

Programación de GPUs con CUDA

Universidad de Chile
Santiago, del 27 al 29 de Enero de 2016



Manuel Ujaldón

Profesor Titular @ Universidad de Málaga
Conjoint Senior Lecturer @ Univ. de Newcastle (Australia)
CUDA Fellow @ Nvidia

Resumen de mis actividades CUDA Fellow

Evento	Lugar	Ciudad
Charla de CUDA	Departamento de Física. Universidad del Bío-Bío	Concepción (Chile)
Charla invitada y seminario de GPUs	Workshop sobre HPC y Big Data. Centro de Modelación y Computación Científica (CEMCC). Facultad de Ingeniería y Ciencias. Universidad de La Frontera	Temuco (Chile)
Curso de CUDA	Pontificia Universidad Católica de Chile (PUC)	Santiago (Chile)
Curso de CUDA	Universidad Andrés Bello (UNAB)	Santiago (Chile)
Curso de CUDA	Facultades de Ingeniería Eléctrica e Ingeniería de Sistemas Universidad Tecnológica de Panamá (UTP)	Ciudad de Panamá (Panamá)
Curso de CUDA	Campus Regional de Coclé Universidad Tecnológica de Panamá (UTP)	Penonomé (Panamá)
Curso de CUDA	Campus Regional de Chiriquí Universidad Tecnológica de Panamá (UTP)	David (Panamá)
Tutorial de CUDA	21st ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'16)	Barcelona (España)
Tutorial de CUDA	22nd IEEE Intl. Symposium on High Performance Computing Architecture (HPCA'16)	Barcelona (España)
Curso de GPUs	ETS de Ingeniería de Telecomunicaciones. Universidad de Málaga	Málaga (España)
Tutorial de CUDA	14th Intl. Conference on Computational Science and Its Applications (ICCSA'16)	Pekín (China)
Charla invitada	14th Intl. Conference on Computational Science and Its Applications (ICCSA'16)	Pekín (China)
Curso de GPUs	XXII Congreso Argentino en Ciencias de la Computación (CACIC). Universidad Nacional de San Luis	San Luis (Argentina)
Curso de GPUs	Escuela de Sistemas y Computación. Facultad de Ingeniería. Universidad Nacional de Chimborazo (UNACH)	Riobamba (Ecuador)
Curso de GPUs	Facultad de Informática y Electrónica. Escuela Superior Politécnica de Chimborazo (ESPOCH)	Riobamba (Ecuador)
Curso de GPUs	Facultad de Ingeniería Mecánica y Ciencias de la Producción. Escuela Superior Politécnica del Litoral (ESPOL)	Guayaquil (Ecuador)

País	2012	2013	2014	2015	2016	Total
España	10	3	12	6	3	34
Chile	2	2	2		4	10
Sudáfrica		4	1	1		6
Argentina		3	2		1	6
Brasil		2	1	3		6
Australia	2	1		2		5
Nueva Zelanda	2			3		5
Italia		1		3		4
Portugal				4		4
Ecuador					3	3
Panamá					3	3
Uruguay	2					2
Francia	1	1				2
Austria	1	1				2
Polonia				2		2
Colombia				2		2
China					2	2
Suecia		1				1
Finlandia		1				1
Perú			1			1
20 países	20	20	19	26	16	101

Mi última visita al CMM dos años atrás



Mi próximo curso la semana próxima en la sede de la UNAB en Viña del Mar

- Si conocéis a alguien que quiera inscribirse, el enlace es:
 - <http://han.ing.unab.cl/index.php/9-yt-sample-data/category1/90-intro-a-programacion-gpu>
- Todo cortesía de Roberto León de la UNAB, organizador del curso en la sede de Viña del Mar.

Agenda del curso

Día	Franja horaria	Contenidos	Nivel
27/1	09:30 - 11:00	Arquitectura de la GPU y diseños many-core	Básico
27/1	11:00 - 11:30	Descanso	
27/1	11:30 - 13:00	Programación CUDA: Hilos, bloques, núcleos, mallas	Básico
27/1	13:00 - 14:30	Almuerzo libre	
27/1	14:30 - 16:00	Herramientas CUDA: Compilador, calculador de ocupación, ...	Intermedio
Día	Franja horaria	Contenidos	Nivel
28/1	09:30 - 11:00	Ejemplos CUDA: VectorAdd, Stencils, ReverseArray, MxM	Intermedio
28/1	11:00 - 11:30	Descanso	
28/1	11:30 - 13:00	Kepler y Maxwell: Hyper-Q, par. dinámico, memoria unificada	Básico
28/1	13:00 - 14:30	Almuerzo libre	
28/1	14:30 - 16:00	OpenACC y otras alternativas de programación para GPGPU	Básico
Día	Franja horaria	Contenidos	Nivel
29/1	09:30 - 12:30	Prácticas individuales (1): Programando GPUs en la nube	Avanzado
29/1	12:30 - 14:00	Almuerzo libre	
29/1	14:00 - 16:00	Prácticas individuales (2): Programando GPUs en la nube	Avanzado

Contenidos [160 diapositivas]

1. Introducción. [27 diapositivas]
2. Arquitectura. [37]
 1. El modelo hardware de CUDA. [3]
 2. Primera generación: Tesla (2007-2009). [3]
 3. Segunda generación: Fermi (2010-2011). [4]
 4. Tercera generación: Kepler (2012-2015). [9]
 5. Cuarta generación: Maxwell (2015-2016). [7]
 6. Quinta generación: Pascal (2016-?). [10]
3. Programación. [23]
4. Sintaxis. [16]
 1. Elementos básicos. [10]
 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. [15]

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.

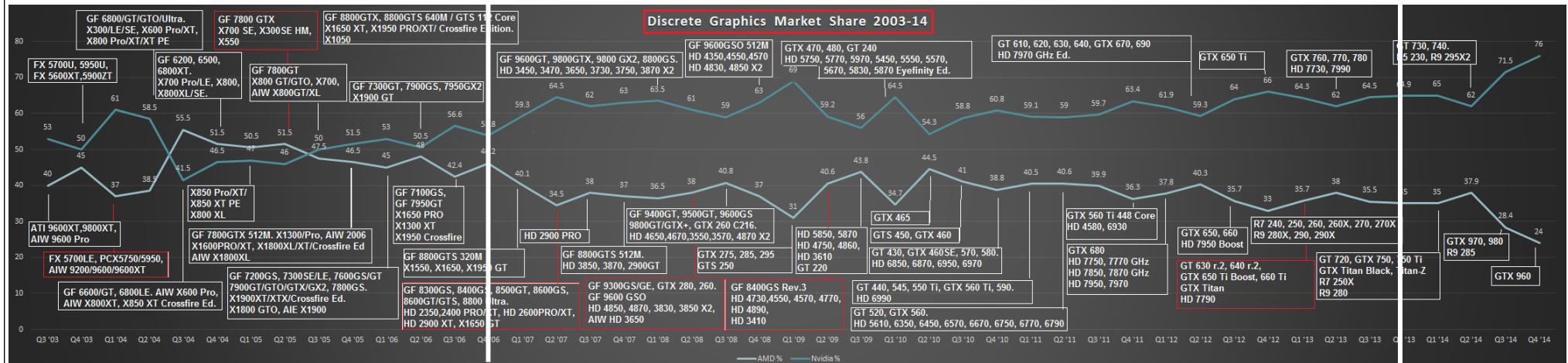


I. Introducción



El pasado: La cuota de mercado de las GPUs

Fuente: Consultora Jon Peddie



	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	1Q15	2Q15
Nvidia	50%	53%	51%	55%	63%	63%	64%	61%	64%	66%	65%	76%	77%	81%
AMD	45%	46%	45%	46%	37%	37%	36%	39%	36%	33%	35%	24%	22%	18%

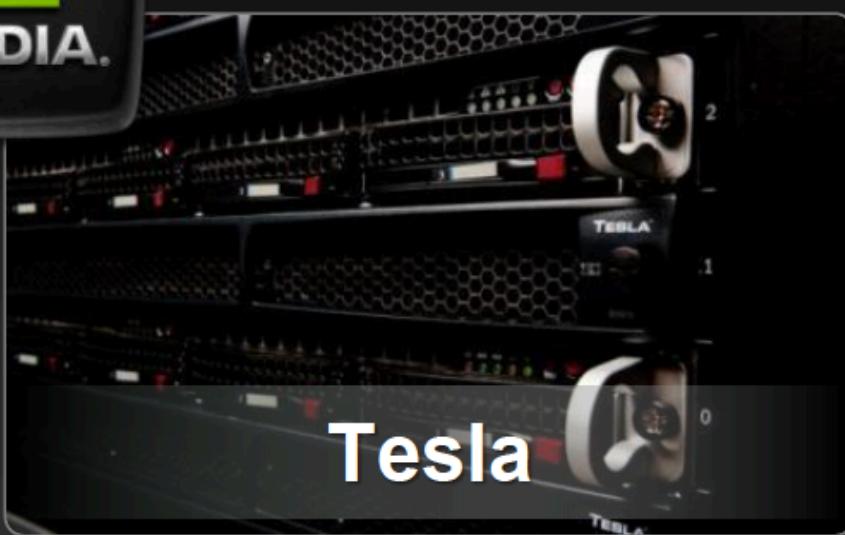
Era pre-CUDA: 1-1

Período estable de 7 años: 2-1

3-1

4-1

Bienvenido al mundo de las GPUs



Modelos comerciales de Kepler: GeForce y Tesla frente a frente



GeForce GTX Titan



Diseñada para jugar:

- El precio es prioritario (<500€).
- Gran disponibilidad/popularidad.
- Poca memoria de vídeo (1-2 GB.).
- Relojes un poco más rápidos.
- Hyper-Q sólo para streams CUDA.
- 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 (6-12 GB.).
- Ejecución sin descanso (24/7).
- Hyper-Q para procesos MPI.
- 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

Año 2008

100.000.000
GPUs aceptan CUDA
(6.000 son Teslas)



150.000
descargas de CUDA



1
supercomputador
en el top500.org
(77 TFLOPS)



60
cursos universitarios



4.000
artículos científicos



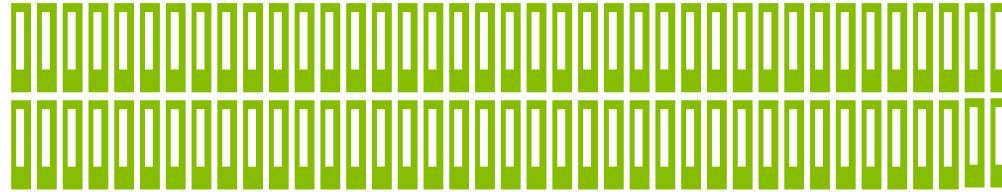
Año 2015



600.000.000 GPUs aceptan CUDA
(y 450.000 son Teslas)



3.000.000 descargas anuales de CUDA
(una cada 9 segundos)



104 supercomputadores
en el TOP500.org
(acumulado: 54.000 TFLOPS)

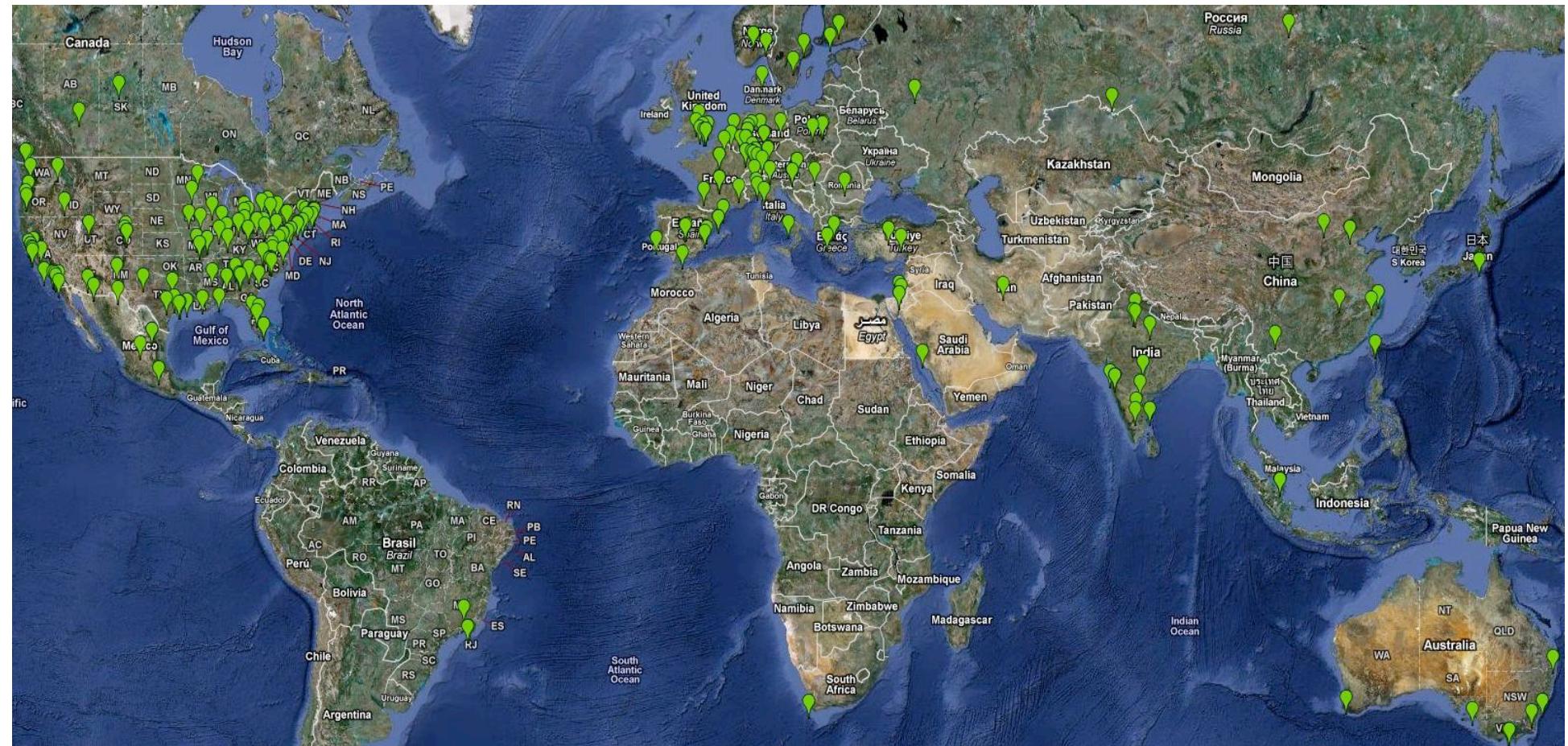


840 cursos universitarios



60.000
artículos científicos

Distribución mundial de las 840 universidades que imparten cursos de CUDA



Las expectativas siguen vigentes

5 PREDICTIONS: WHERE SUPERCOMPUTING IS HEADING IN 2016

Posted 4 days, 3 hours ago

by Barry Bolding, Cray

From new processor technologies to quantum computing, 2016 promises to be another exciting year for supercomputing. Here are five predictions as to how the industry will push ahead in 2016:

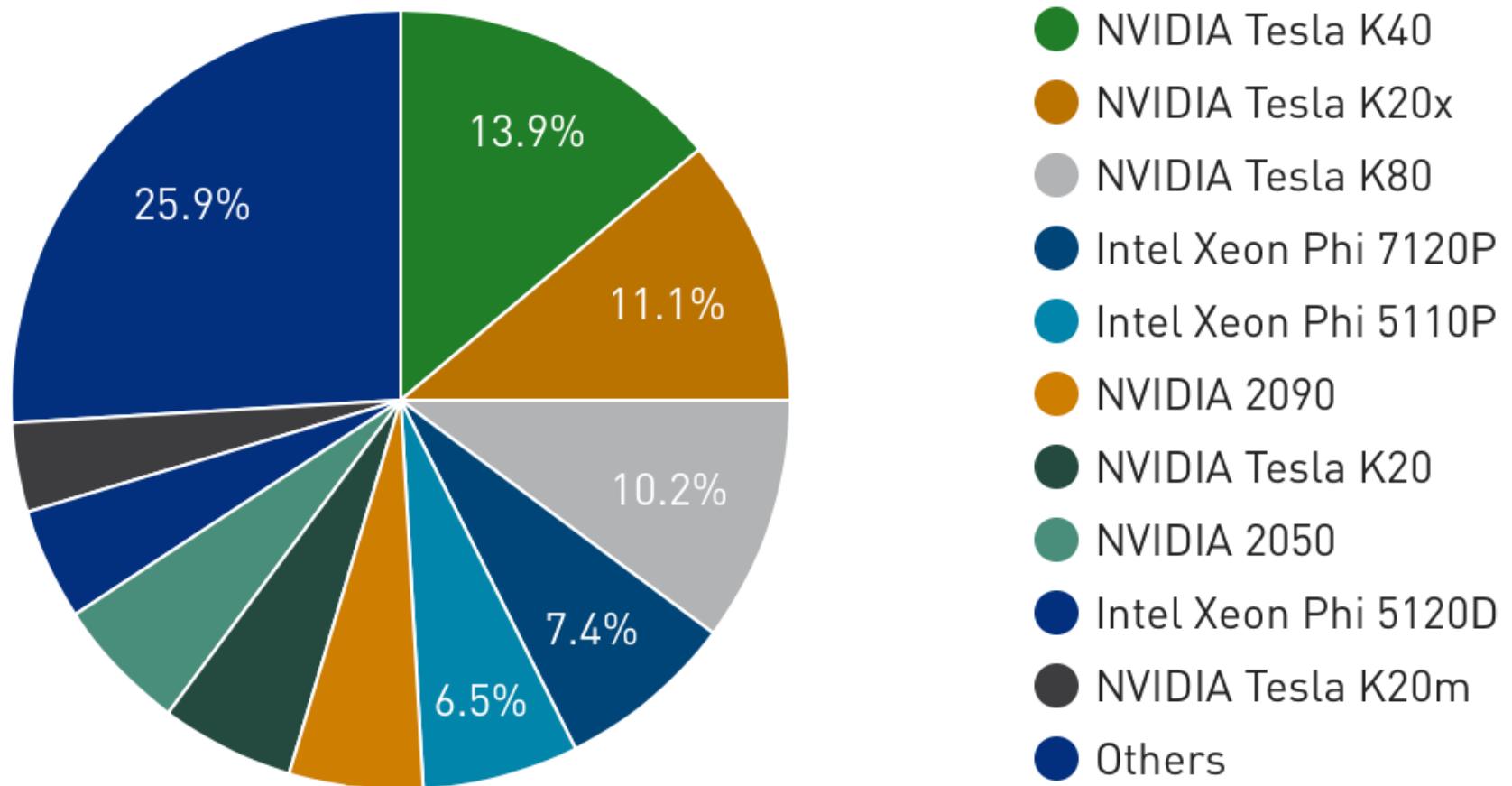
1. The year of progress in new processor technologies

There are a number of exciting technologies we should see in 2016, and a leader will be Intel's next-generation Xeon Phi coprocessor — a hybrid between an accelerator and general purpose processor. This new class of processors will have a large impact on the industry with its innovative design that combines a many-core architecture with general-purpose productivity. Cray, for example, will be delivering Intel Xeon Phi processors with some of our largest systems, including those going to Los Alamos National Labs (the "Trinity" supercomputer) and NERSC (the "Cori" supercomputer).

2016 should also see innovation and advances by NVIDIA with their cutting-edge Pascal GPUs, and the continuing ARM innovations will also drive a competitive environment in the processor game.

104 supercomputadores del top500.org utilizan las GPUs

Accelerator/Co-Processor System Share



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. Un paso decisivo en la convergencia de los modelos de programación para CPU y GPU.
- 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.
- 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, conmutación 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.

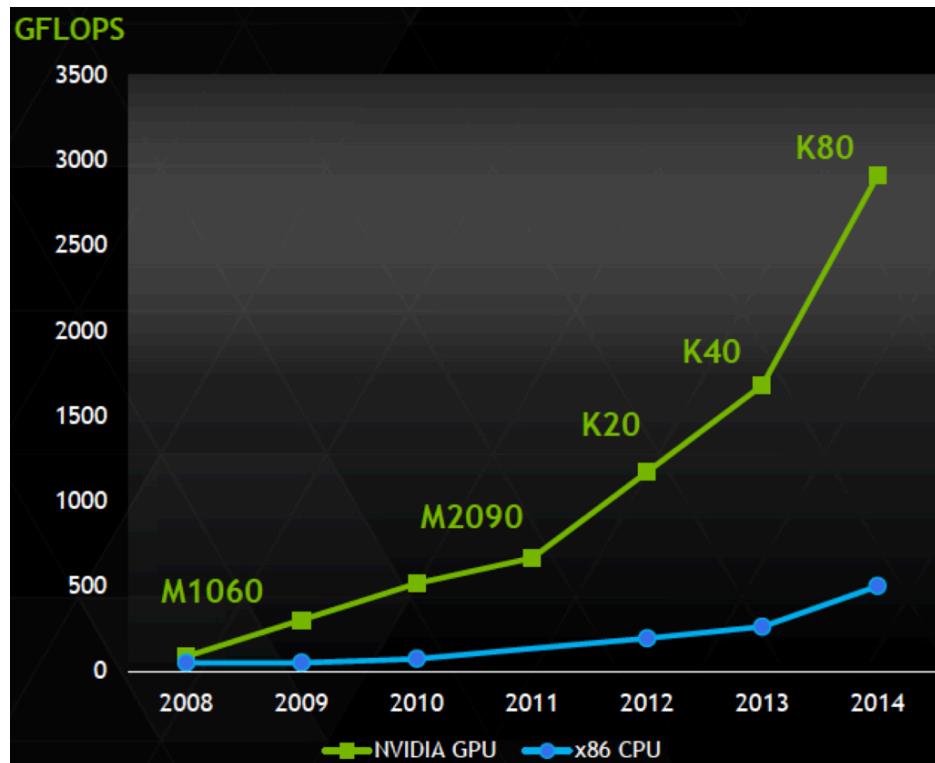
● En memoria dinámica (DRAM):

- 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).

Rendimiento pico frente a la CPU

GFLOPS pico (fp64)

Ancho de banda pico



- GPU 6x mejor en “double”:

- GPU: 3000 GFLOPS
- CPU: 500 GFLOPS

- GPU 6x ancho de banda:

- 7 GHz x 48 bytes = 336 GB/s.
- 2 GHz x 32 bytes = 64 GB/s.

¿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.

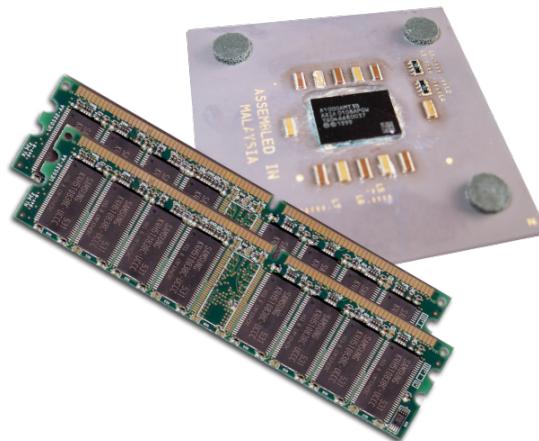
- Objetivos:

- 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 en 2015.



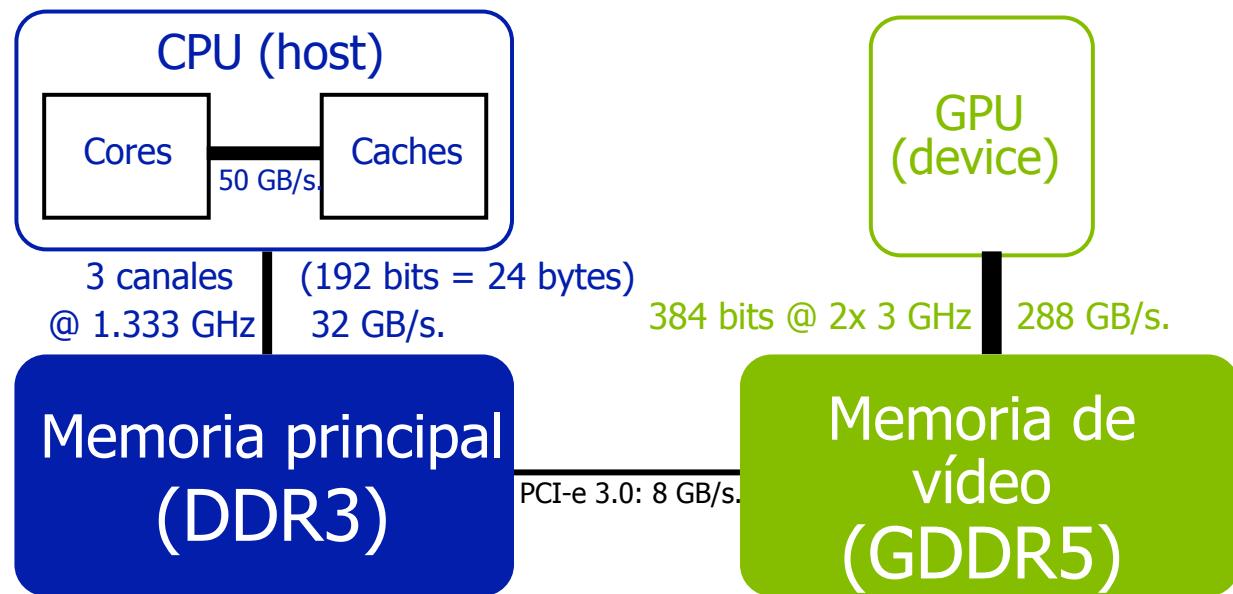
Host



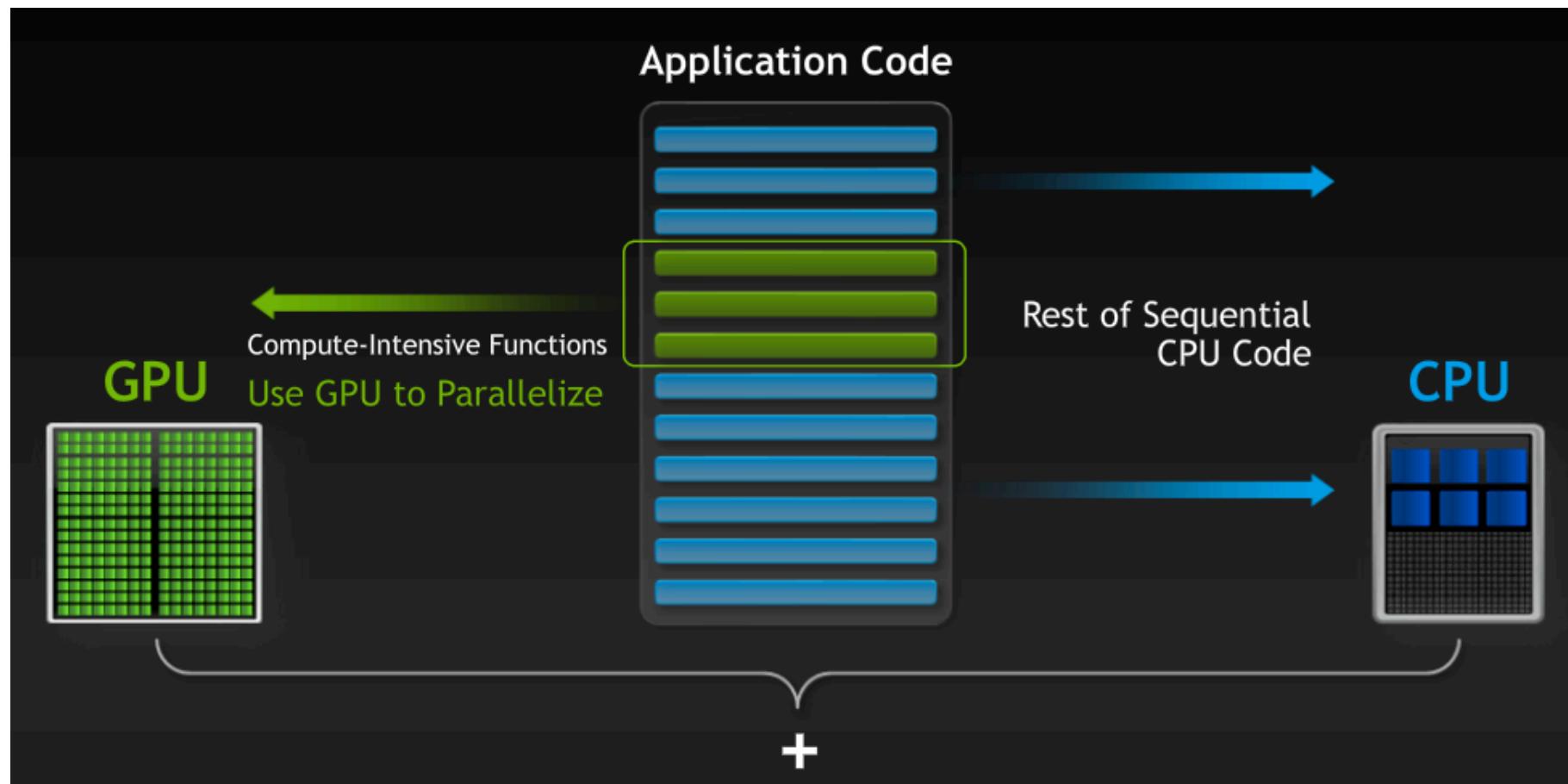
Device

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);

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

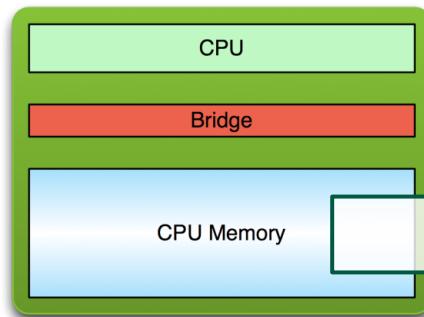
CODIGO DEL DISPOSITIVO:
Función paralela escrita en CUDA.

CODIGO DEL HOST:

- Código serie.
- Código paralelo.
- Código serie.

