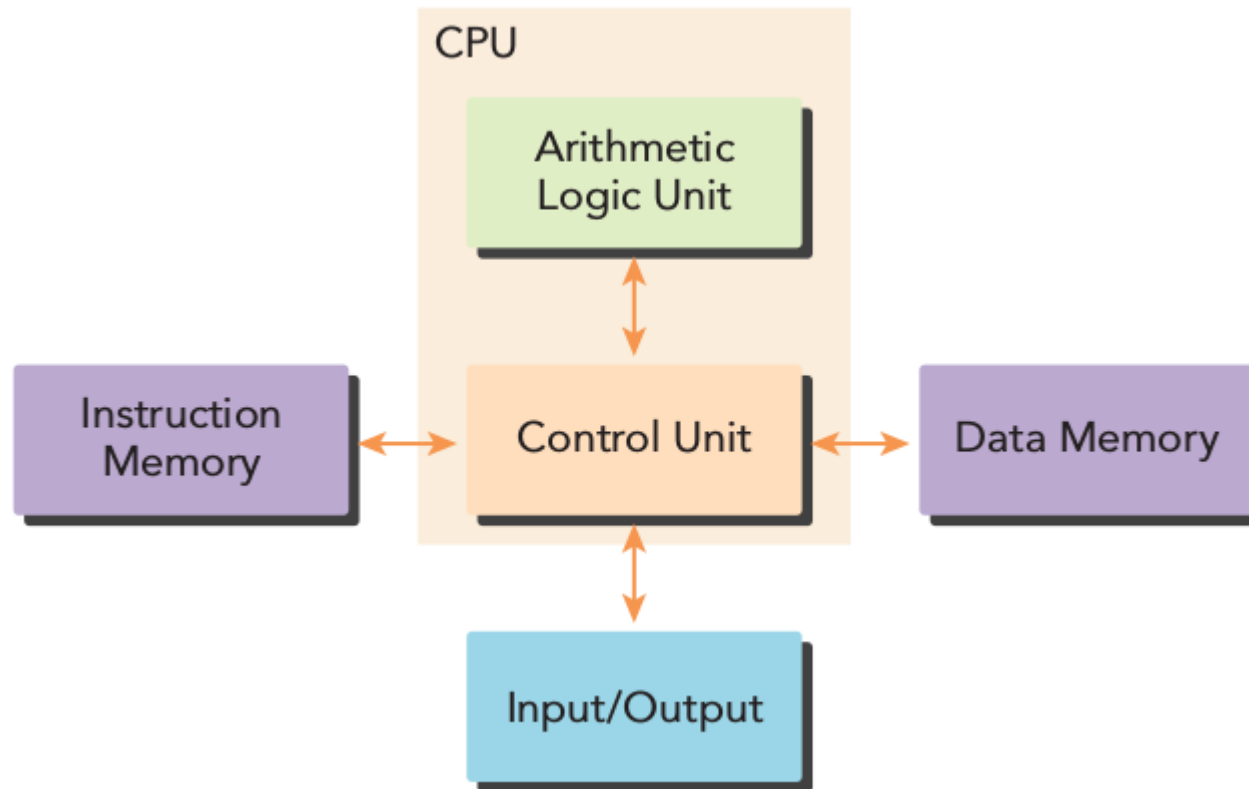


ICNPG 2023

Clase 1: CUDA y acceso a GPUs

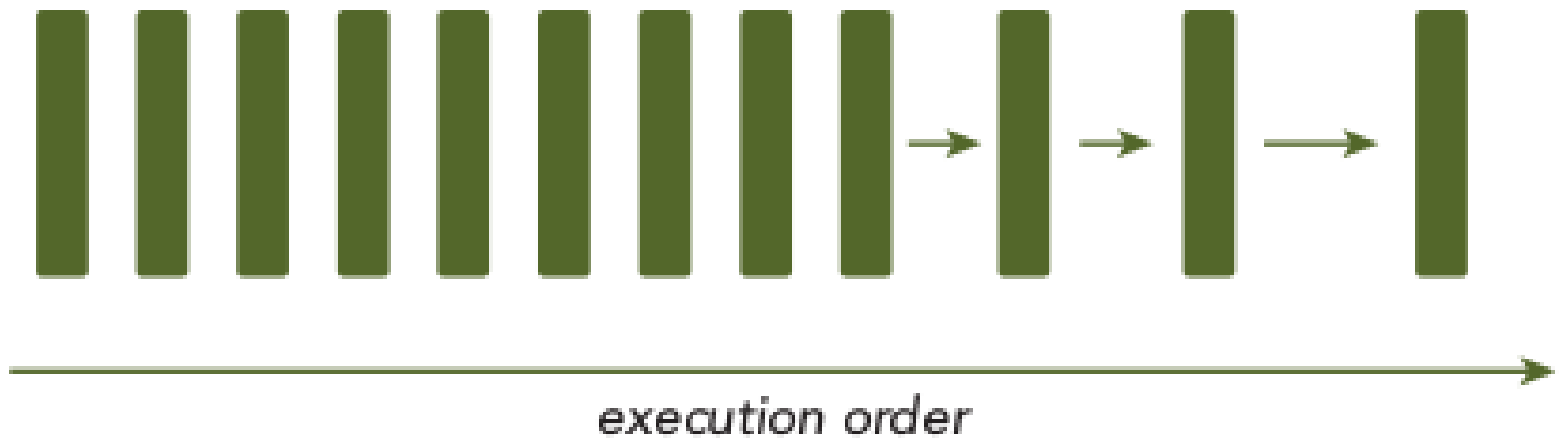


CPU (de un núcleo)

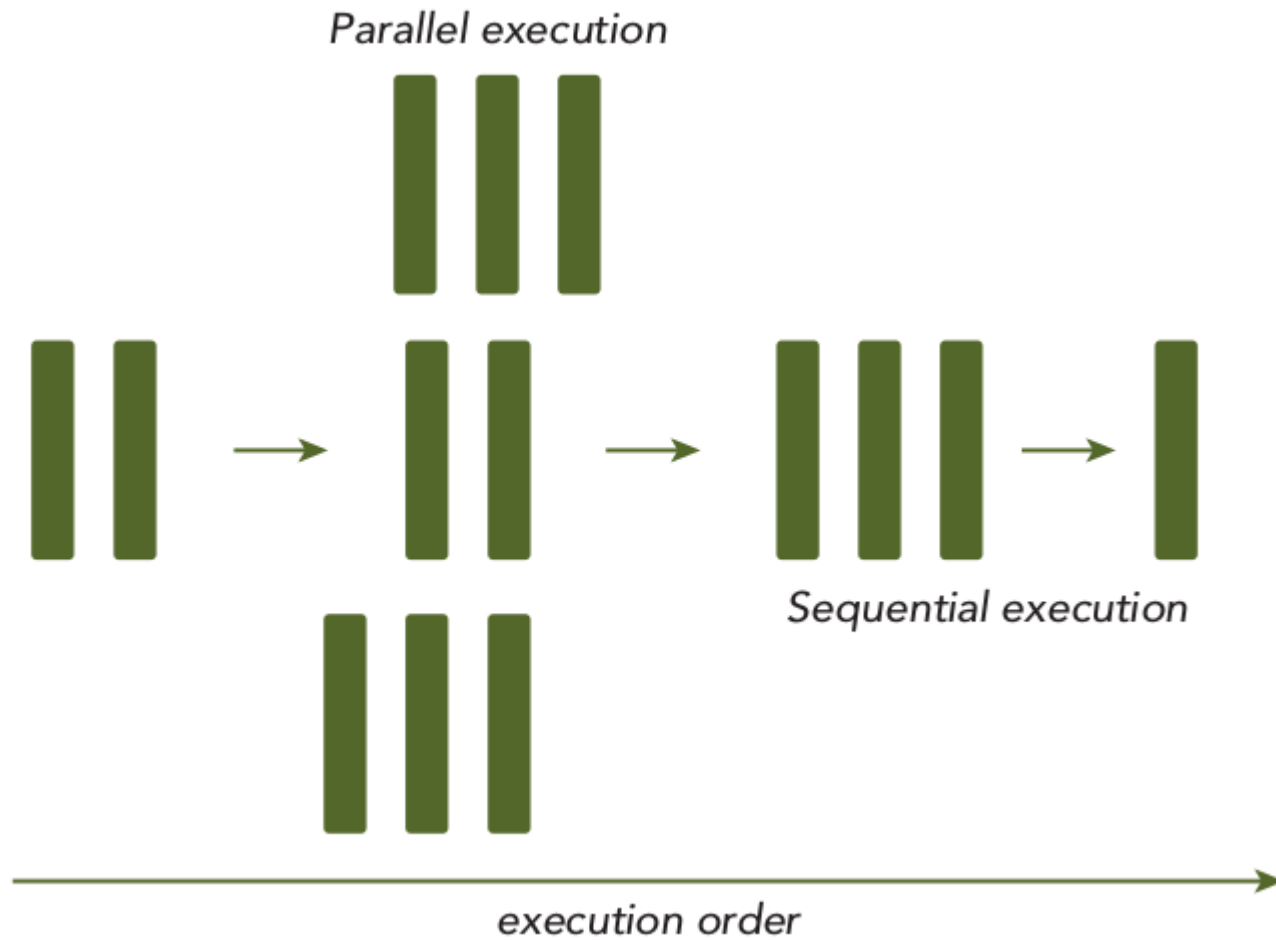


Programa secuencial

The problem is divided into small pieces of calculations.



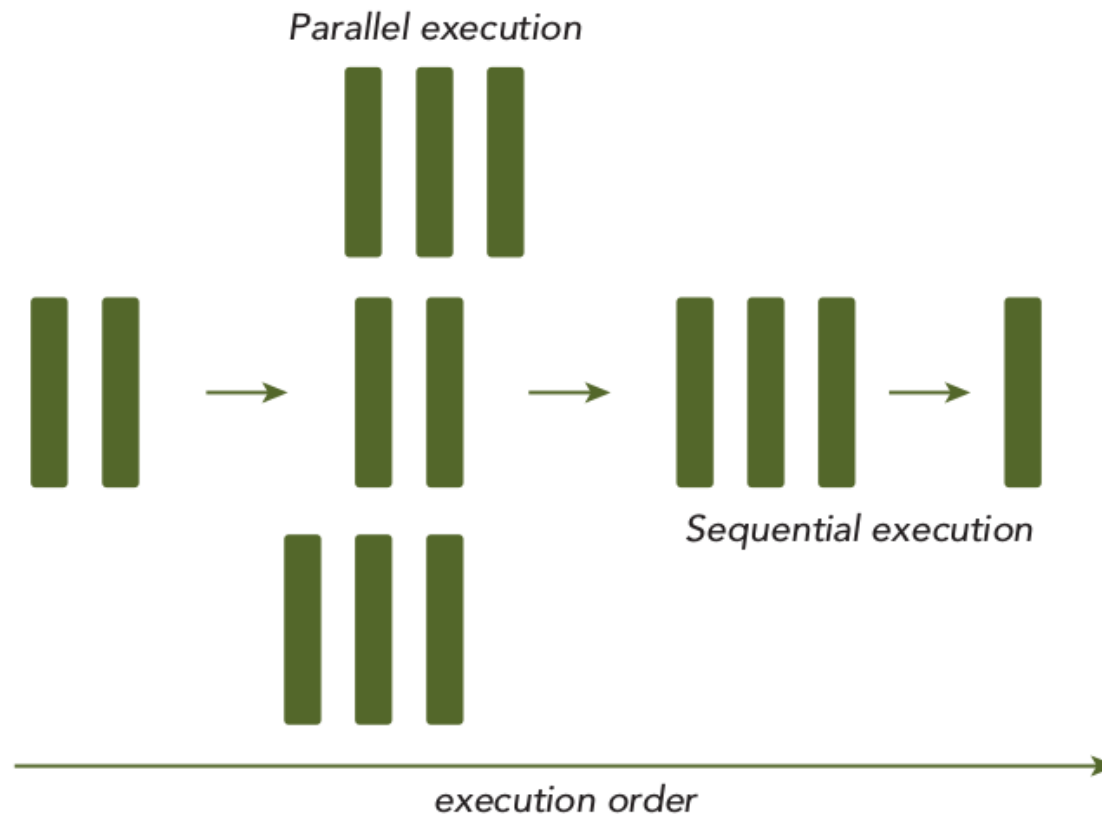
Programa Paralelo



Desafíos

- Coordinar múltiples procesos
- Asegurar la consistencia de los datos
- Minimizar “overheads” asociados a la comunicación y sincronización.

Cuando el problema es lo suficientemente grande los costos (de comunicación, por ej) se amortizan



Paralelismo

- De tareas (cpu multicore)

El sistema operativo ya usa paralelismo en la CPU

```
1  [|||||||||||||||||||||||||||||] 45.4% Tasks: 194, 835 thr; 4 running
2  [|||||||||||||||||||||||||||] 41.9% Load average: 2.70 1.35 1.05
3  [|||||||||||||||||||||||||] 44.5% Uptime: 25 days, 03:29:36
4  [|||||||||||||||||||||||||||] 49.7%
Mem[|||||||||||||||||||||||||||||] 3.31G/3.74G
Swp[|||||||||||||||||||||||||] 2.00G/3.89G
```

PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
14457	ale	20	0	1703M	163M	46604	D	39.8	4.3	17:40.23	/usr/lib/libreoffice/program/soffice.bin
1087	root	20	0	892M	90288	28532	S	27.0	2.3	14h48:19	/usr/lib/xorg/Xorg -core :0 -seat seat0 -
2068	ale	20	0	2079M	203M	48752	R	26.3	5.3	9h14:25	compiz
14050	ale	20	0	8867M	168M	86500	S	25.0	4.4	0:23.27	/opt/google/chrome/chrome --type=render
15802	ale	20	0	895M	194M	66420	S	18.2	5.1	29:58.97	/opt/google/chrome/chrome --type=gpu-proc
14287	ale	20	0	4715M	130M	76744	S	14.2	3.4	0:03.59	/opt/google/chrome/chrome --type=render
14322	ale	20	0	4648M	83576	66808	S	12.8	2.1	0:00.27	/opt/google/chrome/chrome --type=render
15767	ale	20	0	1255M	229M	67032	D	8.8	6.0	43:40.62	/opt/google/chrome/chrome

Paralelismo

- De tareas

- De datos
(GPU)

Existen distintas formas de dividir los datos en los hilos, es decir, distintas formas de asignar qué operaciones (repetitivas) tiene que hacer cada hilo

Mapear hilos (“threads”) a datos



Block partition: each thread takes one data block



Cyclic partition: each thread takes two data blocks

FIGURE 1-4

Matrices



Block partition on one dimension



Block partition on both dimensions



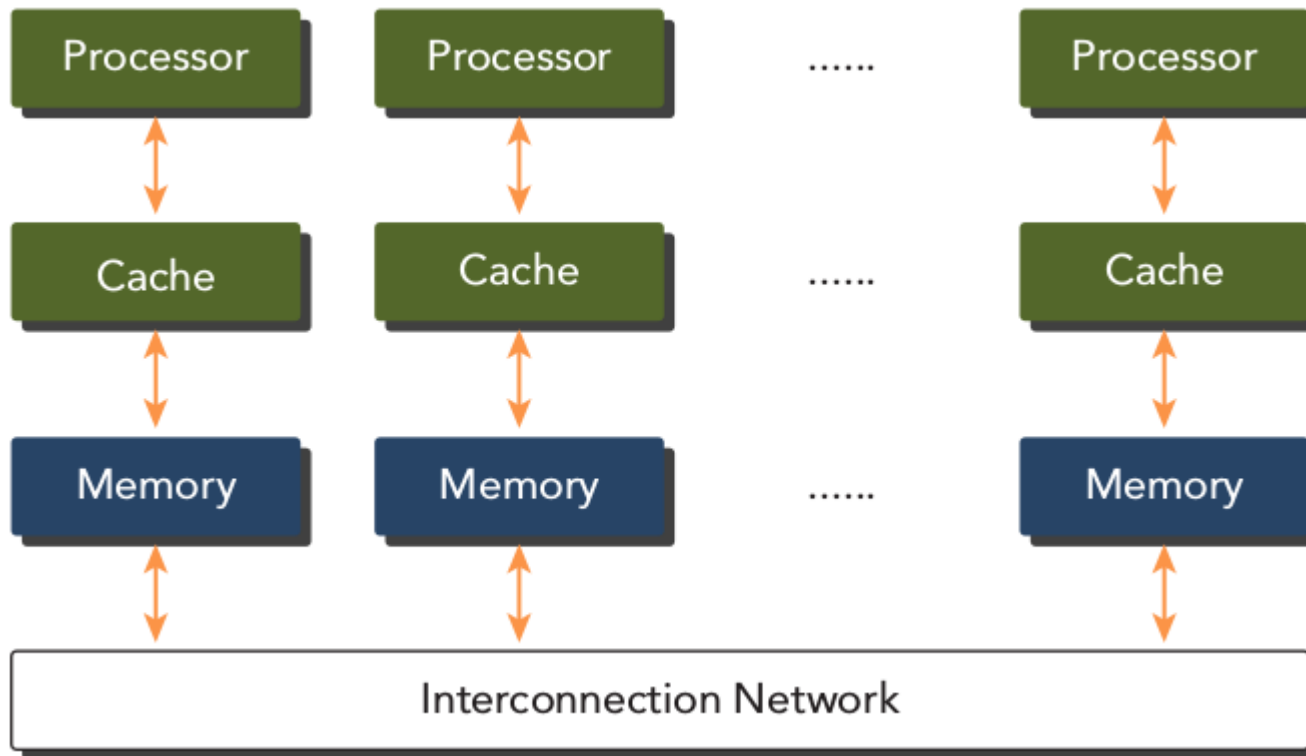
Cyclic partition on one dimension

¿Cuál es el mapeo más eficiente?

