CINECA

Programming GPUs with CUDA: Introduction to GPUs

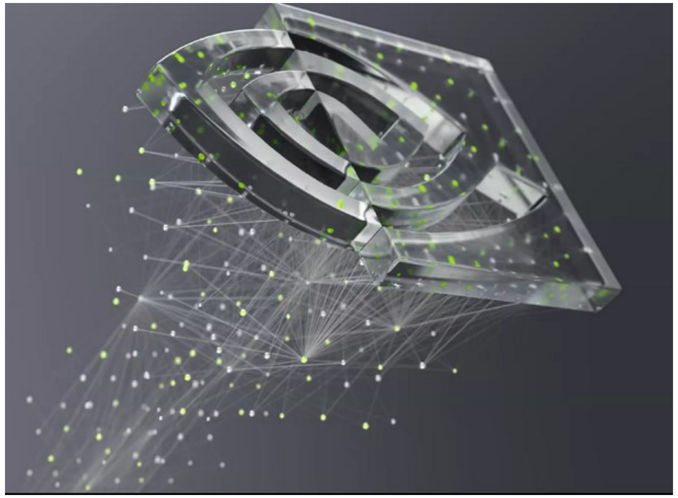
Lara Querciagrossa, Andrew Emerson, Nitin Shukla, Luca Ferraro, Sergio Orlandini

l.querciagrossa@cineca.it

July 12th, 2022

In this lecture…

- What are GPUs
- CPUs vs GPUs
- GPU architecture
- How to accelerate applications
- GPGPU programming model



Introduction to GPU

What are GPUs?

- GPU stands for **Graphics Processing Units.**
- It is a device equipped with:
  - *highly parallel microprocessor* (thousands of cores),
  - *private memory* with very *high bandwidth* (about 900 GB/s).

- GPUs highly parallel structures makes them more efficient w.r.t. CPUs for **embarrassing parallel algorithms**:
  - CPUs race through a series of task requiring lots of interactivity;
  - GPUs break complex problems into thousands or millions of separate tasks and work them out at one.

What are GPUs?

- Born in '90 as a response to the growing demand for high-definition 3D **graphics** (gaming, animations, etc), where textures, lighting and the rendering of shapes should be done on the fly.
- Now routinely used in **HPC** and **machine learning** applications.
- Most important vendors: ***NVIDIA***, AMD, …
  - Different solutions depending on the purpose:
    - Workstations running professional applications
    - Gaming
    - HPC/GPGPU

"The CPU has been called the brain of a PC. The GPU its soul." [1]

Parallel Intensive Computation

- GPUs are designed to render complex 3D scenes composed **of millions of data points/vertex** at high frame rates (60-120 FPS).
- The rendering process requires a **set of transformations based on linear algebra operations and (mostly local) filters**:
  - the *same set of operations* are applied on each data point of the scene;
  - each operation is *independent* of each other;
  - all operations are performed in *parallel* using a *huge number of threads* which process all data independently.

SPMD: Single Program Multiple Data

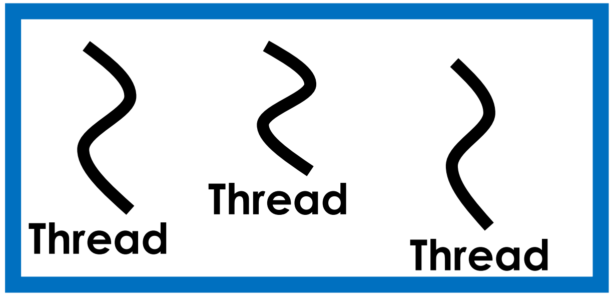
- If the set of transformations can be *applied independently on each point*, the **output result is independent on the order of point computation.**

```
for (int i = 0; i < N; i++) {
        C[i] = A[i] + B[i];
}
```

- If transformations are independent, we can **speed up** the elaboration using **parallel work**:
  - apply the same transformation (Single Program) ...
  - ... to each point (Multiple Data)
  - ... using multiple threads concurrently.

What is a THREAD?

