

# GPU-Ejemplo CUDA

---

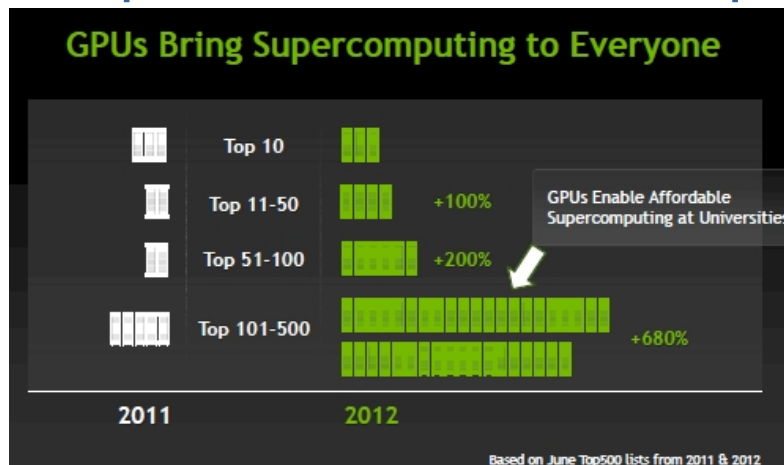
**Carlos García Sánchez**

# Contenidos

- Motivación
- GPU vs. CPU
- GPU vs. Vectoriales
- CUDA
  - Sintaxis
  - Ejemplo

# Motivación

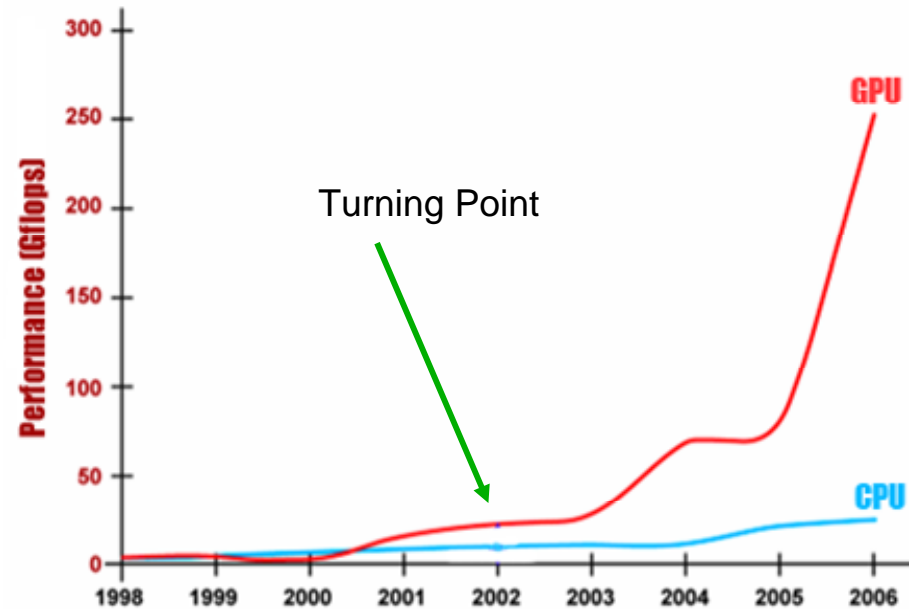
- Computación altas prestaciones: [www.top500.org](http://www.top500.org)



- 1º: Titan (300mil AMD-Opteron + 19mil NVIDIA K20x)
- Green computing: [www.green500.org](http://www.green500.org)
  - Proyecto Montblanc: <http://www.montblanc-project.eu>
    - Objetivo: entre los mejores MFLOP/Watt
    - Basado en Carma: ARM+CUDA
- Electrónica de consumo (tabletas y móviles):
  - Tegra2, Tegra3 (ARM CortexA9 MP + GPU)

