ICNPG 2023

Clase 4: CUDA C

nvprof

<<<...>>>

CudaGetDeviceProp

__device__

grillas

kernels

hilos

blockDim

Paralelismo

Host

CudaMemCopy

punteros

Nvidia

__global__

¿Preguntas de la clase 1,2?

Device

bloques

performances

CUDA

blockId

dim3

threadId

__host__

nvcc

CudaMalloc

gridDim

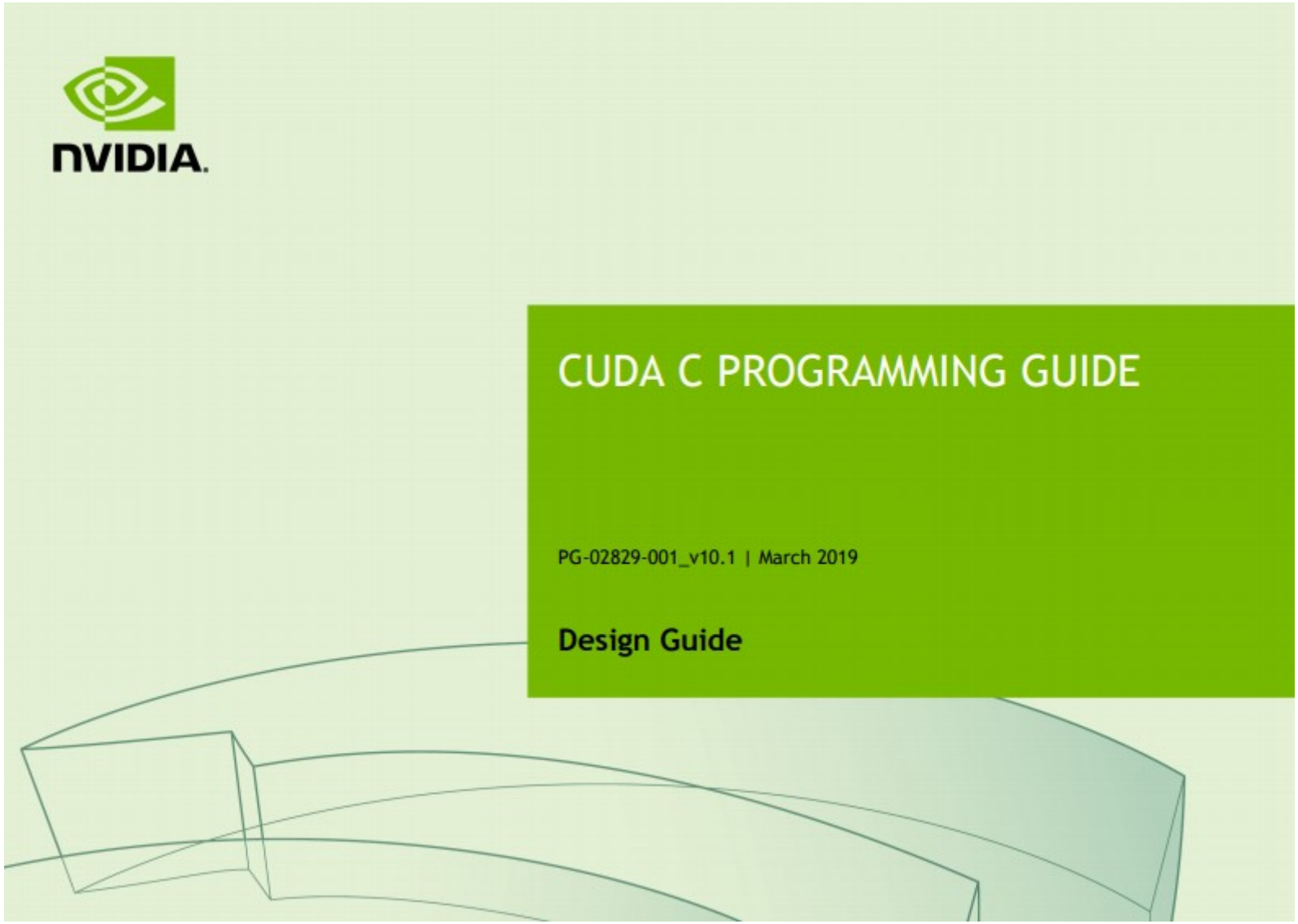
cudaFree

CPU-RAM

GPU-RAM

cudaDeviceSynchronize

https://docs.nvidia.com/cuda/cuda-c-programming-guide/



Tarea 1

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

- Completar el programa de Multiplicación de Matrices

```
// Matrix multiplication kernel called by MatMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
    // Each thread computes one element of C
    // by accumulating results into Cvalue
    float Cvalue = 0;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    for (int e = 0; e < A.width; ++e)
        Cvalue += A.elements[row * A.width + e]
                * B.elements[e * B.width + col];
    C.elements[row * C.width + col] = Cvalue;
}
```

En las librerías se llama "leading dimension"

¿Cual es la mínima modificación de esta función para que multiplique en CPU?
¿ Que tipo de orden de la matriz se está asumiendo, “row-” o “column-major” ?
¿ Que cuidado tengo que tener en el lanzamiento del Kernel ?

La diferencia está en el orden en el que se llevó la matrix a un vector unidimensional

Tarea 1

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

- Completar el programa de Multiplicacion de Matrices

```
// Matrix multiplication kernel called by MatMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
    // Each thread computes one element of C
    // by accumulating results into Cvalue
    float Cvalue = 0;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    for (int e = 0; e < A.width; ++e)
        Cvalue += A.elements[row * A.width + e]
                * B.elements[e * B.width + col];
    C.elements[row * C.width + col] = Cvalue;
}
```

```
// Invoke kernel
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);
```

Multiplicación de matrices

$O(N^3)$

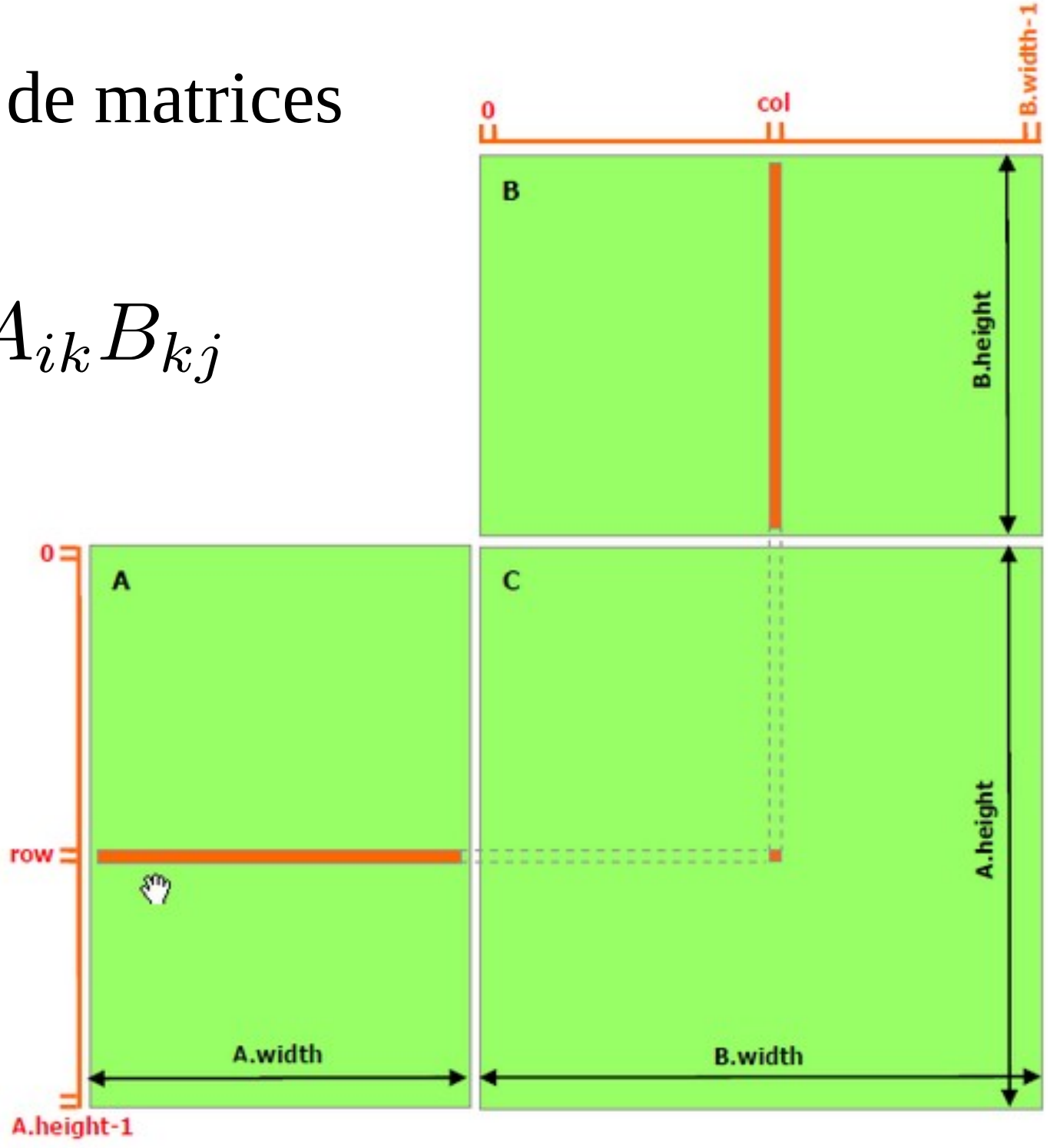
$$C_{ij} = \sum_k A_{ik} B_{kj}$$

Un hilo ← *Paralelización*

threadIdx
blockIdx

Varias formas de mapear a:

ij



Multiplicación de matrices: performance

- Comparar el tiempo de corrida en CPU y en GPU, a partir de las matrices A, B y C alocadas e inicializadas en el HOST.
- ¿ Como escalea con las dimensiones de la matriz ?
- Comparar las multiplicaciones simples de GPU y CPU
- Comparar con rutinas optimizadas de multiplicación de cuda: CUBLAS
- Comparar con rutinas optimizadas de multiplicación para CPU: blas, armadillo, etc..
- **¿ Conclusiones ?**

Multiplicación de matrices

- Ejercicio en el cluster

Premature optimization is the root of all evil (or at least most of it) in programming.

