Introduction to GPU Computing

Advanced Computational Techniques

TCCM Erasmus Mundus course

Sept 26-30, 2022

University of Perugia

Sergio Orlandini
CINECA

s.orlandini@cineca.it

CINECA

3rd Day Outline

- GPU Architectures & Programming Model
- OpenACC Crash-Course
  - Hands-On with OpenACC
- CUDA Crash-Course
  - Hands-On on GPU porting with CUDA
- GPU Memory Hierarchy
- Multi GPU Programming
- GPU/CPU Interaction
- Hands-On
  - Optimizing CUDA Enabled Application
  - CPU-GPU Concurrency

Course Disclaimer

This course is intended to give a **general introduction** to parallel computing and gpu computing.
The topic is very **wide** and **complex**, the intention is not to give an exhaustive explanation of the topic because it would not be possible in such a short time.
The aim is only to introduce the argument and to leave some pointers from which to take inspiration for subsequent in-depth studies and/or courses to follow.
At the end of this course, you will not be an expert in gpu computing, but you will have acquired the **main concepts** of gpu computing and how to use a computing cluster.
This **knowledge** will be very **useful** for future projects, to speed-up your application and consequently your research and/or for writing proposals to submit to a computer centre.

Gentle Starting Poll

http://etc.ch/PvN5

Course Material

All the material, slides and hands-on, is available at the repository:

https://github.com/so07/TCCM-GPU-2022

TCCM-GPU-2022

Advanced Computational Techniques.
TCCM Erasmus Mundus course

September 26-30, 2022

University of Perugia

Download/Clone repository

```
git clone https://github.com/so07/TCCM-GPU-2022.git
```

Graphic Cards

Graphics Processing Unit aka GPU

Graphics Processing Unit (**GPU**) is a device equipped with

- **highly parallel microprocessor** (thousands of cores)
- private memory with very **high bandwidth** (~900 GB/s).

GPU highly parallel structure makes them more efficient than CPUs for embarrassingly parallel algorithms.

Born in '90 as a response to the growing demand for high definition **3D rendering** graphic applications (gaming, animations, etc)

The increasing popularity of 3D-accelerated games caused a rapid growth of computational capabilities of modern GPUs.

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

Parallelism of Single Program Multiple Data (SPMD)

```
// typical loop over each point with the same set of operation
for each point in collection_of_points:
   output_data = trasformations_on_point(point, input_data)
```

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

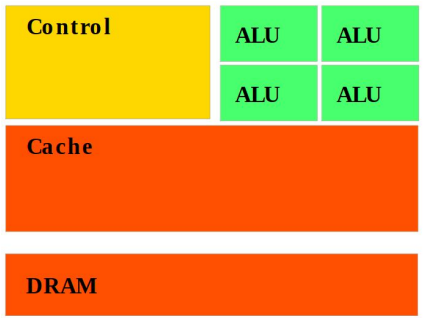
CPU vs GPU

A Central Processing Unit (**CPU**) is a latency-optimized general purpose processor designed to handle a wide range of tasks sequentially, while a 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

CPU	GPU
• **Low** compute density • Few **heavy**-weight cores 8 to 32 cores • Suitable for **Task** parallelism • Low latency • Large caches • Explicit thread management • Optimized for **serial** tasks	• **High** compute density • Many **light**-weight cores 5000+ cores • Suitable for **Data** parallelism • High throughput • High Memory Bandwidth • Threads are managed by hardware • Optimized for **parallel** tasks



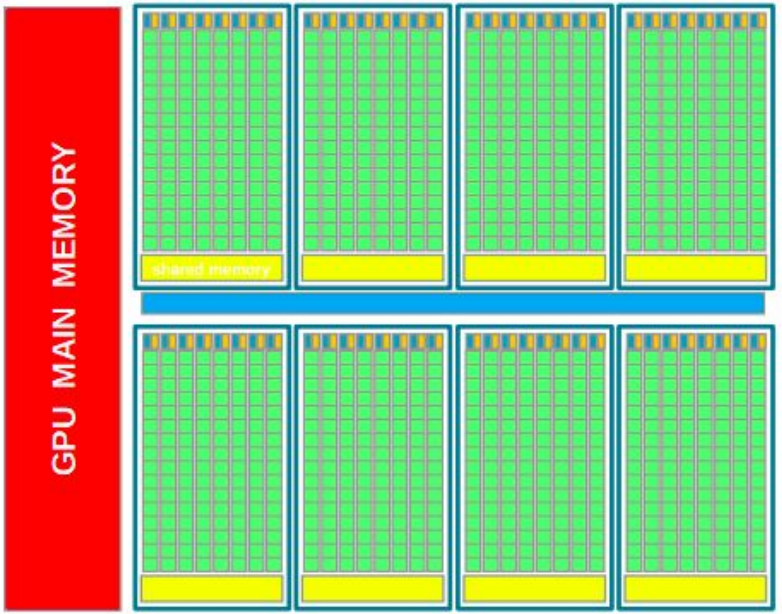
CPU

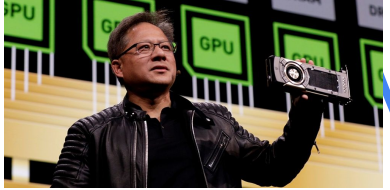
GPU

GPU Architecture Scheme

A typical GPU architecture consists of:

- **Main Global Memory**
  - Medium size (16-40 GB)
  - Very high bandwidth (800-1200 GB/s)
- **Streaming Processors (SM)**
  - Grouping independent cores and control units
  - Number of SM depends on GPU architecture
  - ~16-32 up to ~100 SM on modern GPUs
- 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





VIDIA HPC GPU Solutions

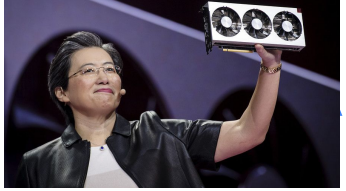
Nvidia's GPU solutions:

- **Tesla** serie is the Nvidia's top gamma GPU solution for general-purpose graphics processing units (GPGPU) and HPC.
- **GeForce** series are for gaming.
- **Quadro** are intended for workstations running professional applications.

