Introduction to GPU Architectures

Master in High Performance Computing (MHPC)
@ICTP Trieste

12-14 February 2025

Sergio Orlandini

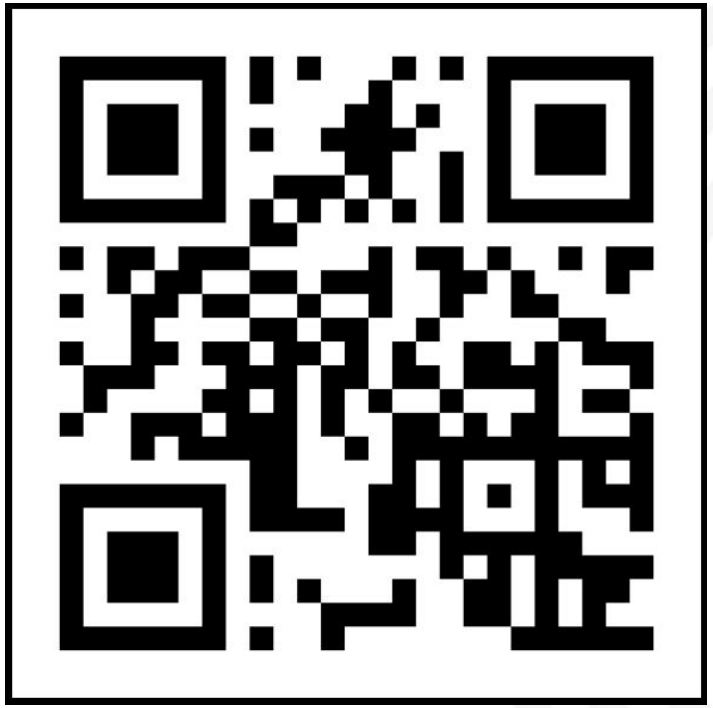
s.orlandini@cineca.it

CINECA

- GPU Architectures
- CINECA Leonardo System
- CUDA Recap
- Hands-on
- Nsight-Compute Demo

Gentle Course Starting Poll

https://etc.ch/hNvy



CINECA GPU-Hackathon 2025

CINECA OPEN HACKATHON

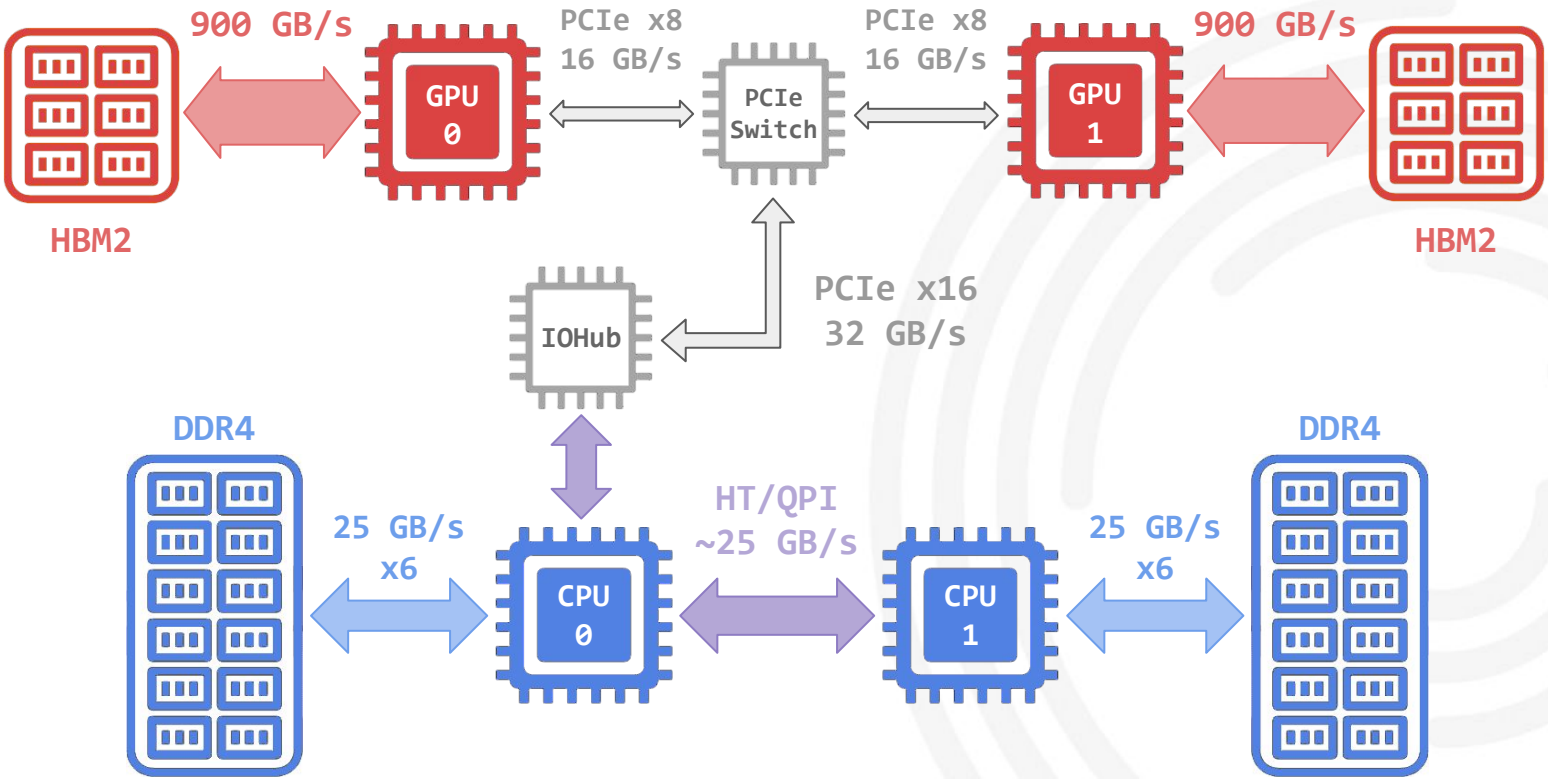
June 24, 2025 - July 03, 2025
Application Deadline: April 29, 2025
Virtual Event

CINECA GPU-HACKATHON-2025

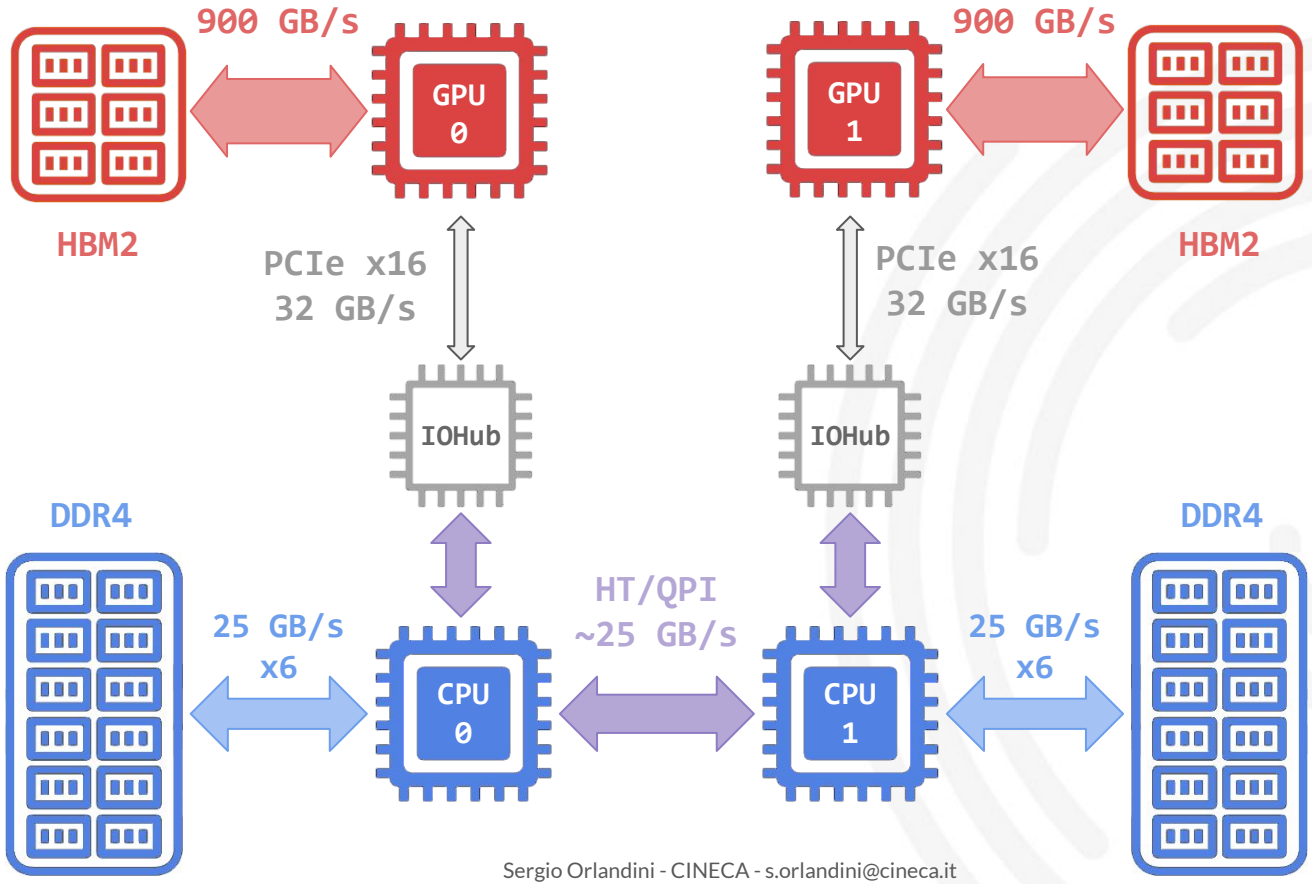
HPC + **AI** Hackathon

https://www.openhackathons.org/s/siteevent/a0CUP00000sOUqi2AG/se000365

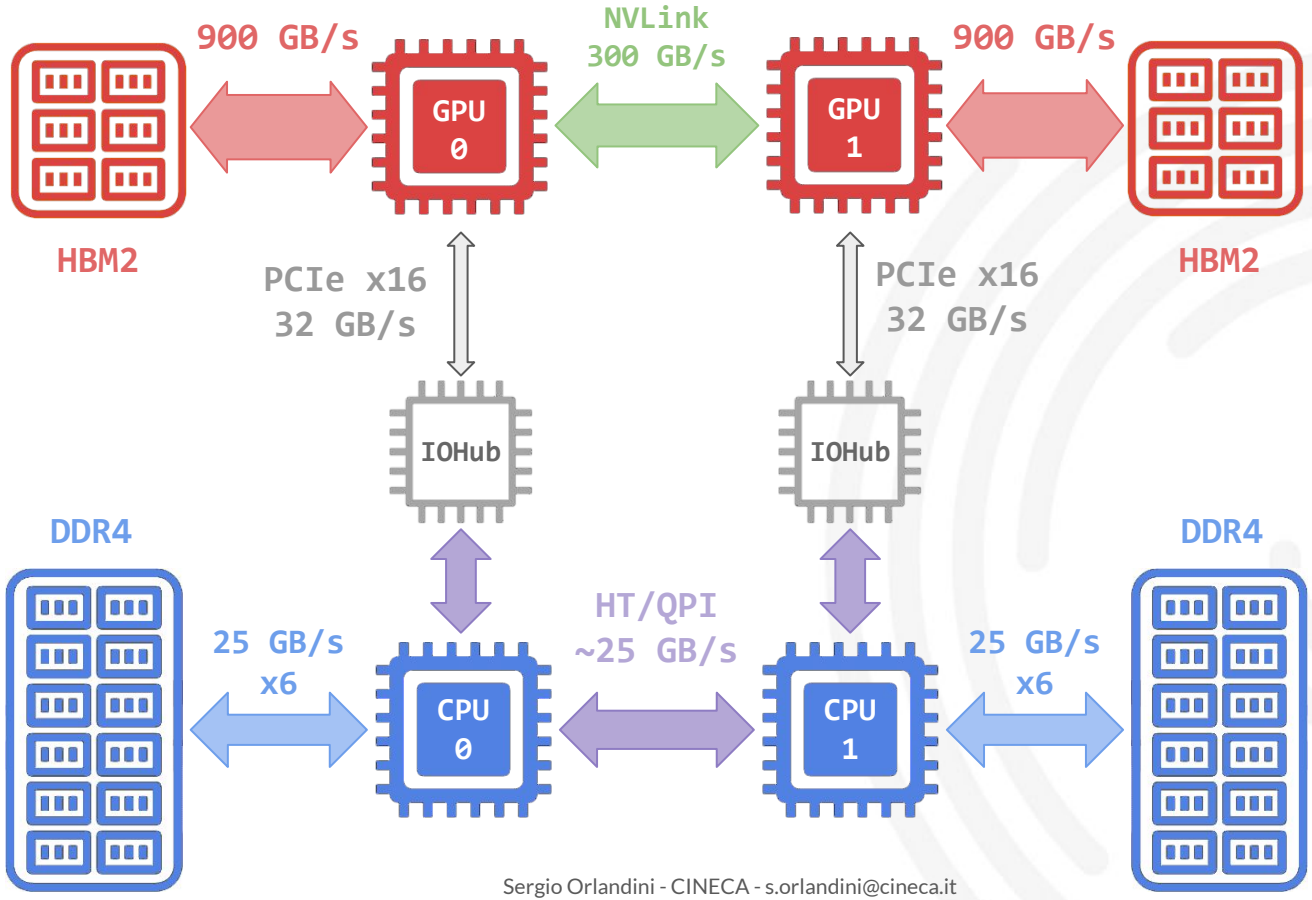
Compute Node Schema



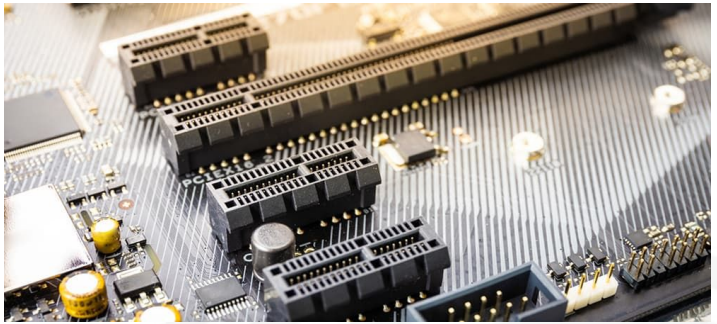
Compute Node Schema



Compute Node Schema



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

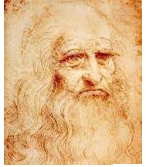
PCI Express link performance		
Version	Year	BW x16 GB/s
1.0	2003	4.0
2.0	2007	8.0
3.0	2010	15.7
4.0	2017	31.5
5.0	2019	63.0

LEONARDO

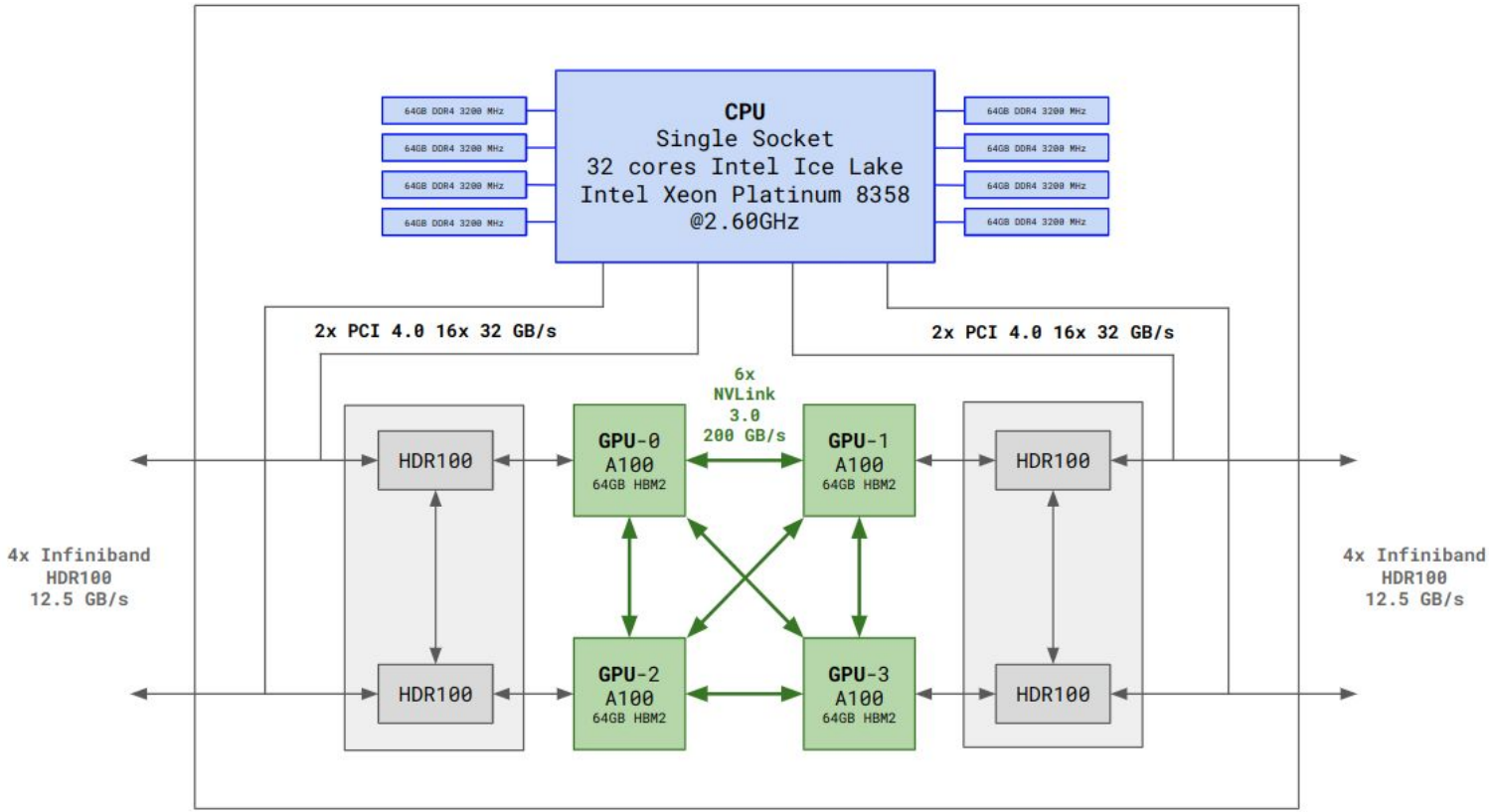
CINECA

TOP500 June 2023

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,699,904	1,194.00	1,679.82	22,703
2	**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
3	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	2,220,288	309.10	428.70	6,016
4	**Leonardo** - BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, NVIDIA A100 SXM4 64 GB, Quad-rail NVIDIA HDR100 Infiniband, Atos EuroHPC/CINECA Italy	1,824,768	238.70	304.47	7,404