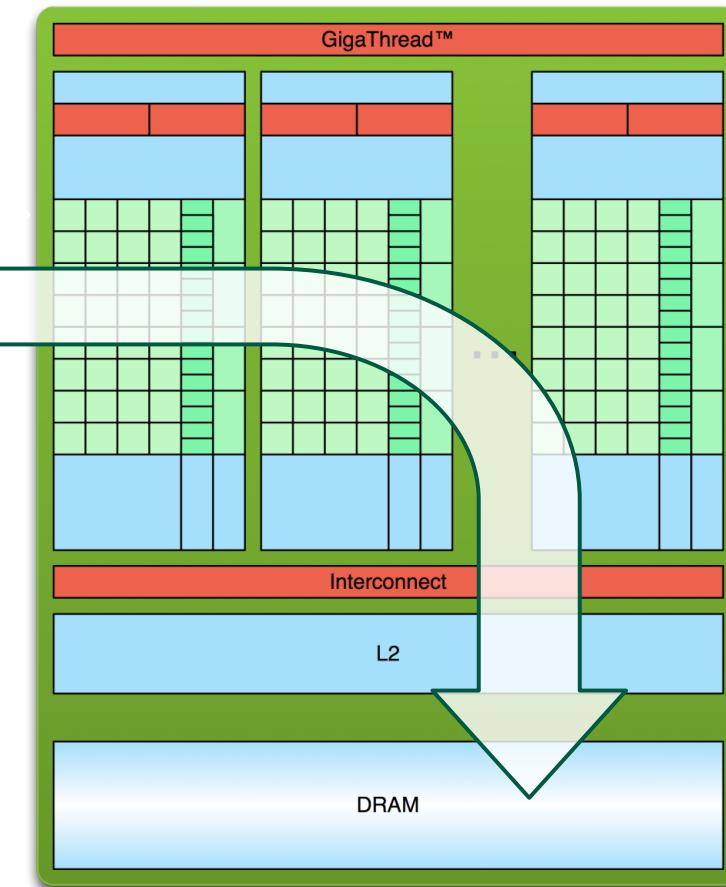


Un sencillo flujo de procesamiento (1/3)

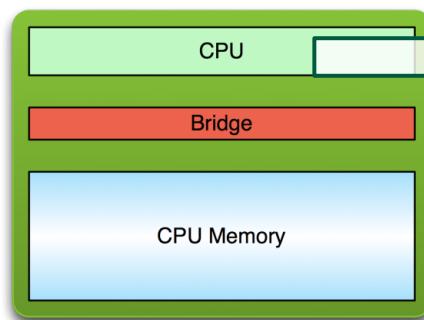


Bus PCI

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

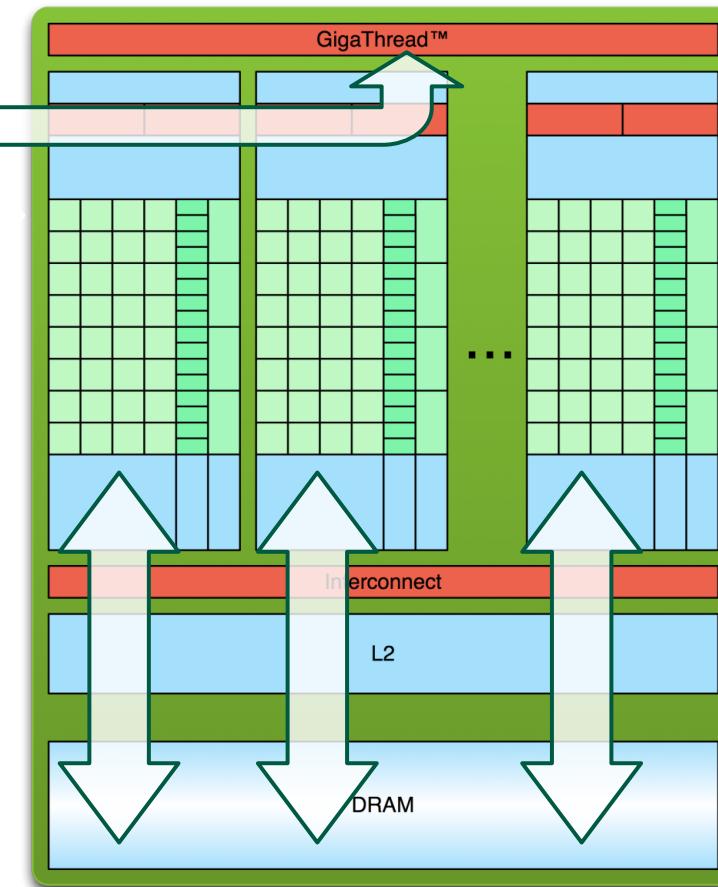


Un sencillo flujo de procesamiento (2/3)

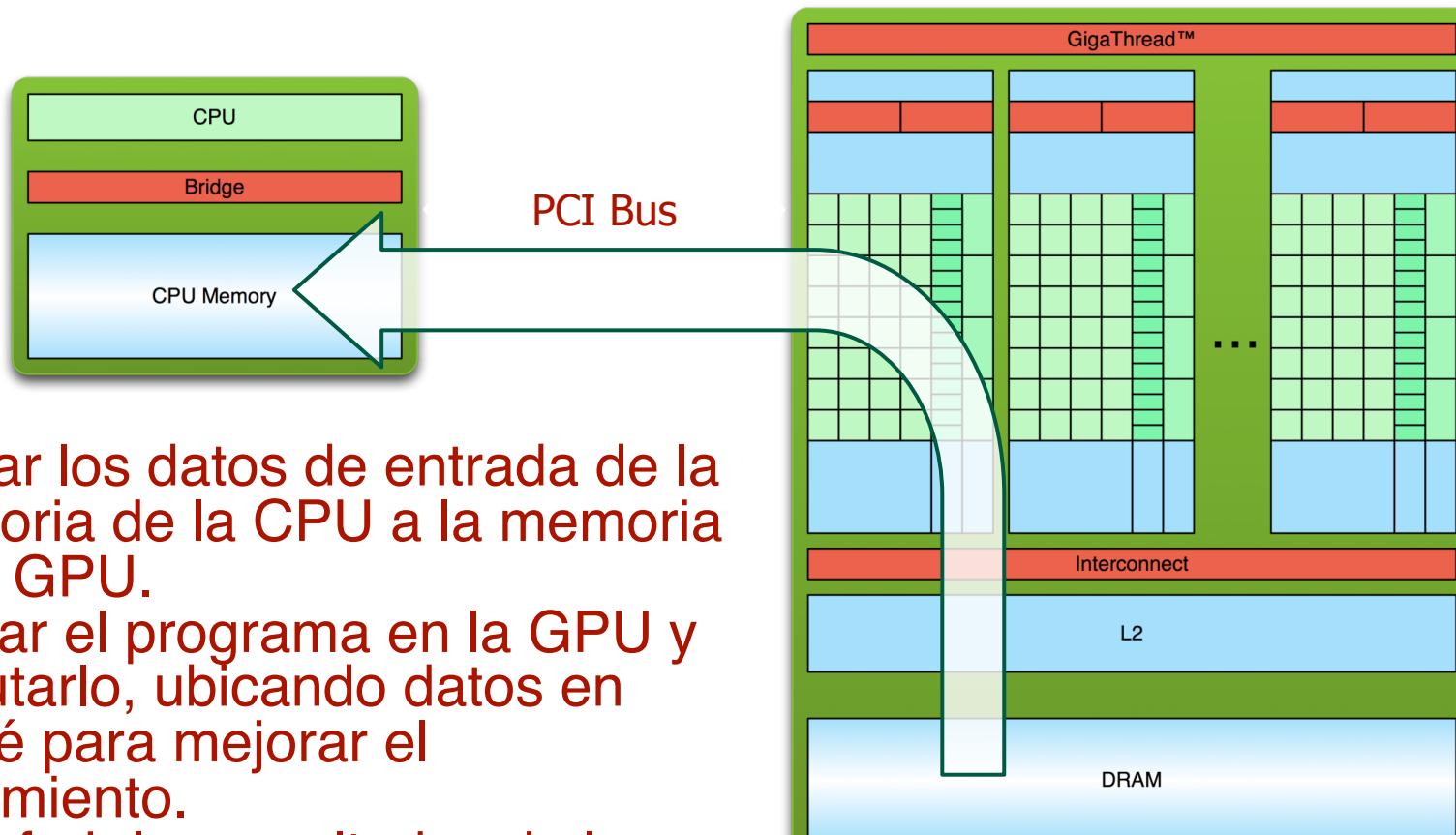


PCI Bus

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.



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)
{
}

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

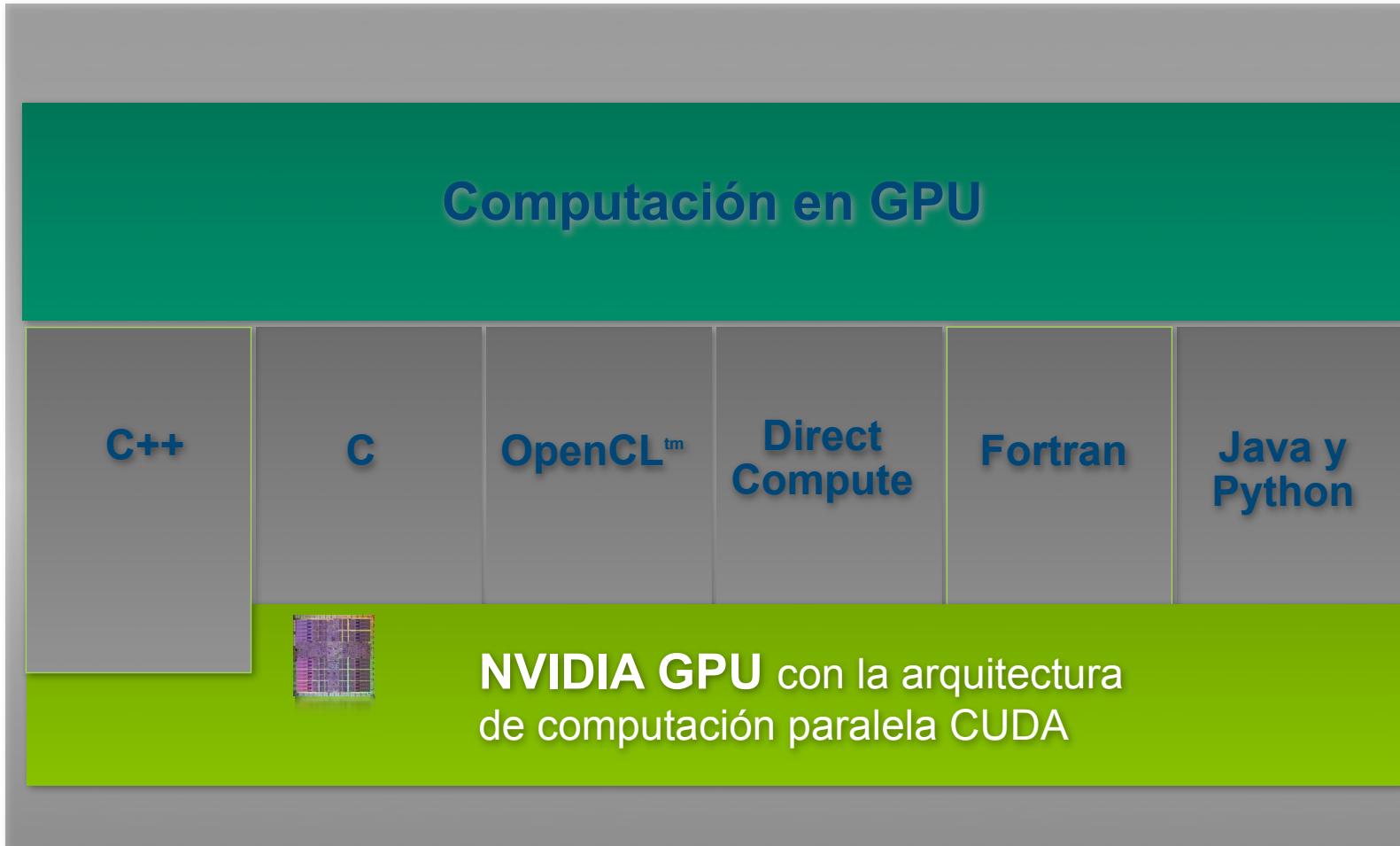
```
global__ void mikernel(void)
{
}
int main(void)
{
    mikernel<<<1,1>>>();
    printf("¡Hola mundo!\n");
    return 0;
}
```

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 paralelismo (bloques e hilos CUDA).

Si tenemos una arquitectura CUDA, podemos programarla de muy diversas formas...



... aunque este tutorial se focaliza sobre CUDA C.

La evolución de CUDA

- En los últimos 7 años, Nvidia ha vendido más de 500 millones de GPUs que aceptan CUDA para su programación.
- Pero CUDA ha evolucionado en la dirección opuesta a la que estamos acostumbrados: Partiendo de los investigadores para poco a poco llegar a usuarios más generalistas.

Versión de CUDA [año]	Usuarios y rasgos más sobresalientes
1.0 [2007]	Muchos investigadores y algunos usuarios madrugadores.
2.0 [2008]	Científicos y aplicaciones para computación de altas prestaciones.
3.0 [2009]	Líderes en la innovación de aplicaciones.
4.0 [2011]	Adopción mucho más extensa de desarrolladores.
5.0 [2012]	Paralelismo dinámico, enlazado de objetos, Remote DMA.
6.0 [2014]	Memoria unificada CPU-GPU
Próximamente	Operandos de punto flotante de 16 bits (half)



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)

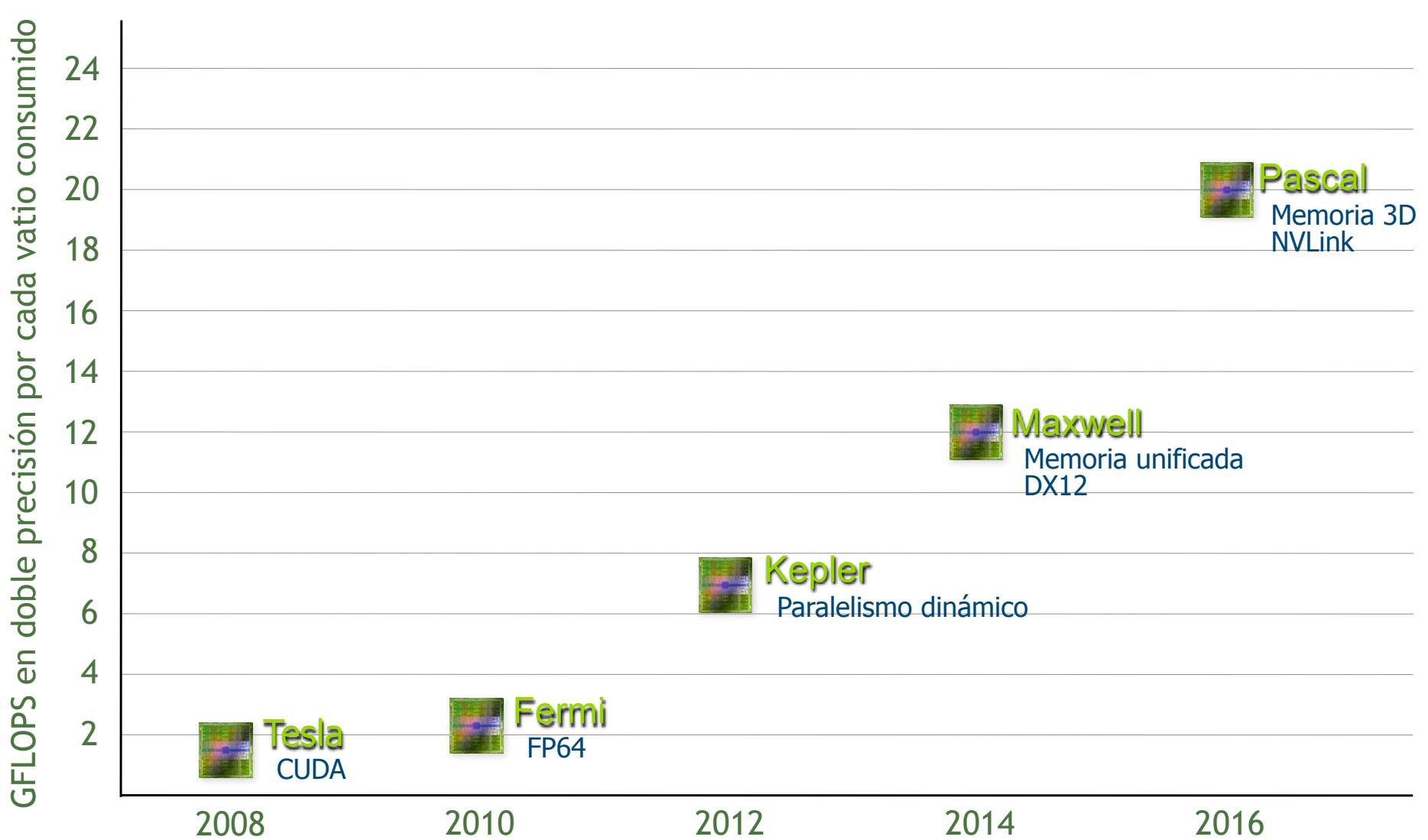
Capítulo 9, página 569



II.1. El modelo hardware de CUDA



Las 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).

● Paralelismo masivo:

- Aplicado sobre miles de hilos.
- Compartiendo datos a diferentes niveles de una jerarquía de memoria.

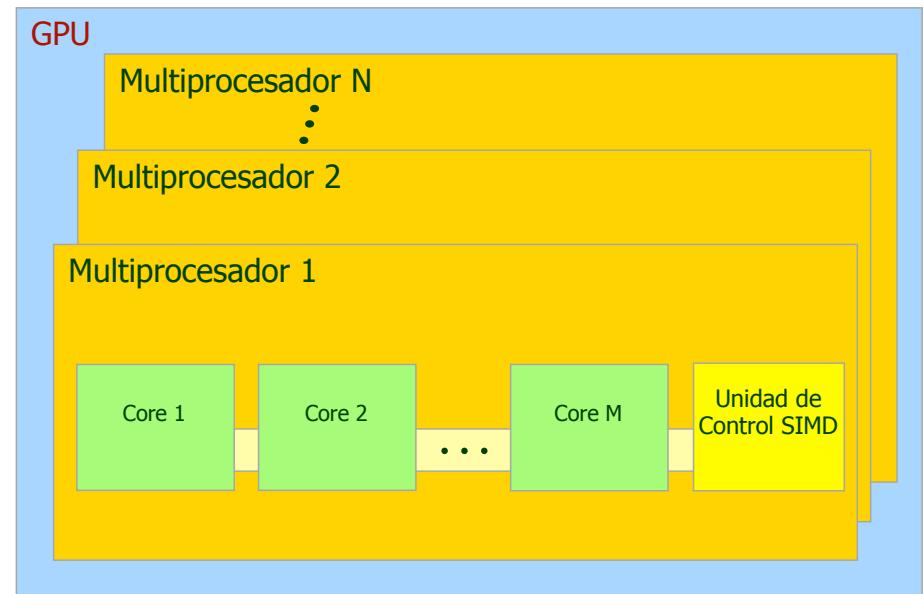
● Computación heterogénea:

● GPU:

- Intensiva en datos.
- Paralelismo fino.

● CPU:

- Gestión y control.
- Paralelismo grueso.



	G80 (Tesla)	GT200 (Tesla)	GF100 (Fermi)	GK110 (Kepler)	(GM200) Maxwell
Marco temporal	2006-07	2008-09	2010-11	2012-13	2014-15
N (multiprocs.)	16	30	14-16	13-15	4-24
M (cores/multip.)	8	8	32	192	128
Número de cores	128	240	448-512	2496-2880	512-3072

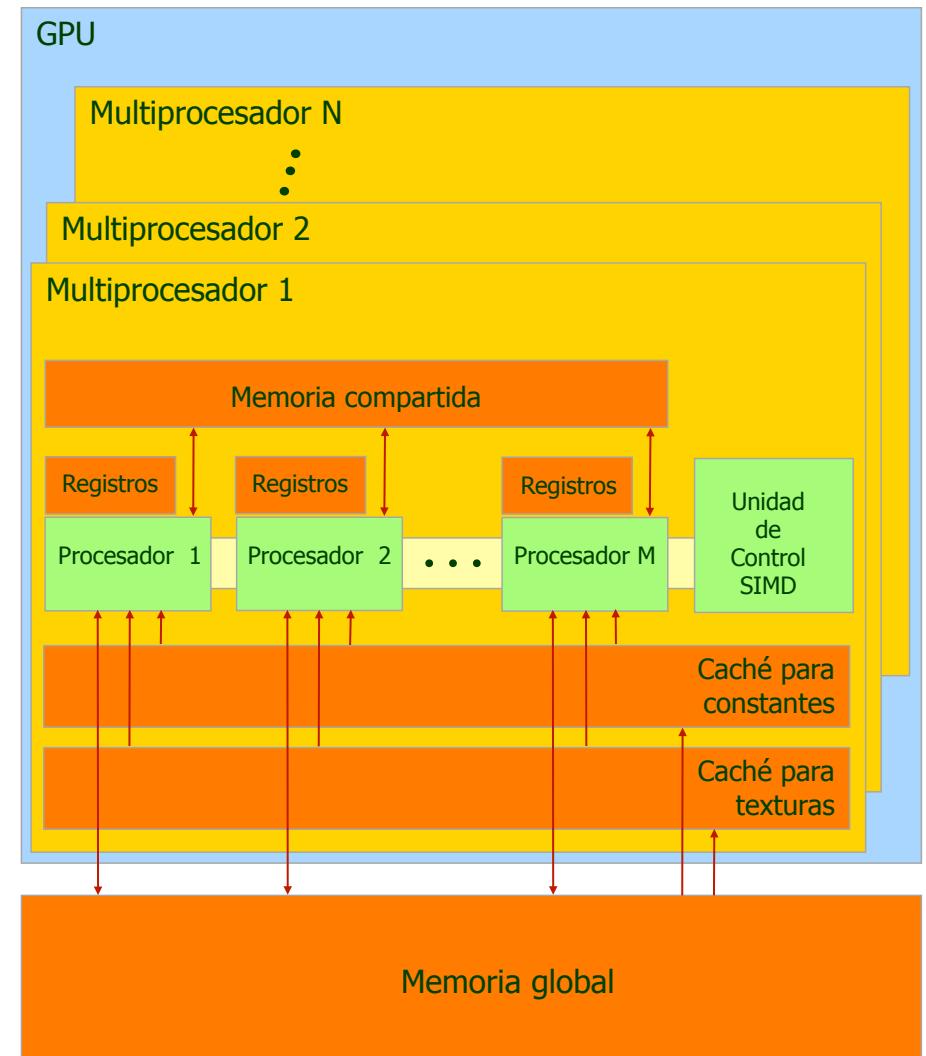
Jerarquía de memoria

- 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. La primera generación: Tesla (G80 y GT200)



La primera generación: G80 (GeForce 8800)

GPU G80 (en torno a 600 MHz, frecuencia muy inferior a la de sus cores)

Multiprocesador 16

Multiprocesador 2

Multiprocesador 1 (los bloques de código CUDA se mapean sobre los multipr.)

Memoria compartida (16 KB)

Registros

Core 1
(1.35 GHz)

Registros

Core 2

Registros

Core 8

Caché de texturas

Unidad de control
(emite instrucciones SIMD)

(los kernels se mapean sobre los cores)

Memoria global (hasta 1.5 GB) (GDDR3 @ 2x 800MHz)

La primera generación: GT200 (GTX 200)

GPU GTX 200 (en torno a 600 MHz)

Multiprocesador 30

Multiprocesador 2

Multiprocesador 1 (los bloques de código CUDA se mapean sobre los multipr.)

Memoria compartida (16 KB)

Registros

Core 1
(1.30 GHz)

Registros

Core 2

Registros

Core 8

Caché de texturas

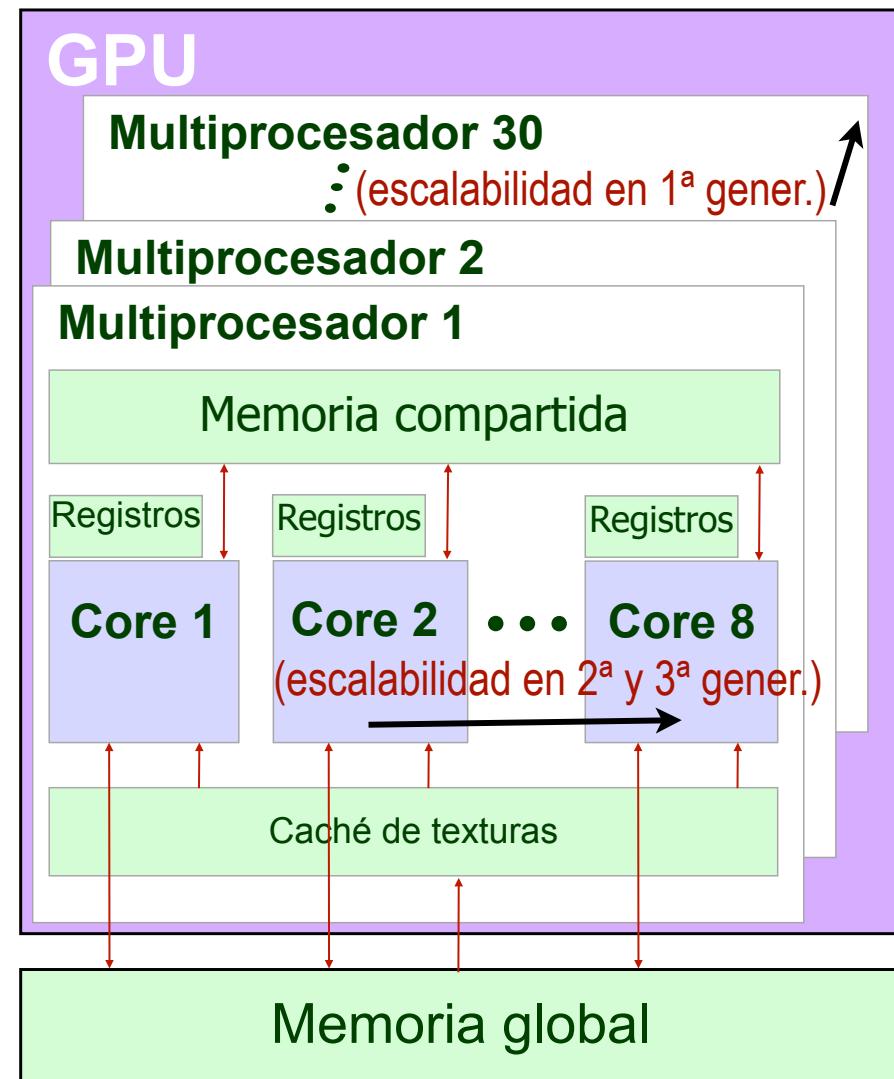
Unidad de control
(emite instrucciones SIMD)

(los kernels se mapean sobre los cores)

Memoria global (hasta 4 GB) (GDDR3, 512 bits @ 2x 1.1GHz = 141.7 GB/s)

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. La segunda generación: Fermi (GFxxx)

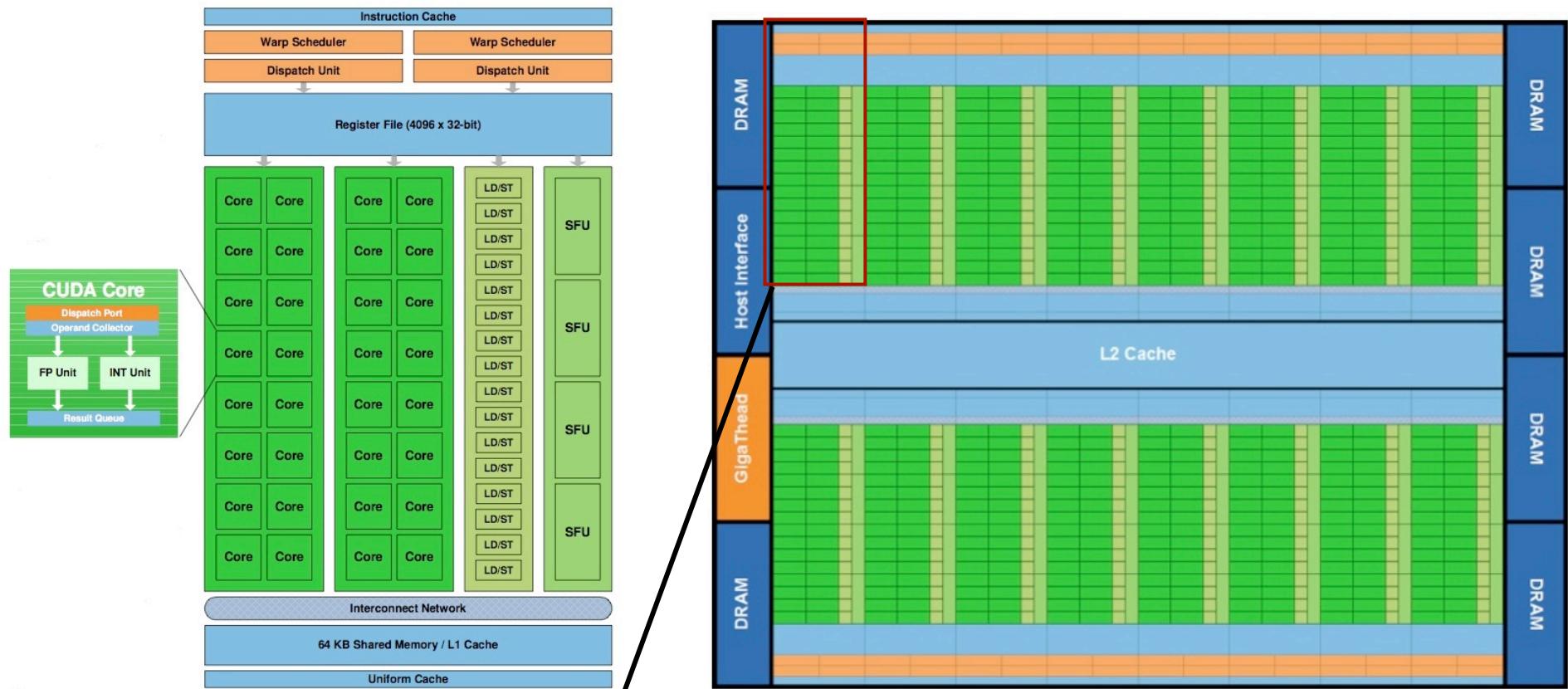


El hardware de Fermi comparado con modelos de la generación anterior

Arquitectura GPU	G80	GT200	GF110 (Fermi)
Nombre comercial	GeForce 8800	GTX 200	GTX 580
Año de lanzamiento	2006	2008	2010
Número de transistores	681 millones	1400 millones	3000 millones
Número de cores (int y fp32)	128	240	512
Número de cores (fp64)	0	30	256
Velocidad de cálculo en fp64	Ninguna	30 madds/ciclo	256 madds/ciclo
Planificadores de warps	1	1	2
Memoria compartida	16 KB	16 KB	16 KB + 48 KB (o viceversa)
Caché L1	Ninguna	Ninguna	
Caché L2	Ninguna	Ninguna	768 KB
Corrección de errores (DRAM)	No	No	Sí
Anchura del bus de direcciones	32 bits	32 bits	64 bits

Arquitectura global de Fermi

- Hasta 512 cores (16 SMs dotados de 32 cores cada uno).
- Doble planificador de hilos en el front-end de cada SM.
- 64 KB. en cada SM: memoria compartida + caché L1.



Mejoras en la aritmética

Unidades aritmético-lógicas (ALUs):

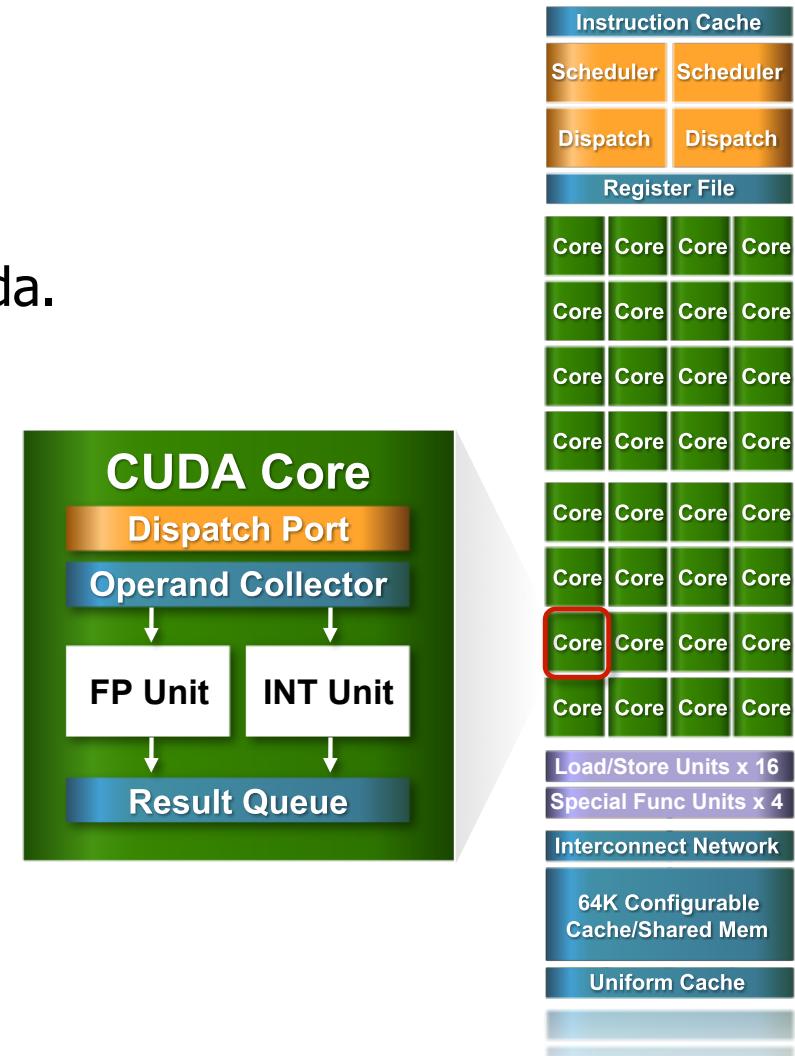
- Rediseñada para optimizar operaciones sobre enteros de 64 bits.
- Admite operaciones de precisión extendida.

La instrucción “madd” (suma y producto simultáneos):

- Está disponible tanto para simple como para doble precisión.