Donald Knuth,
“The Art of Computer Programming”

No hay que optimizar prematuramente. Podemos terminar con un código re optimizado de cosas que no eran necesario optimizar

Manejo de jerarquía de memorias instaladas en la GPU

- CUDA ofrece distintas memorias con distintas características:
  - Registros
  - Memoria compartida
  - Memoria global
  - Memoria constante.
  - Memoria de textura.
- Algunas de ellas están cacheadas.

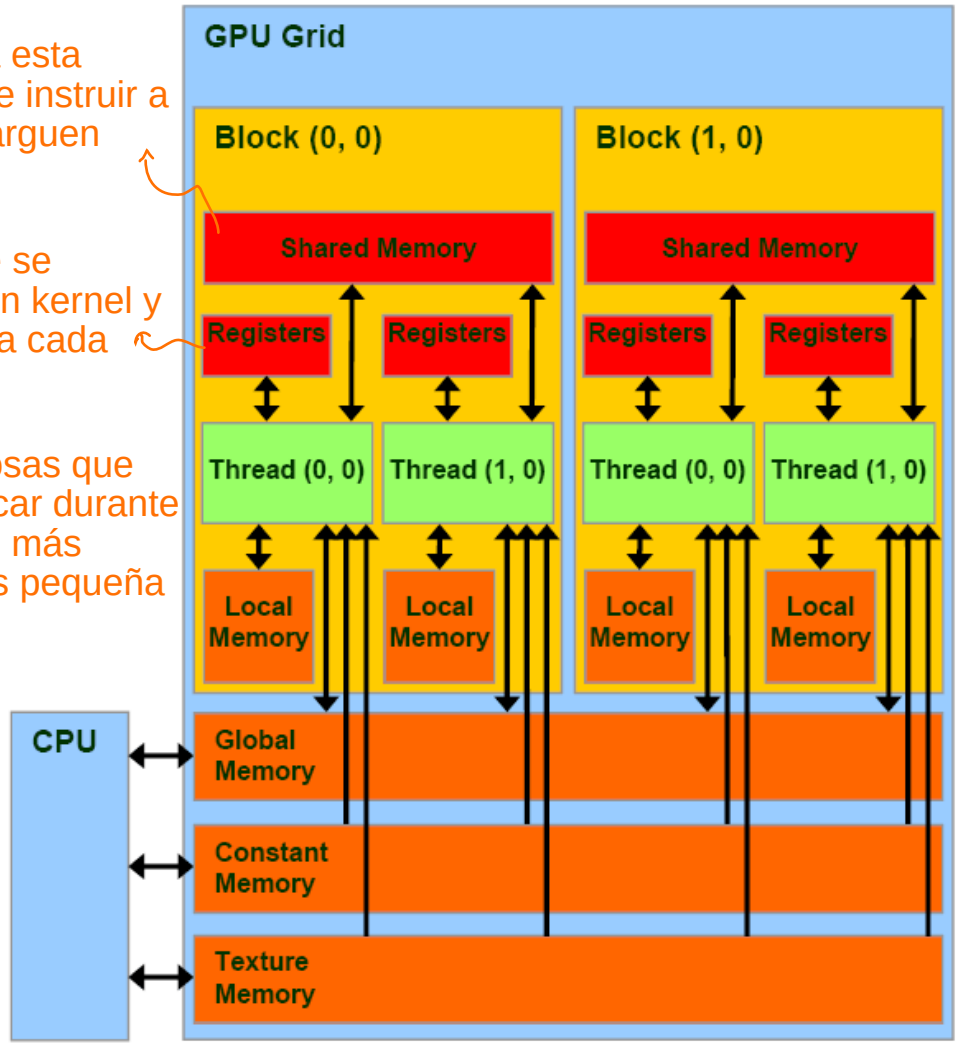
Para acceder a esta memria hay que instruir a los hilos que carguen info. ahí

Variables que se declaran en un kernel y son privadas a cada hilo

para guardar cosas que no voy a modificar durante la ejecución. Es más rápida pero más pequeña que la anterior

Se usa para acceso a memoria del tipo espacial. Ejemplo: dada una imagen, me gustaría acceder al pixel de arriba o abajo. En row-major esas direcciones de memoria están lejos. En esta memoria están cerca

Es una optimización del programa para "tener a mano" una variable que vamos a estar usando muchi. Se borra automáticamente



- **Memoria Global**: es la más grande y la más lenta. Puede ser leída y escrita por la CPU y por los threads de GPU. Permite comunicar datos entre CPU y GPU. El patrón de acceso a memoria por los threads puede afectar el rendimiento. Se puede alocar solo desde host

- **Memoria Constante**: es parte de la memoria global. CPU puede leer y escribir, y es sólo de lectura para los threads. Ofrece mayor ancho de banda cuando grupos de threads acceden al mismo dato.

- **Memoria compartida:** es pequeña y muy rápida y es compartida por todos los threads de un bloque. Es de lectura/escritura por los threads. Puede comunicar datos entre threads del mismo bloque. Puede verse afectada por el patrón de acceso de los threads.

- **Registros**: cada thread utiliza su propio conjunto de registros. El programador no tiene control explícito de los registros, y son utilizados para la ejecución de programas de la misma forma que los registros de propósito general de CPU.

- **Memoria local**: es usada por el compilador automáticamente para alojar variables cuando hace falta. Cuando no alcanzan los registros, se usa memoria local que fisicamente es la global.

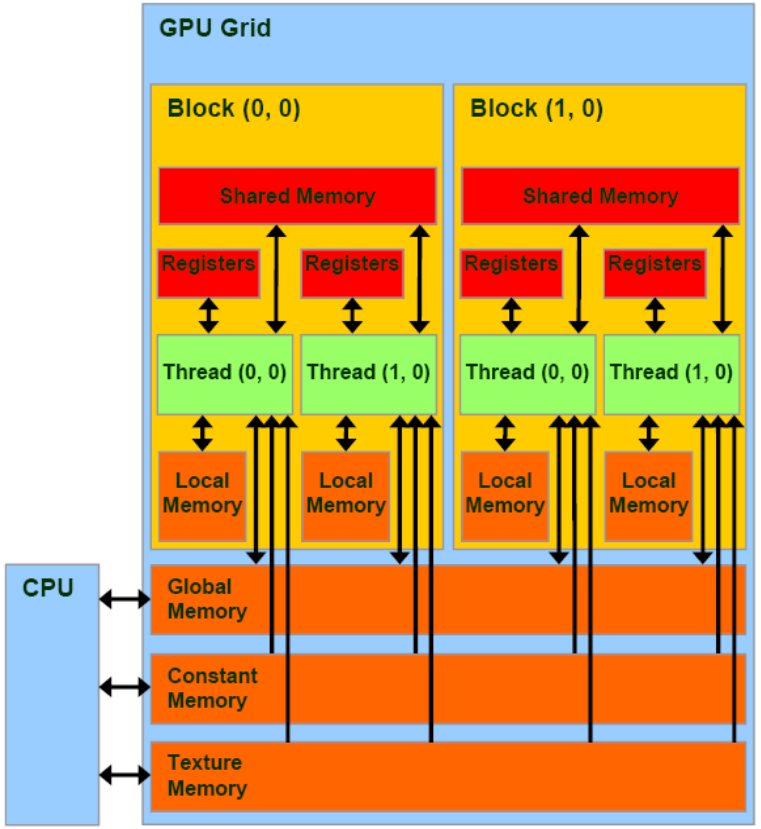
- **Memoria de textura**: es controlada por el programador y puede beneficiar aplicaciones con localidad espacial donde el acceso a memoria global es un cuello de botella.

Table 1. Salient Features of Device Memory

Memory	Location on/off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	Yes††	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	†	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation

Mientras el hilo está, la memoria tmb

solo lectura



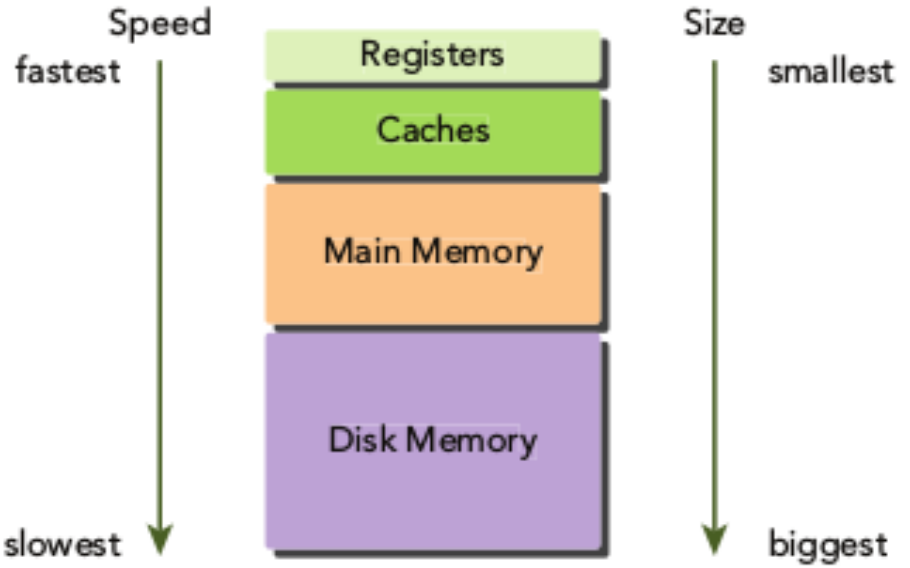
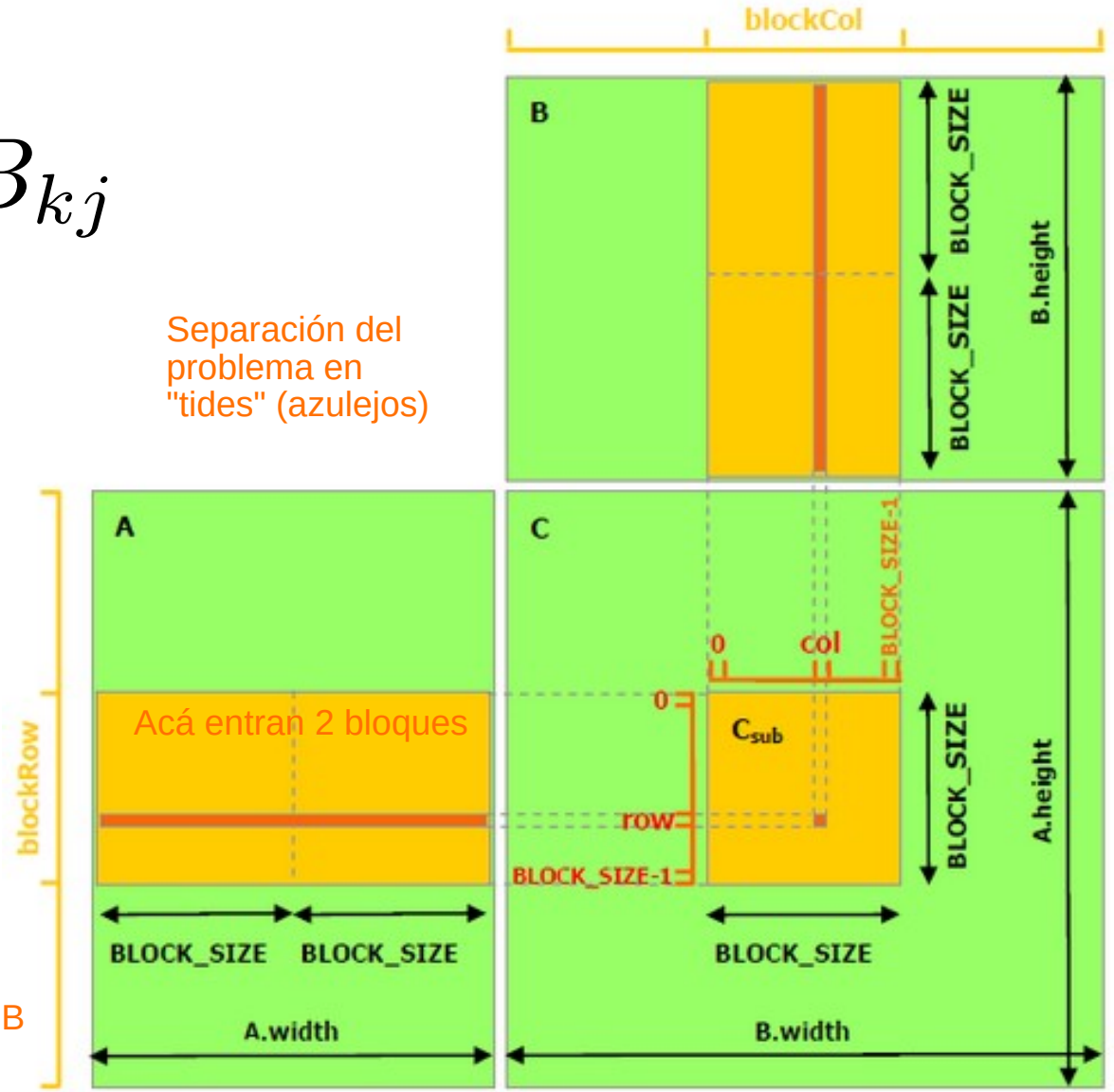


