

Programación de GPUs con CUDA

Curso de Extensión Universitaria

Titulaciones Propias. Universidad de Málaga. Curso 2016/17



Manuel Ujaldón

Catedrático de Arquitectura de Computadores @ Universidad de Málaga
CUDA Fellow @ Nvidia Corporation



Agradecimientos

● Al personal de Nvidia, por compartir conmigo ideas, material, diagramas, presentaciones, ... Alfabéticamente:

- Bill Dally [2010-2011: Consumo energético, Echelon y diseños futuros].
- Simon Green [2007-2009: Los pilares de CUDA].
- Sumit Gupta [2008-2009: El hardware de Tesla].
- Mark Harris [2008, 2012: CUDA, OpenACC, Lenguajes de Programación, Librerías].
- Wen-Mei Hwu [2009: Programación y trucos para mejorar el rendimiento].
- Stephen Jones [2012: Kepler].
- David B. Kirk [2008-2009: Hardware de Nvidia].
- Timothy Lanfear [2012: Kepler y OpenACC].
- David Luebke [2007-2008: Hardware de Nvidia].
- Lars Nyland [2012: Kepler].
- Edmondo Orlotti [2012: CUDA 5.0, OpenACC].
- Cyril Zeller [2008-2009: Fundamentos de CUDA].

● ... y a Mercedes Caravaca por su trabajo en las páginas Web y en los tutoriales de la parte práctica.

Manuel Ujaldon - Nvidia CUDA Fellow

Contenidos [198 diapositivas]



1. Introducción. [27 diapositivas]

2. Arquitectura. [77]

- 1. El modelo hardware de CUDA. [4]
- 2. Primera generación: Tesla (2007-2009). [3]
- 3. Segunda generación: Fermi (2010-2011). [4]
- 4. Tercera generación: Kepler (2012-2015). [4+18]
- 5. Cuarta generación: Maxwell (2015-2016). [7]
- 6. Quinta generación: Pascal (2016-17). [20]
- 7. Sexta generación: Volta (2018-?). [15]
- 8. Síntesis generacional. [2]

3. Programación. [20]

4. Sintaxis. [17]

- 1. Elementos básicos. [11]
- 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]

3

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.

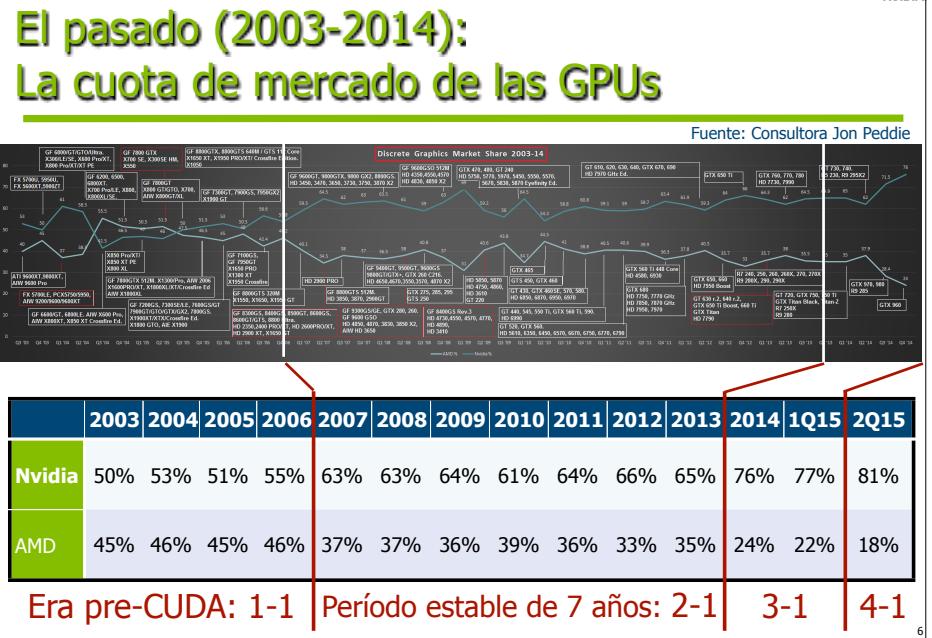
● No es necesario tener experiencia en programación gráfica. Eso era antes cuando GPGPU se basaba en los shaders y Cg. Con CUDA, no hace falta desenvolverse con vértices, píxeles o texturas porque es transparente a todos estos recursos.

Manuel Ujaldon - Nvidia CUDA Fellow

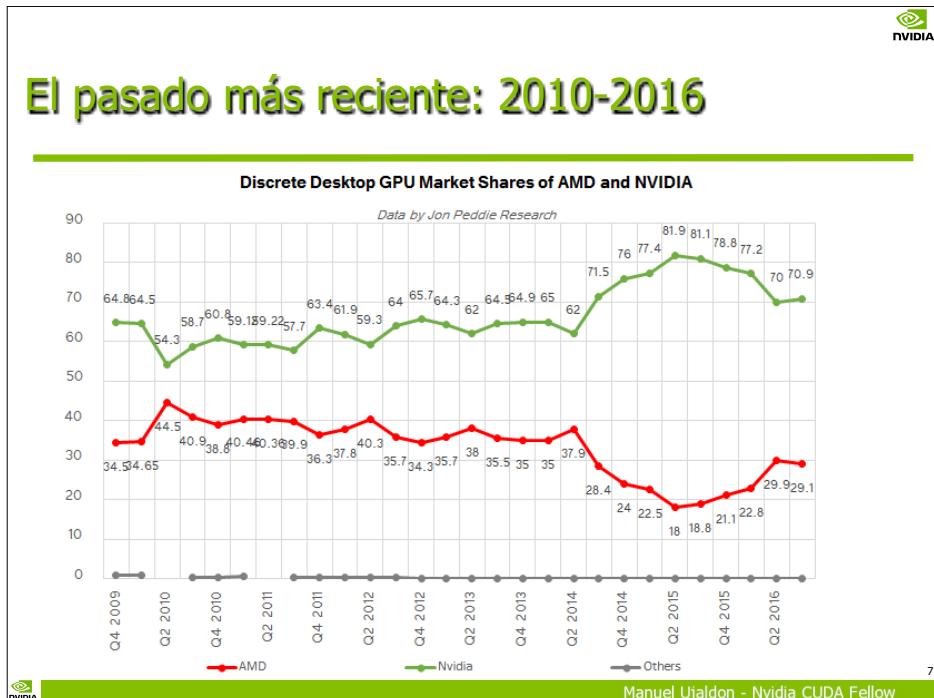
4



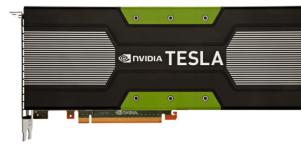
I. Introducción



Manuel Ujaldon - Nvidia CUDA Fellow



Modelos comerciales: GeForce y Tesla frente a frente



Diseñada para jugar:

- El precio es prioritario (<500€).
- Gran disponibilidad/popularidad.
- Poca memoria de vídeo (1-2 GB.).
- Relojes un poco más rápidos.
- Perfecta para desarrollar código que luego pueda disfrutar Tesla.

Orientada a HPC:

- Fiabilidad (tres años de garantía).
- Pensada para conectar en clusters.
- Más memoria de vídeo (6-12 GB.).
- Ejecución sin descanso (24/7).
- GPUDirect (RDMA) y otras coberturas para clusters de GPUs.

9

Manuel Ujaldon - Nvidia CUDA Fellow

10

Manuel Ujaldon - Nvidia CUDA Fellow

10

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	VSIP 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

Manuel Ujaldon - Nvidia CUDA Fellow

Distribución mundial de las 840 universidades que imparten cursos de CUDA



12

Manuel Ujaldon - Nvidia CUDA Fellow

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 2016

600.000.000 GPUs aceptan CUDA
(y 450.000 son Teslas)

3.000.000 descargas de CUDA al año
(esto es, una cada 9 segundos)

63 supercomputadores
en el TOP500.org
Acumulado: 80.000 TFLOPS
(más del 14% de los 567 PFLOPS del top500)

840 cursos universitarios

60.000 artículos científicos

11

Manuel Ujaldon - Nvidia CUDA Fellow

Un resumen de las más de 100 actividades que he desempeñado como CUDA Fellow



País	2012	2013	2014	2015	2016	Total
España	10	3	12	6	5	36
Chile	2	2	2		6	12
Argentina		3	2		2	7
Sudáfrica		4	1	1		6
Brasil		2	1	3		6
Australia	2	1		2		5
Nueva Zelanda	2			3		5
Italia		1		3		4
Portugal				4		4
Francia	1	1			1	3
Panamá					3	3
Uruguay	2					2
Austria	1	1				2
Polonia				2		2
Colombia				2		2
China				2		2
Suecia		1				1
Finlandia		1				1
Perú			1			1
19 países	20	20	19	26	19	104

- ➊ Destaca España con 36 cursos y tutoriales.
- ➋ Le siguen un amplio grupo de países del hemisferio sur: Chile, Argentina, Sudáfrica, Brasil, Australia y Nueva Zelanda.
- ➌ El curso más extenso de todos es el que impartimos en Málaga.

13

Manuel Ujaldon - Nvidia CUDA Fellow

15

14

Manuel Ujaldon - Nvidia CUDA Fellow

El equipo de CUDA Fellows

	Takayuki Aoki Tokyo Tech Awarded: 2012		Lorena Barba George Washington University Awarded: 2012		Massimo Bernaschi National Research Council of Italy Awarded: 2012
	Rich Brower Boston University Awarded: 2014		Esteban Clua Universidade Federal Fluminense Awarded: 2015		Mike Giles Oxford University Awarded: 2009
	Alan Gray The University of Edinburgh Awarded: 2014		Bormin Huang University of Wisconsin, Madison Awarded: 2014		Scott Le Grand Amazon Web Services Awarded: 2011
	Dan Negru University of Wisconsin, Madison Awarded: 2010		John Owens University of California at Davis Awarded: 2012		John Stone University of Illinois, Urbana-Champaign Awarded: 2010
	Manuel Ujaldon University of Malaga Awarded: 2012		Ross Walker San Diego Supercomputer Center Awarded: 2010		Bin Zhou University Science & Technology of China Awarded: 2013

Manuel Ujaldon - Nvidia CUDA Fellow

Síntesis evolutiva de la GPU



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

15

Manuel Ujaldon - Nvidia CUDA Fellow

16

Manuel Ujaldon - Nvidia CUDA Fellow

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

- ➊ Control simplificado.
 - ➊ El control de un hilo se amortiza en otros 31 (**warp size = 32**).
- ➋ Escalabilidad.
 - ➊ Aprovechándose del gran **volumen de datos** que manejan las aplicaciones, se define un modelo de paralelización sostenible.
- ➌ Productividad.
 - ➊ 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**.
- ➍ Palabras clave esenciales para CUDA:
 - ➊ Warp, SIMD, ocultación de latencia, comutación 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

Manuel Ujaldon - Nvidia CUDA Fellow



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):

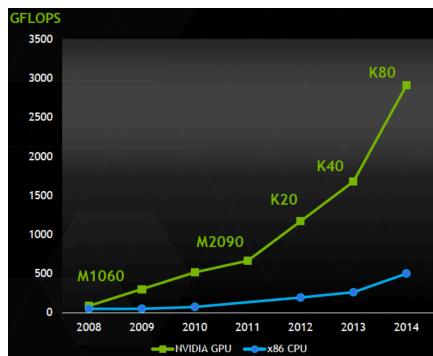
- Aumenta el ancho de banda (OK), pero también la latencia (ups).
- Solución: **Stacked-DRAM**, o cómo resolver por fin el *memory wall* aportando a la vez cantidad (Gbytes) y calidad (cercanía/velocidad).

Manuel Ujaldon - Nvidia CUDA Fellow

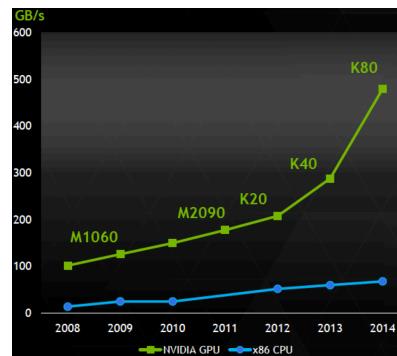


Rendimiento pico frente a la CPU

GFLOPS pico (fp64)



Ancho de banda



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.

19



Manuel Ujaldon - Nvidia CUDA Fellow

19

¿Qué es CUDA?

"Compute Unified Device Architecture"

- Una plataforma diseñada conjuntamente a nivel software y hardware para aprovecharla 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.

20



Manuel Ujaldon - Nvidia CUDA Fellow

Lo esencial de CUDA C

- En general, es lenguaje C con mínimas extensiones:

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

- CUDA define:

- Un modelo de arquitectura:

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

- Un modelo de programación:

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

- Un modelo de gestión de la memoria:

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

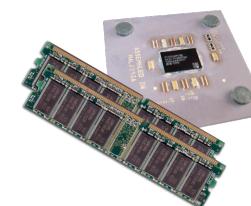
- Objetivos:

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

Computación heterogénea (1/4)

- Terminología:

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



Host



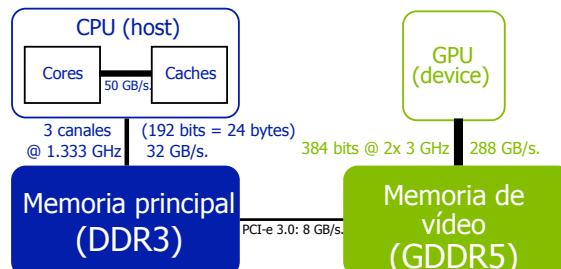
Device

Computación heterogénea (2/4)

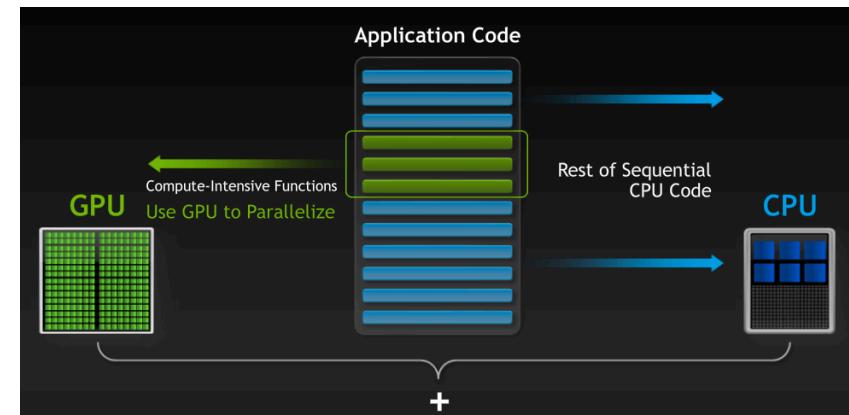
- CUDA ejecuta un programa sobre un dispositivo (la GPU), que actúa como coprocesador de un anfitrión o host (la CPU).

- CUDA puede verse como una librería de funciones que contienen 3 tipos de componentes:

- Host: Control y acceso a los dispositivos.
- Dispositivos: Funciones específicas para ellos.
- Todos: Tipos de datos vectoriales y un conjunto de rutinas soportadas por ambas partes.



Computación heterogénea (3/4)



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

Computación heterogénea (4/4)

```
#include <math.h>
#include <cuda.h>
using namespace std;
#define N 1024
#define RADIUS 3
#define BLOCK_SIZE 16
__global__ void stencil_3d(int *a, int *b) {
    const int i = blockIdx.x * blockDim.x + threadIdx.x;
    const int j = blockIdx.y * blockDim.y + threadIdx.y;
    const int k = blockIdx.z * blockDim.z + threadIdx.z;

    // Read local elements into shared memory
    temp0[i] = a[i];
    temp1[j] = a[j];
    temp2[k] = a[k];
    if (temp0[i] > RADIUS) temp0[i] = a[i - RADIUS];
    if (temp1[j] > RADIUS) temp1[j] = a[j - RADIUS];
    if (temp2[k] > RADIUS) temp2[k] = a[k - RADIUS];

    // Summarize (most of the data is available)
    __syncthreads();
    if (temp0[i] > RADIUS) temp0[i] = a[i + RADIUS];
    if (temp1[j] > RADIUS) temp1[j] = a[j + RADIUS];
    if (temp2[k] > RADIUS) temp2[k] = a[k + RADIUS];

    // Apply the stencil
    int offset = i + RADIUS * N + j * N + k;
    for (int n = -RADIUS; n <= RADIUS; n++) {
        if (n != 0) {
            if (temp0[i + n] > RADIUS) temp0[i + n] = a[i + n];
            if (temp1[j + n] > RADIUS) temp1[j + n] = a[j + n];
            if (temp2[k + n] > RADIUS) temp2[k + n] = a[k + n];
        }
    }

    // Store the result
    out[threadIdx.z * N * N + i * N + j] = result;
}

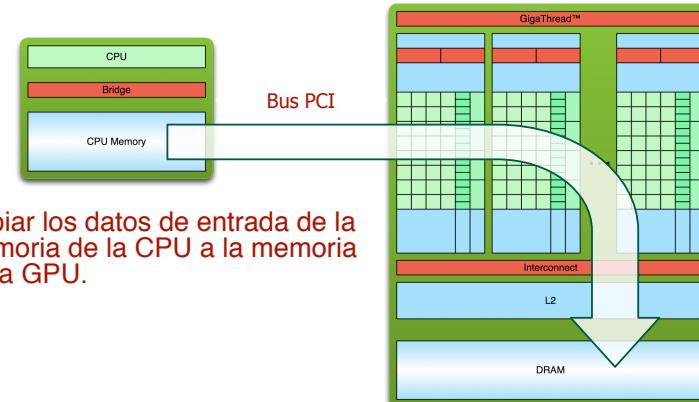
void main() {
    int *a, *b;
    ...
}
```

CODIGO DEL DISPOSITIVO:
Función paralela escrita en CUDA.

CODIGO DEL HOST:

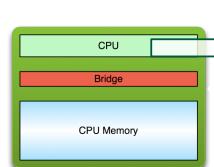
- Código serie.
- Código paralelo.
- Código serie.

