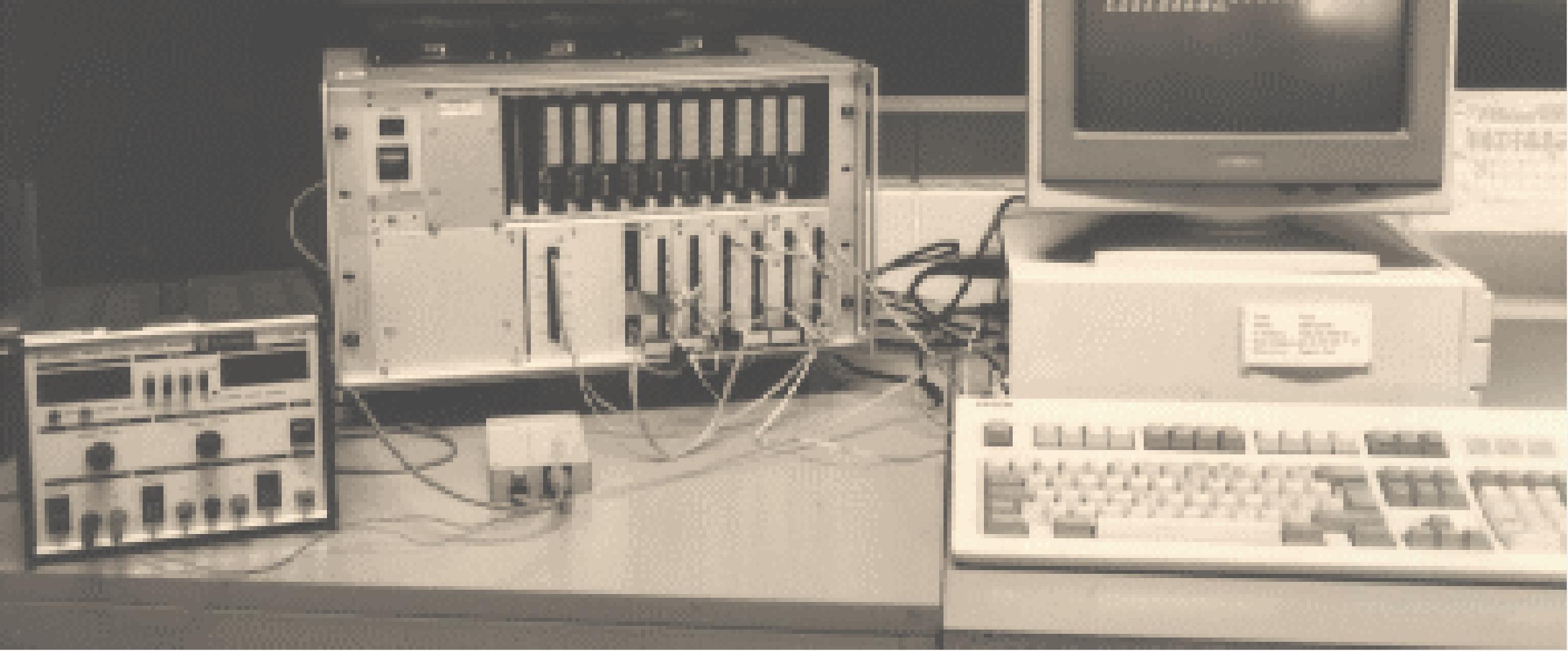


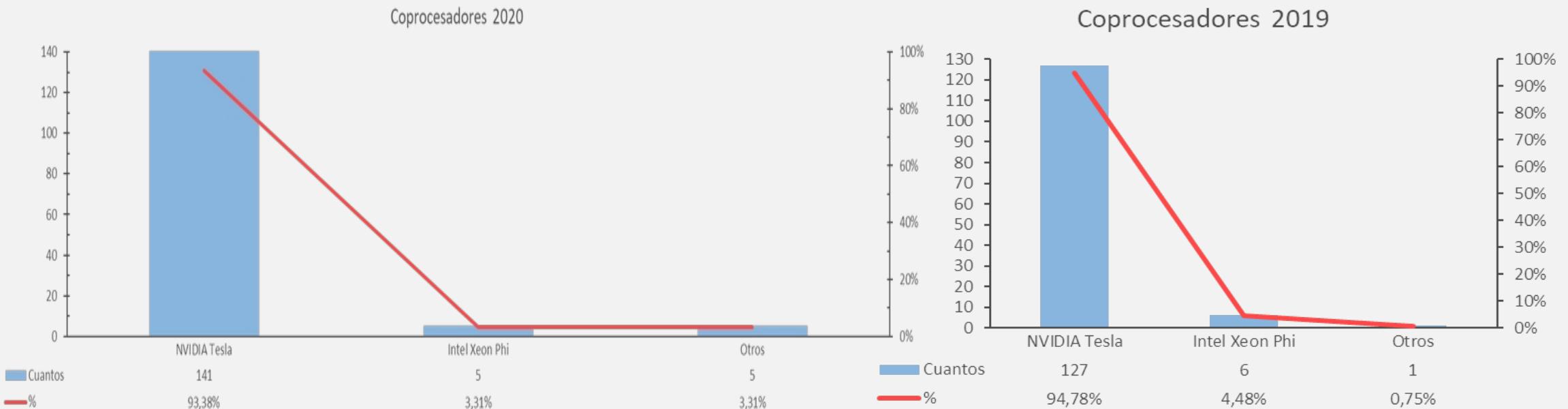
GENERAL-PURPOSE COMPUTING ON GRAPHICS PROCESSING UNITS (GPGPU)



PRESENTACIÓN

Motivación

- Mantener la ley de *Moore* ha implicado el uso de coprocesadores.
- Las siguientes gráficas, del www.top500.org, lo dicen todo.



- No hay indicios de cambio a corto/medio plazo. NVIDIA domina ¿Intel Xe (eXascale for Everyone, 2020) / AMD RDNA 2 (Radeon DNA 2, 2020)?
- La Era de la Computación Elástica (*the Age of Elastic Computing*): adaptación automática de los recursos de la computación a los cambios en la carga de trabajo.

PRESENTACIÓN

Contextualización

- Una introducción
- Memoria Compartida. SIMD – SIMT – Paralelismo de grano fino.
¿Se recuerda la fase 1 de la metodología de diseño?

No son autónomos, necesitan un sistema anfitrión. *PCI Express*, *NVLink*, etc.

- Cuando el sistema anfitrión y el coprocesador se usan conjuntamente para resolver un problema → Sistema Heterogéneo (o híbrido según la fuente y/o la tendencia imperante).
- Tiene su propia memoria, compleja. El espacio de direcciones físico de memoria no es único (no compartido físicamente con el anfitrión).
- El software actual proporciona un espacio de dirección virtual unificado para acceder a toda la memoria.

PRESENTACIÓN

Análisis del Rendimiento

- El direccionamiento virtual transparente (en CUDA se llama *Unified Memory*) es una característica del modelo de programación: las transferencias de información existen.
- Hay que modelar comunicaciones, sincronización y cálculo.
- Encontrar el referente secuencial puede ser complejo.
- Importante equilibrar la carga en función de la potencia de cada subsistema.
- En resumen:
 - Cada subsistema tiene su constante de cálculo. No se puede simplificar.
 - Cada subsistema tiene su tamaño de problema. Se puede expresar en función del tamaño general.
 - Hay comunicaciones, síncronas o asíncronas, con constantes distintas según sea entre el sistema anfitrión o entre los coprocesadores. Como mínimo considerar las comunicaciones anfitrión - coprocesadores.

NVIDIA: A BUSINESS OVERVIEW

- 1993 Jen-Hsun Huang, Chris Malachowsky, y Curtis Priem fundan la compañía NV (*Next Version*). NVIDIA vino después, de la palabra latina *invidia* (de *inviduos*, hostil, envidia), “*la que más les gusto conteniendo NV*”.
- 1995 NV1 primer producto de la compañía.
- 1998 La Fabless Semiconductor Association la elige, por segundo año consecutivo, compañía más respetada de la industria de semiconductores.
- 1999 Oferta pública a \$12 acción (4 *splits* “1000 acciones de 1999= 12000 en 2020”; \$550 acción hoy).
- 2001 La 1^a compañía en facturar más rápidamente \$1000 millones de ingresos. Entra en el S&P 500.
- 2002 Fortune la reconoce como la compañía de mayor crecimiento de EE.UU.
- 2003 La Stanford Business School Alumni Association la nombra compañía emprendedora del año.
- 2006 Más de 500 millones de procesadores vendidos.
- 2007 Forbes la nombra compañía del año. Gana un Emmy (2010, 2011, 2012 presente en los Oscar).
- 2012 Newsweek la designa la sexta compañía más ecológica de América.
- 2013 Compra The Portland Group (antes/después compra más empresas, esta tiene relevancia para PCP).
- 2019 Toyota, Volvo, Audi, Xbox, PlayStation 3, NASA, ORNL-Titán, etc. usan tecnologías NVIDIA.
- 2020 Adquiere MELLANOX y ARM (setiembre, \$40.000 millones).

NVIDIA: A COMPUTATIONAL OVERVIEW

- 1999 NVIDIA “inventa” la GPU (*Graphics Processing Unit*). Presenta las familias **Quadro** y **GeForce** (el nombre GeForce fue por concurso). GeForce 256 la *primera* GPU.
- 2006 Presentación de CUDA (*Compute Unified Device Architecture*).
- 2007 Lanzamiento de la arquitectura **Tesla** (la primera realmente computacional).
- 2008 Lanzamiento del procesador móvil Tegra (consumo 30 veces menor).
- 2009 Lanzamiento de la arquitectura **Fermi**.
- 2012 Lanzamiento de la arquitectura **Kepler**, primera GPU virtualizada y tabletas/Smartphones con Tegra 3.
- 2014 Lanzamiento de la arquitectura **Maxwell**, del procesador Tegra K1 y del sistema Jetson TK1.
- 2015 Lanzamiento del procesador Tegra X1 y del sistema Jetson TX1.
- 2016 Lanzamiento de la arquitectura **Pascal** y del superordenador DGX-1 enfocado a aplicaciones de IA.
- 2017 Lanzamiento de la arquitectura **Volta** y del sistema Jetson TX2 (IA gana protagonismo).
- 2018 Lanzamiento **Turing** (Volta con *Ray-Tracing Cores*), del DGX-2 y del Jetson AGX Xavier. Elimina el apodo Tesla (Tesla T4 fue la última con el apodo) para evitar confusiones con Tesla Automotive.
- 2019 Lanzamiento de los sistemas Jetson Xavier NX y Nano (para IoT).
- 2020 Lanzamiento de la arquitectura **Ampere** (NVIDIA A100 SXM 4 o A100 PCIe, segundo semestre 2020).

ARQUITECTURA

- Cada generación (arquitectura) presenta **características** diferenciadoras y un amplio sustrato común. Cada arquitectura presenta **capacidades** concretas con un núcleo común muy fuerte.
- Hay varias familias (GeForce, Quadro, Tegra, etc.). Las diferencias entre familias de la misma arquitectura son de potencia, memoria, consumo, etc., pero no de características/capacidades. P. ej. NVIDIA A100 PCIe 11000€, NVIDIA V100 5000€, Jetson AGX Xavier 700€, Jetson Nano 100€ (V100, Xavier y Nano son todos arquitectura Volta).

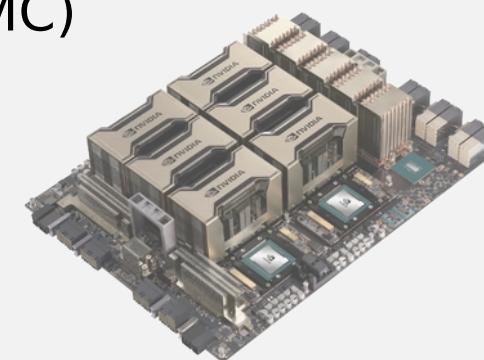
Ecosistema 2020 con GA100 (arquitectura Ampere con litografía de 7 nm fabricada por TSMC)



DGX



A100 SXM4



HGX



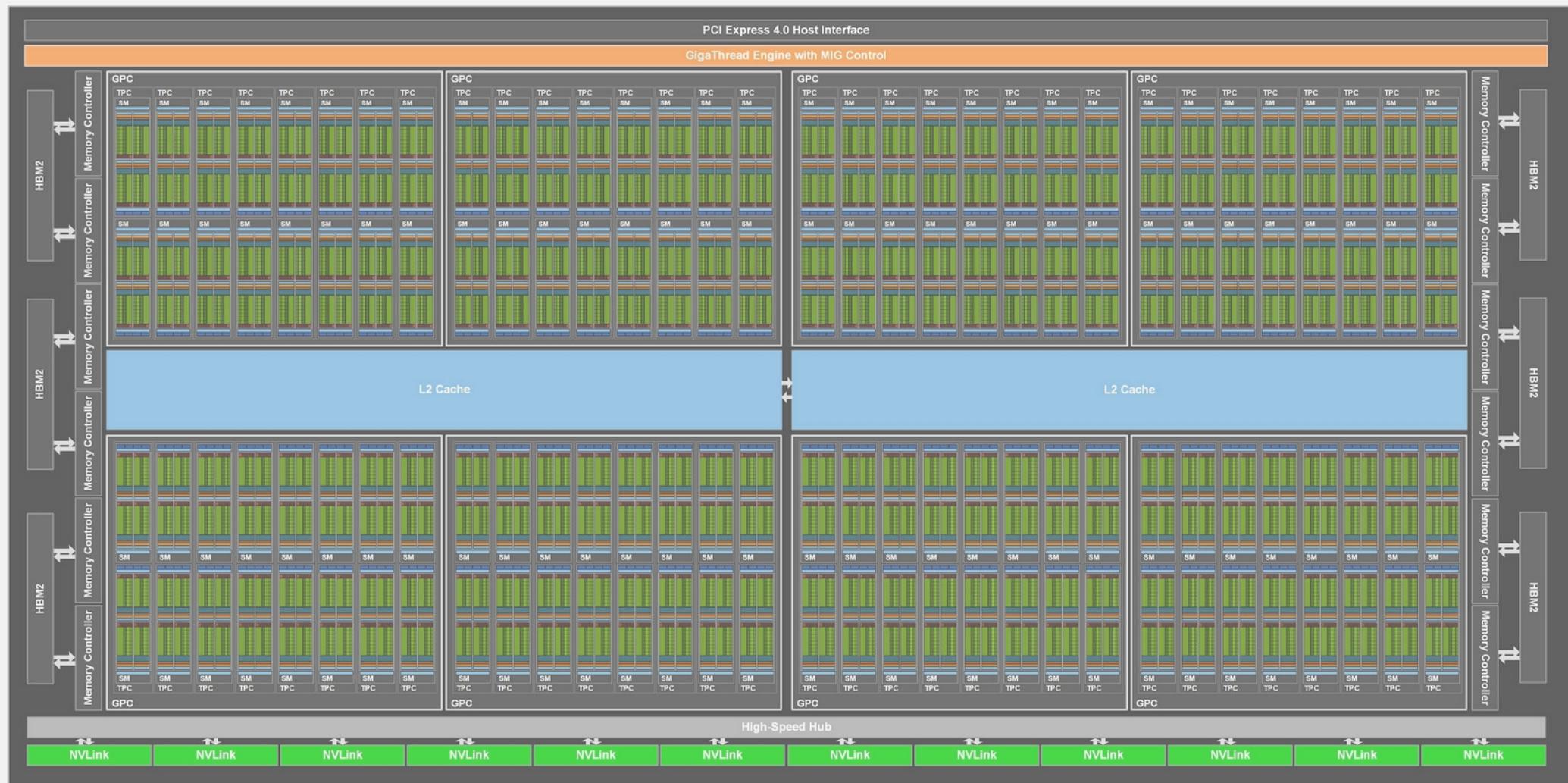
A100 PCIe

ARQUITECTURA GA100: NOVEDADES

- *Multi-Instance GPU (MIG)*. Permite virtualizar y particionar la GPU. Se puede compartir la GPU usando cada usuario una fracción estanca de ella o usar simultáneamente varias GPUs (muy útil para proveer servicios en la nube, QoS, etc. *Elastic Computing*). CUDA 11 lo soporta.
- La cache L1 combina en un solo bloque de memoria las funcionalidades de *data cache* y *shared memory*, mejorando el rendimiento. L1 de 192 KB por SM (se verá muy pronto qué es un *SM*).
- 40 GB de HBM2 DRAM de memoria principal (*global/device memory*) y 40 MB de caché L2.
- Nueva instrucción de copia asíncrona. Carga directamente los datos de la memoria global a la memoria compartida de los SM. Permite copiar a la vez que el SM está haciendo otro tipo de cálculos.
- Incluye *cores FP32* y *INT32* separados (como las anteriores). FP32 y INT32 pueden operar simultáneamente. Útil para aplicaciones con *inner loops* con operaciones de cálculo de direcciones (INT32) y de cálculos (FP32): cada *pipelined loop* direcciona y carga nuevos datos mientras procesa los actuales.
- Soporte optimizado para la nueva característica *Task Graph Acceleration* de CUDA 11.

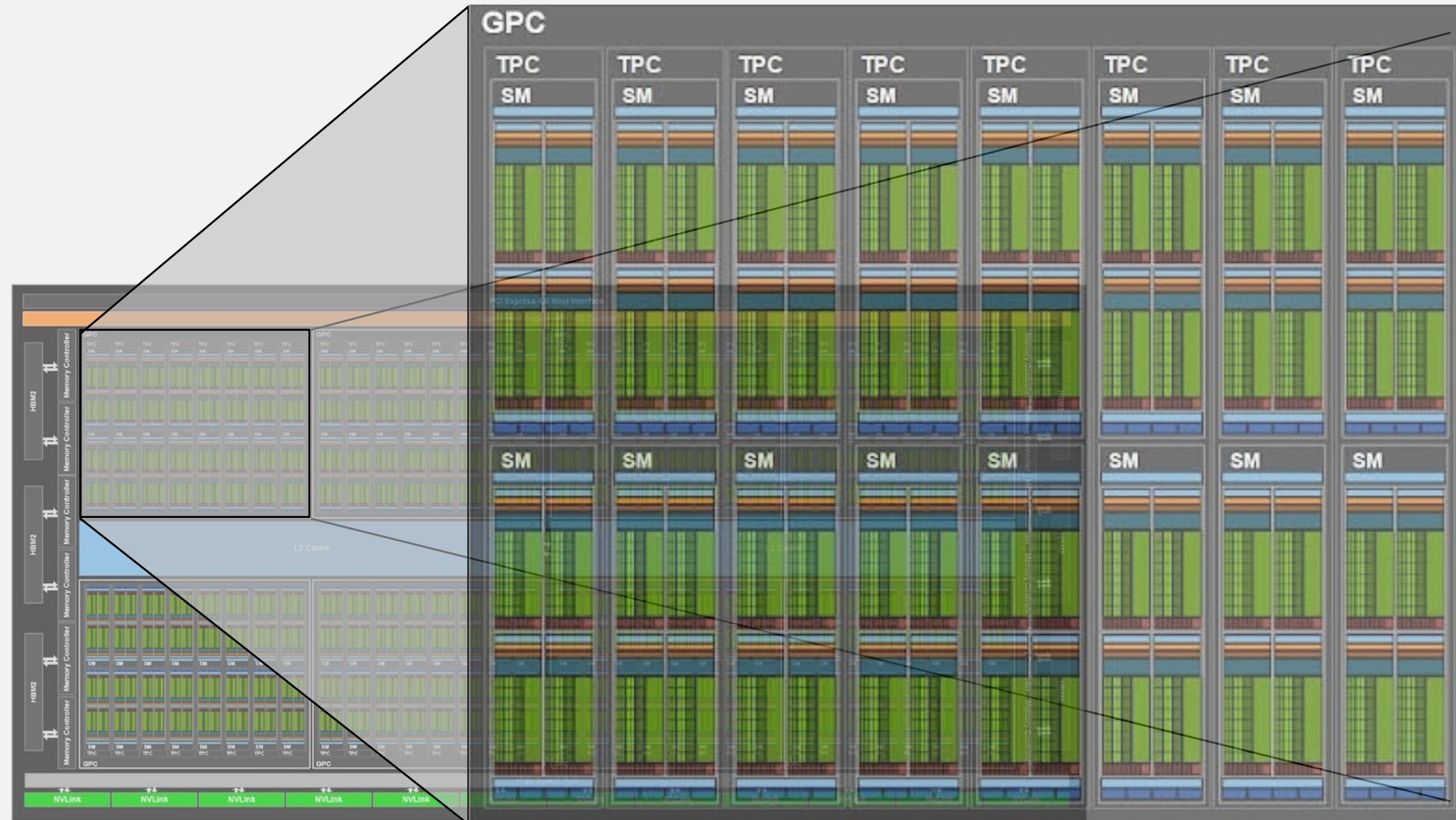
ARQUITECTURA GA100

Similar a GV100 (Volta) y GP100 (Pascal)



ARQUITECTURA GA100

Similar a GV100 (Volta) y GP100 (Pascal)



ARQUITECTURA GA100

Full board

- 8 GPCs (*Graphics Processing Cluster*)
- Cada GPC tiene 8 TPCs (*Texture Processing Cluster*)
- Cada TPC tiene 2 SMs (*Streaming Multiprocessors*)
- Cada SM
 - 64 INT CUDA Cores
 - 64 FP32 CUDA Cores y 32 FP64 CUDA Cores
 - 4 Third-generation Tensor Cores

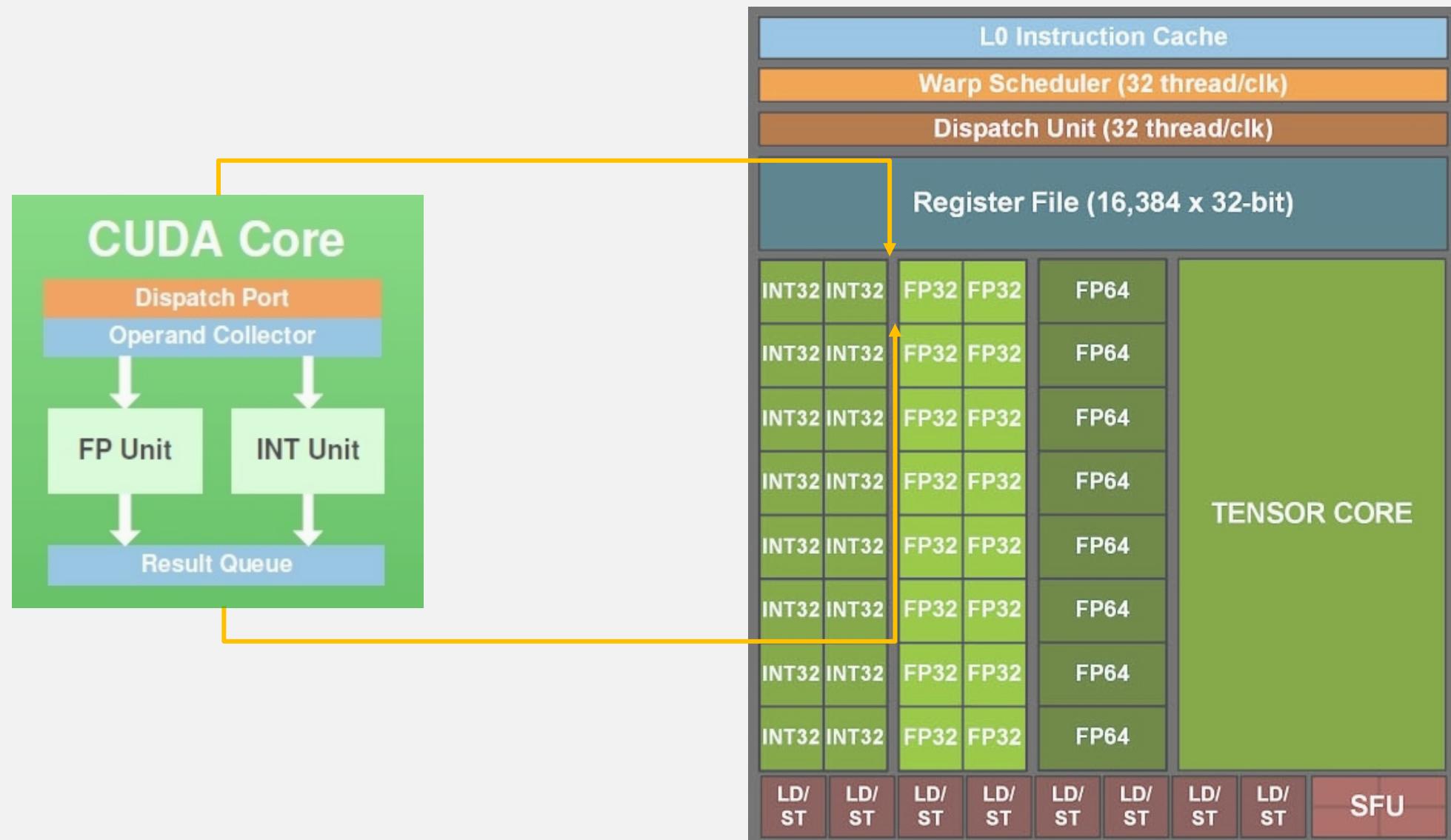
En resumen, una *full board* tiene

- 128 SMs
- 8192 INT, 8192 FP32 y 4096 FP64 CUDA Cores
- 512 Third-generation Tensor Cores
- 6 HBM2 stacks, 12 512-bit Memory Controllers

No todas las tarjetas tienen todos los elementos ni en la misma cantidad.



