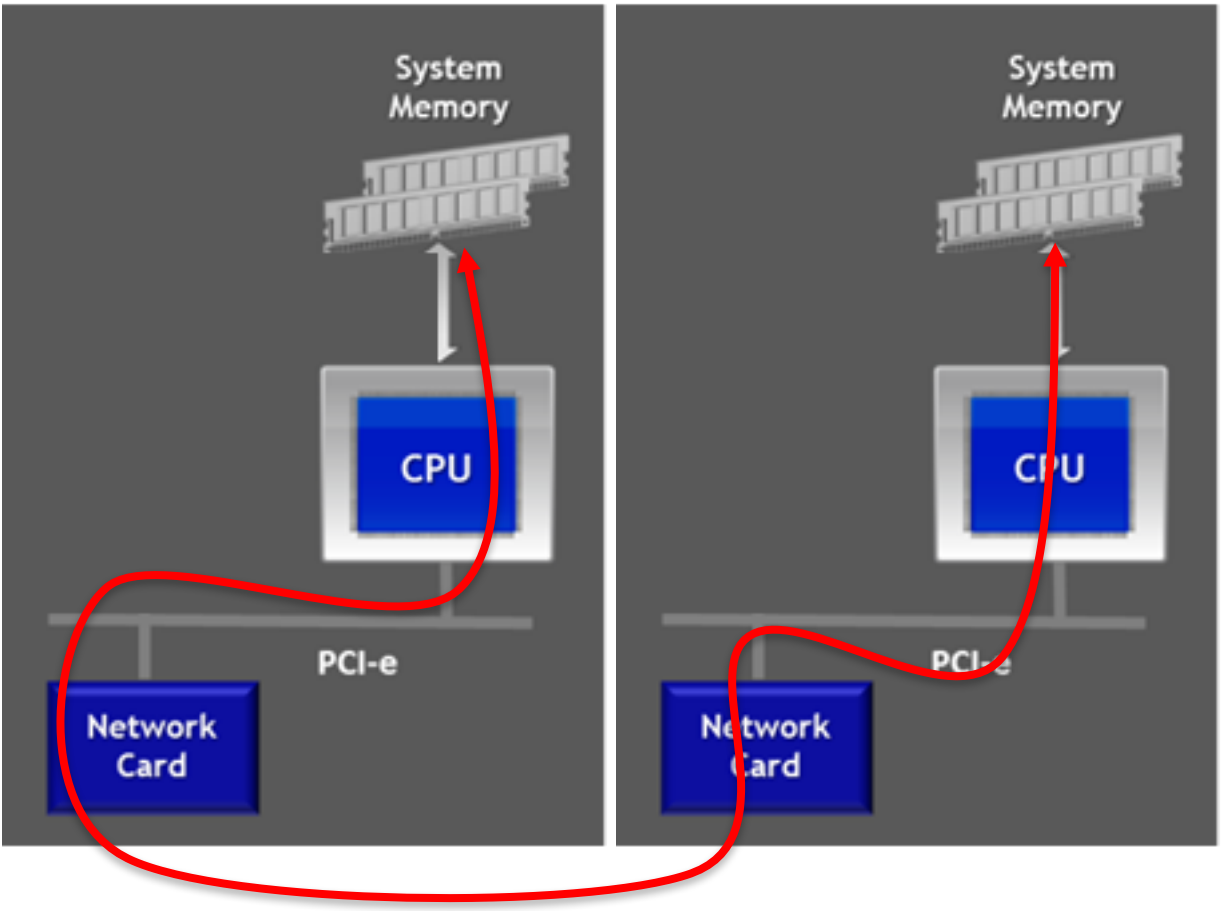
MPI + X programming

UTK resources: Rho Cluster with GPGPU
https://newton.utk.edu/doc/Documentation/Systems/RhoCluster