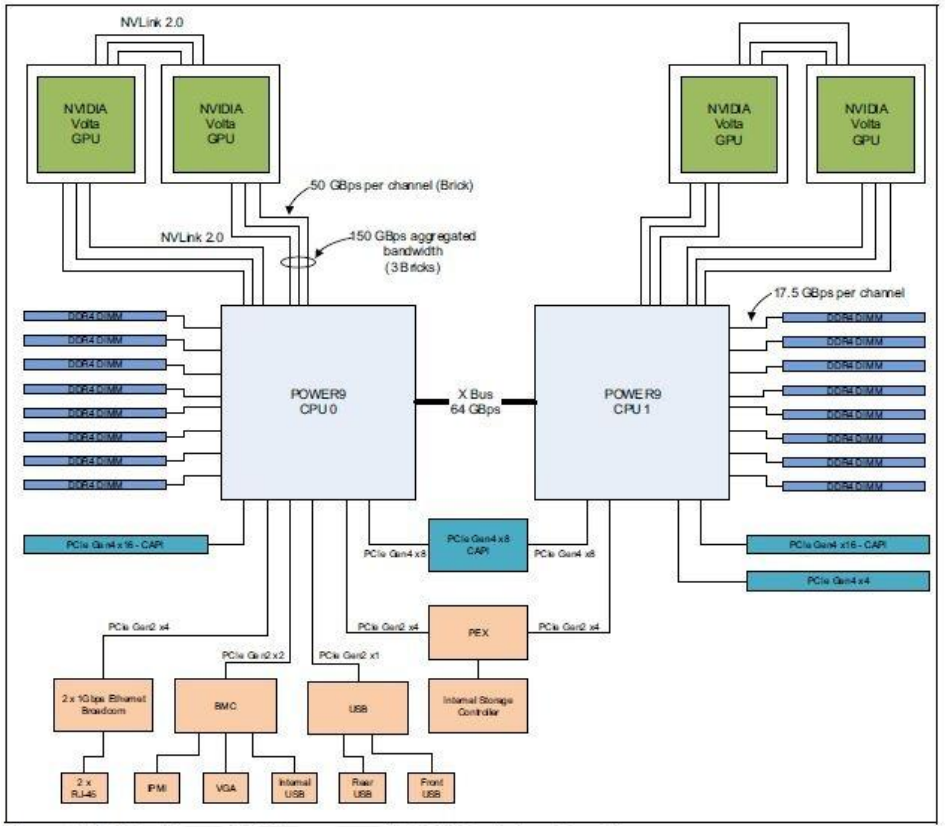
LEONARDO: Booster Module

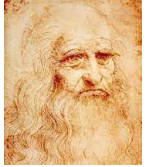


Marconi100 @CINECA aka M100

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





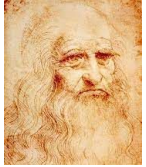
GPU Nvidia Ampere

The NVIDIA System Management Interface (aka **`nvidia-smi`**) is a command for monitoring NVIDIA GPU devices.

Several details are listed such as:

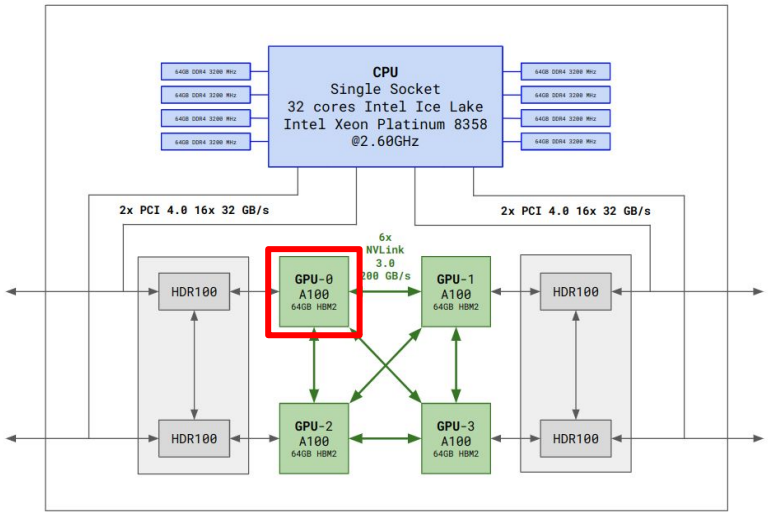
- the CUDA and GPU driver versions,
- the number and type of GPUs available,
- the GPU memory each,
- running GPU process,
- etc.

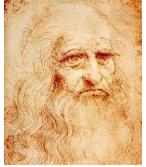
NB: with the option `-l SEC`, `nvidia-smi` continuously reports query data at the specified interval in seconds. Useful for monitoring resource utilisation on a node.



GPU Nvidia Ampere

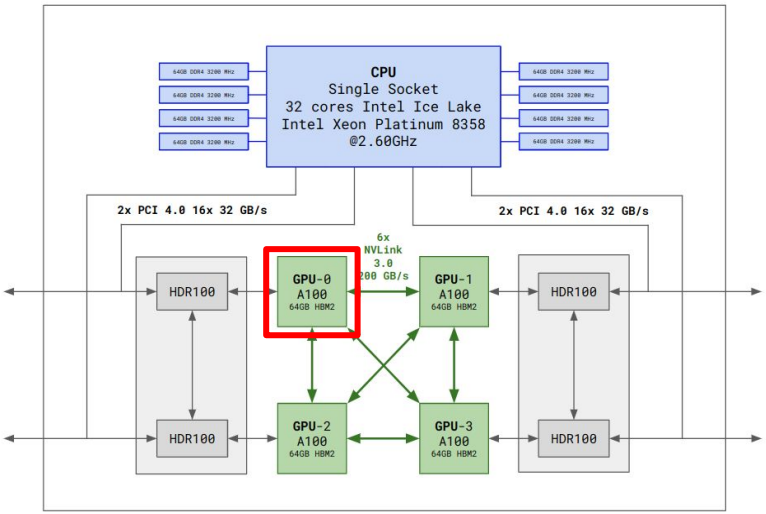
```
$ nvidia-smi

+-----------------------------------------------------------------------------+
| NVIDIA-SMI 520.61.05    Driver Version: 520.61.05    CUDA Version: 11.8     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA PG506-243    On   | 00000000:1D:00.0 Off |                    0 |
| N/A   38C    P0    60W / 467W |      0MiB / 65536MiB |      0%      Default |
|                               |                      |             Disabled |
+-------------------------------+----------------------+----------------------+
|   1  NVIDIA PG506-243    On   | 00000000:56:00.0 Off |                    0 |
| N/A   39C    P0    59W / 456W |      0MiB / 65536MiB |      0%      Default |
|                               |                      |             Disabled |
+-------------------------------+----------------------+----------------------+
|   2  NVIDIA PG506-242    On   | 00000000:8F:00.0 Off |                    0 |
| N/A   38C    P0    61W / 469W |      0MiB / 65536MiB |      0%      Default |
|                               |                      |             Disabled |
+-------------------------------+----------------------+----------------------+
|   3  NVIDIA PG506-242    On   | 00000000:C8:00.0 Off |                    0 |
| N/A   39C    P0    61W / 461W |      0MiB / 65536MiB |      0%      Default |
|                               |                      |             Disabled |
+-------------------------------+----------------------+----------------------+
```

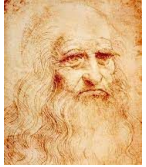




GPU Nvidia Ampere

```
$ DeviceQuery

Device 0: "NVIDIA PG506-243"
  CUDA Driver Version / Runtime Version          11.8 / 11.8
  CUDA Capability Major/Minor version number:    8.0
  Total amount of global memory:                 64969 MBytes (68125065216 bytes)
  (124) Multiprocessors, (064) CUDA Cores/MP:    7936 CUDA Cores
  GPU Max Clock rate:                            1395 MHz (1.39 GHz)
  Memory Clock rate:                             1593 Mhz
  Memory Bus Width:                              4096-bit
  L2 Cache Size:                                 33554432 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 163
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total shared memory per multiprocessor:        167936 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 5 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:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Managed Memory:                Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 29 / 0
```





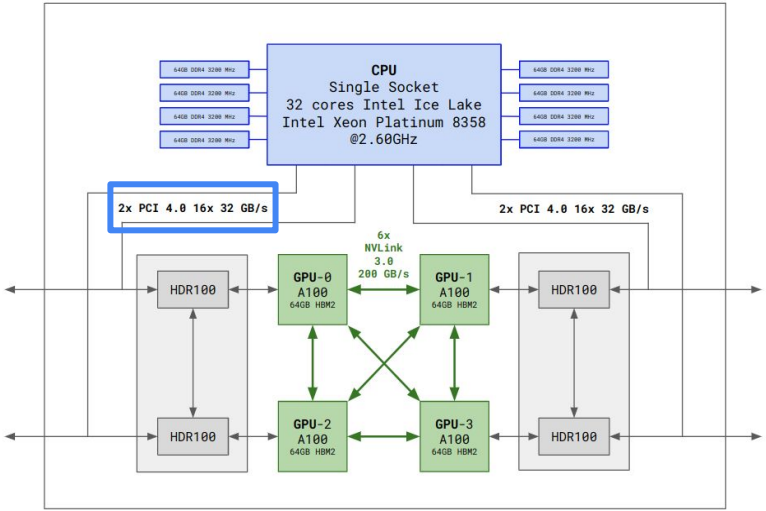
Communication Buses

PCI Express 4th Gen.

- 4x CPU/GPU Connection
- **32 GB/s** Bidirectional Bandwidth (16x lanes)

NVLink 3.0

- NVIDIA high-speed coherent interconnect Technology GPU-to-GPU and GPU-to-CPU
- 6x GPU/GPU Connection
- **200 GB/s** peak Bidirectional Bandwidth



Bandwidth & Latency

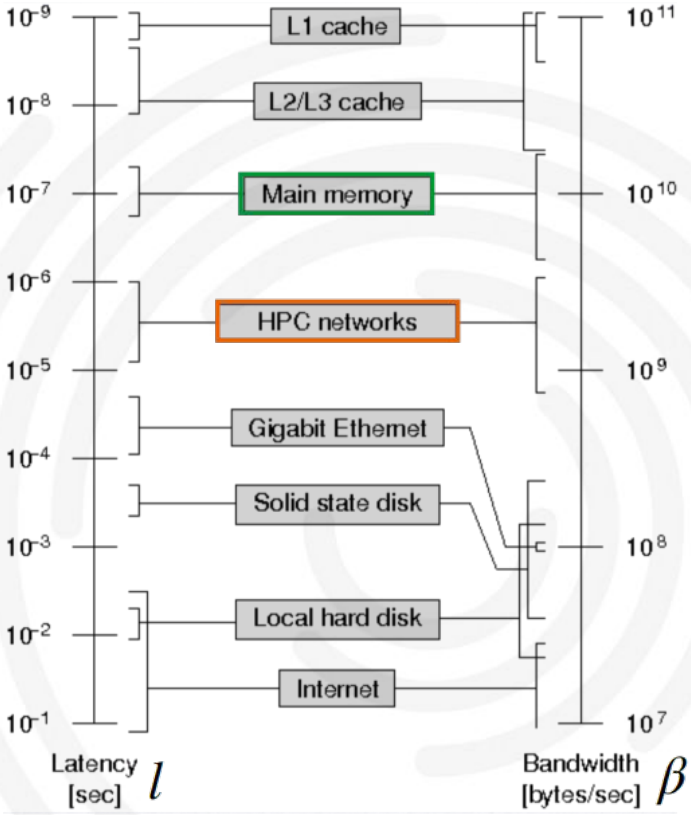
The time t to transmit a message of size S is:

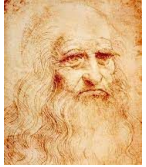
$$t = L + \frac{S}{B}$$

where:

- L is the **latency** (delay in starting communications)
- B is the **bandwidth** (rate of data transfer for a fixed period of time)

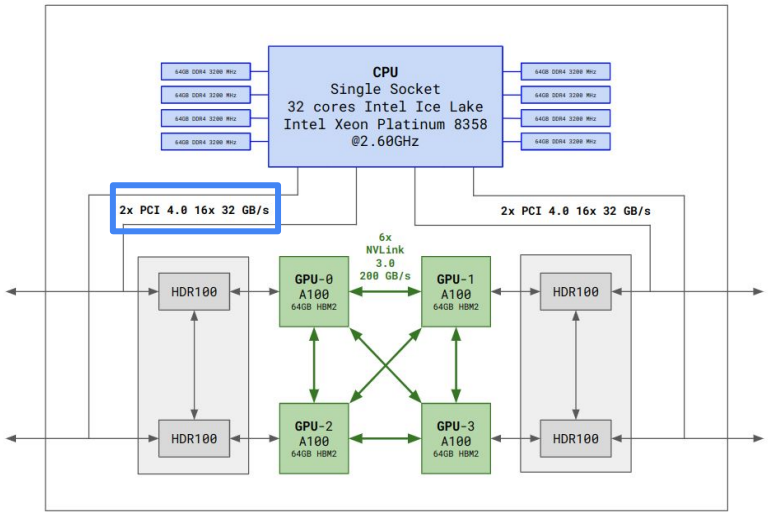
In general **network communication** is much slower than memory access, therefore is good practice to hide the communication behind computation. This can be done thanks to **non-blocking communication**.

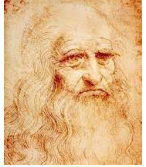




CPU-GPU Communication with PCI Exp.

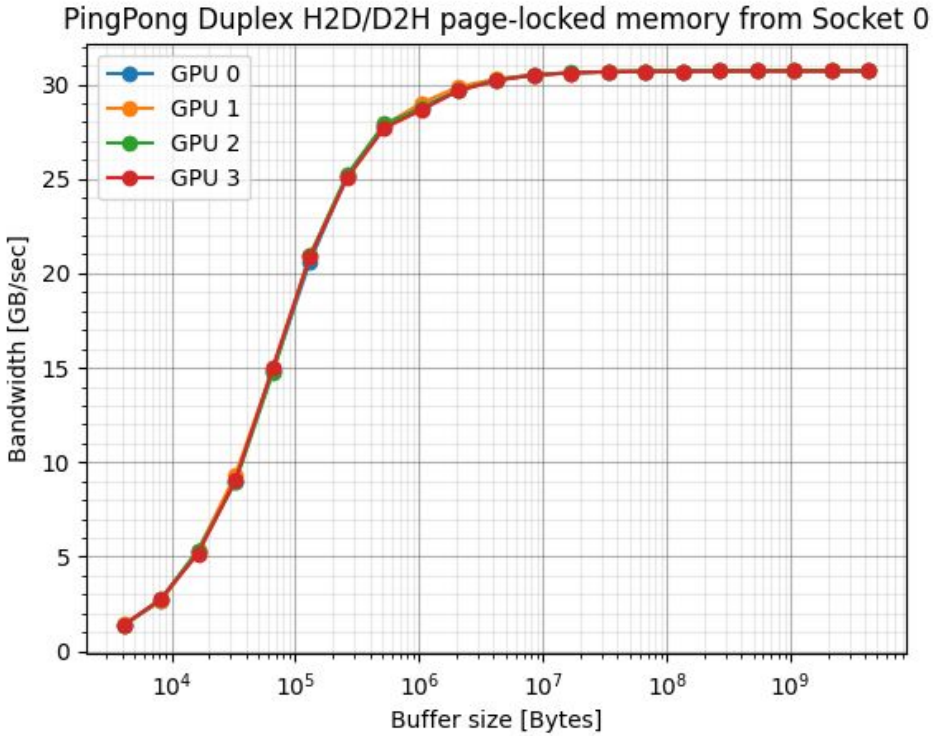
```
$ bandwidthTest

[CUDA Bandwidth Test] - Starting...
Running on...

 Device 0: NVIDIA PG506-243
 Quick Mode

 Host to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)    Bandwidth(GB/s)
   32000000                 24.9

 Device to Host Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)    Bandwidth(GB/s)
   32000000                 25.9

 Device to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)    Bandwidth(GB/s)
   32000000                 1163.9

Result = PASS

NOTE: The CUDA Samples are not meant for performance measurements. Results may vary when GPU Boost is enabled.
```

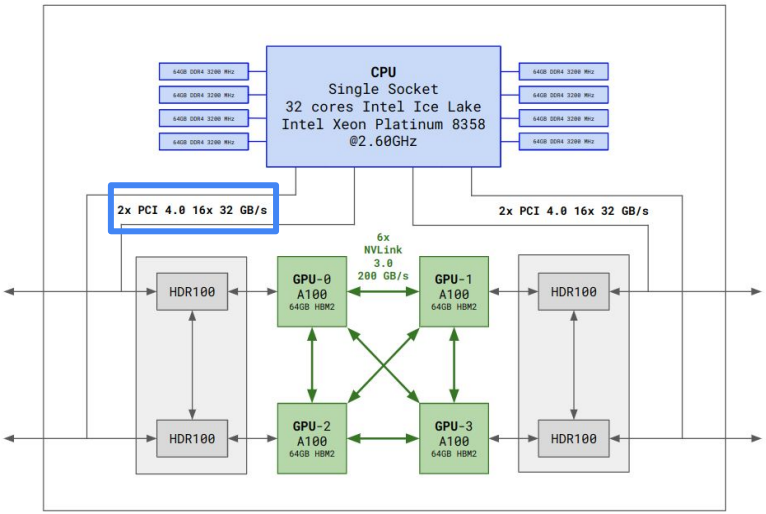




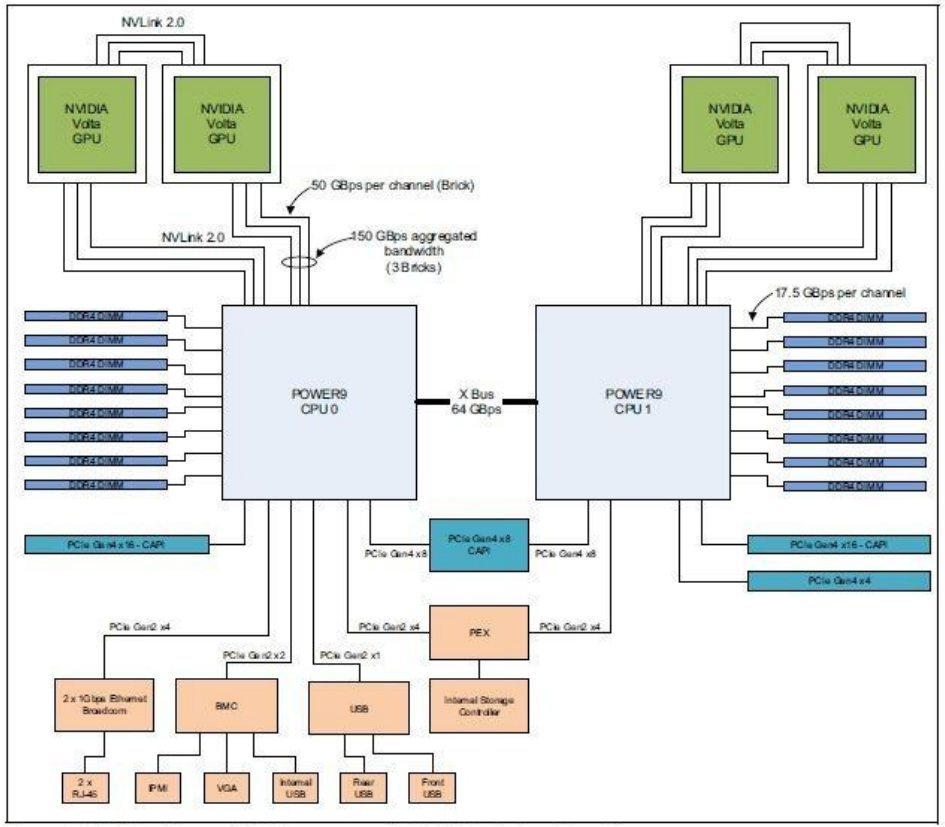
CPU-GPU Communication with PCI Exp.

