Computación Concurrente, Paralela y Distribuida

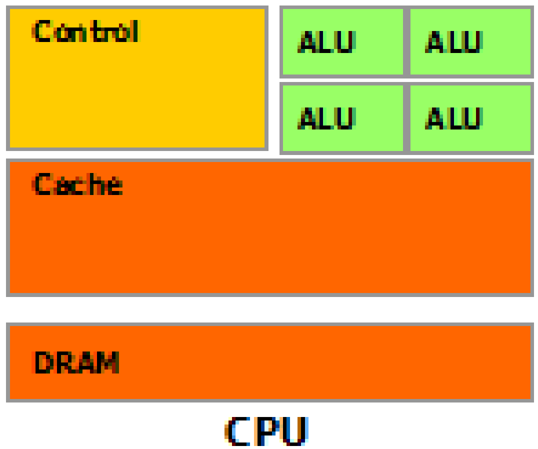
Leandro Rodríguez Liñares – David Olivieri

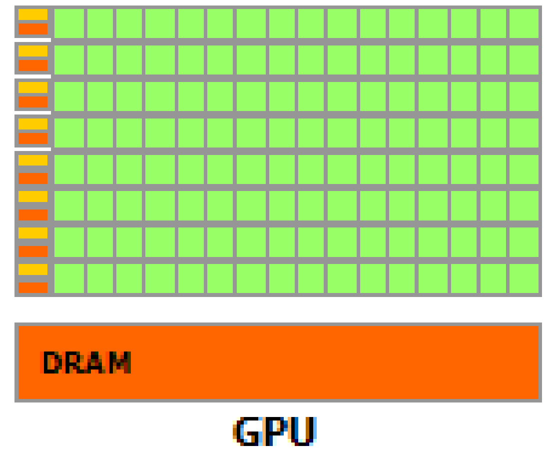
Curso 2024/25

Tema 11: Introducción a Nvidia CUDA

A ¿Qué es NVidia CUDA?

CUDA (or **Compute Unified Device Architecture**) is a parallel computing platform and application programming interface (API) that allows software to use certain types of graphics processing units (GPUs) for general purpose processing, an approach called general-purpose computing on GPUs (GPGPU). CUDA is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels.[1]



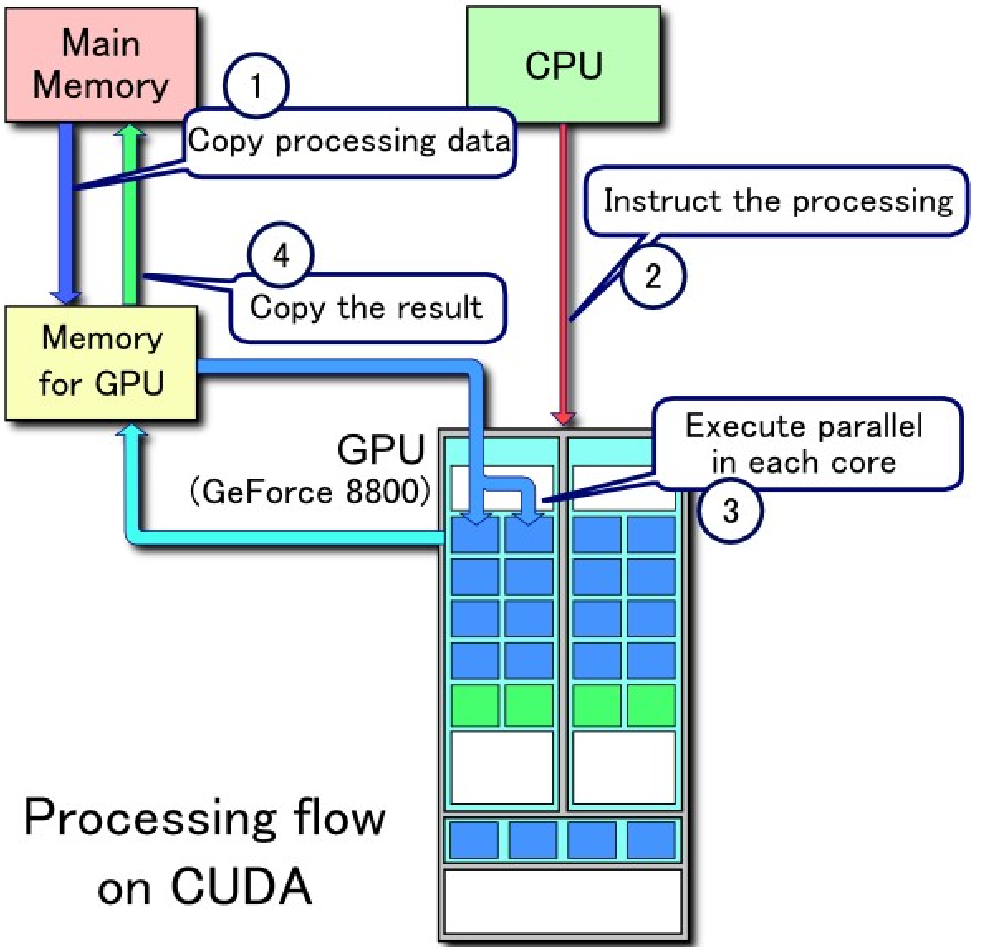


GPUs: herramientas con alta paralelización, multihilo, con numerosos núcleos, con altísima capacidad de cálculo y con alto ancho de banda de acceso a los datos.

GPUs: especializadas en computación intensiva y altamente paralela (necesario para renderizado de gráficos): memorias caché reducidas y poca capacidad de control de flujo.

NVidia CUDA usa un modelo de programación basado en ***offloading***:

1) Los datos se copian de la memoria principal a la memoria de la GPU
2) La CPU ordena el procesamiento en la GPU
3) La GPU procesa los datos de acuerdo a las órdenes de la CPU
4) Los resultados se copian de la memoria de la GPU a la memoria principal



¿Porqué CUDA?