# GPU vs CPU (I)

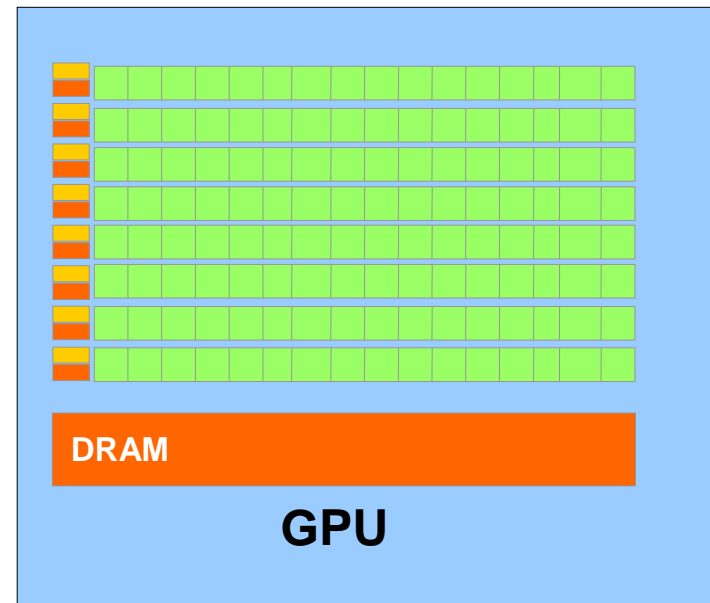
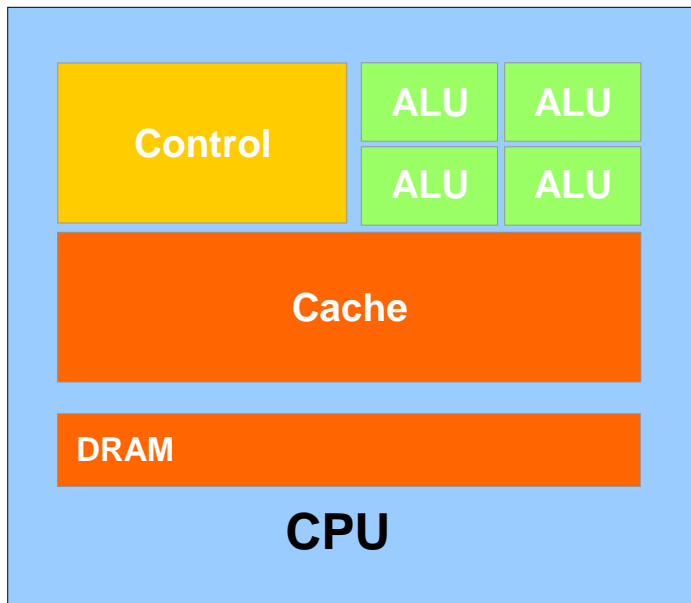
- GPUs son actualmente más potentes
  - 3.0 GHz dual-core2 Duo: 32 GFLOPS
  - NVIDIA G80: 367 GFLOPS
- Acceso a Memoria
  - 1066 MHz FSB Pentium Extreme Edition : 8.5 GB/s
  - ATI Radeon X850 XT Platinum Edition: 37.8 GB/s
- Y la tendencia va a más...



¿Por qué?

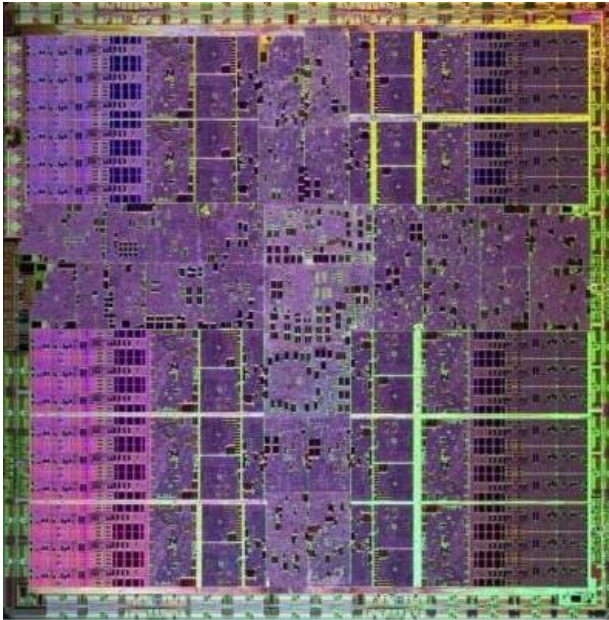
# GPU vs CPU (II)