Depende de cómo estén guardados los datos en memoria, qué operaciones hay que hacer. Encontrar el óptimo es difícil pero encontrar uno bueno es sencillo

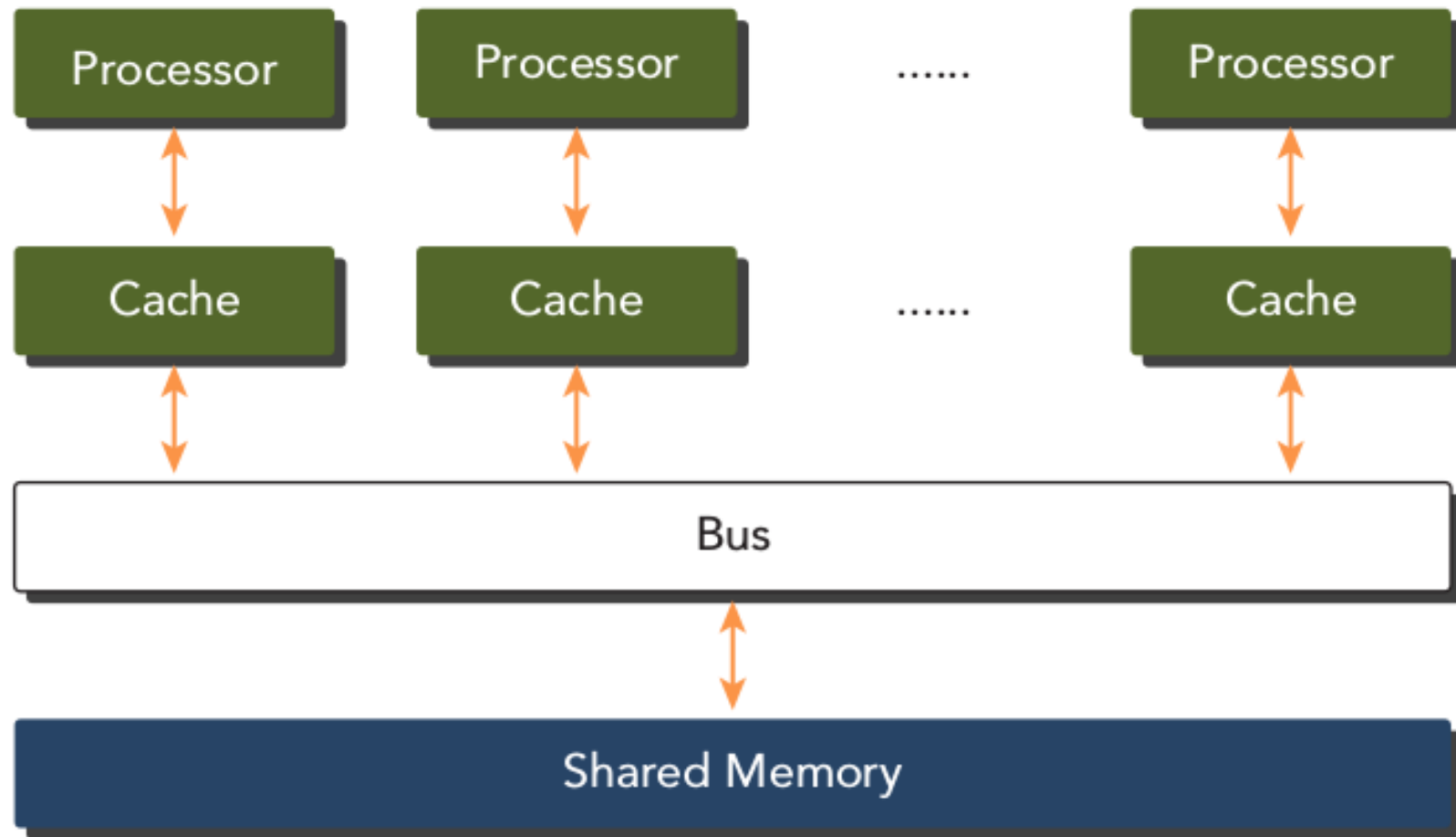
Arquitecturas paralelas: multi-node con memoria distribuída

Hay una red de computadoras conectadas entre sí mediante cables ethernet. Esta red es muy lenta por la comunicación. Si uno busca crear un cluster grande se va a enfrentar inevitablemente a este problema



Arquitecturas paralelas: multiprocessor, con memoria compartida

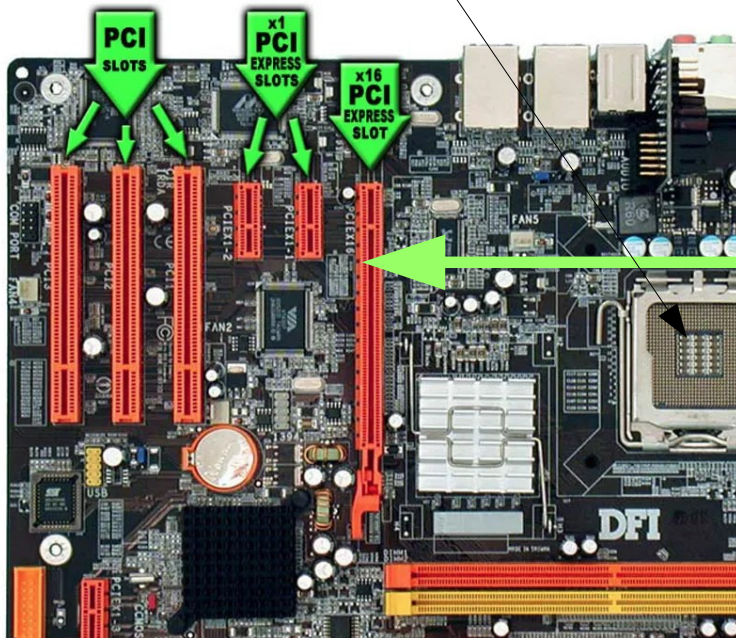
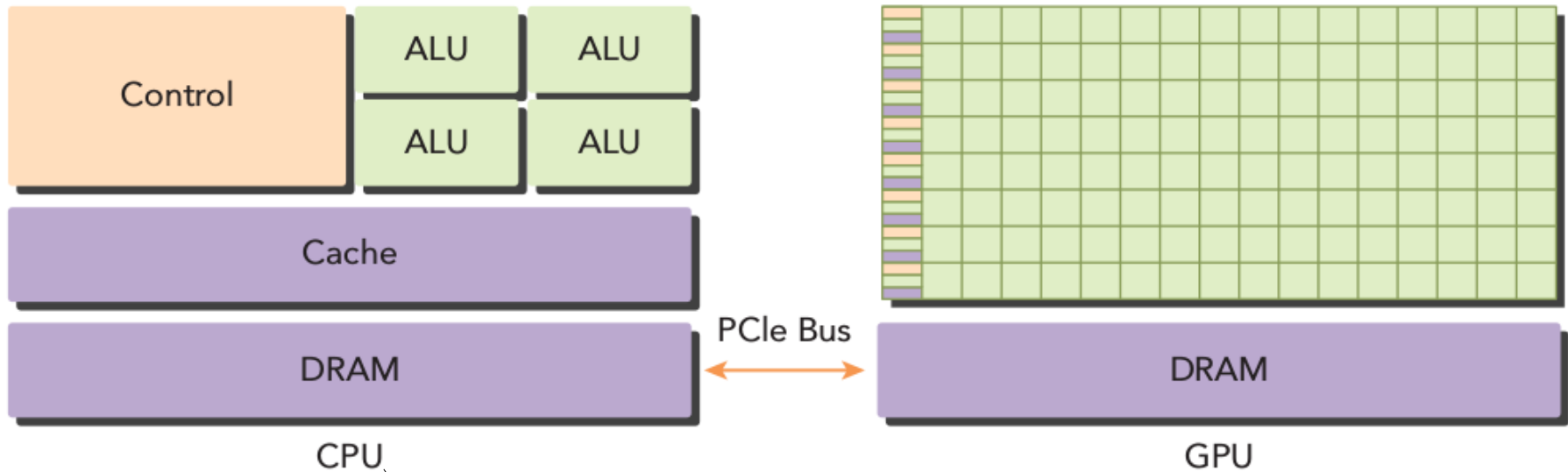
Paralelización en una misma computadora
usando los núcleos de una CPU



Multicore (laptops), manycore (con GPUs).

Arquitectura paralela híbrida

GPU y CPU conectadas. Vamos a "ir y venir" continuamente entre ambas



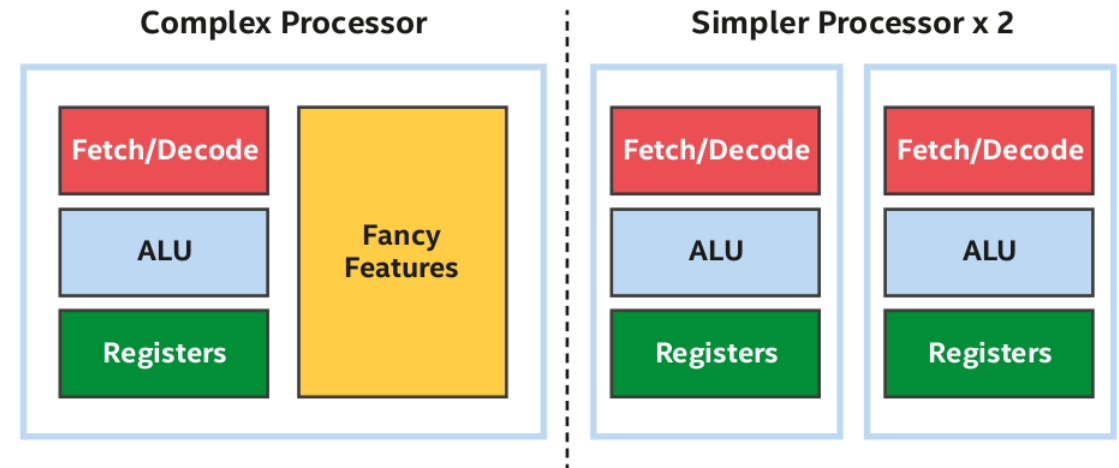
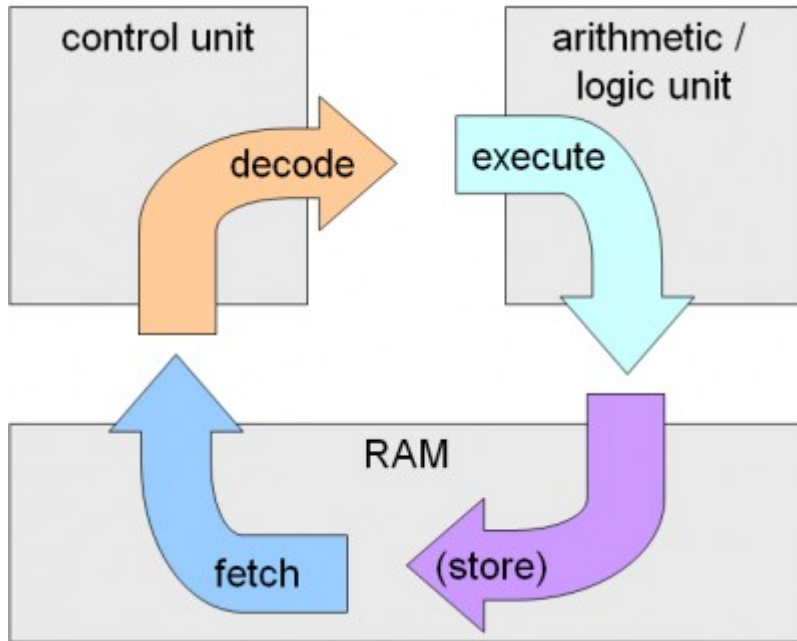


Figure 15-2. GPU processors are simpler, but there are more of them

SIMD Processor

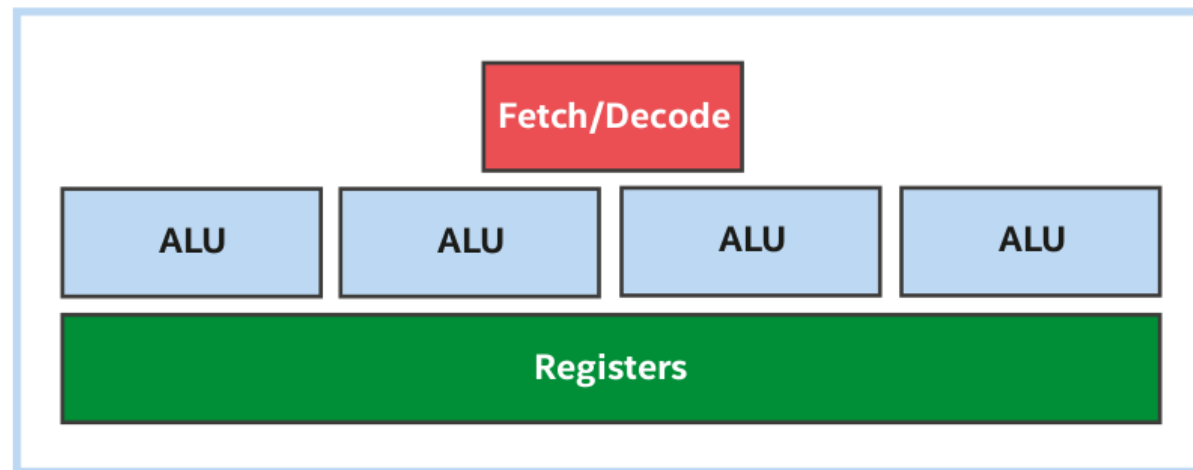
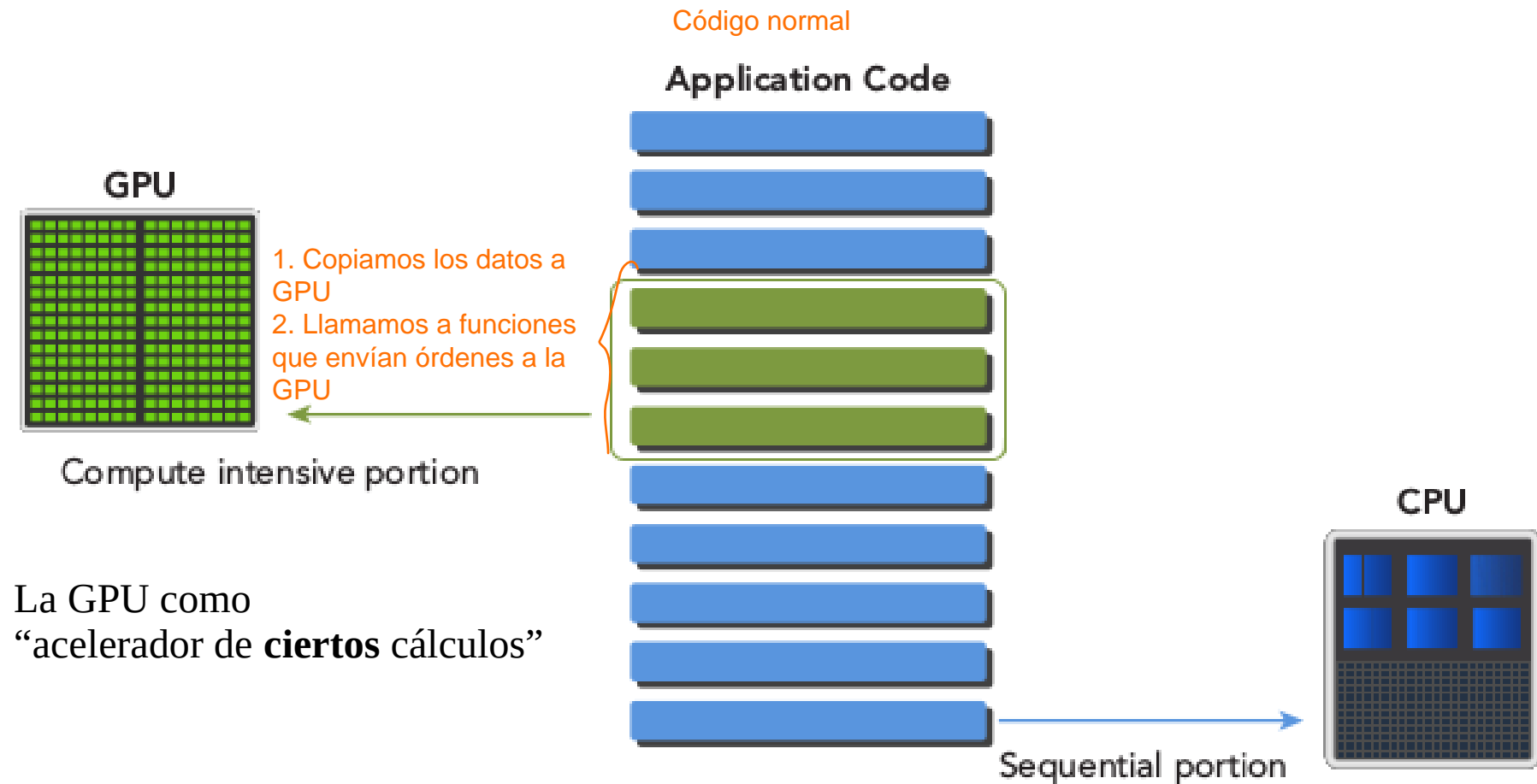


Figure 15-8. Four-wide SIMD processor: The four ALUs share fetch/decode logic

Procesadores SIMD en CPUs son menos poderosos que en GPUs (ancho=32)

Paradigma de computación heterogénea



CPU THREAD VERSUS GPU THREAD

Threads on a CPU are generally heavyweight entities. The operating system must swap threads on and off CPU execution channels to provide multithreading capability. Context switches are slow and expensive.