ARQUITECTURA GA100: CUDA CORE (CC)

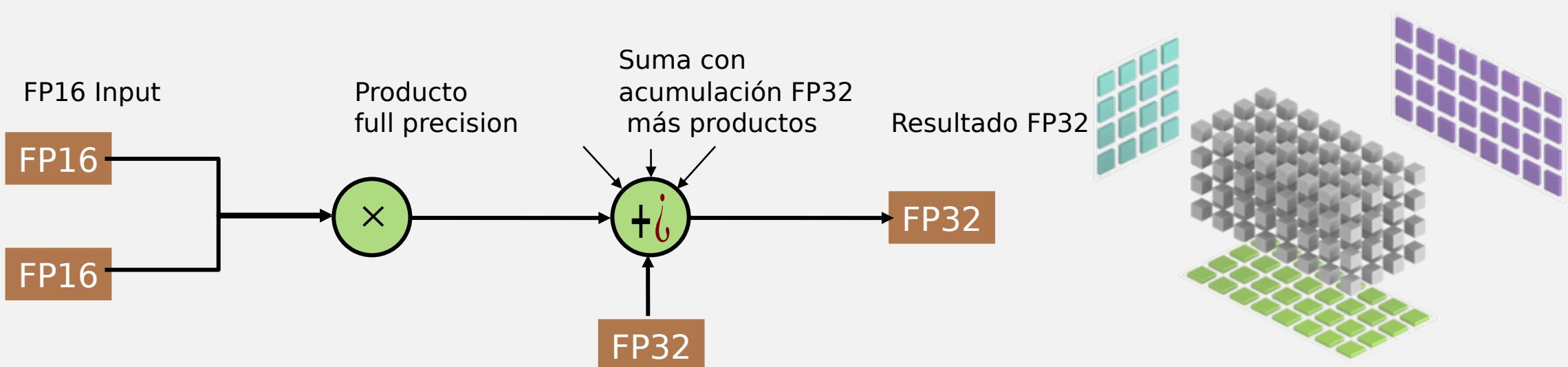


ARQUITECTURA GA100: THIRD-GENERATION TENSOR CORE (TTC)

- Un TTC multiplica y acumula matrices (MMA) en un solo ciclo de reloj, muy útil en DL (*Deep Learning*).
- Algunos datos y nuevas características:
 - Cada TTC puede realizar 256 FMAs FP16 (*Fused Multiply-Add operations*) más acumulación FP32 por ciclo.
 - Esto permite calcular $D = AB + C$ de dimensiones $m=k=8$ y $n=4$ en un ciclo de reloj.
 - Cada SM tiene 4 TTC. Un SM puede hacer 1024 FMAs por ciclo (o 2048 FP16). Como una *full board* tiene 128 SM podrá computar con sus TTCS 528.288 FMAs por ciclo.
 - Una A100 (no es full board) obtiene 312 TFLOP con TTC FP16 (78 con CC FP16), 156 con TTC FP32 (19.5 con CC FP32) y 19.5 con TTC FP64 (9.7 con CC FP34).
 - También puede usar otros tipos (BF16, TF32, FP64, INT8, INT4, etc.) operando a la misma u otra velocidad. Por ejemplo TTC INT8 llegan a los 624 TOP, 1248 TOP con INT4 o 4992 TOP en binario.
 - *Sparcity*. La nueva instrucción *Sparse MMA* permite explotar la dispersión en las estructuras de las redes neuronales en DL, obviando (no operando) con aquellos valores que son *nulos*.

ARQUITECTURA GA100: THIRD-GENERATION TENSOR CORE (TTC)

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



ARQUITECTURA: RAY-TRACING CORE (RTC)

- Para el trazado de rayos de luz en tiempo real en gráficos. Se basa en los objetos que forman parte del gráfico (de la escena).

RTC es la unidad encargada de recorrer el árbol BVH (*Bounding Volume Hierarchy*) de manera continuada. Recorre todos los posibles caminos para dar respuesta a cada uno de ellos. Todo en tiempo real.

Actualmente solo en la serie GeForce.

GeForce no tiene FP64 (hasta ahora).

Por ahora no hay GeForce GA100.

Demo: https://www.youtube.com/watch?v=NgcYLIvlp_k



ARQUITECTURA: COMÚN A TODAS LAS ACTUALES

- Las instrucciones son ejecutadas en orden (no saltos predictivos, no ejecución especulativa), por ahora.
- Los hilos se agrupan en bloques (*thread block*) de topología 1D, 2D ó 3D.
- Los hilos de un bloque se ejecutan concurrentemente en un SM.
- Múltiples bloques pueden ejecutarse concurrentemente en un SM.
- Cuando un bloque termina se lanza la ejecución de nuevos bloques.
- Cuando a un SM llega un bloque para ejecutar lo divide en **wraps** que son planificados (*warp scheduler*) para su ejecución.
- Desde VOLTA cada hilo tiene su propio contador de programa, permitiendo una planificación (*scheduling*) independiente de los hilos. Igual concurrencia entre los hilos, independiente del warp.

COMPUTE UNIFIED DEVICE ARCHITECTURE



Basado en <https://docs.nvidia.com/cuda>

CUDA PRESENTACIÓN

- En el Q4 de 2006 NVIDIA presenta CUDA (*Compute Unified Device Architecture*).
- La GPGPU (*General-Purpose Computing on Graphics Processing Units*) introducida con CUDA busca la *escalabilidad automática*: el software debe escalar transparentemente al aumentar el número de *cores*.
- La estrategia:
 - Dividir el problema en sub-problemas “grandes” que pueden ejecutarse en paralelo de forma independiente (**bloques de hilos**).
 - Dividir cada sub-problema en piezas más finas que se resuelven concurrente y cooperativamente (**hilos**).
 - Cada bloque de hilos puede ejecutarse en cualquiera de los procesadores disponibles, en cualquier orden, simultánea o secuencialmente.
- La *Compute Capability* de una GPU viene definida por su revisión *major.minor*.
 - Igual *major* misma arquitectura. *minor* denotan mejoras incrementales sobre el núcleo de la arquitectura. Los *major* son: ~~1 Tesla~~, ~~2 Fermi~~, 3 Kepler, 5 Maxwell, 6 Pascal, 7 Volta y 8 Ampere.
 - Con el *major.minor* se sabe qué “cosas” CUDA soporta/puede hacer la GPU.

CUDA PRESENTACIÓN

CUDA Occupancy Calculator - Nvidia

(<https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html>)

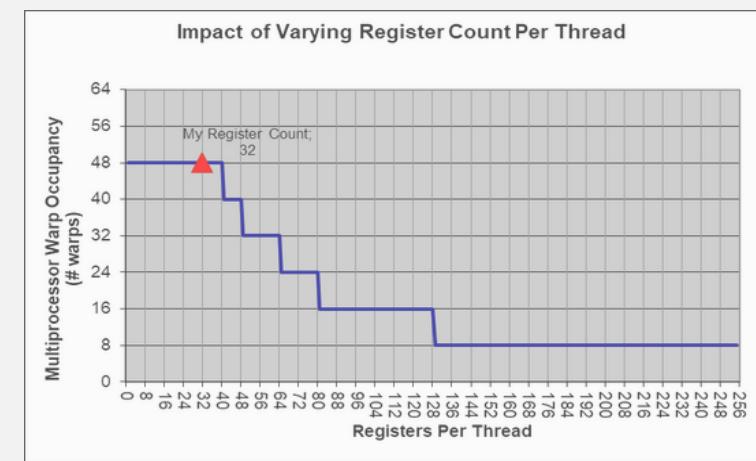
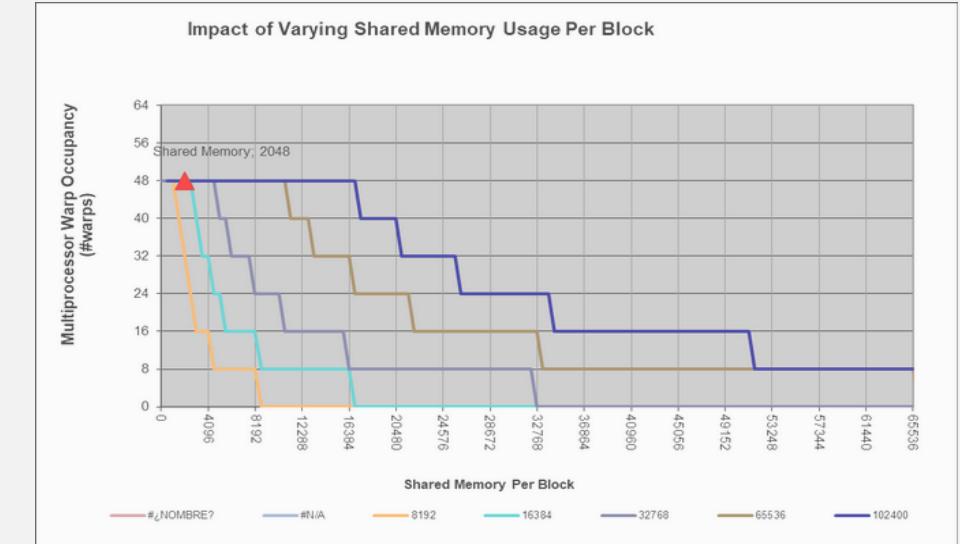
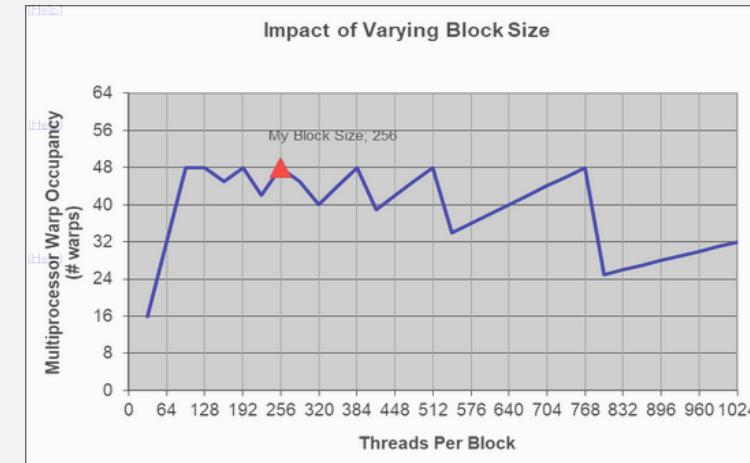
Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click): 8,6
 1.b) Select Shared Memory Size Config (bytes): 65536
 1.c) Select CUDA version: 11,1

2.) Enter your resource usage:
 Threads Per Block: 256
 Registers Per Thread: 32
 User Shared Memory Per Block (bytes): 2048
 (Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:
 Active Threads per Multiprocessor: 1536
 Active Warps per Multiprocessor: 48
 Active Thread Blocks per Multiprocessor: 6
 Occupancy of each Multiprocessor: 100%
 Physical Limits for GPU Compute Capability: 8,6
 Threads per Warp: 32
 Max Warps per Multiprocessor: 48
 Max Thread Blocks per Multiprocessor: 16
 Max Threads per Multiprocessor: 1536
 Maximum Thread Block Size: 1024
 Registers per Multiprocessor: 65536
 Max Registers per Thread Block: 65536
 Max Registers per Thread: 255
 Shared Memory per Multiprocessor (bytes): 65536
 Max Shared Memory per Block: 65536
 Register allocation unit size: 256
 Register allocation granularity: warp
 Shared Memory allocation unit size: 128
 Warp allocation granularity: 4
 Shared Memory Per Block (bytes) (CUDA runtime use): 1024
 Allocated Resources = Allocatable
 Per Block Limit Per SM Blocks Per SM
 Warps (Threads Per Block / Threads Per Warp): 8 48 8
 Registers (Warp limit per SM due to per-warp reg count): 8 64 8
 Shared Memory (Bytes): 2048 65536 32
 Note: SM is an abbreviation for (Streaming) Multiprocessor
 Maximum Thread Blocks Per Multiprocessor: Blocks/SM * Warps/Block = Warps/SM
 Limited by Max Warps or Max Blocks per Multiprocessor: 6 8 48
 Limited by Registers per Multiprocessor: 8
 Limited by Shared Memory per Multiprocessor: 32
 Note: Occupancy limiter is shown in orange
 Physical Max Warps/SM = 48
 Occupancy = 48 / 48 = 100%

CUDA Occupancy Calculator
 Version: 11,1
 Copyright and License



CUDA ECOSISTEMA

- NVIDIA CUDA-X:
 - Math Libraries:
 - cuBLAS, cuFFT, cuRAND, cuSOLVER, cuSPARSE, AmgX y CUDA Math Library.
 - cuTENSOR. GPU-accelerated tensor linear algebra library
 - Parallel Algorithms
 - Image and Video Libraries:
 - nvJPEG, NVIDIA Performance Primitives, NVIDIA Video Codec SDK y NVIDIA Optical Flow SDK.
 - Communication Libraries:
 - NVSHMEM y NCCL
 - Deep Learning:
 - NVIDIA cuDNN, NVIDIA TensorRT™, NVIDIA Jarvis, NVIDIA DeepStream SDK y NVIDIA DALI
- NVIDIA HPC SDK. Incluye compiladores, librerías y herramientas software para maximizar productividad, rendimiento y portabilidad de aplicaciones HPC.
- NVIDIA GPU Cloud™ (NGC). Contenedores para *frameworks* AI y aplicaciones HPC.
- ...

PROGRAMANDO ¿QUÉ ES CUDA?

- CUDA es C/C++ más un conjunto extensiones, tales como:
 - Calificadores de declaración que especifican lugares o ámbitos:

global void xxx(...)	<i>kernel</i> , se ejecuta en la GPU
device int xxx	variable en la memoria de la GPU
shared int xxx	variable en memoria compartida dentro del bloque
 - Sintaxis ampliada para la llamada al *kernel*:

kernel<<<a, b,c,d,>>>(...)	lanza la ejecución del <i>kernel</i>
---	--------------------------------------
 - Nuevos tipos / warpers:

dim3	vector de 3 enteros
-------------	---------------------
 - Variables especiales (intrínsecas) para identificar los hilos en el *kernel*:

threadIdx / blockIdx	ID de los hilos y bloques
blockDim / gridDim	dimensiones de los bloques y las <i>grid</i>
 - Operaciones específicas (intrínsecas) dentro del *kernel*:

_syncthreads()	barreda de sincronización dentro del <i>kernel</i>
-----------------------	--

PROGRAMANDO ¿QUÉ ES CUDA?

- Un *kernel* se ejecuta en paralelo por diferentes hilos CUDA.

Ejemplo definición función en C

```
void VecAdd(float* A, float* B, float* C) {  
    ...  
}
```

Ejemplo definición de kernel

```
_global_ void VecAdd(float* A, float* B, float* C) {  
    ...  
}
```

Ejemplo llamada a función en C

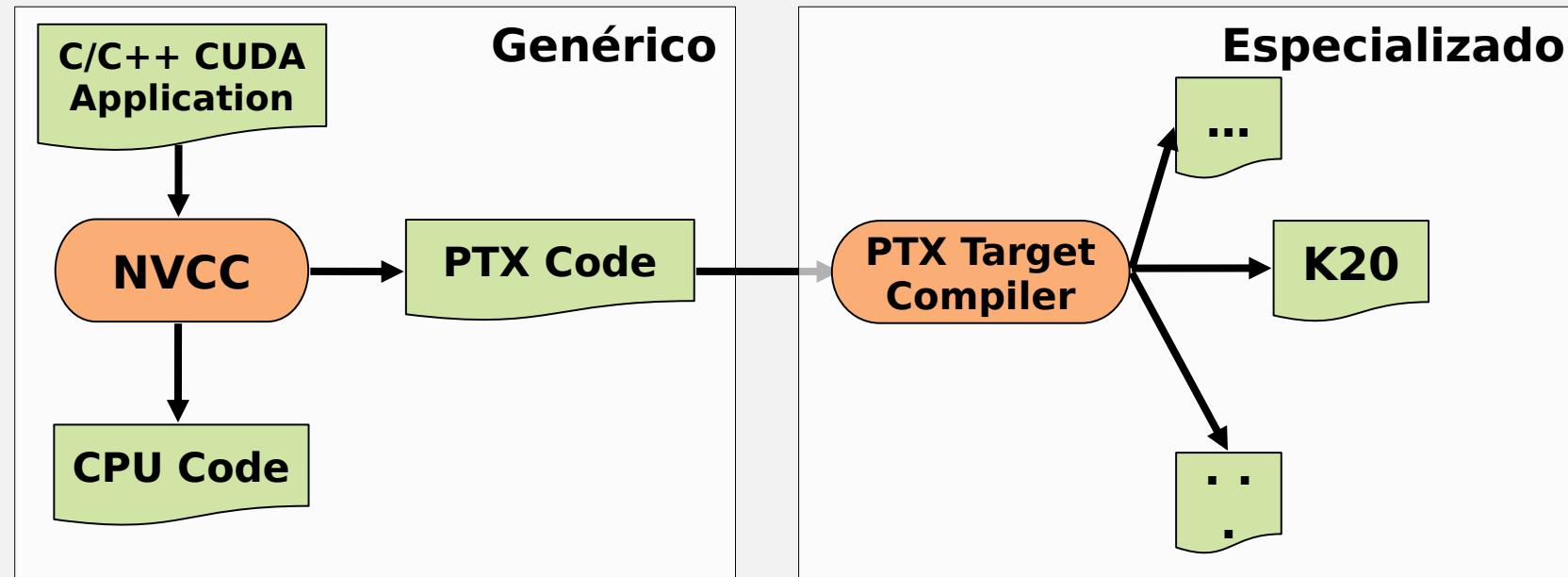
```
VecAdd(A, B, C);
```

Ejemplo llamada a kernel

```
VecAdd<<<1, n>>>(A, B, C);
```

PROGRAMANDO ¿QUÉ ES CUDA?

- El compilador **nvcc** separa los códigos CPU y GPU. Compilación en dos etapas:
 - **Virtual** genera código PTX (*Parallel Thread eXecution*).
 - **Física** genera binarios para GPU y CPU.



CUDA MODELO ORGANIZATIVO

- Los kernels se ejecutan en el dispositivo. Distintos kernels se pueden ejecutar simultáneamente.
- El kernel es ejecutado por hilos. Los hilos ejecutan el mismo código sobre diferentes datos basándose en su **Id**.
- Los hilos se agrupan en bloques de hilos.
- Los hilos del mismo bloque pueden **sincronizarse** (con `_syncthreads()`). Los hilos del mismo bloque pueden **compartir** datos (con *shared memory*).
- Los bloques no pueden sincronizarse: se ejecutan en cualquier orden, secuencial o paralelo.
- **Los kernels son asíncronos respecto a la CPU (retorno inmediato del control)**.
- Los kernels (del mismo *stream*), no comienzan su ejecución hasta que no hayan finalizado todas las llamadas CUDA anteriores ,p. ej. otro kernel (del mismo stream).
- Los kernels no finalizan hasta que finalicen todos sus hilos.

CUDA MODELO PROGRAMACIÓN

- GPU → GPC → SM → Core CUDA
- Bloque → Warp → hilo
- Dato → hilo

{}

¿ Cómo ?

- Cada hilo que ejecuta el kernel recibe un identificador (**Id**).
- Existe, puede existir, una relación entre el dato y el Id.
- A este Id se accede dentro del kernel a través de **variables intrínsecas**.
- Intrínseca **threadIdx** {threadIdx.x, threadIdx.y, threadIdx.z}
 - Identificador del hilo dentro del bloque (rango [0, blockSize-1]).
 - Los hilos están organizados dentro del bloques en 1, 2 ó 3 dimensiones.
 - Vector de 3 componentes. Hay un nuevo tipo de dato (**dim3**) a tal efecto.

CUDA MODELO PROGRAMACIÓN

- Hacer un kernel que sume los elementos de 2 vectores de tamaño n=1000000

Ejemplo V.1

```
__global__ void VecAdd(float* A, float* B,
float* C) {
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
VecAdd<<<1, n>>>(A, B, C);
```

¿ Resultado ejecución ?
*cudaCheckError() failed:
invalid configuration argument*

Ver 1

Ejemplo V.2

```
__global__ void VecAdd(float* A, float* B,
float* C) {
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
VecAdd<<<n/10, 10>>>(A, B, C);
```

¿ Resultado ejecución ?
*Error similar al de la V.1:
algo del número de bloques*

Ver 1

CUDA MODELO PROGRAMACIÓN

- Hacer un kernel que sume los elementos de 2 vectores de tamaño

Ejemplo V.3

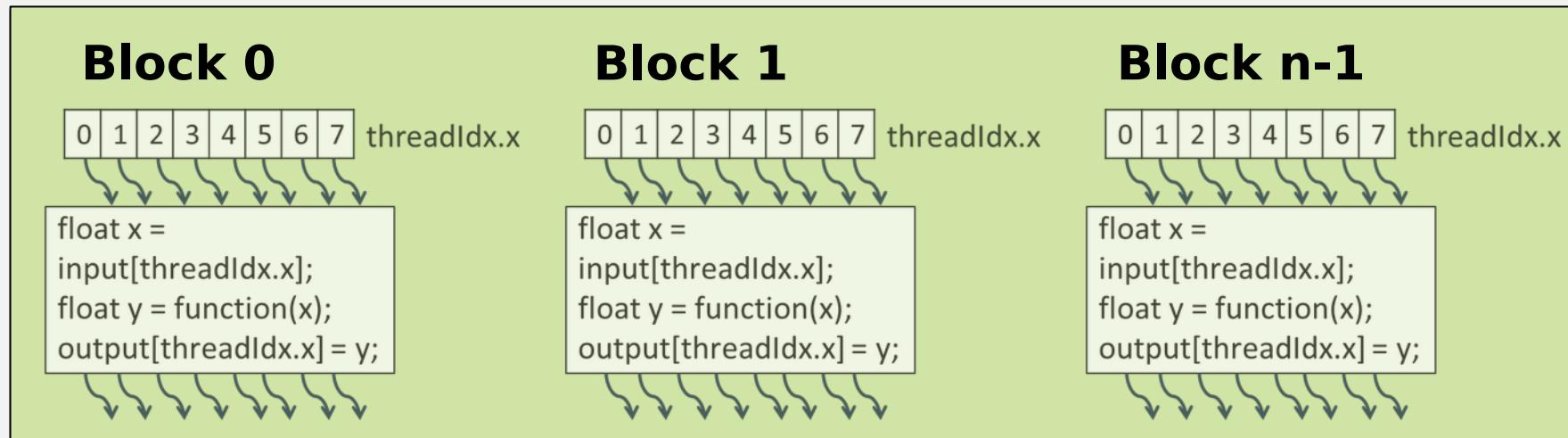
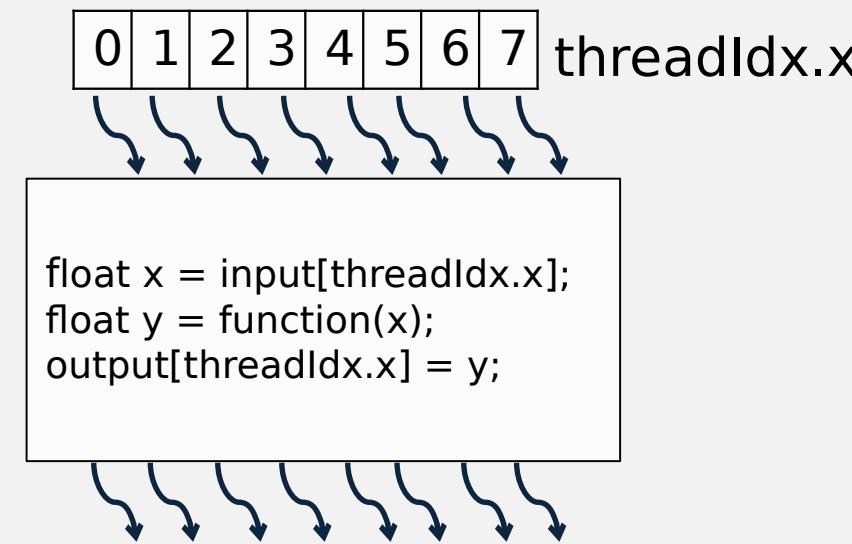
```
__global__ void VecAdd(float* A, float* B,  
float* C) {  
    int i = threadIdx.x;  
    C[i] = A[i] + B[i];  
}  
  
VecAdd<<<1000000/101, 101>>>(A, B, C);
```

¿ Resultado ejecución ?