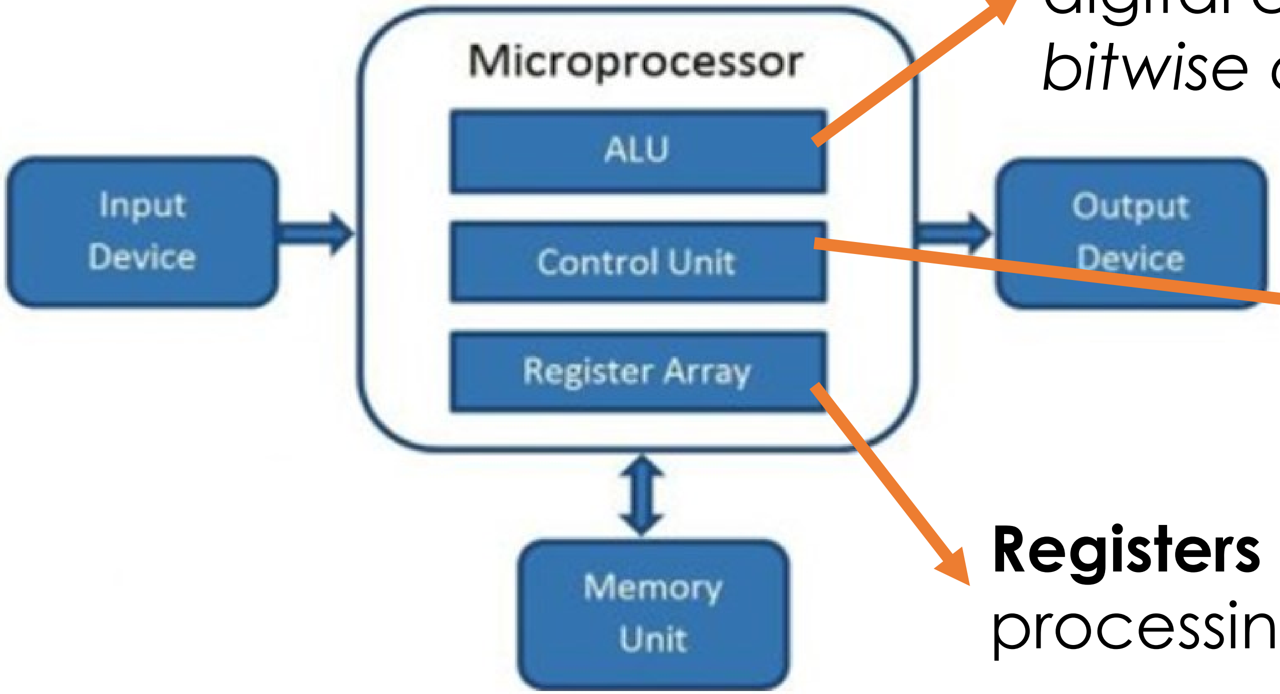
- A thread is an **independent flow of execution** of a program which **share the same memory resources of the main process** (access to main data).
- A process can spawn *multiple threads* of execution, each can follow an independent flow.



- Threads inside a process can:
  - **exchange data** because they share the same memory of the program,
  - **use local memory** not shared with other threads,
  - **synchronize with other threads** (waiting dependences).

Concurrency using CPU threads

- Threads can run **concurrently** as long as there is **available hardware** to use for their execution (*registers* and *ALUs*).



Arithmetic Logic Unit is a combinational digital circuit that performs *arithmetic and bitwise operations* on integer binary number.

Control Unit directs the operations of the processor by providing timing and control signals to synchronize these operations.

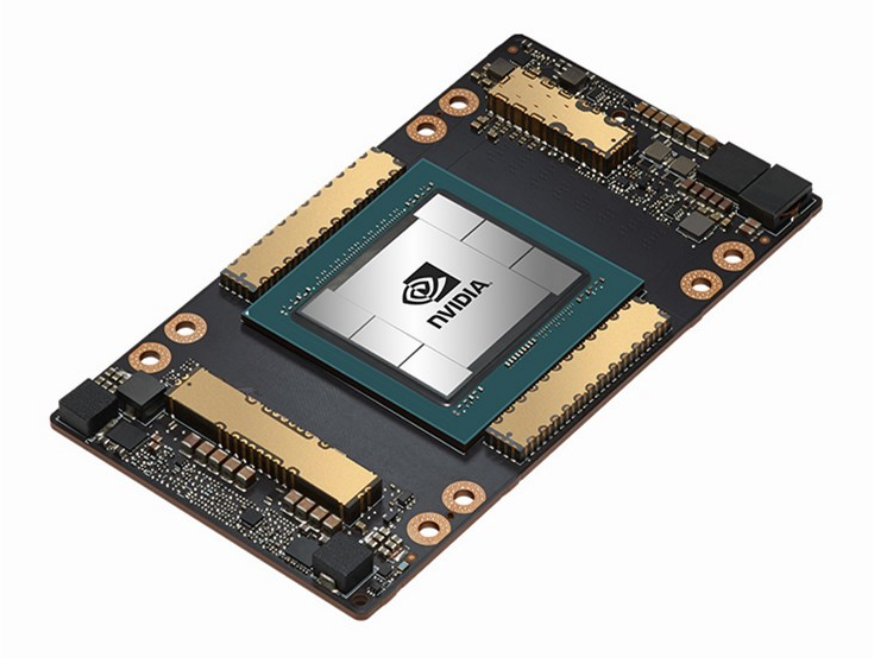
Registers store data before or during processing.

