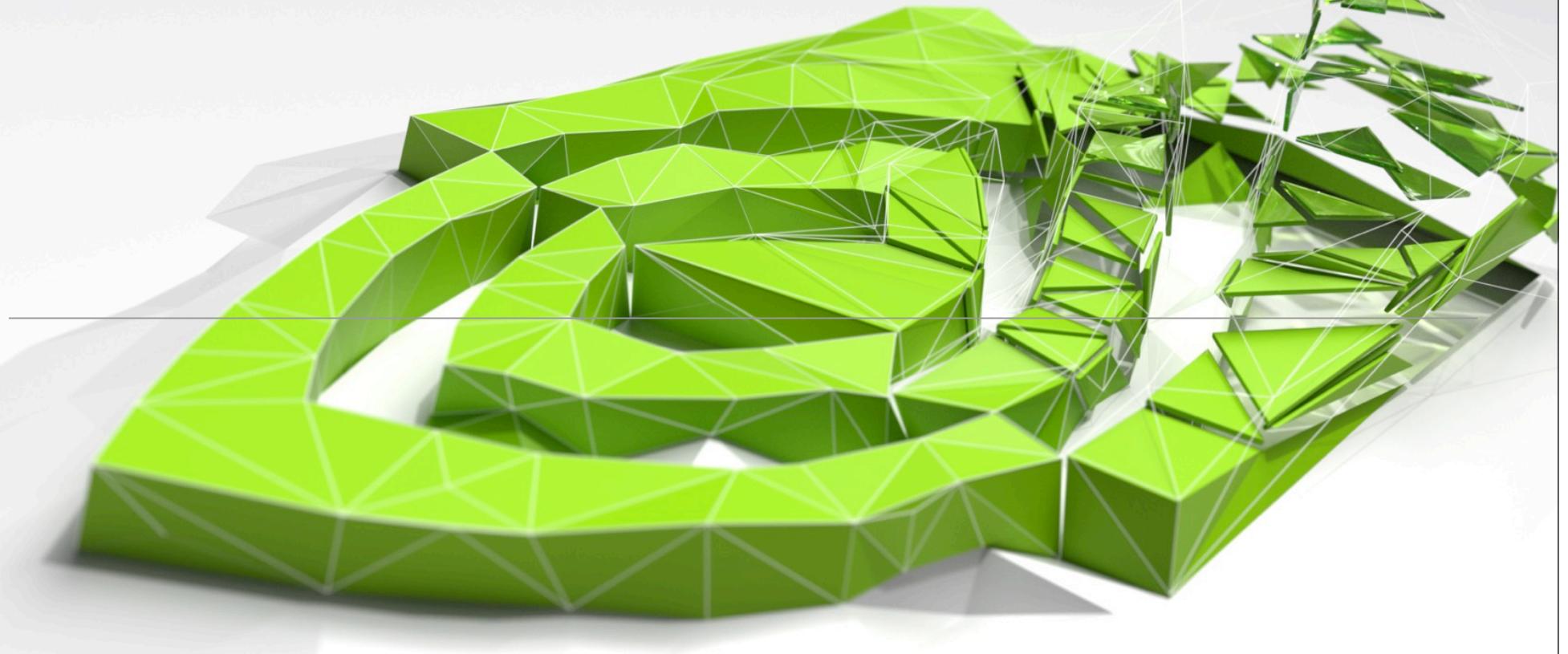


Programación de GPUs con CUDA

Diseño y Evaluación de Infraestructuras Informáticas

Grado de Ingeniería Informática. Univ. Málaga. Curso 2019/20



Manuel Ujaldón

Catedrático de Arquitectura de Computadores @ Universidad de Málaga

CUDA Fellow & DLI Ambassador @ Nvidia Corporation



Contenidos [143 diapositivas]

1. Introducción. [22 diapositivas]
2. Arquitectura. [36]
 1. El modelo hardware de CUDA. [3]
 2. Primera generación: Tesla (2006-2009). [3]
 3. Quinta generación: Pascal (2016-17). [9]
 4. Sexta generación: Volta (2018-?). [7]
 5. Séptima generación: Turing (2019-?). [12]
 6. Síntesis generacional. [2]
3. Programación. [19]
4. Sintaxis. [15]
 1. Elementos básicos. [9]
 2. Un par de ejemplos preliminares. [6]
5. Compilación y herramientas. [12]
6. Ejemplos: Base [2], VectorAdd [5], Stencil [8], Reverse [4], MxM [11]
7. Bibliografía, recursos y herramientas. [9]

Prerrequisitos para este curso

- Se requiere estar familiarizado con el lenguaje C.
- No es necesaria experiencia en programación paralela (aunque si se tiene, ayuda bastante).
- No se necesitan conocimientos sobre la arquitectura de la GPU: Empezaremos describiendo sus pilares básicos.
- No es necesario tener experiencia en programación gráfica. Eso era antes cuando GPGPU se basaba en los shaders y Cg. Con CUDA, no hace falta desenvolverse con vértices, píxeles o texturas porque es transparente a todos estos recursos.



I. Introducción



Bienvenido al mundo de las GPUs



Modelos comerciales: GeForce y Tesla frente a frente



Diseñada para jugar:

- El precio es prioritario (<500€).
- Gran disponibilidad/popularidad.
- Poca memoria de vídeo (3-4 GB.).
- Relojes un poco más rápidos.
- Perfecta para desarrollar código que luego pueda disfrutar Tesla.

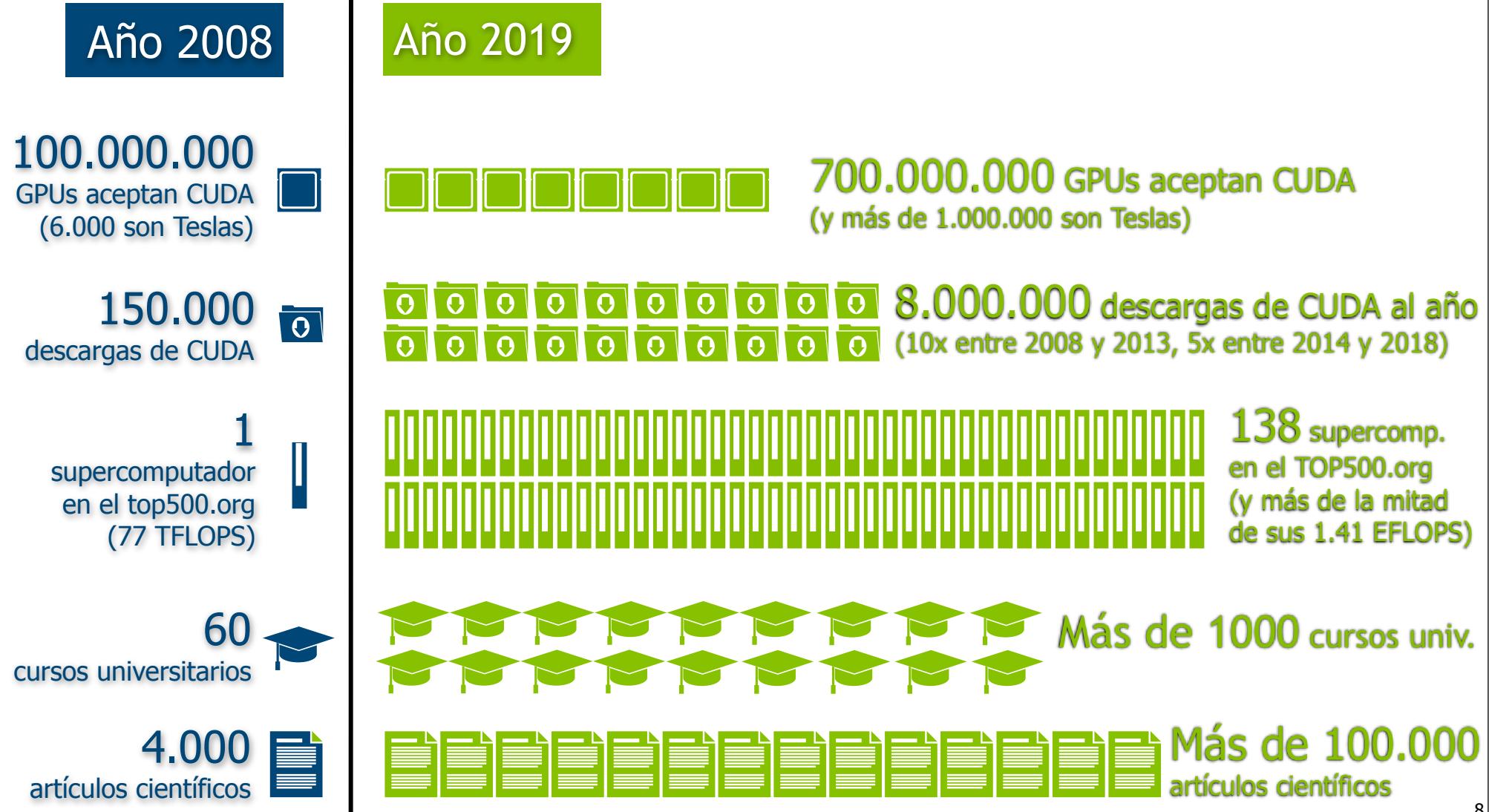
Orientada a HPC:

- Fiabilidad (tres años de garantía).
- Pensada para conectar en clusters.
- Más memoria de vídeo (12-24 GB).
- Ejecución sin descanso (24/7).
- GPUDirect (RDMA) y otras coberturas para clusters de GPUs.

Los personajes de esta historia: La foto de familia de CUDA

GPU Computing Applications						
Libraries and Middleware						
CUFFT CUBLAS CURAND CUSPARSE	CULA MAGMA	Thrust NPP	VSIPL SVM OpenCurrent	PhysX OptiX	iray	MATLAB Mathematica
Programming Languages						
C	C++	Fortran		Java Python Wrappers	DirectCompute	Directives (e.g. OpenACC)
CUDA-Enabled NVIDIA GPUs						
Kepler Architecture (compute capabilities 3.x)	GeForce 600 Series		Quadro Kepler Series		Tesla K20 Tesla K10	
Fermi Architecture (compute capabilities 2.x)	GeForce 500 Series GeForce 400 Series		Quadro Fermi Series		Tesla 20 Series	
Tesla Architecture (compute capabilities 1.x)	GeForce 200 Series GeForce 9 Series GeForce 8 Series		Quadro FX Series Quadro Plex Series Quadro NVS Series		Tesla 10 Series	
	 Entertainment	 Professional Graphics	 High Performance Computing			

La programación CUDA crece a un ritmo vertiginoso



Síntesis evolutiva de la GPU

- 2001: Primeros chips many-core (en los procesadores para vértices y píxeles), mostrando el camino evolutivo.
- 2003: Esos procesadores son programables (con Cg).
- 2006: Esos procesadores se unifican en un solo tipo.
- 2007: Emerge CUDA.
- 2008: Aritmética de punto flotante en doble precisión.
- 2010: Normalización de operandos y memoria ECC.
- 2012: Potenciación de la computación irregular.
- 2014: Unificación del espacio de direcciones CPU-GPU.
- 2016: Memoria 3D. Zócalo NV-link.
- Aún por mejorar: Robustez en clusters y conexión a disco.

Las 3 cualidades que han hecho de la GPU un procesador único

- Control simplificado.
 - El control de un hilo se amortiza en otros 31 (**warp size = 32**).
- Escalabilidad.
 - Aprovechándose del gran **volumen de datos** que manejan las aplicaciones, se define un modelo de paralelización sostenible.
- Productividad.
 - Se habilitan multitud de mecanismos para que cuando un hilo pase a realizar operaciones que no permitan su ejecución veloz, otro **oculte su latencia** tomando el procesador **de forma inmediata**.
- Palabras clave esenciales para CUDA:
 - Warp, SIMD, ocultación de latencia, commutación de contexto gratis.

Las 3 cualidades que han atraido a un mayor número de usuarios

● Coste

- Muy favorable gracias al volumen de ventas.
- Se venden tres GPUs por cada CPU, y este ratio sigue creciendo.

● Ubicuidad

- Cualquier persona tiene ya algunas GPUs.
- Y si no, puede adquirirla en cualquier tienda.

● Consumo

- Hace 10 años consumían más de 200 vatios. Ahora copan el top 25 de la lista Green 500. Progresión para números en punto flotante:

	GFLOPS/w sobre float (32-bit)	GFLOPS/w. sobre double (64-bit)
Fermi (2010)	5-6	3
Kepler (2012)	15-17	7
Maxwell (2014)	40	12

Las novedades más destacadas

- En procesamiento:

- La frecuencia deja de ser el motor: Calor y voltaje lo impiden.
- El paralelismo a nivel de instrucción (ILP), tarea (multi-thread) y zócalo (SMP) van agotando su potencial.
- Solución: Aprovechar el paralelismo de datos SIMD de la GPU, que sí es escalable.

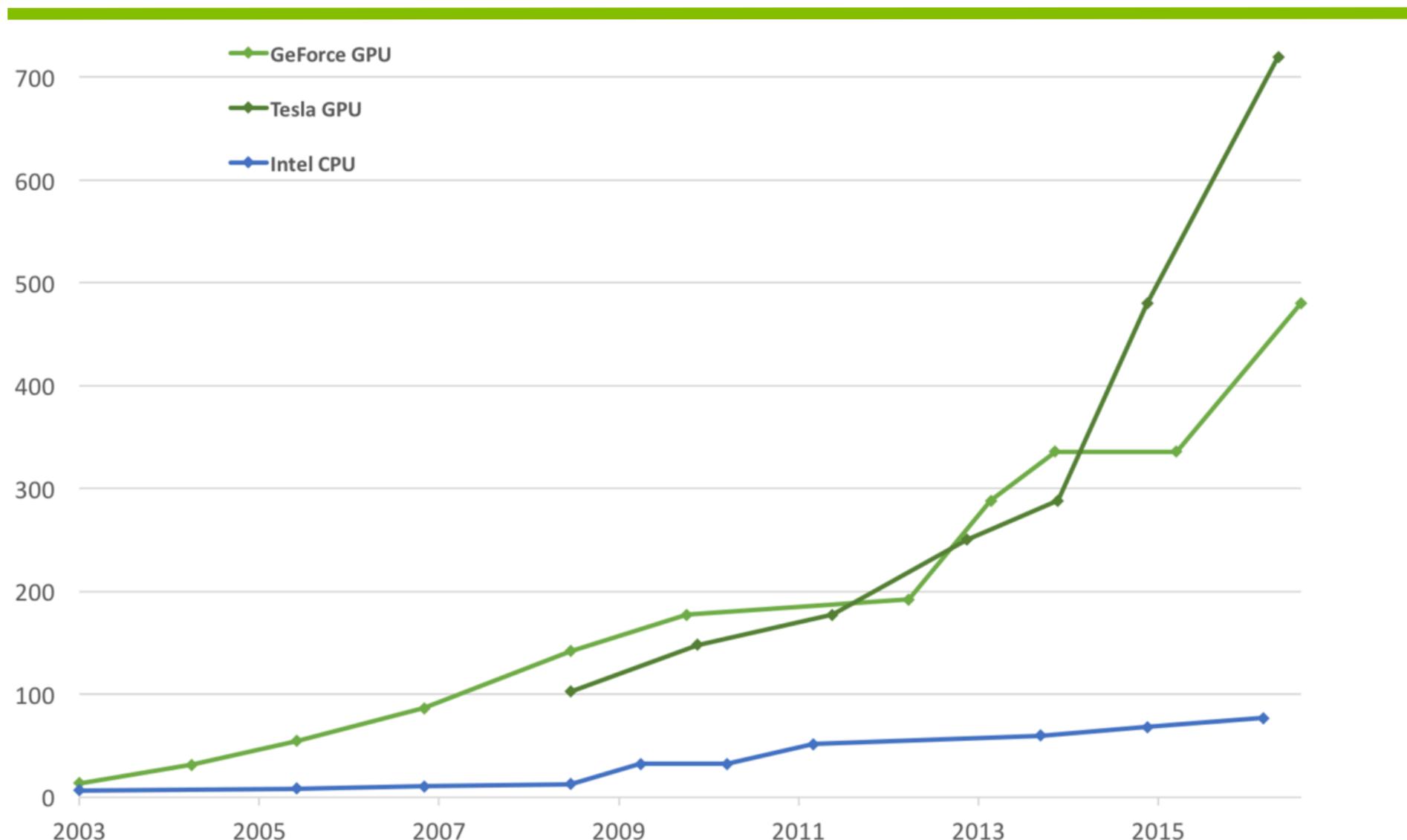
- En memoria estática (SRAM):

- Alternativa: Dejar pequeñas dosis de caché visibles al programador.

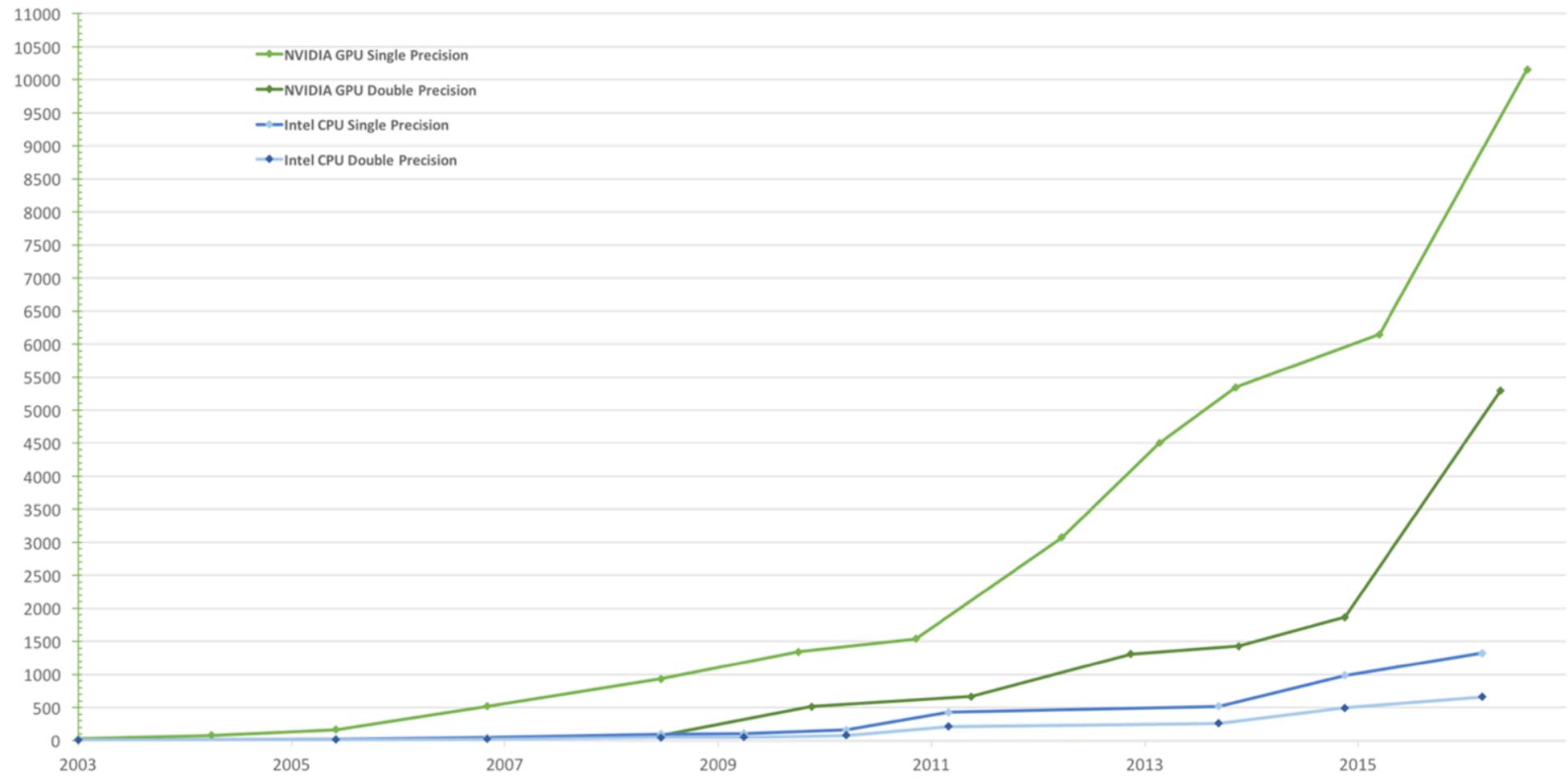
- En memoria dinámica (DRAM):

- Aumenta el ancho de banda (OK), pero también la latencia (ups).
- Solución: **Stacked-DRAM**, o cómo resolver por fin el *memory wall* aportando a la vez cantidad (Gbytes) y calidad (cercanía/velocidad).

Ancho de banda pico: GB/s teórico vs. CPU



Potencia de cálculo pico: GFLOPS teóricos a frecuencia base



¿Qué es CUDA?

“Compute Unified Device Architecture”

- Una plataforma diseñada conjuntamente a nivel software y hardware para aprovechara tres niveles la potencia de una GPU en aplicaciones de propósito general:
 - **Software:** Permite programar la GPU con mínimas pero potentes extensiones SIMD para lograr una ejecución eficiente y escalable.
 - **Firmware:** Ofrece un driver para la programación GPGPU que es compatible con el utilizado para renderizar. Sencillos APIs manejan los dispositivos, la memoria, etc.
 - **Hardware:** Habilita el paralelismo de la GPU para programación de propósito general a través de un número de multiprocesadores gemelos dotados de un conjunto de núcleos computacionales arropados por una jerarquía de memoria.

Lo esencial de CUDA C

- En general, es lenguaje C con mínimas extensiones:

- El programador escribe el programa para un solo hilo (thread), y el código se instancia de forma automática sobre miles de hilos.

- CUDA define:

- Un modelo de arquitectura:

- Con multitud de unidades de proceso (cores), agrupadas en multiprocesadores que comparten una misma unidad de control (ejecución SIMD).

- Un modelo de programación:

- Basado en el paralelismo masivo de datos y en el paralelismo de grano fino.
 - Escalable: El código se ejecuta sobre cualquier número de cores sin recompilar.

- Un modelo de gestión de la memoria:

- Más explícita al programador, con control explícito de la memoria caché.

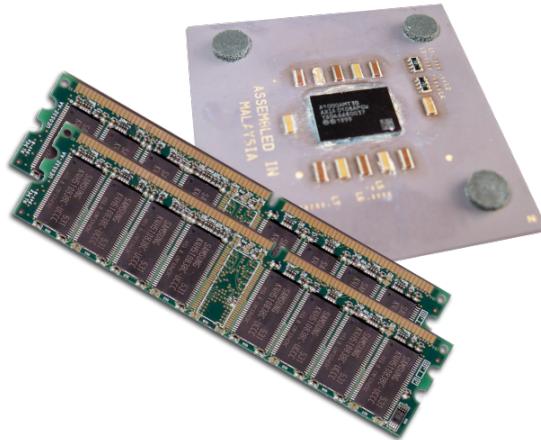
- Objetivos:

- Construir código escalable a cientos de cores, declarando miles de hilos.
 - Permitir computación heterogénea en CPU y GPU.

Computación heterogénea (1/4)

Terminología:

- Host (el anfitrión): La CPU y la memoria de la placa base [DDR3].
- Device (el dispositivo): La tarjeta gráfica [GPU + memoria de vídeo]:
 - GPU: Nvidia GeForce/Tesla.
 - Memoria de vídeo: GDDR5 o memoria 3D.



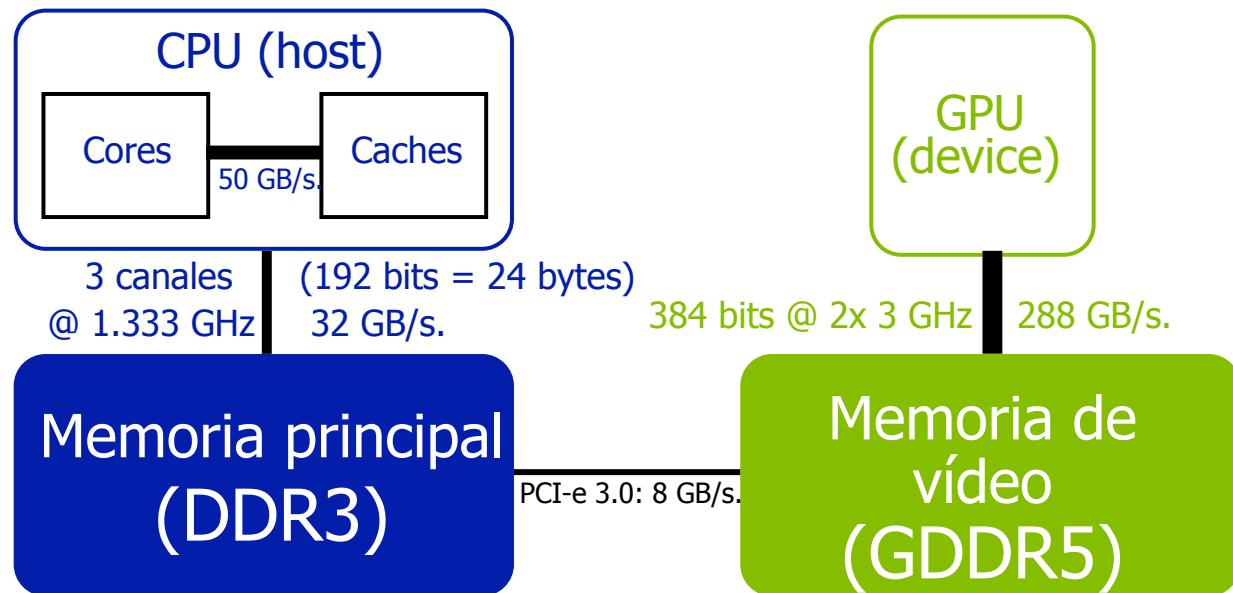
Host



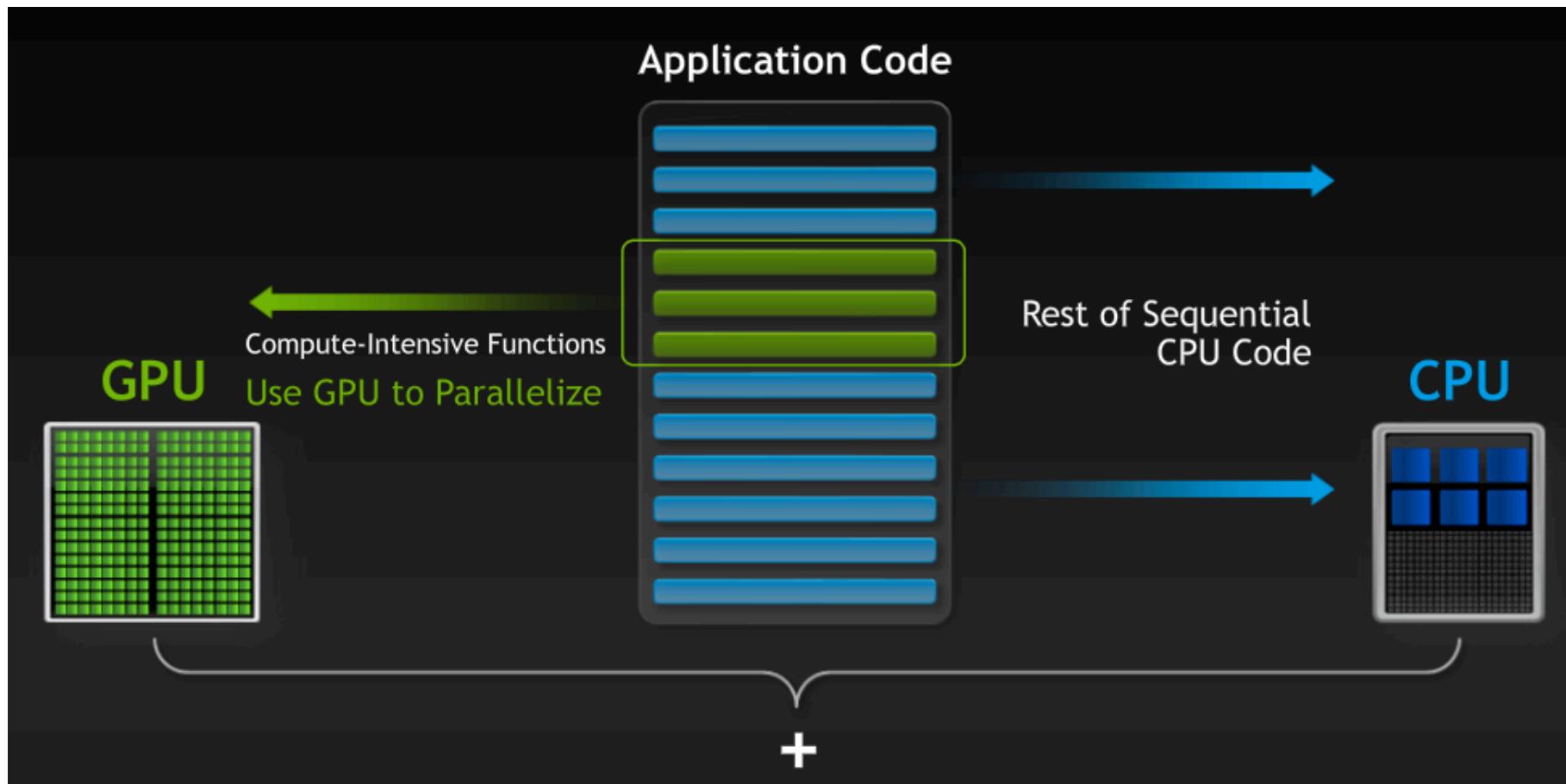
Device

Computación heterogénea (2/4)

- CUDA ejecuta un programa sobre un dispositivo (la GPU), que actúa como coprocesador de un anfitrión o host (la CPU).
- CUDA puede verse como una librería de funciones que contienen 3 tipos de componentes:
 - Host: Control y acceso a los dispositivos.
 - Dispositivos: Funciones específicas para ellos.
 - Todos: Tipos de datos vectoriales y un conjunto de rutinas soportadas por ambas partes.



Computación heterogénea (3/4)



- El código reescrito en CUDA puede ser inferior al 5%, pero consumir más del 50% del tiempo si no migra a la GPU.

Computación heterogénea (4/4)

```
#include <iostream>
#include <algorithm>

using namespace std;

#define N    1024
#define RADIUS 3
#define BLOCK_SIZE 16

_global_ void stencil_1d(int *in, int *out) {
    _shared_ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;

    // Read Input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    // Synchronize (ensure all the data is available)
    __syncthreads();

    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
        result += temp[lindex + offset];

    // Store the result
    out[gindex] = result;
}

void fill_ints(int *x, int n) {
    fill_n(x, n, 1);
}

int main(void) {
    int *in, *out;           // host copies of a, b, c
    int *d_in, *d_out;       // device copies of a, b, c
    int size = (N + 2*RADIUS) * sizeof(int);

    // Alloc space for host copies and setup values
    in = (int *)malloc(size); fill_ints(in, N + 2*RADIUS);
    out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

    // Alloc space for device copies
    cudaMalloc((void **)&d_in, size);
    cudaMalloc((void **)&d_out, size);

    // Copy to device
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

    // Launch stencil_1d() kernel on GPU
    stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS, d_out + RADIUS);

    // Copy result back to host
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

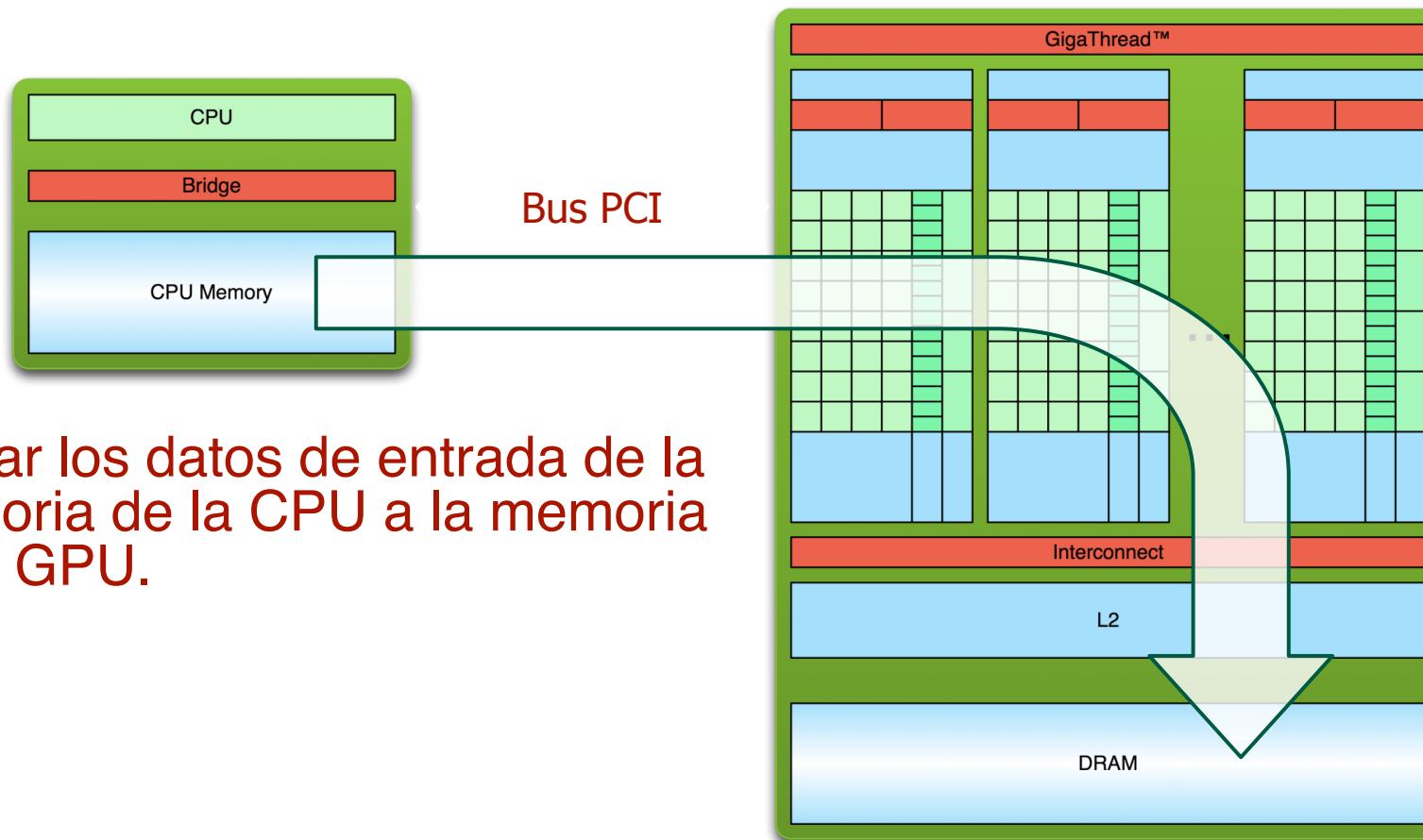
    // Clean up
    free(in); free(out);
    cudaFree(d_in); cudaFree(d_out);
    return 0;
}
```

CODIGO DEL DISPOSITIVO:
Función paralela
escrita en CUDA.

