

ICNPG 2023

Clase 4: CUDA C



nvprof

<<<...>>>

CudaGetDeviceProp

__device__

hilos

kernels

grillas

blockDim

Paralelismo

CudaMemcpy

Host

punteros

Nvidia

__global__

¿Preguntas de la clase 1,2?

Device

bloques

performances

CUDA

blockId

dim3

threadId

__host__

nvcc

CudaMalloc

gridDim

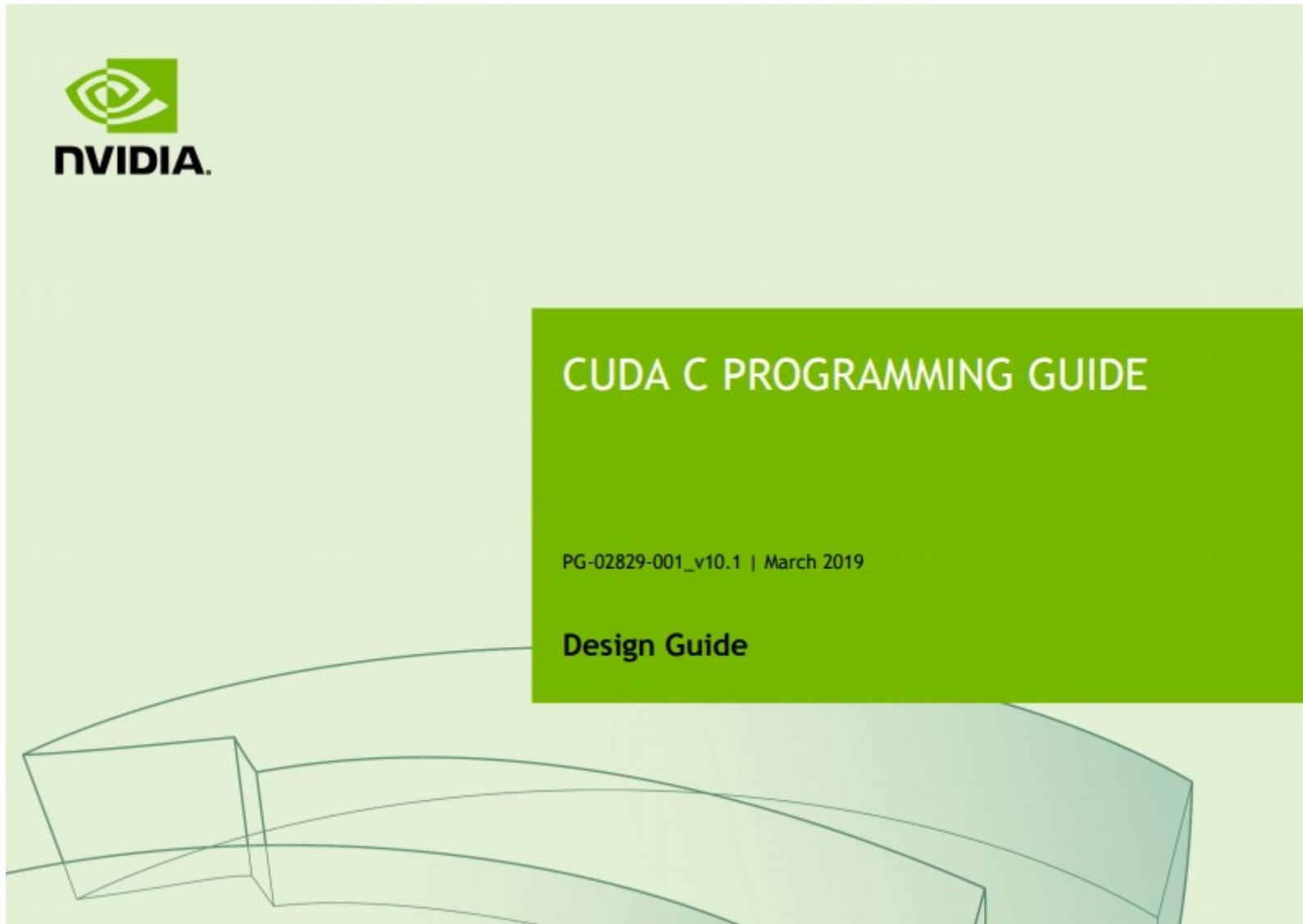
cudaFree

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;  
}
```

- ¿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 ?

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

$O(N^3)$

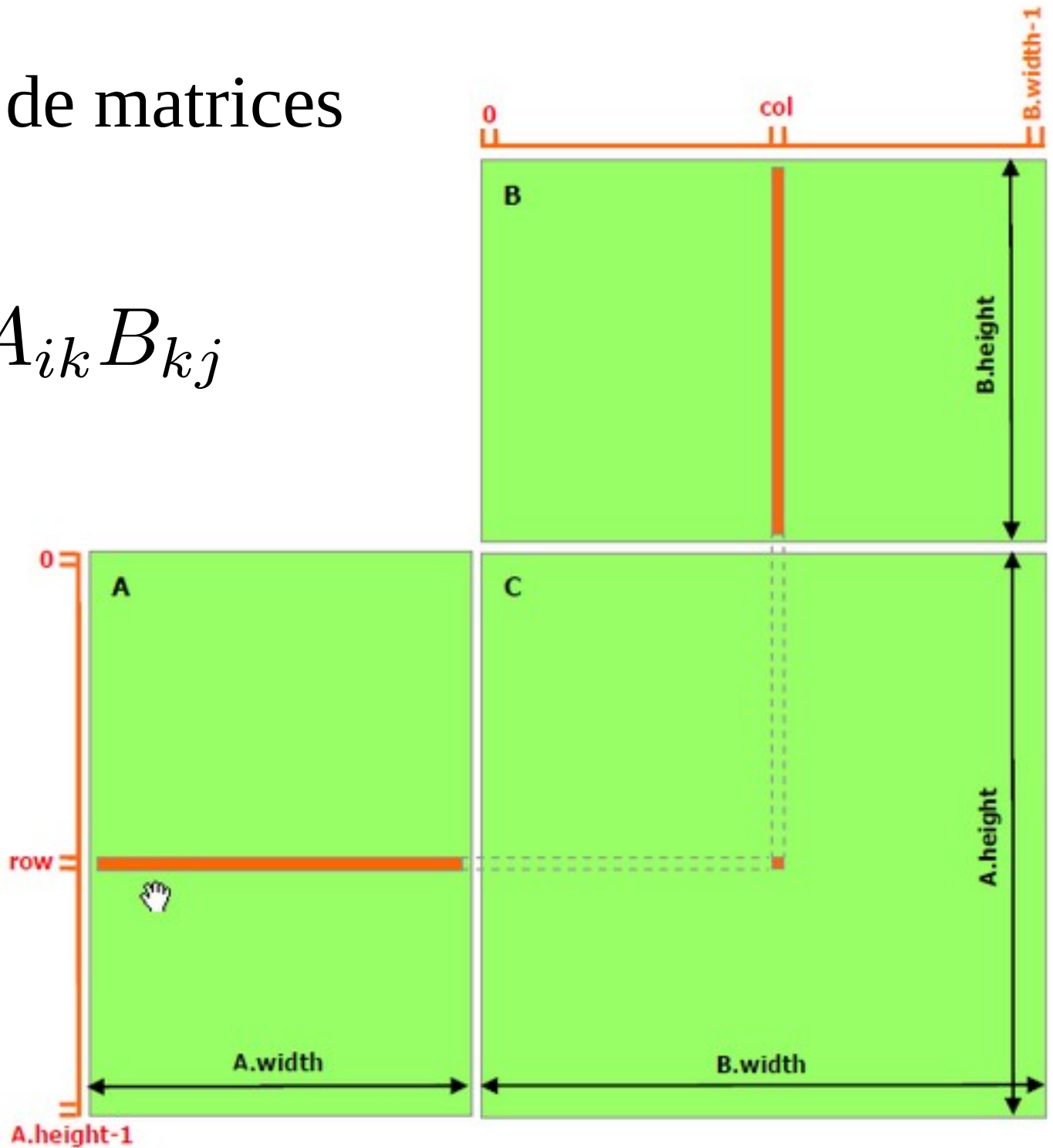
$$C_{ij} = \sum_k A_{ik} B_{kj}$$

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 escala 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

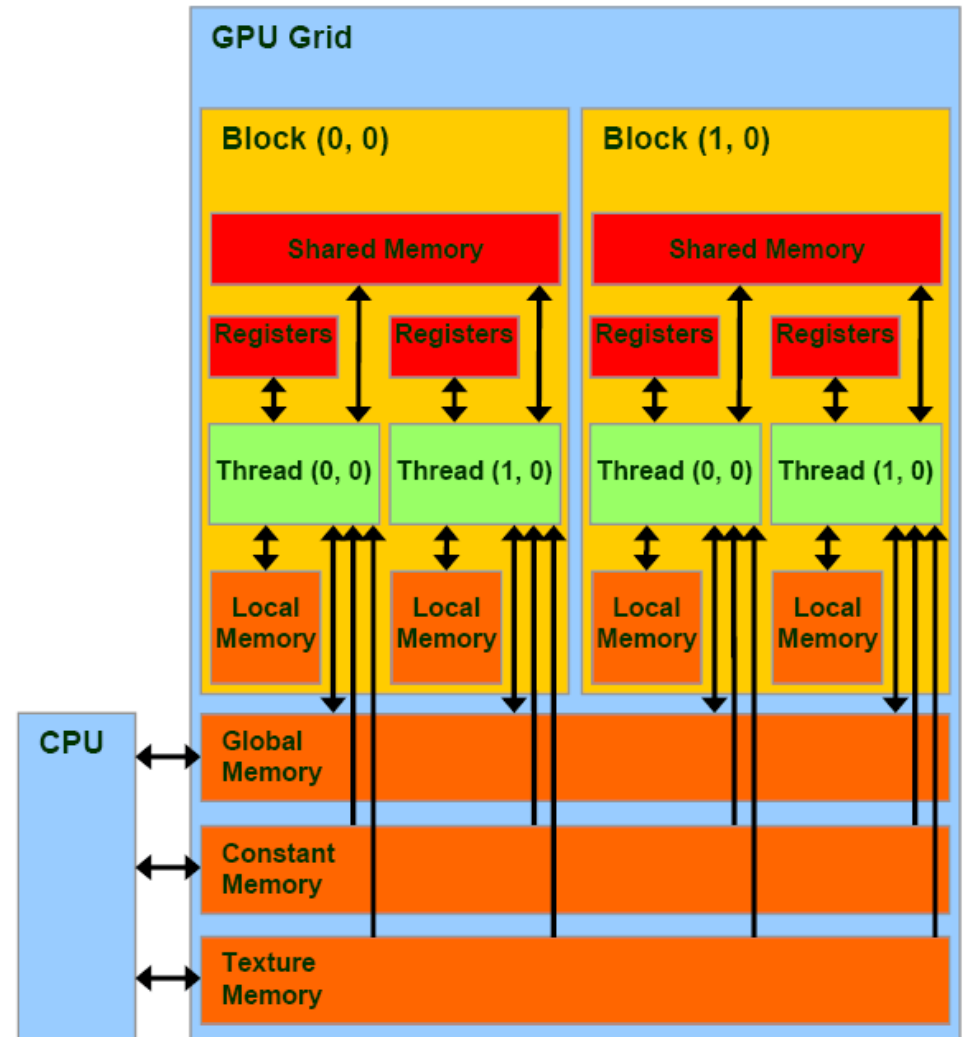
- Ejercicio en el cluster

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

Donald Knuth,
“The Art of Computer Programming”

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.



- **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.
- **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 físicamente 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

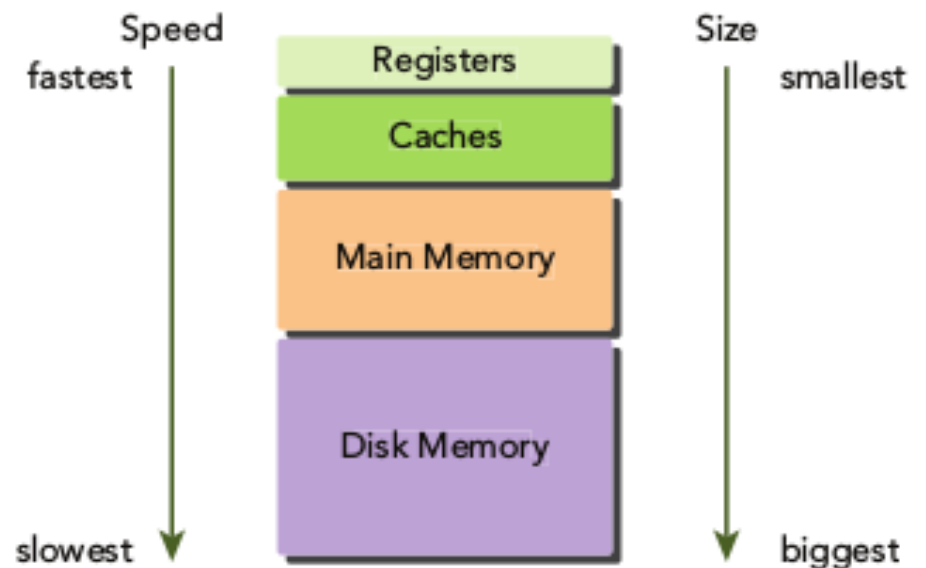