La FPU (Floating-Point Unit):

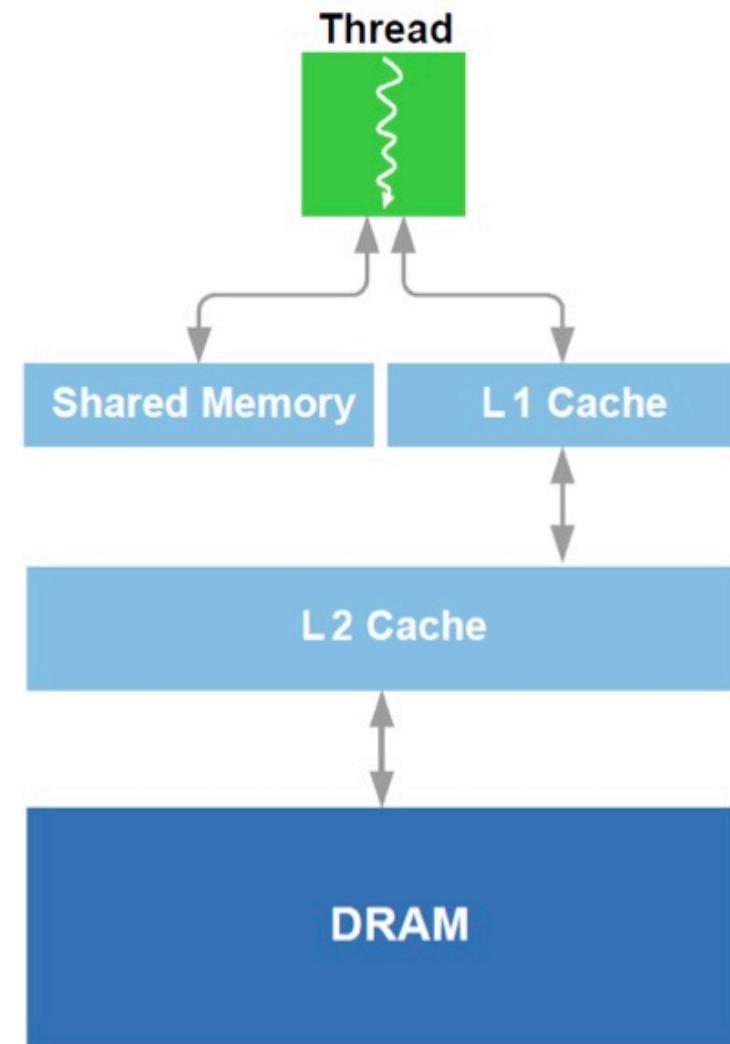
- Implementa el formato IEEE-754 en su versión de 2008, aventajando incluso a las CPUs más avezadas.



La jerarquía de memoria

■ Fermi es la primera GPU que ofrece una caché L1, que combina con la memoria compartida en proporción 3:1 o 1:3 para un total de 64 Kbytes por cada multiprocesador SM.

■ También incluye una L2 de 768 Kbytes que comparten todos los multiprocesadores.



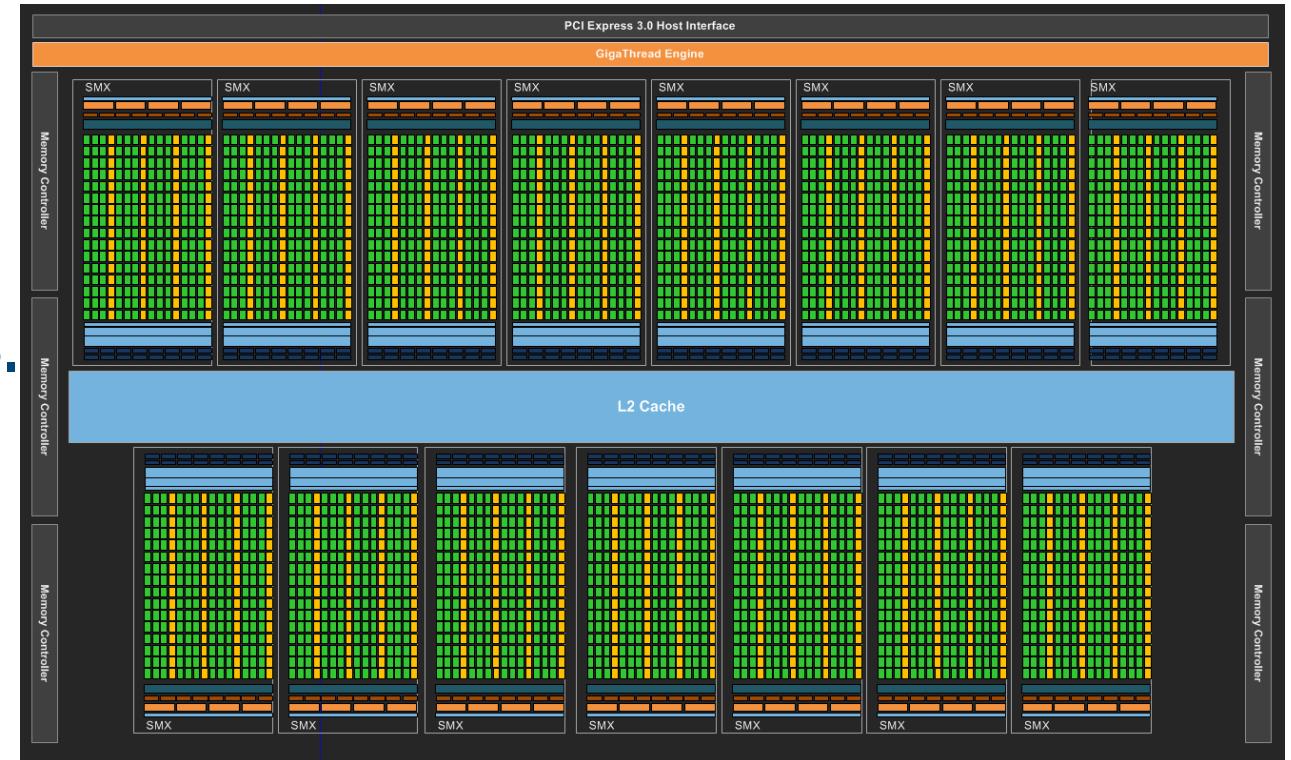


II. 4. La tercera generación: Kepler (GKxxx)



Diagrama de bloques: Kepler GK110

- 7.100 Mt.
- 15 multiprocs. SMX.
- > 1 TFLOP FP64.
- Caché L2 de 1.5 MB.
- GDDR5 de 384 bits.
- PCI Express Gen3.

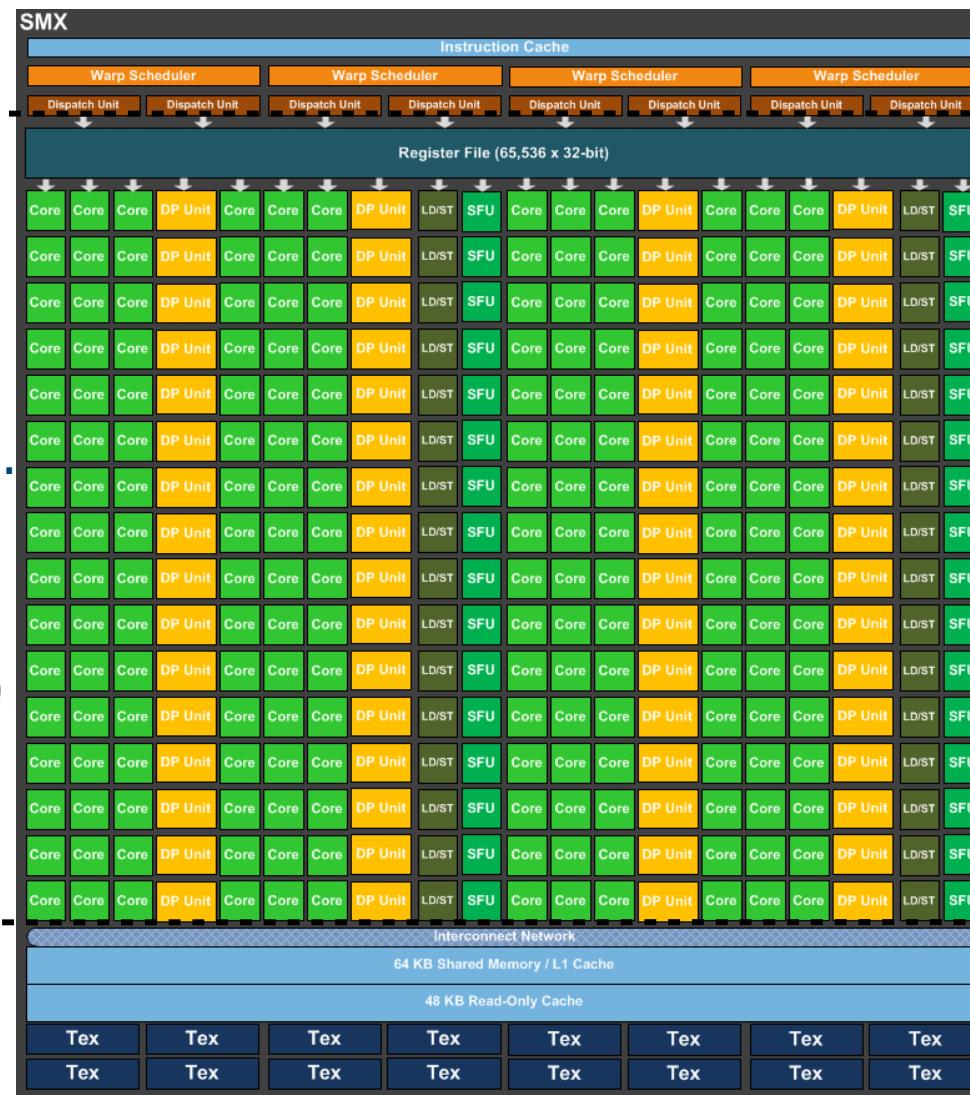


El multiprocesador SMX

Planificación y emisión de instrucciones en warps

Ejecución de instrucciones.
512 unidades funcionales:

- 192 para aritmética entera.
- 192 para aritmética s.p.
- 64 para aritmética d.p.
- 32 para carga/almacen.
- 32 para SFUs (log,sqrt, ...)

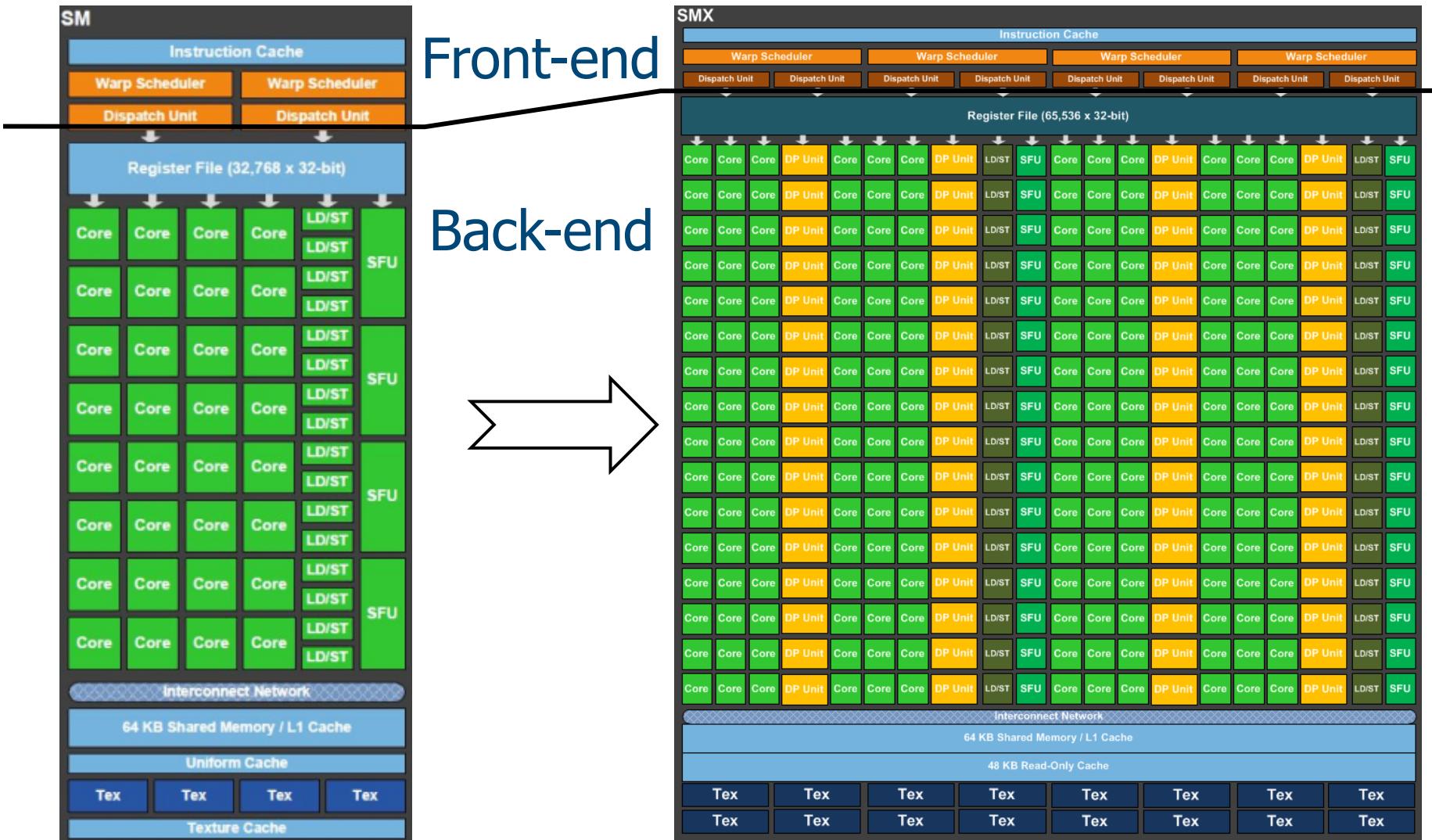


Front-end

Back-end

Interfaz

Del multiprocesador SM de Fermi GF100 al multiprocesador SMX de Kepler GK110



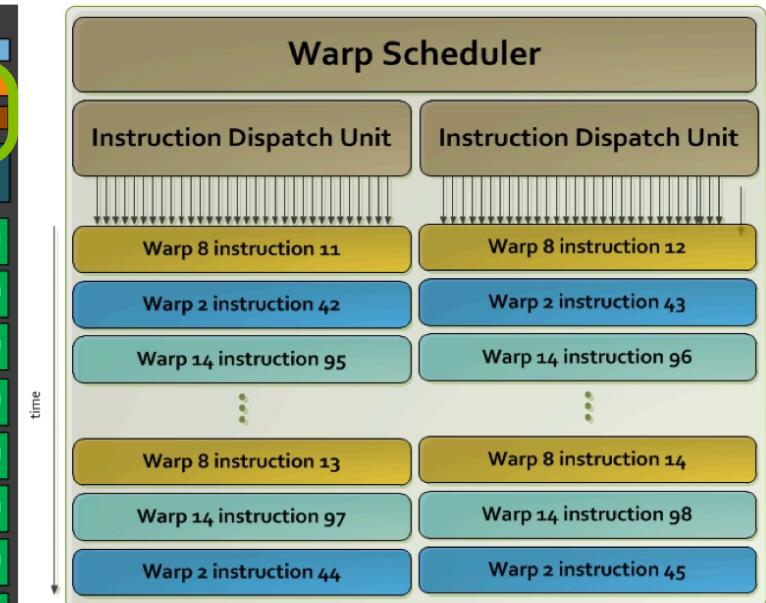
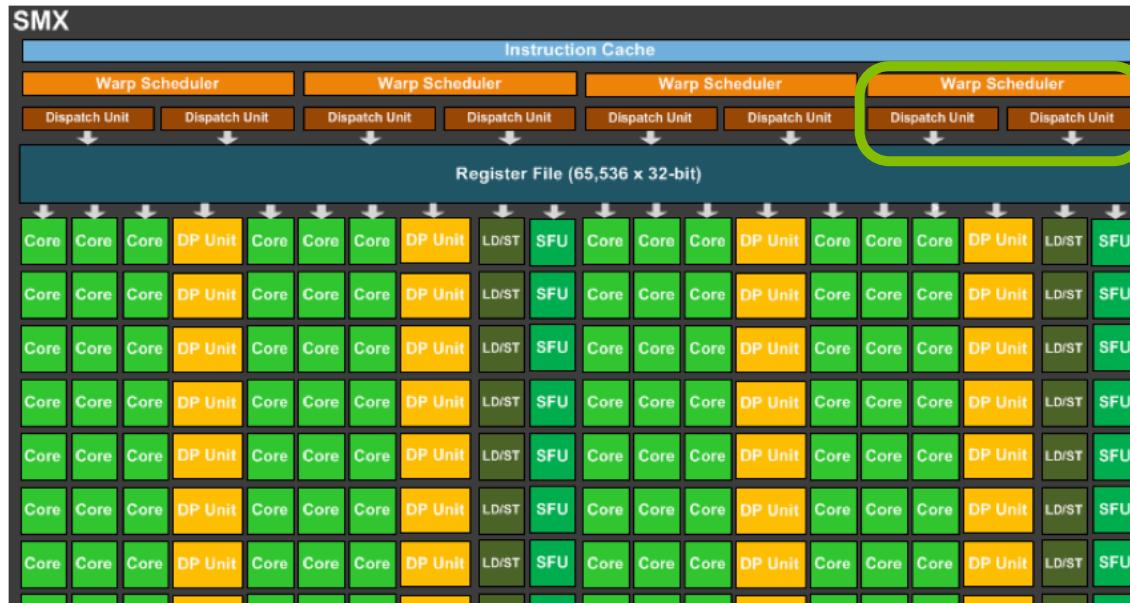
Comparación entre la emisión y ejecución de instrucciones (front-end vs. back-end)

	Búsqueda y emisión (front-end)	Ejecución en SM-SMX (back-end)
Fermi (GF100)	<p>Puede emitir 2 warps, 1 instr. cada uno.</p> <p>Total: Máx. 2 warps por ciclo.</p> <p>Warps activos: 48 en cada SM, seleccionados de entre 8 bloques máx.</p> <p>En GTX580: $16 * 48 = 768$ warps activos.</p>	<p>32 cores (1 warp) para "int" y "float".</p> <p>16 cores para "double" (1/2 warp).</p> <p>16 unids. de carga/almacen. (1/2 warp).</p> <p>4 unids. de funcs. especiales (1/8 warp).</p> <p>Total: Hasta 5 warps concurrentes.</p>
Kepler (GK110)	<p>Puede emitir 4 warps, 2 instrs. cada uno.</p> <p>Total: Máx. 8 warps por ciclo.</p> <p>Warps activos: 64 en cada SMX, seleccionados de entre 16 bloques máx.</p> <p>En K40: $15 * 64 = 960$ warps activos.</p>	<p>192 cores (6 warps) para "int" y "float".</p> <p>64 cores para "double" (2 warps).</p> <p>32 unids. de carga/almacen. (1 warp).</p> <p>32 unids. de funcs. especiales (1 warp).</p> <p>Total: Hasta 16 warps concurrentes.</p>

- En Kepler, cada SMX puede emitir 8 warp-instrucciones por ciclo, pero debido a limitaciones por recursos y dependencias:
 - 7 es el pico sostenible.
 - 4-5 es una buena cantidad para códigos limitados por instrucción.
 - <4 en códigos limitados por memoria o latencia.

Gigathread, o cómo el programa es devorado por el procesador

- Cada malla tiene un número de bloques, que son asignados a los multip. (hasta 32 en Maxwell, 16 en Kepler, 8 en Fermi).
- Los bloques se dividen en warps o grupos de 32 hilos.
- Los warps se ejecutan para cada instrucción de los hilos (hasta 64 warp-instrucción activos en Kepler). Ejemplo:



Mejoras en concurrencia y paralelismo

Generación de GPU	Fermi		Kepler	
Modelo hardware	GF100	GF104	GK104	GK110
CUDA Compute Capability (CCC)	2.0	2.1	3.0	3.5
Número de hilos / warp (tamaño del warp)	32	32	32	32
Máximo número de warps / Multiprocesador	48	48	64	64
Máximo número de bloques / Multiprocesador	8	8	16	16
Máximo número de hilos / Bloque	1024	1024	1024	1024
Máximo número de hilos / Multiprocesador	1536	1536	2048	2048

Mejoras cruciales
para ocultar latencias

Máx. concurrencia
en cada SMX

Caso estudio para explotar la concurrencia de la GPU en Fermi (15 SMs) y Kepler (15 SMXs)

- mykernel <<< 100, 128, ... >>> [Aquí tenemos un déficit en warps]
 - Lanza 100 bloques de 128 hilos (4 warps), esto es, 400 warps.
 - Hay 26.66 warps para cada multiprocesador, ya sea SM o SMX.
 - En Fermi: Hasta 48 warps activos (21 bajo el límite), que no puede aprovecharse.
 - En Kepler: Hasta 64 warps activos (37 bajo el límite), que pueden activarse desde hasta un máx. de 32 llamadas a kernels desde: MPI, threads POSIX, streams CUDA.
- mykernel <<< 100, 384, ... >>>
 - Lanza 100 bloques de 384 hilos (12 warps), esto es, 1200 warps.
 - Hay 80 warps para cada multiprocesador. Hemos alcanzado el máx. de 64 warps activos, así que 16 warps * 15 SMX = 240 warps esperan en colas de Kepler para ser activados con posterioridad.
- mykernel <<< 1000, 32, ... >>> [Aquí tenemos un exceso de bloques]
 - 66.66 bloques para cada SMX, pero el máx. es 16. Mejor <100, 320>

Lecciones a aprender (y conflictos asociados)

- Bloques suficientemente grandes para evitar el límite de 16 por cada SMX.
 - Pero los bloques consumen memoria compartida, y alojar más memoria compartida significa menos bloques y más hilos por bloque.
- Suficientes hilos por bloque como para saturar el límite de 64 warps activos por cada SMX.
 - Pero los hilos consumen registros, y utilizar muchos registros nos lleva a tener menos hilos por bloque y más bloques.
- Sugerencias:
 - Al menos 3-4 bloques activos, cada uno con al menos 128 hilos.
 - Menos bloques cuando la memoria compartida es crítica, pero...
 - ... abusar de ella penaliza la concurrencia y la ocultación de latencia.

Mejora de recursos en los SMX

Recurso	Kepler GK110 frente a Fermi GF100
Ritmo de cálculo con opers. punto flotante	2-3x
Máximo número de bloques por SMX	2x
Máximo número de hilos por SMX	1.3x
Ancho de banda del banco de registros	2x
Capacidad del banco de registros	2x
Ancho de banda de la memoria compartida	2x
Capacidad de la memoria compartida	1x
Ancho de banda de la caché L2	2x
Capacidad de la caché L2	2x

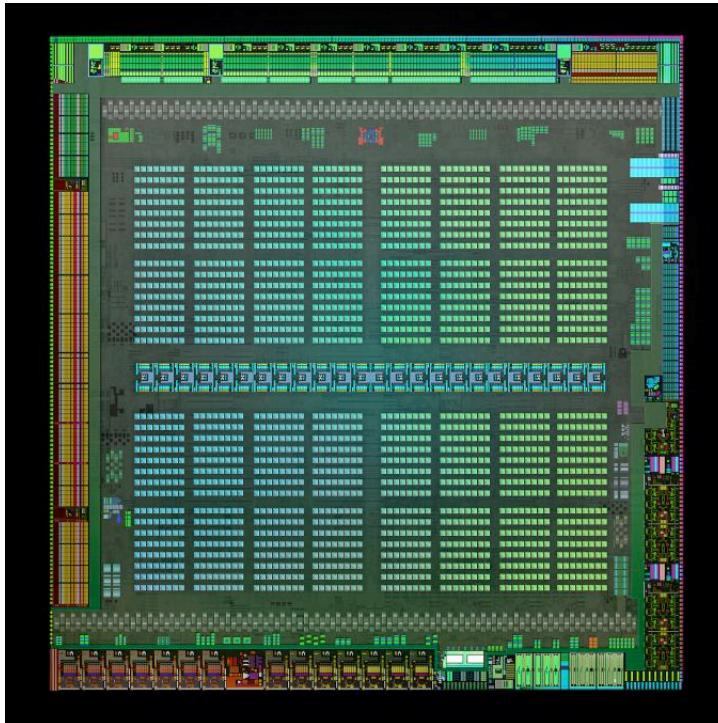


II. 5. La cuarta generación: Maxwell (GMxxx)



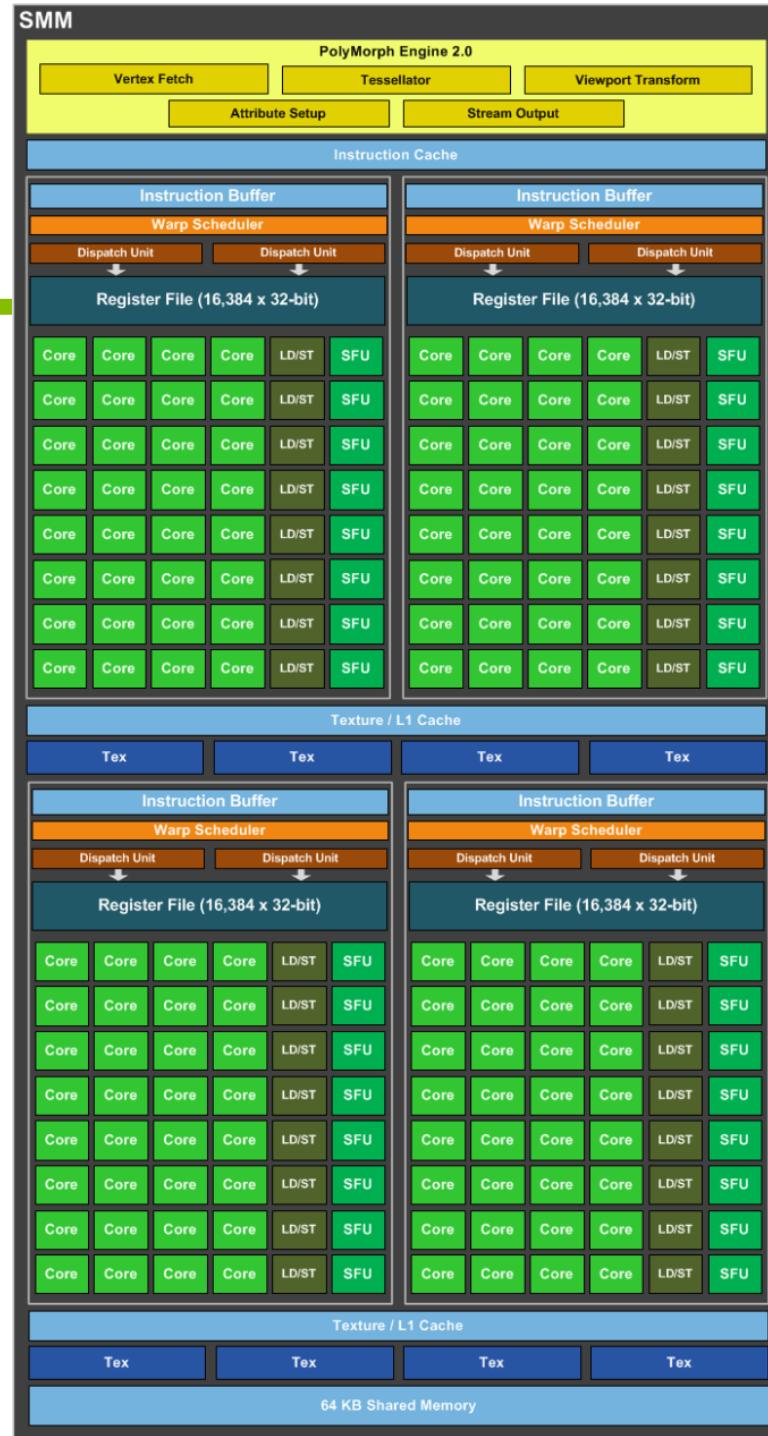
Maxwell y sus SMMs (para GeForce GTX 980, el modelo de 16 multiprocesadores)

- 1870 Mt.
- 148 mm².

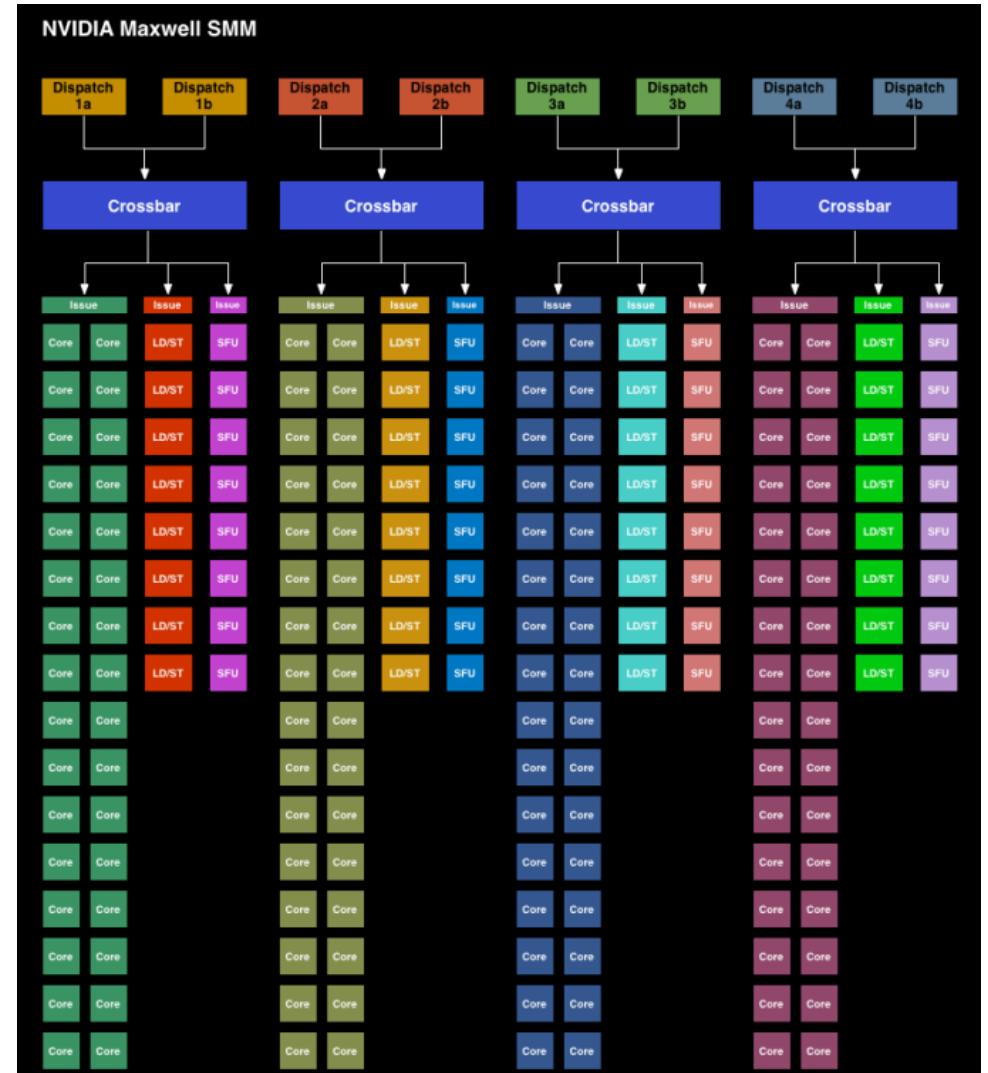
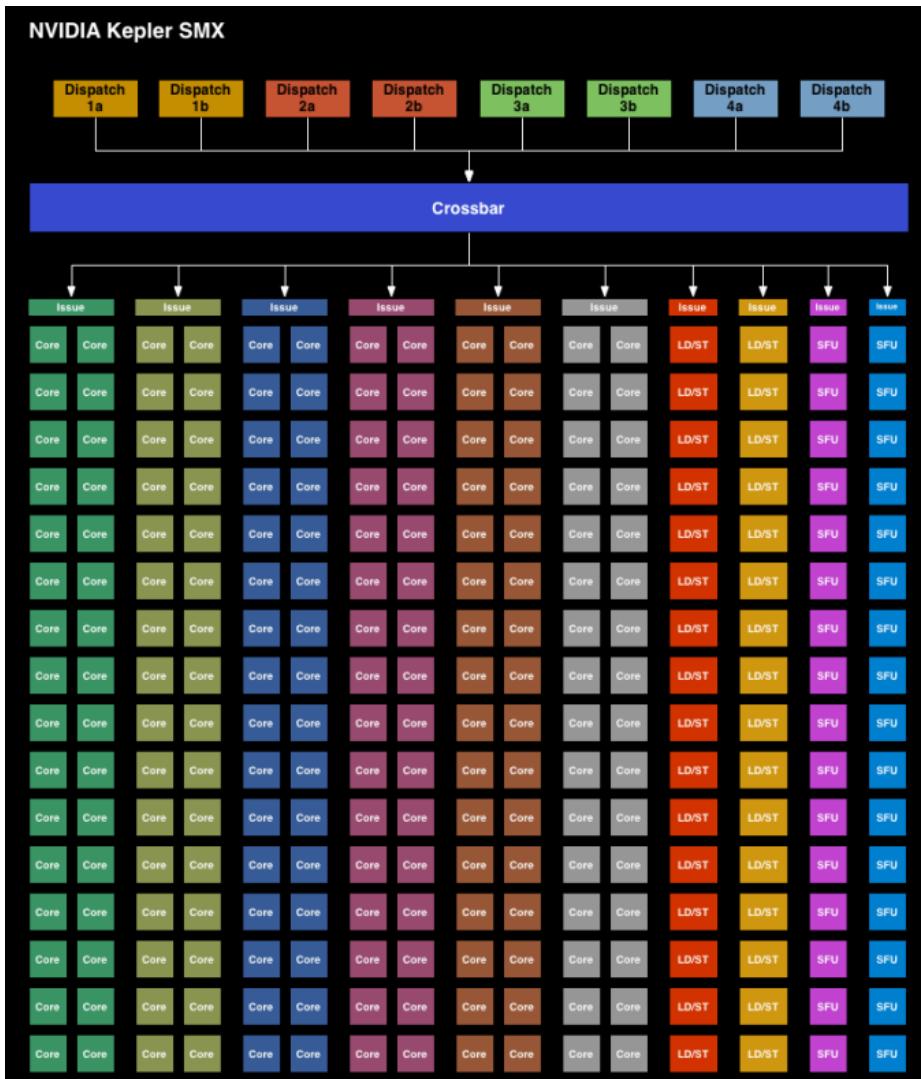


Detalle de los SMMs

- Mantiene los mismos 4 planificadores de warps.
- Mantiene las mismas LD/ST y SFUs.
- Reduce el nº de cores para int y float: De 192 a 128.



