

Fundamentals of Accelerated Computing with CUDA C/C++

Workshop del Deep Learning Institute
Universidad de Málaga



Manuel Ujaldón

Catedrático de Universidad
Departamento de Arquitectura de Computadores
Universidad de Málaga



DEEP
LEARNING
INSTITUTE

UNIVERSITY
AMBASSADOR

La ficha del curso

- **Duración:** 8-10 horas.
- **Certificado:** Los participantes que superen la prueba de evaluación del curso recibirán un certificado del DLI de Nvidia que acredita sus competencias en esta materia y fortalece el crecimiento de su carrera profesional.
- **Prerrequisitos:** Conocimientos básicos de lenguaje C, como tipos de variables, bucles, sentencias condicionales, funciones y manejo de *arrays*. No se requieren conocimientos previos de CUDA.
- **Herramientas, librerías y entornos de trabajo:** nsys, Nsight (profilers).

Objetivos del curso

- Comprender las herramientas fundamentales y las técnicas de la aceleración en GPU que son aplicables a los códigos en lenguaje C y C++, y adquirir competencias para:
 - Escribir código para que se acelere en una GPU.
 - Exponer y expresar el paralelismo de datos en aplicaciones escritas en C utilizando CUDA.
 - Manejar la memoria de CUDA y optimizar el movimiento de datos entre CPU y GPU por medio de la prebúsqueda asíncrona.
 - Utilizar las *streams* o flujos concurrentes (paralelismo de tareas).
 - Desenvolverse con los *profilers* para optimizar la aceleración.

Agenda: Primer día (turno de tarde)

Día	Hora	Módulo	Objetivos
1	16:00 - 17:40	Introducción al hardware de la GPU	Comprender el chip GPU y su arquitectura, bloques constructivos, generaciones y modelos.
1	17:40 - 18:00	Descanso	
1	18:00 - 19:45	Aceleración de Aplicaciones con CUDA C/C++	Conocer la sintaxis esencial y los conceptos que permiten escribir aplicaciones C/C++ con CUDA: (1) Escribir, compilar y ejecutar código GPU. (2) Controlar la jerarquía de hilos paralelos, su organización y la sincronización CPU-GPU. (3) Lanzar kernels básicos.
1	19:45 - 20:00	Descanso	
1	20:00 - 21:00	Aceleración de Aplicaciones con CUDA C/C++	Aplicar paralelismo de datos a un código secuencial: (1) Distinguir los procedimientos que admiten mejor paralelización. (2) Reconocer las dependencias del código que afectan al paralelismo. (3) Manejar las comunicaciones entre la CPU y la GPU.

Agenda: Primer día (turno de mañana)

Día	Hora	Módulo	Objetivos
1	09:00 - 10:40	Introducción al hardware de la GPU	Comprender el chip GPU y su arquitectura, bloques constructivos, generaciones y modelos.
1	10:40 - 11:00	Descanso	
1	11:00 - 12:45	Aceleración de Aplicaciones con CUDA C/C++	Conocer la sintaxis esencial y los conceptos que permiten escribir aplicaciones C/C++ con CUDA: <ul style="list-style-type: none">(1) Escribir, compilar y ejecutar código GPU.(2) Controlar la jerarquía de hilos paralelos, su organización y la sincronización CPU-GPU.(3) Lanzar kernels básicos.
1	12:45 - 13:00	Descanso	
1	13:00 - 14:00	Aceleración de Aplicaciones con CUDA C/C++	Aplicar paralelismo de datos a un código secuencial: <ul style="list-style-type: none">(1) Distinguir los procedimientos que admiten mejor paralelización.(2) Reconocer las dependencias del código que afectan al paralelismo.(3) Manejar las comunicaciones entre la CPU y la GPU.

Nuestra hoja de ruta del primer día (turno de tarde)

Hora	Contenido	Duración	Autor	Disponibilidad del material en:
16:00	Presentación y agenda de trabajo	10'	M.U.	Nuestra web: nvidiaDLI.uma.es
16:10	Introducción a CUDA	30'	M.U.	C.V.: PDF y grabación 1
16:40	Lote 1/6 de diapositivas	10'	DLI	C.V.: PDF y grabación 2 / DLI
16:50	Ejercicios 1.0 a 1.5	20'	DLI	DLI: Guión en <i>jupyter notebook</i>
17:10	- Resolvemos ejercicios 1.0 a 1.5	10'	Ambos	C.V.: PDF de sols. / DLI: Sols
17:20	Hardware CUDA	30'	M.U.	C.V.: PDF y grabación 3
17:50	Primer descanso	20'		
18:10	Programación CUDA	30'	M.U.	C.V.: PDF y grabación 4
18:40	Lote 2/6 de diapositivas	10'	DLI	C.V.: PDF y grabación 5 / DLI
18:50	Ejercicios 2.1 a 2.3	10'	DLI	DLI: Guión en <i>jupyter notebook</i>
19:00	- Resolvemos ejercicios 2.1 a 2.3	10'	Ambos	C.V.: PDF de sols. / DLI: Sols.
19:10	Lote 3/6 de diapositivas	10'	DLI	C.V.: PDF y grabación 6 / DLI
19:20	Ejercicios 3 y 4	15'	DLI	DLI: Guión en <i>jupyter notebook</i>
19:35	- Resolvemos ejercicios 3 y 4	5'	Ambos	C.V.: PDF de sols. / DLI: Sols.
19:40	Segundo descanso	20'		
20:00	Lote 4/6 de diapositivas	10'	DLI	C.V.: PDF y grabación 7 / DLI
20:10	Ejercicio 5	10'	DLI	DLI: Guión en <i>jupyter notebook</i>
20:20	- Resolvemos ejercicio 5	5'	Ambos	C.V.: PDF de sols. / DLI: Sols.
	Lotes 5/6 y 6/6 de diapositivas	Self-paced	DLI	C.V.: PDF / DLI
	Ejercicios 6, 7, 8 y 9	Self-paced	DLI	DLI: Guión y soluciones
20:25	Paralelización de 5 kernels CUDA	20'	M.U.	C.V.: PDF y grabación 8
20:45	Final Exercise: VectorAdd	10'	DLI	DLI: Guión en <i>jupyter notebook</i>
20:55	- Resolvemos VectorAdd	5'	Ambos	C.V.: PDF de sols. / DLI: Sols.
	Gestión de errores	Self-paced	DLI	DLI: Guión de contenidos
	Advanced Exercise: MatrixMul	Self-paced	Ambos	C.V.: PDF de ideas / DLI: Guión
	- Se propone como reto	Self-paced	DLI	DLI: Soluciones
	Advanced Ex. 2: Heat Conduction	Self-paced	DLI	DLI: Guión en <i>jupyter notebook</i>
	- Lo resolvemos el próximo día	10'	Ambos	C.V.: PDF de sols. / DLI: Sols.

Nuestra hoja de ruta del primer día (turno de mañana)

Hora	Contenido	Duración	Autor	Disponibilidad del material en:
9:00	Presentación y agenda de trabajo	10'	M.U.	Nuestra web: nvidiaDLI.uma.es
9:10	Introducción a CUDA	30'	M.U.	C.V.: PDF y grabación 1
9:40	Lote 1/6 de diapositivas	10'	DLI	C.V.: PDF y grabación 2 / DLI
9:50	Ejercicios 1.0 a 1.5	20'	DLI	DLI: Guión en <i>jupyter notebook</i>
10:10	- Resolvemos ejercicios 1.0 a 1.5	10'	Ambos	C.V.: PDF de sols. / DLI: Sols
10:20	Hardware CUDA	30'	M.U.	C.V.: PDF y grabación 3
10:50	Primer descanso	20'		
11:10	Programación CUDA	30'	M.U.	C.V.: PDF y grabación 4
11:40	Lote 2/6 de diapositivas	10'	DLI	C.V.: PDF y grabación 5 / DLI
11:50	Ejercicios 2.1 a 2.3	10'	DLI	DLI: Guión en <i>jupyter notebook</i>
12:00	- Resolvemos ejercicios 2.1 a 2.3	10'	Ambos	C.V.: PDF de sols. / DLI: Sols.
12:10	Lote 3/6 de diapositivas	10'	DLI	C.V.: PDF y grabación 6 / DLI
12:20	Ejercicios 3 y 4	15'	DLI	DLI: Guión en <i>jupyter notebook</i>
12:35	- Resolvemos ejercicios 3 y 4	5'	Ambos	C.V.: PDF de sols. / DLI: Sols.
12:40	Segundo descanso	20'		
13:00	Lote 4/6 de diapositivas	10'	DLI	C.V.: PDF y grabación 7 / DLI
13:10	Ejercicio 5	10'	DLI	DLI: Guión en <i>jupyter notebook</i>
13:20	- Resolvemos ejercicio 5	5'	Ambos	C.V.: PDF de sols. / DLI: Sols.
	Lotes 5/6 y 6/6 de diapositivas	Self-paced	DLI	C.V.: PDF / DLI
	Ejercicios 6, 7, 8 y 9	Self-paced	DLI	DLI: Guión y soluciones
13:25	Paralelización de 5 kernels CUDA	20'	M.U.	C.V.: PDF y grabación 8
13:45	Final Exercise: VectorAdd	10'	DLI	DLI: Guión en <i>jupyter notebook</i>
13:55	- Resolvemos VectorAdd	5'	Ambos	C.V.: PDF de sols. / DLI: Sols.
	Gestión de errores	Self-paced	DLI	DLI: Guión de contenidos
	Advanced Exercise: MatrixMul	Self-paced	Ambos	C.V.: PDF de ideas / DLI: Guión
	- Se propone como reto	Self-paced	DLI	DLI: Soluciones
	Advanced Ex. 2: Heat Conduction	Self-paced	DLI	DLI: Guión en <i>jupyter notebook</i>
	- Lo resolvemos el próximo día	10'	Ambos	C.V.: PDF de sols. / DLI: Sols.



CONFIGURA TU EQUIPO Y TU CUENTA DLI

1. Comprueba que los WebSockets funcionan para tí:

Para ello, valida tu equipo en: <https://websocketstest.com>

- Bajo ENVIRONMENT, confirma que la fila Websockets supported está en “yes”.
- Bajo WEBSOCKETS (Port 443, SSL), confirma que Data Receive, Send y Echo Test están todos en estado “yes”.

Si tuvieras algún problema con WebSockets, trata de actualizar tu navegador. Recomendamos (1) Chrome, (2) Firefox y (3) Safari.

2. Regístrate como alumno DLI en: <https://courses.nvidia.com/join> (escribe tu nombre tal y como quieras que aparezca en tu certificado)

3. Accede gratuitamente al curso en: <https://courses.nvidia.com/dli-event>

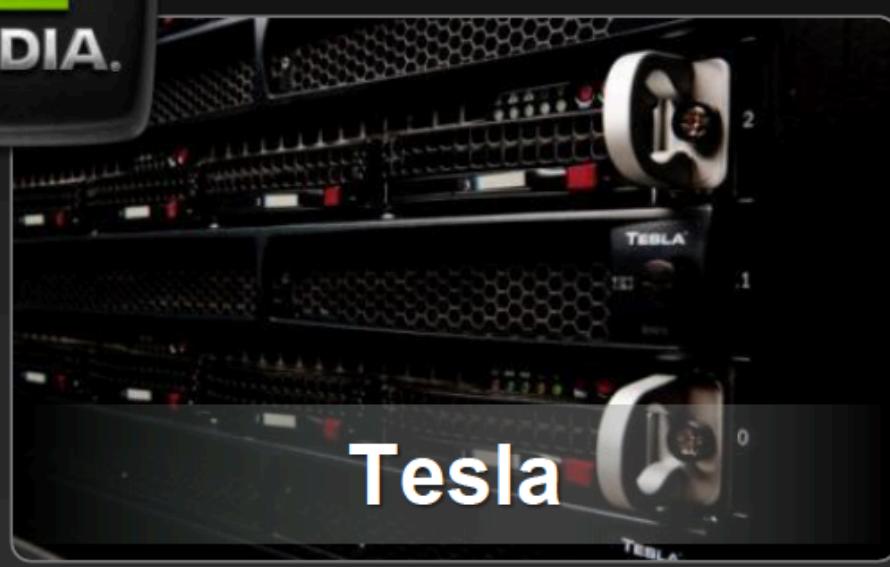
Índice de contenidos

1. Introducción [9 diapositivas]
2. El hardware de CUDA [28]
 1. El modelo global [3]
 2. La arquitectura Volta [6]
 3. La arquitectura Turing [2]
 4. La arquitectura Ampere [4]
 5. La arquitectura Hopper [10]
 6. Resumen generacional [3]
3. Programación CUDA [10]
4. Cinco *kernels* característicos [6]