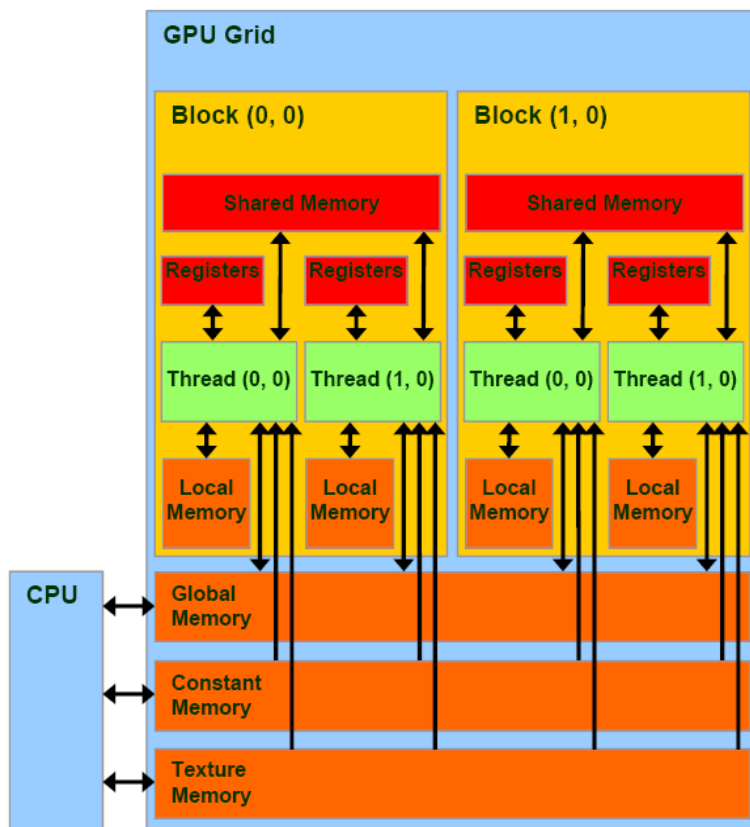


FIGURE 4-1

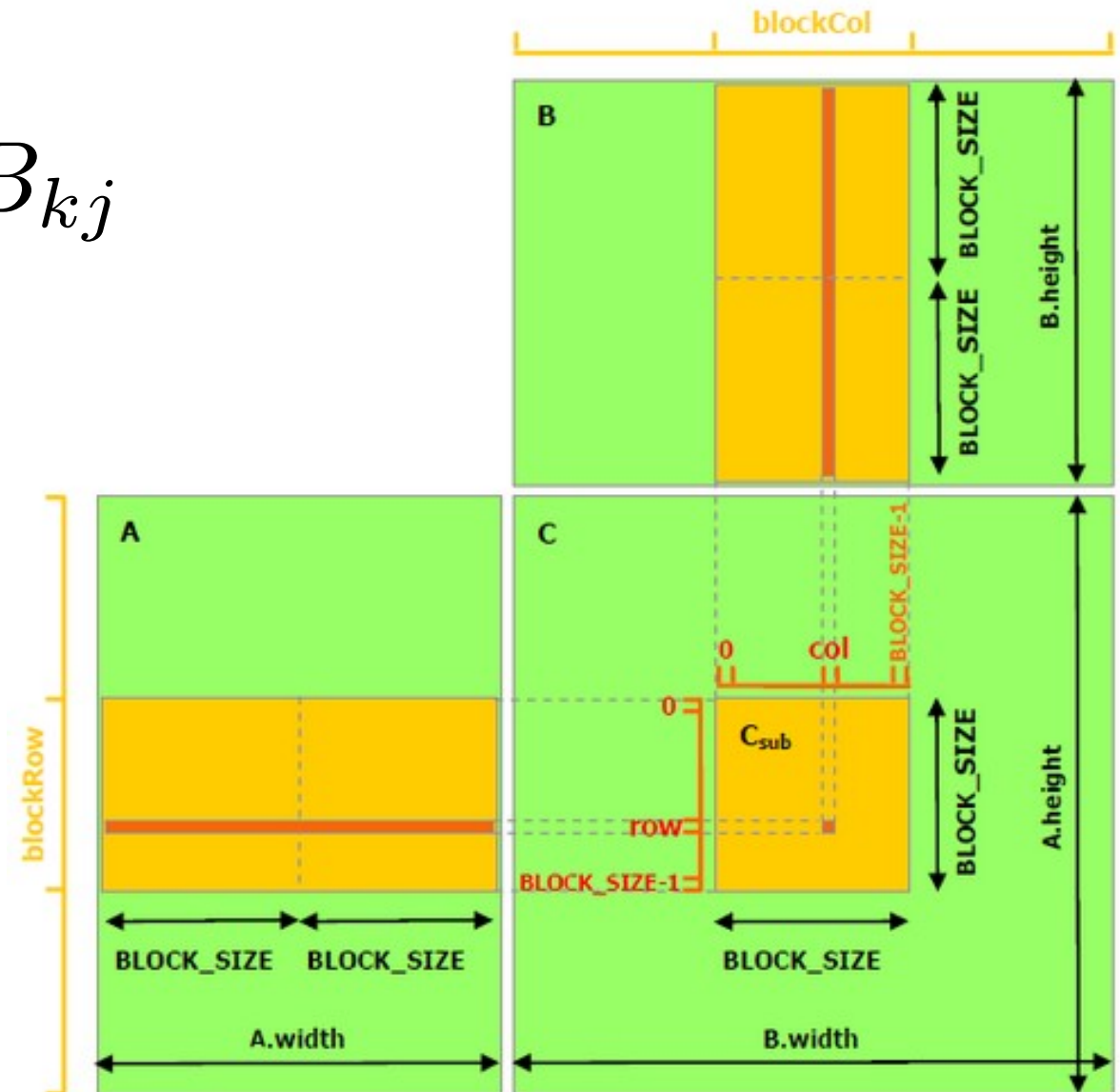
Multiplicación de matrices (optimizada #1)

Figure 10. Matrix Multiplication with Shared Memory

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

Submatriz de C → un bloque
Shared Memory

¿Porque conviene
shared-memory?



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.

Multiplicación de matrices:

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

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>

¿ Cual es preferible ?

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 ?



cuBLAS	
▷ 1. Introduction	⇒
▽ 2. Using the cuBLAS API	
▷ 2.1. General description	
▷ 2.2. cuBLAS Datatypes Reference	
▷ 2.3. CUDA Datatypes Reference	
▷ 2.4. cuBLAS Helper Function Reference	
▷ 2.5. cuBLAS Level-1 Function Reference	
▷ 2.6. cuBLAS Level-2 Function Reference	
▷ 2.7. cuBLAS Level-3 Function Reference	
▷ 2.8. BLAS-like Extension	
▷ 3. Using the cuBLASLt API	
▷ 4. Using the CUBLASXT API	

Tarea 2

- Dada una señal discreta $x[n]$ y un filtro $h[n]$, 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 , $M \ll N$.
- ¿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;  
    }  
}
```