Model	FP32 [TFlops]	cores	RAM [GB]	Bandwidth [GB/s]
Kepler K40	4.3	2280	12 GDDR5	240
Pascal P100	10.6	3584	16 HBM2	720
Volta V100	15.7	5120	16/32 HBM2	900
Ampere A100	19.5	8192	40 HBM2	1500

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



AMD HPC GPU Solutions

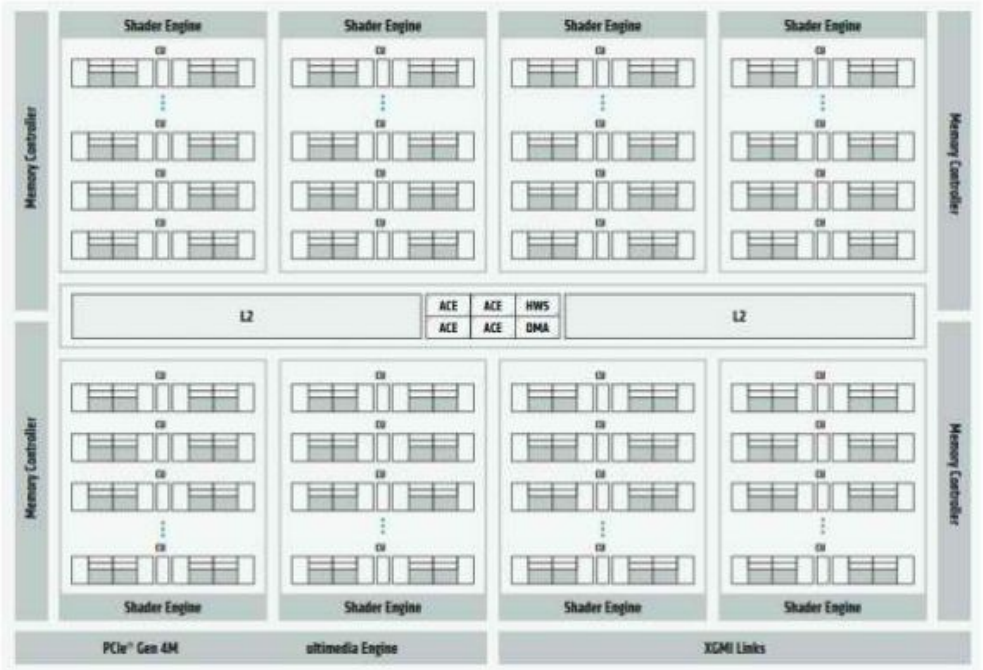
AMD GPU solution lines:

- Radeon Instinct line is intended HPC/GPGPU applications.
- Radeon RX VEGA/500/400 series are for gaming.
- Radeon Pro is the line of professional workstation cards.

Model	FP32 [TFlops]	cores	RAM [GB]	Bandwidth [GB/s]
Radeon MI8	8.2	4096	4 HBM	512
Radeon MI25	12.3	4096	16 HBM2	484
Radeon MI50	13.4	3840	16 HBM2	1024
Radeon MI100	32.1	7680	32 HBM2	1200

AMD Radeon MI100 Architecture (2020)

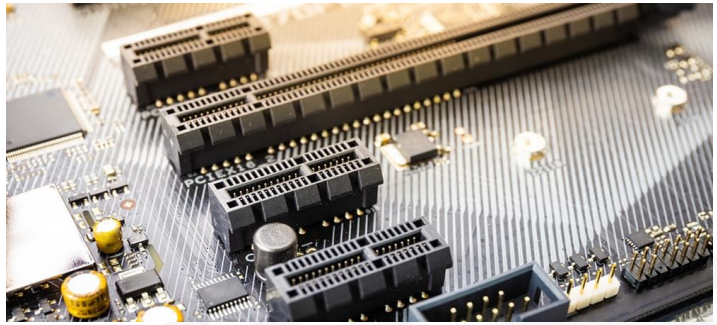
- A full MI100 GPU unit contains a total of 120 Compute Unit (like SMs)
- 7680 FP32 cores
- 8MB L2 cache
- High Bandwidth Memory
  - 32GB HBM2
  - 1200 GB/s bandwidth
- AMD Infinity Fabric
- Peak Performance:
  - 23.0 FP32 TFlops
- Max Power Consumption:
  - 400W



Host/Device Data Movements

- CPU and GPU are separated devices with **separate memory** space addresses.
- GPU is seen as an auxiliary **coprocessor**.
- Data must be moved from Host to Device memory in order to be processed on the GPU.
- When data has been processed on the GPU, they are transferred back to Host.
- The **data movement bottleneck**:
  - Data movement is often the bottleneck of many GPU applications.
  - The bus transfer can be quite slow with respect to the GPU throughput capacity.
  - Sometimes data transfer can take more than the data initialization/computation on GPU.