I. Introducción

Bienvenido al mundo de las GPUs



El ecosistema de CUDA

GPU Computing Applications						
Libraries and Middleware						
cuDNN TensorRT	cuFFT, cuBLAS, cuRAND, cuSPARSE	CULA MAGMA	Thrust NPP	VSIPL, SVM, OpenCurrent	PhysX, OptiX, iRay	MATLAB Mathematica
Programming Languages						
C	C++	Fortran	Java, Python, Wrappers	DirectCompute	Directives (e.g., OpenACC)	
CUDA-enabled NVIDIA GPUs						
Turing Architecture (Compute capabilities 7.x)	DRIVE/JETSON AGX Xavier	GeForce 2000 Series		Quadro RTX Series	Tesla T Series	
Volta Architecture (Compute capabilities 7.x)	DRIVE/JETSON AGX Xavier					Tesla V Series
Pascal Architecture (Compute capabilities 6.x)	Tegra X2	GeForce 1000 Series		Quadro P Series	Tesla P Series	
Maxwell Architecture (Compute capabilities 5.x)	Tegra X1	GeForce 900 Series		Quadro M Series	Tesla M Series	
Kepler Architecture (Compute capabilities 3.x)	Tegra K1	GeForce 700 Series GeForce 600 Series		Quadro K Series	Tesla K Series	
	EMBEDDED	CONSUMER DESKTOP, LAPTOP		PROFESSIONAL WORKSTATION	DATA CENTER	

Este curso ha elegido el lenguaje C como base.

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

● Simplicidad.

- El control de un hilo se amortiza en otros 31 (**warp size = 32**).

● Escalabilidad.

- Aprovechándose del gran **volumen de datos** que manejan las aplicaciones, se define un modelo de paralelización sostenible.

● Productividad.

- Se habilitan multitud de mecanismos para que cuando un hilo pase a realizar operaciones que no permitan su ejecución veloz, otro **oculte su latencia** tomando el procesador **de forma inmediata**.

● Palabras clave esenciales para CUDA:

- Warp, SIMT, ocultación de latencia, commutación de contexto gratis.

¿Qué es CUDA?

“Compute Unified Device Architecture”

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

Lo esencial de CUDA C

- En general, es lenguaje C con mínimas extensiones:

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

- CUDA define:

- Un modelo de arquitectura:

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

- Un modelo de programación:

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

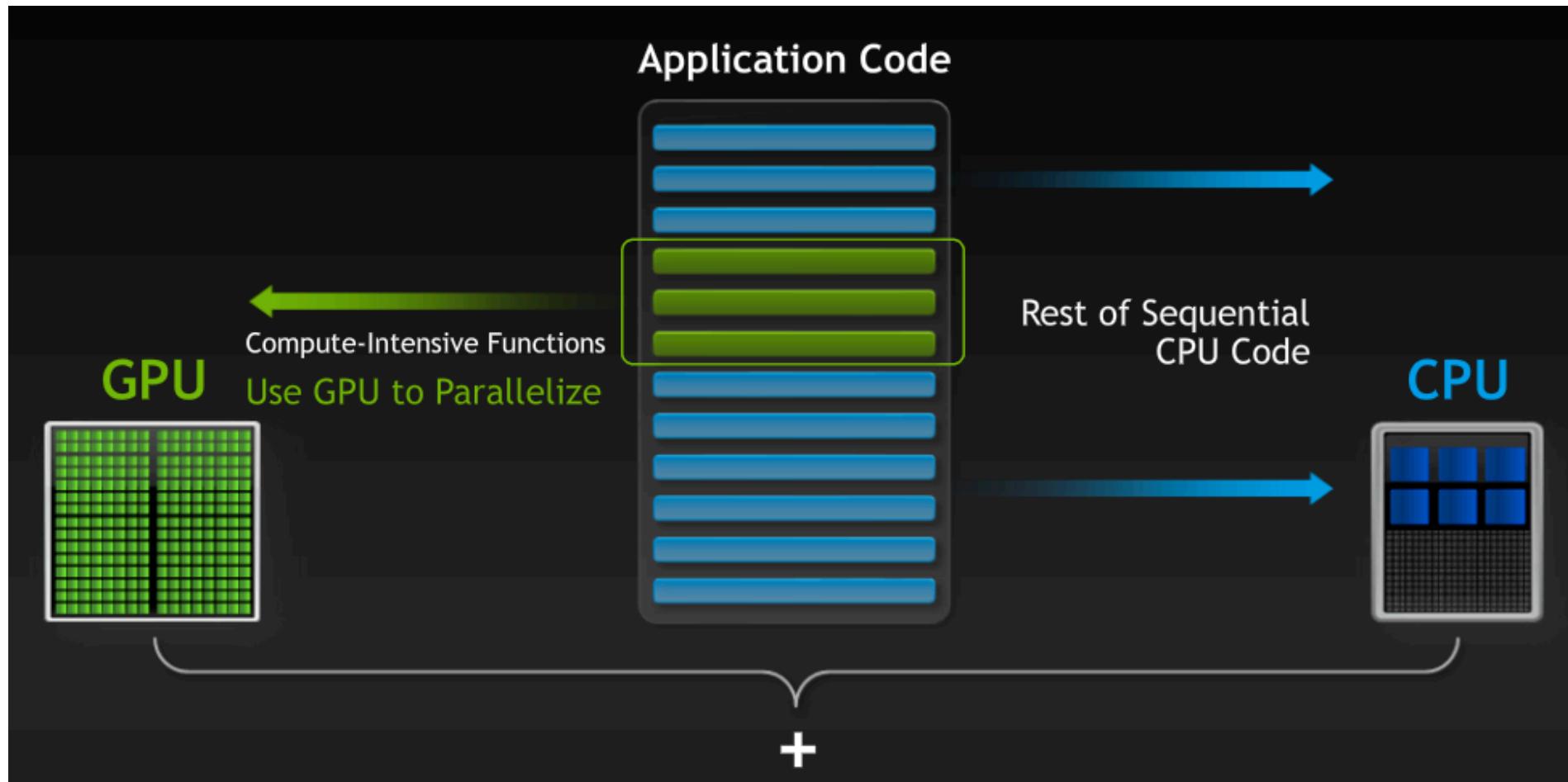
- Un modelo de gestión de la memoria:

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

- Objetivos:

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

Computación heterogénea



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

El código

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

using namespace std;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

CODIGO DEL DISPOSITIVO:
Función paralela escrita en CUDA.

CODIGO DEL HOST:
- Código serie.

} - Código paralelo.
- Código serie.



