Grupo de Ing. Electrónica aplicada a Espacios INteligentes y TRAnsporte – Área Audio-Visual

"PROGRAMACIÓN AVANZADA DE GPUs PARA APLICACIONES CIENTÍFICAS"

Torrevieja (Alicante) Del 19 al 22 de Julio

Álvaro Marcos
Jose Velasco

Raquel Jalvo
David Jiménez

Índice

Evolución histórica

1. *Ordenadores sin tarjeta gráfica*: alto coste para renderizar en pantalla.
2. *Aparición de GPUs* para liberar a la CPU de la representación en pantalla.
3. Las GPUs entran en la *industria del videojuego* → rápido desarrollo (NVIDIA y ATI).
4. En **2004** surge la idea de utilizar *GPUs para HPC* (se usa OpenGL, DirectX...)
5. NVIDIA ve cuota de mercado en el HPC y crea *CUDA* en **2007** (lenguaje específico que aprovecha las altas prestaciones de la GPU).

Motivaciones

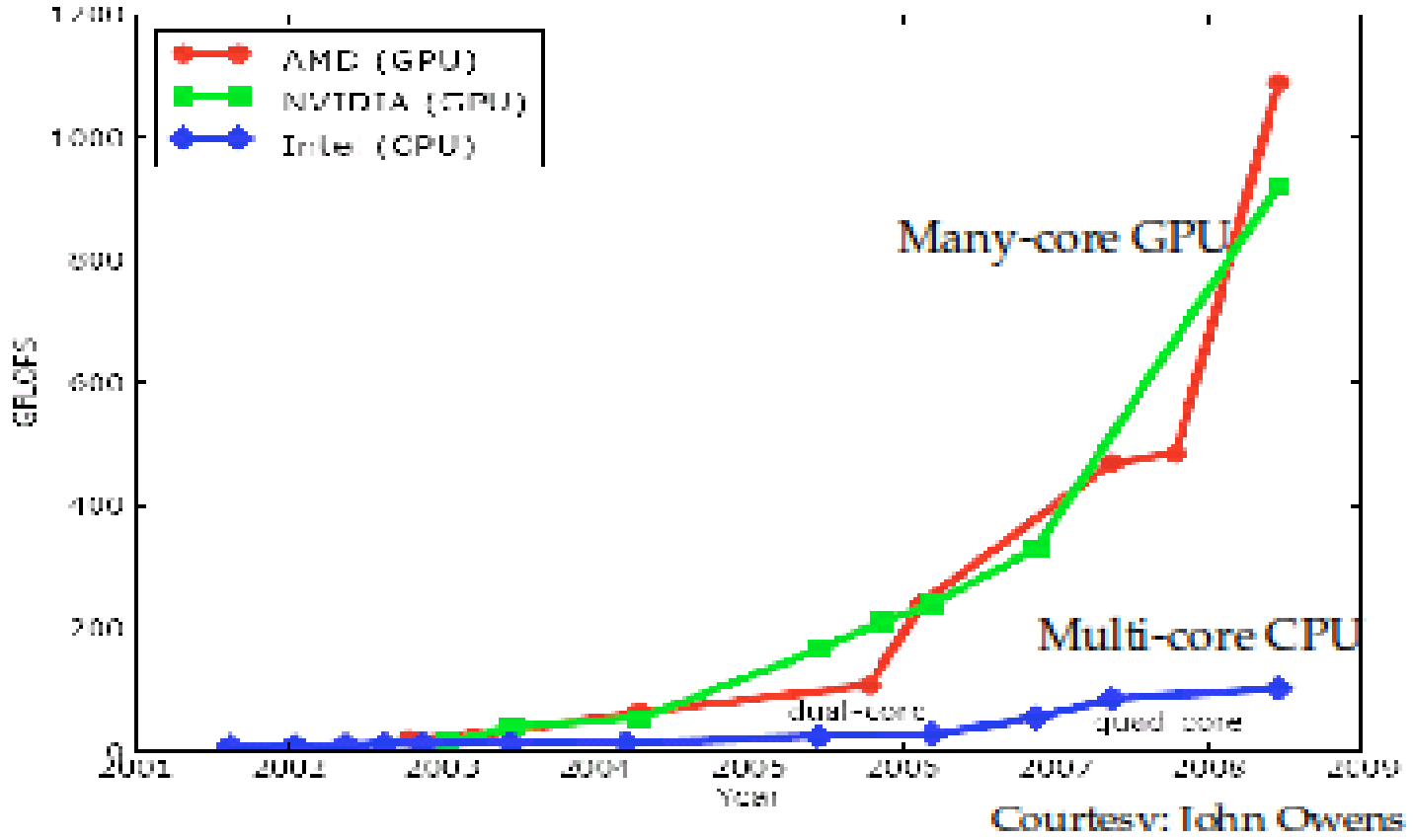
- Análisis de coste vs. eficiencia:

	GFLOPS	Coste	Coste/GFLOPS
CPU	10	800 €	80 €
Super Computador *(“Ben Arabí”)*	14e3	5 M€	357 €
GPU (*)	2e3	300 €	6.67 €

(*) Solo si la aplicación es paralelizable.

Motivaciones

- Las GPUs lideran la lucha del FLOP.
- El mercado del videojuego permite la inversión en I+D.
- Alto rendimiento con costes muy reducidos.