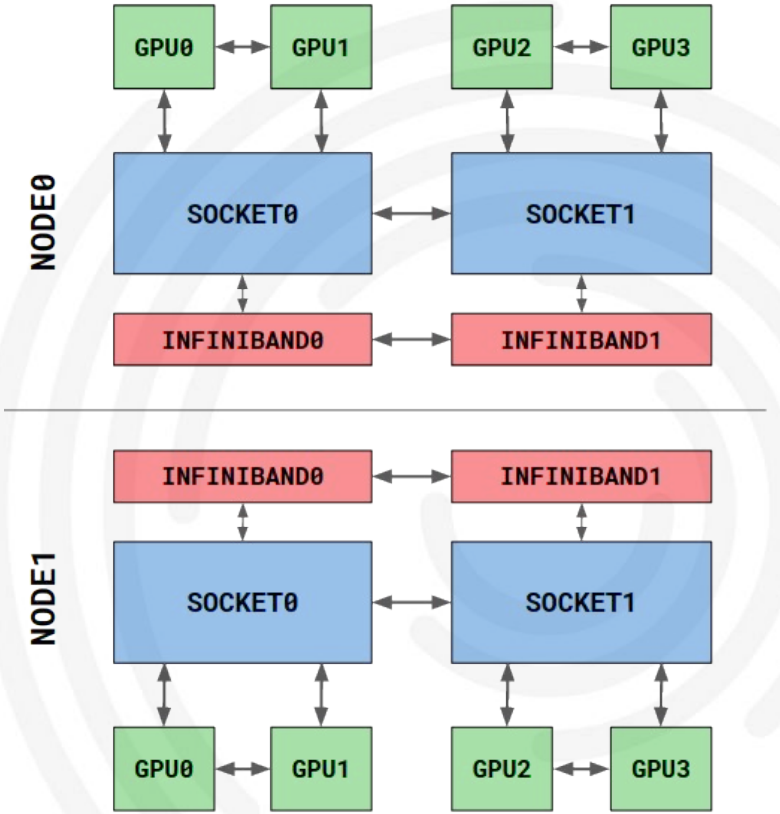
PingPong Duplex mode H2D/D2H Page-locked Memory





Marconi100 @CINECA aka M100

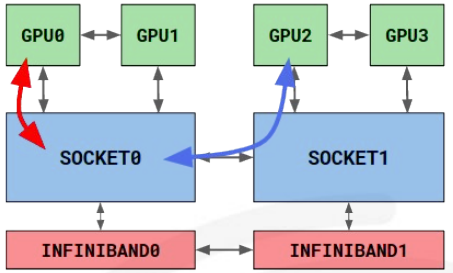


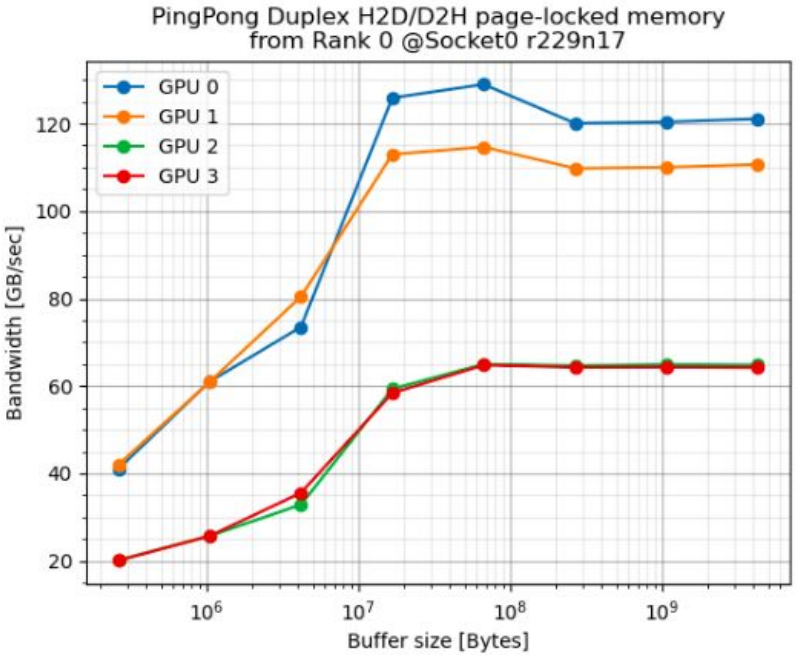


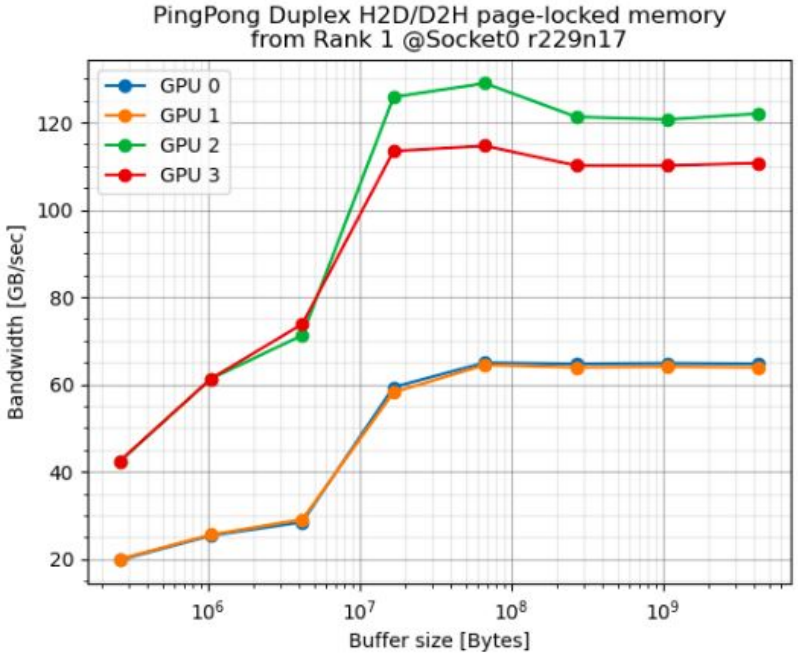
Host-Device Communications

H2D/D2H Bandwidth in duplex mode with **NVLink**

@Marconi100 with 4 V100







Blocking and Non-blocking Functions

- Every CUDA action is submitted to an execution queue on the device
- CUDA runtime functions can be divided in two categories:

- **blocking** (synchronous):
  return control to host thread after execution is completed on device
  - all memory transfer > 64KB
  - all memory allocation on device
  - allocation of page locked memory on host

- **Non-blocking** (asynchronous):
  return control to host immediately, while its execution proceeds on device
  - kernel launches
  - memory transfers < 64KB
  - memory initialization on device (cudaMemset)
  - memory copies from device to device
  - explicit asynchronous memory transfers

- CUDA API provides asynchronous versions of their counterpart synchronous functions
- Asynchronous functions allows to set up concurrent execution of many operations on host and device

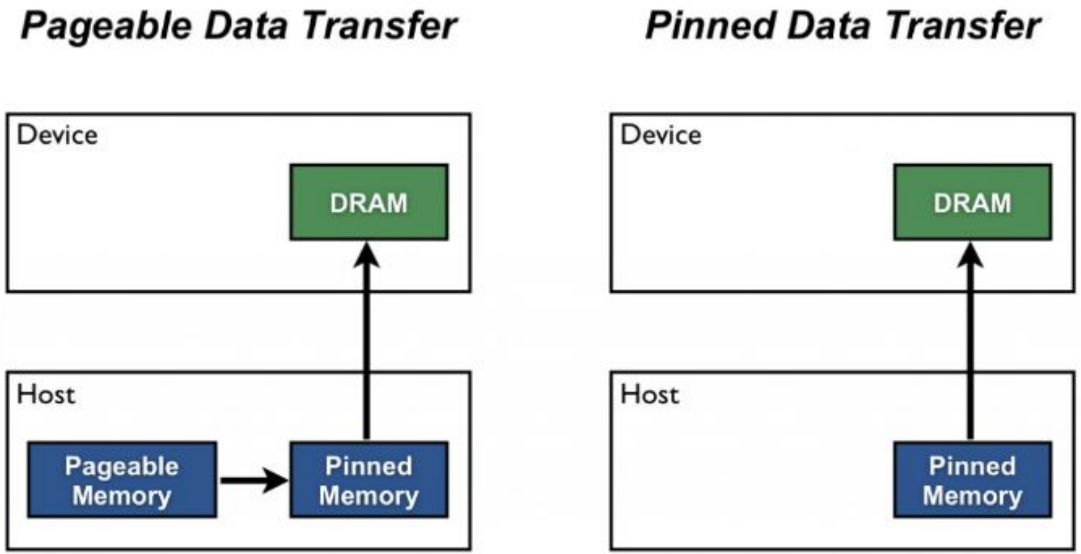
CUDA Streams

- GPU operations are implemented in CUDA using execution queues, called **streams**
- Each operation pushed in a stream will be executed only after all other operations in the same stream are completed (FIFO queue behaviour)
- Operations assigned to different streams can be executed in any order with respect each other
- CUDA runtime provides a **default stream** (aka stream 0) which will be the default queue of all operation if otherwise is not explicitly declared

Memory pinning

Overlapping data movements with computation requires **memory pinning**

compile flag: **`-gpu=pinned`**



PAGEABLE (DEFAULT)

When a data transfer is invoked, the CUDA driver first allocate a temporary page-locked (pinned) host array, copy the host data to the pinned array, and then transfer the data from the pinned array to device memory.

PINNED

The host memory is allocated directly as “pinned” and the copy from pageable to pinned is skipped.

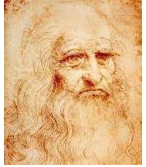
- Allocating pinned memory is slower
- CUDA APIs can pin selectively

Asynchronous Data Transfers

- In order to perform asynchronous data transfers between host and device the host memory must be of **page-locked** type (a.k.a **pinned**)
- CUDA runtime provides the following functions to handle page-locked memory:
  - `cudaMallocHost()`: allocate page-locked memory on host
  - `cudaFreeHost()`: free page-locked allocated memory on host
  - `cudaHostRegister()`: turn host allocated memory into page-locked
  - `cudaHostUnregister()`: turn page-locked memory into ordinary memory
- `cudaMemcpyAsync()` function explicitly performs asynchronous data transfers between host and device memory
- Data transfer operations must queued into a stream different from the default one in order to be asynchronous
- Using page-locked memory allows data transfers between host and device memory with **higher bandwidth**

Asynchronous Data Transfers

```
cudaStreamCreate(stream_a)
cudaStreamCreate(stream_b)

cudaMallocHost(h_buffer_a, buffer_a_size)
cudaMallocHost(h_buffer_b, buffer_b_size)

cudaMalloc(d_buffer_a, buffer_a_size)
cudaMalloc(d_buffer_b, buffer_b_size)

// concurrent and asynchronous data transfer H2D and D2H
cudaMemcpyAsync(d_buffer_a, h_buffer_a, buffer_a_size, cudaMemcpyHostToDevice, stream_a)
cudaMemcpyAsync(h_buffer_b, d_buffer_b, buffer_b_size, cudaMemcpyDeviceToHost, stream_b)

cudaStreamDestroy(stream_a)
cudaStreamDestroy(stream_b)

cudaFreeHost(h_buffer_a)
cudaFreeHost(h_buffer_b)
```



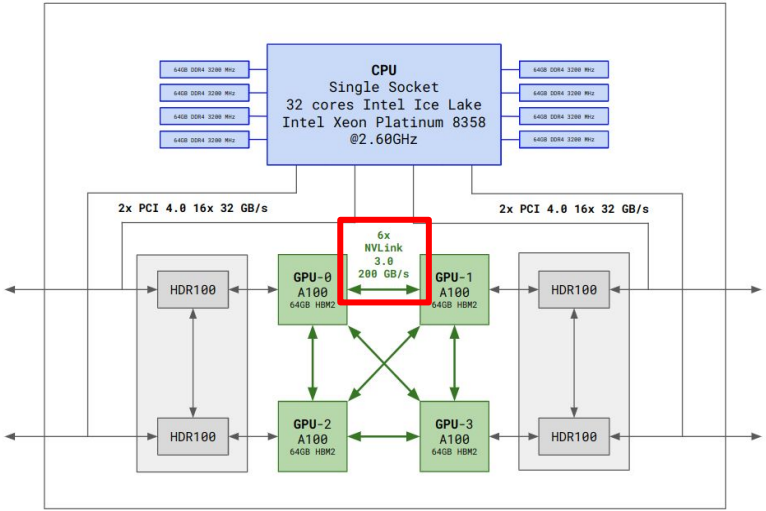
Communication Buses

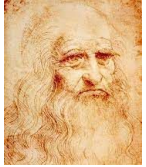
PCI Express 4th Gen.

- 4x CPU/GPU Connection
- **32 GB/s** Bidirectional Bandwidth (16x lanes)

NVLink 3.0

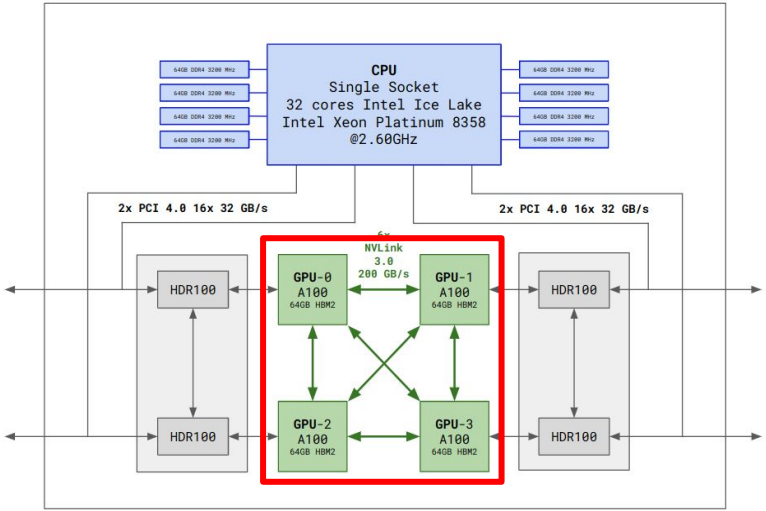
- NVIDIA high-speed coherent interconnect Technology GPU-to-GPU and GPU-to-CPU
- 6x GPU/GPU Connection
- **200 GB/s** peak Bidirectional Bandwidth

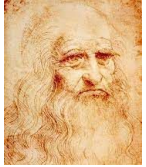




GPU Topology

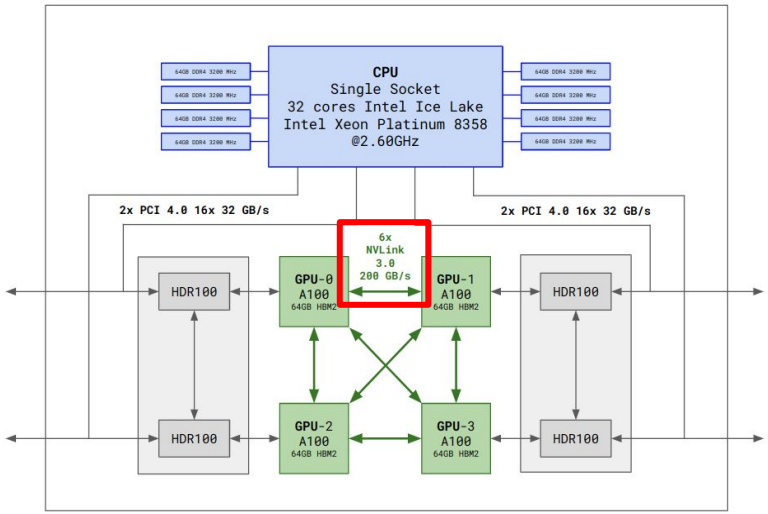
```
$ nvidia-smi topo -m
      GPU0  GPU1  GPU2  GPU3  mlx5_0      mlx5_1      mlx5_2
GPU0   X    NV4   NV4   NV4   PXB   SYS   SYS   SYS   0     0-1
GPU1  NV4    X    NV4   NV4   SYS   PXB   SYS   SYS   0     0-1
GPU2  NV4   NV4    X    NV4   SYS   SYS   PXB   SYS   0     0-1
GPU3  NV4   NV4   NV4    X    SYS   SYS   SYS   PXB   0     0-1
mlx5_0      PXB   SYS   SYS   SYS    X    SYS   SYS   SYS
mlx5_1      SYS   PXB   SYS   SYS   SYS    X    SYS   SYS
mlx5_2      SYS   SYS   PXB   SYS   SYS   SYS    X    SYS
mlx5_3      SYS   SYS   SYS   PXB   SYS   SYS   SYS    X

Legend:

  X    = Self
  SYS  = Connection traversing PCIe as well as the SMP interconnect between NUMA nodes (e.g., QPI/UPI)
  NODE = Connection traversing PCIe as well as the interconnect between PCIe Host Bridges within a NUMA node
  PHB  = Connection traversing PCIe as well as a PCIe Host Bridge (typically the CPU)
  PXB  = Connection traversing multiple PCIe bridges (without traversing the PCIe Host Bridge)
  PIX  = Connection traversing at most a single PCIe bridge
  NV#  = Connection traversing a bonded set of # NVLinks
```