Threads on GPUs are extremely lightweight. In a typical system, thousands of threads are queued up for work. If the GPU must wait on one group of threads, it simply begins executing work on another.

CPU cores are designed to minimize latency for one or two threads at a time, whereas GPU cores are designed to handle a large number of concurrent, lightweight threads in order to maximize throughput.

Today, a CPU with four quad core processors can run only 16 threads concurrently, or 32 if the CPUs support hyper-threading.

Modern NVIDIA GPUs can support up to 1,536 active threads concurrently per multiprocessor. On GPUs with 16 multiprocessors, this leads to more than 24,000 concurrently active threads.

The RTX 3080 graphics card has

- **8704** CUDA cores.
- **68** streaming multiprocessors (Sms) = agrupamiento de núcleos
- A theoretical maximum of 43,264 active threads per SM, or **2,947,072** active threads across all 68 SMs.

Hay una diferencia entre hilos y núcleos. Puede haber más hilos que núcleos aprovechando que a veces un hilo tiene un tiempo de espera hasta volver a ser utilizado

Si resolvemos ecuaciones diferenciales en una grilla no es muy difícil llegar a este nro...

¿número de cores << hilos activos?

→ Esconder latencias

↪ Cuando mandamos a correr el programa con distintos tamaños de grilla, a medida que aumentamos el tamaño de grilla el tiempo de cómputo de la GPU se mantiene cte. Hasta que llegamos a un tamaño que la GPU no puede soportar y lo hace en varios pasos (serializa) y comienza a aumentar el tiempo de cómputo

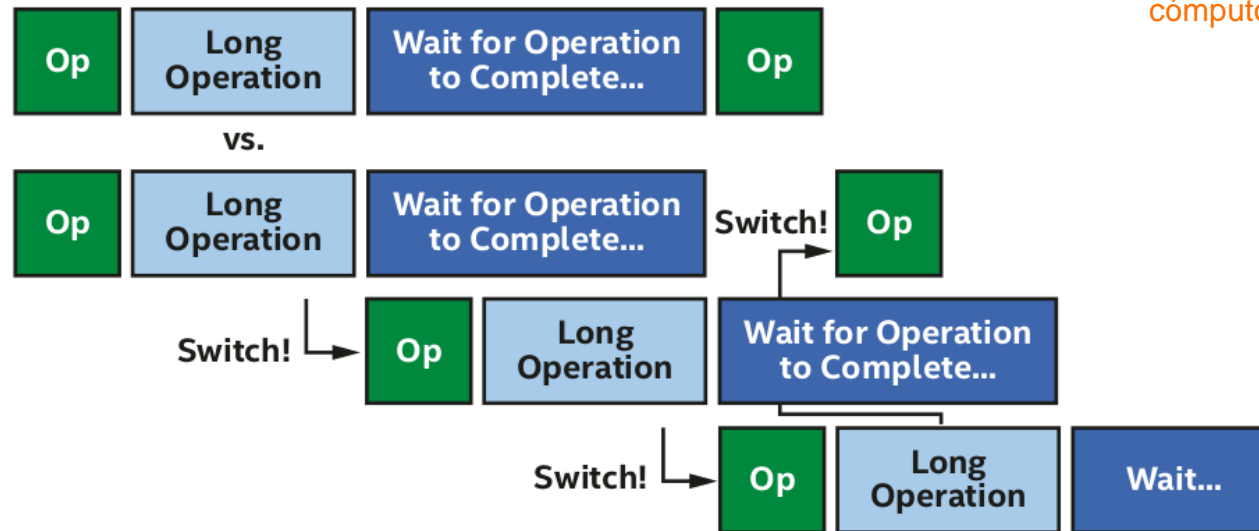
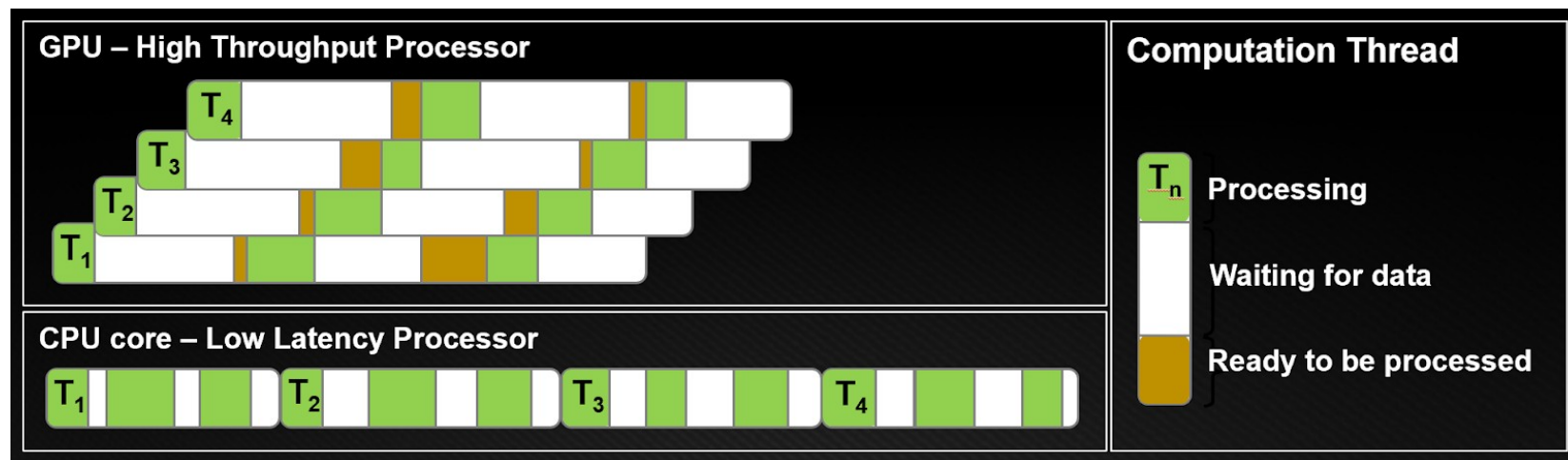


Figure 15-13. Switching instruction streams to hide latency



Buenas noticias:
Cuando programen...

PCI Express 3.0 Host Interface

GigaThread Engine

Memory Controller

Memory Controller

Memory Controller

Memory Controller

GPC

Raster Engine



GPC

Raster Engine



Memory Controller

Memory Controller

Memory Controller

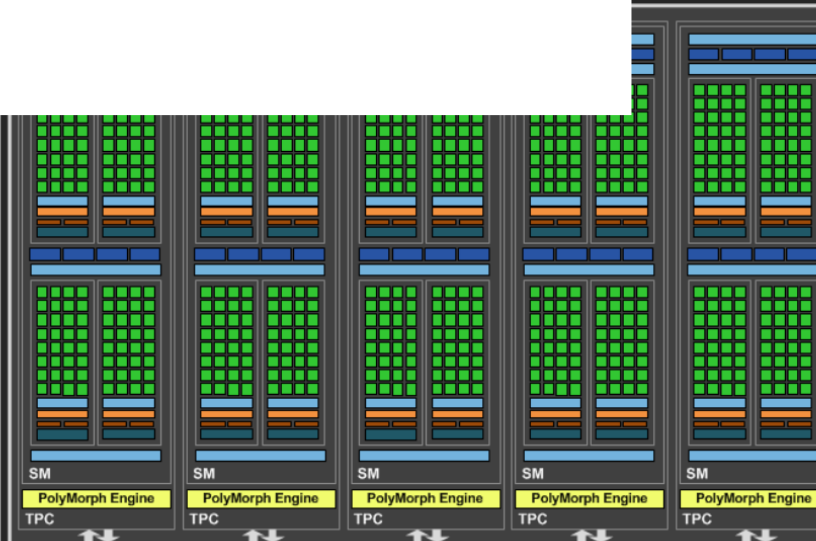
Memory Controller

Olvidéense de este quilombo



GPC

Raster Engine

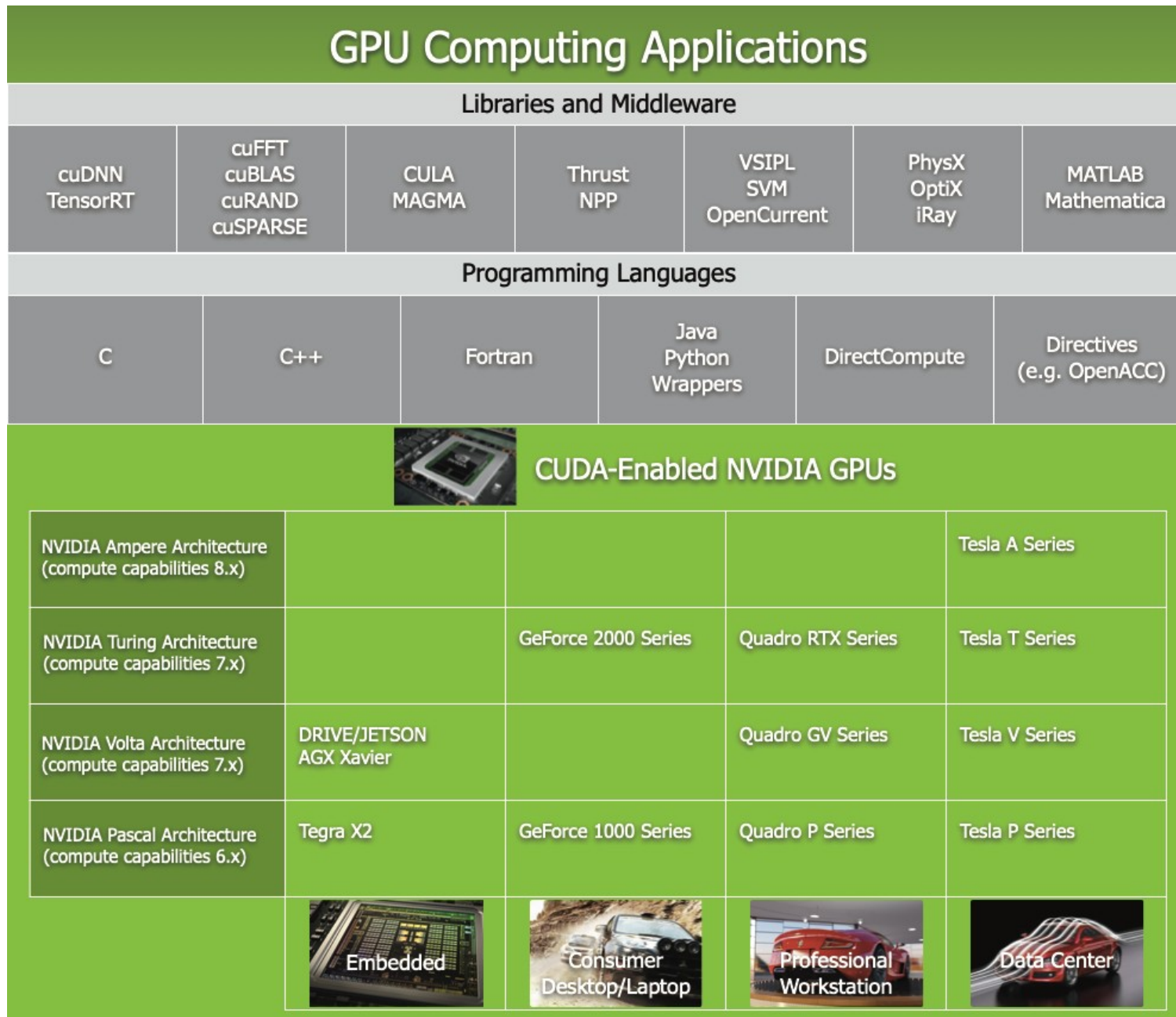


GPC

Raster Engine

CUDA:

Una plataforma para la computación heterogénea CPU+GPU

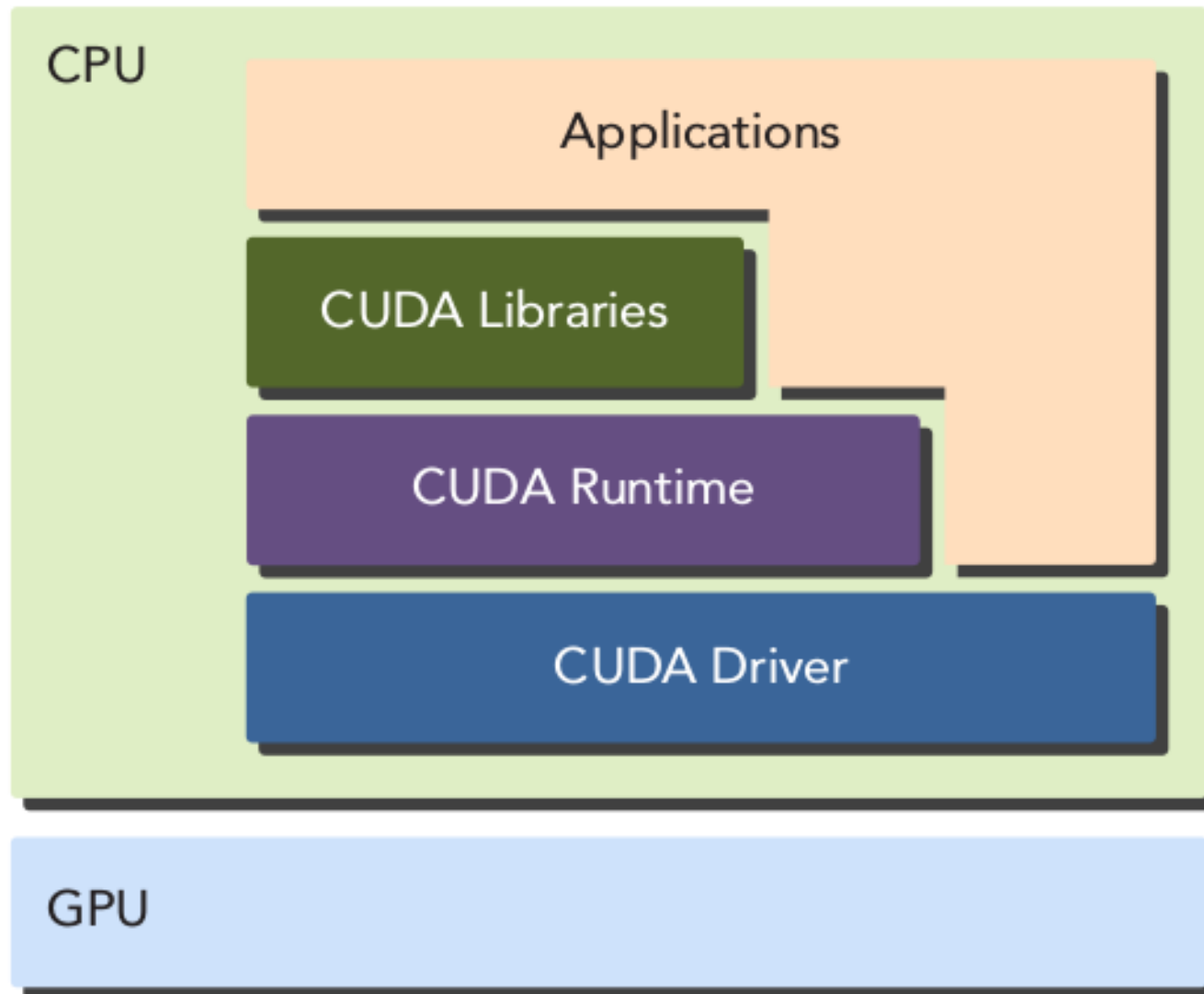


En el curso vamos a trabajar en C. Pero esto tmb será útil en Python porque en algún momento quizás necesitemos hacer un kernel y allí necesitaremos quizás escribir en C

CUDA:

Una plataforma para la computación heterogénea CPU+GPU

El programa es un
programa en CPU que
encola trabajo en la
GPU



<https://es.wikipedia.org/wiki/CUDA>



Documentación oficial:

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

Filosofía

Programen “cualquier” GPU

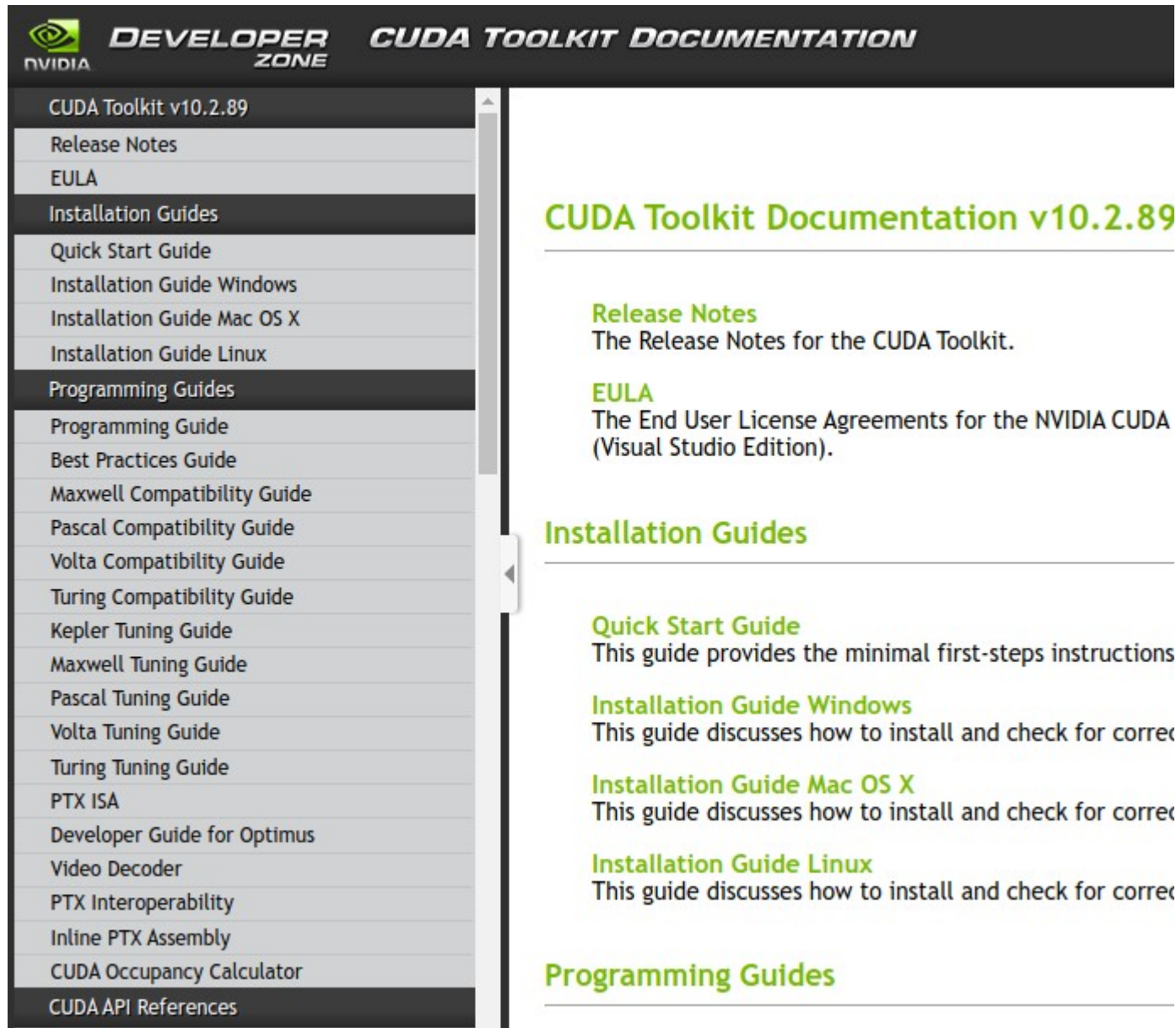
Abstracción

Paralelismo masivo

Usar tantos hilos/threads como sea necesario para procesar todos los datos que entren en la memoria.

El *mismo* programa escalará transparentemente en una GPU con mas cores

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



The screenshot shows the NVIDIA Developer Zone CUDA Toolkit Documentation page for version 10.2.89. The page has a dark header with the NVIDIA logo, 'DEVELOPER ZONE', and 'CUDA TOOLKIT DOCUMENTATION'. A left sidebar contains a table of contents with categories like 'CUDA Toolkit v10.2.89', 'Release Notes', 'EULA', 'Installation Guides', 'Programming Guides', and 'CUDA API References'. The main content area is titled 'CUDA Toolkit Documentation v10.2.89' and lists links to 'Release Notes', 'EULA', 'Installation Guides', and 'Programming Guides' with brief descriptions for each.

CUDA Toolkit v10.2.89

- Release Notes
- EULA
- Installation Guides
 - Quick Start Guide
 - Installation Guide Windows
 - Installation Guide Mac OS X
 - Installation Guide Linux
- Programming Guides
 - Programming Guide
 - Best Practices Guide
 - Maxwell Compatibility Guide
 - Pascal Compatibility Guide
 - Volta Compatibility Guide
 - Turing Compatibility Guide
 - Kepler Tuning Guide
 - Maxwell Tuning Guide
 - Pascal Tuning Guide
 - Volta Tuning Guide
 - Turing Tuning Guide
 - PTX ISA
 - Developer Guide for Optimus
 - Video Decoder
 - PTX Interoperability
 - Inline PTX Assembly
 - CUDA Occupancy Calculator
 - CUDA API References

CUDA Toolkit Documentation v10.2.89

Release Notes
The Release Notes for the CUDA Toolkit.

EULA
The End User License Agreements for the NVIDIA CUDA (Visual Studio Edition).

Installation Guides

Quick Start Guide
This guide provides the minimal first-steps instructions

Installation Guide Windows
This guide discusses how to install and check for correct

Installation Guide Mac OS X
This guide discusses how to install and check for correct

Installation Guide Linux
This guide discusses how to install and check for correct

Programming Guides

principiante



- Programming guide
- CUDA API
- CUDA samples
- Wrappers...
- Stackoverflow...
- Criteroso copy/paste
- Asentar “conceptos” con mucha practica



¡Experto!

Se puede hacer una diferencia importante rapido y con poco esfuerzo

- Acelerar apreciablemente viejos códigos x5, x10, x100.
- Sacarle más el jugo al modesto equipamiento que tenemos, incluída la computadora personal.
- Aprender a programar en paralelo, mas allá del lenguaje y la tecnología. El paralelismo es uno de los grandes paradigmas de la computación hoy en día.
- *El curso apunta a brindarles una breve introducción para que empiecen ya a implementar cosas, y para que luego sigan aprendiendo por su cuenta.*

Threads, Blocks, Grid

En CUDA cada hilo está organizado en un bloque y cada bloque forma parte de una grilla. Las grillas son conjuntos de bloques unidimensionales, bidimensionales o tridimensionales. Cada bloque tiene organizados los hilos de forma unidimensional, bidimensional o tridimensional. Esta abstracción es útil para poder mapear datos a hilos, es solo para indexear los hilos. ¿Por qué solo hasta 3D? Porque es el máximo de utilidad para los juegos

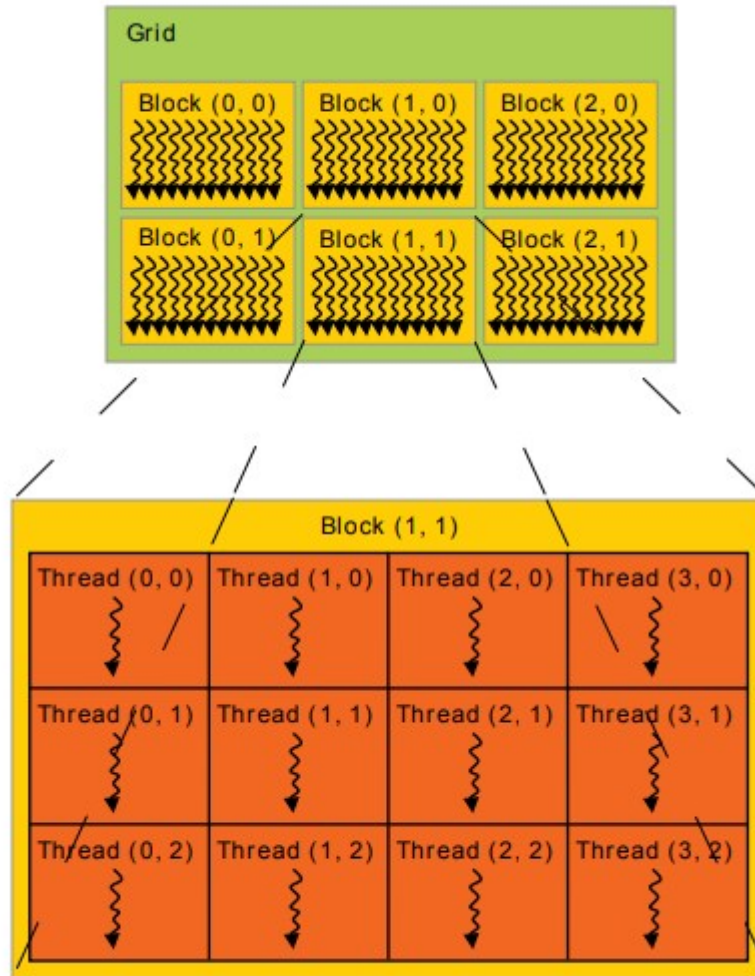
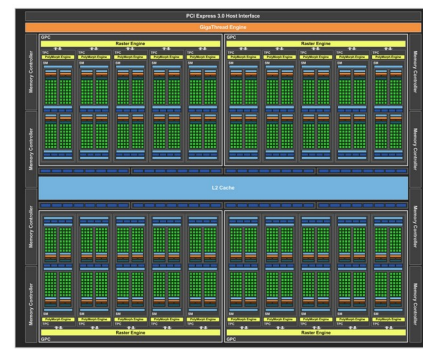


Figure 6 Grid of Thread Blocks

Procesamiento



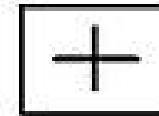
Thread



Executed by



Core



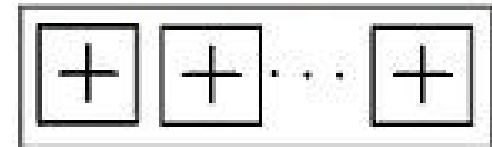
Thread Block



Executed by

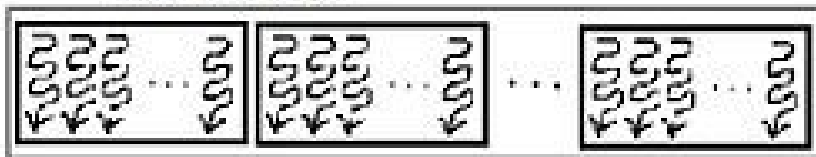


Streaming Multiprocessor



Agrupamiento de procesadores que están "cerca en el espacio"

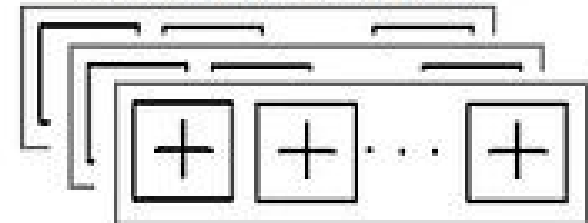
Kernel Grid



Executed by

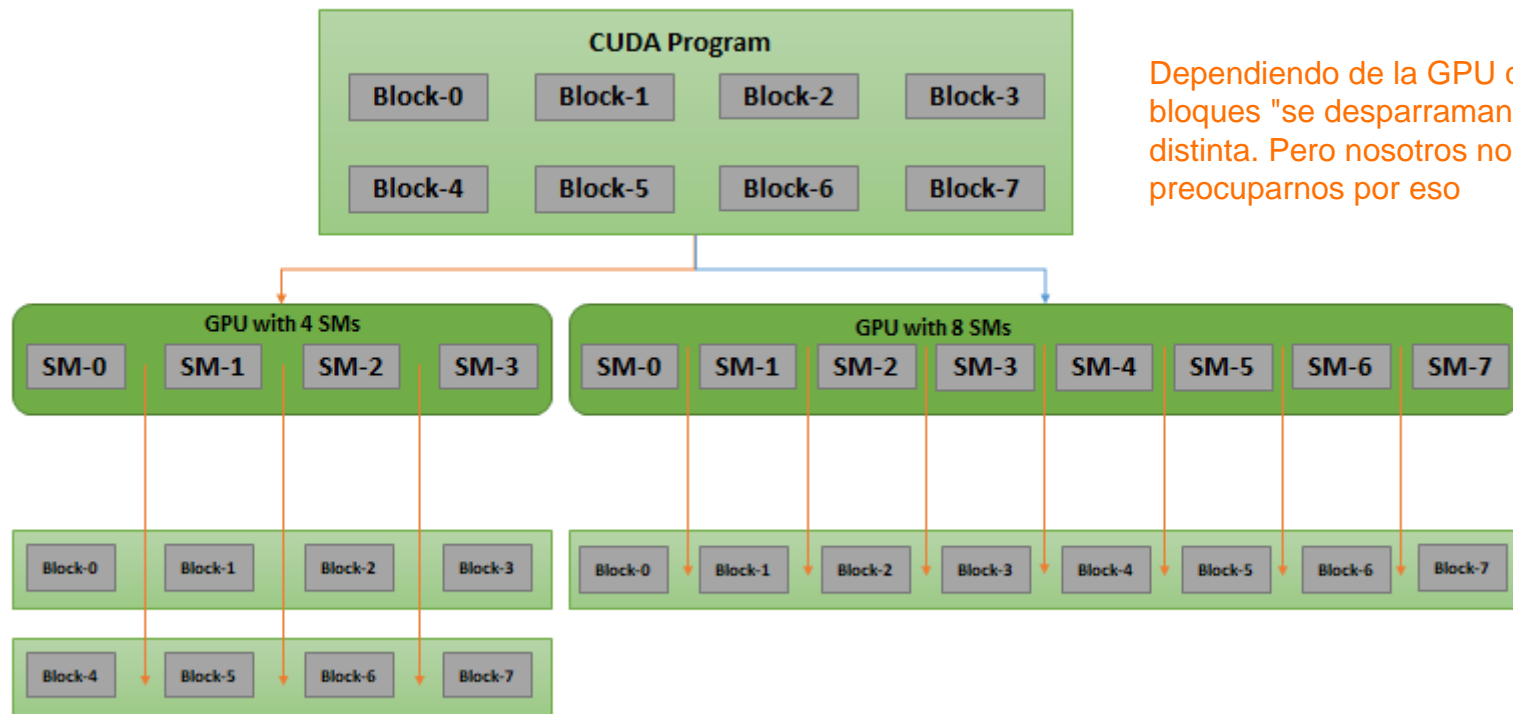


Complete GPU Unit



Escaleo

Mas cores en la GPU más rápido corre el *mismo* programa



Dependiendo de la GPU que se use los bloques "se desparraman" de forma distinta. Pero nosotros no tenemos que preocuparnos por eso

Jerarquía de memorias

Cada hilo hace una operación.
Cada uno tiene asociada una función (lo que hace) y una memoria privada (donde están los datos). Esta memoria es muy rápida

Los bloques de hilo tienen tmb su memoria compartida

Software

Más rápidas/chicas
Más privadas

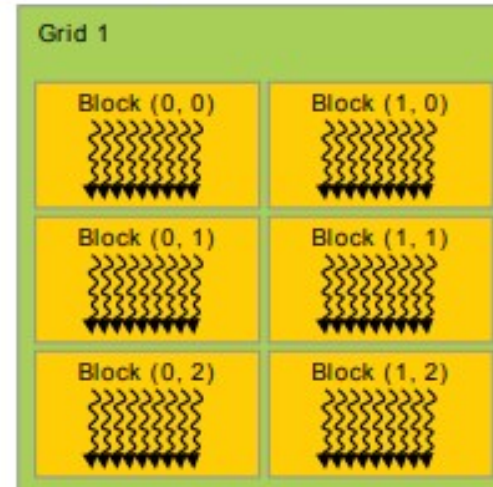
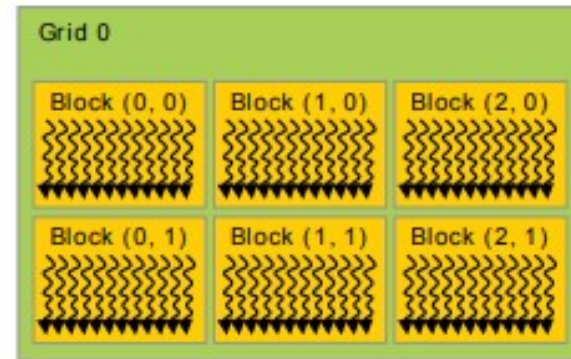
Thread