Un sencillo flujo de procesamiento (1/3)

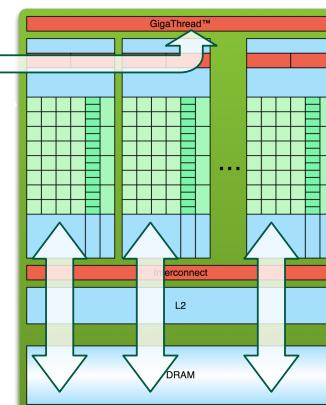


1. Copiar los datos de entrada de la memoria de la CPU a la memoria de la GPU.

Un sencillo flujo de procesamiento (2/3)

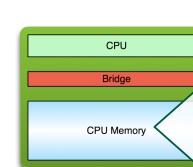


PCI Bus

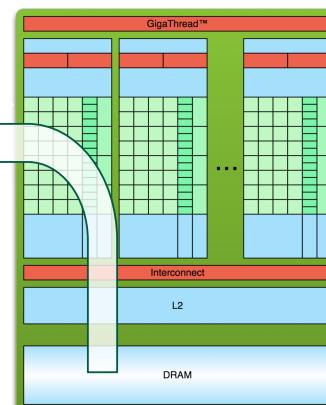


1. Copiar los datos de entrada de la memoria de la CPU a la memoria de la GPU.
2. Cargar el programa en la GPU y ejecutarlo, ubicando datos en caché para mejorar el rendimiento.

Un sencillo flujo de procesamiento (3/3)



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.
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.

29

Manuel Ujaldon - Nvidia CUDA Fellow

30

Manuel Ujaldon - Nvidia CUDA Fellow

¡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).

31

32

33

34

35

36

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

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

- Dos nuevos elementos sintácticos:

- 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>>>**.

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

- nvcc separa el código fuente para la CPU y la GPU.
- Las funciones que corresponden a la GPU (como **mikernel()**) son procesadas por el compilador de Nvidia.
- Las funciones de la CPU (como **main()**) son procesadas por su compilador (**gcc** para Unix, **c1.exe** para Windows).

30

31

32

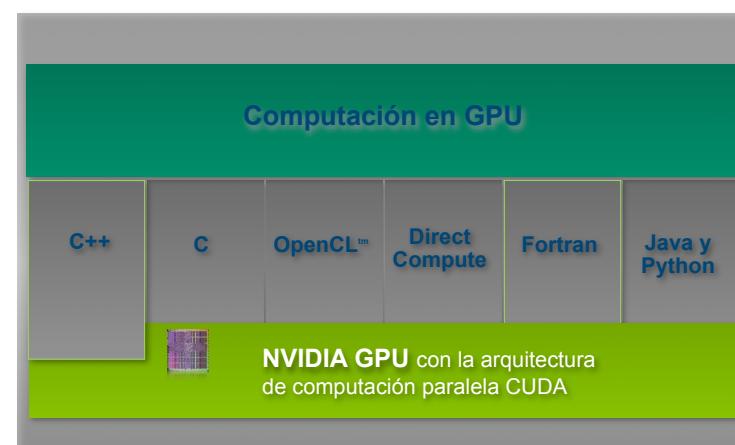
33

34

35

36

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



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

32

33

34

35

36

31

32

33

34

35

36

31

32

33

34

35

36

31

32

33

34

35

36



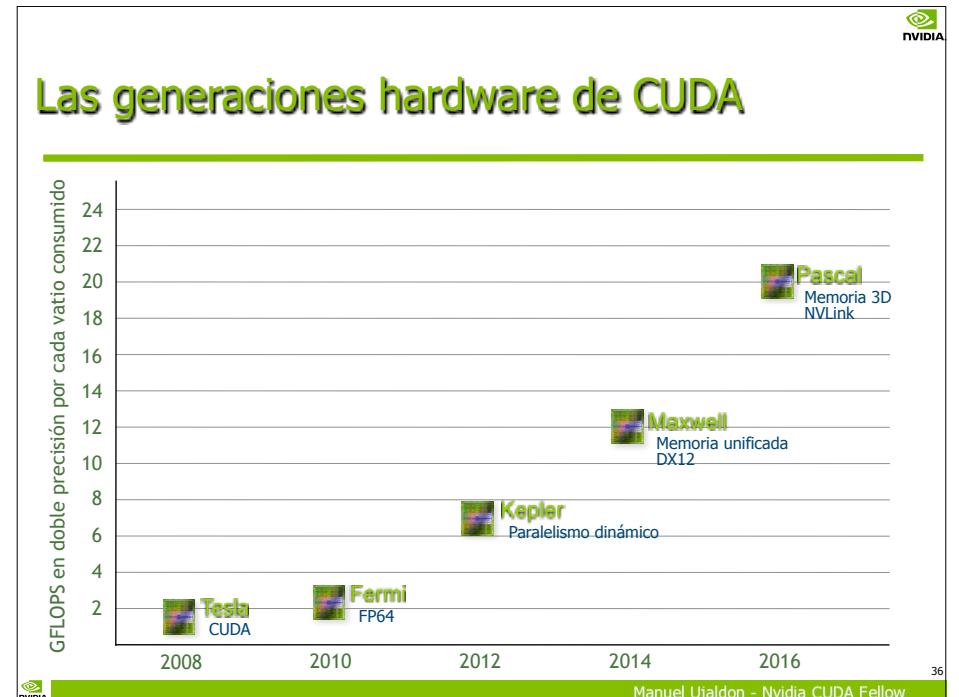
NVIDIA

“...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

34

Manuel Ujaldon - Nvidia CUDA Fellow



El modelo hardware de CUDA: Un conjunto de procesadores SIMD

• La GPU consta de:

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



• Computación heterogénea:

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

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

37

Manuel Ujaldon - Nvidia CUDA Fellow

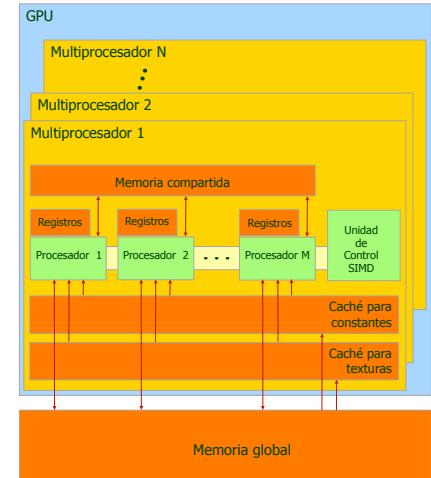
Jerarquía de memoria

• 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).



38

Manuel Ujaldon - Nvidia CUDA Fellow

Latencia y ancho de banda de la memoria en CUDA

• Memoria de la CPU (o memoria principal - DDR3):

- Ancho de banda con mem. vídeo: 3.2 GB/s.(PCIe) y 5.2 GB/s(PCIE2).

• Memoria de vídeo (o memoria global):

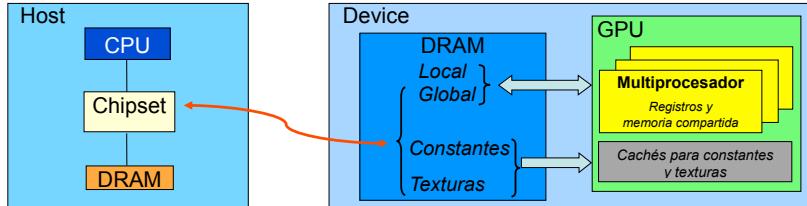
- Gran ancho de banda (80-100 GB/s) y latencia, no pasa por caché.

• Memoria compartida

- Baja latencia, ancho de banda muy elevado, tamaño reducido.
- Actúa como una caché gestionada por el usuario (scratchpad).

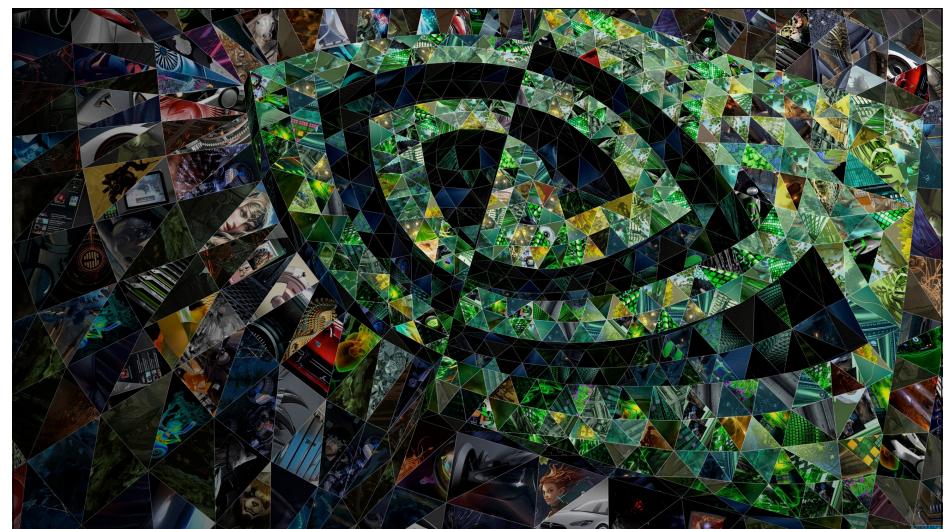
• Memoria de texturas/constantes

- De sólo lectura, alta/baja latencia, pasa por caché.



39

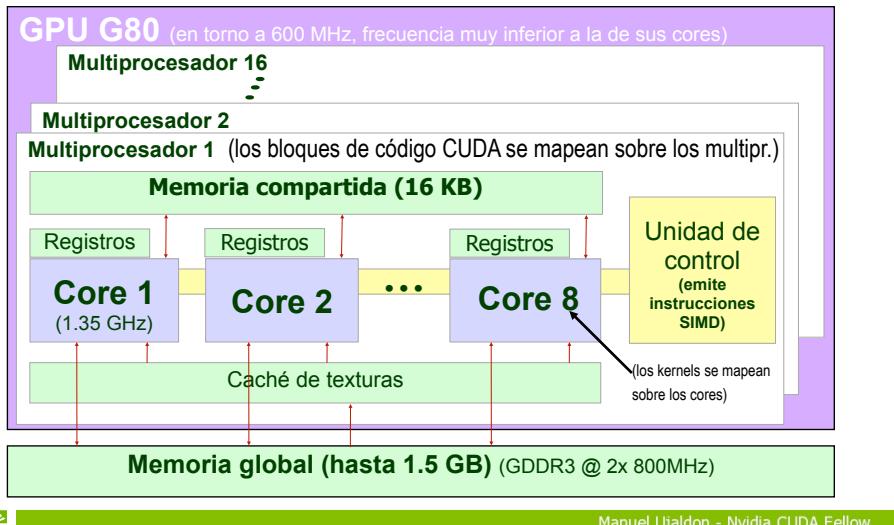
Manuel Ujaldon - Nvidia CUDA Fellow



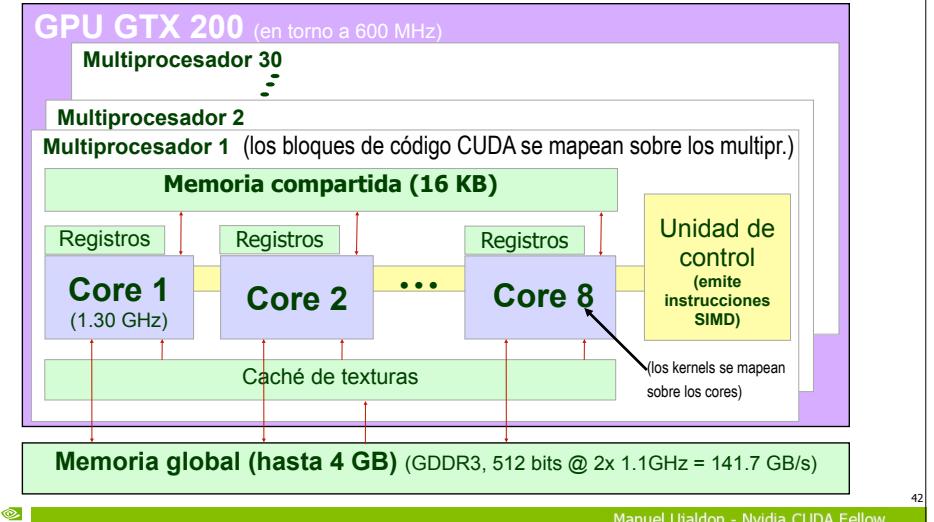
II.2. La primera generación: Tesla (G80 y GT200)



La primera generación: G80 (GeForce 8800)

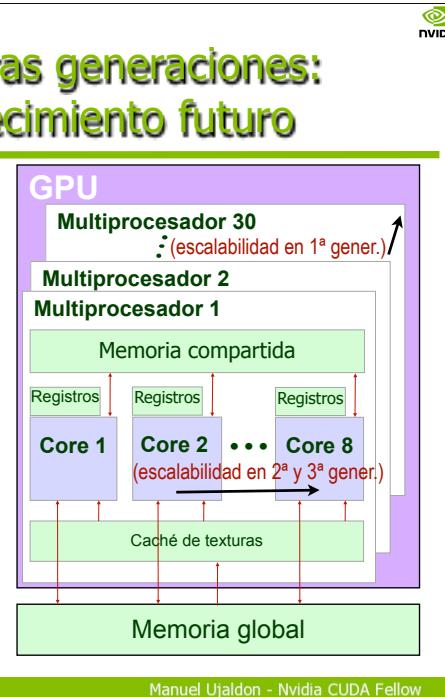


La primera generación: GT200 (GTX 200)



Escalabilidad para futuras generaciones: Alternativas para su crecimiento futuro

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



II. 3. 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	(o viceversa)
Caché L2	Ninguna	Ninguna	768 KB
Corrección de errores (DRAM)	No	No	Sí
Anchura del bus de direcciones	32 bits	32 bits	64 bits

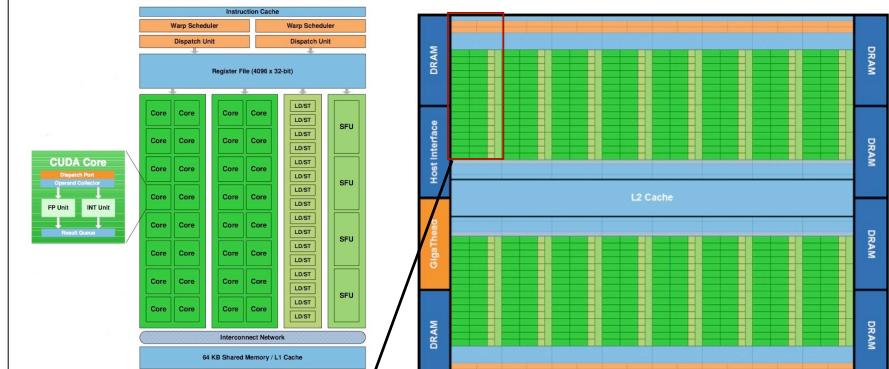
45

Manuel Ujaldon - Nvidia CUDA Fellow



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.



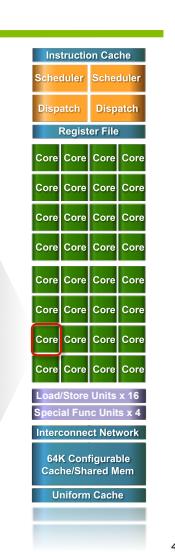
46

Manuel Ujaldon - Nvidia CUDA Fellow

Mejoras en la aritmética

Unidades aritmético-lógicas (ALUs):

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



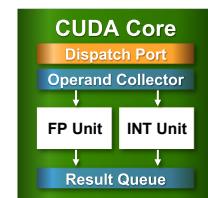
47

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.



47

48

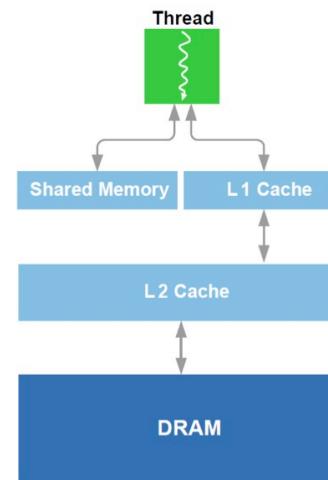
Manuel Ujaldon - Nvidia CUDA Fellow



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.



48

Manuel Ujaldon - Nvidia CUDA Fellow





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



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, ...)

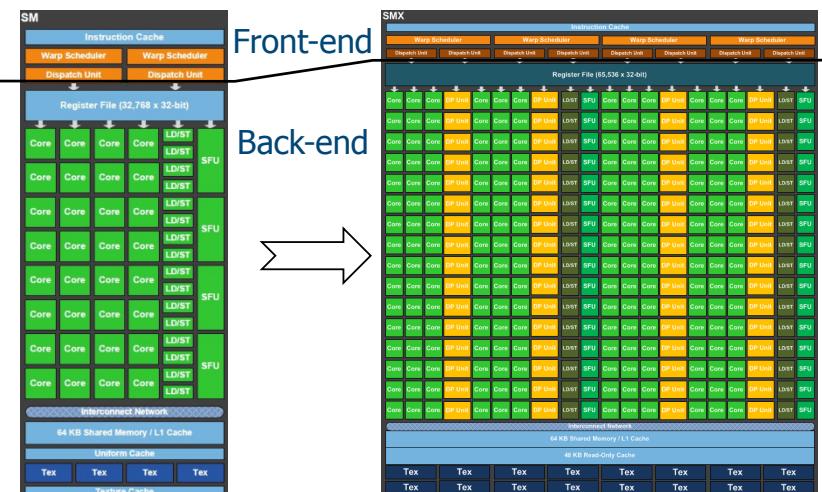
Acceso a memoria

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.



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

- 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.

