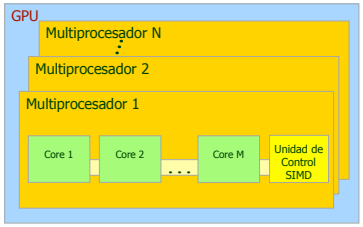
Breve introducción a CUDA

Manuel Ujaldón
Nvidia CUDA Fellow y Profesor Titular
Departamento de Arquitectura de Computadores
Universidad de Málaga (España)

El modelo hardware de CUDA: Un conjunto de procesadores SIMD

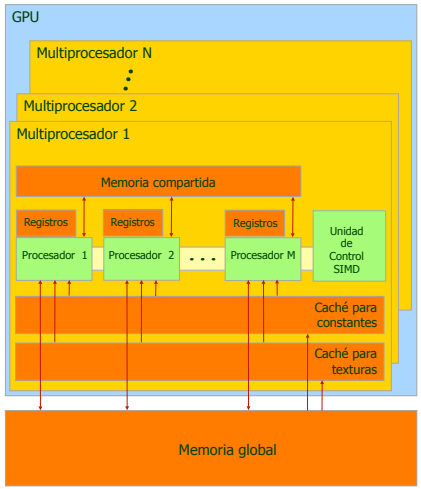
- La GPU consta de:
  - N multiprocesadores x M cores.
- Paralelismo masivo:
  - Aplicado sobre miles de hilos.
  - Compartiendo datos a diferentes niveles.
- Computación heterogénea:
  - GPU:
    - Intensiva en datos.
    - Paralelismo de grano fino.
  - CPU:
    - Gestión y control.
    - Paralelismo de grano grueso.



	G80	GT200	GF100 (Fermi)	K10 (Kepler) (2 x GK104)
Marco temporal	2006-07	2008-09	2010-11	2012-13
N (multiprocs.)	16	30	16	8
M (cores/multip.)	8	8	32	192
Núm. de cores	128	240	512	1536

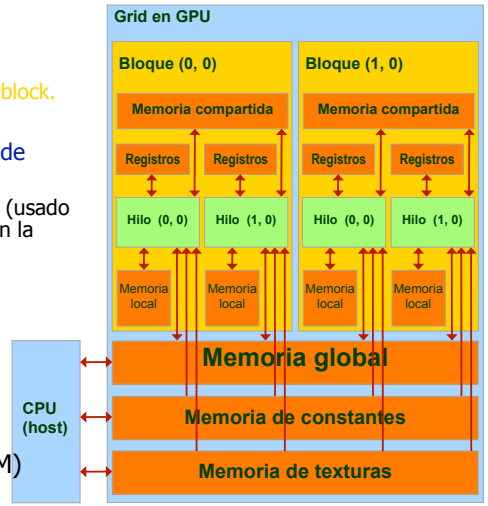
Arquitectura del sistema de memoria

- Cada multiprocesador tiene:
  - Su banco de registros.
  - Memoria compartida.
  - Una caché de constantes y otra de texturas, ambas de sólo lectura y uso marginal.
- La memoria global es la memoria de vídeo (GDDR5):
  - Tres veces más rápida que la de la CPU, pero...
  - ... ¡500 veces más lenta que la memoria compartida! (que es SRAM en realidad).



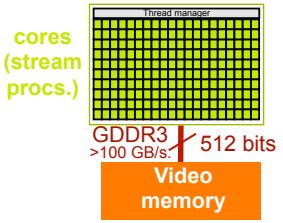
Acceso a memoria y sus limitaciones

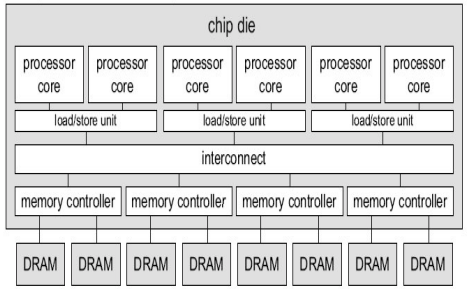
- Cada hilo puede:
  - Leer/escribir registros per-thread.
  - Leer/escribir memoria compartida per-block.
  - Leer/escribir memoria global per-grid.
- Cada hilo también puede, por motivos de comodidad o rendimiento:
  - Leer/escribir memoria local per-thread (usado por el compilador para volcar registros en la memoria global).
  - Leer memoria de constantes per-grid.
  - Leer memoria de texturas per-grid.
- La CPU puede:
  - Leer/escribir en memoria global, de constantes y de texturas (mapeadas sobre DRAM)