George Bosilca

CS462

MPI

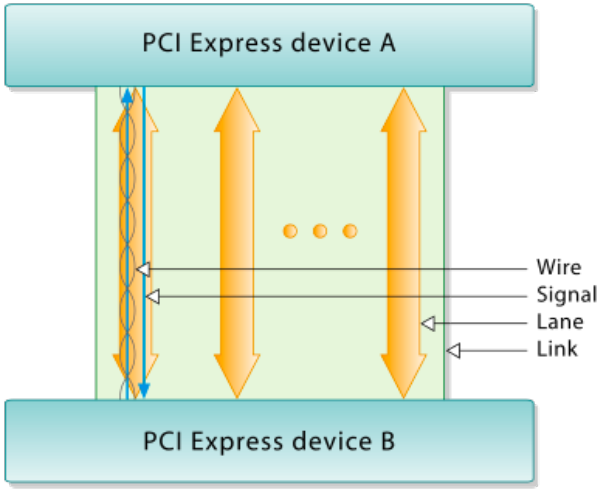


- Each programming paradigm only covers a particular spectrum of the hardware capabilities
  - MPI is about moving data between distributed memory machines
  - CUDA is about accessing the sheer computations power of a single GPU
  - OpenMP is about taking advantage of the multicores architectures
- What is involved in moving data between 2 machines
  - Bus (PCI/PCI-X)
  - Memory (pageable, pinned, virtual)
  - OS (security)

Applications need to fully take advantage of all available hardware capabilities . It became imperative to combine different programming paradigms together !

PCI* performance

PCI - Peripheral Component Interconnect
PCI-X - Peripheral Component Interconnect **eXtended**
PCIe - Peripheral Component Interconnect **Express**



https://en.wikipedia.org/wiki/PCI_Express

- split transactions (transactions with request and response separated by time)
- has a protocol and processing overhead due to the additional transfer robustness (line code below)
  - CRC and acknowledgements

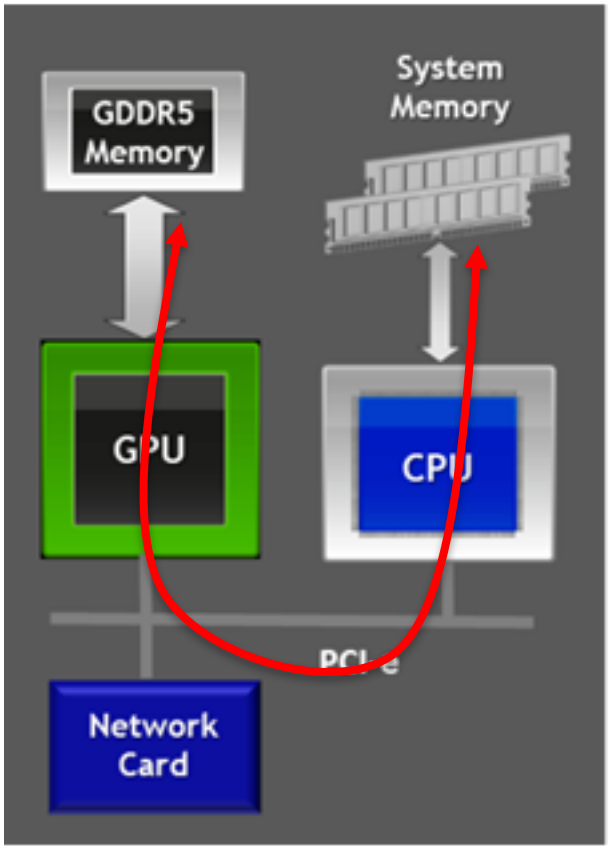
PCI Express link performance[29][32]