53

Manuel Ujaldon - Nvidia CUDA Fellow

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:



Manuel Ujaldon - Nvidia CUDA Fellow

54

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>

55

Manuel Ujaldon - Nvidia CUDA Fellow

56

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

55

Manuel Ujaldon - Nvidia CUDA Fellow

Lecciones a aprender (y conflictos asociados)

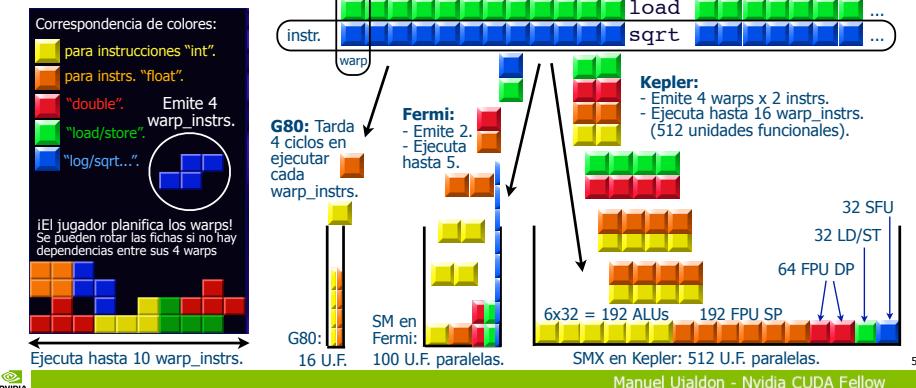
- Bloques suficientemente grandes para evitar el límite de 16 por cada SMX.
- Pero los bloques consumen memoria compartida, y alojar más memoria compartida significa menos bloques y más hilos por bloque.
- Suficientes hilos por bloque como para saturar el límite de 64 warps activos por cada SMX.
 - Pero 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.

57

Manuel Ujaldon - Nvidia CUDA Fellow

Expresando todo el paralelismo

- Tetris (baldosa = warp_instr.):
- Emite 4 warp_instrs.
 - Ejecuta hasta 10 warps = 320 hilos.
 - Warp_instrs. son simétricos y se ejecutan todos en 1 ciclo.



58

Manuel Ujaldon - Nvidia CUDA Fellow

Paralelismo en SMX: A nivel de hilo (TLP) y a nivel de instrucción (ILP)

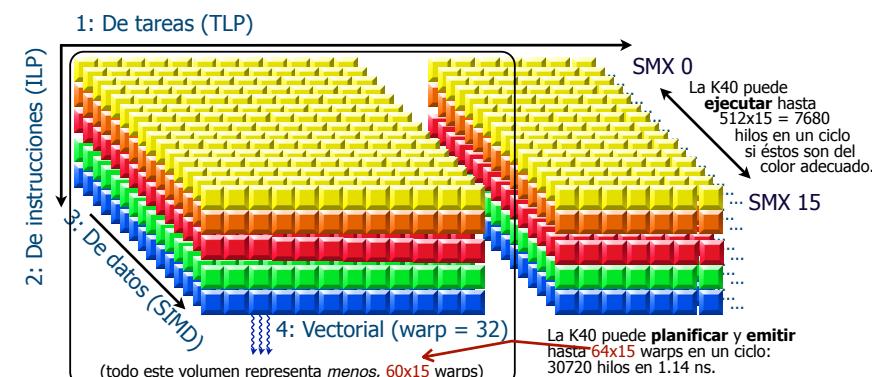


- Los SMX pueden potenciar el ILP disponible de forma intercambiable con el TLP:
 - Es mucho mejor que Fermi para esto.
 - Algunas veces es más fácil incrementar el ILP que el TLP (por ejemplo, desenrollar un lazo en un pequeño factor):
 - El número de hilos puede estar limitado por el algoritmo o límites hardware.
 - Necesitamos el ILP para lograr un elevado IPC (Instrs. Per Cycle).

59

Manuel Ujaldon - Nvidia CUDA Fellow

En las GPUs Kepler concurren todas las formas de paralelismo. Ejemplo: K40.

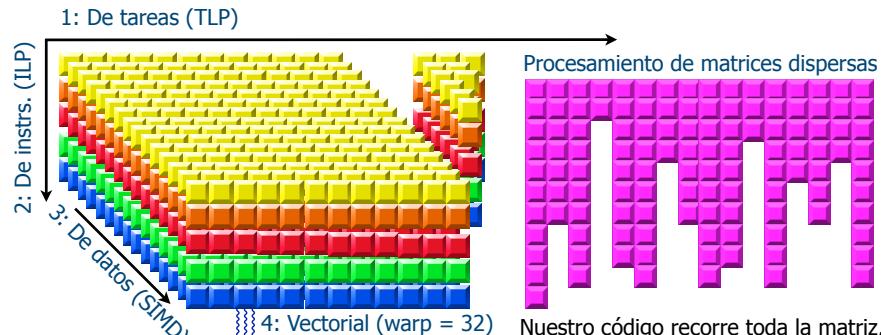


- Imagina un tetris 3D con 15 cubos y hasta 64 baldosas cayendo simultáneamente en cada uno de ellos, porque así funciona la K40 planificando warps con el máx. paralelismo.

60

Manuel Ujaldon - Nvidia CUDA Fellow

Una breve incursión en el concurso



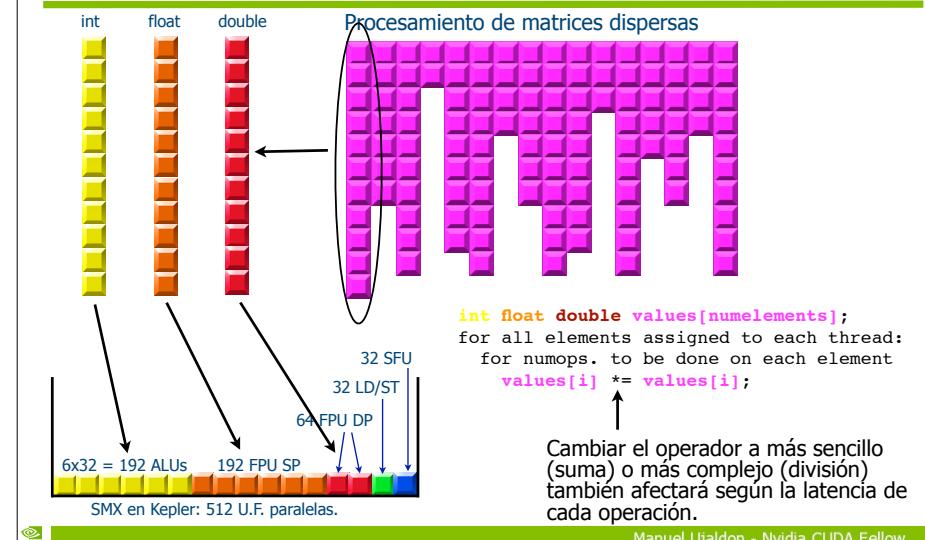
Estrategia base:

- Lanzaremos un kernel CUDA por cada columna de la matriz.
- Cada kernel tendrá el menor número posible de bloques.
- Cada bloque tendrá el mayor número posible de warps.

61

Manuel Ujaldon - Nvidia CUDA Fellow

Una breve incursión en el concurso (2)



Manuel Ujaldon - Nvidia CUDA Fellow

62

Caso estudio: Polinomios de Zernike

Recursos GPU	ALU	FPU 32 bits	FPU 64 bits	Carga/almacen.	SFU
Fermi	32%	32%	16%	16%	4%
Kepler	37.5%	37.5%	12.5%	6.25%	6.25%
Kernel de los polinomios de Zernike	54%	21%	0%	25%	0%
Mejor	Kepler	Fermi	Kepler	Fermi	Fermi

- Fermi se encuentra más equilibrada para este caso.
- La distribución de recursos en Kepler mejora la ejecución de la aritmética entera, pero empeora la de punto flotante y carga/almacenamiento. El resto no se utilizan.

63

Manuel Ujaldon - Nvidia CUDA Fellow

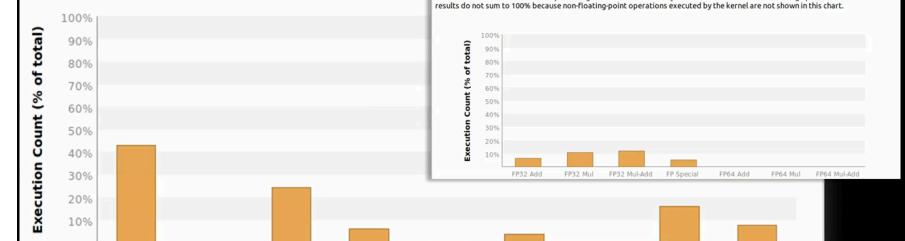
Utilizar el CUDA Visual Profiler para conocer qué tal se adapta nuestra aplicación

Detailed Instruction Mix Visualization

Visual Profiler and NSight EE

1 Instruction Execution Counts

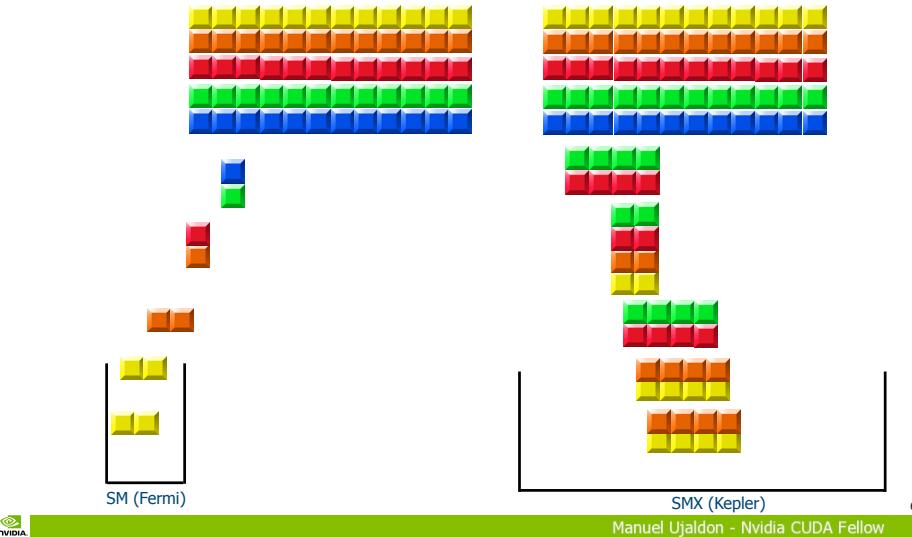
The following chart shows the mix of instructions executed by the kernel. The instructions are grouped into classes and for each class the chart shows the percentage of thread execution cycles that were devoted to executing instructions in that class. The "inactive" result shows the thread executions that did not execute any instruction because the thread was predicated or inactive due to divergence.



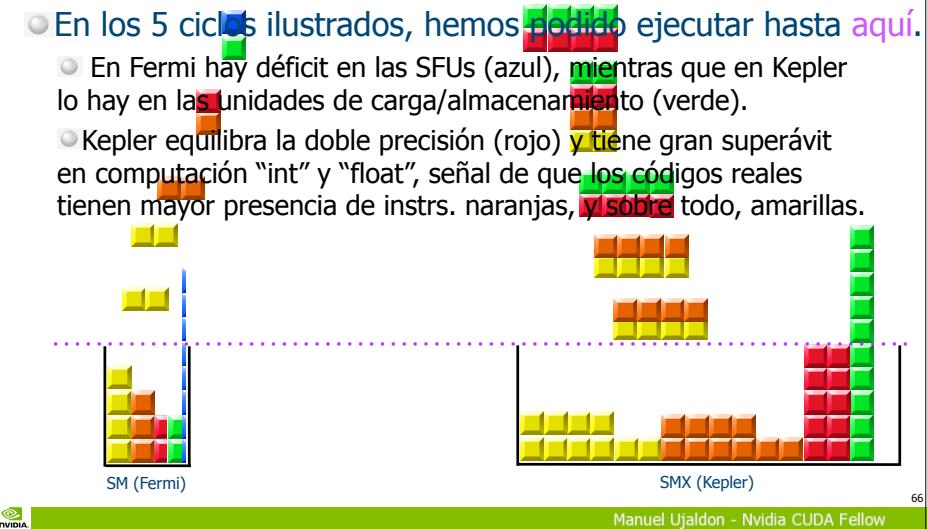
Manuel Ujaldon - Nvidia CUDA Fellow

64

Cómo trabaja el front-end de la GPU: (1) Planificación de warps

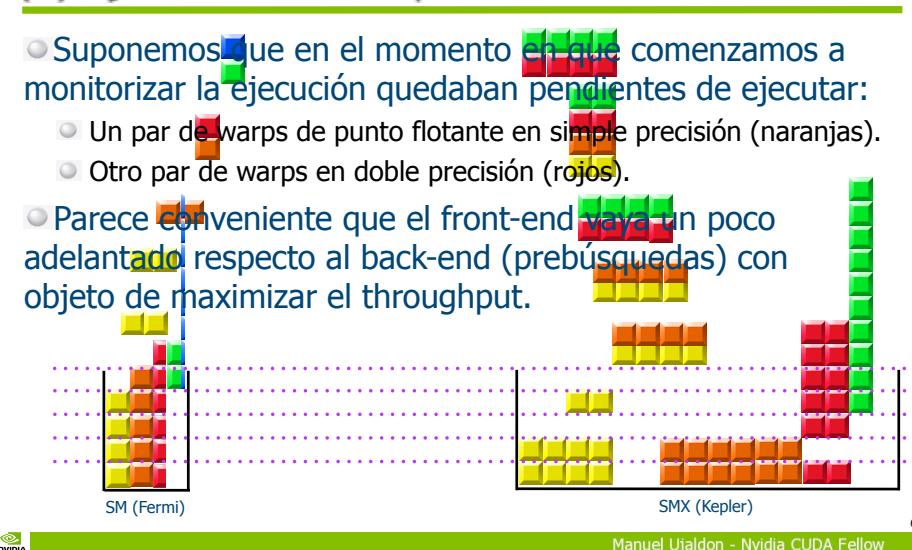


La conexión entre front-end y back-end: (2) Emisión de warps



Cómo trabaja el back-end de la GPU: (3) Ejecución de warps

- Suponemos que en el momento en que comenzamos a monitorizar la ejecución quedaban pendientes de ejecutar:
 - Un par de warps de punto flotante en simple precisión (naranjas).
 - Otro par de warps en doble precisión (rojos).
- Parece conveniente que el front-end vaya un poco adelantado respecto al back-end (prebúsquedas) con objeto de maximizar el throughput.



Puntuaciones al modelo "tetris"

- En Fermi, las baldosas rojas no pueden combinarse con otras.
- En Kepler, no se pueden tomar 8 warp_instrs. en horizontal, deben ser de altura 2 como mínimo.
- Las instrucciones tienen distinta latencia, así que las que consuman más de un ciclo (como los operandos en doble precisión) deben expandirse verticalmente en proporción.
- Si el warp tiene divergencias, tarda más de un ciclo: Es la carencia del vectorial. Se podría representar como el anterior.
- Los códigos tienen más fichas amarillas ("int" domina).
- Algunas fichas vienen "cojas" porque el planificador de warps no puede encontrar un lote de 4x2 sin dependencias.
- Las fichas pueden ensamblar baldosas no contiguas.

Latencia de los warps

• Aunque todas las baldosas tardaran un ciclo en ejecutarse, la duración de los warps no sería ésa. El tiempo que un warp gasta en la máquina es la suma de tres:

- Tiempo en planificación.
- Tiempo en emisión.
- Tiempo en ejecución.

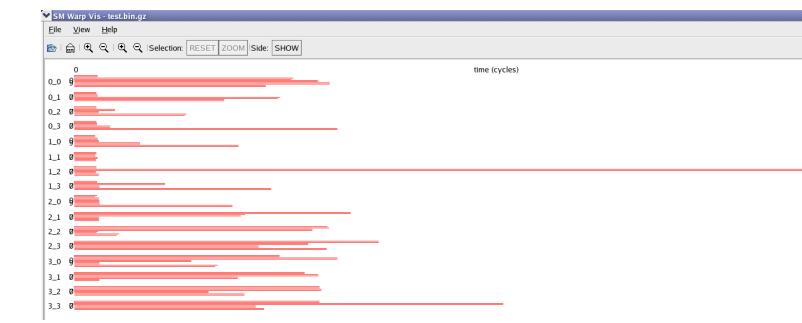
• La planificación y ejecución son bastante regulares, pero la emisión no: Depende de la montaña de baldosas que haya en el cubilete (estaciones de reserva). De ahí la varianza que experimenta su duración.

69

Manuel Ujaldon - Nvidia CUDA Fellow

El comportamiento de los warps nos enseña que la GPU no es un procesador regular

• Factores impredecibles en tiempo de ejecución dificultan un reparto equilibrado de la carga computacional entre los multiprocesadores. Aquí vemos un ejemplo de la varianza existente entre los 8 últimos warps ejecutados en cada SM:



70

Manuel Ujaldon - Nvidia CUDA Fellow

Mejora de recursos en los SMX

Recurso	Kepler GK110 frente a Fermi GF100
Ritmo de cálculo con oper. 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

71

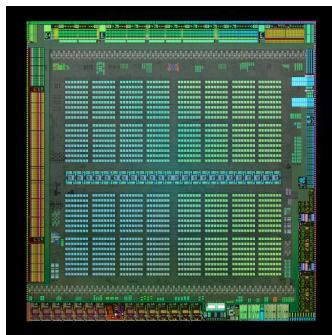
Manuel Ujaldon - Nvidia CUDA Fellow



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

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

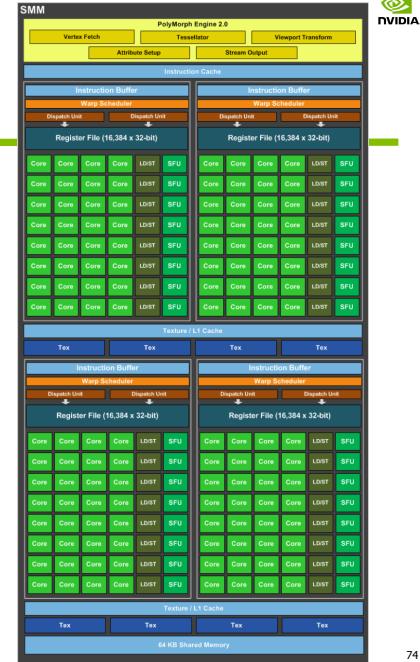
- 1870 Mt.
- 148 mm².