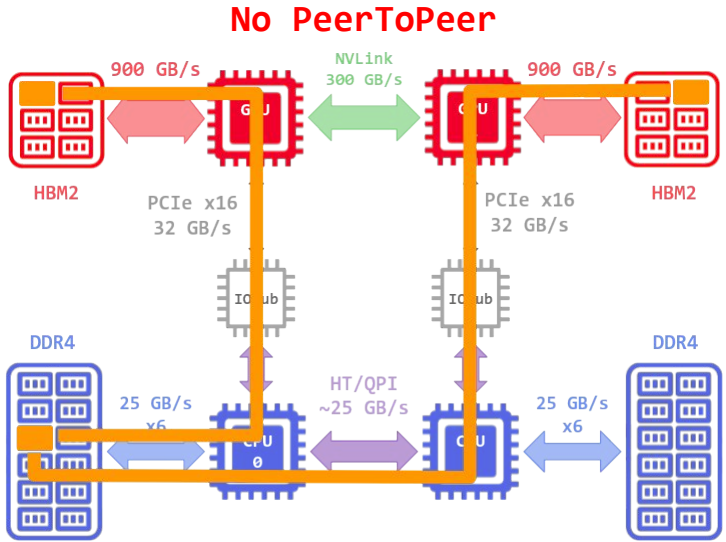
GPU-GPU Communication with NVLink

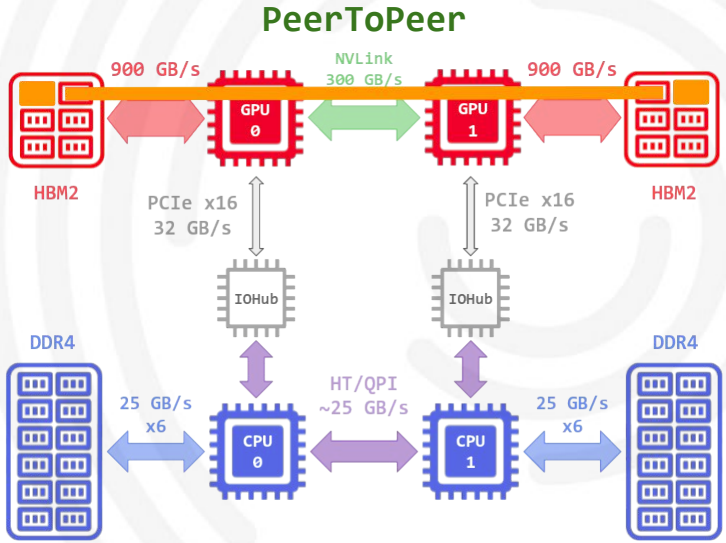
```
$ p2pBandwidthLatencyTest

Unidirectional P2P=Enabled Bandwidth (P2P Writes) Matrix (GB/s)
  D\D     0       1       2       3
    0 1286.01   93.55   93.60   93.39
    1   93.57 1335.47   93.35   93.48
    2   93.55   93.37 1335.47   93.61
    3   93.29   93.56   93.55 1337.76

Bidirectional P2P=Enabled Bandwidth Matrix (GB/s)
  D\D     0       1       2       3
    0 1309.72  185.13  185.35  184.87
    1  185.40 1344.66  185.00  185.46
    2  185.36  184.96 1345.82  185.33
    3  185.11  185.48  185.28 1348.14

P2P=Enabled Latency (P2P Writes) Matrix (us)
  GPU     0       1       2       3
    0    2.37    2.24    2.27    2.25
    1    2.30    2.28    2.23    2.19
    2    2.27    2.22    2.38    2.26
    3    2.19    2.27    2.28    2.52
```



Peer to Peer Transfers

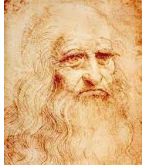
- A device can directly transfer or access data to/from another device.
- This kind of direct transfer is called Peer to Peer (P2P).
- P2P transfers are more efficient and do not require a host buffer.
- Direct access avoid host memory copy.





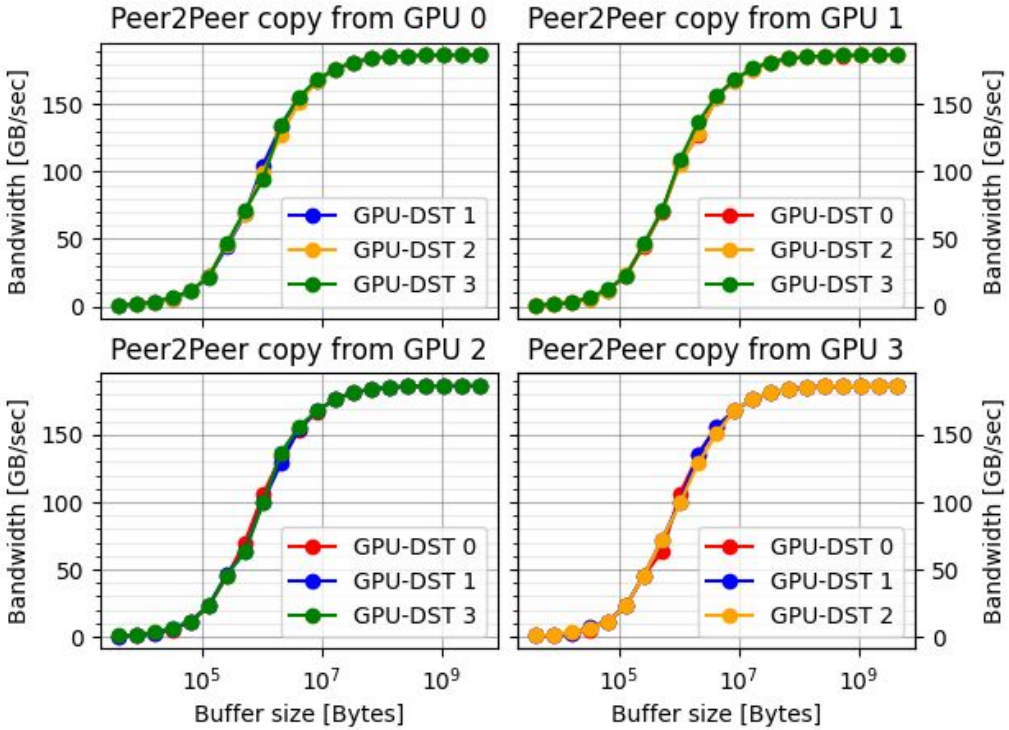
Peer to Peer Transfer Pseudocode

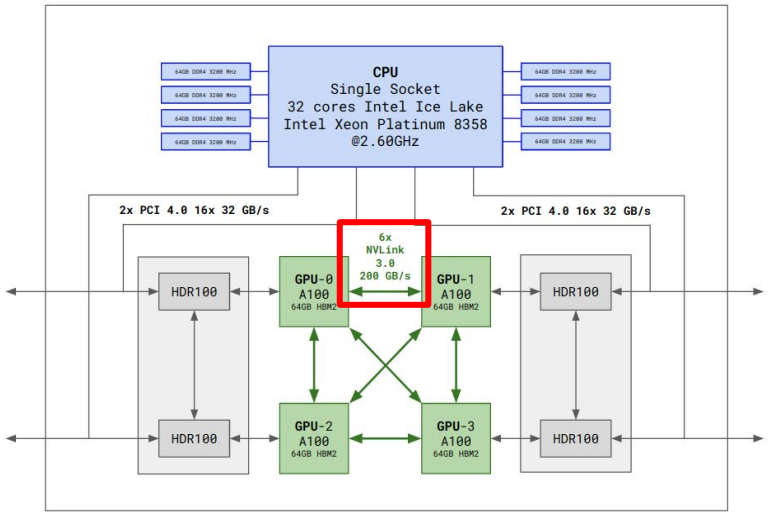
```
gpuA=0, gpuB=1
cudaSetDevice(gpuA)
cudaMalloc(buffer_A, buffer_size)

cudaSetDevice(gpuB)
cudaMalloc(buffer_B, buffer_size)

cudaSetDevice(gpuA)
cudaDeviceCanAccessPeer(answer, gpuA, gpuB)

If answer is true:
    cudaDeviceEnablePeerAccess(gpuB, 0)
    // gpuA performs copy from gpuA to gpuB
    cudaMemcpyPeer(buffer_B, gpuB, buffer_A, gpuA, buffer_size)
    // gpuA performs copy from gpuB to gpuA
    cudaMemcpyPeer(buffer_A, gpuA, buffer_B, gpuB, buffer_size)
```



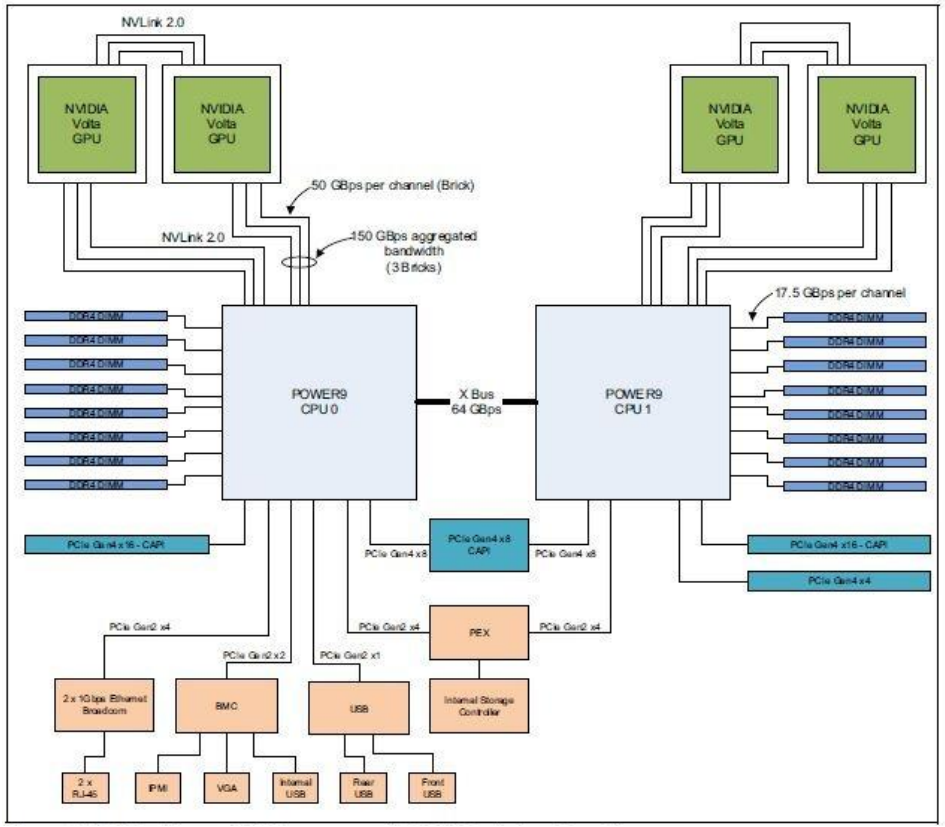
GPU-GPU Communication with NVLink

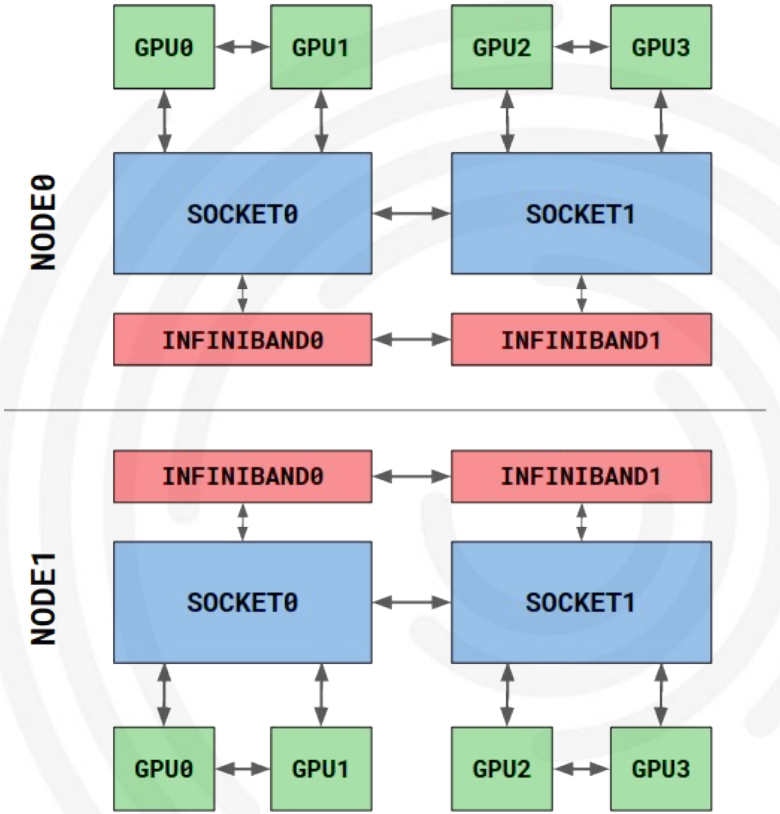
PingPong Peer2Peer copies - P2P Enabled





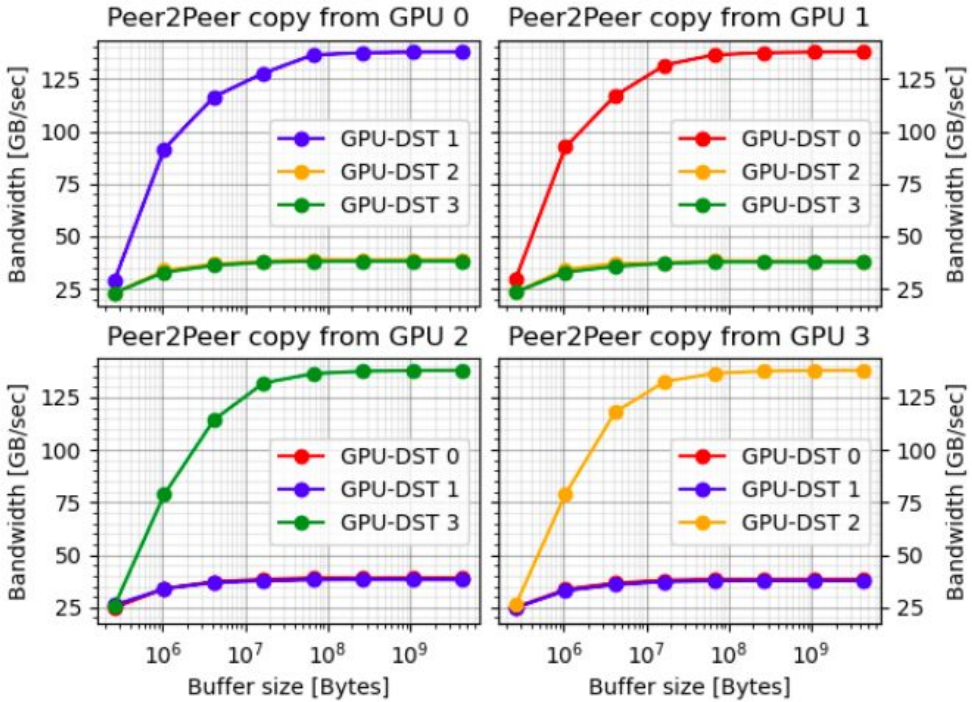
Marconi100 @CINECA aka M100

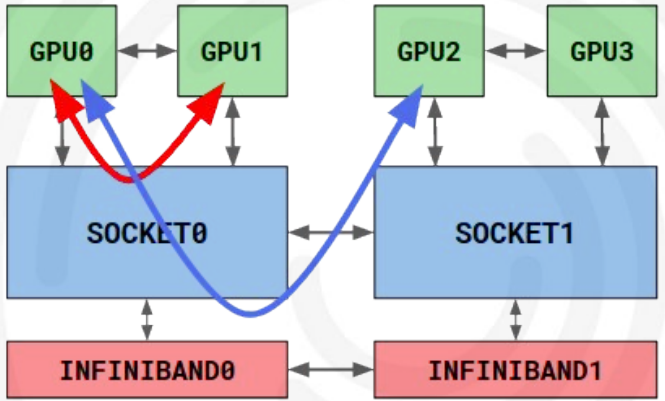




Device-Device Communications

PingPong **Peer2Peer** copies **@Marconi100** with 4 V100





CUDA-AWARE MESSAGE PASSING INTERFACE

MPI implementations may also be **aware of CUDA** programming model.

With CUDA-Aware MPI, messages can be sent directly between **GPU memories**, even between GPUs of **different nodes**.

This greatly simplifies the code required for MPI communications and memory management. Indeed, copies between device and host are no longer required before exchanging MPI messages.

Nowadays, MPI and CUDA interaction are supported by ***OpenMPI***, ***MVAPICH2*** and IBM ***Spectrum_MPI***.