Mi primer programa en GPU (*device code*)

```
__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, cl.exe para Windows).

Cambiando el programa para saludar desde la CPU

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

Salida:

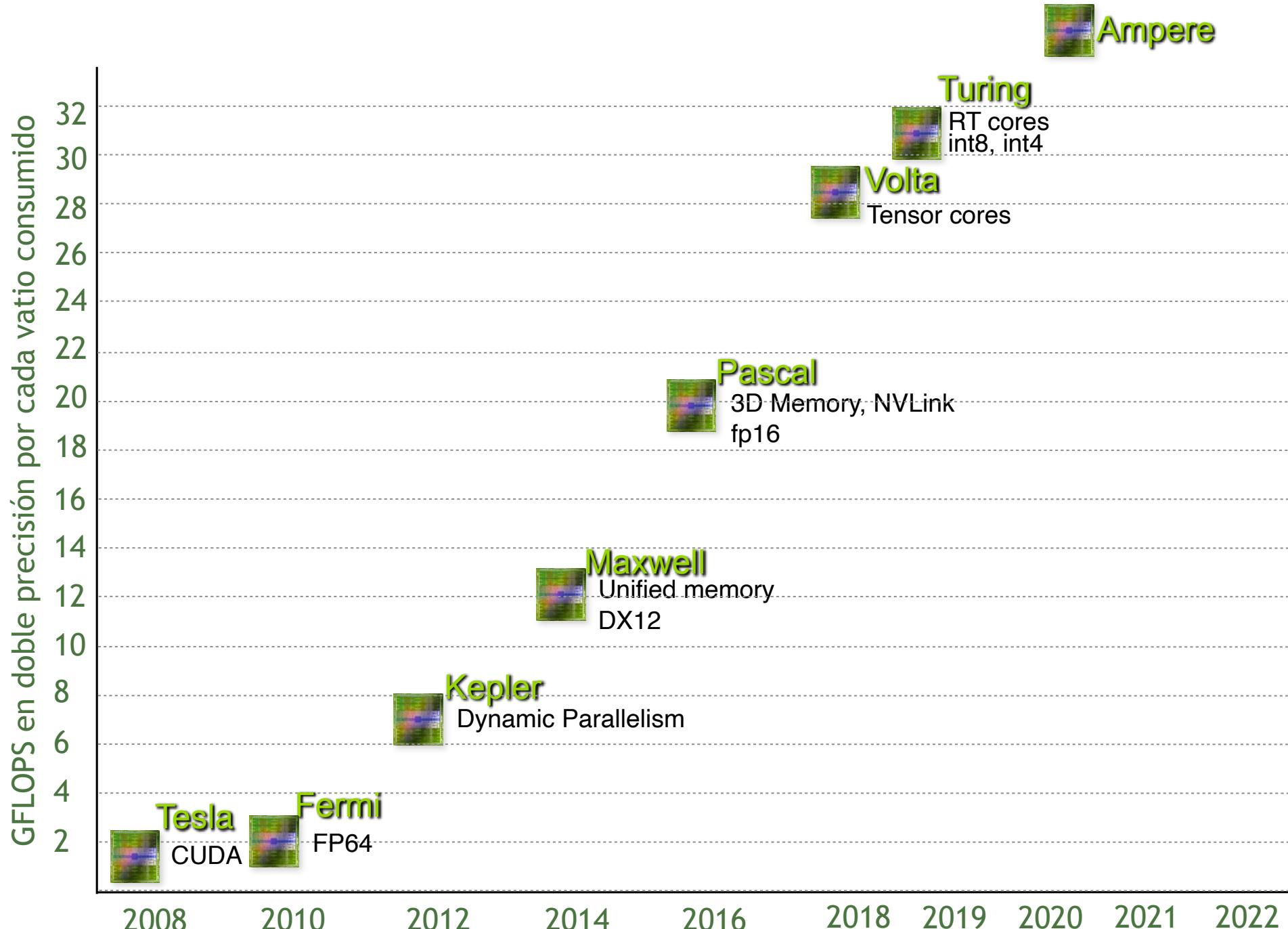
```
$ nvcc hola.cu
$ a.out
¡Hola mundo!
$
```

- `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).



II. El hardware de CUDA

Las 8 generaciones hardware de CUDA



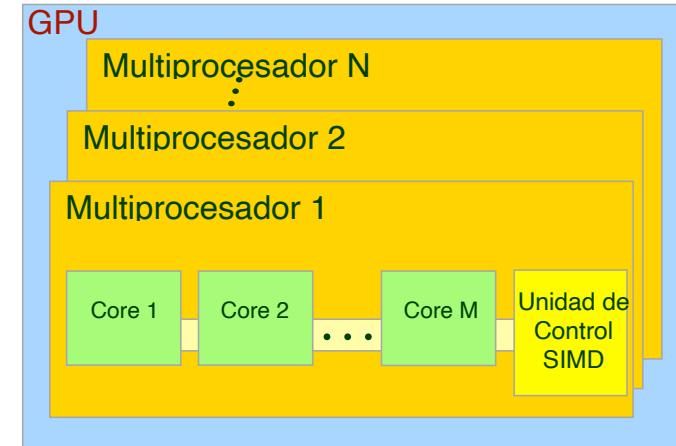
El modelo hardware de CUDA: Un conjunto de procesadores SIMD

- La GPU consta de:

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

- Computación heterogénea:

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



	G80 (Tesla)	GF100 (Fermi)	GK110 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)	GV100 (Volta)	TU102 (Turing)	A100 (Ampere)	H100 (Hopper)
Marco temporal	2006-09	2010-11	2012-13	2014-15	2016-17	2018-20	2019-20	2020-22	2022-?
N (multiprocesadores)	16-30	14-16	13-15	4-24	56	80	72	108	132
M (fp32 cores/multip.)	8	32	192	128	64	64	64	64	128
# cores	128-240	448-512	2496-2880	512-3072	3584	5120	4608	6912	16896

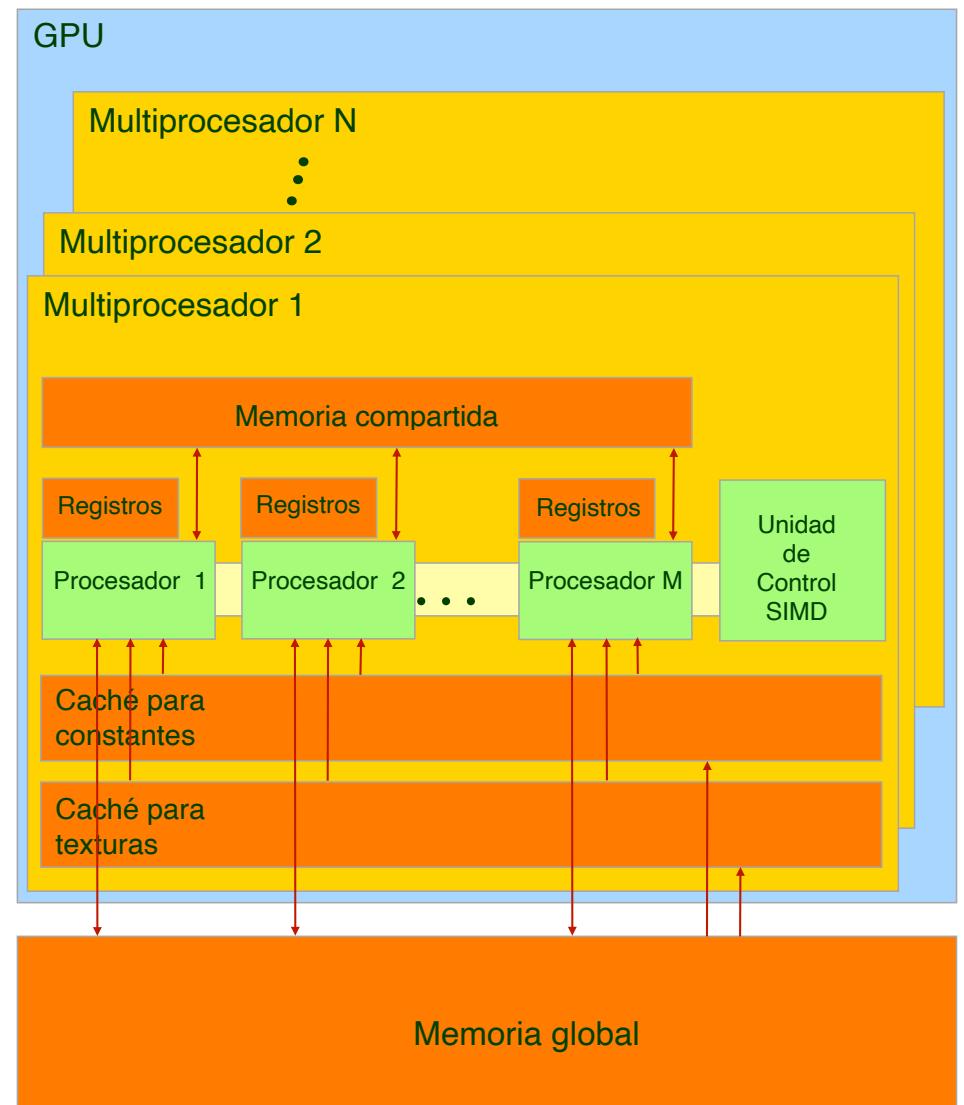
Jerarquía de memoria

- Cada multiprocesador tiene:

- Su banco de registros.
- Memoria compartida.
- Una caché de constantes y otra de texturas, ambas de sólo lectura y uso marginal.

- La memoria global es la memoria de vídeo (GDDR5):

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



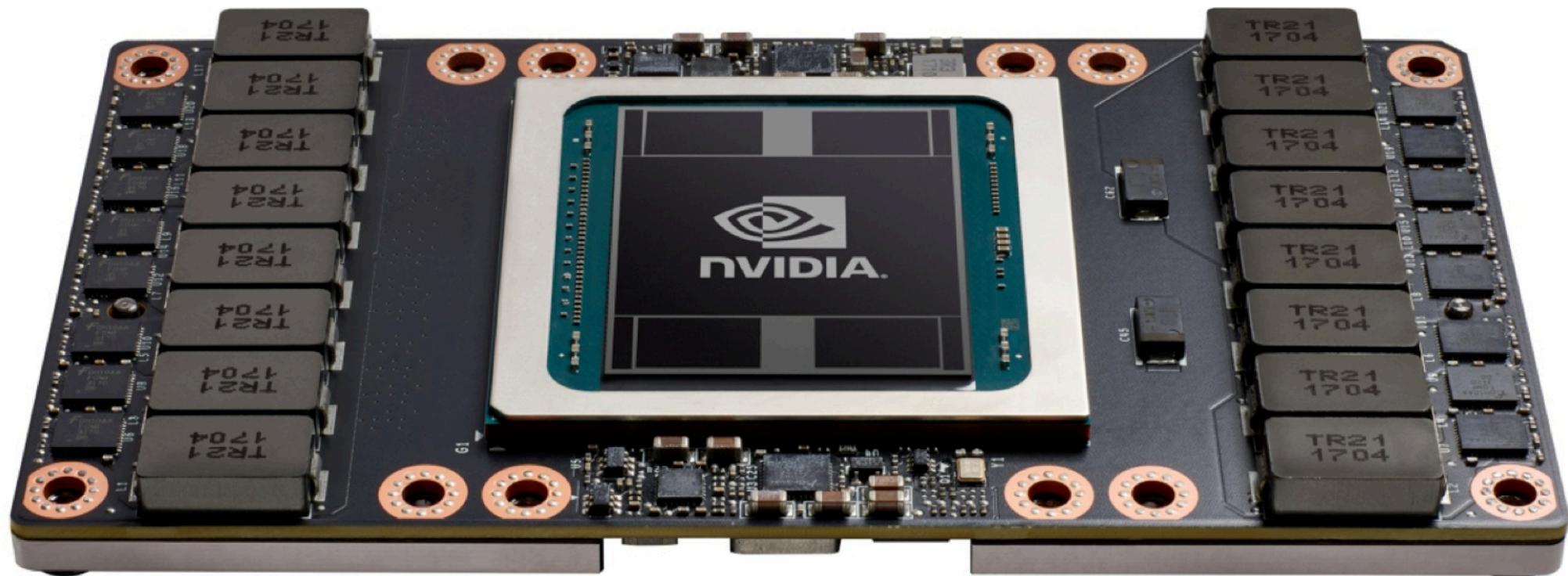


II. 1. Sexta generación: Volta (GVxxx)

Evolución desde la tercera a la sexta generación

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

Aspecto externo del producto comercial



La GPU GV100: 84 SMs y 8 controladores de memoria de 512 bits (Tesla V100 sólo usa 80 SMs)



El multiprocesador de Volta

Cores:

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

Unidades de almacen.:

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

Memoria:

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



Evolución del multiprocesador: Desde Pascal a Volta



Estructura del core tensor (8 por multiprocesador)

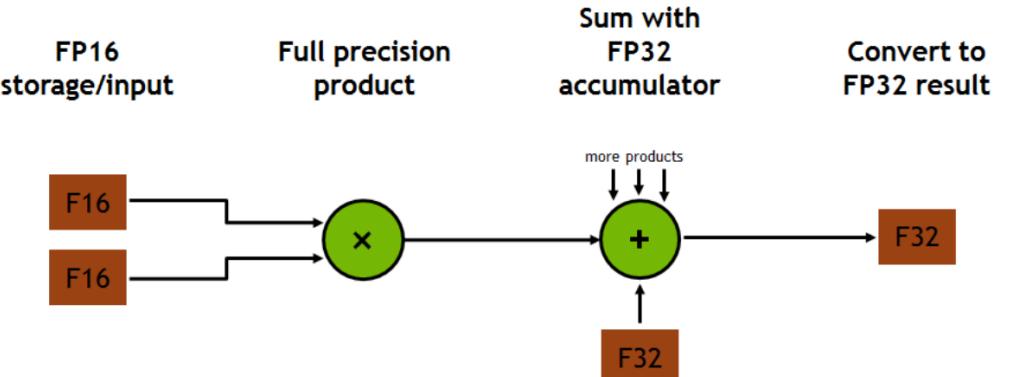
- Matriz de procesamiento 4x4x4 para computar $D = A^*B + C$.