Manuel Ujaldon - Nvidia CUDA Fellow

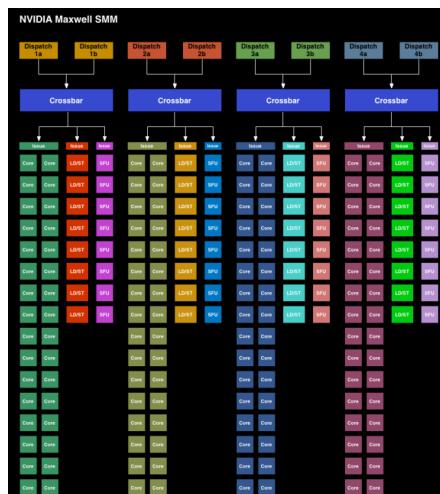
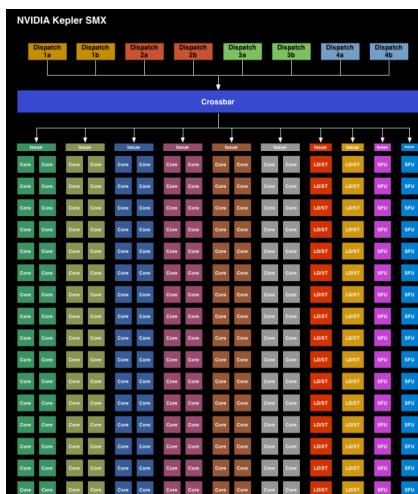
Detalle de los SMMs

- 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.



Manuel Ujaldon - Nvidia CUDA Fellow

Comparativa respecto a Kepler



Manuel Ujaldon - Nvidia CUDA Fellow

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/s.	86.4 GB/s.	80.2 GB/s.	144 GB/s.	86.4 GB/s.
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]

Manuel Ujaldon - Nvidia CUDA Fellow

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



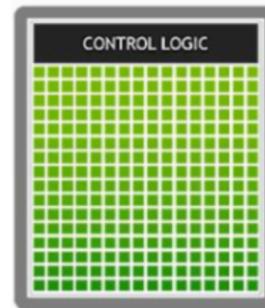
GeForce	GTX 950	GTX 960	GTX 970	GTX980	GTX 980 Ti	Titan X
Fecha de lanzamiento	Ago'15	Ago'15	Sept'14	Sept'14	Junio'15	Marzo'15
GPU (nombre del chip)	GM206-250	GM206-300	GM204-200	GM204-400	GM200-310	GM200-400
Multiprocesadores	6	8	13	16	22	24
Número de cores	768	1024	1664	2048	2816	3072
Frec. cores (MHz)	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
Conector 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



Manuel Ujaldon - Nvidia CUDA Fellow

Principales mejoras

KEPLER

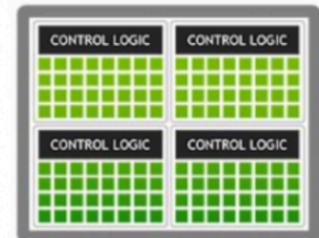


MAXWELL

1st Generation

135%
Performance/Core

2x
Performance/Watt



Manuel Ujaldon - Nvidia CUDA Fellow

Eficiencia del consumo



79

Manuel Ujaldon - Nvidia CUDA Fellow

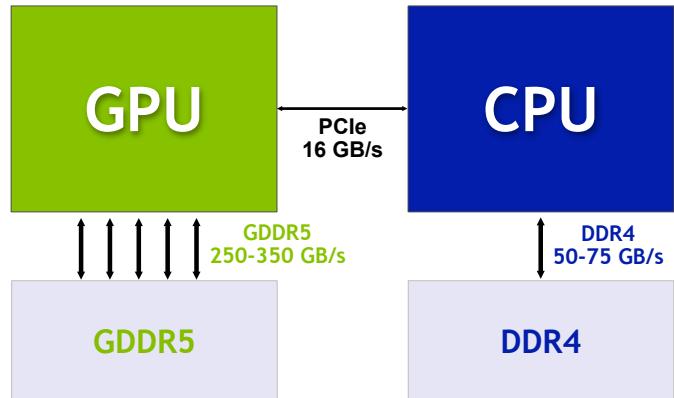


II. 6. La quinta generación: Pascal (GPxxx)



78

Hoy



81

Manuel Ujaldon - Nvidia CUDA Fellow

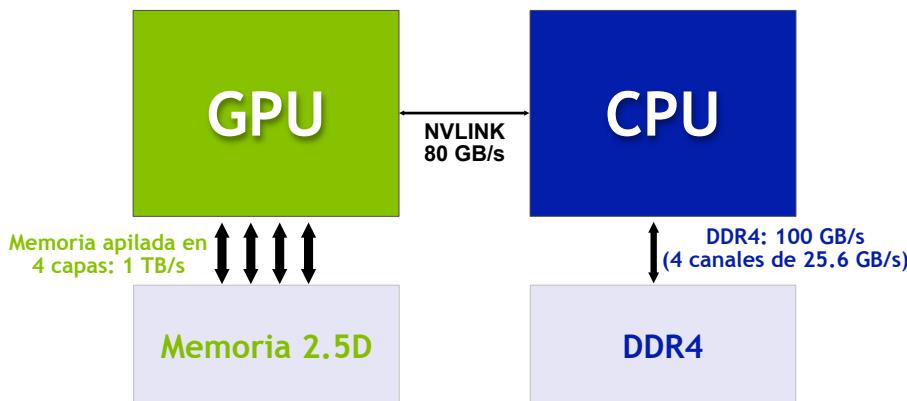
82

Manuel Ujaldon - Nvidia CUDA Fellow

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



En 2017



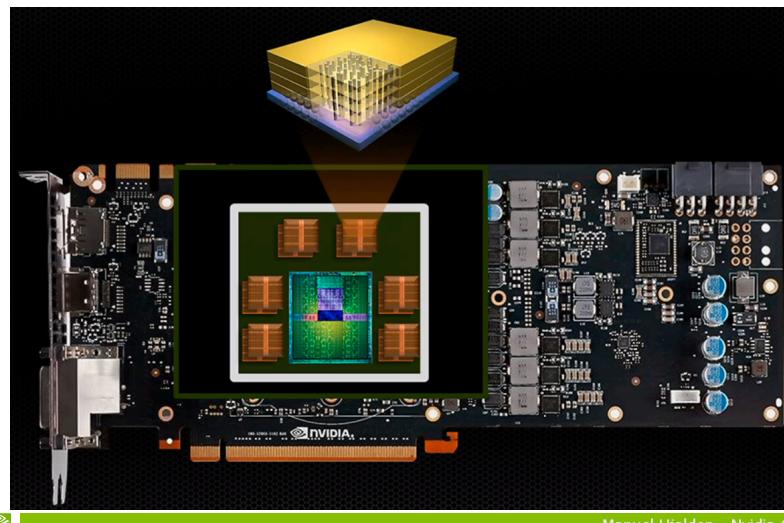
83

Manuel Ujaldon - Nvidia CUDA Fellow

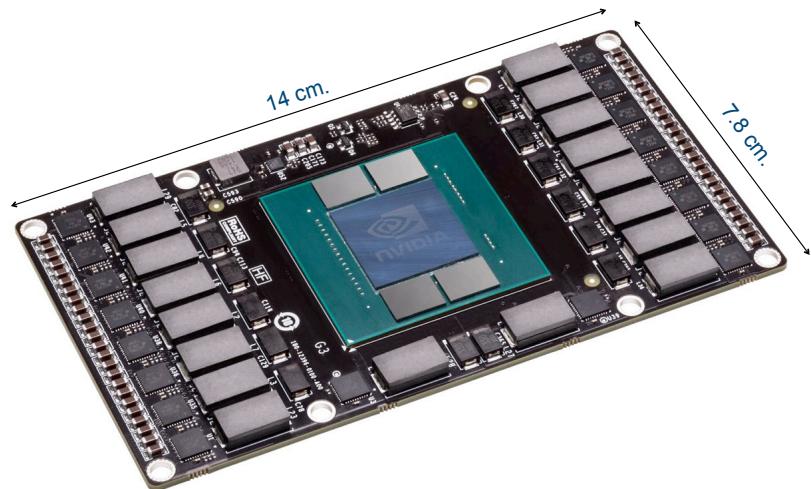
84

Manuel Ujaldon - Nvidia CUDA Fellow

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



Un prototipo de GPU Pascal

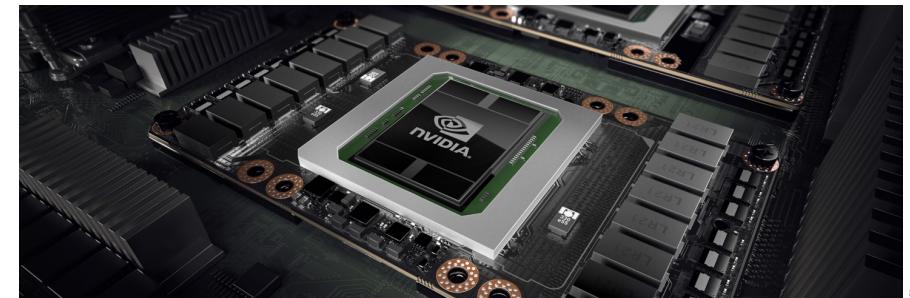


85

Manuel Ujaldon - Nvidia CUDA Fellow

La placa de circuito impreso de Pascal

- 3 veces más eficiente en rendimiento.
- 15.300 M. transistores FinFET 16 nm. en un área de 610 mm²:
- Fabricado por TSMC.
- Cubos de memoria HBM2 dotados de 4096 líneas:
- Fabricado por Samsung.



86

Manuel Ujaldon - Nvidia CUDA Fellow

Primer modelo comercial: GeForce GTX 1080. Comparativa con las 2 generaciones anteriores

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

87

Manuel Ujaldon - Nvidia CUDA Fellow

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

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

88

Manuel Ujaldon - Nvidia CUDA Fellow

Nueva gama de productos para aplicaciones de Deep Learning

Modelo comercial	Chip	Perfil (coste) [mem.]	Aplicaciones	Requerimientos y características	Hardware exclusivo
Tarjetas para entrenamiento: Tesla P100	GP100	High-end (5000 €) [16 GB]	Entrenamiento de redes neuronales	Rendimiento de fp16 y potencia	HPC fp16. Memoria HBM2
Tarjetas para inferencias: Tesla P40	GP102	Mid-end (5000 €) [24 GB]	Que escalen bien sobre un puñado de GPUs	Menos precisión y rendimiento. 50% más energía por GFLOPS vs P4.	Operaciones int8 de alta velocidad (producto escalar de 8 bits con acumulado de 32 bits) [47 TOPS]
Tarjetas para inferencias: Tesla P40	GP104	Low-end (1700 €) [8 GB]	Que escalen masivamente sobre multitud de GPUs	Menor rendimiento. Mayor densidad.	

89



Manuel Ujaldon - Nvidia CUDA Fellow

Tarjetas Tesla para centros de datos

	Tesla M4 (Maxwell)	Tesla M40 (Maxwell)	Tesla P4 (Pascal)	Tesla P40 (Pascal)
Chip de fabricación	GM206 (227 mm ²)	GM200 (601 mm ²)	GP104 (314 mm ²)	GP102 (471 mm ²)
Transistores	2940M @ 28 nm.	8000M @ 28 nm.	7200M @ 16 nm.	12000M @ 16 nm.
CUDA Compute Capabil.	5.2	5.2	6.1	6.1
Solución térmica	50-75 W	250 W	50-75 W	250 W
# multiprocesadores	8	24	40	60
fp32 cores / Multiproc.	128	128	64	64
fp32 cores / GPU	1024	3072	2560	3840
fp64 cores / Multiproc.	4	4	2	2
Frecuencia de reloj	872, 1072 MHz	948, 1114 MHz	810, 1063 MHz	1303, 1531 MHz
TFLOPS (fp32/fp64)	2195 / 69	6844 / 213	5442 / 170	11758 / 367
Banco de regs. / Multip.	65536	65536	65536	65536
Memoria comp. / Multip.	96 KB	96 KB	96 KB	96 KB
Tamaño de caché L2	2048 KB	3072 KB	2048 KB	3072 KB
Interfaz de memoria	128-bit GDDR5	384-bit GDDR5	256-bit GDDR5	384 GDDR5
Tamaño de la memoria	4 GB	12, 24 GB	8 GB	24 GB
Ancho de banda de la "	88 GB/s	288 GB/s	192	345.6 GB/s

90



Manuel Ujaldon - Nvidia CUDA Fellow

Productos comerciales de Pascal

	Tesla P4 (GP104)	Tesla P40 (GP102)	P100 w. NV-link (GP100)
Fecha de lanzamiento	Nov, 2016	Oct, 2016	Apr, 2016
Gama y zócalo	Alta (PCI-e 3.0 x16)	Media (PCI-e 3.0 x16)	Baja (SXM2)
# multiprocesadores	40	60	56
fp32 cores / Multip.	64	64	64
fp32 cores / GPU	2560	3840	3584
fp64 cores / Multip.	32	32	32
Frecuencia de reloj	810, 1063	1303, 1531	1328, 1480 MHz
Rendimiento pico (SP)	5442 GFLOPS	11758 GFLOPS	10608 GFLOPS
Banco de regs / Multip.	65536	65536	65536
Memoria comp. / Multip.	96 KB	96 KB	64 KB
Tamaño de caché L2	2048 KB	3072 KB	4096 KB
Interfaz de memoria	256-bit GDDR5 2x3000 MHz	384-bit GDDR5 2x3615 MHz	4096-bit HBM2
Tamaño de la memoria	8 GB	12+12 GB	16 GB
Ancho de banda de "	192 GB/s	345.6 GB/s	720 GB/s

91



Manuel Ujaldon - Nvidia CUDA Fellow

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

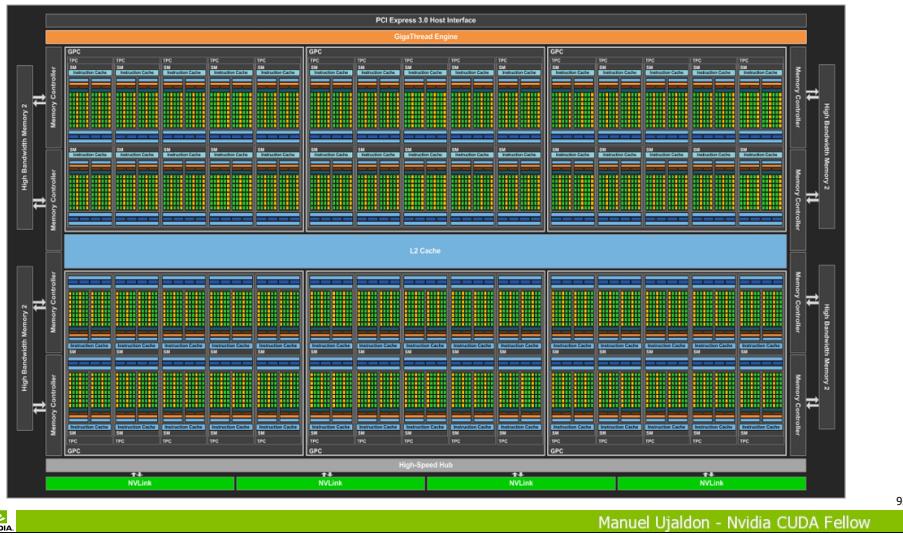


92



Manuel Ujaldon - Nvidia CUDA Fellow

Disposición física de sus multiprocesadores, buses y controladores de memoria

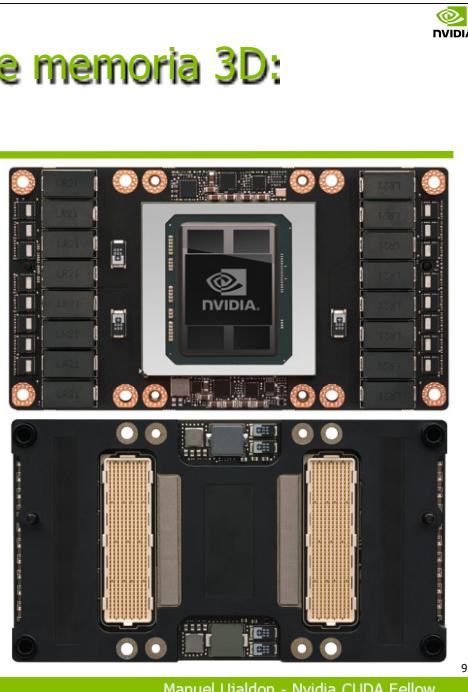


El multiprocesador CUDA de Pascal



Pascal y sus 4 cubos de memoria 3D: Envés y revés

- **Modelo Tesla P100:**
 - GPU: 56 SMs de 64 cores.
Rendimiento pico:
 - 5.3 TFLOPS (FP64).
 - 10.6 TFLOPS (FP32).
 - 21.2 TFLOPS (FP16).
 - Memoria:
 - Más bancos de registros y memoria compartida que Maxwell (mismos tamaños por cada SMX) pero hay 56 SMX2, frente al total de 24 en Maxwell).
 - Bus NVLINK: 160 GB/s.
 - 80 GB/s. bidireccionales.