CODIGO DEL HOST:
- Código serie.
- Código paralelo.
- Código serie.

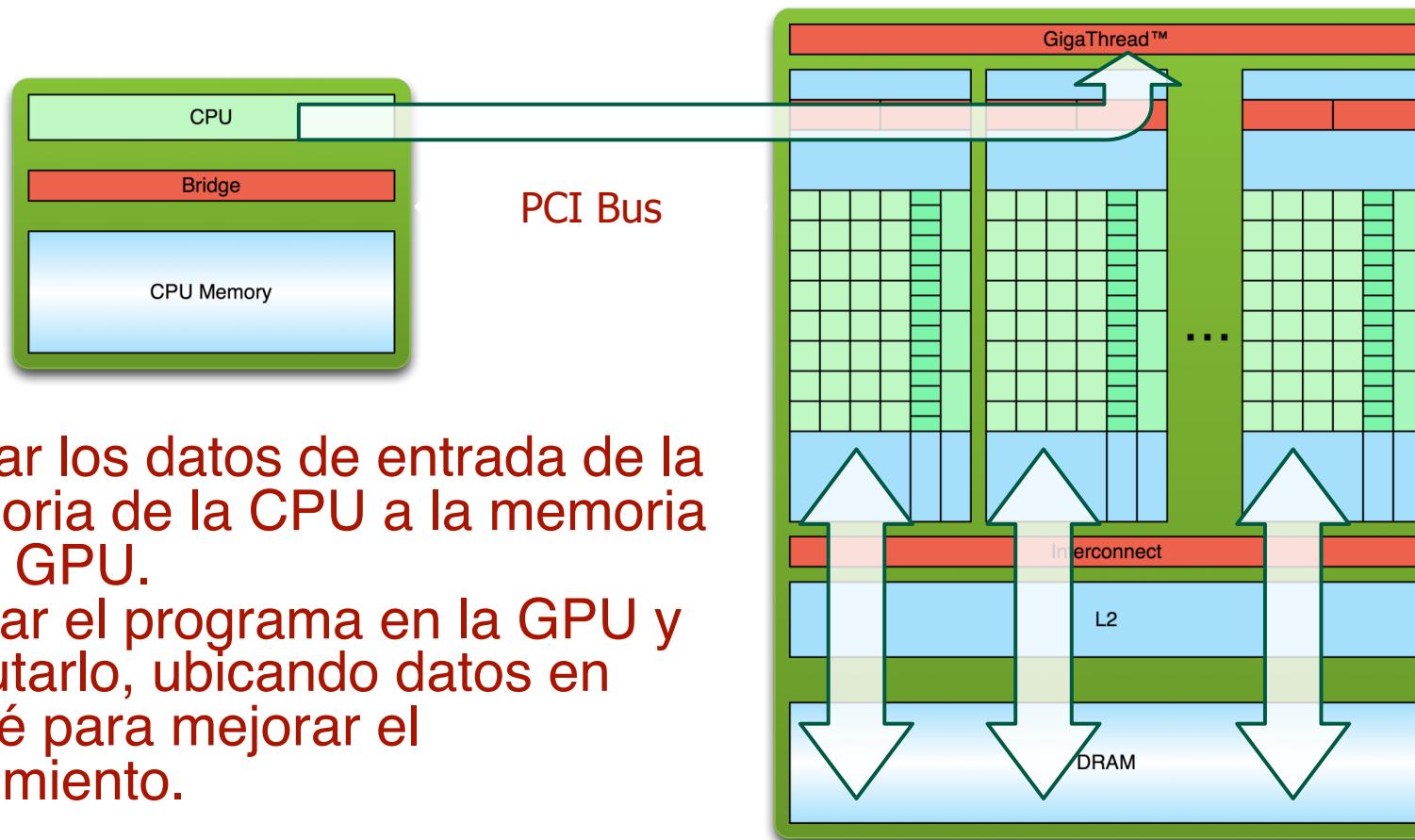


Un sencillo flujo de procesamiento (1/3)

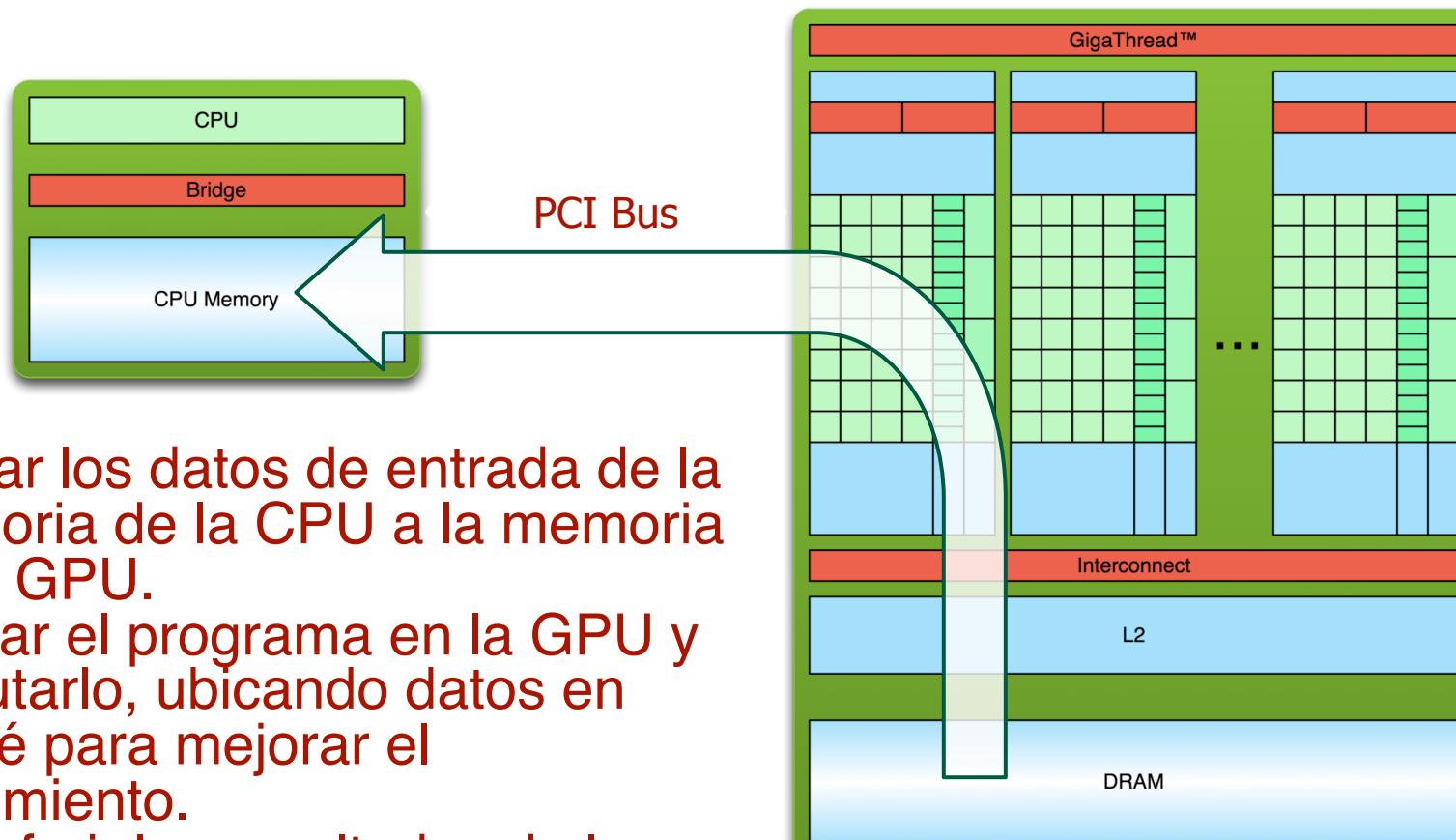


1. Copiar los datos de entrada de la memoria de la CPU a la memoria de la GPU.

Un sencillo flujo de procesamiento (2/3)



Un sencillo flujo de procesamiento (3/3)



1. Copiar los datos de entrada de la memoria de la CPU a la memoria de la GPU.
2. Cargar el programa en la GPU y ejecutarlo, ubicando datos en caché para mejorar el rendimiento.
3. Transferir los resultados de la memoria de la GPU a la memoria de la CPU.

El clásico ejemplo

```
int main(void) {  
    printf("¡Hola mundo!\n");  
    return 0;  
}
```

Salida:

```
$ nvcc hello.cu  
$ a.out  
¡Hola mundo!  
$
```

- Es código C estándar que se ejecuta en el host.
- El compilador nvcc de Nvidia puede utilizarse para compilar programas que no contengan código para la GPU.

¡Hola mundo! con código para la GPU (1/2)

```
__global__ void mikernel(void)
{
    printf("¡Hola mundo!\n");
}
int main(void)
{
    mikernel<<<1,1>>>();
    return 0;
}
```

- ➊ Dos nuevos elementos sintácticos:

- ➌ La palabra clave de CUDA `__global__` indica una función que se ejecuta en la GPU y se lanza desde la CPU. Por ejemplo, `mikernel<<<1,1>>>`.

- ➌ Eso es todo lo que se requiere para ejecutar una función en GPU.

- ➌ nvcc separa el código fuente para la CPU y la GPU.
- ➌ Las funciones que corresponden a la GPU (como `mikernel()`) son procesadas por el compilador de Nvidia.
- ➌ Las funciones de la CPU (como `main()`) son procesadas por su compilador (gcc para Unix, cl.exe para Windows).

¡Hola mundo! con código para la GPU (2/2)

```
global__ void mikernel(void)
{
}
int main(void)
{
    mikernel<<<1,1>>>();
    printf("¡Hola mundo!\n");
    return 0;
}
```

Salida:

```
$ nvcc hello.cu
$ a.out
¡Hola mundo!
$
```

- `mikernel()` no hace nada esta vez.
- Los símbolos "`<<<`" y "`>>>`" delimitan la llamada desde el código de la CPU al código de la GPU, también denominado "lanzamiento de un kernel".
- Los parámetros 1,1 describen el parallelismo (bloques e hilos CUDA).



II. Arquitectura



“...y si la gente del software quiere buenas máquinas, deben aprender más sobre hardware para poder así influir en los diseñadores de hardware ...”

David A. Patterson & John Hennessy

Organización y Diseño de Computadores

Mc-Graw-Hill (1995)

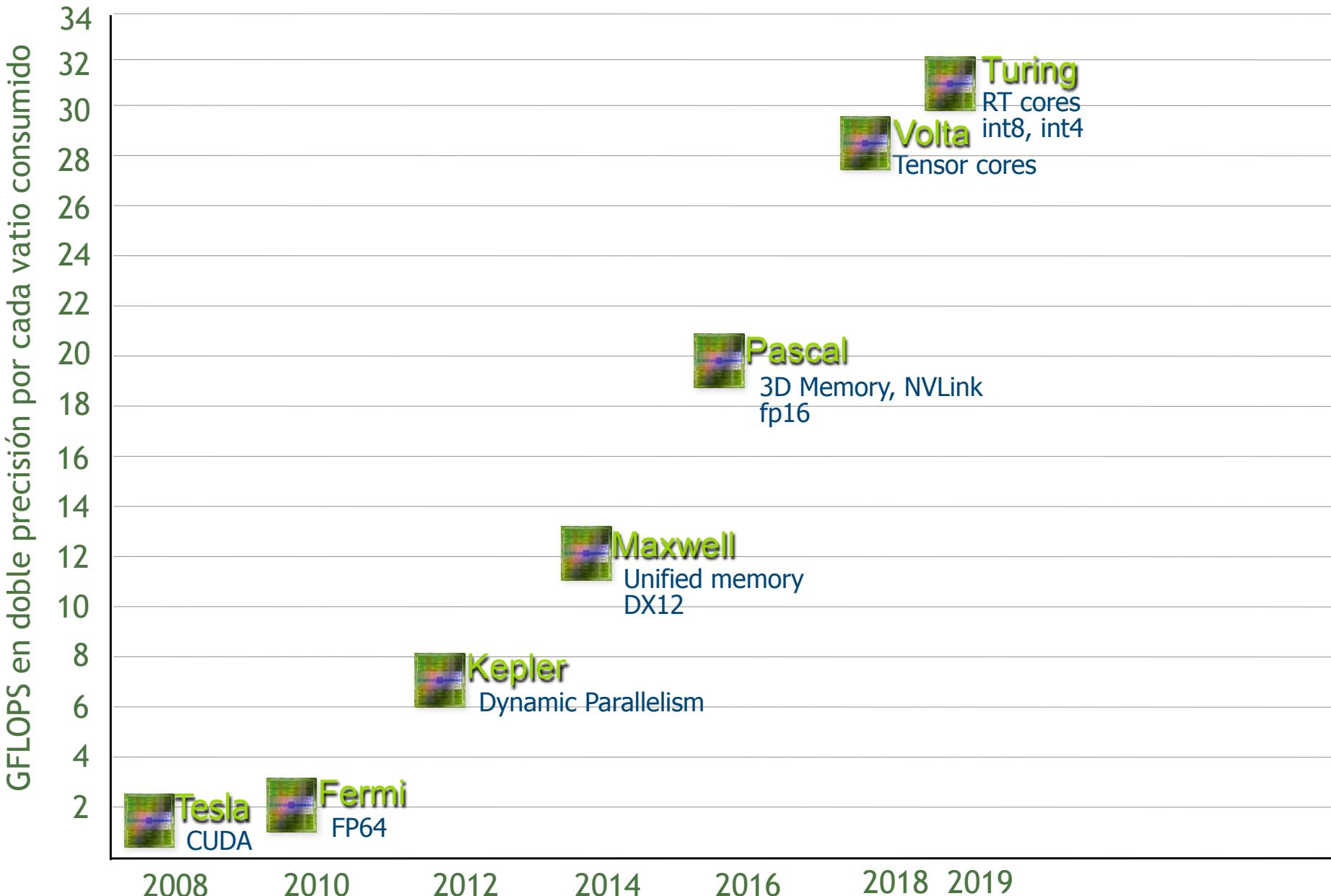
Capítulo 9, página 569



II.1. El modelo hardware de CUDA



Las 7 generaciones hardware de CUDA



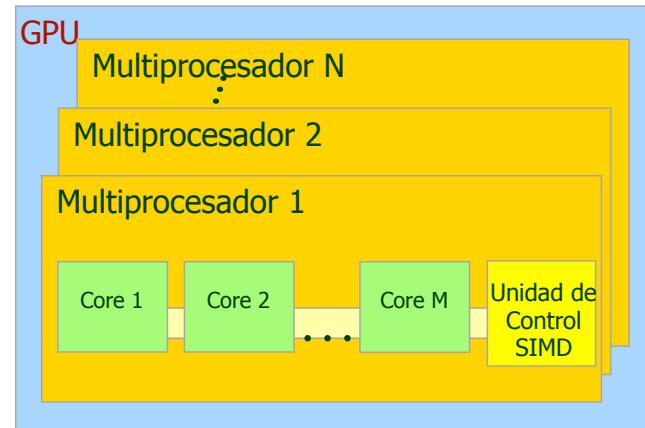
El modelo hardware de CUDA: Un conjunto de procesadores SIMD

- La GPU consta de:

- N multiprocesadores, cada uno dotado de M cores (o procesadores streaming).

- Computación heterogénea:

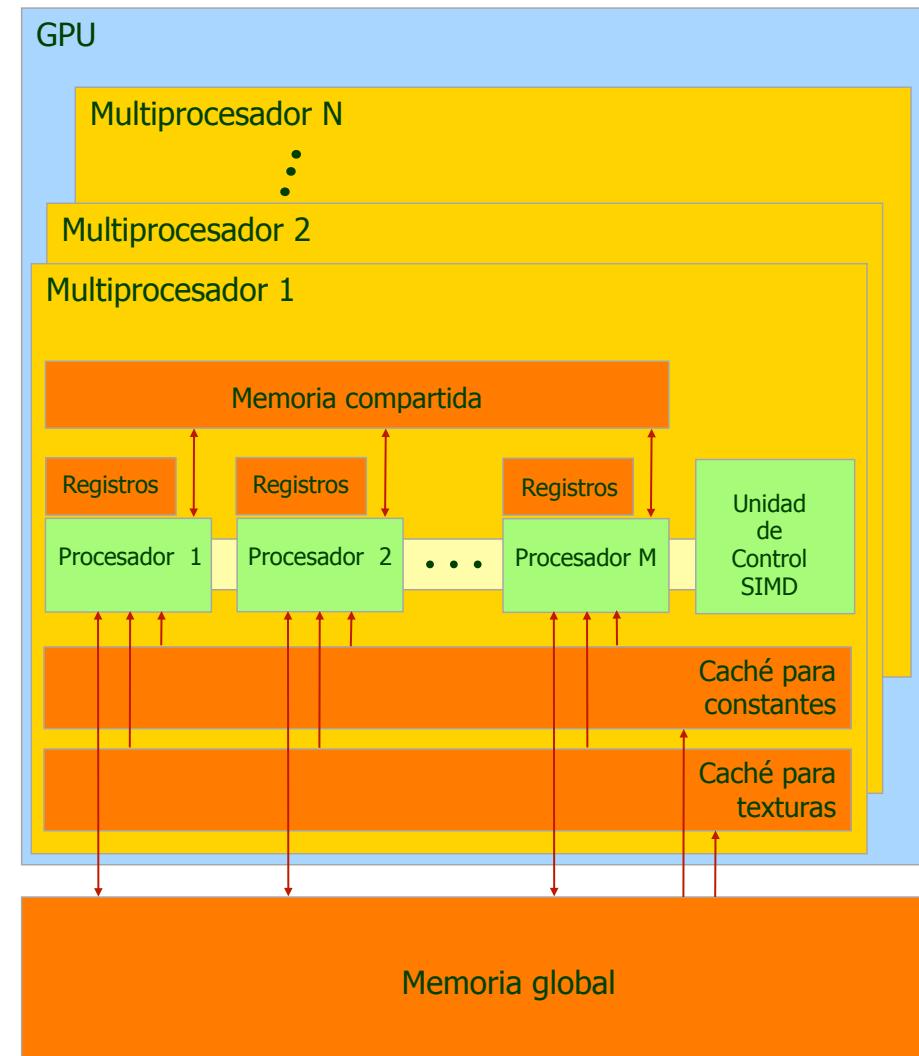
- GPU: Intensiva en datos. Paralelismo fino.
- CPU: Saltos y bifurcaciones. Paralelismo grueso.



	G80 (Tesla)	GT200 (Tesla)	GF100 (Fermi)	GK110 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)	GV100 (Volta)	TU102 (Turing)
Marco temporal	2006-07	2008-09	2010-11	2012-13	2014-15	2016-17	2018-?	2019-?
N (multiprocesadores)	16	30	14-16	13-15	4-24	56	80	72
M (cores/multiproc.)	8	8	32	192	128	64	64	64
# cores	128	240	448-512	2496-2880	512-3072	3584	5120	4608

Jerarquía 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 memoria principal de la CPU, pero... ¡500 veces más lenta que la memoria compartida! (que es SRAM en realidad).

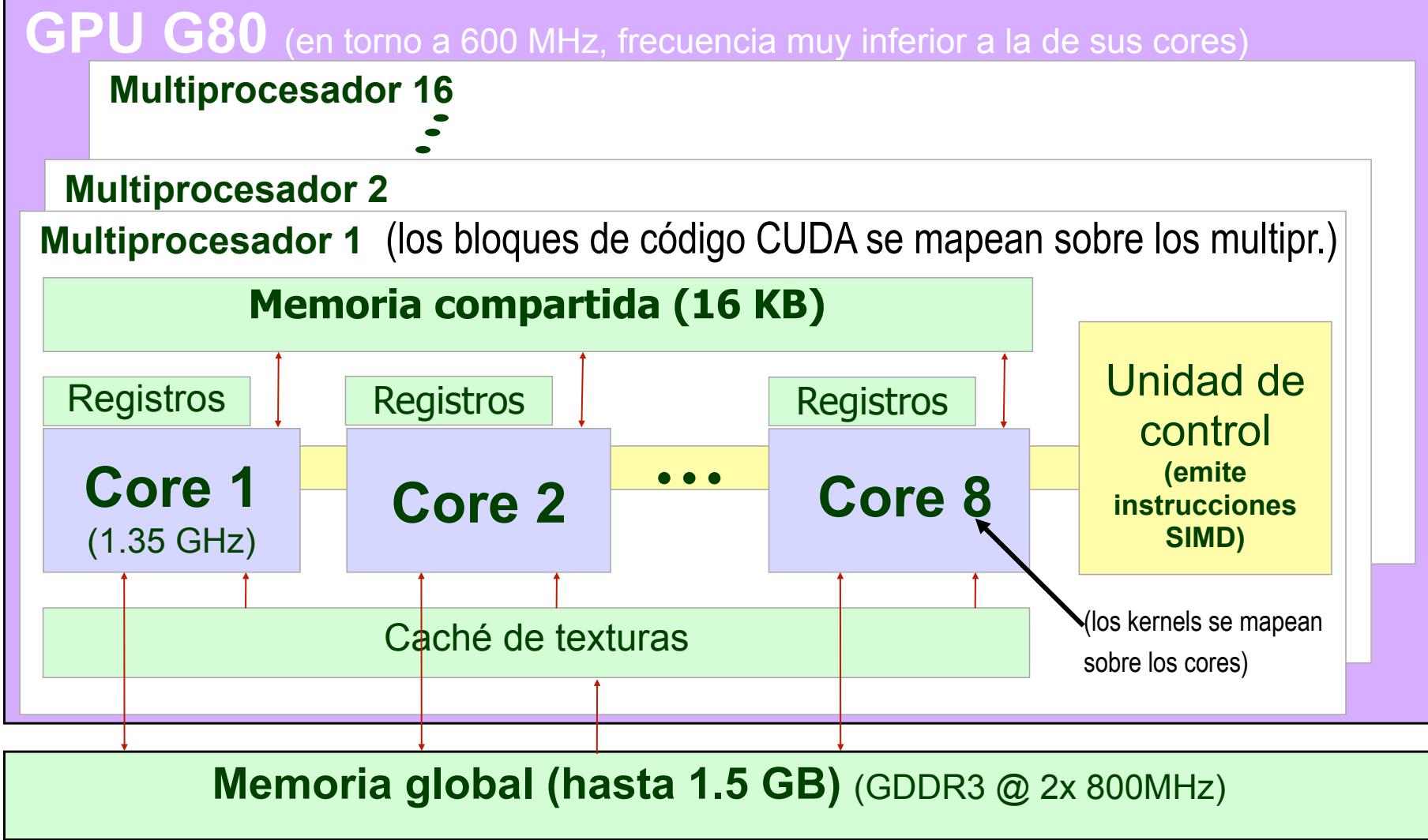




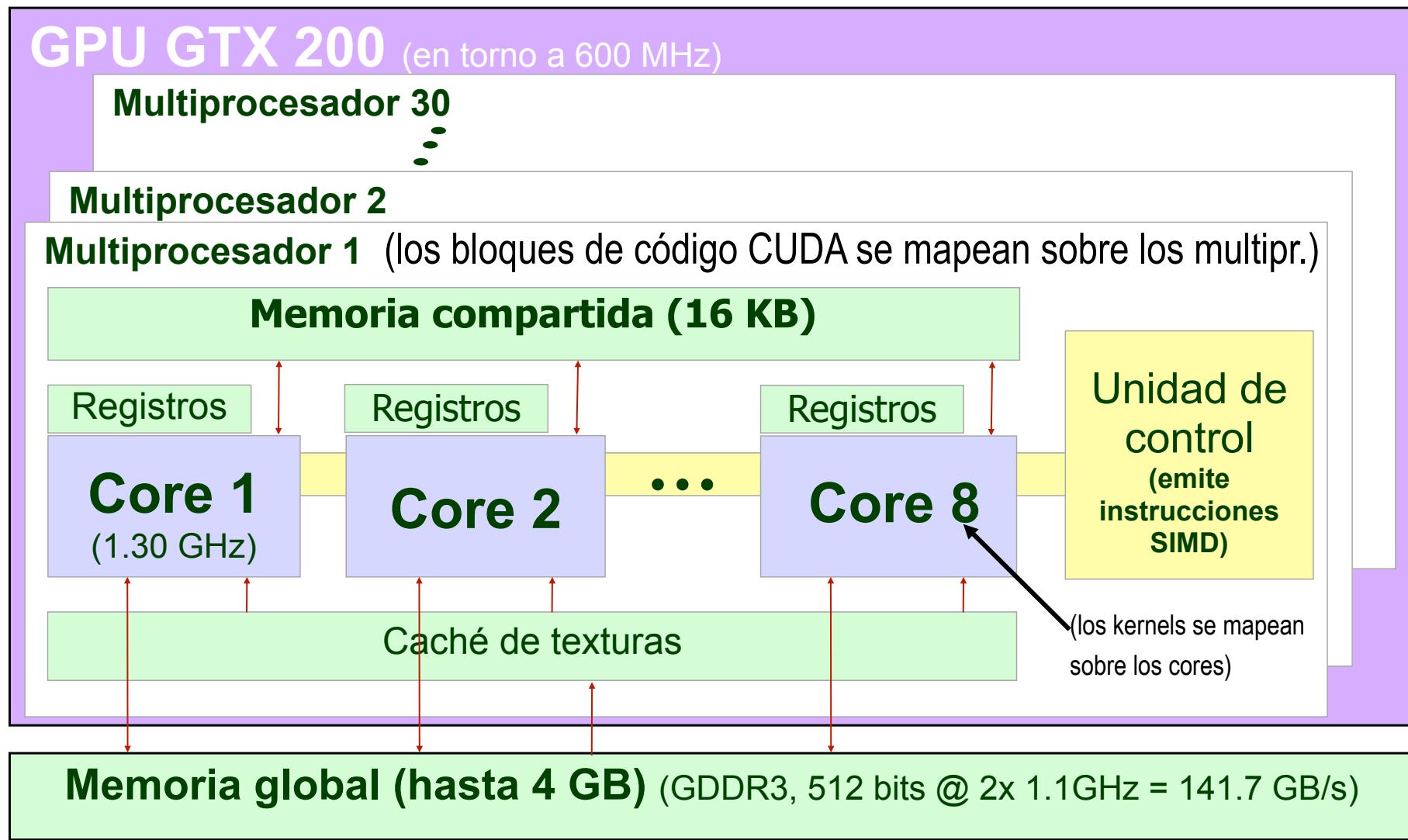
II.2. Primera generación: Tesla (G80 y GT200)



La primera generación: G80 (GeForce 8800)

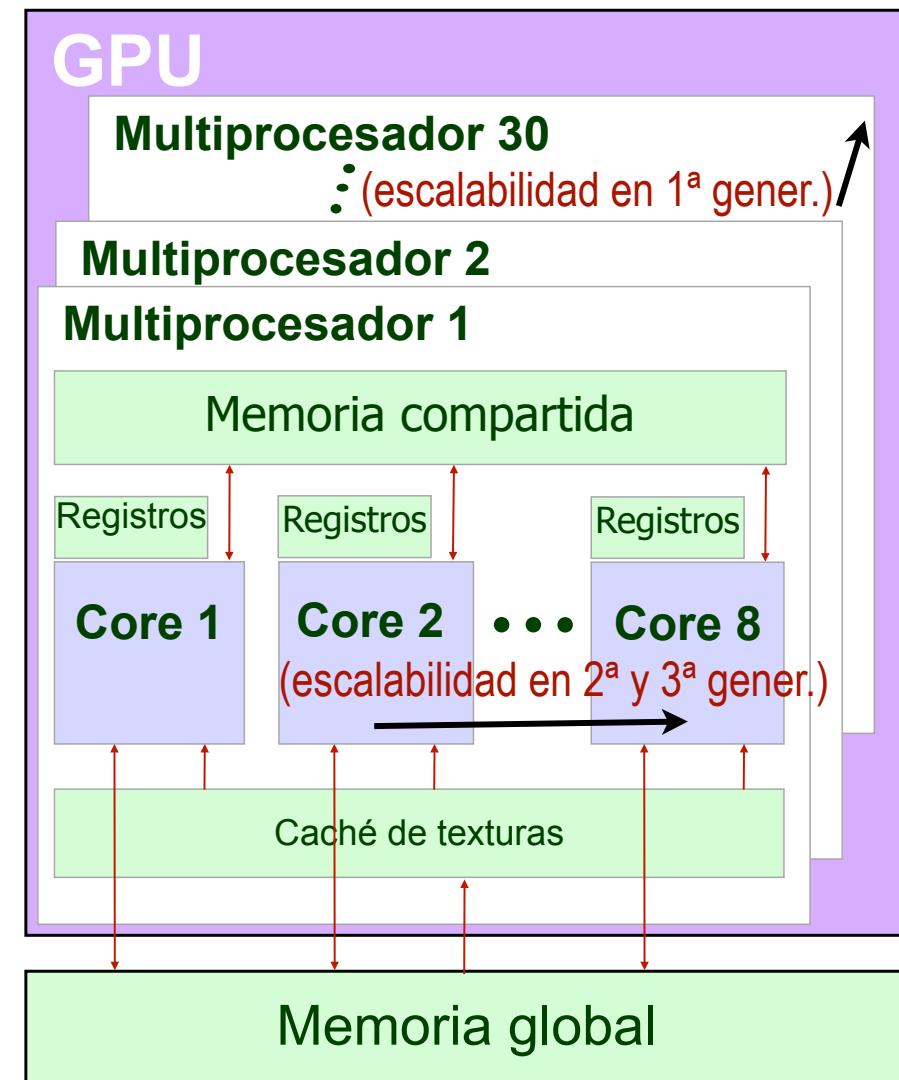


La primera generación: GT200 (GTX 200)



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 1^a gener. (de 16 a 30).
- Aumentar el número de procesadores de cada multiprocesador, o crecer en la dimensión X. Es el camino trazado por la 2^a y 3^a gener.
- Aumentar el tamaño de la memoria compartida, esto es, crecer en la dimensión Y.