Concurrency using CPU threads

- Modern CPUs often have **multiple cores on chip**: each core has registers and ALU units to run a thread independently from other cores.
- When there are more threads on the fly than available cores, the operating system can make a **context switch:**
  - the running thread is freezed: all information about its status is saved and put back for later restore,
  - a new thread take the hardware resources, load its previous status and restart its flow until a new context switch will take place.
- CPU threads context switch involves many backup operations, plus the fact that data is no longer in registers and caches.

CPU threads vs GPU threads

- GPU are able to manage **thousands of threads efficiently.**
- GPU threads are extremely **light weighted:**
  - GPU have *thousands of available registers,*
  - each thread running on a GPU *maintains its status* in its own registers,
  - *no penalty in case of a context-switch.*
- The more threads are in flight, the more the GPU hardware is able to **hide** memory or computational **latencies.**



Summing up: CPUs vs GPUs

CPU vs GPU

- A Central Processing Unit (**CPU**) is a latency-optimized general purpose processor designed to handle a wide range of tasks sequentially.
- Graphics Processing Unit (**GPU**) is a **throughput-optimized specialized processor designed for high-end parallel computing.**

- **Massive Parallel Computing**: extensive calculations with similar operations.
- **High Data Throughput**: thousands of cores performing the same operation on multiple data items in parallel.
- **High Computing Throughput**: high-performance computing power.