Motivaciones

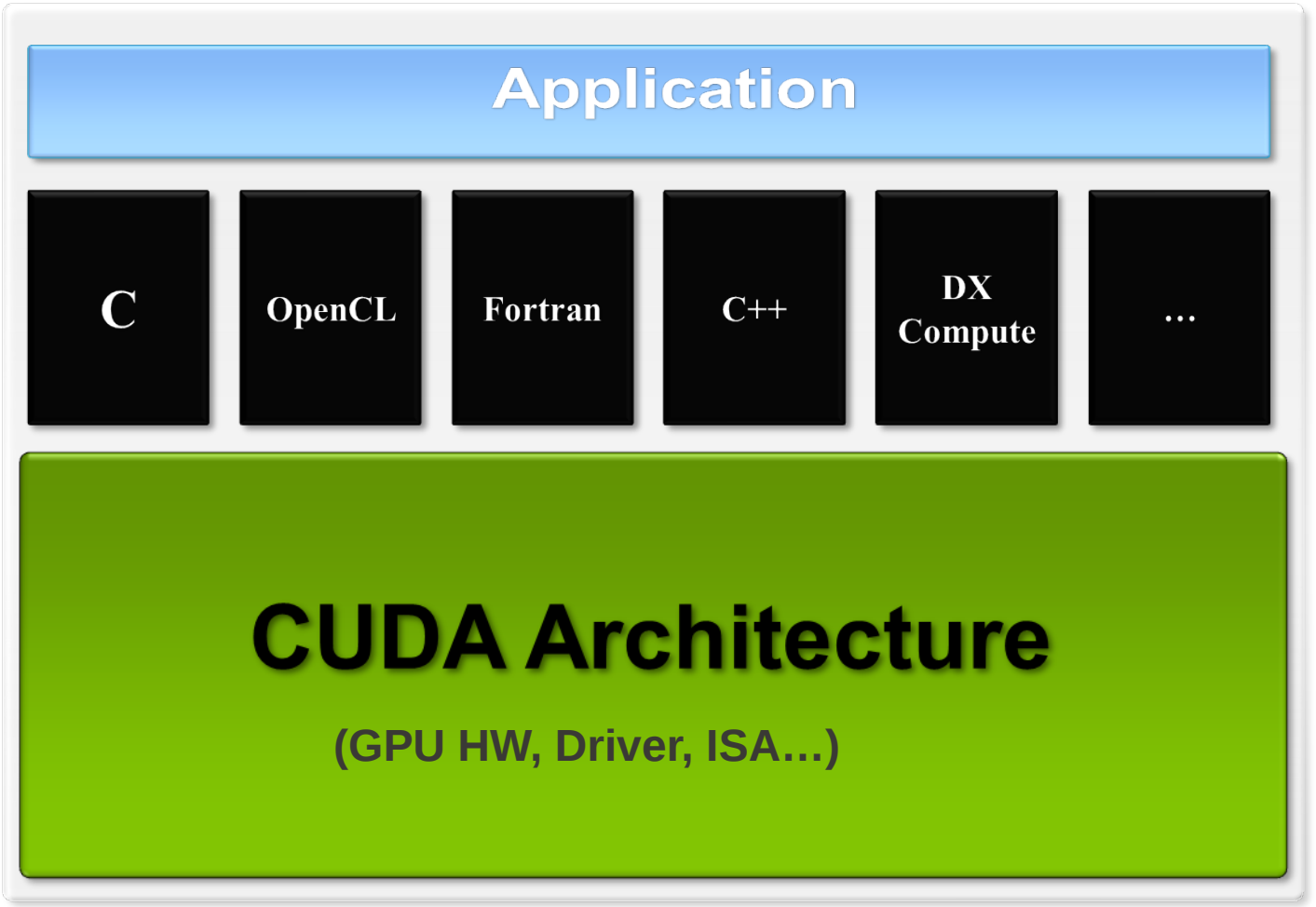
- **¿ Por qué esa gran diferencia en rendimiento de cómputo?**
  - Diferencias en la filosofía de diseño:
    - CPU optimizado para código secuencial.
      - *Hasta 4 cores muy complejos.*
      - *Unidad de control: ejecución en paralelo, fuera de orden, etc → Pero resultado secuencial.*
      - *Más memoria caché.*
    - GPU optimizado para videojuegos
      - *Hasta 512 cores muy sencillos.*
      - *Gran número de operaciones aritméticas*
      - *Ingente número de hilos. Unos esperan (latencias, etc) y otros ejecutan.*

Motivaciones

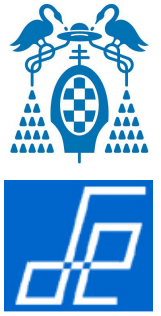
- **Conclusión**: GPUs diseñadas para cálculos numéricos.
- GPUs no son CPUs, pero son compatibles.
- Computación híbrida CPU + GPU → **CUDA**.
  - CPU partes secuenciales.
  - GPU partes intensivas en cómputo paralelo.
    - Operaciones matriciales (2D y 3D)

Motivaciones

- Varias posibilidades de lenguajes.