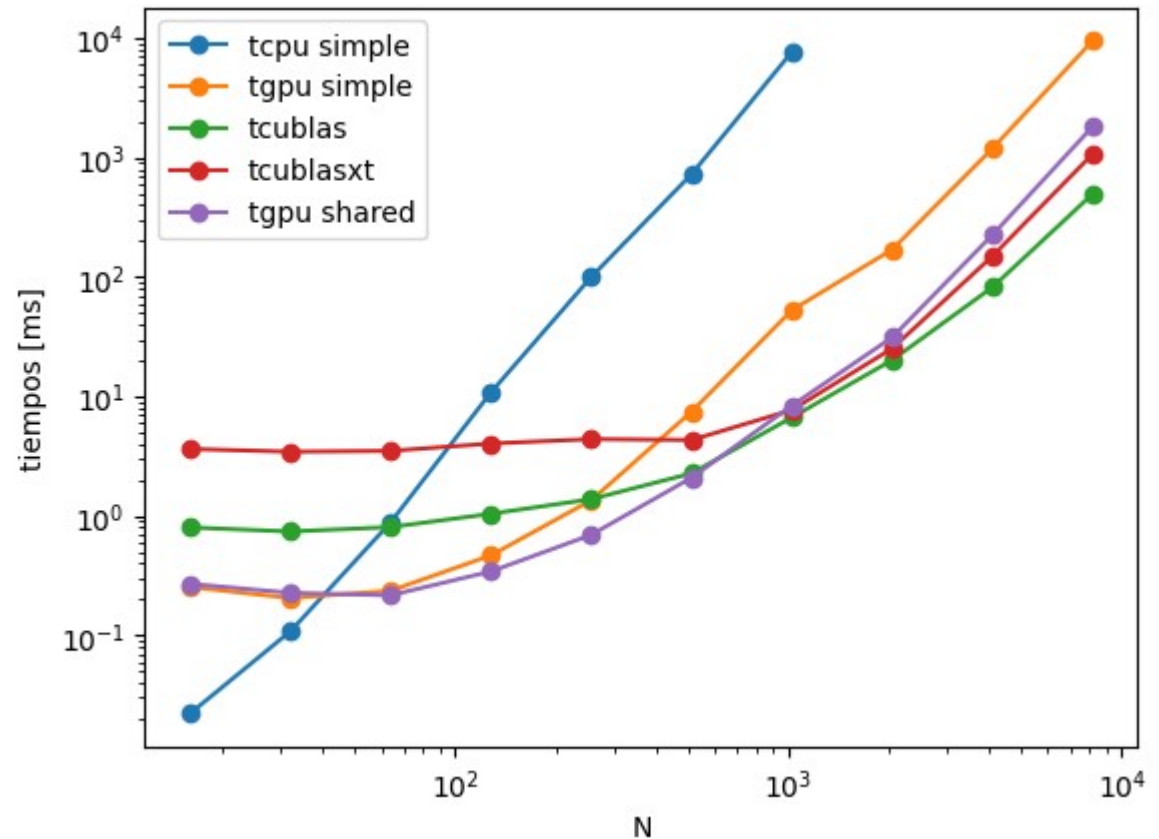
CPU para sistemas chicos, Cublas para grandes

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

```
1 #@title veamos los tiempos
2 !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?



python

numpy

```
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

```
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")
```