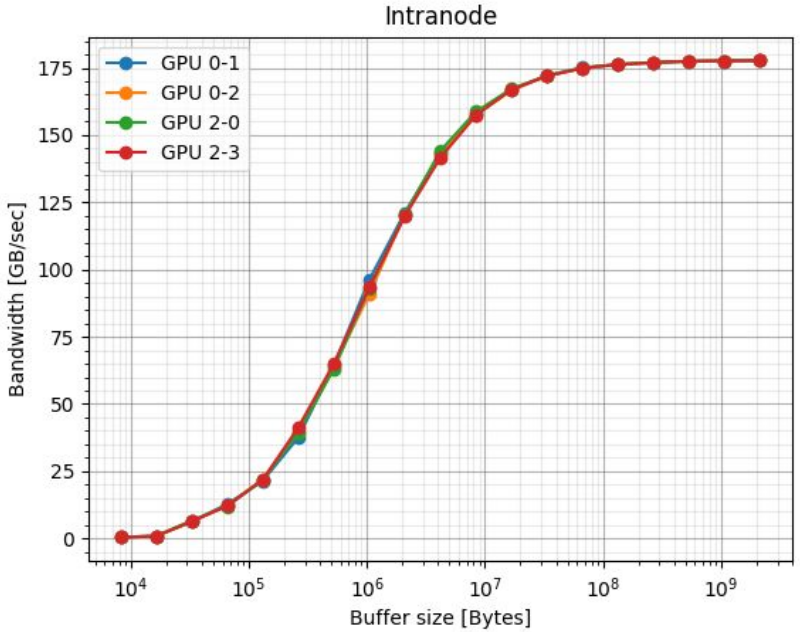
MPI-CUDA-AWARE

```
MPI_Init(&argc, &argv)
MPI_Comm_rank(MPI_COMM_WORLD, &rank)
cudaSetDevice(rank)

cudaMalloc(buffer_send, buffer_size)
cudaMalloc(buffer_recv, buffer_size)

if(rank == 0) {

  MPI_Irecv(buffer_recv, buffer_size, MPI_INT, 1, 43, MPI_COMM_WORLD, &request[1]);

  MPI_Isend(buffer_send, buffer_size, MPI_INT, 1, 42, MPI_COMM_WORLD, &request[0]);

} else if(rank == 1) {

  MPI_Irecv(buffer_recv, buffer_size, MPI_INT, 0, 42, MPI_COMM_WORLD, &request[0]);

  MPI_Isend(buffer_send, buffer_size, MPI_INT, 0, 43, MPI_COMM_WORLD, &request[1]);

}

MPI_Waitall(2, request, status); // Returns only after all the requests have completed

cudaFree(buffer_send)
cudaFree(buffer_recv)

MPI_Finalize()
```

MPI-CUDA-AWARE Intra-Node Point-to-Point

MPI-CUDA-AWARE Point2Point **Intra-Node** Bidirectional Bandwidth

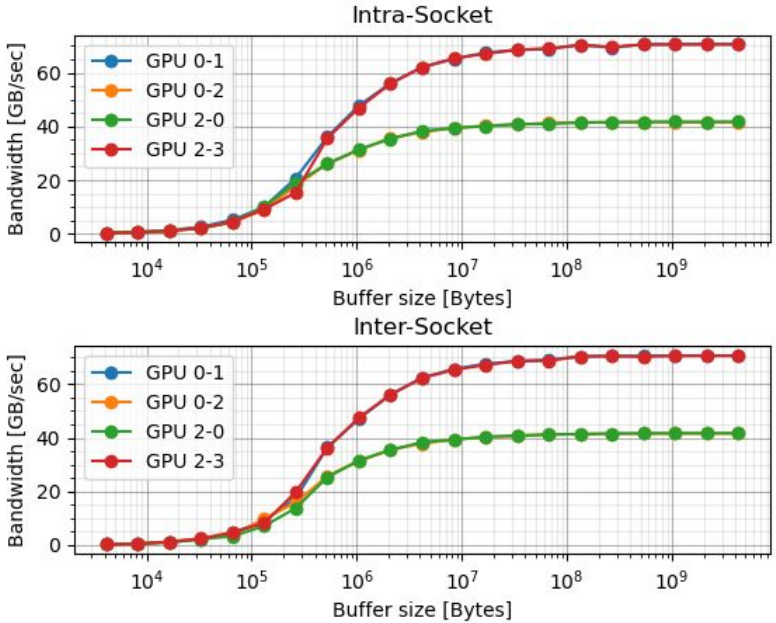
@Leonardo with 4 A100

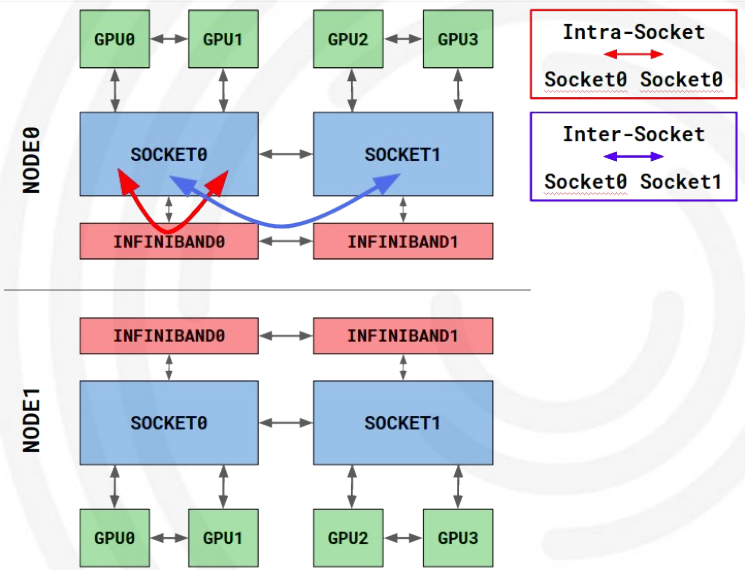


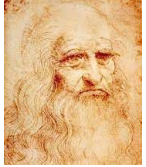
MPI-CUDA-AWARE Intra-Node Point-to-Point

MPI-CUDA-AWARE Point2Point **Intra-Node** Bidirectional Bandwidth

@Marconi100 with 4 V100

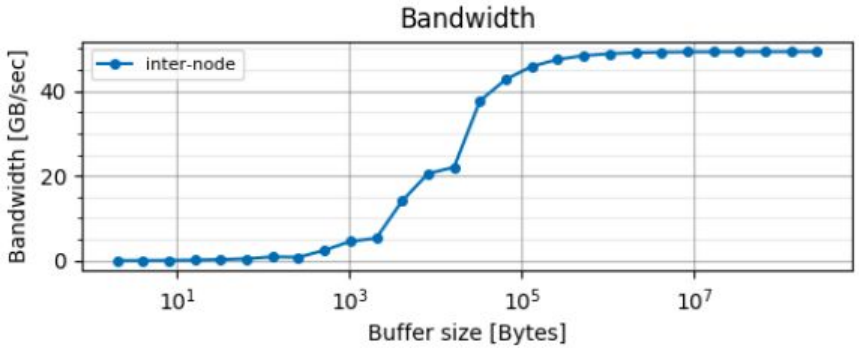




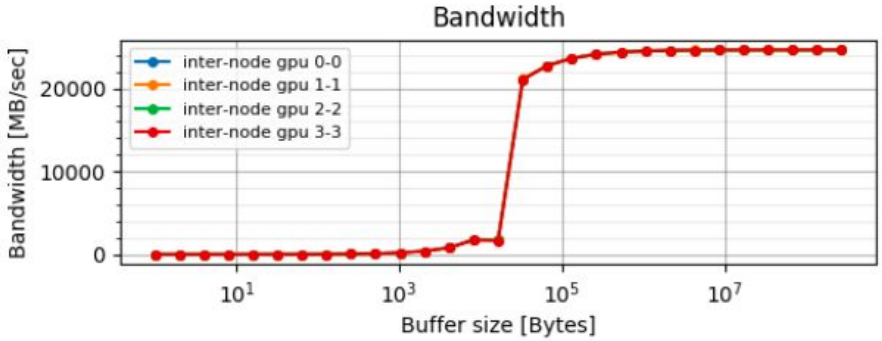


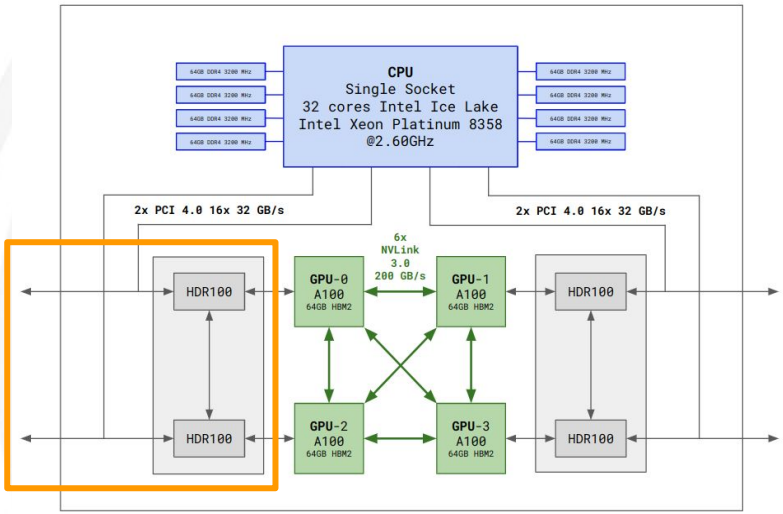
Inter-Node Commun. with MPI-CUDA-AWARE

- MPI Inter-node pt2pt Bidirectional Host-Host



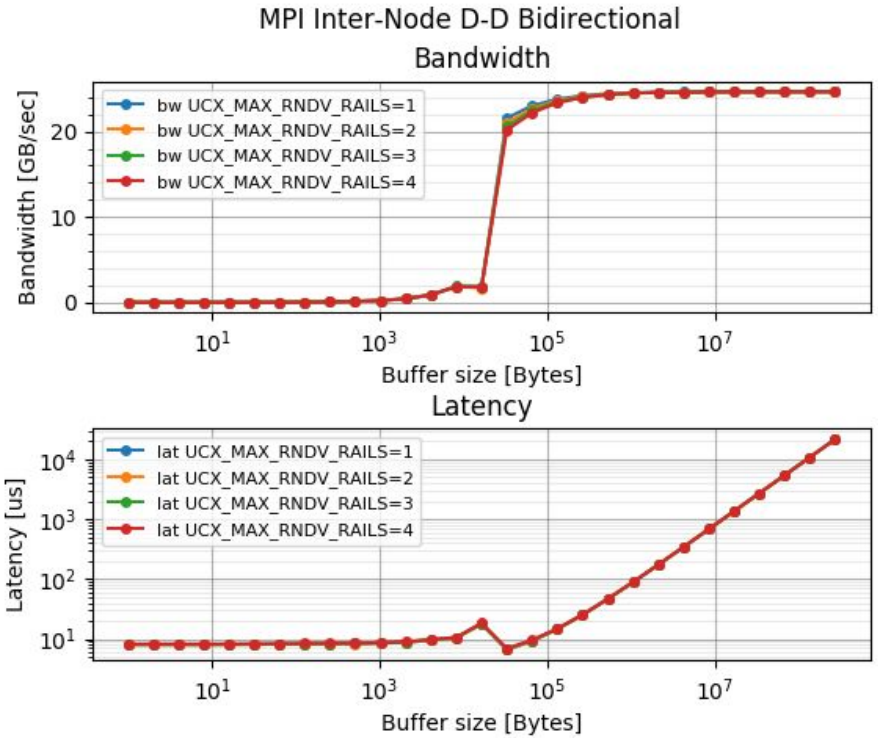
- MPI Inter-node pt2pt Bidirectional Device-Device





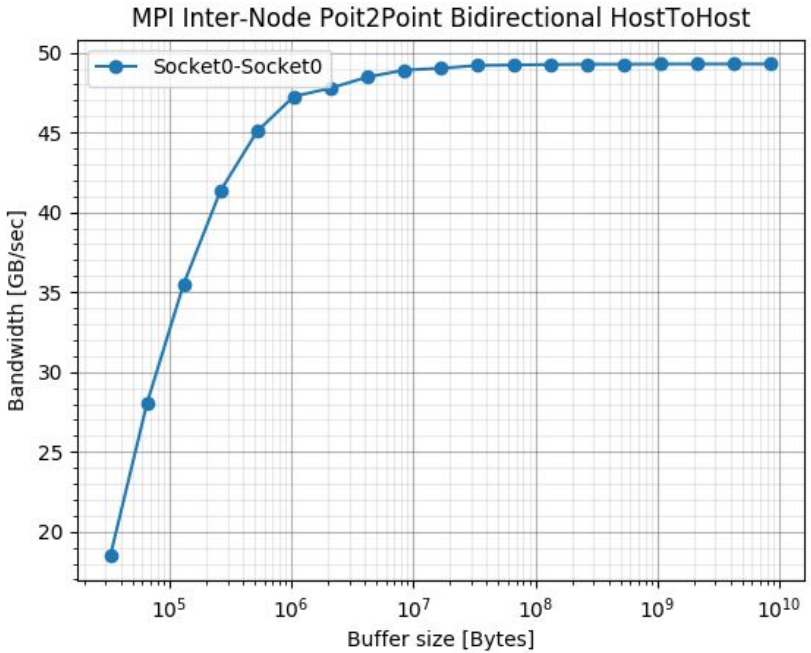
MPI-CUDA-AWARE Inter-Node Point-to-Point

MPI-CUDA-AWARE Point2Point **Inter-Node** Bidirectional Bandwidth
@Leonardo with 4 A100



MPI-CUDA-AWARE Inter-Node Point-to-Point

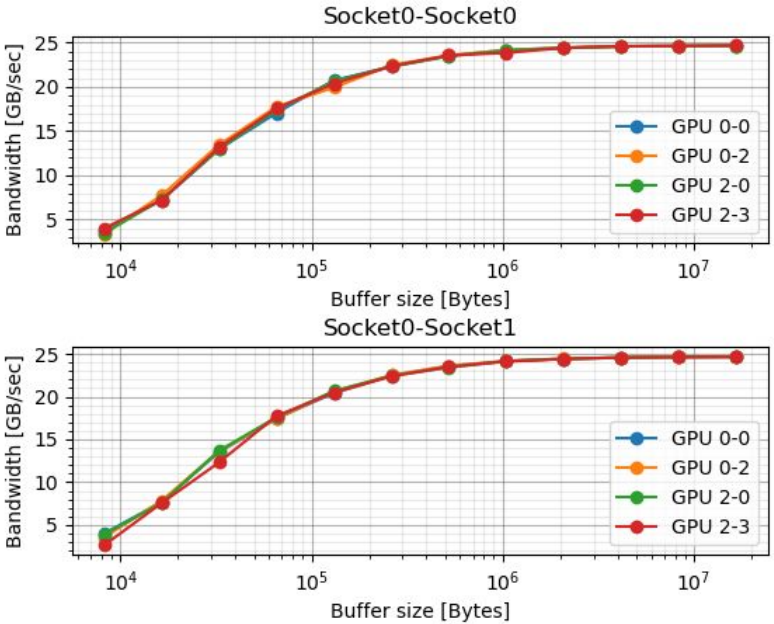
MPI-CUDA-AWARE Point2Point **Inter-Node** Bidirectional Bandwidth
@Leonardo with 4 A100

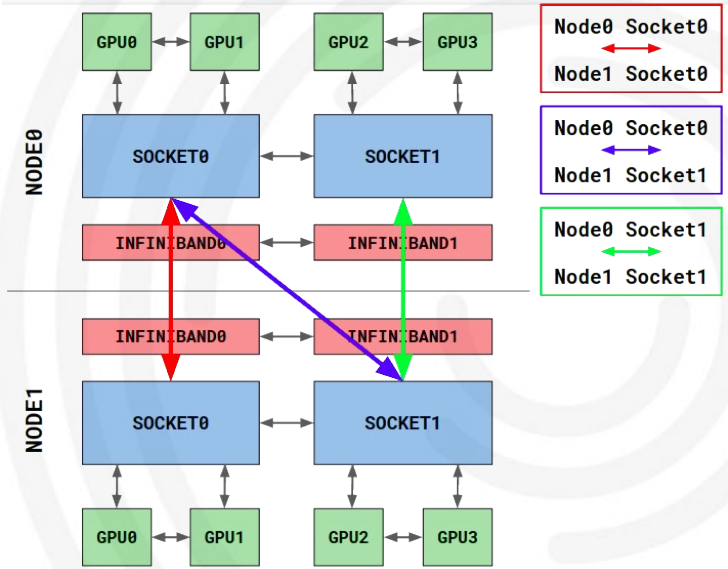


MPI-CUDA-AWARE Inter-Node Point-to-Point

MPI-CUDA-AWARE Point2Point **Inter-Node** Bidirectional Bandwidth

@Marconi100 with 4 V100

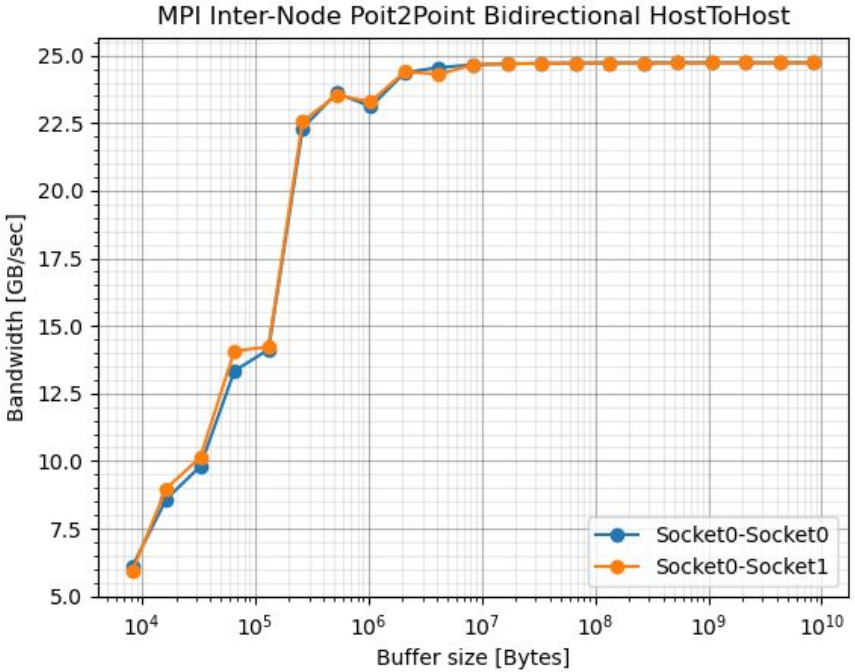


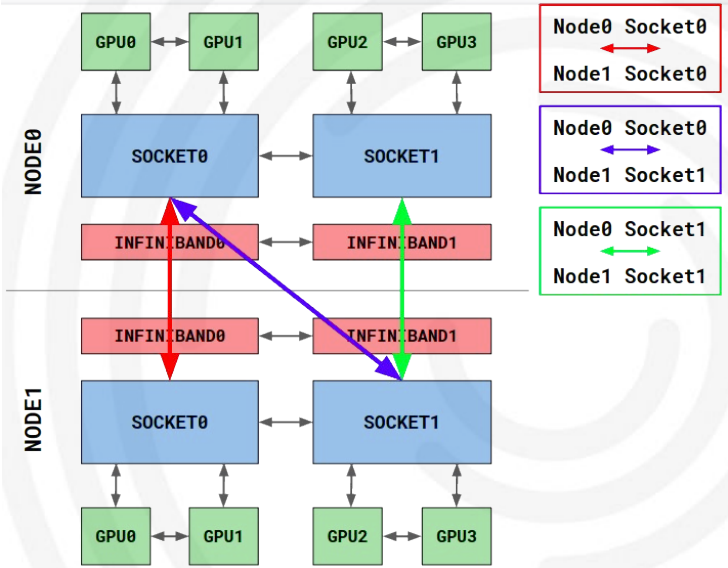


MPI Inter-Node Point-to-Point Host-Host

MPI-CUDA-AWARE Point2Point **Inter-Node** Bidirectional Bandwidth

@Marconi100 with 4 V100





GPU Architectures

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

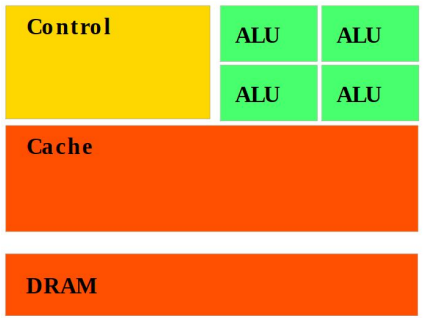
CPU vs GPU

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



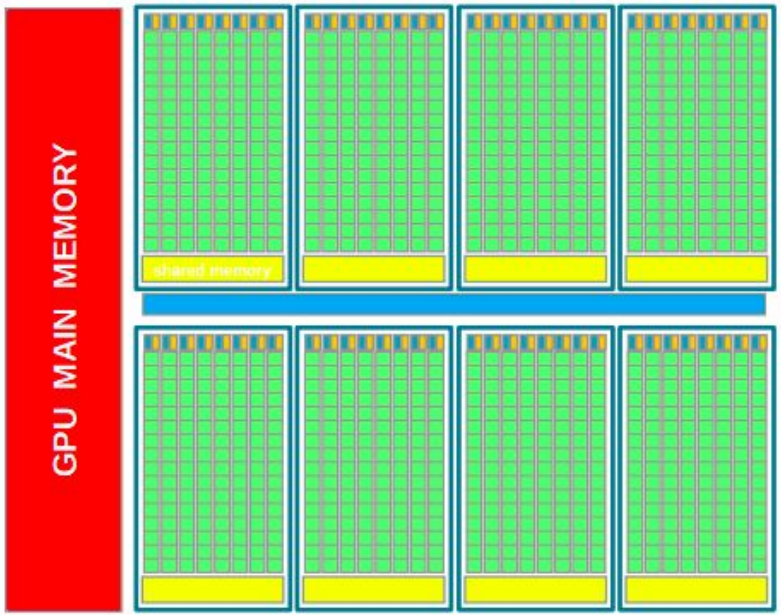
CPU

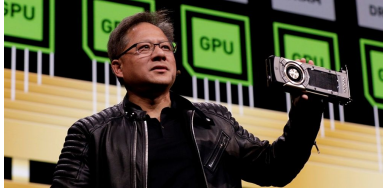
GPU

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





NVIDIA HPC GPU Solutions

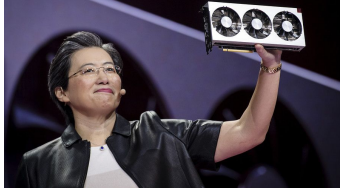
Nvidia's GPU solutions:

- **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 Ampere A100 Architecture (2020)

- https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/
- A full GA100 GPU unit contains 8 Compute Graphic Clusters (CGC) with 16 SM each, total 128 SMs
- 8192 FP32 cores
- 8192 INT32 cores
- 40MB L2 cache
- High Bandwidth Memory
  - 40GB HBM2
  - 1500 GB/s bandwidth
- NVLink technology
- Peak Performance:
  - 19.5 FP32 TFlops
- Max Power Consumption:
  - 400W



AMD HPC GPU Solutions

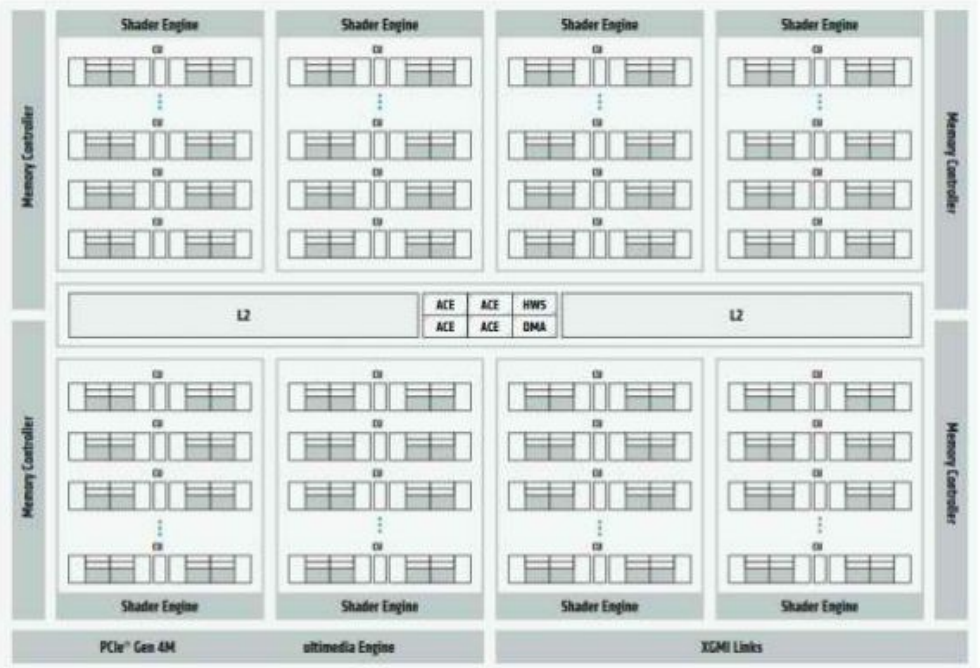
AMD GPU solution lines:

- 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

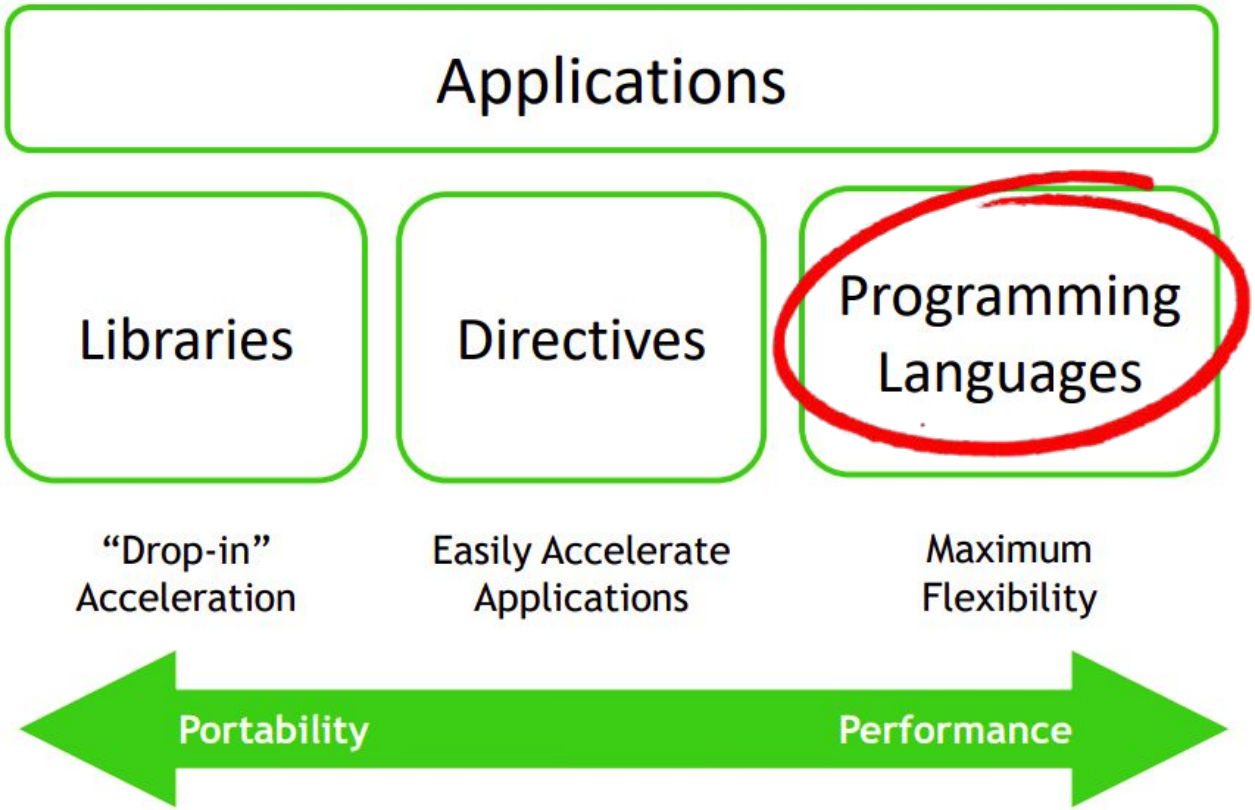
AMD Radeon MI100 Architecture (2020)

- 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



CUDA Recap

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:

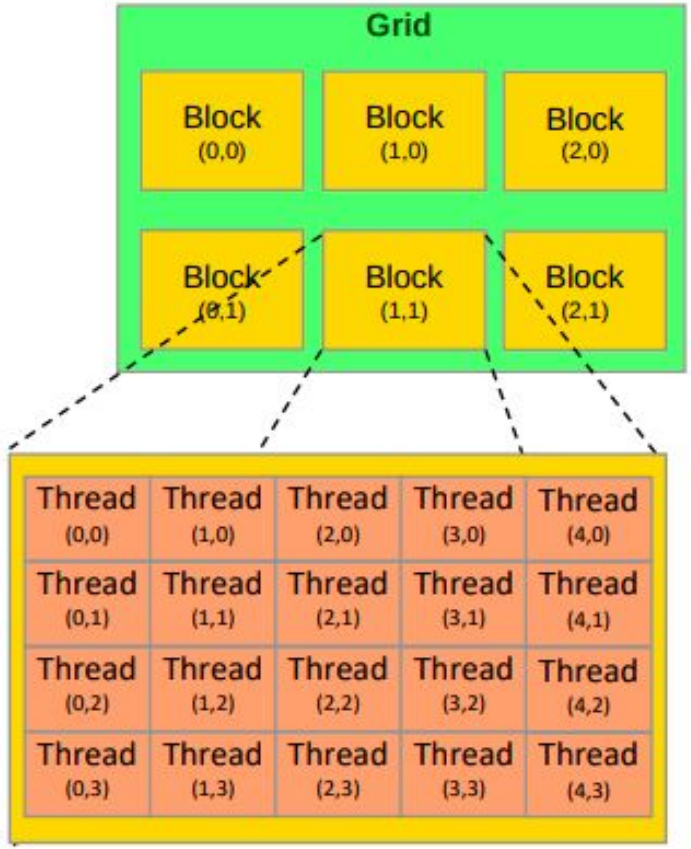
- a hierarchical multi-threaded **programming paradigm** that matches GPU hardware structure,
- a set of **extensions to higher level programming languages** (C/C++ and Fortran) to use GPU as a coprocessor for heavy parallel task and to express thread parallelism within a familiar programming environment,
- 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.

Three steps for a CUDA porting

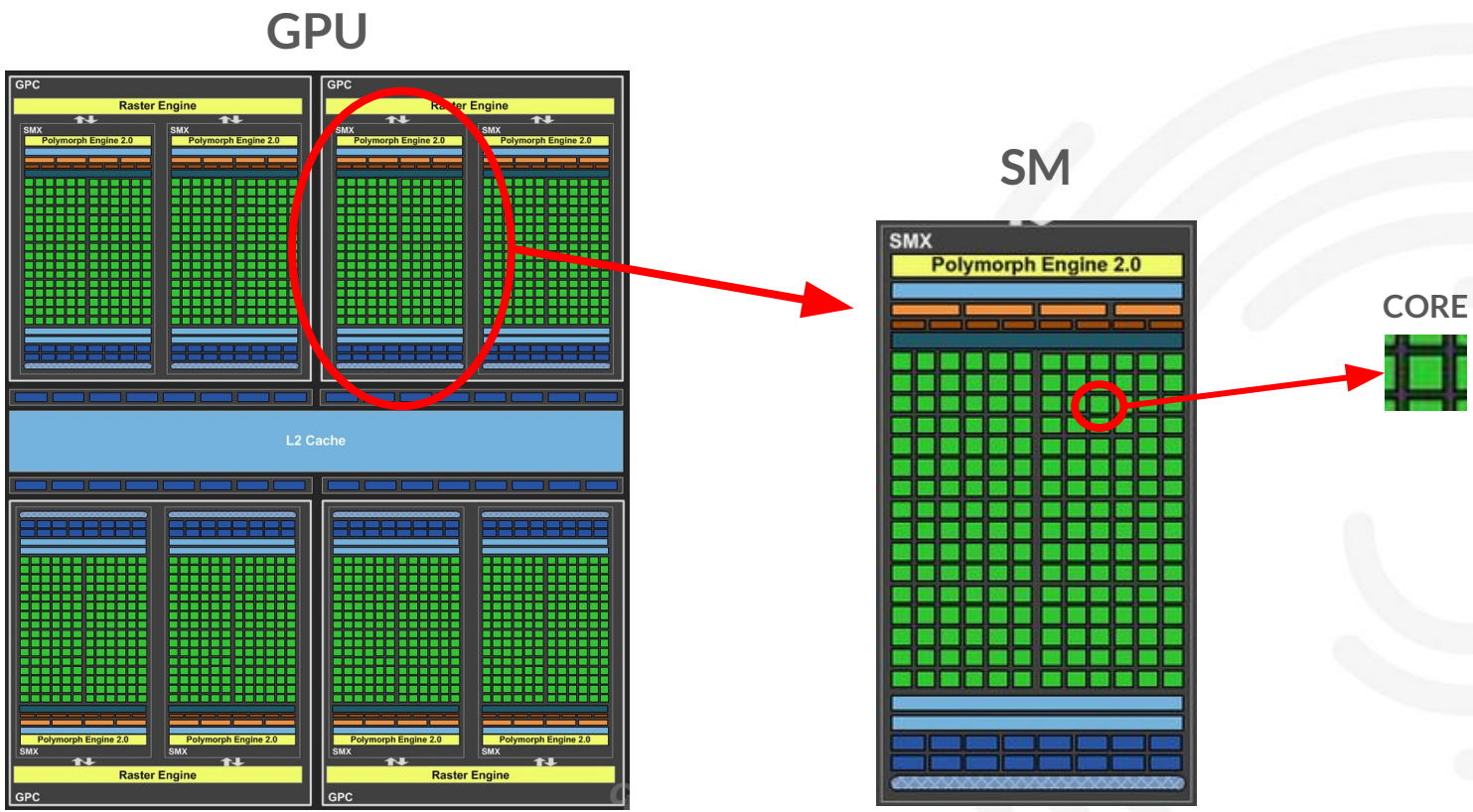
1. Identify **data-parallel** and **computational intensive** portions:
   - Isolate them into functions (CUDA kernels candidates).
   - Identify involved data to be moved between CPU and GPU.
2. Translate identified CUDA kernel candidates into real **CUDA kernels** functions:
   - Choose the appropriate thread index map to access data.
   - Change code so that each thead acts on its own data.
3. Modify code in order to **manage memory** and **kernel calls**:
   - Allocate memory on the device.
   - Transfer needed data from host to device memory.
   - Insert calls to CUDA kernel with execution configuration syntax.
   - Transfer resulting data from device to host memory.

GPU Thread Hierarchy

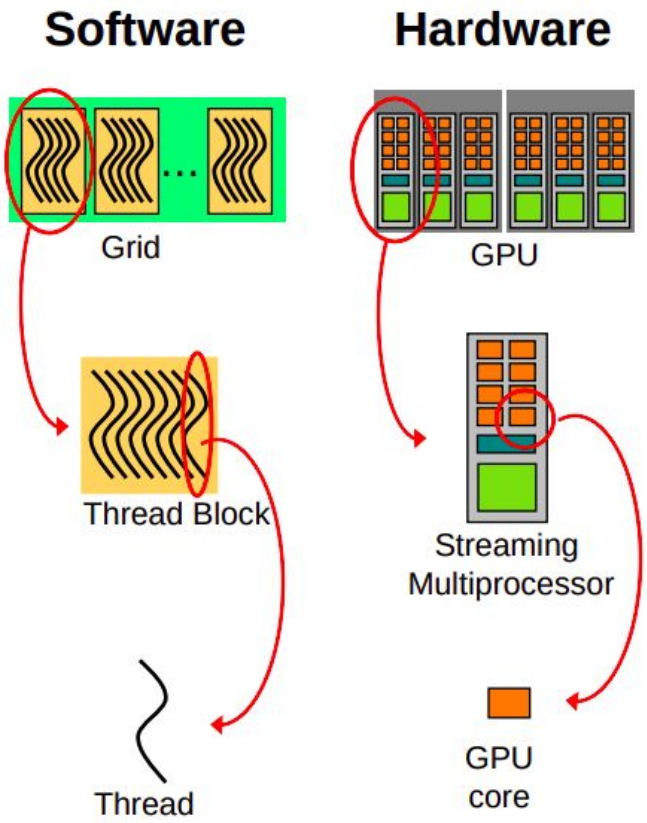
- In order to compute N elements on the GPU in parallel, at least N concurrent threads must be created on the device.
- GPU threads are grouped together in teams or blocks of threads.
- Threads belonging to the same block or team can cooperate together exchanging data through a shared memory cache area.



GPU Hardware Recap



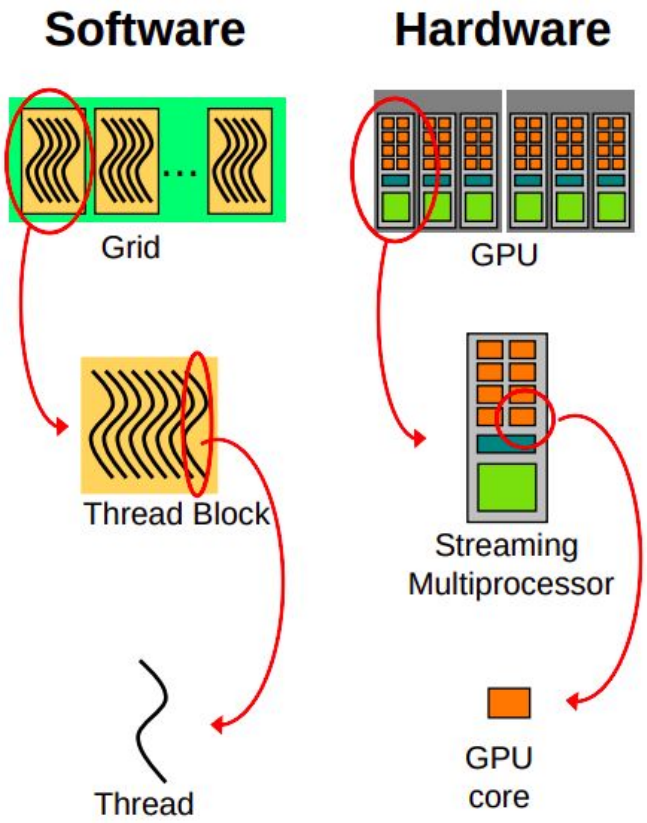
GPU Execution Model



When a GPU kernel is invoked:

- Each thread block is assigned to a SM in a round-robin mode.
- A maximum number of blocks can be assigned to each SM:
  - upper bound depending on hardware capabilities
  - upper bound on how many resources a kernel requires (registers, shared memory, etc)
- The runtime system maintains a list of active blocks and assigns new blocks to SMs as they complete.
- Once a block is assigned to a SM, it remains on that SM until the work for all threads in the block is completed.
- Each block execution is independent from the other (no synchronization is possible among them).