$$D = \left(\begin{array}{cccc} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{array} \right) \left(\begin{array}{cccc} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{array} \right) + \left(\begin{array}{cccc} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{array} \right)$$

FP16 or FP32 FP16 FP16 FP16 or FP32

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

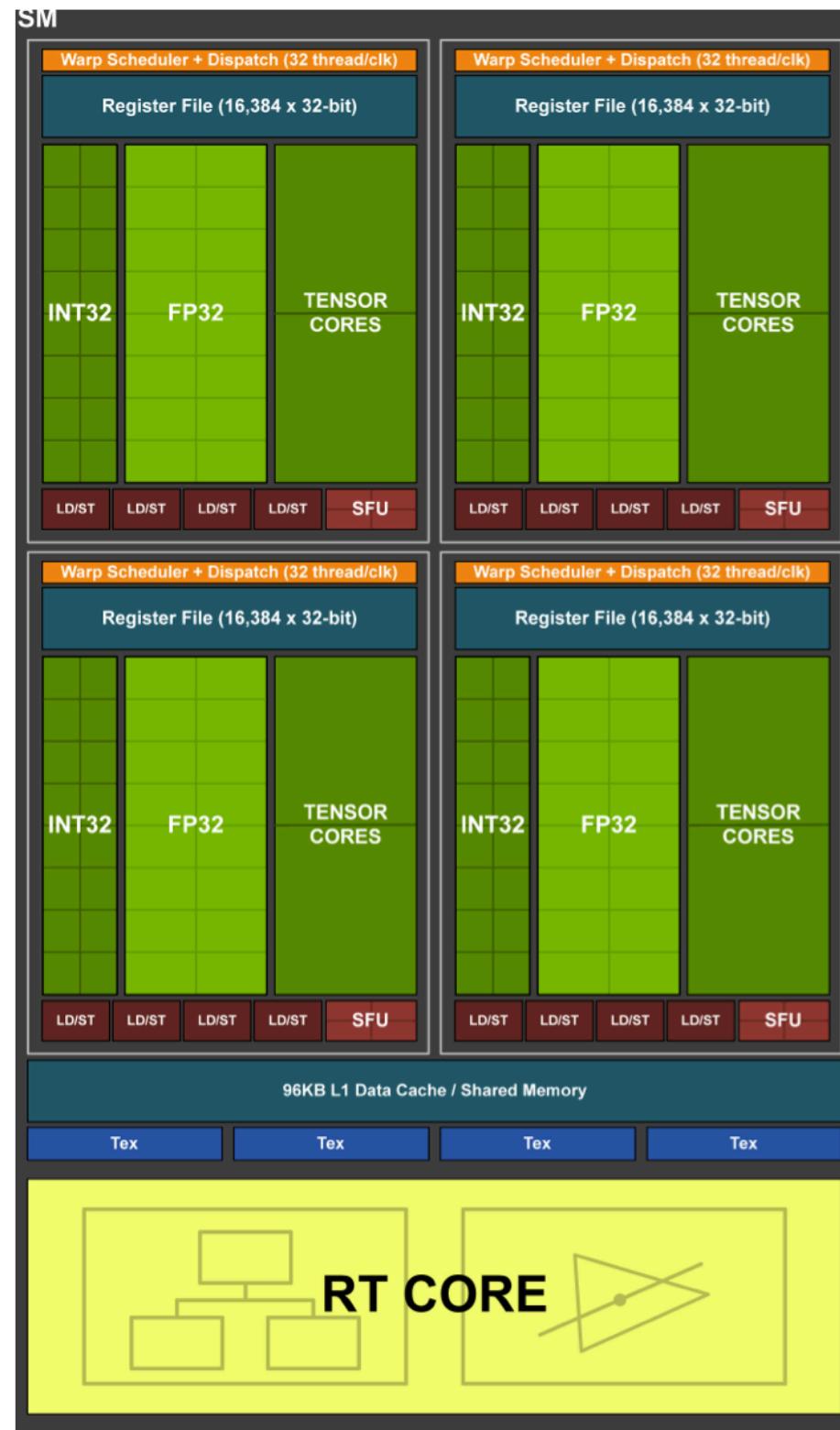
- Dos matrices de entrada de precisión mixta (fp16) con producto de precisión simple (fp32).
- Acumulación y resultado final precisión simple (fp32).



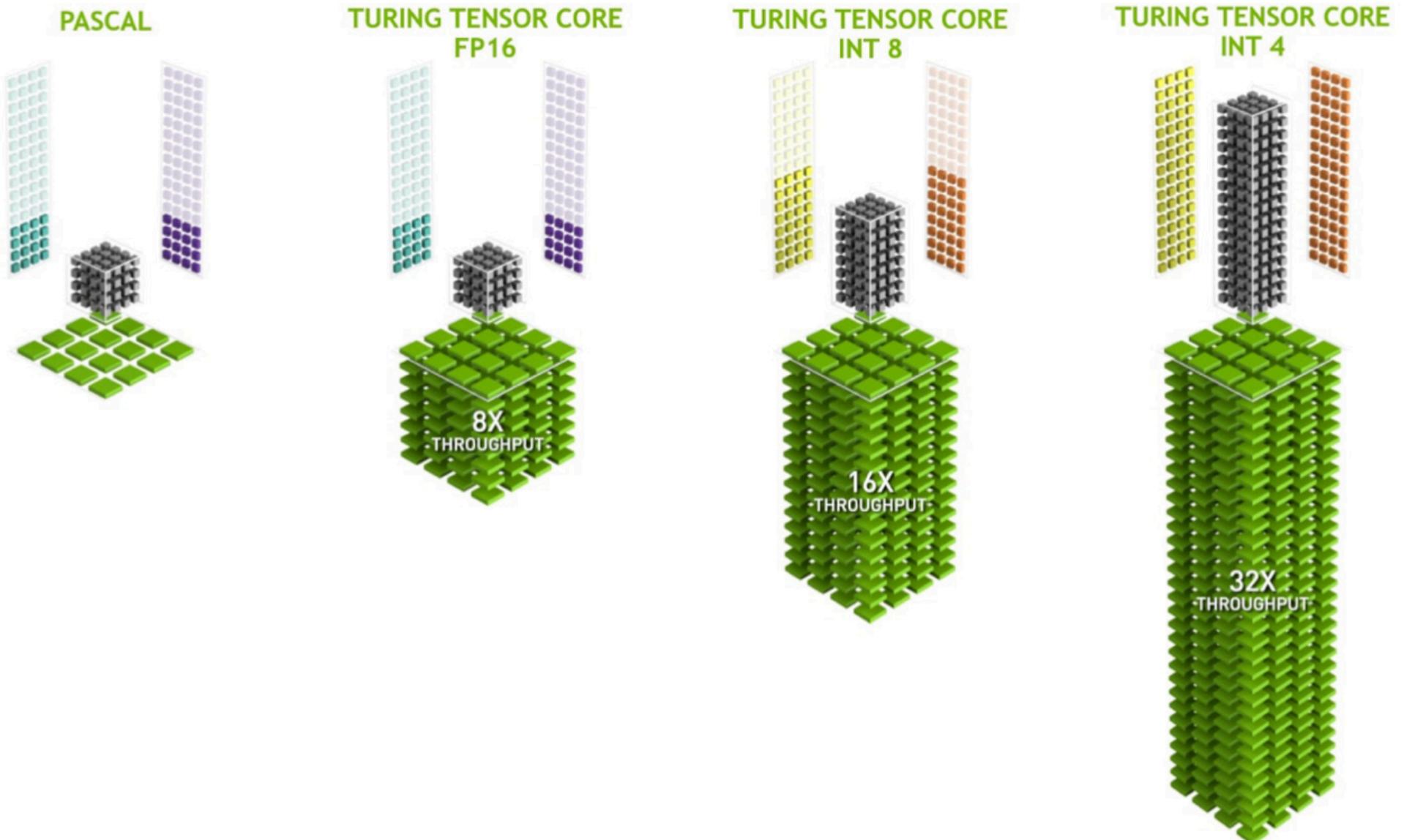


II. 2. Turing (TUxxx)

El multiprocesador Turing



Precisión INT8 y INT4 en los cores Tensor





II. 3. Séptima generación: Ampere (Axxx)

Comparación con generaciones previas

	Tesla V100 (Volta)	GeForce Titan RTX & Quadro RTX 6000 (Turing)	Full implementation of Ampere (GA100)	First commercial Ampere model (A100 Tensor Core)
GPU (chip)	GV100	TU102	GA100	GA100
Million of transistors	21100	18600	54200	54200
Die size	815 mm ²	754 mm ²	826 mm ²	826 mm ²
Manufacturing process	12 nm. FinFET	12 nm. FinFET	7 nm. N7	7 nm. N7
Thermal Design Power	300 W.	280 W.	400 W.	400 W.
Number of fp32 cores	5120 (80 x 64)	4608 (72 x 64)	8192 (128 x 64)	6912 (108 x 64)
Number of fp64 cores	2560	144	4096	3456
Frequency (base-boost)	1370-1455 MHz	1440-1770 MHz	1410 MHz	1410 MHz
TFLOPS (fp16, fp32, fp64)	30, 15, 7.5	32.6, 16.3, 0.51	92, 23, 11.5	78, 19.5, 9.7
Memory interface	HBM2 4 stacks (4096 bits)	GDDR6 384 bits @ 14 GHz	HBM2 6 stacks (6144 bits)	HBM2 5 stacks (5120 bits)
Memory bandwidth	900 GB/s	672 GB/s	1866 GB/s	1555 GB/s
Video memory	16 or 32 GB	24 GB	48 GB	40 GB
L2 cache	6 MB	6 MB	40 MB	40 MB
Shared memory / SM	96 KB (of 128)	64 KB (out of 96)	192 KB	192 KB

La memoria HBM2

- La GPU A100 GPU incluye 40 GB de memoria HBM2.
- La memoria se organiza en 5 torres HBM2 activas con 8 planos en cada torre (de 16 bytes cada uno). Con una frecuencia de 1215 MHz (DDR), la memoria HBM2 de A100 proporciona 1555 GB/sg:
 - $2 \times 1215 \text{ MHz} \times 5 \text{ stacks} \times 2 \text{ memory controllers / stack} \times 512 \text{ bits / memory controller} = 1555.2 \text{ GB/s}$.
 - Cada torre contribuye con 8 GB, y la implementación completa de la GA100 tiene 6 torres en lugar de 5, para llegar a 48 GB y 1866 GB/s.

GA100 completa con 128 SMs & 6 HBM2

(A100 tiene 108 SMs y 5 memorias apiladas HBM2)



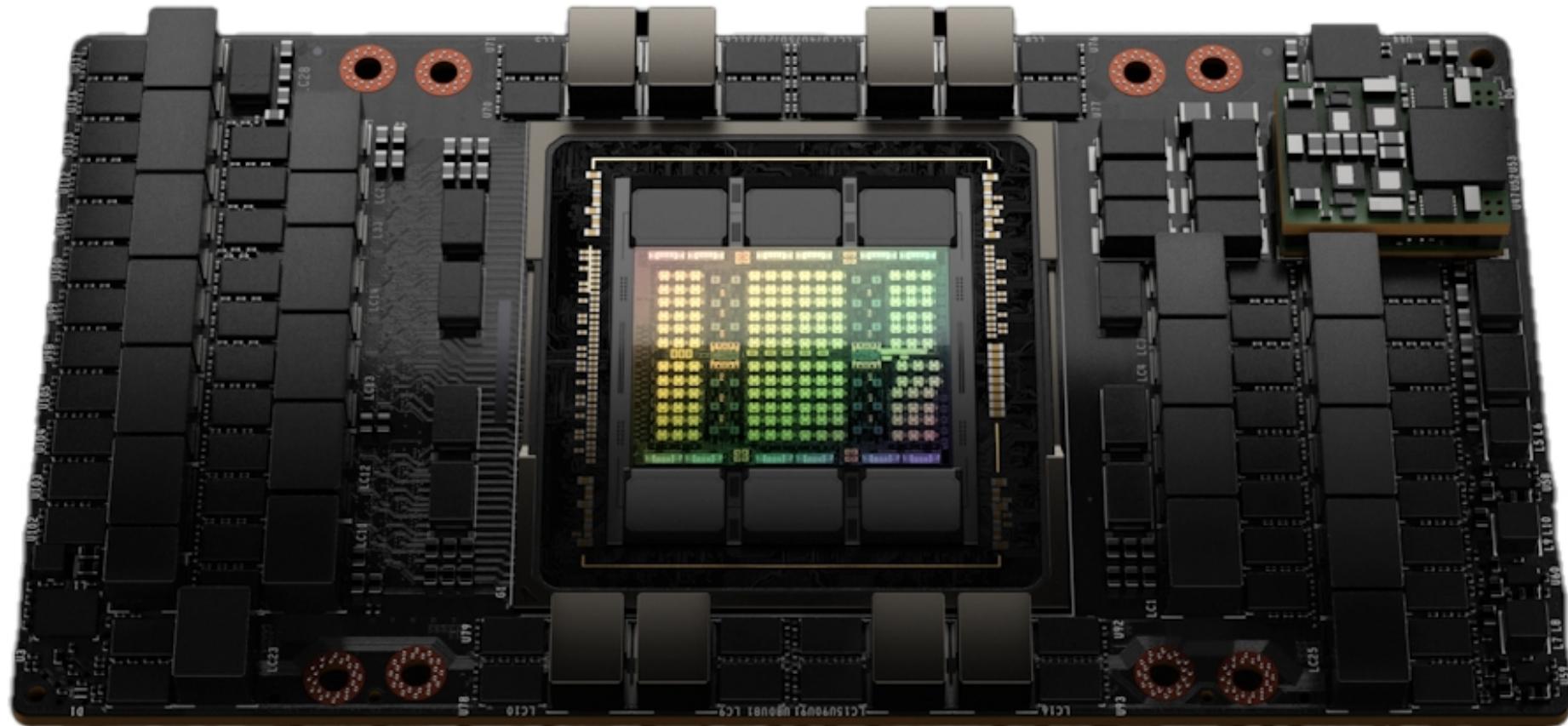
GA100 Streaming Multiprocessor





II. 4. Octava generación: Hopper (Hxxx)