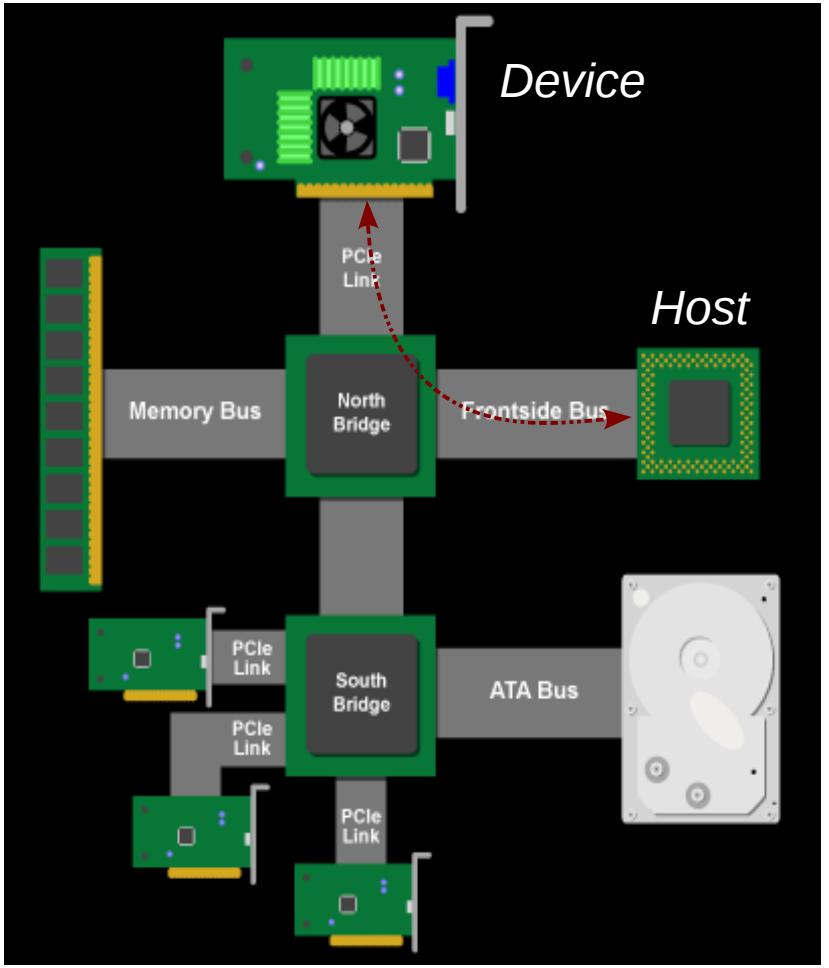
Paradigma del Paralelismo



- **Filosofía SPMD** (*Single Program Multiple Data)*
  - Tipos:
    - Memoria compartida (Hilos)
    - Memoria distribuida (Master & Slave)
  - Los núcleos son más complejos → CPU
- **Filosofía SIMD** *(Single Instruction Multiple Data)*
  - Todos los hilos ejecutan la misma instrucción
  - La instrucción debe ser muy sencilla
  - Cantidad ingente de hilos (hasta 65536 en la G80) en arquitecturas many-cores.

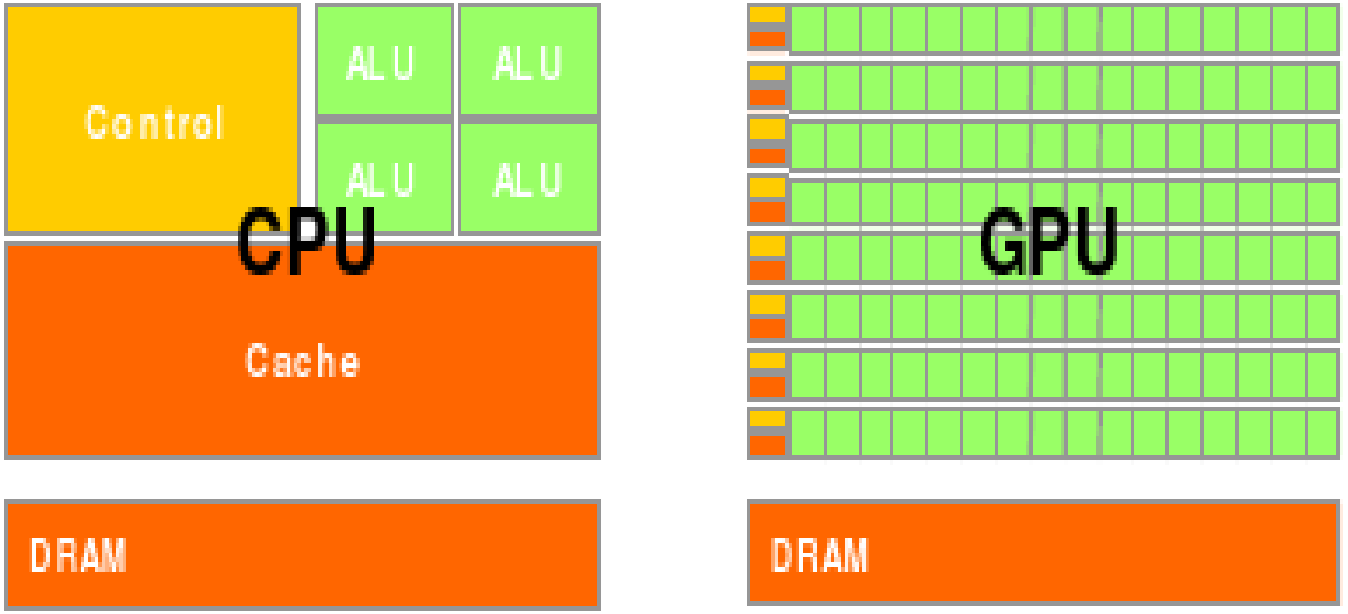
Arquitectura Hardware

- Arquitectura del PC
  - El Northbridge conecta componentes que se deben comunicar a alta velocidad (CPU, DRAM, video)
  - PCIe une la GPU al sistema
  - Aquí reside el cuello de botella para altas prestaciones:
    - *Host* ↔ *Device*



Arquitectura HW. CPU vs. GPU

- **CPU (multi-core)**: pocos núcleos – instrucciones complejas
- **GPU (many-core)**: muchos núcleos – instrucciones sencillas



Arquitectura HW. La GPU

- Arquitectura many-cores (240 cores simples en la G80)
- 1 core es 1 **SP**. La agrupación de 8 SPs es 1 **SM**.
- 1 SM comparte memoria privada, caché y unidad de control.