- Anticipandose a la Era Manycore
  - NVIDIA G80: 128 Cores – Noviembre 2006 –
    - Multiprocesador Especializado pero de Gran Consumo (Millones)
    - 681M trans. en 470 mm<sup>2</sup> (90nm), 1.35 GHz, 518 GFLOPS
    - 1.5 GB DRAM, Interfaz DRAM 384 pines, 76 GB/s
    - PCI Express x16, 170 W max

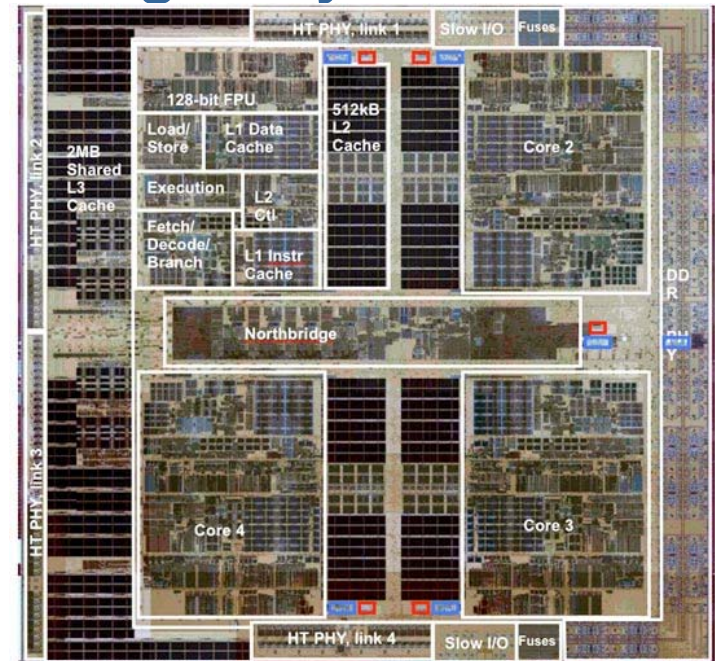


# GPU vs CPU (III)

- GPUs son
  - Relativamente baratas (<600€)
  - Dedicadas a operaciones gráficas
- Emplean los transistores para “lógica” y no a *cache*



NVIDIA T10P (ISC'08) tiene 240 *cores* implementados como "*thread processor*" con unidades enteras+ floats (32/64bits)  
= 500 GFLOPS hasta 1 TFLOPS



AMD-K10 (Quad-Core)  
L1 y L2 replicada por core  
L3 compartida

# GPU vs. Vectoriales (I)

- GPU: procesadores de hilos

- SIMT: Single Instruction Multiple Threads

SIMT thread registers

a[i]	a[i+1]	a[i+2]	a[i+3]
b[i]	b[i+1]	b[i+2]	b[i+3]
a	a	a	a
b	b	b	b
i	i+1	i+2	i+3
...	...	...	...

$a = b + c$

- Analogía procesadores vectoriales

- SIMD: Single Instruction Multiple Data

SIMD vector registers

a[i], a[i+1], a[i+2], a[i+3]
b[i], b[i+1], b[i+2], b[i+3]
...