PCI Express version	Line code	Transfer rate[i]	Throughput[i]				
			×1	×2	×4	×8	×16
1.0	8b/10b	2.5 GT/s	250 MB/s	500 MB/s	1 GB/s	2 GB/s	4 GB/s
2.0	8b/10b	5.0 GT/s	500 MB/s	1 GB/s	2 GB/s	4 GB/s	8 GB/s
3.0	128b/130b	8.0 GT/s	984.6 MB/s	1.97 GB/s	3.94 GB/s	7.9 GB/s	15.8 GB/s
4.0	128b/130b	16.0 GT/s	1969 MB/s	3.94 GB/s	7.9 GB/s	15.8 GB/s	31.5 GB/s
5.0[30][31] *(expected in Q2 2019)*[33]	128b/130b	32.0 or 25.0 GT/s[ii]	3938 or 3077 MB/s	7.9 or 6.15 GB/s	15.8 or 12.3 GB/s	31.5 or 24.6 GB/s	63.0 or 49.2 GB/s

i. ^ a b In each direction (each lane is a dual simplex channel).

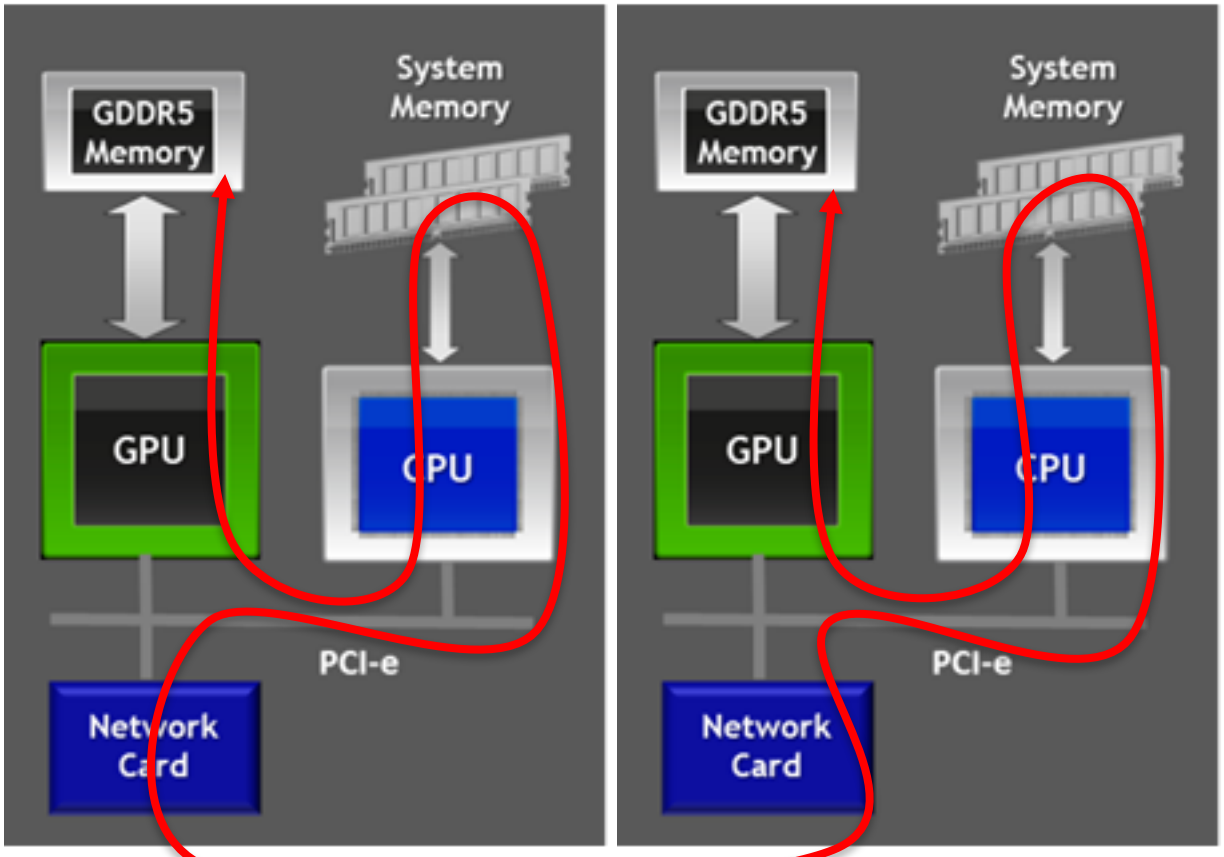
ii. ^ Both rates are being considered for technical feasibility.