La arquitectura en general

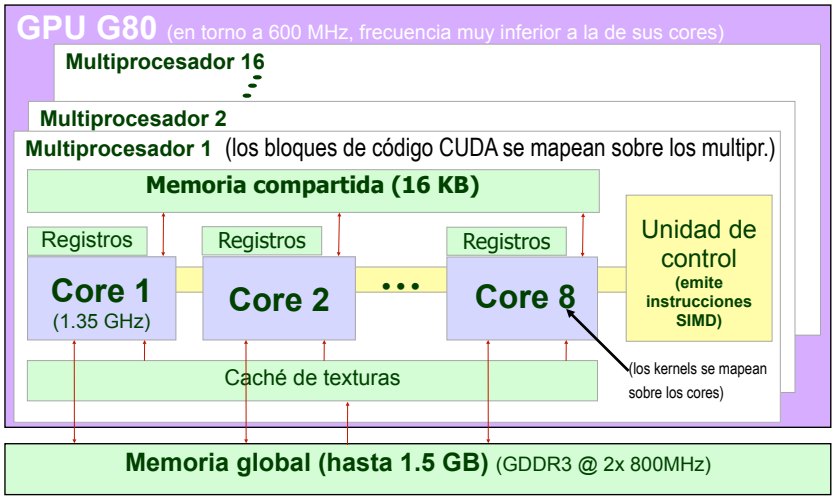
- Se compone de multiprocesadores dotados de 8 cores, donde los GFLOPS escalan con el número de cores (stream processors), y el ancho de banda escala con el número de controladores de memoria según el modelo comercial:



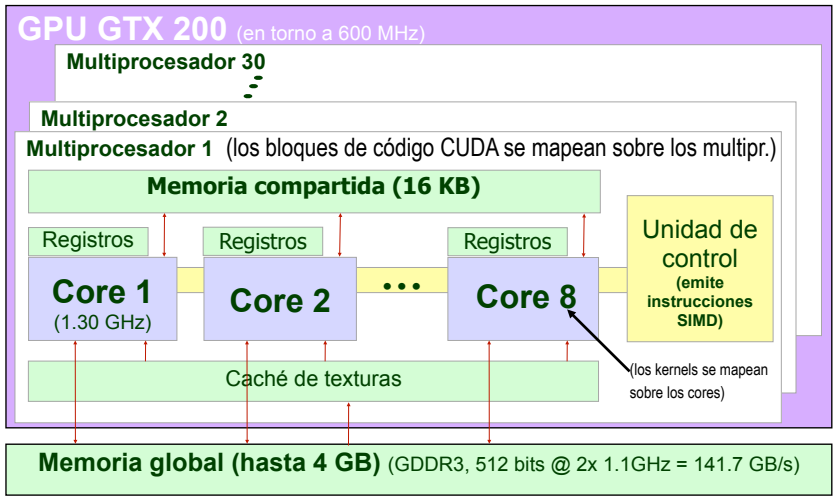


Modelo GeForce	GTS8600	GTX9800	GTX200
Multiprocesadores	4	16	30
Cores	32	128	240
GFLOPS	93	429	624
Control. de mem.	2	4	8
Anchura del bus	64	128	256
A. banda (GB/s)	32	70	141

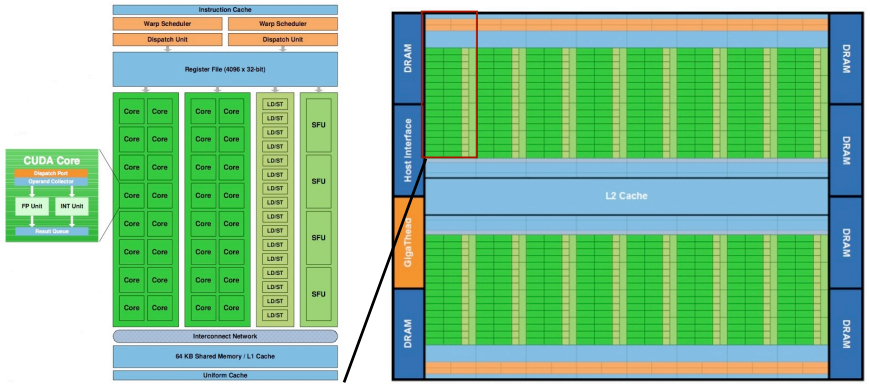
La primera generación: G80 (GeForce 8800)