(Suponiendo que no da los errores de V.1 y V.2)

Suma mal: suma (1000000/101) veces las 101 primeras posiciones.

CUDA MODELO ORGANIZATIVO



CUDA MODELO PROGRAMACIÓN

- Hacer un kernel que sume los elementos de 2 vectores de tamaño n=1000000

Ejemplo V.3

```
__global__ void VecAdd(float* A, float* B,  
float* C) {  
    int i = threadIdx.x;  
    C[i] = A[i] + B[i];  
}  
  
VecAdd<<<1000000/101, 101>>>(A, B, C);
```

Intrínseca **blockIdx** {blockIdx.x, blockIdx.y, blockIdx.z}

- Identificador del bloque dentro de la *grid*.
- Los bloques están organizados en *grids* de bloques de 1, 2 o 3 dimensiones.
- blockDim** es otra intrínseca *dim3* con las dimensiones del bloque.

CUDA MODELO PROGRAMACIÓN

- Hacer un kernel que sume los elementos de 2 vectores de tamaño n=1000000

Ejemplo V.4

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

VecAdd<<<1000000/101, 101>>>(A, B, C);
```

¿ Resultado ejecución ?

(Suponiendo que no da los errores de V.1, V.2 y V.3)

Suma mal: solo suma 999.900 posiciones.

Ejemplo V.5

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

¿ Resultado ?

```
VecAdd<<< (n + threadsPerBlock - 1) /  

threadsPerBlock, threadsPerBlock>>>(A, B, C);
```

CUDA MODELO PROGRAMACIÓN

- Hacer un kernel que sume los elementos de 2 vectores de tamaño

Ejemplo V.6

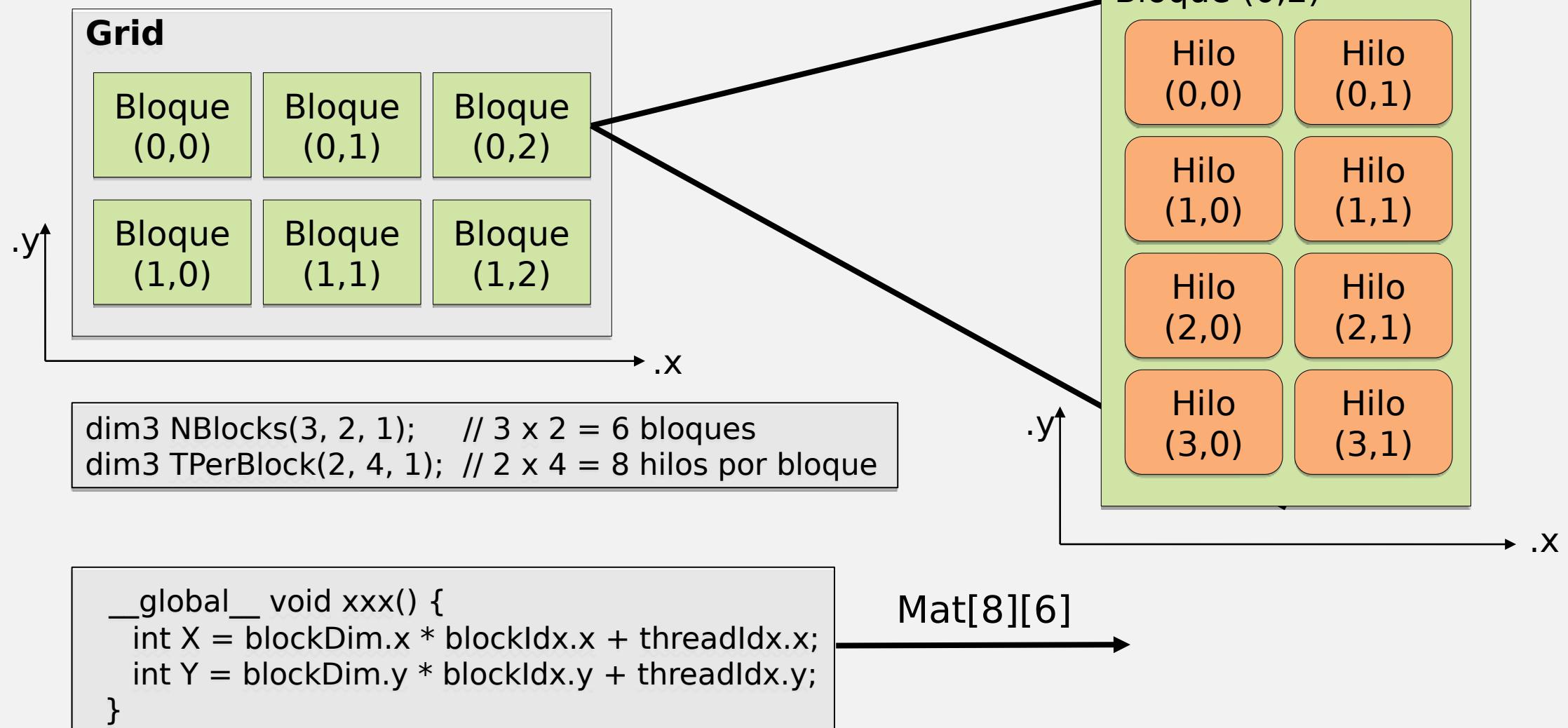
```
__global__ void VecAdd(float* A, float* B, float* C, int n) {  
    int i = blockIdx.x * blockDim.x + threadIdx.x;  
    if (i < n)  
        C[i] = A[i] + B[i];  
}  
VecAdd<<<(n + threadsPerBlock - 1)/threadsPerBlock,  
threadsPerBlock>>>(A, B, C, n);
```

¿ Resultado para **n** muy grande ?

*Podemos volver a los errores de la V.1 ó V.2
según sean los valores de threadsPerBlock y de n.*

CUDA MODELO ORGANIZATIVO

- La Grid: cómo se organizan los bloques



CUDA MODELO PROGRAMACIÓN

- Hacer un *kernel* que sume los elementos de 2 vectores de tamaño 4000x4000

Ejemplo V.7

```
__global__ void VecAdd(float* A, float* B, float* C, int rows, int cols) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    if ((y < rows) && (x < cols))
        C[y*cols+x] = A[y*cols+x] + B[y*cols+x];
}
dim3 TPB2d(threadsPerBlock, threadsPerBlock);
dim3 NB2d( (sqrt(n) + TPB2d.x - 1) / TPB2d.x, (sqrt(n) + TPB2d.y - 1) / TPB2d.y );
VecAdd<<< NB2d, TPB2d >>>(A, B, C, sqrt(n), sqrt(n));
```

Puede dar el mismo error que V.1, según el valor *threadPerBlock*

V.6 vs V.7

n = 4000x4000

Tiempo V.6 con: 1.0213000E-02 segundos

Tiempo V.7 con: 2.6340000E-03 segundos

CUDA MODELO PROGRAMACIÓN

Ejemplo V.8 (también válido para sumar matrices)

```
#define BLOCKSIZE 32

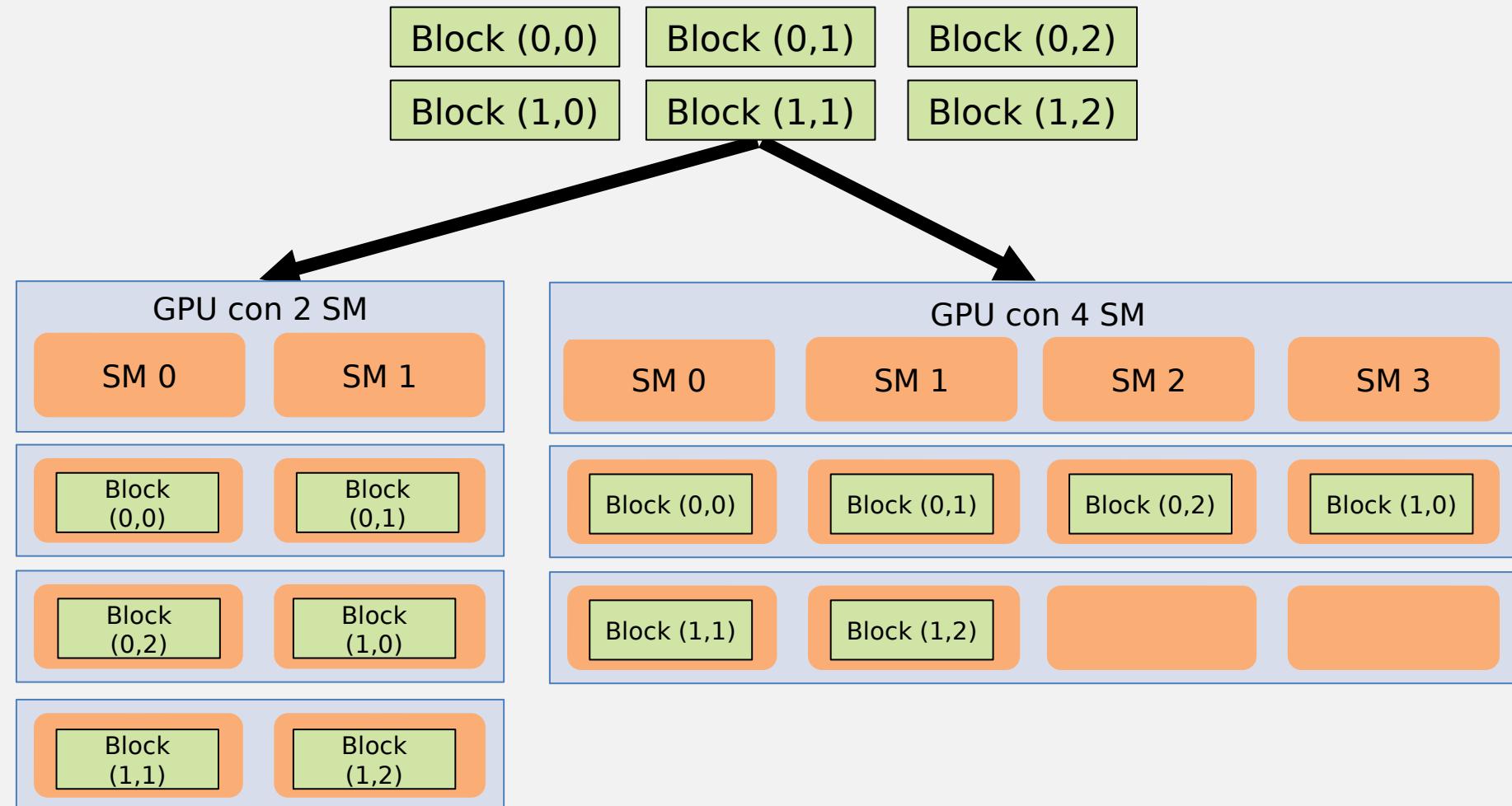
__global__ void MatSum(double *A, double *B, double *C, int rows, int cols) {
    int X= blockIdx.x * blockDim.x + threadIdx.x;
    int Y= blockIdx.y * blockDim.y + threadIdx.y;
    int index = Y * cols + X;

    if (Y < rows && X < cols) {
        C[index] = A[index] + B[index];
    }
}

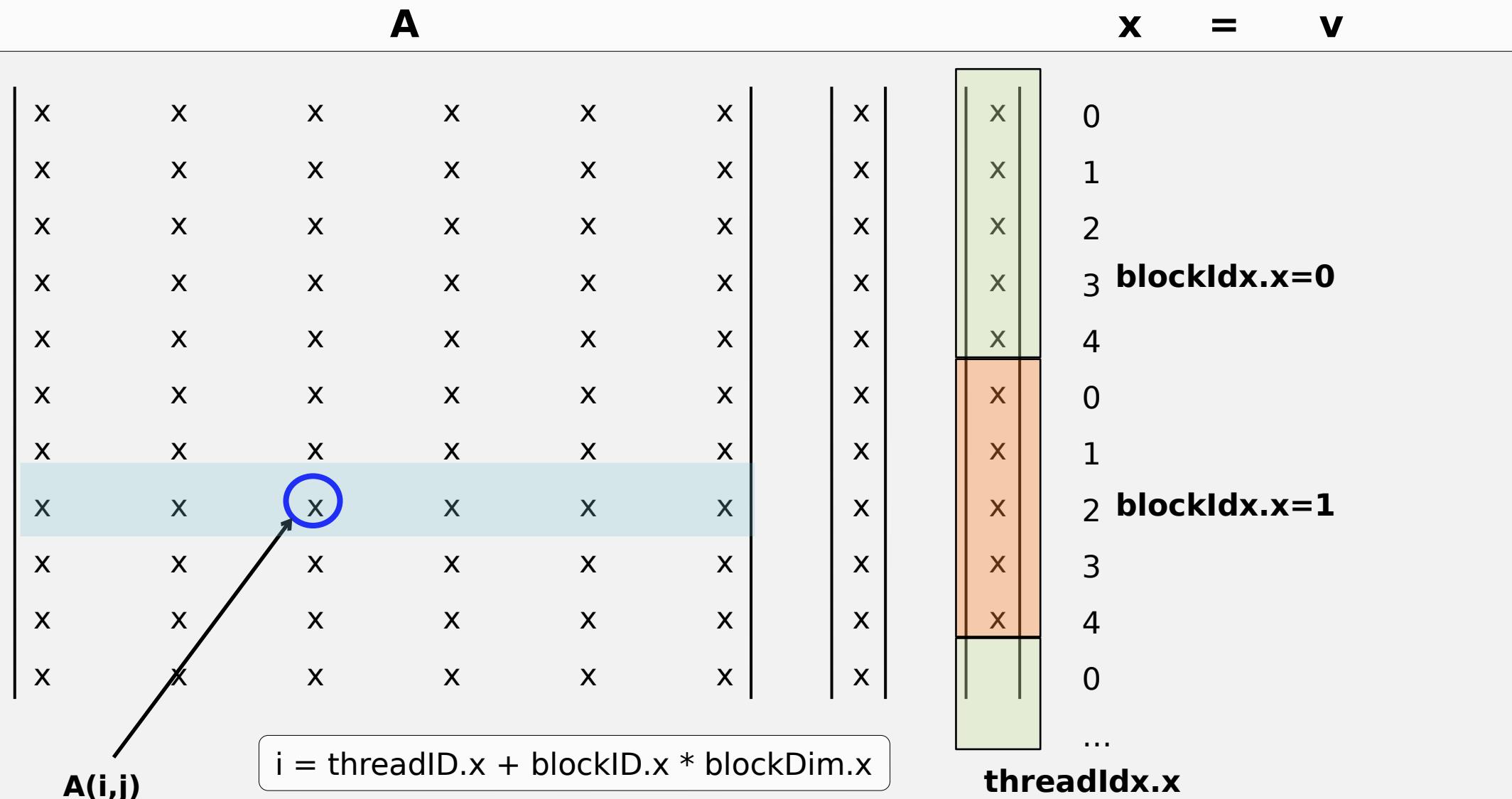
int main() {
    ...
    X = (int) ceil( (float)cols / BLOCKSIZE);
    Y = (int) ceil( (float)rows / BLOCKSIZE);
    dim3 TPBLCK(BLOCKSIZE, BLOCKSIZE);
    dim3 NBLCK(X, Y);
    MatSum<<<NBLCK, TPBLCK>>>(A, B, C, rows, cols);
    ...
}
```

CUDA MODELO ORGANIZATIVO

- Escalabilidad automática



EJEMPLO



EJEMPLO

Ejemplo $v = A * x$

```
__global__ void MatdotVec(double *A, double *x, double *v, int rows, int cols) {
    int i = blockIdx.x * blockDim.x + threadIdx.x; // global matrix row index
    int j;
    double tmp=0.0;

    if (i < rows)
        for (j=0; j < cols; j++)
            tmp += A[i*cols+j] * x[j];
        v[i] = tmp;
    }

int main() {
    ...
    dim3 TPBLCK(XXX, 1, 1); dim3 NBLCK(ZZZ, 1, 1);
    MatdotVec<<<NBLCK, TPBLCK>>>(A, x, v, rows, cols);
    ...
}
```

CUDA MODELO ORGANIZATIVO

¿Por qué, en general, almacenar matrices como vectores?

- Por todo lo visto hasta ahora: localidad, librerías, etc.

Y entonces ¿por qué en CUDA usan *grids/blocks* 2D si lo *recomendado* es almacenar en 1D? ¿No es complicarlo?

- Por flexibilidad, limitaciones, rendimiento potencialmente, etc.
- La localidad en GPU es igual de importante que en CPU, pero se *entiende* de otra manera (segunda parte).

Ejemplo binarizado

binariza 1000 2000 16

El tiempo en CPU: 3.8437440E-03 segundos.

El tiempo con CuBLAS 1D: vecbin.cu(114): getLastCudaError() CUDA error...

El tiempo con CuBLAS 2D: 2.1923199E-04 segundos.

binariza 1000 2000 32

El tiempo en CPU: 3.8437440E-03 segundos.

El tiempo con CuBLAS 1D: 4.7323201E-04 segundos.

El tiempo con CuBLAS 2D: 2.7491199E-04 segundos.

CUDA CONTROL ERRORES

- Todas las funciones de la API de CUDA (cuBLAS, cuFFT, etc.) retornan un valor. Se debe comprobar.

Ejemplo

```
int main() {  
    ...  
    if (cudaMemcpy(...) != cudaSuccess)  
}
```

Mejor así

```
#define CUDAERR(x) do { if((x)!=cudaSuccess) { \  
    printf("CUDA error: %s : %s, line %d\n", cudaGetErrorString(x), __FILE__, __LINE__);\  
    return EXIT_FAILURE; } } while(0)  
int main() {  
    ...  
    CUDAERR(cudaMemcpy(...));  
}
```

CUDA CONTROL ERRORES

- Los kernel son *void* (no devuelven nada) y asíncronos con la CPU. Sus errores se gestionan con *cudaGetLastError*.

Ejemplo

```
kernel<<<...>>>(...);  
if (cudaSuccess != cudaGetLastError()) ...
```

O también

```
#define CUDACHECK() __getLastCudaError(__FILE__, __LINE__)  
inline void __getLastCudaError(const char *file, const int line) {  
    cudaError_t err = cudaGetLastError();  
    if (cudaSuccess != err) {  
        fprintf(stderr, "%s(%i): getLastCudaError() error: (%d) %s.\n", file, line, (int)err, cudaGetString(err));  
        exit(-1);  
    }  
}
```

CUDA EVENTOS

- Muchos usos, uno es medir tiempos.

Aquí ¿ Cuántos segundos tarda el kernel ?

```
Ctimer(&elapsed, &ucpu, &scpu, 0);  
    kernel<<<...>>>(...)  
Ctimer(&elapsed, &ucpu, &scpu, 1);
```

En todo caso

```
Ctimer(&elapsed, &ucpu, &scpu, 0);  
    kernel<<<...>>>(...);  
    cudaDeviceSynchronize()  
Ctimer(&elapsed, &ucpu, &scpu, 0);
```

O si procede y tiene sentido

```
Ctimer(&elapsed, &ucpu, &scpu, 0);  
    kernel<<<...>>>(...);  
    cudaMemcpy(...)  
Ctimer(&elapsed, &ucpu, &scpu, 0);
```

CUDA EVENTOS

Pero tal vez sea más apropiado

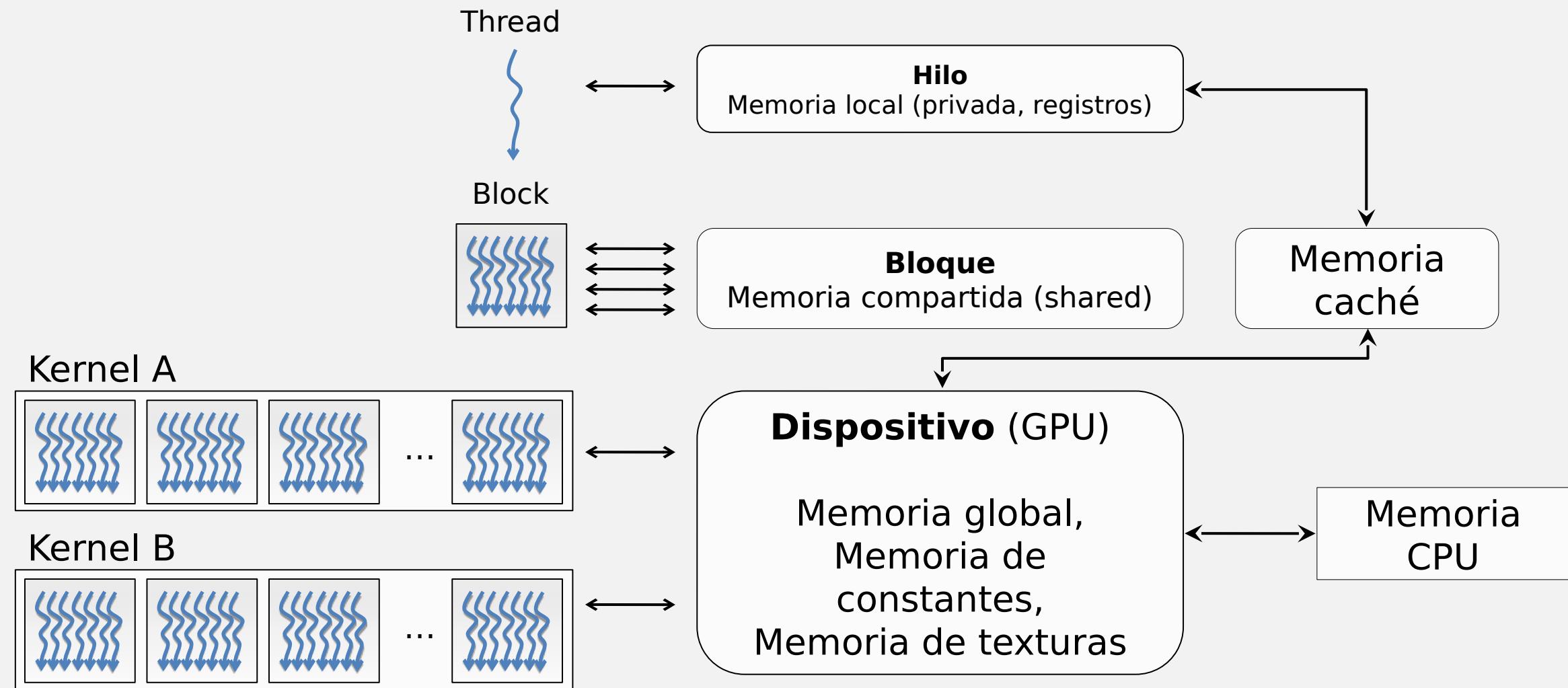
```
int main() {
    cudaEvent_t INICIO, FIN;
    float time;
    cudaEventCreate(&INICIO); cudaEventCreate(&FIN);
    ...
    // Record event INICIO starts on stream 0
    cudaEventRecord(INICIO, 0);
    kernel<<<...>>>(...);
    // Record event FIN starts on stream 0
    cudaEventRecord(FIN, 0);