FIGURE 4-1

Multiplicación de matrices (optimizada #1)

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

Figure 10. Matrix Multiplication with Shared Memory



Separación del problema en "tides" (azulejos)

Submatriz de C → un bloque
Shared Memory

¿Porque conviene shared-memory?

Se gana velocidad respecto a usar memoria global porque los elementos del bloque en naranja de A y B son accedidos muchas veces

Multiplicación de matrices usando shared memory

```
__global__ void matmul_kernel(float *A, float *B, float *C, int n) {
    // Declare shared memory arrays for input matrices A and B
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    // Compute thread and block indices
    int bx = blockIdx.x;
    int by = blockIdx.y;
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Compute row and column indices of output matrix C
    int row = by * blockDim.y + ty;
    int col = bx * blockDim.x + tx;
```

```
// Initialize the output element to 0
float Csub = 0.0;

// Loop over input matrices in blocks of size BLOCK_SIZE
for (int i = 0; i < n / BLOCK_SIZE; i++) {
    // Load block of input matrix A into shared memory
    As[ty][tx] = A[row * n + i * BLOCK_SIZE + tx];
    // Load block of input matrix B into shared memory
    Bs[ty][tx] = B[(i * BLOCK_SIZE + ty) * n + col];
    // Synchronize threads to ensure all shared memory loads have completed
    __syncthreads();

    // Compute dot product of the corresponding row of A and column of B
    for (int j = 0; j < BLOCK_SIZE; j++) {
        Csub += As[ty][j] * Bs[j][tx];
    }
    // Synchronize threads to ensure all shared memory stores have completed
    __syncthreads();
}

// Write the output element to global memory
C[row * n + col] = Csub;
```

- Cada bloque de hilos de BLOCK_SIZE x BLOCK_SIZE calcula un bloque de de la matriz C, pero cada hilo tiene su elemento de C. Colaboran porque usan memoria compartida.
- Necesito cargar varios bloques BLOCK_SIZE x BLOCK_SIZE
  de las matrices A y B, una “franja horizontal/vertical”
  de toda la matriz A/B.
- Por cada bloque de A y B que cargo, sincronizo hilos, y cada hilo hace una multiplicación de vectores y suma una de las contribuciones a C.

Esto se puede hacer con todos los hilos de un mismo bloque

No puedo arrancar la multiplicación sin haber terminado de cargar los datos

Ahora tenemos que cargar memoria en shared memory y sincronizar hilos.... Esto tiene un costo que se amortiza a mayor tamaño de la matriz

Multiplicación de matrices:

$$C = \alpha \mathrm{op}(A)\mathrm{op}(B) + \beta C$$

por si uno quiere implementar el valor de C por un coeficiente

operación aplicada a A: conjugar, transponer,...

CUBLAS API

cublasSgemm(manija,CUBLAS_OP_N,CUBLAS_OP_N,m,m,m,&al, d_A.elements,m,d_B.elements,m, &bet, d_C.elements,m);

https://docs.nvidia.com/cuda/cublas/index.html#using-the-cublas-api

CUBLASXt API

cublasXtSgemm(manija,CUBLAS_OP_N,CUBLAS_OP_N,m,m,m,&al,A.elements,m,B.elements,m, &bet,C.elements,m);

https://docs.nvidia.com/cuda/cublas/index.html#using-the-cublasXt-api

Para medir tiempos hay que considerar que la primera vez que uno lanza un kernel tarda más tiempo

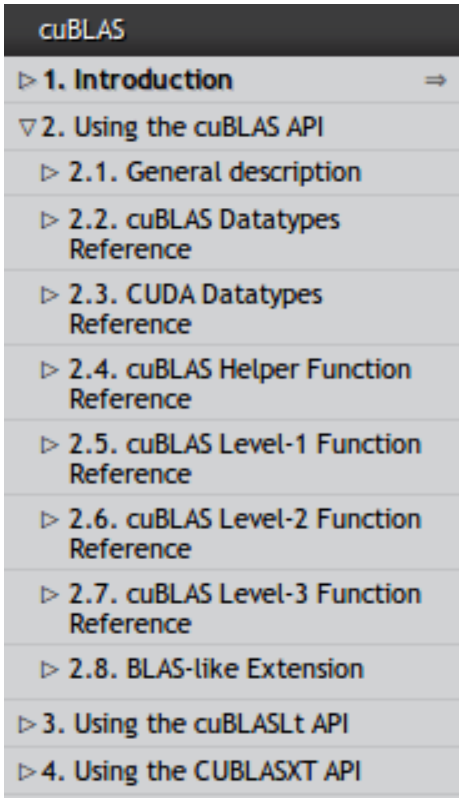
Incluso las librerías de TensorFlow o las de Python llaman por debajo a estas librerías

¿ Cual es preferible ?

La única diferencia es que la segunda acepta punteros de host y devuelve punteros a host, con lo cual uno se olvida que existe una GPU. Mientras que la primera es de más bajo nivel y tengo que darle punteros a GPU

Bibliotecas de Algebra Lineal Básica

- Operaciones: vector-vector, matriz-vector, matriz-matriz
- Las que manejan la memoria de device por uno: cublasXt API
- Las que esperan que los datos ya estén en device: cublas API
- ¿ Cual es preferible ?



Tarea 2

- Dada una señal discreta x[n] y un flitro h[n], la convolución $y = x * h$ se define como:

$$y[n] = [x * h][n] = \sum_k x[k+n]h[k]$$

  - Considerar un array x de números reales de tamaño N que representa la señal en un dominio discreto y un array h de tamaño M que describe el filtro h, M << N.

- ¿Como paralelizamos esto?

```
/* convolucion en la cpu: requiere dos loops */
void conv_sec(FLOAT* input, FLOAT* output, FLOAT * filter)
{
        FLOAT temp;
        for(int j=0;j<N;j++){
                temp=0.0;
                for(int i=0;i<M;i++){
                        temp += filter[i]*input[i+j];
                }
                output[j] = temp;
        }
}
```

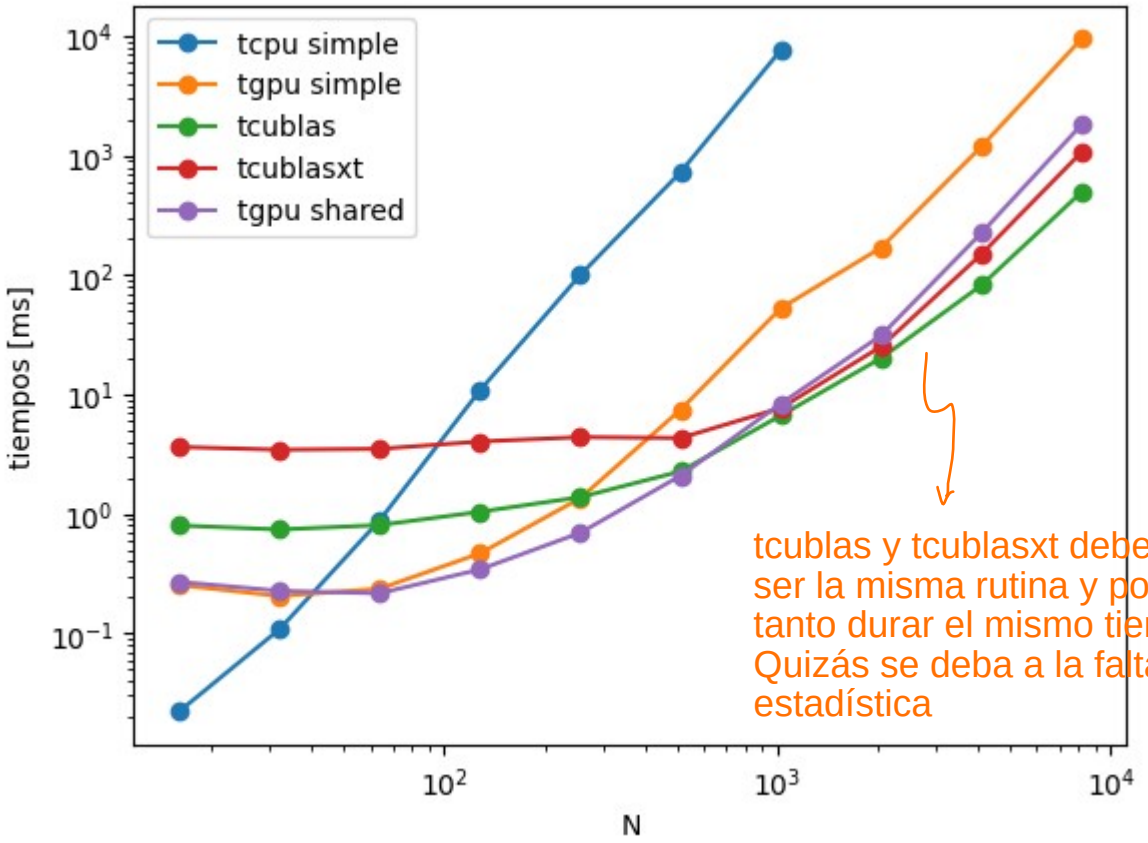
CPU para sistemas chicos, Cublas para grandes

```
std::cout << N << ", " << tcpu << ", " << tgpu << ", " << tcublas << ", " << tcublasxt << "," << tgpu2 << std::endl;
```

```
#@title veamos los tiempos
!cat tiempos.csv
```

```
16, 0.025829, 0.212272, 0.973593, 3.56597,0.186802
32, 0.109172, 0.194161, 0.778927, 3.41522,0.198833
64, 1.46313, 0.244008, 0.761812, 3.343,0.224012
128, 6.7704, 0.280491, 1.14203, 3.61958,0.3284
256, 56.5966, 0.618724, 1.09887, 3.64755,0.599275
512, 526.593, 2.1065, 2.37428, 4.3372,2.06049
1024, 8249.42, 9.529, 6.21019, 7.11964,9.4273
2048, 87878.3, 60.0578, 24.213, 29.2775,45.7572
```