Arquitectura HW. Tipos de memoria

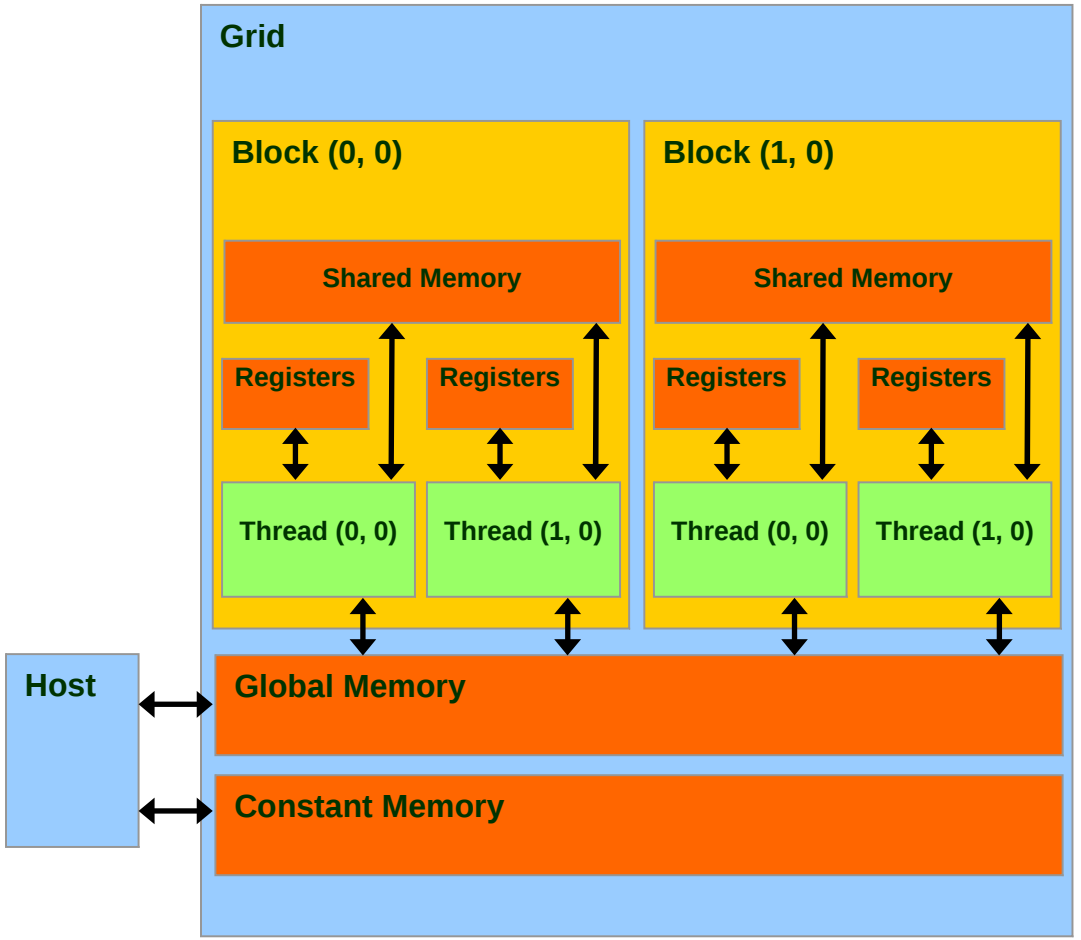
- En función de la latencia de acceso:

  - **Global Memory** → accesible en R/W por todos los hilos y por la CPU.
  - **Local Memory** → parte de la memoria global, privada para cada hilo (memoria virtual).
  - **Constant Memory** → global de solo lectura. Puede cargarse en caché de SM para acelerar las transferencias.
  - **Shared Memory** → de tamaño limitado (16KB) (impuesto por los videojuegos). Accesible en R/W por los hilos de un mismo SM.
  - **Registro** → mismo número para todos los hilos, accesible en R/W de forma privada.