Scalar registers

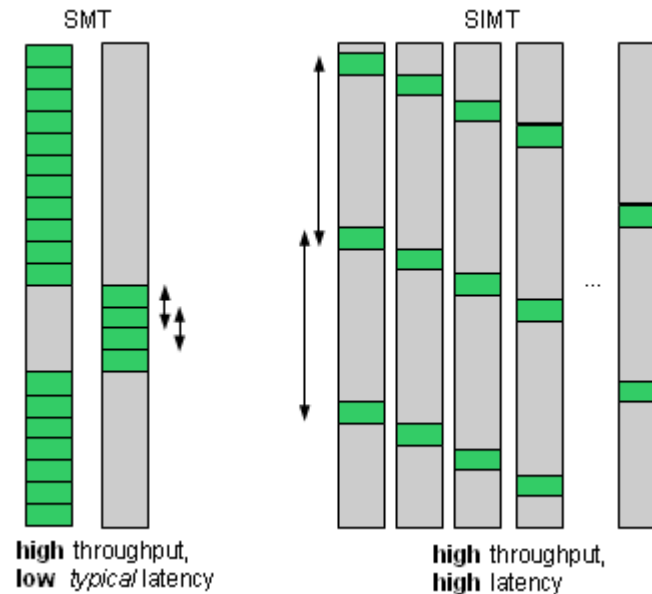
a
b
i
...

```
__global__ void add(float *a, float *b, float *c)
{ int i = blockIdx.x * blockDim.x + threadIdx.x;
  a[i]=b[i]+c[i]; //no loop!
}
```

$a[0:63] = b[0:63] + c[0:63]$

# GPU vs. Vectoriales (II)

- GPU: procesadores de hilos
  - SIMT: Single Instruction Multiple Threads
  - Multiples threads ~ warp (terminología NVIDIA)
  - Muchos threads para ocultar latencias



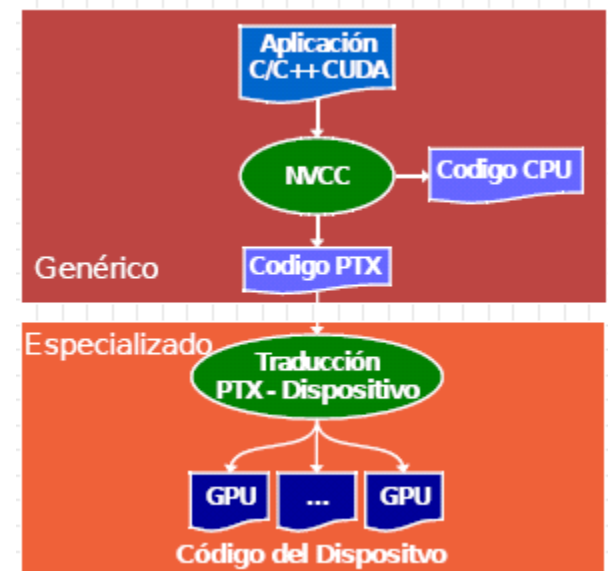
- Muchos registros: 64 registros/thread para evitar register spilling



