

GPU programming



27-th Summer School on Parallel Computing

Rome edition

9 - 20 July 2018

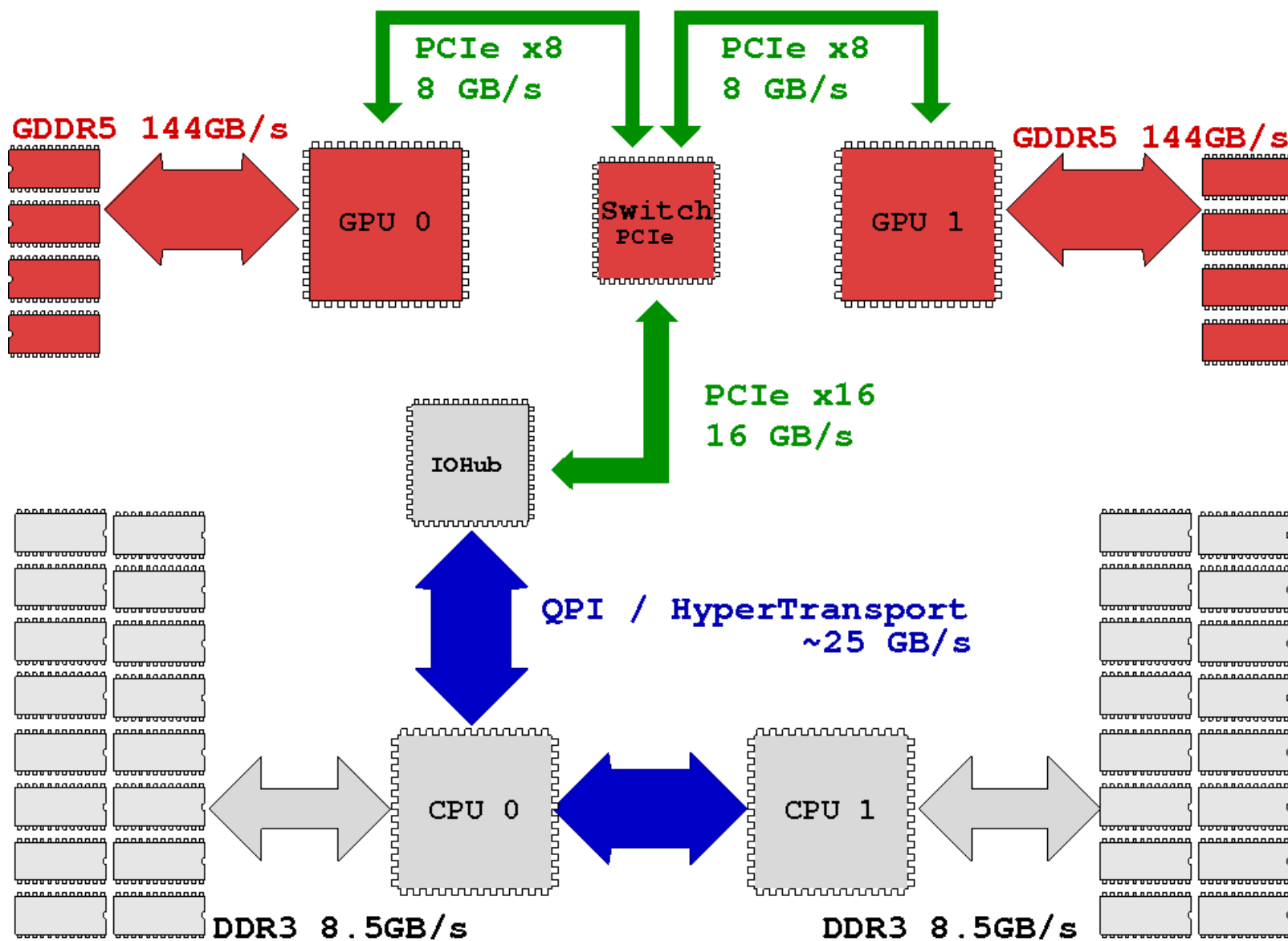
Sergio Orlandini

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*



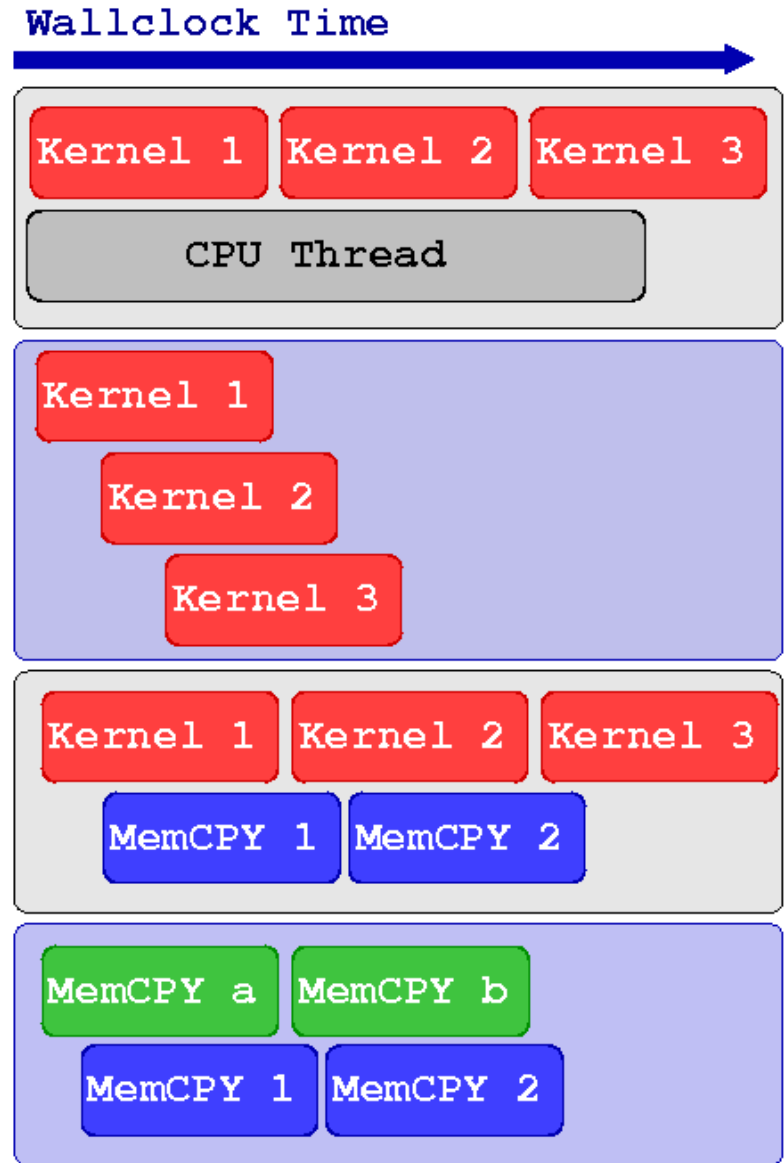
Blocking and Non-blocking Functions

- 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 $> 64\text{KB}$
 - 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 $< 64\text{KB}$
 - 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 transferring data from *device* to *host*