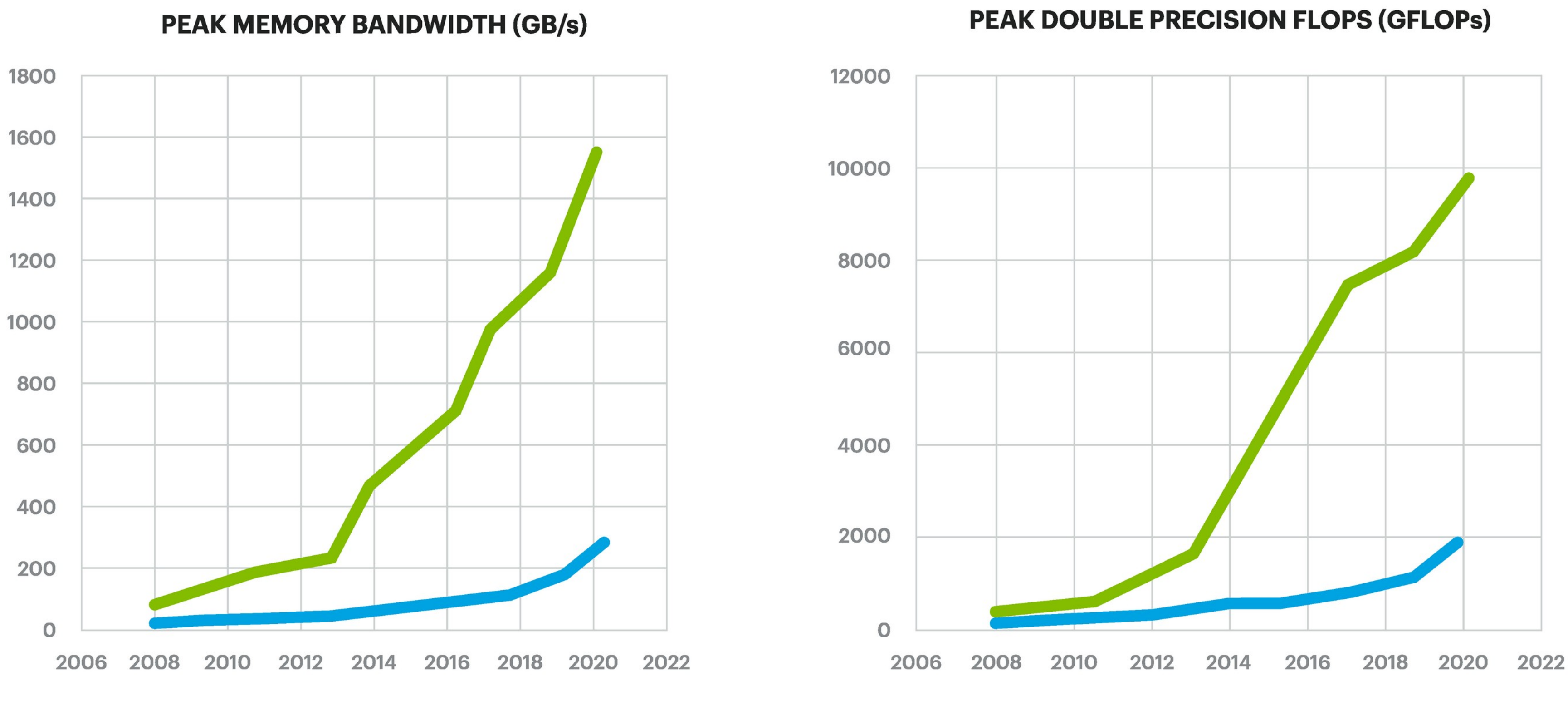
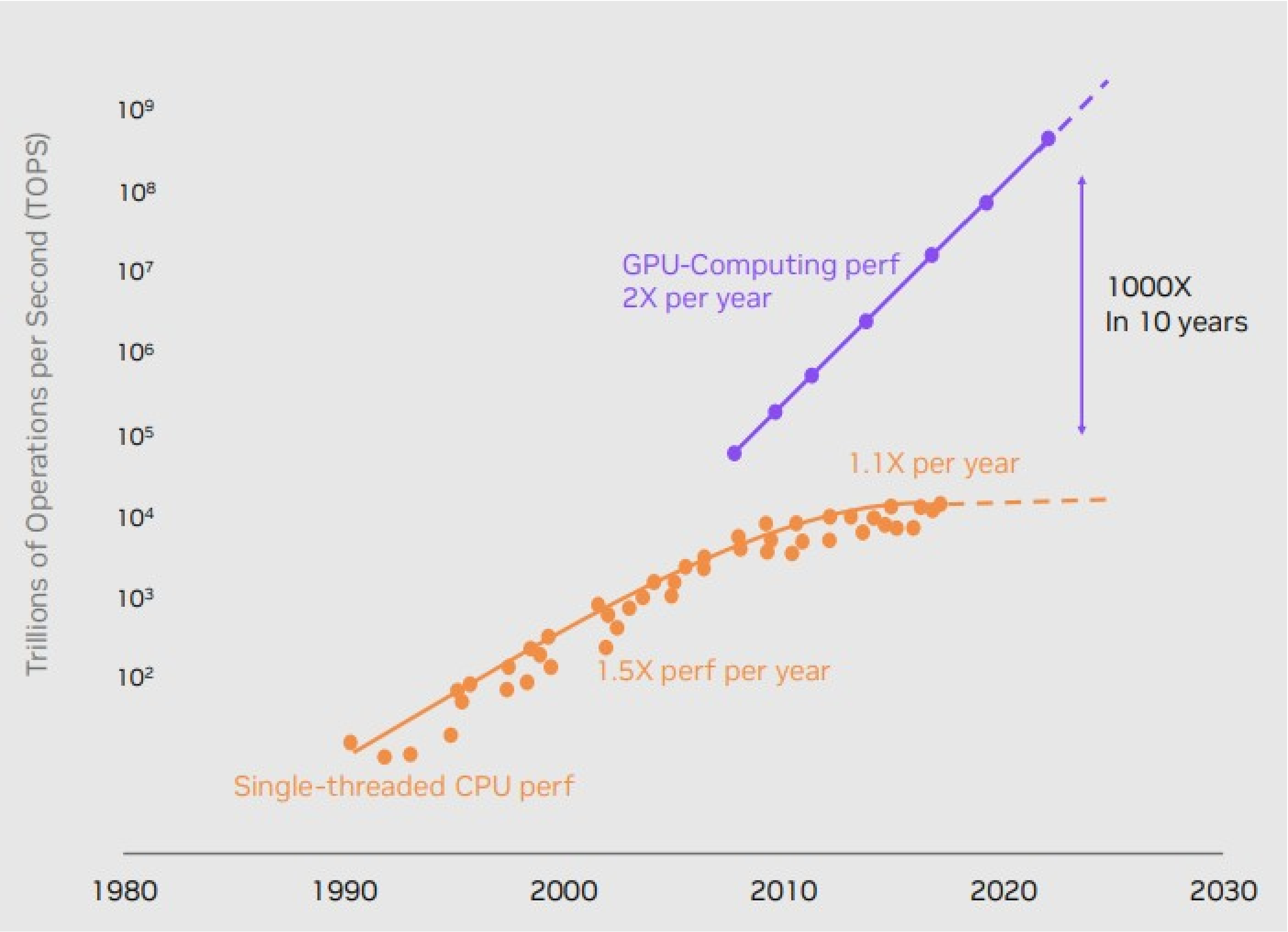
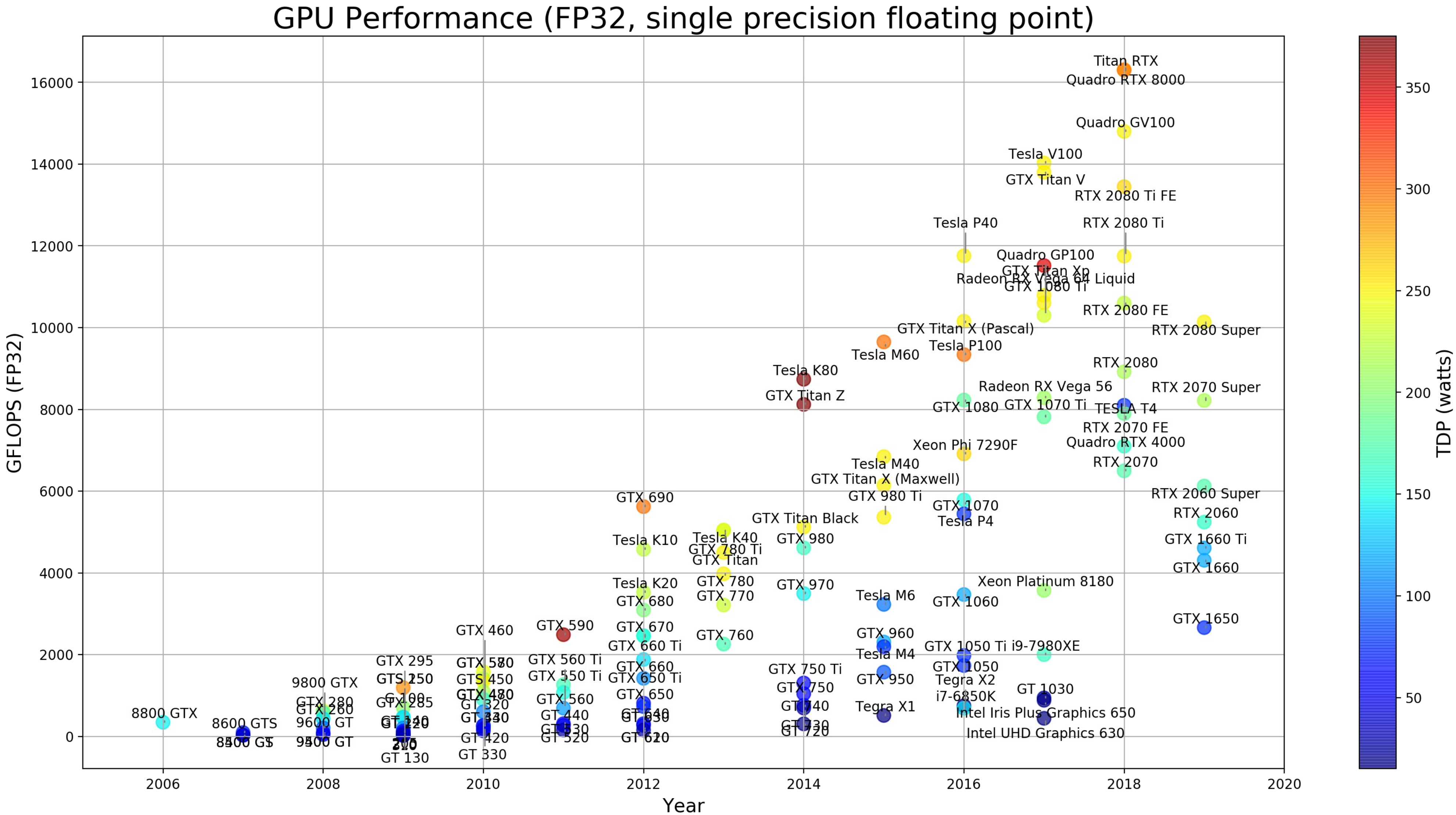


Figure 2. Comparison of evolution of memory bandwidth (left) and double precision flops (right) on GPU and CPU

https://s201.q4cdn.com/141608511/files/doc_presentations/2023/02/nvda-f4q23-investor-presentation-final.pdf



https://blog.inten.to/hardware-for-deep-learning-part-3-gpu-8906c1644664



Noviembre 2024

TOP 500 The List.

https://www.top500.org/

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**El Capitan** - HPE Cray EX255a, AMD 4th Gen EPYC 24C 1.8GHz, AMD Instinct MI300A, Slingshot-11, TOSS, HPE DOE/NNSA/LLNL United States	11,039,616	1,742.00	2,746.38	29,581
2	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE Cray OS, HPE DOE/SC/Oak Ridge National Laboratory United States	9,066,176	1,353.00	2,055.72	24,607
3	**Aurora** - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, Intel DOE/SC/Argonne National Laboratory United States	9,264,128	1,012.00	1,980.01	38,698
4	**Eagle** - Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA H100, NVIDIA Infiniband NDR, Microsoft Azure Microsoft Azure United States	2,073,600	561.20	846.84	
5	**HPC6** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, RHEL 8.9, HPE Eni S.p.A. Italy	3,143,520	477.90	606.97	8,461
6	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899

Nuevo

Nº 1 en 2023

Nº 2 en 2023

Nº 3 en 2023

Nuevo

Nº 4 en 2023

7	**Alps** - HPE Cray EX254n, NVIDIA Grace 72C 3.1GHz, NVIDIA GH200 Superchip, Slingshot-11, HPE Cray OS, HPE Swiss National Supercomputing Centre (CSCS) Switzerland	2,121,600	434.90	574.84	7,124
8	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	2,752,704	379.70	531.51	7,107
9	**Leonardo** - BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, NVIDIA A100 SXM4 64 GB, Quad-rail NVIDIA HDR100 Infiniband, EVIDEN EuroHPC/CINECA Italy	1,824,768	241.20	306.31	7,494
10	**Tuolumne** - HPE Cray EX255a, AMD 4th Gen EPYC 24C 1.8GHz, AMD Instinct MI300A, Slingshot-11, TOSS, HPE DOE/NNSA/LLNL United States	1,161,216	208.10	288.88	3,387
11	**MareNostrum 5 ACC** - BullSequana XH3000, Xeon Platinum 8460Y+ 32C 2.3GHz, NVIDIA H100 64GB, Infiniband NDR, EVIDEN EuroHPC/BSC Spain	663,040	175.30	249.44	4,159