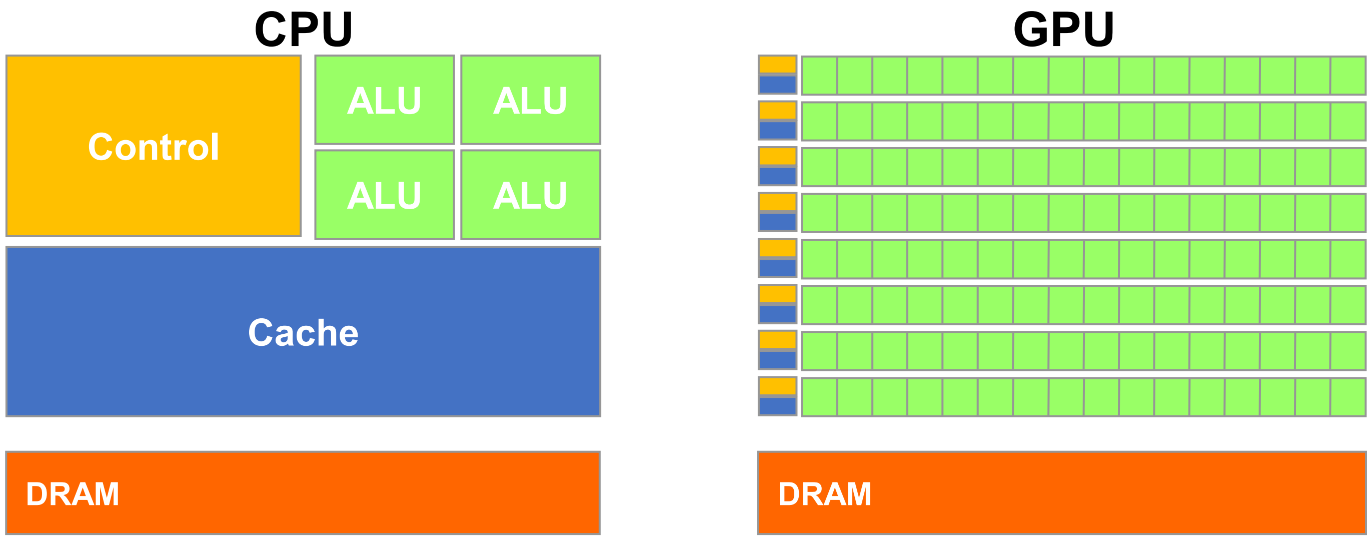
CPU vs GPU architecture

CPU

- just a few cores (8-32)
- large cache, low latency
- tens of software threads at a time
- task parallelism

GPU

- thousands of cores (5k)
- small memory, large bandwidth
- thousands of threads simultaneously
- data parallelism



Why using GPUs?

- Very *large number of "cores"* (better performance w.r.t. CPU applications).
- Often *low cost* ($/Flop) and *low energy consumption* (Watts/Flop).

- May need *complete rewrite of application.*
- *Low device memory* (max 16/32 Gb).
- Depending on model, possible low transfer speeds.

GPU architecture

GPU architecture scheme

- **Main global memory**
  - medium size (16-40 GB)
  - very high bandwidth (800-1200 GB/s)

- **Streaming Multiprocessors** (SM)
  - grouping independent cores and control units
  - number of SM depends on GPU architecture
  - ~16-32 up to ~100 SM on modern GPUs