    // Wait until all device work before its call on stream 0 ends.
    cudaEventSynchronize(FIN);
    cudaEventElapsedTime(&time, INICIO, FIN);
    printf("%f milisegundos\n", time);
    cudaEventDestroy(INICIO); cudaEventDestroy(FIN);
}
```

GPU / CUDA JERARQUÍA DE MEMORIA

- Los hilos CUDA pueden acceder a los datos desde múltiples espacios de memoria durante su ejecución.
- Cada hilo tiene su memoria local privada. Todos los hilos tienen acceso a la memoria global.
- Cada bloque de hilos tiene su espacio de memoria compartida (shared) accesible/visible a todos sus hilos y con el mismo ciclo de vida que el bloque.
- Las GPUs tienen cachés L2 y L1, que se comportan de forma similar a las cachés de la CPU. La L1 comparte espacio (recursos físicos) con la shared.
- Las memorias de texturas y constantes son espacios de memoria adicionales accesibles a todos los hilos.
- Los espacios de memoria global, texturas y constantes tiene el mismo ciclo de vida que la aplicación (persistente durante su ejecución).

GPU / CUDA JERARQUÍA DE MEMORIA



CUDA API RESERVANDO MEMORIA

- Para reservar:

cudaMalloc(), **cudaMallocPitch()**, **cudaMalloc3D()**, **cudaMallocHost()**, etc.

Permiten reservar memoria lineal (CUDA arrays, *layouts* de memoria opacas optimizadas).

- Para liberar:

cudaFree(), **cudaFreeHost()**, etc.

- Para inicializar:

cudaMemset(), etc.

- Para copiar:

cudaMemcpy(), **cudaMemcpyAsync()**, etc. Hay que indicar el tipo: cudaMemcpyDeviceToDevice, cudaMemcpyDeviceToHost, cudaMemcpyHostToDevice, cudaMemcpyHostToHost, cudaMemcpyDefault. En cudaMemcpyDefault el tipo de transferencia se infiere del valor del puntero y se permite en sistemas con soporte UVA (Unified Virtual Addressing).

- Para prebúsquedas:

cudaMemPrefetchAsync(), etc.

CUDA MODELO CLÁSICO (HANDMADE) RESERVA MEMORIA

Ejemplo

```
int main() {
    double *Host_Mat, *Device_Mat;
    int size=m*n*sizeof(double);

    Host_Mat=(double *)calloc(m*n, sizeof(double));
    cudaMalloc((void **) &Device_Mat, size);

    cudaMemcpy(Device_Mat, Host_Mat, size, cudaMemcpyHostToDevice);

    // hacer cosas en CPU con Host_Mat y en GPU con Device_Mat

    cudaMemcpy(Host_Mat, Device_Mat, size, cudaMemcpyDeviceToHost);

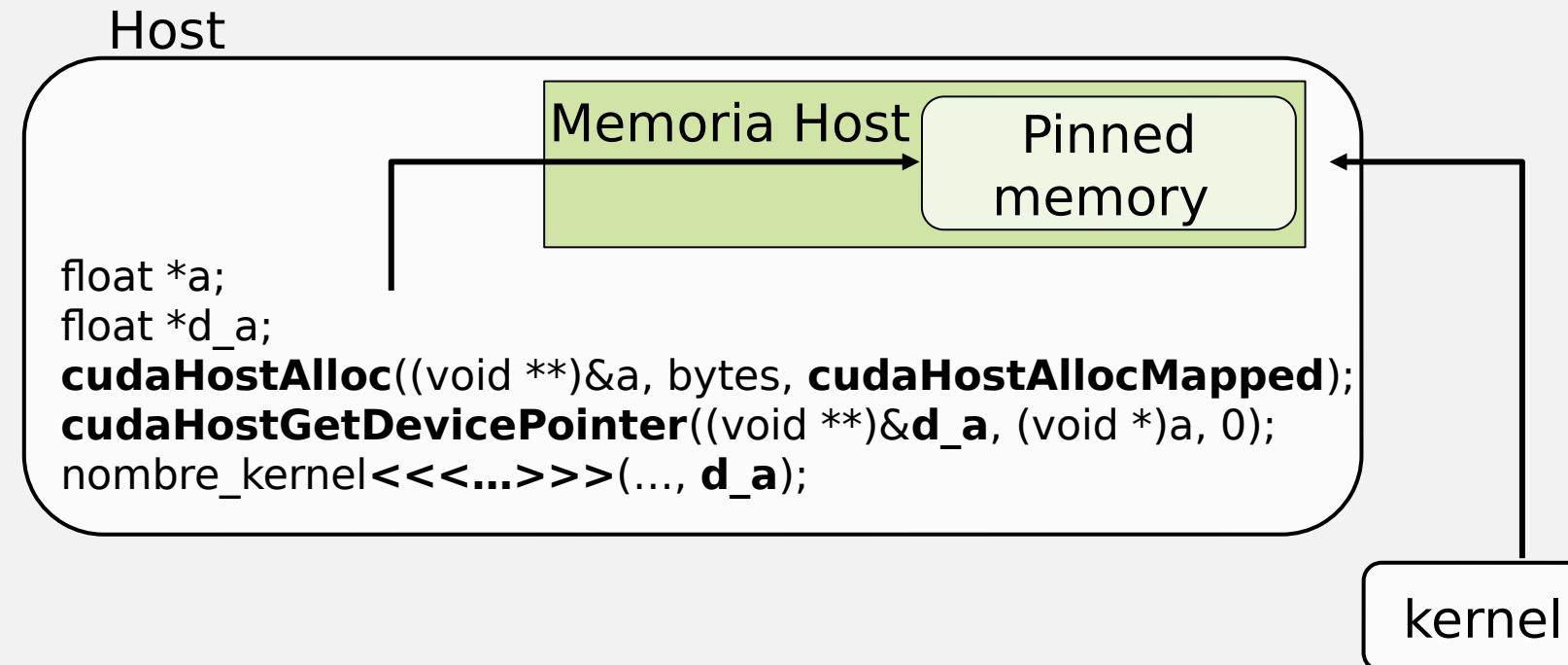
    // hacer más cosas

    cudaFree(Device_Mat);
    free(Host_Mat);
}
```

CUDA PINNED MEMORY

cudaHostAlloc(void ptr, size_t size, unsigned int flag)**

- Reserva (como **cudaMallocHost**) memoria en la RAM del Host (CPU), potencialmente accesible a la GPU.
- La reserva es de tipo *page-locked (pinned memory)*, que acelera las transferencias de información.
- Su uso excesivo puede degradar el rendimiento.



CUDA PINNED MEMORY

cudaHostAlloc(void ptr, size_t size, unsigned int flag)**

- Valores de **flag**
 - *cudaHostAllocDefault*. Equivalente a `cudaMallocHost()`.
 - *cudaHostAllocPortable*. *Pinned para todos los contextos CUDA.*
 - *cudaHostAllocMapped*. *Mapea el espacio de direcciones de la CPU en la GPU. El puntero se obtiene con `cudaHostGetDevicePointer()`. Usado en el ejemplo anterior.*
 - *cudaHostAllocWriteCombined*. Es *write-combined (WC)*. Según Nvidia: “*WC memory can be transferred across the PCI Express bus more quickly on some system configurations, but cannot be read efficiently by most CPUs. WC memory is a good option for buffers that will be written by the CPU and read by the device via mapped pinned memory or host→device transfers*”.

CUDA PINNED MEMORY

Ejemplo Clásica

```
int main() {
    double *Host_Mat, *Device_Mat;
    int size=m*n*sizeof(double);

    cudaHostAlloc((void **) &Host_Mat, size, cudaHostAllocDefault);
    cudaMalloc ((void **) &Device_Mat, size);

    cudaMemcpy(Device_Mat, Host_Mat, size,
    cudaMemcpyHostToDevice);

    // hacer cosas en CPU con Host_Mat y en GPU con Device_Mat
    cudaMemcpy(Host_Mat, size, Device_Mat,
    cudaMemcpyDeviceToHost);
    // hacer más cosas

    cudaFree(Device_Mat);
    cudaFreeHost(Host_Mat);
}
```

CUDA PINNED MEMORY

Ejemplo

```
int main() {
    double *Host_Mat, *Device_Mat;
    int size=m*n*sizeof(double);

    cudaHostAlloc((void **) &Host_Mat, size, cudaHostAllocMapped);

    cudaHostGetDevicePointer((void **)&Device_Mat, (void *)Host_Mat, 0);

    // hacer cosas en CPU con Host_Mat y en GPU con Device_Mat

    cudaFreeHost(Host_Mat);
}
```

Hay ciertas limitaciones (quién, cuándo y cómo se accede a los datos) en este caso (*cudaHostAllocMapped*).

CUDA UNIFIED MEMORY

- *Pinned* tiene limitaciones y problemas de estabilidad si no se usa correctamente.
- Memoria Unificada (*Unified Memory*) hace *lo mismo* de forma más eficiente y con un mayor rango de uso.
- La reserva se hace en “alguna memoria” (¿CPU? ¿GPU? Depende...). El *mismo* puntero es accesible por la CPU y la GPU.
- No todos los sistemas la soportan y en arquitecturas previas a *Pascal* su comportamiento no es bueno.

cudaMallocManaged(void devPtr, size_t size, unsigned int flag)**

Flag especifica el *stream* asociado, que puede ser:

- *cudaMemAttachGlobal*: Accesible por cualquier *stream* en cualquier dispositivo.
- *cudaMemAttachHost*: No accesible para dispositivos con atributo *cudaDevAttrConcurrentManagedAccess* con valor 0.
- Se libera memoria de forma normal: *cudaFree*.

CUDA UNIFIED MEMORY

Ejemplo

```
_global_ void add(int n, float *x, float *y) {
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;
    for (int i = index; i < n; i += stride) y[i] = x[i] + y[i];
}
int main() {
    int N = 1<<20, blockSize = 256, numBlocks = (N + blockSize - 1) / blockSize;
    float *x, *y;
    cudaMallocManaged(&x, N*sizeof(float));
    cudaMallocManaged(&y, N*sizeof(float));
    for (int i = 0; i < N; i++) { x[i] = 1.0f; y[i] = 2.0f; }
    add<<<numBlocks, blockSize>>>(N, x, y);
    cudaDeviceSynchronize();
    cudaFree(x);  cudaFree(y);
    return 0;
}
```

CUDA MEMORIA DE CONSTANTES

En CPU

```
__constant__ double GPUCostes[5];  
...  
int main() {  
    double Costes[5] = {1.0, 4.0, 6.0, 4.0, 1.0};  
    cudaMemcpyToSymbol(GPUCostes, Costes,  
    sizeof(Costes));  
    ...  
    kernel_ejemplo<<<...,...>>(devMat);  
    ...  
}
```

En GPU

```
__global__ void kernel_ejemplo(double *Mat) {  
    ...  
    Mat[i] = Mat[i] * GPUCostes[0] + ...;
```

CUDA SHARED MEMORY

- Se reserva usando el calificador `_shared_`
- Es más rápida (se espera que sea) que la memoria *global*.
- Se divide en módulos de igual tamaño (*banks*) a los que se puede acceder simultáneamente.
- El número de bancos es 32 (¿casualidad?) y ancho de banda de 32 bits por 2 ciclos de reloj (puede cambiar).
- Los accesos a n direcciones de n bancos distintos se pueden hacer simultáneamente: ancho de banda n veces mayor que el ancho de banda de un solo módulo.
- 2 accesos simultáneos al mismo banco generan *conflicto de acceso* y se produce *serialización* en el acceso.
- Los hilos de un *warp* no generan conflicto de banco si acceden dentro de la misma palabra de 32 bits.

CUDA SHARED MEMORY

En tiempo de compilación

```
_global_ void kernel(...) {  
    _shared_ float sData[256];  
    ...  
}  
void main() {  
    kernel<<<nB, nT>>>(...);  
}
```

En tiempo de ejecución

```
_global_ void kernel(...) {  
    extern _shared_ float sData[ ];  
    ...  
}  
void main() {  
    kernel<<<nB, nT, nT*sizeof(float)>>>(...);  
}
```

CUDA SHARED MEMORY

Cache

```
__global__ void EjemploCache(float* A, float *F, ...) {  
    int i = blockDim.x * blockIdx.x + threadIdx.x;  
    int j = blockDim.y * blockIdx.y;  
    ...  
    temp += A[t]*F[0];  
    ...  
}
```

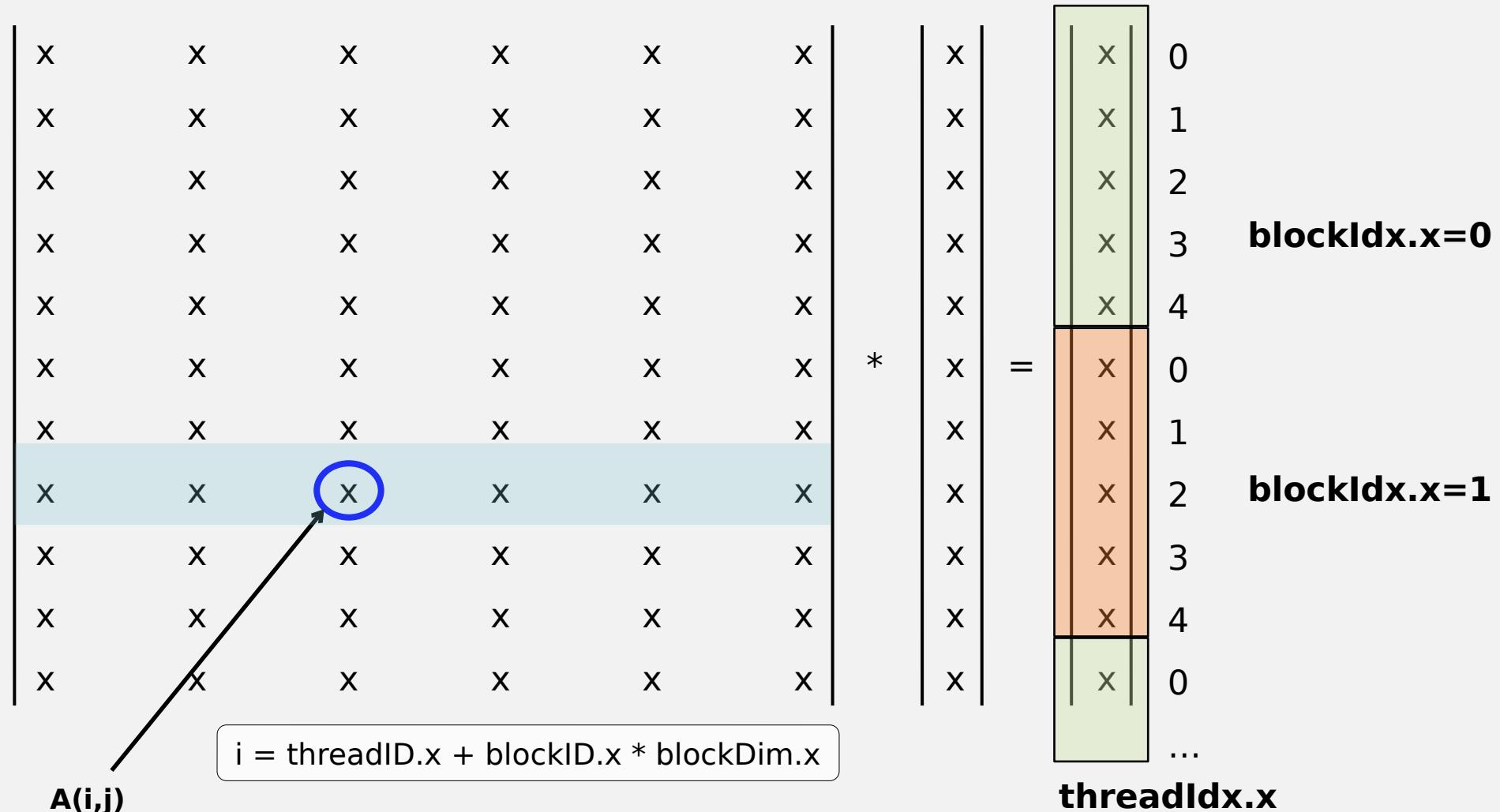
Shared

```
__global__ void EjemploShared(float* A, float *F, ...) {  
    __shared__ float copia[(256+2)*3];  
    int i = blockDim.x * blockIdx.x + threadIdx.x;  
    int j = blockDim.y * blockIdx.y;  
    ...  
    // write data to shared memory  
    copia[threadIdx.x + 2] = A[t];  
    t += N;  
    copia[dim + threadIdx.x + 2] = A[t];  
    ...  
    __syncthreads();  
    ...  
    temp += copia[t]*F[0];  
    ...  
}
```

CUDA SHARED MEMORY

Recordando

$$\mathbf{v} = \mathbf{A} * \mathbf{x}$$



CUDA SHARED MEMORY

Ejemplo $v = A * x$

```
__global__ void MatdotVec(double *A, double *x, double *v, int rows, int cols) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j;
    double tmp=0.0;

    if (i < rows)
        for (j=0; j < cols; j++)
            tmp += A[i*cols+j] * x[j];
    v[i] = tmp;
}

int main() {
    ...
    dim3 TPBLCK(XXX, 1, 1);  dim3 NBLCK(ZZZ, 1, 1);
    MatdotVec<<<NBLCK, TPBLCK>>>(A, x, v, rows, cols);
    ...
}
```

CUDA SHARED MEMORY

$$v = A * x$$

Shared memory



CUDA SHARED MEMORY

Ejemplo $v = A * x$ con Shared

```
_global_ void MatdotVecSh(double *A, double *x, double *v, int rows, int cols) {
    int j, i=blockIdx.x * blockDim.x + threadIdx.x;
    extern __shared__ double sh_x[];
    double tmp=0.0;
    if (i < rows){
        if (threadIdx.x == 0)
            for (j=0; j < cols; j++) { sh_x[j] = x[j]; }
        __syncthreads();
        for (j=0; j < cols; j++) { tmp += A[i*cols+j] * sh_x[j]; }
        v[i] = tmp;
    }
}
int main() {
    dim3 TpBlock (XXX, 1, 1);
    dim3 Nblocks (ZZZ, 1, 1);
    MatdotVec<<<Nblocks, TpBlock, n*sizeof(double)>>>(A, x, v, rows, cols);
}
```

CUDA SHARED MEMORY

```
$ matdotvec
```

Uso: matdotvec <rows> <cols> <seed> <threadsPerBlock>

```
$ matdotvec 10000 4000 13 32
```

INFO: Hay 1 GPUs con capacidades CUDA, seguimos

El tiempo en CPU es 4.7871487E-02 segundos.

El tiempo en GPU es 1.2955520E-02 segundos.

El tiempo con SH es 3.0112000E-05 segundos.

“los tiempos de GPU incluyen la copia al Host del resultado”

```
$ matdotvec 4000 10000 13 32 (si compila...)
```

INFO: Hay 1 GPUs con capacidades CUDA, seguimos

El tiempo en CPU es 4.8071648E-02 segundos.

El tiempo en GPU es 1.0892768E-02 segundos.

El tiempo con SH es 2.7676960E-02 segundos.

¿ La memoria compartida es infinita ?

CUDA SHARED MEMORY

Ejemplo $v = A * x$ con Shared V2: “n debe ser divisible por 4”

```
__global__ void MatdotVecSH2(double *A, double *x, double *v, int rows, int cols) {
    int k, j, i=blockIdx.x * blockDim.x + threadIdx.x;
    __shared__ double sh_x[4];           // o cualquier otro valor razonable
    double tmp=0.0;

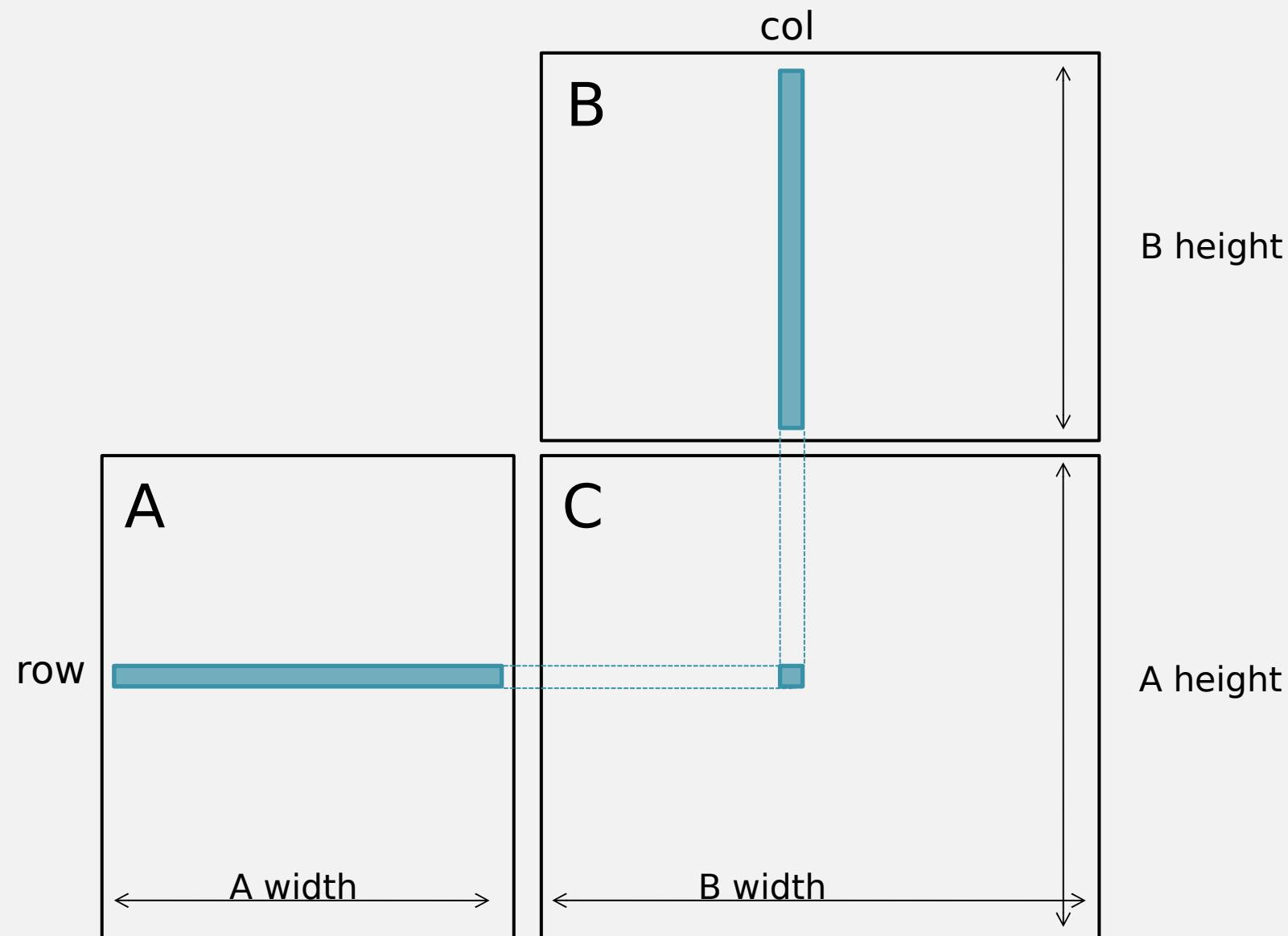
    if (i < rows)
        for (k=0; k < cols; k+=4)
    {
        if (threadIdx.x == 0)
            for (j=0; j<4; j++) { sh_x[j] = x[k+j]; }
        _syncthreads();

        for (j=0; j<4; j++) { tmp += A[i*cols+k+j] * sh_x[j]; }
        _syncthreads();           // evita que si el 0 avanza antes cambie sh_x antes de ...
    }
    v[i] = tmp;
```

Puede ser más lento o rápido dependiendo de *rows*, hilos por bloque, etc.

CUDA SHARED MEMORY

$$C = A * B$$



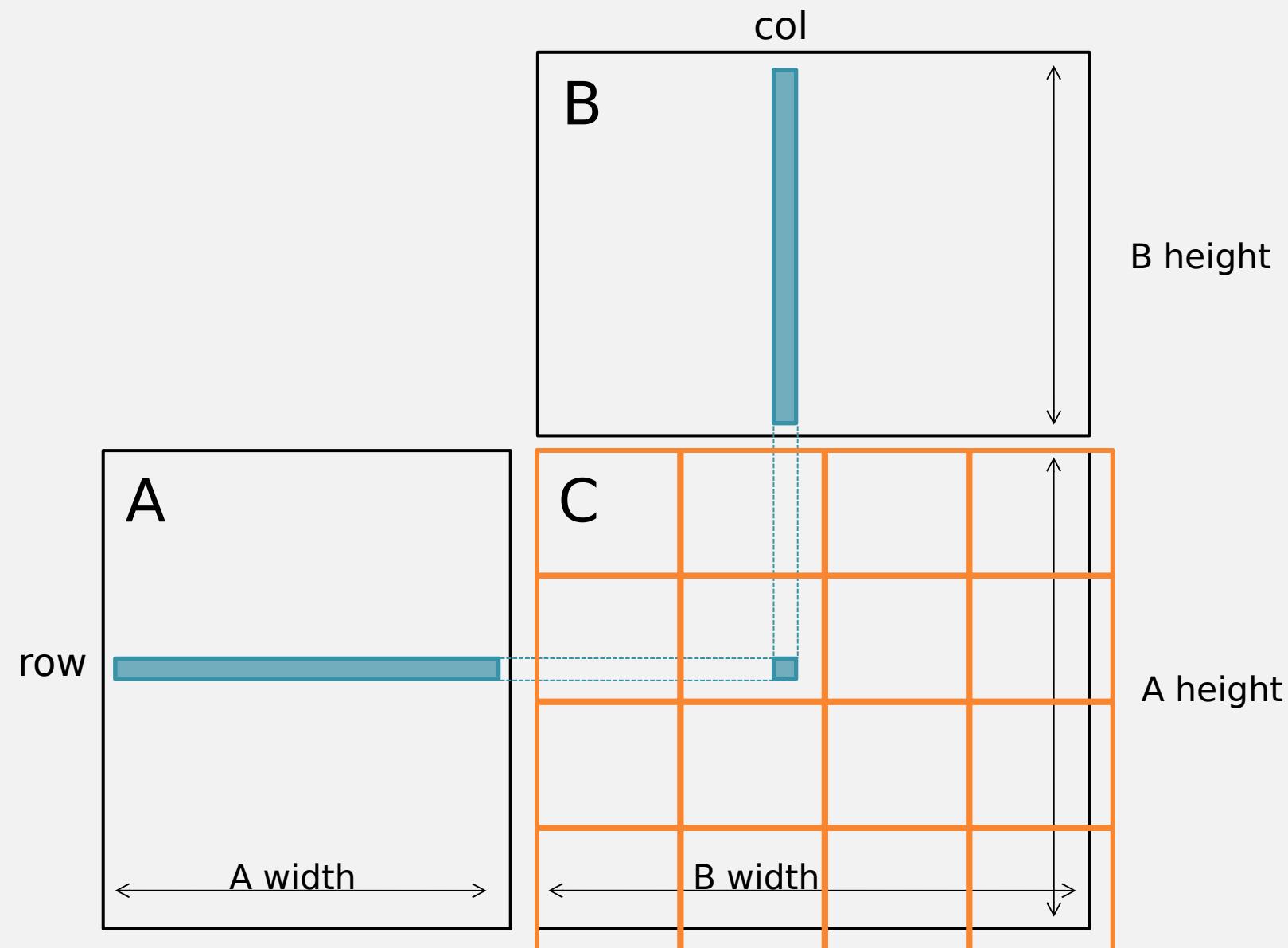
CUDA SHARED MEMORY

C = A * B