Comparativa respecto a Kepler



Algunos modelos comerciales para CCC 5.0 y comparativa con Kepler (en 28 nm.)

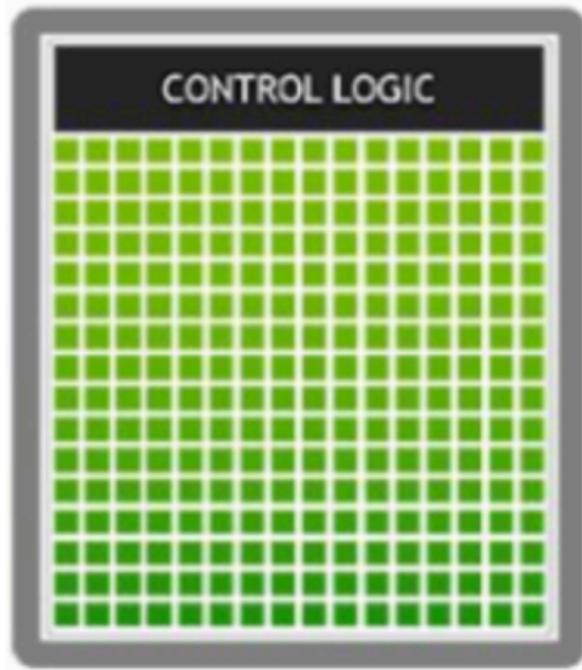
	GeForce GTX 650	GeForce GTX 650 Ti	GeForce GTX 750	GeForce GTX 660	GeForce GTX 750 Ti
GPU (code name)	GK107	GK106	GM107	GK106	GM107
Arquitectura	Kepler	Kepler	Maxwell	Kepler	Maxwell
Multiprocesadores	2 SMX	4 SMX	4 SMM	5 SMX	5 SMM
Número de cores	192 x 2 = 384	192 x 4 = 768	128 x 4 = 512	192 x 5 = 960	128 x 5 = 640
Frecuencia cores	1058 MHz	925 MHz	1020- 1085 MHz	980-1033 MHz	1020- 1085 MHz
Anchura bus DRAM	128 bits	128 bits	128 bits	192 bits	128 bits
Frecuencia DRAM	2x 2500 MHz	2x 2700 MHz	2x 2500 MHz	2x 3000 MHz	2x 2700 MHz
Ancho banda RAM	80 GB/sg.	86.4 GB/sg.	80.2 GB/sg.	144 GB/sg.	86.4 GB/sg.
Tamaño GDDR5	1 ó 2 GB	1 ó 2 GB	1 GB	2 GB	1 ó 2 GB
Conector potencia	6 pines	6 pines	Ninguno	6 pines	Ninguno
Máx. TDP	64 W.	110 W.	55 W.	140 W.	60 W.
Millones de transis.	1300	2540	1870	2540	1870
Área de integración	118 mm ²	214 mm ²	148 mm ²	214 mm ²	148 mm ²
Coste aproximado	100 € [2 GB]	110 € [2 GB]	80 € [1 GB]	150 € [2 GB]	110 € [2 GB]

Algunos modelos comerciales para CCC 5.2 (todos sobre 28 nm)

GeForce	GTX 950	GTX 960	GTX 970	GTX980	GTX 980 Ti	Titan X
Fecha de lanzamiento	Ago'15	Ago'15	Sept'14	Sept'14	Junio'15	Marzo'15
GPU (code name)	GM206-250	GM206-300	GM204-200	GM204-400	GM200-310	GM200-400
Multiprocesadores	6	8	13	16	22	24
Número de cores	768	1024	1664	2048	2816	3072
Frec. cores (MHz)	1024-1188	1127-1178	1050-1178	1126-1216	1000-1075	1000-1075
Anchura bus DRAM	128 bits	128 bits	256 bits	256 bits	384 bits	384 bits
Frecuencia DRAM	2x 3.3 GHz	2x 3.5 GHz				
Ancho de banda RAM	105.6 GB/s	112 GB/s	224 GB/s	224 GB/s	336.5 GB/s	336.5 GB/s
Tamaño GDDR5	2 GB	2 GB	4 GB	4 GB	6 GB	12 GB
Millones de transistores	2940	2940	5200	5200	8000	8000
Área de integración	228 mm ²	228 mm ²	398 mm ²	398 mm ²	601 mm ²	601 mm ²
Consumo (TDP)	90 W	120 W	145 W	165 W	250 W	250 W
Conecotor de potencia	1 x 6 pines	1 x 6 pines	2 x 6 pines	2 x 6 pines	6 + 8 pines	6 + 8 pines
Precio (\$ en estreno)	149	199	329	549	649	999

Principales mejoras

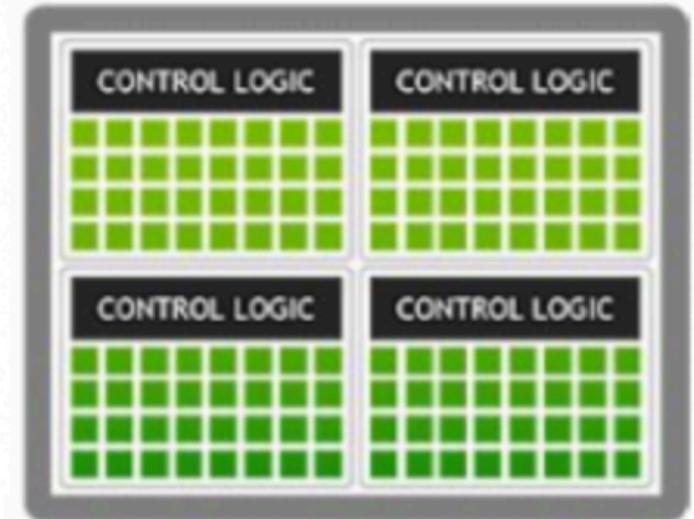
KEPLER



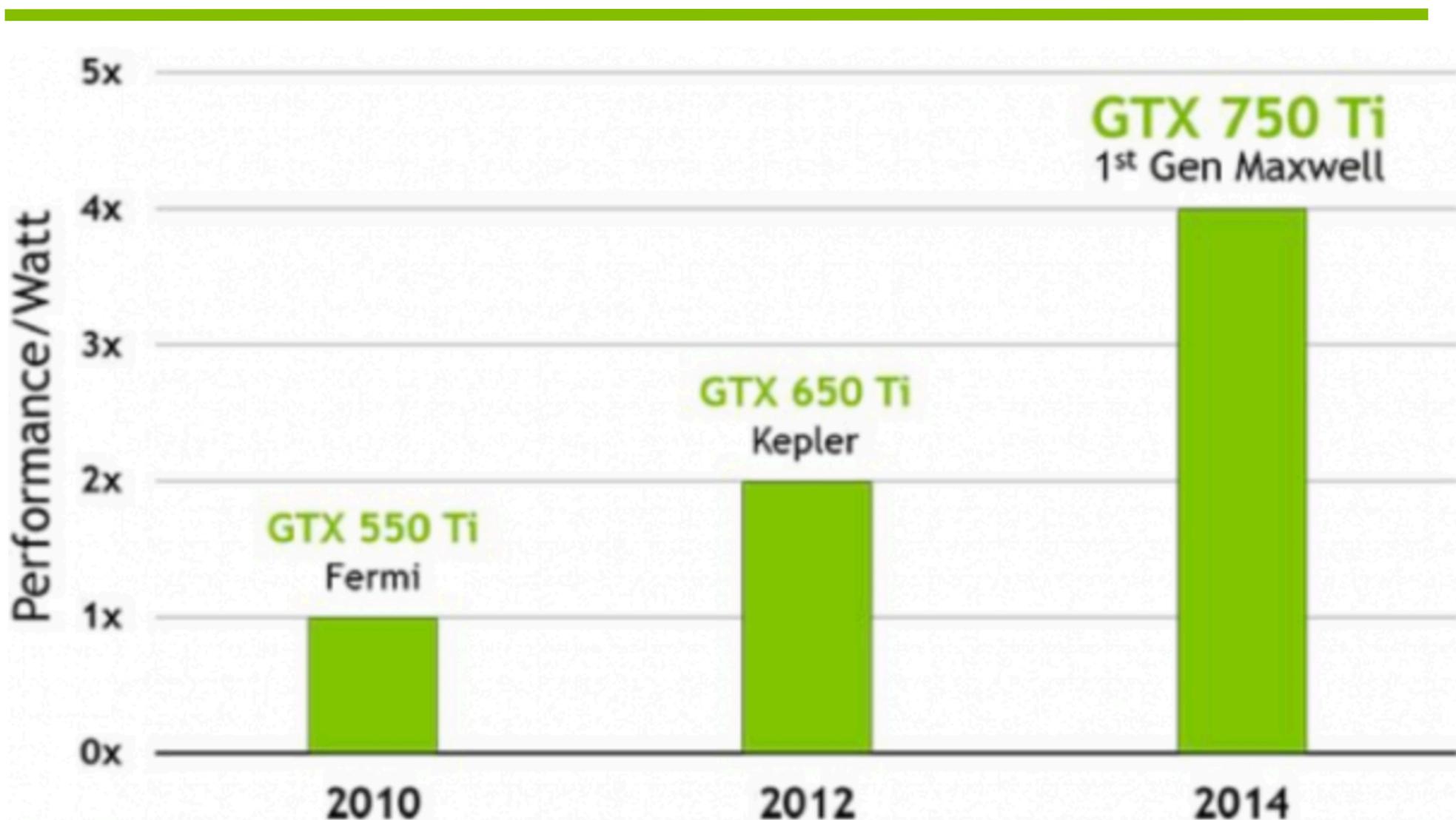
MAXWELL 1st Generation

135%
Performance/Core

2x
Performance/Watt



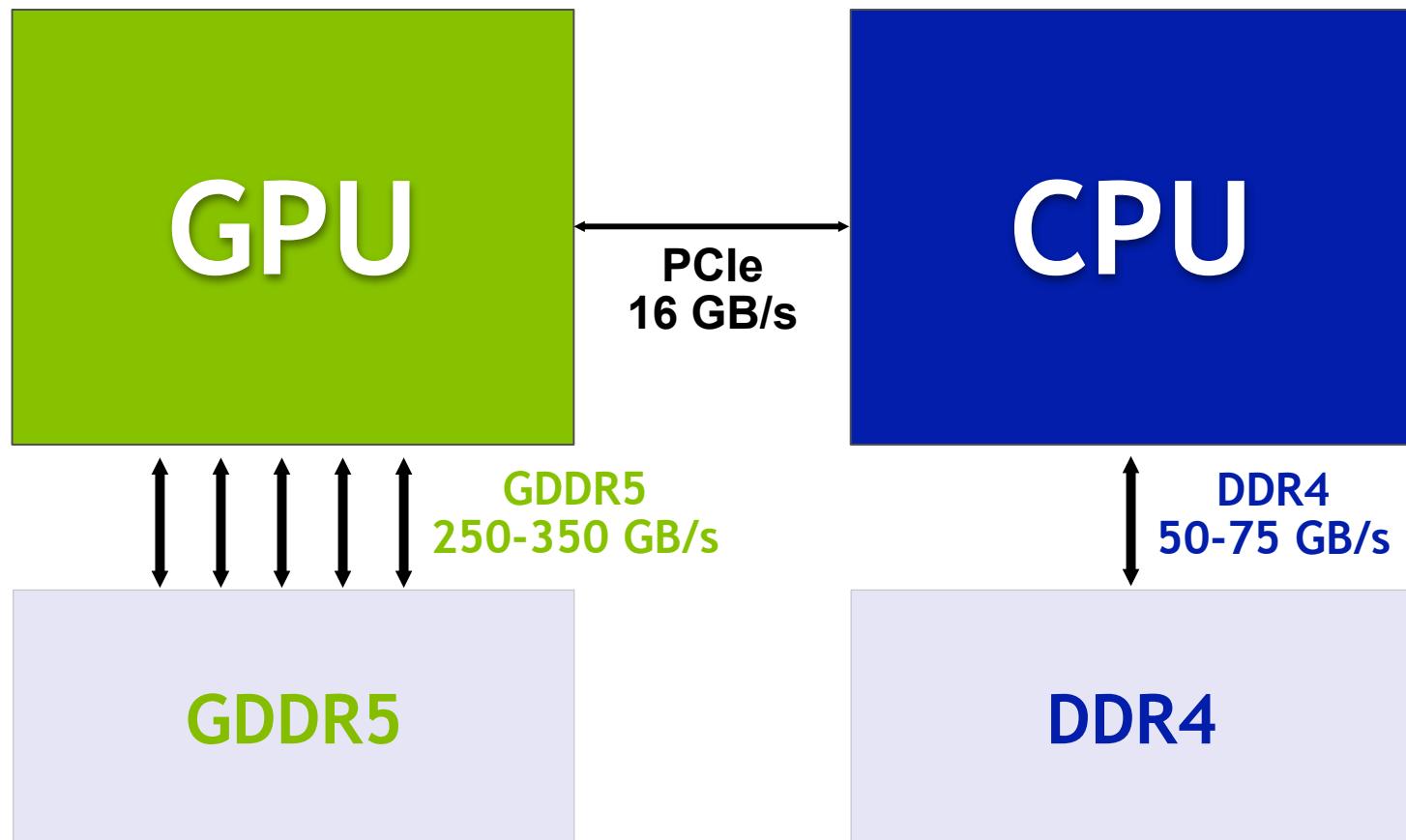
Eficiencia del consumo



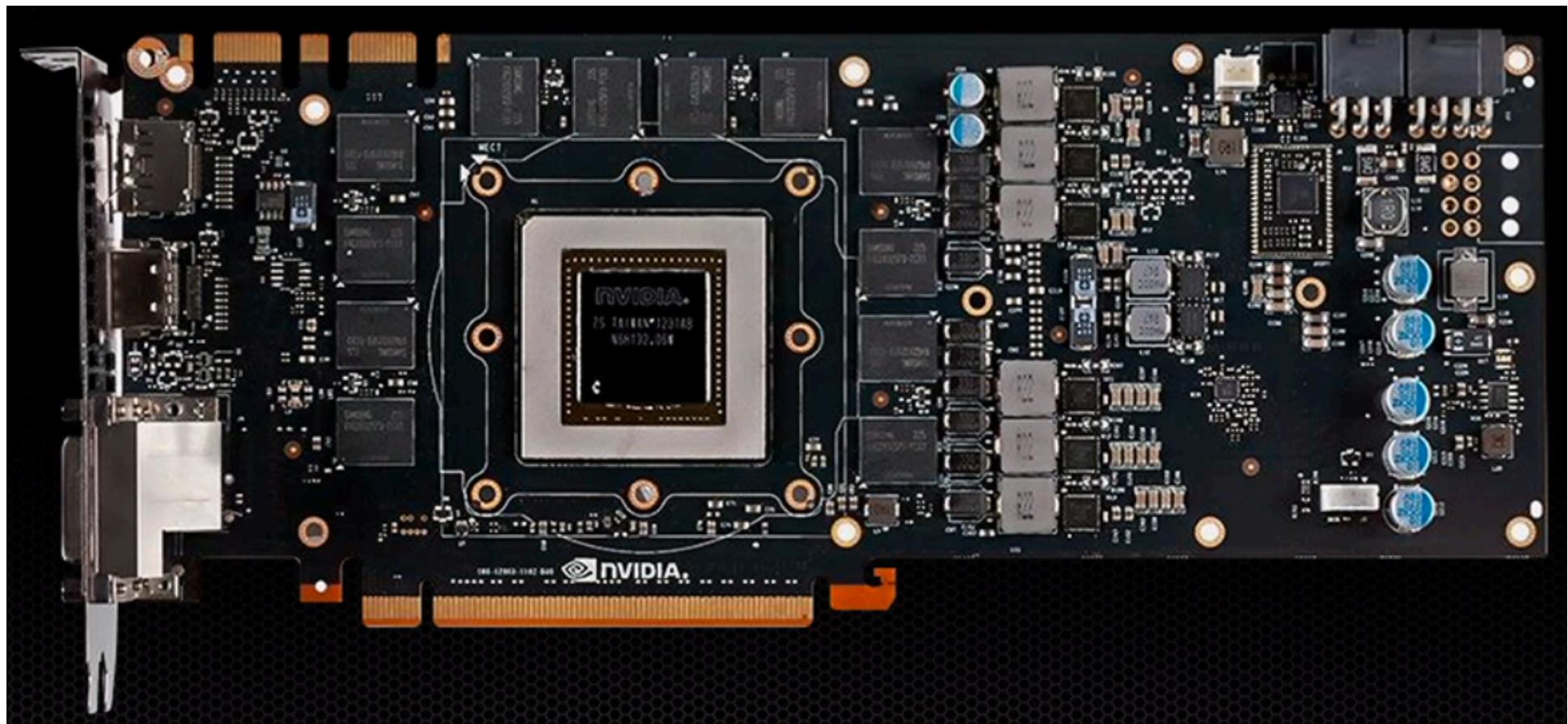


II. 6. La quinta generación: Pascal

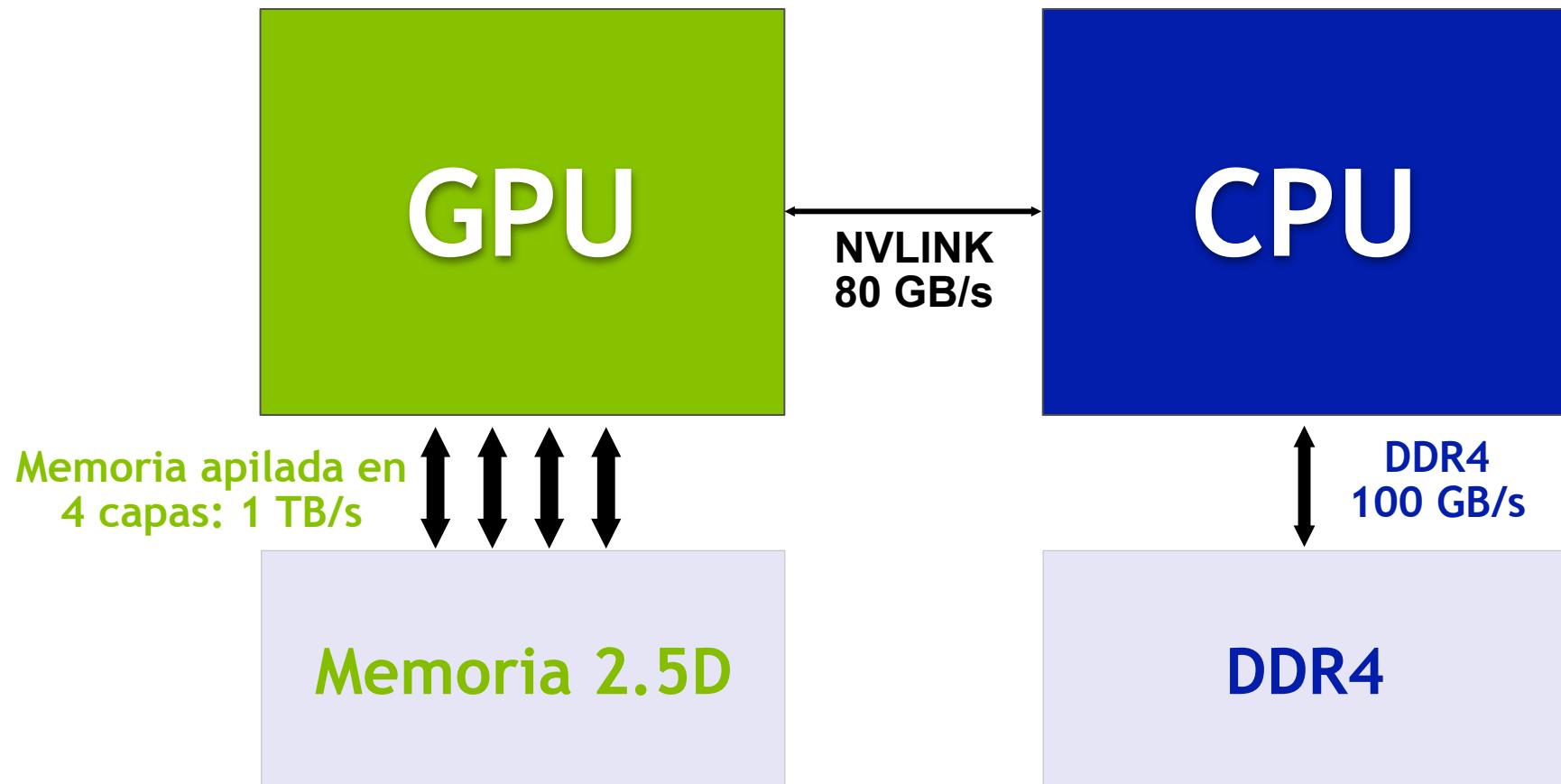
Hoy



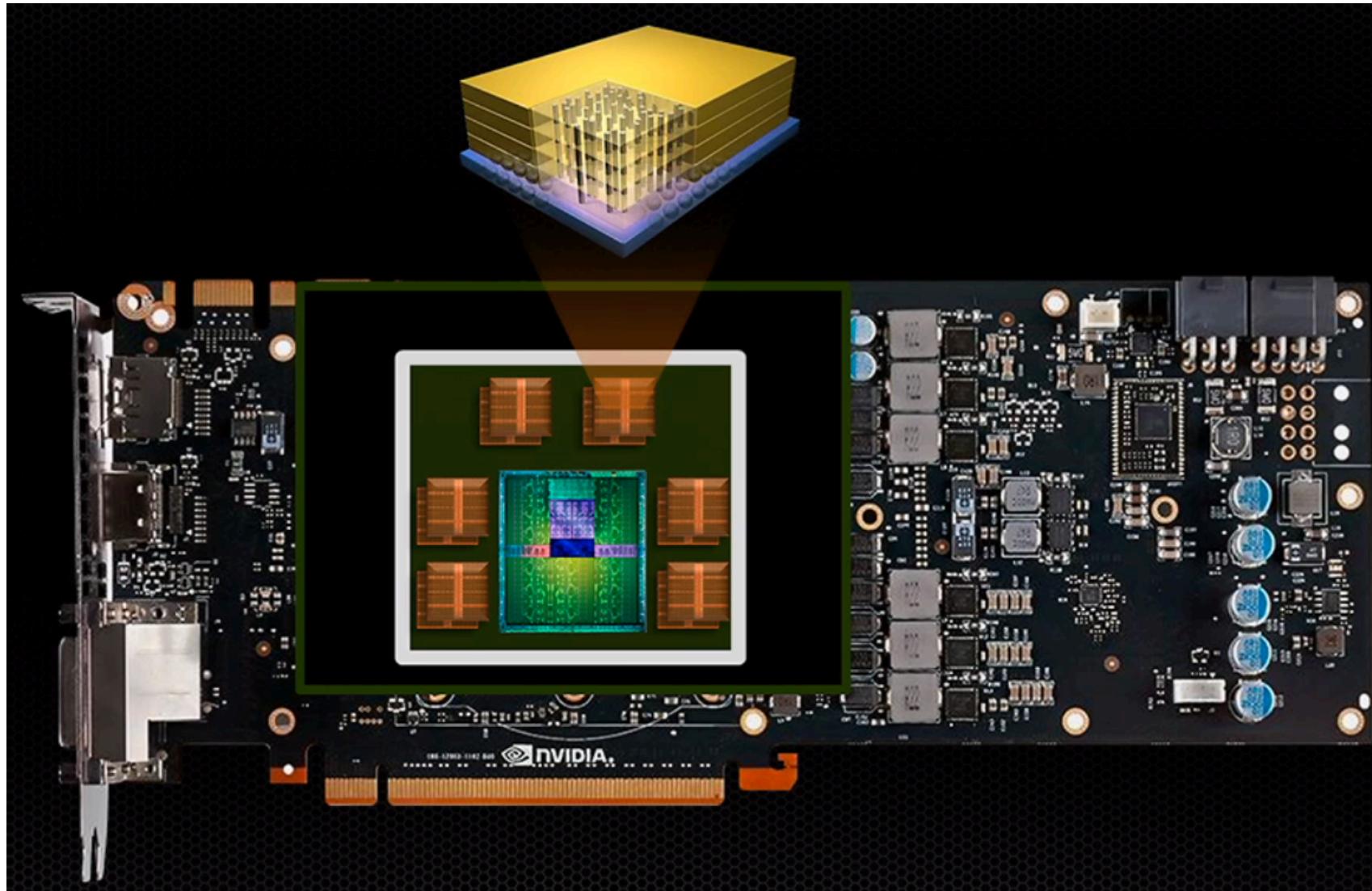
La tarjeta gráfica de 2014/15: GPU Kepler/Maxwell con memoria de GDDR5



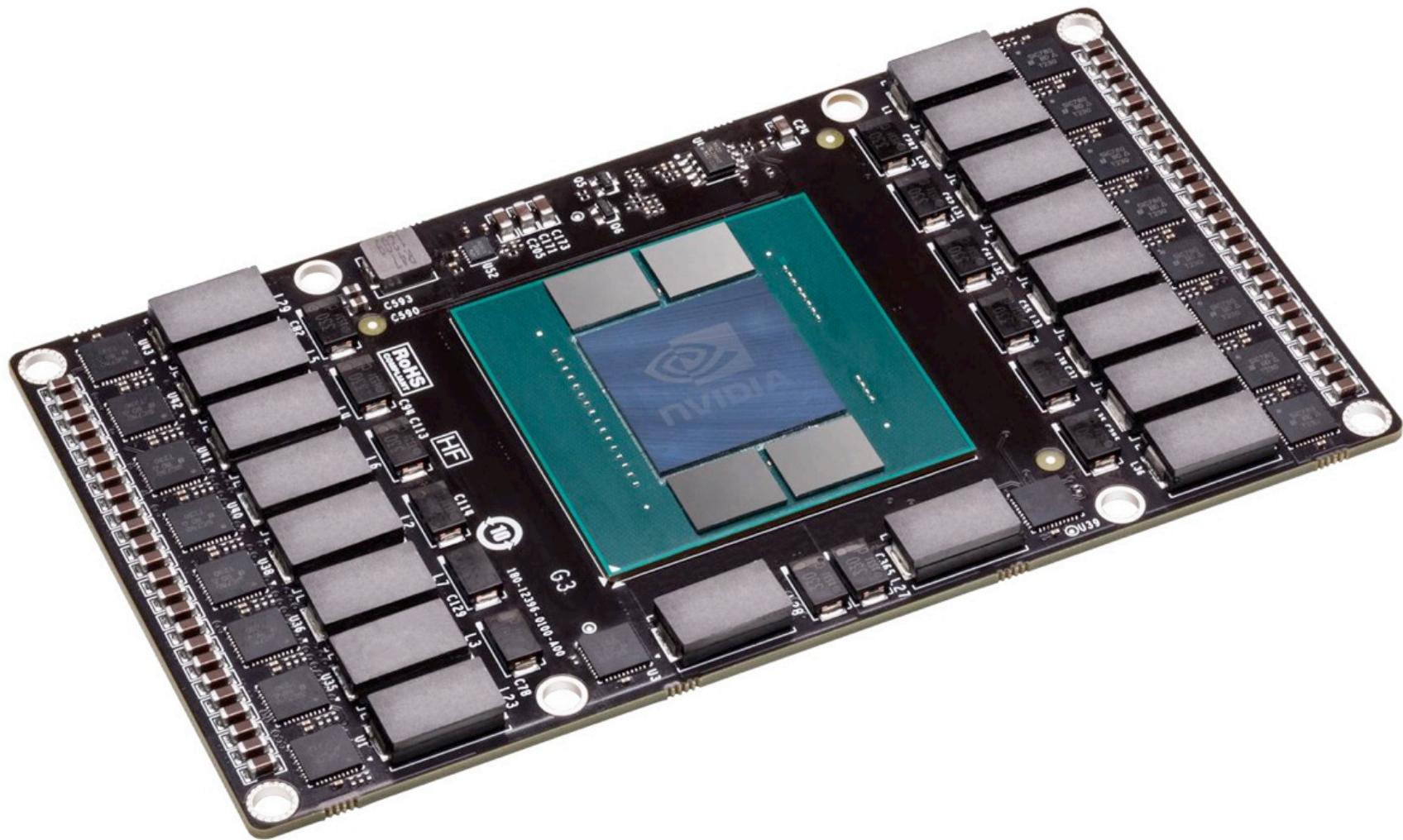
En 2016



La tarjeta gráfica de 2016: GPU Pascal con memoria Stacked (3D) DRAM



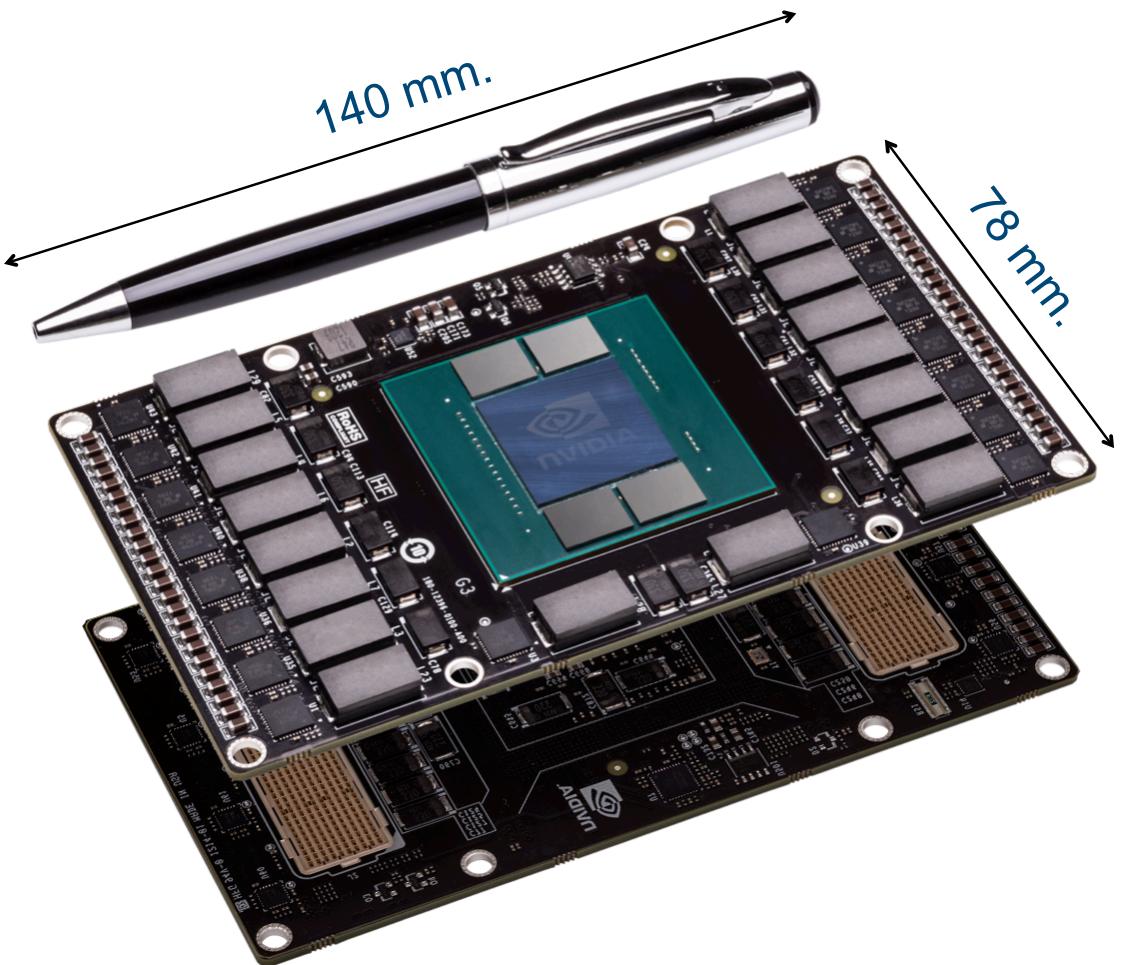
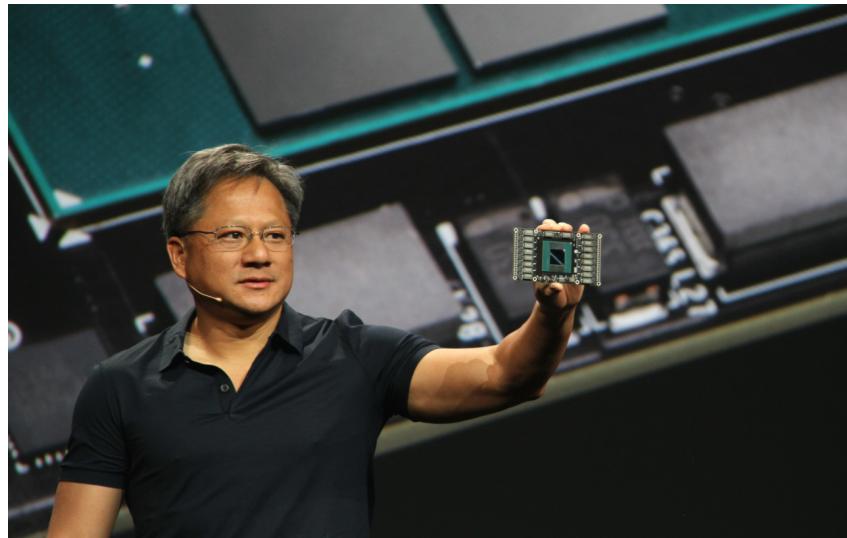
Prototipo de GPU Pascal