Example of Concurrent Execution

```
cudaSetDevice(0)
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 evaluate the `CPU_Function()` while GPU is involved in kernel execution (*concurrent execution*).

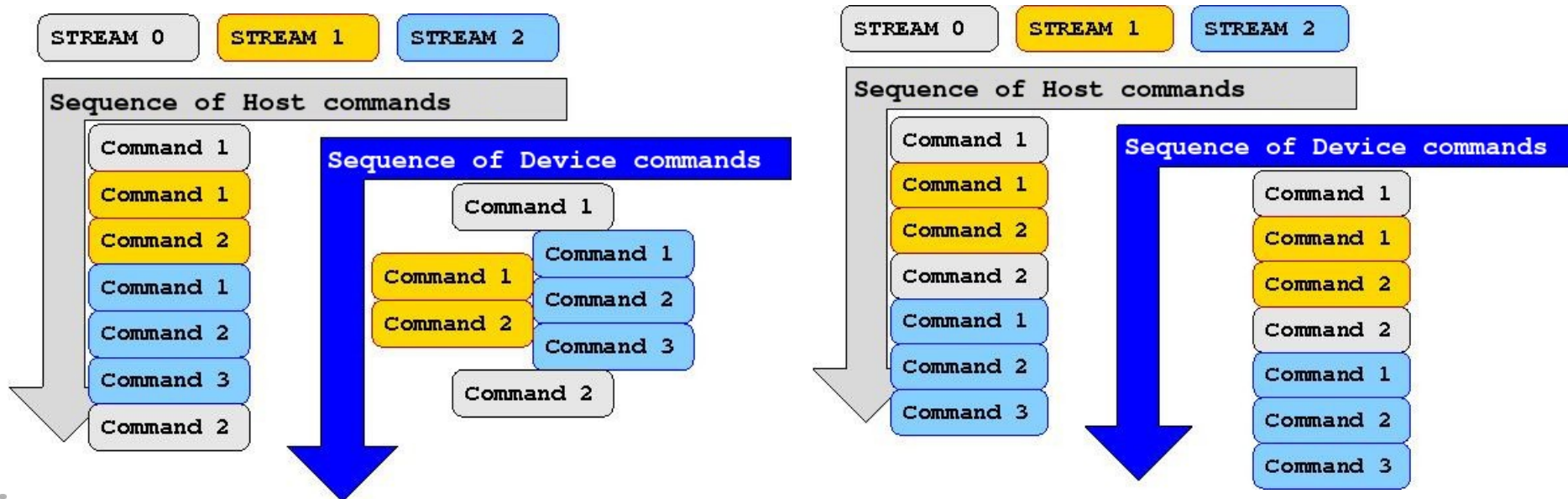
Before using results from CUDA kernel, synchronization between *host* and *device* is required.

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

- All operations assigned to the default stream will be executed only after all preceeding operations assigned to other streams are completed
- Any further operation assigned to stream different from default will begin only after all operations on the default stream are completed
- Operations assigned to the default stream act as implicit synchronization barriers among other streams



Synchronization

▪ **Explicit Synchronizations** :

- `cudaDeviceSynchronize()`
 - Blocks host code until all operations on device are completed
- `cudaStreamSynchronize(stream)`
 - Blocks host code until all operations on a stream are completed
- `cudaStreamWaitEvent(stream, event)`
 - Blocks all operations assigned to a stream until event is reached

▪ **Implicit Synchronizations** :

- All operations assigned to the default stream
- Page-locked memory allocations
- Memory allocations on device
- Settings operations on device
- ...

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

```
cudaSetDevice(0)

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 performs asynchronous data transfers between *host* and *device* memory
- Data transfer operations must be 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



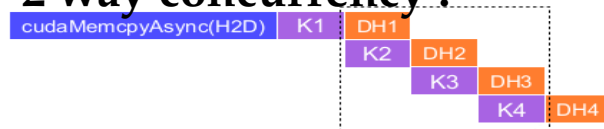
Concurrency

- 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

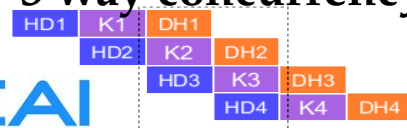
Serial :



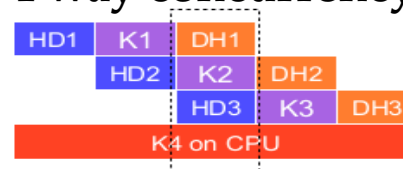
2 way concurrency :



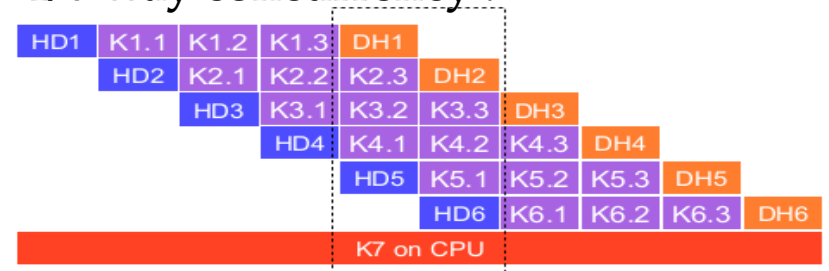
3 way concurrency :