```
__global__ void kernel_MatMul(int m, int n, int k, double *C, int IdC, double *A, int IdA, double *B, int IdB) {  
    int i = blockIdx.y * blockDim.y + threadIdx.y; //rows  
    int j = blockIdx.x * blockDim.x + threadIdx.x; //cols  
    int r;  
    double dtmp=.0;  
  
    if (i<m && j<n)  
    {  
        for (r = 0; r < k; r++)  
            dtmp += A[i * IdA + r] * B[r * IdB + j];  
        C[i * IdC + j] = dtmp;  
    }  
}  
...  
dim3 NBlk((int)ceil((float)n/threadsPerBlock), (int)ceil((float)m/threadsPerBlock), 1);  
dim3 TpBlk(threadsPerBlock, threadsPerBlock, 1);  
kernel_MatMult<<<NBlk, TpBlk>>>(m, n, k, Devi_MatC, n, Devi_MatA, n, Devi_MatB, k);
```

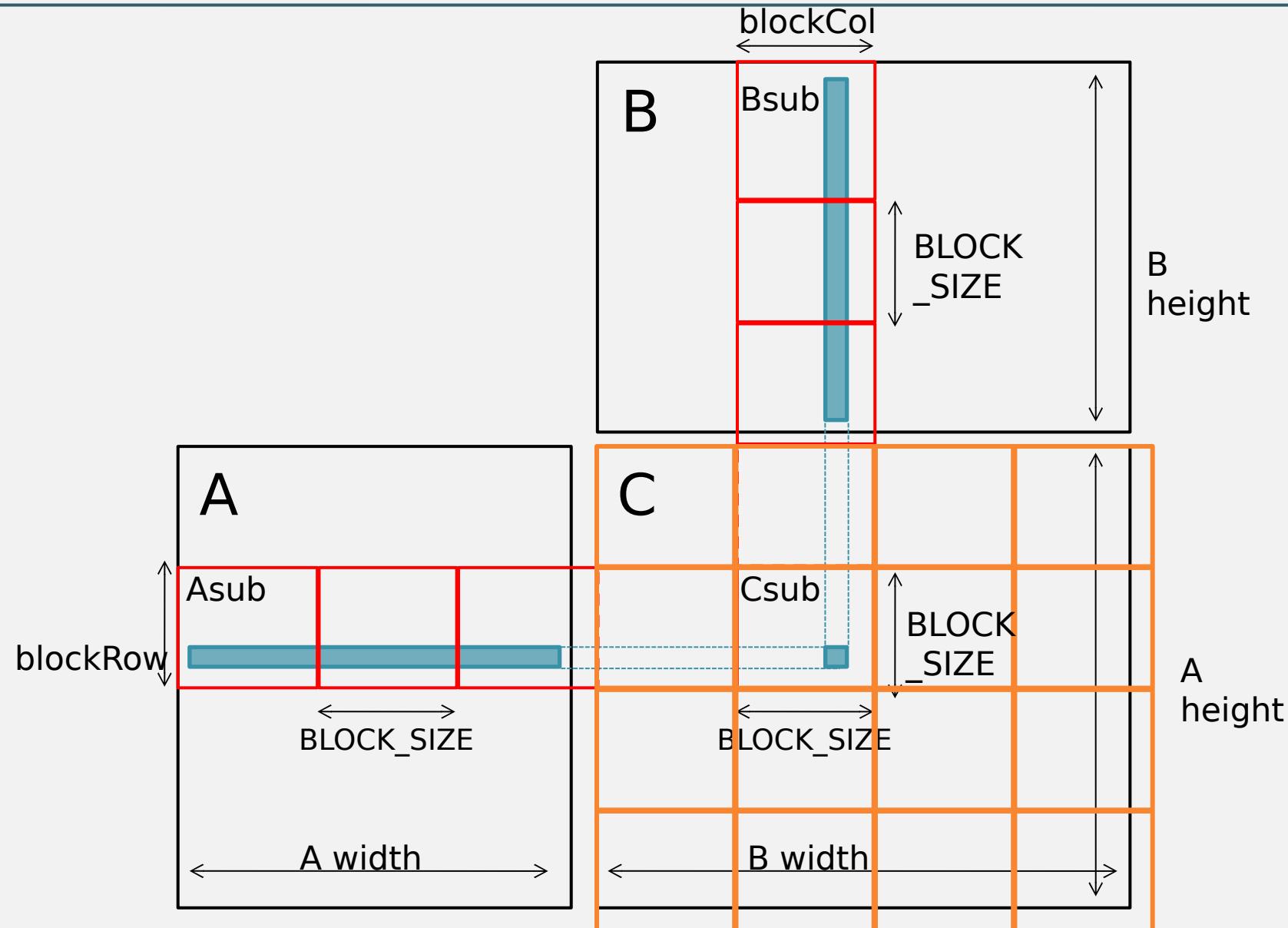
CUDA SHARED MEMORY

$$C = A * B$$



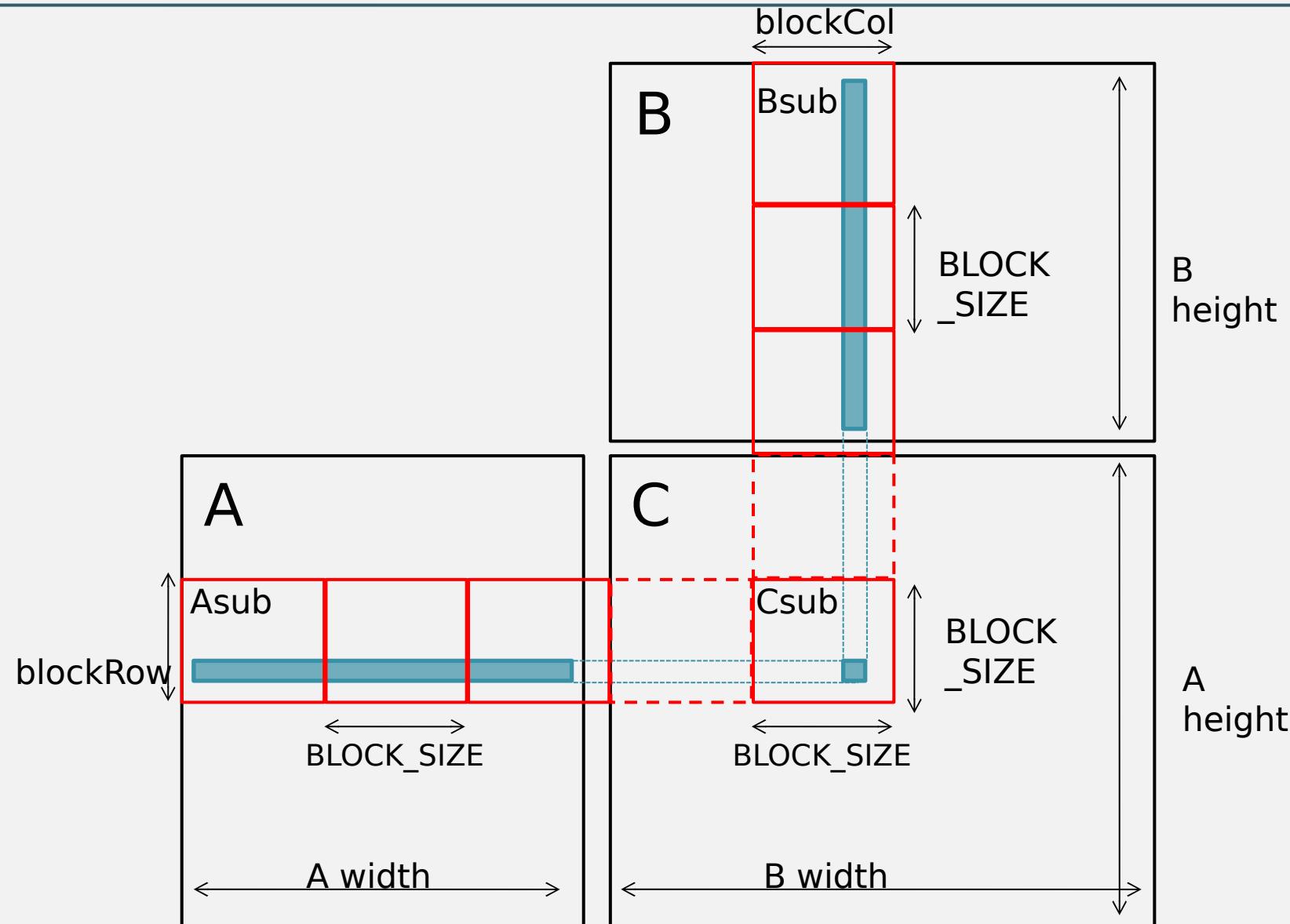
CUDA SHARED MEMORY

$$C = A * B$$



CUDA SHARED MEMORY

$$C = A * B$$



CUDA SHARED MEMORY

C = A * B

```
$ matmul_cuda  
Uso: matmul_cuda <m> <n> <k> <seed> <threadsPerBlock>
```

```
$ matmul_cuda 4000 4000 4000 11 16  
INFO: Hay 1 GPUs con capacidades CUDA, seguimos  
El tiempo en CPU con MKL es 3.2251323E+00 segundos.  
El tiempo en GPU es 2.1978037E+00 segundos.  
El tiempo en GPU Shared es 1.0863396E+00 segundos.
```

```
$ matmul_cuda 4000 4000 4000 11 32  
INFO: Hay 1 GPUs con capacidades CUDA, seguimos  
El tiempo en CPU con MKL es 3.8699146E+00 segundos.  
El tiempo en GPU es 2.9133772E+00 segundos.  
El tiempo en GPU Shared es 1.0914811E+00 segundos.
```

CUDA NIVELES DE CONCURRENCIA

- Las siguientes operaciones se pueden ejecutar simultáneamente entre sí:
 - Computación en el Host (CPU).
 - Ejecución de kernels en el *device* (GPU).
 - Transferencia de información del Host a la GPU.
 - Transferencia de información de la GPU al Host.
 - En la copia de datos entre direcciones de memoria del mismo subsistema.
 - Transferencia de información entre memorias de varias GPUs.
- El nivel de concurrencia alcanzado depende de las características y de la CC del dispositivo.
- En las transferencias de datos con *pinned* y la ejecución de los *kernels*.
- En la copia de datos entre la CPU y las GPUs cuando son bloques de tamaño ≤ 64 KB.
- En los movimientos de datos con funciones *Async*.
- ...

COMPUTE UNIFIED DEVICE ARCHITECTURE

BEST PRACTICES



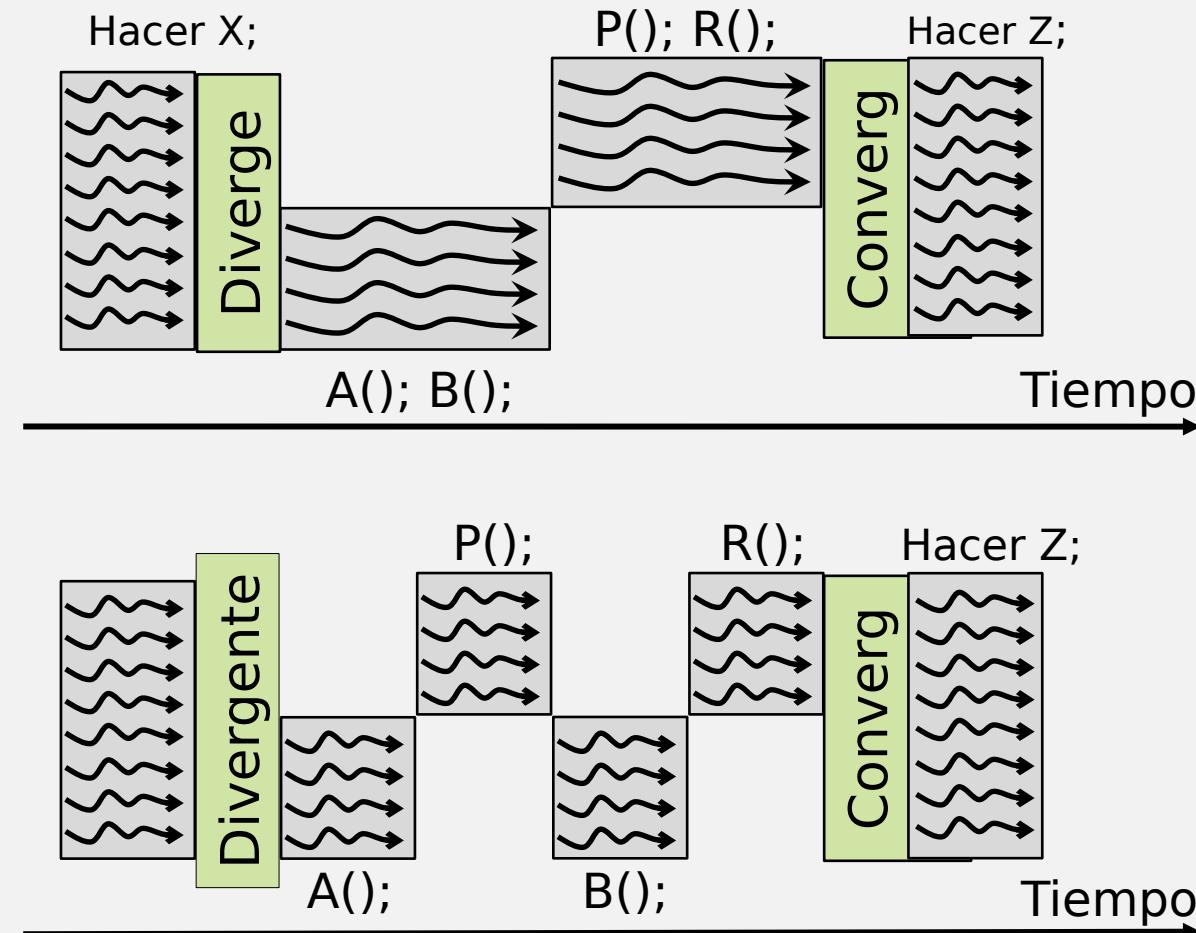
SALTOS DIVERGENTES (*BRANCH DIVergENCE*)

- La importancia de los *Saltos Divergentes* es a nivel de *warp*.
(Un warp es un grupo de 32 hilos (también hay *half-* y *quarter-warps*)).
- Todos los hilos de un warp comienzan en la misma dirección de programa, pero tienen su propio contador de direcciones de instrucción por lo que tienen independencia de ejecución.
- Un warp ejecuta una instrucción común a cada instante de tiempo:
 - La mayor productividad se alcanza cuando todos los hilos del warp ejecutan el camino programado.
 - Si los hilos del warp divergen (p. ej. salto condicional) el warp ejecuta *secuencialmente* todos los caminos.
 - En cada camino los hilos que no estén en él son deshabilitados.
 - Ejecutados todos los posibles caminos, los hilos convergen en su ejecución.
- Evitar los saltos divergentes por la pérdida de rendimiento que implican (en CUDA todo puede cambiar).

SALTOS DIVERGENTES (BRANCH DIVERSION)

- PreVolta: todas las sentencias de un camino se ejecutan antes de cualquier sentencia del otro camino de ejecución.

```
Hacer X;
if (threadIdx.x < 4) {
    A();
    B();
} else {
    P();
    R();
}
Hacer Z;
```



- Volta: permiten la *interleaved execution* de las sentencias de los distintos caminos de ejecución.

COALESCENCIA

Sea v un vector de tamaño n . Resolver $\text{suma} = \sum_{i=0}^{n-1} v[i]$

- Secuencial en CPU:

```
...
    suma=0.0;
    for (i=0; i<n; i++)
        suma += v[i];
...
...
```

- Paralelo OpenMP CPU:

```
...
    suma=0.0;
#pragma omp parallel for reduction(+: suma) num_threads(ncores)
    for (i=0; i<n; i++)
        suma += v[i];
...
...
```

- Explota localidad espacial

COALESCENCIA

Sea v un vector de tamaño n . Resolver $\text{suma} = \sum_{i=0}^{n-1} v[i]$

- Paralelo OpenMP CPU “*handmade localidad espacial CPU*”:

```
...
    ncores = omp_get_num_procs();
    for (i=0; i<ncores; i++) { tmpVec[i]=0.0; }
    #pragma omp parallel num_threads(ncores) private(hilo)
    {
        hilo=omp_get_thread_num();
        for (i=0; i<(n/ncores); i++)
            tmpVec[hilo] += v[hilo*(n/ncores)+i];
    }
    suma=0.0;
    for (i=0; i<ncores; i++) { suma += tmpVec[i]; }
...
}
```

- ¿Mejor, peor o igual que el anterior?

COALESCENCIA

Sea v un vector de tamaño n . Resolver $\text{suma} = \sum_{i=0}^{n-1} v[i]$

- Paralelo OpenMP CPU “*handmade rompiendo localidad espacial CPU*”

```
...
    ncores = omp_get_num_procs();
    for (i=0; i<ncores; i++) { tmpVec[i]=0.0; }
    #pragma omp parallel num_threads(ncores) private(hilo)
    {
        hilo=omp_get_thread_num();
        for (i=0; i<n; i+=ncores)
            tmpVec[hilo] += Vector[hilo+i];
    }
    suma=0.0;
    for (i=0; i<ncores; i++) { suma += tmpVec[i]; }
...
}
```

- ¿Mejor, peor o igual que los anteriores?

COALESCENCIA

Sea v un vector de tamaño n . Resolver $\text{suma} = \sum_{i=0}^{n-1} v[i]$

- 16 cores CPU:

<code>./Coalescente (n=2²³= 8388608)</code>	
Tiempo secuencial	1.6161119E-02
Tiempo paralelo reduction	4.1111679E-02
Tiempo paralelo hand made	2.7210464E-02
Tiempo paralelo bad	3.0115871E-02
<code>./Coalescente (n=2²⁴=16777216)</code>	
Tiempo secuencial	3.2298271E-02
Tiempo paralelo reduction	3.6256226E-02
Tiempo paralelo hand made	2.6870016E-02
Tiempo paralelo bad	4.4180351E-02
<code>./Coalescente (n=2²⁵=33554432)</code>	
Tiempo secuencial	4.6381504E-02
Tiempo paralelo reduction	3.0885887E-02
Tiempo paralelo hand made	2.8962816E-02
Tiempo paralelo bad	7.5936546E-02

COALESCENCIA

Sea v un vector de tamaño n . Resolver $\text{suma} = \sum_{i=0}^{n-1} v[i]$

- CUDA “*localidad espacial CPU*”:

```
localidad<<<1, 32>>>(v, n);
...
__global__ void localidad(double *Vector, const int n) {
    __shared__ double parciales[32];
    int   i, chunk=n/32;
    double suma=0.0;
    for (i=0; i<chunk; i++)
        suma += Vector[threadIdx.x*chunk+i];
    parciales[threadIdx.x]=suma;  suma=0.0;
    __syncthreads();
    if (threadIdx.x == 0) {
        for (i=0; i<32; i++) {suma += parciales[i]; }
        Vector[0] = suma;
    }
}
```

COALESCENCIA

Sea v un vector de tamaño n . Resolver $\text{suma} = \sum_{i=0}^{n-1} v[i]$

- CUDA “*rompiendo localidad espacial CPU*”:

```
sin_localidad<<<1, 32>>>(v, n);
...
__global__ void sin_localidad(double *Vector, const int n) {
    __shared__ double parciales[32];
    int   i;
    double suma=0.0;
    for (i=0; i<n; i+=32)
        suma += Vector[threadIdx.x+i];
    parciales[threadIdx.x]=suma;  suma=0.0;
    __syncthreads();
    if (threadIdx.x == 0) {
        for (i=0; i<32; i++) {suma += parciales[i]; }
        Vector[0] = suma;
    }
}
```

COALESCENCIA

Sea v un vector de tamaño n . Resolver suma = $\sum_{i=0}^{n-1} v[i]$

- 16 cores CPU y kernels GPU con 1 bloque de 32 hilos:

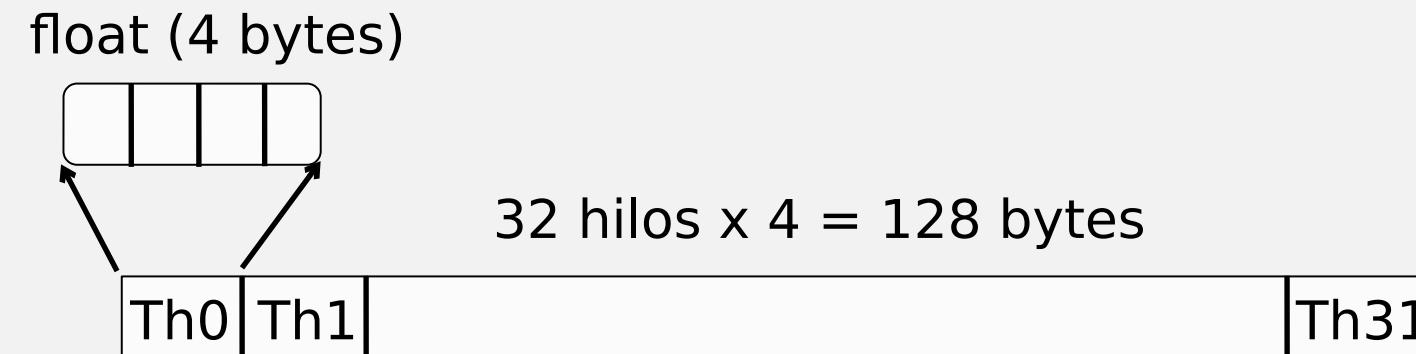
./Coalescente ($n=2^{23}= 8388608$)	
Tiempo secuencial	1.6161119E-02
Tiempo paralelo reduction	4.1111679E-02
Tiempo paralelo hand made	2.7210464E-02
Tiempo paralelo bad	3.0115871E-02
Tiempo GPU localidad	3.7153248E-02
Tiempo GPU sin_localidad	1.4544736E-02
./Coalescente ($n=2^{25}=33554432$)	
Tiempo secuencial	4.6381504E-02
Tiempo paralelo reduction	3.0885887E-02
Tiempo paralelo hand made	2.8962816E-02
Tiempo paralelo bad	7.5936546E-02
Tiempo GPU localidad	1.4267780E-01
Tiempo GPU sin_localidad	5.0520161E-02

COALESCENCIA

- *Localidad Espacial* CPU: Orientada a reducir el número de accesos a memoria central a nivel de hilo.
- El enfoque de la localidad espacial en CPU no funciona bien en GPU, como se ha visto en el ejemplo anterior.
- Coalescencia es la *Localidad Espacial* GPU: Orientada a reducir el número de accesos a nivel de warp.
- Cuando un *warp* accede a memoria global junta (**Coalescencia**) los accesos de los hilos dentro del *warp*.
- Una operación de acceso a memoria global puede tener que *repetirse* múltiples veces dependiendo de la distribución de los datos (distribución de los accesos a memoria) entre los hilos del *warp*.
- El acceso a la memoria global se realiza en transacciones de 32, 64 ó 128 bytes.
- Los segmentos de memoria que estén alineados (cuya primera dirección sea múltiplo del tamaño del dispositivo) pueden ser leídos o escritos por transacciones de memoria.
- Lo anterior afecta al rendimiento, más o menos, en función del tipo de memoria al que se esté accediendo.
- A mayor dispersión de las direcciones mayor pérdida de rendimiento.
- Se debe maximizar la coalescencia para un mayor rendimiento en los accesos a memoria global.

COALESCENCIA

- Un acceso se traduce en una sola instrucción de acceso a memoria global si y solo si el tamaño del tipo es 1, 2, 4, 8 o 16 bytes y los datos están alineados de forma natural.
- Los requisitos de alineación se cumplen automáticamente para los tipos incorporados (*char, short, int, long, float, doble, etc.*).
- Para que estos accesos sean totalmente coalescentes la anchura del bloque de hilos y de la estructura de datos deben ser múltiplos de *warp*.

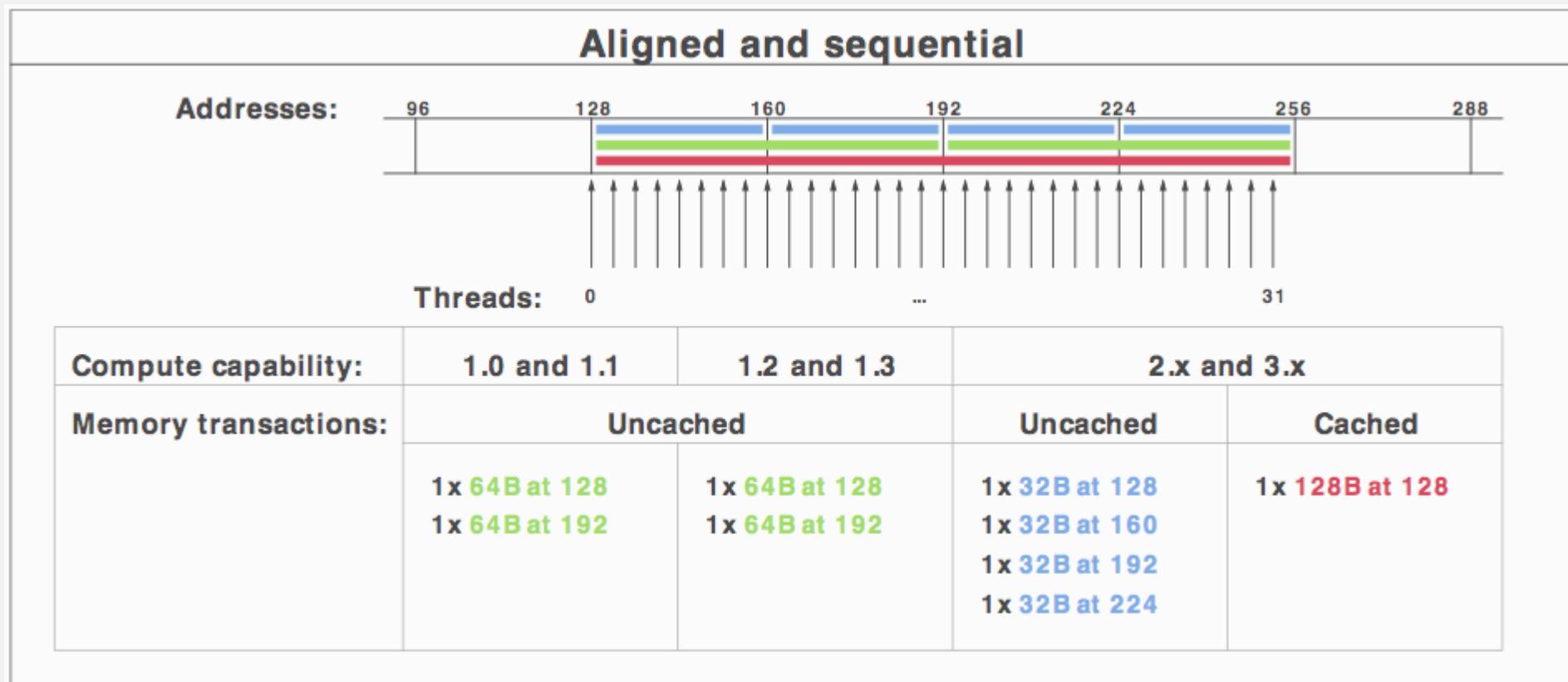


- Un patrón de acceso a memoria *global* para una matriz 2D *adecuado* es aquel donde un hilo genérico (tx, ty) usa $BaseAddress + width * ty + tx$ para acceder a su elemento, siendo *width* la anchura de la matriz.

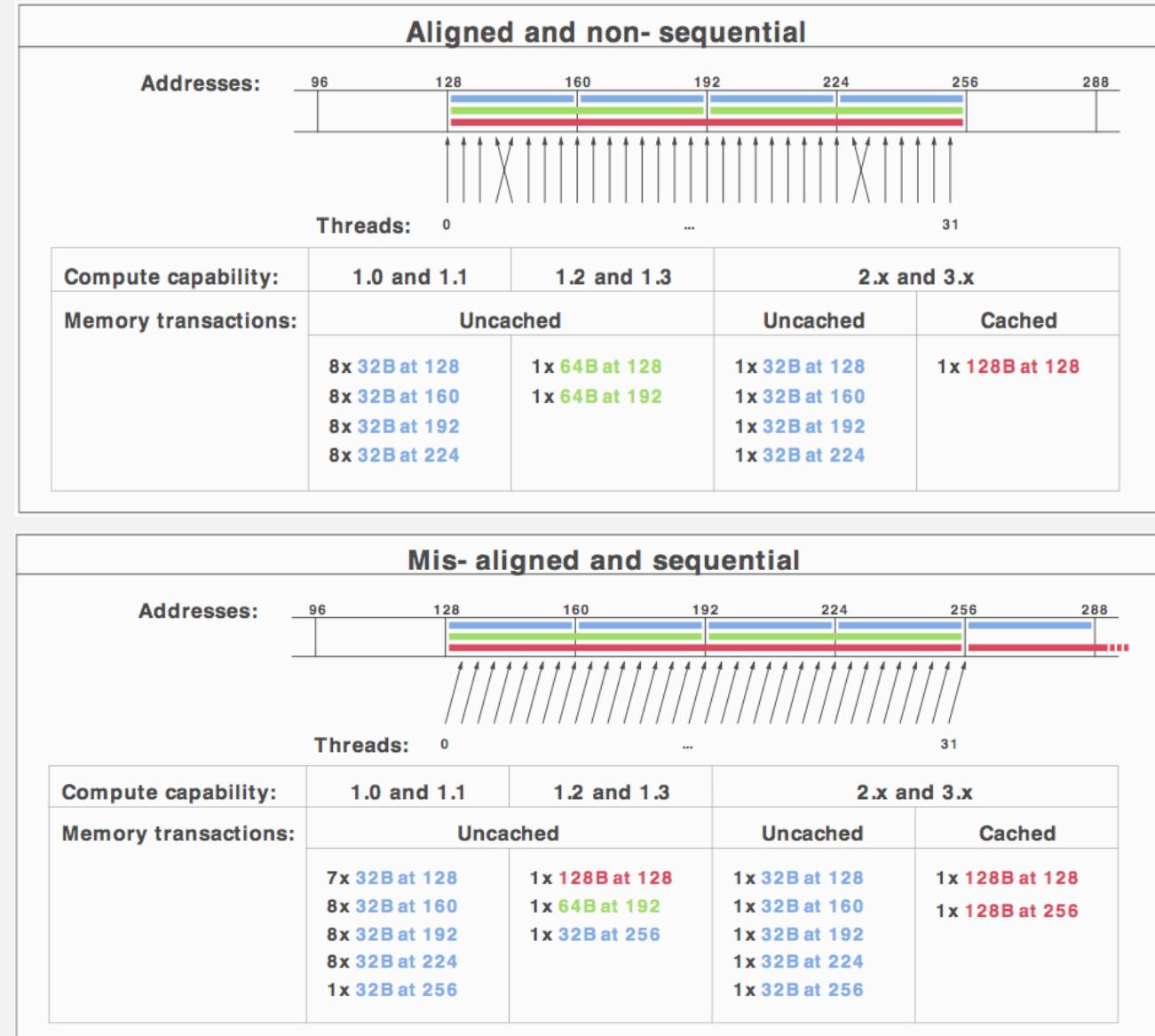
COALESCENCIA

Guía para maximizar la productividad en el acceso a memoria:

- Usar los patrones de acceso óptimos para cada arquitectura (CUDA Compute Capability).
- Usar tipos de datos que cumplan con el requisito de alineación a 32, 64 ó 128 bytes.
- *Padding, shared memory* y la existencia de la caché ayudan a mitigar el problema de la coalescencia.



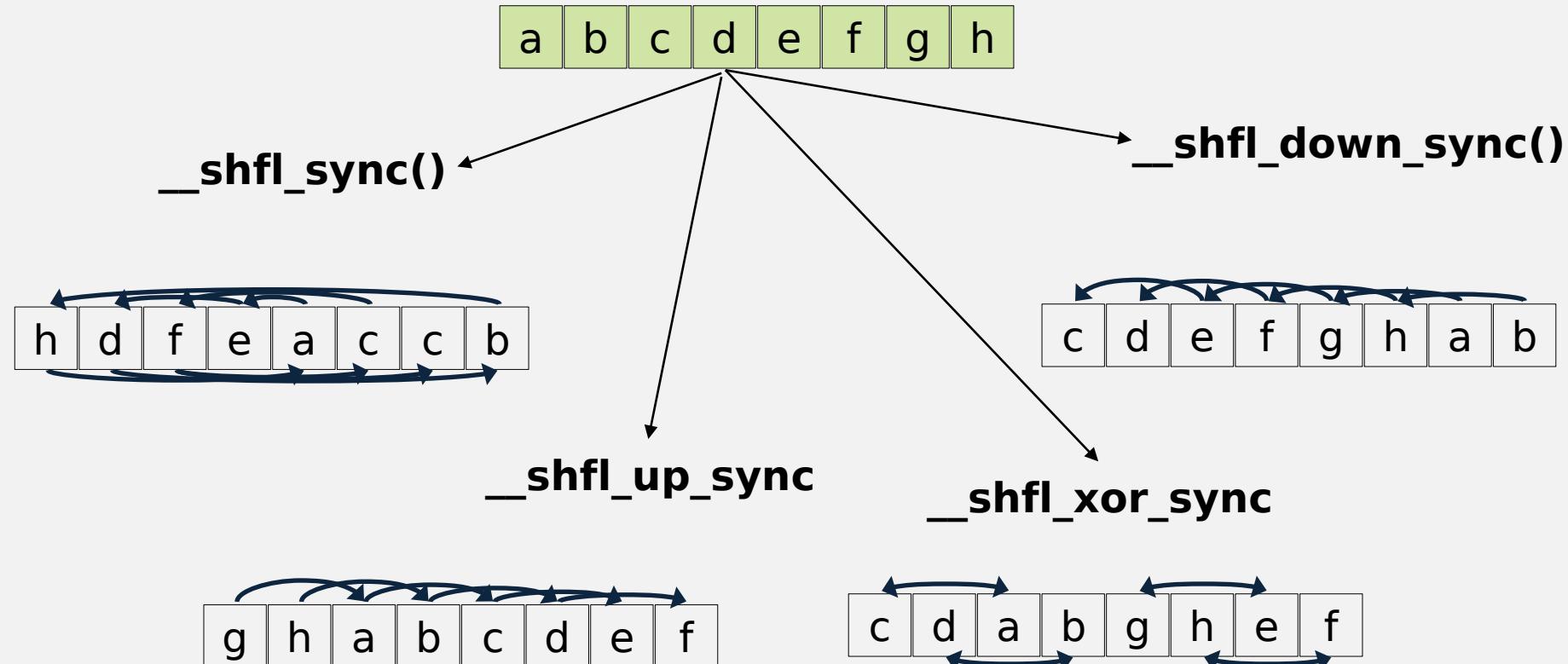
COALESCENCIA



WARP-LEVEL PRIMITIVES

Para gestionar la interacción entre los hilos de un warp a *bajo nivel*.

- Tres tipos: intercambio, máscaras (`_activemask`) y sincronización (`_syncwarp`)
- Evita el uso de *shared memory* y pueden mejorar el rendimiento.

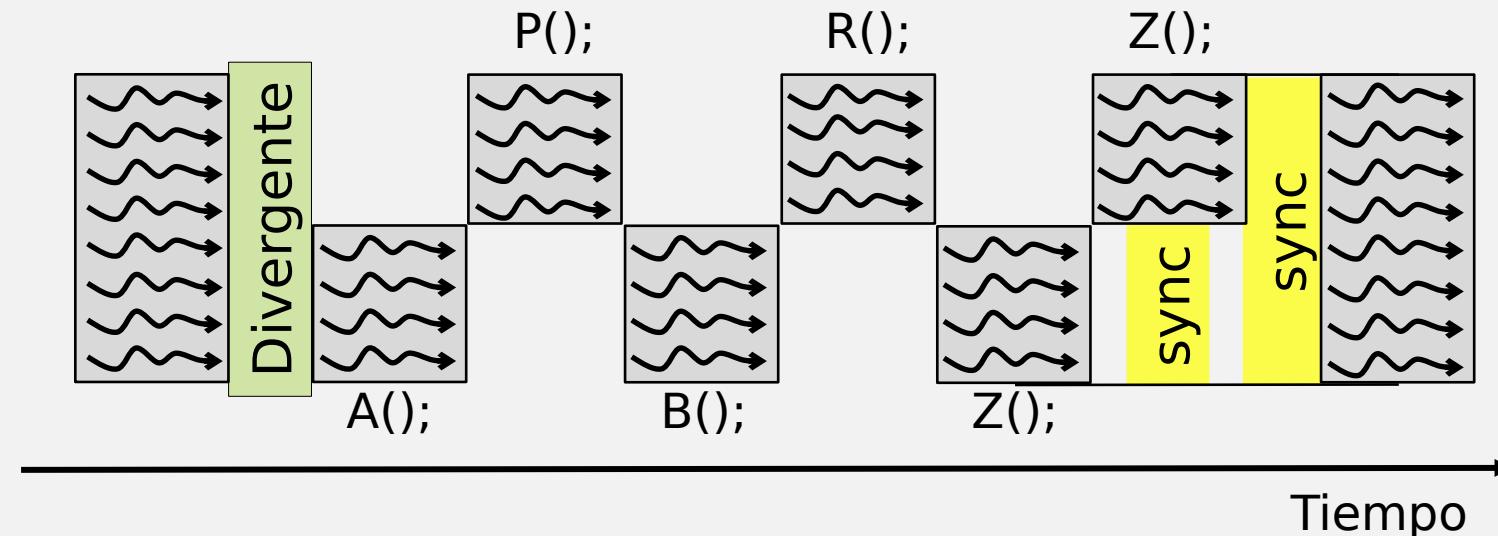


WARP-LEVEL PRIMITIVES

El caso `_syncwarp()`

- En Volta no es necesario para otras *warp-level primitives*, pero puede serlo para otro tipo de operaciones.

```
X();  
if (threadIdx.x < 4) { Hacer A(); Hacer B(); }  
else { Hacer P(); Hacer R(); }  
Z();  
_syncwarp();
```



WARP-LEVEL PRIMITIVES

El caso `_shfl_down_sync()`

```
int __shfl_down_sync(unsigned mask, int v, unsigned s, int width=warpSize)
```

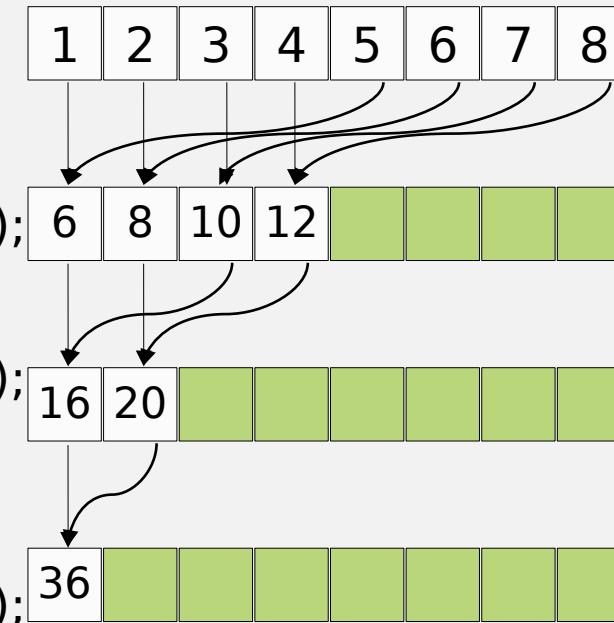
- **mask** Mascara de 32-bit que denota los hilos del warp que participan en la operación. Lo primero que hace la primitiva es sincronizar los hilos que participan (si no están ya sincronizados).

- `unsigned m=0xffffffff`

```
v += __shfl_down_sync(m, v, 4);
```

```
v += __shfl_down_sync(m, v, 2);
```

```
v += __shfl_down_sync(m, v, 1);
```



WARP-LEVEL PRIMITIVES

Para ajustar la máscara si el número de hilos es menor que el tamaño del warp

```
#define MASK 0xffffffff  
__inline__ __device__ double ReduceWarpFull(double val, const  
unsigned int N) {  
    unsigned mask = __ballot_sync(MASK, threadIdx.x < N);  
    if (threadIdx.x < N)  
        for (unsigned int offset = 16; offset > 0; offset /= 2)  
            val += __shfl_down_sync(mask, val, offset);  
}
```

En Volta (y posteriores) funcionan con saltos divergentes

```
if (threadIdx.x % 2) {  
    ....  
    val += __shfl_sync(MASK, val, 0);  
} else {  
    val += __shfl_sync(MASK, val, 0);  
    ...
```

WARP-LEVEL PRIMITIVES

Resolver

- CUDA “coalescente”:

```
__global__ void sin_localidad(double *Vector, const int n) {  
    __shared__ double parciales[32];  
    double suma = 0.0;  
    for (unsigned int i=0; i<n; i+=32)  
        suma += Vector[threadIdx.x+i];  
    parciales[threadIdx.x]=suma;  suma=0.0;  
    __syncthreads();  
  
    if (threadIdx.x == 0) {  
        for (unsigned int i=0; i<32; i++)  {summa += parciales[i]; }  
        Vector[0] = summa;  
    }  
}
```

WARP-LEVEL PRIMITIVES

Resolver

```
#define MASK 0xffffffff
__inline__ __device__ double ReduceWarp(double val) {
    for (unsigned int offset = 16; offset > 0; offset /= 2)
        val += __shfl_down_sync(MASK, val, offset, 32);
    return val;
}

__global__ void warp_prim(double *V, const unsigned int N)
{
    double suma=0.0;
    for (unsigned int i=0; i<N; i+=32)
        suma += V[threadIdx.x+i];
    suma=ReduceWarp(suma);
    if (threadIdx.x == 0)    Vector[0] = suma;
}
```

WARP-LEVEL PRIMITIVES

Resolver

- <<<1, 32>>>(Vector, n);

./Coalescente (n=2 ²³ =8388608)	
Tiempo GPU localidad	2.9422655E-02
Tiempo GPU sin_localidad	1.4593856E-02
Tiempo GPU warp_prim	1.4574624E-02
./Coalescente (n=2 ²⁴ =16777216)	
Tiempo GPU localidad	5.8586655E-02
Tiempo GPU sin_localidad	2.9044737E-02
Tiempo GPU warp_prim	2.9019840E-02
./Coalescente (n=2 ²⁵ =33554432)	
Tiempo GPU localidad	1.1697629E-01
Tiempo GPU sin_localidad	4.7669121E-02
Tiempo GPU warp_prim	4.1792000E-02

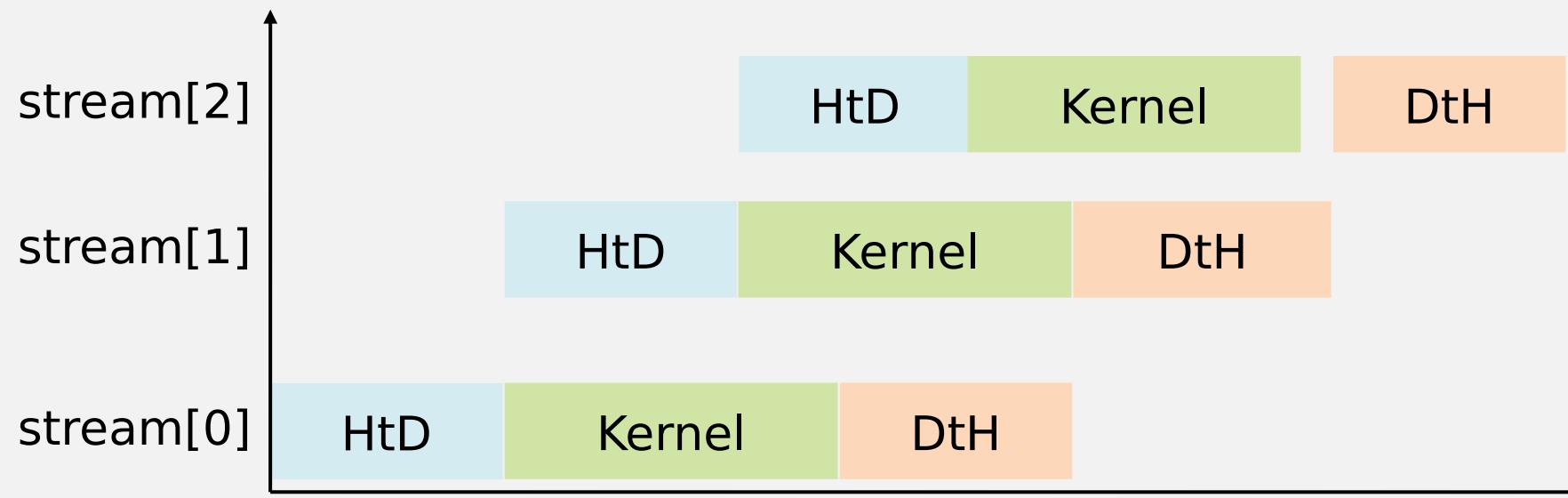
STREAMS

- Un *Stream* es una secuencia de comandos que se ejecutan en orden (equivalencia: cola FIFO).
- Si no se especifica los comandos van al *stream por defecto*, que siempre existe.
- Los *streams* se ejecutan concurrentemente entre sí.
- Cada hilo CPU puede tener su “*stream por defecto*” independiente (con “nvcc --default-stream=per-thread”).

Ejemplo de uso

```
cudaStream_t stream1, stream2;  
cudaStreamCreate(&stream1);  
cudaStreamCreate(&stream2);  
...  
cudaMalloc (&dev1, size);  
cudaMallocHost (&host1, size); //tiene que ser pinned  
...  
cudaMemcpyAsync(dev1, host1, size, dir, stream1);  
kernel2<<<grid, block, 0, stream2>>>(..., dev2, ...);  
...  
alguna_función_cpu()
```

STREAMS



- `cudaDeviceSynchronize()` Espera a que todos los comandos anteriores en todas las secuencias de todos los hilos de *Host* estén completados.
- `cudaStreamSynchronize()` Espera hasta que todos los comandos anteriores en la secuencia dada como parámetro estén completados.
- `cudaStreamWaitEvent()` Partiendo de un *stream* y un evento hace que todo el flujo de ese *stream* pare su ejecución hasta que el evento dado se haya completado.
- `cudaStreamQuery()` Permite saber si todos los comandos anteriores de un *stream* han finalizado.

STREAMS

Plantilla

// Creando

```
cudaStream_t *streams=(cudaStream_t*)malloc(nstr*sizeof(cudaStream_t));           //nstr = número de streams  
for (int i=0; i<nstr; i++)  
    cudaStreamCreate(&(streams[i]));
```

// Lanzamiento asíncrono de los *kernel*s cada uno con sus propios datos

```
for (int i=0; i<nstr; i++)  
    kernel<<<blocks, threads, 0, streams[i]>>>(dA + (i*n/nstr), dB, niters);
```

// Lanzamiento asíncrono de las copias. No empiezan hasta el fin de su *kernel*