Volta GV100 Full GPU with 84 SM Units

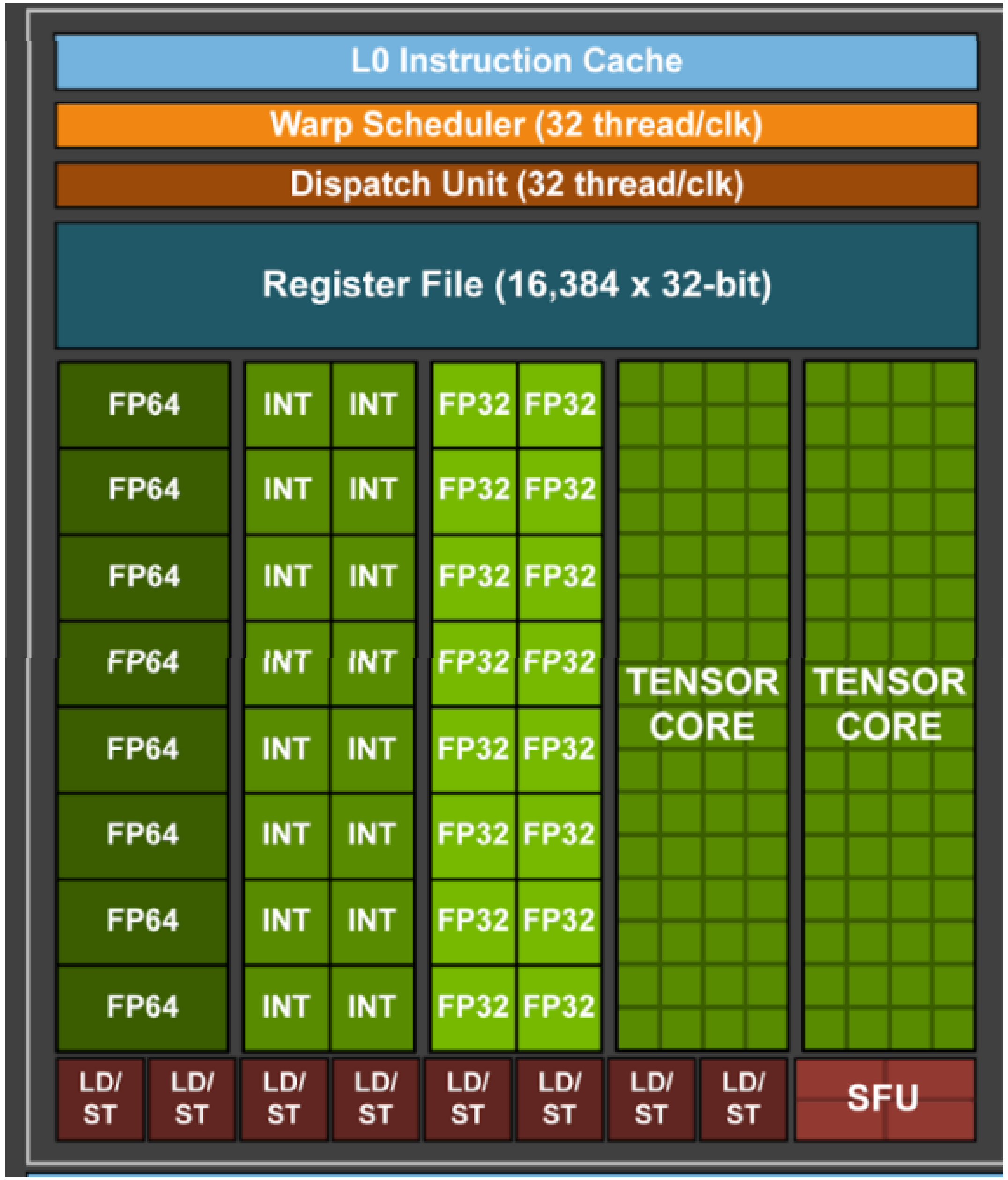
GPU architecture scheme

- Each **SM** unit has:
  - many cores (> 100 cores)
  - lots of registers (32K-64K)
  - instruction scheduler dispatchers
  - shared memory with fast access to data
  - several caches
- Host/Device connection technology (NVLink, PCIExpress, AMD Infinity Fabric) → **data movement bottleneck**

Volta GV100 Streaming Multiprocessor

GPU functional unit types

- **FP32**: performs 32-bit floating point operations.
- **INT32**: performs 32-bit and maybe some logical operations.
- **FP64**: executes 64-bit FP operations.
- **Special Functional Unit (SFU):** performs reciprocal (1x) and transcendental instructions.
- **Load/Store (LS):** performs loads and stores from all memory address spaces.
- **Tensor Core:** specialized units to compute A*B+C matrix product.



Volta GV100 functional unit types

NVIDIA Volta V100 architecture (2017)

- https://developer.nvidia.com/blog/inside-volta
- A full GV100 GPU unit contains **6 Compute Graphic Clusters (CGC)** with 14 SM each, **total 84 SMs**
- 5376 FP32 cores
- 5376 INT32 cores
- 6MB L2 cache
- **High Bandwidth Memory**
  - 16 GB HBM2 SDRAM
  - 900 GB/s bandwidth
- **NVLink** technology
  - 300GB/s bandwidth
- **Peak Performance**:
  - **15.7 FP32 TFlops**
- **Max Power Consumption**:
  - 300W

Volta GV100 Full GPU with 84 SM Units

NVIDIA Volta V100 architecture (2017)

- SM composed of 4 independent blocks
- each block supports:
  - 1 warps x 2 dispatchers
  - 16FP32 + 16INT32 ALU units
  - separate FP32 and INT32 cores, allowing simultaneous execution of FP32 and INT32 operations at full throughput
  - 8 FP64 ALU units
  - 2 Tensor Core units (HW matmul)
  - 8 Load/Store units
  - 4 SFU units
  - 32768 32bits registers
- each block accesses:
  - 128KB for L1/shared memory
  - 4 texture units

Volta GV100 Streaming Multiprocessor

Accelerating applications

How do I accelerate applications?

Libraries	Directives	Programming languages
«Drop in» accelerations	Easily accelerate applications	Maximum flexibility

← Portability

Performance →

How do I accelerate applications?

Libraries

«Drop in»
accelerations

Directi

Easily
acceler
applications

amming
uages

imum
ibility

cuBLAS: CUDA Basic Linear Algebra Subroutines

- cuSPARSE for sparse matrices
- support for all 152 standard BLAS routines
- mixed and low precision support
- CUDA streams support

How do I accelerate applications?

Libraries

«Drop in» accelerations

Directi

Easil
accele
applications

mming
uages

mum
xibility

MAGMA: Matrix Algebra for GPU and Multicore Architecture

- *hybrid approach*: small non-parallelizable tasks are run on CPU, large and parallel ones on GPU, bounded with communication
- data dependencies among tasks used to properly *scheduling tasks' execution* over multicore and GPU hardware components
- built on top of *CUDA*

How do I accelerate applications?