Pascal: Multiprocesadores SXM 2.0

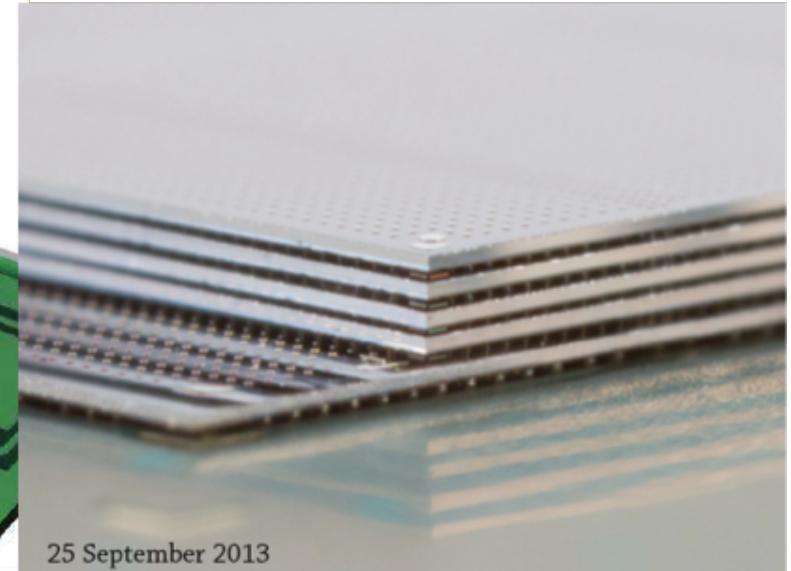
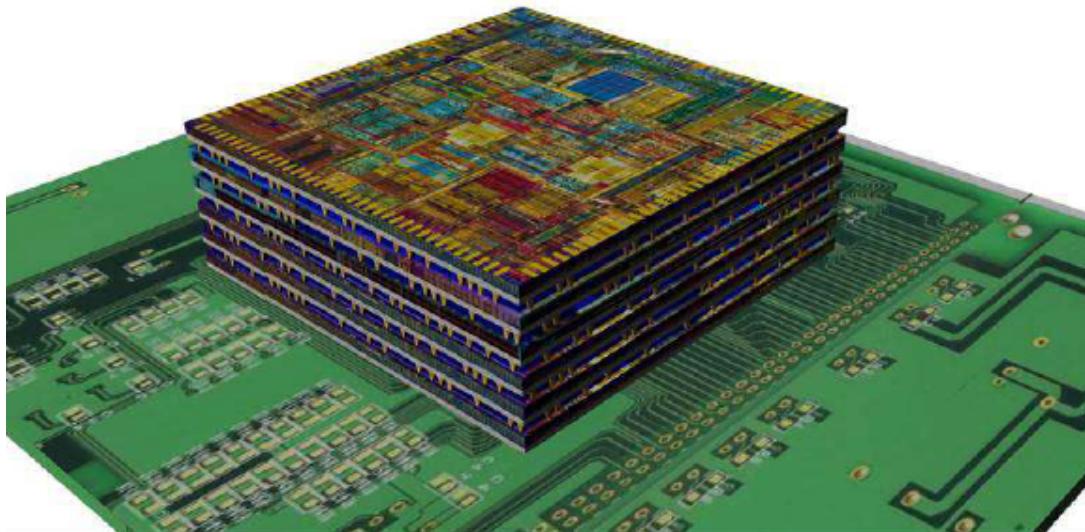


SMX 2.0*:
3 veces mejores
prestaciones en
densidad de circuitos

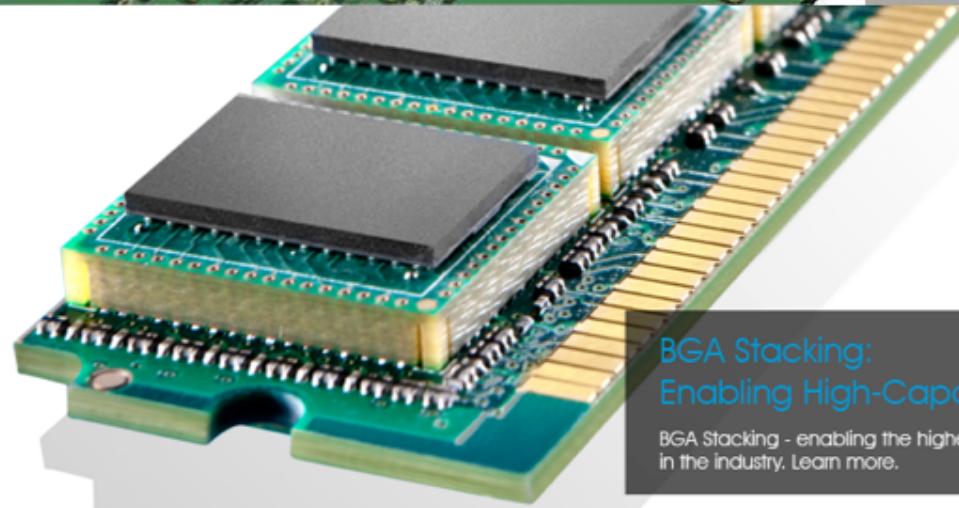


(* Nombre provisional para marketing).

Desarrollos comerciales: Prototipos de Micron (arriba) y productos de Viking (abajo)



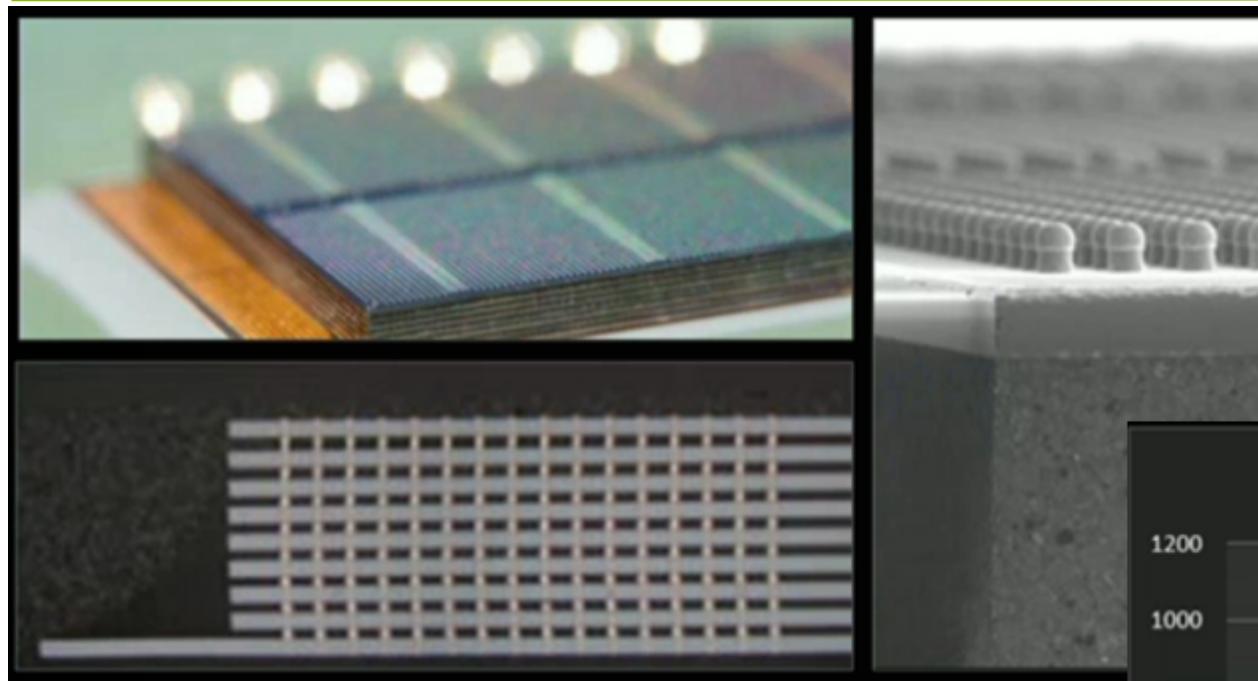
25 September 2013



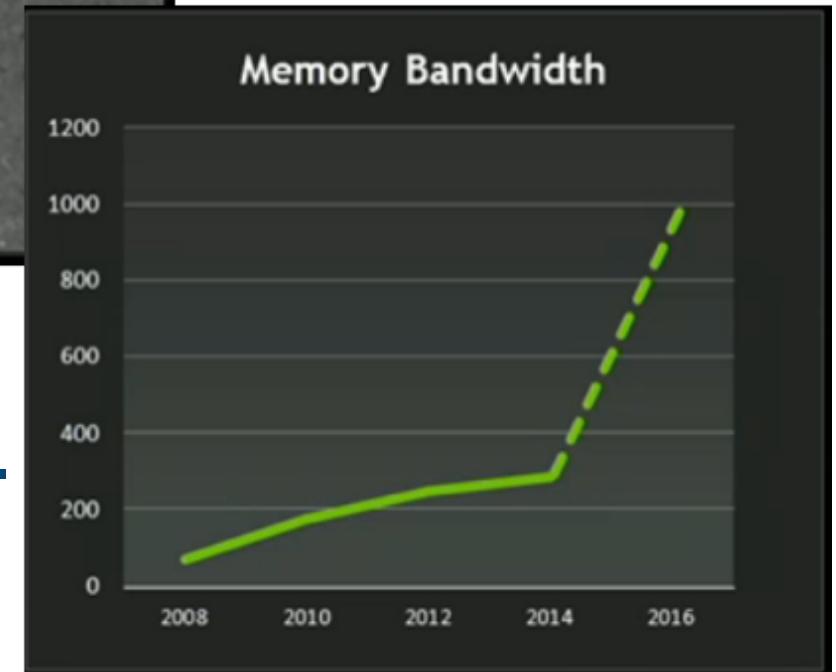
BGA Stacking:
Enabling High-Capacity DRAM Modules

BGA Stacking - enabling the highest capacity memory modules available in the industry. Learn more.

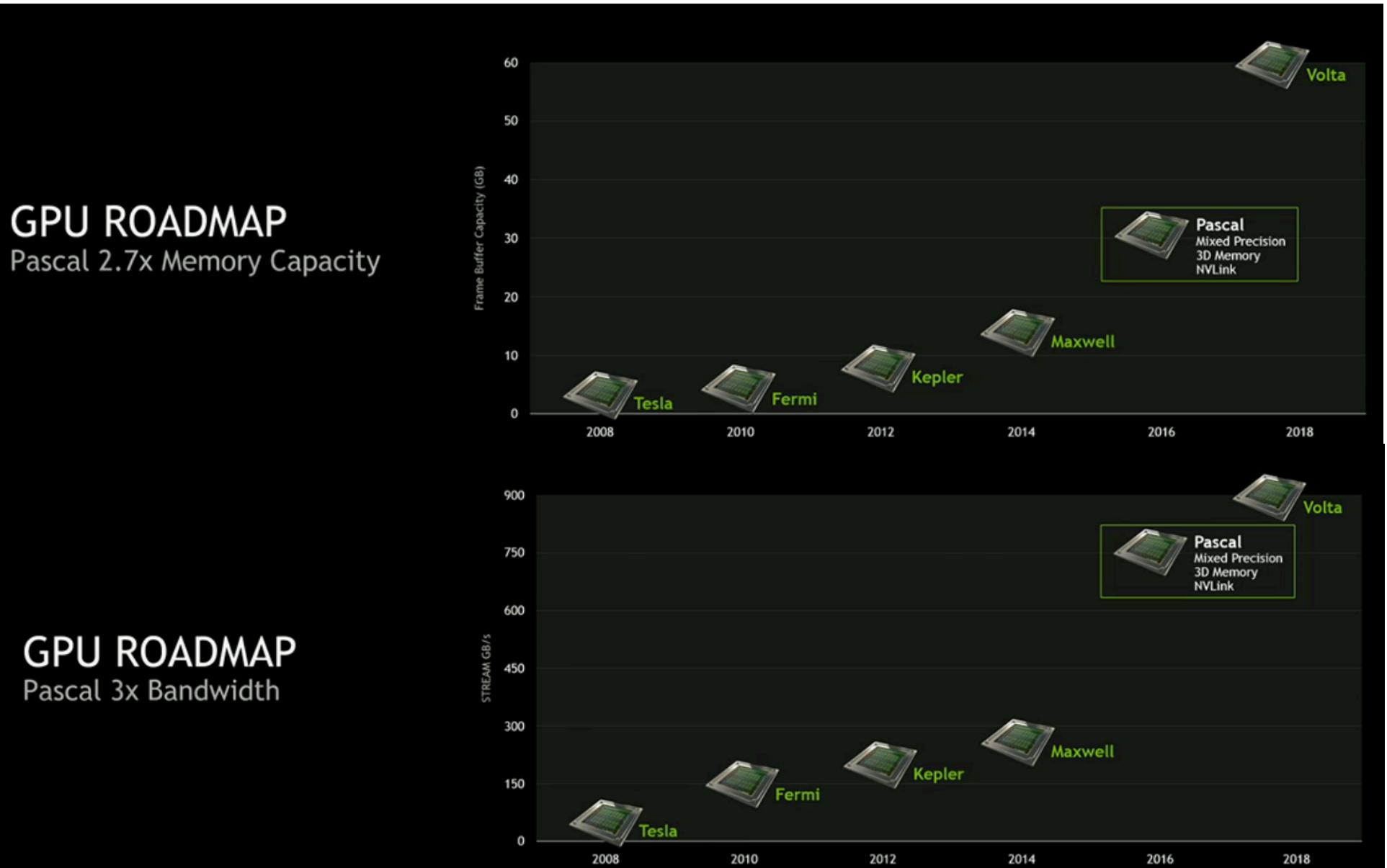
Memoria 3D de Pascal



- Integración chip-en-oblea 3D.
- 3x ancho de banda (vs. GDDR5).
- 2.5x capacidad.
- 4x eficiencia energética.

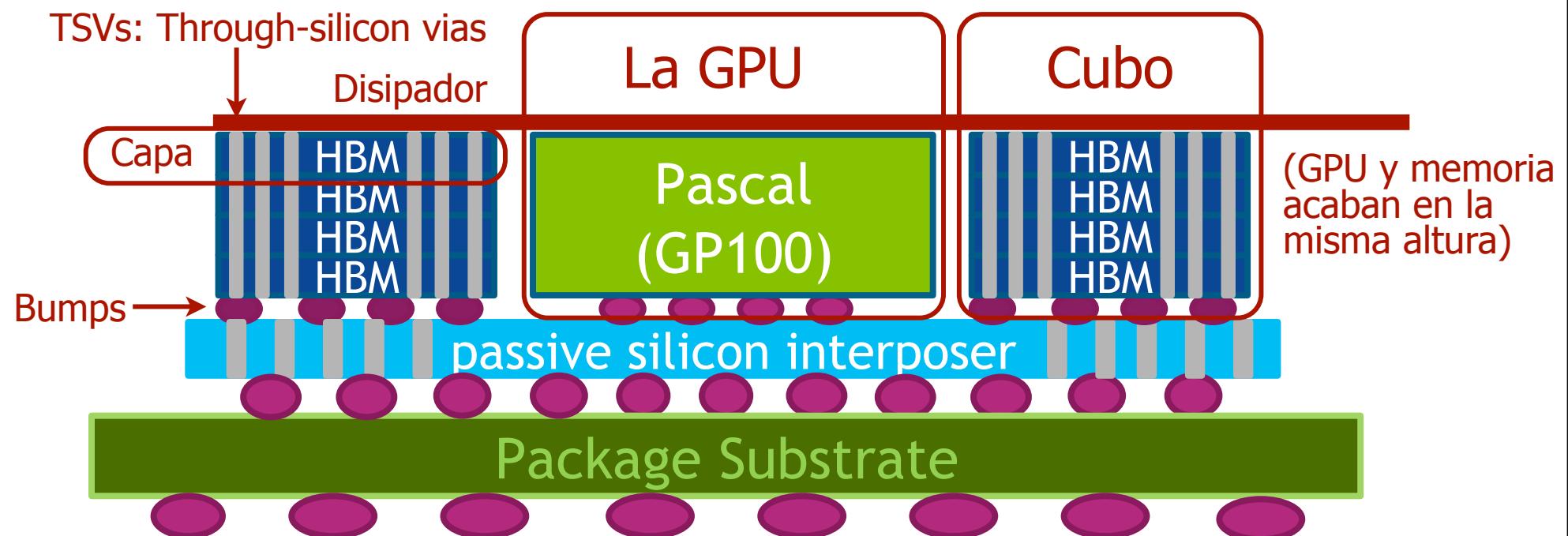


Futuro: Anuncios oficiales en el GTC'15



Cómo flanquear la barrera del TFLOP con un reloj de 2x 500 MHz

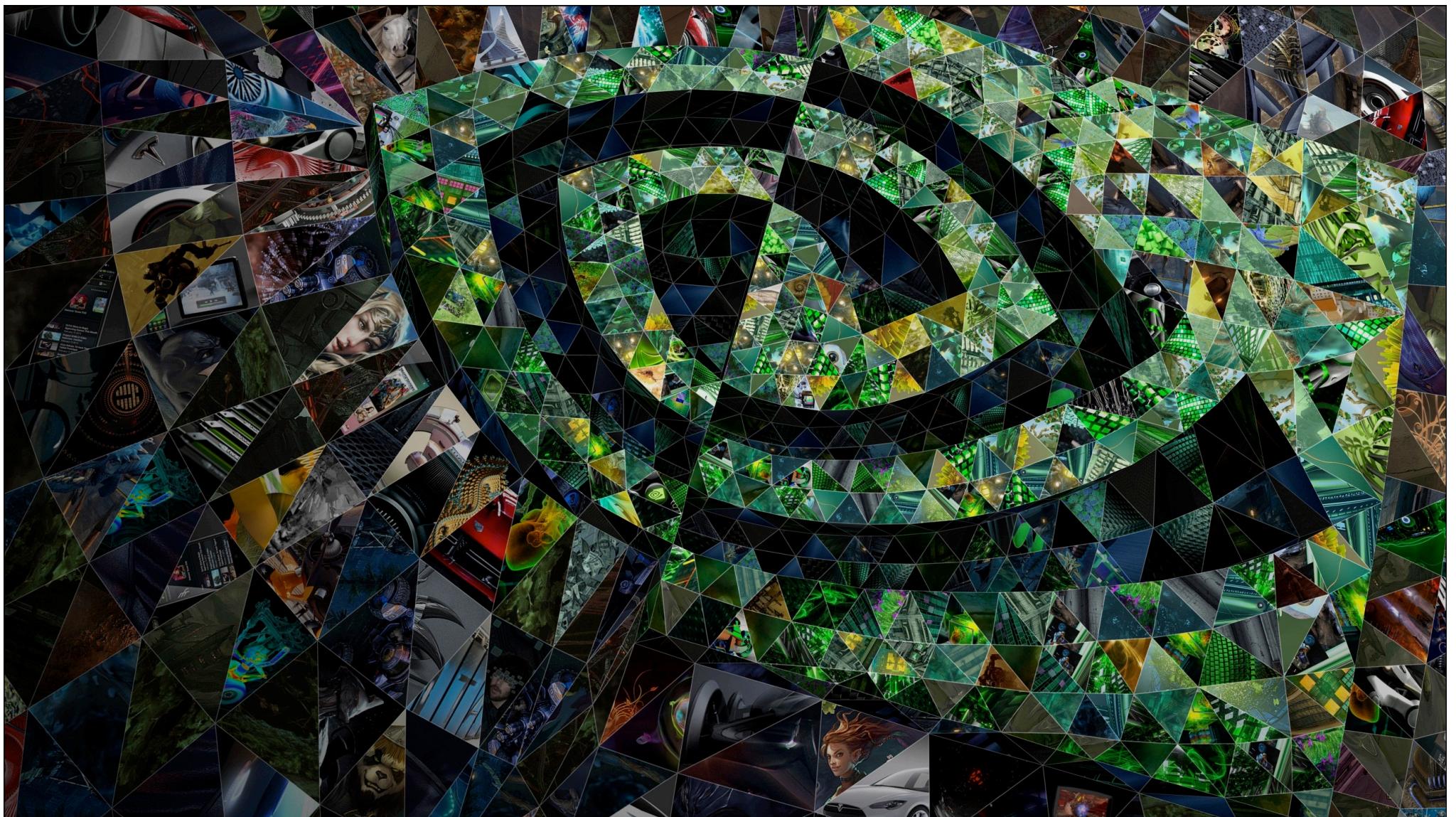
- BW = Frecuencia*anchura => $1 \text{ TB/s} = 2 \times 500 \text{ MHz} * \text{anchura}$
 $\Rightarrow \text{anchura} = 8000 \text{ Gbits/s} / 1 \text{ GHz} = 8000 \text{ bits}$
- Ancho de Titan X: 384 bits. Máx. histórico en GPU: 512 bits.



- ¡Hay una jerarquía de interconexión!

Para más información sobre DRAM apilada (Stacked DRAM o 3D DRAM)

- HMCC (Hybrid Memory Cube Consortium).
 - Mentores: Micron y Samsung.
 - <http://www.hybridmemorycube.org> (HMC 1.0, 1.1, 2.0 disponibles)
- HBM (High Bandwidth Memory).
 - Mentores: AMD and SK Hynix.
 - <https://www.jedec.org/standards-documents/docs/jesd235> (acceso via JEDEC).
- Echa un vistazo a lo que predicen los gurús a final de año (informe emitido por la ITRS a finales de 2015):
 - <http://www.itrs.net>



II. 7. Síntesis generacional

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

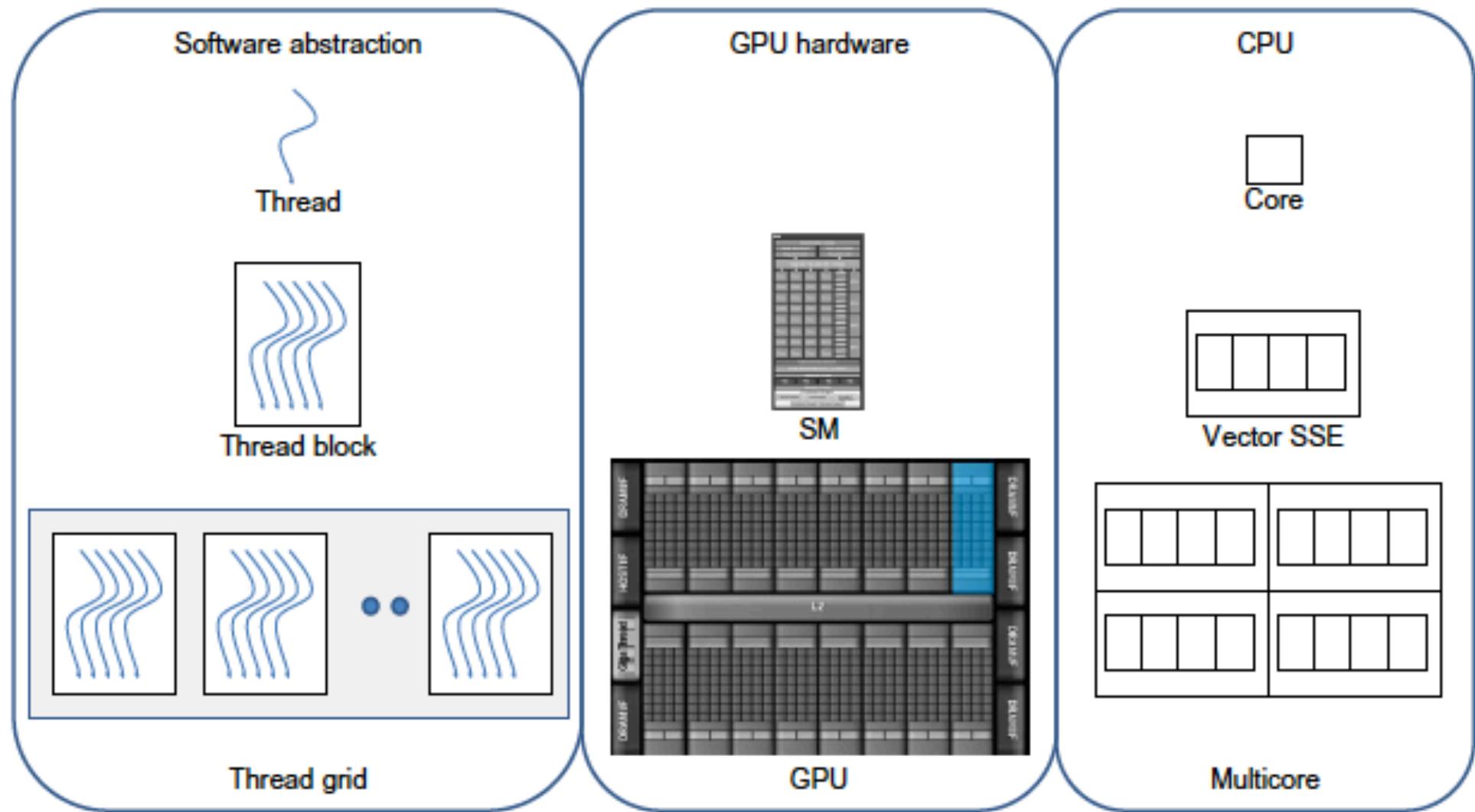
	Tesla		Fermi		Kepler				Maxwell	
Arquitectura	G80	GT200	GF100	GF104	GK104 (K10)	GK110 (K20X)	GK110 (K40)	GK210 (K80)	GM107 (GTX750)	GM204 (GTX980)
Marco temporal	2006 /07	2008 /09	2010	2011	2012	2013	2013 /14	2014	2014 /15	2014 /15
CUDA Compute Capability	1.0	1.3	2.0	2.1	3.0	3.5	3.5	3.7	5.0	5.2
N (multiprocs.)	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



III. Programación



Comparativa con la CPU



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 commutación 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

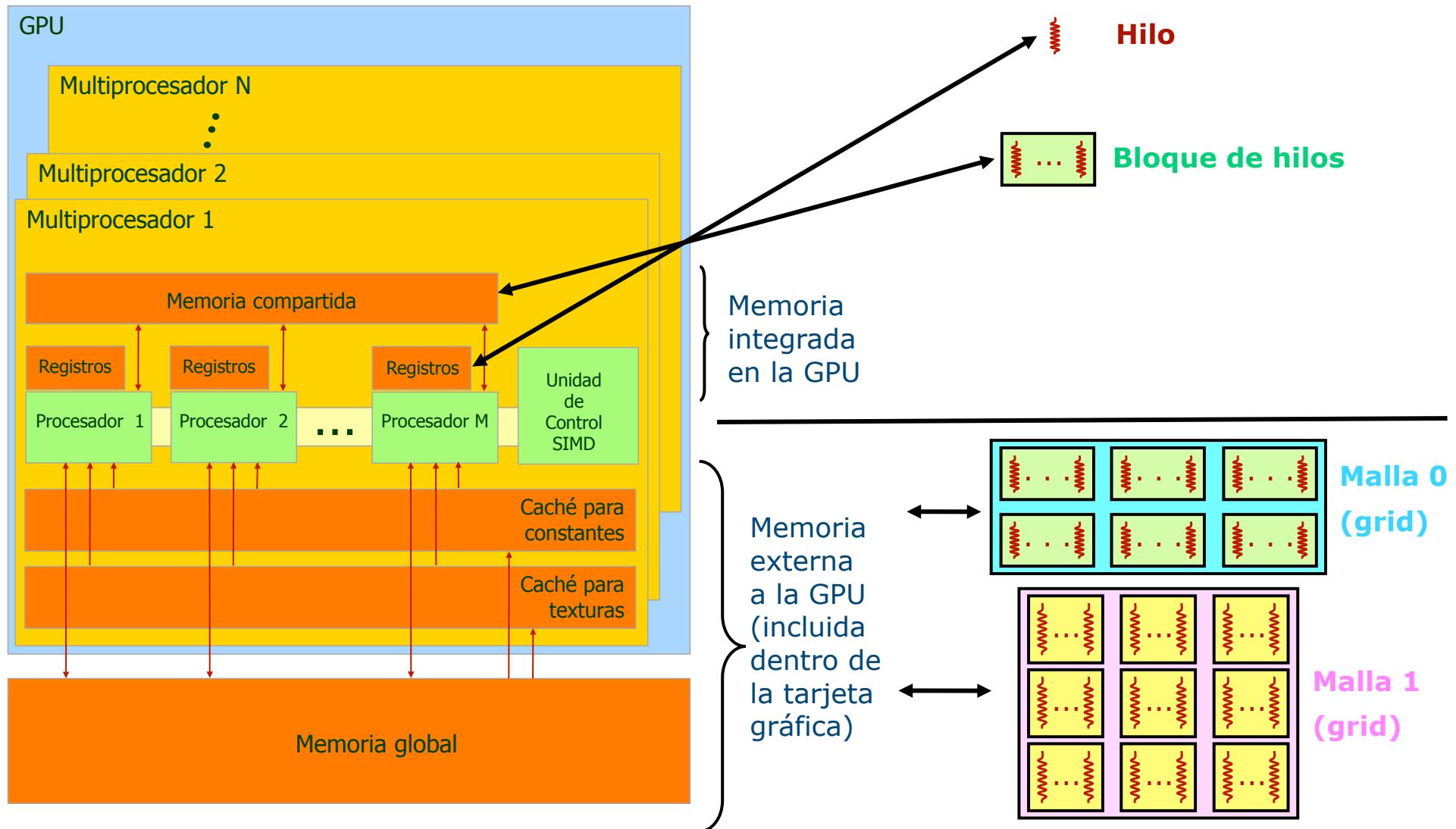
- Cada multiprocesador procesa lotes de bloques, uno detrás de otro
 - Bloques activos = los bloques procesados por un multiprocesador en un lote.
 - Hilos activos = todos los que provienen de los bloques que se encuentren activos.
- Los registros y la memoria compartida de un multiprocesador se reparten 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:

- Dispositivo = GPU = Conjunto de multiprocesadores.
- Multiprocesador = Conjunto de procesadores y memoria compartida.
- Kernel = Programa listo para ser ejecutado en la GPU.
- Malla (grid) = Conjunto de bloques cuya compleción ejecuta un kernel.
- Bloque [de hilos] = Grupo de hilos SIMD que:
 - Ejecutan un kernel delimitando su dominio de datos según su threadID y blockID.
 - Pueden comunicarse a través de la memoria compartida del multiprocesador.
 - 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)

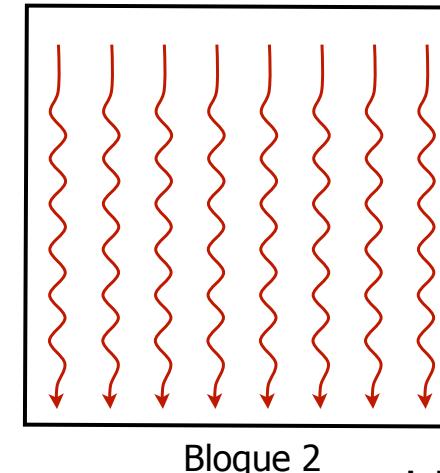
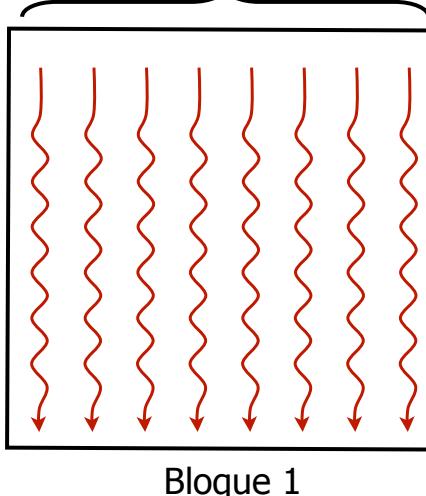
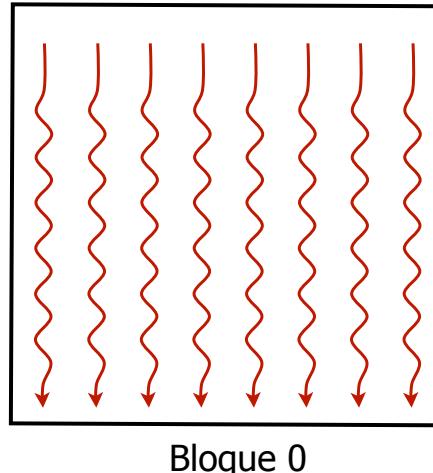
	CUDA Compute Capability (CCC)					Limitación	Impacto
	1.0, 1.1	1.2, 1.3	2.0, 2.1	3.0, 3.5, 3.7	5.0, 5.2		
Multiprocesadores / GPU	16	30	14-16	13-16	4, 5, ...	Hardware	Escala- bilidad
Cores / Multiprocesador	8	8	32	192	128	Hardware	Ritmo de salida de datos
Hilos / Warp	32	32	32	32	32	Software	Paralelismo
Bloques / Multiprocesador	8	8	8	16	32	Software	Conjunto de trabajo
Hilos / Bloque	512	512	1024	1024	1024	Software	
Hilos / Multiprocesador	768	1024	1536	2048	2048	Software	
Registros de 32 bits / Mult.	8K	16K	32K	64K	64K	Hardware	
Memoria compartida / Mult.	16K	16K	16K 48K	16K, 32K, 48K	64K (5.0) 96K (5.2)	Hardware	

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	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/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	Titan	2014	28 nm.
5.0	GM107,8	Muchos	7xx	2014-15	28 nm.
5.2	GM200,4,6	Muchos	9xx/Titan	2014-15	28 nm.