Host/Device Data Movements



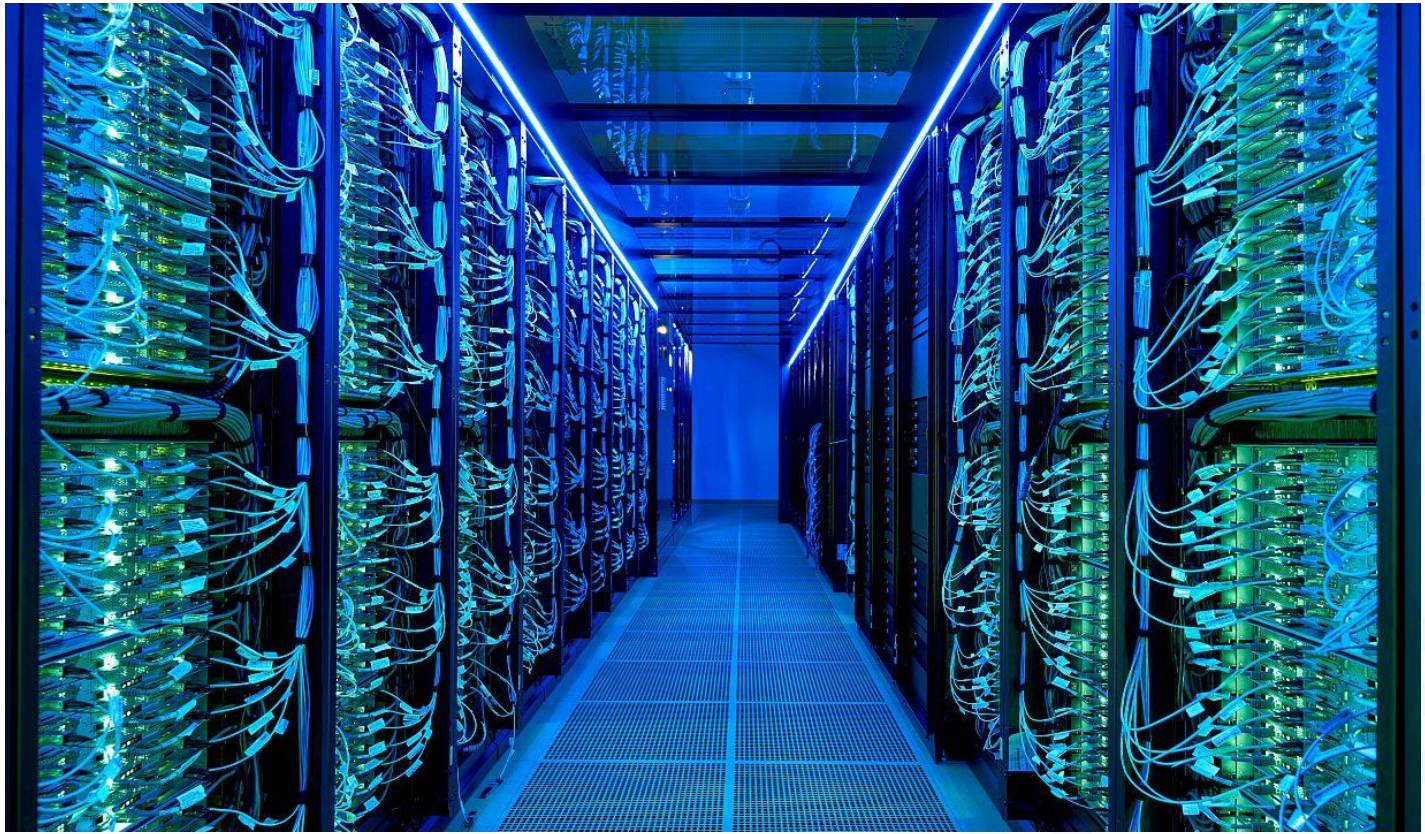
Available Connection Technologies:

- **PCI Express** link
- **NVLink** nVIDIA
- **Infinity Fabric** AMD

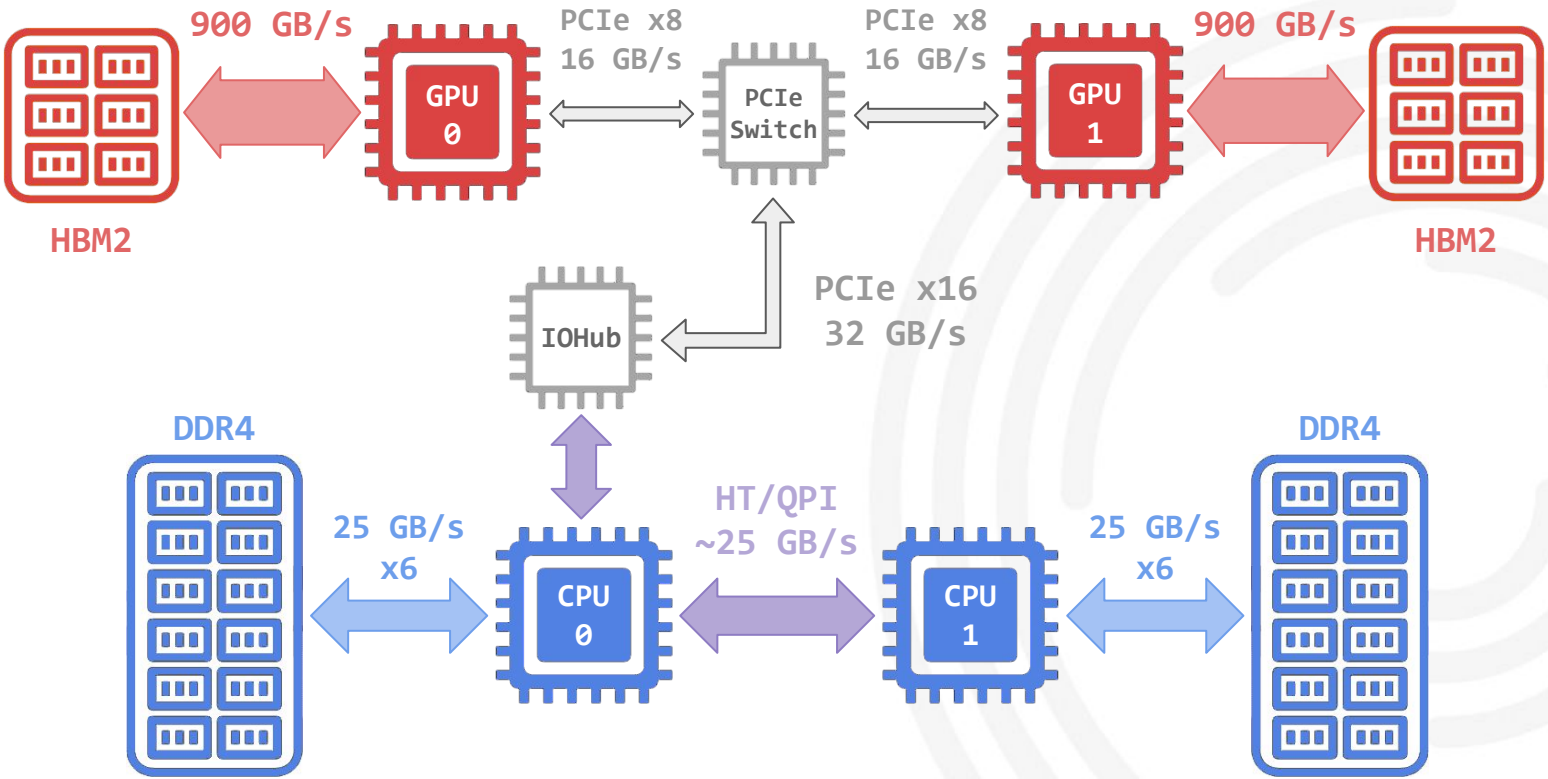
NVLink performance				
Version	Year	TR GT/s	Lane	Tot BW GB/s
1.0	2016	20	x4	80
2.0	2017	25	x6	150
3.0	2020	25/50	x6	300