# CUDA (I)

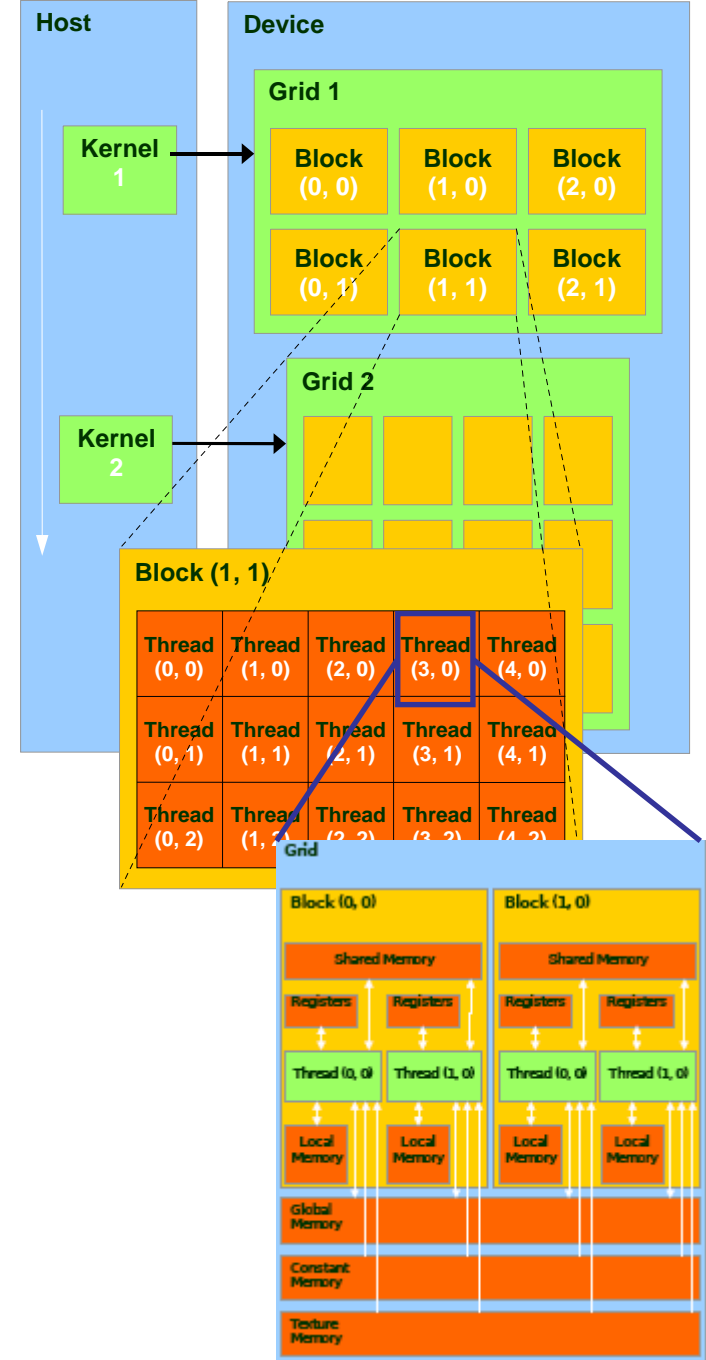
## ■ Características

- Lenguaje C estándar
  - Librerías numéricas estándar
    - cuFFT (Fast Fourier Transform)
    - cuBLAS (Basic Linear Algebra Subroutines).
    - Algebra (Densa: Magma, Dispersa:cuSparse)
  - El controlador de CUDA interacciona con los controladores de gráficos OpenGL y DirectX.
- ## ■ La GPU vista como un Coprocesador
- Objetivo: Computación Heterogénea Altamente Escalable



# CUDA (II)

- **Single Instruction Multiple Threads**
  - Cores/PE Organizados en Multiprocesadores SIMT
  - Unidad de Planificación *Warp* (scoreboarding)
  - Multithreading → Ocultar Latencia
- Modelo de *threads* y *blocks*
  - Threads Organizados en Grid (1D,2D) de Blocks (1D, 2D, 3D)
    - Kernel<<<GridSize,BlockSize>>> (parametros)
    - Los Threads de un Block se Ejecutan en Mismo Multiprocesador
    - En un Bloque es Posible **Sincronización y Cooperación** Eficiente



# CUDA (III)

## ■ Extensiones de C

- Funciones en GPU como (*global, device, shared, local, constant*)
  - `__device__ float filter[N];`
  - `__global__ void convolve (float *image)`
- Descriptores del bloque/thread
  - `int tx = threadIdx.x;`
- Sincronizaciones
  - `__syncthreads()`
- Memoria
  - `void *myimage = cudaMalloc(bytes)`
  - `cudaFree( ), cudaMemcpy2D( ), ...`
- Invocación del kernel
  - `convolve<<<100, 10>>> (myimage);`