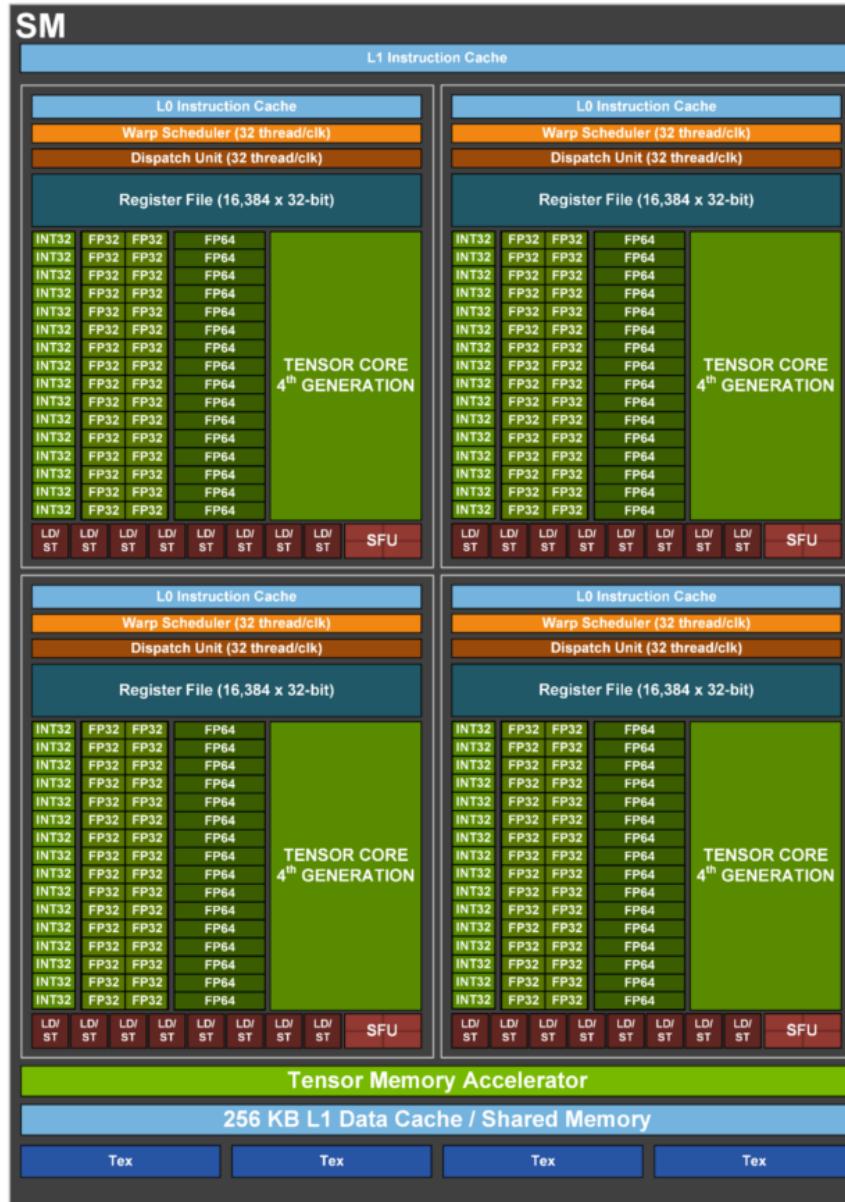
Aspecto físico de la placa de circuito impreso



La GPU GH100 con 144 SMs y 6 pilas HBM3



El multiprocesador de la GPU GH100

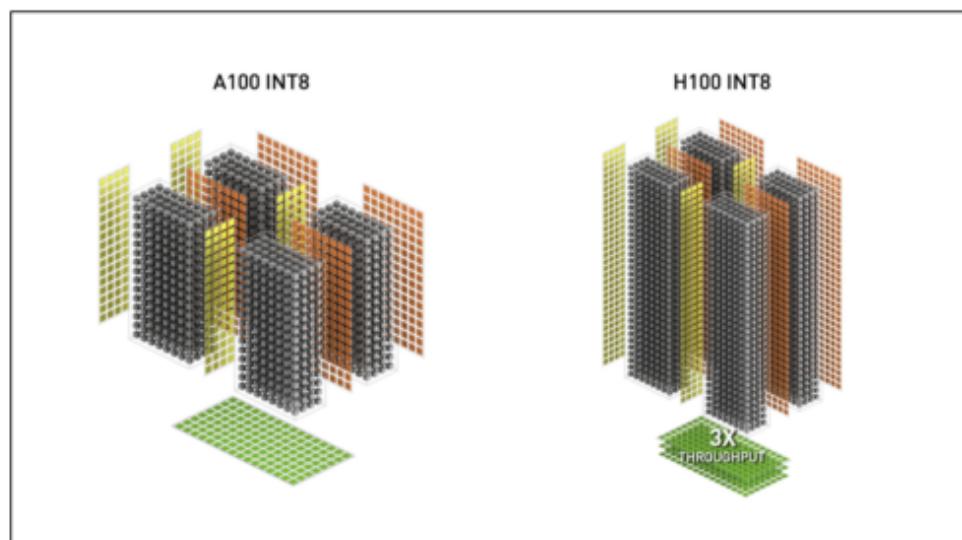
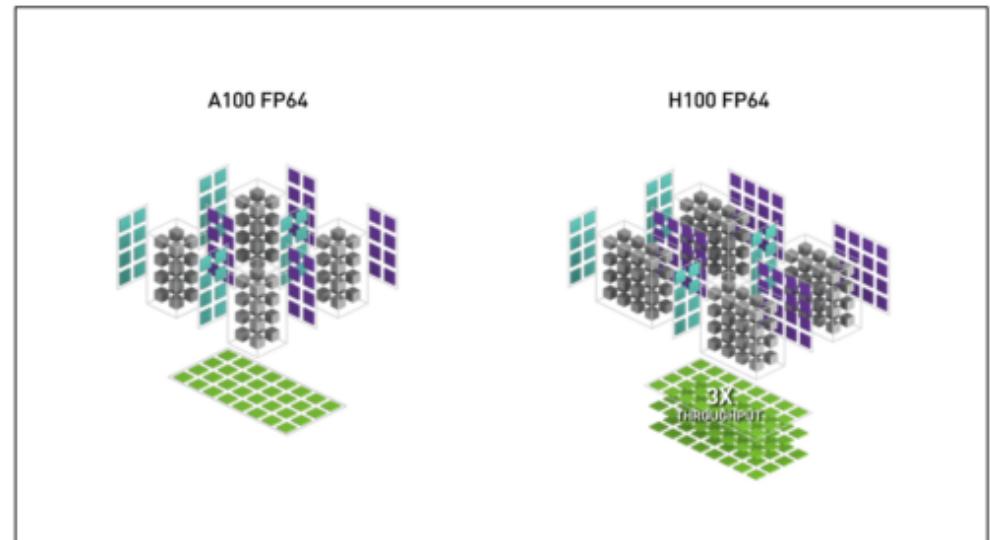
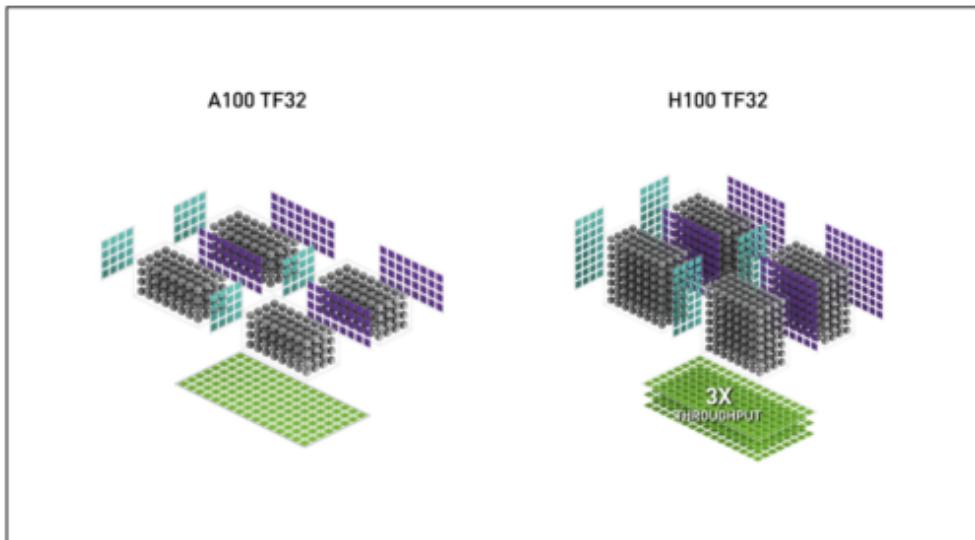


Recursos computacionales y de memoria en las cuatro últimas GPUs insignia

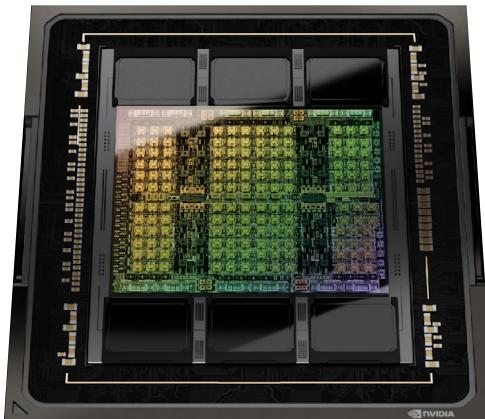
	Tesla V100 (Volta)	Titan RTX (Turing)	A100 (Ampere)	H100 * (Hopper)
GPU (chip)	GV100	TU102	GA100	GH100
fp32 cores	5120	4608	6912	16896
fp64 cores	2560	144	4096	8448
Frecuencia (base-boost)	1370-1455 MHz	1440-1770 MHz	1410 MHz	n/a
TFLOPS (fp16, fp32, fp64)	30, 15, 7.5	32.6, 16.3, 0.51	78, 19.5, 9.7	120, 60, 30
Interfaz de memoria	HBM2 4096 bits	GDDR6 384 bits	HBM2 5 stacks	HBM3 5 stacks
Ancho de banda de memoria	900 GB/s.	672 GB/s.	1555 GB/s.	3000 GB/s.
Memoria de video	16 ó 32 GB	24 GB	48 GB	80 GB
Memoria caché L2	6 MB	6 MB	40 MB	50 MB
Memoria compartida / multip.	Hasta 96 KB	Hasta 64 KB	Hasta 164 KB	Hasta 228 KB.

(*) Especificaciones preliminares para la H100 basadas en las expectativas actuales, pueden cambiar en el producto final.

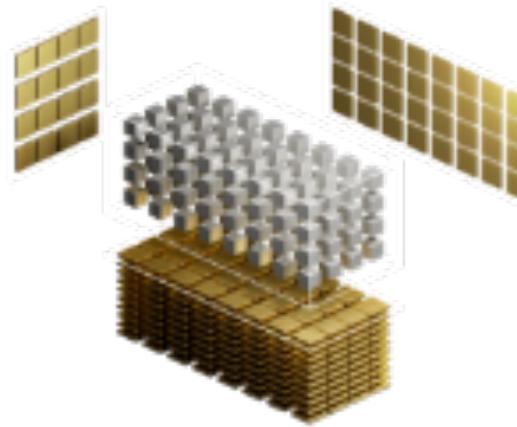
Computación matricial por hardware



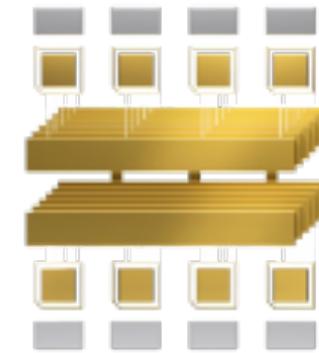
Hopper: Nuevo motor para infraestructura IA. Rendimiento, escalabilidad & seguridad



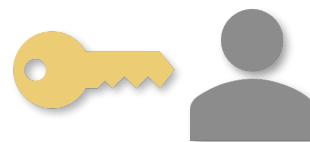
Fabricación a 4N por TSMC
80.000 millones de transistores



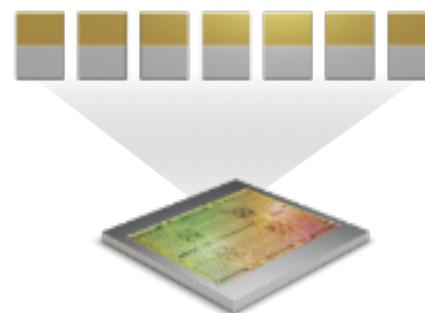
Motor para
transformers



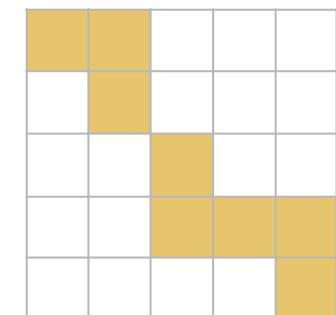
4^a generación NVLink



Computación
confidencial



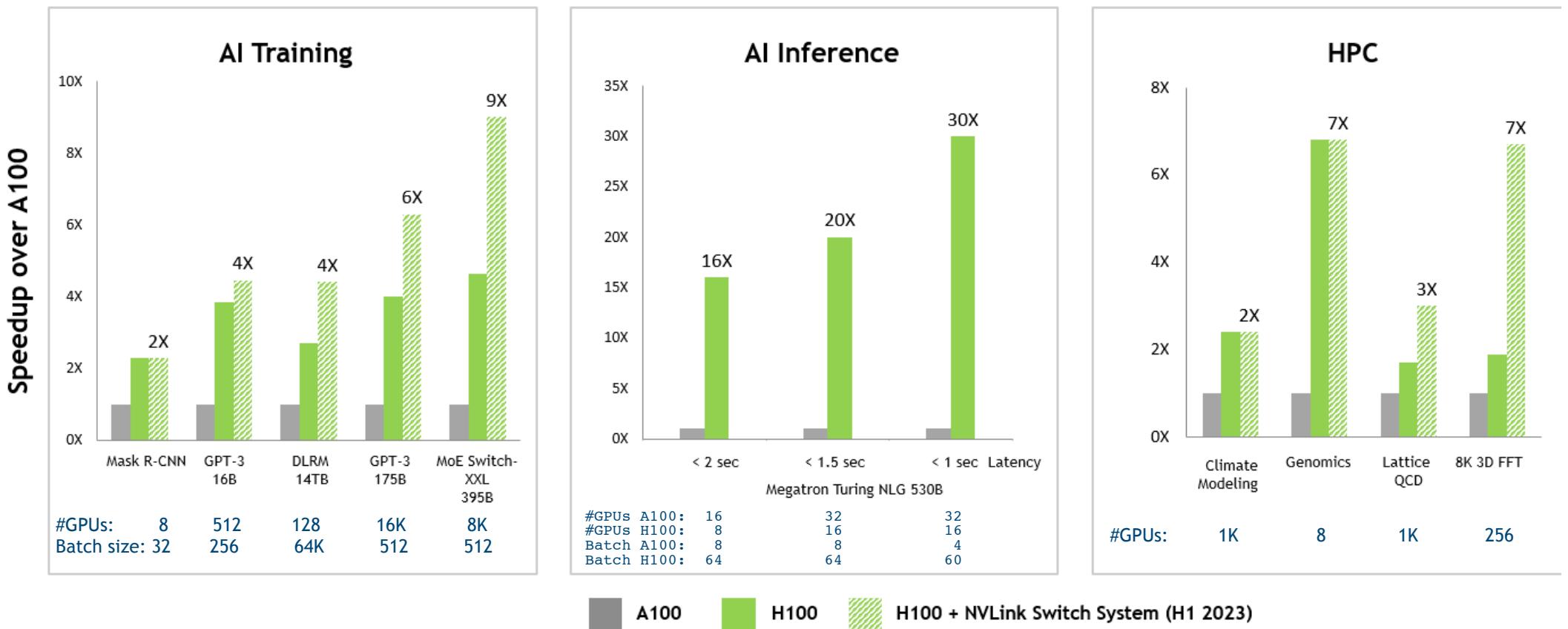
2^a generación MIG



Instrucciones DPX

Aceleración sustancial en todas las áreas

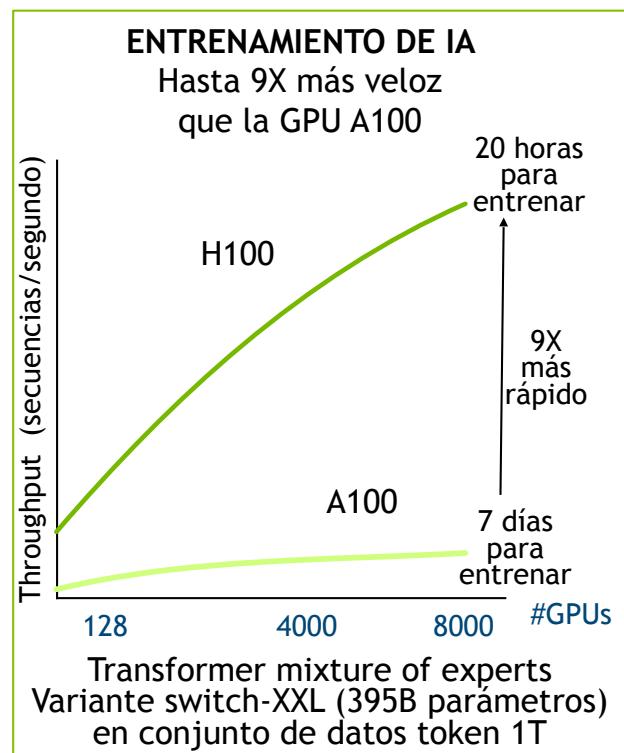
- Factores de mejora H100 vs. A100 sobre múltiples GPUs:



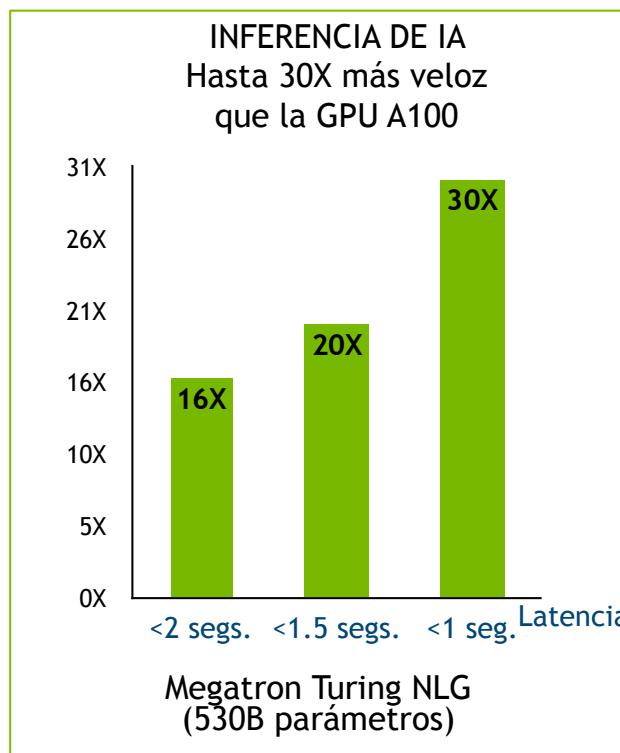
Proyecciones de rendimiento sujetas a cambio

H100 supone un salto en rendimiento de un orden de magnitud

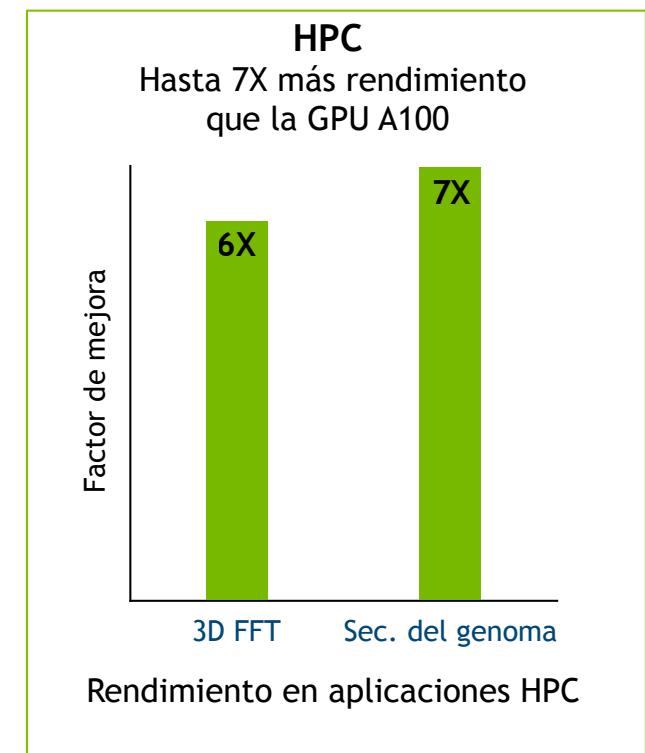
Rendimiento y escalabilidad para las aplicaciones actuales:



Proyecciones de rendimiento sujetas a cambio

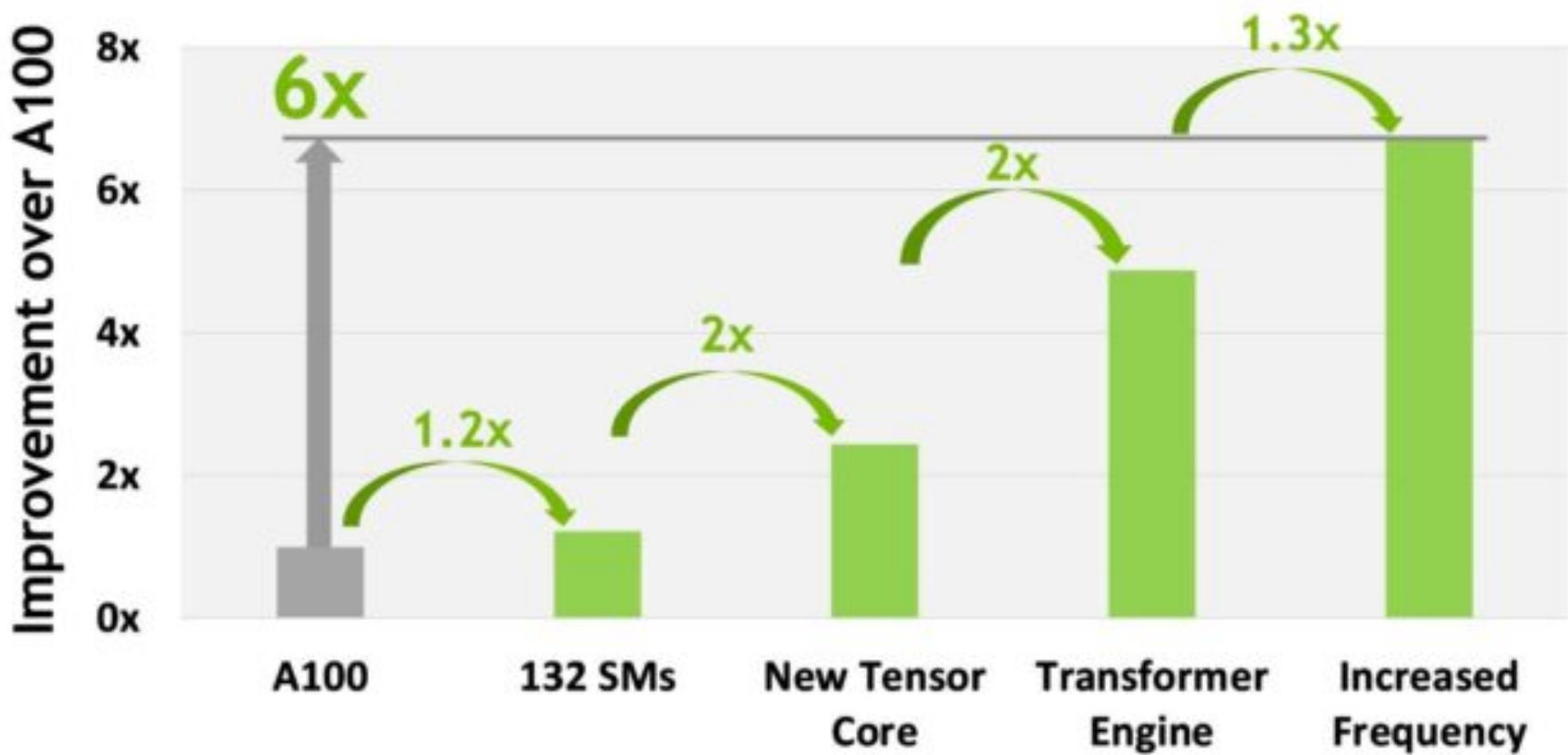


Longitud de la secuencia de entrada: 128
 Longitud de la secuencia de salida: 20
 Clúster A100: Red HDR Infiniband
 Clúster H100: Red NDR Infiniband para 16 H100