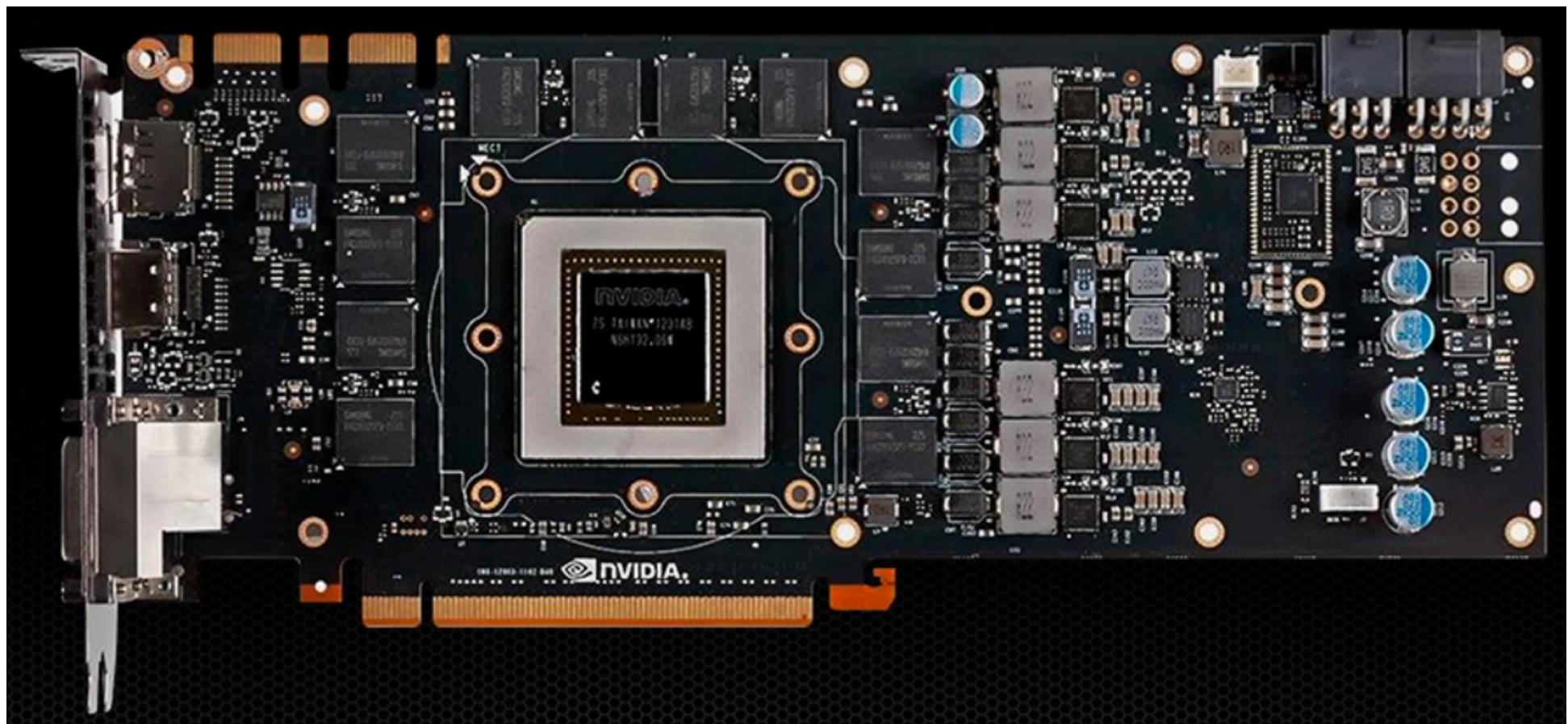




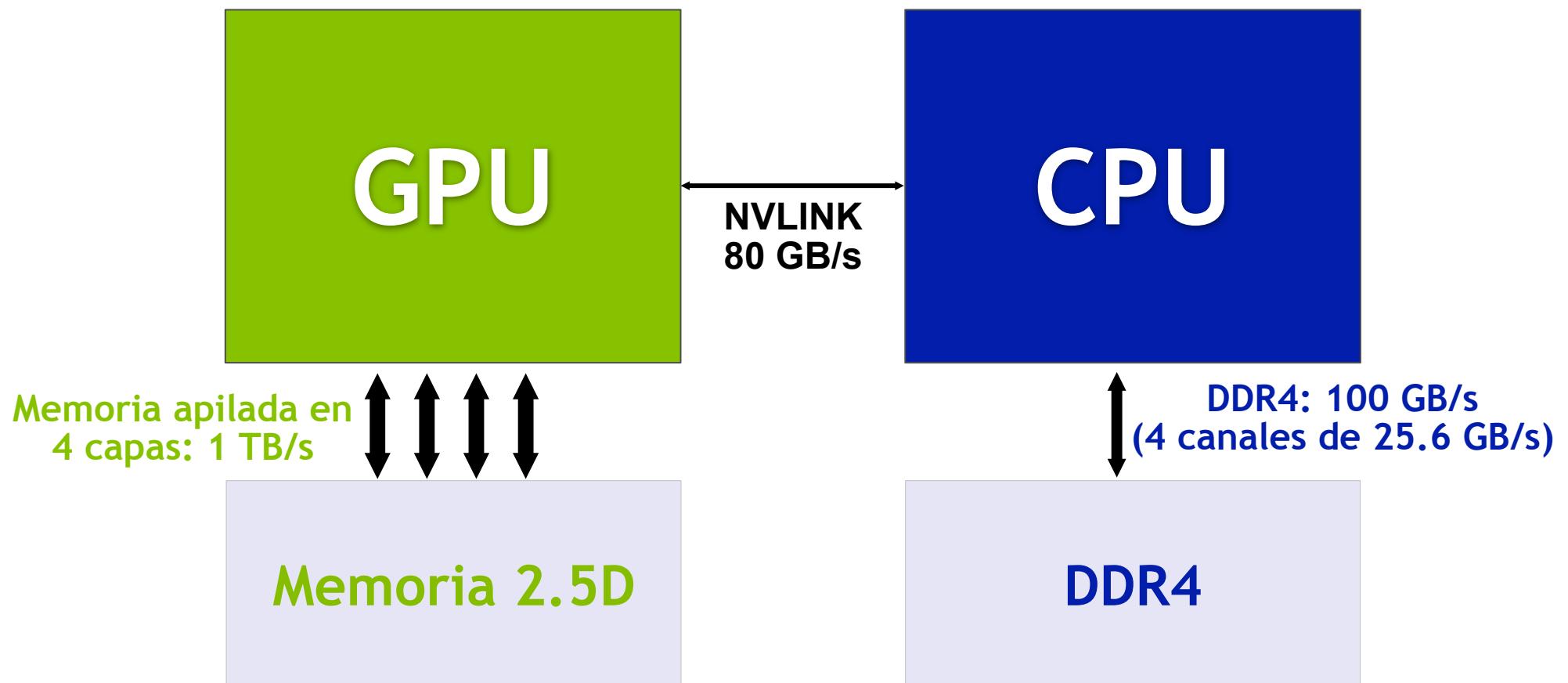
II. 3. Quinta generación: Pascal (GPxxx)



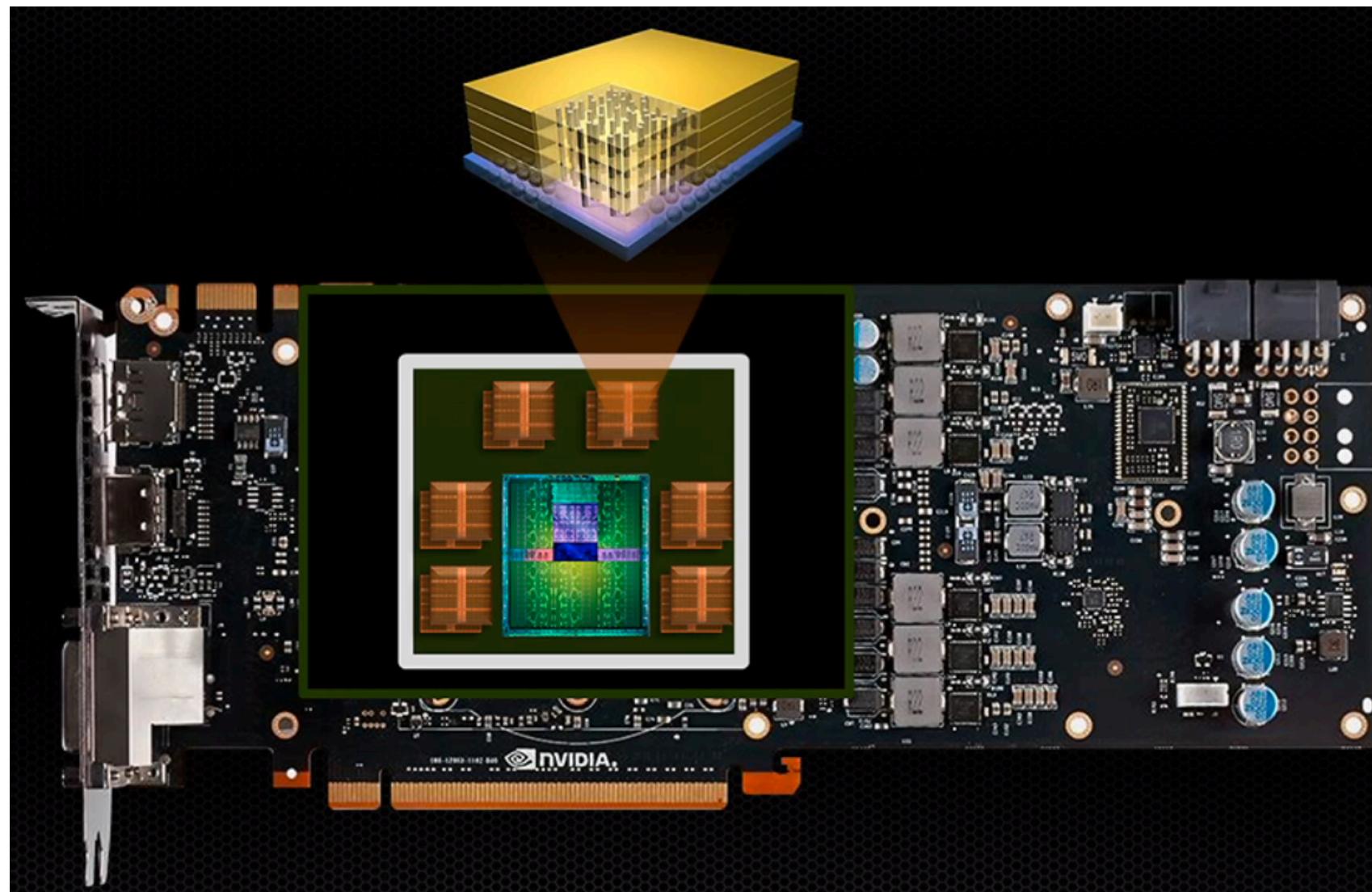
La tarjeta gráfica de 2015: GPU Kepler/Maxwell con memoria GDDR5



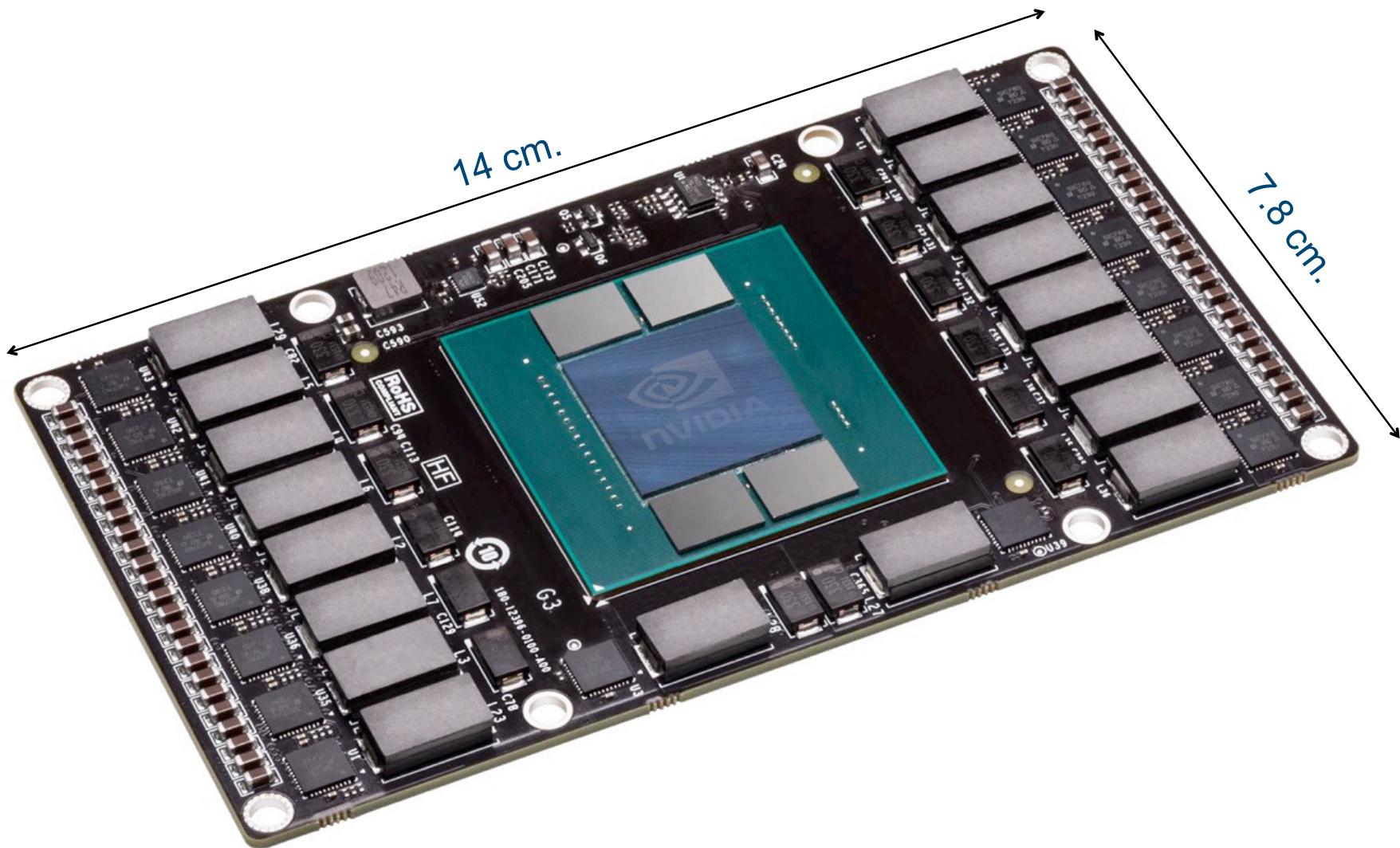
En 2017



La tarjeta gráfica de 2017: GPU Pascal con memoria Stacked (3D) DRAM



Un prototipo de GPU Pascal



Primer modelo comercial: GeForce GTX 1080.

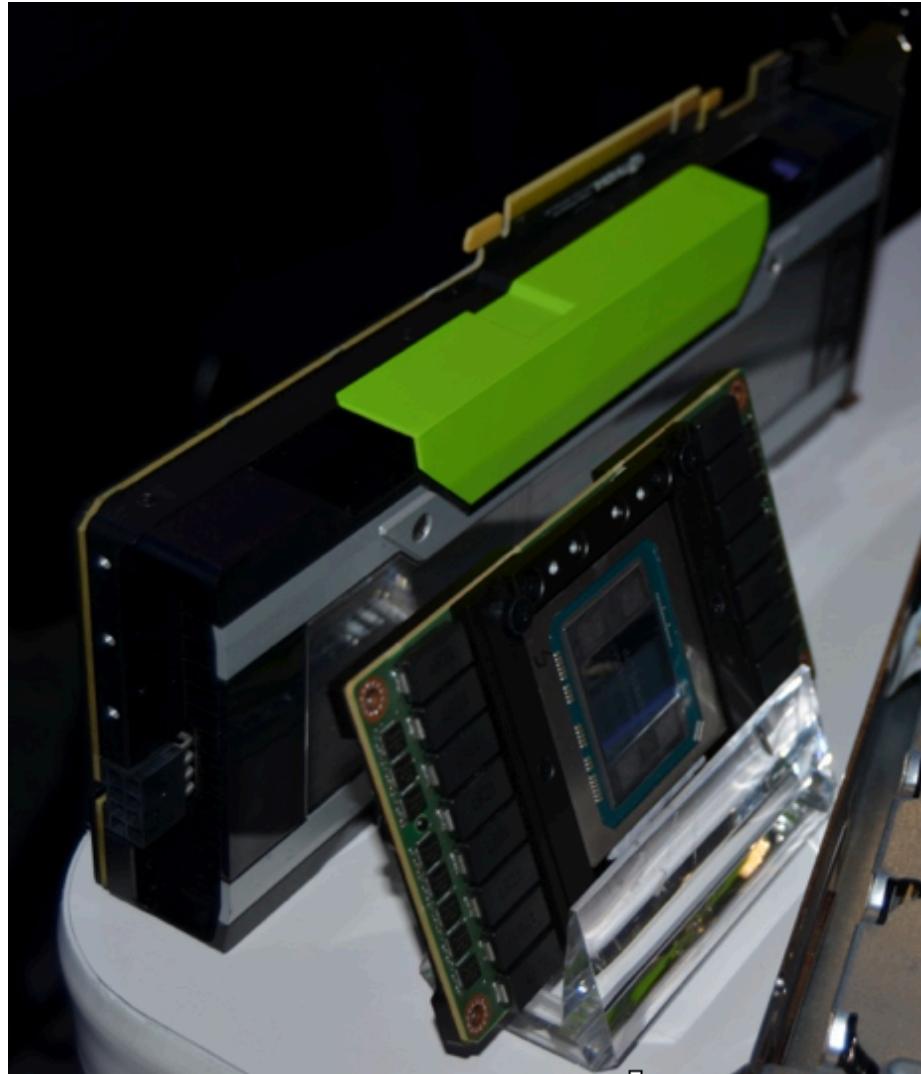
Comparativa con las 2 generaciones anteriores

	GTX 680 (Kepler)	GTX 980 (Maxwell)	GTX 1080 (Pascal)
Fecha de lanzamiento	2012	2014	2016
Transistores	3.54 B @ 28 nm.	5.2 B @ 28 nm.	7.2 B @ 16 nm.
Consumo y área int.	195 W & 294 mm ²	165 W & 398 mm ²	180 W & 314 mm ²
Multiprocesadores	8	16	40
Cores / Multiproc.	192	128	64
Cores / GPU	1536	2048	2560
Reloj (sin y con GPU Boost)	1006, 1058 MHz	1126, 1216 MHz	1607, 1733 MHz
Rendimiento pico	3250 GFLOPS	4980 GFLOPS	8873 GFLOPS
Memoria compartida	16, 32, 48 KB	64 KB	
Tamaño de caché L1	48, 32, 16 KB	Integrada con la caché de texturas	
Tamaño de caché L2 (recortada respecto a Teslas)	512 KB	2048 KB	
Memoria DRAM: Interfaz	256-bit GDDR5	256-bit GDDR5	256-bit GDDR5X
Memoria DRAM: Frecuencia	2x 3000 MHz	2x 3500 MHz	4x 2500 MHz
Memoria DRAM: Ancho banda	192.2 GB/s	224 GB/s	320 GB/s

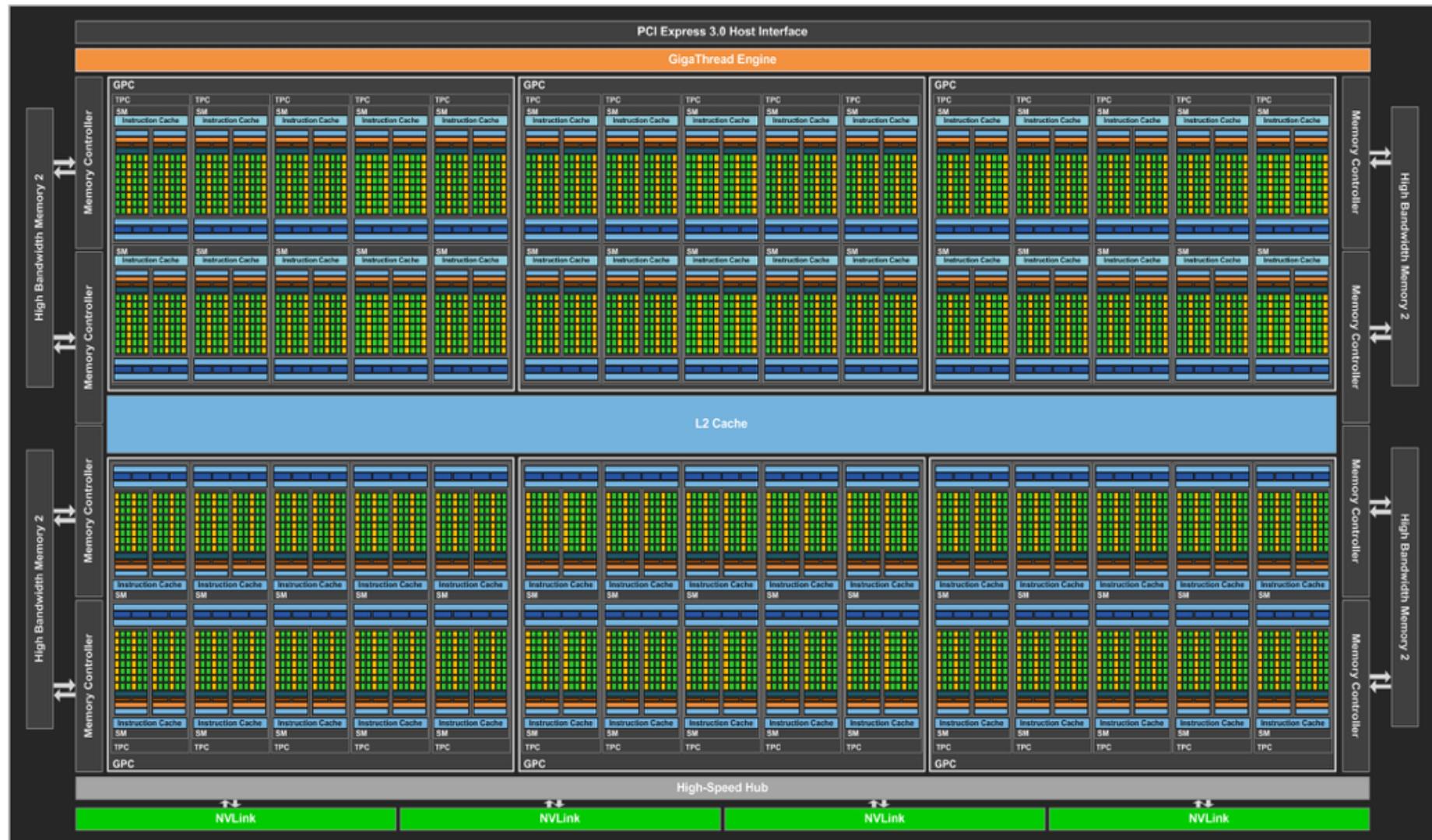
Primer modelo Tesla para Pascal: P100. y comparativa con las 2 generaciones previas

	Tesla K40 (Kepler)	Tesla M40 (Maxwell)	P100 & NV-link	P100 & PCI-e
Fecha de lanzamiento	2012	Noviembre, 2015	Abril, 2016	
Transistores	7.1 B @ 28 nm.	8 B @ 28 nm.	15.3 B @ 16 nm. FinFET (610 mm ²)	
Multiprocesadores	15	24	56	
Cores fp32 / Multiproc.	192	128	64	
Cores fp32 / GPU	2880	3072	3584	
Cores fp64 / Multiproc.	64	4	32	
Cores fp64 / GPU	960 (1/3 fp32)	96 (1/32 fp32)	1792 (1/2 fp32)	
Frecuencia de reloj	745,810,875 MHz	948,1114 MHz	1328, 1480 MHz	1126, 1303 MHz
Consumo energético	235 W.	250 W.	300 W.	250 W.
Rendimiento pico (DP)	1680 GFLOPS	213 GFLOPS	5304 GFLOPS	4670 GFLOPS
Tamaño de la caché L2	1536 KB	3072 KB	4096 KB	
Memoria: Interfaz	384-bit GDDR5	384-bit GDDR5	4096-bit HBM2	
Memoria: Tamaño	Hasta 12 GB	Hasta 24 GB	16 GB	
Memoria: Ancho banda	288 GB/s	288 GB/s	720 GB/s	

Los dos formatos: Zócalo PCI-e vs. Socket NVLINK (SXM2)



Disposición física de sus multiprocesadores, buses y controladores de memoria



El multiprocesador CUDA de Pascal





II. 4. Sexta generación: Volta (GVxxx)



Comparativa con generaciones anteriores en la gama Tesla

	K40 (Kepler)	M40 (Maxwell)	P100 (Pascal)	V100 (Volta)
GPU (chip)	GK110	GM200	GP100	GV100
Millones de transistores	7100	8000	15300	21100
Área de integración	551 mm ²	601 mm ²	610 mm ²	815 mm ²
Fabricación	28 nm.	28 nm.	16 nm. FinFET	12 nm. FinFET
Disipación de calor (TDP)	235 W.	250 W.	300 W.	300 W.
Número de cores fp32	2880 (15 x 192)	3072 (24 x 128)	3584 (56 x 64)	5120 (80 x 64)
Número de cores fp64	960	96	1792	2560
Frecuencia nominal y boost	745 y 875 MHz	948 y 1114 MHz	1328 y 1480 MHz	1370 y 1455 MHz
TFLOPS (fp16, fp32 y fp64)	No, 5.04, 1.68	No, 6.8, 2.1	21.2, 10.6, 5.3	30, 15, 7.5
Interfaz de memoria	GDDR5 de 384 bits		HBM2 de 4096 bits	
Memoria de vídeo	Hasta 12 GB	Hasta 24 GB	16 GB	16 ó 32 GB
Caché L2	1536 KB	3072 KB	4096 KB	6144 KB
Memoria compartida / SM	48 KB	96 KB	64 KB	Hasta 96 KB
Banco de registros / SM	65536	65536	65536	65536

Aspecto externo del producto comercial



La GPU GV100: 6 GPC, 84 SM, 42 TPC y 8 contr. de memoria de 512 bits (Tesla V100 sólo usa 80 SMs)



El multiprocesador de Volta

Cores:

- 64 int32 ("int").
- 64 fp32 ("float").
- 32 fp64 ("double").
- 8 unidades tensor.

Unidades de almacen.:

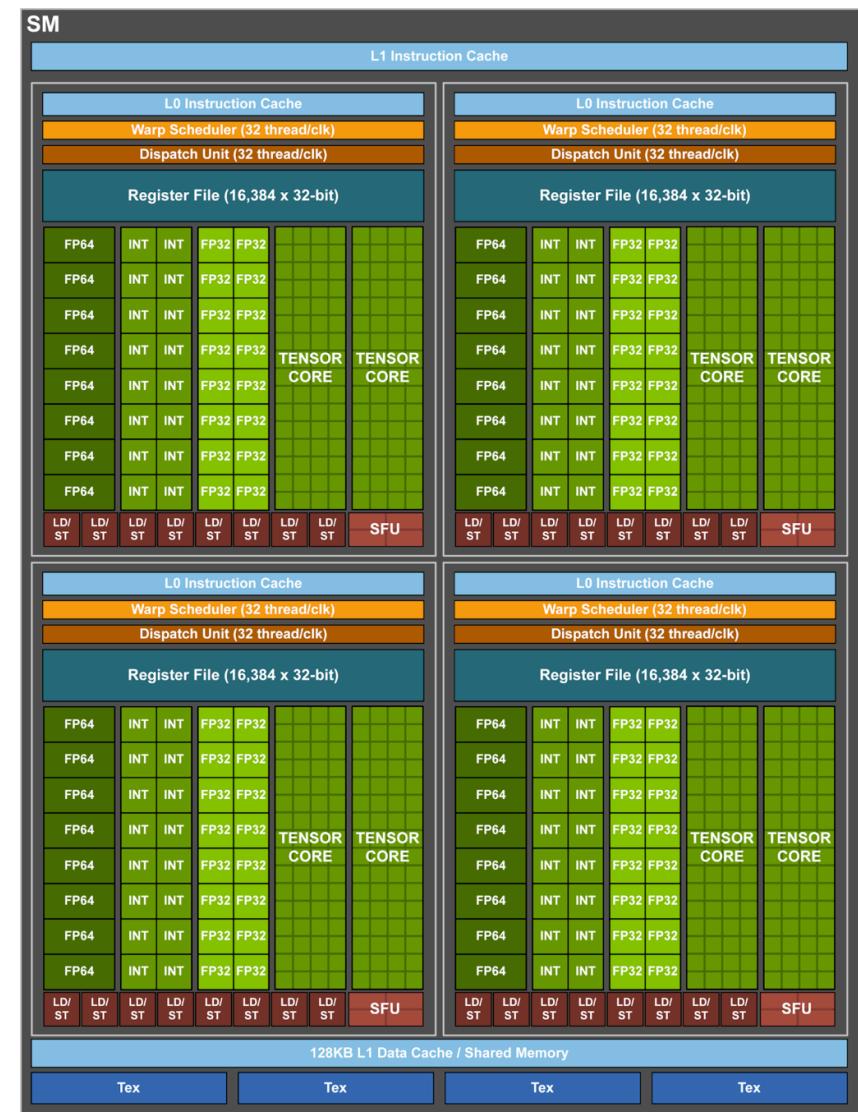
- 8 de carga/almacenamiento.
- 4 de texturas.

Memoria:

- 64K registros de 32 bits.
- Caché de instrucciones L0 (en lugar buffers de instrucción).
- 128 KB de caché L1 para datos y memoria compartida.

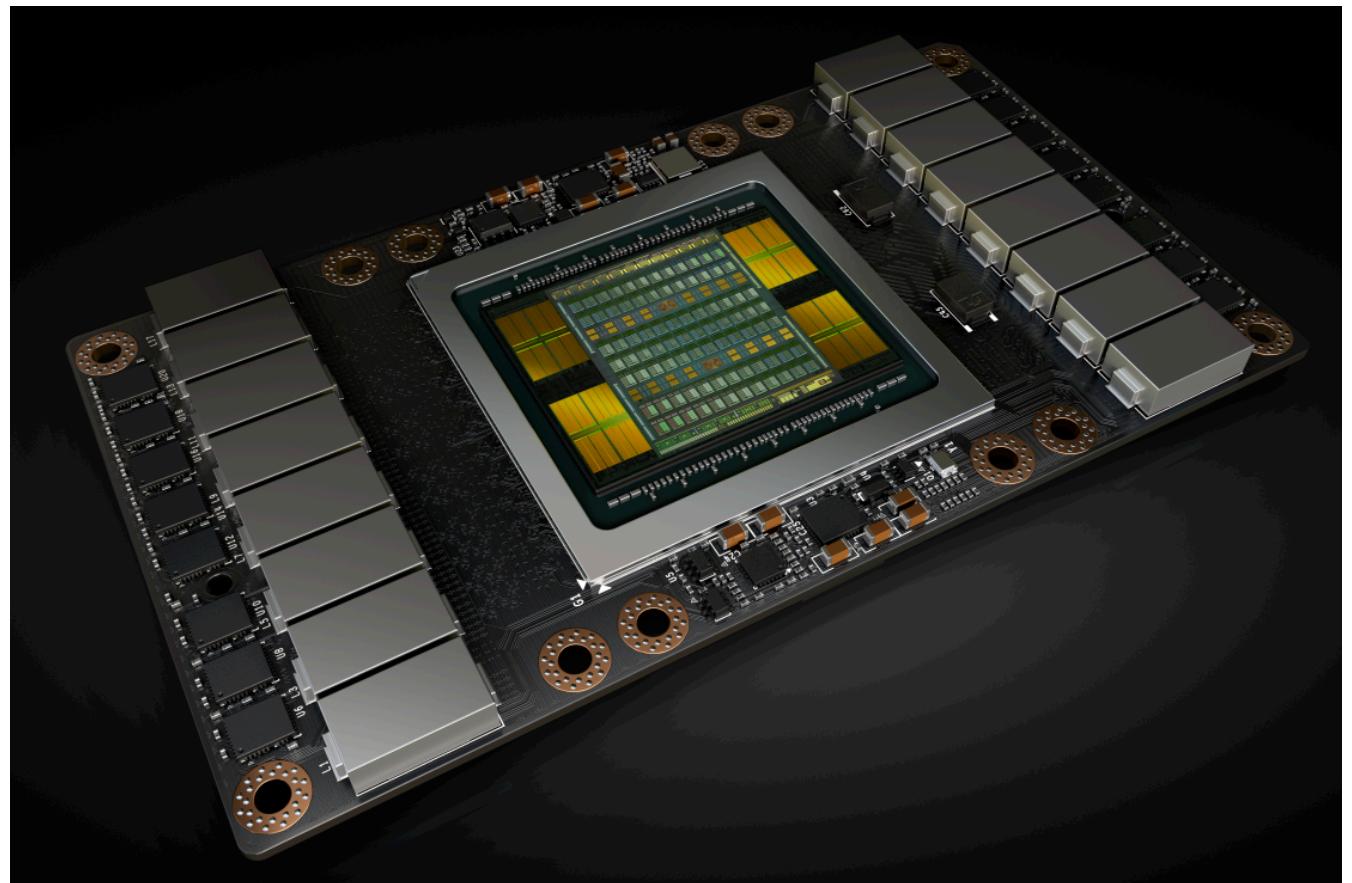
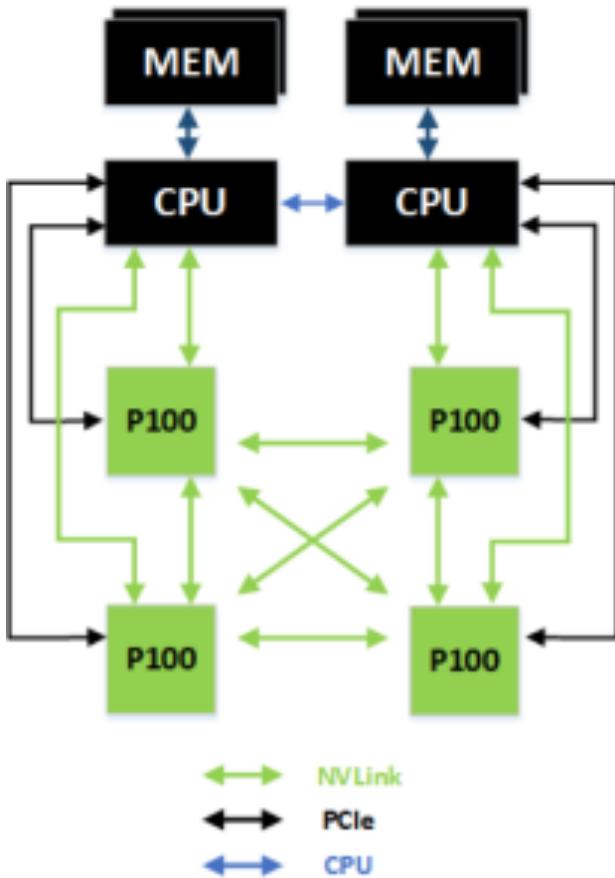


Evolución del multiprocesador: Desde Pascal a Volta



Interconexión: Zócalos y sockets

- Interconexión NV-link de 2^a generación con 6 enlaces de 25 GB/s. (frente a 4 enlaces de 20 GB/s. en Pascal).