Cupy “raw kernels”



```
# 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)

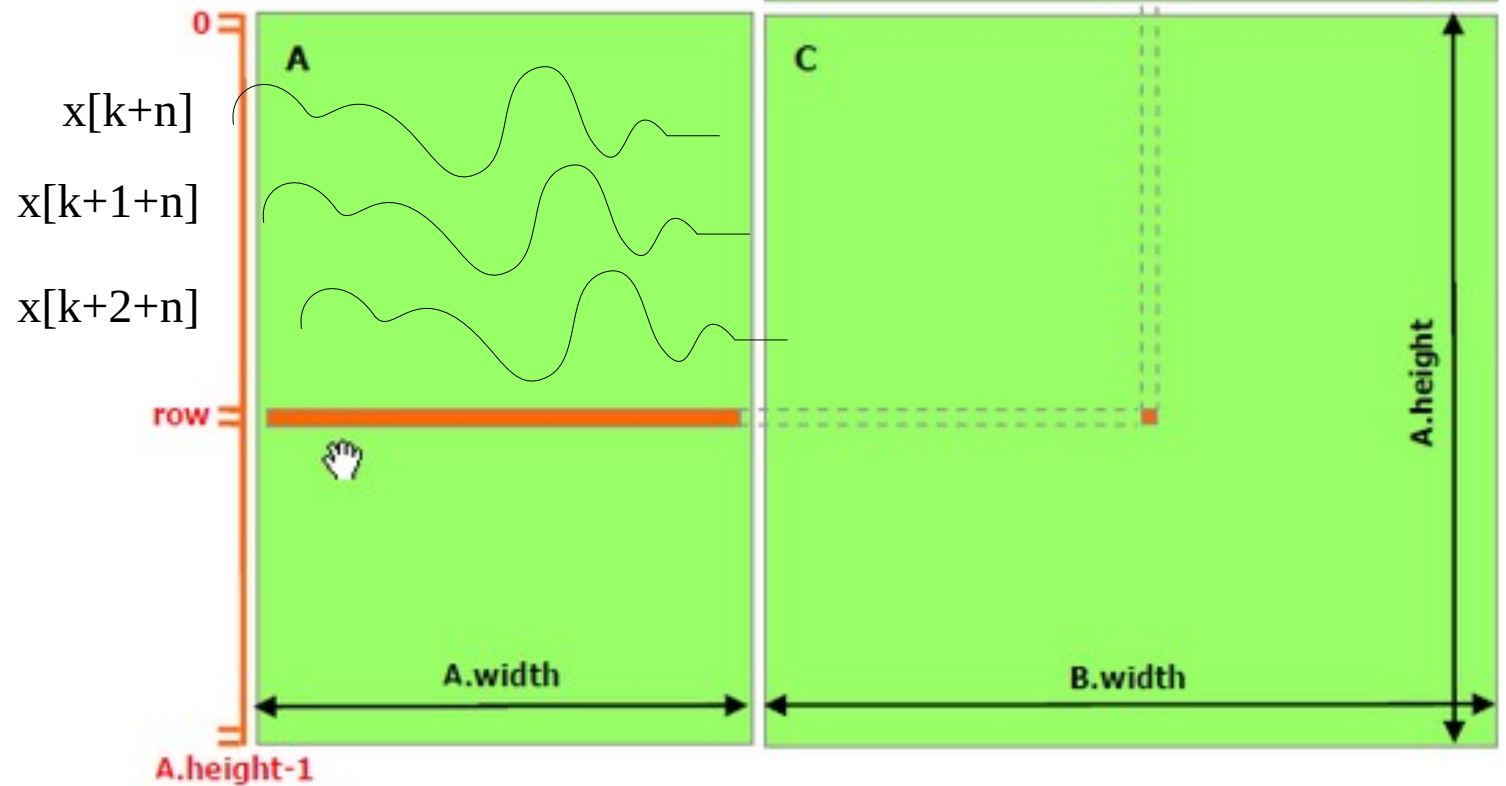
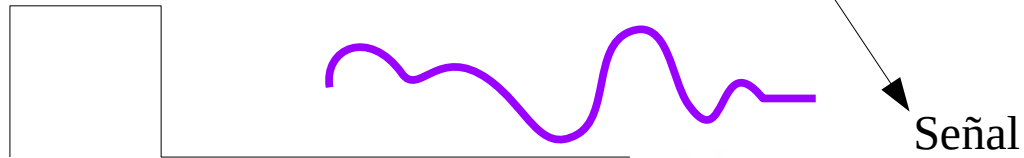
start_time = time.time()

# 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()
```