 16 A100 vs 8 H100 para 2 segs.
 32 A100 vs 16 H100 para 1 y 1.5 segs.

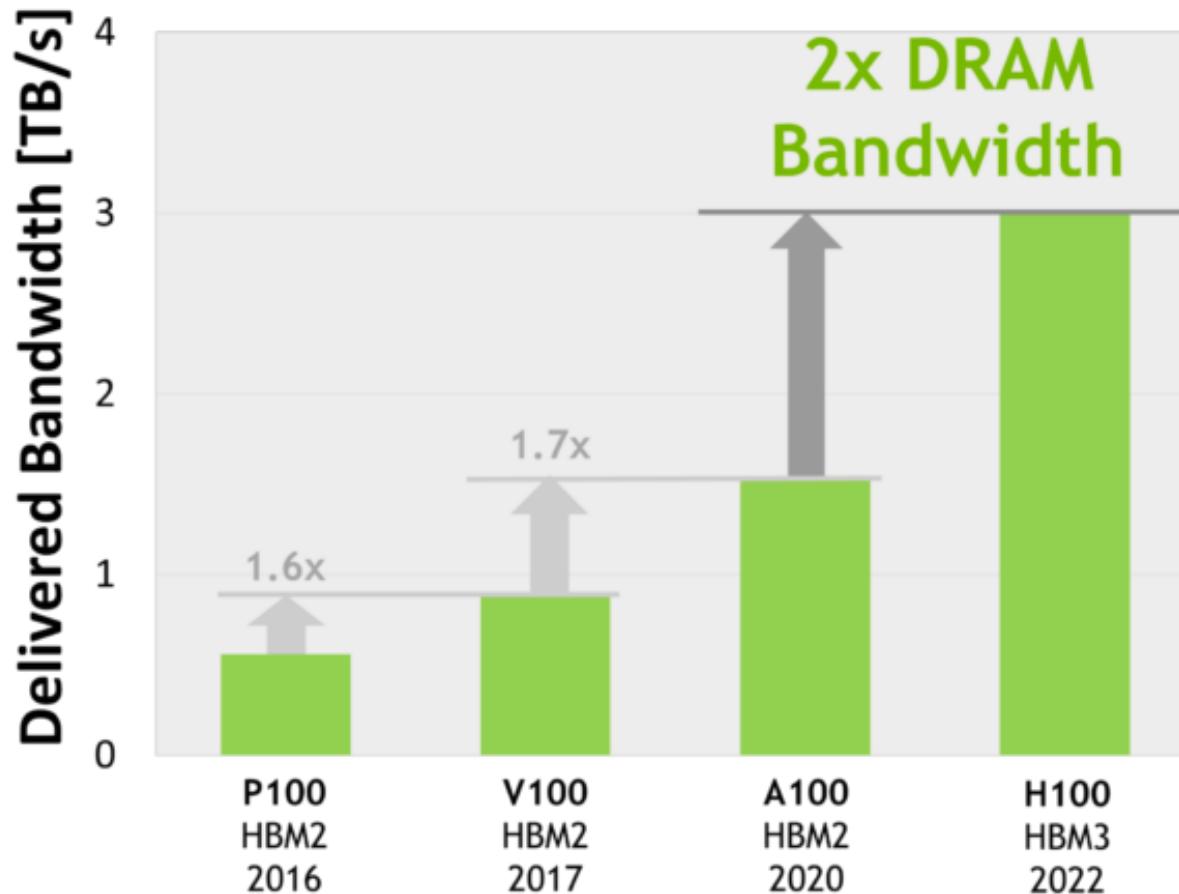


3D FFT ($4K^3$):
 - Clúster A100: Red Infiniband
 - Clúster H100: NVLink Switch System, Red Infiniband NDR
 Secuenciación del genoma (Smith-Waterman):
 - 1 A100
 - 1 H100

Desglose de los factores de mejora vs. A100



Mejoras en el ancho de banda de la memoria desde la adopción de HBM





II. 5. 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)	GM200 (Titan X) (Tesla M40)
Marco temporal	2006/07	2008/09	2010	2011	2012	2013	2013/14	2014	2014/15	2014/15	2016
CUDA Compute Capability	1.0	1.3	2.0	2.1	3.0	3.5	3.5	3.7	5.0	5.2	5.3
N (multiprocs.)	16	30	16	7	8	14	15	30	5	16	24
M (fp32 cores / multip.)	8	8	32	48	192	192	192	192	128	128	128
Número de fp32 cores	128	240	512	336	1536	2688	2880	5760	640	2048	3072

Las nuevas generaciones (2016-2022)

	Pascal			Volta	Turing	Ampere		Hopper	
Arquitectura	GP104 (GTX1080)	GP100 (Titan X) (Tesla P100)	GP102 (Tesla P40)	GV100 (Tesla V100)	TU102 (Titan RTX)	A100	GA100	H100	GH100
Marco temporal	2016	2017	2017	2018	2019	2020	2020	2022	2022
CUDA Compute Capability	6.0	6.0	6.1	7.0	7.5	8.0	8.x	9.0	9.x
N (multiprocs.)	40	56	60	80	72	108	128	114	132
M (fp32 cores / multip.)	64	64	64	64	64	64	64	128	128
Número de fp32 cores	2.560	3.584	3.840	5.120	4.608	6.912	8.192	14.592	16.896

Recursos computacionales y de memoria en las cuatro últimas GPUs insignia

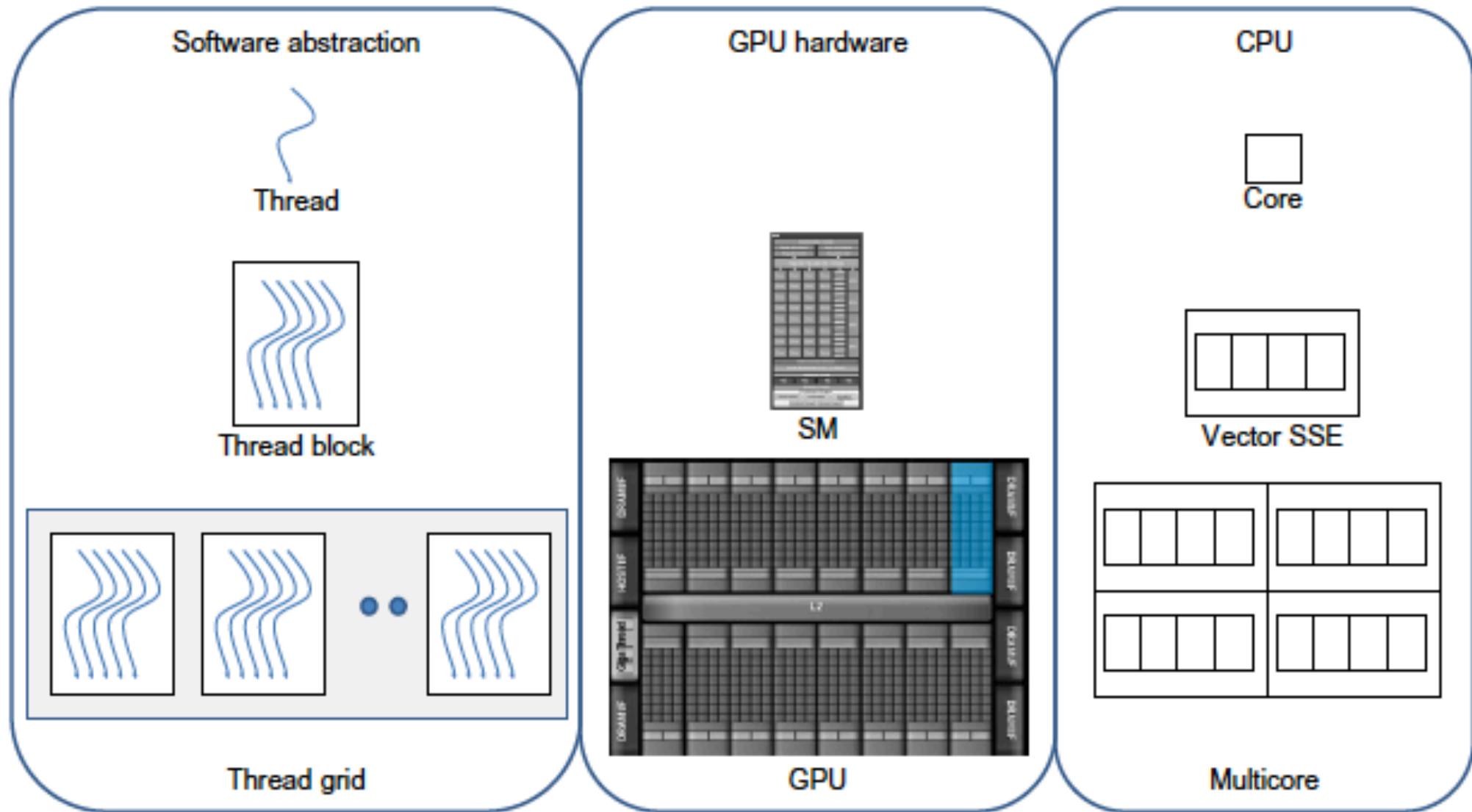
	Tesla V100 (Volta)	Titan RTX (Turing)	A100 (Ampere)	H100 * (Hopper)
GPU (chip)	GV100	TU102	GA100	GH100
fp32 cores	5120	4608	6912	16896
fp64 cores	2560	144	4096	8448
Frecuencia (base-boost)	1370-1455 MHz	1440-1770 MHz	1410 MHz	n/a
TFLOPS (fp16, fp32, fp64)	30, 15, 7.5	32.6, 16.3, 0.51	78, 19.5, 9.7	120, 60, 30
Interfaz de memoria	HBM2 4096 bits	GDDR6 384 bits	HBM2 5 stacks	HBM3 5 stacks
Ancho de banda de memoria	900 GB/s.	672 GB/s.	1555 GB/s.	3000 GB/s.
Memoria de video	16 ó 32 GB	24 GB	48 GB	80 GB
Memoria caché L2	6 MB	6 MB	40 MB	50 MB
Memoria compartida / multip.	Hasta 96 KB	Hasta 64 KB	Hasta 164 KB	Hasta 228 KB.

(*) Especificaciones preliminares para la H100 basadas en las expectativas actuales, pueden cambiar en el producto final.



III. Programación

Comparativa con la CPU: Dos formas de construir supercomputadores



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

POSIX-threads en CPU

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