Per-thread local memory

Thread Block

Per-block shared memory

Hardware

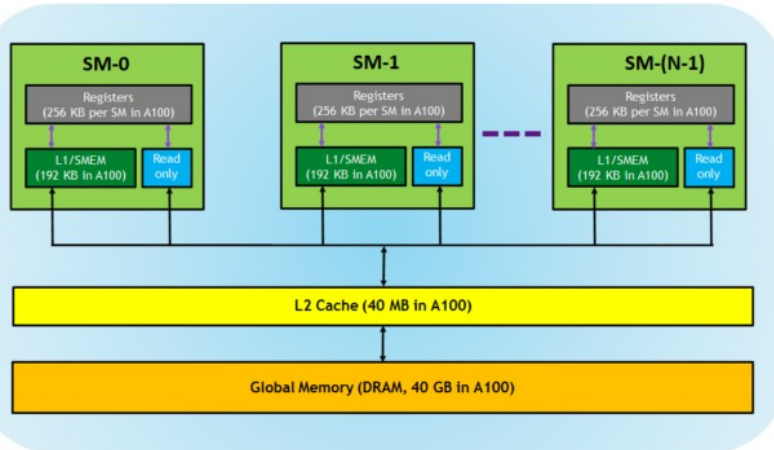


Global memory

Solo puede acceder a ella la GPU, pero la CPU no puede acceder a esta memoria global si los hilos están trabajando. La CPU tiene que esperar a que una función interna de la GPU lleve los datos a la CPU

Figure 7 Memory Hierarchy

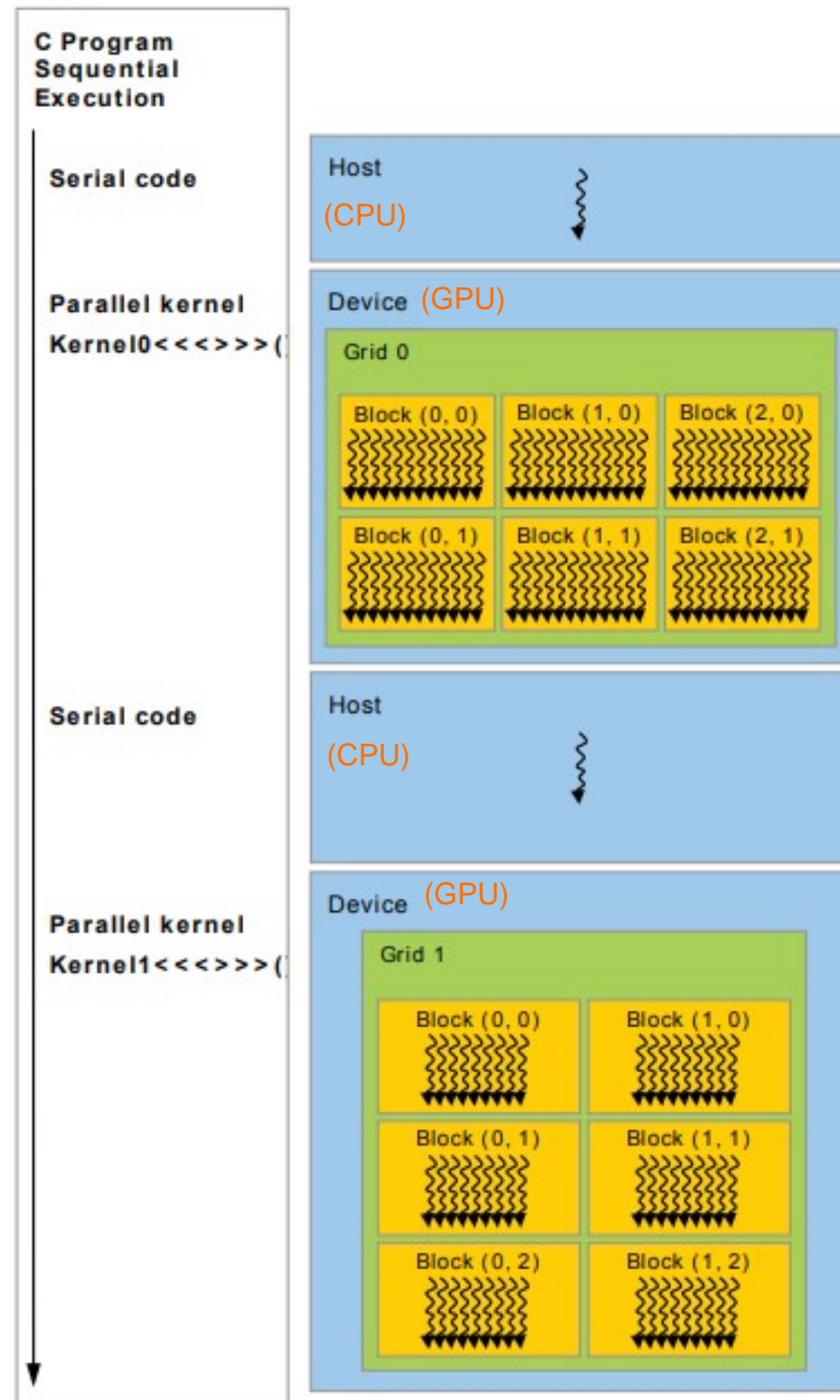
Más lentas/grandes
Mas públicas



Programación heterogénea

La mejor estrategia es analizar dónde se pierde más tiempo y ver si es paralelizable

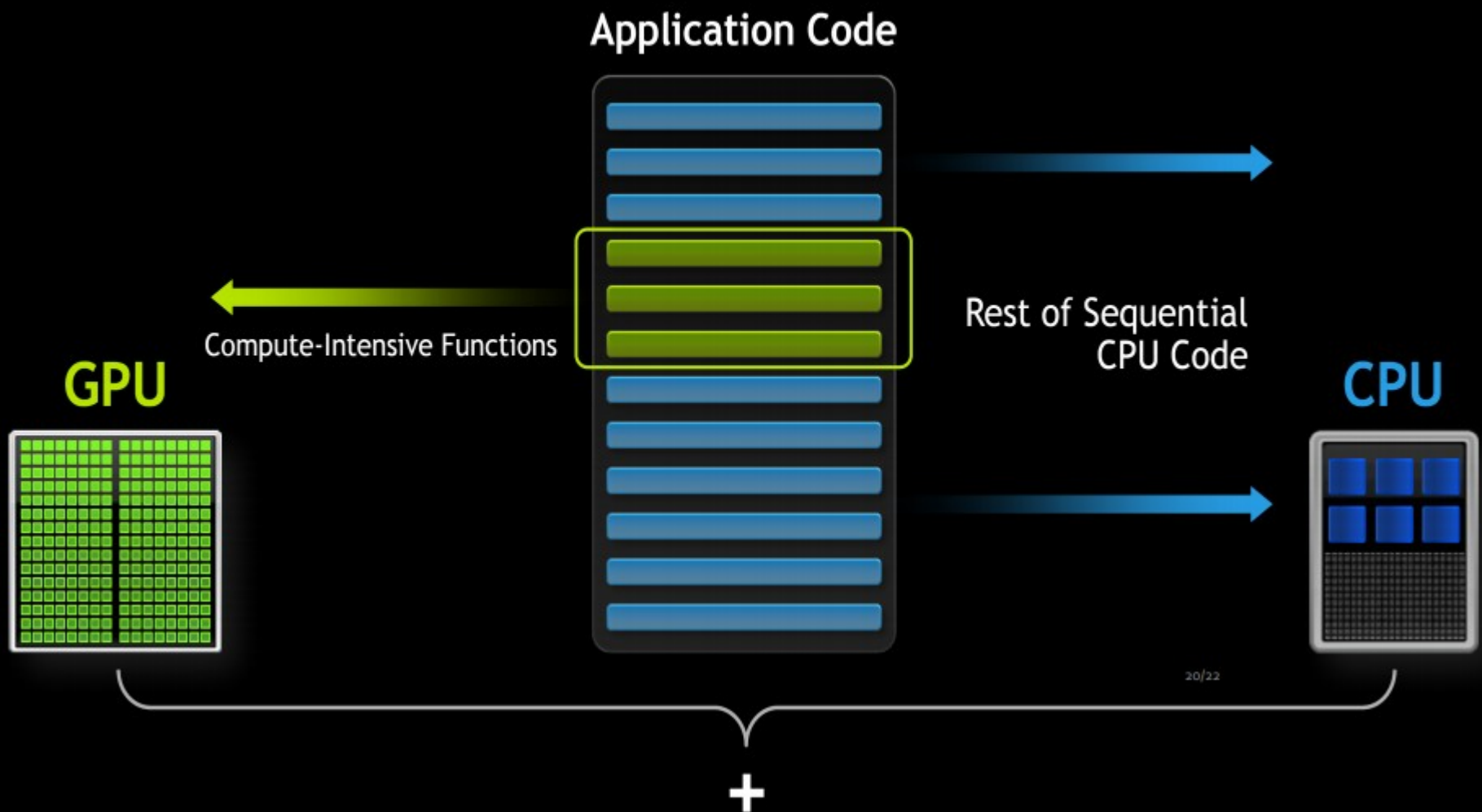
Un código, 2 devices
CPU+GPU



Mientras se ejecuta código en la GPU podría seguir trabajando en paralelo en la CPU

Programacion híbrida... que vale la pena

Minimum Change, Big Speed-up



¡Manos a la obra!

Plan para hoy

- **Cluster de GPUs**

- Acceso a su cuenta vía ssh. Terminal linux.
- Línea de comandos linux básica. Editores.
- Compilación: nvcc, gcc. Makefiles.
- Sistema de colas: qsub, qstat, qdel
- Edición de códigos remota.

- **CUDA C/C++ Básico**

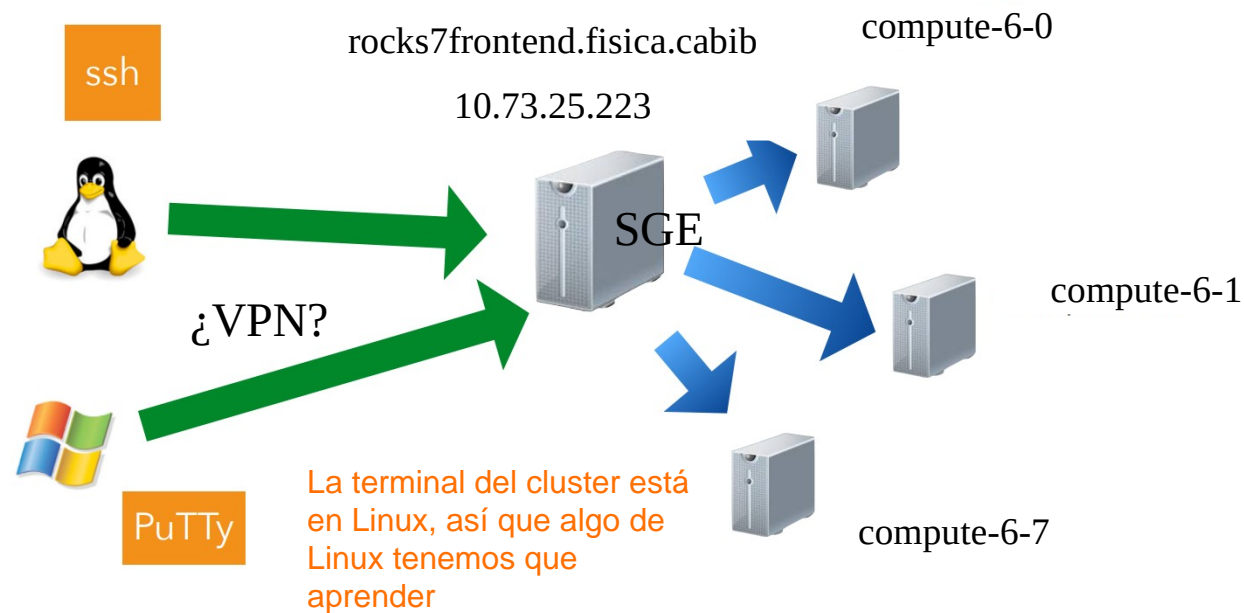
- Hola Mundo
- Kernels, hilos y grillas
- Device query
- Suma de vectores
- Profiling
- Ejercicios

- **Google colab**

- Correr los ejemplos

`ssh usuario@rocks7frontend.fisica.cabib / ssh usuario@10.73.25.223`

Acceso al cluster



sysadmin: Gustavo Berman

soporte.fisica@cab.cnea.gov.ar

Controle el espacio de almacenamiento en su home: `$ du -hs ~`

ssh username@10.73.25.223 ... ¿y ahora?

```
ale@coltrane:~$ ssh koltona@10.73.25.223
Last login: Fri Mar 31 15:15:08 2023 from 10.73.16.104
Rocks 7.0 (Manzanita)
Profile built 14:12 18-Oct-2018

Kickstarted 14:46 18-Oct-2018

#####
*****          INFORMACION IMPORTANTE          *****
En este sistema se utiliza SGE como administrador de trabajos.
Los compiladores, bibliotecas y herramientas se puede acceder a
traves del comando module
module avail          (muestra los modulos disponibes)
module load  <MODULE> (carga el modulo indicado)
module purge          (descarga todos los modulos cargados)
*****          Ejemplos de uso del cluster en          *****
*****          https://fisica.cab.cnea.gov.ar/utilizacion *****
*****          Y en /opt/ejemplos          *****
!!!!!!  Email de soporte: soporte.fisica@cab.cnea.gov.ar  !!!!!
#####

Disk quotas for user koltona (uid 346000032):
   Filesystem    space   quota   limit   grace   files   quota   limit   grace
nas-0-0:/export/data2
                432G    600G    800G          10845k         0         0
(base)
```

Leer!!!

<https://fisica.cab.cnea.gov.ar/utilizacion>

Línea de comandos de Linux: tipo “bash”

Si no está familiarizado con linux, en la terminal del cluster, hacer “man xxxxx”, donde xxxxx es algun comando de estos

- ls
- cd
- mkdir
- rm
- nano
- vi
- ssh
- exit
- man
- cat
- more
- less
- top
- htop
- Pwd
- Scripts...

Muchos Tutoriales



linux command line bash tutorials

Que hay en el cluster de GPUs del CAB?

(base) qghost											
HOSTNAME	ARCH	NCPU	NSOC	NCOR	NTHR	LOAD	MEMTOT	MEMUSE	SWAPTO	SWAPUS	
global	-	-	-	-	-	-	-	-	-	-	-
compute-0-0	lx-amd64	8	2	8	8	0.04	15.5G	394.4M	23.6G	0.0	
compute-0-1	lx-amd64	8	2	8	8	1.79	15.5G	3.0G	7.9G	0.0	
compute-0-2	lx-amd64	8	2	8	8	0.02	15.5G	370.4M	7.9G	0.0	
compute-0-3	lx-amd64	8	2	8	8	0.03	15.5G	398.9M	23.6G	0.0	
compute-0-4	lx-amd64	8	2	8	8	2.45	15.5G	3.1G	7.9G	0.0	
compute-0-5	lx-amd64	8	2	8	8	0.02	15.5G	394.7M	23.6G	0.0	
compute-0-6	lx-amd64	8	2	8	8	0.03	15.5G	378.7M	23.6G	0.0	
compute-0-7	lx-amd64	8	2	8	8	0.02	15.5G	392.2M	23.6G	0.0	
compute-3-0	lx-amd64	8	2	8	8	0.02	23.3G	488.5M	35.3G	0.0	
compute-3-1	lx-amd64	8	2	8	8	0.01	23.3G	493.9M	35.3G	0.0	
compute-3-10	lx-amd64	8	2	8	8	0.02	23.3G	496.8M	35.3G	0.0	
compute-3-11	lx-amd64	8	2	8	8	0.01	23.3G	494.7M	35.3G	0.0	
compute-3-2	lx-amd64	8	2	8	8	0.03	23.3G	488.6M	35.3G	0.0	
compute-3-3	lx-amd64	8	2	8	8	0.03	23.3G	494.3M	35.3G	0.0	
compute-3-4	lx-amd64	8	2	8	8	0.04	11.5G	377.2M	17.6G	0.0	
compute-3-5	lx-amd64	8	2	8	8	0.02	11.5G	371.6M	17.6G	0.0	
compute-3-6	lx-amd64	8	2	8	8	0.03	23.3G	490.0M	35.3G	0.0	
compute-3-7	lx-amd64	8	2	8	8	0.01	23.3G	494.2M	35.3G	0.0	
compute-3-8	lx-amd64	8	2	8	8	0.02	23.3G	493.1M	35.3G	0.0	
compute-3-9	lx-amd64	8	2	8	8	0.04	23.3G	493.1M	35.3G	0.0	
compute-4-0	lx-amd64	16	2	16	16	0.01	31.2G	629.2M	47.2G	0.0	
compute-4-1	lx-amd64	16	2	16	16	0.03	54.8G	939.4M	94.4G	0.0	
compute-4-10	lx-amd64	16	1	16	16	0.02	62.4G	963.9M	93.9G	0.0	
compute-4-11	lx-amd64	16	1	16	16	0.06	62.4G	964.8M	93.9G	0.0	
compute-4-12	lx-amd64	10	1	10	10	0.03	125.4G	1.4G	4.0G	0.0	
compute-4-13	lx-amd64	20	2	20	20	-	125.4G	-	4.0G	-	
compute-4-14	lx-amd64	32	2	32	32	0.01	188.4G	5.3G	4.0G	0.0	
compute-4-15	lx-amd64	32	2	32	32	0.02	188.4G	5.3G	4.0G	0.0	
compute-4-16	lx-amd64	32	2	32	32	0.01	188.4G	5.3G	4.0G	0.0	
compute-4-17	lx-amd64	32	2	32	32	0.01	188.4G	5.3G	4.0G	0.0	
compute-4-18	lx-amd64	32	2	32	32	0.03	188.4G	5.3G	4.0G	0.0	
compute-4-2	lx-amd64	16	2	16	16	0.02	62.7G	989.1M	94.4G	0.0	
compute-4-3	lx-amd64	40	2	20	40	0.01	94.2G	1.4G	4.0G	0.0	
compute-4-4	lx-amd64	20	2	20	20	0.02	188.7G	3.0G	4.0G	121.6M	
compute-4-5	lx-amd64	20	2	20	20	0.01	251.6G	3.6G	94.4G	120.8M	
compute-4-6	lx-amd64	20	2	20	20	0.02	125.7G	1.5G	4.0G	0.0	
compute-4-7	lx-amd64	20	2	20	20	0.02	30.9G	6.1G	46.7G	2.6G	
compute-4-8	lx-amd64	20	2	20	20	0.01	31.1G	640.7M	47.1G	211.2M	
compute-4-9	lx-amd64	16	1	16	16	0.01	62.4G	956.2M	93.9G	0.0	
compute-6-0	lx-amd64	8	1	4	8	0.24	23.4G	4.2G	35.4G	0.0	
compute-6-1	lx-amd64	8	1	4	8	1.04	31.2G	1.4G	47.3G	0.0	
compute-6-10	lx-amd64	12	2	12	12	0.02	92.9G	4.6G	4.0G	0.0	
compute-6-2	lx-amd64	4	1	4	4	0.02	11.6G	926.7M	17.6G	0.0	
compute-6-3	lx-amd64	16	2	16	16	0.13	31.2G	1.2G	47.2G	0.0	
compute-6-4	lx-amd64	16	2	16	16	0.04	31.2G	1.3G	47.3G	0.0	
compute-6-5	lx-amd64	32	2	16	32	0.27	31.2G	6.8G	47.2G	0.0	
compute-6-6	lx-amd64	28	2	28	28	0.35	15.4G	1.2G	23.4G	0.0	
compute-6-7	lx-amd64	28	2	28	28	0.22	15.4G	1.0G	23.4G	0.0	
compute-6-8	lx-amd64	12	2	12	12	0.03	62.4G	1.5G	93.9G	0.0	
compute-6-9	lx-amd64	12	2	12	12	0.03	92.9G	4.8G	4.0G	0.0	

