradicional en el procesamiento gráfico y en muchas aplicaciones científicas

Software: CUDA

- **CUDA (Compute Unified Device Architecture)**
  - Hace referencia tanto a un compilador como a un conjunto de herramientas de desarrollo creadas por NVIDIA
  - **Basado en C** con algunas extensiones
  - Soporte C++ , Fortran. Envoltorios para otros lenguajes: *.NET, Python, Java, etcétera*
  - **Gran cantidad de ejemplos** y buena documentación, lo cual reduce la curva de aprendizaje para aquellos con experiencia en lenguajes como OpenMPI y MPI.
  - Extensa comunidad de usuarios en los **foros de NVIDIA**

Software: CUDA

- **Compilando un programa CUDA**



Software: Programación CUDA

- Un programa en cuda tiene dos partes:
  - Código Host en la CPU que hace interfaz con la GPU
  - Código Kernel que se ejecuta sobre la GPU
- En el nivel Device, existen dos APIs
  - Runtime: Simplificada, más sencilla de usar.
  - Driver: más flexible, más compleja de usar.
    - La versión driver no implica más rendimiento, sino más flexibilidad a la hora de trabajar con la GPU

Software: Conceptos básicos

- **Kernel**: es una función la cual al ejecutarse lo hará en N distintos hilos en lugar de en secuencial.
- **Grid**: forma de estructurar los bloques en el kernel
  - Bloques: 1D, 2D
  - Hilos/Bloque: 1D,2D o 3D
- **Bloque**:
  - Agrupación de hilos
  - Cada bloque se ejecuta sobre un solo SM
  - Un SM puede tener asignados varios bloques

Software: Programación CUDA

- Invocación kernel:
  - `kernel_routine<<<gridDim, blockDim>>>(args);`
    - `gridDim` : número de bloques. Tamaño del grid.
    - `blockDim`: número de hilos que se ejecutan dentro de un bloque.
    - `Args`: número limitado de argumentos, normalmente punteros a memoria de la GPU.
  - `gridDim y blockDim` permiten definirse con estructuras que representan 2 y 3 dimensiones para facilitar la programación de algunas aplicaciones.
    - `dim3 dimBlock(256, 256, 1);` `1024 hilos organizados matriz`
    - `dim3 dimGrid(4, 4, 1);` `16 bloques organizados matriz`

Software: Programación CUDA

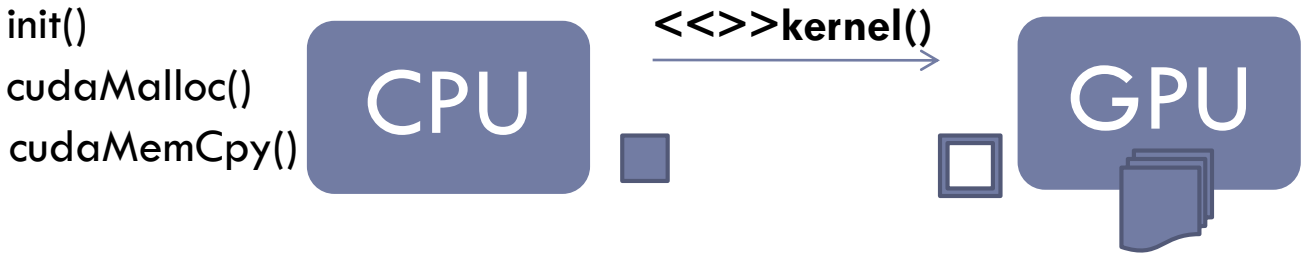
- Una vez ejecutado el kernel, dentro de cada copia del mismo tenemos la siguiente información (propia de cada hilo):
  - Variables pasadas por argumento
  - Punteros a memoria de la GPU
  - Constantes globales en memoria GPU
  - Variables especiales:
    - `gridDim`: tamaño o dimensión de la malla de bloques
    - `blockIdx`: índice del bloque (propio de cada bloque)
    - `blockDim`: tamaño o dimensión de cada bloque
    - `threadIdx`: índice del hilo (propio de cada hilo)

Software: Programación CUDA

- Si utilizamos conjuntos de hilos y bloques definidos en 2 y 3 dimensiones accederemos a sus índices de la siguiente forma:
  - `blockDim.x, blockDim.y`
  - `threadIdx.x, threadIdx.y`
- Existen tipos de datos especiales:
  - `Dim3 nthreads(16,4)`
- Dim3 es un tipo especial de CUDA con 3 componentes x,y,z por defecto inicializas a 1.