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

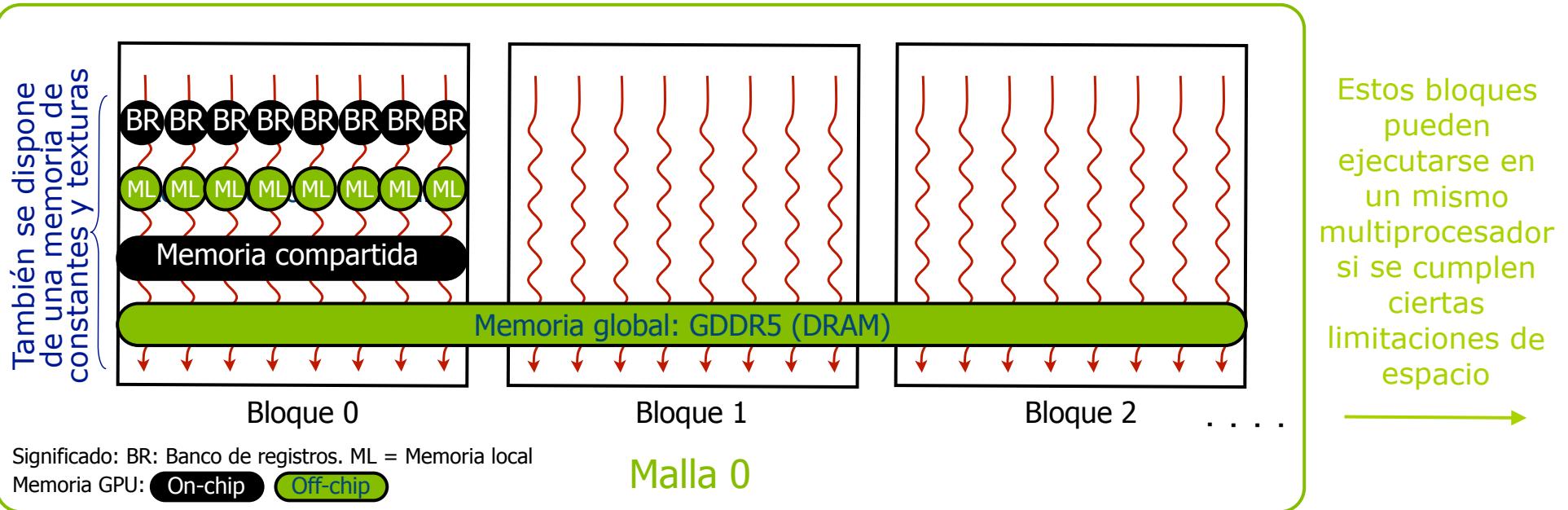
[Límite:
16-32 bloques concurrentes en cada multiprocesador]

- 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 paralelismo 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.

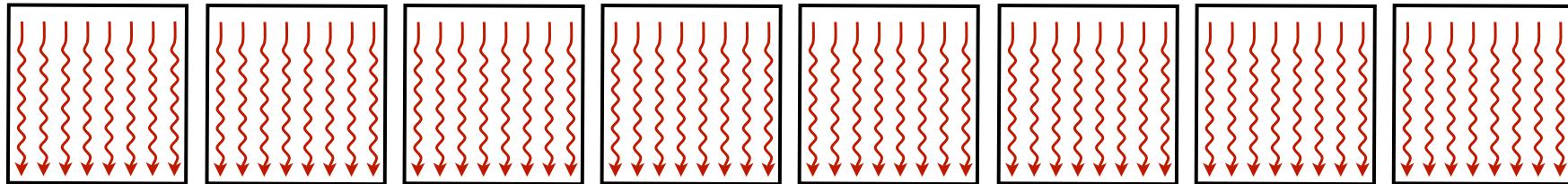
Jugando con las restricciones de memoria en Maxwell para maximizar el uso de recursos

- 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.

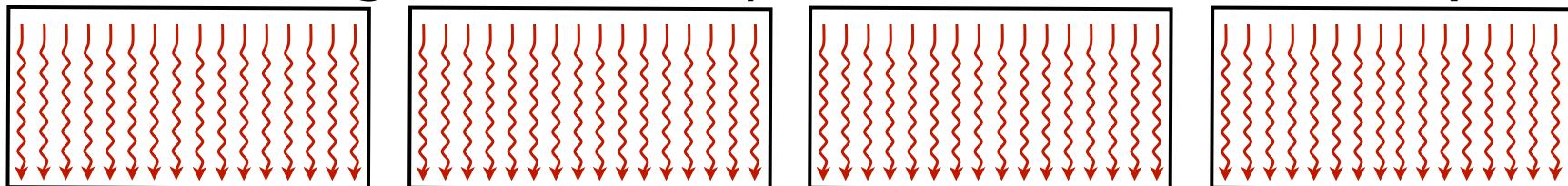
Pensando el pequeño: Particionamiento 1D de un vector de 64 elementos

- 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).

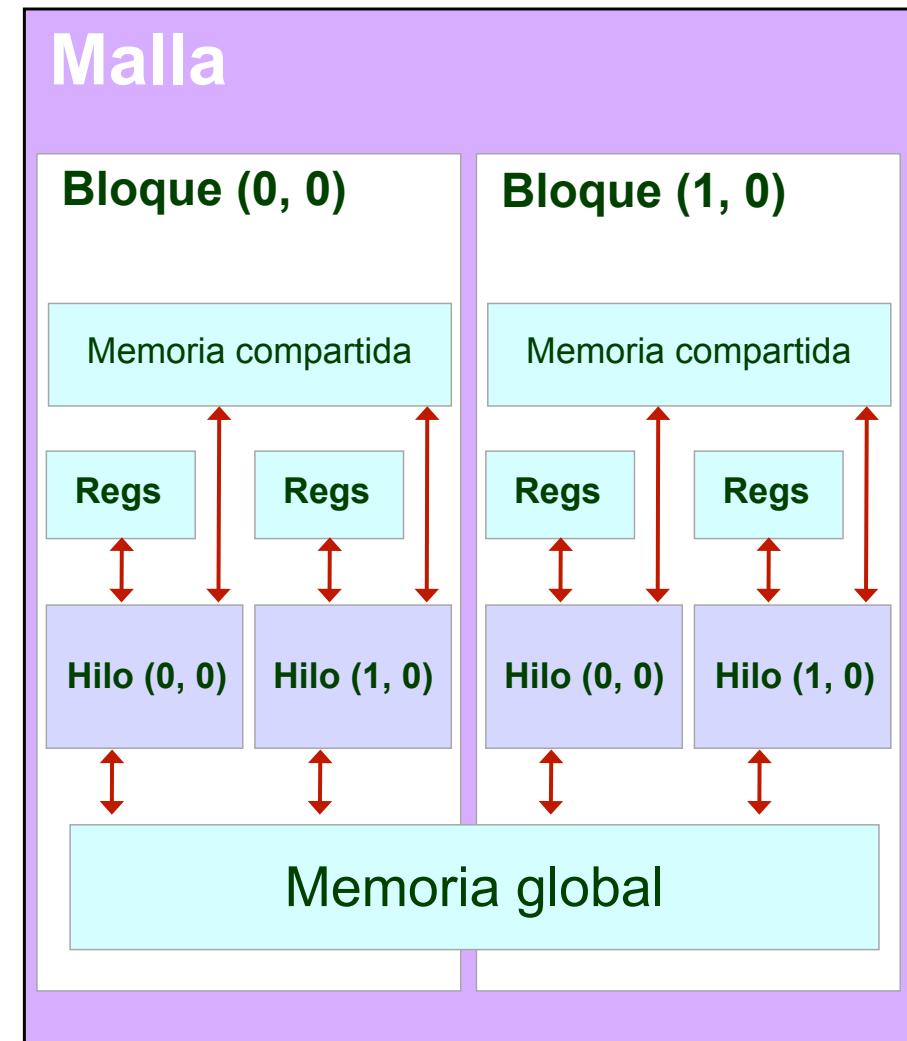


Pensando a lo grande: Particionamiento 1D de un vector de 64 millones de elementos

- 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.

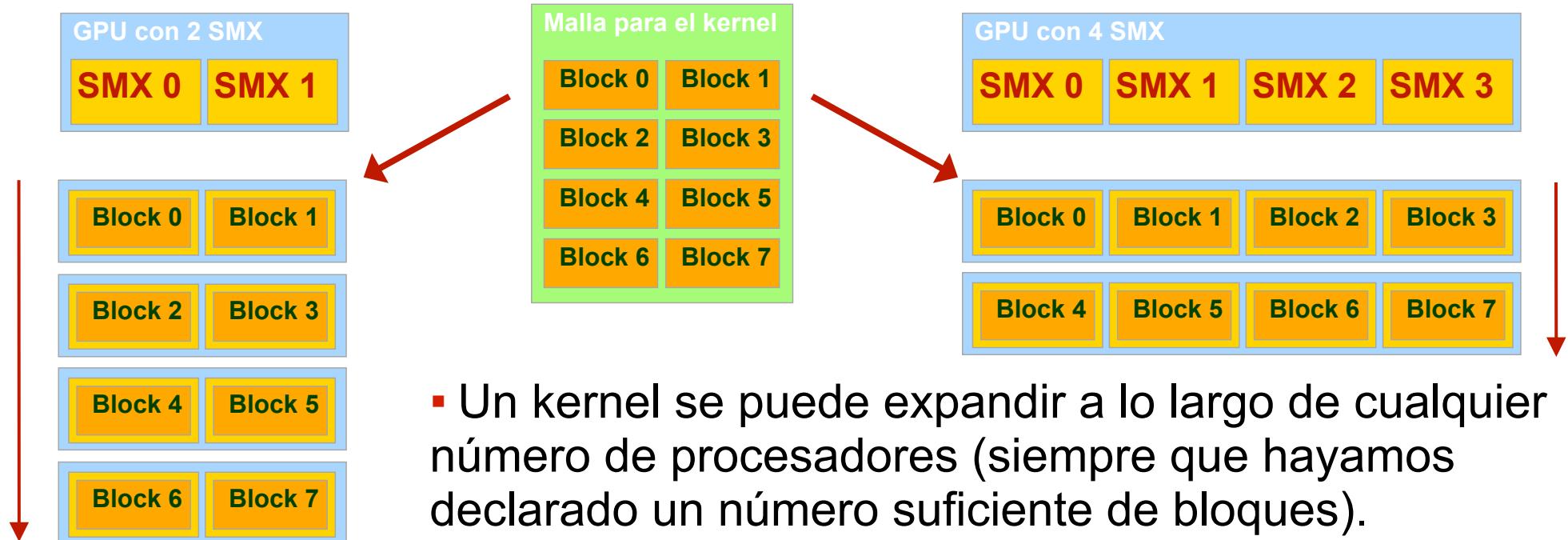
Recopilando sobre kernels, bloques y parallelismo

- Los kernels se lanzan en grids.
- 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.



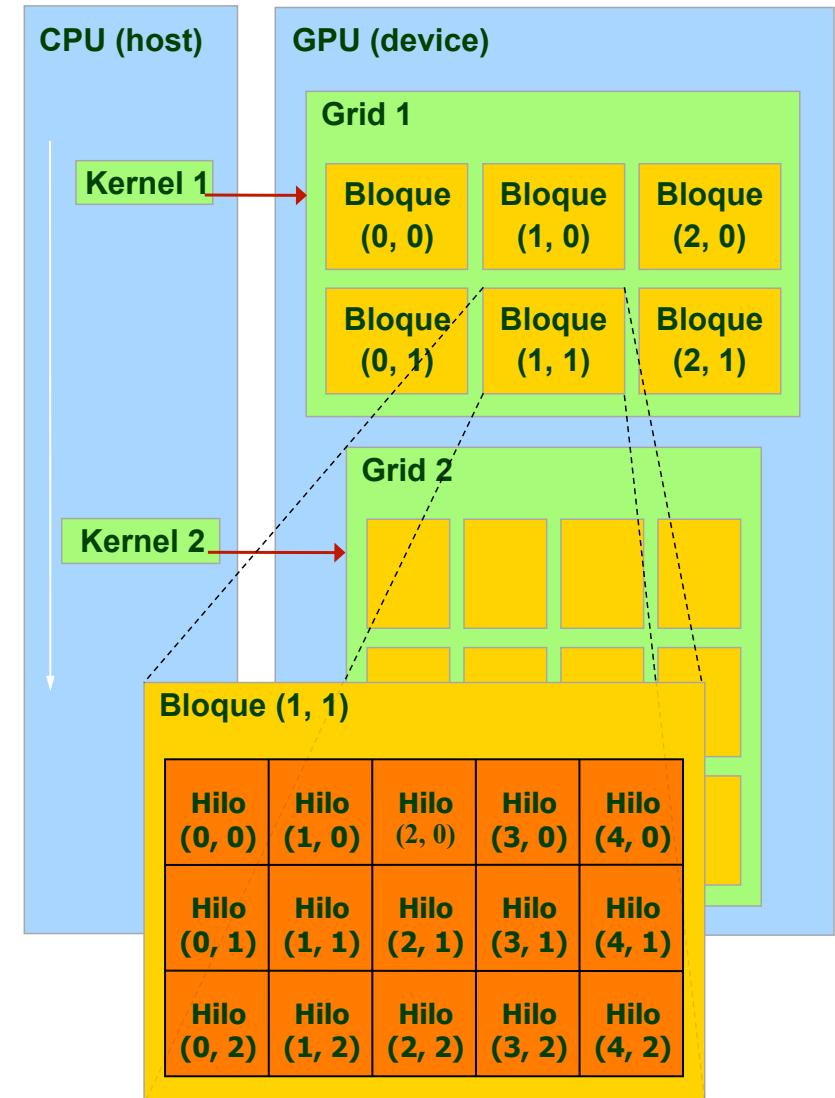
Escalabilidad transparente

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



Particionamiento de computaciones y datos

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



Espacios de memoria

- La CPU y la GPU tiene **espacios de memoria separados**:

- Para comunicar ambos procesadores, se utiliza el bus PCI-express.
- 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).

- Los punteros son sólo direcciones:

- 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.
- 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.)

```

__global__ kernelA(){...}
__global__ kernelB(){...}
int main()
...
kernelA <<< dimGridA, dimBlockA >>> (params.);
...
kernelB <<< dimGridB, dimBlockB >>> (params.);
...

```

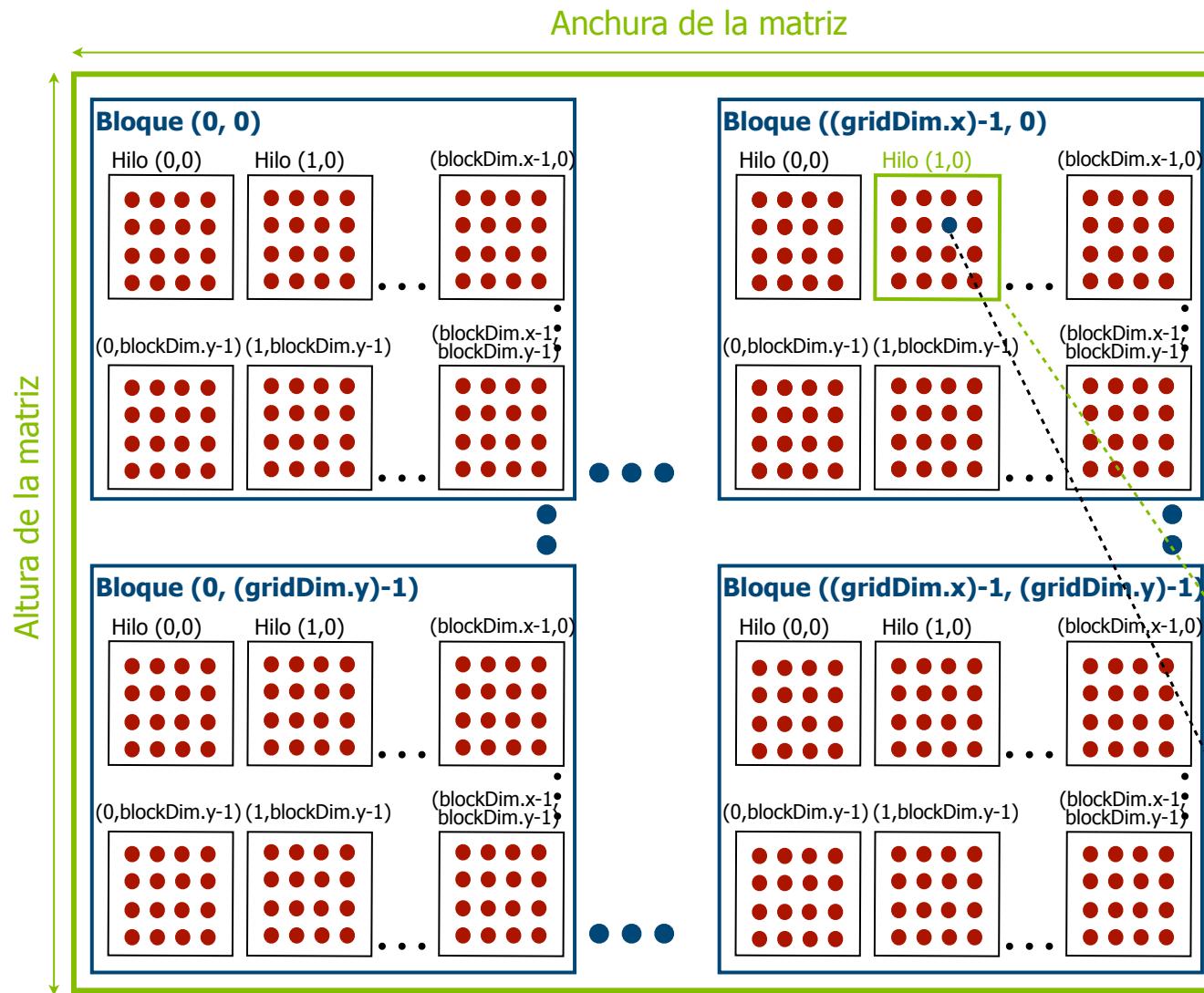
Ejecución ↓

} CPU
→ GPU
} CPU
→ GPU
} CPU



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

Partición de datos para una matriz 2D [para un patrón de acceso paralelo, ver ejemplo 2]



- La posición (hor,ver) para el bloque dentro de la malla es (`blockIdx.x`, `blockIdx.y`).
- La posición (hor,ver) para el hilo dentro del bloque es (`threadIdx.x`, `threadIdx.y`).

Para un caso de 16x16 bloques de 8x8 hilos, y cada hilo encargado de una matriz de 4x4 datos:

Para este hilo:

- `blockIdx.x` es (`gridDim.x`-1, o sea, 15.
- `blockIdx.y` es 0.
- `threadIdx.x` es 1.
- `threadIdx.y` es 0.

Este dato está en la columna:
 $[blockIdx.x * blockDim.x * 4] + (threadIdx.x * 4) + 2$, esto es,
 $[15 * 8 * 4] + (1 * 4) + 2 = 486$.
Y en la fila:
 $[blockIdx.y * blockDim.y * 4] + (threadIdx.y * 4) + 1 = 0 + 0 + 1 = 1$.

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

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

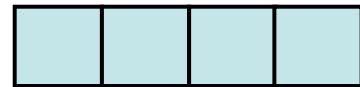
void main()
{
    ....
    dim3 dimBlock (blocksize);
    dim3 dimGrid (ceil(N/(float)blocksize));
    incremento_en_gpu<<<dimGrid, dimBlock>>>(a, b, N);
}
```

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

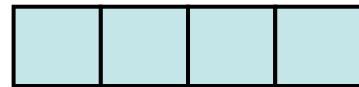


Con $N=16$ y $\text{blockDim}=4$, tenemos 4 bloques de hilos, encargándose cada hilo de computar un elemento del vector. Es lo que queremos: Paralelismo de grano fino en la GPU.

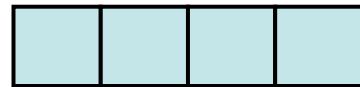
Extensiones al lenguaje



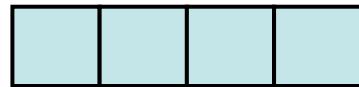
```
blockIdx.x = 0
blockDim.x = 4
threadIdx.x = 0,1,2,3
idx = 0,1,2,3
```



```
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

```

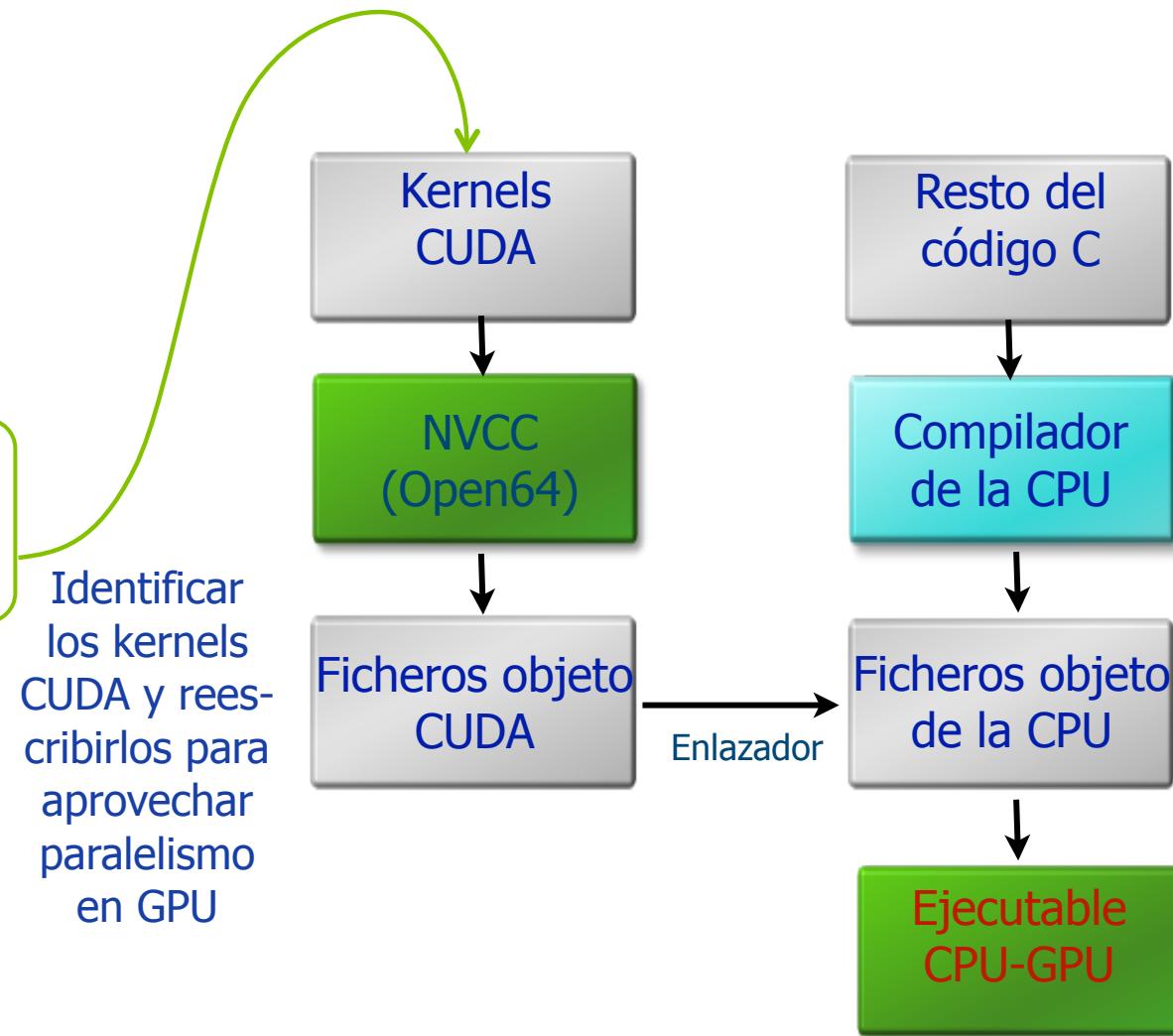
void funcion_en_CPU( ... )
{
    ...
}

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(...);
    ...
}

```



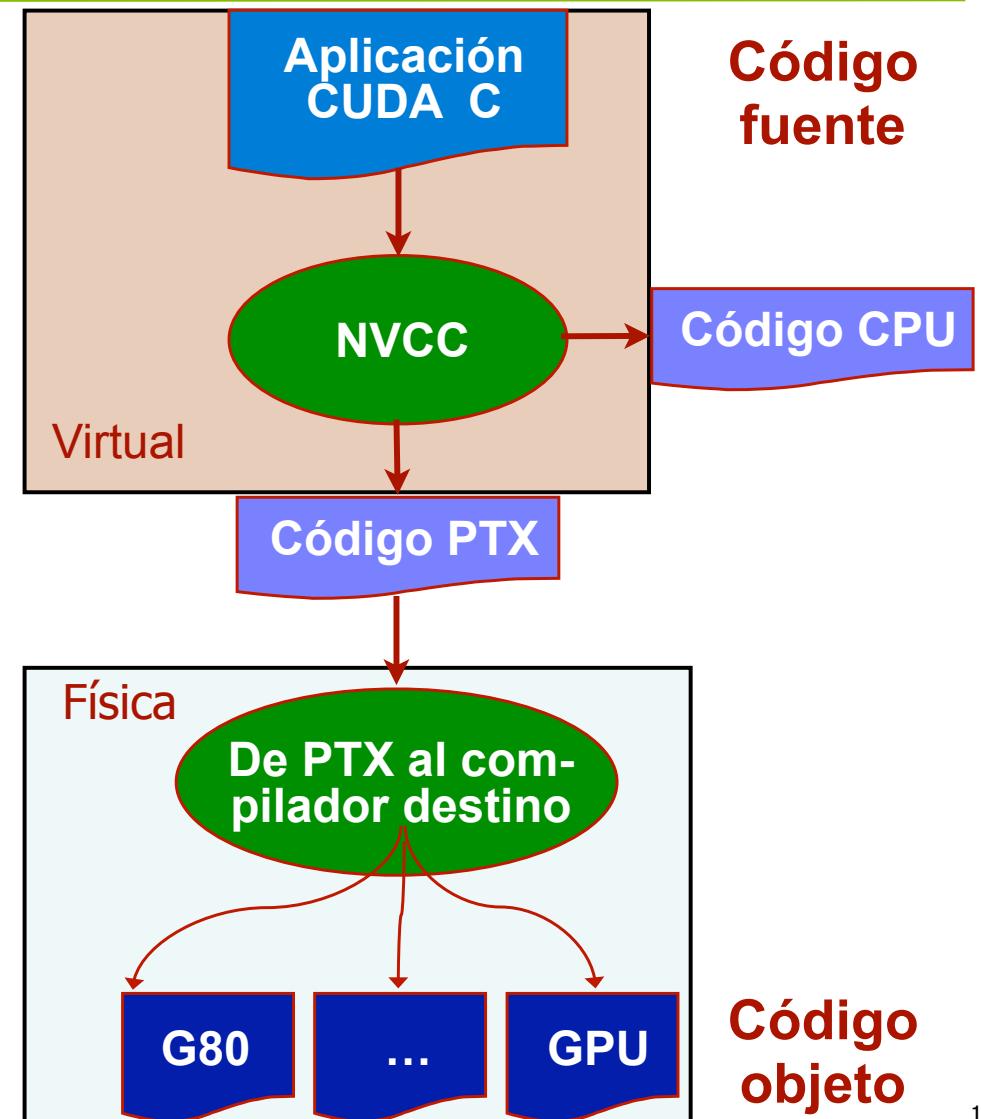
Los diferentes módulos de compilación

- El código fuente CUDA se compila con NVCC.