CUDA



- The CPU is the main driver, it launches kernels on the GPU that perform computations sum<<<1,1>>>(2, 3, device_z);
  - Data must be moved between main memory and GPU prior to the computations
  - And must be fetched back once the computation is completed
  - In general these are explicit operations (cudaMemcpy)

MPI + CUDA

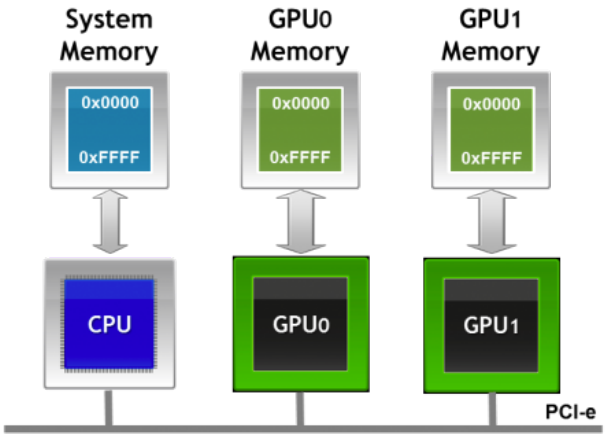


- MPI is handling main memory while CUDA kernels update the GPU memory. Explicit memory copy from the device to the CPU is necessary to ensure coherence.

```
if( 0 == rank ) {
    cudaMemcpy(host_buffer, device_buffer, size, cudaMemcpyDeviceToHost);
    MPI_Send(host_buffer, size, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
} else { // assume MPI rank 1
    MPI_Recv(host_buffer, size, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
    cudaMemcpy(device_buffer, host_buffer, size, cudaMemcpyHostToDevice);
}
```

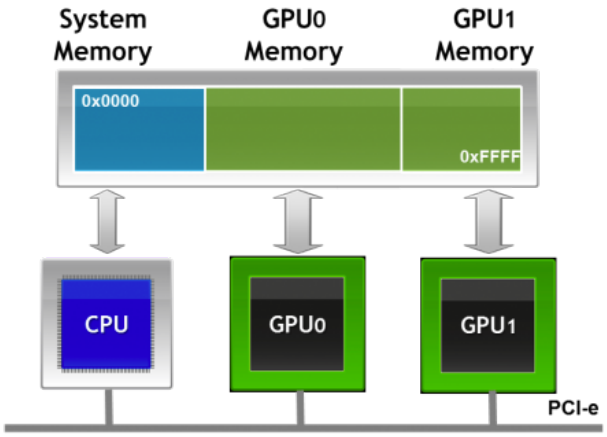
Unified Virtual Addressing (UVA)

No UVA: Multiple Memory Spaces



Devices have similar ranges of memory.
Impossible to know where a memory range belongs to