```
for (int i=0; i<nstr; i++)  
    cudaMemcpyAsync(hA + (i*n/nstr), dA + (i*n/nstr), size/nstr, cudaMemcpyDeviceToHost, streams[i]);
```

// Liberando recursos

```
for (int i=0; i<nstr; i++)  
    cudaStreamDestroy(streams[i]);
```

STREAMS

$C = \beta C + \alpha AB$; usando el *modo estándar*

```
NBlk.x = NBlk.y = (int)ceil((float)n/32);
TpBlk.x = TpBlk.y = 32; NBlk.z = TpBlk.z = 1;

cudaEventRecord(start, 0);
    cudaMemcpy(devA, cpuA, n*n*sizeof(double), cudaMemcpyHostToDevice));
    cudaMemcpy(devB, cpuB, n*n*sizeof(double), cudaMemcpyHostToDevice));
    cudaMemcpy(devC, cpuC, n*n*sizeof(double), cudaMemcpyHostToDevice));

    MatMult<<<NBlk, TpBlk>>>(devC, devA, devB, alfa, beta, n, n);

    cudaMemcpy(cpuC, devC, n*n*sizeof(double), cudaMemcpyDeviceToHost));
    CHECKLASTERR();
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&time, start, stop);
```

STREAMS

C=bC+aAB; usando 1 hilo CPU

```
chunk=n/nstreams; stride=chunk*n;
NBlk.x=(int)ceil((float) n/32);
NBlk.y=(int)ceil((float)chunk/32);
NBlk.z=TpBlk.z=1; TpBlk.x=TpBlk.y=32;

cudaEventRecord(start, 0);
cudaMemcpy(devB, cpuB, n*n*sizeof(double), cudaMemcpyHostToDevice);
for(i=0; i<nstreams; i++) {
    cudaMemcpyAsync(&devA[stride*i], &cpuA[stride*i], stride*sizeof(double), cudaMemcpyHostToDevice, streams[i]);
    cudaMemcpyAsync(&devC[stride*i], &cpuC[stride*i], stride*sizeof(double), cudaMemcpyHostToDevice, streams[i]);

    MatMult<<<NBlk, TpBlk, 0, streams[i]>>>(&devC[stride*i], &devA[stride*i], devB, alfa, beta, chunk, n);

    cudaMemcpyAsync(&cpuC[stride*i], &devC[stride*i], stride*sizeof(double), cudaMemcpyDeviceToHost, streams[i]);
}
CHECKLASTERR();
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
```

STREAMS

C=bC+aAB; usando tantos hilos CPU como streams

```
chunk=n/nstreams; stride=chunk*n;
NBlk.x=(int)ceil((float)n/32);
NBlk.y = (int)ceil((float)chunk/32);
NBlk.z=TpBlk.z=1; TpBlk.x=TpBlk.y=32;

cudaEventRecord(start, 0);
cudaMemcpy(...);
#pragma omp parallel for num_threads(nstreams)
for(i=0; i<nstreams; i++) {
    cudaMemcpyAsync(...);
    cudaMemcpyAsync(...);
    MatMult<<<NBlk, TpBlk, 0, streams[i]>>>(...);
    cudaMemcpyAsync(...);
    cudaStreamSynchronize(streams[i]);
}
CHECKLASTERR();
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&time, start, stop);
```

DINAMISMO

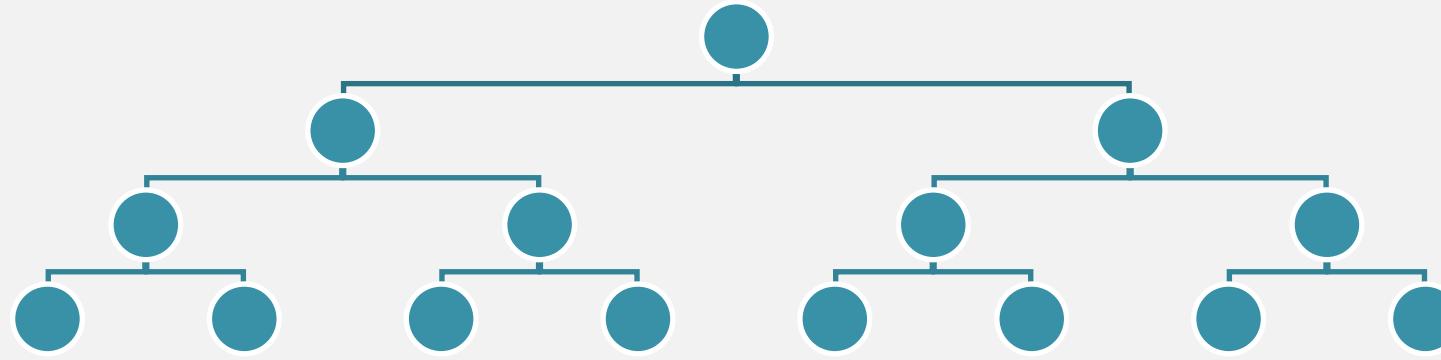
Los kernels puede lanzar, a su vez, la ejecución de otros kernels.

Tomado de una aplicación propia

```
_global_ void kernel_doGPU(...) {  
    ...  
    kernel_Cfreq<<<NMIDI, sizeWarp>>>(...);  
    if (BETA == 0.0)  
        kernel_CompDisB0<<<Grid2D, TPBI2D>>>(...);  
    else if (BETA == 1.0) {  
        kernel_Reduction<<<1, sizeWarp>>>(...);  
        kernel_CompDisB1<<<Grid2D, TPBI2D>>>(...);  
    } else {  
        kernel_PowToReal<<<GridNMID32, sizeWarp>>>(...);  
        kernel_CompDisBG<<<Grid2D, TPBI2D>>>(...);  
    }  
}
```

REDUCCIÓN PARALELA EN CUDA

- Muy común. Importante que sea eficiente.
- Enfoque basado en divide y vencerás a nivel de bloque de hilo.



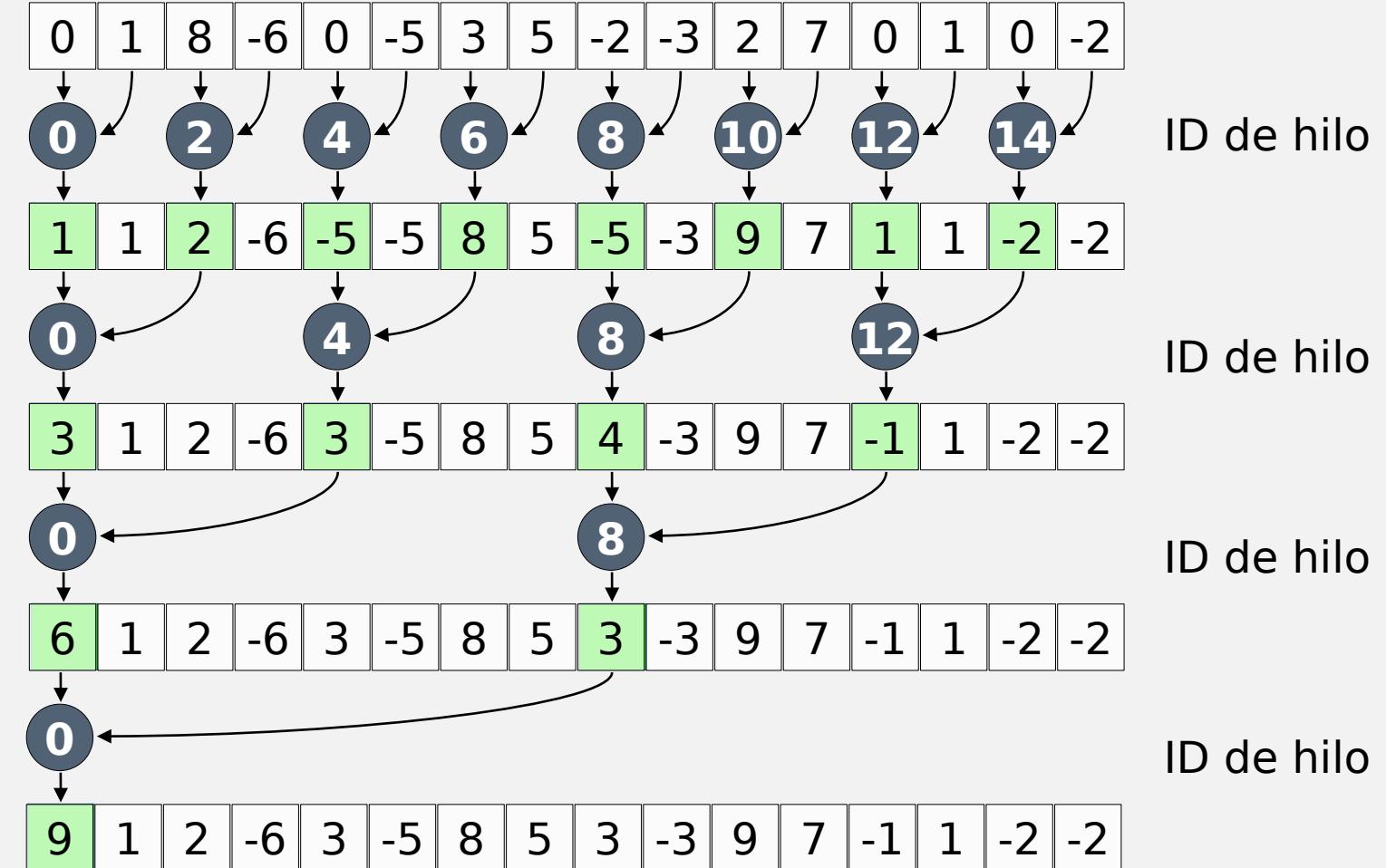
- Importante poder lanzar múltiples bloques:
 - Para procesar estructuras de datos muy grandes.
 - Para mantener ocupados todos los SMs de la GPU.
 - Cada bloque reduce una parte de la estructura.
- La GPU usada en el siguiente ejemplo tiene una tasa de transferencia de memoria teórica de 86.4 GB/s

REDUCCIÓN PARALELA EN CUDA

- Reducción Nº 1:

Valores en compartida

1 *Stride 1*



REDUCCIÓN PARALELA EN CUDA

- Reducción Nº 1:

```
__global__ void reduce1(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

    // Cada hilo carga en memoria compartida un elemento de la
    memoria global
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();

    // Haciendo la reducción en compartida
    for(unsigned int s=1; s < blockDim.x; s *= 2) {
        if (tid % (2*s) == 0)
            sdata[tid] += sdata[tid + s];
        __syncthreads();
    }
    // Escribiendo resultado a la memoria global
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

REDUCCIÓN PARALELA EN CUDA

- Reducción Nº 1:

```
__global__ void reduce1(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

    // Cada hilo carga en memoria compartida un elemento de la memoria global
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();

    // Haciendo la reducción en compartida
    for(unsigned int s=1; s < blockDim.x; s *= 2) {
        if (tid % (2*s) == 0) → Saltos divergentes, bajo rendimiento
            sdata[tid] += sdata[tid + s];
        __syncthreads();
    }
    ...
}
```

Rendimiento: 2.083 GB/s

REDUCCIÓN PARALELA EN CUDA

- Reducción Nº 2: stride como índice – sin divergencia

```
__global__ void reduce2(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

    // Cada hilo carga en memoria compartida un elemento de la memoria global
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();

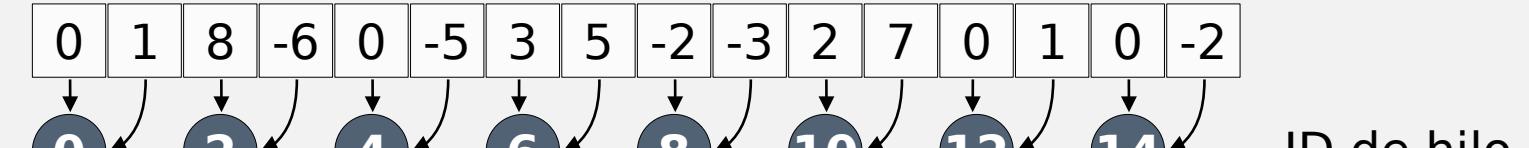
    // Haciendo la reducción en compartida
    for(unsigned int s=1; s < blockDim.x; s *= 2) {
        int index = 2 * s * tid;
        if (index < blockDim.x) sdata[index] += sdata[index + s];
        __syncthreads();
    }
    // Escribiendo resultado a la memoria global
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

REDUCCIÓN PARALELA EN CUDA

- Reducción Nº 1: (repetido aquí para poder comparar fácilmente)

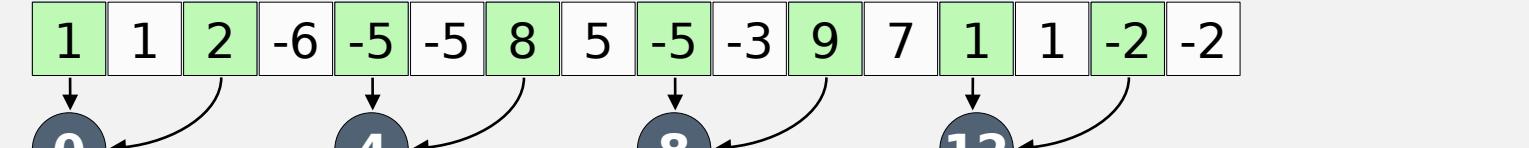
Valores en compartida

1 *Stride 1*



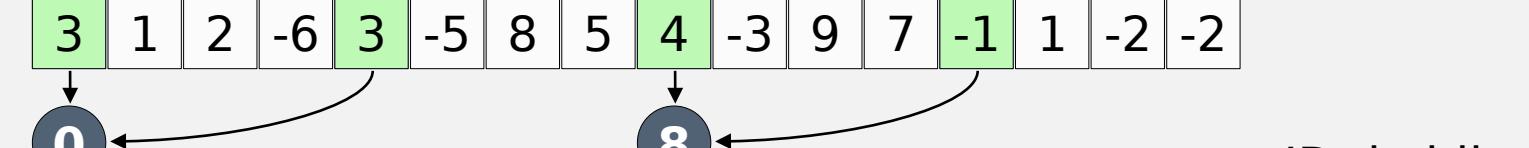
Valores en compartida

2 *Stride 2*



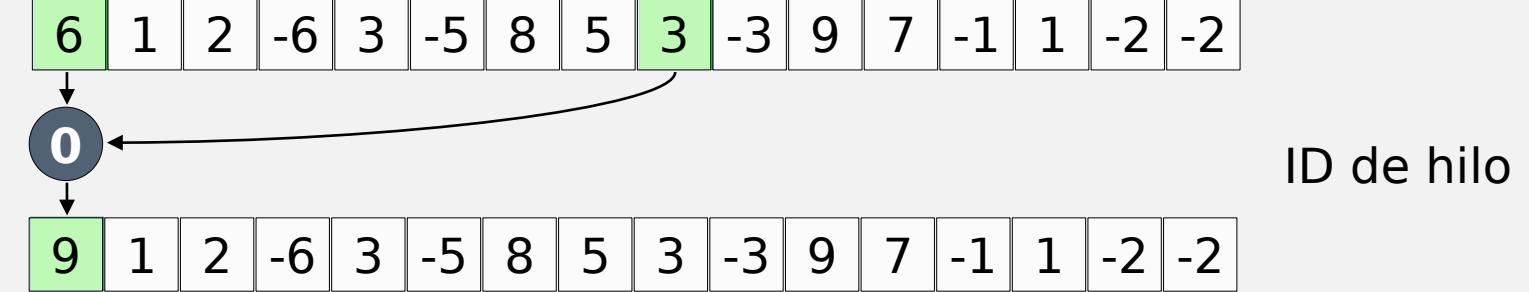
Valores en compartida

3 *Stride 4*



Valores en compartida

4 *Stride 8*

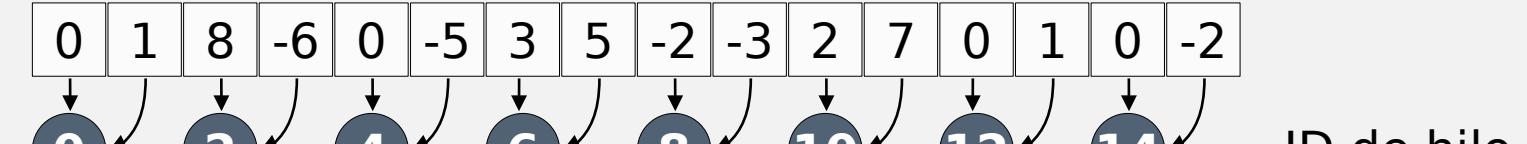


REDUCCIÓN PARALELA EN CUDA

Reducción Nº 2: stride como índice – sin divergencia

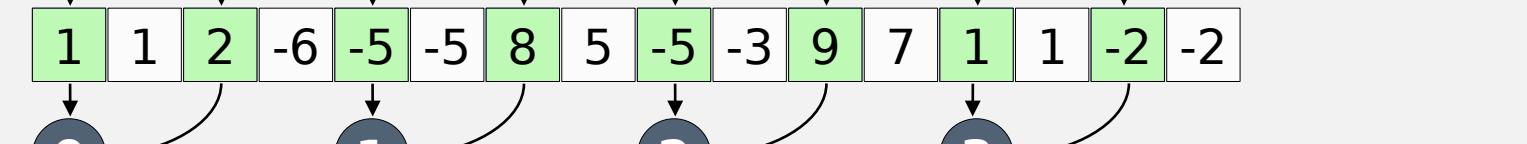
Valores en compartida

1 *Stride 1*



Valores en compartida

2 *Stride 2*



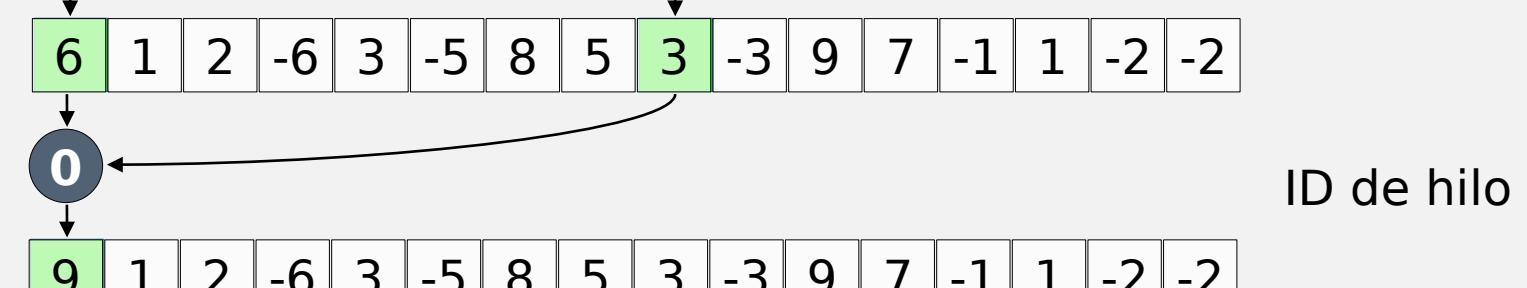
Valores en compartida

3 *Stride 4*



Valores en compartida

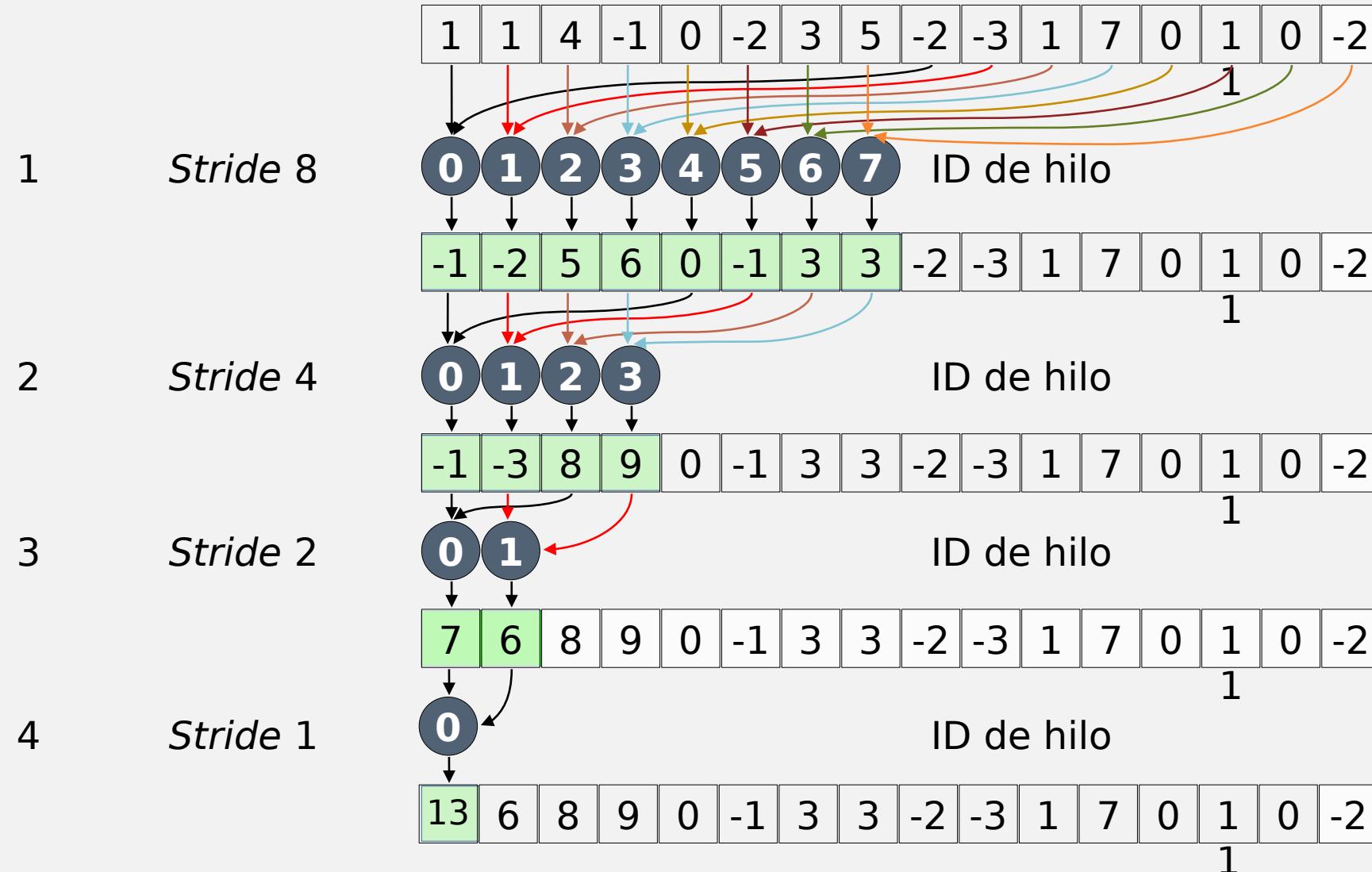
4 *Stride 8*



Rendimiento: 4.854 GB/s. El problema: Conflicto de acceso a bancos de memoria compartida

REDUCCIÓN PARALELA EN CUDA

- Reducción Nº 3: direccionamiento secuencial



REDUCCIÓN PARALELA EN CUDA

- Reducción Nº 3: direccionamiento secuencial

```
__global__ void reduce3(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

    // Cada hilo carga en memoria compartida un elemento de la memoria global
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();

    // Haciendo la reducción en compartida
    for(unsigned int blockDim.x/2; s>0; s>>=1) {
        if (tid < s)
            sdata[tid] += sdata[tid + s];
        __syncthreads();
    }
    // Escribiendo resultado a la memoria global
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

REDUCCIÓN PARALELA EN CUDA

- Reducción Nº 3: direccionamiento secuencial

```
__global__ void reduce3(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

    // Cada hilo carga en memoria compartida un elemento de la memoria global
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();

    // Haciendo la reducción en compartida
    for(unsigned int blockDim.x/2; s>0; s>>=1) { //La mitad de los hilos idle
        if (tid < s)
            sdata[tid] += sdata[tid + s];
        __syncthreads();
    }
    ...
}
```

Rendimiento: 9.741 GB/s

REDUCCIÓN PARALELA EN CUDA

- Reducción Nº 4: suma durante la carga

```
__global__ void reduce4(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

    // Cada hilo carga en memoria compartida un elemento de la memoria global
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
    sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
    __syncthreads();
}

...
```

Rendimiento: 17.377 GB/s

REDUCCIÓN PARALELA EN CUDA

- A medida que avanza la reducción el número de hilos activos disminuye.
 - Cuando $s \leq 32$ solo queda un *warp* activo.
- Las instrucciones son SIMD sincrónicas dentro del warp.
- Eso significa que cuando $s \leq 32$:
 - No es necesario `_syncthreads()`
 - No es necesario "*if (tid < s)*"
- Se puede, por tanto, recurrir al desenrollado de las últimas iteraciones del bucle interno.