- *“No existe asignación de memoria dinámica a través de la GPU”.*

Arquitectura HW. Tipos de memoria

- Distribución de memorias en la GPU



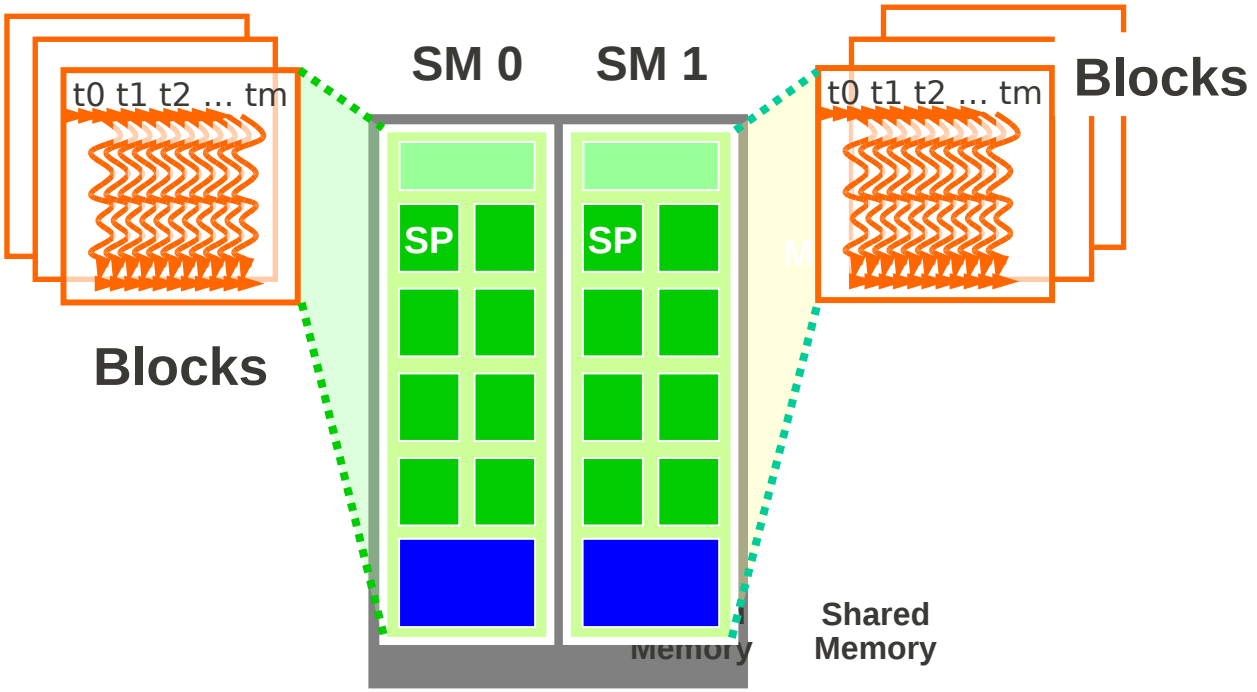
Arquitectura Software

- Cambio de mentalidad de secuencial a programación en **paralelo**.
- Máximo **aprovechamiento** de la GPU:
  - *Mayor número de hilos* trabajando de forma simultánea.
  - *Mayor uso de memoria local* (shared memory y registros) y *menor uso de memoria compartida* (global memory).
- **Comparativa** de eficiencia:
  - Un η=*80%* en la *CPU* → INSUFICIENTE!
  - Un η=*20%* en la *GPU* → LOGRO!
    - ***Cuidado**: la información viene dada en rendimientos pico-teóricos.*

"Es fácil programar en CUDA, pero es difícil conseguir rendimientos elevados".

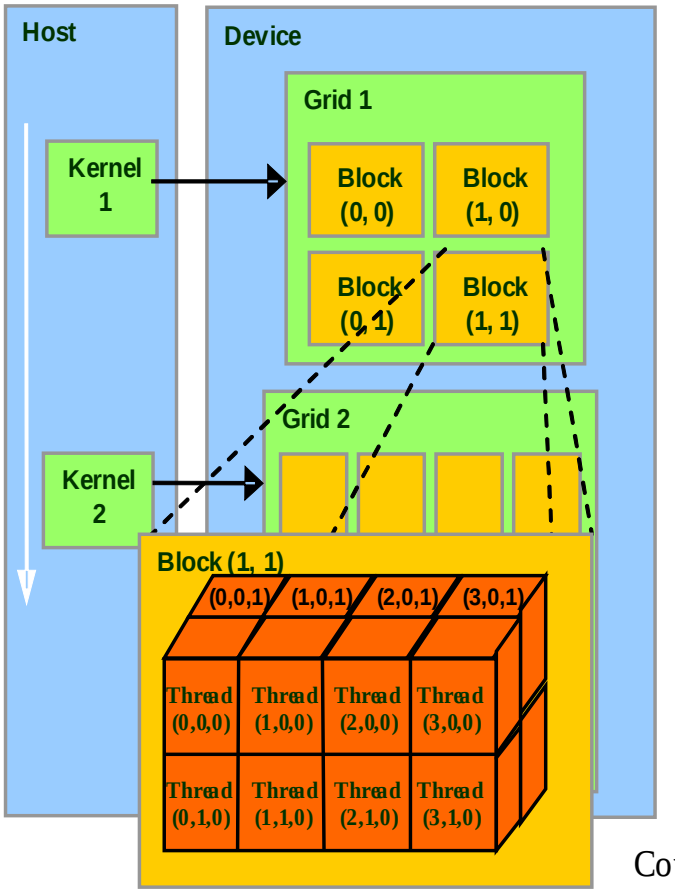
Arquitectura SW. Conceptos básicos

- **Bloque:** agrupación fija de hilos definida a priori.
  - Cada bloque se ejecuta en un solo SM
  - Un SM puede tener asignados varios bloques.