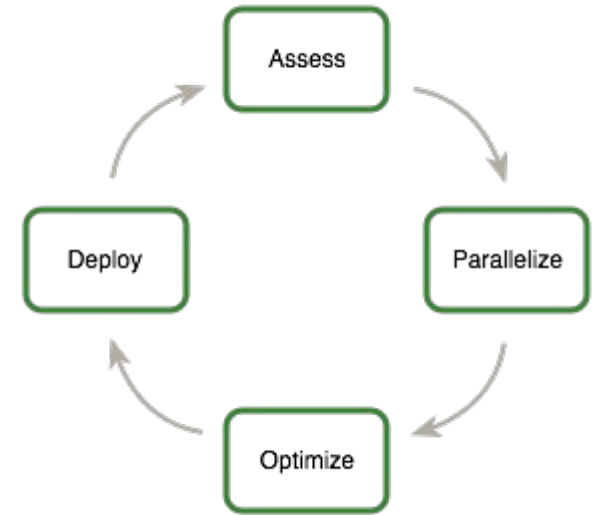
Convolución

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



Profiling

- 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();`



CUDA C Best Practices Guide

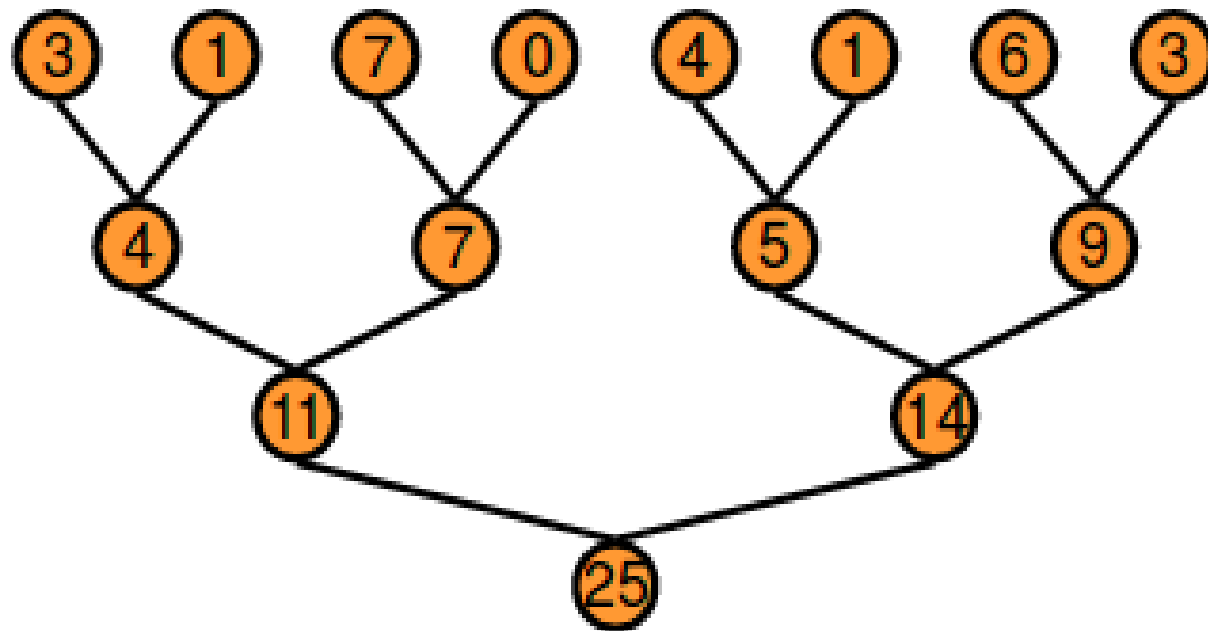
Reducción

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

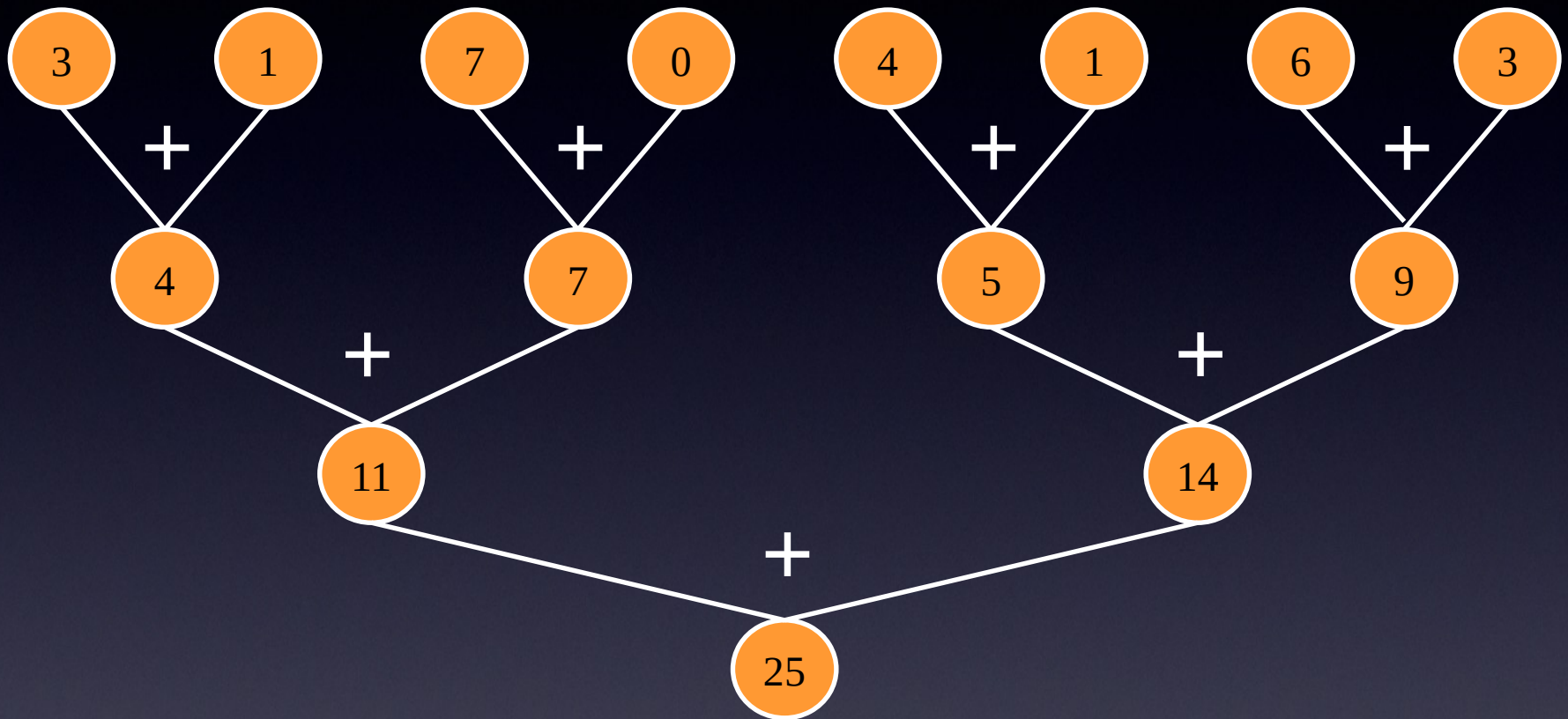
$$S = \sum_{i=0}^{N-1} x_i$$

¿Para que sirve?

Reducción paralela: “se organiza un torneo”