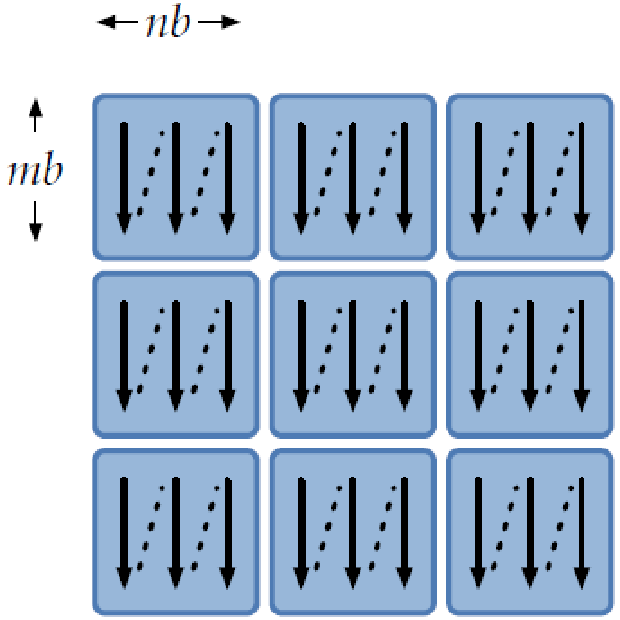
Libraries

«Drop in» accelerations

Easily accelerate applications

Pr

I

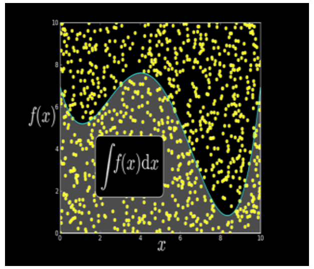


SLATE: Software for Linear Algebra Targeting Exascale

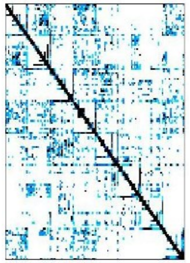
- based on *tile layout* and tile algorithms
- *flexible*: not-uniform size blocks, arbitrary matrix distribution
- *standard based*: MPI, OpenMP, GPU support with cuBLAS
- *task-based parallelism*

How do I accelerate applications?

NVIDIA cuBLAS



NVIDIA cuRAND



NVIDIA cuSPARSE

NVIDIA NPP

GPU VSIPL

Vector Signal Image Processing

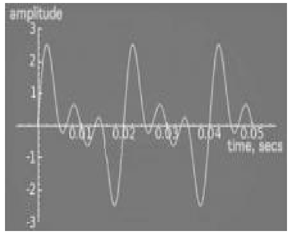
CULA tools

GPU Accelerated Linear Algebra

MAGMA

Matrix Algebra on GPU and Multicore

open source initiative



NVIDIA cuFFT

ROGUE WAVE SOFTWARE

IMSL Library

ArrayFire Matrix Computations

CUSP

Sparse Linear Algebra

open source initiative

Thrust

C++ STL Features for CUDA

open source initiative

How do I accelerate applications?

Directives

Programming

«Drop in» accelerations

Easily accelerate applications

Maximum flexibility

- **openMP** from v 4.0 allows offloading of tasks onto GPUs.
- **openACC** allows to annotate areas of code that should be accelerated using compiler directives and additional functions.
- ...

How do I accelerate applications?

«Drop in» accelerations

Directives

applications

Programming languages

Maximum flexibility

- **CUDA**: C extension, developed by NVIDIA, PGI compiler, FORTRAN extension.
- **OpenCL**: General framework for writing programs across heterogenous devices. Often used for non-NVIDIA GPUs and FPGAs. Open source. Low level and verbose language.

How do I accelerate applications?

«Drop in» accelerations

CuPy

sily lerate applications

Programming languages

Maximum flexibility

- **HIP**: developed by AMD, interface similar to CUDA, designed to easily convert existing CUDA code (HIPify tool to automate this conversion).
- **Python support**: **pyCUDA**, **cuPy** (open source NumPy/SciPy compatible lib), **numba** (decorators or CUDA language).