2024

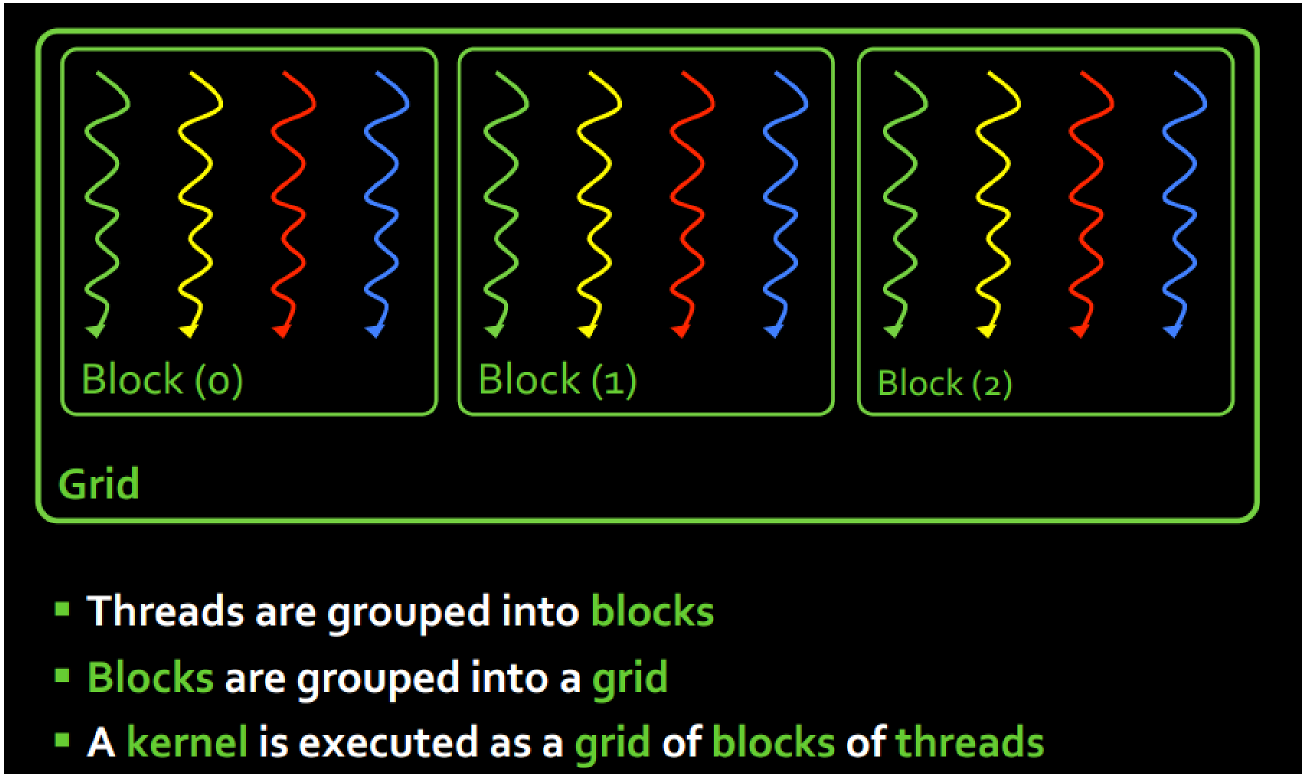
8	**MareNostrum 5 ACC** - BullSequana XH3000, Xeon Platinum 8460Y+ 40C 2.3GHz, NVIDIA H100 64GB, Infiniband NDR200, EVIDEN EuroHPC/BSC Spain	680,960	138.20	265.57	2,560

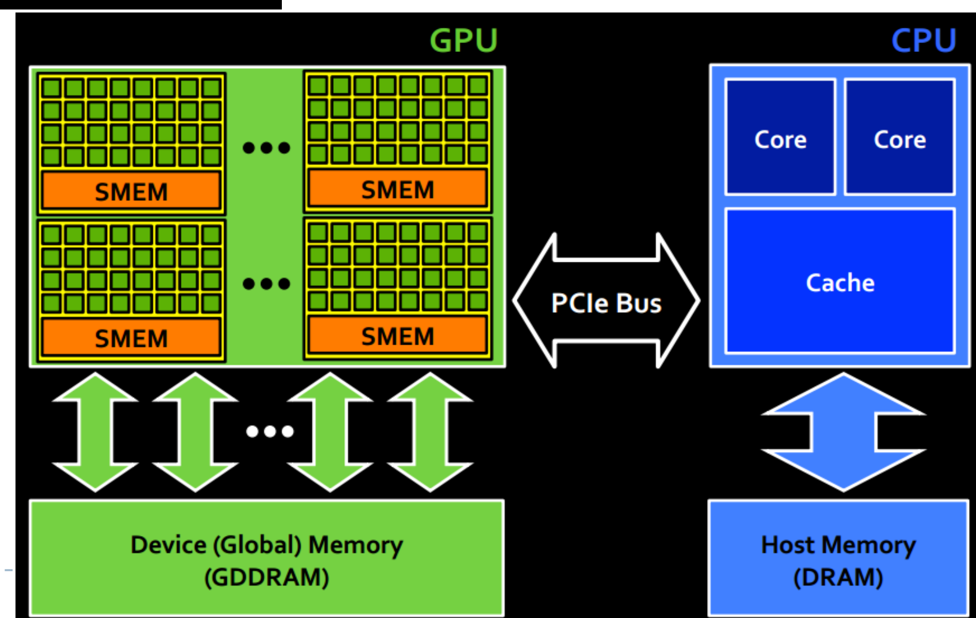
2023

88	**MareNostrum** - Lenovo SD530, Xeon Platinum 8160 24C 2.1GHz, Intel Omni-Path, Lenovo Barcelona Supercomputing Center Spain	153,216	6.47	10.30	1,632

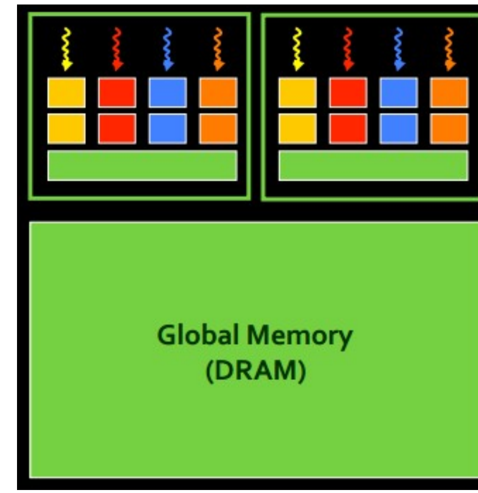
2022

B Modelo de ejecución CUDA



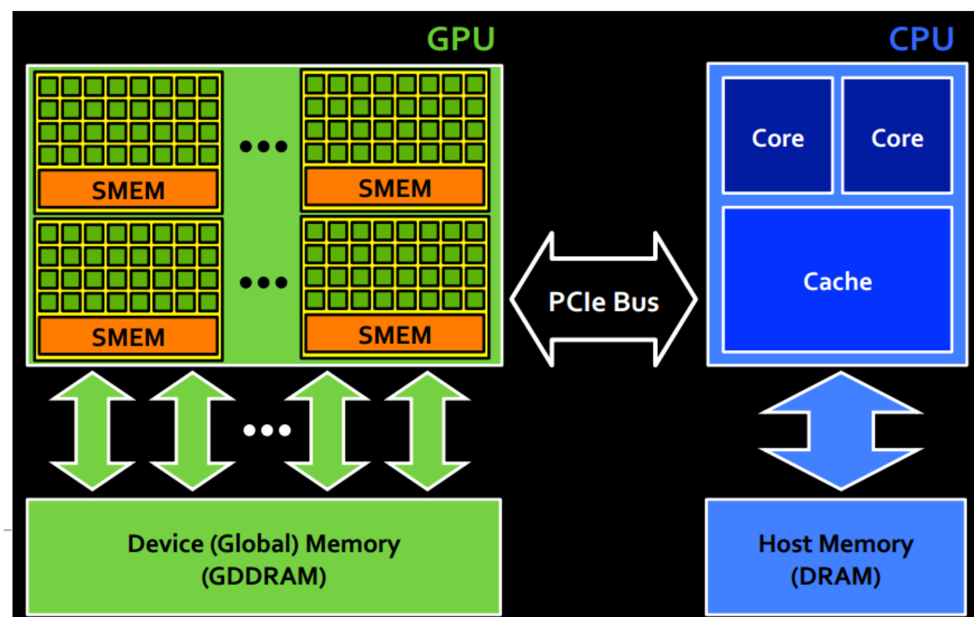


- A thread executes on a single streaming processor
  - Allows use of familiar scalar code within kernel
- A block executes on a single streaming multiprocessor
  - Threads and blocks do not migrate to different SMs
  - All threads within block execute in concurrently, in parallel
- A streaming multiprocessor may execute multiple blocks
  - Must be able to satisfy aggregate register and memory demands
- A grid executes on a single device (GPU)
  - Blocks from the same grid may execute concurrently or serially*
  - Blocks from multiple grids may execute concurrently
  - A device can execute multiple kernels concurrently



En la GPU tenemos varios tipos de memoria:

- **Memoria global**
- **Memoria compartida**
- Registros
- Memoria local
- Memoria de constantes
- Memoria de texturas



Memoria Global

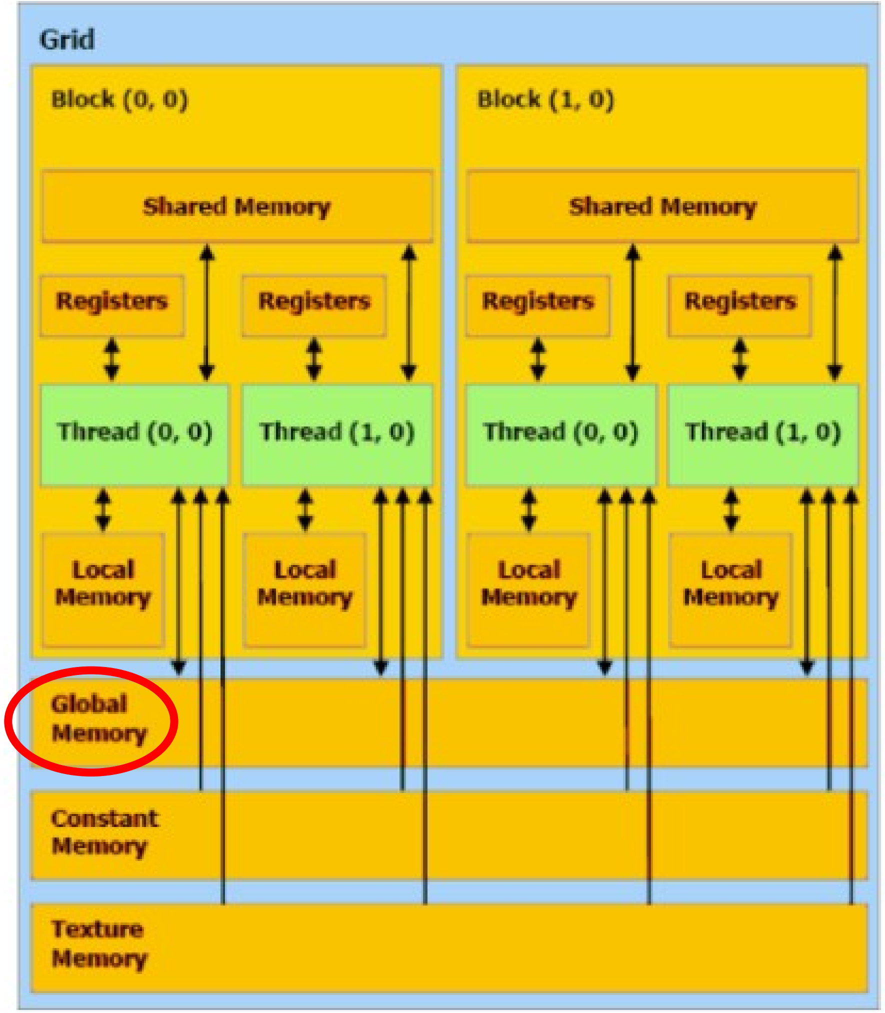
- Es la memoria “principal” de la GPU. Todos los hilos pueden acceder a ella
- La memoria permanece reservada mientras se ejecuta el kernel que la reservó o se libera explícitamente
- Reserva:

```
m = cuda.device_array(n, dtype=np.float32)
```