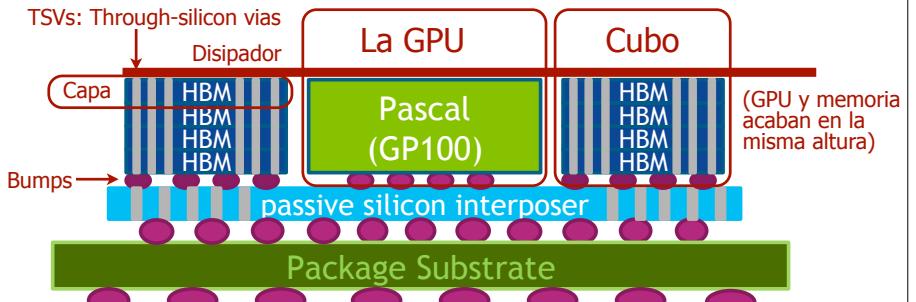


La memoria de Pascal: Cómo alcanzar 1 TB/s con un reloj de 2x 500 MHz

- BW = Frecuencia*anchura => 1 TB/s = 2x500MHz*anchura => anchura = 8000 Gbits/s / 1 GHz = 8000 bits

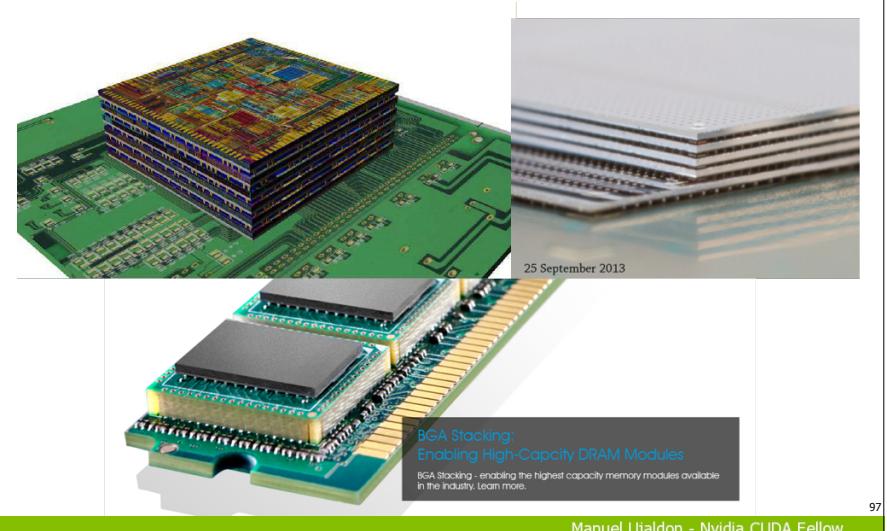
- Ancho de Titan X: 384 bits. Máx. histórico en GPU: 512 bits.

TSVs: Through-silicon vias



- ¡Hay una jerarquía de interconexión!

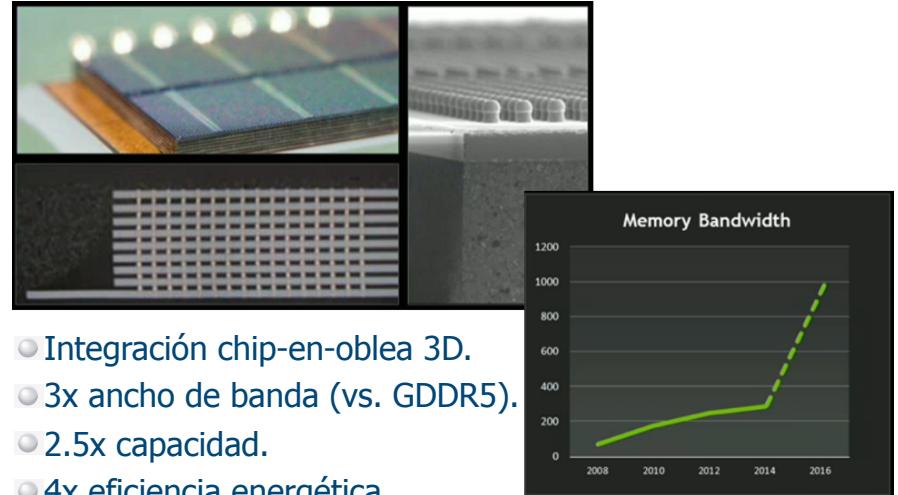
Desarrollos comerciales: Prototipos de Micron (arriba) y productos de Viking (abajo)



97

Manuel Ujaldon - Nvidia CUDA Fellow

Memoria 3D de Pascal en producción: Disponible comercialmente en 1Q'17



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

98

Manuel Ujaldon - Nvidia CUDA Fellow

La hoja de ruta de Nvidia



99

Manuel Ujaldon - Nvidia CUDA Fellow

Para más información sobre memorias 3D, visitar uno de los dos consorcios:

- HMCC (Hybrid Memory Cube Consortium).
 - Mentores: Micron y Samsung.
 - <http://www.hybridmemorycube.org> (HMC 1.0, 1.1, 2.0 disponibles)
 - Adoptado por los aceleradores Xeon Phi de Intel con leves retoques.
- HBM (High Bandwidth Memory).
 - Mentores: AMD and SK Hynix.
 - <https://www.jedec.org/standards-documents/docs/jesd235> (acceso via JEDEC).
 - Adoptado por las GPUs de Nvidia (Pascal, HBM2).

100

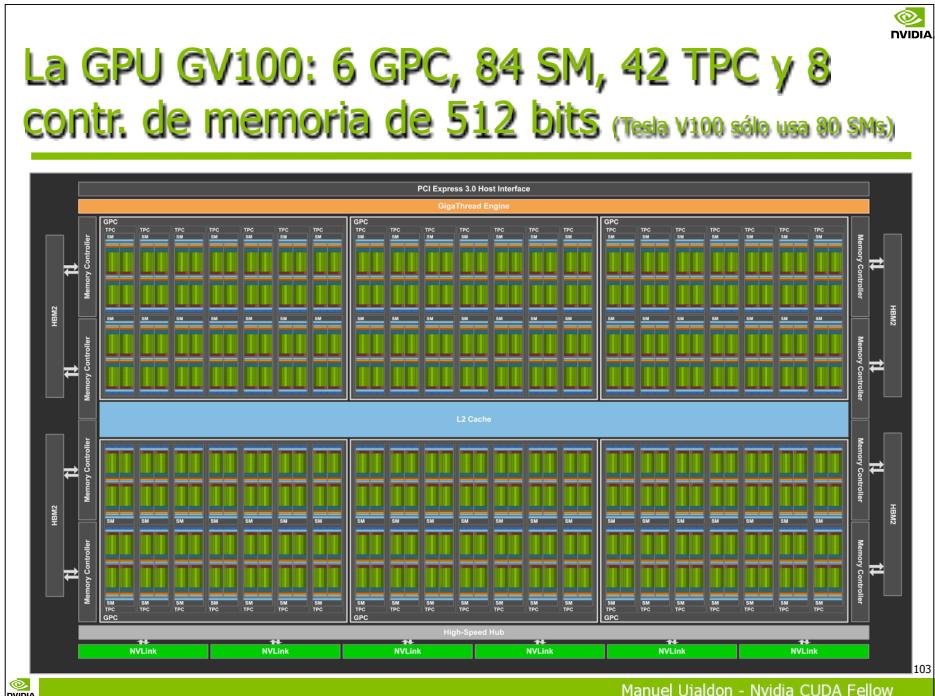
Manuel Ujaldon - Nvidia CUDA Fellow



II. 7. La sexta generación: Volta (GVxxx)



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



Comparativa con generaciones anteriores en la gama Tesla

	K40 (Kepler)	M40 (Maxwell)	P100 (Pascal)	V100 (Volta)
GPU (chip)	GK110	GM200	GP100	GV100
Millones de transistores	7100	8000	15300	21100
Área de integración	551 mm ²	601 mm ²	610 mm ²	815 mm ²
Fabricación	28 nm.	28 nm.	16 nm. FinFET	12 nm. FFN
Dissipación de calor (TDP)	235 W.	250 W.	300 W.	300 W.
Número de cores	2880 (15 x 192)	3072 (24 x 128)	3584 (56 x 64)	5120 (80 x 64)
Número de unidades fp64	960	96	1792	2560
Frecuencia máx. (boost)	875 MHz	1114 MHz	1480 MHz	1455 MHz
TFLOPS (fp32 / fp64)	5.04 / 1.68	6.8 / 2.1	10.6 / 5.3	15 / 7.5
Interfaz de memoria	GDDR5 de 384 bits		HBM2 de 4096 bits	
Memoria de vídeo	Hasta 12 GB	Hasta 24 GB	16 GB	16 GB
Caché L2	1536 KB	3072 KB	4096 KB	6144 KB
Memoria compartida / SM	48 KB	96 KB	64 KB	Hasta 96 KB
Banco de registros / SM	65536	65536	65536	65536

Manuel Ujaldon - Nvidia CUDA Fellow



Más sobre la GPU GV100

- Los productos comerciales utilizan diferentes configuraciones de GV100.
- Cada controlador de memoria maneja 768 KB de L2, para un total de 6144 KB de caché L2 en la GV100 más completa.
- Cada torre de memoria HBM2 se gestiona desde un par de controladores de 512 bits.
- Disminuyen las latencias de instrucción y acceso a caché.
- Aumentan los relojes y la eficiencia energética.
- Más hilos, warps y bloques de hilos activos frente a Pascal.
- Separa las unidades int32 y fp32 para que puedan funcionar simultáneamente.
- Se reduce de 6 a 4 la latencia para despachar una op. "madd".



Manuel Ujaldon - Nvidia CUDA Fellow

104

El multiprocesador de Volta

Cores:

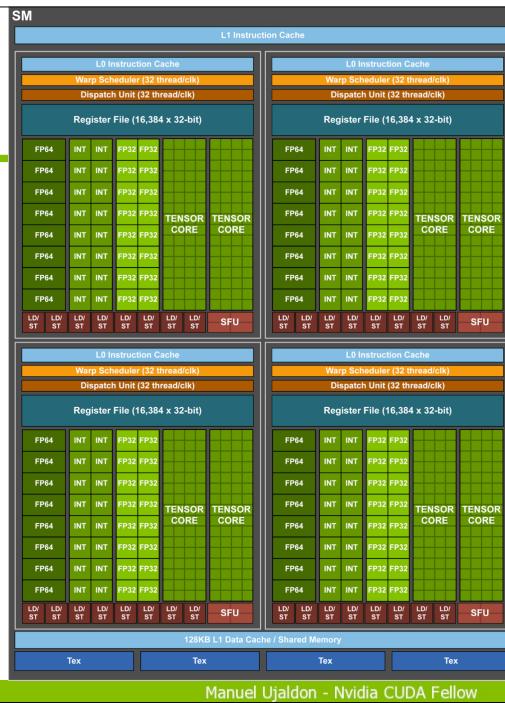
- 64 int32 ("int").
- 64 fp32 ("float").
- 32 fp64 ("double").
- 8 unidades tensor.

Unidades de almacen.:

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

Memoria:

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



Manuel Ujaldon - Nvidia CUDA Fellow

106

Composición del multiprocesador Volta frente al de Pascal

	GP100	GV100
Conjuntos de procesamiento ("plantilla clonada")	2	4
Cores int32 / conjunto	32	16
Cores fp32 / conjunto	32	16
Cores fp64 / conjunto	16	8
Cores tensor / conjunto	Ninguno	2
Caché de instrucciones L0 / conjunto	Ninguno (buffer de instrucciones en su lugar)	1
Banco de registros / conjunto	128 K	64 K
Planificadores de warp / conjunto	1	1
Unidades de emisión de instrs. / conjunto	1	1

Manuel Ujaldon - Nvidia CUDA Fellow

106

Evolución del multiprocesador: Desde Pascal a Volta



Manuel Ujaldon - Nvidia CUDA Fellow

107

El nuevo multiprocesador (SM) optimizado para Deep Learning

- Rendimiento pico (basado en el máximo reloj GPU Boost):
 - fp64: 7.5 TFLOPS.
 - fp32: 15 TFLOPS.
- Cores tensor.
 - Hasta 12x TFLOPS pico para la fase de entrenamiento (120 TFLOPS para fp32).
- Planificación de hilos independiente.
 - Habilita la sincronización de grano fino y la cooperación entre hilos.
- Caché de datos L1 y memoria compartida combinadas.
 - Mejora el rendimiento y simplifica la programación.

Manuel Ujaldon - Nvidia CUDA Fellow

108

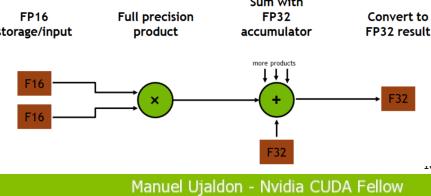
Estructura del core tensor (8 por multiprocesador)

- Matriz de procesamiento 4x4x4 para computar $D = A \cdot B + C$.

$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix}_{\text{FP16 or FP32}} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix}_{\text{FP16}} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}_{\text{FP16 or FP32}}$$

- 64 madd operaciones de precisión mixta por ciclo de reloj:

- Dos matrices de entrada de precisión mixta (fp16) con producto de precisión simple (fp32).
- Acumulación y resultado final en precisión simple (fp32).



La memoria compartida y la caché L1 se fusionan en una sola unidad

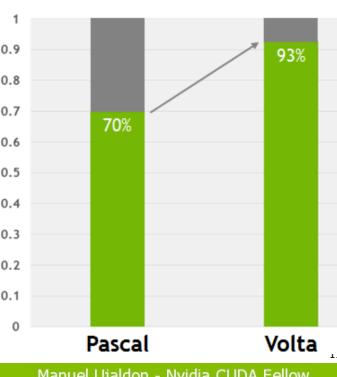
- De esta manera se obtiene un rendimiento más cercano

Ejemplo experimental: % de ralentización cuando se prescinde del uso explícito de la memoria compartida, para dejar todo en manos de una gestión automática de la caché L1:

- Now L1 cache can get all SRAM when shared memory is not used.

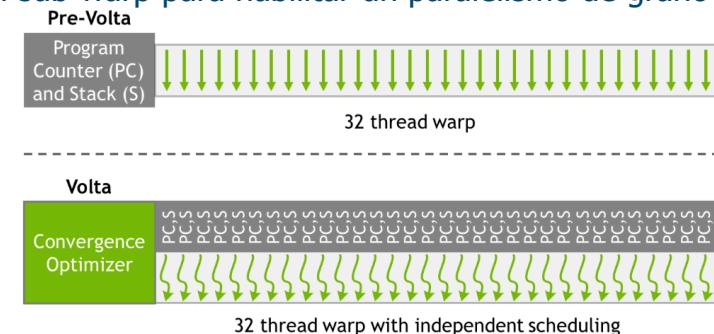
- L1 has much lower latency and higher bandwidth than in the past.

- Texture units also use the cache.



Planificación de hilos independiente

- Volta mantiene un estado de ejecución por hilo, incluyendo el contador de programa y la pila de llamadas.
- Ahora el programador puede definir el comportamiento a nivel sub-warp para habilitar un paralelismo de grano fino.



Rompiendo la uniformidad del warp

- “Cooperative Groups” es un nuevo modelo de programación que inaugura CUDA 9 para poder definir grupos organizados de hilos cooperativos. Esto permite al programador expresar la granularidad con la que pueden comunicarse los hilos, logrando descomposiciones paralelas más eficientes.

Evolución:

- Kepler ya soportaba la funcionalidad básica de “Cooperative Groups”.
- Pascal incluía soporte para nuevas APIs de lanzamientos cooperativos que permiten la sincronización entre bloques de hilos CUDA.
- Volta incorpora soporte para patrones de nueva sincronización.



Eficiencia energética

- Volta es un 50% más eficiente energéticamente que Pascal.

Nuevos modos de Gestión de Consumo:

- Máximo rendimiento: Opera sin restricciones hasta el TDP (300W)
- Maxima eficiencia: Rendimiento/vatio óptimo. Se puede establecer un límite para el consumo en el conjunto de GPUs pertenecientes a un rack.



Acceso a memoria y rendimiento

Migración: Memoria unificada

- En GV100: Nuevos contadores de acceso para mejorar la migración de las páginas de memoria al procesador que accede a ellas de forma más frecuente.

- En plataformas Power de IBM: Nuevos servicios de traducción de direcciones para permitir a la GPU acceder a las tablas de páginas de la CPU de forma directa.

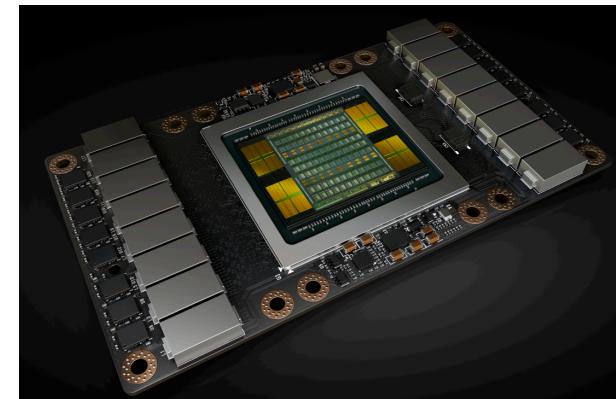
Ancho de banda: Memoria HBM2 de 16 GB

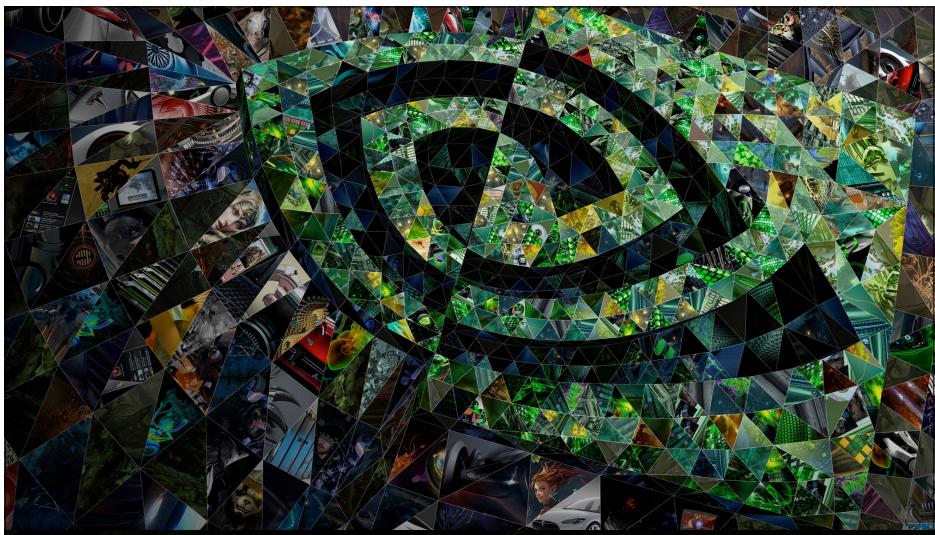
- Memoria HBM2 de nueva generación (de Samsung): Ancho de banda pico de 900 GB/s (1.25x frente a los 720 GB/s pico en Pascal).
- Nuevo controlador de memoria (de Nvidia): Eficiencia del ancho de banda superior al 95% ejecutando muchos programas de prueba.