How do I accelerate applications?

braries

Directives

Programming languages

«Drop

acceler

Easily accelerate applications

Maximum flexibility

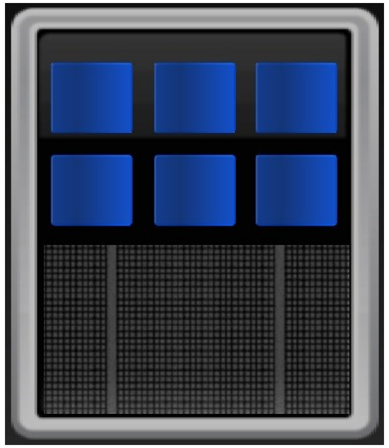
- **oneAPI**: developed by Intel.
- **SYCL**: developed by Khronos Group (as OpenCL).
- **DirectCompute**: developed by Microsoft.

GPGPU Programming Model

GPGPU programming model

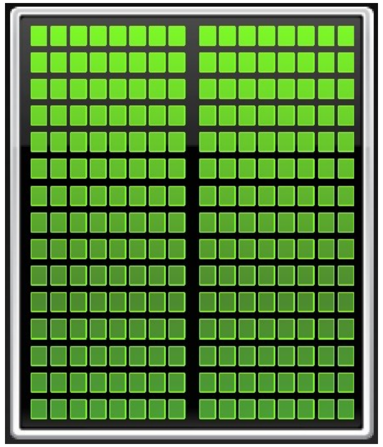
- **General Purpose GPU Programming** relates to use GPU computational power to solve problems other than graphics.
- CPU and GPU are separate devices with separate memory space addresses.
- GPU is seen as an *auxiliary coprocessor* equipped with thousands of cores and a high bandwidth memory.
- They *should work together* for best benefit and performances.