¿Que tiene de bueno conocer las versiones simples de CUDA?



tcublas y tcublasxt deberían ser la misma rutina y por lo tanto durar el mismo tiempo. Quizás se deba a la falta de estadística

python

El código completo está en un Google Colab de la materia (ver Google Classroom)

numpy

```python
import numpy as np
import time

# Set the sizes of the matrices
m = 2048
n = 2048
p = 2048

# Generate random matrices with the given sizes
A = np.random.rand(m, n)
B = np.random.rand(n, p)

# Compute the matrix product with a timer
start_time = time.time()
C = np.dot(A, B)
end_time = time.time()


# Print the elapsed time
print("Elapsed time: ", (end_time - start_time)*1000, "mseconds")
```

cupy

```python
import cupy as cp
import time

# Set the sizes of the matrices
m = 2048
n = 2048
p = 2048

device = cp.cuda.Device()

# Generate random matrices with the given sizes
A = cp.random.rand(m, n)
B = cp.random.rand(n, p)

# Compute the matrix product with a timer
start_time = time.time()
C = cp.dot(A, B)
device.synchronize()
end_time = time.time()

# Print the elapsed time
print("Elapsed time: ", (end_time - start_time)*1000, "mseconds")
```

para obtener el identificador del device

corre en GPU

Si no hacemos esto el tiempo sería mucho más corto porque la función time() corre en CPU que corre en paralelo a la GPU

Acá tmb se está considerando el tiempo de alocación de la matriz C

En Python tmb es asincrónico

Da una mejora respecto a usar CPU PERO no es tan grande como si usásemos solo CUDA C. Cupy corre sobre CUDA. Incluso cupy averigua qué placa tenemos y busca las rutinas más eficientes para tal caso

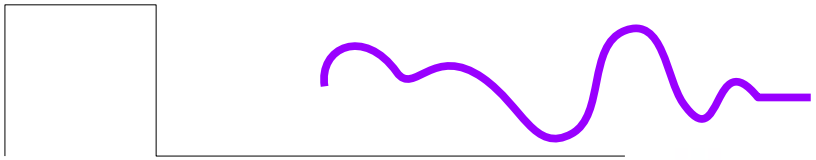
Cupy “raw kernels”

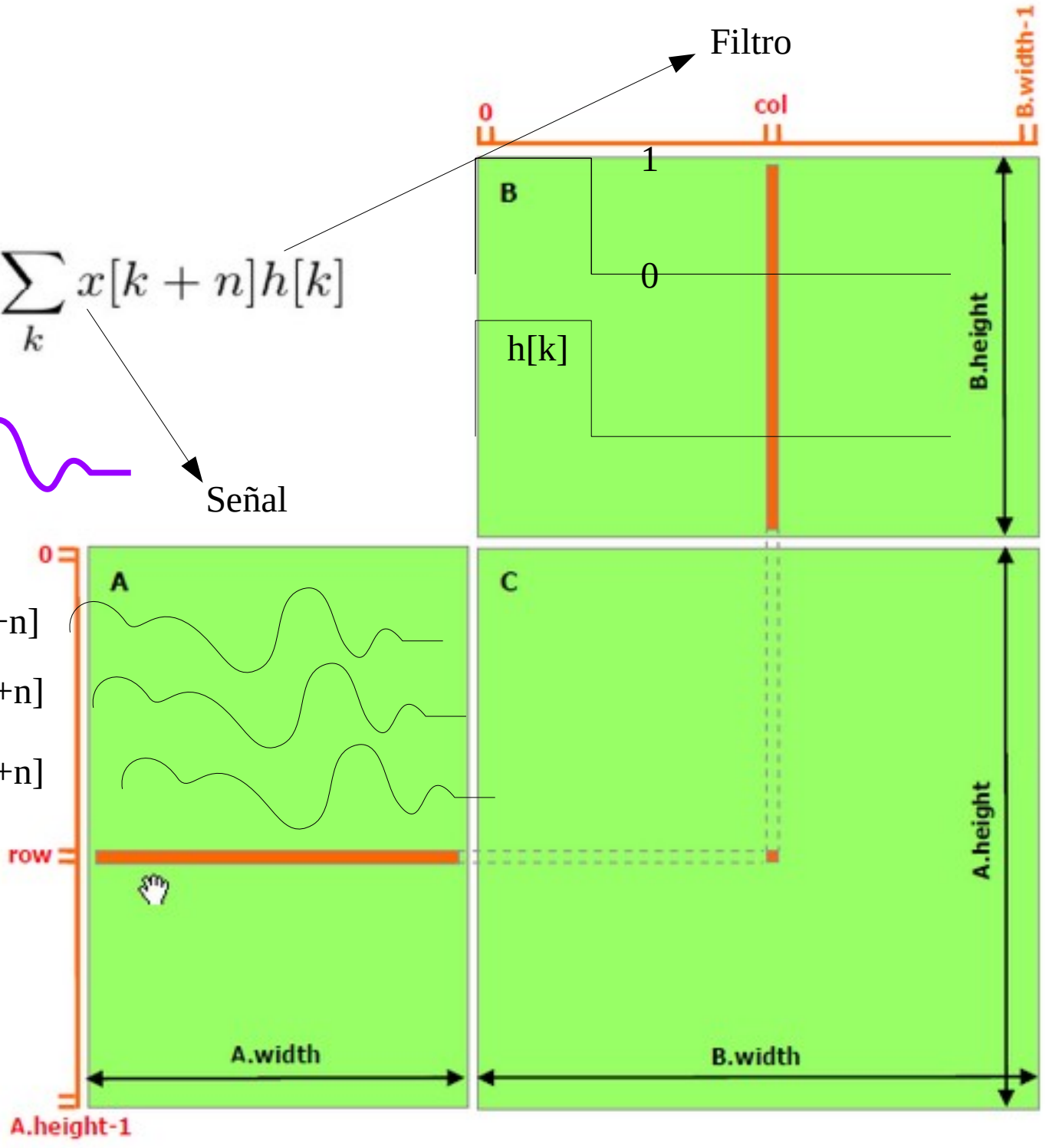
Da mejores tiempos que el caso anterior. Se está contando solo el tiempo de la multiplicación

```
# Define the kernel code as a string
kernel_code = """
extern "C" __global__ void matrix_multiply(float *a, float *b, float *c, int m, int n, int k) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < m && col < k) {
        float sum = 0.0f;
        for (int i = 0; i < n; i++) {
            sum += a[row * n + i] * b[i * k + col];
        }
        c[row * k + col] = sum;
    }
}
"""

# Compile the kernel code into a function
matrix_multiply = cp.RawKernel(kernel_code, 'matrix_multiply')

# Define the block and grid sizes for the kernel
block_size = (16, 16, 1)
grid_size = ((m + block_size[0] - 1) // block_size[0], (k + block_size[1] - 1) // block_size[1], 1)

start_time = time.time()

# Call the kernel function with the input matrices and sizes
matrix_multiply(grid_size, block_size, (a_gpu, b_gpu, c_gpu, m, n, k))

device.synchronize()
end_time = time.time()
```

Convolución

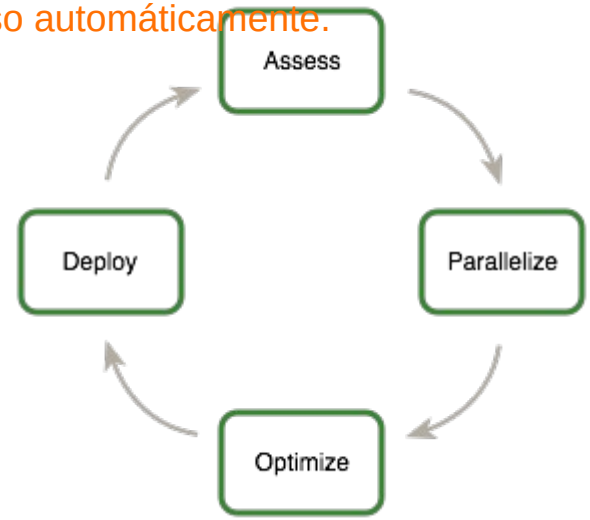
$$y[n] = [x * h][n] = \sum_k x[k+n]h[k]$$





Una forma es poner relojes y ver cuánto tiempo tarda cada función. Esto es manual y rústico.
Sin embargo, existen librerías que hacen todo el proceso automáticamente.