GPU Execution Model



- Threads of each block are partitioned into warps of consecutive threads.
- A **warp** execute one common set of instruction at a time, mapping onto the multiple core ALU of the SM units
  - each GPU core take care of one thread in the warp
  - fully efficiency when all threads agree on their execution path
- The scheduler select for execution an eligible warp from one of the residing blocks in each SM.
- A warp is eligible for execution when:
  - next instruction was fetched and all its arguments are ready
  - resources are ready: fp32 cores / fp64 cores / ldstunit / special function units

GPU Thread Hierarchy

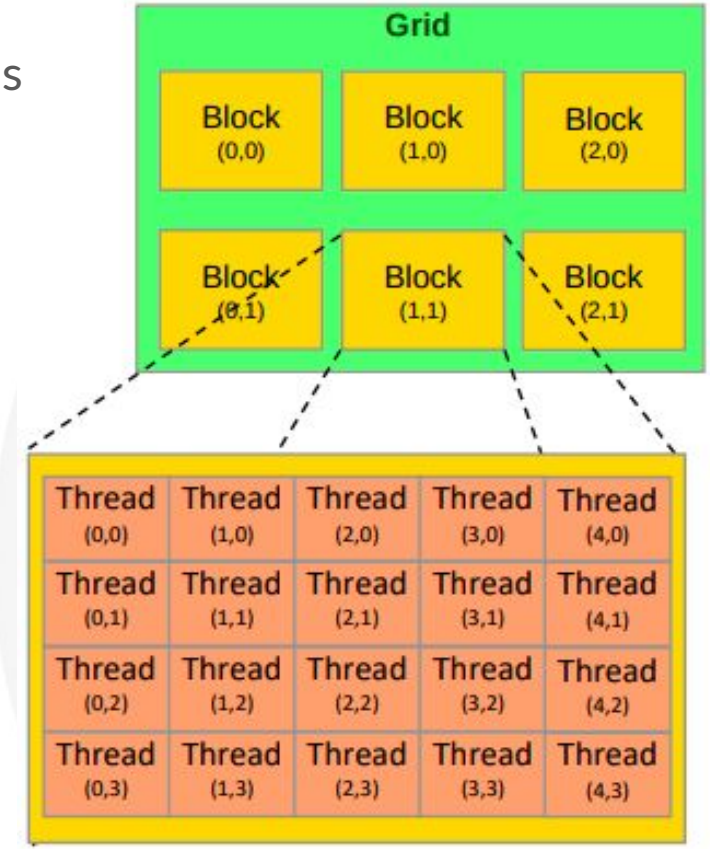
- Threads are organized into blocks of threads
  - Blocks can be 1D, 2D, 3D sized in threads
- Blocks are organized in a grid of blocks
  - Blocks can be organized into a 1D, 2D, 3D grid of blocks
- Each block or thread has a unique ID
  - Use .x, .y, .z to access its components

`threadIdx`: thread coordinates inside the block

`blockIdx`: block coordinates inside the grid

`blockDim`: block dimensions in thread units

`gridDim`: grid dimensions in block units



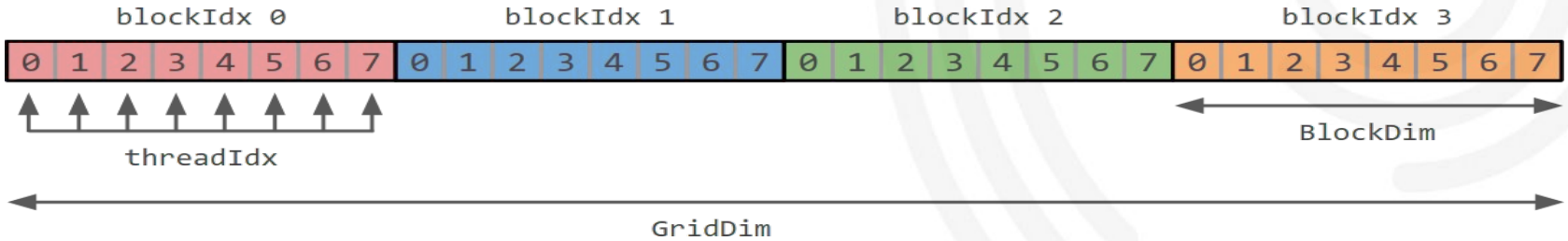
GPU Thread Hierarchy 1D Example

Each thread execute the same kernel, but acts on different data:

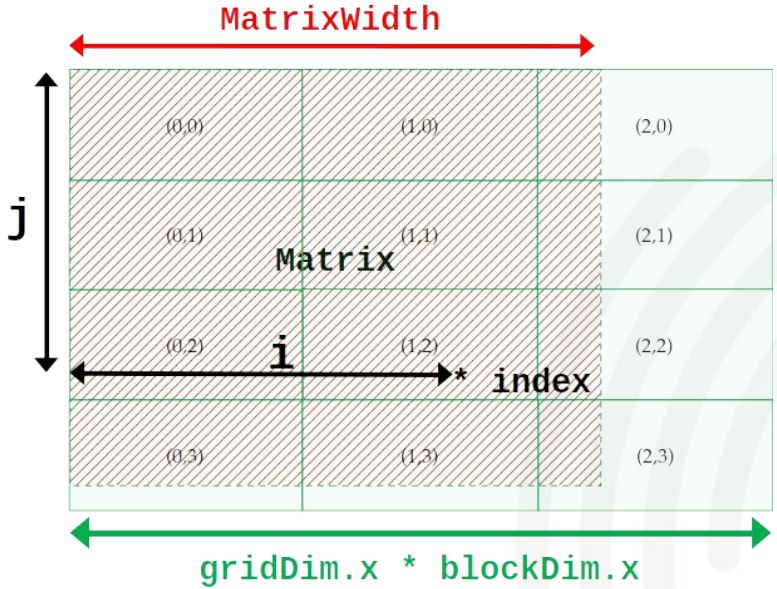
- turn the loop into a CUDA kernel function
- map each CUDA thread onto a unique index to access data
- let each thread retrieve, compute and store its own data using the unique address
- prevent out of border access to data if data is not a multiple of thread block size

```
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];
}
```

```
__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];
}
```

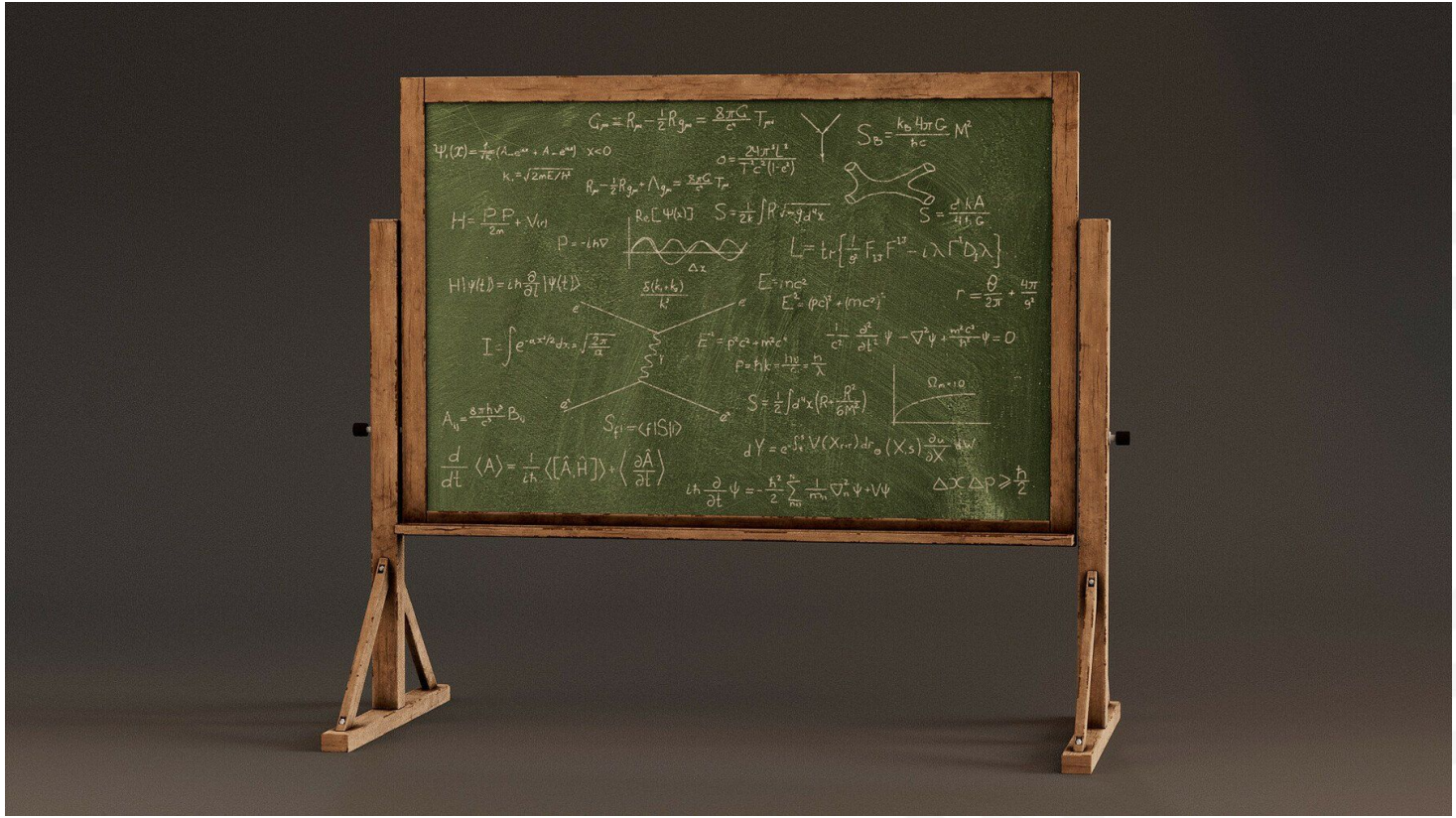


GPU Thread Hierarchy 2D Example




```
i = blockIdx.x * blockDim.x + threadIdx.x;
j = blockIdx.y * blockDim.y + threadIdx.y;

index = j * MatrixWidth + i;
```

Hands-on



Nsight-Compute Hands-on

- Kernel Copy (stride, offset)
- Jacobi
- Matrix Multiplication
- Matrix Transpose

Nsight-Compute Kernel Profiler

https://developer.nvidia.com/nsight-compute

- From **GUI**:

  ```
  ncu-ui
  ```

- From **CLI** examples:

  ```
  ncu ./EXE
  ```

  ```
  ncu --set full -o output ./EXE
  ```

NB: compile with **`-lineinfo`** option to analyze source code in the profiler

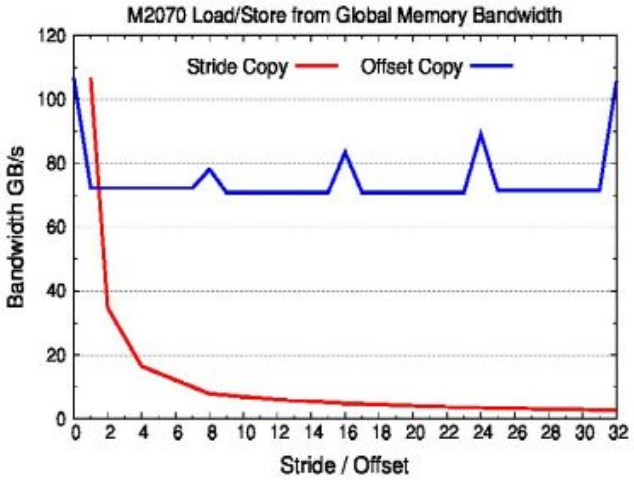
@Leonardo

1. Reserve a compute node
2. run **`ncu`**
3. copy profiler output on a local machine
4. run **`ncu-ui`** on a local machine

Stride vs Offset Kernel Copy

Strided copy	
Stride	Bandwidth GB/s
1	106.6
2	34.8
8	7.9
16	4.9
32	2.7

Offset copy	
Offset	Bandwidth GB/s
0	106.6
1	72.2
8	78.2
16	83.4
32	105.7




```
// strided copy
__global__ void strideCopy(float *odata,
                           float* idata,
                           int stride) {
   int xid = (blockIdx.x*blockDim.x + threadIdx.x) * stride;
   odata[xid] = idata[xid];
}
```

```
// offset copy
__global__ void strideCopy(float *odata,
                           float* idata,
                           int stride) {
   int xid = (blockIdx.x*blockDim.x + threadIdx.x) + offset;
   odata[xid] = idata[xid];
}
```

```
Measured on a M2070; Total elements = 16776960; Num. Blocks = 65535; Block length = 256
```

Memory Hierarchy

All CUDA threads in a block have access to:

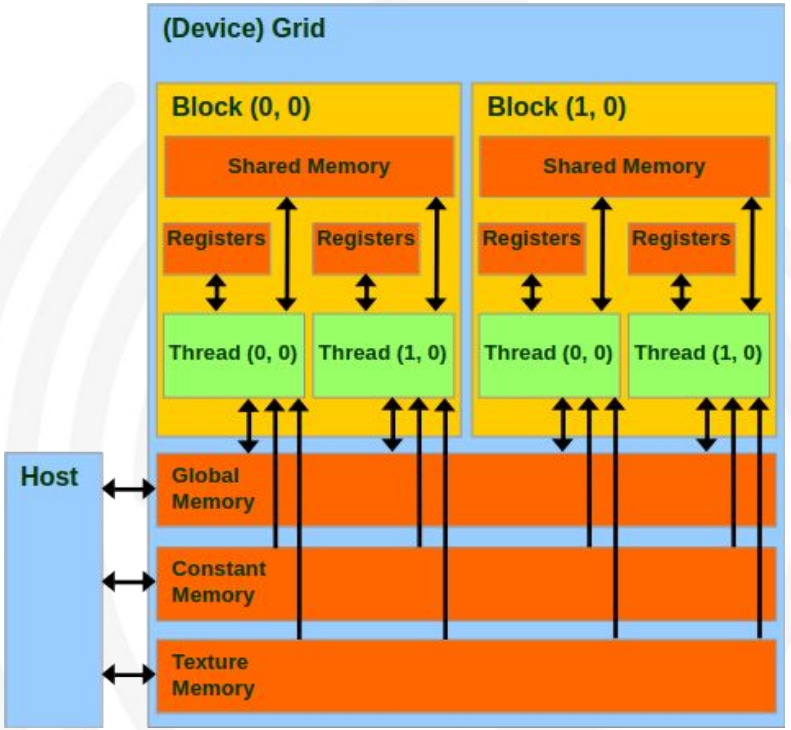
- resources of the SM assigned to its block:
  - Registers
  - Shared Memory

NB: thread belonging to different blocks cannot share these resources

- all memory type available on GPU:
  - Global Memory
  - Constant Memory (read only)
  - Texture Memory (read only)

NB: CPU can access and initialize both constant and texture memory

NB: global, constant and texture memory have persistent storage duration



Loads from Global Memory

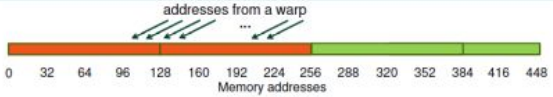
All load/store request in global memory are issued per warp (as all other instructions)

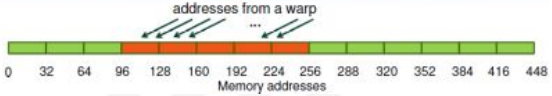
1. each thread in a warp compute the address to access
2. load/store units calculate in which memory segments data resides
3. load/store units start up requests for segment to transfer

Warp requires 32 consecutive 4-byte word aligned to segment (total 128 bytes)	
Caching Load	**Non-caching Load**
addresses fall whitin 1 cache line	addresses fall whitin 4 cache segments
128 bytes are moved across the bus	128 bytes are moved across the bus
bus utilization: **100%**	bus utilization: **100%**

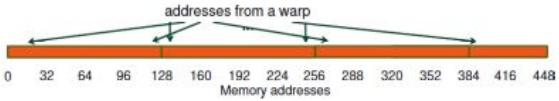
Loads from Global Memory

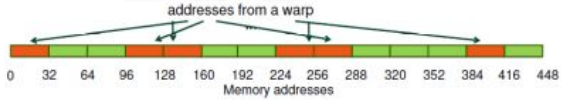
Warp requests 32 consecutive 4-bytes words not aligned to a segment (total 128 bytes)	
Caching Load	**Non-caching Load**
addresses fall within 2 cache lines	addresses fall within at most 5 segments
256 bytes are moved across the bug	256 bytes are moved across the bus
bus utilization: **50%**	bus utilization: at least **80%**





Warp requests 32 not contiguous 4-bytes words (total 128 bytes)	
Caching Load	**Non-caching Load**
addresses fall within N different cache lines	addresses fall within N different segments
N*128 bytes are moved across the bus	N*32 bytes are moved across the bus
bus utilization: **128 / (N*128)**	bus utilization: **128 / (N*32)**



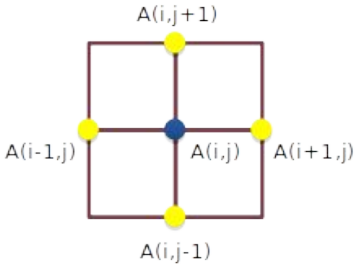


Jacobi

- Jacobi method is an iterative method to solve a system of linear equation
- It is a very common and useful algorithm
- Example: Jacobi method can be used to solve the Laplace differential equation in two variables (2D)
- The Laplace equation models the steady state of a function f defined in a physical 2D space where f is a physical quantity (e.g. Temperature)

$$\nabla^2 f(x, y) = 0$$

- Iteratively converges to correct value by computing new values at each point from the average of neighboring points

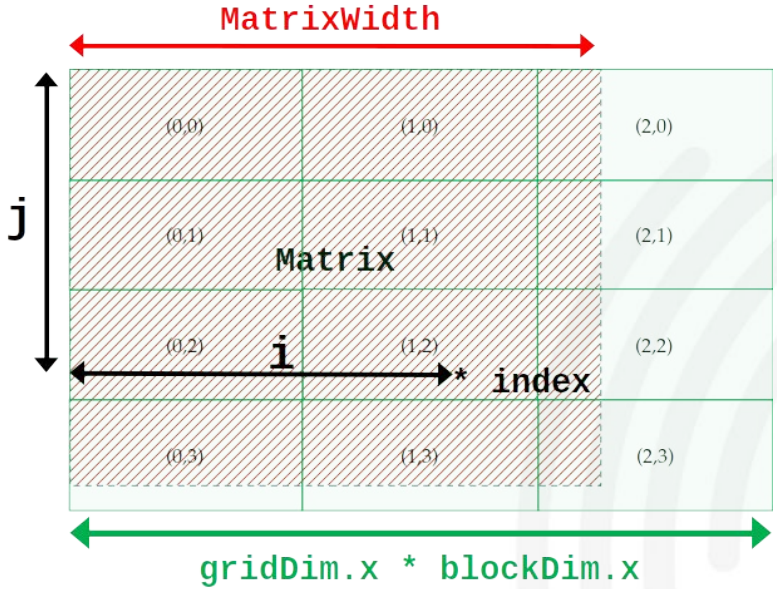


$$A_{k+1}(i,j) = \frac{A_k(i-1,j) + A_k(i+1,j) + A_k(i,j-1) + A_k(i,j+1)}{4}$$

Jacobi

```
__global__ void jacobi(float * curr, float * next, float * errs, int n, int stride)
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int j = threadIdx.y + blockIdx.y * blockDim.y;

    int idx = i + j * stride;

    if ( (i > 0) && (i < n-1) && (j > 0) && (j < n-1) ) {

        next[idx] = 0.25 * (
                curr[(i-1) + j * stride] +
                curr[(i+1) + j * stride] +
                curr[i + (j-1) * stride] +
                curr[i + (j+1) * stride]
                ) ;

        errs[idx] = fabs(next[idx] - curr[idx]);

    }
}
```