Interconexión: Zócalos y sockets

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





II. 8. 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

Manuel Ujaldon - Nvidia CUDA Fellow

118

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

119

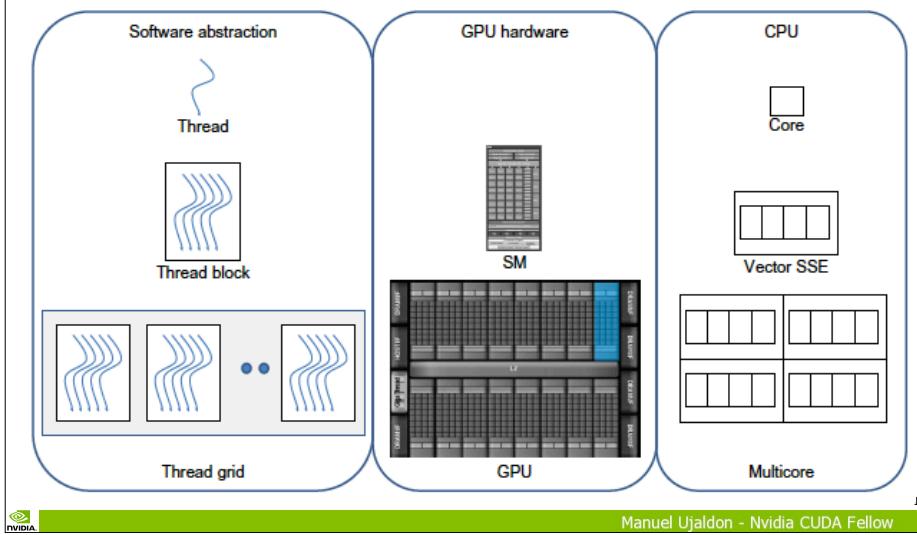
Manuel Ujaldon - Nvidia CUDA Fellow



III. Programación



Comparativa con la CPU: Dos formas de construir supercomputadores



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

POSIX-threads en CPU

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

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

```
#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;
}
```

```
#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;
}
```



Manuel Ujaldon - Nvidia CUDA Fellow

122

Unos pocos parámetros distinguen los modelos comerciales

- 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 conmutación de contexto es inmediata.
- Objetivo del programador: Declarar miles de hilos, que la GPU necesita para lograr rendimiento y escalabilidad.



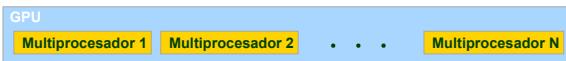
Manuel Ujaldon - Nvidia CUDA Fellow

124

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 conmutación de contexto es inmediata.

- Objetivo del programador: Declarar miles de hilos, que la GPU necesita para lograr rendimiento y escalabilidad.



Manuel Ujaldon - Nvidia CUDA Fellow

123

Estructura de un programa CUDA

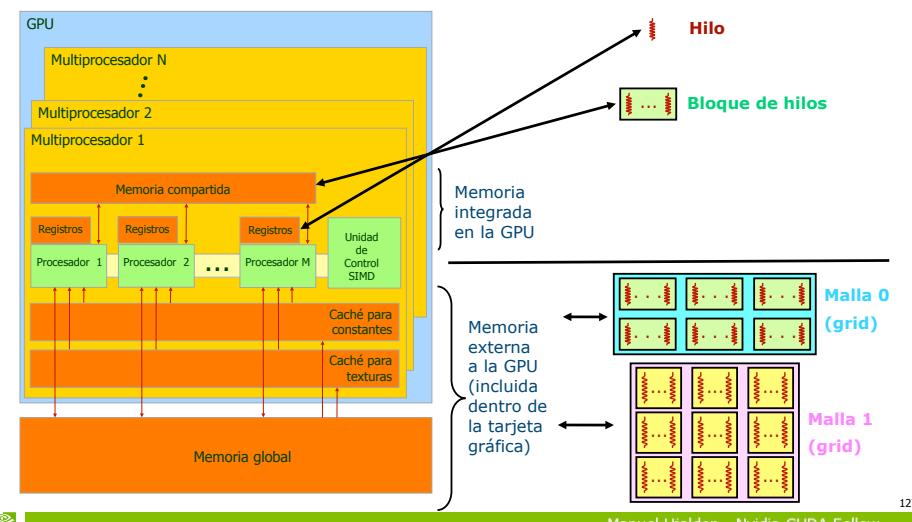
- Cada multiprocesador procesa lotes de bloques, uno detrás de otro
 - Bloques activos = los bloques procesados por un multiprocesador en un lote.
 - Hilos activos = todos los que provienen de los bloques que se encuentren activos.
- Los registros y la memoria compartida de un multiprocesador se reparten 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)							Limi-tación	Im-pacto
	1.0, 1.1	1.2, 1.3	2.0, 2.1	3.0, 3.5, 3.7	5.0, 5.2, 5.3	6.0, 6.1	7.0		
Multiprocesadores / GPU	16	30	14-16	13-16	4, 5, ...	40,56,60	80	HW	Escala-bilidad
fp32 cores / Multip.	8	8	32	192	128	64	64		
Hilos / Warp	32	32	32	32	32	32	32	SW	Ritmo de salida de datos
Bloques / Multiprocesador	8	8	8	16	32	32	32		
Hilos / Bloque	512	512	1024	1024	1024	1024	1024	SW	Paralelismo
Hilos / Multiprocesador	768	1024	1536	2048	2048	2048	2048		
Regs. de 32 bits / Multip.	8K	16K	32K	64K	64K	64K	64K	HW	Conjunto de trabajo
Mem. compartida / Multip.	16K	16K	16KB	16KB,	64K (5.0)	64KB,(6.0)	Configurable hasta 96KB.		

La relación entre CCC y el mercado de GPUs



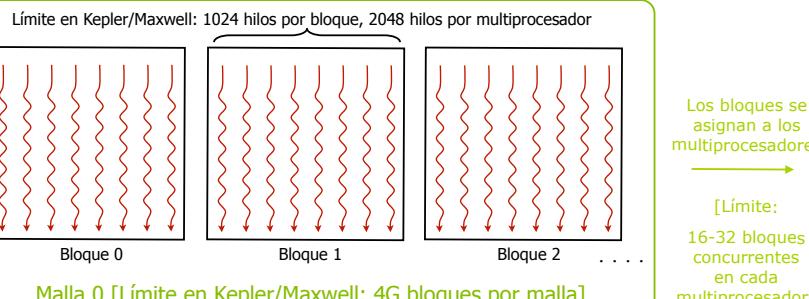
CCC	"Code names"	Modelos vinculados a CUDA	Series comerciales	Marco temporal (año)	Proceso de fabricación @ TSMC
1.0	G80	Muchos	8xxx	2006-07	90 nm.
1.1	G84,6 G92,4,6,8	Muchos	8xxx/9xxx	2007-09	80, 65, 55 nm.
1.2	GT215,6,8	Pocos	2xx	2009-10	40 nm.
1.3	GT200	Muchos	2xx	2008-09	65, 55 nm.
2.0	GF100, GF110	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.
6.0	GP100	Todos	P100	2016-17	16 nm. finFET
6.1	GP102,4,6,7,8	Muchos	10xx	2017	16 nm. finFET
7.0	GV100	Todos	V100	2017-18	12 nm. FFN

129

Manuel Ujaldon - Nvidia CUDA Fellow



Bloques e hilos en GPU



- 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()`.

131

Manuel Ujaldon - Nvidia CUDA Fellow

Para identificar las series de Nvidia en un catálogo comercial



- 200: Desarrollada durante 3T'08, hasta 4T'09. Mejora la G80 con 240 cores (GTX260 and GTX280).
- 400: Arranca en 1T'10, inaugurando Fermi. Hasta 3T'11.
- 500: Empieza en 4T'10 con la GTX580 [GF110], y termina la generación Fermi en 1T'12.
- 600: 2012-13. Presenta Kepler, pero también tiene Fermis.
- 700: 2013-14. Centrada en Kepler, incluye las últimas Fermis [GF108] y las primeras Maxwells [GM107, GM108].
- 800M: 1T'14 sólo para portátiles, combinando Fermi [GF117], Kepler [GK104] y Maxwell [GM107, GM108].
- 900: Comienza en 4T'14, mejorando Maxwell [GM20x].
- 1000: Empieza en 2T'16 con las primeras Pascal [GP10x].

130

Manuel Ujaldon - Nvidia CUDA Fellow

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

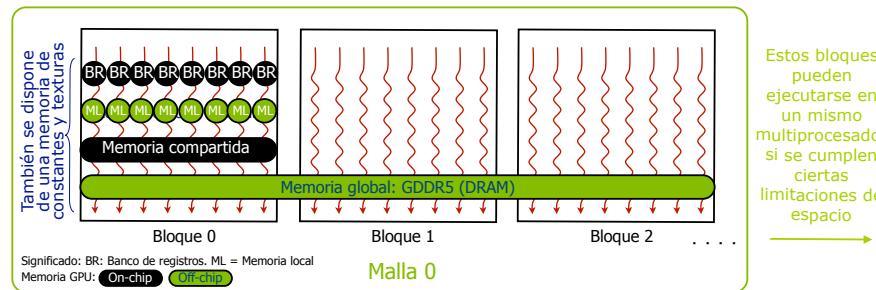


- Límites para un multiprocesador SMM: [1] 32 bloques, [2] 1024 hilos/bloque y [3] 2048 hilos en total.
- 1 bloque de 2048 hilos. No lo permite [2].
- 2 bloques de 1024 hilos. Posible en el mismo multiproc.
- 4 bloques de 512 hilos. Posible en el mismo multiproc.
- 4 bloques de 1024 hilos. No lo permite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

132

Manuel Ujaldon - Nvidia CUDA Fellow

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.

133

Manuel Ujaldon - Nvidia CUDA Fellow

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

- Límites dentro de un multiprocesador SMM: 64 Kregs. y 96 KB. de memoria compartida. Así:
 - 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**.
 - 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**.
 - ...y así sucesivamente. Cuanto menos memoria usemos, mayor concurrencia para la ejecución de bloques.
- El programador debe establecer el compromiso adecuado entre apostar por memoria o paralelismo en cada código.

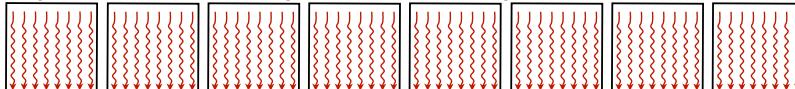
134

Manuel Ujaldon - Nvidia CUDA Fellow

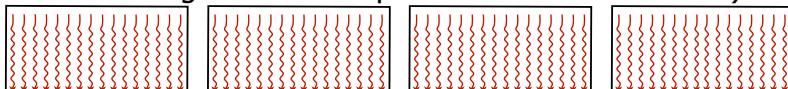
Pensando en pequeño: Particionamiento 1D de un vector de 64 elementos

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

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



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



135

Manuel Ujaldon - Nvidia CUDA Fellow

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

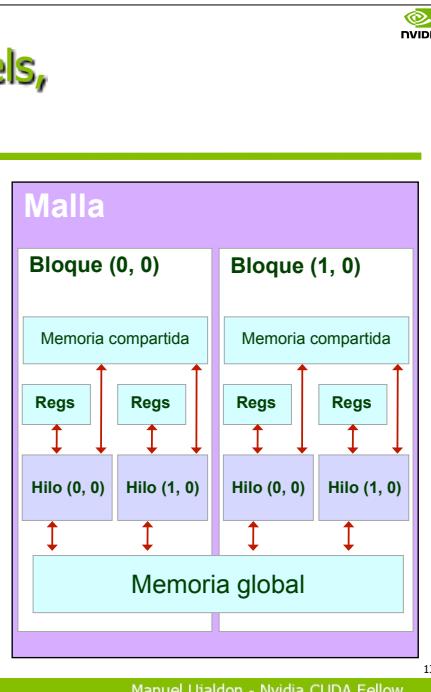
- Máximo número de hilos por bloque: 1024.
- Maximo número de bloques:
 - 64K en Fermi.
 - 4G en Kepler/Maxwell.
- Tamaños más grandes para las estructuras de datos sólo pueden implementarse mediante un número ingente de bloques (si queremos preservar el paralelismo fino).
- Posibilidades a elegir:
 - 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.

136

Manuel Ujaldon - Nvidia CUDA Fellow

Recopilando sobre kernels, bloques y paralelismo

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

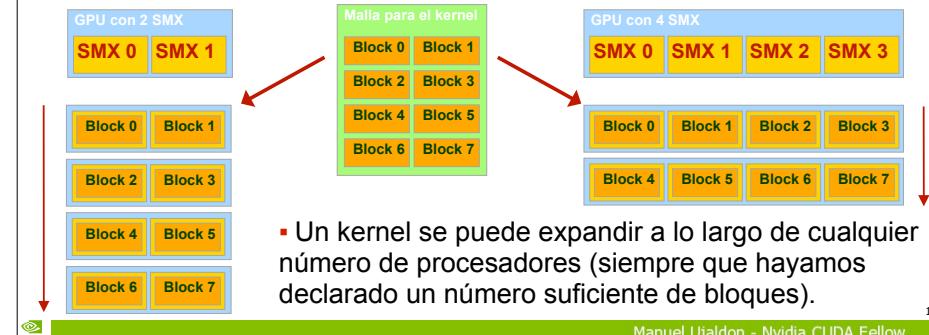


Manuel Ujaldon - Nvidia CUDA Fellow

Escalabilidad transparente

- Dado que los bloques no se pueden sincronizar:

- El hardware tiene libertad para planificar la ejecución de un bloque en cualquier multiprocesador.
- Los bloques pueden ejecutarse secuencialmente o concurrentemente según la disponibilidad de recursos.

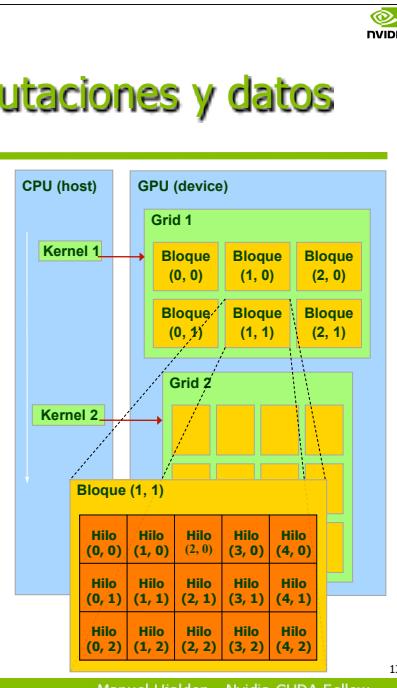


- Un kernel se puede expandir a lo largo de cualquier número de procesadores (siempre que hayamos declarado un número suficiente de bloques).

Manuel Ujaldon - Nvidia CUDA Fellow

Particionamiento de computaciones y datos

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



Manuel Ujaldon - Nvidia CUDA Fellow

Espacios de memoria

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

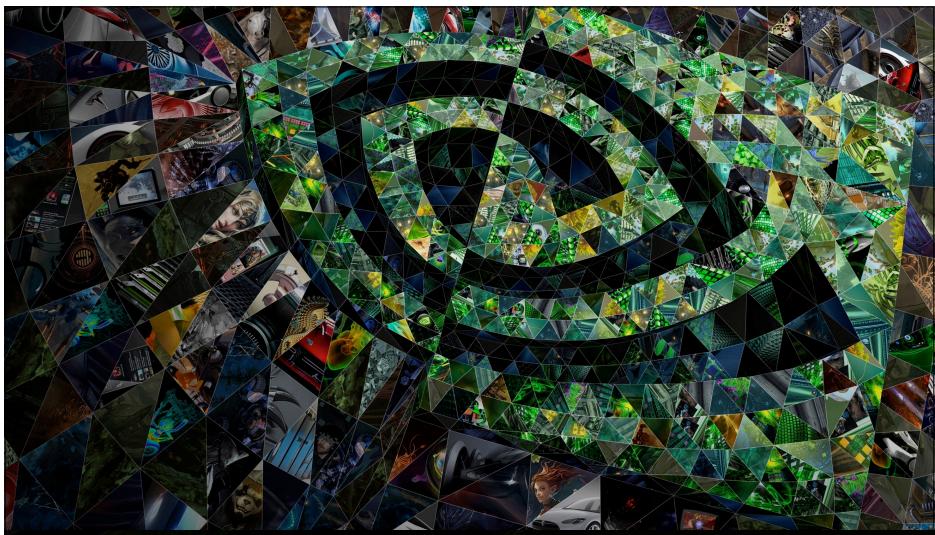
- Para comunicar ambos procesadores, se utiliza el bus PCI-express.
- En la GPU se utilizan funciones para alojar memoria y copiar datos de la CPU de forma similar a como la CPU procede en lenguaje C (malloc para alojar y free para liberar).

- 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**).