Síntesis comparativa de Volta frente a Pascal

	GP100	GV100	Ratio
Rendimiento pico en P.F. 32 y 64 bits	10 & 5 TFLOPS	15 & 7.5 TFLOPS	1.5x
Entrenamiento para aprendizaje profundo	10 TFLOPS	120 TFLOPS	12x
Inferencias de aprendizaje profundo	21 TFLOPS	120 TFLOPS	6x
Cachés L1 (una por cada multiprocesador)	1.3 MB	10 MB	7.7x
Caché L2	4 MB	6 MB	1.5x
Ancho de banda HBM2	720 GB/s	900 GB/s	1.2x
Rendimiento sobre STREAM Triad (benchmark)	557 GB/s	855 GB/s	1.5x
Ancho de banda de la conexión NV-link	160 GB/s	300 GB/s	1.8x



II. 5. Séptima generación: Turing (TUxxx)



Comparativa con generaciones anteriores en la gama Tesla

	Tesla M40 (Maxwell)	Tesla P100 (Pascal)	Tesla V100 (Volta)	GeForce Titan RTX y Quadro RTX 6000 (Turing)
GPU (chip)	GM200	GP100	GV100	TU102
Millones de transistores	8000	15300	21100	18600
Área de integración	601 mm ²	610 mm ²	815 mm ²	754 mm ²
Fabricación	28 nm.	16 nm. FinFET	12 nm. FinFET	12 nm. FinFET
Disipación de calor (TDP)	250 W.	300 W.	300 W.	280 W.
Número de cores fp32	3072 (24 x 128)	3584 (56 x 64)	5120 (80 x 64)	4608 (72 x 64)
Número de cores fp64	96	1792	2560	144
Frecuencia base y boost	948 y 1114 MHz	1328 y 1480 MHz	1370 y 1455 MHz	1440 y 1770
TFLOPS (fp16, fp32, fp64)	No, 6.8, 2.1	21, 10.6, 5.3	30, 15, 7.5	32.6, 16.3, 0.51
Interfaz de memoria	GDDR5 de 384 bits	HBM2 de 4096 bits	GDDR6 de 384 bits @ 14 GHz	
Memoria de vídeo	Hasta 24 GB	16 GB	16 ó 32 GB	24 GB
Caché L2	3072 KB	4096 KB	6144 KB	6144 KB
Memoria compartida / SM	96 KB	64 KB	96 KB (de 128)	64 KB (de 96)

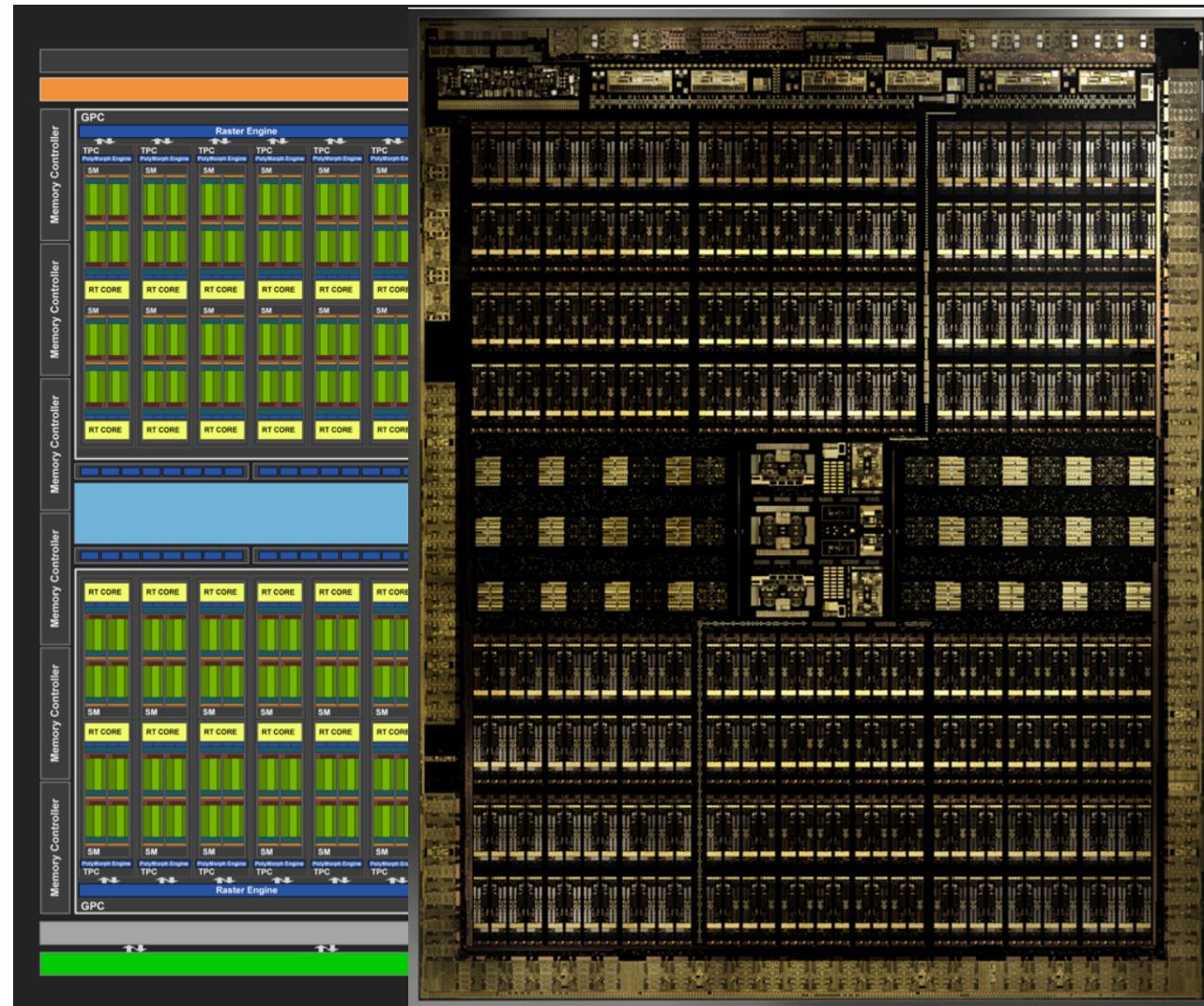
Los modelos comerciales GeForce RTX

	2060	2060 Super	2070	2070 Super	2080	2080 Super	2080 Ti	Titan RTX
Chip interno	TU-106	TU-106	TU-106	TU-104	TU-104	TU-104	TU-102	TU-102
Fecha de lanzamiento	Nov'16	Jul'19	Oct'18	Jul'19	Sep'18	Jul'19	Sep'18	Dec'18
Coste (en dólares EEUU)	349	399	499	499+	699	699+	999	2499
# multiprocesadores	30	34	36	40	46	48	68	72
# CUDA cores	1920	2176	2304	2560	2944	3072	4352	4608
Memoria GDDR6 (GB.)	6	8	8	8	8	8	11	24
Bus de memoria (bits)	192	256	256	256	256	256	352	384
Ancho de banda (GB/s.)	336	448	448	448	448	496	616	672

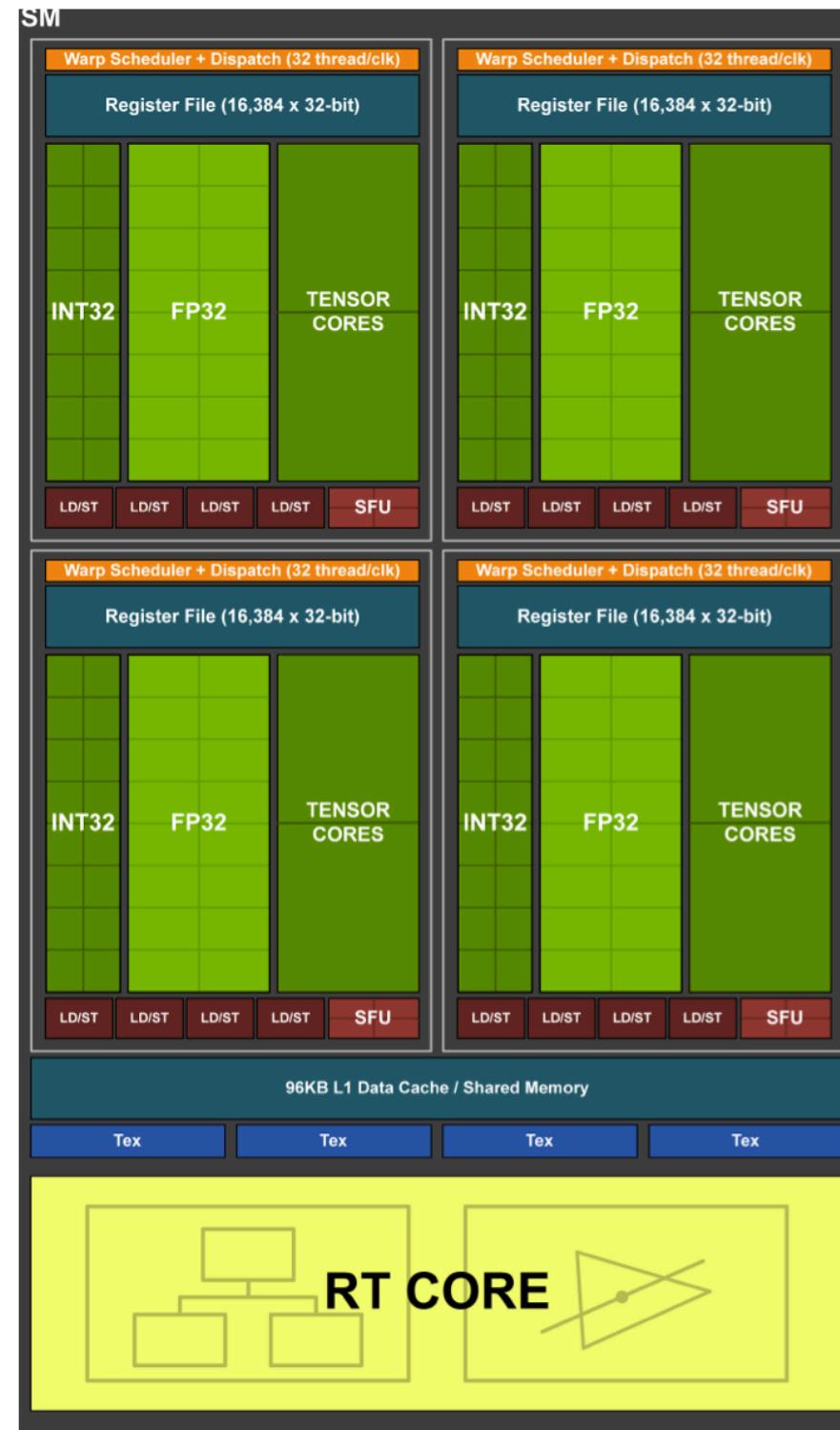
- Nuestro análisis se basará en el chip TU102, que es el más representativo de la generación Turing.

El chip TU-102 chip dotado de 72 multiprocesadores Turing (o SMs)

- 4608 cores CUDA (64 en cada SM).
- 576 cores Tensor (8 en cada SM).
- 72 cores Ray Tracing (1 por SM).
- 288 unidades de tectura (4 por SM).
- 12 controladores de memoria GDDR6 de 32 bits (384 bits en total).



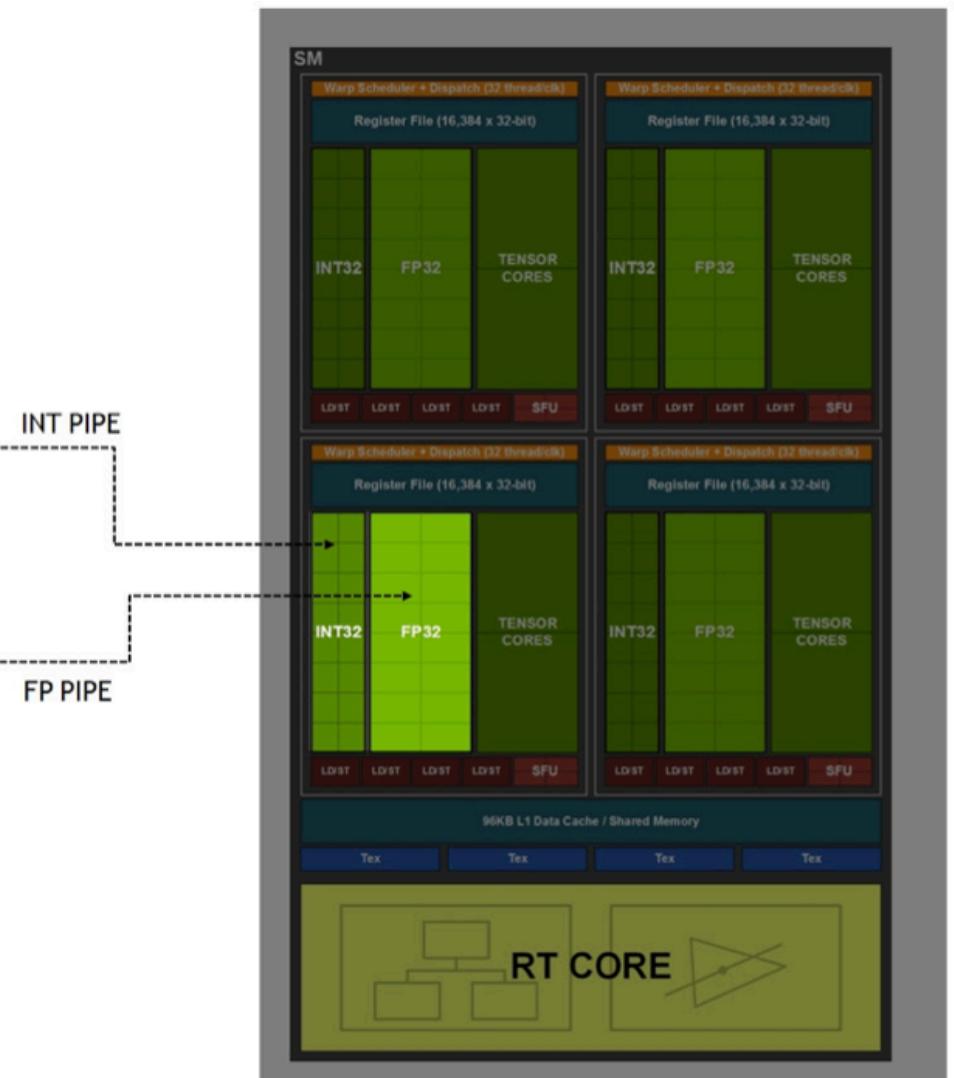
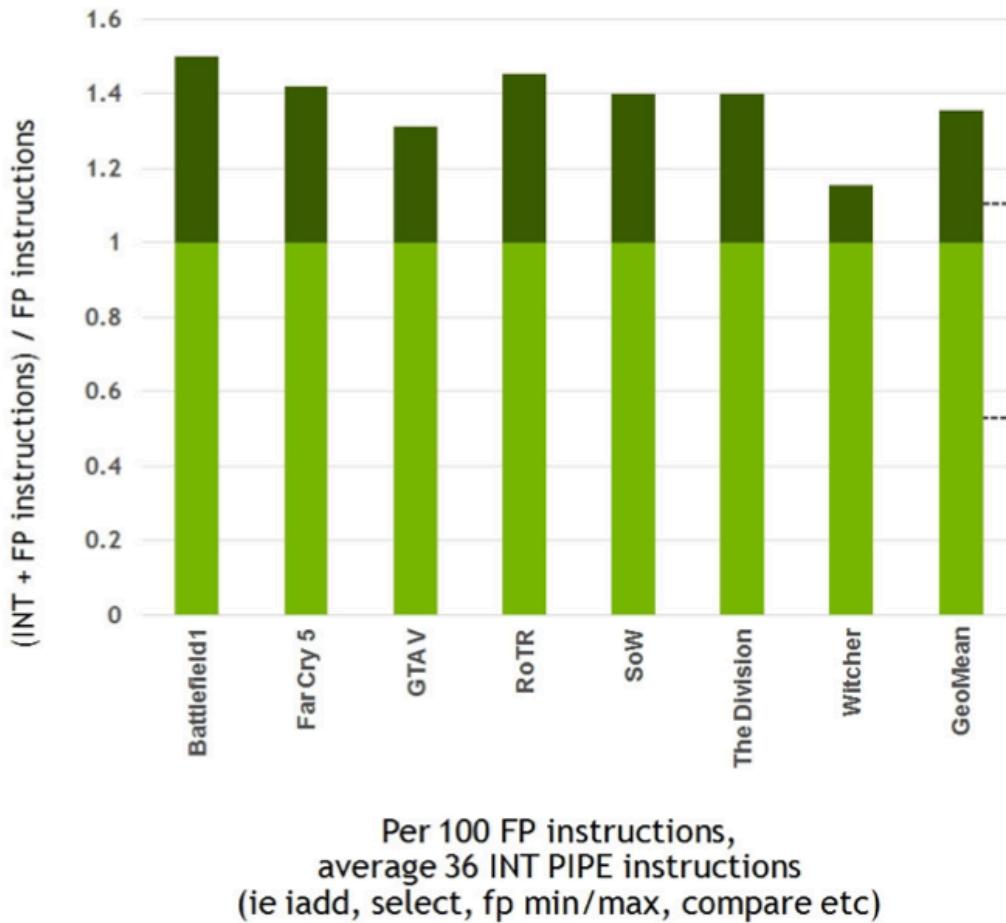
El multiprocesador Turing



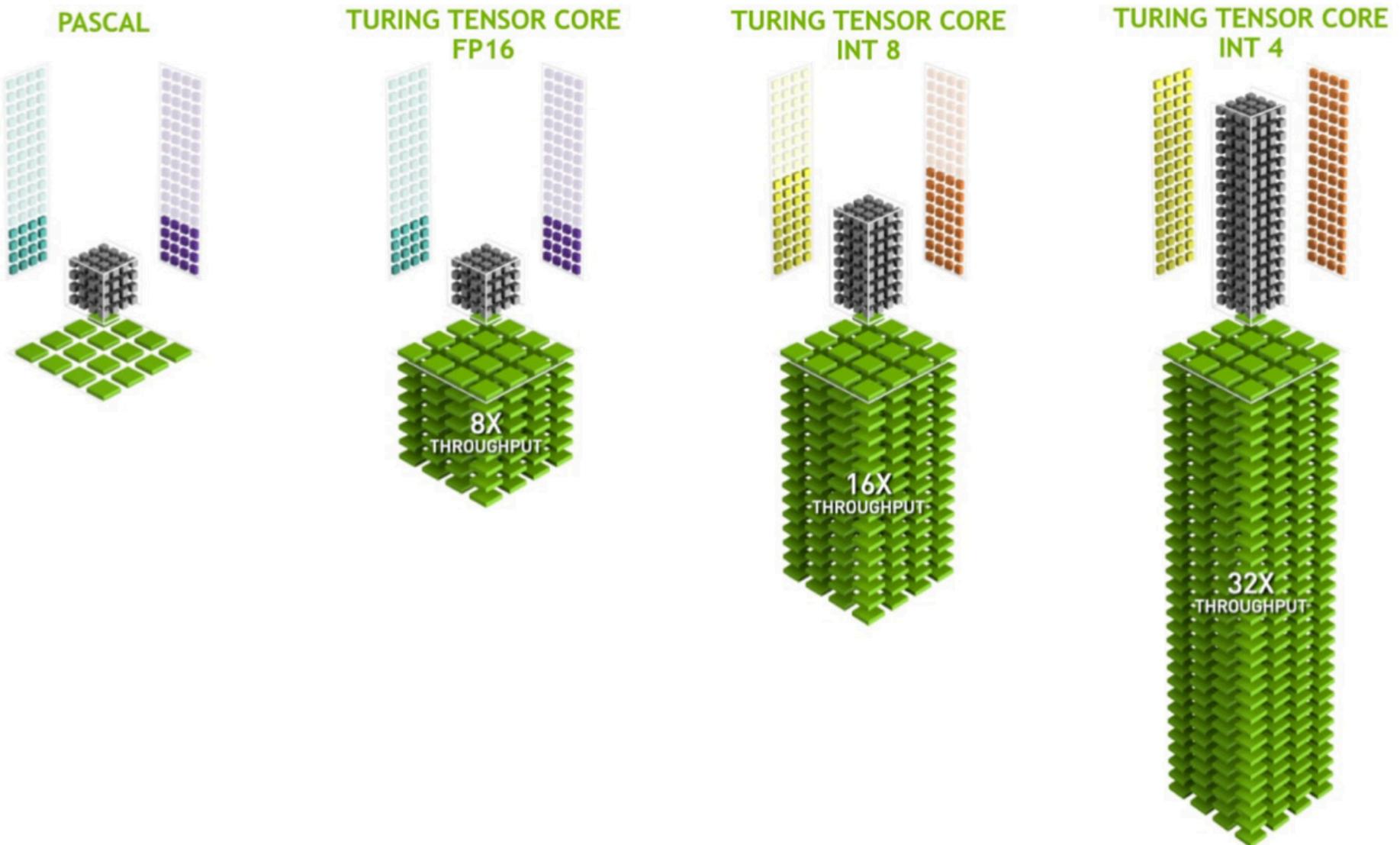
Los cambios arquitecturales más importantes

- El flujo de los datos enteros es independiente, pudiéndose ejecutar concurrentemente con las instrucciones de punto flotante. En las generaciones anteriores, ambos flujos de instrucciones se bloqueaban entre sí, no pudiendo emitirse de forma simultánea.
- Los tensor cores incorporan **precisión para INT8 e INT4** para utilizarlos en los procesos de inferencia de las redes neuronales donde se han mostrado efectivos.

CONCURRENT EXECUTION



Precisión INT8 y INT4 en los cores Tensor



Principales mejoras hardware

- El mayor salto arquitectural en una década. Grandes avances para:
 - Usuarios GeForce: Eficiencia y rendimiento en los juegos para PC.
 - Usuarios Quadro: Aplicaciones para los profesionales de los gráficos.
 - Usuarios GPGPU: Aprendizaje profundo y aceleración HPC.
- Nuevos aceleradores se fusionan con una renderización híbrida:
 - Ray tracing en tiempo real.
 - Inteligencia Artificial.
 - Rasterización y simulaciones.
- Nuevo multiprocesador para la GPU que:
 - Introduce nuevos cores para Ray Tracing.
 - Integra las unidades de memoria.
 - Mejora la ejecución de los sombreadores.

Principales mejoras software

- Los cores Tensor impulsan un conjunto de Servicios Neuronales:
 - Espectaculares efectos gráficos para video-juegos y profesionales.
 - Rápida inferencia para IA en sistemas basados en la nube.
- Nuevos cores Ray Tracing para renderizado en tiempo real, combinados con DirectML para IA y las APIs de Microsoft DirectX Raytracing (DXR) (desde principios de 2018 en adelante).
- Avances en los sombreadores para:
 - Mejorar su rendimiento.
 - Avanzar en la calidad de imagen.
 - Aportar nuevos niveles de complejidad geométrica.

Clarificando las frecuencias de reloj

Producto comercial	Tipo de reloj	GeForce		Quadro	
		GTX 1080 Ti [Pascal]	RTX 2080 Ti [Turing]	P6000 [Pascal]	GeForce Titan RTX / Quadro RTX 6000 / Quadro RTX 8000 [Turing]
Modelo de referencia	GPU Base	1480	1350	1506	1350 / 1395 / 1440
	GPU Boost	1582	1545	1645	1770
Founders Edition	GPU Base	1480	1350	1506	1455 (GeForce)
	GPU Boost	1582	1635	1645	1770

- Cifras en negrita denotan el rendimiento pico oficial (ver la diapositiva siguiente).
- Sin GPU Boost, se pierde un 21% de eficiencia en TODAS las ejecuciones.

Diseños de GPU que se traducen en modelos comerciales

- GeForce GTX: Para jugones y usuarios domésticos, punto de partida para científicos CUDA y usuarios de propósito general.
 - GTX 1050 Ti (4 GB): Alternativa de bajo coste (<200€).
 - GTX 1060: Mejor ratio rendimiento/coste (3 GB 280€, 6 GB 380€).
 - GTX 1070: Punto de entrada para los que quieran experimentar con DL.
 - GTX 1080: Rentable, pero cara.
- Quadro: Para los profesionales gráficos. Cara. Mercado estable.
- Titan: Variante de gama alta de la GeForce.
 - Mejor ratio rendimiento/coste para los usuarios HPC (approx. 1000 €).
- Tesla: Para HPC y aplicaciones científicas muy exigentes.
 - Cuando la memoria (16-32 GB HBM2) importa y el dinero no.
- GeForce RTX: Para la serie Turing. El chip TU102 es el modelo estandarte que tiene la 2080 Ti (1200€) y la Quadro RTX 6000.

Turing (GeForce RTX) frente a Pascal (GeForce GTX)

● Rasgos:

- RTX contiene GDDR6, Tensors, hasta 5120 cores.
- GTX contiene GDDR5, NO Tensors, hasta 3584 cores.

● Para aprovechar las RTX para jugar:

- Aumentar la resolución (4K): Para beneficiarse del ancho de banda.
- Usar HDR (High Dynamic Range): Para explotar mejor FP16.
- Usar rasgos IA en gráficos (DLSS - Deep Learning Super Sampling), Ray Tracing, y más en el futuro): Para utilizar los Tensor cores.

● Si prefieres cómputo HPC en vez de jugar:

- Usa los entornos Deep Learning: Caffe, mxnet, TensorFlow, Pytorch.
- Usa las librerías Deep Learning: CUDA 9 cuBLAS y cuDNN.
- Usa CUDA C++ API.



II. 6. Síntesis generacional

Escalabilidad de la arquitectura: Síntesis de cuatro generaciones (2006-2015)

	Tesla		Fermi		Kepler				Maxwell	
Arquitectura	G80	GT200	GF100	GF104	GK104 (K10)	GK110 (K20X)	GK110 (K40)	GK210 (K80)	GM107 (GTX750)	GM204 (GTX980)
Marco temporal	2006 /07	2008 /09	2010	2011	2012	2013	2013 /14	2014	2014 /15	2014 /15
CUDA Compute Capability	1.0	1.3	2.0	2.1	3.0	3.5	3.5	3.7	5.0	5.2
N (multiprocs.)	16	30	16	7	8	14	15	30	5	16
M (cores/multip.)	8	8	32	48	192	192	192	192	128	128
Número de cores	128	240	512	336	1536	2688	2880	5760	640	2048

Las nuevas generaciones (2016-2019)

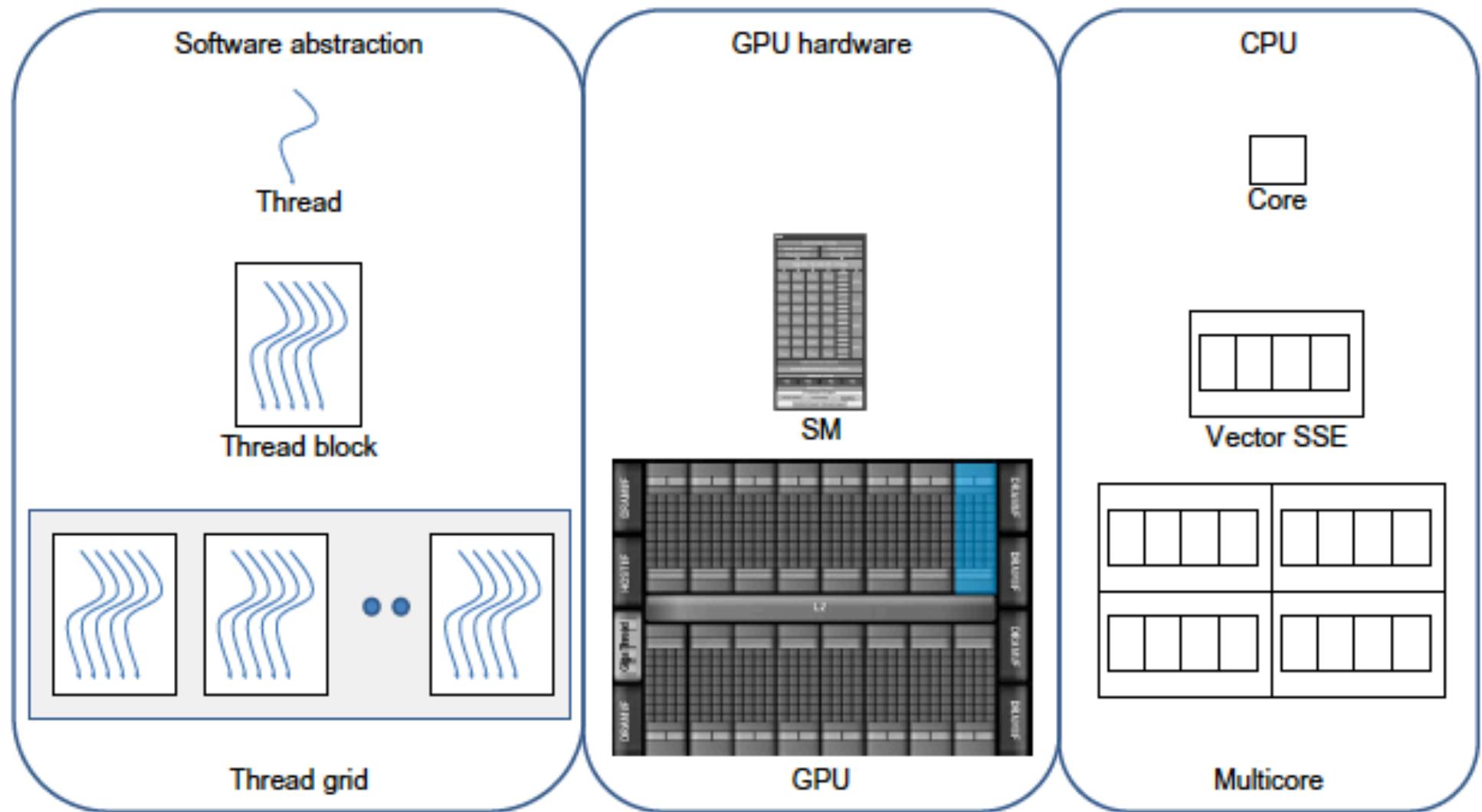
	Maxwell			Pascal			Volta	Turing
Arquitectura	GM107 (GTX750)	GM204 (GTX980)	GM200 (Titan X) (Tesla M40)	GP104 (GTX1080)	GP100 (Titan X) (Tesla P100)	GP102 (Tesla P40)	GV100 (Tesla V100)	TU102 (Titan RTX)
Marco temporal	2014 /15	2014 /15	2016	2016	2017	2017	2018	2019
CUDA Compute Capability	5.0	5.2	5.3	6.0	6.0	6.1	7.0	7.5
N (multiprocs.)	5	16	24	40	56	60	80	72
M (cores/multip.)	128	128	128	64	64	64	64	64
Número de cores	640	2048	3072	2560	3584	3840	5120	4608



III. Programación



Comparativa con la CPU: Dos formas de construir supercomputadores



De la programación de hilos POSIX en CPU a la programación de hilos CUDA en GPU

POSIX-threads en CPU

```
#define NUM_THREADS 16
void *mifunc (void *threadId)
{
    int tid = (int) threadId;
    float result = sin(tid) * tan(tid);
    pthread_exit(NULL);
}

void main()
{
    int t;
    for (t=0; t<NUM_THREADS; t++)
        pthread_create(NULL,NULL,mifunc,t);
    pthread_exit(NULL);
}
```

CUDA en GPU, seguido del código host en CPU

```
#define NUM_BLOCKS 1
#define BLOCKSIZE 16
__global__ void mikernel()
{
    int tid = threadIdx.x;
    float result = sin(tid) * tan(tid);
}

void main()
{
    dim3 dimGrid (NUM_BLOCKS);
    dim3 dimBlock (BLOCKSIZE);
    mikernel<<<dimGrid, dimBlock>>>();
    return EXIT_SUCCESS;
}
```

Configuración 2D: Malla de 2x2 bloques de 4 hilos

```
#define NUM_BLX 2
#define NUM_BLY 2
#define BLOCKSIZE 4
__global__ void mikernel()
{
    int bid=blockIdx.x*gridDim.y+blockIdx.y;
    int tid=bid*blockDim.x+ threadIdx.x;
    float result = sin(tid) * tan(tid);
}

void main()
{
    dim3 dimGrid (NUM_BLX, NUM_BLY);
    dim3 dimBlock(BLOCKSIZE);
    mikernel<<<dimGrid, dimBlock>>>();
    return EXIT_SUCCESS;
}
```

El modelo de programación CUDA

- La GPU (device) ofrece a la CPU (host) la visión de un coprocesador altamente ramificado en hilos.
 - Que tiene su propia memoria DRAM.
 - Donde los hilos se ejecutan en paralelo sobre los núcleos (cores o stream processors) de un multiprocesador.



- Los hilos de CUDA son **extremadamente ligeros**.
 - Se crean en un tiempo muy efímero.
 - La comutación de contexto es inmediata.
- Objetivo del programador: Declarar miles de hilos, que la GPU necesita para lograr rendimiento y escalabilidad.