CPU



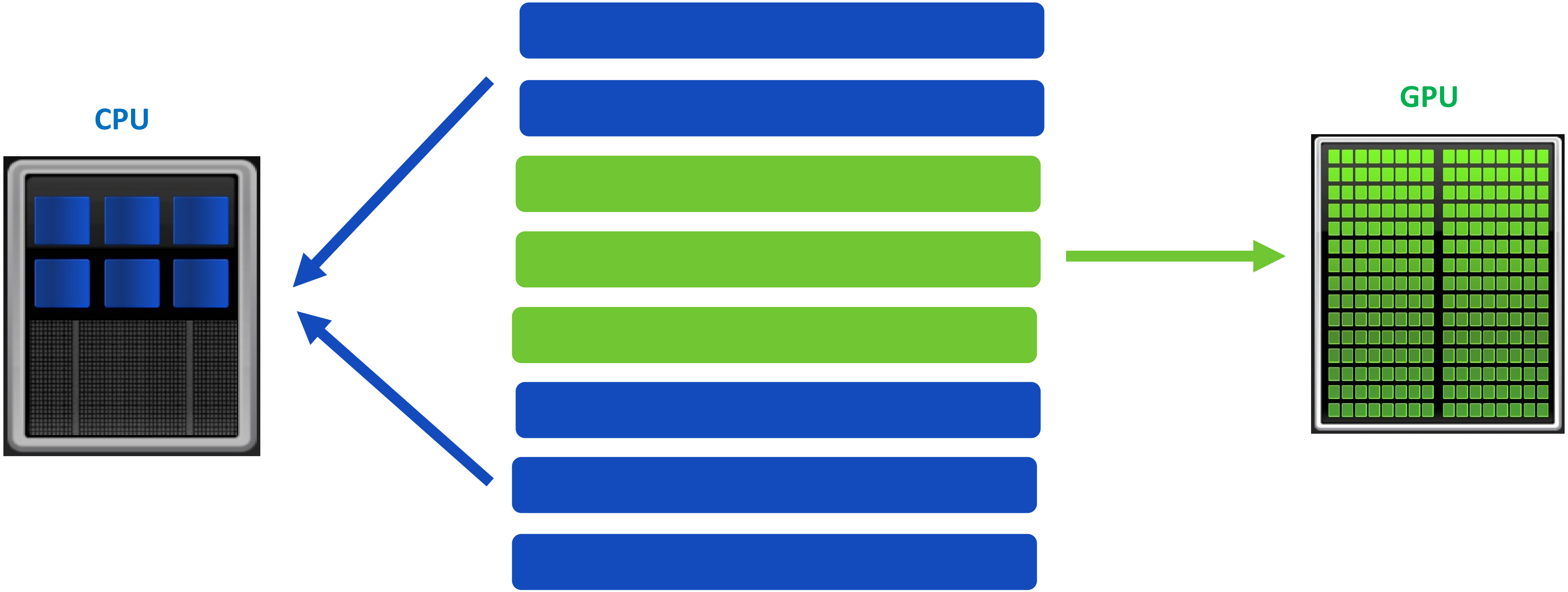
+

GPU



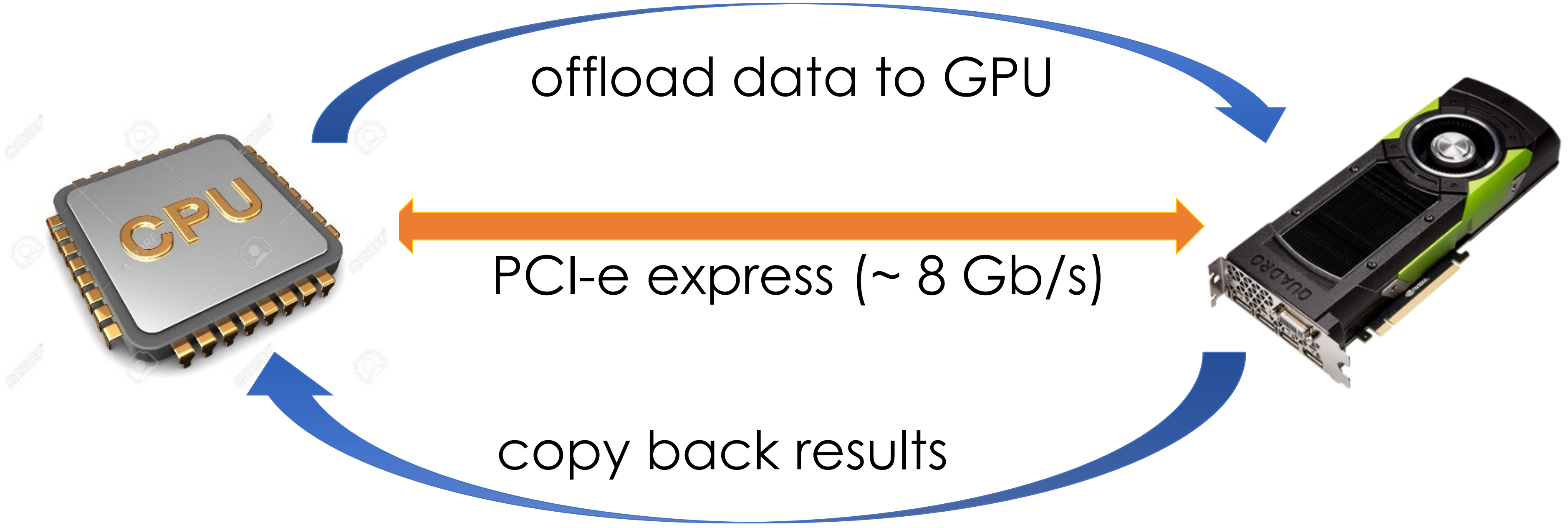
GPGPU programming model

- *Serial*, or processes with *low parallelism*, remain on the **CPU.**
- *Computational intensive high parallel* regions are ***offloaded*** to the **GPU** for processing.



GPGPU programming model

- Required **data** is **moved** on GPU memory and back to the CPU: *connection* to GPU is *relatively slow* so **minimise data transfers**.
- The GPU's threads need to be ***saturated*** otherwise the result will be slower than CPU.
- For max performance, *execution on the CPU should continue* as much as possible while GPU is being used.



References

References

- Previous editions of this school at CINECA
- Oakridge National Laboratory's "Introduction to CUDA C++": https://www.olcf.ornl.gov/calendar/introduction-to-cuda-c/
- NVIDIA DL Institute Online Course
- blogs.nvidia.com
- unsplash.com
- Wikipedia
- https://www.javatpoint.com/microprocessor-introduction
- Computer register: https://www.javatpoint.com/computer-registers, https://www.geeksforgeeks.org/different-classes-of-cpu-registers/, https://alltechqueries.com/what-are-cpu-registers/
- ALU: https://www.javatpoint.com/what-is-alu
- https://www.learncomputerscienceonline.com/central-processing-unit/

THANK YOU!

Lara Querciagrossa
l.querciagrossa@cineca.it