Profiling



CUDA C Best Practices Guide

- CPU (C/C++)
  - **gprof**
  - g++ -pg suma_vectores_cpu.cpp -o a.out;
  - ./a.out; gprof ./a.out
- GPU (CUDA)
  - **nvprof & nvvp**
  - https://devblogs.nvidia.com/cuda-pro-tip-nvprof-your-handy-universal-gpu-profiler/
  - nvcc suma_vectores_gpu.cu -o a.out;
  - Texto: nvprof ./a.out (texto)
  - Visual: nvprof -o out.prof ./a.out; nvvp out.prof
- CPU y GPU (runtime)
  - Incluir **cpu_timer.h** y **gpu_timer.h** y usar como esta en los ejemplos.
  - gpu_timer Reloj; Reloj.tic();…; ms=Reloj.tac();

Reducción

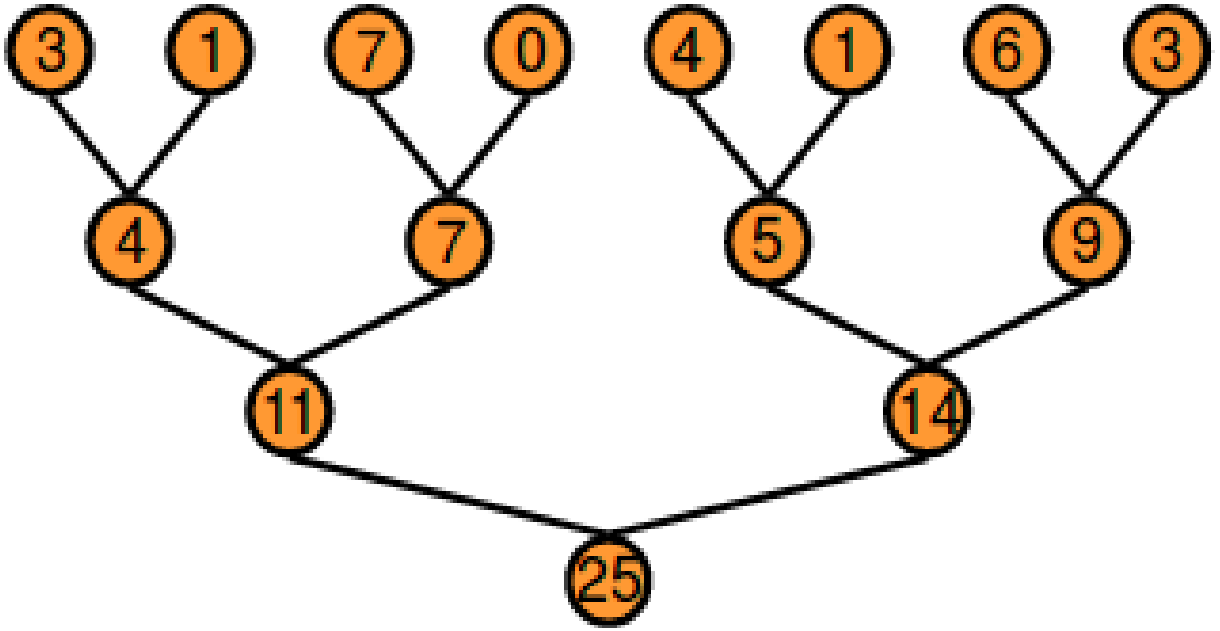
La reducción es posible para determinadas operaciones. Creo que hay que pedir que sea distributiva y asociativa

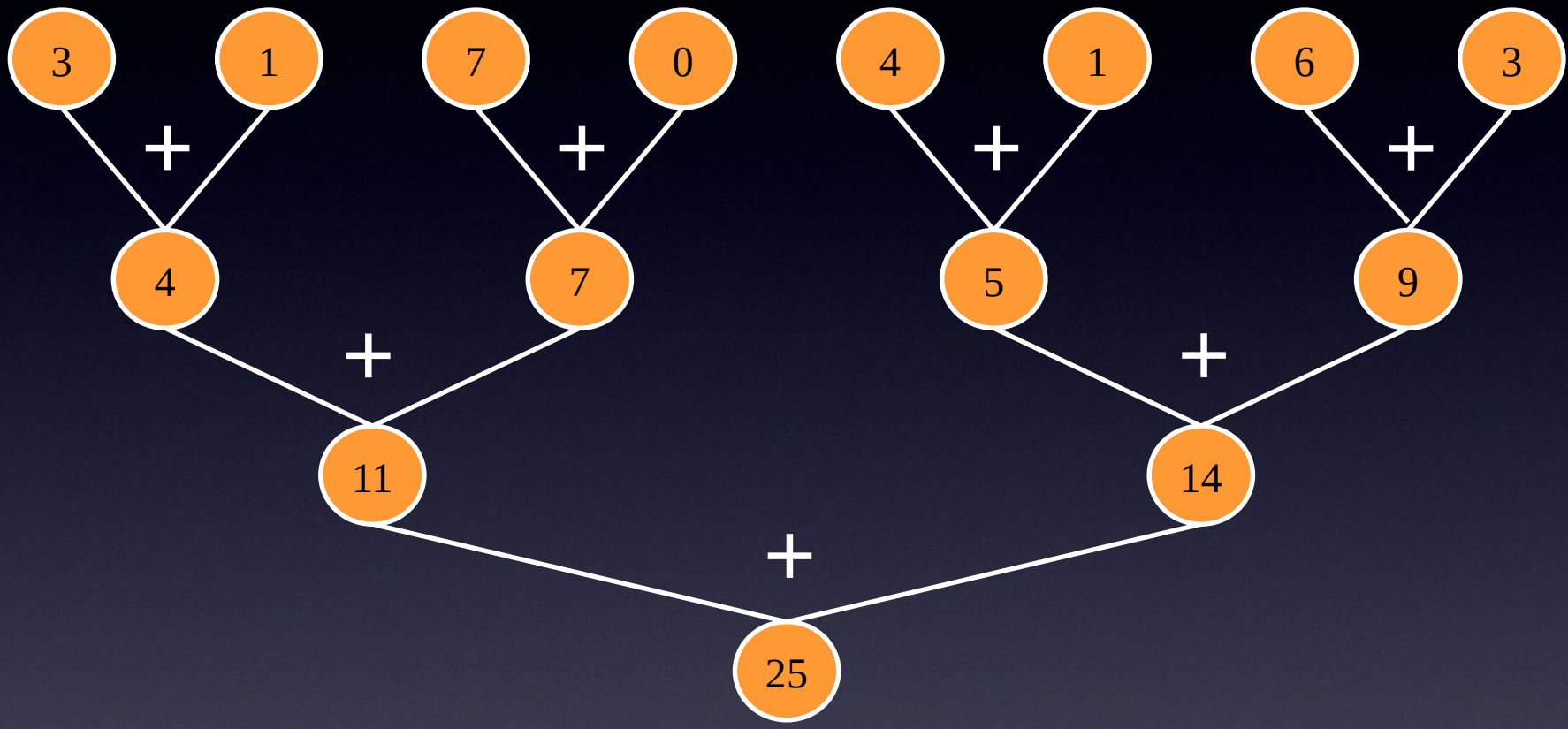
- Dado un vector de N elementos, calcular “la suma” de los mismos.

$$S = \sum_{i=0}^{N-1} x_i$$

¿Para que sirve?

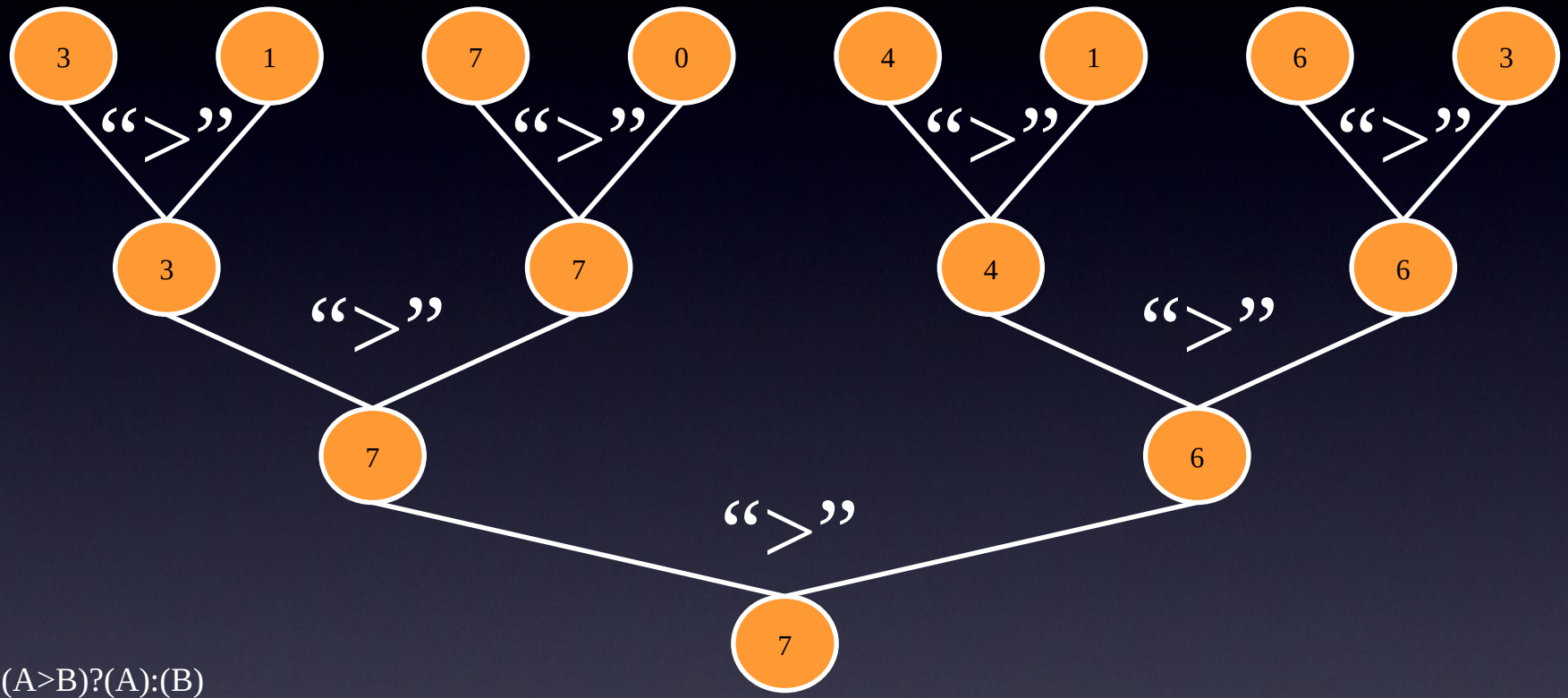
Reducción paralela: “se organiza un torneo”





Calcula la *suma*

Vale para int, float, double,o cualquier tipo de dato con la operación “+” *bien definida*...