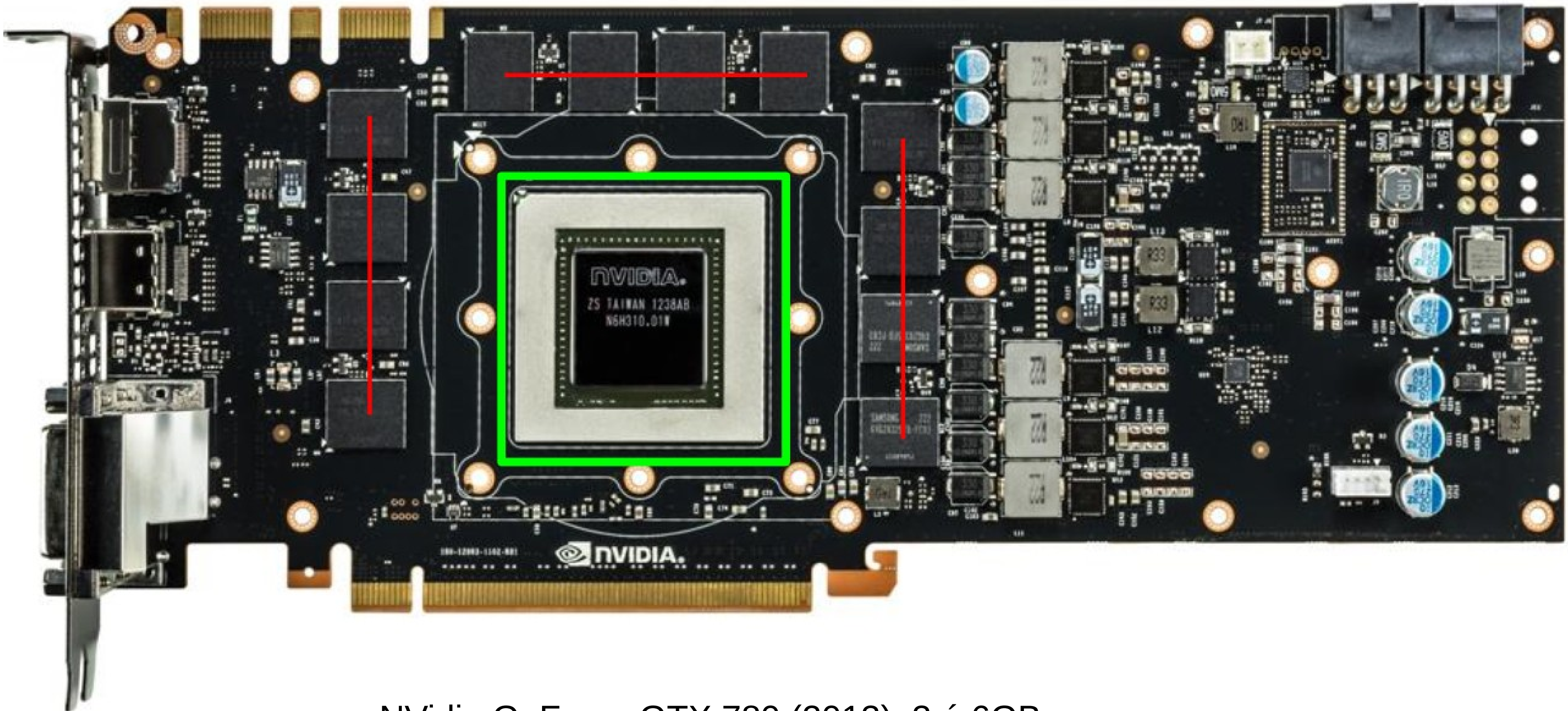
- Liberación:

```
m.close()
```

- La mayoría de la memoria de las GPUs es memoria global
- La latencia en torno a 300 ns para GPUs Kepler GTX 700 (la latencia de la memoria principal de la CPU ≥ 100 ns)
- Potencialmente 150 veces más lenta que la memoria compartida o los registros
- Tamaño: hasta 48-64 GBs



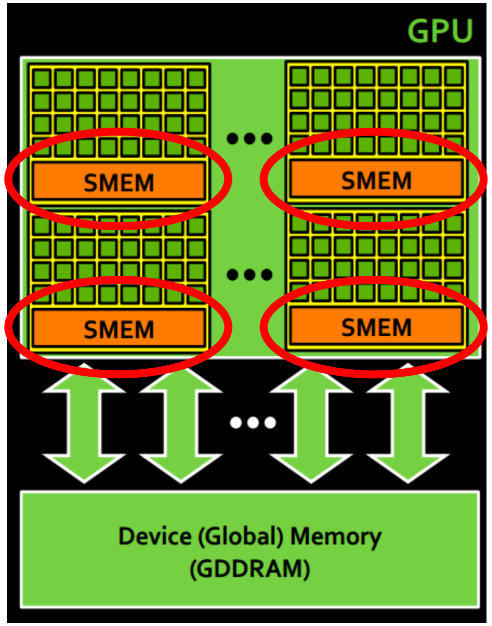
https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units



NVidia GeForce GTX 780 (2013): 3 ó 6GB

Las líneas rojas marcan la memoria global

Memoria Compartida



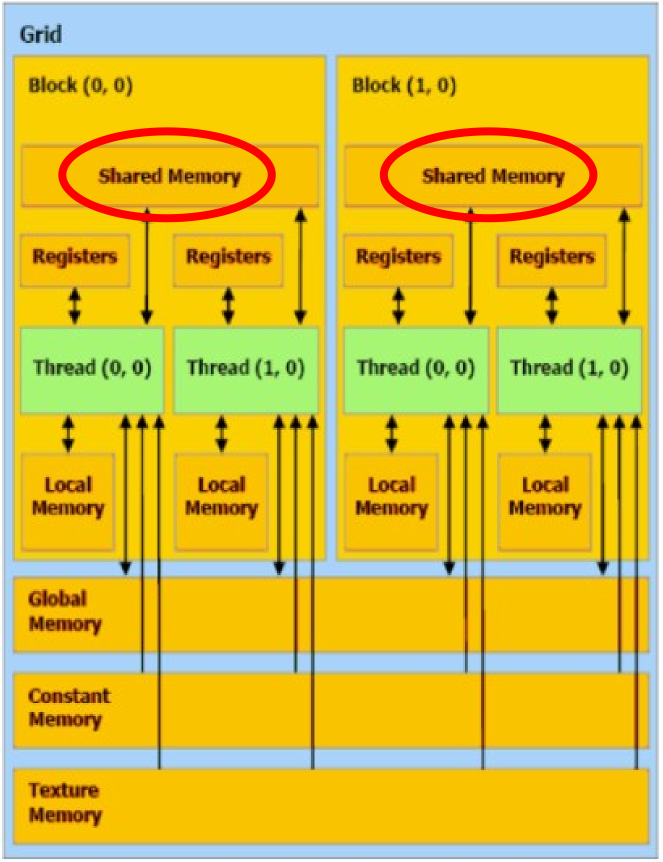
- Memoria muy rápida localizada en los SMs
- Mismo hardware que el caché L1
- Declarada estáticamente <u>en el kernel</u>:

```
sm = cuda.shared.array(128, dtype=numba.float32)
```

- Declarada dinámicamente en host y kernel:

```
kernel[grid_dim, block_dim, numBytesShMem](args)
```

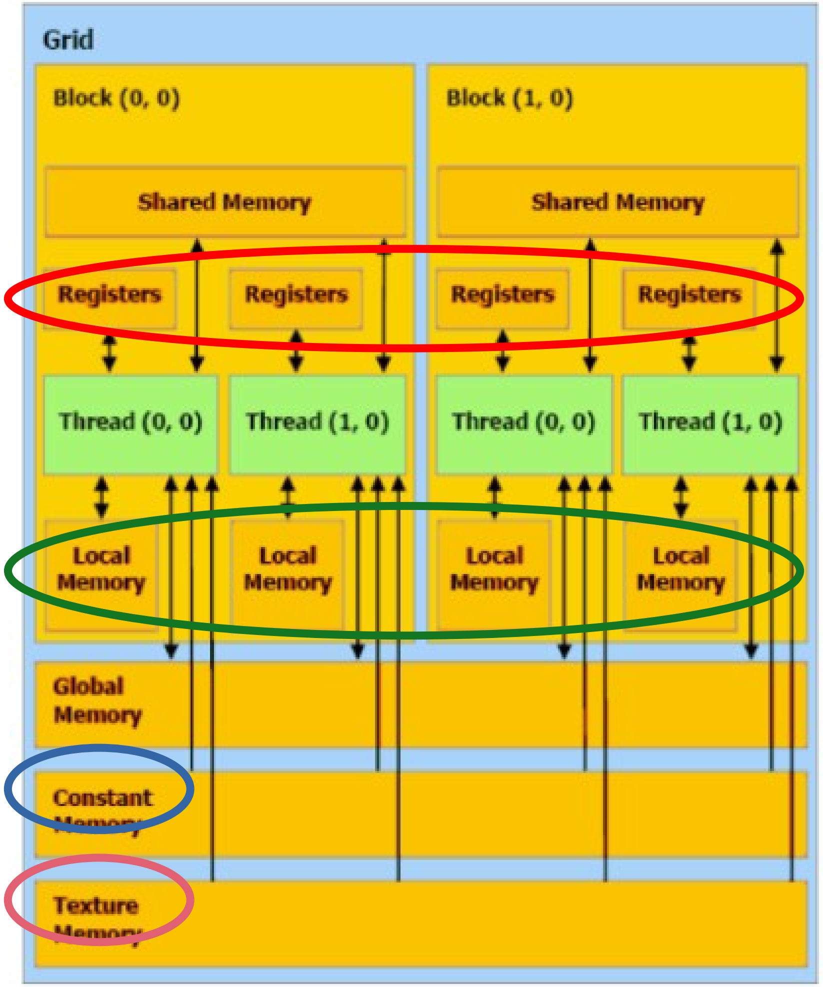
- Latencia en torno a 5 ns
- Tamaño: 48 – 128 kiB. Reparto entre caché L1 y memoria compartida es configurable



https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units

Memoria para constantes (constant memory)