Arquitectura SW. Conceptos básicos

- **Kernel**: función invocada desde la CPU que se ejecuta en la GPU.
  - Solo 1 kernel de manera simultánea por CPU.
- **Grid**: forma de estructurar los bloques en el kernel.
  - Bloques: 1D, 2D
  - Hilos/bloque: 1D, 2D o 3D



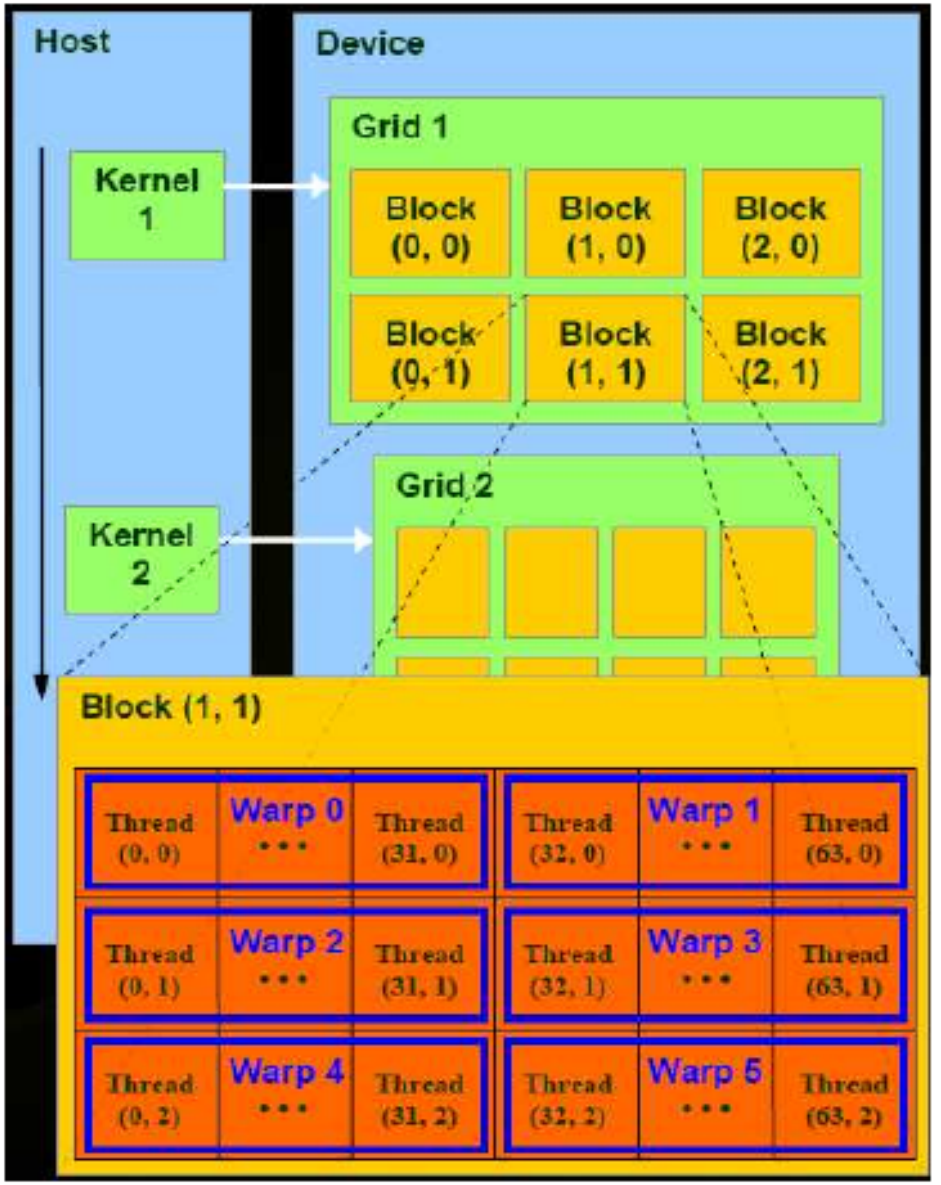
Courtesy: NDVIA

Arquitectura SW. Concepto de warp

- Todos los hilos de un mismo bloque se agrupan en conjuntos de 32 → "**warp**".
- Los hilos de un warp se planifican en los 8 cores de un SM para aprovechar el pipeline de 4 etapas.
  - *Resultado*: ejecución paralela de 32 hilos en el mismo instante de tiempo → SIMD
- *Ejemplo:*
  - 1 bloque de 256 hilos se divide en 256/32=8 warps que son planificados para la mejorar la eficiencia.

Arquitectura SW. Modelo de ejecución

- Cada bloque activo se divide en warps.
- Los hilos de un warp son ejecutados *físicamente en paralelo.*
- Warps y bloques se *planifican* en el SM.



Arquitectura SW. Concepto de tiling

- La *global memory* es mucho más lenta que la shared memory.
- **Estrategia "tiling"**: técnica de aprovechamiento de los recursos de la shared memory (16KB).
  - Se divide el problema en bloques de computación con subconjuntos que quepan en shared memory.
  - Se maneja a nivel de bloque de hilos:
    1. Carga de un subconjunto de global memory a shared memory, usando varios hilos para aprovechar el paralelismo a nivel de memoria.
    2. Se realizan las operaciones sobre cada subconjunto en shared memory.
    3. Se copian los resultados desde shared memory a global memory

Arquitectura SW. Warps divergentes

- **Concepto de divergencia**: hilos de un mismo warp ejecutan instrucciones diferentes.
  - La ejecución se serializa (*if/else*).
  - Pérdida de eficiencia.
- **Solución**: todos los hilos de un mismo warp deben ejecutar el mismo juego de instrucciones.
  - Los hilos se agrupan en warps que ejecuten la misma instrucción.
- Ejemplo con divergencia:
  - *If (threadIdx.x > 2) { }*
- Ejemplo sin divergencias:
  - *If (threadIdx.x / WARP_SIZE > 2) { }*