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

CUDA en GPU, seguido del código host en CPU

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

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

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

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

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

El modelo de programación CUDA

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



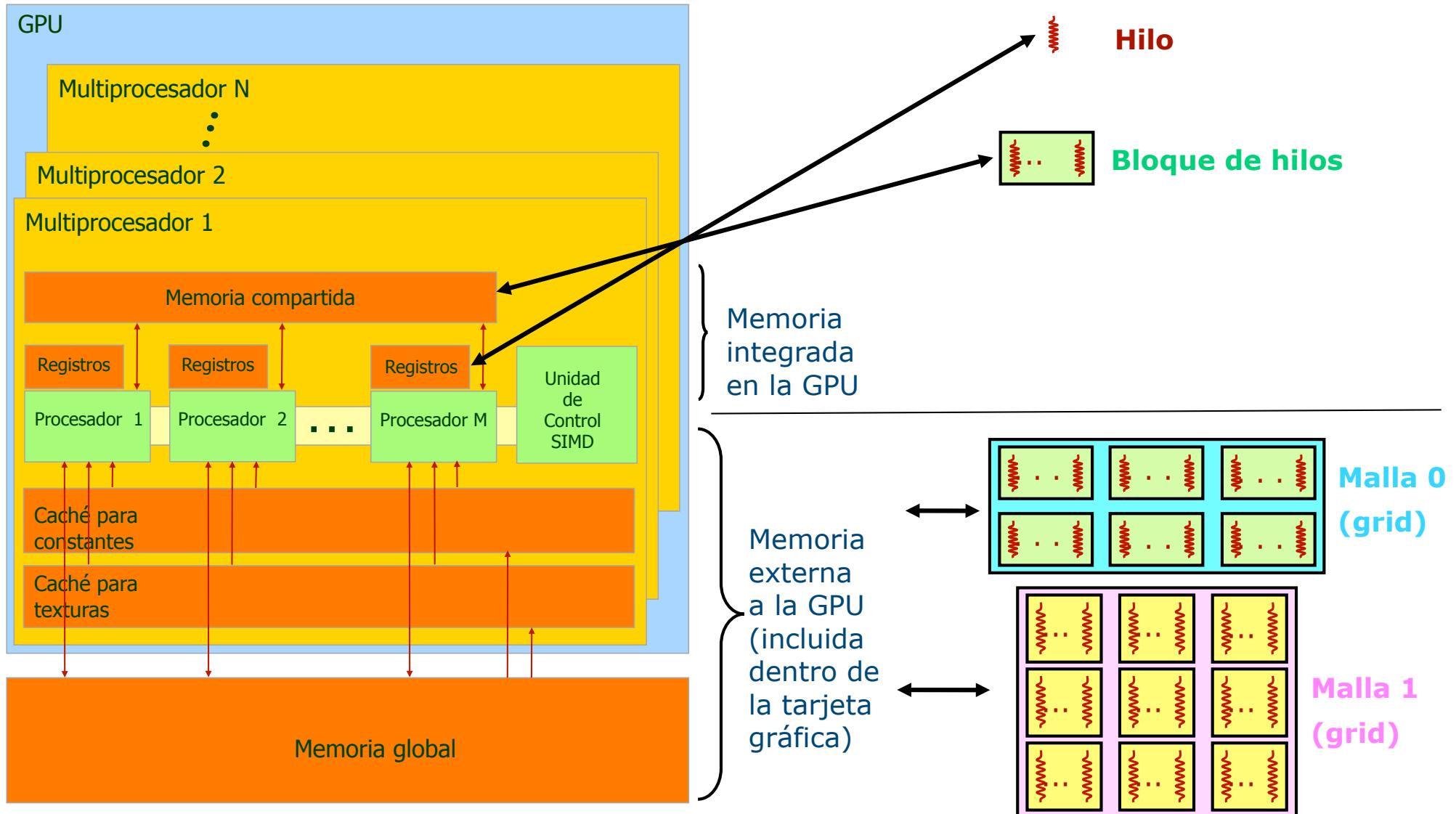
- Los hilos de CUDA son **extremadamente ligeros**.
 - Se crean en un tiempo muy efímero.
 - La commutación de contexto es inmediata.
- Objetivo del programador: Declarar miles de hilos, que la GPU necesita para lograr rendimiento y escalabilidad.

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	3.2*	3.5	3.7		
fp32 cores / Multip.	8	8	32	192	192	192	192	HW	Escala-bilidad
fp64 cores / Multip.	0	0	16	64	64	64	64		
Hilos / Warp	32	32	32	32	32	32	32	SW	Ritmo de salida de datos
Bloques / Multiprocesador	8	8	8	16	16	16	16		
Hilos / Bloque	512	512	1024	1024	1024	1024	1024	SW	Paralelis-mo
Hilos / Multiprocesador	768	1024	1536	2048	2048	2048	2048		
Regs. de 32 bits / Multip.	8K	16K	32K	64K	32K	64K	128K	HW	Conjunto de trabajo
Mem. compartida / Bloque	16K	16K	48K	48K	48 KB	48 KB	48 KB		
Mem. compartida / Multip.	16K	16K	48K	48 KB	48 KB	48 KB	112 KB		

(*) Establecida sólo para dispositivos Tegra y Jetson.

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

CCC	5.0	5.2	5.3*	6.0	6.1	6.2*	7.0, 7.2*	7.5	8.0	8.6
fp16 cores / Multip.	0	128	128	64	128	128	64	64	64	64 (x2 ops.)
fp32 cores / Multip.	128	128	128	64	128	128	64	64	64	64 (x2 ops.)
fp64 cores / Multip.	4	4	4	32	32	32	32	2	32	32
Tensor cores / Multip.	0	0	0	0	0	0	8	8	4	4
Bloques / Multiprocesador	32	32	32	32	32	32	32	16	32	16
Hilos / Bloque	1024	1024	1024	1024	1024	1024	1024	1024	1024	1024
Hilos / Multiprocesador	2048	2048	2048	2048	2048	2048	2048	1024	2048	1536
Regs. de 32 bits / Bloque	64K	64K	32K	64K	64K	32K	64K	64K	64K	64K
Regs. de 32 bits / Multip.	64K	64K	64K	64K						
Mem. compartida / Bloque	48 K	48K	48K	48K	48K	48K	48K, 96K	64K	163K	99K
Mem. compartida / Multip.	64K	96K	64K	64K	96K	64K	96K (de 128)	64K (de 96)	164K (de 192)	100K (de 128)

(*) Establecida sólo para dispositivos Tegra y Jetson.

Ejemplo: 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);
}
```

