Introduction to CUDA programming models

Concurrency

September 26-30, 2022

University of Perugia

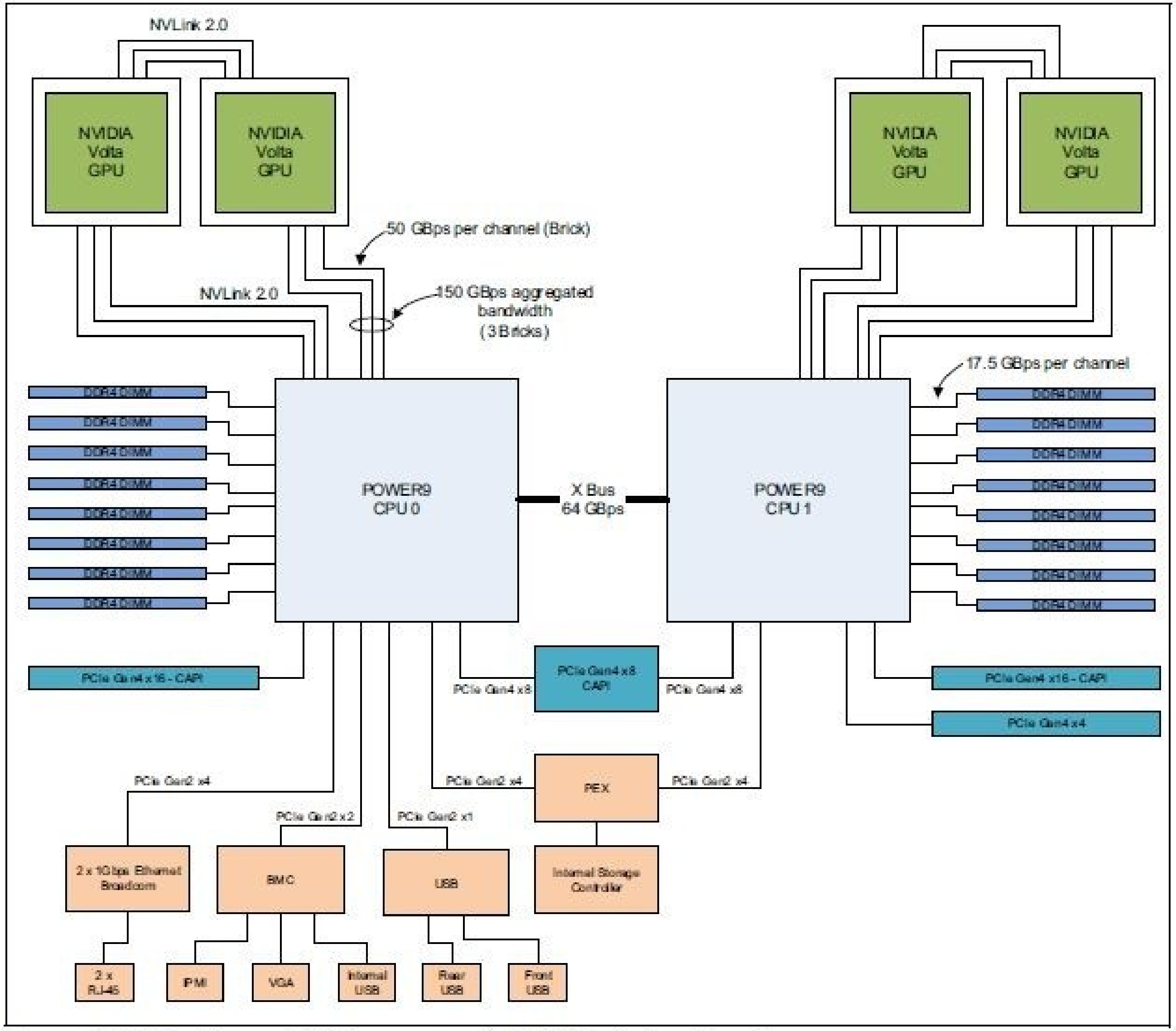
Sergio Orlandini

CINECA

s.orlandini@cineca.it

- Synchronous and Asynchronous API
- Concurrent Execution
- CPU and GPU interaction
    - concurrent execution on CPU and GPU
    - overlapping transfers and kernels
- Multi-device management
- GPU/GPU interactions