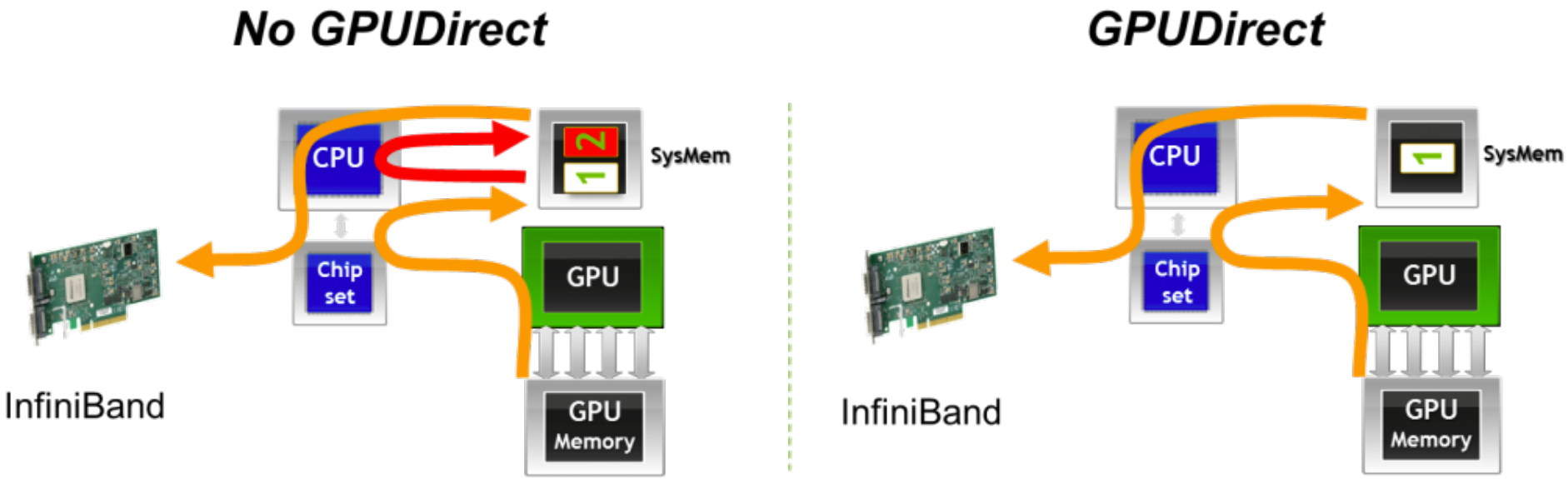
UVA: Single Address Space



Devices have continuous ranges of memory (managed by the hardware and OS).
A memory address clearly identifies the hardware device hosting the memory

UVA: One address space for all CPU and GPU memory
No need to alter libraries, they can how identify on which device the memory is located