PCI Express link performance		
Version	Year	BW x16 GB/s
1.0	2003	4.0
2.0	2007	8.0
3.0	2010	15.7
4.0	2017	31.5
5.0	2019	63.0

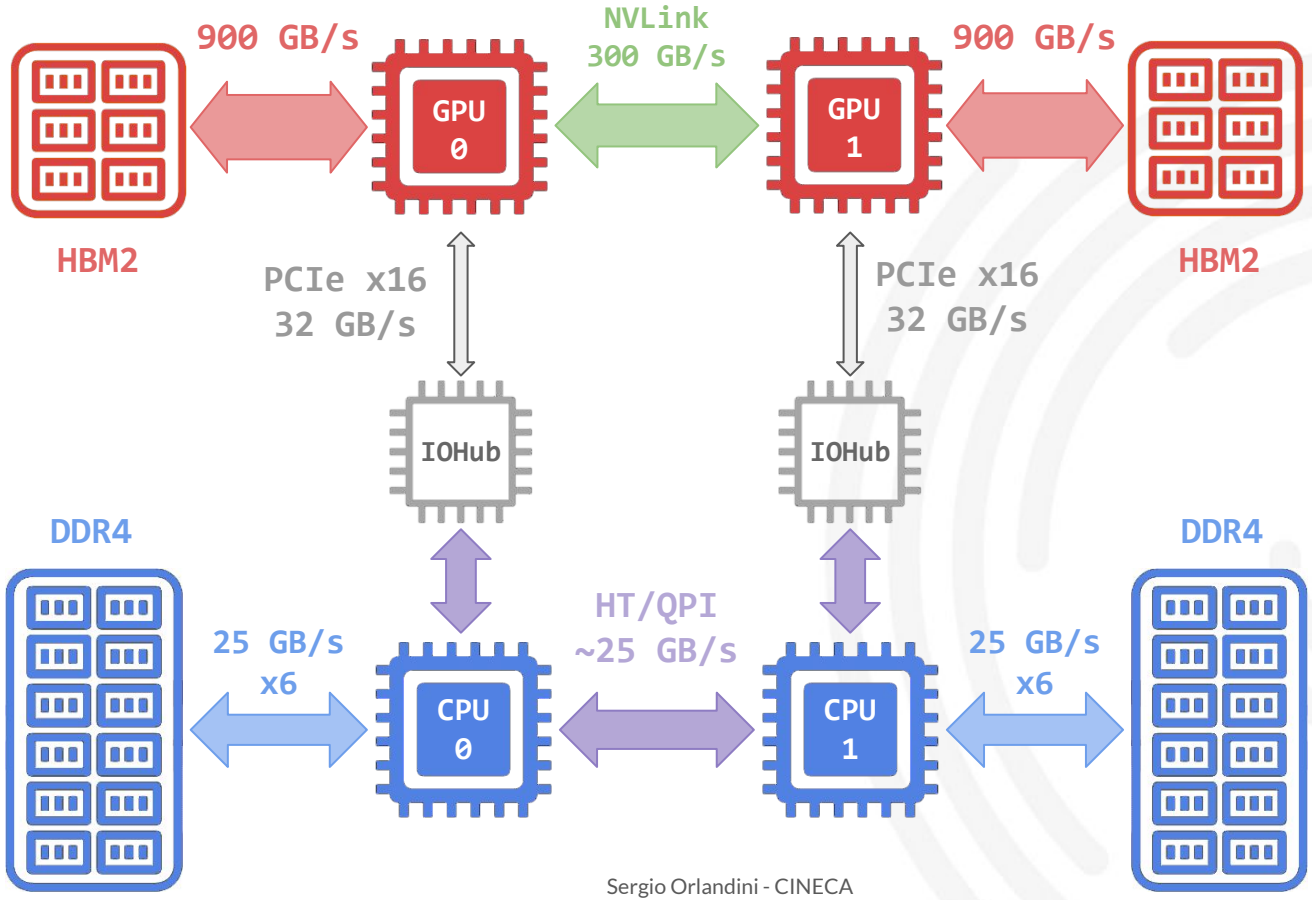
Supercomputers



Compute Node Schema



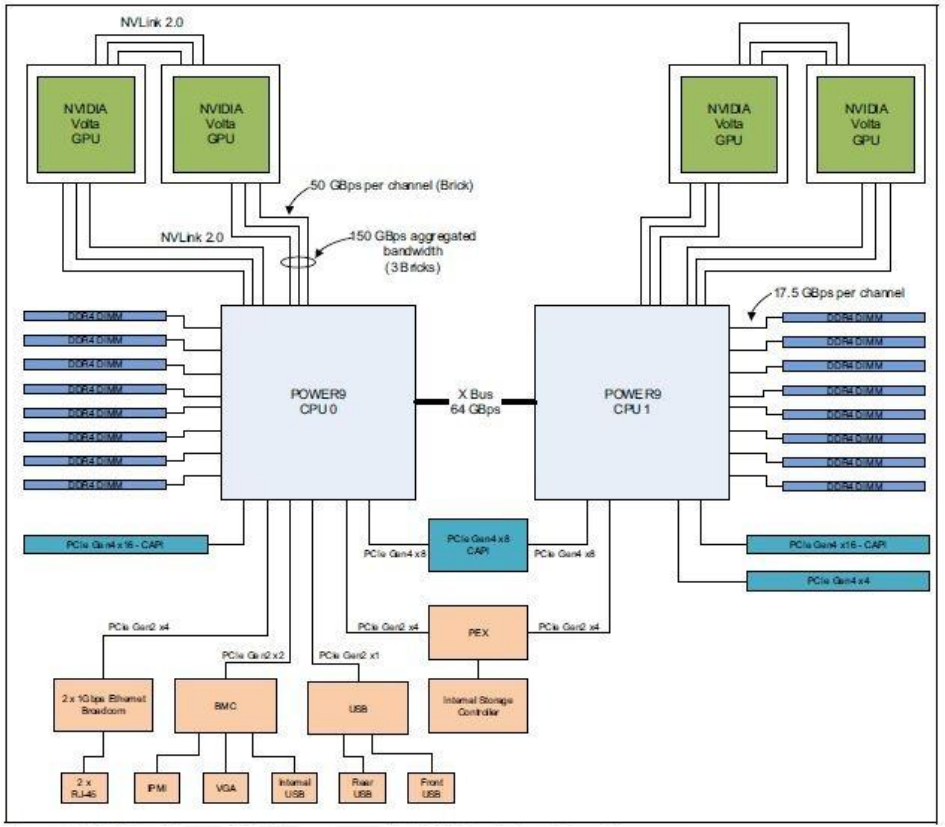
Compute Node Schema



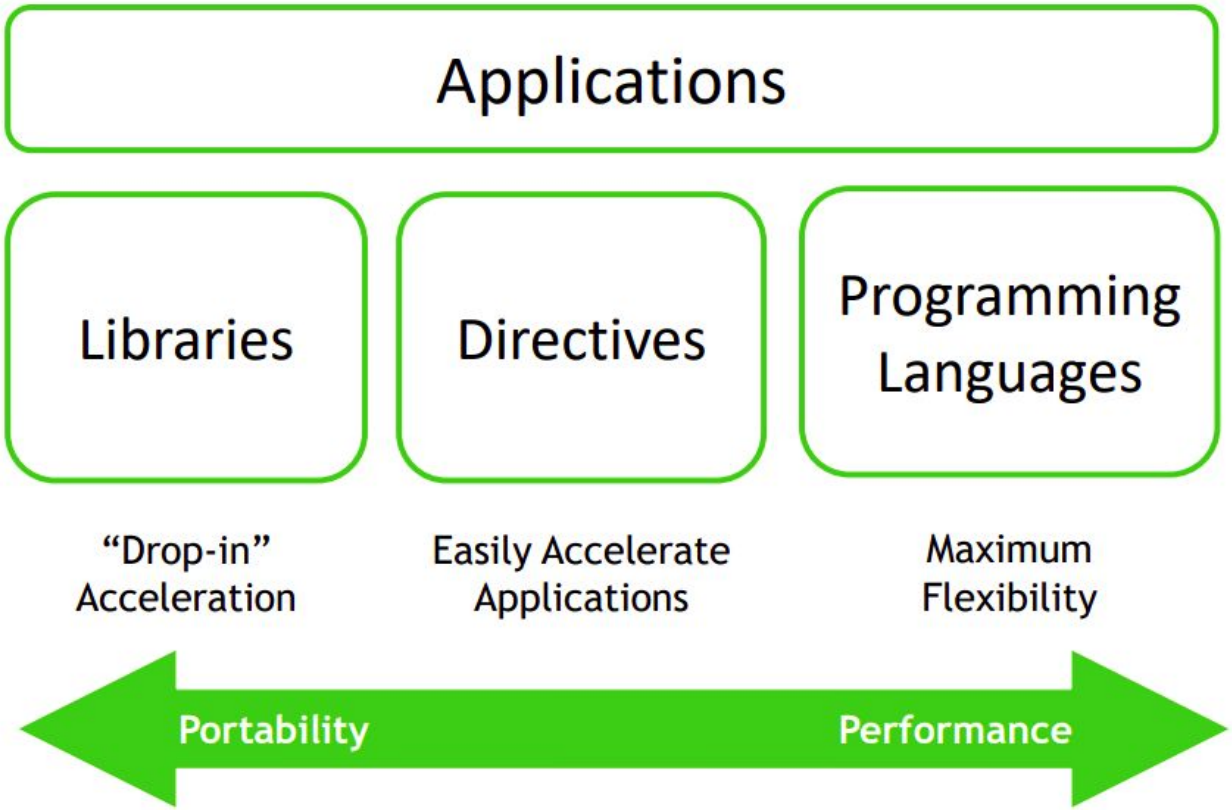
Marconi100 @CINECA aka M100

M100 Architecture:

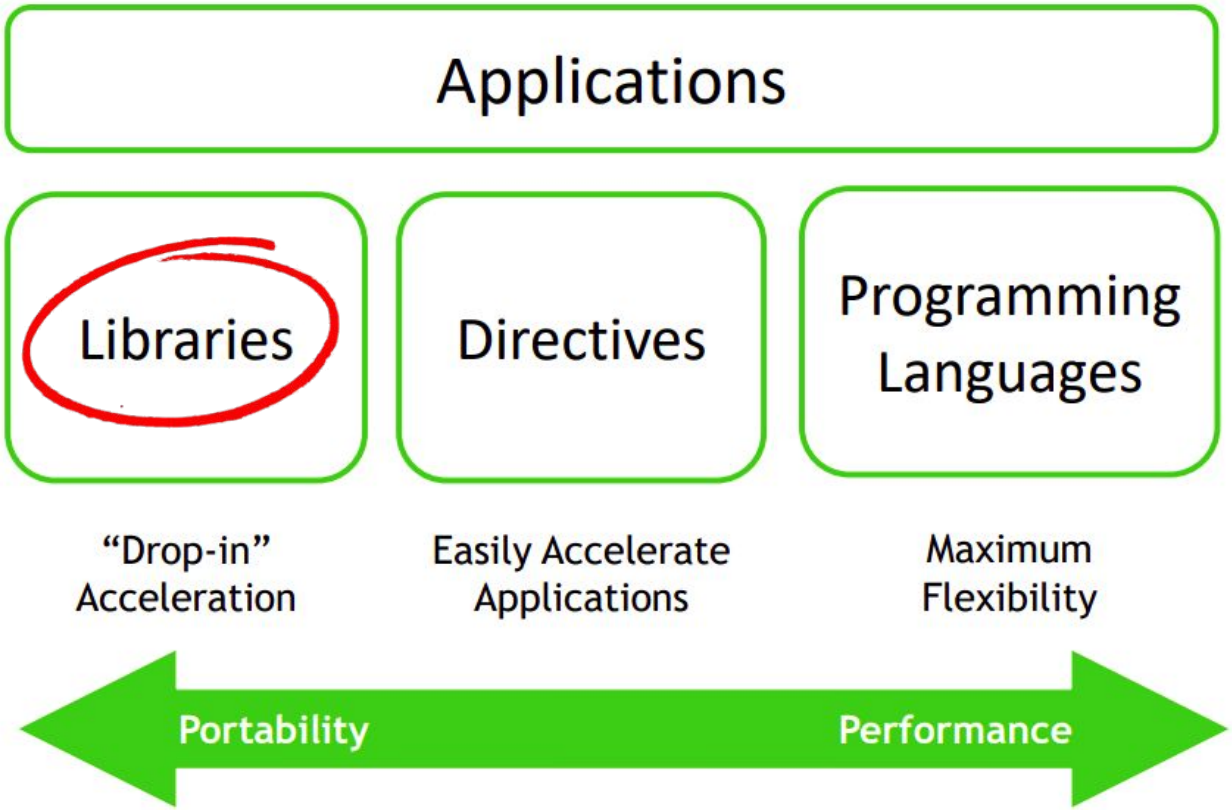
- **980** compute nodes + 3 login nodes
- 2x16 cores IBM **POWER9** AC922 at 2.6 GHz
  - 32 cores/node, Hyperthreading x4 (SMT4)
- RAM: **256** GB/node
- **4x** NVIDIA Volta **V100** GPUs/node 16GB
- CPU/GPU interconnection **Nvlink** 2.0
- Mellanox Infiniband EDR **DragonFly+**
- Peak Performance: about **32 Pflop/s**
  - 32 TFlops per compute node
- Disk Space: **8PB** raw GPFS storage
- Local Disk: **1.6TB** NVMe