Connection Scheme of *host/device*



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*

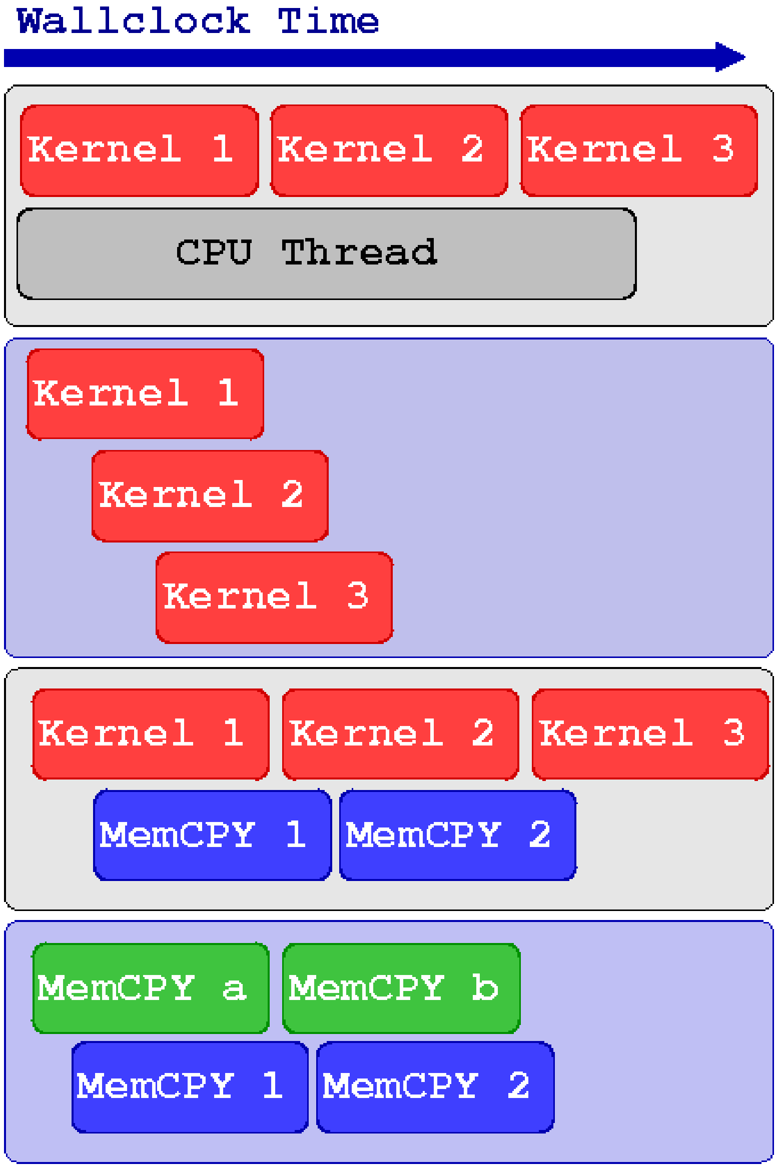
- **Non-blocking** (asynchronous): return control to *host* immediately, while its execution proceeds on *device*
  - kernel launches
  - memory transfers < 64KB
  - memory initialization on *device* (cudaMemset)
  - memory copies from *device* to *device*
  - explicit asynchronous memory transfers

- 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*

Concurrent and Asynchronous Execution

Asynchronous functions allows to expose concurrent executions:

1. Overlap computation on *host* and on *device*
2. execution of more than one kernel on *device*
3. data transfers between *host* and *device* while executing a kernel
4. data transfers from *host* to *device*, while transfering data from *device* to *host*



Example of Concurrent Execution

```
kernel <<<threads, Blocks>>> (a, b, c)

// work on CPU while GPU is working
CPU_Function()

// Stop CPU until GPU has finished to compute
cudaDeviceSynchronize()

// Use device results on host
CPU_uses_the_GPU_kernel_results()
```

Since CUDA kernel invocation is an asynchronous operation, CPU can proceed and evalutate the `CPU_Function()`while GPU is involved in kernel execution (*concurrent execution*).