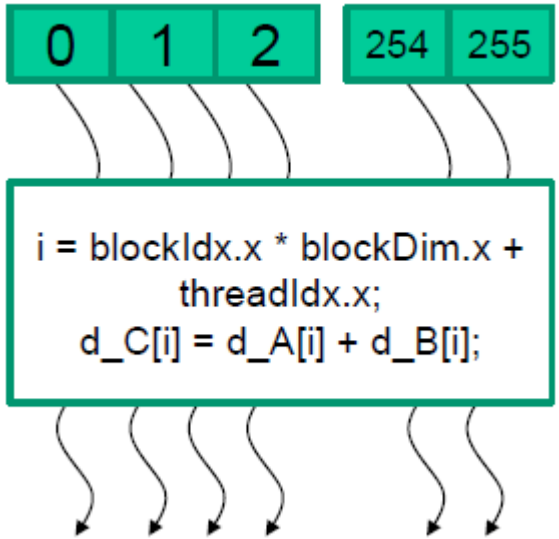
Software: Flujo de ejecución

- En el nivel más alto encontramos un proceso sobre la CPU (Host) que realiza los siguientes pasos:
  1. Inicializa GPU
  2. Reserva memoria en la parte host y device
  3. Copia datos desde el host hacia la memoria device
  4. Lanza la ejecución de múltiples copias del kernel
  5. Copia datos desde la memoria device al host
  6. Se repiten los pasos 3-5 tantas veces como sea necesario
  7. Libera memoria y finaliza la ejecución proceso maestro

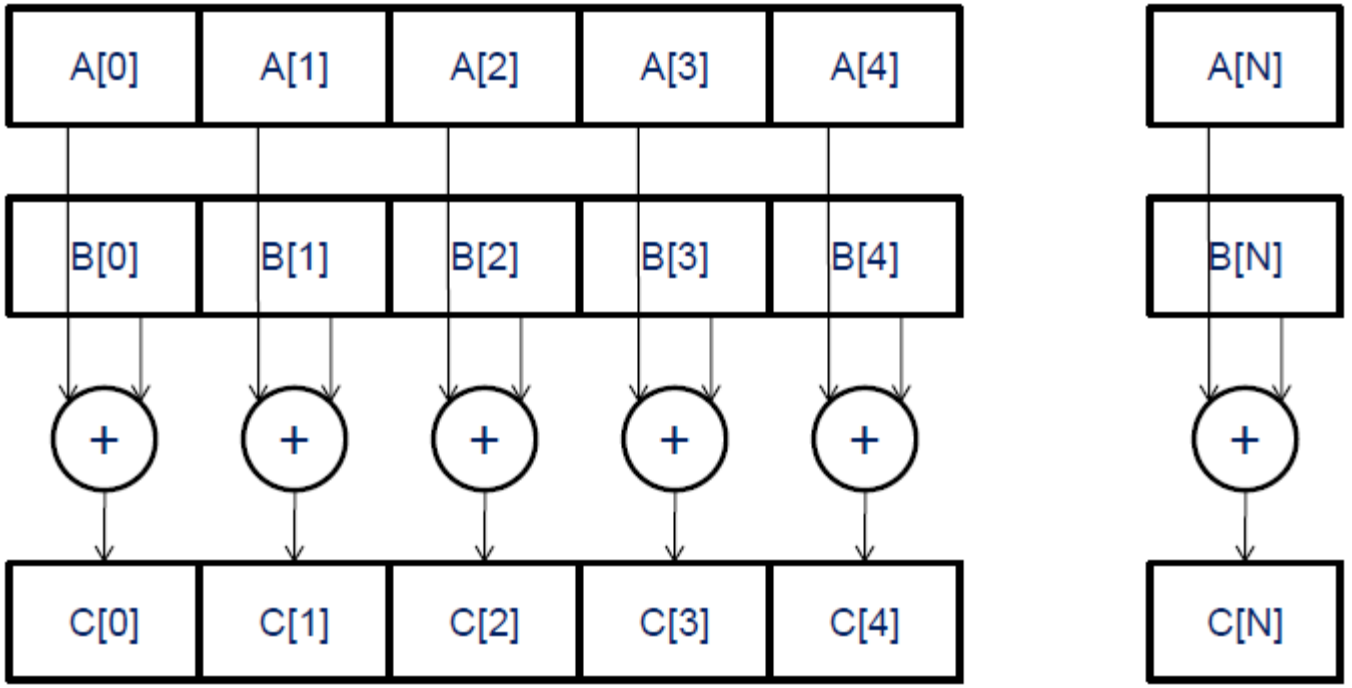


Ejecución kernel

- Un kernel CUDA se ejecuta sobre un grid de hilos:
  - Todos los threads del grid ejecutan el mismo código
  - Cada hilo tiene sus índices y lo utiliza para direccionar la memoria y realizar su lógica



Ejemplo: Suma de vectores



Ejemplo: Suma de vectores

Código tradicional en la CPU

```
// Compute vector sum C = A+B
void vecAdd(float* A, float* B, float* C, int n)
{
  for (i = 0, i < n, i++)
    C[i] = A[i] + B[i];
}
```