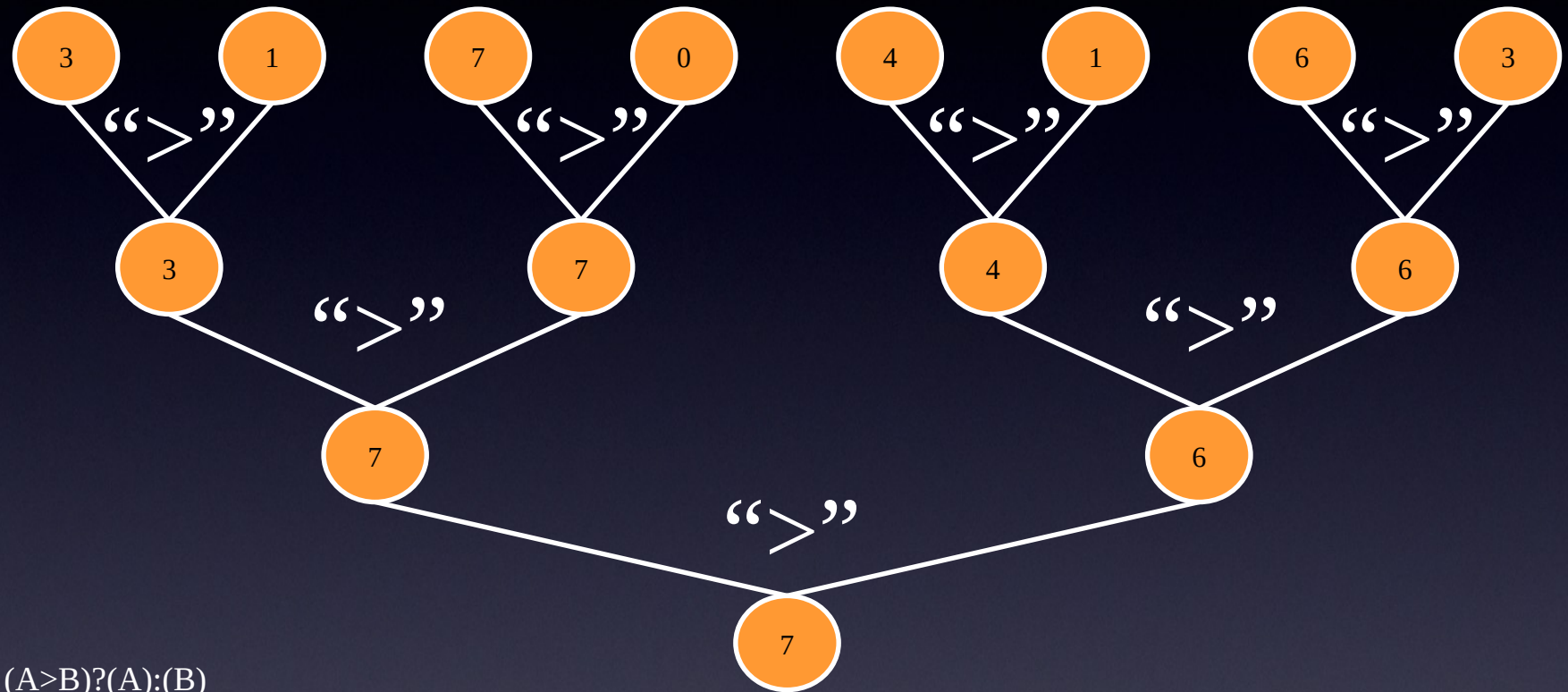


Parallel Primitives



Calcula la *suma*

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

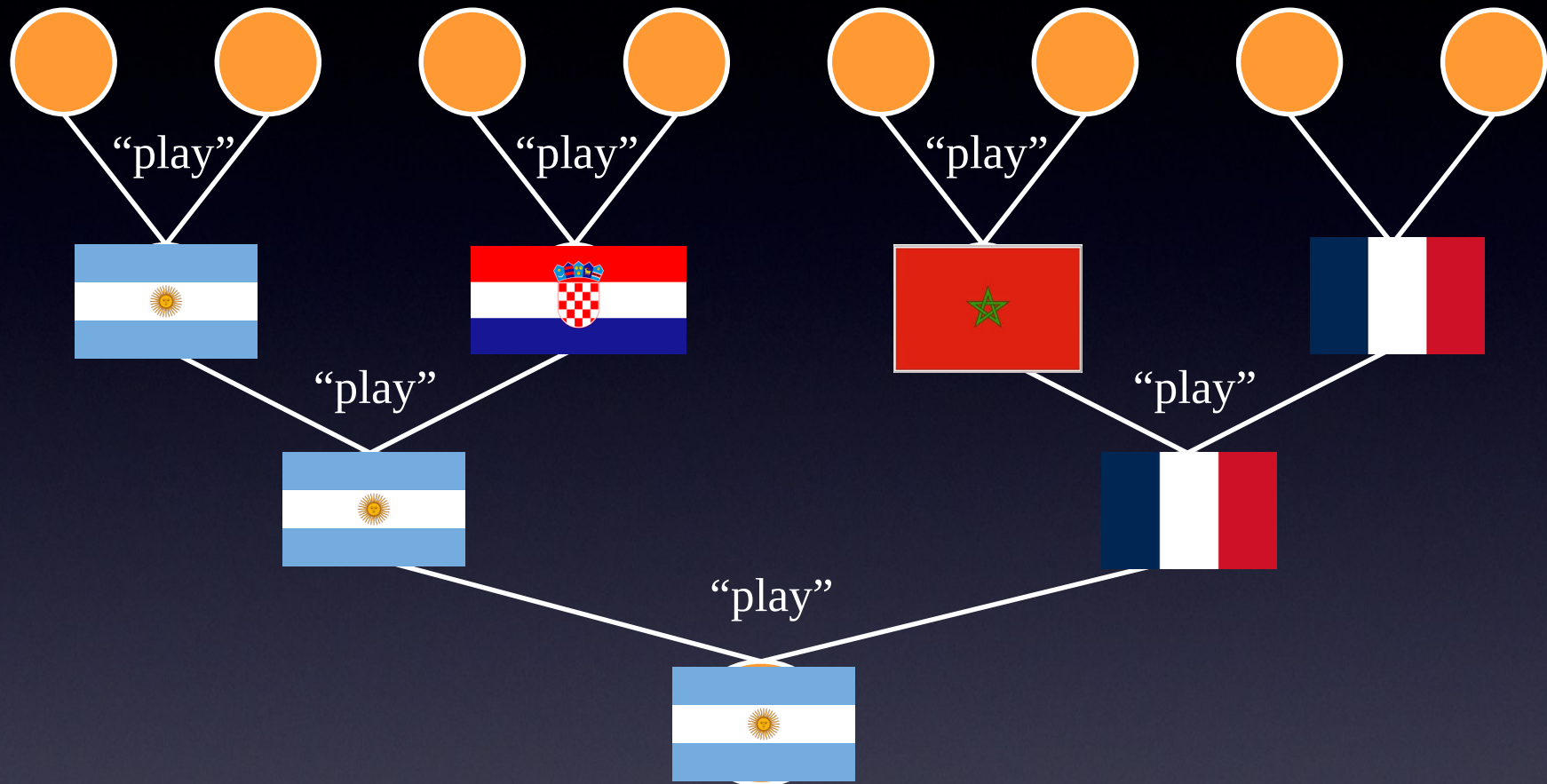


$A > B = (A > B) ? (A) : (B)$

Calcula el *máximo*

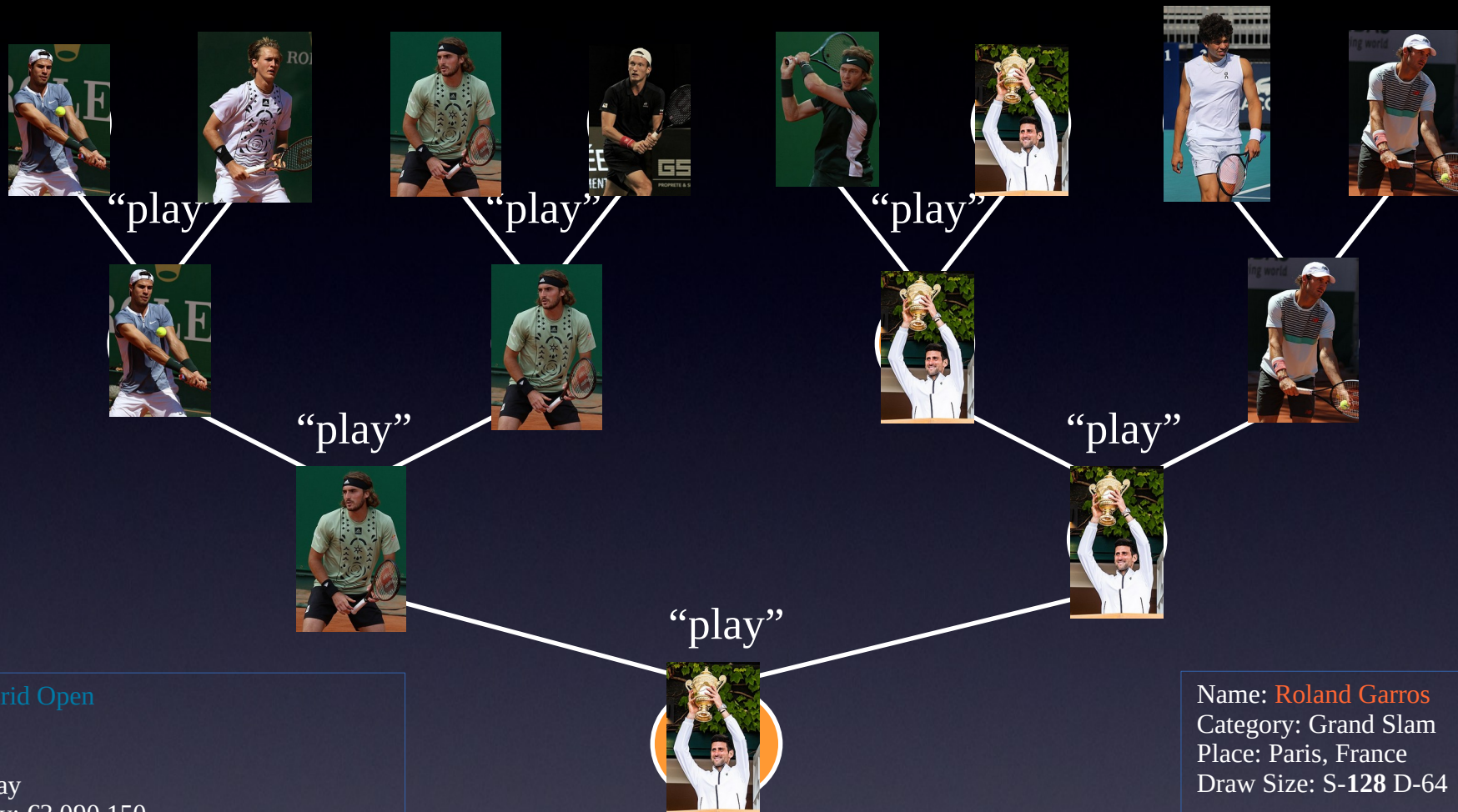
Vale para int, float, double, o cualquier tipo de dato con la operación ">" bien definida...

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



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

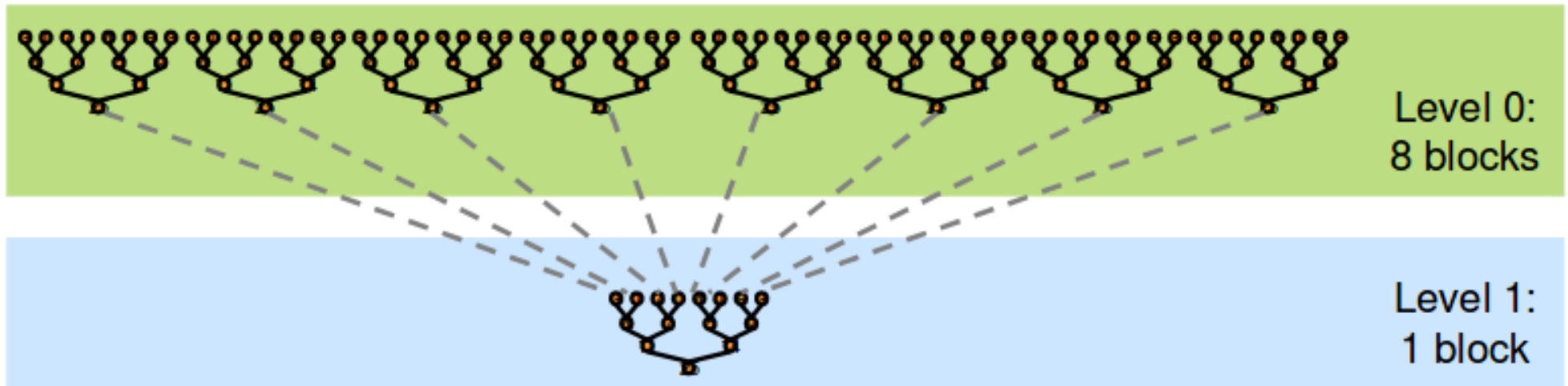
Name: Roland Garros
Category: Grand Slam
Place: Paris, France
Draw Size: S-128 D-64