Before using results from CUDA kernel, synchronization between *host* and *device* is required.

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

CUDA Streams Management

- Stream management:
  - Constructor: `cudaStreamCreate()`
  - Synchronization: `cudaStreamSynchronize()`
  - Destructor: `cudaStreamDestroy()`
- Stream allows various execution modes, depending on the compute capability:
  - concurrent execution of more than one kernel per GPU
  - concurrent asynchronous data transfers in both H2D and D2H directions
  - concurrent execution on device/host and data transfers from host and device

Kernel Concurrent Execution

```
cudaStreamCreate(stream1)
cudaStreamCreate(stream2)

// concurrent execution of the same kernel
Kernel_1<<<blocks, threads, SharedMem, stream1>>>(inp_1, out_1)
Kernel_1<<<blocks, threads, SharedMem, stream2>>>(inp_2, out_2)

// concurrent execution of different kernels
Kernel_1<<<blocks, threads, SharedMem, stream1>>>(inp, out_1)
Kernel_2<<<blocks, threads, SharedMem, stream2>>>(inp, out_2)

cudaStreamDestroy(stream1)
cudaStreamDestroy(stream2)
```

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 performes 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

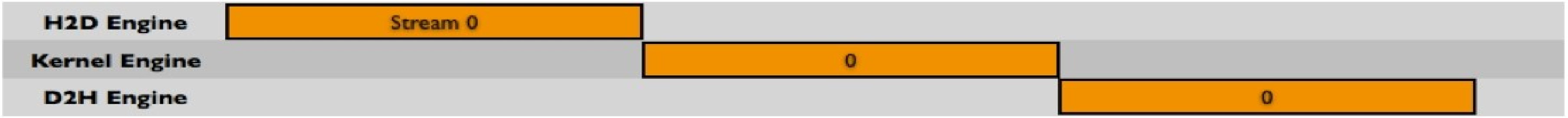
Asynchronous Data Transfers

```
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)
```

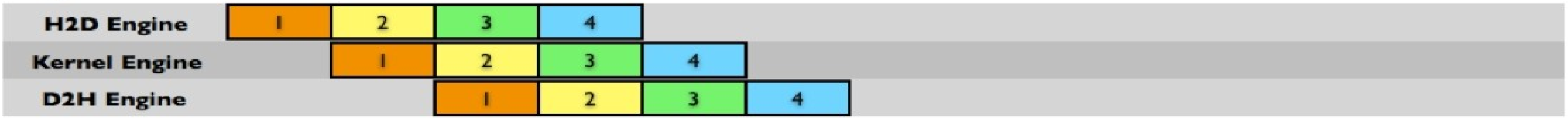
Asynchronous Data Transfers

```
cudaStream_t stream[4];
for (int i=0; i<4; ++i) cudaStreamCreate(&stream[i]);

float* hPtr; cudaMallocHost((void**)&hPtr, 4 * size);

for (int i=0; i<4; ++i) {
  cudaMemcpyAsync(d_inp + i*size, hPtr + i*size,
                  size, cudaMemcpyHostToDevice, stream[i]);

  MyKernel<<<100, 512, 0, stream[i]>>>(d_out+i*size, d_inp+i*size, size);

  cudaMemcpyAsync(hPtr + i*size, d_out + i*size,
                  size, cudaMemcpyDeviceToHost, stream[i]);
}
cudaDeviceSynchronize();

for (int i=0; i<4; ++i) cudaStreamDestroy(&stream[i]);
```