3 Ways to Accelerate Applications

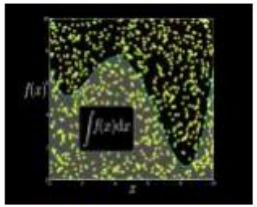


3 Ways to Accelerate Applications



A Pinch of GPU Enabled Libraries


NVIDIA cuBLAS


NVIDIA cuRAND


NVIDIA cuSPARSE


NVIDIA NPP


cuSOLVER

GPU VSIPL

Vector Signal Image Processing

CULA tools

GPU Accelerated Linear Algebra

MAGMA

Matrix Algebra on GPU and Multicore


NVIDIA cuFFT

C A B

cuTENSOR

ROGUE WAVE SOFTWARE

IMSL Library

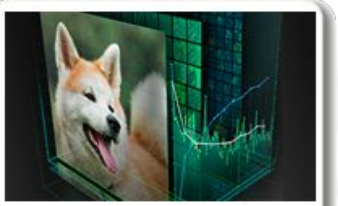
ArrayFire Matrix Computations

CUSP

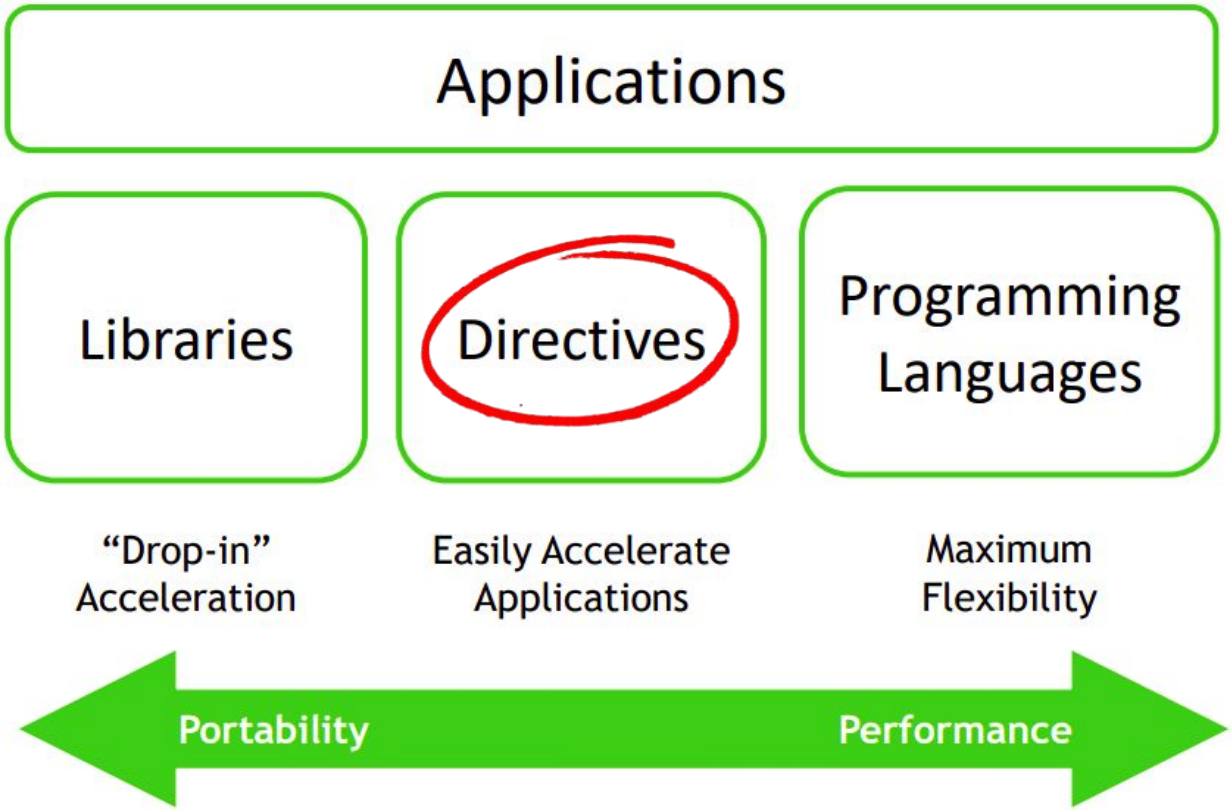
Sparse Linear Algebra

Thrust

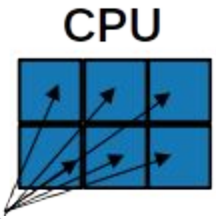
C++ STL Features for CUDA

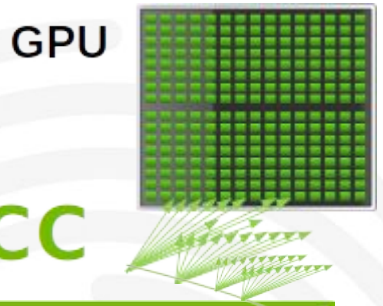

nvJPEG

3 Ways to Accelerate Applications



OpenMP *vs* OpenACC





OpenMP

```
main() {
  double pi = 0.0; long i;

  #pragma omp parallel for reduction(+:pi)
  for (i=0; i<N; i++)
  {
    double t = (double)((i+0.05)/N);
    pi += 4.0/(1.0+t*t);
  }

  printf("pi = %f\n", pi/N);
}
```

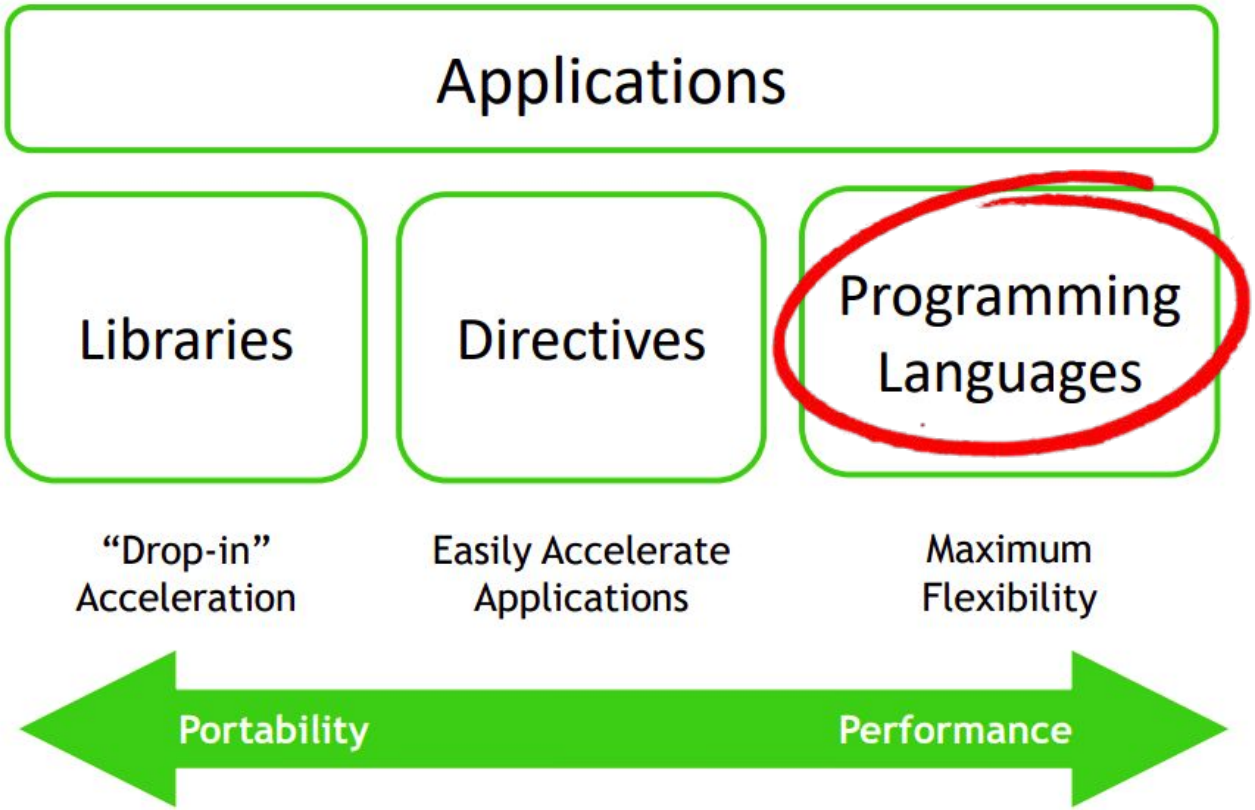
OpenACC

```
main() {
  double pi = 0.0; long i;

  #pragma acc parallel loop reduction(+:pi)
  for (i=0; i<N; i++)
  {
    double t = (double)((i+0.05)/N);
    pi += 4.0/(1.0+t*t);
  }

printf("pi = %f\n", pi/N);
}
```

3 Ways to Accelerate Applications



Graphics Processing Unit - A bit of History

- Born in 1990s as a response to the growing demand for high definition **3D rendering graphic** applications (gaming, animations, etc).
- The increasing popularity of 3D-accelerated games caused a rapid growth of computational capabilities of modern GPUs.