4 way concurrency :



4/+ way concurrency :



Stream Priorities

- Relative priorities of streams can be specified at creation
- If not specified, all streams get the same priority
- runtime will choose which operation start first among equivalent priority streams

```
// get the range of stream priorities for this device
int priority_high, priority_low;
cudaDeviceGetStreamPriorityRange(&priority_low, &priority_high);

// create streams with highest and lowest priorities
cudaStream_t st_high, st_low;
cudaStreamCreateWithPriority(&st_high, cudaStreamNonBlocking,
priority_high);
cudaStreamCreateWithPriority(&st_low, cudaStreamNonBlocking,
priority_low);
```


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)
```

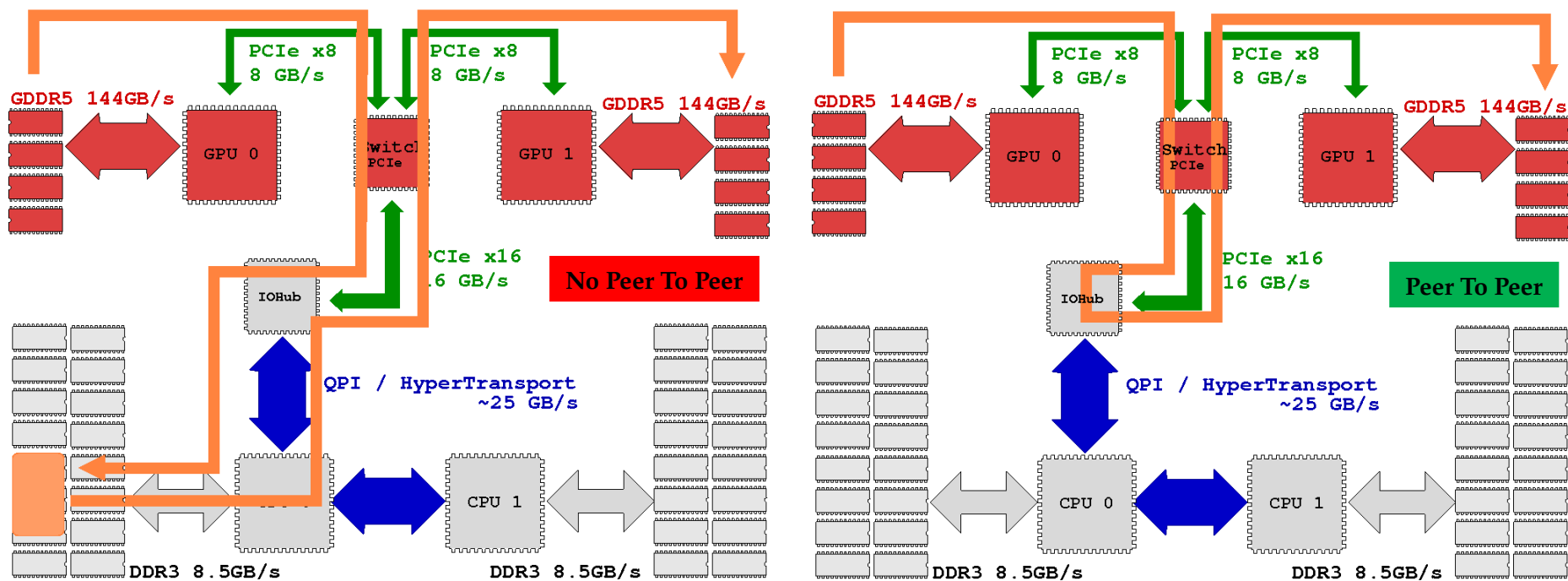
```
For each device:
    cudaSetDevice(device)
    cudaDeviceSynchronize()
```

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)
```

Rights & Credits

These slides are CINECA 2014 and are released under the Attribution-NonCommercial-NoDerivs (CC BY-NC-ND) Creative Commons license, version 3.0.

Uses not allowed by the above license need explicit, written permission from the copyright owner. For more information see:

<http://creativecommons.org/licenses/by-nc-nd/3.0/>

Slides and examples were authored by:

Isabella Baccarelli, Luca Ferraro, Sergio Orlandini