## ■ Jerarquía de memoria (II)

### ■ Cada kernel puede leer:

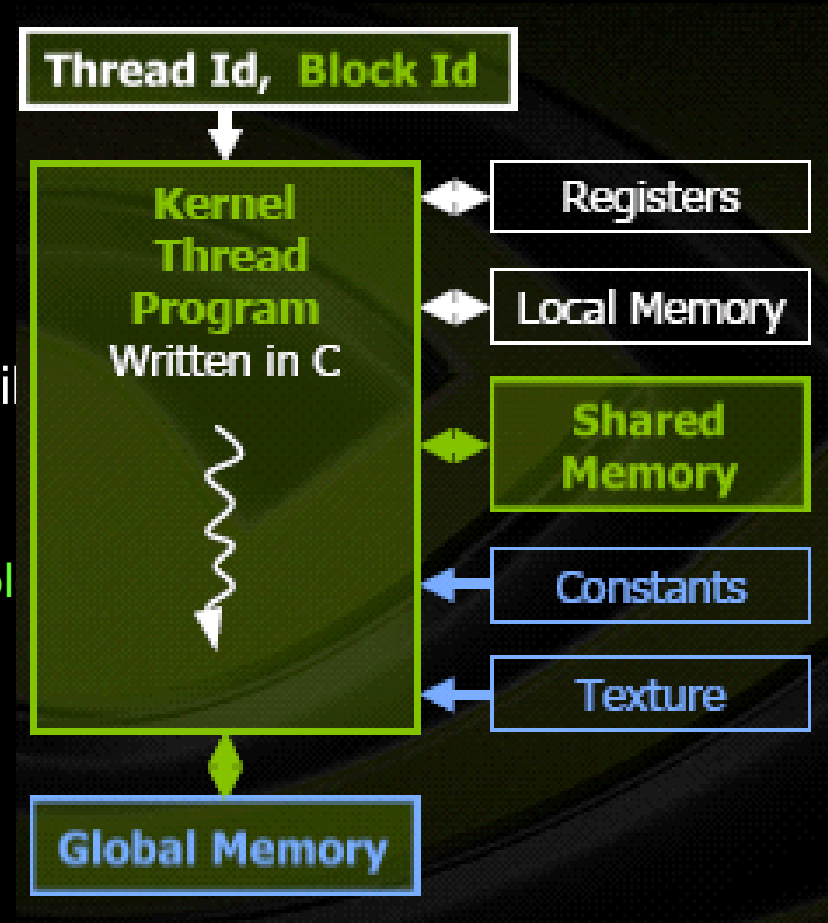
- Thread Id      por thread
- Block Id      por block
- Constants      por grid
- Texture      por grid

### ■ Cada thread puede leer y escribir:

- Registers      por thread
- Local memory      por thread
- Shared memory      por block
- Global memory      por grid

### ■ Host CPU puede leer/escribir:

- Constants      por grid
- Texture      por grid
- Global memory      por grid



# CUDA sintaxis (I)

## ■ Gestión memoria

### ■ `cudaMalloc()`

- Reserva espacio de memoria en el dispositivo (memoria de vídeo)
- Parametros: puntero a esa región memoria y tamaño

### ■ `cudaFree()`

- Libera el espacio de memoria a partir del puntero dado

### ■ `cudaMemcpy()`

- Transfiere datos
- 4 parámetros: puntero fuente/destino, número bytes y dirección transferencia (Host a Host/Host a Device/Device a Host/Device a Device)

# CUDA sintaxis (II)

## ■ Declaración de funciones:

	Se ejecuta en:	Sólo se puede llamar desde:
<code>__device__ float deviceFunc()</code>	device	device
<code>__global__ void KernelFunc()</code>	device	host
<code>__host__ void HostFunc()</code>	host	host