- in 2000s manufacturers extended the GPUs' core functionality with the introduction of **pixel and vertex** shader languages.
- Brave computer scientists used the increasing computational power of GPUs to implement more general algorithms by expressing them in shader languages. This was the birth of the so-called general-purpose computing on GPUs (**GPGPU**).
- In 2006 NVIDIA released the Compute Unified Device Architecture (**CUDA**) extension to C/C++ for GPU architecture.

Programming Languages for GPU Paradigms

Nowadays , there are several programming languages/framework aimed at GPU programming:

- Compute Unified Device Architecture (**CUDA**) / Nvidia,
- Heterogeneous Interface for Portability (**HIP**) / AMD,
- Open Computing Language (**OpenCL**) / Khronos Group,
- **OneAPI** / Intel,
- **SYCL** / Khronos Group,
- **DirectCompute** / Microsoft,
- …

Compute Unified Device Architecture aka CUDA

CUDA is a general purpose parallel computing platform and programming model created by NVIDIA.

It consists of:

- a hierarchical multi-threaded **programming paradigm** that matches GPU hardware structure,
- a set of **extensions to higher level programming languages** (C/C++ and Fortran) to use GPU as a coprocessor for heavy parallel task and to express thread parallelism within a familiar programming environment,
- a new **architecture instruction set** called PTX (Parallel Thread eXecution) to match GPU typical hardware,
- a **developer toolkit** to compile, debug, profile programs and run them easily in a heterogeneous systems,
- a set of **GPU accelerated libraries** for common scientific algorithms.