Solo CPUs

CPUs y GPUs

Que hay en el cluster de GPUs del CAB?

```
(base) rocks run host compute-6-% command="nvidia-smi -L " collate=yes| sort
compute-6-0: GPU 0: GeForce RTX 2080 Ti (UUID: GPU-e445d9be-0ed4-0cd6-d6a8-2241b0b25d98)
compute-6-10:
compute-6-10: NVIDIA-SMI has failed because it couldn't communicate with the NVIDIA driver. Make
compute-6-1: GPU 0: NVIDIA GeForce GTX 1080 Ti (UUID: GPU-2506bdf5-fa2a-44c9-d2a1-6825ef62a08c)
compute-6-2: GPU 0: NVIDIA GeForce GTX TITAN X (UUID: GPU-b428df97-2ca3-a8c2-0c86-e9ca5351e7a3)
compute-6-3: GPU 0: NVIDIA GeForce RTX 2070 (UUID: GPU-8a4a9a30-98f3-91b6-508b-52b4da956574)
compute-6-3: GPU 1: NVIDIA GeForce RTX 2070 (UUID: GPU-bc5cd294-bf56-5549-b001-d53324fe4bb9)
compute-6-4: GPU 0: NVIDIA GeForce RTX 2080 Ti (UUID: GPU-959e4ef1-35a5-bcf9-b7dc-f2c9374a980d)
compute-6-5: GPU 0: GeForce RTX 2070 (UUID: GPU-4f46469b-c32b-9f61-f9c9-a6fff97eefc)
compute-6-6: GPU 0: Tesla K20Xm (UUID: GPU-d8da7ebe-f8f9-645d-6e7b-88776d25ede3)
compute-6-6: GPU 1: Tesla K20Xm (UUID: GPU-93cad715-5b8e-220e-76b4-5a477d6430ec)
compute-6-7: GPU 0: Tesla K20Xm (UUID: GPU-0afae843-73c4-e2c5-03ec-d5fbd48d56d8)
compute-6-7: GPU 1: Tesla K20Xm (UUID: GPU-e2e754a7-e502-c044-6d4f-c460d94bc1f8)
compute-6-8: GPU 0: NVIDIA GeForce RTX 3080 (UUID: GPU-e7f77454-3e7a-b62b-fe1d-88f8419216dd)
compute-6-8: GPU 1: NVIDIA GeForce RTX 3080 Ti (UUID: GPU-58756a66-fff0-2549-6f85-ebcec9211929)
compute-6-9: GPU 0: NVIDIA GeForce RTX 3080 (UUID: GPU-6c89e3cc-8fd5-900c-a64f-426832b435bb)
compute-6-9: GPU 1: NVIDIA GeForce RTX 3080 (UUID: GPU-b59bdb32-e9c1-dcf3-a018-31f972d77d77)
compute-6-9: GPU 2: NVIDIA GeForce RTX 3080 Ti (UUID: GPU-8c292c9b-a0ea-45fe-6107-0fa5f653e934)
```



GPUs para “jueguitos”
(tienen salida HDMI)

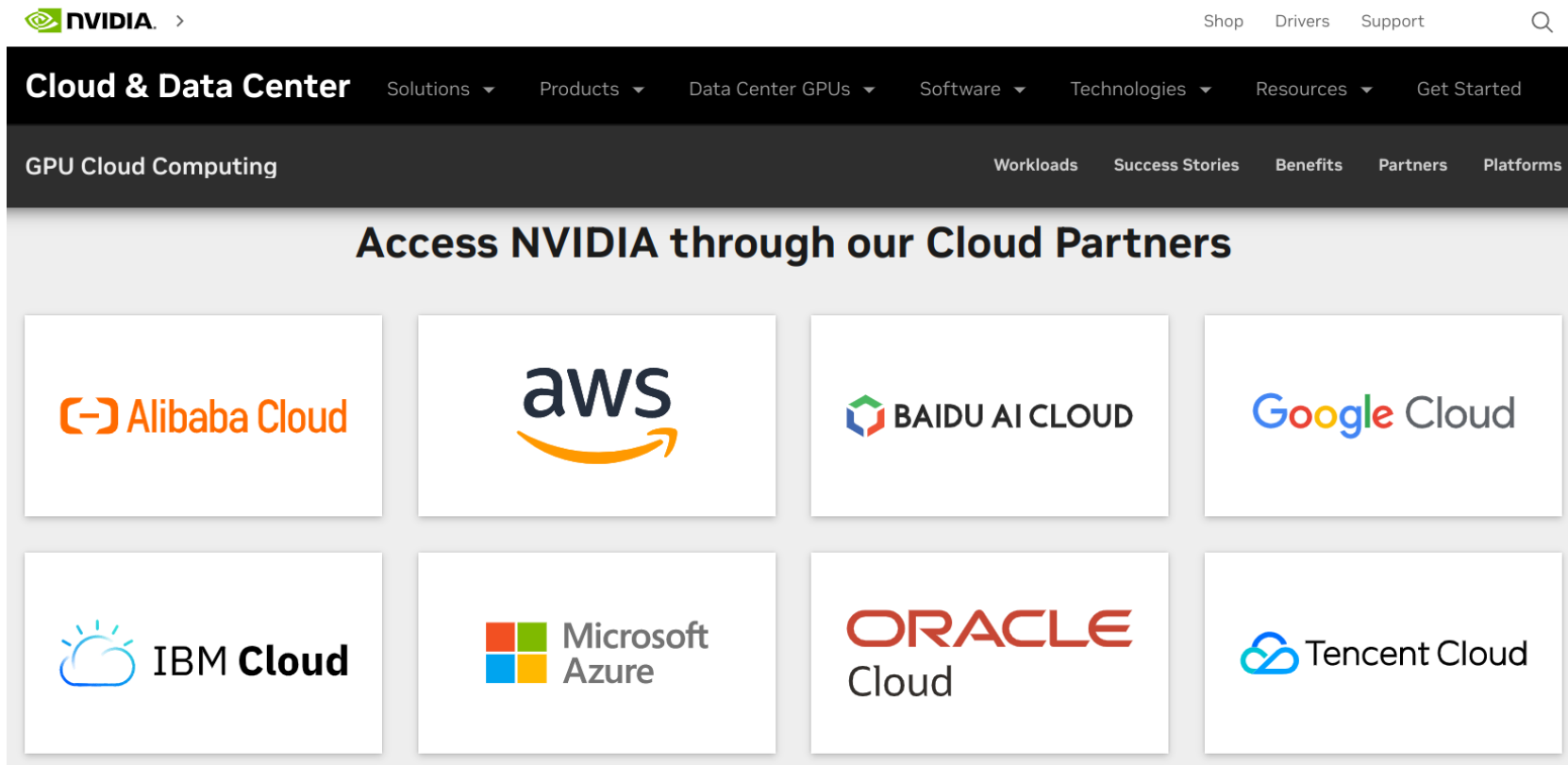
Visita al cluster

Computación de alto desempeño → Aprender a usar clusters



Los centros de investigación de todo el mundo en general tienen acceso a clusters de cómputo, que usan software muy parecido.

<https://www.argentina.gob.ar/ciencia/sistemasnacionales/computacion-de-alto-desempeno>



Dilema...

Sabiendo que existen librerías y lenguajes de más alto nivel y que cada día hay mas aplicaciones que usan GPU...

¿Vale la pena aprender a programar en CUDA C/C++?

- No es necesario para empezar a programar GPUS.
- Cuda C es el lenguaje nativo para las GPUs de NVIDIA.
- Es útil para tener un entendimiento más profundo.
- Hay detalles que solo se pueden acceder con CUDA C/C++.
- Hay optimizaciones que por el momento solo son posibles con CUDA C/C++.
- Los conceptos son siempre los mismos, no importa el lenguaje o framework.
- Siempre se puede hacer algo híbrido, entre alto (interface) y bajo (optimización) nivel.

Códigos de la clase 1

```
$ cp -r /share/apps/icnpg/clases/clase1 .
```


Hola Mundo

```
#include <stdio.h>
int main(void)
{
    printf("Hello World from CPU!\n");
}
```

- Compilación (dos opciones):

\$ **nvcc** hello.**cu** -o hello

Compilador CUDA C/C++

- Correr (tres opciones):

\$./hello

nvcc

El compilador nvcc compila cualquier código normal

Hola Mundo: ¡Primer kernel!

```
#include "common.h"
#include <stdio.h>

__global__ void helloFromGPU()
{
    printf("hola mundo desde la GPU!\n");
}

int main(int argc, char **argv)
{
    printf("Hola mundo desde la CPU!\n\n");

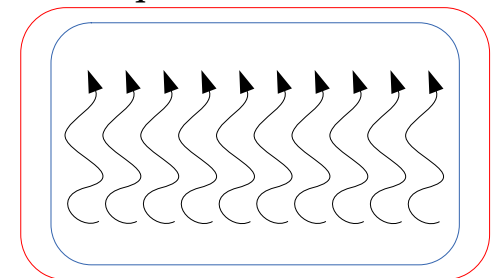
    helloFromGPU<<<1, 10>>>();

    CHECK(cudaDeviceSynchronize());
    return 0;
}
```

kernel

lanzamiento

La grilla:
1 bloque con 10 hilos



Hola Mundo

\$ make

nvcc -o hello hello.cu

\$ make run

???

\$ make submit

nvcc -o hello hello.cu

????

\$ qsub jobGPU

Your job 155838 ("holamundo") has been submitted

\$ qstat

???

\$ make submitwatch

???

\$ cat holamundo.o1558838

El cluster y su sistema de colas

https://es.wikipedia.org/wiki/Rocks_Clusters

\$ **man qstat**

???

Nuestros nodos con GPUs

\$ **man qsub**

???

\$ **man qdel**

???

\$ **qstat**

???

\$ **qstat -f**

???

\$ **qstat -f -u “*”**

???

gpu@compute-6-0.local	BIP	0/0/4	-NA-	lx-amd64	
auE					
gpu@compute-6-1.local	BIP	0/1/4	-NA-	lx-amd64	au
gpu@compute-6-2.local	BIP	0/1/4	1.01	lx-amd64	
gpu@compute-6-4.local	BIP	0/2/16	1.96	lx-amd64	
gpu@compute-6-5.local	BIP	0/0/16	0.26	lx-amd64	
gpu@compute-6-6.local	BIP	0/1/28	1.01	lx-amd64	
gpu@compute-6-7.local	BIP	0/1/28	1.01	lx-amd64	
gpushort@compute-6-3.local	BIP	0/0/16	0.07	lx-amd64	

Línea de comandos de Linux: “bash”

Si no está familiarizado con la línea de comandos de linux, en la terminal del cluster, hacer “man xxxxx”, donde xxxxx es algun comando de estos.

- ls
- cd
- mkdir
- rm
- nano
- vi
- ssh
- exit
- man
- cat
- more
- less
- top
- htop
- pwd

Muchos Tutoriales



linux command line bash tutorials

Kernels y sus grillas

- Programar en paralelo en GPUs implica **mapear índices de hilos a datos**.
- Para eso, los kernels disponen de variables reservadas, identificadoras de los hilos.
- El indexado de hilos se organiza en grillas-de-bloques-de-hilos, de 1d, 2d, o 3d.

- A thread block is a batch of threads that can cooperate with each other by:
 - Synchronizing their execution
 - For hazard-free shared memory accesses
 - Efficiently sharing data through a low-latency shared memory
- Two threads from two different blocks cannot cooperate

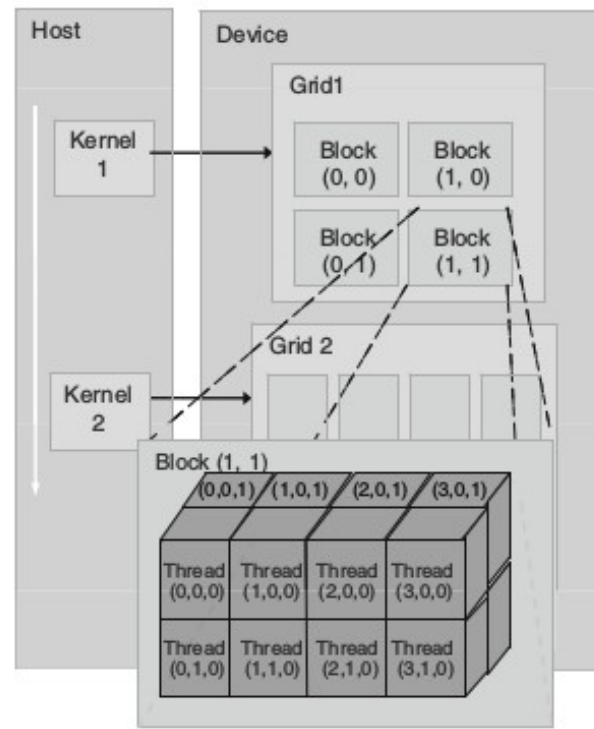


FIGURE 3.13

CUDA thread organization.