Estructura de un programa CUDA

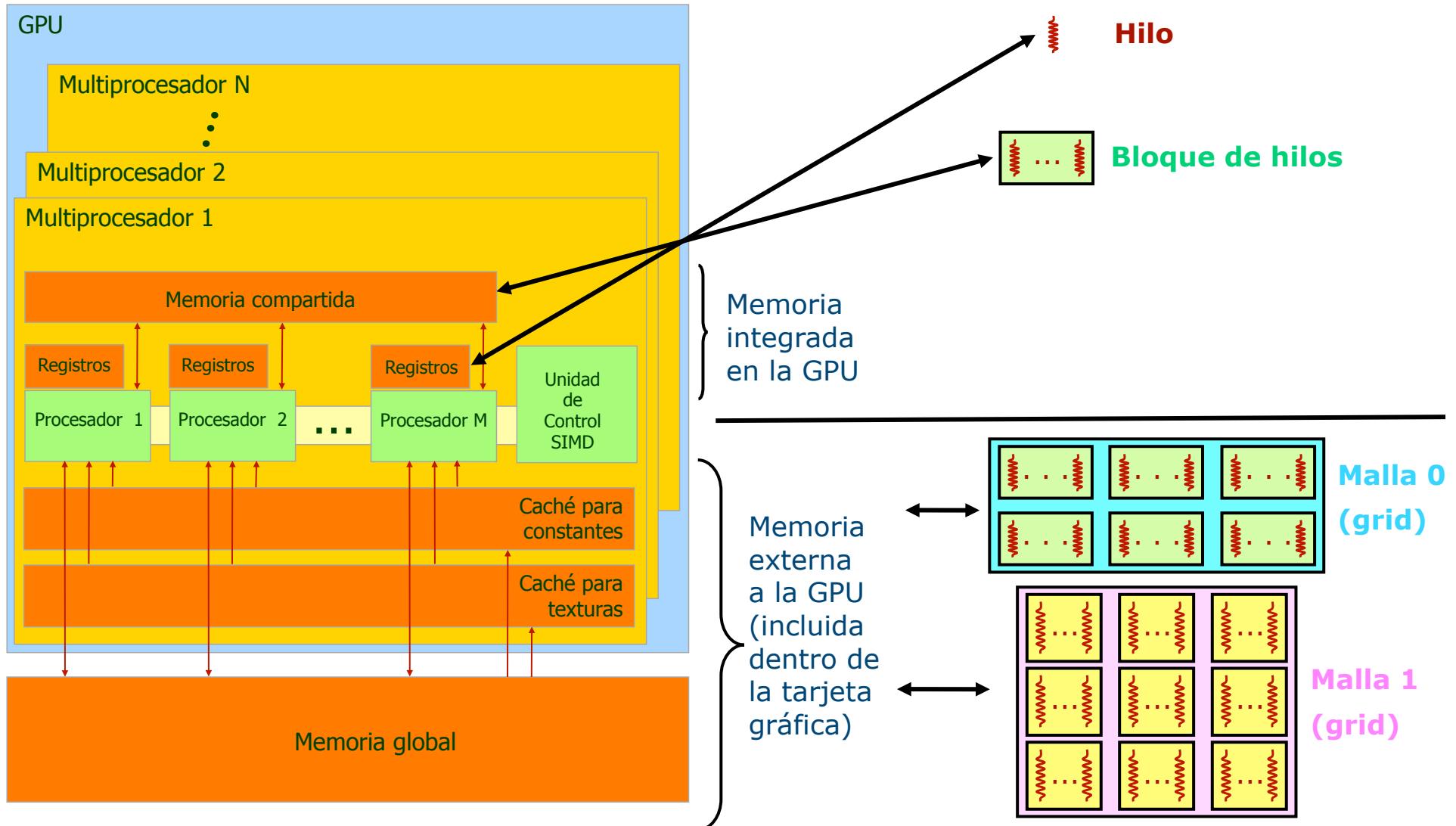
- Cada multiprocesador procesa lotes de bloques, uno detrás de otro
 - Bloques activos = los bloques procesados por un multiprocesador en un lote.
 - Hilos activos = todos los que provienen de los bloques que se encuentren activos.
- Los registros y la memoria compartida de un multiprocesador se reparten entre sus hilos activos. Para un kernel dado, el número de bloques activos depende de:
 - El número de registros requeridos por el kernel.
 - La memoria compartida consumida por el kernel.

Conceptos básicos

Los programadores se enfrentan al reto de exponer el paralelismo para múltiples cores y múltiples hilos por core. Para ello, deben usar los siguientes elementos:

- ➊ Dispositivo = GPU = Conjunto de multiprocesadores.
- ➋ Multiprocesador = Conjunto de procesadores y memoria compartida.
- ➌ Kernel = Programa listo para ser ejecutado en la GPU.
- ➍ Malla (grid) = Conjunto de bloques cuya compleción ejecuta un kernel.
- ➎ Bloque [de hilos] = Grupo de hilos SIMD que:
 - ➏ Ejecutan un kernel delimitando su dominio de datos según su threadID y blockID.
 - ➏ Pueden comunicarse a través de la memoria compartida del multiprocesador.
 - ➏ Tamaño del warp = 32. Esta es la resolución del planificador para emitir hilos por grupos a las unidades de ejecución.

Relación entre el hardware y el software desde la perspectiva del acceso a memoria



Recursos y limitaciones según la GPU que utilicemos para programar (CCC)

	CUDA Compute Capability (CCC)							Limi-tación	Im-pacto
	1.0, 1.1	1.2, 1.3	2.0, 2.1	3.0	3.2*	3.5	3.7		
fp32 cores / Multip.	8	8	32	192	192	192	192	HW	Escala-bilidad
fp64 cores / Multip.	0	0	16	64	64	64	64		
Hilos / Warp	32	32	32	32	32	32	32	SW	Ritmo de salida de datos
Bloques / Multiprocesador	8	8	8	16	16	16	16		
Hilos / Bloque	512	512	1024	1024	1024	1024	1024	SW	Paralelismo
Hilos / Multiprocesador	768	1024	1536	2048	2048	2048	2048		
Regs. de 32 bits / Multip.	8K	16K	32K	64K	32K	64K	128K	HW	Conjunto de trabajo
Mem. compartida / Bloque	16K	16K	48 KB	48 KB	48 KB	48 KB	48 KB		
Mem. compartida / Multip.	16K	16K	48 KB	48 KB	48 KB	48 KB	112 KB		

(*) Establecida sólo para dispositivos Tegra y Jetson.

Recursos y limitaciones según la GPU que utilicemos para programar (CCC) (cont.)

ccc	5.0	5.2	5.3*	6.0	6.1	6.2*	7.0, 7.2*	7.5
fp16 cores / Multip.	0	128	128	64	64	64	64	64
fp32 cores / Multip.	128	128	128	64	64	64	64	64
fp64 cores / Multip.	4	4	4	32	32	32	32	2
Tensor cores / Multip.	0	0	0	0	0	0	8	8
Bloques / Multiprocesador	32	32	32	32	32	32	32	16
Hilos / Bloque	1024	1024	1024	1024	1024	1024	1024	1024
Hilos / Multiprocesador	2048	2048	2048	2048	2048	2048	2048	1024
Regs. de 32 bits / Bloque	64K	64K	32K	64K	64K	32K	64K	64K
Regs. de 32 bits / Multip.	64K	64K						
Mem. compartida / Bloque	48 K	48K	48K	48K	48K	48K	48/96K	64K
Mem. compartida / Multip.	64K	96K	64K	64K	96K	64K	96K (de 128)	64K (de 96)

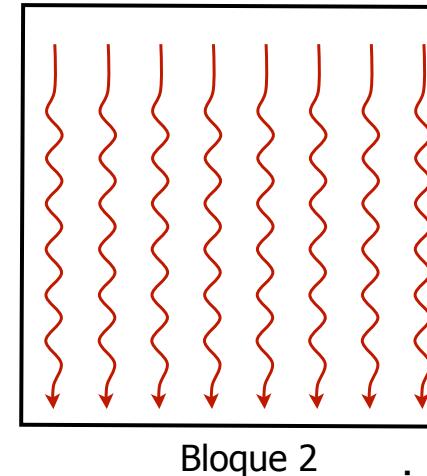
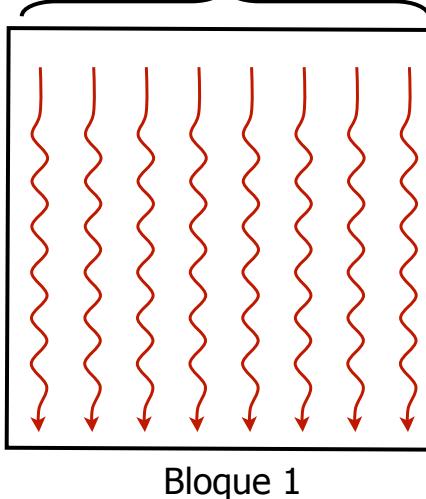
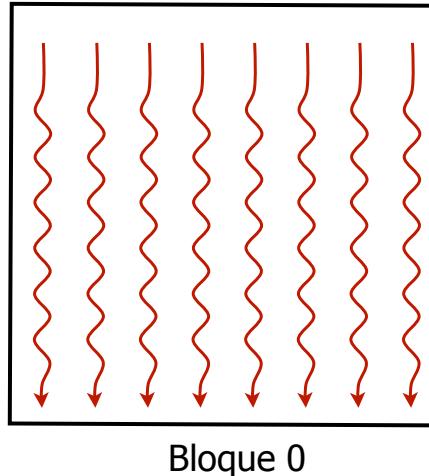
(*) Establecida sólo para dispositivos Tegra y Jetson.

La relación entre CCC y el mercado de GPUs

CCC	"Code names"	Modelos vinculados a CUDA	Series comerciales	Marco temporal (año)	Proceso de fabricación @ TSMC
1.0	G80	Muchos	8xxx	2006-07	90 nm.
1.1	G84,6 G92,4,6,8	Muchos	8xxx/9xxx	2007-09	80, 65, 55 nm.
1.2	GT215,6,8	Pocos	2xx/3xx	2009-10	40 nm.
1.3	GT200	Muchos	2xx	2008-09	65, 55 nm.
2.0	GF100, GF110	Muchísimos	4xx/5xx	2010-11	40 nm.
2.1	GF104,6,8, GF114,6,8,9	Pocos	4xx/5xx/6xx/7xx	2010-13	40 nm.
3.0	GK104,6,7	Bastantes	6xx/7xx	2012-14	28 nm.
3.5	GK110, GK208	Muchísimos	6xx/7xx/Titan	2013-14	28 nm.
3.7	GK210 (2xGK110)	Muy pocos	Tesla K80	2014	28 nm.
5.0	GM107,8	Muchos	7xx/8xx/9xx	2014-15	28 nm.
5.2	GM200,4,6	Muchos	9xx/Titan	2014-15	28 nm.
6.0	GP100	Todos	Tesla P100	2016-17	16 nm. finFET
6.1	GP102,4,6,7,8	Muchos	10xx / Titan X	2017	16 nm. finFET
7.0	GV100	Todos	Titan V / V100	2017-18	12 nm. FFN
7.5	TU102,4,6, TU116-7	Todos	GeForce RTX, GTX 1650-60	2019-20	12 nm. FFN

Bloques e hilos en GPU

Límite en Kepler/Maxwell: 1024 hilos por bloque, 2048 hilos por multiprocesador



Malla 0 [Límite en Kepler/Maxwell: 4G bloques por malla]

Los bloques se asignan a los multiprocesadores

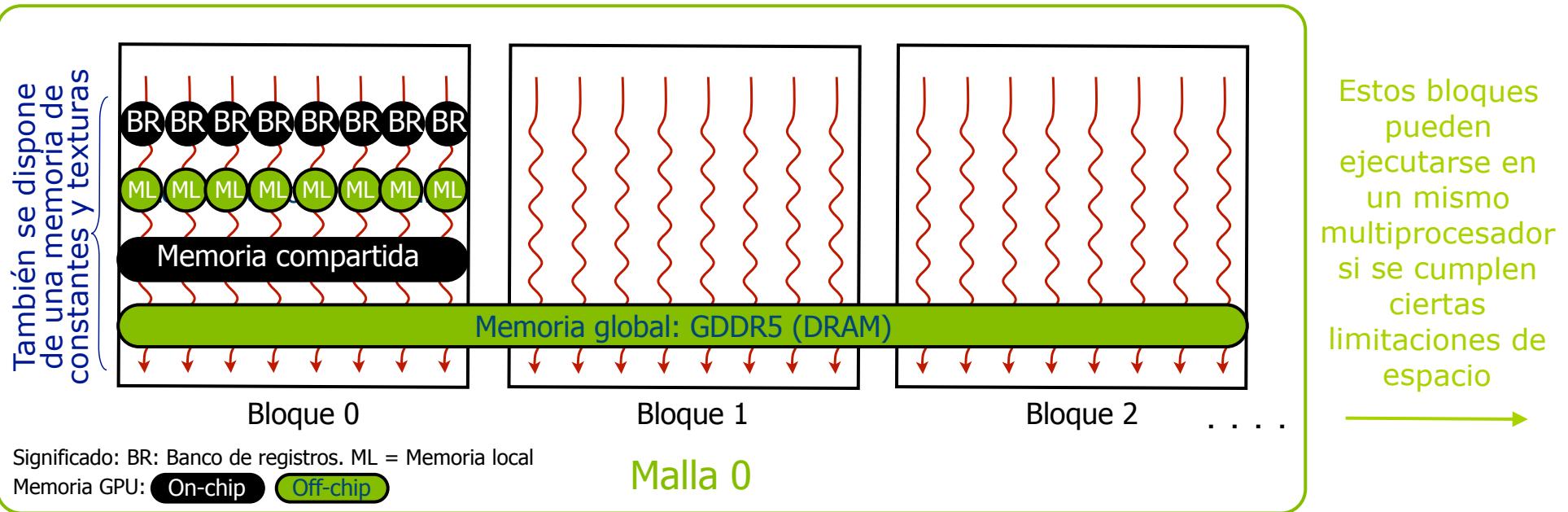
[Límite:
16-32 bloques concurrentes
en cada multiprocesador]

- Los hilos se asocian a los multiprocesadores en bloques, y se asignan a los cores en átomos de 32 llamados warps.
- Los hilos de un bloque comparten la memoria compartida y pueden sincronizarse mediante llamadas a `syncthreads()`.

Ejemplos de restricciones de parallelismo en Maxwell al tratar de maximizar concurrencia

- ➊ Límites para un multiprocesador SMM: [1] 32 bloques, [2] 1024 hilos/bloque y [3] 2048 hilos en total.
- ➋ 1 bloque de 2048 hilos. No lo permite [2].
- ➌ 2 bloques de 1024 hilos. Posible en el mismo multiproc.
- ➍ 4 bloques de 512 hilos. Posible en el mismo multiproc.
- ➎ 4 bloques de 1024 hilos. No lo permite [3] en el mismo multiprocesador, pero posible usando dos multiprocs.
- ➏ 8 bloques de 256 hilos. Posible en el mismo multiproc.
- ➐ 256 bloques de 8 hilos. No lo permite [1] en el mismo multiprocesador, posible usando 8 multiprocesadores.

La memoria en la GPU: Ámbito y aplicación



- Los hilos de un bloque pueden comunicarse a través de la memoria compartida del multiprocesador para trabajar de forma más cooperativa y veloz.
- La memoria global es la única visible a hilos, bloques y kernels.

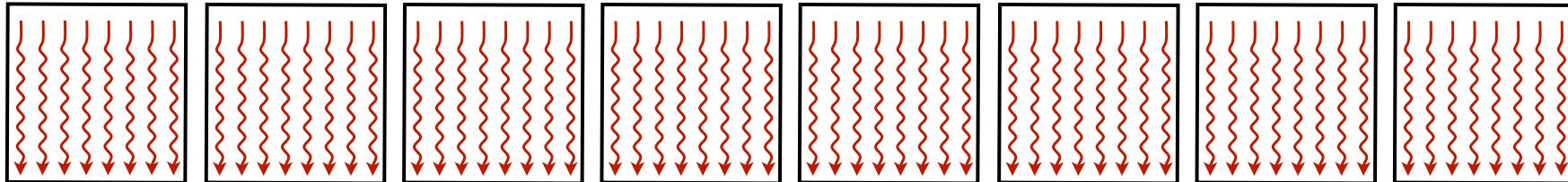
Jugando con las restricciones de memoria en Maxwell para maximizar el uso de recursos

- ➊ Límites dentro de un multiprocesador SMM: 64 Kregs. y 96 KB. de memoria compartida. Así:
 - ➊ Para que un **segundo bloque** se ejecute en el mismo multiprocesador, cada bloque debe usar a lo sumo 32 Kregistros y 48 Kbytes de memoria compartida.
 - ➋ Para que un **tercer bloque** se ejecute en el mismo multiprocesador, cada bloque debe usar a lo sumo 21.3 Kregistros y 32 Kbytes de memoria compartida.
- ➌ ...y así sucesivamente. Cuanto menos memoria usemos, mayor concurrencia para la ejecución de bloques.
- ➍ El programador debe establecer el compromiso adecuado entre apostar por memoria o paralelismo en cada código.

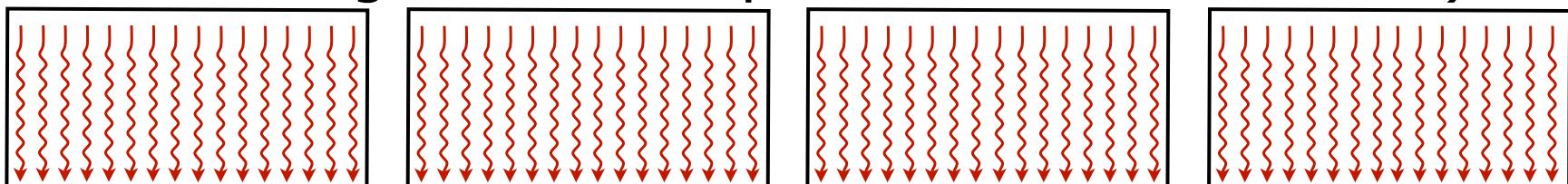
Pensando en pequeño: Particionamiento 1D de un vector de 64 elementos

- Recordar: Mejor paralelismo de grano fino (asignar un dato a cada hilo). Despliegue de bloques e hilos:

8 bloques de 8 hilos cada uno. Riesgo en bloques pequeños: Desperdiciar paralelismo si se alcanza el máximo de 16-32 bloques en cada multiprocesador con pocos hilos en total.



4 bloques de 16 hilos cada uno. Riesgo en bloques grandes: Estrangular el espacio de cada hilo (la memoria compartida y el banco de registros se comparte entre todos los hilos).

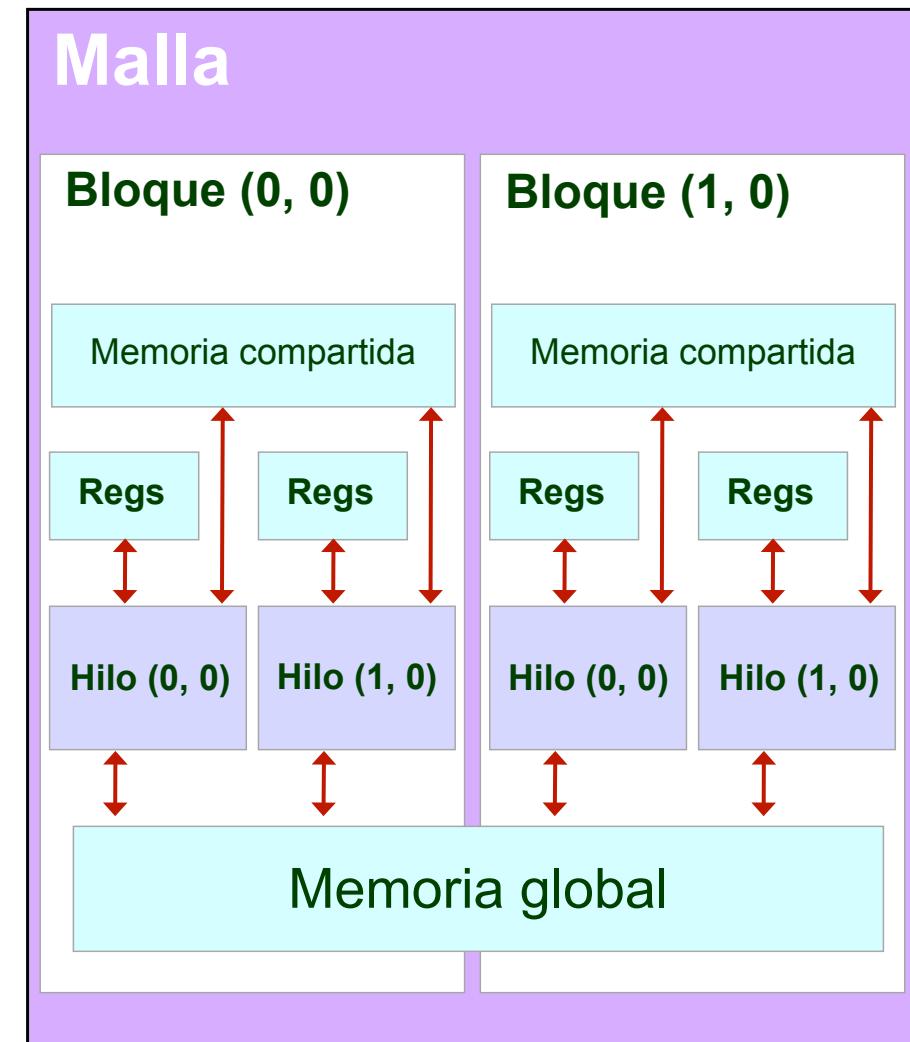


Pensando a lo grande: Particionamiento 1D de un vector de 64 millones de elementos

- Máximo número de hilos por bloque: 1024.
- Máximo número de bloques:
 - 64K en Fermi.
 - 4G en Kepler/Maxwell.
- Tamaños más grandes para las estructuras de datos sólo pueden implementarse mediante un número ingente de bloques (si queremos preservar el paralelismo fino).
- Posibilidades a elegir:
 - 64K bloques de 1024 hilos cada uno (tope para Fermi).
 - 128K bloques de 512 hilos cada uno (ya no es factible en Fermi).
 - 256K bloques de 256 hilos cada uno (ya no es factible en Fermi).
 - ... y así sucesivamente.

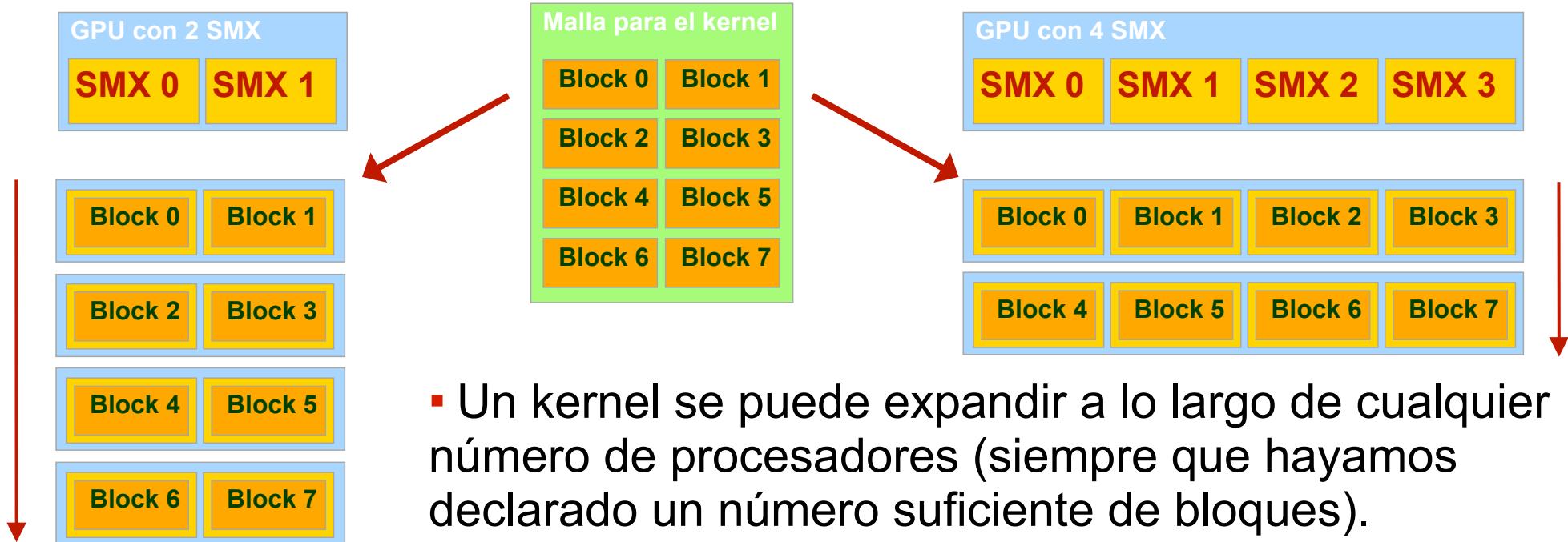
Recopilando sobre kernels, bloques y parallelismo

- Los kernels se lanzan en mallas.
- Un bloque se ejecuta en un multiprocesador (SMX/SMM).
 - El bloque no migra.
- Varios bloques pueden residir concurrentemente en un SMX/SMM.
 - Con ciertas limitaciones:
 - Hasta **16/32** bloques concurrentes.
 - Hasta **1024** hilos en cada bloque.
 - Hasta **2048** hilos en cada SMX/SMM.
 - Otras limitaciones entran en juego debido al uso conjunto de la memoria compartida y el banco de registros según hemos visto hace 3 diapositivas.



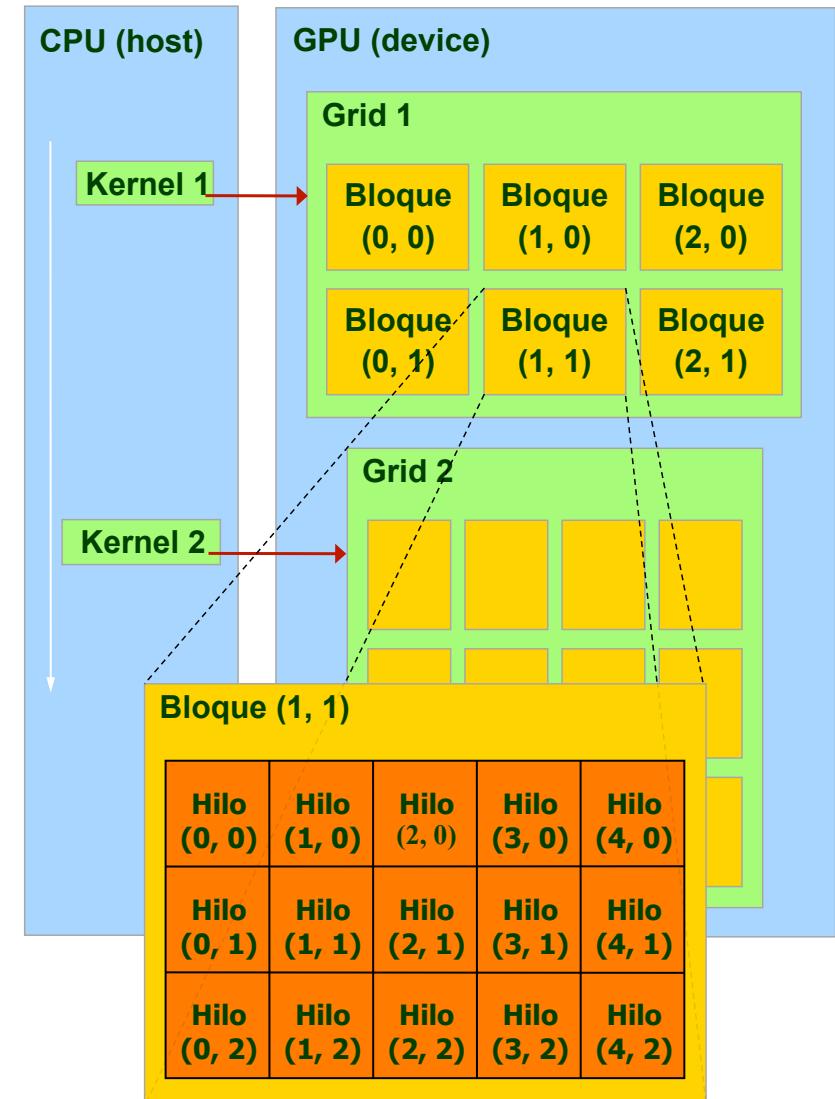
Escalabilidad transparente

- Dado que los bloques no se pueden sincronizar:
 - El hardware tiene libertad para planificar la ejecución de un bloque en cualquier multiprocesador.
 - Los bloques pueden ejecutarse secuencialmente o concurrentemente según la disponibilidad de recursos.



Particionamiento de computaciones y datos

- Un bloque de hilos es un lote de hilos que pueden cooperar:
 - Compartiendo datos a través de memoria compartida.
 - Sincronizando su ejecución para acceder a memoria sin conflictos.
- Un kernel se ejecuta como una malla o grid 1D ó 2D de bloques de hilos 1D, 2D ó 3D.
- Los hilos y los bloques tienen IDs para que cada hilo pueda acotar sobre qué datos trabaja, y simplificar el direccionamiento al procesar datos multidimensionales.



Espacios de memoria

- ➊ La CPU y la GPU tiene espacios de memoria separados:
 - ➌ Para comunicar ambos procesadores, se utiliza el bus PCI-express.
 - ➌ En la GPU se utilizan funciones para alojar memoria y copiar datos de la CPU de forma similar a como la CPU procede en lenguaje C (`malloc` para alojar y `free` para liberar).
- ➋ Los punteros son sólo direcciones:
 - ➌ No se puede conocer a través del valor de un puntero si la dirección pertenece al espacio de la CPU o al de la GPU.
 - ➌ Hay que ir con mucha cautela a la hora de acceder a los datos a través de punteros, ya que si un dato de la CPU trata de accederse desde la GPU o viceversa, el programa fallará (**esta situación cambia a partir de CUDA 6.0 con la memoria unificada**).



IV. Sintaxis





IV. 1. Elementos básicos



CUDA es C con algunas palabras clave más. Un ejemplo preliminar

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

Código C estándar

Código CUDA equivalente de ejecución paralela en GPU:

```
__global__ void saxpy_paralelo(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 numero_de_bloques = (n + 255) / 256;
saxpy_paralelo<<<numero_de_bloques, 256>>>(n, 2.0, x, y);
```

Lista de extensiones sobre el lenguaje C

- Modificadores para las variables (type qualifiers):

- global, device, shared, local, constant.

- Palabras clave (keywords):

- threadIdx, blockIdx, blockDim, gridDim.

- Funciones intrínsecas (intrinsics):

- __syncthreads

- API en tiempo de ejecución:

- Memoria, símbolos, gestión de la ejecución.

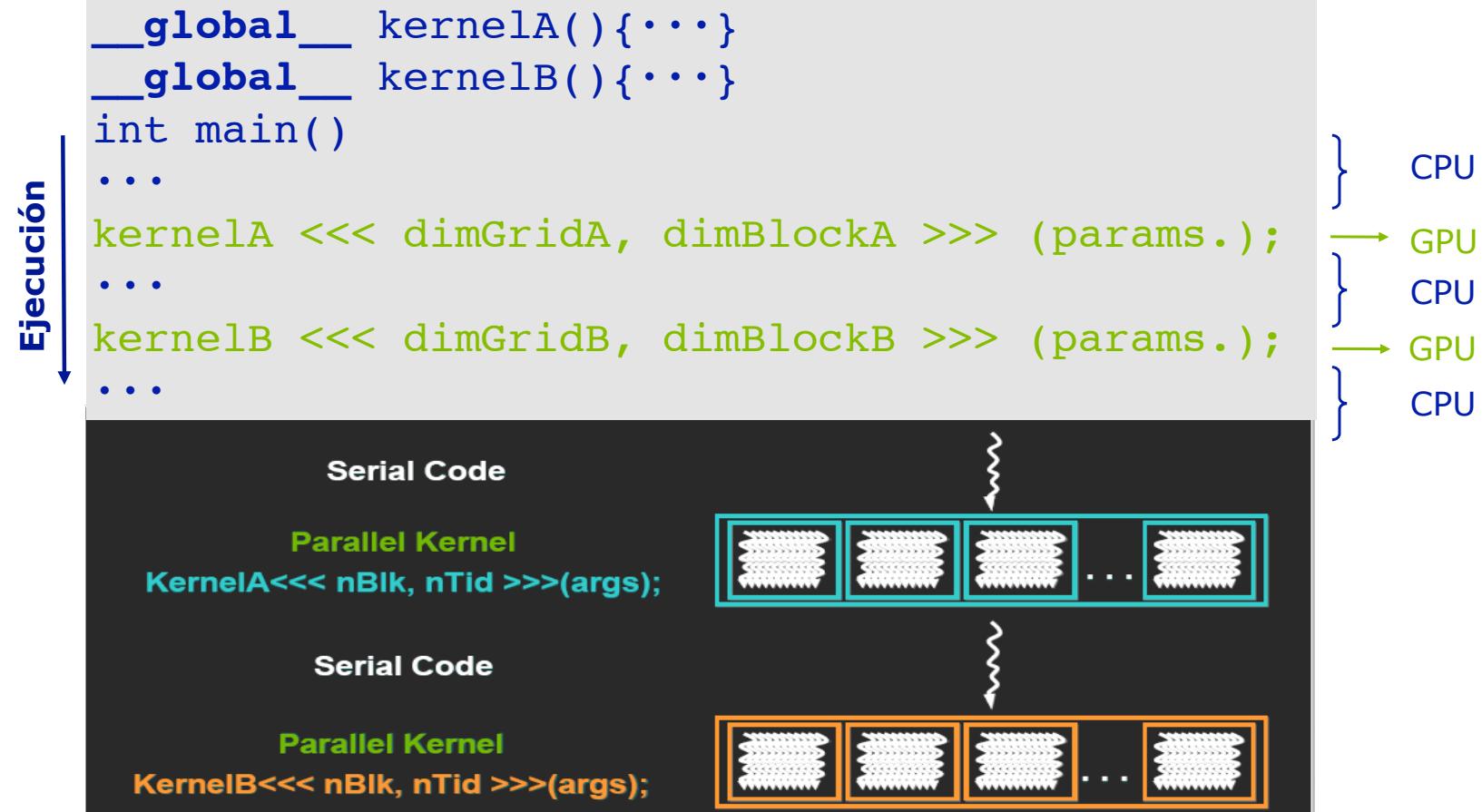
- Funciones kernel para lanzar código a la GPU desde la CPU.

```
__device__ float vector[N];  
  
__global__ void filtro (float *image) {  
  
    __shared__ float region[M];  
    ...  
  
    region[threadIdx.x] = image[i];  
  
    __syncthreads();  
    ...  
    imagen[j] = result;  
}  
  
// Alojar memoria en la GPU  
void *myimage;  
cudaMalloc(&myimage, bytes);  
  
// 100 bloques de hilos, 10 hilos por bloque  
convolucion <<<100, 10>>> (myimage);
```

La interacción entre la CPU y la GPU

- CUDA extiende el lenguaje C con un nuevo tipo de función, kernel, que ejecutan en paralelo los hilos activos en GPU.
- El resto del código es C nativo que se ejecuta sobre la CPU de forma convencional.
- De esta manera, el típico `main()` de C combina la ejecución secuencial en CPU y paralela en GPU de kernels CUDA.
- Un kernel se lanza siempre de forma asíncrona, esto es, el control regresa de forma inmediata a la CPU.
- Cada kernel GPU tiene una barrera implícita a su conclusión, esto es, no finaliza hasta que no lo hagan todos sus hilos.
- Aprovecharemos al máximo el biprocesador CPU-GPU si les vamos intercalando código con similar carga computacional.