Compute Unified Device Architecture aka CUDA

CUDA C/C++ and FORTRAN API allow you to:

- Control and query available GPU devices,
- Manage GPU memory allocation and data transfers,
- Manage and control independent work queues,
- Define special language extension to express thread parallelism of selected functions (kernel) which will be executed on the GPU by thousands of threads.

```
void vecAdd_CPU (int N, const float *A,
                        const float *B,
                        float *C) {
for ( int i = 0; i < N; i++ )
    c[i] = a[i] + b[i];
}
// call vecAdd_CPU on N elements
vecAdd_CPU ( N, a, b, c );
```

```
__global__ void vecAdd_GPU (int N, const float *A,
                                   const float *B,
                                   float *C) {
int i = threadIdx.x + BlockDim.x * BlockIdx.x
if (i < N) c[i] = a[i] + b[i];
}
// call vecAdd_GPU on N elements
dim3 block(32);
dim3 grid( (N-1)/threads.x + 1); // num. of blocks
vecAdd_GPU<<<grid, block>>> ( N, a, b, c );
```

Heterogeneous Interface for Portability aka HIP

HIP is **AMD's GPU** programming environment for designing high performance kernels on GPU hardware.

It provides a C-style API / C++ runtime API and a programming language that allows developers to create portable applications.
Programmers familiar with other GPGPU languages will find HIP easy to use.
It is an interface that uses the underlying Radeon Open Compute (**ROCm**) or **CUDA** platform.
The API is similar to CUDA so porting existing codes from CUDA to HIP should be straightforward.
It is designed to **ease conversion** of CUDA applications to portable C++ code.
HIPify tool automates the conversion by performing a source-to-source transformation from CUDA to HIP source code.

CUDA *vs* HIP

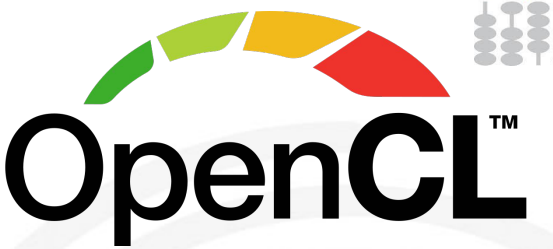
CUDA

```
size_t nbytes = N*sizeof(float);
// allocate memory on device
float *a, *b, *c;
cudaMalloc((void **) &a, nbytes);
cudaMalloc((void **) &b, nbytes);
cudaMalloc((void **) &c, nbytes);
// copy data from host to device
cudaMemcpy(a, a_h, nbytes, cudaMemcpyHostToDevice);
cudaMemcpy(b, b_h, nbytes, cudaMemcpyHostToDevice);
// the kernel
__global__ void vecAdd_GPU(int N,
     const float *A, const float *B, float *C) {
   int i = threadIdx.x + BlockDim.x * BlockIdx.x
   if (i < N) c[i] = a[i] + b[i];
}
// call vecAdd_GPU on N elements
dim3 block(32);
dim3 grid( (N-1)/threads.x + 1);
vecAdd_GPU<<<grid, block>>> ( N, a, b, c );
// copy output from device to host
cudaMemcpy(c_h, c, nbytes, cudaMemcpyDeviceToHost);
```

HIP

```
size_t nbytes = N*sizeof(float);
// allocate memory on device
float *a, *b, *c;
hipMalloc((void **) &a, nbytes);
hipMalloc((void **) &b, nbytes);
hipMalloc((void **) &c, nbytes);
// copy data from host to device
hipMemcpy(a, a_h, nbytes, hipMemcpyHostToDevice);
hipMemcpy(b, b_h, nbytes, hipMemcpyHostToDevice);
// the kernel
__global__ void vecAdd_GPU(int N,
     const float *A, const float *B, float *C) {
  int i = hipBlockDim_x * hipBlockIdx_x + hipThreadIdx_x;
  if (i < N) c[i] = a[i] + b[i];
}
// call vecAdd_GPU on N elements
dim3 block(32);
dim3 grid( (N-1)/threads.x + 1);
hipLaunchKernel(vecAdd_GPU, block, grid, 0, 0,
                N, a, b, c);
// copy output from device to host
hipMemcpy(c_h, c, nbytes, hipMemcpyDeviceToHost);
```

Open Computing Language aka OpenCL



OpenCL is a standard **open-source** programming model developed by major brands of hardware manufacturers (IBM, Intel, AMD, nVIDIA, ARM, Samsung, Qualcomm, Altera).
It provides extensions to **C/C++** for writing programs that execute across heterogeneous platforms with specific hardware like: central processing units (**CPUs**), graphics processing units (**GPUs**), digital signal processors (**DSPs**), field-programmable gate arrays (**FPGAs**) and other processors or hardware accelerators.
OpenCL provides a standard interface for parallel computing for both **task** and **data** parallelism.
It is very **low level** and **verbose** programming language.
OpenCL is an open standard maintained by the non-profit technology consortium **Khronos Group**.

GPU programming in Python

- **pyCUDA**
  - pyCUDA provides Cython/Python wrappers for CUDA driver and runtime APIs.
- **cupy**
  - CuPy is an open-source **NumPy/SciPy compatible** array library for GPU-accelerated computing with Python.
  - CuPy utilizes CUDA Toolkit libraries including cuBLAS, cuRAND, cuSOLVER, cuSPARSE, cuFFT, cuDNN and NCCL to make full use of the GPU architecture.
  - CuPy's interface is highly compatible with NumPy and SciPy; in most cases it can be used as a drop-in replacement. `import cupy as cp`
- **numba**
  - Numba allows to write standard Python functions and run them on a CUDA-capable GPU.
  - Two approaches can be used:
    - A function **decorator** to instruct Numba to compile for the GPU. For example, the `@vectorize` with `target='cuda'` that process arrays of data in parallel on the GPU.
    - Numba exposes the **CUDA programming model**, just like in CUDA C/C++, but using pure python syntax.