Indexado de hilos

- **\$ less grillas.cu**
- **\$ make submit** (cluster)
- **\$ make run** (su maquina)

Indices de hilo en el bloque

Indices de bloque en la grilla

threadIdx, blockIdx
blockDim, gridDim

Dimensiones de bloque

Dimensiones de grilla

ejemplo 1: Quiensoy<<< 3, 2>>>

Soy el thread (0,0,0) del bloque (0,0,0) [blockDim=(2,1,1),gridDim=(3,1,1)]

Soy el thread (1,0,0) del bloque (0,0,0) [blockDim=(2,1,1),gridDim=(3,1,1)]

Soy el thread (0,0,0) del bloque (1,0,0) [blockDim=(2,1,1),gridDim=(3,1,1)]

Soy el thread (1,0,0) del bloque (1,0,0) [blockDim=(2,1,1),gridDim=(3,1,1)]

Soy el thread (0,0,0) del bloque (2,0,0) [blockDim=(2,1,1),gridDim=(3,1,1)]

Soy el thread (1,0,0) del bloque (2,0,0) [blockDim=(2,1,1),gridDim=(3,1,1)]

ejemplo 2: Quiensoy<<< dim3(2,2), dim3(2,1) >>>();

Soy el thread (0,0,0) del bloque (0,0,0) [blockDim=(2,1,1),gridDim=(2,2,1)]

Soy el thread (1,0,0) del bloque (0,0,0) [blockDim=(2,1,1),gridDim=(2,2,1)]

Soy el thread (0,0,0) del bloque (1,0,0) [blockDim=(2,1,1),gridDim=(2,2,1)]

Soy el thread (1,0,0) del bloque (1,0,0) [blockDim=(2,1,1),gridDim=(2,2,1)]

Soy el thread (0,0,0) del bloque (1,1,0) [blockDim=(2,1,1),gridDim=(2,2,1)]

Soy el thread (1,0,0) del bloque (1,1,0) [blockDim=(2,1,1),gridDim=(2,2,1)]

Soy el thread (0,0,0) del bloque (0,1,0) [blockDim=(2,1,1),gridDim=(2,2,1)]

Soy el thread (1,0,0) del bloque (0,1,0) [blockDim=(2,1,1),gridDim=(2,2,1)]

Grillas

- **dim3** es una variable tipo vector tridimensional.
- **Inicialización sobrecargada:**
 - `dim3 n; n.x=7; nx=8; nz=1; // 1) declaración, 2) inicialización`
 - `dim3 n(7,8,1); //declaración+inicialización`
 - `dim3 n(7,8); //sobreentiende que n.z=1`
- **Todos estos lanzamientos son equivalentes:**
 - `dim3 nb(4,1,1); dim3 nt(3,1,1); Quiensoy<<<nb, nt>>>();`
 - `Quiensoy<<< dim3(4,1,1), dim3(3,1,1) >>>();`
 - `Quiensoy<<< 4,3 >>>();`
 - `Quiensoy<<< dim3(4,1), dim3(3,1) >>>();`

Grillas (más simple)

```
$ cp -r /share/apps/icnpg/clase1/grillas_simple .
```

```
$ cd grillas_simple; ls
```

```
$ nvcc grillas.cu -o grillas
```

```
$ qsub jobGPU (cluster)
```

```
$ ./grillas (local)
```

jobGPU

```
#$ -cwd  
#$ -j y  
#$ -S /bin/bash  
#$ -q gpushort  
#$ -l gpu=1  
#$ -l memoria_a_usar=1G  
#$ -N Grillas  
#
```

#ejecutar el o los binarios con sus respectivos argumentos
./grillas

```
#include <stdio.h>  
  
// kernel  
__global__ void Quiensoy()  
{  
    printf("Soy el thread (%d,%d,%d) del bloque (%d,%d,%d) [blockDim=(%d,%d,%d),gridDim=(%d,%d,%d)]  
\\n",threadIdx.x,threadIdx.y,threadIdx.z,blockIdx.x,blockIdx.y,blockIdx.z,  
blockDim.x,blockDim.y,blockDim.z,gridDim.x,gridDim.y,gridDim.z);  
}  
  
int main()  
{  
    //TODO: pruebe distintas grillas  
  
    //ejemplo1: 4 blocks, y 3 threads/block:  
    dim3 nb(4,1,1); dim3 nt(3,1,1);  
    Quiensoy<<< nb, nt>>>();  
  
    // espera a que los threads hayan terminado  
    cudaDeviceSynchronize();  
    return 0;  
}
```

GPUs en la nube

- Google colaboratory <https://colab.research.google.com/> (ver tutorials)
- Usar y compartir Notebooks (similar a jupyter notebook)
- Notebooks con Markdown (latex, secciones, figuras, animaciones, html, etc)
- Cambiar runtime (CPU → GPU)
- Probar distintos lenguajes de programación
- Conexión con drive/github



Ordenando las frecuencias tenemos entonces

$$\begin{aligned}\omega_0 &= 0, \\ \omega_1 &= \sqrt{k/m}, \\ \omega_2 &= \sqrt{k/m + 2k/M}\end{aligned}$$

Podemos entonces ya plantear las ecuaciones para los autovectores correspondientes. Pero para o *mathematica* para controlar y completar las cuentas y de paso aprendemos a usarlos.

Double-click (or enter) to edit

• Con sympy necesitamos cargar esto al principio

```
[ ] 1 #@title Con sympy necesitamos cargar esto al principio
    2 from sympy import *
    3 init_printing(use_unicode=True)
```

Definimos los símbolos para las masas y la matriz M, a la que le llamamos "masas"

• Armate la matriz de masas \mathbb{M}

```
[ ] 1 #@title Armate la matriz de masas  $\mathbb{M}$ 
    2
    3 m = Symbol("m", positive = True)
    4 M = Symbol("M", positive = True)
    5
    6 masas = Matrix([[m, 0, 0], [0, M, 0], [0, 0, m]])
    7 masas
```

$$\begin{bmatrix} m & 0 & 0 \\ 0 & M & 0 \\ 0 & 0 & m \end{bmatrix}$$

Experimentar

- Modificar el programa hello.cu
 - Para que imprima "hola mundo desde la GPU desde el thread número ...”
 - Largar mas hilos en un bloque y ver que pasa...
 - Para que imprima "hola mundo desde la GPU!” solo si es el quinto hilo del bloque.
 - Comentar la línea CHECK(cudaDeviceSynchronize()), ver que pasa...
 - Largar varios bloques

Device query

- A veces es útil que el programa, al empezar a correr conozca las propiedades y limitaciones de la placa (memoria, compute capability, etc).
- El cuda runtime provee funciones y estructuras para entrevistar a la gpu...

```
#include <stdio.h>

int main(int argc, char **argv)
{
    cudaDeviceProp deviceProp;

    int deviceCount = 0;
    cudaError_t error_id = cudaGetDeviceCount(&deviceCount);

    printf("En este nodo hay %d placas\n\n", deviceCount);
    for(int dev=0; dev<deviceCount; dev++){
        cudaSetDevice(dev);
        cudaGetDeviceProperties(&deviceProp, dev);
        printf("Hola!, yo soy [Device %d: \"%s\"], tu acelerador grafico personal\n", dev, deviceProp.name);
    }

    int dev; cudaGetDevice(&dev);
    printf("\nle asigno la device %d, que esta desocupada\n", dev);

    return 0;
}
```

Ejemplo completo

https://github.com/NVIDIA/cuda-samples/tree/master/Samples/1_Uutilities/deviceQuery

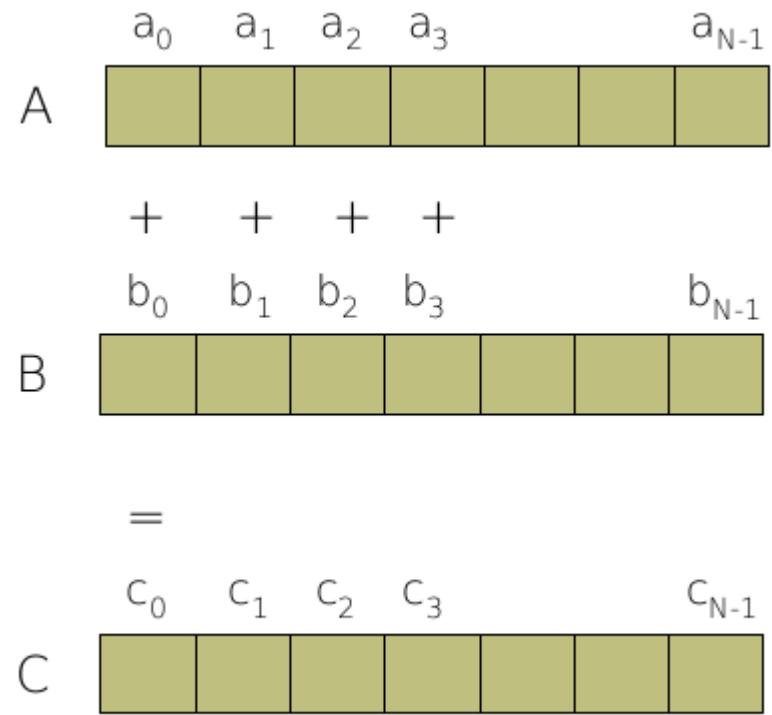
Hagamos algo útil con toda esa cuadrilla de hilos trabajando en paralelo

Suma de dos arrays

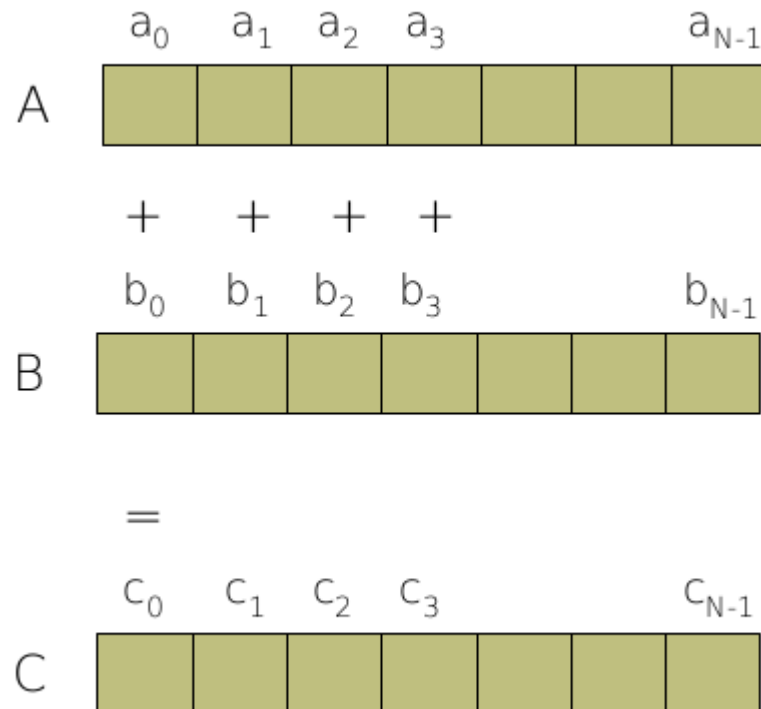
- Escribir un programa que dados dos arrays de N elementos **a** y **b**, los rellene con valores, y los sume tirando el resultado en un vector de N elementos **c=a+b**

1) Versión Secuencial en CPU

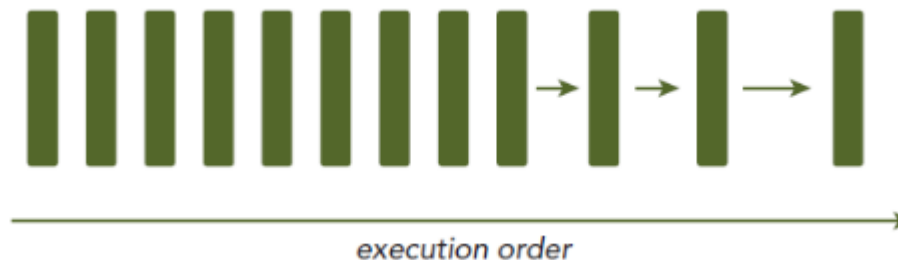
2) Versión Paralela en GPU



Suma secuencial de dos arrays



```
void VectorAdd(int *a, int *b, int *c, int n)
{
    for(int i=0; i<n; i++)
    {
        c[i] = a[i] + b[i];
    }
}
```



Sumar vectores en la CPU

- Ver código:

```
$less 0_suma_vectores_cpu.cpp
```

- Compilar:

```
$g++ 0_suma_vectores_cpu.cpp
```

```
$nvcc 0_suma_vectores_cpu.cpp
```

```
$make 0
```

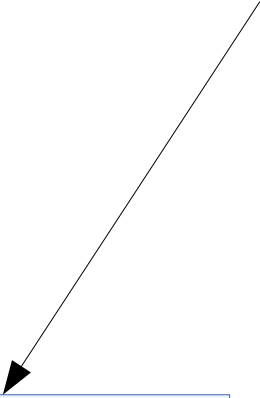
- Correr en el cluster:

```
$qsub jobGPU
```

- Correr en maquina con GPU:

```
$./a.out 1024
```

Los Makefiles nos permitirán
ahorrar mucho tiempo...



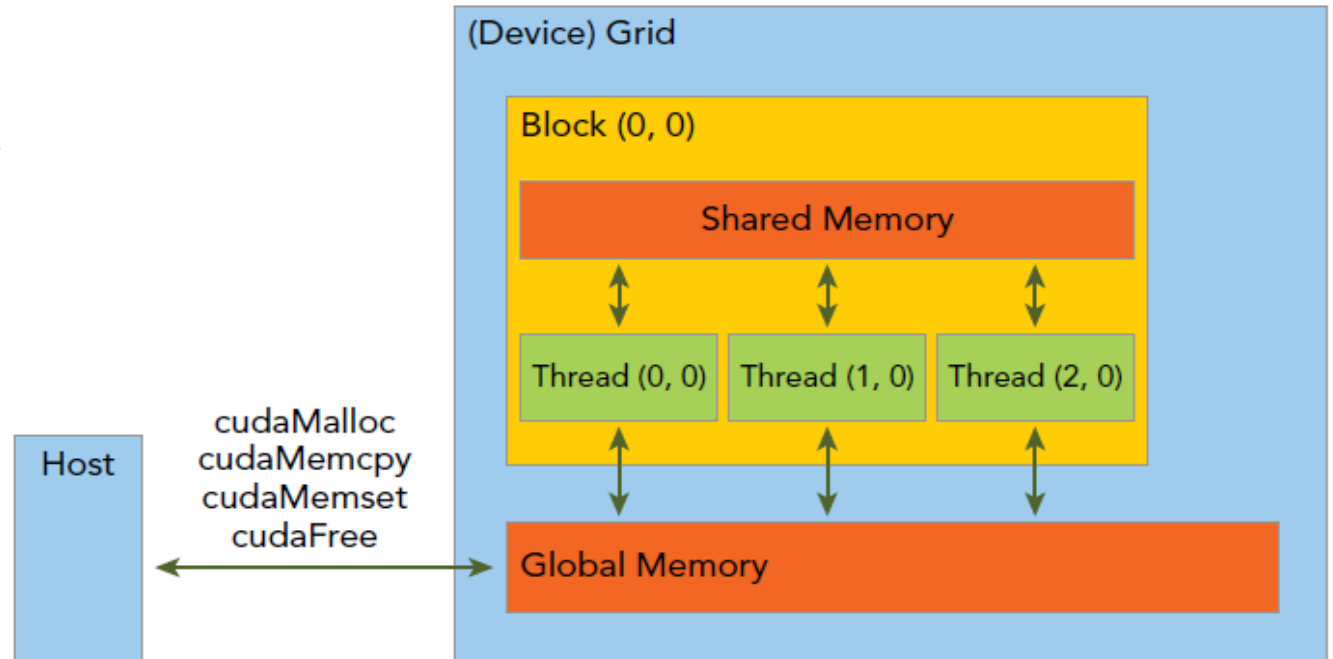
- Compilar y correr en un nodo del cluster:

```
$make qsub0 N=1024
```

- Compilar y correr en su máquina:

```
$make run0 N=1024
```


Memoria global



```
// allocacion memoria de device
cudaMalloc( &d_a, N*sizeof(int));
cudaMalloc( &d_b, N*sizeof(int));
cudaMalloc( &d_c, N*sizeof(int));
```

```
// copia de host a device
cudaMemcpy( d_a, a, N*sizeof(int), cudaMemcpyHostToDevice );
cudaMemcpy( d_b, b, N*sizeof(int), cudaMemcpyHostToDevice );
cudaMemcpy( d_c, c, N*sizeof(int), cudaMemcpyHostToDevice );
```

```
// copia (solo del resultado) del device a host
cudaMemcpy( c, d_c, N*sizeof(int), cudaMemcpyDeviceToHost );
```

```
// liberacion memoria de device
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);
```

Organización de los threads y los bloques elegí 1d, 2d o 3d para indexarlos

- A thread block is a batch of threads that can cooperate with each other by:
 - Synchronizing their execution
 - For hazard-free shared memory accesses
 - Efficiently sharing data through a low-latency shared memory
- Two threads from two different blocks cannot cooperate

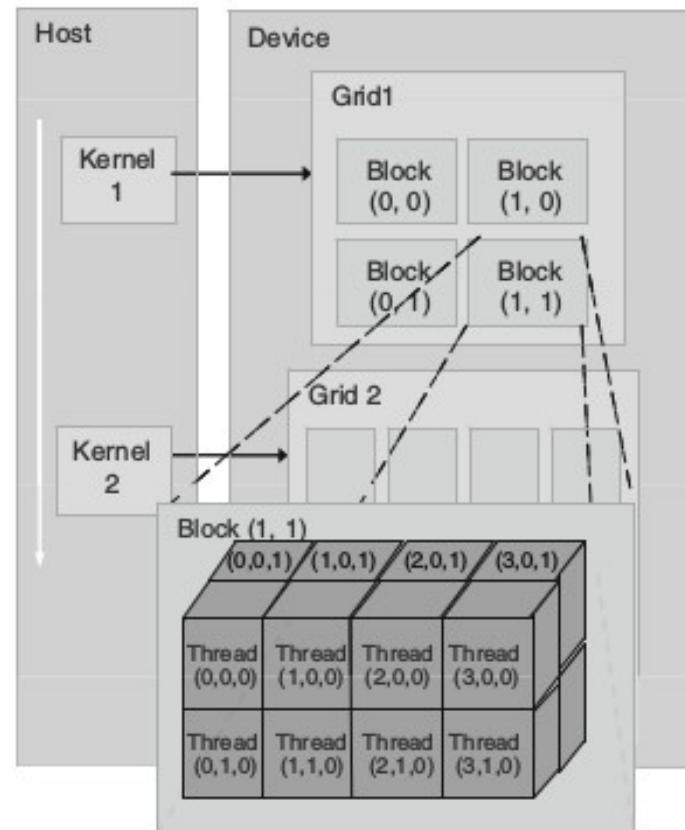


FIGURE 3.13

CUDA thread organization.