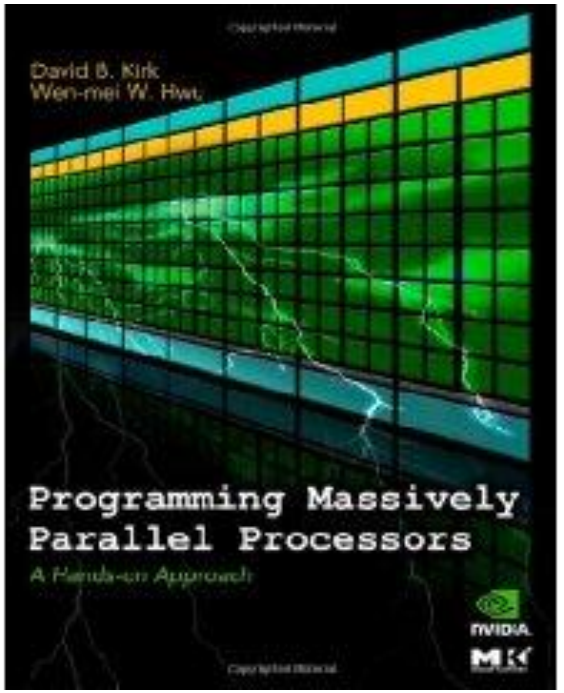
Código paralelizado GPU

```
__global__
void vecAddkernel(float* d_A, float* d_B, float* d_C, int n)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<n) d_C[i] = d_A[i] + d_B[i];
}
```

Bibliografía

- Programming Massively Parallel Processors: A Hands-on Approach
  - By David Kirk and Wen-mei Hwu

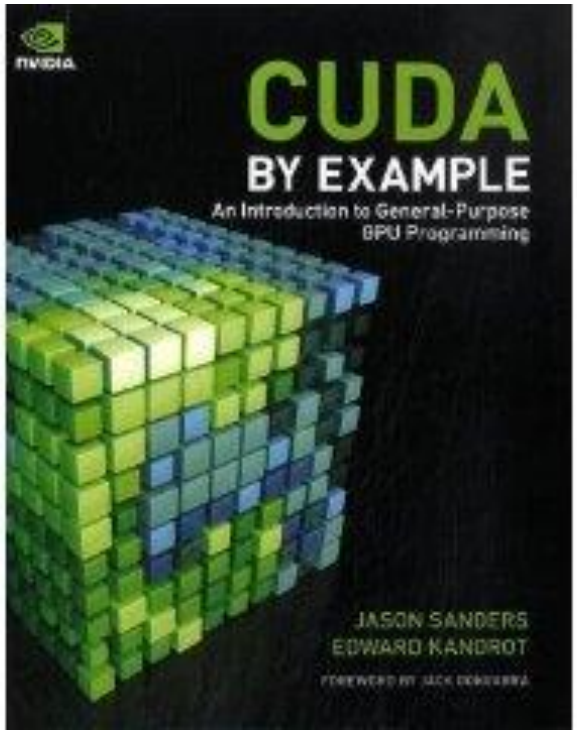
- Libro indispensable para aprender CUDA
- Escrito por gente de NVIDIA
- Conceptos básicos y avanzados de CUDA



Bibliografía

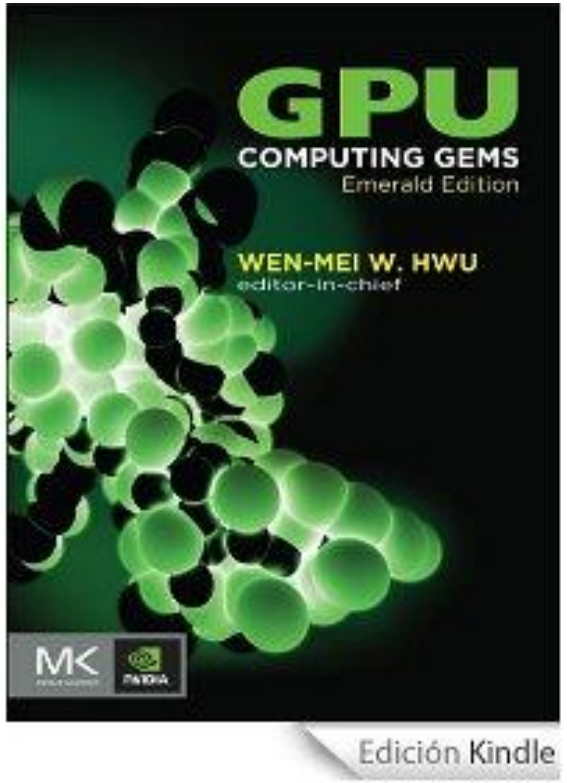
- CUDA by Example: An Introduction to General-Purpose GPU Programming
  - Jason Sanders, Edward Kandrot

- Aprendizaje basado en ejemplos



Bibliografía

- GPU Computing GEMS
  - Wen-mei W. Hwu
- Recopilación de los mejores artículos sobre la tecnología CUDA.
- Aplicaciones sobre distintos campos:
  - Física
  - Visión artificial
  - Librerías
  - Álgebra lineal aplicada



¿Preguntas?