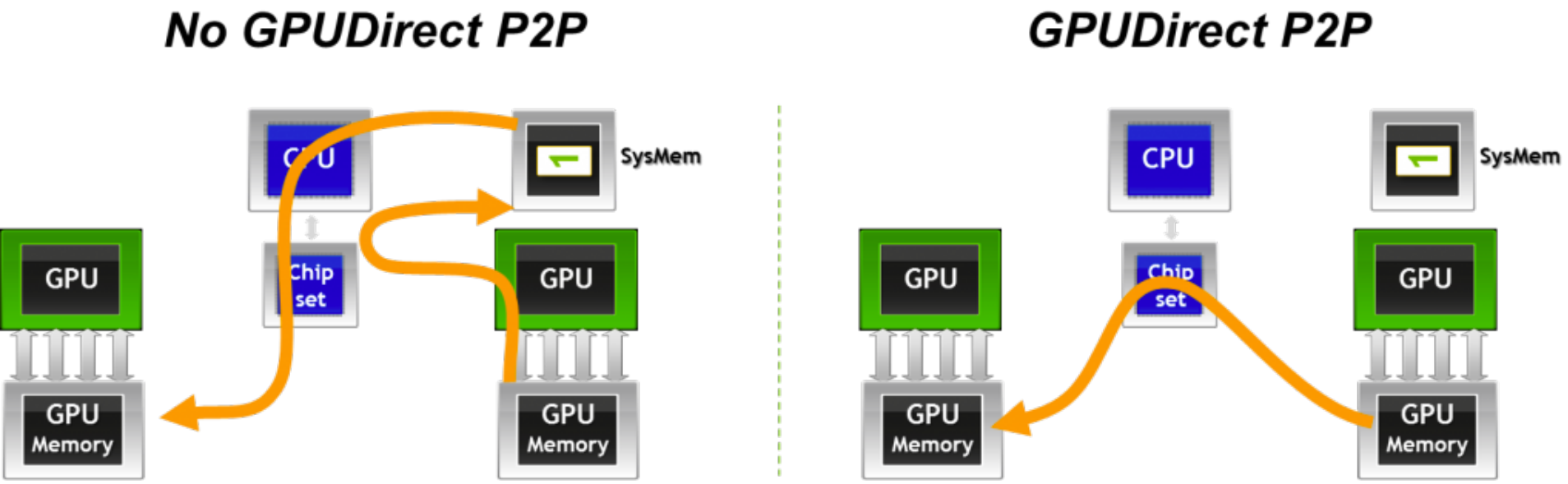
Nvidia GPUDirect



- Allowed pinned pages to be shared between different users
  - Ne need for multiple intermediary buffers to ready the data to be sent over the NiC

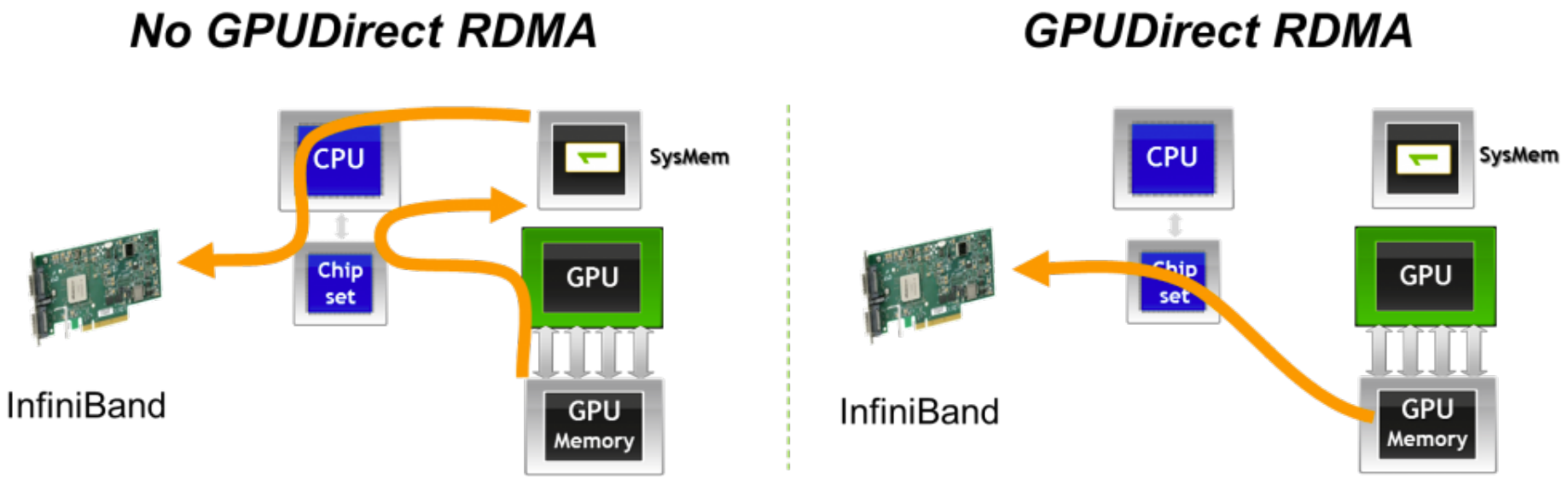
CUDA 3.1

Nvidia GPUDirect P2P



- P2P (Peer-to-Peer) allows memory to be copied between devices on the same node without going through the main memory.