140

Manuel Ujaldon - Nvidia CUDA Fellow



IV. Sintaxis



**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);
```



143



IV. 1. Elementos básicos



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);
```

144

La interacción entre la CPU y la GPU

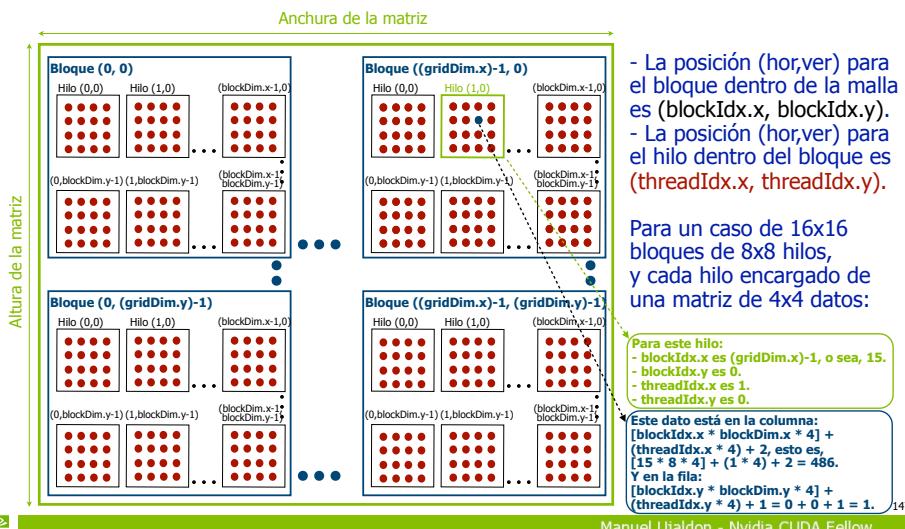
- CUDA extiende el lenguaje C con un nuevo tipo de función, kernel, que ejecutan en paralelo los hilos activos en GPU.
- El resto del código es C nativo que se ejecuta sobre la CPU de forma convencional.
- De esta manera, el típico `main()` de C combina la ejecución secuencial en CPU y paralela en GPU 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.

145

Manuel Ujaldon - Nvidia CUDA Fellow



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

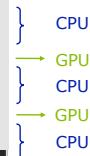


Manuel Ujaldon - Nvidia CUDA Fellow

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

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

    Serial Code
    Parallel Kernel
    KernelA<<< nBlk, nTid >>>(args);
    Serial Code
    Parallel Kernel
    KernelB<<< nBlk, nTid >>>(args);
```



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

146

Manuel Ujaldon - Nvidia CUDA Fellow

Identificación de los entes al programar (sobre un ejemplo de suma de matrices)

- Para cada uno de los 4x4 bloques del grid:

- BlockIdx: vector (1D o 2D) que identifica el bloque dentro del grid.

- Para cada uno de los 4x4 hilos del bloque:

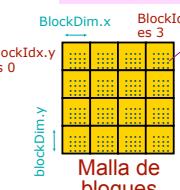
- ThreadId: vector (1D, 2D o 3D) que identifica el hilo dentro de su bloque.

```
__global__ void matAdd (float A[N][N],float B[N][N],float C[N][N])
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    int j = blockIdx.y*blockDim.y + threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main(){
    dim3 dimBlock(4,4);
    dim3 dimGrid (N/BlockDim.x, N/BlockDim.y);
    matAdd <<< dimGrid, dimBlock >>> (A, B, C);
}
```

Matrices A, B y C de 16x16 elementos
Las tres matrices se partitionan igual, otorgando un elemento a cada hilo

¿Qué hilo computa C[15,2]?
blockIdx = (3,0)
threadIdx = (3,2)



Malla de bloques
blockIdx.y es 0
blockIdx.x es 3

148

Manuel Ujaldon - Nvidia CUDA Fellow

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.
```

149

Manuel Ujaldon - Nvidia CUDA Fellow



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);
```

151

Manuel Ujaldon - Nvidia CUDA Fellow



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.

150

Manuel Ujaldon - Nvidia CUDA Fellow



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
```

152

Manuel Ujaldon - Nvidia CUDA Fellow



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.

153

Manuel Ujaldon - Nvidia CUDA Fellow



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.

155

Manuel Ujaldon - Nvidia CUDA Fellow



IV. 2. Un par de ejemplos



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 disposit. (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);
}
```

156

Manuel Ujaldon - Nvidia CUDA Fellow



Transferencias de memoria asíncronas

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

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

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

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

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

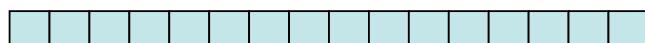
El kernel CUDA que se ejecuta en GPU,
seguido del código host para CPU.
Este archivo se compila con `nvcc`

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

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

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

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



Extensiones al lenguaje

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

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

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

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

`int idx = (blockIdx.x * blockDim.x) + threadIdx.x;` } Patrón de acceso común a todos los hilos
Se mapeará del índice local `threadIdx.x` al índice global

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);

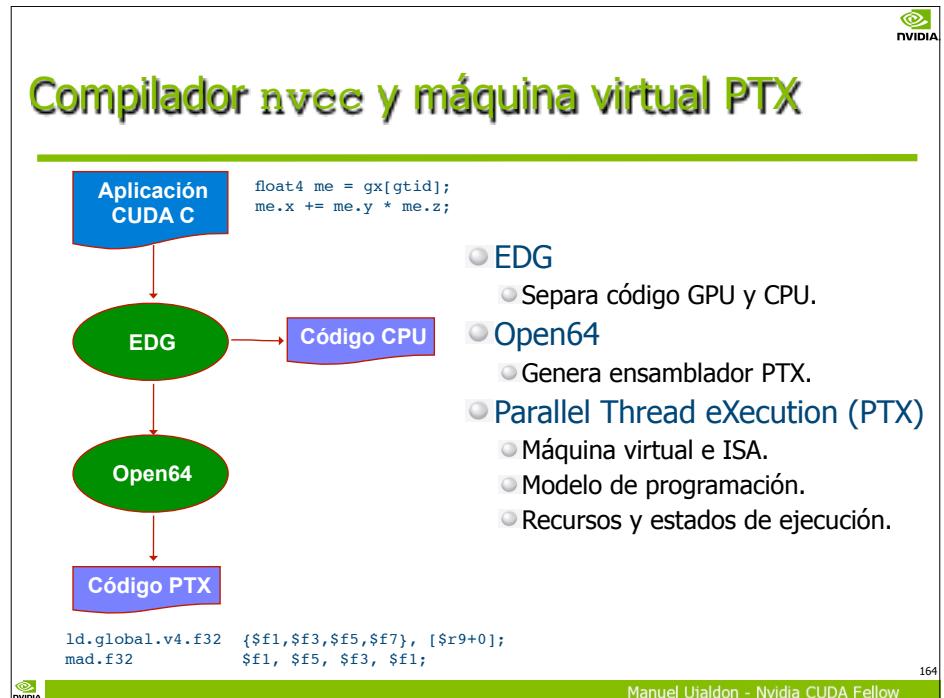
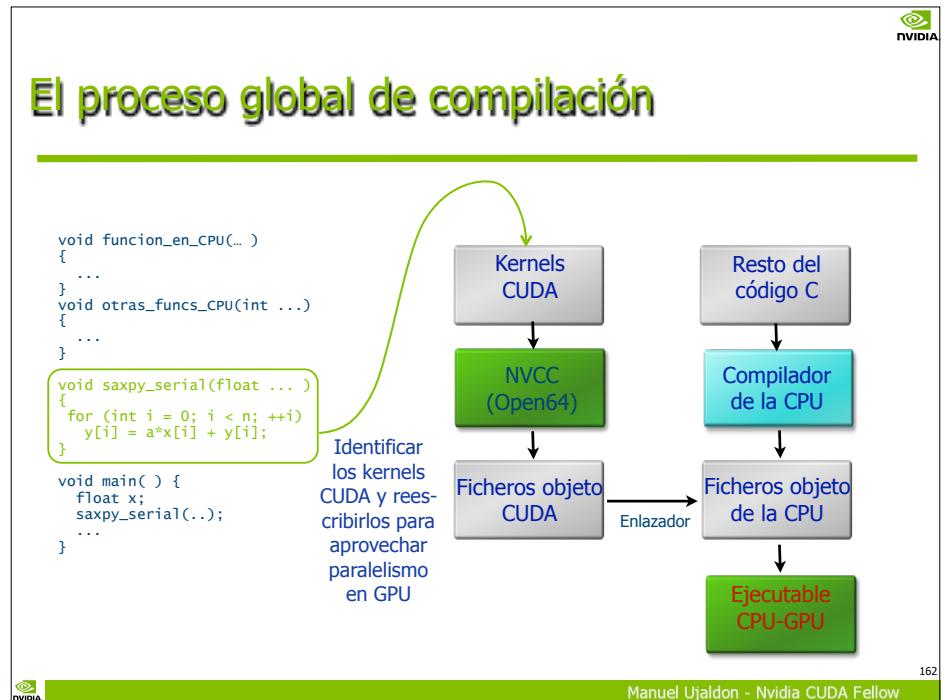
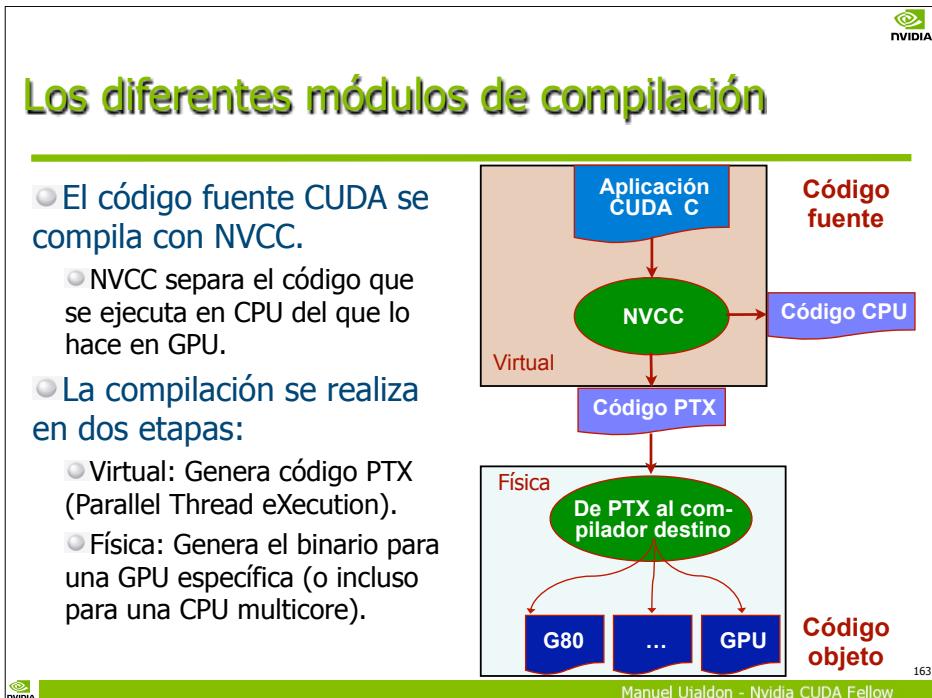
// Ejecuta el kernel con un número de bloques y tamaño de bloque
incremento_en_gpu <<< N/blockSize, blockSize >>> (d_A, b);

// 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

NVIDIA.



NVCC (NVidia CUDA Compiler)

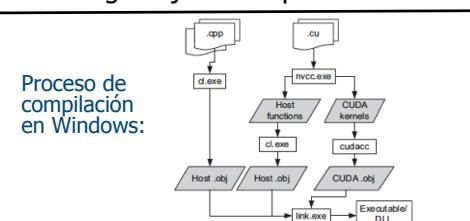
- NVCC es un driver del compilador.

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

- NVCC produce como salida:

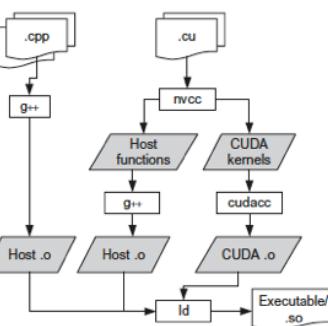
- Código C para la CPU, que debe luego compilarse con el resto de la aplicación utilizando otra herramienta.

- Código objeto PTX para la GPU.



Proceso de compilación en Windows:

Proceso de compilación Linux:



Manuel Ujaldon - Nvidia CUDA Fellow

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

Manuel Ujaldon - Nvidia CUDA Fellow

Heurísticos para la configuración de la ejecución

- El número de hilos por bloque debe ser un múltiplo de 32.

- Para no desperdiciar la ejecución de warps incompletos.

- Debemos declarar más bloques que multiprocesadores haya (1), y a ser posible, ser más del doble (2):

- (1) Para que todos ellos tengan al menos un bloque que ejecutar.
- (2) Para tener al menos un bloque activo que garantice la actividad del SMX cuando el bloque en ejecución sufra un parón debido a un acceso a memoria, no disponibilidad de UFs, conflictos en bancos, puntos de sincronización (`syncthreads()`), etc.

- Los recursos por bloque (registros y memoria compartida) deben ser al menos la mitad del total disponible.

- De lo contrario, resulta mejor fusionar bloques.

Heurísticos (cont.)

- Reglas generales para que el código sea escalable en futuras generaciones y para que el flujo de bloques pueda ejecutarse de forma segmentada (pipeline):

- (1) Pensar a lo grande para el número de bloques.
- (2) Pensar en pequeño para el tamaño de los hilos.

- Conflicto: Más hilos por bloque significa mejor ocultación de latencia, pero menos registros disponibles para cada hilo.

- Sugerencia: Utilizar un mínimo de 64 hilos por bloque, o incluso mejor, 128 ó 256 hilos (si aún disponemos de suficientes registros).

- Conflicto: Incrementar la ocupación no significa necesariamente aumentar el rendimiento, pero una baja ocupación del multiprocesador no permite ocultar latencias en kernels limitados por el ancho de banda a memoria.

- Sugerencia: Vigilar la intensidad aritmética y el paralelismo.

Parametrización de una aplicación

- Todo lo que concierne al rendimiento es dependiente de la aplicación, por lo que hay que experimentar con ella para lograr resultados óptimos.

Las GPUs evolucionan muy rápidamente, sobre todo en:

- El nº de multiprocesadores (SMs) y el número de cores por SMs.
- El ancho de banda con memoria: Entre 100 GB/s. y 1 TB/s.
- El tamaño del banco de registros de cada SM: 8K, 16K, 32K, 64K.
- El tamaño de la memoria compartida: 48 KB. (en Fermi y Kepler), 64 KB., 96 KB. (en algunos modelos de Maxwell y Pascal)
- Hilos: Comprobar el límite por bloque y el límite total.
 - Por bloque: 1024 (a partir de Fermi).
 - Total: 2048 (a partir de Kepler).

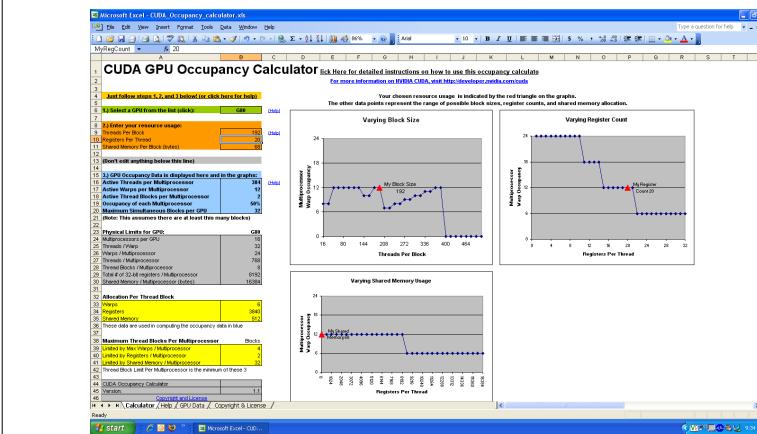
169

Manuel Ujaldon - Nvidia CUDA Fellow

CUDA Occupancy Calculator

- Asesora en la selección de los parámetros de configuración

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



170

Manuel Ujaldon - Nvidia CUDA Fellow

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

El primer dato es el número de hilos por bloque:

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

171

Manuel Ujaldon - Nvidia CUDA Fellow

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

El segundo dato es el nº de registros que usa cada hilo.

- Esto lo obtendremos del archivo .cubin.
- El límite es 64 Kregistros en cada SM (a partir de Kepler), así que consumiendo 32 registros/hilo es posible maximizar la ocupación:
 - 8 bloques * 256 hilos/bloque * 32 registros/hilo = 64 Kregistros.
 - ... disponiendo así del máximo de 2048 hilos activos en cada uno de los multiprocesadores disponibles.

172

Manuel Ujaldon - Nvidia CUDA Fellow

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

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

Manuel Ujaldon - Nvidia CUDA Fellow



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.



175

Manuel Ujaldon - Nvidia CUDA Fellow

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()`;



Manuel Ujaldon - Nvidia CUDA Fellow

176



VI. 1. Suma de dos vectores



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);

```

179

Manuel Ujaldon - Nvidia CUDA Fellow



Código para GPU y su invocación desde CPU

```

// Suma de dos vectores de tamaño N: C[1..N] = A[1..N] + B[1..N]
// Cada hilo computa un solo componente del vector resultado C
__global__ void vecAdd(float* A, float* B, float* C) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    C[tid] = A[tid] + B[tid];
}
int main() { // Lanza N/256 bloques de 256 hilos cada uno
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}

```

Código GPU Código CPU

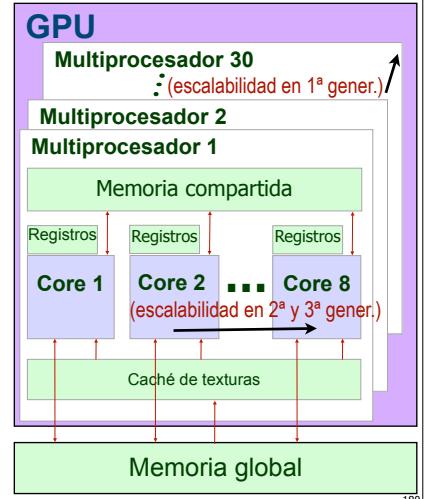
- El prefijo `__global__` indica que `vecAdd()` se ejecuta en la GPU (device) y será llamado desde la CPU (host).
- A, B y C son punteros a la memoria de vídeo de la GPU, por lo que necesitamos:
 - Alojar/liberar memoria en GPU, usando `cudaMalloc/cudaFree`.
 - Estos punteros no pueden ser utilizados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

178

Manuel Ujaldon - Nvidia CUDA Fellow

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.