- Memoria que es utilizada desde el host
- Visible (read-only) para todos los hilos del kernel
- Tamaño constante: 64 KB
- Uso típico: almacenar constantes



Registros

- Variables locales de los threads
- Hasta 128K / 32 bits cada uno

Memoria local

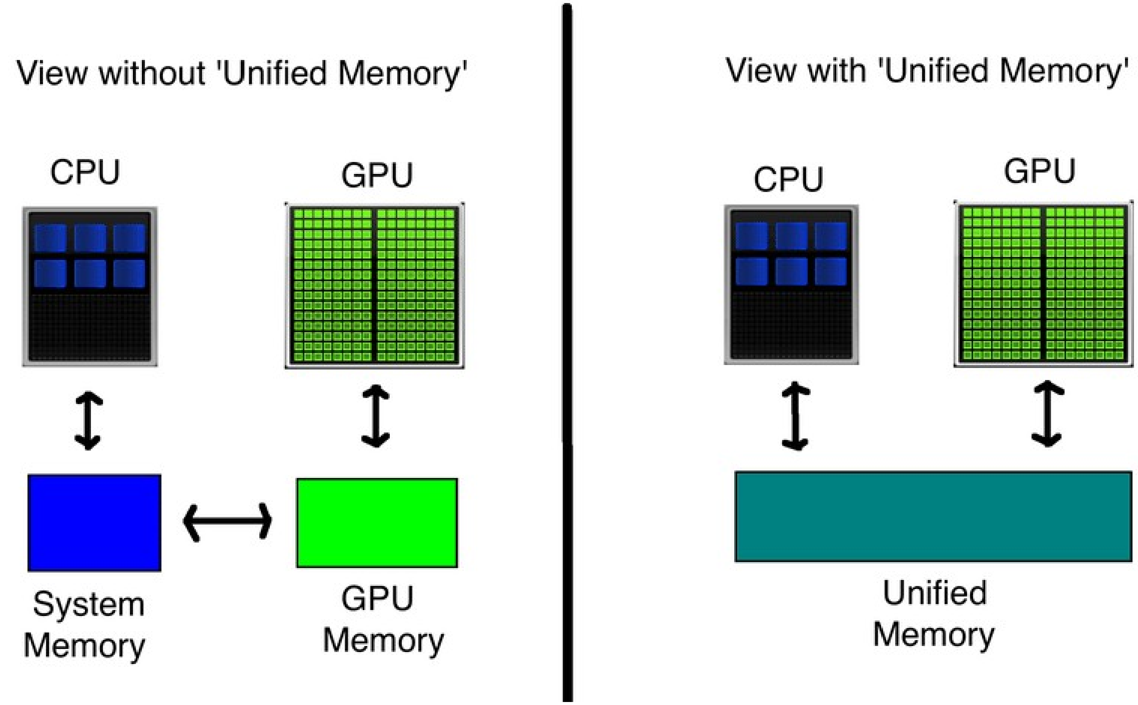
- Si se acaban los registros, se reserva un espacio en memoria global

Texture memory

- Parte de la memoria global. Se obtiene localidad espacial mediante cachés. Ideal para imágenes. Es de acceso read-only desde el kernel

Memoria unificada

- Unifica las memorias de GPU y CPU en un espacio único
- Bajas prestaciones
- *Compute capability* ≥ 3.0



Feature support (unlisted features are supported for all compute capabilities)	Compute capability (version)								
	1.0, 1.1	1.2, 1.3	2.x	3.0	3.2	3.5, 3.7, 5.x, 6.x, 7.0, 7.2	7.5	8.x	9.0
Warp vote functions (__all(), __any())	No	Yes							
Warp vote functions (__ballot())	No		Yes						
Memory fence functions (__threadfence_system())	No		Yes						
Synchronization functions (__syncthreads_count(), __syncthreads_and(), __syncthreads_or())	No		Yes						
Surface functions	No		Yes						
3D grid of thread blocks	No		Yes						
Warp shuffle functions	No			Yes					
Unified memory programming	No			Yes					
Funnel shift	No				Yes				
Dynamic parallelism	No					Yes			

```
Device 0: "GeForce GT 720"
  CUDA Driver Version / Runtime Version          7.0 / 7.0
  CUDA Capability Major/Minor version number:    3.5
  Total amount of global memory:                 2047 MBytes (2146762752 bytes)
  ( 1) Multiprocessors, (192) CUDA Cores/MP:     192 CUDA Cores
  GPU Max Clock rate:                            797 MHz (0.80 GHz)
  Memory Clock rate:                             900 Mhz
  Memory Bus Width:                              64-bit
  L2 Cache Size:                                 524288 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D=(4096,
4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 7.0, CUDA Runtime Version = 7.0,
NumDevs = 1, Device0 = GeForce GT 720
Result = PASS
```

Equipo de mi despacho

Tarjeta: 250€ aprox.

```
Device 0: "NVIDIA T400 4GB"
  CUDA Driver Version / Runtime Version          12.0 / 12.1
  CUDA Capability Major/Minor version number:    7.5
  Total amount of global memory:                 3901 MBytes (4090494976 bytes)
  ( 6) Multiprocessors x (192) CUDA Cores/MP:    1152 CUDA Cores
  GPU Clock rate:                                1425 MHz (1.42 GHz)
  Memory Clock rate:                             5001 Mhz
  Memory Bus Width:                              64-bit
  L2 Cache Size:                                 524288 bytes
  Max Texture Dimension Size (x,y,z)             1D=(131072), 2D=(131072,65536),
3D=(16384,16384,16384)
  Max Layered Texture Size (dim) x layers        1D=(32768) x 2048, 2D=(32768,32768) x 2048
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1024
  Maximum number of threads per block:           1024
  Maximum sizes of each dimension of a block:    1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:     2147483647 x 65535 x 65535
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 3 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Bus ID / PCI location ID:           1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 12.0, CUDA Runtime Version = 12.1,
NumDevs = 1, Device0 = NVIDIA T400 4GB
```

Equipo de prácticas curso 2021/22:

Tarjeta: 800€ aprox.

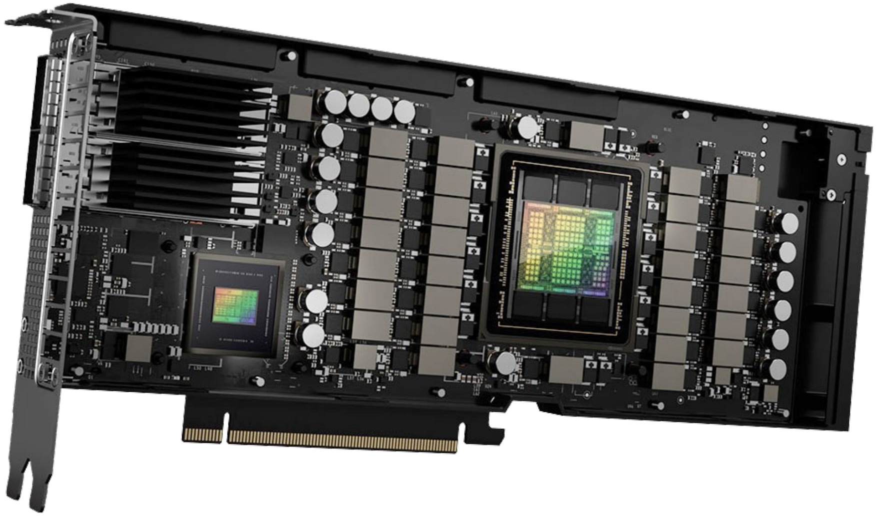
```
Detected 2 CUDA Capable device(s)

Device 0: "GeForce GTX 1080 Ti"
  CUDA Driver Version / Runtime Version          10.1 / 10.1
  CUDA Capability Major/Minor version number:    6.1
  Total amount of global memory:                 11178 MBytes (11721506816 bytes)
  (28) Multiprocessors x (192) CUDA Cores/MP:    5376 CUDA Cores
  GPU Clock rate:                                1633 MHz (1.63 GHz)
  Memory Clock rate:                             5505 Mhz
  Memory Bus Width:                              352-bit
  L2 Cache Size:                                 2883584 bytes
  Max Texture Dimension Size (x,y,z)             1D=(131072), 2D=(131072,65536),
3D=(16384,16384,16384)
  Max Layered Texture Size (dim) x layers        1D=(32768) x 2048, 2D=(32768,32768) x 2048
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Maximum sizes of each dimension of a block:    1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:     2147483647 x 65535 x 65535
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Bus ID / PCI location ID:           23 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >
```