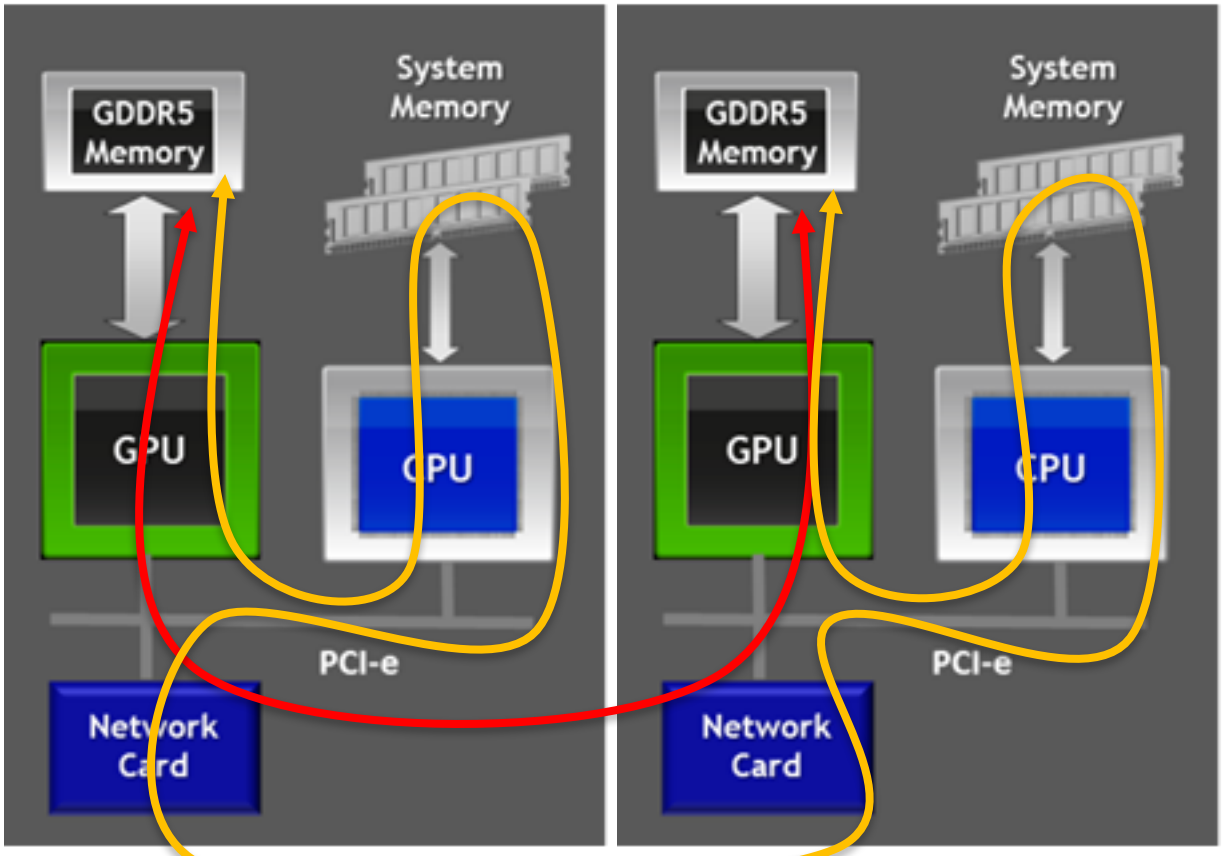
Nvidia GPUDirect RDMA



- Push the data out of the GPU directly into the NiC (or other hardware component).
  - Implement standard parts of the PCI-X protocol