La interacción entre la CPU y la GPU (cont.)



- Un kernel no comienza hasta que terminan los anteriores.
- Emplearemos **streams** para definir kernels paralelos.

Modificadores para las funciones y lanzamiento de ejecuciones en GPU

● Modificadores para las funciones ejecutadas en la GPU:

- **`__global__`** void MyKernel() { } // Invocado por la CPU
- **`__device__`** float MyFunc() { } // Invocado por la GPU

● Modificadores para las variables que residen en la GPU:

- **`__shared__`** float MySharedArray[32]; // Mem. compartida
- **`__constant__`** float MyConstantArray[32];

● Configuración de la ejecución para lanzar kernels:

- `dim2 gridDim(100,50); // 5000 bloques de hilos`
- `dim3 blockDim(4,8,8); // 256 hilos por bloque`
- `MyKernel <<< gridDim,blockDim >>> (pars.); // Lanzam.`
- Nota: Opcionalmente, puede haber un tercer parámetro tras `blockDim` para indicar la cantidad de memoria compartida que será alojada dinámicamente por cada kernel durante su ejecución.

Variables y funciones intrínsecas

- `dim3 gridDim; // Dimension(es) de la malla`
- `dim3 blockDim; // Dimension(es) del bloque`

- `uint3 blockIdx; // Indice del bloque dentro de la malla`
- `uint3 threadIdx; // Indice del hilo dentro del bloque`

- `void __syncthreads(); // Sincronización entre hilos`

- El programador debe elegir el tamaño del bloque y el número de bloques para explotar al máximo el paralelismo del código durante su ejecución.

Funciones para conocer en tiempo de ejecución con qué recursos contamos

- Cada GPU disponible en la capa hardware recibe un número entero que la identifica, comenzando por el 0.

- Para conocer el número de GPUs disponibles:

- `cudaGetDeviceCount(int* count);`

- Para conocer los recursos disponibles en la GPU dev (caché, registros, frecuencia de reloj, ...):

- `cudaGetDeviceProperties(struct cudaDeviceProp* prop, int dev);`

- Para conocer la mejor GPU que reúne ciertos requisitos:

- `cudaChooseDevice(int* dev, const struct cudaDeviceProp* prop);`

- Para seleccionar una GPU concreta:

- `cudaSetDevice(int dev);`

- Para conocer en qué GPU estamos ejecutando el código:

- `cudaGetDevice(int* dev);`

Ejemplo: Salida de la función cudaGetDeviceProperties

- El programa se encuentra dentro del SDK de Nvidia.

```
There are 4 devices supporting CUDA
```

```
Device 0: "GeForce GTX 480"
  CUDA Driver Version:          4.0
  CUDA Runtime Version:         4.0
  CUDA Capability Major revision number: 2
  CUDA Capability Minor revision number: 0
  Total amount of global memory: 1609760768 bytes
  Number of multiprocessors:    15
  Number of cores:             480
  Total amount of constant memory: 65536 bytes
  Total amount of shared memory per block: 49152 bytes
  Total number of registers available per block: 32768
  Warp size:                   32
  Maximum number of threads per block: 1024
  Maximum sizes of each dimension of a block: 1024 x 1024 x 64
  Maximum sizes of each dimension of a grid: 65535 x 65535 x 65535
  Maximum memory pitch:          2147483647 bytes
  Texture alignment:            512 bytes
  Clock rate:                  1.40 GHz
  Concurrent copy and execution: Yes
  Run time limit on kernels:     No
  Integrated:                  No
  Support host page-locked memory mapping: Yes
  Compute mode:                 Default (multiple host threads can use this device simultaneously)
  Concurrent kernel execution:   Yes
  Device has ECC support enabled: No
```

Para gestionar la memoria de vídeo

- Para reservar y liberar memoria en la GPU:
 - `cudaMalloc(puntero, tamaño)`
 - `cudaFree(puntero)`
- Para mover áreas de memoria entre CPU y GPU:
 - En la CPU, declaramos `malloc(h_A)`.
 - En la GPU, declaramos `cudaMalloc(d_A)`.
 - Y una vez hecho esto, podemos:
 - Pasar los datos desde la CPU a la GPU:
 - `cudaMemcpy(d_A, h_A, numBytes, cudaMemcpyHostToDevice);`
 - Pasar los datos desde la GPU a la CPU:
 - `cudaMemcpy(h_A, d_A, numBytes, cudaMemcpyDeviceToHost);`
- El prefijo “`h_`” suele usarse para punteros en memoria principal. Idem “`d_`” para punteros en memoria de vídeo.



IV. 2. Un par de ejemplos

Ejemplo 1: Descripción del código a programar

- Alojar N enteros en la memoria de la CPU.
- Alojar N enteros en la memoria de la GPU.
- Inicializar la memoria de la GPU a cero.
- Copiar los valores desde la GPU a la CPU.
- Imprimir los valores.

Ejemplo 1: Implementación

[código C en rojo, extensiones CUDA en azul]

```
int main()
{
    int N = 16;
    int num_bytes = N*sizeof(int);
    int *d_a=0, *h_a=0; // Punteros en dispos. (GPU) y host (CPU)

    h_a = (int*) malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes);

    if( 0==h_a || 0==d_a ) printf("No pude reservar memoria\n");

    cudaMemset( d_a, 0, num_bytes);
    cudaMemcpy( h_a, d_a, num_bytes, cudaMemcpyDeviceToHost);

    for (int i=0; i<N; i++) printf("%d ", h_a[i]);

    free(h_a);
    cudaFree(d_a);
}
```

Transferencias de memoria asíncronas

- Las llamadas a `cudaMemcpy()` son síncronas, esto es:
 - No comienzan hasta que no hayan finalizado todas las llamadas CUDA que le preceden.
 - El retorno a la CPU no tiene lugar hasta que no se haya realizado la copia en memoria.
- A partir de CUDA Compute Capabilities 1.2 es posible utilizar la variante `cudaMemcpyAsync()`, cuyas diferencias son las siguientes:
 - El retorno a la CPU tiene lugar de forma inmediata.
 - Podemos solapar comunicación y computación.

Ejemplo 2: Incrementar un valor “b” a los N elementos de un vector

Programa C en CPU.
Este archivo se compila con **gcc**

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

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

El kernel CUDA que se ejecuta en GPU,
seguido del código host para CPU.
Este archivo se compila con **nvcc**

```
__global__ void incremento_en_gpu(float *a, float b, int N)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N)
        a[idx] = a[idx] + b;
}

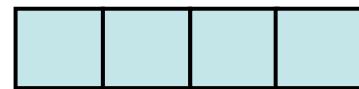
void main()
{
    ....
    dim3 dimBlock (blocksize);
    dim3 dimGrid (ceil(N/(float)blocksize));
    incremento_en_gpu<<<dimGrid, dimBlock>>>(a, b, N);
}
```

Ejemplo 2: Incrementar un valor “b” a los N elementos de un vector

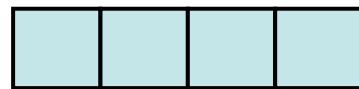


Con $N=16$ y $\text{blockDim}=4$, tenemos 4 bloques de hilos, encargándose cada hilo de computar un elemento del vector. Es lo que queremos: Paralelismo de grano fino en la GPU.

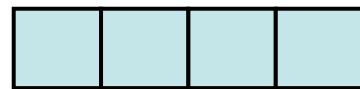
Extensiones
al lenguaje



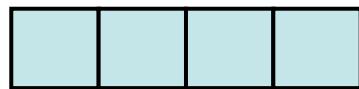
`blockIdx.x = 0`
`blockDim.x = 4`
`threadIdx.x = 0,1,2,3`
`idx = 0,1,2,3`



`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`

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

Se mapeará del índice local `threadIdx.x` al índice global

} Patrón de acceso común a todos los hilos

Nota: `blockDim.x` debería ser ≥ 32 (tamaño del warp), esto es sólo un ejemplo.

Código en CPU para el ejemplo 2

[rojo es C, verde son variables, azul es CUDA]

```
// Aloja memoria en la CPU
unsigned int numBytes = N * sizeof(float);
float* h_A = (float*) malloc(numBytes);

// Aloja memoria en la GPU
float* d_A = 0;  cudaMalloc(&d_A, numbytes);

// Copia los datos de la CPU a la GPU
cudaMemcpy(d_A, h_A, numBytes, cudaMemcpyHostToDevice);

// Ejecuta el kernel con un número de bloques y tamaño de bloque
incremento_en_gpu <<< N/blockSize, blockSize >>> (d_A, b);

// Copia los resultados de la GPU a la CPU
cudaMemcpy(h_A, d_A, numBytes, cudaMemcpyDeviceToHost);

// Libera la memoria de vídeo
cudaFree(d_A);
```



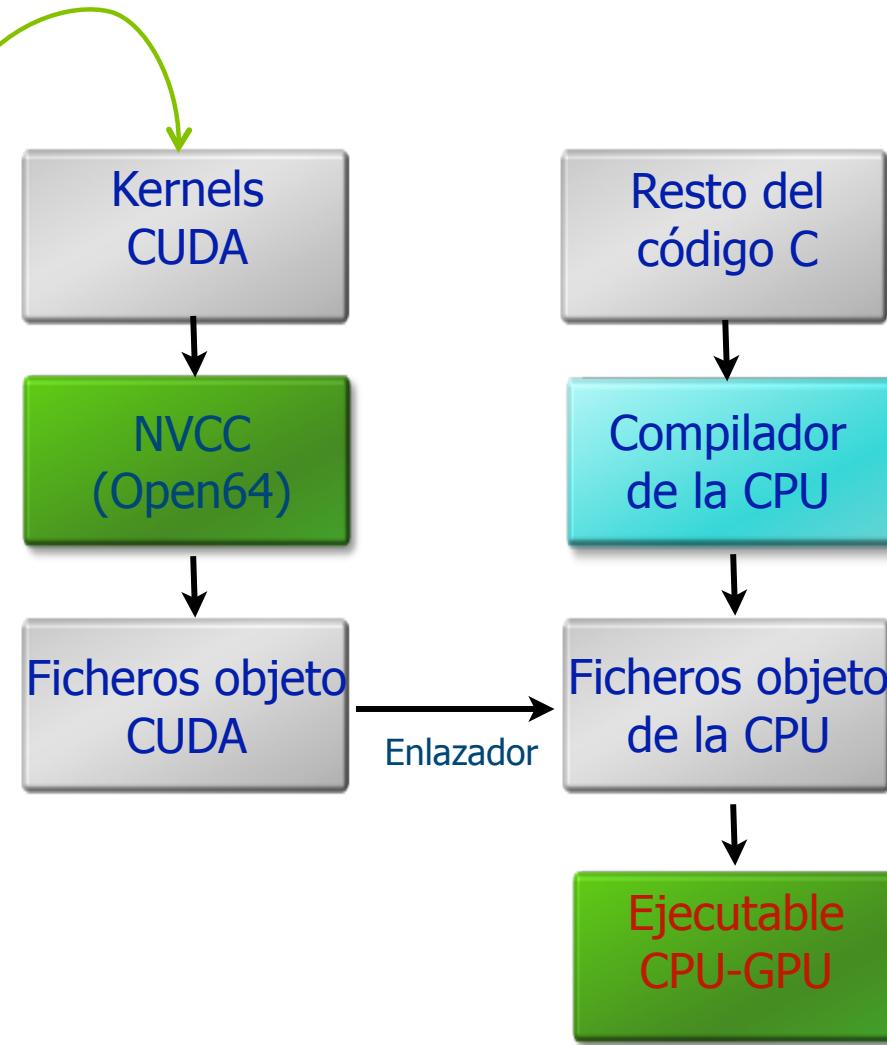
V. Compilación



El proceso global de compilación

```
void funcion_en_CPU(... )  
{  
    ...  
}  
void otras_funcs_CPU(int ...)  
{  
    ...  
}  
  
void saxpy_serial(float ... )  
{  
    for (int i = 0; i < n; ++i)  
        y[i] = a*x[i] + y[i];  
}  
  
void main( ) {  
    float x;  
    saxpy_serial(...);  
    ...  
}
```

Identificar los kernels CUDA y reescribirlos para aprovechar paralelismo en GPU



Los diferentes módulos de compilación

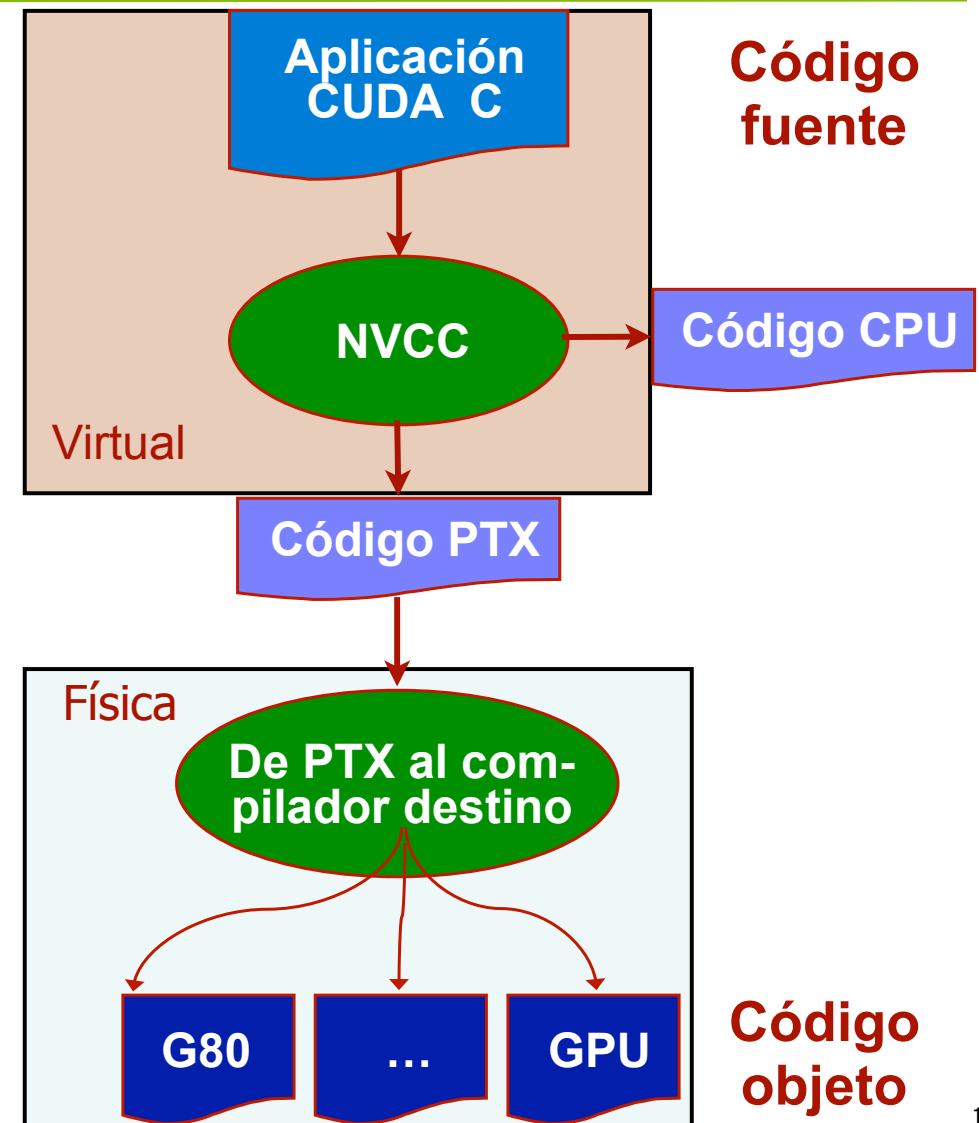
- El código fuente CUDA se compila con NVCC.

- NVCC separa el código que se ejecuta en CPU del que lo hace en GPU.

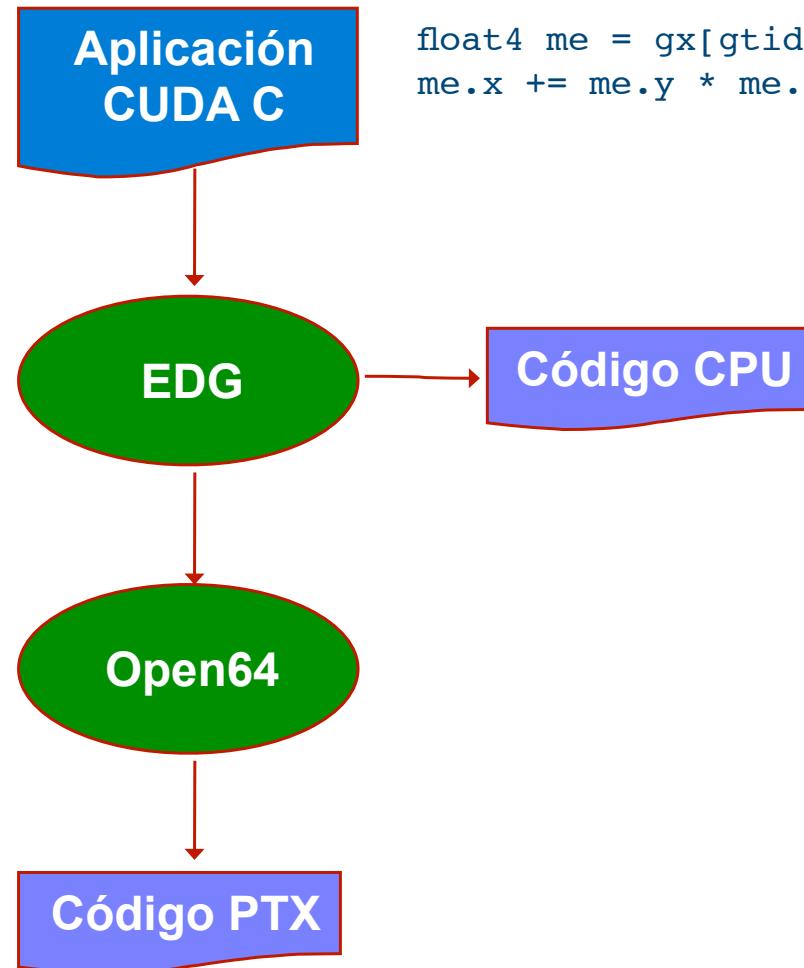
- La compilación se realiza en dos etapas:

- Virtual: Genera código PTX (Parallel Thread eXecution).

- Física: Genera el binario para una GPU específica (o incluso para una CPU multicore).



Compilador nvcc y máquina virtual PTX



EDG

Separá código GPU y CPU.

Open64

Genera ensamblador PTX.

Parallel Thread eXecution (PTX)

Máquina virtual e ISA.

Modelo de programación.

Recursos y estados de ejecución.

ld.global.v4.f32 {\$f1,\$f3,\$f5,\$f7}, [\$r9+0];
mad.f32 \$f1, \$f5, \$f3, \$f1;

NVCC (NVidia CUDA Compiler)

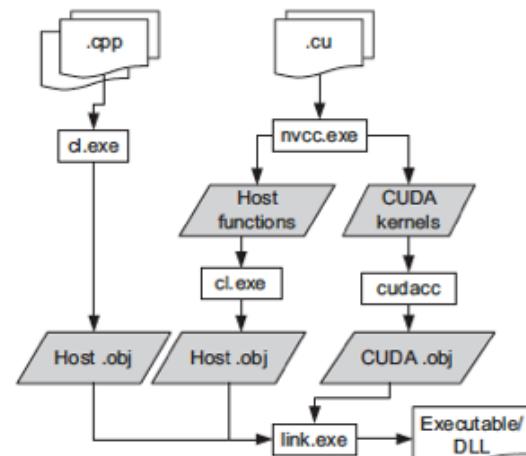
- NVCC es un driver del compilador.

- Invoca a los compiladores y herramientas, como cudacc, g++, cl, ...

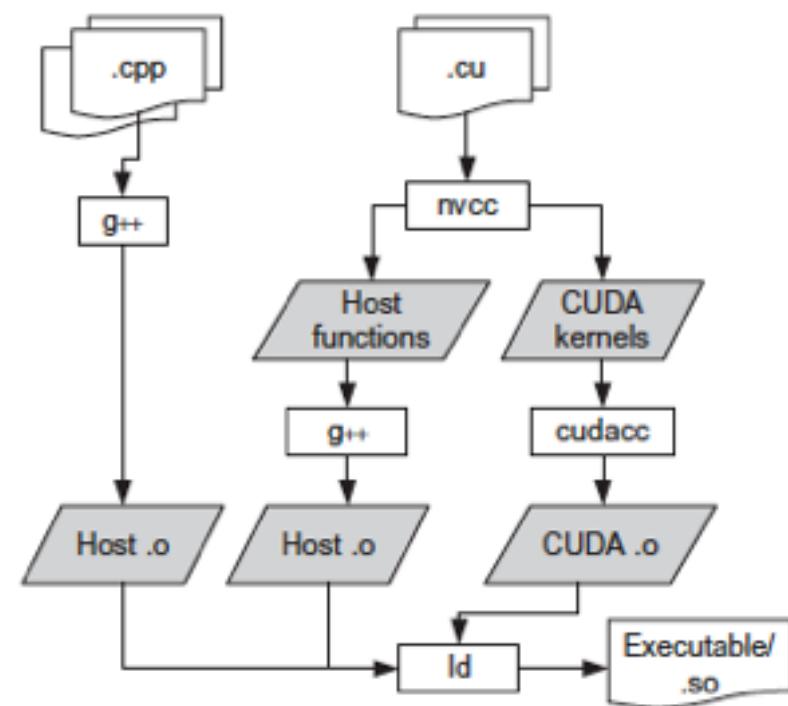
- NVCC produce como salida:

- Código C para la CPU, que debe luego compilarse con el resto de la aplicación utilizando otra herramienta.
 - Código objeto PTX para la GPU.

Proceso de compilación en Windows:



Proceso de compilación Linux:



Para conocer la utilización de los recursos

- Compilar el código del kernel con el flag `-cubin` para conocer cómo se están usando los registros.
 - Alternativa on-line: `nvcc --ptxas-options=-v`
- Abrir el archivo de texto `.cubin` y mirar la sección “code”.

```
architecture {sm_10}
abiversion {0}
modname {cubin}
code {
    name = myGPUcode
    lmem = 0
    smem = 68
    reg = 20
    bar = 0
    bincode {
        0xa0004205 0x04200780 0x40024c09 0x0020
        ...
    }
}
```

Memoria local para cada hilo
(usada por el compilador para volcar contenidos de los registros en memoria)

Memoria compartida usada por cada bloque de hilos

Registros usados por cada hilo

Heurísticos para la configuración de la ejecución

- El número de hilos por bloque debe ser un múltiplo de 32.
 - Para no desperdiciar la ejecución de warps incompletos.
- Debemos declarar más bloques que multiprocesadores haya (1), y a ser posible, ser más del doble (2):
 - (1) Para que todos ellos tengan al menos un bloque que ejecutar.
 - (2) Para tener al menos un bloque activo que garantice la actividad del SMX cuando el bloque en ejecución sufra un parón debido a un acceso a memoria, no disponibilidad de UFs, conflictos en bancos, puntos de sincronización (`syncthreads()`), etc.
- Los recursos por bloque (registros y memoria compartida) deben ser al menos la mitad del total disponible.
 - De lo contrario, resulta mejor fusionar bloques.

Heurísticos (cont.)

- Reglas generales para que el código sea escalable en futuras generaciones y para que el flujo de bloques pueda ejecutarse de forma segmentada (pipeline):
 - (1) Pensar a lo grande para el número de bloques.
 - (2) Pensar en pequeño para el tamaño de los hilos.
- Conflicto: Más hilos por bloque significa mejor ocultación de latencia, pero menos registros disponibles para cada hilo.
- Sugerencia: Utilizar un mínimo de 64 hilos por bloque, o incluso mejor, 128 ó 256 hilos (si aún disponemos de suficientes registros).
- Conflicto: Incrementar la ocupación no significa necesariamente aumentar el rendimiento, pero una baja ocupación del multiprocesador no permite ocultar latencias en kernels limitados por el ancho de banda a memoria.
- Sugerencia: Vigilar la intensidad aritmética y el paralelismo.

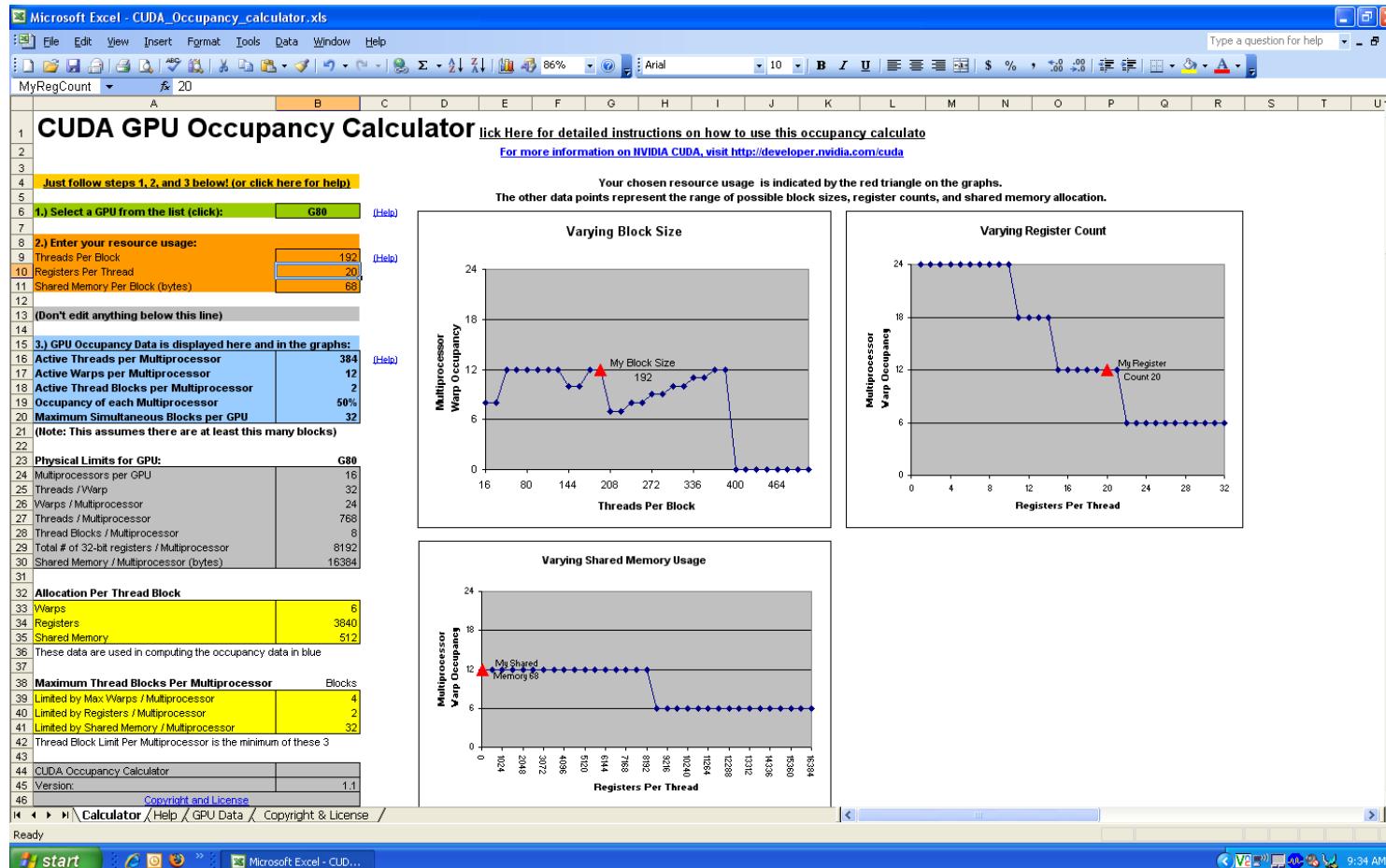
Parametrización de una aplicación

- ➊ Todo lo que concierne al rendimiento es dependiente de la aplicación, por lo que hay que experimentar con ella para lograr resultados óptimos.
- ➋ Las GPUs evolucionan muy rápidamente, sobre todo en:
 - ➌ El nº de multiprocesadores (SMs) y el número de cores por SMs.
 - ➌ El ancho de banda con memoria: Entre 100 GB/s. y 1 TB/s.
 - ➌ El tamaño del banco de registros de cada SM: 8K, 16K, 32K, 64K.
 - ➌ El tamaño de la memoria compartida: 48 KB. (en Fermi y Kepler), 64 KB., 96 KB. (en algunos modelos de Maxwell y Pascal)
 - ➌ Hilos: Comprobar el límite por bloque y el límite total.
 - ➍ Por bloque: 1024 (a partir de Fermi).
 - ➍ Total: 2048 (a partir de Kepler).

CUDA Occupancy Calculator

Asesora en la selección de los parámetros de configuración

http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls



Para alcanzar el mayor grado de paralelismo, fijarse en el área naranja del CUDA Occup.(1)

- El primer dato es el número de hilos por bloque:
 - El límite es 1024.
 - Las potencias de dos son las mejores elecciones.
 - Lista de candidatos: 2, 4, 8, 16, 32, 64, 128, **256**, 512, 1024.
 - Pondremos 256 como primera estimación, y el ciclo de desarrollo nos conducirá al valor óptimo aquí, aunque normalmente:
 - Valores pequeños [2, 4, 8, 16] no explotan el tamaño del warp ni los bancos de memoria compartida.
 - Valores intermedios [32, 64] comprometen la cooperación entre hilos y la escalabilidad en futuras generaciones de GPUs.
 - Valores grandes [512, 1024] nos impiden tener suficiente número de bloques concurrentes en cada multiprocesador, ya que el máximo de hilos por bloque y SMX son dos valores muy próximos. Además, la cantidad de registros disponibles para cada hilo es baja.

Para alcanzar el mayor grado de paralelismo, fijarse en el área naranja del CUDA Occup.(2)

- El segundo dato es el nº de registros que usa cada hilo.
 - Esto lo obtendremos del archivo .cubin.
 - El límite es 64 Kregistros en cada SM (a partir de Kepler), así que consumiendo 32 registros/hilo es posible maximizar la ocupación:
 - $8 \text{ bloques} * 256 \text{ hilos/bloque} * 32 \text{ registros/hilo} = 64 \text{ Kregistros.}$
 - ... disponiendo así del máximo de 2048 hilos activos en cada uno de los multiprocesadores disponibles.

Para alcanzar el mayor grado de paralelismo, fijarse en el área naranja del CUDA Occup.(3)

- El tercer dato es el gasto de memoria compartida en cada bloque de hilos:
 - Esto también lo obtenemos del archivo .cubin, aunque podemos llevar una contabilidad manual, ya que todo depende de la declaración de variables `__shared__` que elegimos como programador.
 - Límites: 48 KB (Kepler), 64 KB (unas Maxwell/Pascal), 96 KB (otras).
 - Para el caso intermedio de 64 KB, en el ejemplo anterior de 8 bloques de 256 hilos, para mantener el 100% de ocupación, cada bloque puede usar un máximo de 8 KB de memoria compartida:
 - $8 \text{ bloques} \times 8 \text{ KB./bloque} = 64 \text{ KB.}$
 - Y si incrementamos, por ejemplo, a:
 - 9 KB/bloque, entonces sólo tendremos 7 bloques activos (sacrificamos el 12.5%).
 - 10 KB/bloque, sólo tendremos 6 bloques activos (sacrificamos el 25% del paralel.).
 - Como era de esperar, hay un conflicto entre paralelismo y memoria.