Sequential Version



Asynchronous Versions

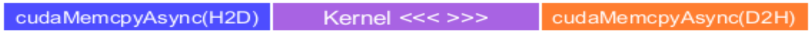


Concurrency

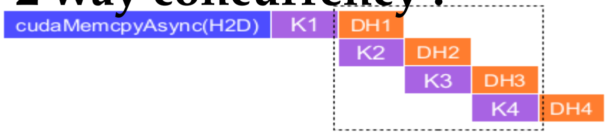
- Concurrency: when two or more CUDA operations proceed at the same time
  - **Fermi** : up to 16 kernel CUDA / **Kepler** : up to 32 kernel CUDA
  - 2 data transfers host/device (duplex)
  - concurrency with host operations

- Requirements for concurrency:
  - operations must be assigned to streams different from the default stream
  - host/device data transfers should be asynchronous and host memory must be page-locked
  - only if there are enough hw resources left to use (SharedMem, Registers, Blocks, PCIe bus, ...)
    - ❑ No kernel concurrency if all SM on the device are in use
    - ❑ data transfers won't take place if other transfers are still going on

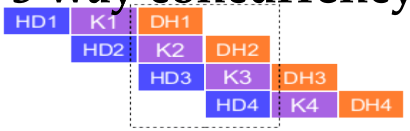
Serial :



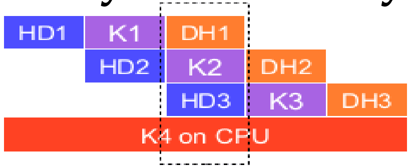
2 way concurrency :



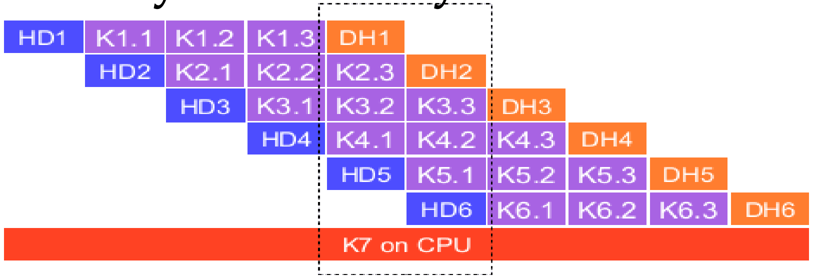
3 way concurrency :



4 way concurrency :



4/+ way concurrency :



Device Management

CUDA runtime allows to control more than one GPU device available on a computing node (multi-GPU programming):

- CUDA 3.2 and previous versions
  - a multi-thread or multi-process parallel paradigm was required to access and use more than one device
- CUDA 4.0 and later versions
  - new runtime API to select and to control all available devices from a serial program (single host core)
  - you can still use a parallel programming approach (multi-thread or multi-process):
    - each process or thread will be always able to access all devices
    - you can select which devices a thread/process can control