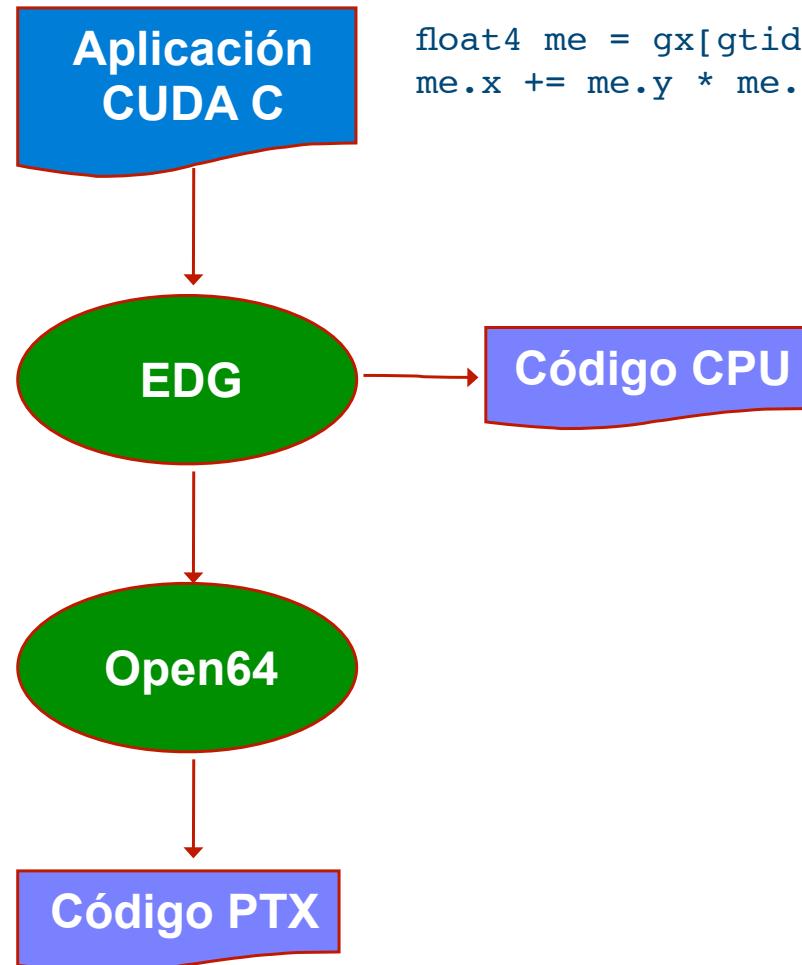
- NVCC separa el código que se ejecuta en CPU del que lo hace en GPU.

- 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**

- Separa código GPU y CPU.

- **Open64**

- Genera ensamblador PTX.

- **Parallel Thread eXecution (PTX)**

- Máquina virtual e ISA.

- Modelo de programación.

- Recursos y estados de ejecución.

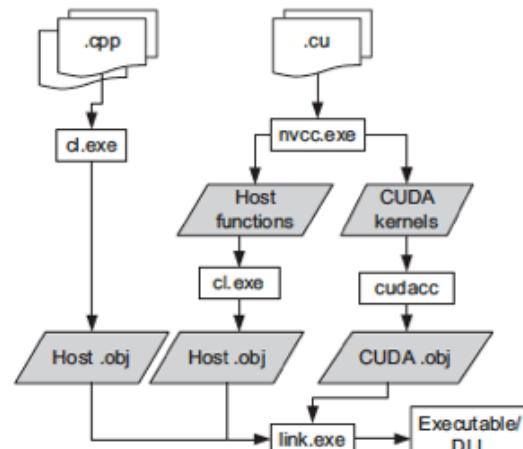
```

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

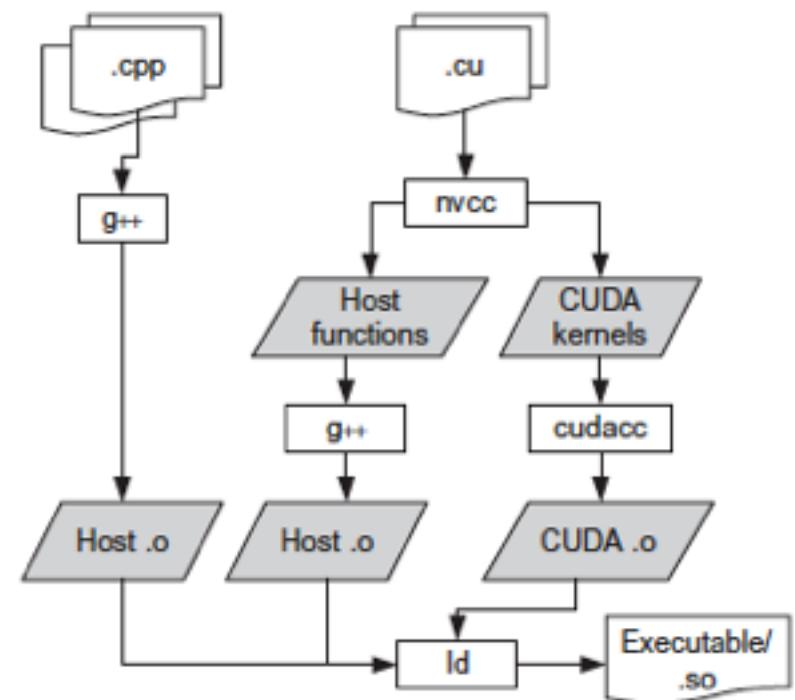
NVCC (NVIDIA CUDA Compiler)

- 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, esperas de todos los hilos en 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.

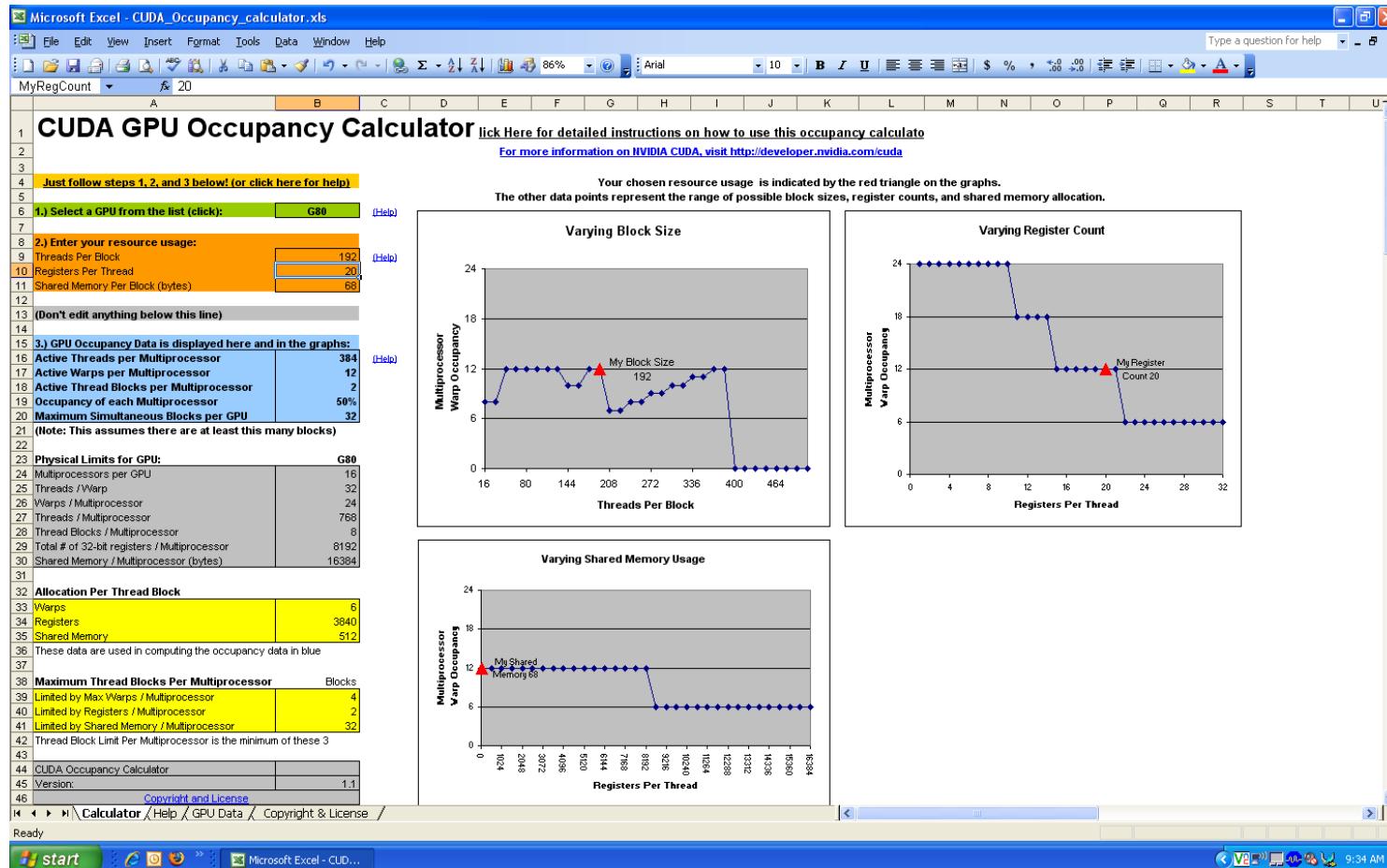
Parametrización de una aplicación

- 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 y 500 GB/s.
 - ➌ El tamaño del banco de registros de cada SM: 8K, 16K, 32K, 64K.
 - ➌ El tamaño de la memoria compartida: 16 KB., ampliable a 48 KB. en Fermi y Kepler.
 - ➌ Hilos: Comprobar el límite por bloque y el límite total.
 - ➍ Por bloque: 512 (G80 y GT200), 1024 (Fermi y Kepler).
 - ➍ Total: 768 (G80), 1024 (GT200), 1536 (Fermi), 2048 (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 en las generaciones Fermi y Kepler.
- 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 Kepler, Maxwell y 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 8K (G80), 16K (GT200), 32K (Fermi), o 64K (Kepler), así que consumiendo 10 registros/hilo es posible ejecutar:
 - En G80: 768 hilos/SM, o sea, 3 bloques de 256 hilos [$3 \times 256 \times 10 = 7680$] (< 8192).
 - En Kepler: Alcanzamos el máximo de 2048 hilos por SMX, aunque nos quedamos muy cortos en el uso de los registros (podríamos haber usado hasta 29 regs./hilo):
 $8 \text{ bloques} \times 256 \text{ hilos/bloque} \times 10 \text{ registros/hilo} = 20480 \text{ regs.} (< 65536 \text{ máx.})$.
 - En el caso de la G80, si hubiéramos consumido 11 registros/hilo, ya no podríamos acomodar 3 bloques en el SM, sino sólo 2, por lo que perderíamos 1/3 del paralelismo => Habría que esforzarse en reducir de 11 a 10 el número de registros.
 - En el caso de Kepler, podríamos usar hasta 29 registros sin sacrificar paralelismo.

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: 16 KB (CCC 1.x), 16/48 KB (CCC 2.x), 16/32/48 KB (3.x).

- Para el caso anterior sobre la G80, no gastaremos más de 5 KB de memoria compartida por cada bloque para que podamos ejecutar el máximo de 3 bloques en paralelo en cada multiprocesador:

- $3 \text{ bloques} \times 5 \text{ KB./bloque} = 15 \text{ KB} (< 16 \text{ KB.})$

- A partir de 5.33 KB. de memoria compartida usada por cada bloque, estamos sacrificando un 33% del paralelismo, igual que sucedía antes si no éramos capaces de bajar a 10 registros/kernel.



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 = threadIdx.x + (blockDim.x* blockIdx.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 >>> ()`

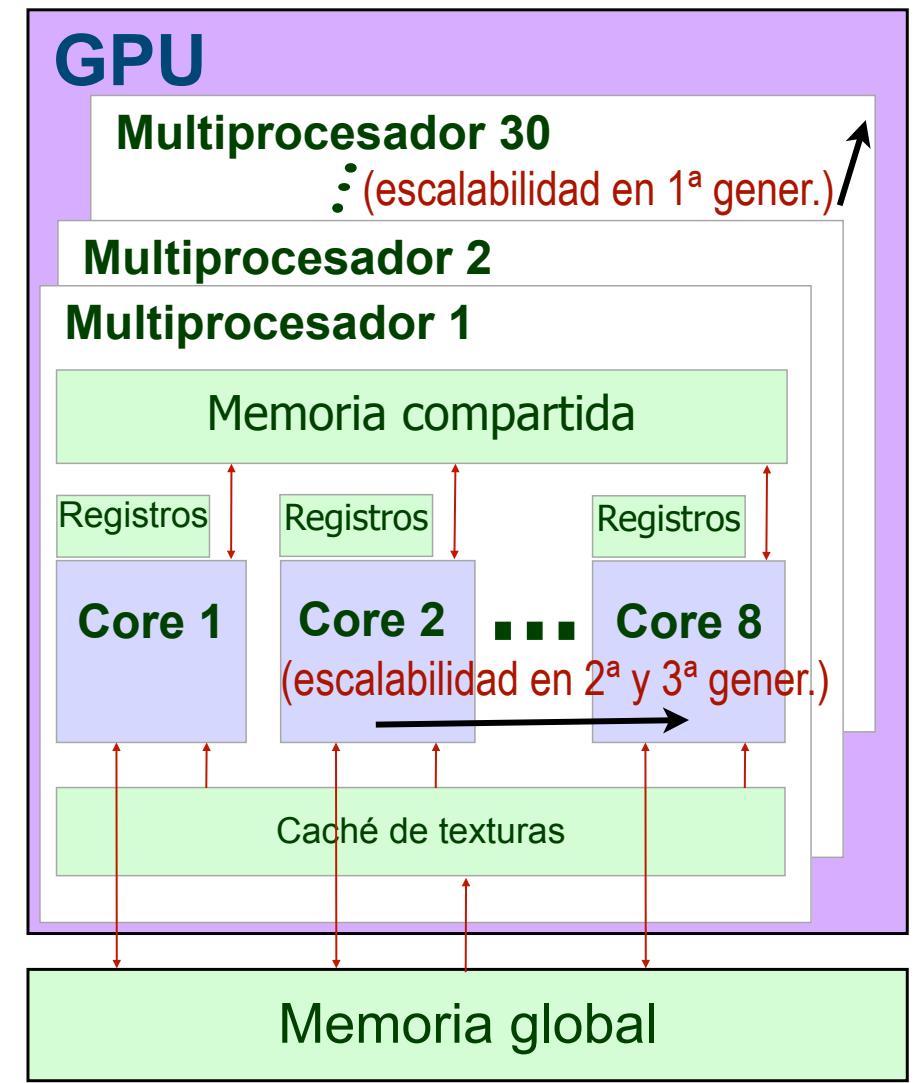
Ejecuta 1 bloque de 1 hilo.
No hay paralelismo.

• `vecAdd <<< B, 1 >>> ()`

Ejecuta B bloques de 1 hilo.
Hay paralelismo entre multiprocesadores.

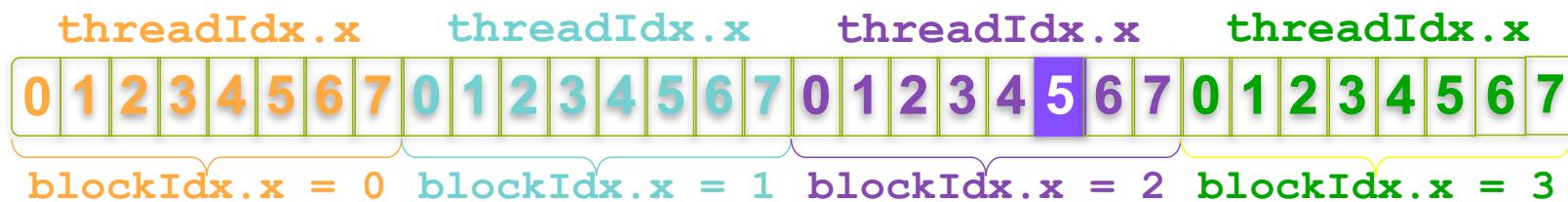
• `vecAdd <<< B, M >>> ()`

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 = threadIdx.x + blockDim.x * blockIdx.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.
 - $tid = 5 + (8 * 2) = 21$ (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 = threadIdx.x + (blockDim.x * blockIdx.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)

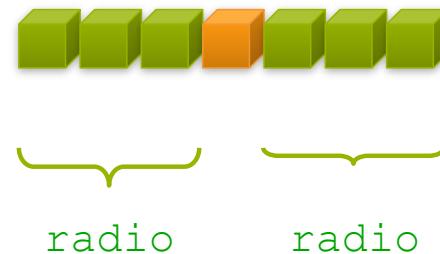
Por qué hemos seleccionado este código

- 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)

- Apliquemos 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

- ➊ Los hilos de un mismo bloque pueden compartir datos a través de memoria compartida.
- ➋ El programador gestiona la memoria compartida de forma explícita, insertando el prefijo __shared__ en la declaración de la variable.
 - ➌ Los datos se alojan para cada uno de los bloques.
 - ➍ La memoria compartida es extremadamente rápida:
 - ➎ 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).
 - ➏ 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

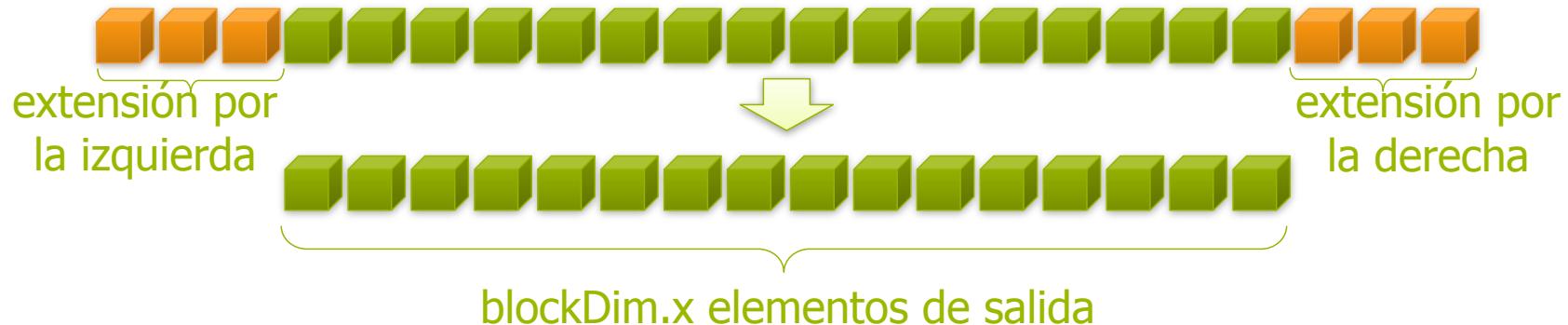
- El uso de los registros y la memoria compartida limita el paralelismo.
 - Si dejamos espacio para un segundo bloque en cada multiprocesador, el banco de registros y la memoria compartida se partitionan (aunque la ejecución no es simultánea, el cambio de contexto es inmediato).
- Ya mostramos ejemplos 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

```

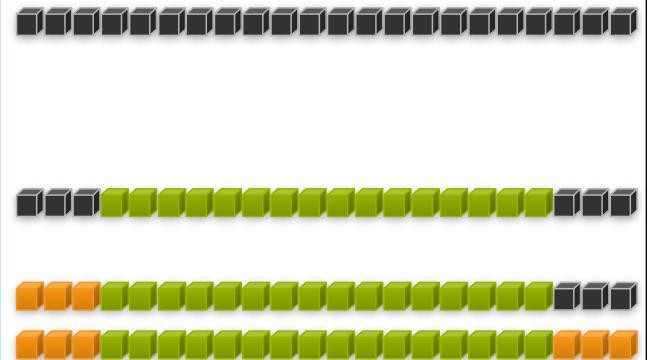
__global__ void stencil_1d(int *d_in, int *d_out)
{
    __shared__ int temp[BLOCKSIZE + 2 * RADIO];
    int gindex = threadIdx.x
                + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIO;

    // 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

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

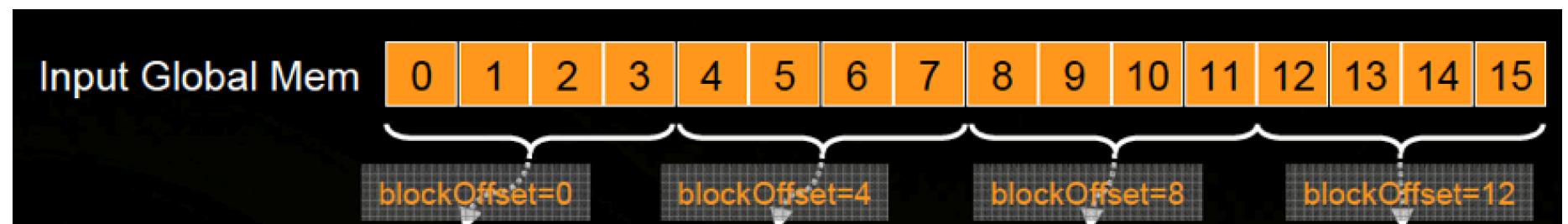
- Es una solución demasiado simplista: No aspira a aplicar paralelismo masivo porque el máximo tamaño de bloque es 1024 hilos, con lo que ése sería el mayor vector que este código podría aceptar como entrada.

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];
}
```

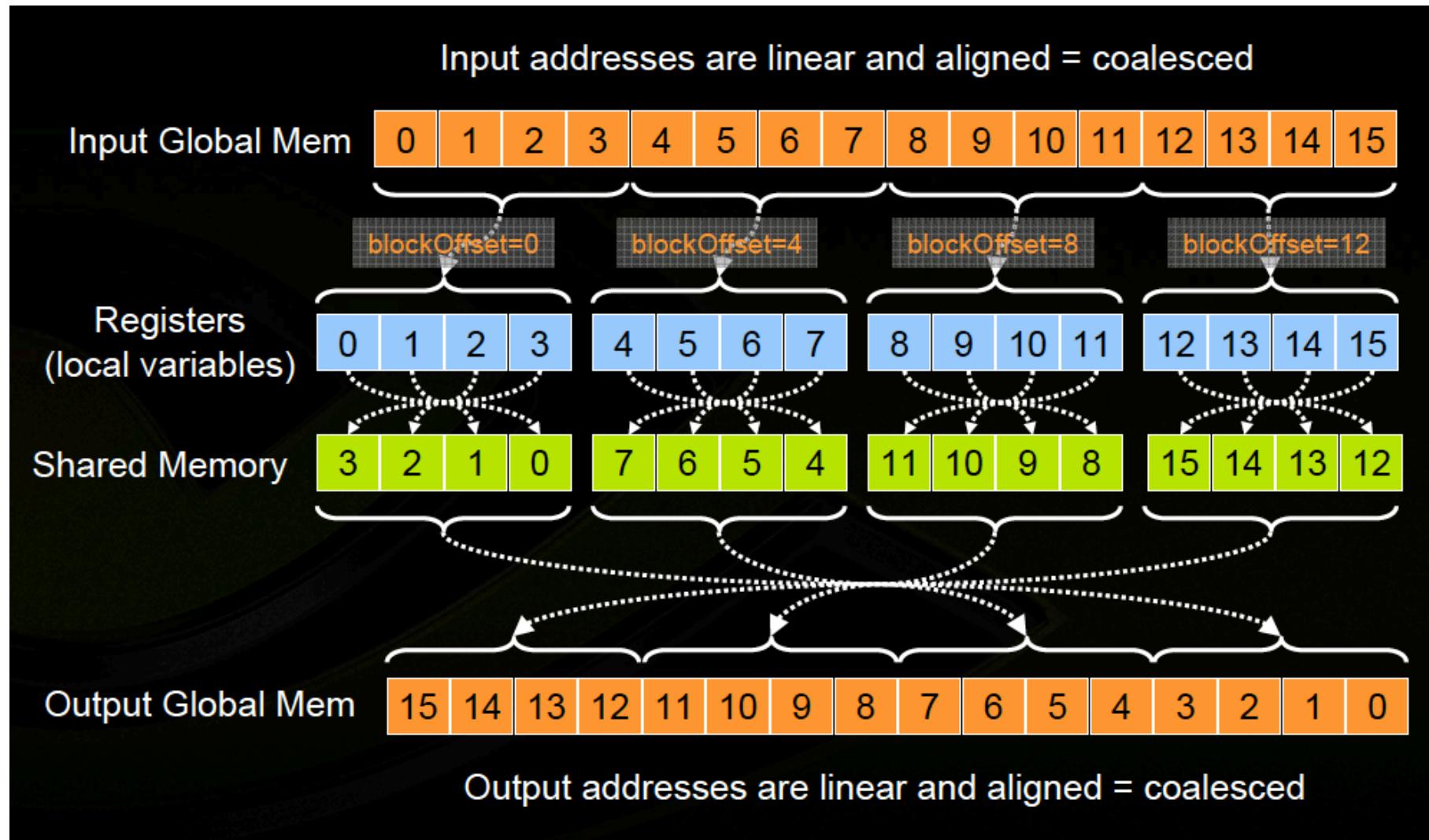
- Por ejemplo, para 4 bloques de 4 hilos, tendríamos:



Output Global Mem

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
----	----	----	----	----	----	---	---	---	---	---	---	---	---	---	---

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 = threadIdx.x + blockIdx.x * blockDim.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.

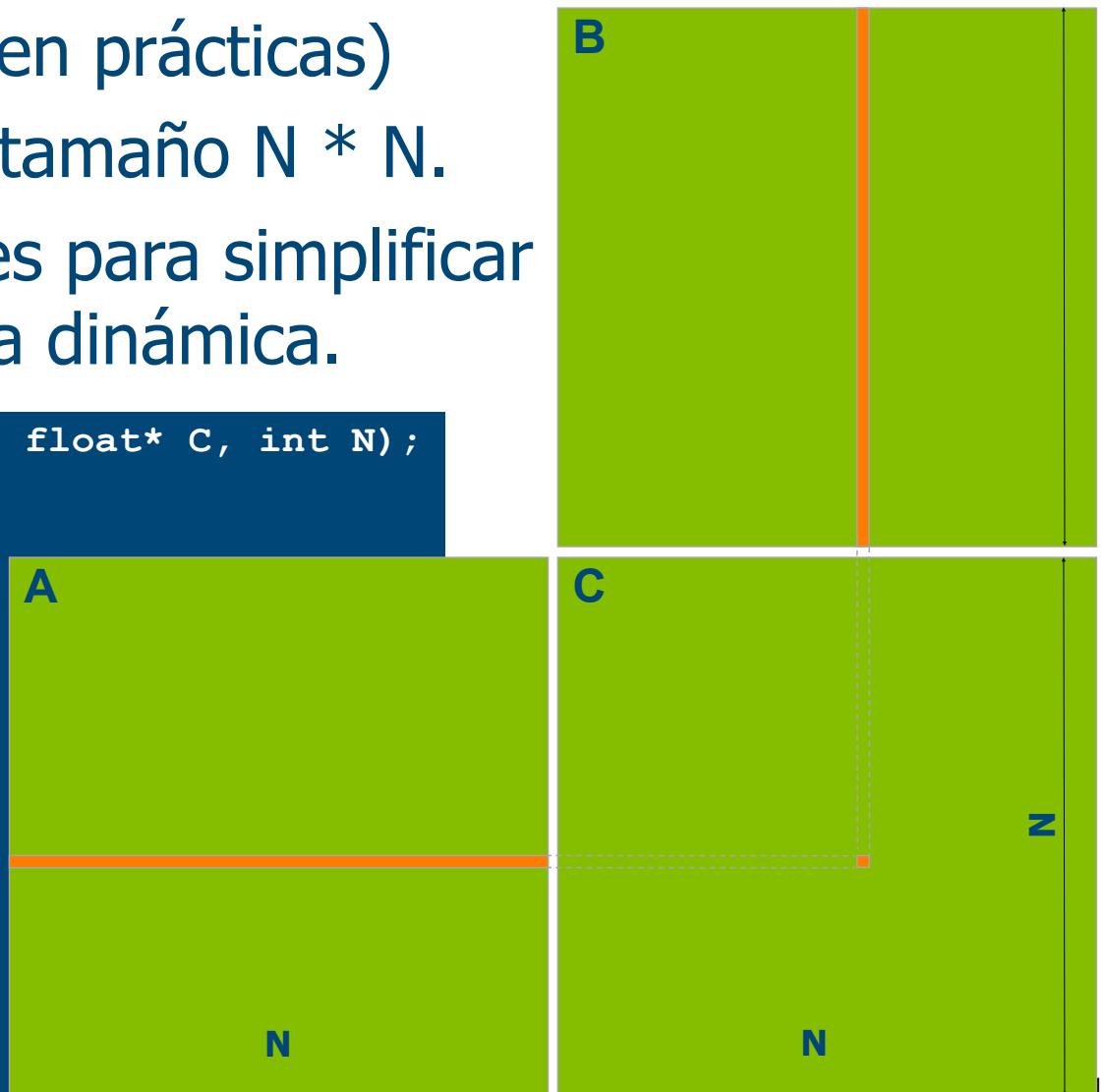


VI. 4. Producto de matrices

Versión de código CPU escrita en lenguaje C

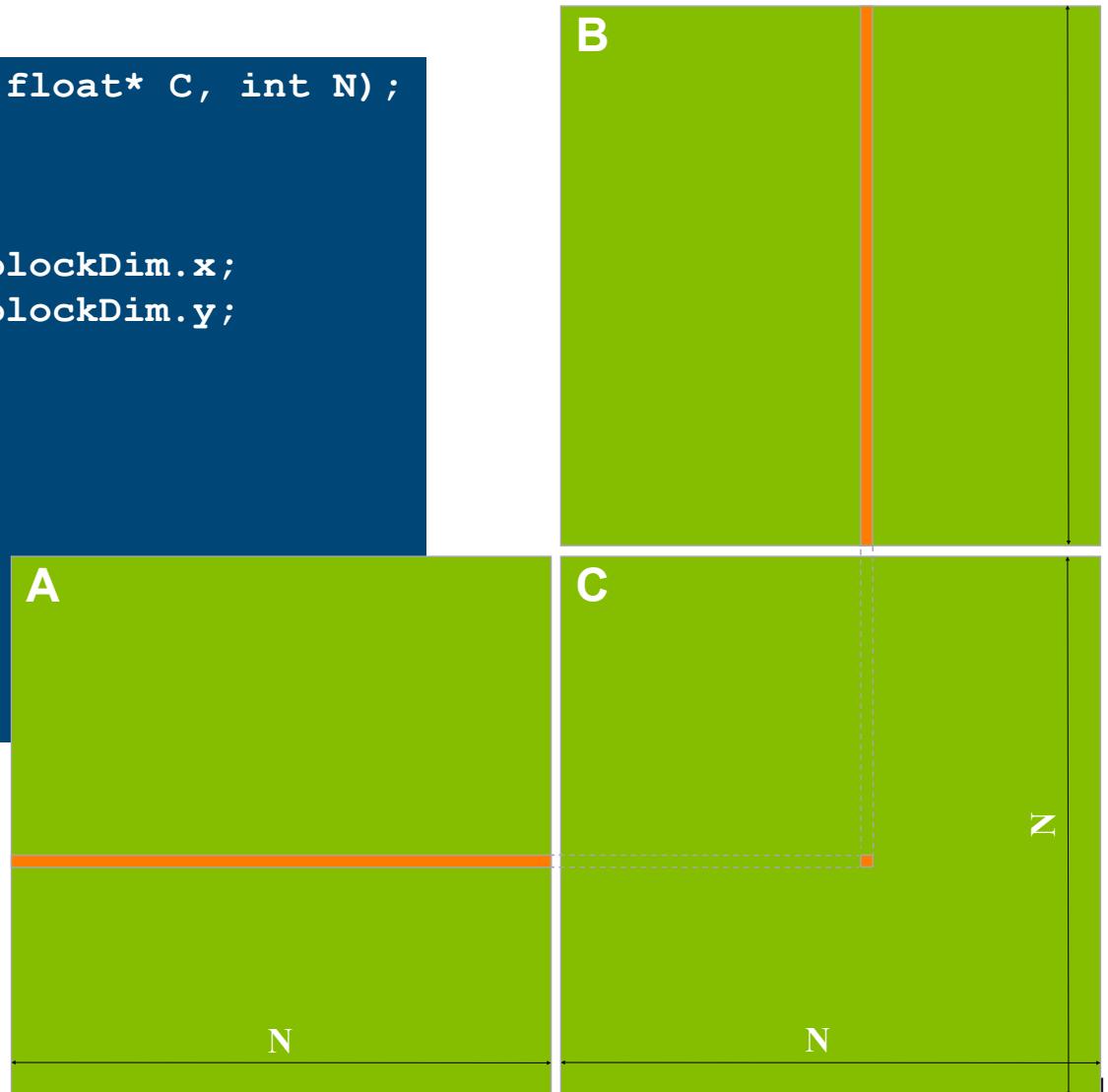
- $C = A * B$. ($P = M * N$ en prácticas)
- Matrices cuadradas de tamaño $N * N$.
- Linearizadas en vectores para simplificar el alojamiento de memoria dinámica.