VI. Ejemplos: VectorAdd, Stencil, ReverseArray, MxM



Pasos para la construcción del código CUDA

1. Identificar las partes del código con mayor potencial para beneficiarse del paralelismo de datos SIMD de la GPU.
2. Acotar el volumen de datos necesario para realizar dichas computaciones.
3. Transferir los datos a la GPU.
4. Hacer la llamada al kernel.
5. Establecer las sincronizaciones entre la CPU y la GPU.
6. Transferir los resultados desde la GPU a la CPU.
7. Integrarlos en las estructuras de datos de la CPU.

Se requiere cierta coordinación en las tareas paralelas

- El paralelismo viene expresado por los bloques e hilos.
- Los hilos de un bloque pueden requerir sincronización si aparecen dependencias, ya que sólo dentro del warp se garantiza su progresión conjunta (SIMD). Ejemplo:

```
a[i] = b[i] + 7;  
syncthreads();  
x[i] = a[i-1]; // El warp 1 lee aquí el valor a[31],  
                // que debe haber sido escrito ANTES por el warp 0
```

- En las fronteras entre kernels hay barreras implícitas:
 - Kernel1 <<<nblocks,nthreads>>> (a,b,c);
 - Kernel2 <<<nblocks,nthreads>>> (a,b);
- Bloques pueden coordinarse usando operaciones atómicas.
 - Ejemplo: Incrementar un contador atomicInc();



VI. 1. Suma de dos vectores

Código para GPU y su invocación desde CPU

```
// Suma de dos vectores de tamaño N: C[1..N] = A[1..N] + B[1..N]
// Cada hilo computa un solo componente del vector resultado C
__global__ void vecAdd(float* A, float* B, float* C) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    C[tid] = A[tid] + B[tid];                                Código GPU
}
```

```
int main() { // Lanza N/256 bloques de 256 hilos cada uno
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);          Código CPU
}
```

- ➊ El prefijo **__global__** indica que **vecAdd()** se ejecuta en la GPU (device) y será llamado desde la CPU (host).
- ➋ A, B y C son punteros a la memoria de vídeo de la GPU, por lo que necesitamos:
 - ➌ Alojar/liberar memoria en GPU, usando **cudaMalloc/cudaFree**.
 - ➍ Estos punteros no pueden ser utilizados desde el código de la CPU.

Código CPU para manejar la memoria y recoger los resultados de GPU

```
unsigned int numBytes = N * sizeof(float);
// Aloja memoria en la CPU
float* h_A = (float*) malloc(numBytes);
float* h_B = (float*) malloc(numBytes);
... inicializa h_A y h_B ...
// Aloja memoria en la GPU
float* d_A = 0;  cudaMalloc((void**)&d_A, numBytes);
float* d_B = 0;  cudaMalloc((void**)&d_B, numBytes);
float* d_C = 0;  cudaMalloc((void**)&d_C, numBytes);
// Copiar los datos de entrada desde la CPU a la GPU
cudaMemcpy(d_A, h_A, numBytes, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, numBytes, cudaMemcpyHostToDevice);
(aquí colocamos la llamada al kernel VecAdd de la pág. anterior)
// Copiar los resultados desde la GPU a la CPU
float* h_C = (float*) malloc(numBytes);
cudaMemcpy(h_C, d_C, numBytes, cudaMemcpyDeviceToHost);
// Liberar la memoria de vídeo
cudaFree(d_A);  cudaFree(d_B);  cudaFree(d_C);
```

Ejecutando en paralelo (independientemente de la generación HW)

• `vecAdd <<< 1, 1 >>> ()`

Ejecuta 1 bloque de 1 hilo.

No hay paralelismo.

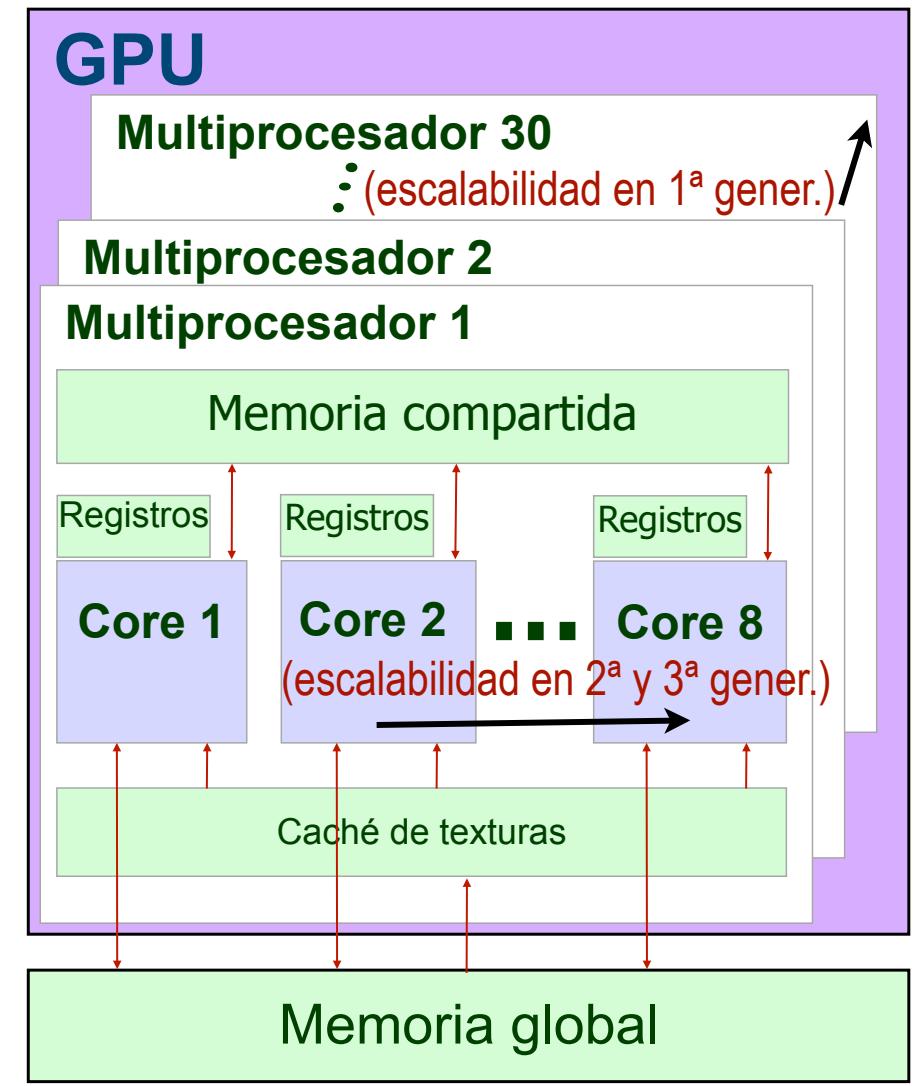
• `vecAdd <<< B, 1 >>> ()`

Ejecuta B bloques de 1 hilo.

Hay paralelismo entre multiprocesadores.

• `vecAdd <<< B, M >>> ()`

Ejecuta B bloques compuestos de M hilos. Hay paralelismo entre multiprocesadores y entre los cores del multiprocesador.

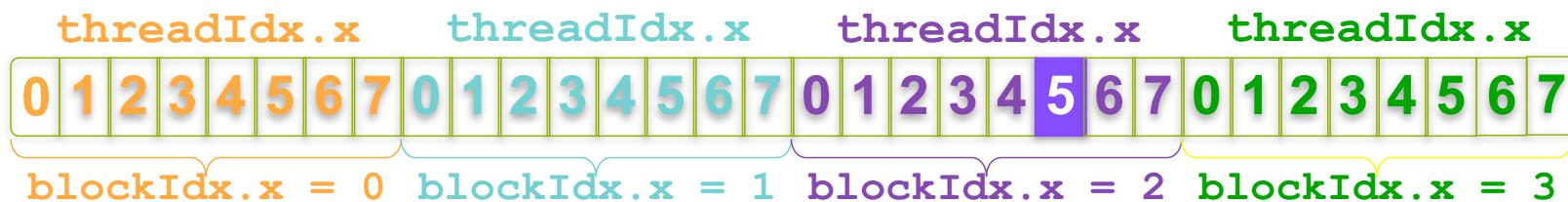


Calculando el índice de acceso a un vector según el bloque y el hilo dentro del mismo

- Con M hilos por bloque, un índice único viene dado por:

`tid = blockIdx.x * blockDim.x + threadIdx.x;`

- Para acceder a un vector en el que cada hilo computa un solo elemento (queremos paralelismo de grano fino), B=4 bloques de M=8 hilos cada uno:



- ¿Qué hilo computará el vigésimo segundo elemento del vector?

- `gridDim.x` es 4. `blockDim.x` es 8. `blockIdx.x = 2`. `threadIdx.x = 5`.
- $tid = (2 * 8) + 5 = 21$ (al comenzar en 0, éste es el elemento 22).

Manejando vectores de tamaño genérico

- Los algoritmos reales no suelen tener dimensiones que sean múltiplos exactos de `blockDim.x`, así que debemos desactivar los hilos que computan fuera de rango:

```
// Suma dos vectores de tamaño N: C[1..N] = A[1..N] + B[1..N]
__global__ void vecAdd(float* A, float* B, float* C, N) {
    int tid = (blockIdx.x * blockDim.x) + threadIdx.x;
    if (tid < N)
        C[tid] = A[tid] + B[tid];
}
```

- Y ahora, hay que incluir el bloque "incompleto" de hilos en el lanzamiento del kernel (redondeo al alza en la div.):

```
vecAdd<<< (N + M-1)/256, 256>>>(d_A, d_B, d_C, N);
```



VI. 2. Kernels patrón (stencils)

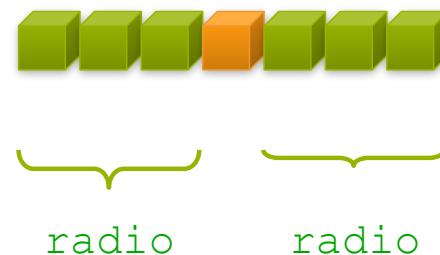
Por qué hemos seleccionado este código

- En el ejemplo anterior, los hilos añaden complejidad sin realizar contribuciones reseñables.
- Sin embargo, los hilos pueden hacer cosas que no están al alcance de los bloques paralelos:
 - Comunicarse (a través de la memoria compartida).
 - Sincronizarse (por ejemplo, para salvar los conflictos por dependencias de datos).
- Para ilustrarlo, necesitamos un ejemplo más sofisticado ...

Patrón unidimensional (1D)

- Aplicaremos un patrón 1D a un vector 1D.

- Al finalizar el algoritmo, cada elemento contendrá la suma de todos los elementos dentro de un radio próximo al elemento dado.
- Por ejemplo, si el radio es 3, cada elemento agrupará la suma de 7:



- De nuevo aplicamos paralelismo de grano fino, por lo que cada hilo se encargará de obtener el resultado de un solo elemento del vector resultado.
- Los valores del vector de entrada deben leerse varias veces:
 - Por ejemplo, para un radio de 3, cada elemento se leerá 7 veces.

Compartiendo datos entre hilos. Ventajas

- Los hilos de un mismo bloque pueden compartir datos a través de memoria compartida.
- El programador gestiona la memoria compartida de forma explícita, insertando el prefijo shared en la declaración de la variable.
 - Los datos se alojan para cada uno de los bloques.
 - La memoria compartida es extremadamente rápida:
 - 500 veces más rápida que la memoria global (que es memoria de video GDDR5). La diferencia es tecnológica: estática (construida con transistores) frente a dinámica (construida con mini-condensadores).
 - La memoria compartida puede verse como una extensión del banco de registros, pero es más versátil que éstos, que son privados a cada hilo.

Compartiendo datos entre hilos. Limitaciones

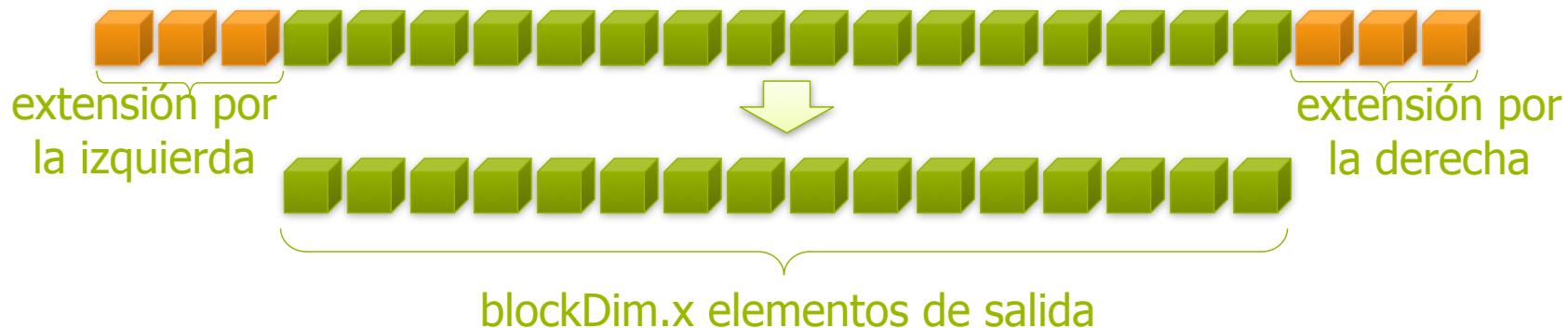
- El uso de los registros y la memoria compartida limita el paralelismo.
 - Si dejamos espacio para un segundo bloque en cada multiprocesador, el banco de registros y la memoria compartida se partitionan (aunque la ejecución no es simultánea, el cambio de contexto es inmediato).
- Ya mostramos ejemplos para Kepler anteriormente. Con un máximo de 64K registros y 48 Kbytes de memoria compartida por cada multiprocesador SMX, tenemos:
 - Para 2 bl./SMX: No superar 32 Kregs. ni 24 KB. de memoria comp.
 - Para 3 bl./SMX: No superar 21.33 Kregs. ni 16 KB. de memoria comp.
 - Para 4 bl./SMX: No superar 16 Kregs. ni 12 KB. de memoria comp.
 - ... y así sucesivamente. Usar el CUDA Occupancy Calculator para asesorarse en la selección de la configuración más adecuada.

Utilizando la memoria compartida

- Pasos para cachear los datos en memoria compartida:

- Leer (`blockDim.x + 2 * radio`) elementos de entrada desde la memoria global a la memoria compartida.
- Computar `blockDim.x` elementos de salida.
- Escribir `blockDim.x` elementos de salida a memoria global.

- Cada bloque necesita una extensión de `radio` elementos en los extremos del vector.

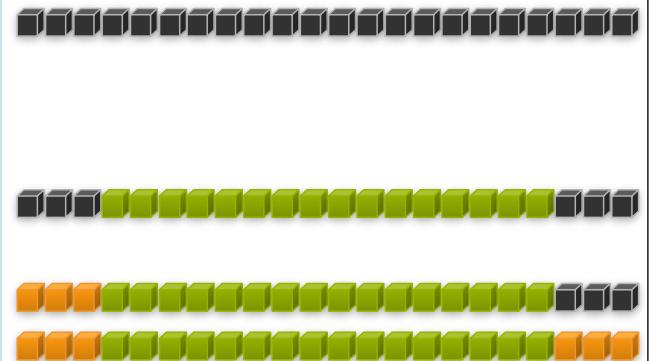


Kernel patrón

```
__global__ void stencil_1d(int *d_in, int *d_out)
{
    __shared__ int temp[BLOCKSIZE + 2 * RADIO];
    int gindex = blockIdx.x * blockDim.x + threadIdx.x;
    int lindex = threadIdx.x + RADIO;

    // Pasar los elementos a memoria compartida
    temp[lindex] = d_in[gindex];
    if (threadIdx.x < RADIO) {
        temp[lindex-RADIO] = d_in[gindex-RADIO];
        temp[lindex+blockDim.x] = d_in[gindex+blockDim.x];
    }

    // Aplicar el patrón
    int result = 0;
    for (int offset=-RADIO; offset<=RADIO; offset++) {
        result += temp[lindex + offset];
    }
    // Almacenar el resultado
    d_out[gindex] = result;
}
```



Pero hay que prevenir condiciones de carrera. Por ejemplo, el último hilo lee la extensión antes del que el primer hilo haya cargado esos valores (en otro warp). Se hace necesaria una **sincronización entre hilos**.

Sincronización entre hilos

- Usar `__syncthreads()` para sincronizar todos los hilos de un bloque:

- Todos los hilos deben alcanzar la barrera antes de proseguir.
- Puede utilizarse para prevenir riesgos del tipo RAW / WAR / WAW.
- En sentencias condicionales, la condición debe ser uniforme a lo largo de todo el bloque.

```
__global__ void stencil_1d(...)  
{  
    < Declarar variables e índices >  
    < Pasar el vector a memoria compartida >  
  
    __syncthreads();  
  
    < Aplicar el patrón >  
    < Almacenar el resultado >  
}
```

Recopilando los conceptos puestos en práctica en este ejemplo

- Lanzar N bloques con M hilos por bloque para ejecutar los hilos en paralelo. Emplear:
 - `kernel <<< N, M >>> ();`
- Acceder al índice del bloque dentro de la malla y al índice del hilo dentro del bloque. Emplear:
 - `blockIdx.x` y `threadIdx.x`;
- Calcular los índices globales donde cada hilo tenga que trabajar en un área de datos diferente según la partición. Emplear:
 - `int index = blockIdx.x * blockDim.x + threadIdx.x;`
- Declarar el vector en memoria compartida. Emplear:
 - `__shared__` (como prefijo antecediendo al tipo de dato en la declaración).
- Sincronizar los hilos para prevenir los riesgos de datos. Emplear:
 - `__syncthreads();`



VI. 3. Invertir el orden a los elementos de un vector



Código en GPU para el kernel ReverseArray

(1) utilizando un único bloque

```
__global__ void reverseArray(int *in, int *out) {
    int index_in = threadIdx.x;
    int index_out = blockDim.x - 1 - threadIdx.x;

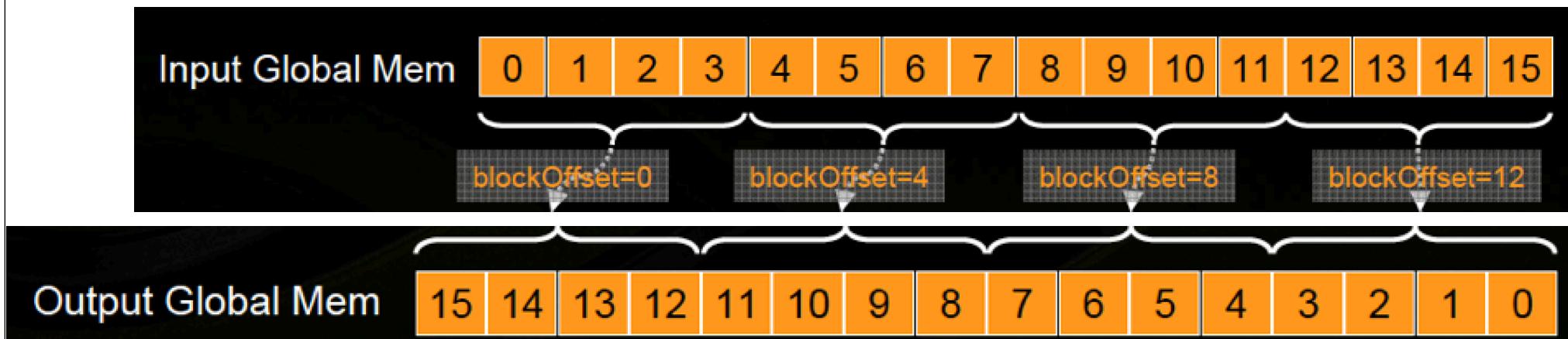
    // Invertir los contenidos del vector con un solo bloque
    out[index_out] = in[index_in];
}
```

- Es una solución demasiado simplista: No aspira a aplicar paralelismo masivo porque el máximo tamaño de bloque es 1024 hilos, con lo que ése sería el mayor vector que este código podría aceptar como entrada.

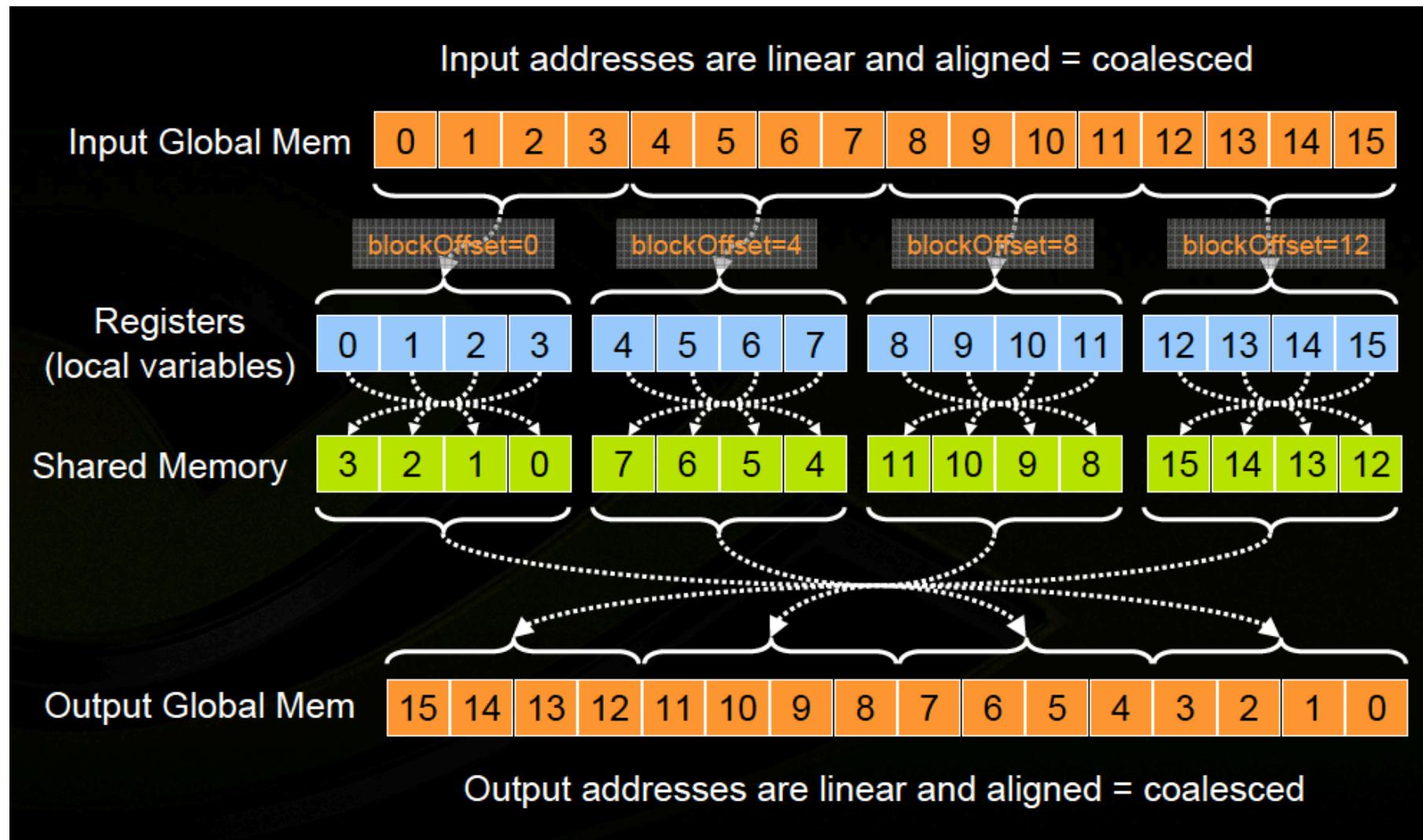
Código en GPU para el kernel ReverseArray (2) con múltiples bloques

```
__global__ void reverseArray(int *in, int *out) { // Para el hilo 0 del bloque 0:  
int in_offset = blockIdx.x * blockDim.x; // in_offset = 0;  
int out_offset = (gridDim.x - 1 - blockIdx.x) * blockDim.x; // out_offset = 12;  
int index_in = in_offset + threadIdx.x; // index_in = 0;  
int index_out = out_offset + (blockDim.x - 1 - threadIdx.x); // index_out = 15;  
  
// Invertir los contenidos en fragmentos de bloques enteros  
out[index_out] = in[index_in];  
}
```

- Por ejemplo, para 4 bloques de 4 hilos, tendríamos:



Versión utilizando la memoria compartida



Código en GPU para el kernel ReverseArray (3) con múltiples bloques y mem. compartida

```
__global__ void reverseArray(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE];
    int gindex = blockIdx.x * blockDim.x + threadIdx.x;
    int lindex = threadIdx.x;

    temp[lindex] = in[gindex];      // Pasar el vector de entrada a memoria compartida
    syncthreads();                // (i1)
    temp[lindex] = temp[blockDim.x-lindex-1]; // Invertir dentro de cada bloque (i2)
    syncthreads();                // (i3)
                                    // Invertir los contenidos en fragmentos de bloques enteros (i4)
    out[threadIdx.x + (((N/blockDim.x)-blockIdx.x-1) * blockDim.x)] = temp[lindex];
}
```

- Dependencias de datos: En (i2), los valores que escribe un warp deben ser leídos por otro.
- Solución: Usar otro vector `temp2[BLOCK_SIZE]` para guardar los resultados (tb. en (i4)).
- Mejora: (i3) no es necesario. Además, si intercambiamos los índices dentro de `temp[]` y `temp2[]` en (i2), entonces (i1) no es necesario (pero (i3) se hace imprescindible).
- Si sustituimos todas las apariciones de `temp` y `temp2` por sus expresiones equivalentes, esta versión converge a la implementación anterior sin memoria compartida.
- Cada elemento de `in` y `out` se accede una sola vez, así que no hay localidad de acceso.



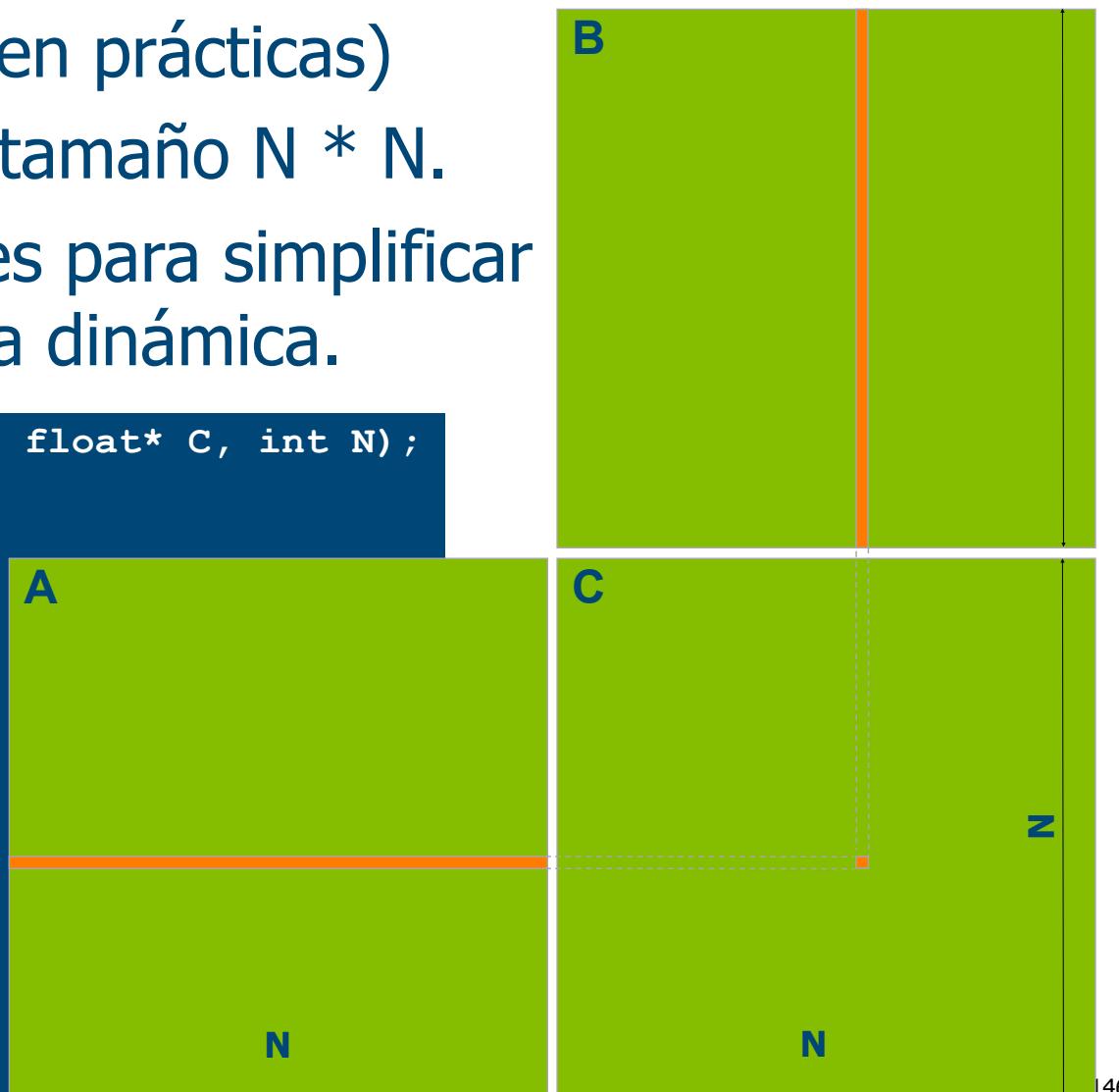
VI. 4. Producto de matrices



Versión de código CPU escrita en lenguaje C

- $C = A * B$. ($P = M * N$ en prácticas)
- Matrices cuadradas de tamaño $N * N$.
- Linearizadas en vectores para simplificar el alojamiento de memoria dinámica.

```
void MxMonCPU(float* A, float* B, float* C, int N)
{
    forall (int i=0; i<N; i++)
        forall (int j=0; j<N; j++)
        {
            float sum=0;
            for (int k=0; k<N; k++)
            {
                A[i][k] float a = A[i*N + k];
                B[k][j] float b = B[k*N + j];
                sum += a*b;
            }
            C[i*N + j] = sum;
        }
}
```

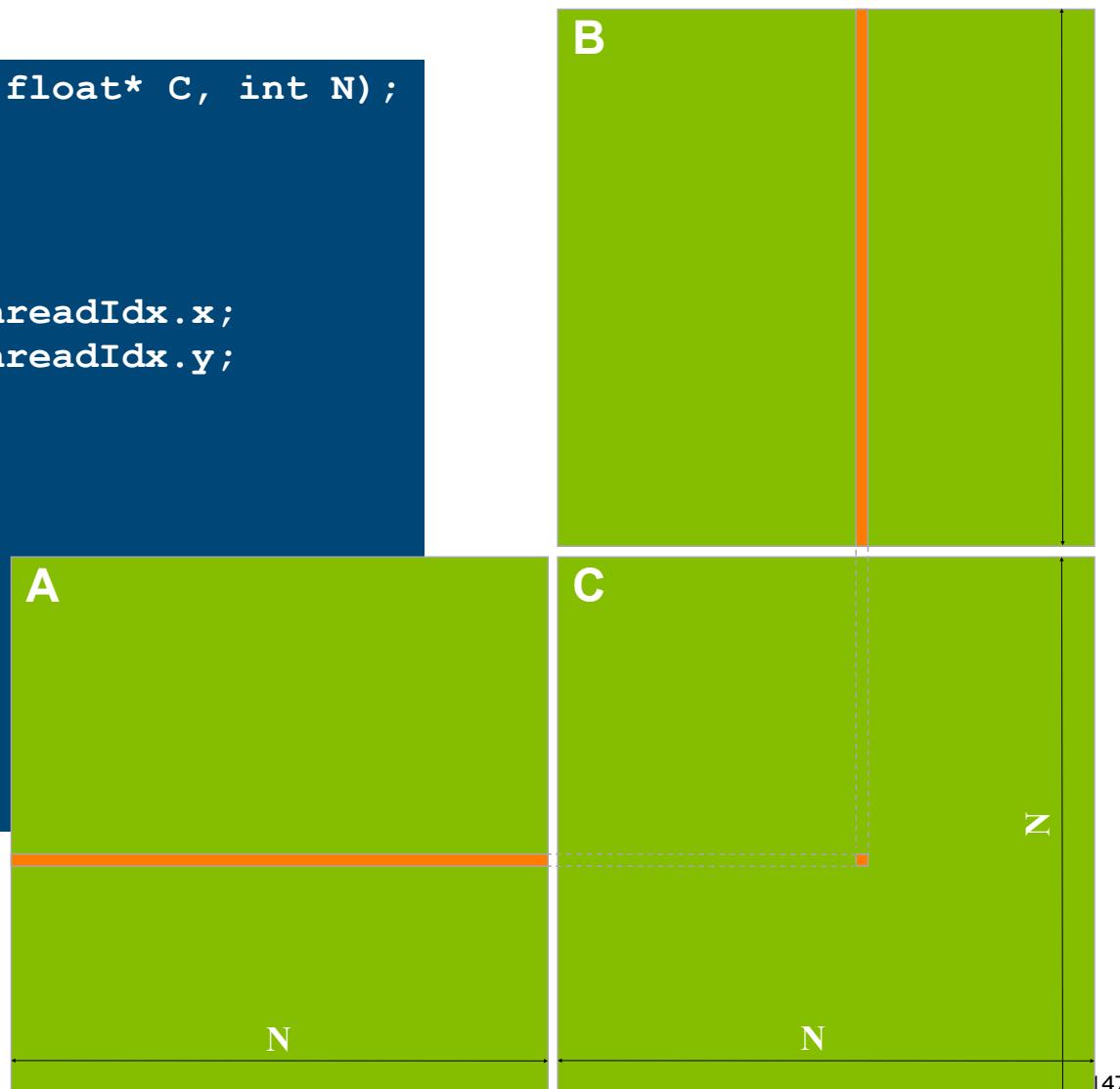


Versión CUDA para el producto de matrices: Un primer borrador para el código paralelo

```
void MxMonGPU(float* A, float* B, float* C, int N)
{
    float sum=0;
    int i, j;

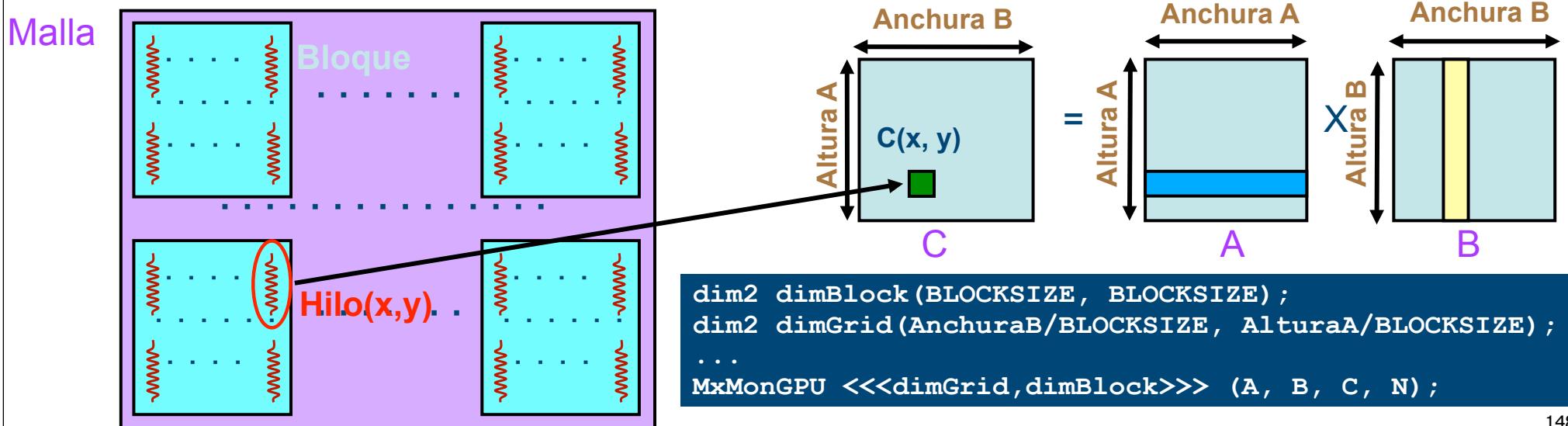
    i = blockIdx.x * blockDim.x + threadIdx.x;
    j = blockIdx.y * blockDim.y + threadIdx.y;

    for (int k=0; k<N; k++)
    {
        float a = A[i*N + k];
        float b = B[k*N + j];
        sum += a*b;
    }
    C[i*N + j] = sum;
}
```



Versión CUDA para el producto de matrices: Descripción de la paralelización

- Cada hilo computa un elemento de la matriz resultado C.
 - Las matrices A y B se cargan N veces desde memoria de vídeo.
- Los bloques acomodan los hilos en grupos de 1024 (limitación interna en arquitecturas Fermi y Kepler). Así podemos usar bloques 2D de 32x32 hilos cada uno.