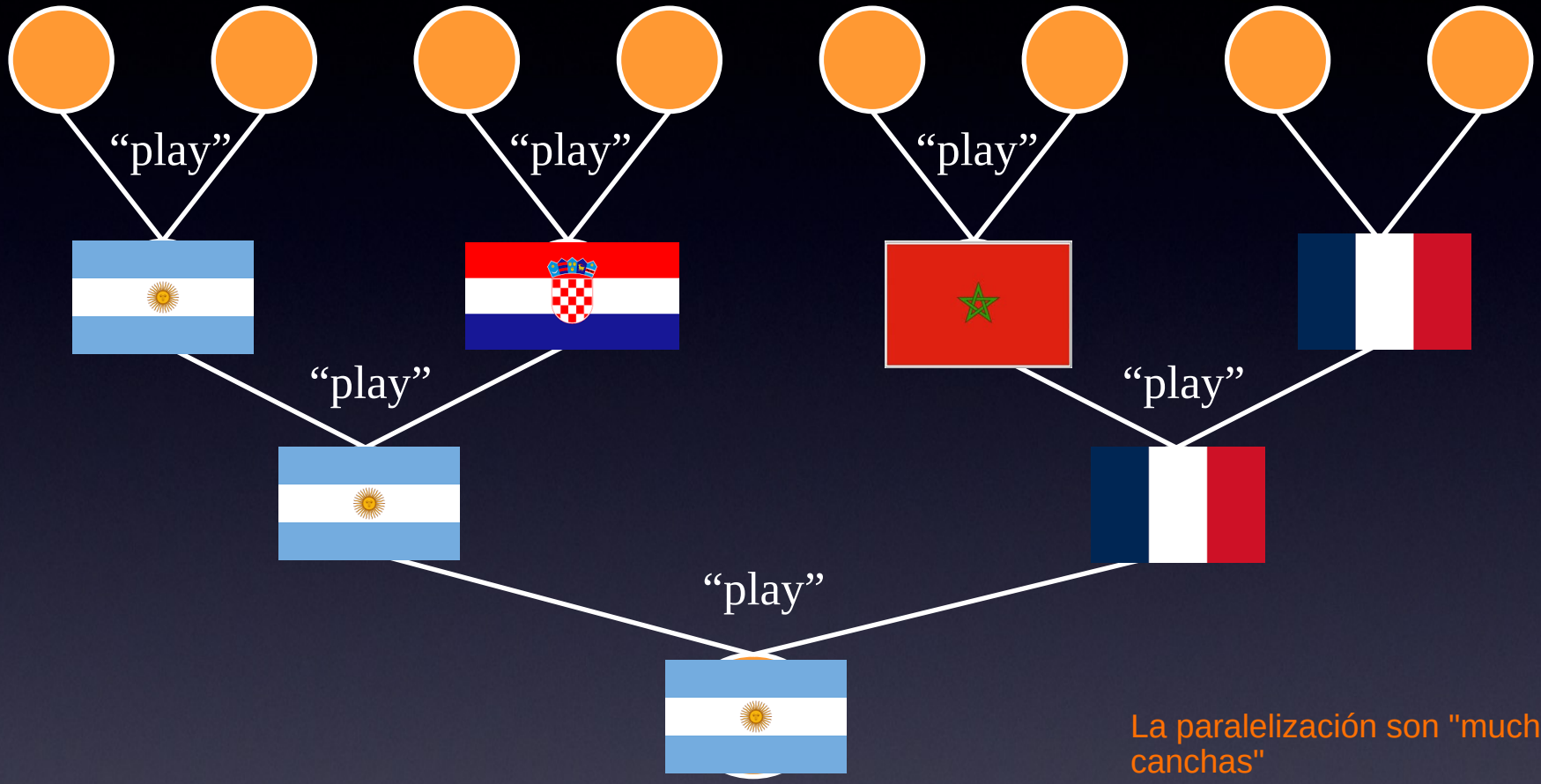


A “>” B = (A>B)?(A):(B)

Calcula el *máximo*

Vale para int, float, double,o cualquier tipo de dato con la operación “>” bien definida...

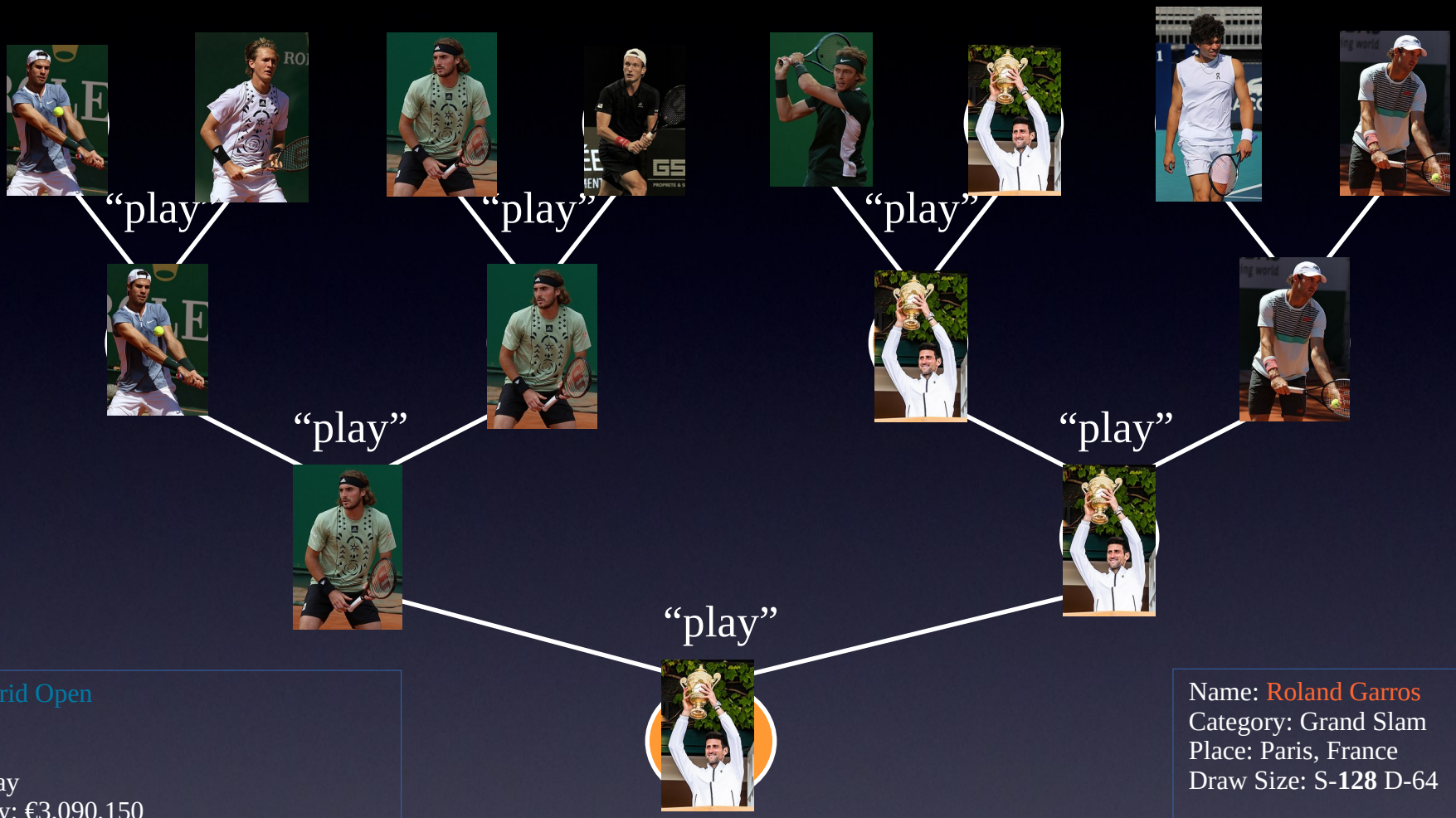
Copa del Mundo de Futbol: 32 equipos → 16 a octavos de final



La paralelización son "muchas canchas"

Calcula el *campeón*

Australian open 2023 (128→ 1)



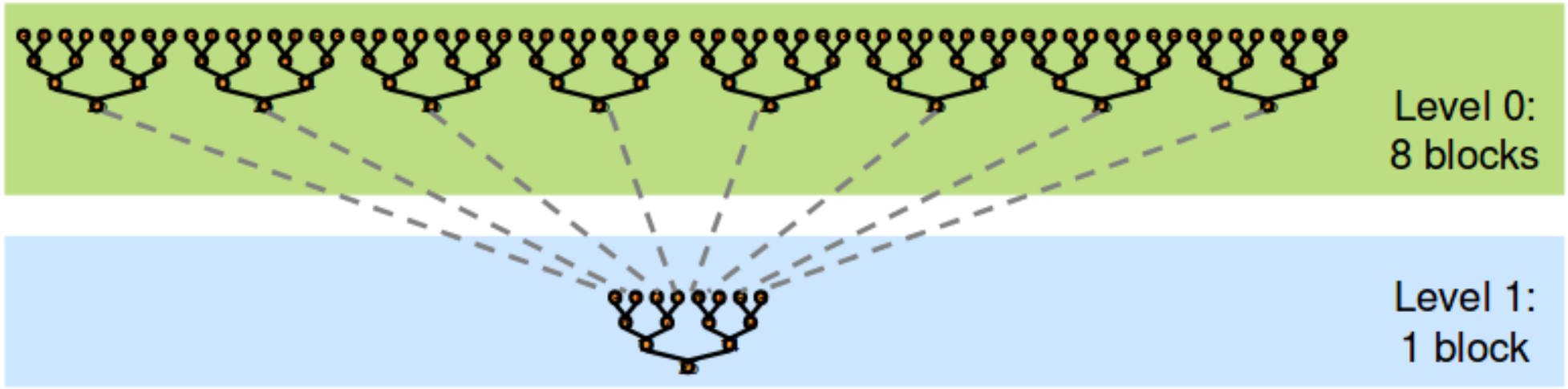
Mutua Madrid Open
Spain
Draw: **64**
Surface: Clay
Prize Money: €3,090,150
63 partidos en 6 jornadas

Calcula el *campeón*

Name: Roland Garros
Category: Grand Slam
Place: Paris, France
Draw Size: S-**128** D-64

log_2(N) operaciones:
Grand-Grand-Slam: 2^20 = 1048576 jugadores → solamente 20 “jornadas” !!!