Device Management

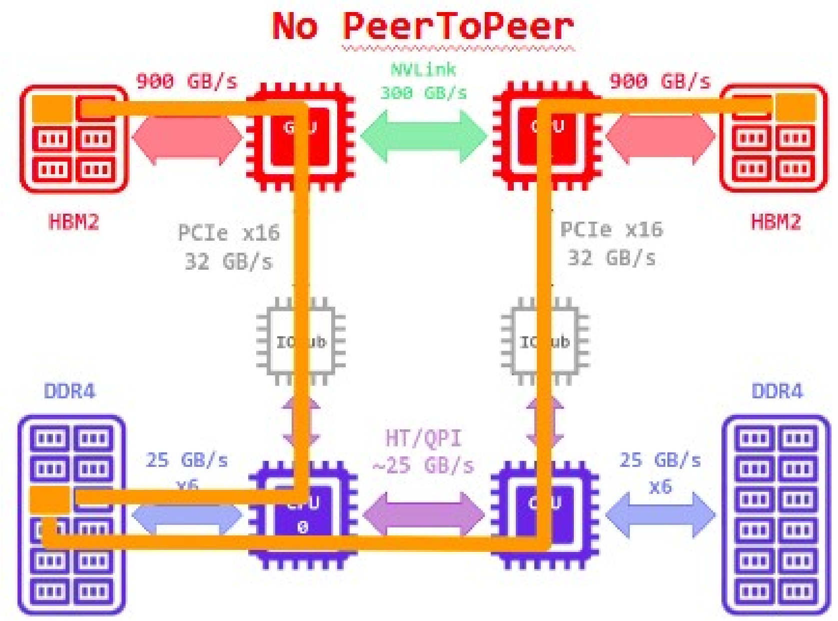
```
cudaDeviceCount(number_gpu)
cudaGetDeviceProperties(gpu_property, gpu_ID)

cudaSetDevice(0)
kernel_0 <<<threads, Blocks>>> (a, b, c)

cudaSetDevice(1)
kernel_1 <<<threads, Blocks>>> (d, e, f)

For each device:
  cudaSetDevice(device)
  cudaDeviceSynchronize()
```

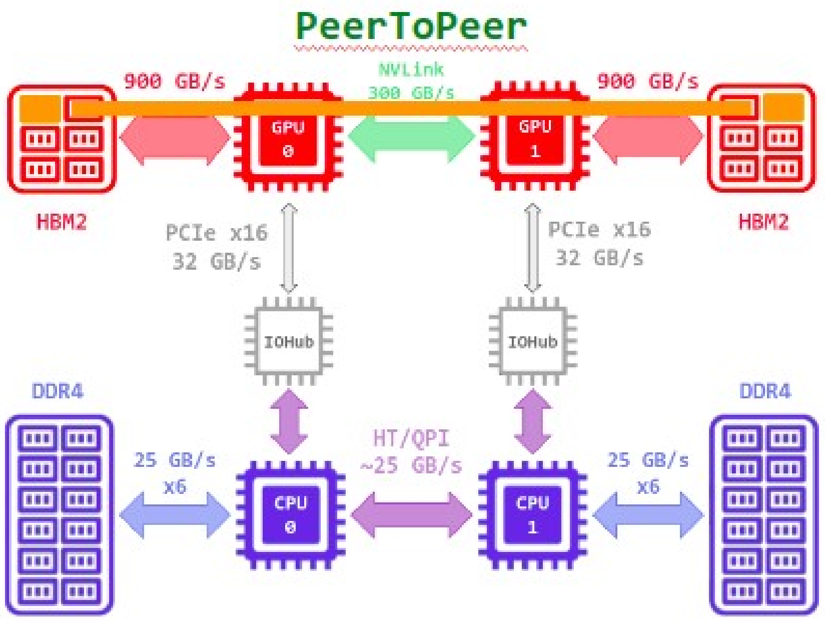
CUDA runtime allows to:

- get information on available CUDA enabled devices
- get properties of each CUDA device (cc, memory sizes, clock, etc)
- select a device and queue CUDA operations on that device
- manage synchronization among available devices