Calcula el campeón

$\log_2(N)$ operaciones:

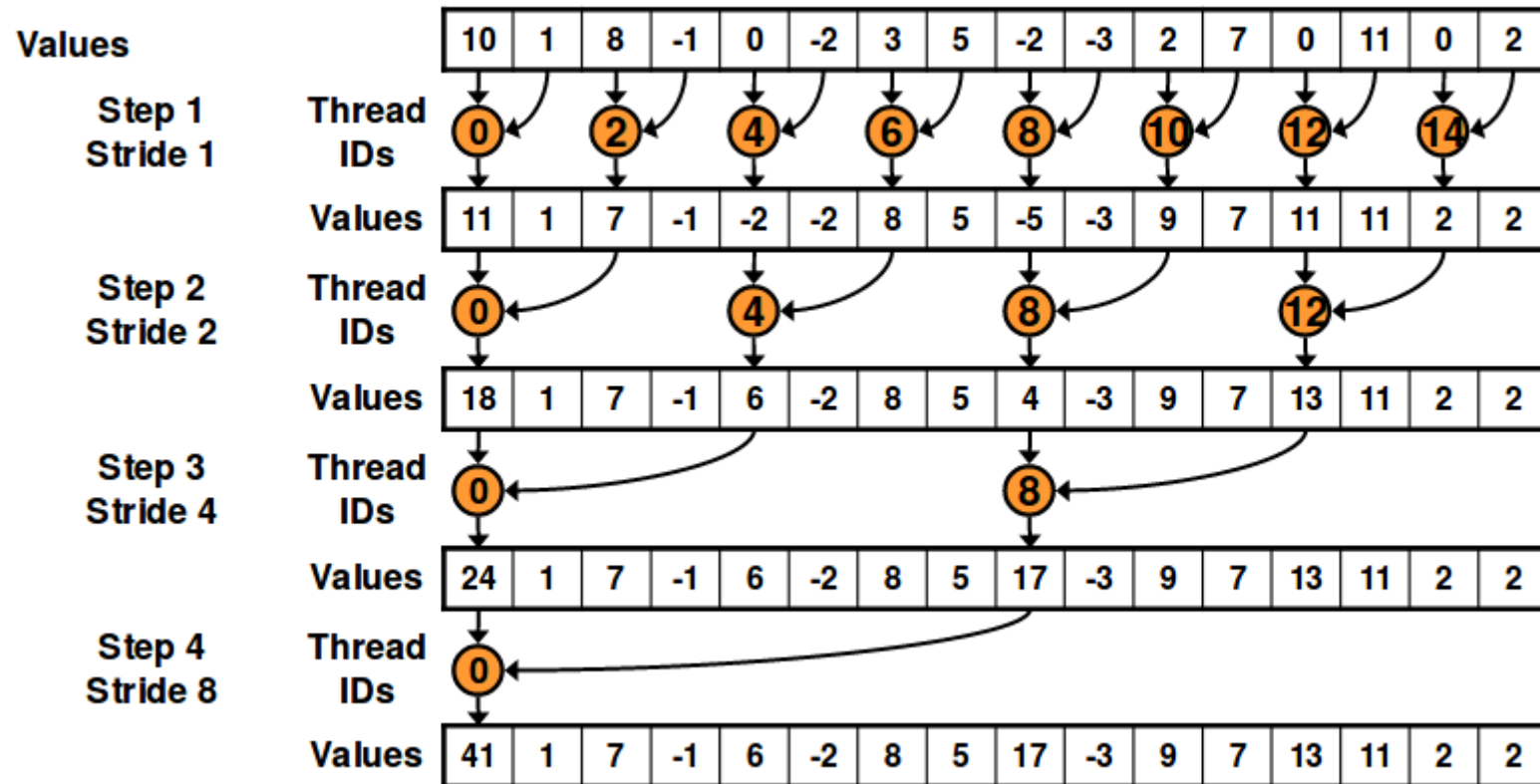
Grand-Grand-Slam: $2^{20} = 1048576$ jugadores → solamente 20 “jornadas” !!!