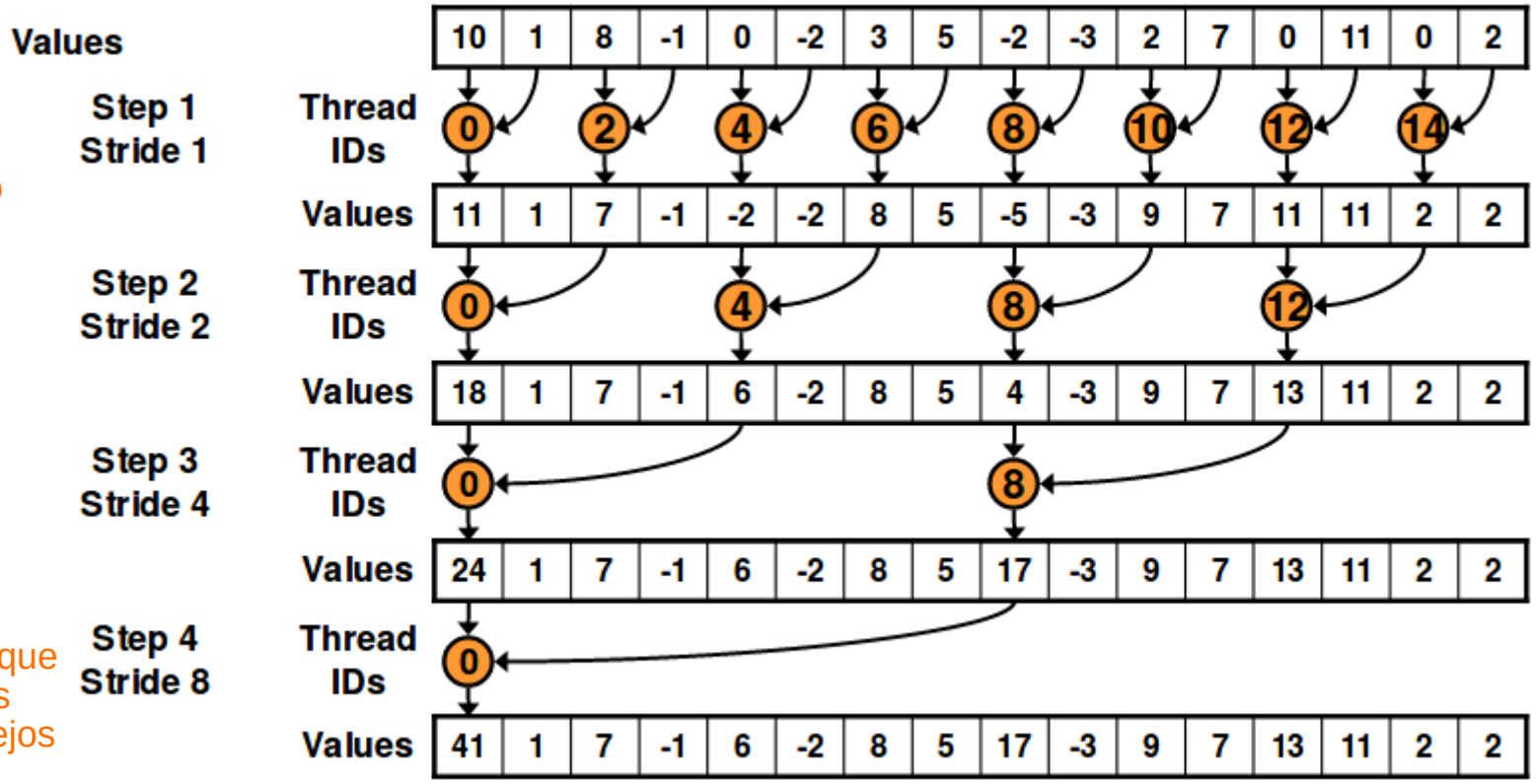
Reducción: N vs log(N)



https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

Reducción #1

Stride: "cuánto están separados"

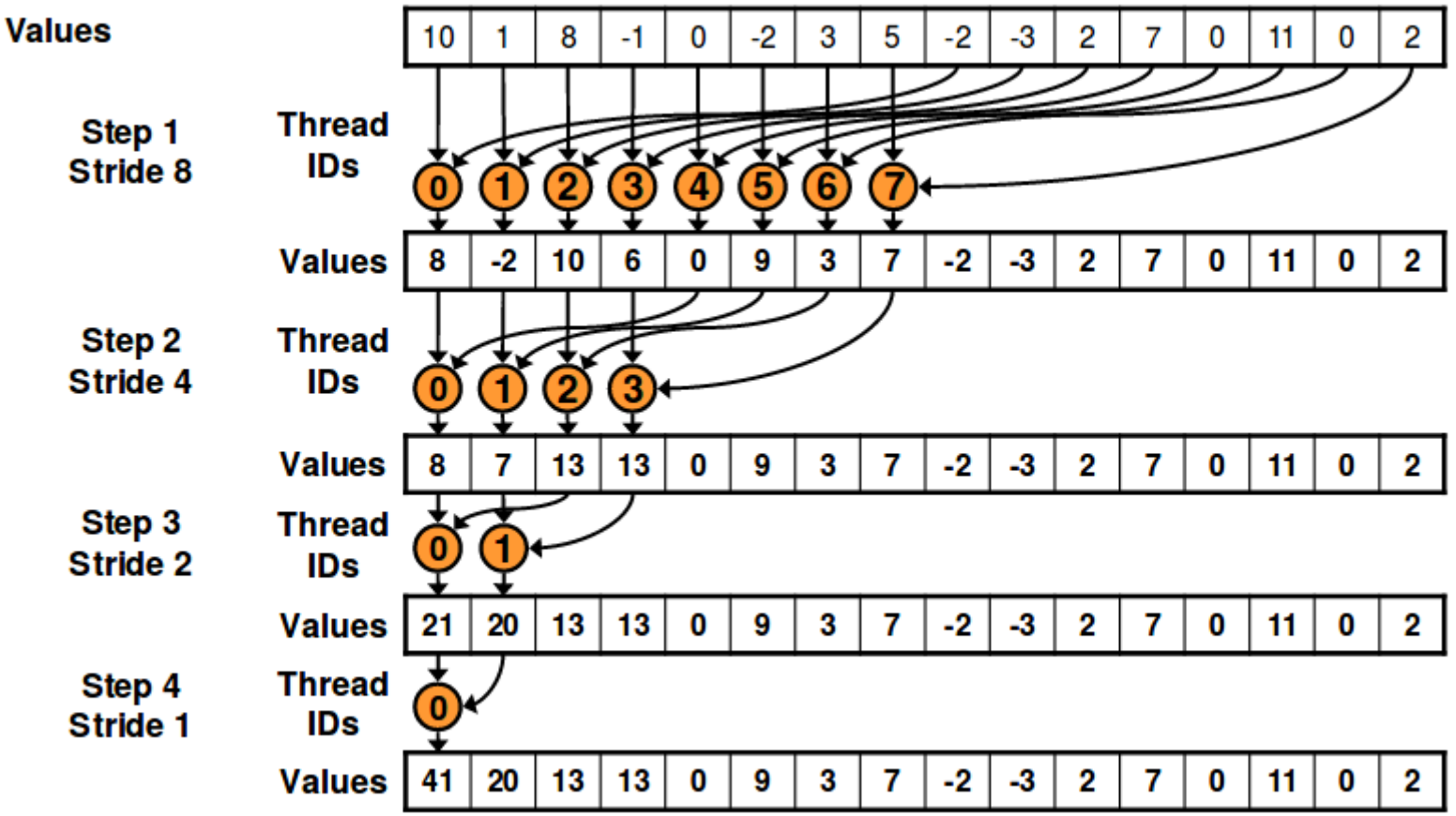


Cada vez tengo que sumar elementos que están más lejos en memoria

https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

Reducción #2

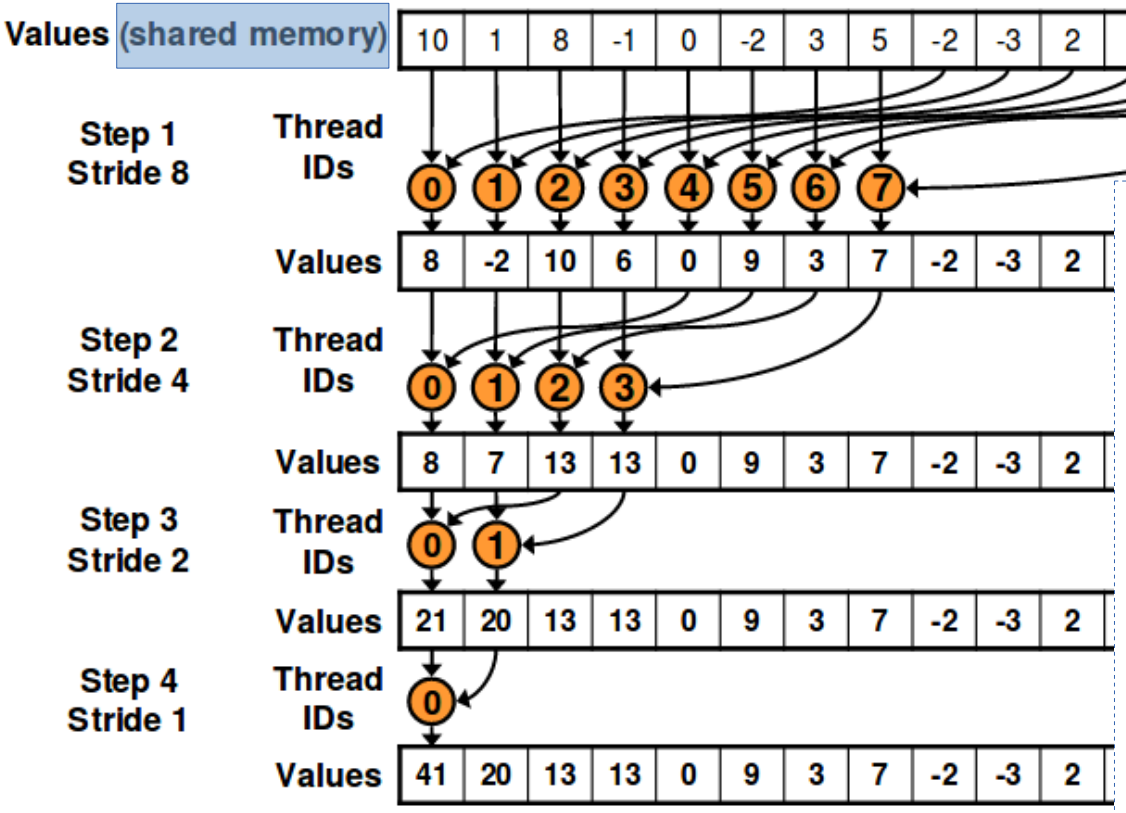
Por cuestiones de cómo se accede a la memoria, este es más eficiente que el anterior