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

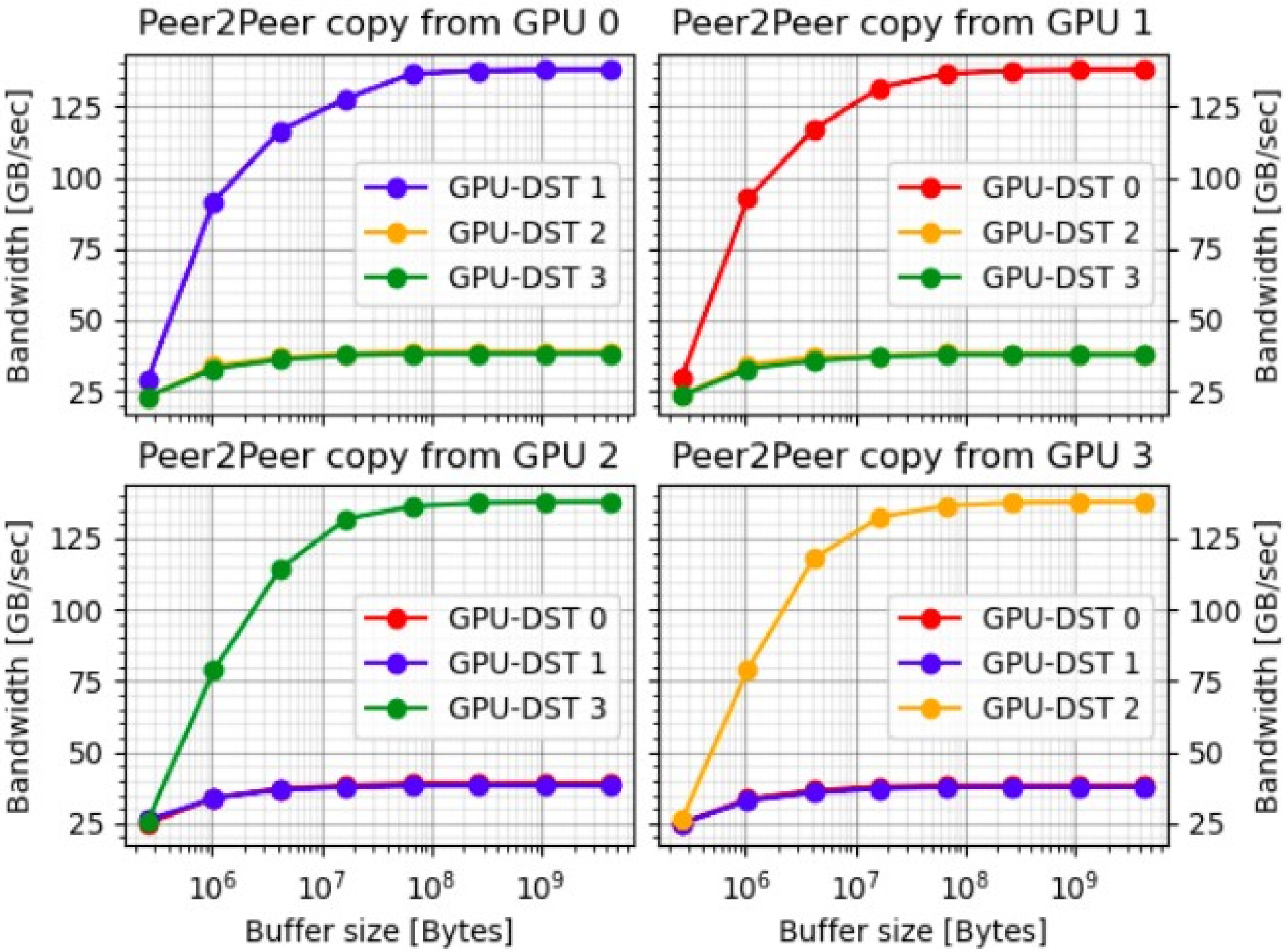
```
gpuA=0, gpuB=1
cudaSetDevice(gpuA)
cudaMalloc(buffer_A, buffer_size)

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)
```