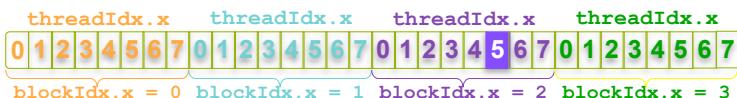


180

Manuel Ujaldon - Nvidia CUDA Fellow

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

- Con M hilos por bloque, un índice único viene dado por:
$$tid = blockIdx.x * blockDim.x + threadIdx.x;$$
- Para acceder a un vector en el que cada hilo computa un solo elemento (queremos paralelismo de grano fino), B=4 bloques de M=8 hilos cada uno:



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

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

181

Manuel Ujaldon - Nvidia CUDA Fellow

Manejando vectores de tamaño genérico

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

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

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

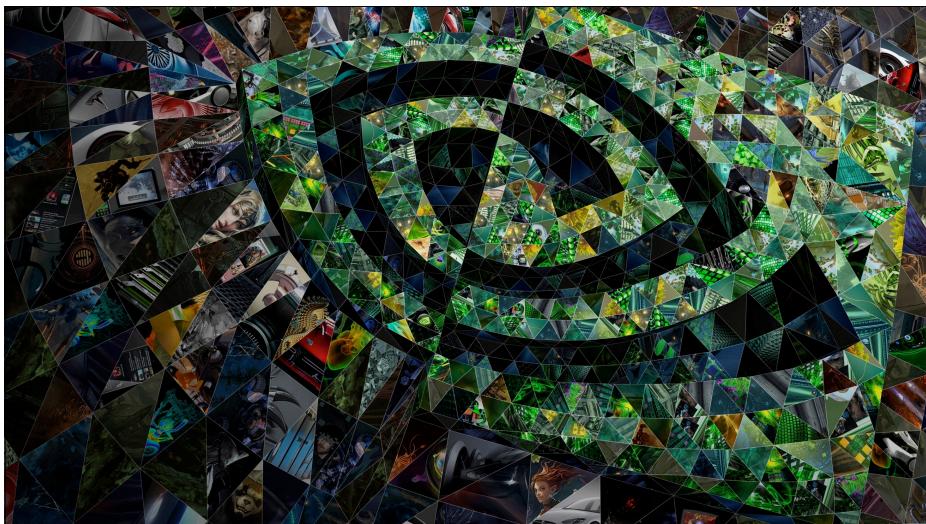
```
vecAdd<<< (N + M-1)/256, 256>>>(d_A, d_B, d_C, N);
```

182

Manuel Ujaldon - Nvidia CUDA Fellow

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 ...



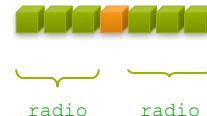
VI. 2. Kernels patrón (stencils)

Patrón unidimensional (1D)

- Aplicemos 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.

185

Manuel Ujaldon - Nvidia CUDA Fellow

Compartiendo datos entre hilos. Ventajas

- Los hilos de un mismo bloque pueden compartir datos a través de memoria compartida.

- El programador gestiona la memoria compartida de forma explícita, insertando el prefijo `__shared__` en la declaración de la variable.

- Los datos se alojan para cada uno de los bloques.

- La memoria compartida es extremadamente rápida:

- 500 veces más rápida que la memoria global (que es memoria de video GDDR5). La diferencia es tecnológica: estática (construida con transistores) frente a dinámica (construida con mini-condensadores).

- La memoria compartida puede verse como una extensión del banco de registros, pero es más versátil que éstos, que son privados a cada hilo.

186

Manuel Ujaldon - Nvidia CUDA Fellow

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.

187

Manuel Ujaldon - Nvidia CUDA Fellow

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.



188

Manuel Ujaldon - Nvidia CUDA Fellow

Kernel patrón

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

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

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



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

189



Manuel Ujaldon - Nvidia CUDA Fellow

Recopilando los conceptos puestos en práctica en este ejemplo

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

191



Manuel Ujaldon - Nvidia CUDA Fellow

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 >
}
```

190



Manuel Ujaldon - Nvidia CUDA Fellow



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.



193

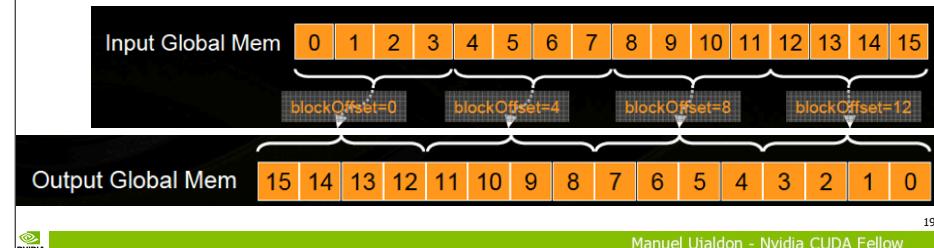
Manuel Ujaldon - Nvidia CUDA Fellow

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];
}
```

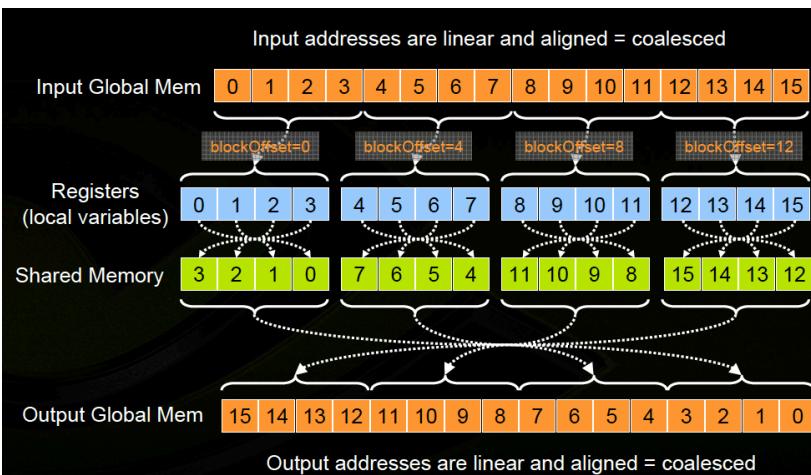
- Por ejemplo, para 4 bloques de 4 hilos, tendríamos:



Manuel Ujaldon - Nvidia CUDA Fellow

194

Versión utilizando la memoria compartida



195

Manuel Ujaldon - Nvidia CUDA Fellow

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

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

    temp[lindex] = in[gindex]; // Pasar el vector de entrada a memoria compartida // (i1)
    syncthreads();
    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.

196

Manuel Ujaldon - Nvidia CUDA Fellow



VI. 4. Producto de matrices



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

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

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

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

199

Manuel Ujaldon - Nvidia CUDA Fellow

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;
        }
}
```

198

Manuel Ujaldon - Nvidia CUDA Fellow

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

- 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.

```
dim2 dimBlock(BLOCKSIZE, BLOCKSIZE);
dim2 dimGrid(AnchuraB/BLOCKSIZE, AlturaA/BLOCKSIZE);
...
MxMonGPU <<<dimGrid, dimBlock>>> (A, B, C, N);
```

200

Manuel Ujaldon - Nvidia CUDA Fellow

Versión CUDA para el producto de matrices: Análisis

- Cada hilo utiliza 10 registros, lo que nos permite alcanzar el mayor grado de paralelismo en Kepler:
 - 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].
- **Problemas:**
 - Baja intensidad aritmética.
 - Exigente en el ancho de banda a memoria, que termina erigiéndose como el cuello de botella para el rendimiento.
- **Solución:**
 - Utilizar la memoria compartida de cada multiprocesador.

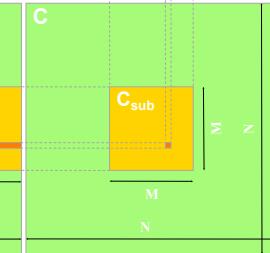
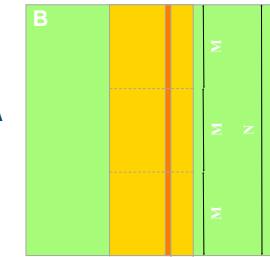
201

Manuel Ujaldon - Nvidia CUDA Fellow



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.



202

Manuel Ujaldon - Nvidia CUDA Fellow



Tiling: Detalles de la implementación

- Tenemos que gestionar todos los mosaicos de fila y columna que necesita cada bloque de hilos:
 - 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.
 - `__syncthreads()` (para asegurarnos que hemos cargado las matrices completamente antes de comenzar la computación).
 - Computar todos los productos y sumas para C utilizando los mosaicos de memoria compartida.
 - Cada hilo puede ahora iterar independientemente sobre los elementos del mosaico.
 - `__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).

203

Manuel Ujaldon - Nvidia CUDA Fellow

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**.

204

Manuel Ujaldon - Nvidia CUDA Fellow



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:
 $a[0][0], a[0][1], a[0][2], \dots$

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

205

Manuel Ujaldon - Nvidia CUDA Fellow

Tiling: El código CUDA para el kernel en GPU

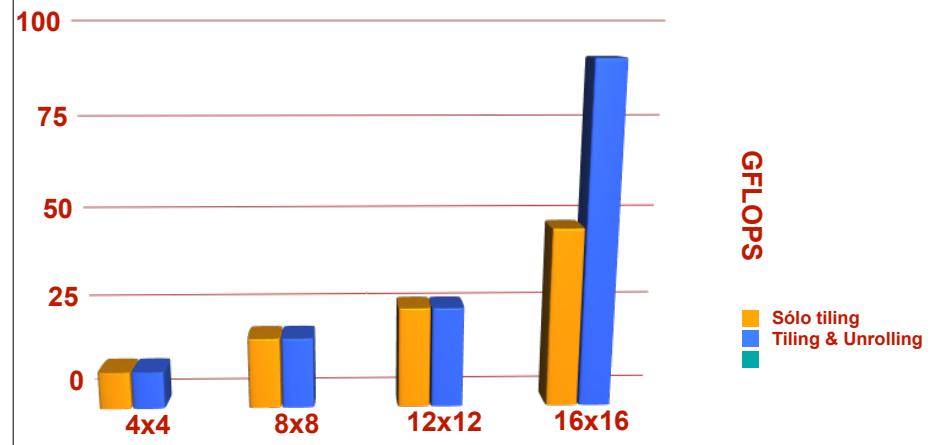
```
__global__ void MxMonGPU(float *A, float *B, float *C, int N)
{
    int sum=0, tx, ty, i, j;
    tx = threadIdx.x; ty = threadIdx.y;
    i = blockIdx.x * blockDim.x + tx; j = blockIdx.y * blockDim.y + ty;
    __shared__ float As[32][32], float Bs[32][32];

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

206

Manuel Ujaldon - Nvidia CUDA Fellow

Rendimiento con tiling & unrolling en la G80



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

208

Manuel Ujaldon - Nvidia CUDA Fellow

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;
```

207

Manuel Ujaldon - Nvidia CUDA Fellow



VII. Bibliografía y herramientas



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.	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 GPU-Accelerated Libraries	Programming Language Develop your own parallel applications and libraries using a programming language you already know. Get Started With: <ul style="list-style-type: none">• C/C++ using CUDA C• Fortran using CUDA Fortran• Python
----------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

211

Manuel Ujaldon - Nvidia CUDA Fellow

CUDA Zone:
La web raíz para el programador CUDA

[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

210

Manuel Ujaldon - Nvidia CUDA Fellow

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.

212

Manuel Ujaldon - Nvidia CUDA Fellow



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 >](#)

213

Manuel Ujaldon - Nvidia CUDA Fellow





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

215

Manuel Ujaldon - Nvidia CUDA Fellow



CUDA Downloads

Get the latest and greatest version of the CUDA Toolkit

[NVIDIA CUDA ZONE](#) Getting Started Downloads Training Ecosystem Forum

Windows Linux x86 Linux POWER Mac OSX

Version	Network Installer	Local Package Installer	Rpmfile Installer
Fedora 21	Coming Soon	Coming Soon	Coming Soon
OpenSUSE 13.2	RPM (3KB)	RPM (108B)	RPM (1.10B)
OpenSUSE 13.1	RPM (3KB)	RPM (108B)	RPM (1.10B)
RHEL 7	RPM (3KB)	RPM (108B)	RPM (1.10B)
RHEL 6	RPM (3KB)	RPM (108B)	RPM (1.10B)
SLES 12	RPM (3KB)	RPM (1.10B)	RPM (1.10B)
SLES 11 SP3	RPM (3KB)	RPM (1.10B)	RPM (1.10B)
SLES 11 SP2	RPM (3KB)	RPM (1.10B)	RPM (1.10B)
SLES 11 SP1	RPM (3KB)	RPM (1.10B)	RPM (1.10B)
Ubuntu 14.10	DEB (3MB)	DEB (1.50B)	DEB (1.10B)
Ubuntu 14.04	DEB (10MB)	DEB (1.2MB)	DEB (1.10B)
Ubuntu 12.04	DEB (3MB)	DEB (1.30B)	DEB (1.10B)
GPU Deployment Kit	Included in Installer	Included in Installer	RPM (1MB)
cuFFT Patch			TAR (122MB), README

Check out:

- CUDA 7 Performance Report and Webinar Recording
- An informative webinar by Ujwal 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 cuStreamNonBlocking flag of the CUDA Runtime API or the CU_STREAM_NON_BLOCKING flag of the CUDA Driver API).

Windows Linux x86 Linux POWER 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	

Documentation Release Notes End User License Agreement Online Documentation CUDA Toolkit Overview Checksums

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

Windows Linux x86 Linux POWER Mac OSX

Version	Network Installer	Local Package Installer	Rpmfile Installer
Ubuntu 14.10	DEB (3MB)	DEB (1.6MB)	
Ubuntu 14.04	DEB (3MB)	DEB (1.6MB)	
GPU Deployment Kit	n/a	n/a	RPM (1.2MB), README
cuFFT Patch			TAR (110MB), README

Windows Linux x86 Linux POWER Mac OSX

Version	Network Installer	Local Installer
Ubuntu 10.10	DEB (3MB)	DEB (1.7MB)
Ubuntu 10.10	DEB (3MB)	DEB (1.7MB)
cuFFT Patch		

Manuel Ujaldon - Nvidia CUDA Fellow

214

Libros sobre CUDA: Desde 2007 hasta 2015

La serie GPU Gems: 1, 2 y 3 [developer.nvidia.com/gpugems]

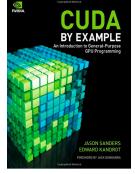
Una lista de libros de CUDA [developer.nvidia.com/suggested-reading]



Sep'07



Feb'10



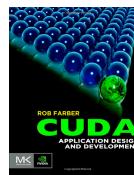
Jul'10



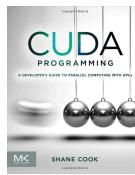
Abr'11



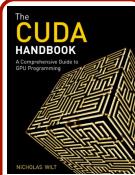
Oct'11



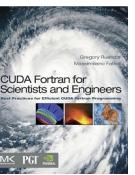
Nov'11



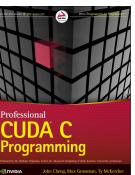
Dic'12



Jun'13



Oct'13



Sep'14

216

Manuel Ujaldon - Nvidia CUDA Fellow

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).



Manuel Ujaldon - Nvidia CUDA Fellow

217

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

- [\[developer.nvidia.com/cuda-education-training\]](https://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

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!](#)

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

Tweets by @GPUComputing [Follow](#)

Manuel Ujaldon - Nvidia CUDA Fellow

218

Cursos on-line de acceso gratuito

Impartidos por reputados pedagogos:

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

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.
- [\[https://developer.nvidia.com/udacity-cs344-intro-parallel-programming\]](https://developer.nvidia.com/udacity-cs344-intro-parallel-programming)



UDACITY

Programación paralela heterogénea (en UIUC): [coursera](#)

- 9 semanas, cada una con teoría (vídeos de 20 minutos), desafíos o ejercicios
- [\[https://www.coursera.org/course/hetero\]](https://www.coursera.org/course/hetero) (temporalmente descatalogado).
- Enlace alternativo en UIUC: [\[https://courses.engr.illinois.edu/ece408/index.html\]](https://courses.engr.illinois.edu/ece408/index.html)



Manuel Ujaldon - Nvidia CUDA Fellow

219

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 a list of short tutorials for C/C++, Fortran, and Python. Each tutorial card includes a thumbnail, title, duration, and access level. The titles include "NVIDIA CUDA GPU Optimization", "OpenACC", "NVIDIA CUDA", "Accelerating Applications with GPU-Accelerated Libraries", and "CUDA Cloud Training". The access levels range from "Free" to "\$29.99".

Title	Duration	Access Level
NVIDIA CUDA GPU Optimization	1 hr 30 m	Free
OpenACC	2 hr 45 m	Free
NVIDIA CUDA	1 hr 30 m	\$29.99
Accelerating Applications with GPU-Accelerated Libraries	1 hr 30 m	\$29.99
CUDA Cloud Training	1 hr 30 m	Free

Manuel Ujaldon - Nvidia CUDA Fellow

220

Charlas y webinarios

- Charlas grabadas @ GTC (Graphics Technology Conference):
 - Más de 500 charlas en cada una de las últimas ediciones (2013-16):
 - [\[www.gputechconf.com/gtcnew/on-demand-gtc.php\]](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)



Manuel Ujaldon - Nvidia CUDA Fellow

221

Desarrolladores (2)

- Listado oficial de GPUs que soportan CUDA:
 - [\[developer.nvidia.com/cuda-gpus\]](http://developer.nvidia.com/cuda-gpus)

	CUDA-Enabled Tesla Products
	CUDA-Enabled Quadro Products
	CUDA-Enabled NVS Products
	CUDA-Enabled GeForce Products
	CUDA-Enabled TEGRA / Jetson Products

- Y una última herramienta: El CUDA Occupancy Calculator
 - [\[developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls\]](http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls)



Manuel Ujaldon - Nvidia CUDA Fellow

223

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.
- Noticias y eventos de GPGPU (cada vez menos actualizado):
 - [\[www.gpgpu.org\]](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)



Manuel Ujaldon - Nvidia CUDA Fellow

222

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.



Manuel Ujaldon - Nvidia CUDA Fellow

224