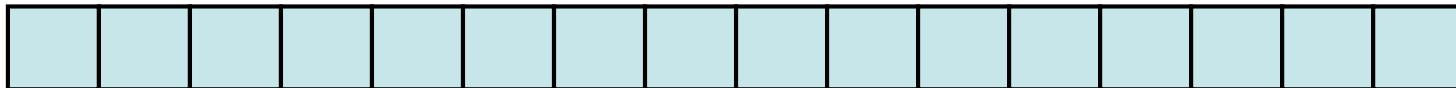
El kernel CUDA que se ejecuta en GPU,
seguido del código host para CPU.

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;
}

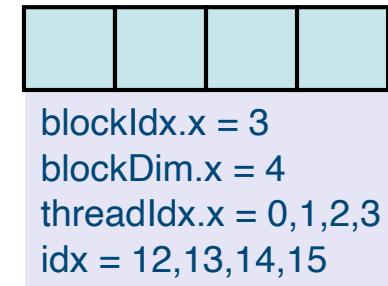
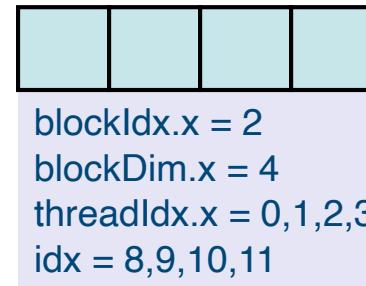
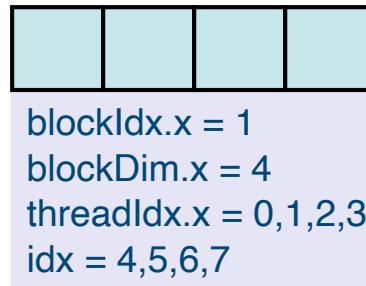
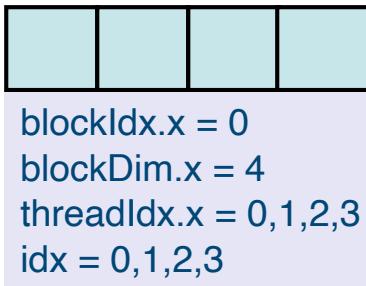
void main()
{
    .....
    dim3 dimBlock (blocksize);
    dim3 dimGrid (ceil(N/(float)blocksize));
    incremento_en_gpu<<<dimGrid, dimBlock>>>(a, b, N);
}
```

Acceso a los datos en paralelo



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: Parallelismo de grano fino en la GPU.

Extensiones
al lenguaje



`int idx = (blockIdx.x * blockDim.x) + threadIdx.x;`
Se mapeará del índice local `threadIdx.x` al índice global

Patrón de acceso común a todos los hilos

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

Código en CPU (host)

[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);
```



IV. Cinco kernels característicos

1. Operadores *streaming*

- Constituyen la expresión más simple de un bucle `forall`.
- La implementación CUDA aprovecha todo el paralelismo declarando tantos hilos como píxeles, para que cada uno se encargue de una sola iteración (paralelismo de grano fino).
- Ejemplo:

```
#define N 1920*1080

float r[N], g[N], b[N], luminancia[N];

for (i=0; i<N; i++)
    luminancia[i] = 255 * (0.299 * r[i] + 0.587 * g[i] + 0.114 * b[i]);
```

2. Operadores sobre vectores

- Otra expresión típica de un bucle `forall`.
 - Declararíamos un hilo CUDA por cada iteración del bucle para aprovechar al máximo paralelismo y escalabilidad.
- Ejemplo:

```
#define N (1 << 30)

float a[N], b[N], c[N];

for (i=0; i<N; i++)
    c[i] = a[i] + b[i];
```

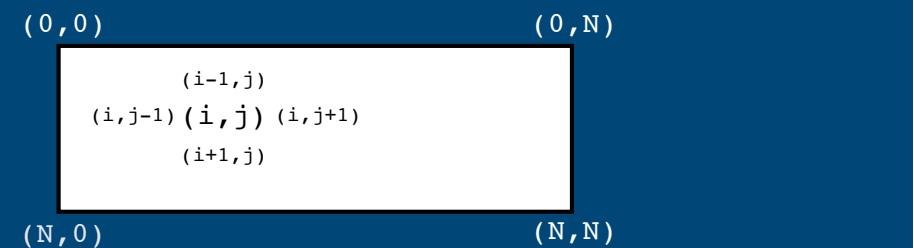
3. Operadores patrón (*stencil operators*)

```

int i, j, iter, N, Niters;
float in[N][N], out[N][N];

for (iter=0; iter<Niters; iter++) {
    for (i=1; i<N-1; i++)
        for (j=1; j<N-1; j++)
            out[i][j] = 0.2 * (in[i][j] + in[i-1][j] + in[i+1][j] + in[i][j-1] + in[i][j+1]);
    for (i=1; i<N-1; i++)
        for (j=1; j<N-1; j++)
            in[i][j] = out[i][j];
}

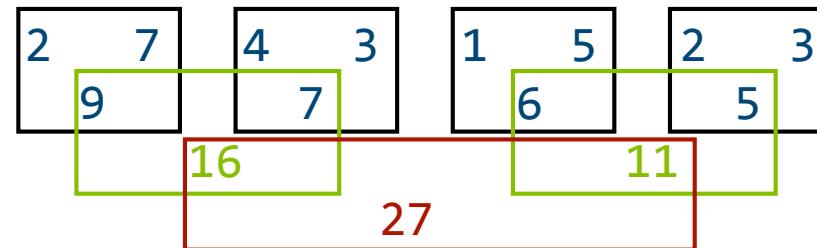
```



- Las iteraciones del bucle externo deben serializarse debido a la presencia de dependencias (*carried-loop dependencies*), pero podemos explotar todo el paralelismo para cada partícula (los bucles 2, 3, 4 y 5 son todos `forall`).
- La carga computacional depende de `Niters`, pero el paralelismo depende del tamaño de la matriz 2D (N^2).
- Todos los hilos deben sincronizarse entre asignaciones.

4. Operadores de reducción

```
float sum, x[N];  
sum = 0;  
for (i=0; i<N; i++)  
    sum += x[i];
```

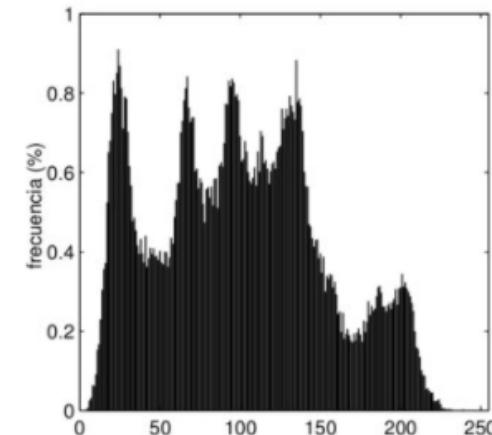


- El código tiene dependencias entre las iteraciones, pero el paralelismo puede explotar la asociatividad del operador para desplegar paralelismo en forma de árbol binario, resultando en $\log(N)$ pasos que van reduciendo el grado de paralelismo a la mitad hasta concluir con un solo hilo.
- El reto está en utilizar el patrón de acceso a memoria que explota mejor la jerarquía de memoria de la GPU.

5. Histogramas

```
int histo[Nbins], image[N][N];
sum = 0;

for (i=0; i<Nbins; i++)
    histo[Nbins] = 0;
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        histo[image[i][j]]++;
```



- El primer bucle `forall` apenas tiene carga computacional.
- Los dos bucles siguientes tienen dependencias, pero los valores de `image[][]` se pueden leer en paralelo si los asignamos a hilos CUDA. A partir de ahí, CUDA proporciona operaciones atómicas (`atomicInc(histo[image[i][j]])`) para serializar accesos concurrentes al vector `histo[]`, y que usaremos aquí para prevenir condiciones de carrera.

Análisis final

Code feature	Streaming	Array	Stencil	Reduction	
	Operator	Constructor	Operator	Operator	Histogram
Input size	$3*N$	$2*N$	N^2	N	N^2
Output size	N	N	N^2	1	No. bins
Computational weight	Low	Low	Heavy	Low	High
Computat. complexity	$O(N)$	$O(N)$	$O(it*N^2)$	$O(N)$	$O(N^2)$
Memory usage	Average	Average	Intensive	Low	High
Data reuse	None	None	High	Low	Low
Op. intensity (FLOP/byte)	0.375	0.083	0.625	0.25	0.25
Shared memory candidates	No	No	in []	sum	histo []
Estimated GPU speed-up	High	High	Fair	Low	Low

- El operador *streaming* es el que más acelera en GPU.
- El operador patrón es el que mejor aprovecha la memoria compartida.
- El operador de reducción depende más del programador.
- El histograma supone el mayor reto para el programador.