Peer to Peer Direct Access 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 invokes a kernel that accesses to gpuB memory
 kernel<<<threads, blocks>>>(buffer_A, buffer_B)
```

GPU-GPU Peer to Peer Transfers

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.

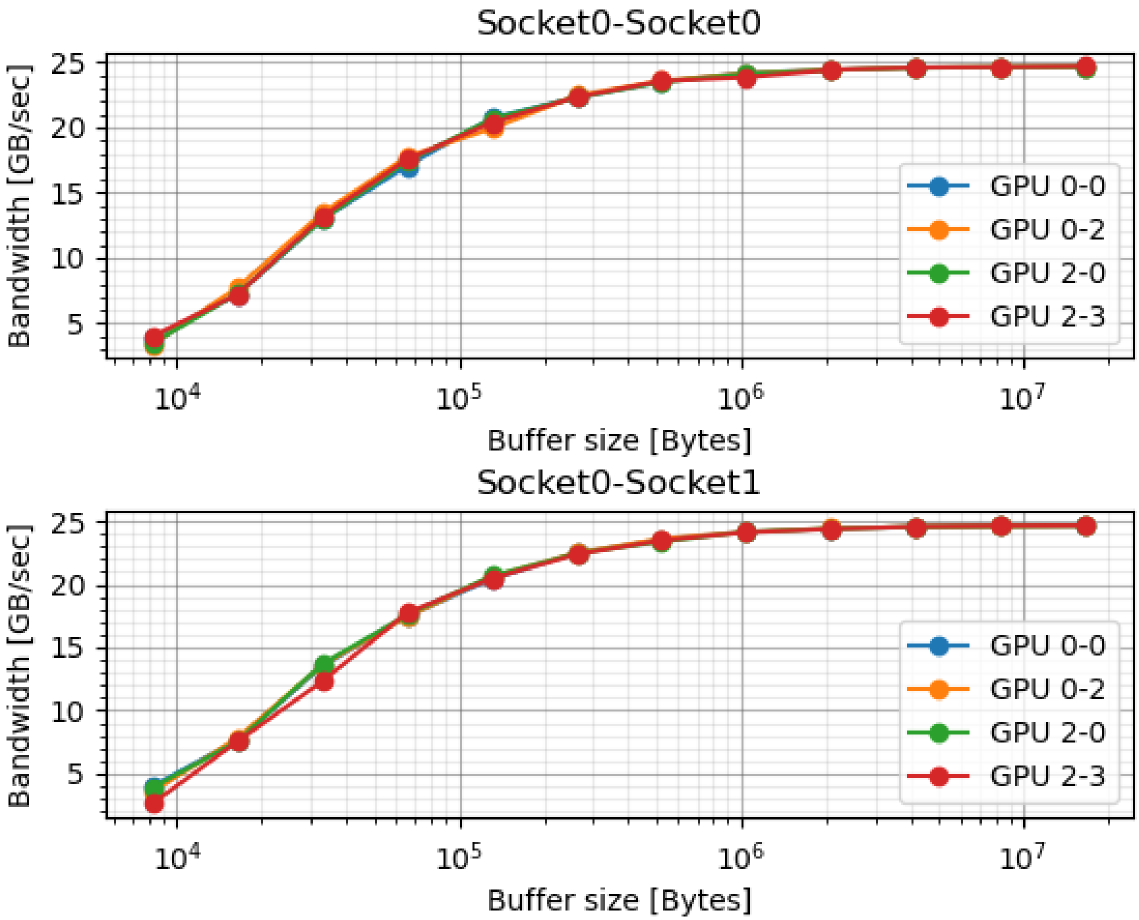
MPI-CUDA-AWARE

```
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_Isend(buffer_send, buffer_elem, MPI_INT, 1, 42, MPI_COMM_WORLD, &request)

  MPI_Waitall(1, request, status)
  MPI_Recv(buffer_recv, 4, MPI_INT, 1, 43, MPI_COMM_WORLD, &status)

} else if(rank == 1) {

  MPI_Irecv(buffer_recv, buffer_elem, MPI_INT, 0, 42, MPI_COMM_WORLD, &request)

  MPI_Waitall(1, request, status)
  MPI_Send(buffer_send, 4, MPI_INT, 0, 43, MPI_COMM_WORLD)

}

cudaFree(buffer_send)
cudaFree(buffer_recv)