Reducción: N vs $\log(N)$

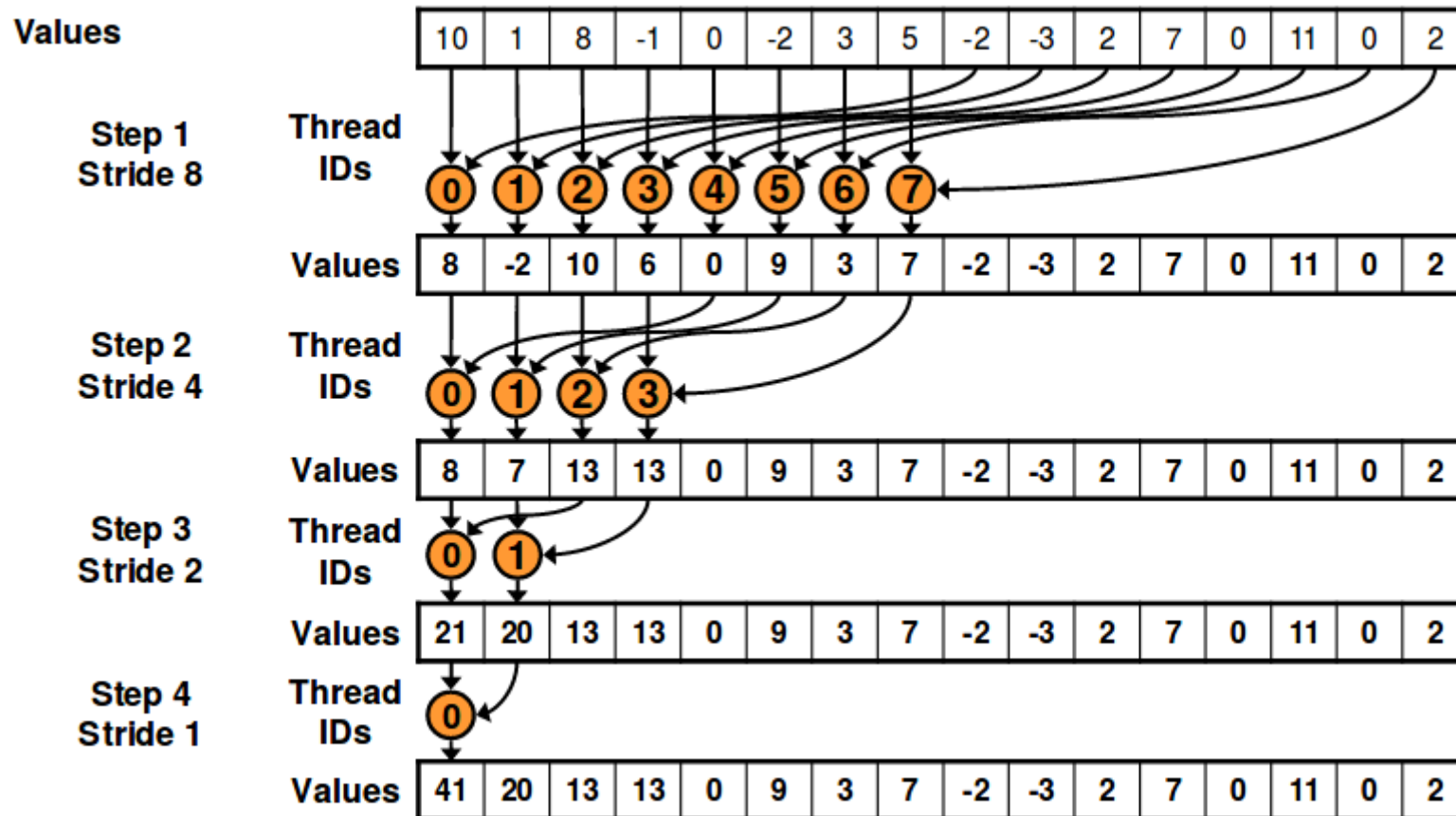


<https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf>

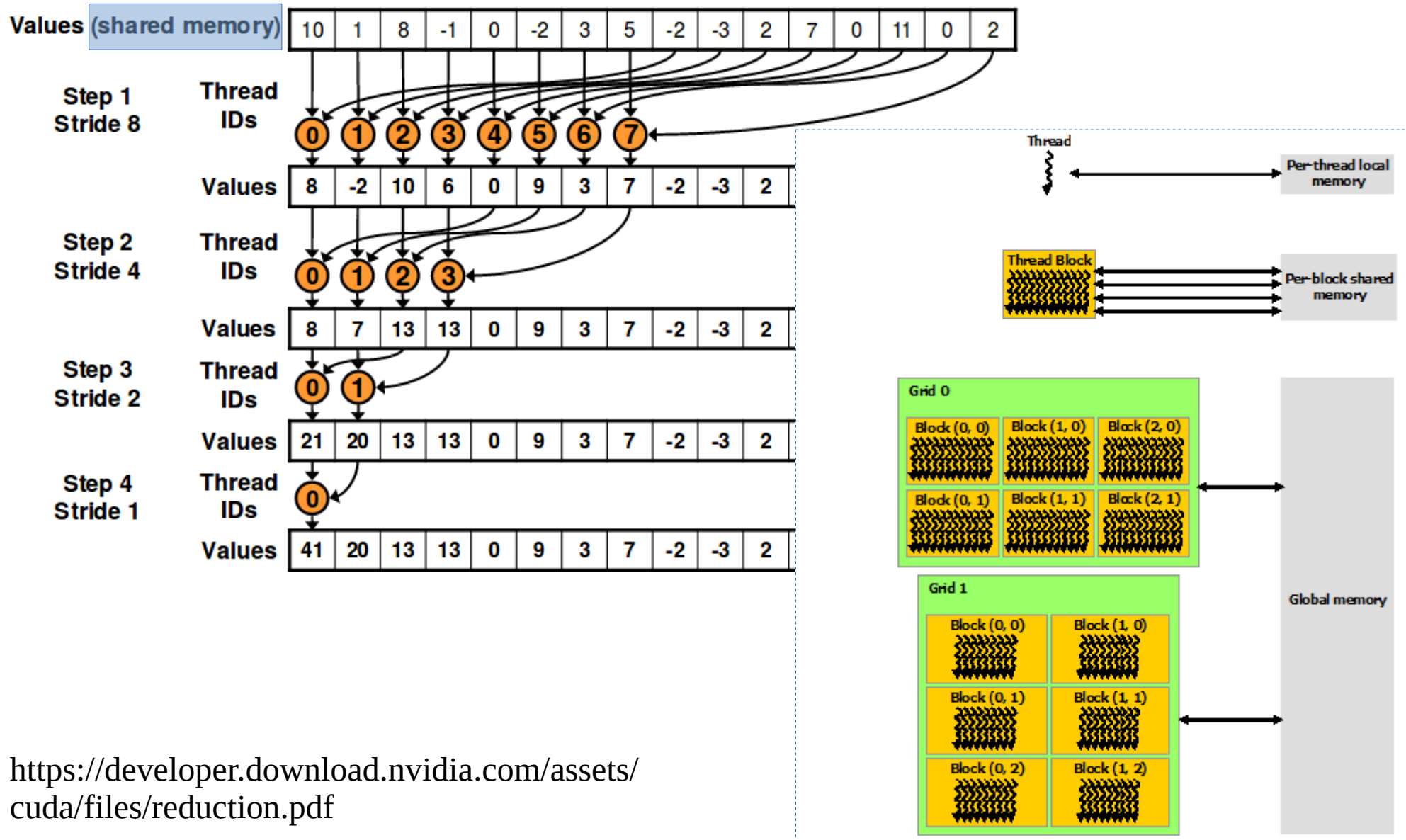
Reducción #1



Reducción #2

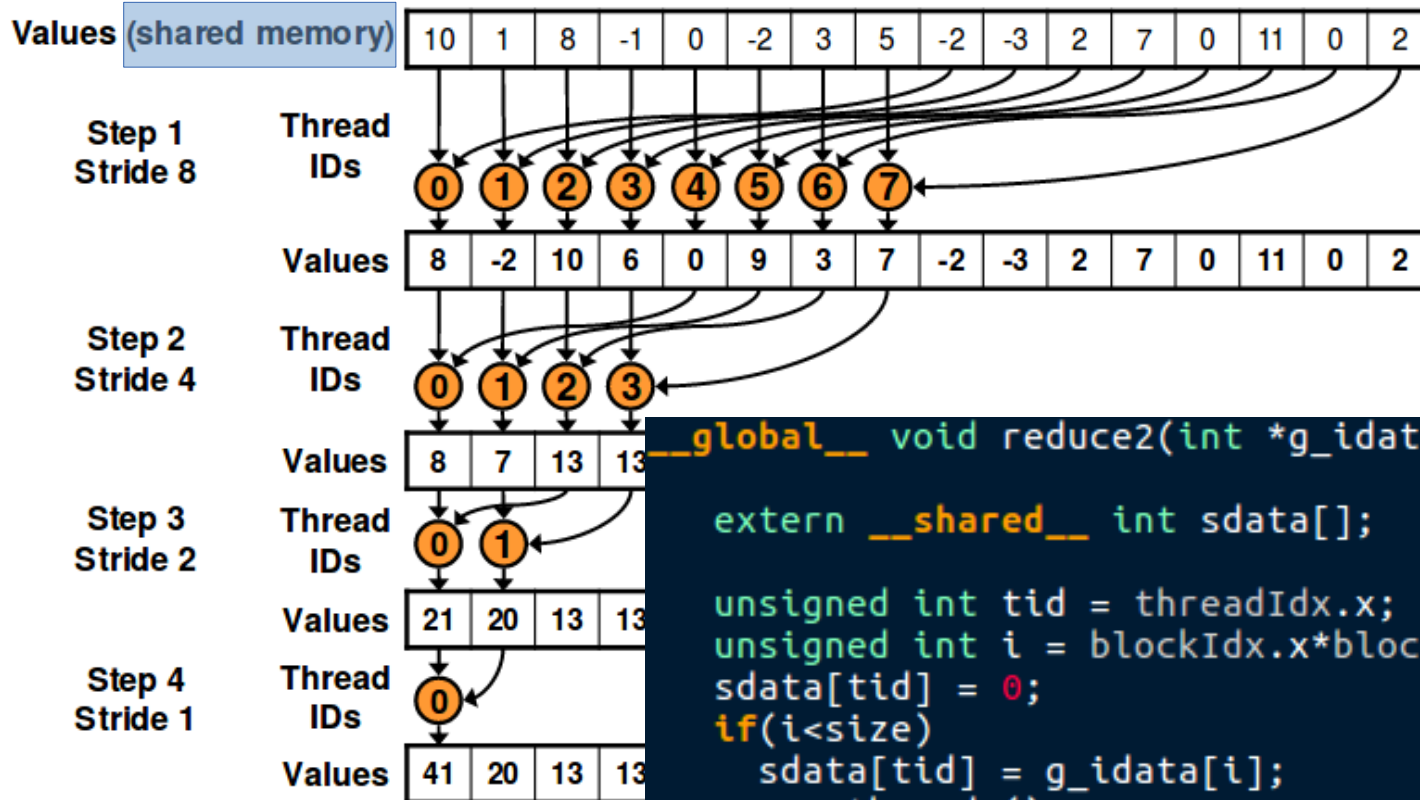


Reduction #2 (optimizado)



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

Reduction #2 (optimizado)



```
__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];
}
```

Los threads ahora
“colaboran”, cargan y
comparten Shared-memory

Reduction #2 (optimizado)

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



lanzamiento

```
__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];  
}
```

*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.