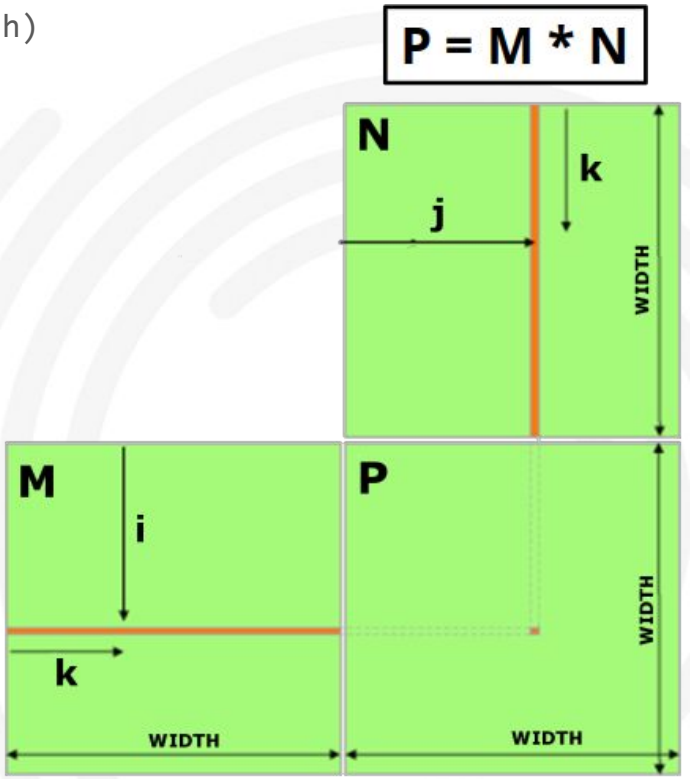
GPU Thread Hierarchy 2D Example




```
i = blockIdx.x * blockDim.x + threadIdx.x;
j = blockIdx.y * blockDim.y + threadIdx.y;

index = j * MatrixWidth + i;
```

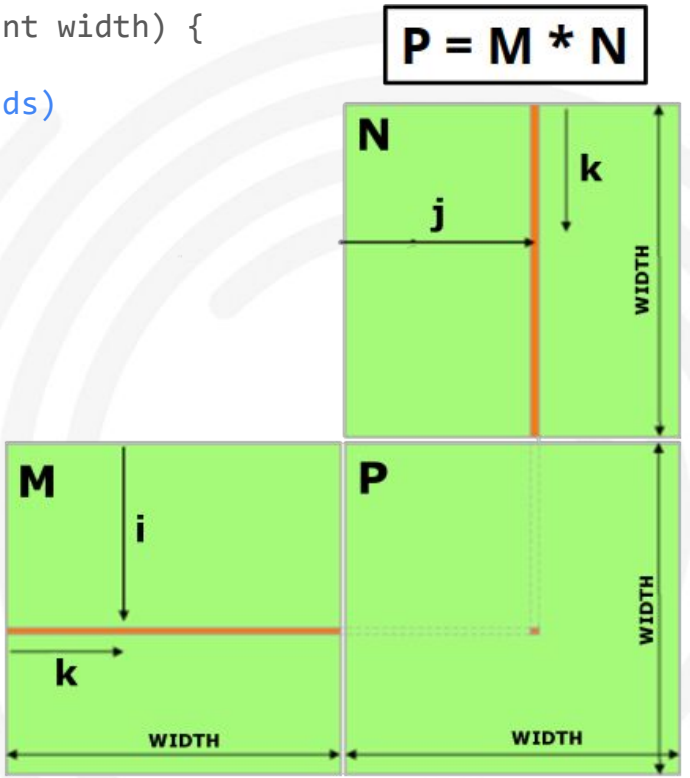
Matrix Multiplication in Global Memory

```
void MatrixMulOnHost (float* M, float* N, float* P, int Width)
{
  // loop on rows
  for (int row = 0; row < Width; ++row) {
    // loop on columns
    for (int col = 0; col < Width; ++col) {

      // accumulate element-wise products
      float pval = 0;
      for (int k = 0; k < Width; ++k) {
        float a = M[row * Width + k];
        float b = N[k * Width + col];
        pval += a * b;
      }

      // store final results
      P[row * Width + col] = pval;
    }
  }
}
```



Matrix Multiplication in Global Memory

```
__global__ void MMKernel (float* dM, float *dN, float *dP, int width) {

  // row,col from built-in thread indices (2D block of threads)
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  int row = blockIdx.y * blockDim.y + threadIdx.y;

  // check if current CUDA thread is inside matrix borders
  if (row < width && col < width) {

    // accumulate element-wise products
    // NB: pval stores the dP element computed by the thread
    float pval = 0;
    for (int k=0; k < width; k++)
       pval += dM[row * width + k] * dN[k * width + col];

    // store final results (each thread writes one element)
    dP[row * width + col] = Pvalue;
  }
}
```



Matrix Multiplication in Global Memory

Which is the best thread block size to select (i.e. **TILE_WIDTH**)?

On **Fermi** architectures: each SM can handle up to ***1536*** total threads

- **TILE_WIDTH** = **8**

  8x8 = 64 threads >>> 1536/64 = 24 blocks needed to fully load a SM

  … yet there is a limit of maximum 8 resident blocks per SM for cc 2.x

  so we end up with just 64x8 = 512 threads per SM on a maximum of 1536 (only **33%** occupancy)

- **TILE_WIDTH** = **16**

  16x16 = 256 threads >>> 1536/256 = 6 blocks to fully load a SM

  6x256 = 1536 threads per SM … reaching **full occupancy** per SM!

- **TILE_WIDTH** = **32**

  32x32 = 1024 threads >>> 1536/1024 = 1.5 = 1 block fully loads SM

  1024 threads per SM (only **66%** occupancy)

TILE_WIDTH = 16

Matrix Multiplication in Global Memory

Which is the best thread block size to select (i.e. **TILE_WIDTH**)?

On **Kepler** architectures: each SM can handle up to ***2048*** total threads

- **TILE_WIDTH** = 8

  8x8 = 64 threads >>> 2048/64 = 32 blocks needed to fully load a SM

  … yet there is a limit of maximum 16 resident blocks per SM for cc 3.x

  so we end up with just 64x16 = 1024 threads per SM on a maximum of 2048 (only **50%** occupancy)

- **TILE_WIDTH** = 16

  16x16 = 256 threads >>> 2048/256 = 8 blocks to fully load a SM

  8x256 = 2048 threads per SM … reaching **full occupancy** per SM!

- **TILE_WIDTH** = 32

  32x32 = 1024 threads >>> 2048/1024 = 2 blocks fully loads SM

  2x1024 = 2048 threads per SM … reaching **full occupancy** per SM

TILE_WIDTH = 16 or 32

Matrix Multiplication in Global Memory

Each thread compute one element of C, using 2N elements and performing 2N floating-point operations (N add , N mul)

NB: every element of C shares same row or column retrieved N times the same elements from A or B

This implementation results in $2N^3$ loads !!!

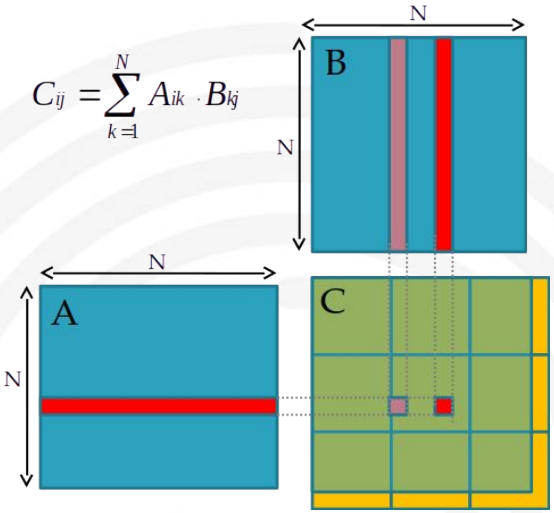
We can avoid requesting the same elements many times, sharing them through the shared memory

each thread can retrieve just one data element data in parallel and store it into shared memory

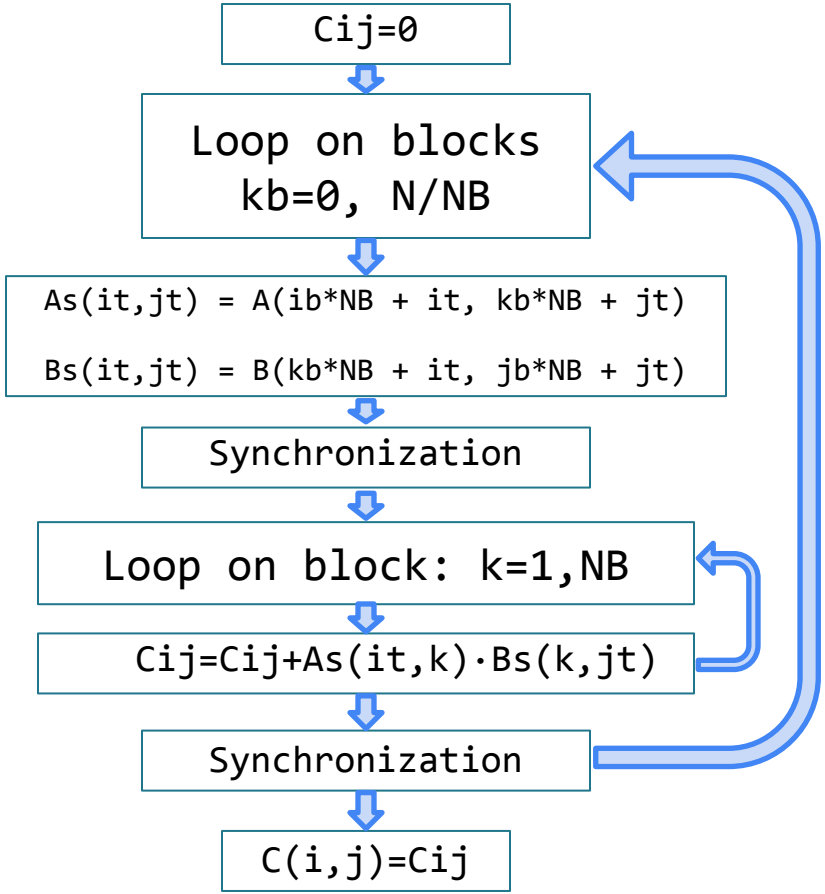
when all threads have loaded needed data, they can access all the elements by the threads belonging to the same block, for example sharing a full row or column

Unfortunately shared memory size is small

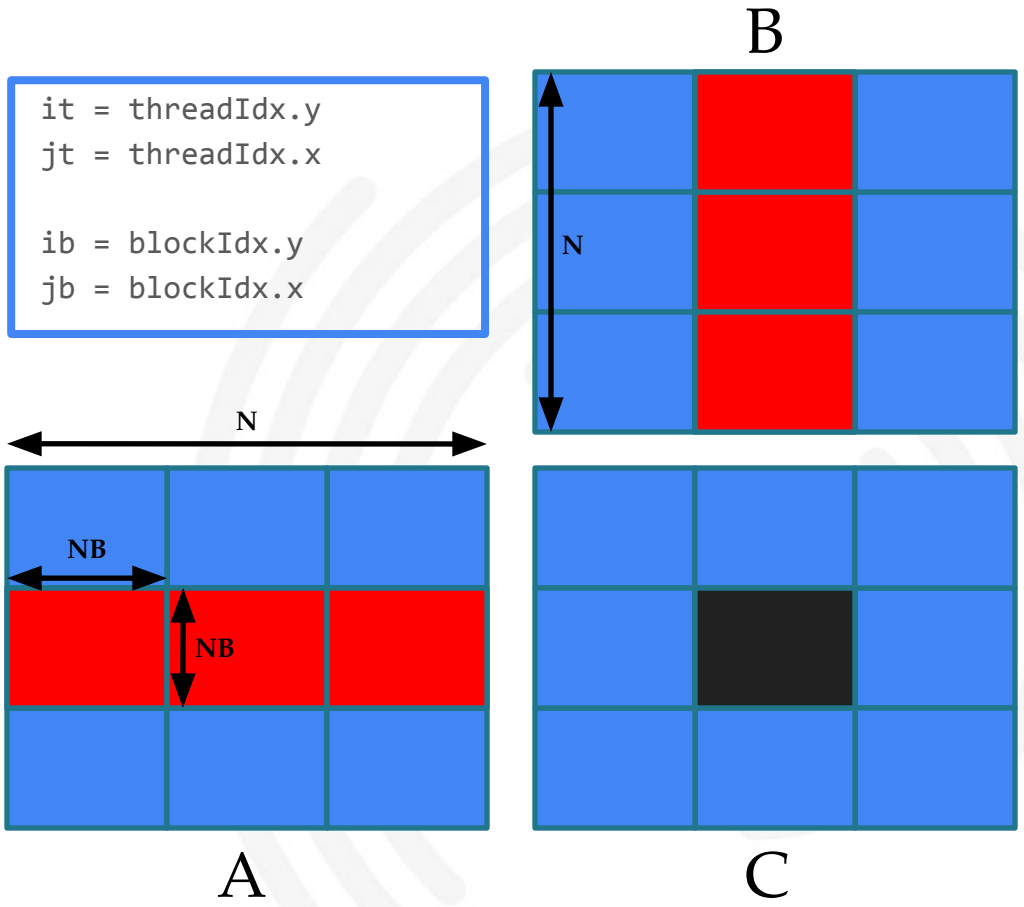
16/48 KB depending on the compute capability



Matrix Multiplication in Shared Memory




```
it = threadIdx.y
jt = threadIdx.x

ib = blockIdx.y
jb = blockIdx.x
```



Matrix Transpose

Matrix transpose with the following simple design:

- out-of-place buffers
- square matrices with size modulo 32 elements

This is a memory-bounded kernel

- no computation on elements
- just load and stores

We will use effective bandwidth (GB/s) as a metric to measure the performance of such kernels

Simple Matrix Copy

```
__global__ void copy(float *idata, float *odata, int width, int height)
{
  int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;

  int index  = width * yIndex + xIndex;

  odata[index] = idata[index];
}
```

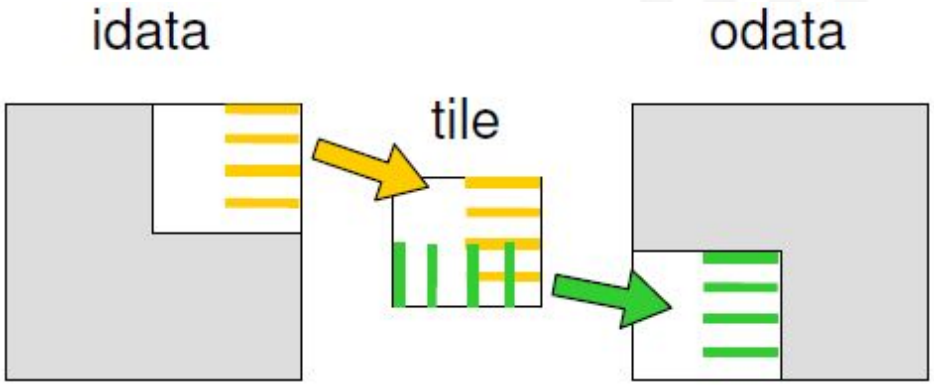
Matrix Transpose in Global Memory

```
__global__ void transpose(float *idata, float *odata, int width, int height)
{
  int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;

  int index_in  = width  * yIndex + xIndex;
  int index_out = height * xIndex + yIndex;

  odata[index_out] = idata[index_in];
}
```

Matrix Transpose in Shared Memory

To avoid non-coalesced store we should store by row:

- fill a tile in shared memory with data to be transposed
- we don't get any penalty writing elements by columns into shared-memory
- the transpose operation is now performed in shared-memory
- then write back info global memory by rows



Matrix Transpose in Shared Memory

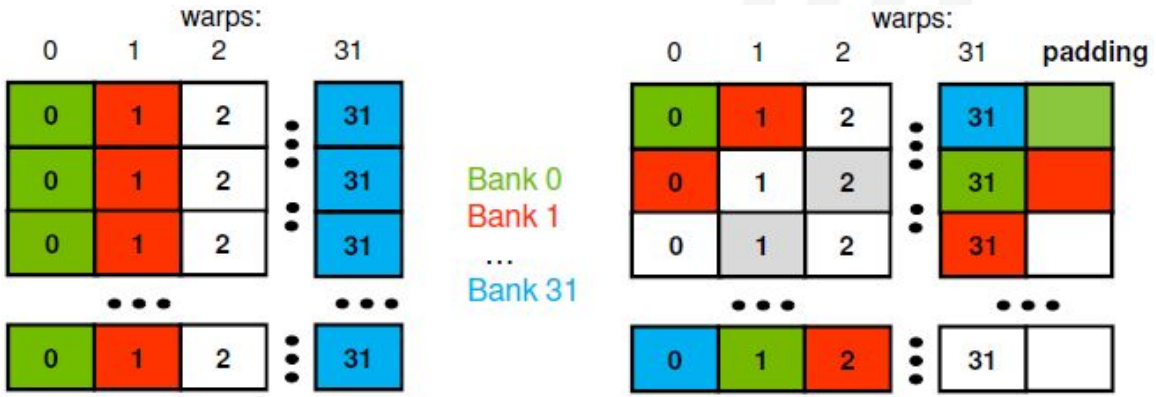
```
__global__ void transpose (float *idata, float *odata, int width, int height)
{
  __shared__ float tile[TILE_DIM][TILE_DIM];

  int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;

  int index_in  = width * yIndex + xIndex;

  xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
  yIndex = blockIdx.x * TILE_DIM + threadIdx.y;

  int index_out = height * yIndex + xIndex;

  tile[threadIdx.y][threadIdx.x] = idata[index_in];

  __syncthreads();

  odata[index_out] = tile[threadIdx.x][threadIdx.y];
}
```

Matrix Transpose in Shared Memory

Matrix Transpose in shared memory use a tile of 32x32 float

- each element resides on successive bank (4-byte)
- accessing elements with a 32 size stride will fetch them from the same bank
- any read/write access to this tile by column will get a 32 bank conflict
- use the trick! a new tile of 32x33 elements
- element of the same tile column will resides on different banks
- no more bank conflicts at all

```
__shared__ float tile[TILE_DIM][TILE_DIM+1];
```



Matrix Transpose in Shared Memory