- Las direcciones de las funciones device `__device__` no se puede acceder a ellas desde la CPU → acceder a datos mediante *cudaMemcpy*
- Para las funciones ejecutadas en la GPU:
  - No se permite la recursión
  - No podemos tener variables estáticas
  - Ni un número variable de argumentos

# CUDA sintaxis (III)

- Llamadas a *kernels* en la GPU
  - La función del kernel debe ser invocada con la junto con la configuración de ejecución

```
__global__ void KernelFunc(...);  
dim3 DimGrid(100, 50); // 5000 thread blocks  
dim3 DimBlock(4, 8, 8); // 256 threads per block  
size_t SharedMemBytes = 64; // 64 bytes of shared memory  
...  
KernelFunc<<< DimGrid DimBlock SharedMemBytes DimGrid, DimBlock, >>>(...);
```

# CUDA ejemplo (I)

- Necesario tener instalado:
  - CUDA Toolkit
  - SDK (incluye ejemplos)
  - Drivers
- <https://developer.nvidia.com/cuda-downloads>
- Utilizaremos un ejemplo sencillo: Suma de matrices
  - $a=b+c$
  - Compilación con nvcc: `nvcc -o add main.cu -O3 -lm`



# CUDA ejemplo (II)

- Ejemplo sencillo: suma de matriz punto a punto:  $a = b + c$

## CODIGO C


```
void addMatrix (float *a, float *b,
               float *c, int N)
{
    int i, j, idx;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            idx = i*N + j;
            a[idx] = b[idx] + c[idx];
        }
    }
}

void main()
{
    .....
    addMatrix(a, b, c, N);
}
```

## CODIGO CUDA

```
__global__ void addMatrixG(float *a, float *b,
                          float *c, int N)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    int j = blockIdx.y*blockDim.y + threadIdx.y;
    int idx = i*N + j;
    if (i < N && j < N)
        a[idx] = b[idx] + c[idx];
}

void main()
{
    ...
    dim3 dimBlock (blocksize, blocksize);
    dim3 dimGrid (N/dimBlock.x, N/dimBlock.y);
    addMatrixGPU<<<dimGrid, dimBlock>>>(a, b, c, N);
    ....
}
```



# CUDA ejemplo (III)

- Ejemplo sencillo: suma de matriz punto a punto:  $a = b + c$ 
  - Memoria GPU-CPU

```
// Reservar Memoria en la CPU (host)
float* a = (float*) malloc(N*N*sizeof(float));
float* b = (float*) malloc(N*N*sizeof(float));
```

```
// Reservar Memoria en la GPU (device)
float* a_GPU, b_GPU, c_GPU;
cudaMalloc( (void**) &a_GPU, N*N*sizeof(float));
cudaMalloc( (void**) &b_GPU, N*N*sizeof(float));
cudaMalloc( (void**) &c_GPU, N*N*sizeof(float));
```

```
// Transferencia Host - Device
cudaMemcpy(b_GPU, b, N*N*sizeof(float), HostToDevice));
cudaMemcpy(c_GPU, c, N*N*sizeof(float), HostToDevice));
```

```
// Transferencia Device - Host
a_host = (float*) malloc(N*N*sizeof(float));
cudaMemcpy(a_host, a_GPU, N*N*sizeof(float), DeviceToHost));
```

## ■ Positivo

- GPU Útil Como Dispositivo Acelerador Códigos con Paralelismo de Datos
  - Eficiencia Basada en Ejecución Concurrente de Miles de Threads
  - Gran Número de Unidades Funcionales, Control Relativamente sencillo

## ■ No tanto ...

- Sintonización Código Compleja
  - Optimizar Accesos a Memoria Global
  - Gestión Explícita Memoria Compartida
  - Ajuste Tamaño de *Block*
  - Uso de Texturas
- Aspectos no visibles
  - Asignación y planificación de *threads*
- Rendimiento Pobre en Doble Precisión

} Jerarquía memoria