Ejemplos prácticos

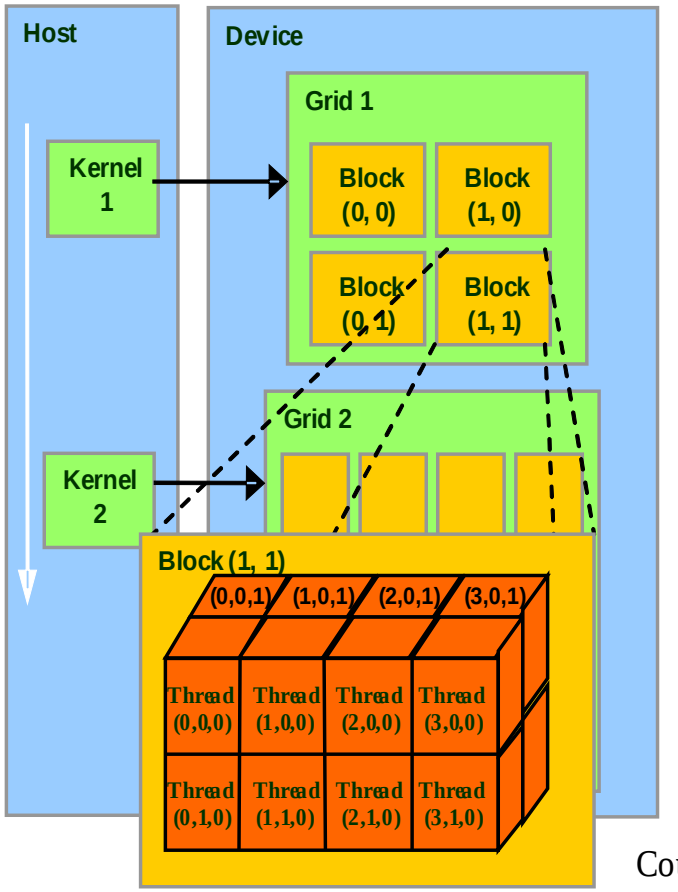
- Modelo de Programación
  - Agentes
  - Niveles Jerárquicos
  - Extensiones C
- Ejemplo de Interfaz de Programación
  - Suma de Vectores
  - Multiplicación de Matrices (con Tiling)

Modelo de programación: Agentes

- Host (CPU)
  - Comportamiento Secuencial.
  - Ejecuta pocos hilos pero muy complejos
  - Controla al Device
- Device (GPU):
  - Actúa en paralelo a la CPU
  - Posee su propio Hardware (Memoria, procesadores, etc..)
  - Las partes de datos paralelas de una aplicación son expresadas como kernels (SPMD)
  - Un kernel ejecuta muchos hilos a la vez

Modelo de Programación: Niveles Jerárquicos en CUDA

- Hilos
  - Arrays de hilos
  - SPMD
  - Identificadores de Hilo
- Bloques
  - Un array de hilos está dividido en múltiples bloques
  - Los hilos de un bloque comparten ciertos recursos
- GRID
  - Varios Bloques conforman un GRID



Courtesy: NDVIA

Modelo de Programación: Indexación

- Se puede identificar un bloque y un hilo concreto dentro del grid:
  - *blockIdx.{x,y}*
  - *threadIdx.{x,y,z}*
- Empleado para acceder a la porción de memoria relativa a cada hilo.
- Ejemplo:
  - Acceso a una posición de un vector:

    *indice = blockIdx.x * block_size + threadIdx.x;*

    - block_size → número de hilos por bloque.

Modelo de Programación: Extensiones C para CUDA

- Prefijos
- Palabras Clave:
- Funciones Intrinsecas:
- Runtime API:
  - *Memoria*
  - *Transferencia de datos*
  - Lanzamiento del kernel
  - Tiling

```
__device__ float filter[N];

__global__ void convolve (float *image)  {

  __shared__ float region[M];
  ...

  region[threadIdx] = image[i];

  __syncthreads()
  ...

  image[j] = result;
}

// Allocate GPU memory
void *myimage = cudaMalloc(bytes)

// 100 blocks, 10 threads per block
convolve<<<100, 10>>> (myimage);
```

Extensiones C para CUDA: Prefijos

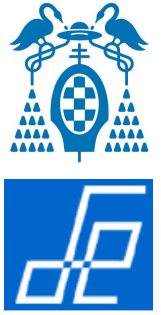
	Ejecutado en el:	Sólo se puede llamar desde:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

- __global__ define una función como kernel
- Cada "__" son 2 barras bajas
- Un kernel devuelve void
- __device__ y __host__ pueden usarse juntas

Extensiones C para CUDA Palabras Claves

- Indican con que datos tiene que trabajar con cada hilo.
- dim3 gridDim;
  - Dimensiones del grid en bloques (gridDim.z)
- dim3 blockDim;
  - Dimensionds del bloque en threads
- dim3 BlockIdx
  - Permite relacionar un bloque con sus datos
  - 1 ó 2 Dimensiones
- dim3 ThreadIdx
  - Permite relacionar un hilo con sus datos
  - 1, 2 ó 3 Dimensiones
- El número de dimensiones simplifican el acceso a memoria

Extensiones C para CUDA Funciones Intrínsecas



- Son ejecutadas en la GPU
  - Funciones matemáticas: `__pow`, `__log`, `__log2`, `__log10`, `__exp`, `__sin`, `__cos`, `__tan`
    - Tienen menos precisión (float) pero son mucho más rápidas
  - Otras Funciones: `__syncthreads`
    - Sincroniza todos los hilos de un bloque.