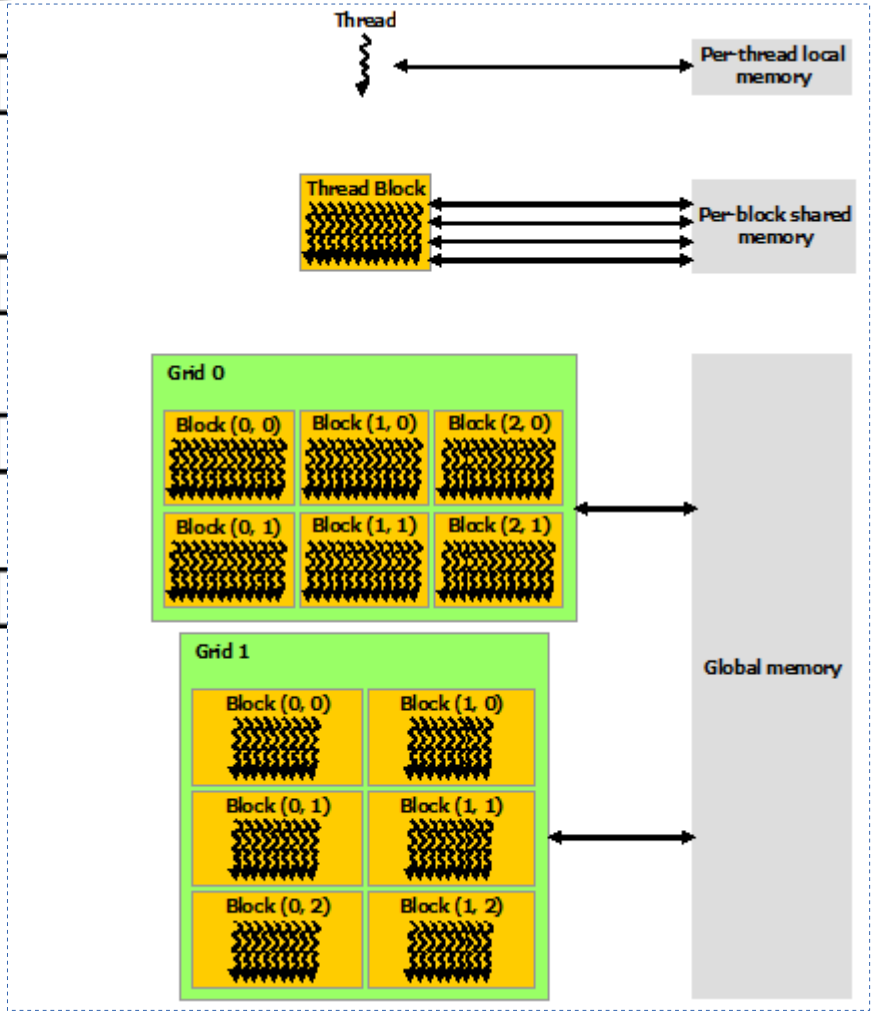


https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

Reduction #2 (optimizado)

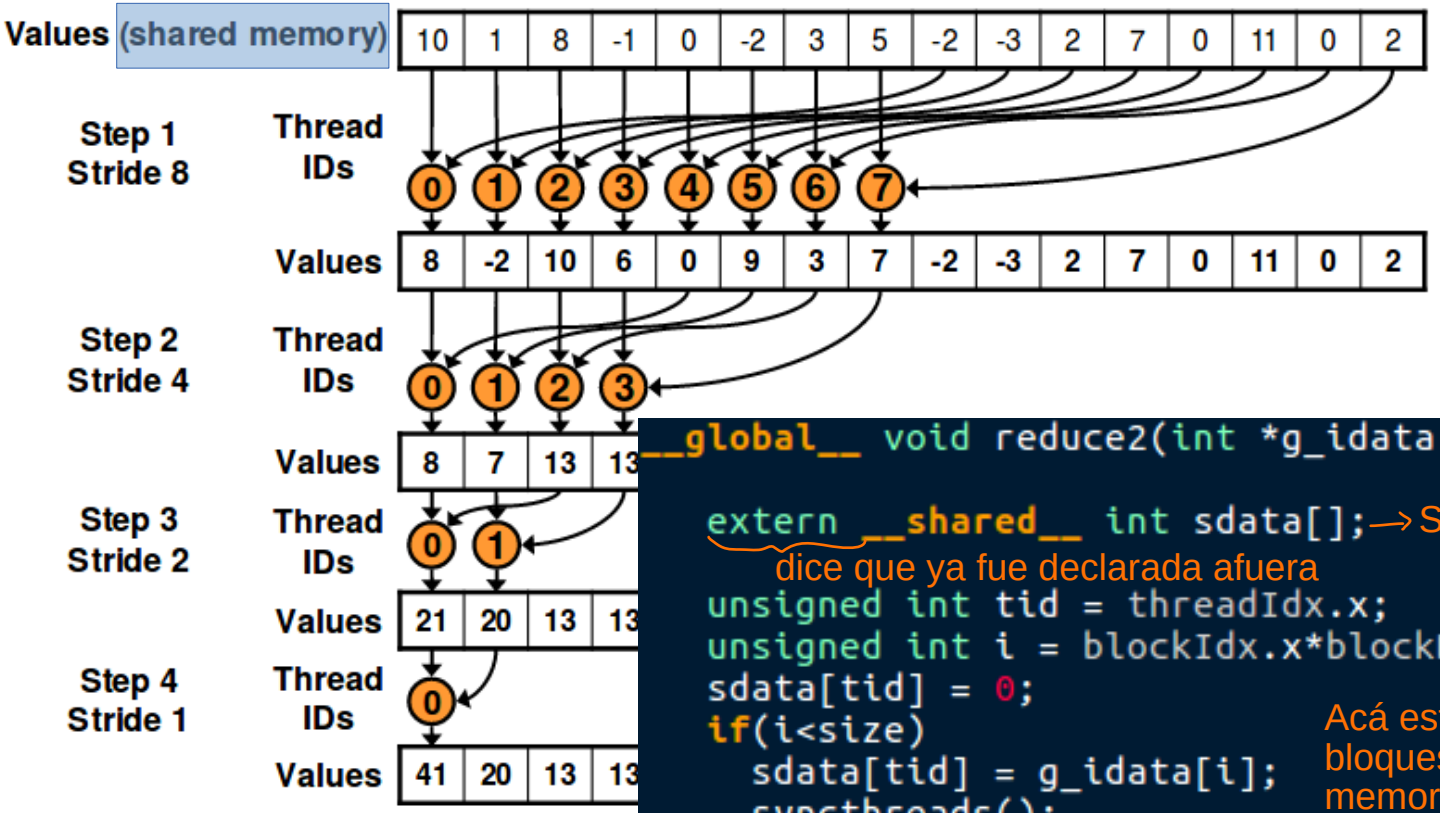
¿Por qué lo guardamos en shared memory? Porque muchos hilos van a usar esa info.





https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

Reduction #2 (optimizado)



Tiene los resultados parciales de cada bloque. Luego hay que hacer una reducción de g_odata. Podría usarse el mismo kernel o hacerlo secuencial porque ya se ganó bastante haciendo la reducción

Acá se pone el resultado

```
__global__ void reduce2(int *g_idata, int *g_odata, int size){

  extern __shared__ int sdata[];
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
  sdata[tid] = 0;
  if(i<size)
    sdata[tid] = g_idata[i];
  __syncthreads();

  for(unsigned int s=blockDim.x/2; s>0; s>>=1)
  {
      if (tid < s) {
          sdata[tid] += sdata[tid + s];
      }
      __syncthreads();
  }
  if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Se está declarando un puntero

dice que ya fue declarada afuera

Acá estoy cargando sdata en varios bloques! No hay problemas porque las memorias compartidas de cada bloque son distintas

"s" es la variable de "salto"

Vamos a ir modificando sdata

Esta tarea se la podría encargar a cualq hilo

Los threads ahora "colaboran", cargan y comparten Shared-memory

Divide s por 2. Es equivalente a correr un bit

Reduction #2 (optimizado)

reduce2<<<totalBlocks, threadsPerBlock, **threadsPerBlock*sizeof(int)**>>>
(input, output, size);

Cuánta memoria quiero alocar en la memoria compartida. Podría haber sacado el extern y puesto directamente la dimensión de sdata dentro del kernel


```
__global__ void reduce2(int *g_idata, int *g_odata, int size){

   extern __shared__ int sdata[];

   unsigned int tid = threadIdx.x;
   unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
   sdata[tid] = 0;
   if(i<size)
     sdata[tid] = g_idata[i];
   __syncthreads();

   for(unsigned int s=blockDim.x/2; s>0; s>>=1)
   {
        if (tid < s) {
                sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
   }
   if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

RECORDEMOS que no existe un modo de sincronizar los hilos de todos los bloques!!

No existe porque sería costoso y rompería la paralelización

Los threads ahora "colaboran", cargan y comparten la shared-memory

Parallel primitives

- Buenas noticias: hay librerías genéricas para implementarlas
- Reducción en **Thrust**:

```
int suma = reduce(device,input,input+size);
```

Donde
- “device” indica que el “reduce” se hace en paralelo en el device.
- input es el puntero a los datos, ya copiados en device.
- Size es el tamaño del array.

No hay kernels a la vista, la implementación está “delegada”
El input queda en device.

CUDA C/C++, librerías, etc

Sus proyectos

Un curso “project driven”

- Vayan investigando y pensando en un problema para resolver usando GPGPU.
- Si les sirve para su trabajo buenísimo (sino, que sea instructivo o divertido!).
- Con que sea conceptual, fundacional o prueba piloto ok!
- Herramientas: Cuda C/C++, pycuda, bibliotecas, aplicaciones, etc, combinadas con otras que no sean del curso.
- Que cualquiera pueda ver el código, correrlo y evaluar performance.

- **Evaluación final de la materia:** *Charlita* al final del curso explicando el problema, la motivación, la implementación, los resultados, las perspectivas + *Códigos* que compilen y corran.

Hay que ir entregando problemas?