La segunda generación: GT200 (GTX 200)

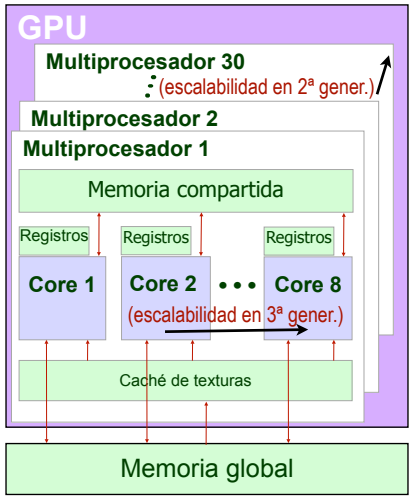


La tercera generación: Fermi (GF100)



Escalabilidad para futuras generaciones: Alternativas para su crecimiento futuro

- Aumentar el número de multiprocesadores por pares (nodo básico), esto es, crecer en la dimensión Z. Es lo que hizo la 2ª gener. (de 16 a 30).
- Aumentar el número de procesadores de cada multiprocesador, o crecer en la dimensión X. Es lo que hizo la 3ª gener. (de 8 a 32).
- Aumentar el tamaño de la memoria compartida, esto es, crecer en la dimensión Y.



Recursos y limitaciones según la GPU que utilicemos para programar CUDA

Parámetro	Valor según gener. GPU			Limi-tación	Impacto
CUDA Compute Capabilities (CCC)	1.0 y 1.1	1.2 y 1.3	Fermi		
Multiprocesadores / GPU	16	30	14-16	HW.	Escalabilidad
Cores / Multiprocesador	8	8	32	HW.	Escalabilidad
Hilos / Warp	32	32	32	SW.	Throughput
Bloques de hilos / Multiprocesador	8	8	8	SW.	Throughput
Hilos / Bloque	512	512	1024	SW.	Paralelismo
Hilos / Multiprocesador	768	1 024	1 536	SW.	Paralelismo
Registros de 32 bits / Multiprocesador	8K	16K	32K	HW.	Working set
Memoria compartida / Multiprocesador	16K	16K	16K 48K	HW.	Working set

Cinco claves para maximizar el rendimiento del código

1. Expresar explícitamente todo el paralelismo posible aplicando SIMD de grano fino para definir multitud de hilos.
   1. Si los hilos de un mismo bloque necesitan comunicarse, utilizar la memoria compartida y __syncthreads()
   2. Si los hilos de diferentes bloques necesitan comunicarse, utilizar la memoria global y descomponer la computación en múltiples kernels.
2. Aprovechar el ancho de banda con memoria: Pocas transferencias grandes en lugar de muchas pequeñas.
3. Optimizar la localidad de acceso: Reutilización de datos.
4. Ocultar latencias con memoria global maximizando la ocupación de unidades funcionales. Intensidad aritmética.
5. Maximizar el CPI del código (throughput): Seleccionar la instrucción de menor latencia en el repertorio CUDA.

Finalizamos con un ejemplo: CUDA es C con algunas palabras clave más

```
void saxpy_serial(int n, float a, float *x, float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
// Invocar al kernel SAXPY secuencial
saxpy_serial(n, 2.0, x, y);
```

Código C estándar

Código CUDA equivalente de ejecución paralela en GPU:

```
__global__ void saxpy_parallel(int n, float a, float *x,
float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n)  y[i] = a*x[i] + y[i];
}
// Invocar al kernel SAXPY paralelo con 256 hilos/bloque
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

Detalle del patrón de acceso común

Con N = 16 y blockDim = 4, tenemos 4 bloques de hilos, encargándose cada hilo de computar un elemento del vector.

blockIdx.x = 0
blockDim.x = 4
threadIdx.x = 0,1,2,3
idx = 0,1,2,3

blockIdx.x = 1
blockDim.x = 4
threadIdx.x = 0,1,2,3
idx = 4,5,6,7

blockIdx.x = 2
blockDim.x = 4
threadIdx.x = 0,1,2,3
idx = 8,9,10,11

blockIdx.x = 3
blockDim.x = 4
threadIdx.x = 0,1,2,3
idx = 12,13,14,15

Extensiones al lenguaje

```
int idx = (blockId.x * blockDim.x) + threadIdx.x;
```

Se mapeará del índice local threadIdx al índice global

Patrón de acceso común

Nota: blockDim debería ser >= 32 (warp size) en código real, esto es sólo un ejemplo