Versión CUDA para el producto de matrices: Análisis

- ➊ Cada hilo utiliza 10 registros, lo que nos permite alcanzar el mayor grado de paralelismo en Kepler:
 - ➊ 2 bloques de 1024 hilos (32x32) en cada SMX. [$2 \times 1024 \times 10 = 20480$ registros, que es inferior a la cota de 65536 regs. disponibles].
- ➋ Problemas:
 - ➊ Baja intensidad aritmética.
 - ➋ Exigente en el ancho de banda a memoria, que termina erigiéndose como el cuello de botella para el rendimiento.
- ➌ Solución:
 - ➊ Utilizar la memoria compartida de cada multiprocesador.

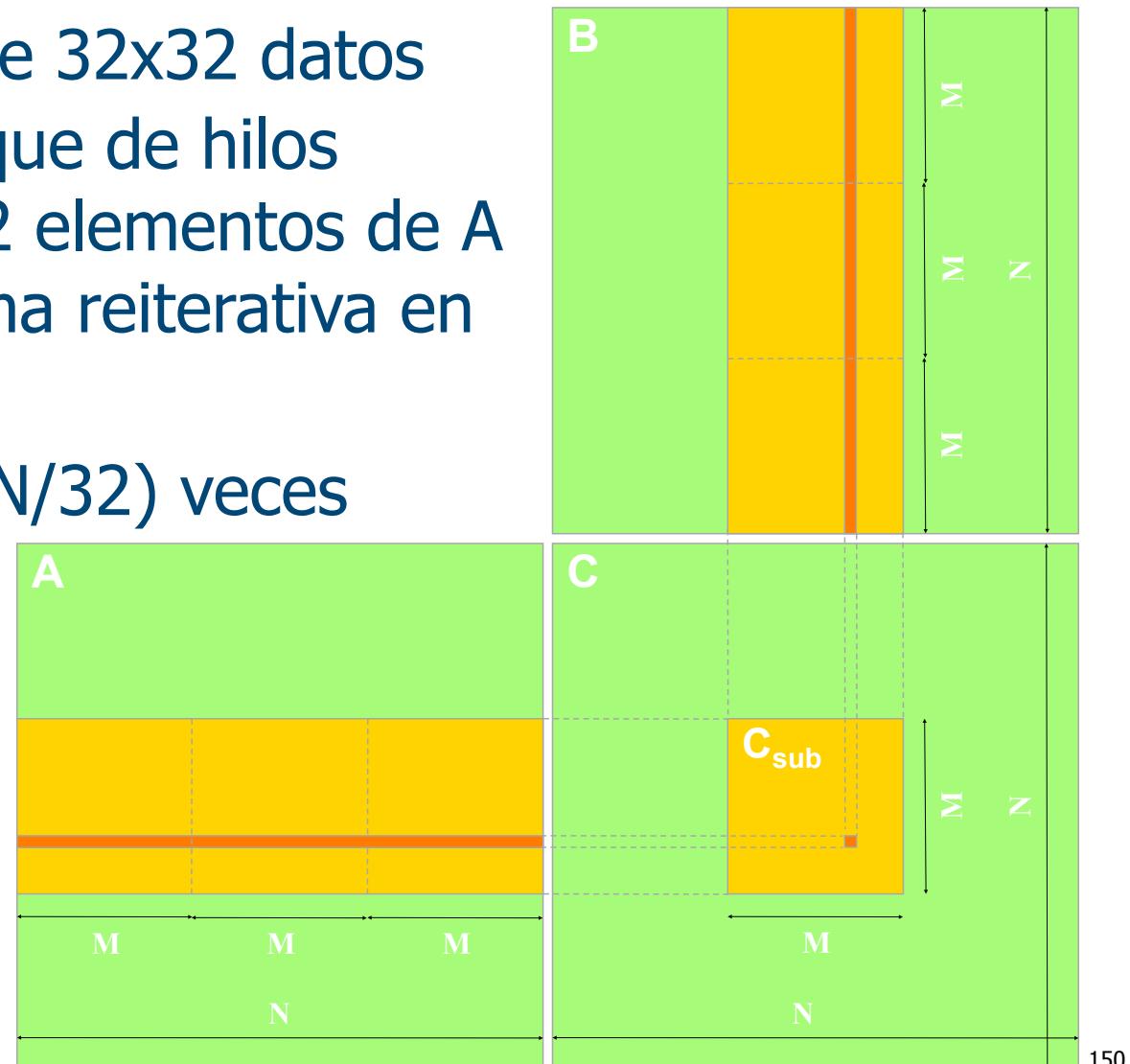
Utilizando la memoria compartida: Versión con mosaicos (tiling) para A y B

- La submatriz de C_{sub} de 32×32 datos computada por cada bloque de hilos utiliza mosaicos de 32×32 elementos de A y B que se alojan de forma reiterativa en memoria compartida.

- A y B se cargan sólo $(N/32)$ veces desde memoria global.

- Logros:

- Menos exigente en el ancho de banda a memoria.
- Más intensidad aritmética.



Tiling: Detalles de la implementación

- ➊ Tenemos que gestionar todos los mosaicos de fila y columna que necesita cada bloque de hilos:
 - ➊ Se cargan los mosaicos de entrada (A y B) desde memoria global a memoria compartida **en paralelo** (todos los hilos contribuyen). Estos mosaicos reutilizan el espacio de memoria compartida.
 - ➊ `__syncthreads()` (para asegurarnos que hemos cargado las matrices completamente antes de comenzar la computación).
 - ➊ Computar todos los productos y sumas para C utilizando los mosaicos de memoria compartida.
 - ➊ Cada hilo puede ahora iterar independientemente sobre los elementos del mosaico.
 - ➊ `__syncthreads()` (para asegurarnos que la computación con el mosaico ha acabado antes de cargar, en el mismo espacio de memoria compartida, dos nuevos mosaicos para A y B en la siguiente iteración).

Un truco para evitar conflictos en el acceso a los bancos de memoria compartida

- Algunos rasgos de CUDA:

- La memoria compartida consta de 16 (pre-Fermi) ó 32 bancos.
- Los hilos de un bloque se enumeran en orden "column major", esto es, hilos consecutivos difieren en la dimensión x (no en la y).
- Si accedemos de la forma habitual a los vectores en memoria compartida: **As [threadIdx.x] [threadIdx.y]**, los hilos de un mismo warp leerán de la misma columna, esto es, del mismo banco en memoria compartida.
- En cambio, usando **As [threadIdx.y] [threadIdx.x]**, leerán de la misma fila, accediendo a un banco diferente.
- Por tanto, los mosaicos se almacenan y acceden en memoria compartida de forma invertida o **traspuesta**.

Ejemplo de resolución de conflictos a los bancos de memoria compartida

(0,0)(1,0)	warp 0	(31,0)	(0,0)(1,0)	warp 0	(31,0)
(0,1)(1,1)	warp 1	(31,1)	(0,1)(1,1)	warp 1	(31,1)
(0,2)(1,2)	warp 2	(31,2)	(0,2)(1,2)	warp 2	(31,2)

Bloque (0,0)

Bloque (1,0)

(0,29)(1,29)	warp 29	(31,29)	(0,29)(1,29)	warp 29	(31,29)
(0,30)(1,30)	warp 30	(31,30)	(0,30)(1,30)	warp 30	(31,30)
(0,31)(1,31)	warp 31	(31,31)	(0,31)(1,31)	warp 31	(31,31)
(0,0)(1,0)	warp 0	(31,0)	(0,0)(1,0)	warp 0	(31,0)
(0,1)(1,1)	warp 1	(31,1)	(0,1)(1,1)	warp 1	(31,1)
(0,2)(1,2)	warp 2	(31,2)	(0,2)(1,2)	warp 2	(31,2)

Bloque (0,1)

Bloque (1,1)

(0,29)(1,29)	warp 29	(31,29)	(0,29)(1,29)	warp 29	(31,29)
(0,30)(1,30)	warp 30	(31,30)	(0,30)(1,30)	warp 30	(31,30)
(0,31)(1,31)	warp 31	(31,31)	(0,31)(1,31)	warp 31	(31,31)

... (más bloques de 32 x 32 hilos)

→ Hilos consecutivos de un mismo warp difieren en la primera de sus dos dims.

Pero posiciones consecutivas de memoria de una matriz bidimensional alojan datos que difieren en la segunda de sus dims:
 $a[0][0]$, $a[0][1]$, $a[0][2]$, ...

dato	Está en el banco	Si el hilo (x,y) usa $a[x][y]$, el warp accede a	Si el hilo (x,y) usa $a[y][x]$, el warp accede a
$a[0][0]$	0	X	X
$a[0][1]$	1		X
$a[0][31]$	31		X
$a[1][0]$	0	X	
$a[31][0]$	0	X	

100% conflictos

Ningún conflicto

Tiling: El código CUDA para el kernel en GPU

```
__global__ void MxMonGPU(float *A, float *B, float *C, int N)
{
    int sum=0, tx, ty, i, j;
    tx = threadIdx.x;                      ty = threadIdx.y;
    i = blockIdx.x * blockDim.x + tx;      j = blockIdx.y * blockDim.y + ty;
    __shared__ float As[32][32], float Bs[32][32];

    // Recorre los mosaicos de A y B necesarios para computar la submatriz de C
    for (int tile=0; tile<(N/32); tile++)
    {
        // Carga los mosaicos (32x32) de A y B en paralelo (y de forma traspuesta)
        As[ty][tx]= A[(i*N) + (ty+(tile*32))];
        Bs[ty][tx]= B[((tx+(tile*32))*N) + j];
        __syncthreads();
        // Computa los resultados para la submatriz de C
        for (int k=0; k<32; k++) // Los datos también se leerán de forma traspuesta
            sum += As[k][tx] * Bs[ty][k];
        __syncthreads();
    }
    // Escribe en paralelo todos los resultados obtenidos por el bloque
    C[i*N+j] = sum;
}
```

Una optimización gracias al compilador: Desenrollado de bucles (loop unrolling)

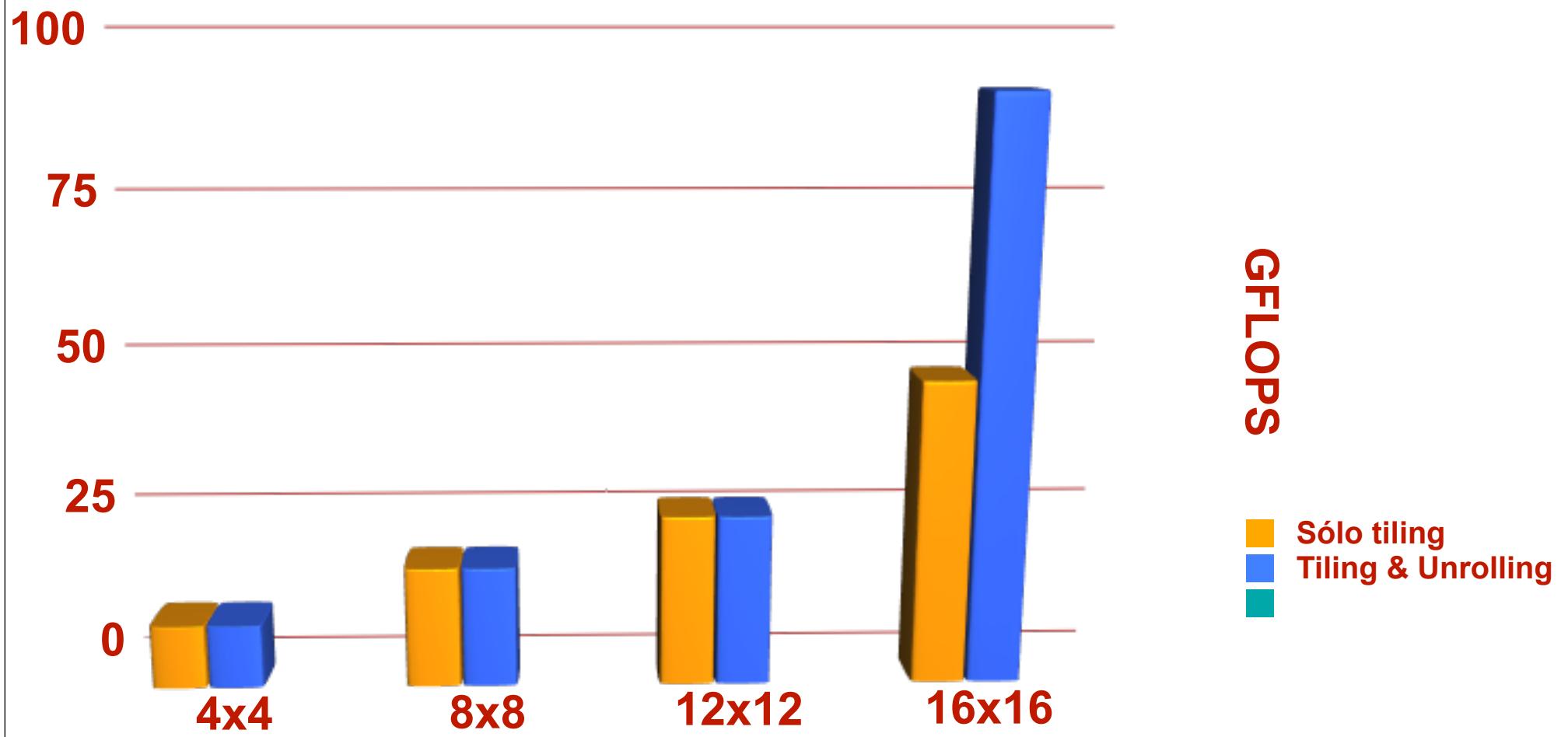
Sin desenrollar el bucle

```
...  
  
__syncthreads();  
  
// Computar la parte de ese mosaico  
for (k=0; k<32; k++)  
    sum += As[tx][k]*Bs[k][ty];  
  
__syncthreads();  
}  
  
C[indexC] = sum;
```

Desenrollando el bucle

```
__syncthreads();  
  
// Computar la parte de ese mosaico  
sum += As[tx][0]*Bs[0][ty];  
sum += As[tx][1]*Bs[1][ty];  
sum += As[tx][2]*Bs[2][ty];  
sum += As[tx][3]*Bs[3][ty];  
sum += As[tx][4]*Bs[4][ty];  
sum += As[tx][5]*Bs[5][ty];  
sum += As[tx][6]*Bs[6][ty];  
sum += As[tx][7]*Bs[7][ty];  
sum += As[tx][8]*Bs[8][ty];  
...  
sum += As[tx][31]*Bs[31][ty];  
__syncthreads();  
}  
  
C[indexC] = sum;
```

Rendimiento con tiling & unrolling en la G80



Tamaño del mosaico (32x32 no es factible en la G80)



VII. Bibliografía y herramientas



CUDA Zone: La web raíz para el programador CUDA

[developer.nvidia.com/cuda-zone]

Libraries

cuRAND NPP
Math Library cuFFT
nvGRAPH NCCL

[See More Libraries](#)

Tools and Integrations

Nsight Visual Profiler
CUDA GDB CUDA MemCheck

OpenACC

CUDA Profiling Tools Interface

[See More Tools](#)

CUDA accelerates applications across a wide range of domains from image processing, to deep learning, numerical analytics and computational science.

COMPUTATIONAL CHEMISTRY MACHINE LEARNING DATA SCIENCE
BIOINFORMATICS COMPUTATIONAL FLUID DYNAMICS WEATHER AND CLIMATE

[More Applications](#)

Get started with CUDA by downloading the CUDA Toolkit and exploring introductory resources including videos, code samples, hands-on labs and webinars.

[Download Now >](#)

[Get Started with CUDA >](#)

Guías de desarrollo y otros documentos

- Para iniciarse con CUDA C: La guía del programador.
 - [\[docs.nvidia.com/cuda/cuda-c-programming-guide\]](https://docs.nvidia.com/cuda/cuda-c-programming-guide)
- Para optimizadores de código: La guía con los mejores trucos.
 - [\[docs.nvidia.com/cuda/cuda-c-best-practices-guide\]](https://docs.nvidia.com/cuda/cuda-c-best-practices-guide)
- La web raíz que aglutina todos los documentos ligados a CUDA:
 - [\[docs.nvidia.com/cuda\]](https://docs.nvidia.com/cuda)
- donde encontramos, además de las guías anteriores, otras para:
 - Instalar CUDA en Linux, MacOS y Windows.
 - Optimizar y mejorar los programas CUDA sobre GPUs Kepler y Maxwell.
 - Consultar la sintaxis del API de CUDA (runtime, driver y math).
 - Aprender a usar librerías como cuBLAS, cuFFT, cuRAND, cuSPARSE, ...
 - Manejar las herramientas básicas (compilador, depurador, optimizador).

Recursos educacionales



Educator Resources

[developer.nvidia.com/educators]

Equipping Educators with Teaching Materials and GPU Computing Tools

The NVIDIA supports educators by providing Teaching Kits and GPU resources for use in university classrooms and labs to empower today's students with the deep learning and accelerated computing skills they'll need tomorrow.

NVIDIA Teaching Kits contain everything instructors need to teach full-term curriculum courses with GPUs in machine and deep learning, robotics, accelerated/parallel computing, and a variety of other academic disciplines. **Click "Join now" to request access to the Teaching Kits or "Member area" if already approved.**

[Join now](#)

Community Forum and Support

Please read our [FAQs](#), visit our forum and email us with any questions, feedback, or suggestions.

[Learn more](#)

DLI Self-Paced Labs and Workshops

The NVIDIA Deep Learning Institute (DLI) offers hands-on- training for those looking to solve challenging problems with deep learning.

[Learn more](#)

Getting Started with GPUs

Suggested labs, libraries, and reference materials for those new to GPU-accelerated computing.

[Learn more](#)

GPU Access and Development Tools

Recommended development tools, CUDA downloads, and other resources.

[Learn more](#)

Existing Course Material

Learn more about real university classes, labs, and MOOCs currently using GPUs.

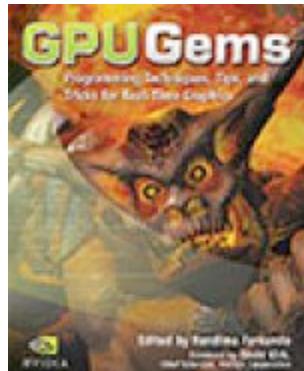
[Learn more](#)

Training Material and Code Samples

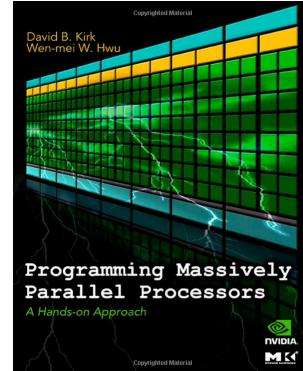
Tutorials, seminars, training slides, and code samples that help teach an array of parallel programming concepts.

[Learn more](#)

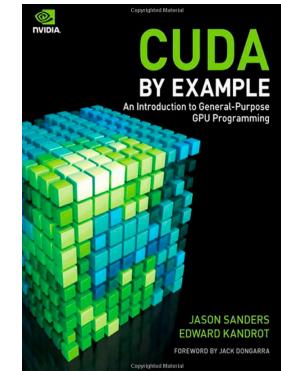
Libros sobre CUDA: Desde 2007 hasta 2015



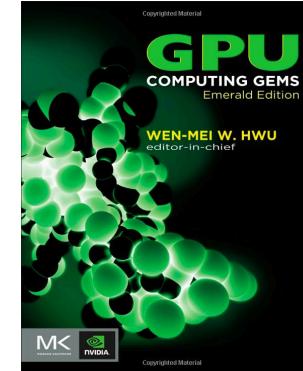
Sep'07



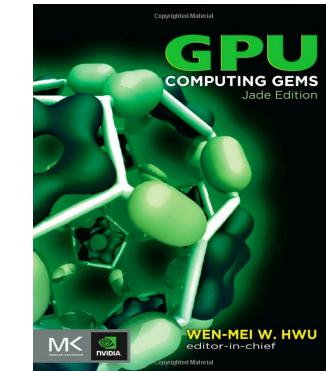
Feb'10



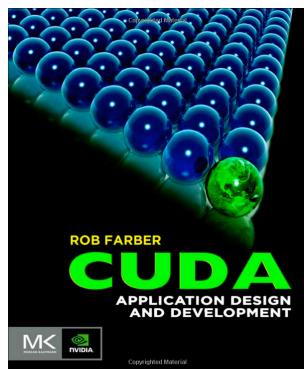
Jul'10



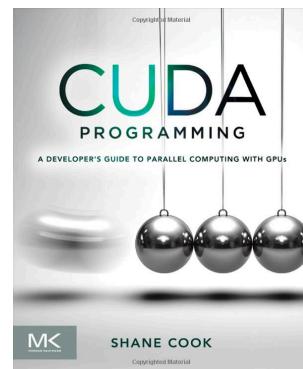
Abr'11



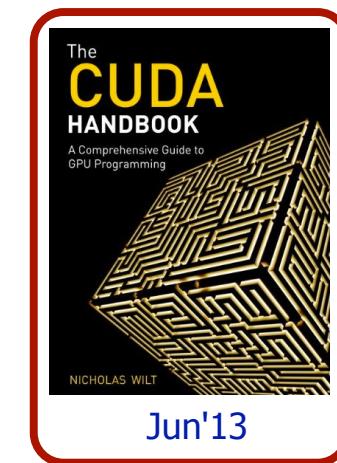
Oct'11



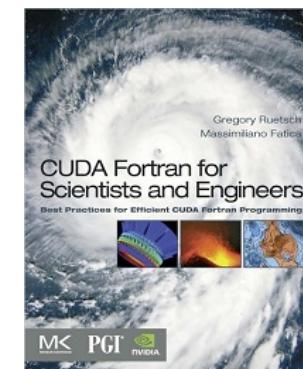
Nov'11



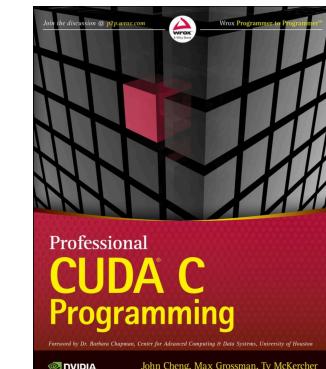
Dic'12



Jun'13



Oct'13



Sep'14

Tutoriales cortos sobre C/C++, Fortran y Python

- Hay que registrarse en la Web de los tutoriales que hay dados de alta en los servicios en la nube de Amazon EC2: [nvidia.qwiklab.com]
- Suelen ser sesiones de 90 minutos.
- Sólo se necesita un navegador de Web y una conexión SSH.
- Algunos tutoriales son gratuitos, otros requieren tokens de \$29.99.

The screenshot shows the NVIDIA QwikLab website interface. At the top, there are links for 'Sign in', 'Create New Account', and 'Language' (English/Japanese). On the left, a sidebar lists categories: 'C/C++ Labs' (4), 'Fortran Labs' (3), and 'Python Labs' (2). A red arrow points from the 'First time?' callout to the 'C/C++ Labs' section. Below the sidebar, a message states: 'This class contains all labs related to the Python programming language.' Two thumbnail cards are shown: 'Accelerating Applications with CUDA Python' and 'Accelerating Applications with GPU-Accelerated Libraries in Python'. In the center, a callout box titled 'Popular tags' lists: 'self-paced', 'Python', 'CUDA Libraries', 'Fortran', 'OpenACC', and 'C++ Optimization'. To the right, several lab cards are displayed:

- NVIDIA CUDA**: GPU Memory Optimizations. Price: 1 Token. Duration: 01 h:30 m. Tags: OpenACC. Access: 02 h:00 m. Levels: Beginner. Unrated.
- OpenACC**: Directives For Accelerators. OpenACC - 2X in 4 Steps in C/C++. Price: 1 Token. Duration: 01 h:30 m. Tags: OpenACC. Access: 02 h:00 m. Levels: Beginner. Unrated.
- NVIDIA CUDA**: Accelerating Applications with CUDA C/C++. Price: 1 Token. Duration: 01 h:30 m. Tags: self-paced. C. Access: 02 h:00 m. Levels: Beginner. Unrated.
- NVIDIA CUDA**: Accelerating Applications with GPU-Accelerated Libraries in C/C++. Price: 1 Token. Duration: 01 h:30 m. Tags: CUDA. C. C++. Access: 02 h:00 m. Levels: Beginner. Unrated.

Each card includes the NVIDIA logo and 'CUDA Cloud Training' text at the bottom.

Charlas y webinarios

- Charlas grabadas @ GTC (Graphics Technology Conference):
 - Más de 500 charlas en cada una de las últimas ediciones (2013-18):
 - [\[www.gputechconf.com/gtcnew/on-demand-gtc.php\]](http://www.gputechconf.com/gtcnew/on-demand-gtc.php)
- Webinarios sobre computación en GPU:
 - Listado histórico de charlas en vídeo (mp4/wmv) y diapositivas (PDF).
 - Listado de próximas charlas on-line a las que poder apuntarse.
 - [\[developer.nvidia.com/gpu-computing-webinars\]](http://developer.nvidia.com/gpu-computing-webinars)

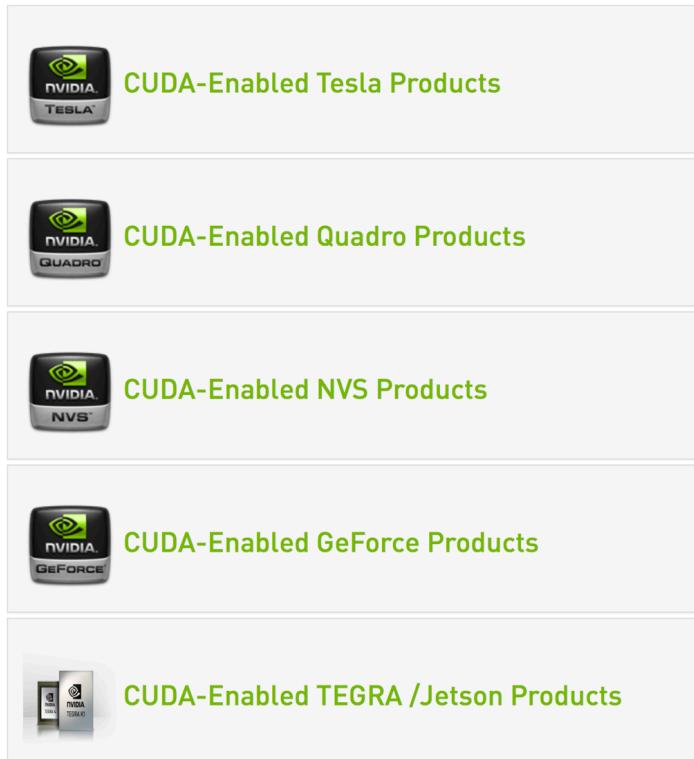
Desarrolladores

- Para firmar como desarrollador registrado:
 - [developer.nvidia.com/developer-program]
 - Acceso a las descargas exclusivas para desarrolladores.
 - Acceso exclusivo a las versiones más avanzadas de CUDA.
 - Actividades exclusivas y ofertas especiales.
- Lanzar cuestiones técnicas o preguntar dudas on-line:
 - NVIDIA Developer Forums: [devtalk.nvidia.com]
 - Busca respuestas o suscribe preguntas: [stackoverflow.com/tags/cuda]

Desarrolladores (2)

- Listado oficial de GPUs que soportan CUDA:

● [\[developer.nvidia.com/cuda-gpus\]](http://developer.nvidia.com/cuda-gpus)



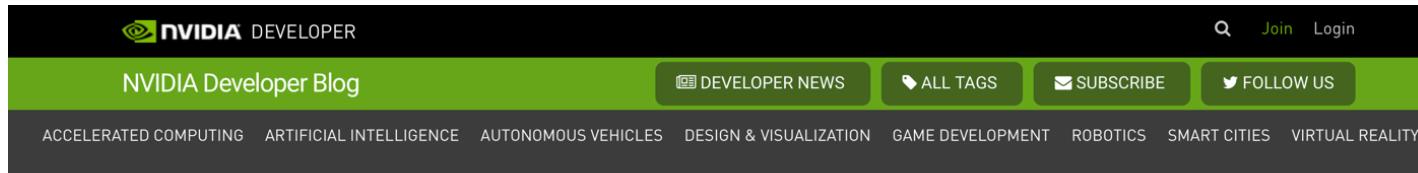
- Y una última herramienta:
El CUDA Occupancy Calculator

● [\[developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls\]](http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls)

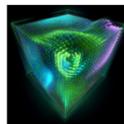
Los artículos más actuales sobre computación acelerada en GPU

- El blog de Nvidia, con los posts más novedosos en torno a CUDA y computación acelerada en GPU:

● <https://devblogs.nvidia.com>



Accelerated Computing



GPU-accelerated computing is the use of a graphics processing unit (GPU) together with a CPU to accelerate deep learning, analytics, and engineering applications. With **NVIDIA ComputeWorks** SDKs, you can develop, optimize and deploy GPU-accelerated applications using widely-used languages such as C, C++, Python, Fortran and MATLAB.

A grid of three blog post thumbnails. The first thumbnail on the left shows the NVIDIA logo and the TensorFlow logo. The second thumbnail in the middle shows a close-up of a GPU card. The third thumbnail on the right shows a close-up of a printed circuit board (PCB) with various components. Below each thumbnail is a brief description of the article and the author's name.

TensorRT Integration Speeds Up
TensorFlow Inference

By Sami Kama, Julie Bernauer and Siddharth Sharma |

Storage Performance Basics for Deep
Learning

By James Mauro | March 21, 2018

Using CUDA Warp-Level Primitives

By Yuan Lin and Vinod Grover | January 15, 2018

Muchas gracias por vuestra atención

- Siempre a vuestra disposición en el Departamento de Arquitectura de Computadores de la Universidad de Málaga
 - e-mail: ujaldon@uma.es
 - Teléfono: +34 952 13 28 24.
 - Página Web: <http://manuel.ujaldon.es>
(versiones en castellano e inglés).