```
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;
        }
}
```



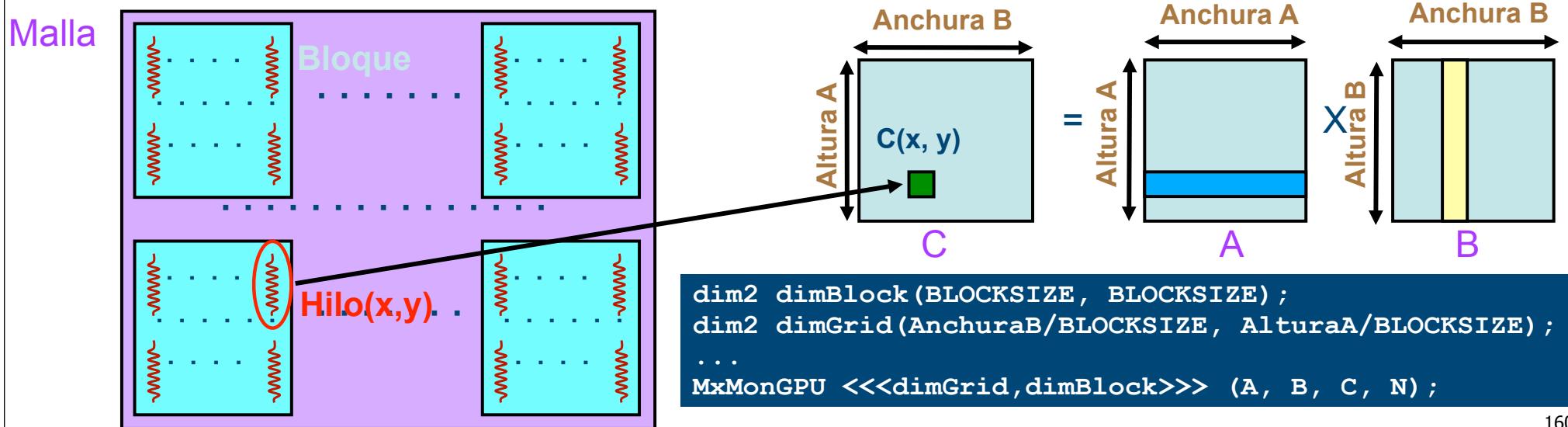
Versión CUDA para el producto de matrices: Un primer borrador para el código paralelo

```
void MxMonGPU(float* A, float* B, float* C, int N) {
{
    float sum=0;
    int i, j;
    i = threadIdx.x + blockIdx.x * blockDim.x;
    j = threadIdx.y + blockIdx.y * blockDim.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.

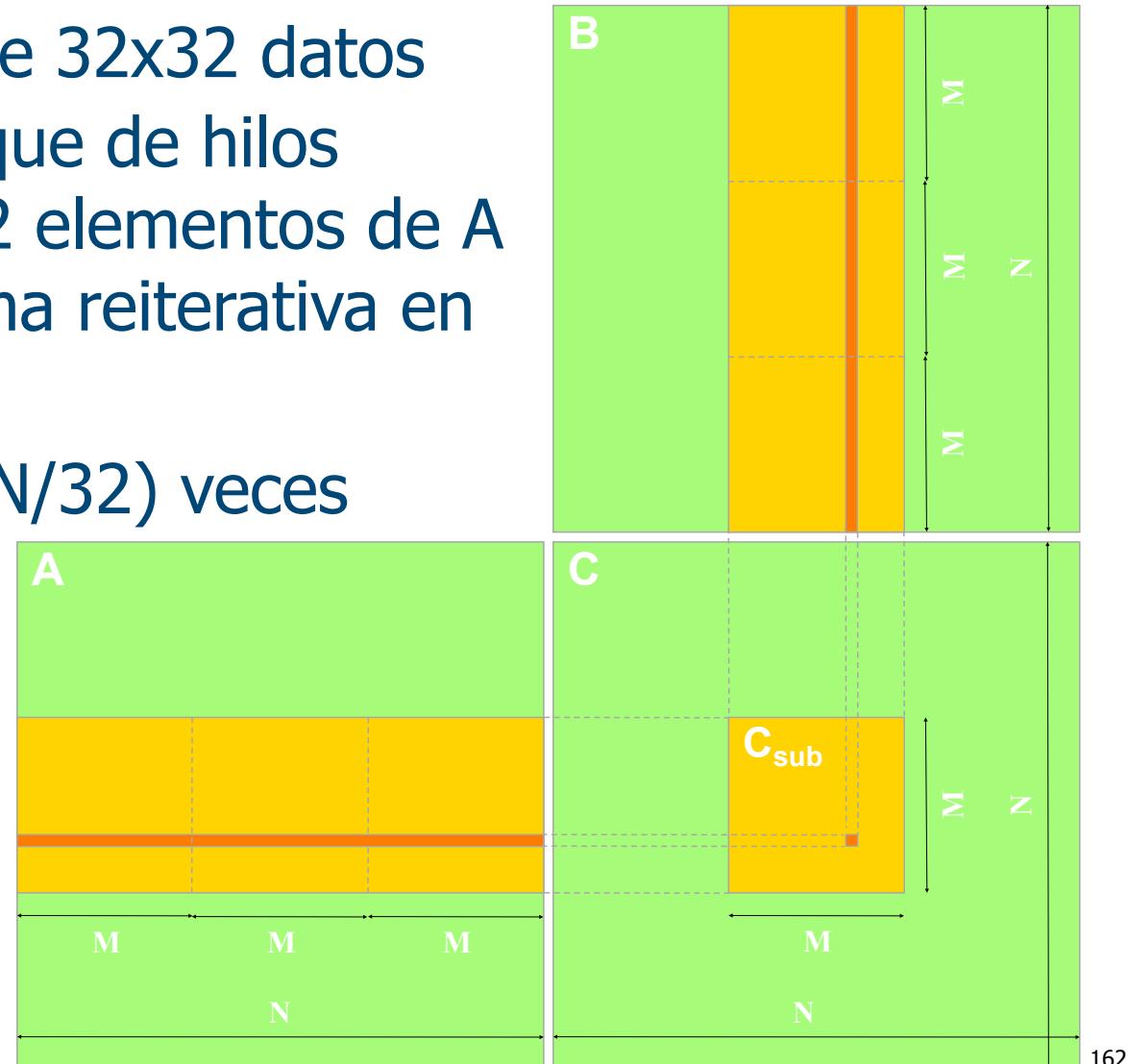
Utilizando la memoria compartida: Versión con mosaicos (tiling) para A y B

- La submatriz de C_{sub} de 32x32 datos computada por cada bloque de hilos utiliza mosaicos de 32x32 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 = tx + blockIdx.x * blockDim.x;      j = ty + blockIdx.y * blockDim.y;
    __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;
```

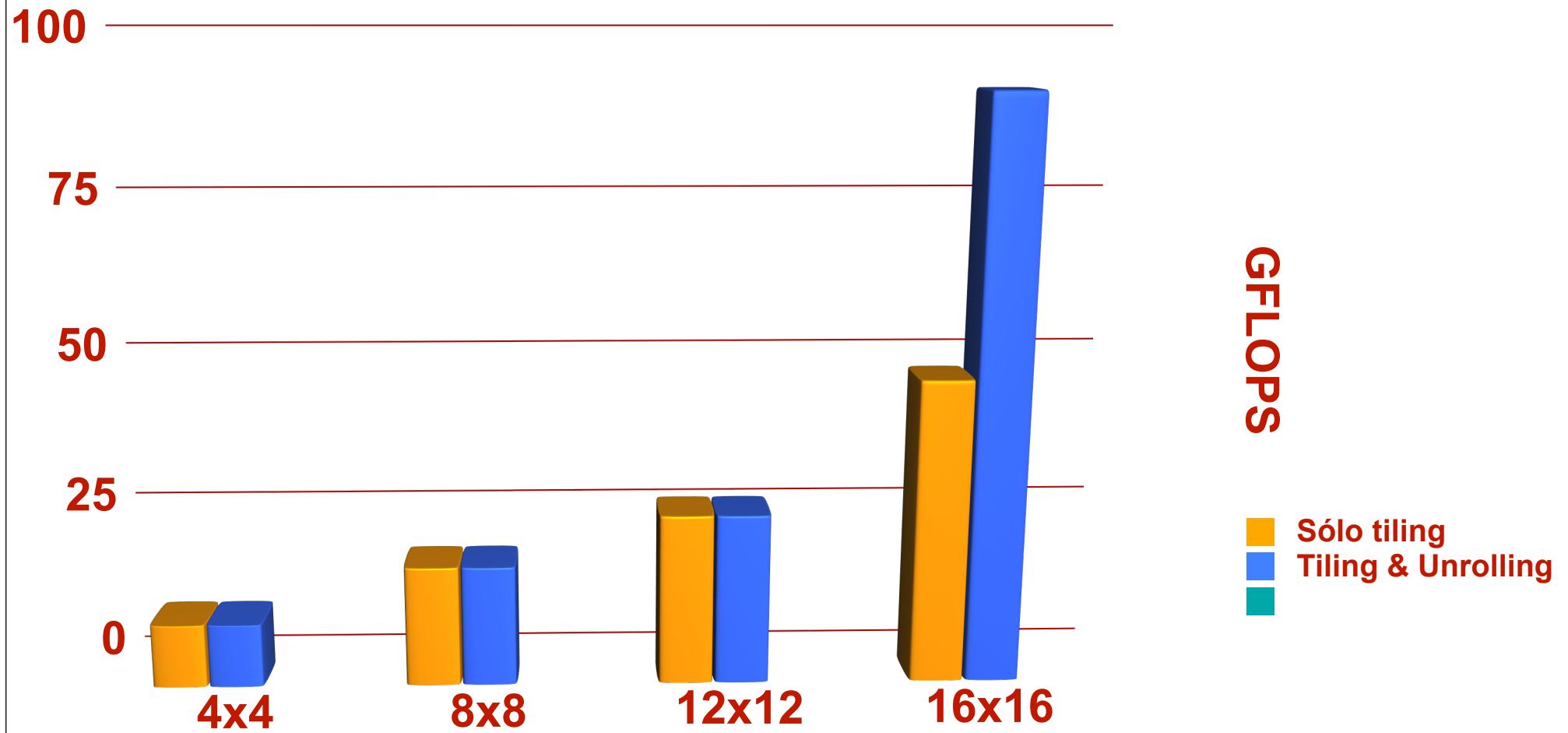
Desenrollando el bucle

```
__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

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



About CUDA

All about the NVIDIA CUDA parallel computing platform

[Learn more >](#)



Getting Started

First steps for getting started in parallel computing

[Learn more >](#)



Tools & Ecosystem

From accelerated cloud appliances to profiling tools, a gold mine of information

[Learn more >](#)



Academic Collaboration

Partner with NVIDIA to advance parallel computing education and research

[Learn more >](#)



CUDA Downloads

Get the latest and greatest version of the CUDA Toolkit

[Learn more >](#)



Resources

Materials and links especially for GPU Computing professionals and developers



Getting Started

First steps for getting started in parallel computing

[Learn more >](#)

Optimized Libraries

Drop-in, Industry standard libraries replace MKL, IPP, FFTW and other widely used libraries. Some feature automatic multi-GPU scaling,

[Get Started with GPU-Accelerated Libraries](#)

Compiler Directives

Easy: simply insert hints in your code
Open: run on either CPU or GPU
Powerful: tap into the power of GPUs within minutes

[Get Started with Directives](#)

Programming Language

Develop your own parallel applications and libraries using a programming language you already know.

Get Started With:

- C/C++ using CUDA C
- Fortran using CUDA Fortran
- Python

Tools & Ecosystem



From accelerated cloud appliances to profiling tools, a gold mine of information



Accelerated Solutions

GPUs are accelerating many applications across numerous industries.

[Learn more >](#)



Numerical Analysis Tools

Applications with high arithmetic density can enjoy amazing GPU acceleration.

[Learn more >](#)



GPU-Accelerated Libraries

Adding acceleration to your application can be as easy as calling a library function.

[Learn more >](#)



Language and APIs

GPU acceleration can be accessed from most popular programming languages.

[Learn more >](#)



Performance Analysis Tools

Find the best solutions for analyzing your application's performance profile.

[Learn more >](#)



Debugging Solutions

Powerful tools can help debug complex parallel applications in intuitive ways.

[Learn more >](#)



Key Technologies

Learn more about parallel computing technologies and architectures.



Cluster Management

Managing your GPU cluster will help achieve maximum performance.



Job Scheduling

Scheduling jobs on your GPU Cluster can be simple and intuitive.



Academic Collaboration

Partner with NVIDIA to advance parallel computing education and research



Academic Programs

All about our investment in academia through our four core programs.

[Learn more >](#)



GPU Centers

A showcase of all our GPU Centers – check for your institution.

[Learn more >](#)



CUDA Fellows

Our partners who are committed to leading the use and adoption of CUDA.

[Learn more >](#)



Educators Network

A collaborative area for those looking to educate others on massively parallel programming.

[Learn more >](#)



Curriculum & Teaching Resources

Hands-on exercises and access to GPUs for your parallel programming courses.

[Learn more >](#)



Additional Resources

Materials and links especially for academia.

[Learn more >](#)

CUDA Downloads



Get the latest and greatest version of the CUDA Toolkit

 **NVIDIA CUDA ZONE**

[Getting Started](#)

[Downloads](#)

[Training](#)

[Ecosystem](#)

[Forum](#)



[Register Now](#)

[Login](#)

[Home](#) > [CUDA ZONE](#) > [Tools & Ecosystem](#) > [CUDA Toolkit](#) > [CUDA 7 Downloads](#)

CUDA 7 Downloads

Check out:

- [CUDA 7 Performance Report](#) and [Webinar Recording](#)
- An informative webinar by Ujval Kapasi, NVIDIA's CUDA Product Manager [CUDA 7 Features and Overview](#)
- [The Power of C++11 in CUDA 7](#), another technical blog on Parallel Forall.

If you find any issues please file a bug [requires membership of the [CUDA Registered Developer Program](#)].

Please Note: There is a recommended patch for CUDA 7.0 which resolves an issue in the cuFFT library that can lead to incorrect results for certain input sizes less than or equal to 1920 in any dimension when `cufftSetStream()` is passed a [non-blocking stream](#) [e.g., one created using the `cudaStreamNonBlocking` flag of the CUDA Runtime API or the `CU_STREAM_NON_BLOCKING` flag of the CUDA Driver API].

[Windows](#)

[Linux x86](#)

[Linux POWER8](#)

[Mac OSX](#)

Version

Network Installer

Local Installer

Windows 8.1

[EXE](#) {8.0MB}

[EXE](#) {939MB}

Windows 7

Win Server 2012 R2

Win Server 2008 R2

cuFFT Patch

[ZIP](#) {52MB} , [README](#)

[Windows Getting Started Guide](#)

[Windows FAQ](#)

Q: Where is the notebook installer?

A: Previous releases of the CUDA Toolkit had separate installation packages for notebook and desktop

[Documentation](#)

[Release Notes](#)

[End User License Agreement](#)

[Online Documentation](#)

[CUDA Toolkit Overview](#)

[Checksums](#)

[Windows](#)

[Linux x86](#)

[Linux POWER8](#)

[Mac OSX](#)

Version

Network Installer

Local Package Installer

Runfile Installer

Fedora 21

Coming Soon

Coming Soon

Coming Soon

OpenSUSE 13.2

[RPM](#) {3KB}

[RPM](#) {1GB}

[RUN](#) {1.1GB}

OpenSUSE 13.1

[RPM](#) {3KB}

[RPM](#) {1GB}

[RUN](#) {1.1GB}

RHEL 7

[RPM](#) {10KB}

[RPM](#) {1GB}

[RUN](#) {1.1GB}

CentOS 7

[RPM](#) {18KB}

[RPM](#) {1GB}

[RUN](#) {1.1GB}

RHEL 6

[RPM](#) {18KB}

[RPM](#) {1GB}

[RUN](#) {1.1GB}

CentOS 6

[RPM](#) {18KB}

[RPM](#) {1GB}

[RUN](#) {1.1GB}

SLES 12

[RPM](#) {3KB}

[RPM](#) {1.1GB}

[RUN](#) {1.1GB}

SLES 11 (SP3)

[RPM](#) {3KB}

[RPM](#) {1.1GB}

[RUN](#) {1.1GB}

SteamOS 1.0-beta

[DEB](#) {3KB}

[DEB](#) {1.5GB}

[RUN](#) {1.1GB}

Ubuntu 14.10

[DEB](#) {3KB}

[DEB](#) {1.5GB}

[RUN](#) {1.1GB}

Ubuntu 14.04*

[DEB](#) {10KB}

[DEB](#) {902MB}

[RUN](#) {1.1GB}

Ubuntu 12.04

[DEB](#) {3KB}

[DEB](#) {1.3GB}

[RUN](#) {1.1GB}

GPU Deployment Kit

Included in Installer

Included in Installer

[RUN](#) {4MB}

cuFFT Patch

[TAR](#) {122MB} , [README](#)

[Linux Getting Started Guide](#)

[Windows](#)

[Linux x86](#)

[Linux POWER8](#)

[Mac OSX](#)

Version

Network Installer

Local Package Installer

Runfile Installer

Ubuntu 14.10

[DEB](#) {3KB}

[DEB](#) {588MB}

[RUN](#) {1.7MB}

Ubuntu 14.04

[DEB](#) {3KB}

[DEB](#) {588MB}

[RUN](#) {1.7MB}

GPU Deployment Kit

n/a

n/a

[RUN](#) {1.7MB}

cuFFT Patch

[TAR](#) {105MB} , [README](#)

[Windows](#)

[Linux x86](#)

[Linux POWER8](#)

[Mac OSX](#)

Version

Network Installer

Local Installer

10.9

[DMG](#) {0.4MB}

[PKG](#) {977MB}

10.10

cuFFT Patch

[TAR](#) {104MB} , [README](#)



Resources

Materials and links especially for GPU Computing professionals and developers

Downloads

- CUDA Toolkit
- CUDA Downloads
- CUDA Archives
- CUDA Developer Home
- CUDA Developer Sign Up

Docs and References

- Online Documentation
- Architecture References

Education & Training

- Training Materials
- GTC Express Webinars
- GTC Presentations
- Udacity -Free Courses
- Coursera

CUDA Powered Processors

- Tesla
- Quadro
- GeForce
- GRID
- Tegra

Community

- GPU Computing Forums
- Meetups in Your City
- Parallel Forall Blog
- YouTube
- Stackoverflow
- gpgpu.org
- gpucomputing.net

Keep Informed

- Twitter
- Facebook
- CUDA Newsletter
- Parallel Forall – RSS feed

Partners & Ecosystem

- Tools & Ecosystem
- Accelerated Libraries
- Programming Language

Success Stories

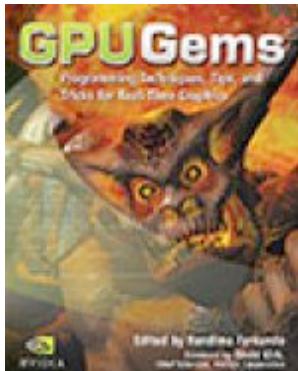
- Industry Domains
- Industry Applications
- Industry Success Stories
- CUDA Spotlights

Contact Us

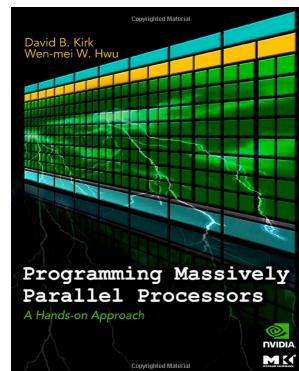
- Contact Form
- Submit Bugs
- View Your Submitted Bugs
- Forums
- Academic Programs

Libros sobre CUDA: Desde 2007 hasta 2015

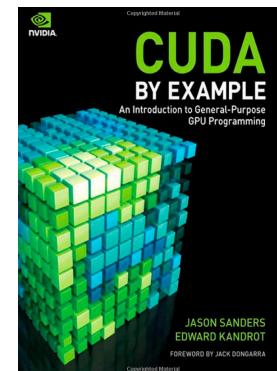
- La serie GPU Gems: 1, 2 y 3 [developer.nvidia.com/gpugems]
- Una lista de libros de CUDA [developer.nvidia.com/suggested-reading]



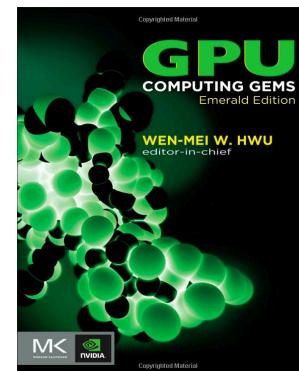
Sep'07



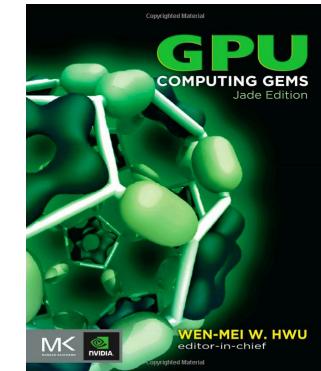
Feb'10



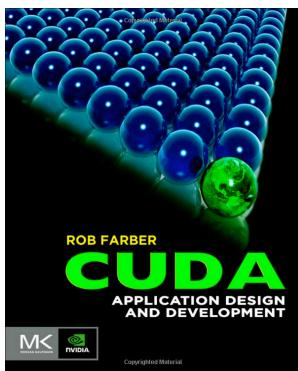
Jul'10



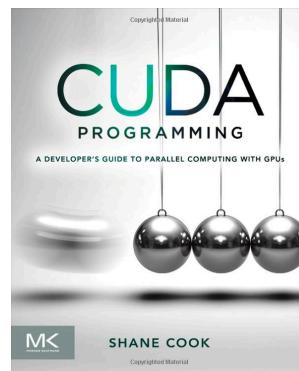
Abr'11



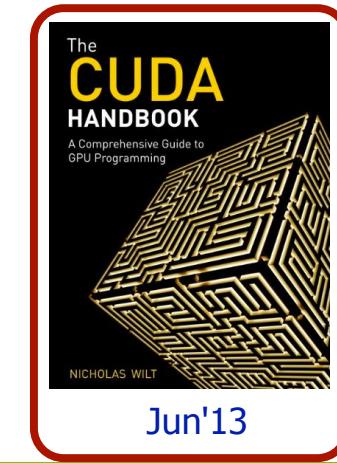
Oct'11



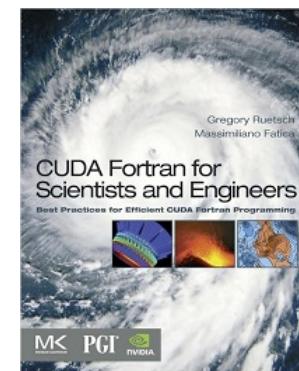
Nov'11



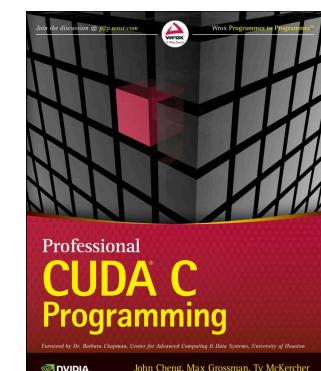
Dic'12



Jun'13



Oct'13



Sep'14

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).

Opciones para acelerar tus aplicaciones en CUDA y material para enseñar CUDA

● [developer.nvidia.com/cuda-education-training] (accesible también desde CUDA Zone -> Resources -> Training materials)

CUDA Education & Training

Accelerate Your Applications

Learn using step-by-step instructions, video tutorials and code samples.

- Accelerated Computing with C/C++
- Accelerate Applications on GPUs with OpenACC Directives
- Accelerated Numerical Analysis Tools with GPUs
- Drop-in Acceleration on GPUs with Libraries
- GPU Accelerated Computing with Python

QUICKLINKS

- | |
|-------------------------------------|
| Downloads |
| CUDA GPUs |
| NVIDIA Nsight Visual Studio Edition |
| Get Started - Parallel Computing |
| Tools & Ecosystem |
| CUDA FAQ |

Tweets by @GPUComputing  Follow

Teaching Resources

Get the latest educational slides, hands-on exercises and access to GPUs for your parallel programming courses.

- Parallel Programming Training Materials
- NVIDIA Research & Academic Programs

Sign up to join the Accelerated Computing Educators Network. This network seeks to provide a collaborative area for those looking to educate others on massively parallel programming. Receive updates on new educational material, access to CUDA Cloud Training Platforms, special events for educators, and an educators focused news letter.

[Sign-up Today!](#)

Cursos on-line de acceso gratuito

● Realizados por más de 50.000 programadores de 127 países en los últimos 6 meses. Impartidos por reputados pedagogos:

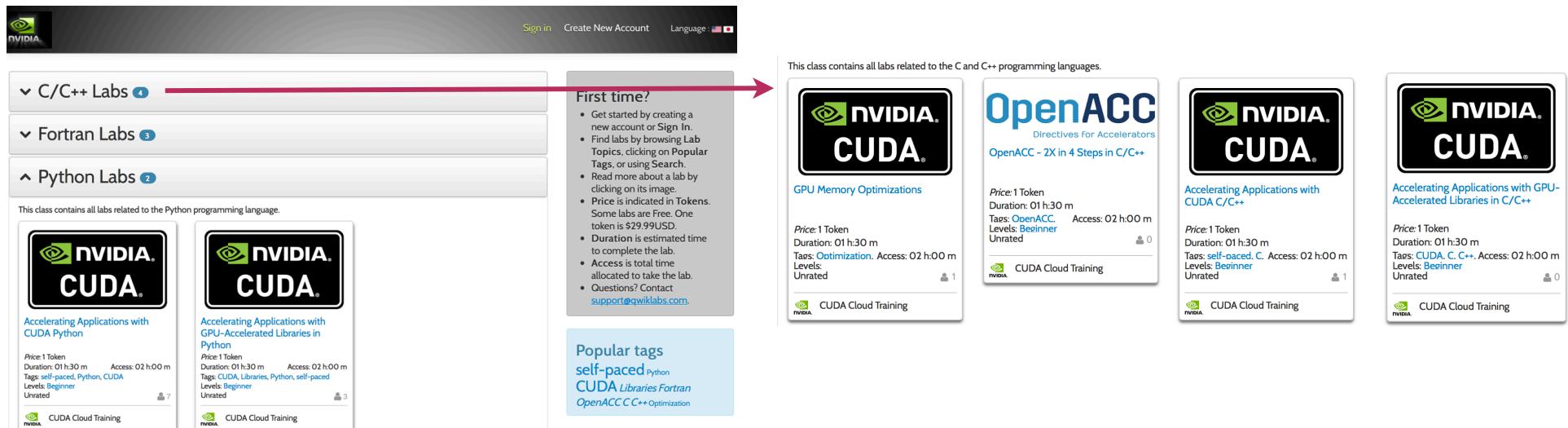
- Prof. Wen-Mei Hwu (Univ. of Illinois).
- Prof. John Owens (Univ. of California at Davis).
- Dr. David Luebke (Nvidia Research).

● Hay dos opciones, igualmente recomendables:

- Introducción a la programación paralela:
 - 7 unidades de 3 horas = 21 horas de esfuerzo.
 - Proporciona GPUs de gama alta para realizar los ejercicios.
 - [<https://developer.nvidia.com/udacity-cs344-intro-parallel-programming>]
- Programación paralela heterogénea (en UIUC): 
 - 9 semanas, cada una con teoría (vídeos de 20 minutos), desafíos o ejercicios
 - [<https://www.coursera.org/course/hetero>]

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: [\[nvidia.qwiklab.com\]](https://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's a navigation bar with the NVIDIA logo, 'Sign in', 'Create New Account', and language selection. The main menu on the left includes 'C/C++ Labs' (selected), 'Fortran Labs', and 'Python Labs'. A red arrow points from the 'First time?' section to the 'CUDA' lab card.

First time?

- Get started by creating a new account or Sign In.
- Find labs by browsing Lab Topics, clicking on Popular Tags, or using Search.
- Read more about a lab by clicking on its image.
- Price is indicated in Tokens. Some labs are Free. One token is \$29.99USD.
- Duration is estimated time to complete the lab.
- Access is total time allocated to take the lab.
- Questions? Contact support@qwiklabs.com.

Popular tags

self-paced Python CUDA Libraries Python self-paced CUDA Libraries Fortran OpenACC C/C++ Optimization

CUDA Labs

- GPU Memory Optimizations**: Price: 1 Token, Duration: 01 h:30 m, Tags: Optimization, Access: 02 h:00 m, Levels: Beginner, Unrated
- 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
- 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
- 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

Python Labs

- Accelerating Applications with CUDA Python**: Price: 1 Token, Duration: 01 h:30 m, Access: 02 h:00 m, Tags: self-paced, Python, CUDA, Levels: Beginner, Unrated
- Accelerating Applications with GPU-Accelerated Libraries in Python**: Price: 1 Token, Duration: 01 h:30 m, Access: 02 h:00 m, Tags: CUDA Libraries, Python, self-paced, Levels: Beginner, Unrated

Charlas y webinarios

- Charlas grabadas @ GTC (Graphics Technology Conference):
 - 383 charlas de la edición de 2013.
 - Más de 500 charlas de la edición de 2014 y 2015.
 - [\[www.gputechconf.com/gtcnew/on-demand-gtc.php\]](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)
- CUDACasts:
 - [\[devblogs.nvidia.com/parallelforall/category/cudacasts\]](http://devblogs.nvidia.com/parallelforall/category/cudacasts)

Desarrolladores

- Para firmar como desarrollador registrado:

- [\[www.nvidia.com/paralleldeveloper\]](http://www.nvidia.com/paralleldeveloper)

- Acceso a las descargas exclusivas para desarrolladores.

- Acceso exclusivo a las versiones más avanzadas de CUDA.

- Actividades exclusivas y ofertas especiales.

- Sitio de encuentro con otros muchos desarrolladores:

- [\[www.gpucomputing.net\]](http://www.gpucomputing.net)

- Noticias y eventos de GPGPU:

- [\[www.gpgpu.org\]](http://www.gpgpu.org)

- Lanzar cuestiones técnicas o preguntar dudas on-line:

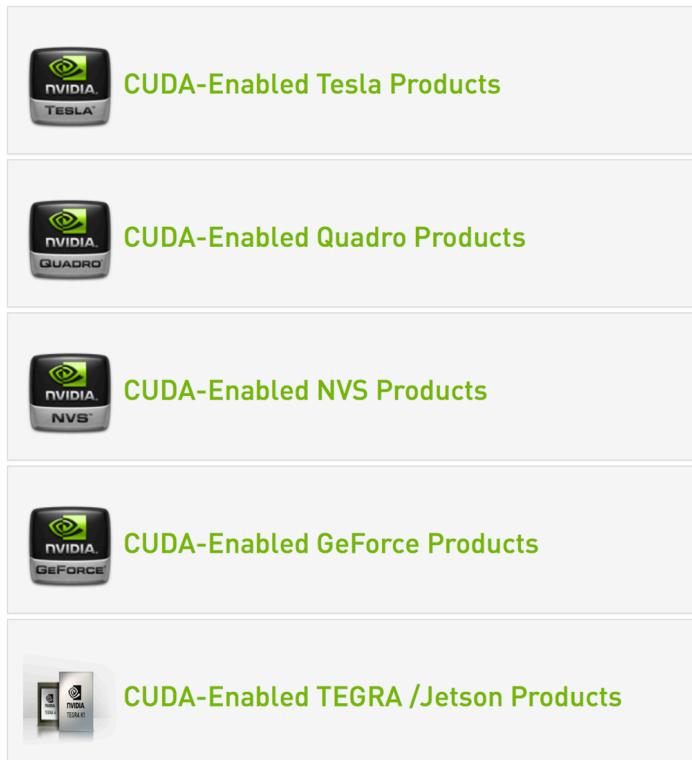
- NVIDIA Developer Forums: [\[devtalk.nvidia.com\]](http://devtalk.nvidia.com)

- Busca respuestas o suscribe preguntas: [\[stackoverflow.com/tags/cuda\]](http://stackoverflow.com/tags/cuda)₁₈₂

Desarrolladores (2)

- Listado oficial de GPUs que soportan CUDA:

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



- Y una última herramienta:
El CUDA Occupancy Calculator

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

Tendencias futuras

- El blog de Nvidia, “Parallel for all” contiene los artículos más jugosos y fiables sobre todo lo nuevo que está apareciendo en torno a CUDA (conviene suscribirse):
 - [\[devblogs.nvidia.com/parallelforall\]](http://devblogs.nvidia.com/parallelforall)
- Algunos artículos especialmente interesantes:
 - “Getting Started with OpenACC”, por Jeff Larkin.
 - “New Features in CUDA 7.5”, por Mark Harris.
 - “CUDA Dynamic Parallelism API and Principles”, por Andrew Adinetz.
 - “NVLINK, Pascal and Stacked Memory: Feeding the Appetite for Big Data”, por Denis Foley.
 - “CUDA Pro Tip: Increase Application Performance with NVIDIA GPU Boost”, por Mark Harris.

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).

● O de forma más específica sobre GPUs,
visita mi página Web como CUDA Fellow:

● <http://research.nvidia.com/users/manuel-ujaldon>