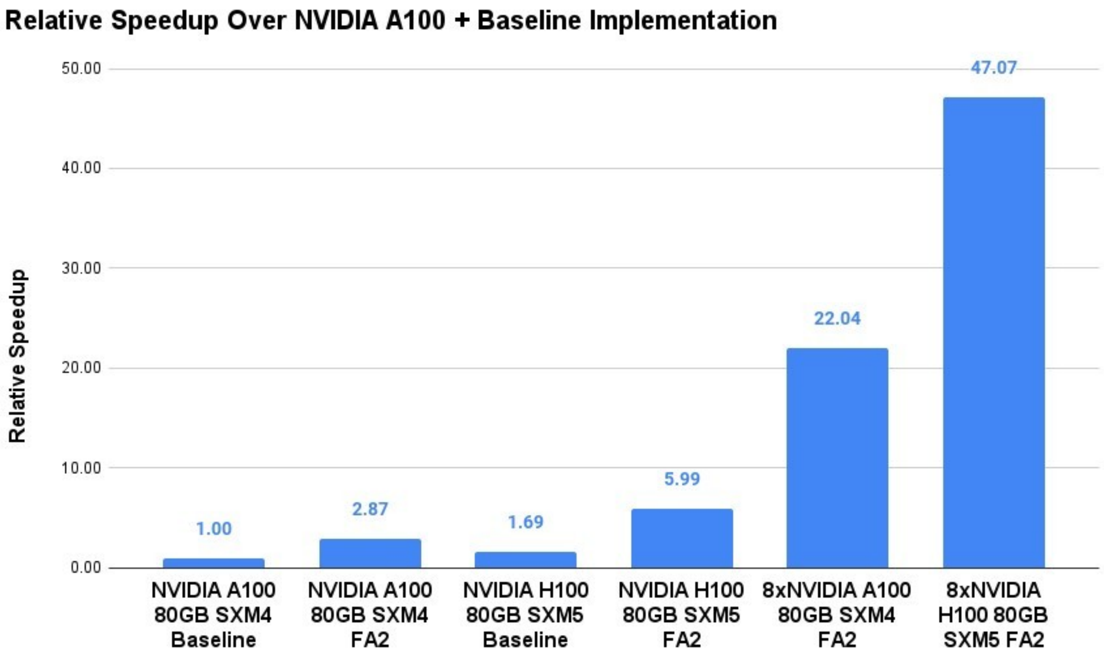
Tarjeta: 5000€ aprox.

```
Detected 1 CUDA Capable device(s)

Device 0: "NVIDIA RTX A6000"
  CUDA Driver Version / Runtime Version          12.2 / 12.1
  CUDA Capability Major/Minor version number:    8.6
  Total amount of global memory:                 48669 MBytes (51032686592 bytes)
  (84) Multiprocessors x (192) CUDA Cores/MP:    16128 CUDA Cores
  GPU Clock rate:                                1800 MHz (1.80 GHz)
  Memory Clock rate:                             8001 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 6291456 bytes
  Max Texture Dimension Size (x,y,z)             1D=(131072), 2D=(131072,65536),
3D=(16384,16384,16384)
  Max Layered Texture Size (dim) x layers        1D=(32768) x 2048, 2D=(32768,32768) x 2048
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1536
  Maximum number of threads per block:           1024
  Maximum sizes of each dimension of a block:    1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:     2147483647 x 65535 x 65535
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Bus ID / PCI location ID:           1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >
```

Arquitectura NVidia Hopper

- Incluye *tensor cores*
- Diseñadas específicamente para *deep learning*
- Precio ≥ 25000 euros



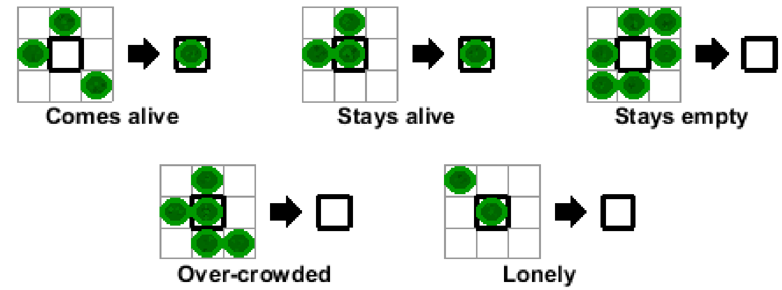


- NVidia H100 vs A100
- Precio A100 ≥ 5000 euros

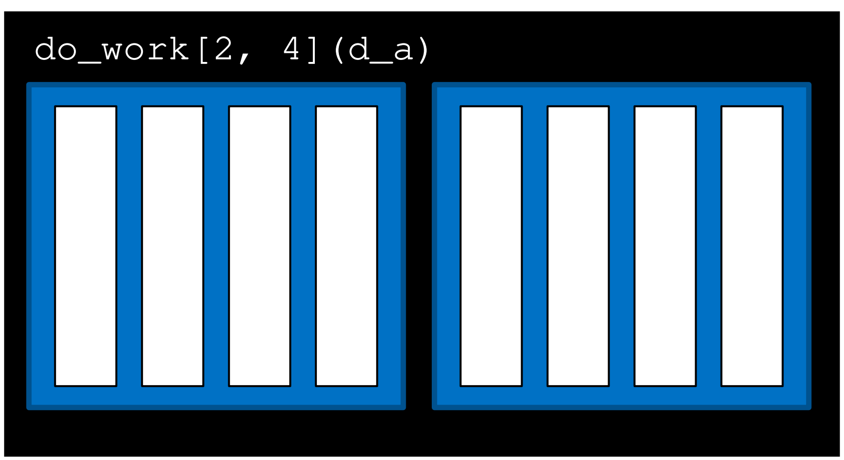
C Kernels en CUDA

¿Porqué hay que implementar Kernels en CUDA?

- Ufuncs muy útiles, pero limitados a casos en que es la misma operación a todos los elementos de una estructura
- Normalmente hay que acceder a más de un elemento para hacer cálculos



Los kernel lanzan hilos (*threads*) agrupados en bloques (*blocks*). El conjunto de bloques forma un *grid.* Todos los bloques poseen el mismo número de hilos



`gridDim.x` is the number of blocks in the grid, in this case **2**

`blockIdx.x` is the index of the current block within the grid

`blockDim.x` describes the number of threads in a block. In this case **4**

Inside a kernel **`threadIdx.x`** describes the index of the thread within a block

Primer kernel CUDA

```python
from numba import cuda
import numpy as np

# Kernels decorados con `@cuda.jit` no devuelven valores
# No es necesaria signatura de tipos
@cuda.jit
def add_kernel(x, y, out):
    idx = cuda.grid(1)
        # 1 = grid unidimensional
        # cuda.grid(1) = cuda.threadIdx.x + cuda.blockIdx.x*cuda.blockDim.x
    out[idx] = x[idx] + y[idx]


n = 4096
h_x = np.arange(n).astype(np.float32) # [0.0 ... 4095.0]
h_y = np.ones_like(h_x)              # [1.0 ... 1.0]

d_x = cuda.to_device(h_x)
d_y = cuda.to_device(h_y)
d_out = cuda.device_array_like(d_x)

# Necesitamos un hilo para cada elemento (4096)
threads_per_block = 128
blocks_per_grid = 32

add_kernel[blocks_per_grid, threads_per_block](d_x, d_y, d_out)
cuda.synchronize() # Esto sería innecesario
print(d_out.copy_to_host().astype(np.int16)) # Resultado: [1...4096]
```

```
[   1    2    3 ... 4094 4095 4096]
/home/leandro/anaconda3/envs/tftorch/lib/python3.11/site-packages/numba/cuda/
ikely result in GPU under-utilization due to low occupancy.
  warn(NumbaPerformanceWarning(msg))
```