REDUCCIÓN PARALELA EN CUDA

- Reducción Nº 5: desenrollado del último warp

```
__global__ void reduce5(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];
    ...

    // Haciendo la reducción en compartida
    for(unsigned int blockDim.x/2; s>>=1) {
        if (tid < s)
            sdata[tid] += sdata[tid + s];
        __syncthreads();
    }
    if (tid < 32) {
        sdata[tid] += sdata[tid + 32];  sdata[tid] += sdata[tid + 16];
        sdata[tid] += sdata[tid +  8];  sdata[tid] += sdata[tid +  4];
        sdata[tid] += sdata[tid +  2];  sdata[tid] += sdata[tid +  1];
    }
    ...
}
```

Rendimiento: 17.377 GB/s

REDUCCIÓN PARALELA EN CUDA

- Reducción Nº 6: múltiples sumas por carga

```
__global__ void reduce6(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
    unsigned int gridSize = blockSize*2*gridDim.x;
    sdata[tid] = 0;

    while (i < n) {
        sdata[tid] += g_idata[i] + g_idata[i+blockSize];
        i += gridSize;                                // stride para mantener coalescencia
    }
    __syncthreads();
}
```

...

Rendimiento: 17.377 GB/s

REDUCCIÓN PARALELA EN CUDA

- Reducción Final:

```
__global__ void reduce(int *g_idata, int *g_odata, unsigned int n) {  
    extern __shared__ int sdata[];  
  
    unsigned int tid = threadIdx.x;  
    unsigned int i = blockIdx.x*blockDim.x*2 + tid;  
    unsigned int gridSize = blockSize*2*gridDim.x;  
    sdata[tid] = 0;  
  
    while (i < n) {  
        sdata[tid] += g_idata[i] + g_idata[i+blockSize];  
        i += gridSize;  
    }  
    __syncthreads();  
  
    ...
```

REDUCCIÓN PARALELA EN CUDA

- Reducción Final:

```
...
    if (blockSize >= 512) {
        if (tid < 256) sdata[tid] += sdata[tid + 256];
        __syncthreads();
    }

    if (blockSize >= 256) {
        if (tid < 128) sdata[tid] += sdata[tid + 128];
        __syncthreads();
    }

    if (blockSize >= 128) {
        if (tid < 64) sdata[tid] += sdata[tid + 64];
        __syncthreads();
    }

...
}
```

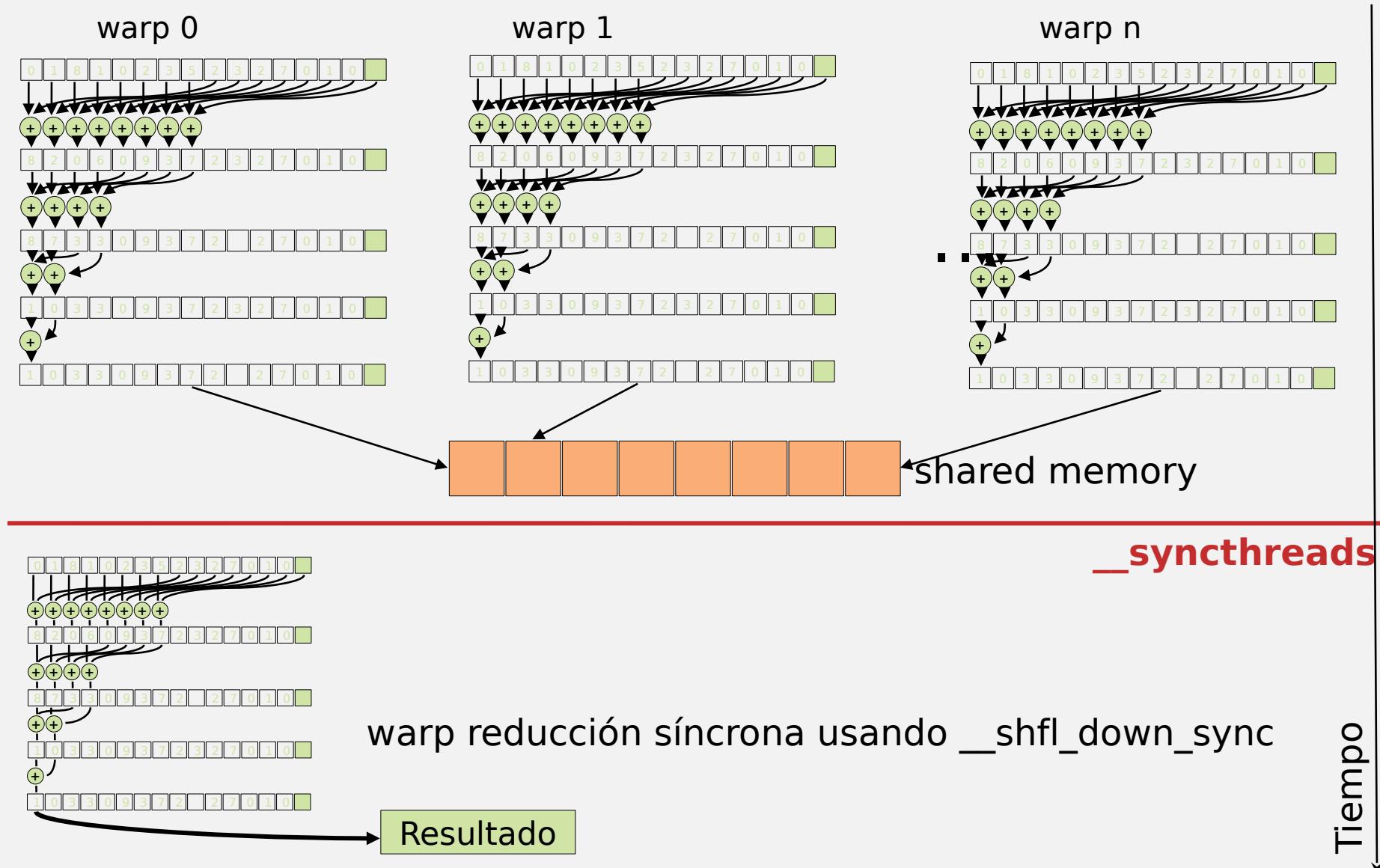
REDUCCIÓN PARALELA EN CUDA

- Reducción Final:

```
...
    if (tid < 32) {
        sdata[tid] += sdata[tid + 32];
        sdata[tid] += sdata[tid + 16];
        sdata[tid] += sdata[tid +  8];
        sdata[tid] += sdata[tid +  4];
        sdata[tid] += sdata[tid +  2];
        sdata[tid] += sdata[tid +  1];
    }

    // Escribiendo resultado a la memoria global
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

REDUCCIÓN PARALELA EN CUDA

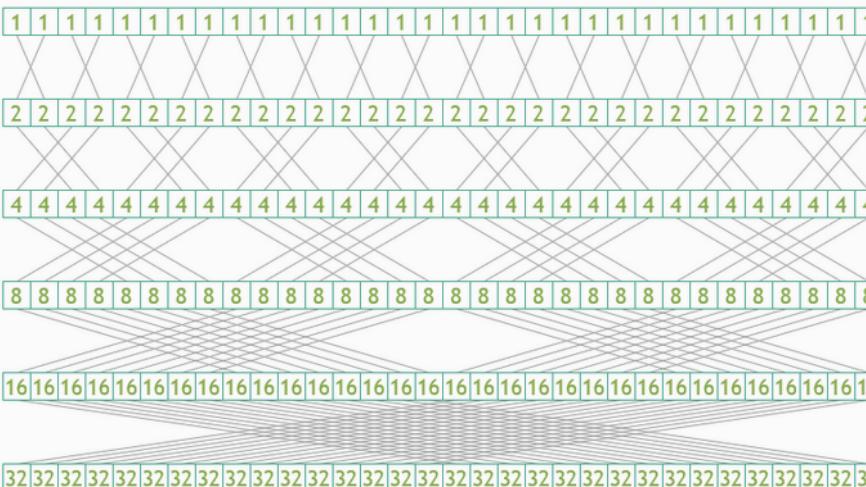


JUNTANDO CONCEPTOS: BUSCAR EL ÚLTIMO MÍNIMO

```
int Onelmin(double *odata, int *opos, double *idata, int maxGrid, int N) {  
    int nBlocks=0, nThreads=0, sShared=0, S;  
  
    BlocksAndThreads(&nBlocks, &nThreads, &sShared, maxGrid, N);  
  
    kernel_Onelmin<<<nBlocks, nThreads, 2*sShared>>>(odata, opos, idata, nThreads,  
    IsPow2(N), N);  
  
    S = nBlocks;  
    while (S > 1) {  
        BlocksAndThreads(&nBlocks, &nThreads, &sShared, maxGrid, S);  
        kernel_OnelminLast<<<nBlocks, nThreads, 2*sShared>>>(odata, opos, odata, opos,  
        nThreads, IsPow2(S), S);  
        S = (S + (nThreads*2-1)) / (nThreads*2);  
    }  
  
    cudaDeviceSynchronize();  
    return opos[0];  
}
```

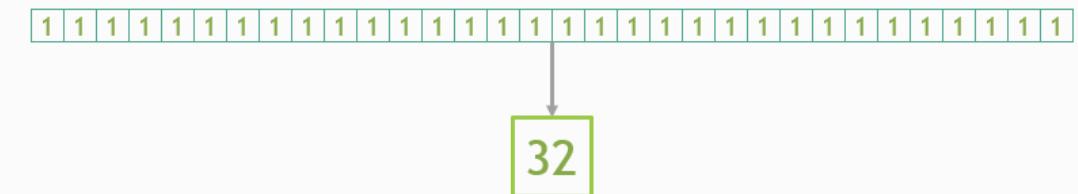
WARP-WIDE REDUCTION

- Usando la nueva instrucción de reducción de la A100 se puede realizar operaciones de reducción en un solo paso. Antes requería 5 pasos con operaciones SHFL como muestra la figura.



```
__device__ int reduce(int value) {
    value += __shfl_xor_sync(0xFFFFFFFF, value, 1);
    value += __shfl_xor_sync(0xFFFFFFFF, value, 2);
    value += __shfl_xor_sync(0xFFFFFFFF, value, 4);
    value += __shfl_xor_sync(0xFFFFFFFF, value, 8);
    value += __shfl_xor_sync(0xFFFFFFFF, value, 16);

    return value;
}
```



```
int total = __reduce_add_sync(0xFFFFFFFF, value);
```

CUDA TASK GRAPH

- La ejecución del trabajo en la GPU se divide en tres etapas: lanzamiento, inicialización del grid y ejecución del kernel. Para kernels con tiempos de ejecución reducidos los costes *generales* pueden ser una fracción significativa del tiempo total de ejecución.
- Muchas aplicaciones intensivas en GPU tienen una estructura iterativa en la que el mismo *trabajo* se ejecuta repetidamente. El uso de CUDA Streams requiere que la CPU vuelva a enviar el trabajo a la GPU con cada iteración lo que consume tiempo y recursos de la CPU.
- Los grafos de tareas CUDA proporcionan un modelo más eficiente para enviar trabajo a la GPU.
- Un grafo de tareas consta de una serie de operaciones, como copias de memoria y lanzamientos de kernel, conectadas por dependencias y se define por separado de su ejecución.
- Los grafos de tareas permiten definir una vez y ejecutar repetidamente flujos de trabajo.
- Un grafo de tareas permite el lanzamiento de cualquier número de kernels en una sola operación, mejorando enormemente la eficiencia y el rendimiento de la aplicación.
- Separar la definición del grafo de tareas de su ejecución reduce significativamente los costes en CPU del lanzamiento del kernel. Además, permiten que CUDA realice optimizaciones porque todo el flujo de trabajo es visible para el driver.

CONCLUSIONES

- “Cuando aprendemos a manejar un martillo, sólo vemos clavos”. No usar GPUs para menos de 10^7 datos.
- Primero buscar en internet. Segundo artesanía para adaptar código. Lema de Google Scholar:....
- Ojo a las dependencias: calcula el propietario. Si no, `_syncthreads` dentro de un bloque. Entre bloques no hay control. También tenemos `atomic`, que atomiza la celda, no toda la variable.
- Matlab: democratización de la programación matemática. CUDA: Quien quiera peces...
- Grano fino a la GPU. Grano grueso/tareas a la CPU.
- No es grave si los hilos tienen más carga: lo que importa es tener los cores al 100%.
- 70% programas son “compute bound”, mientras 30% son “memory bound” → ¡¡Usar memoria!!
- No hacerse el/la listx: la fuerza bruta es caballo ganador.
- Empezar lanzando los kernels con `<<<N/256,256>>>`, luego ir variando arriba/abajo (no menos de 64) en potencias de 2.

COMPUTE UNIFIED DEVICE ARCHITECTURE

...Y PYTHON



PYTHON: ENTRE DOS MUNDOS

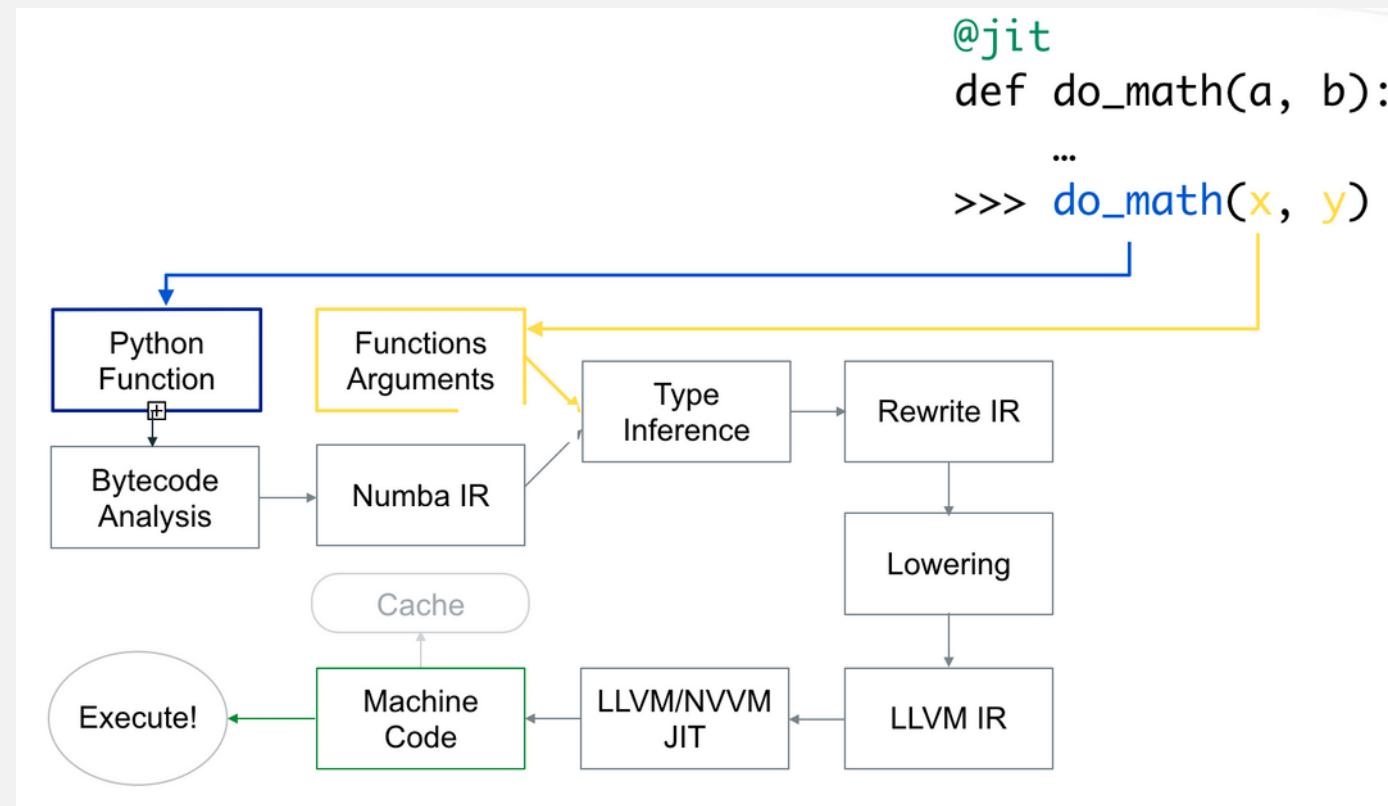
- Python es un lenguaje interpretado, EXTREMADAMENTE lento, pero muy flexible gracias a infinidad de librerías.
- Entre otras, NUMBA permite el uso y gestión de CPUs y GPUs, mediante “decoradores” y compiladores:

```
from numba import jit
import math
@jit
def hypot(x, y):
    x = abs(x);
    y = abs(y);
    t = min(x, y);
    x = max(x, y);
    t = t / x;
    return x * math.sqrt(1+t*t)
```

El `@jit` le dice a python que debe compilar para CPU esa función → Ganaremos velocidad.
Ojo: no cualquier código python puede ser compilado por NUMBA (diccionarios... mirar la [documentación](#)).

PYTHON: ENTRE DOS MUNDOS

- Esquema de compilación para CPU:



- (Imagen del DLI, Nvidia, 2022)

PYTHON: CAMINITO DE LA SINTAXIS CUDA

- La primera ejecución sobre GPU se puede realizar mediante vectorización:

```
@vectorize(['int64(int64, int64)'], target='cuda') # La firma del tipo de dato y el target son necesarios para las vectorizaciones
def add_ufunc(x, y):
    return x + y
a = np.array([1, 2, 3, 4])
b = np.array([10, 20, 30, 40])
add_ufunc(a, b)
```

La función “universal” `add_ufunc` suma dos elementos, pero su operación puede realizarse en paralelo, elemento a elemento.

Su implementación sobre C usaría un bucle *forall*.

Otros tipos de datos pueden ser `float32`, `complex128`... [y muchos más](#), que deberemos declarar al generar el array:

```
x = np.random.uniform(-3, 3, size=1000000).astype(np.float32)
```

PYTHON: CAMINITO DE LA SINTAXIS CUDA

- ¿Y si la función no tiene estructura vectorizable? Tenemos `@cuda`:

```
from numba import cuda
@cuda.jit(device=True [, inline=True])
def polar_to_cartesian(rho, theta):
    x = rho * math.cos(theta)                      # Función CUDA, no kernel CUDA. No vectorizable.
    y = rho * math.sin(theta)                      # No llamable desde función python estándar.
    return x, y                                     # Puede forzarse "inline"
                                                    #Puede devolver datos

@vectorize(['float32(float32, float32, float32, float32)'], target='cuda')
def polar_distance(rho1, theta1, rho2, theta2):
    x1, y1 = polar_to_cartesian(rho1, theta1)      # Llamada a función CUDA desde ufuncs vectorizables
    x2, y2 = polar_to_cartesian(rho2, theta2)
    return ((x1 - x2)**2 + (y1 - y2)**2)**0.5
```

Limitaciones de código en GPU:

- if, elif, else
- while, for
- Operadores matemáticos básicos
- Algunas funciones de los módulos math y cmath
- Tuplas
- [Y algunas cosas más](#)

PYTHON: CAMINITO DE LA SINTAXIS CUDA

- Movimientos entre memorias:
 - NUMBA tiene herramientas para:
 - Mover datos entre memorias
 - Crear datos directamente en alguna memoria concreta
 - Manejar memoria pinned...

```
fn = 100000
noise = (np.random.normal(size=n) * 3).astype(np.float32)    # Datos en memoria Host
t = np.arange(n, dtype=np.float32)
period = n / 23

d_noise = cuda.to_device(noise)                                # Datos movidos a GPU
d_t = cuda.to_device(t)                                      # Datos movidos a GPU
d_pulses = cuda.device_array(shape=(n,), dtype=np.float32)   # Datos directamente generados en memoria de GPU, más adelante podremos copiar
                                                               # de/al host con d_pulses_host=d_pulses.copy_to_host()

make_pulses(d_t, period, 100.0, out=d_pulses)                 # Función vectorizada con target=cuda
waveform = add_ufunc(d_pulses, d_noise)                      # Función vectorizada con target=cuda
```

PYTHON: CAMINITO DE LA SINTAXIS CUDA

- Variables intrínsecas. NUMBA ofrece mecanismos para acceder a las variables sintrínsecas de forma análoga a como lo hace CUDA:

```
from numba import cuda
```

```
# Hace falta un array de salida "out" como parámetro. Los kernels CUDA (con @cuda.jit) no pueden devolver nada, como los kernels de C.  
@cuda.jit('void(int32[:,], int32[:,], int32[:,])')  
def add_kernel(x, y, out):  
    # Los valores de hilos y bloques no se saben hasta el momento de ejecución del kernel.  
  
    # Esto obtiene el id de un solo hilo de todos los generados  
    idx = cuda.grid(1)    # 1 = malla unidimensional, devuelve un valor escalar.  
                        # Esta función de NUMBA es equivalente a cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x  
  
    # Cada hilo procesará un solo valor del array general  
    out[idx] = x[idx] + y[idx]
```

- O también:

```
tx = cuda.threadIdx.x  ty = cuda.threadIdx.y  bx = cuda.blockIdx.x  by = cuda.blockIdx.y  bw = cuda.blockDim.x  bh = cuda.blockDim.y
```

PYTHON: CAMINITO DE LA SINTAXIS CUDA

- Variables intrínsecas. También encontraremos opciones para 2 y 3 dimensiones ([y algunas variables más](#)):

```
from numba import cuda

@cuda.jit
def get_2D_indices(A):
    x, y = cuda.grid(2)                      # El 2 señala que queremos dos valores en las dos dimensiones disponibles
                                                # Esa línea es equivalente a las dos siguientes:
                                                # x = cuda.blockIdx.x * cuda.blockDim.x + cuda.threadIdx.x
                                                # y = cuda.blockIdx.y * cuda.blockDim.y + cuda.threadIdx.y

    # Ya podemos operar en 2D
    A[x][y] = x + y / 10

Blocks = (2, 2)                            # Bloques definidos en 2D: malla de 2x2 bloques. Para una dimensión bastaría con Blocks = 10, p.e.
threads_per_block = (2, 2)                  # Hilos por bloque definidos también en 2D: bloques de 2x2 hilos
get_2D_indices[blocks, threads_per_block](d_A) # Llamada estándar a un kernel en GPU desde python
result = d_A.copy_to_host()
```

PYTHON: CAMINITO DE LA SINTAXIS CUDA

- Configuración de ejecución. NUMBA tiene una sintaxis específica para determinar bloques e hilos por bloque de forma análoga a como lo hace CUDA:

```
add_kernel[blocks_per_grid, threads_per_block](d_x, d_y, d_out)
cuda.synchronize()
print(d_out.copy_to_host())
```

Configuración de ejecución (en rojo) entre corchetes
Barreras de sincronización necesarias, ya sabemos por qué
Copia final al Host para poder imprimir.

- Por supuesto, hay que elegir valores adecuados para la configuración de ejecución, en las dimensiones adecuadas.
- Ej: uso de strides para estructuras muy grandes:

```
@cuda.jit
def add_kernel(x, y, out):
    start = cuda.grid(1)
    stride = cuda.gridsize(1)

    for i in range(start, x.shape[0], stride):
        out[i] = x[i] + y[i]
```

Obtenemos el tamaño total de la malla, equivalente a *cuda.blockDim.x * cuda.gridDim.x*
Cada hilo trabajará sobre su índice y los demás elementos separados *stride* de dicho índice.

PYTHON: CAMINITO DE LA SINTAXIS CUDA

- NUMBA implementa, también operaciones atómicas:

```
@cuda.jit  
def thread_counter_safe(global_counter):  
    cuda.atomic.add(global_counter, 0, 1) # Incrementar 1 en offset 0 del array global global_counter
```

- Para memoria compartida:

```
@cuda.jit  
def swap_with_shared(vector, swapped):  
    temp = cuda.shared.array(4, dtype=types.int32) # Reservar memoria shared de tipo int32 con 4 posiciones.  
    idx = cuda.grid(1)  
    temp[idx] = vector[idx] # Copiar a la memoria shared los valores que necesitamos. Cada hilo copia un dato.  
    cuda.syncthreads() # Ya sabemos para qué es esto... Recordad: ¡a nivel de bloque!  
    swapped[idx] = temp[3 - cuda.threadIdx.x] # Cada hilo lee un dato que escribió otro hilo en la memoria shared.
```

- Y para manejar streams, memoria local (de cada hilo), la de constantes...
- Cualquier kernel de NUMBA se verá afectado por la coalescencia (y conflictos entre bancos...), exactamente igual que con los kernels de CUDA.