Extensiones C para CUDA Runtime API

- Ejecutadas en el Host.
- Controlan la ejecución de la GPU.
  - Operaciones con Memoria
  - Transferencia de datos
  - Llamadas al Kernel
  - Manejo de errores

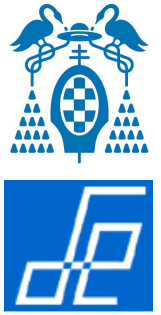
Runtime API
Operaciones con memoria

- La memoria global es visible para todos los hilos GPU. Es la que se comunica con el Host.
- `CudaMalloc()`
  - Reserva memoria global de la GPU.
- `CudaFree()`
  - Libera memoria global.
- OJO! No se puede reasignar memoria durante la ejecución del Kernel.

Runtime API
Transferencia de datos

- `CudaMemcpy()`
  - Transferenecia de datos
    - Host-Host
    - Host-Device
    - Device-Host
    - Device-Device
  - Es una instrucción bloqueante .

Runtime API
Llamadas al Kernel



- Son asíncronas
- Pero existen funciones de sincronización

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

Ejemplo Interfaz de Programación Suma de Vectores

```
int main()                  //CODIGO HOST
{
    // Reserva e inicializa memoria en la CPU
    float *h_A = …,   *h_B = …;
    // Reserva memoria en la GPU
    float *d_A, *d_B, *d_C;
    cudaMalloc( (void**) &d_A, N * sizeof(float));
    cudaMalloc( (void**) &d_B, N * sizeof(float));
    cudaMalloc( (void**) &d_C, N * sizeof(float));

    // copia host memory en device
    cudaMemcpy(d_A, h_A, N * sizeof(float), cudaMemcpyHostToDevice) );
    cudaMemcpy(d_B, h_B, N * sizeof(float), cudaMemcpyHostToDevice) );

    // Ejecuta el kernel en ceil(N/256) blocks de 256 threads cada uno
    vecAdd<<<ceil(N/256), 256>>>(d_A, d_B, d_C, n);

    cudaMemcpy(h_C, d_C, N * sizeof(float), cudaMemcpyDeviceToHost) );

    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```

Ejemplo Interfaz de Programación Suma de Vectores

```
// Calcula la suma de vectores C = A+B
// Cada hilo hace la suma de sus elementos
__global__
void vecAdd(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];
}
```

Ejemplo Interfaz de Programación
Multiplicación de matrices usando tiling

Configuracion del Kernel en el Host:

```
// Bloques del grid e hilos por bloque
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);
dim3 dimGrid(Width/TILE_WIDTH,Width/TILE_WIDTH);


// Lanzar la computación en la GPU
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd,
Width);
```

Ejemplo Interfaz de Programación
Multiplicación de matrices usando tiling

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width) {
1.  __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
2.  __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
3.  int bx = blockIdx.x;  int by = blockIdx.y;
4.  int tx = threadIdx.x; int ty = threadIdx.y;
// Identificamos la columna y la fila de Pd para trabajar
5.  int Row = by * TILE_WIDTH + ty;
6.  int Col = bx * TILE_WIDTH + tx;
7.  float Pvalue = 0;
// Bucle sobre los tiles de Md y Nd para calcular Pd
8.   for (int m = 0; m < Width/TILE_WIDTH; ++m) {
// Lectura colaborativa de los tiles Md y Nd en shared memory
9.      Mds[tx][ty] = Md[(m*TILE_WIDTH + tx)*Width+Row];
        Nds[tx][ty] = Nd[Col*Width+(m*TILE_WIDTH + ty)];
        __syncthreads();
12.     for (int k = 0; k < TILE_WIDTH; ++k)
13.         Pvalue += Mds[tx][k] * Nds[k][ty];
14.    __synchthreads();
15.  }
16.   Pd[Row*Width+Col] = Pvalue;
}
```

CUDA orientado al Ispace

- Pensado para el cálculo matricial.
  - Tanto 2D como 3D.
- La precisión es suficiente para las aplicaciones de audio y vídeo:
  - Se estiman mejoras en un futuro cercano.
- Coste de evaluación reducido.
- Existen plugins para Matlab.

Vídeos

- Reconstrucción 3D mediante par estéreo.
- Simulación y renderizado en tiempo real de dinámica de fluidos y partículas.
- Cálculo de flujo óptico denso en tiempo real.
- Mapa de distancias en superficies parametrizadas.

Conclusiones

- Limitaciones
  - Precisión del float
  - Cuellos de botella
    - Comunicaciones Host-Device
    - Comunicación con la global memory
- Tarjetas que admiten CUDA
  - Familias G80, GT200y GF100. *http://www.nvidia.es/object/cuda_gpus_es.html*
- Conclusiones finales
  - La curva de esfuerzo desciende exponencialmente con el tiempo.
  - “Es un avión con depósito de Vespino”.

El futuro de CUDA

- Corto plazo:
  - Aparición de nuevas arquitecturas:
    - *AMD* → Fusion: integra GPU y CPU en un mismo chip compartiendo memoria.
    - *Intel* → Larrabee: futuro incierto, basado en arquitectura X86 (realmente propósito general)
    - *Nvidia* → Fermi: evolución de Tesla.
  - **OpenCL**: estándar abierto basado en CUDA

- Largo plazo:
  - Mayor integración
  - “CUDA desaparecerá pero el paralelismo de datos NO”

Preguntas.