MPI_Finalize()
```

MPI-CUDA-AWARE

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

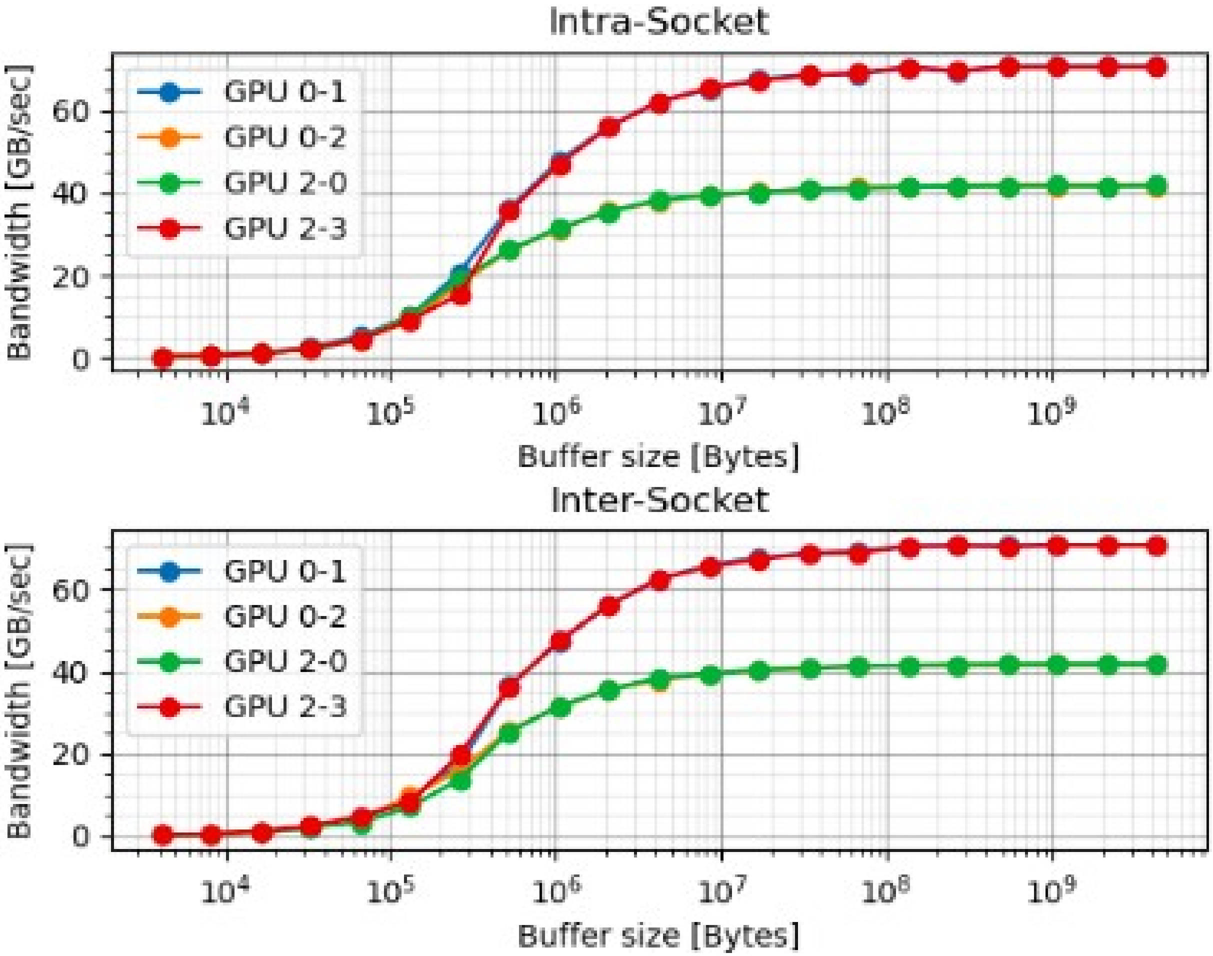
@**Marconi100** with 4 V100



MPI-CUDA-AWARE

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

@**Marconi100** with 4 V100



Rights & Credits

Slides and examples were authored by:

Isabella Baccarelli, Luca Ferraro, Sergio Orlandini