Mejores prestaciones

128*32=4096

Ejercicio 1: crear un kernel a partir de una función

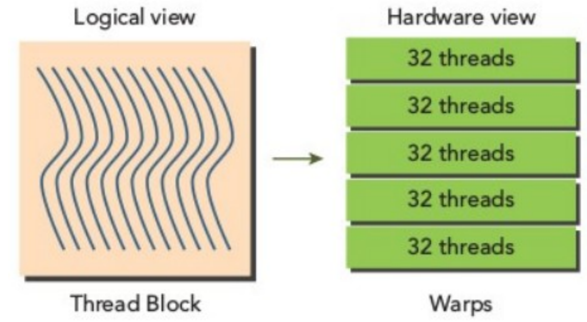
Partiendo del código suministrado, modificarlo para realizar el cálculo en la GPU

```
import numpy as np

n = 16384
def h_square(a):
    return a**2

# TODO: implementar un kernel d_square()

a = np.arange(n, dtype=np.float32)
# TODO: crear vector d_a y copiar al kernel
# TODO: crear vector en GPU para obtener la salida

# TODO: modificar estos valores e invocar kernel
blocks = 0
threads = 0

# TODO: Launch as a kernel with an appropriate execution configuration
out = h_square(a)

out_aux = a**2
np.testing.assert_almost_equal(out, out_aux)
# TODO: reemplazar out_aux con lo obtenido en el kernel (usar .copy_to_host())
```

```
%timeit h_square(h_a)
%timeit d_square[blocks,threads](d_a,d_out)
```

```
er loop (mean ± std. dev. of 7 runs, 10,000 loops each)
 per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

Rendimiento de los kernel CUDA



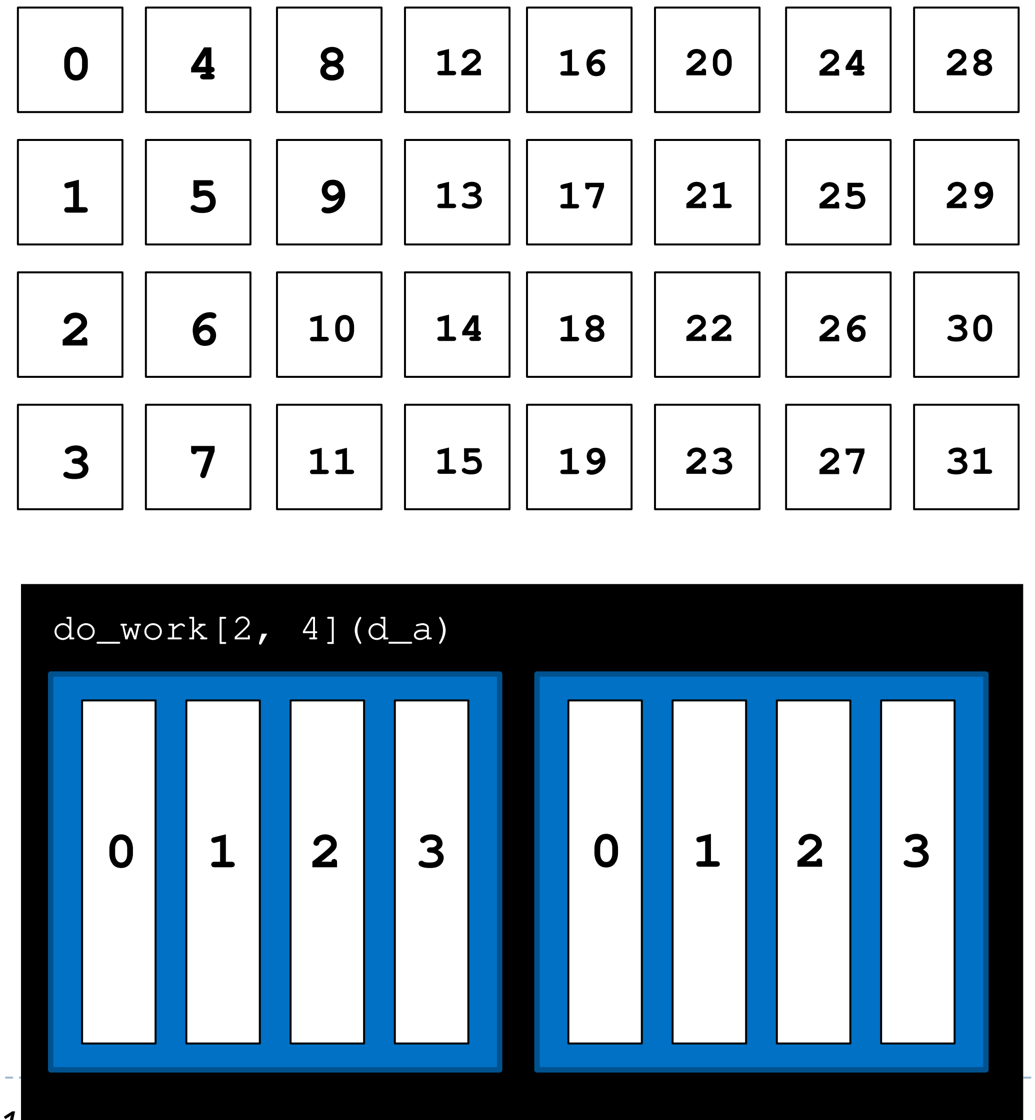
Notas sobre la ejecución de kernels CUDA:

- En la ejecución, cada bloque se asigna a un SM, con potencialmente muchos bloques asignados a cada SM
- Los hilos de un bloque se dividen en grupos de 32 llamados **warps**. Los warps se ejecutan en paralelo
- **CUDA gestiona de manera automática los cambios de los warps en ejecución. No hay latencia apreciable entre cambios de warp**
- Un kernel debe estar compuesto de un número de warps suficiente (configurando el número de bloques)
- El tamaño de un grid vendrá dado por el problema y la compute capability de la GPU. Algunas ideas:
  - Tamaño de bloque múltiplo de 32 (hilos/warp): típicamente entre 128 y 512 hilos/bloque
  - Tamaño de grid que ocupe eficientemente la GPU. Típicamente nº de bloques 2x-4x el nº de SMs de la GPU
  - Si el tamaño de los datos es muy grande, en vez de aumentar mucho el número de bloques, hacer kernels que procesen más datos. Ejecutar bloques tiene latencia

D Uso de stride en Kernels CUDA

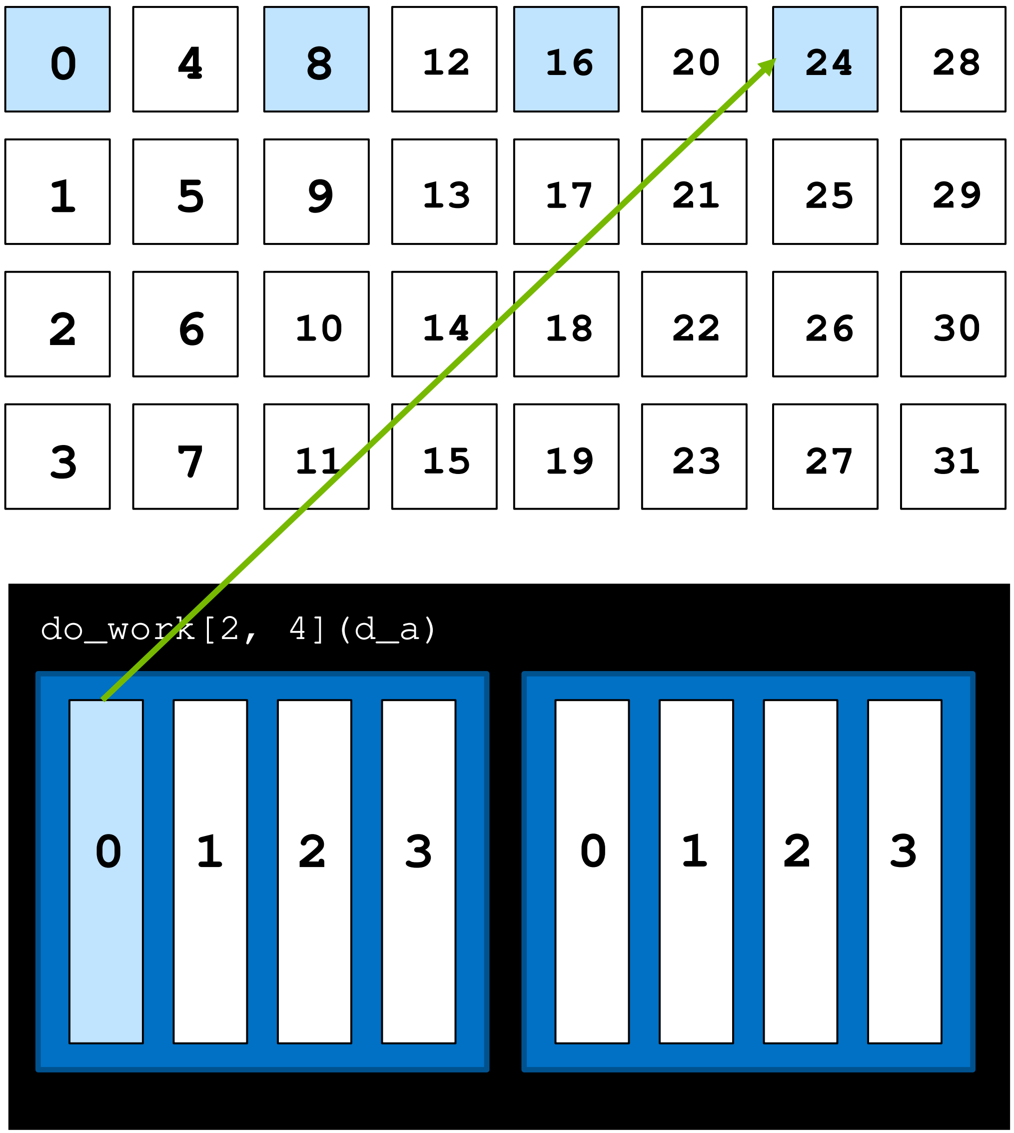
A menudo el conjunto de datos es muy grande:

- No es conveniente aumentar el número de bloques en exceso
- Mejor que cada hilo procese más de un elemento: uso de **stride**



Ejemplo

- 2 bloques
- 4 hilos/bloque
- 32 elementos a procesar



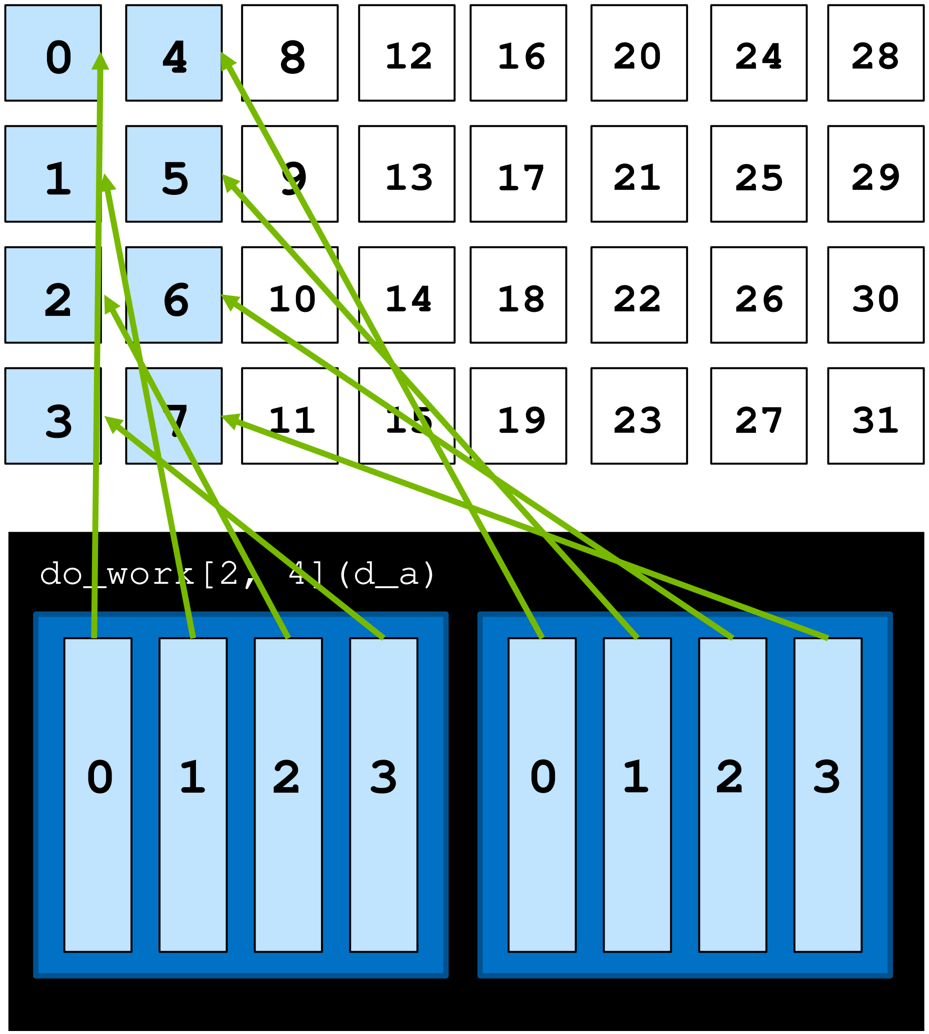
One way to address this programmatically is with a **grid-stride loop**

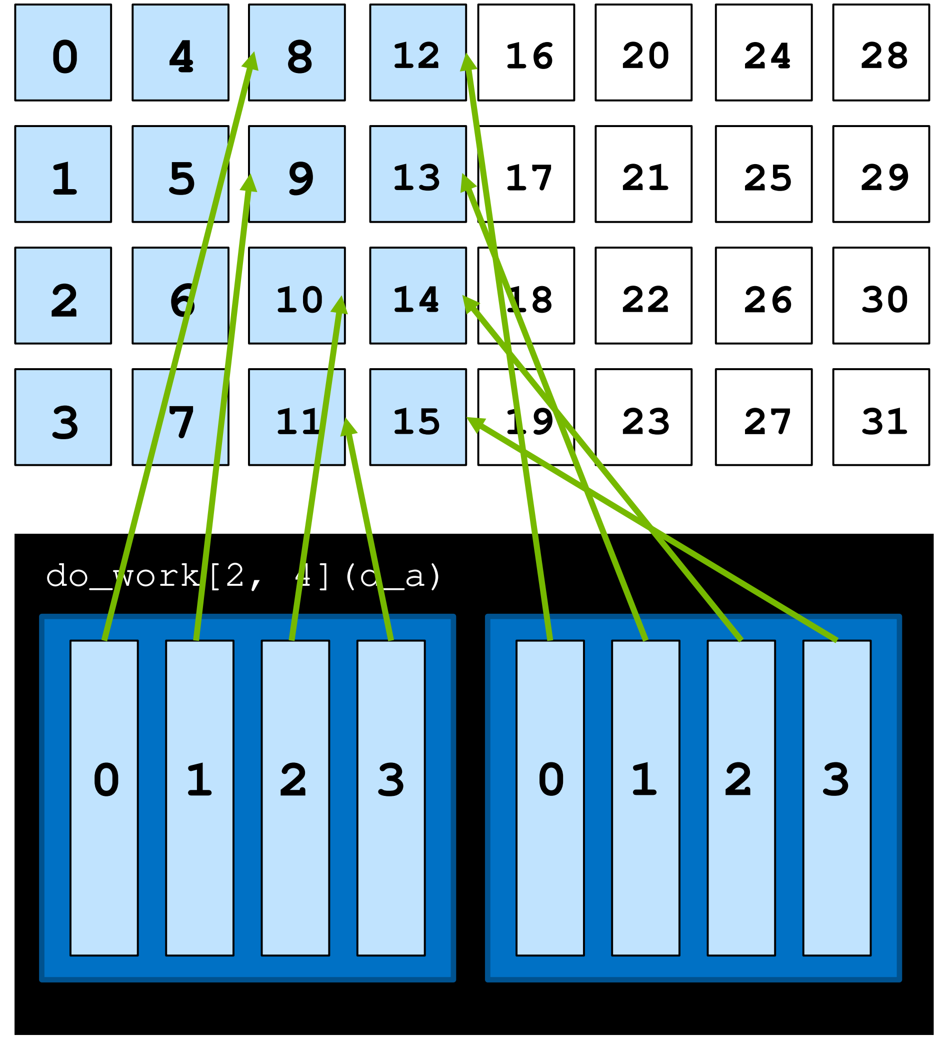
In a grid-stride loop, the thread's first element is calculated as usual, with **`cuda.grid()`**

The thread then strides forward by the total number of threads in the grid **`(blockDim.x * gridDim.x)`**, in this case **`8`**

Numba provides another convenience function for this common calculation: **`cuda.gridsize()`**, returning the number of threads in the grid

The thread continues in this way until its data index is greater than the number of data elements





And grid stride loops support this **memory coalescing** because threads executing in parallel will access adjacent data elements

With all threads working in this way, all elements are covered with the performance advantage of memory coalescing

Sin stride:

```
from numba import cuda
import numpy as np

@cuda.jit
def add_kernel(x, y, out):
    idx = cuda.grid(1)
    out[idx] = x[idx] + y[idx]

n = 4096
h_x = np.arange(n).astype(np.float32)
h_y = np.ones_like(h_x)

d_x = cuda.to_device(h_x)
d_y = cuda.to_device(h_y)
d_out = cuda.device_array_like(d_x)

threads_per_block = 128
blocks_per_grid = 32

add_kernel[blocks_per_grid, threads_per_block](d_x, d_y, d_out)
d_out.copy_to_host().astype(np.int16)
```

```
/home/leandro/anaconda3/envs/tftorch/lib/python3.11/site-package
ikely result in GPU under-utilization due to low occupancy.
  warn(NumbaPerformanceWarning(msg))
```

```
array([   1,    2,    3, ..., 4094, 4095, 4096], dtype=int16)
```

Con stride:

```
from numba import cuda
import numpy as np

@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]

n = 125000
h_x = np.arange(n).astype(np.float32)
h_y = np.ones_like(h_x)

d_x = cuda.to_device(h_x)
d_y = cuda.to_device(h_y)
d_out = cuda.device_array_like(d_x)

threads_per_block = 128
blocks_per_grid = 56

add_kernel[blocks_per_grid, threads_per_block](d_x, d_y, d_out)
d_out.copy_to_host().astype(np.float16)
```

```
/home/leandro/anaconda3/envs/tftorch/lib/python3.11/site-package
ikely result in GPU under-utilization due to low occupancy.
  warn(NumbaPerformanceWarning(msg))
/tmp/ipykernel_6055/1658409643.py:24: RuntimeWarning: overflow e
  d_out.copy_to_host().astype(np.float16)
```

```
array([ 1.,  2.,  3., ..., inf, inf, inf], dtype=float16)
```

Ejercicio 2: kernel CUDA con stride

En el código proporcionado, implementar el kernel

$$r = \sqrt{x^2 + y^2} = \sqrt{x^2\left(1 + \left(\frac{y}{x}\right)^2\right)} = |x|\sqrt{1 + \left(\frac{y}{x}\right)^2}$$

```
import numpy as np
from math import hypot
from numba import cuda

def cpu_hypot(a,b):
    return np.hypot(a,b)

# TODO: implementar esta función
# usando stride
def gpu_hypot_stride(a, b, c):
    None

# No modificar a partir de aquí
n = 1000000
h_a = np.random.uniform(-12, 12, n).astype(np.float32)
h_b = np.random.uniform(-12, 12, n).astype(np.float32)
d_a = cuda.to_device(h_a)
d_b = cuda.to_device(h_b)
d_c = cuda.device_array_like(d_b)

blocks = 128
threads_per_block = 64
gpu_hypot_stride[blocks, threads_per_block](d_a, d_b, d_c)
np.testing.assert_almost_equal(np.hypot(h_a, h_b), d_c.copy_to_host(), decimal=5)
```

```
%timeit cpu_hypot(h_a,h_b)
%timeit gpu_hypot_stride[128, 64](d_a, d_b, d_c)
```

```
 per loop (mean ± std. dev. of 7 runs, 100 loops each)
er loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

Operaciones atómicas

Con los hilos en paralelo, es posible que se den *race conditions*. Por ejemplo, un contador global:

- Leemos el valor del contador
- Incrementamos el valor del contador
- Escribimos el contador

Para evitar estos problemas, tenemos **operaciones atómicas**

```
import numpy as np
from numba import cuda

@cuda.jit
def thread_counter_race_condition(global_counter):
    global_counter[0] += 1  # Mal

@cuda.jit
def thread_counter_safe(global_counter):
    cuda.atomic.add(global_counter, 0, 1)

# Esto no funciona bien
global_counter = cuda.to_device(np.array([0], dtype=np.float32))
thread_counter_race_condition[64, 64](global_counter)
print('Debería dar %d:' % (64*64), global_counter.copy_to_host().astype(np.int16))

# Esto sí funciona bien
global_counter = cuda.to_device(np.array([0], dtype=np.float32))
thread_counter_safe[64, 64](global_counter)
print('Debería dar %d:' % (64*64), global_counter.copy_to_host().astype(np.int16))
```

```
Debería dar 4096: [1]
Debería dar 4096: [4096]
```

F Kernels bidimensionales y tridimensionales

Dependiendo de la naturaleza del cálculo y del problema, puede ser interesante tener grids y bloques de más de una dimensión

```
import numpy as np
from numba import cuda

@cuda.jit
def get_2D_indices(A):
    x, y = cuda.grid(2) # Obtenemos las dos dimensiones
    # Equivalente a:
    # x = cuda.blockIdx.x * cuda.blockDim.x + cuda.threadIdx.x
    # y = cuda.blockIdx.y * cuda.blockDim.y + cuda.threadIdx.y

    # Escribimos índice x + '.' + índice y
    A[x][y] = x + y / 10

d_A = cuda.device_array(shape=(4,4), dtype=np.float32)
    # Matriz 4x4 en la GPU

blocks = (2, 2) # Grid = 2x2 bloques
threads_per_block = (2, 2) # Bloque = 2x2 threads

get_2D_indices[blocks, threads_per_block](d_A)

np.set_printoptions(precision=1, floatmode="fixed")
print(d_A.copy_to_host())
```

```
/home/leandro/anaconda3/envs/tftorch/lib/python3.11/site-packages/
kely result in GPU under-utilization due to low occupancy.
  warn(NumbaPerformanceWarning(msg))
[[0.0 0.1 0.2 0.3]
 [1.0 1.1 1.2 1.3]
 [2.0 2.1 2.2 2.3]
 [3.0 3.1 3.2 3.3]]
```

Kernel bidimensional: suma de matrices

```
from numba import cuda
import numpy as np

@cuda.jit  # Adjust block size as needed
def add_matrices(a, b, c):
    i, j = cuda.grid(2)  # Get thread indices in two dimensions (row, column)
    c[i, j] = a[i, j] + b[i, j]

# Example usage
rows = 4096
cols = 4096

h_a = np.random.rand(rows, cols).astype(np.float32)  # Allocate matrices on CPU
h_b = np.random.rand(rows, cols).astype(np.float32)
d_a = cuda.to_device(h_a)  # Transfer matrices to GPU
d_b = cuda.to_device(h_b)
d_c = cuda.device_array_like(d_b)

threads_per_block = (32, 32)
blocks = (128, 128)

add_matrices[blocks, threads_per_block](d_a, d_b, d_c)  # Launch kernel with appropriate grid size

h_c = d_c.copy_to_host()

np.testing.assert_almost_equal(h_c, h_a+h_b)
```

```
%timeit c_aux= (h_a + h_b)
%timeit add_matrices[blocks, threads_per_block](d_a, d_b, d_c)
```

```
19.2 ms ± 11.3 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
11.3 ms ± 153 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

Ejercicio 3: kernel bidimensional para procesar una imagen

```
# Necesitamos skimage
# Instalar con
#       conda install scikit-image

import matplotlib.pyplot as plt
from skimage import data, color
import numpy as np

@cuda.jit
def blur(input, output):
    x, y = cuda.grid(2)
    if x>0 and y>0 and x<(input.shape[0]-1) and y<(input.shape[1]-1):
        output[x][y] = 0.25*(input[x-1][y]+input[x+1][y]+input[x][y-1]+input[x][y+1])
    else:
        output[x][y] = input [x][y]

# TODO: definir tamaño de grid y de bloque
num_cycles = 100

astronaut = (255.-color.rgb2gray(data.astronaut()))/255.0
print("Image size: ",astronaut.shape)
fig, ax = plt.subplots()
im = ax.imshow(astronaut,  cmap='Greys')

# TODO: datos a GPU (duplicar imagen)

# TODO: ejecutar num_cycles veces un el kernel blur

# TODO: copiar imagen desenfocada al host

fig, ax = plt.subplots()
im = ax.imshow(astronaut_blurred, cmap='Greys')
```