Mismo programa en cada thread pero sobre distintos datos

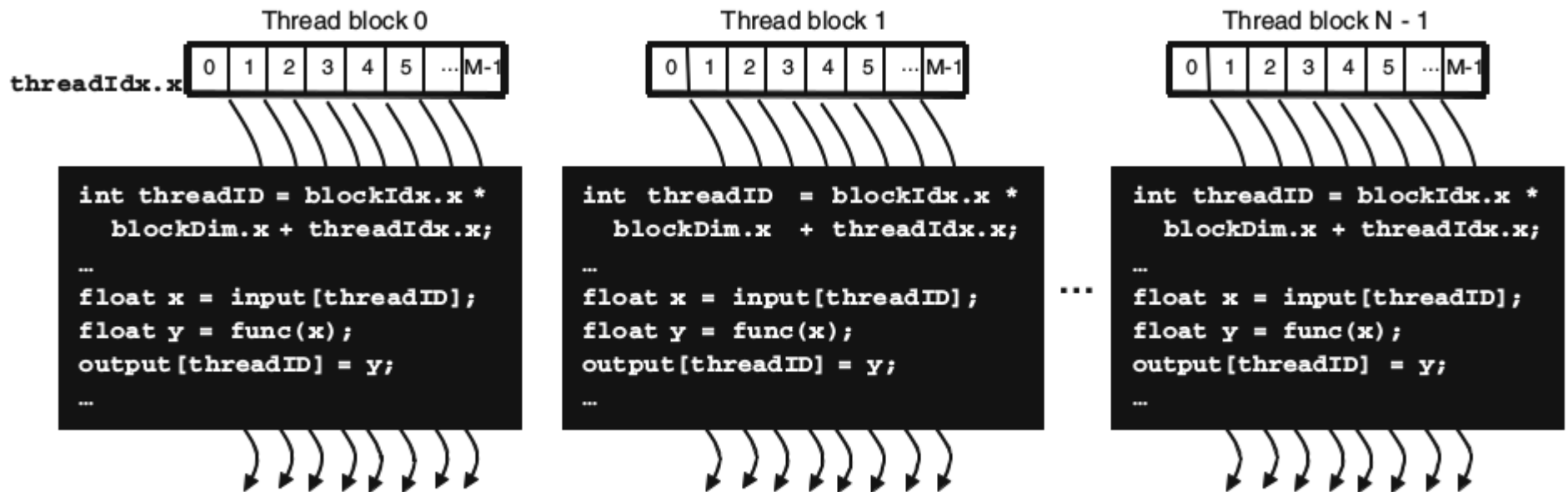


FIGURE 4.1

Overview of CUDA thread organization.

Threads del mismo bloque pueden cooperar: sincronizar, compartir “shared memory”, etc

Sumar vectores en la GPU: ¿1 bloque de N hilos?

- Ver código:

\$less 1_suma_vectores_gpu_1bl1024th_naive.cu

- Compilar y correr en el cluster:

\$make qsub1 N=1024

- Compilar y correr en su máquina:

\$make run1 N=1024

¿Qué pasa si cambio N?

```
// kernel
__global__ void VectorAdd(int *a, int *b, int *c, int n)
{
    int i = threadIdx.x;

    if (i < n)
        c[i] = a[i] + b[i];
}
```

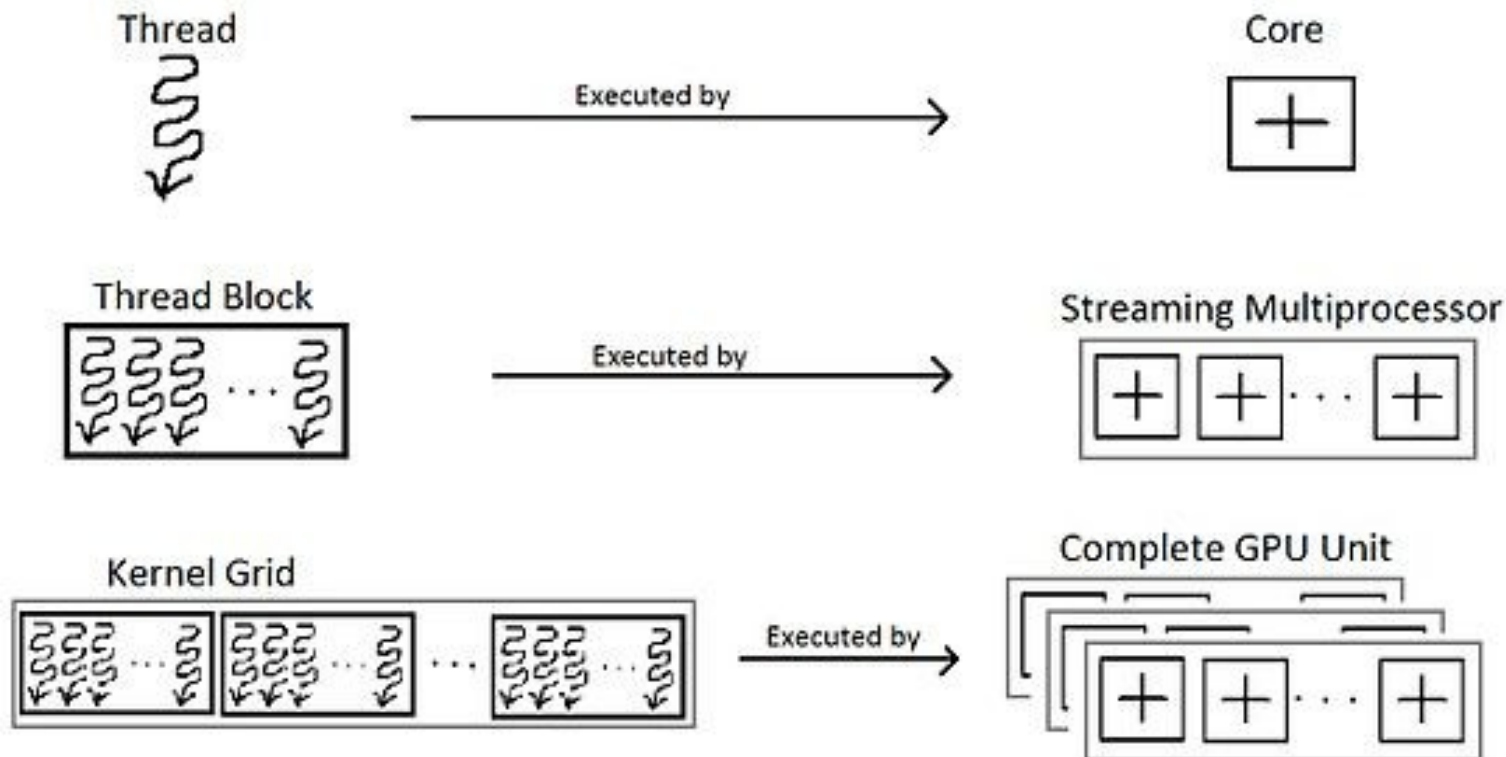
```
// suma paralela en el device
VectorAdd<<< 1, N >>>>(d_a, d_b, d_c, N);
```

Grillas permitidas



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):	(65535, 65535, 65535)

Guía 1: deviceQuery



Sumar vectores en la GPU: ¿1 bloques de N hilos?

- Ver código:

\$less 2_suma_vectores_gpu_1bl1024th_serializado.cu

- Compilar y correr en el cluster:

\$make qsub2 N=2048

¿Qué pasa si cambio N?

```
// kernel
__global__ void VectorAdd(int *a, int *b, int *c, int n)
{
    int i = threadIdx.x;

    while(i<n){
        c[i] = a[i] + b[i];
        i+=blockDim.x;
    }
}
```

```
// suma paralela en el device
VectorAdd<<< 1, N >>>>(d_a, d_b, d_c, N);
```

Sumar vectores en la GPU: ¿N bloques de 1 hilo?

- Ver código:
`$ 3_suma_vectores_gpu_muchos_bloques_un_thread_por_bloque.cu`

- Compilar y correr en el cluster:
`$ make qsub3 N=2048`

¿Qué pasa si cambio N?

```
// kernel
__global__ void VectorAdd(int *a, int *b, int *c, int n)
{
    int i = blockIdx.x;

    if(i < n){
        c[i] = a[i] + b[i];
    }
}
```

```
// suma paralela en el device
VectorAdd<<< 1, N >>>>(d_a, d_b, d_c, N);
```

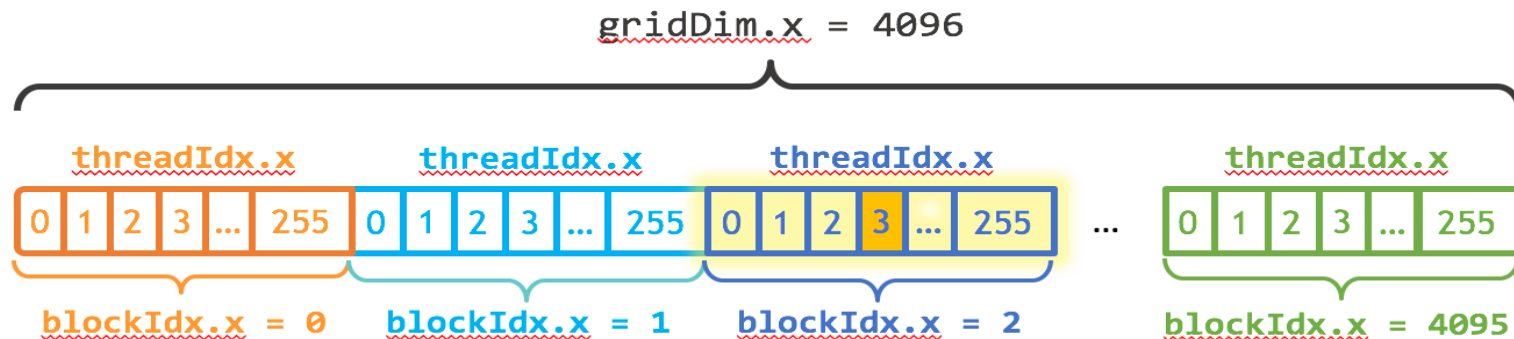
¿Cuál es el máximo número de hilos permitido?

Depende la placa

```
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)
```

- Grilla unidimensional de bloques unidimensionales: ??????
 - Grilla bidimensional de bloques unidimensionales: ??????
 - Grilla tridimensional de bloques unidimensionales: ??????
 - Etc...
 - Grilla tridimensional de bloques tridimensionales: ??????
-
- *La grilla puede tener muchísimos hilos, muchos más que el vector más grande que pueda caber en la memoria del device.*
 - *El asunto es como mapear sus índices a los datos.*
 - *Pero, ¿da igual cualquier grilla?*

Grilla y bloques unidimensionales



$$\text{index} = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}$$

$$\text{index} = (2) * (256) + (3) = 515$$

```
// grilla de threads suficientemente grande...
dim3 nThreads(256);
dim3 nBlocks((N + nThreads.x - 1) / nThreads.x);
// suma paralela en el device
VectorAdd<<< nBlocks, nThreads >>>(d_a, d_b, d_c, N);
```

Lanzamiento del Kernel

```
// kernel
__global__ void VectorAdd(int *a, int *b, int *c, int n)
{
    // indice de thread mapeado a indice de array
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n)
        c[i] = a[i] + b[i];
}
```

Kernel

Gilla unidimensional de bloques unidimensionales

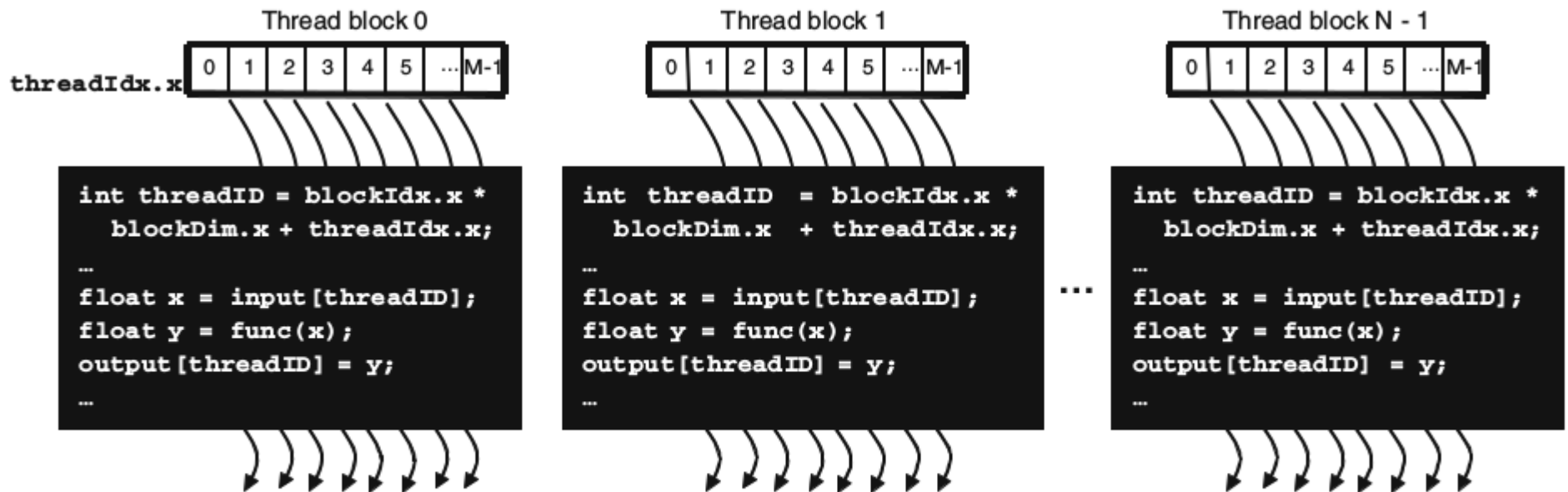


FIGURE 4.1

Overview of CUDA thread organization.

En general, se usan variables reservadas de tipo “dim3”
threadIdx.x, threadIdx.y, threadIdx.z, blockIdx.x, blockIdx.y, blockIdx.z,
blockDim.x, blockDim.y, blockDim.z, gridDim.x, gridDim.y, gridDim.z

Funciones “Kernel”

CUDA KERNELS ARE FUNCTIONS WITH RESTRICTIONS

The following restrictions apply for all kernels:

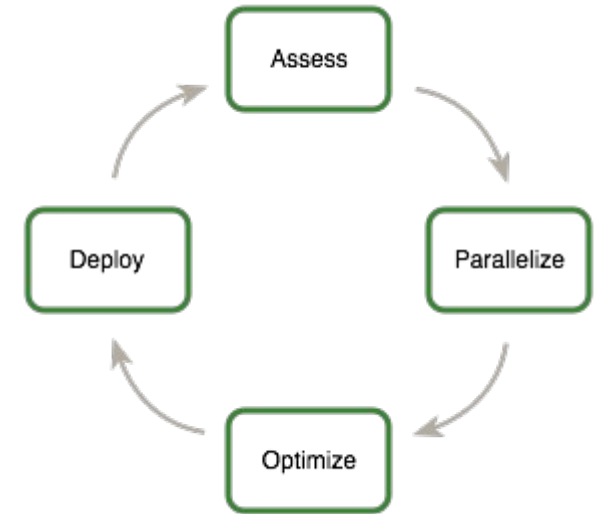
- Access to device memory only
- Must have void return type
- No support for a variable number of arguments
- No support for static variables
- No support for function pointers
- Exhibit an asynchronous behavior

```
// kernel
__global__ void VectorAdd(int *a, int *b, int *c, int n)
{
    // indice de thread mapeado a indice de array
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n)
        c[i] = a[i] + b[i];
}
```

QUALIFIERS	EXECUTION	CALLABLE	NOTES
__global__	Executed on the device	Callable from the host Callable from the device for devices of compute capability 3	Must have a void return type
__device__	Executed on the device	Callable from the device only	
__host__	Executed on the host	Callable from the host only	Can be omitted

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-handly-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

CUDA PROGRAM STRUCTURE

A typical CUDA program structure consists of five main steps:

1. Allocate GPU memories.
2. Copy data from CPU memory to GPU memory.
3. Invoke the CUDA kernel to perform program-specific computation.
4. Copy data back from GPU memory to CPU memory.
5. Destroy GPU memories.

THREE RULES OF GPGPU PROGRAMMING

Observation has shown that there are three general rules to creating high-performance GPGPU programs:

1. Get the data on the GPGPU and keep it there.
2. Give the GPGPU enough work to do.
3. Focus on data reuse within the GPGPU to avoid memory bandwidth limitations.

These rules make sense, given the bandwidth and latency limitations of the PCIe bus and GPGPU memory system as discussed in the following subsections.

Mini-Proyectos

- ¿ Qué problema quieren resolver usando GPGPU ?
- Si les sirve para su trabajo/tesis/etc buenísimo. Sino también, siempre que sea divertido y aprendamos algo.
- Escribir una versión serial primero, acelerarla luego.
- Medir la ambición, con que sea conceptual o prueba piloto ok!
- Usar: Cuda C/C++, python, bibliotecas, aplicaciones, nuevas herramientas, etc.
- Que cualquiera pueda verlo, correrlo y evaluar performance.
- *Charlita abierta al final del curso explicando el problema, la motivación, la implementación y los resultados.*

Tarea para la próxima clase

- Escribir un código secuencial SAXPY, $Y = a * X + Y$, donde X e Y son vectores de dimensión N .
- Escribir un código secuencial de multiplicación de matrices cuadradas de $N \times N$.
- Escribir un código secuencial que sume los elementos de X .
- Cronometrar los tiempos para distintos tamaños de vectores.

Links

- <https://developer.nvidia.com/blog/even-easier-introduction-cuda/>
- <https://www.youtube.com/@NVIDIADeveloper>