CUDA 5.0

MPI + CUDA: integration/awarness

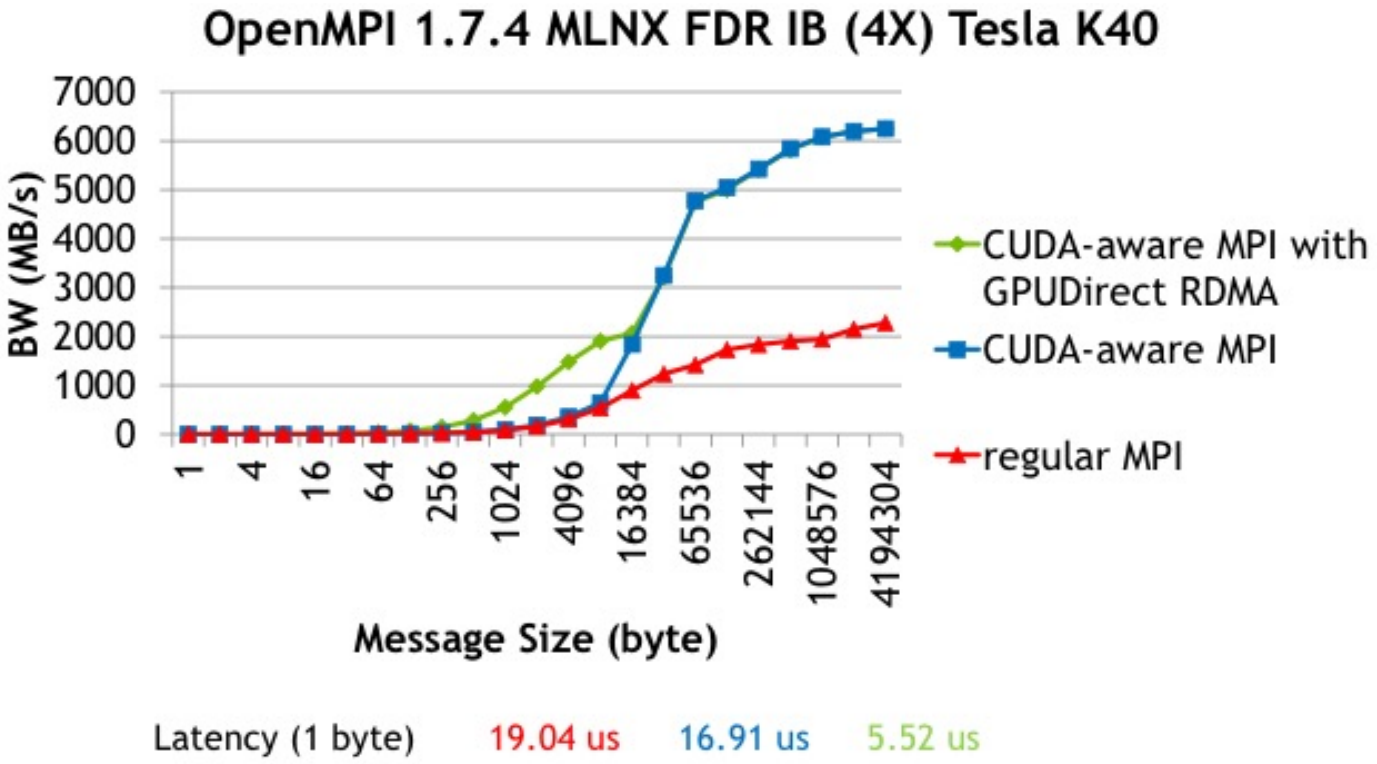


- Explicit memory copy from the device to the CPU is not necessary to ensure coherence.
- Data now flows directly between the local and remote memory (independent on the location of the memory).

```
if( 0 == rank ) {
   cudaMemcpy(host_buffer, device_buffer, size, cudaMemcpyDeviceToHost);
   MPI_Send(device_buffer, size, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
} else { // assume MPI rank 1
   MPI_Recv(device_buffer, size, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
   cudaMemcpy(device_buffer, host_buffer, size, cudaMemcpyHostToDevice);
}
```

CUDA-aware MPI

```
if( 0 == rank ) {
    MPI_Send(device_buffer, size, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
} else { // assume MPI rank 1
    MPI_Recv(device_buffer, size, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
}
```



MVAPICH2 1.8/1.9b

OpenMPI 1.10

CRAY MPI (MPT 5.6.2)

IBM Platform MPI (8.3)

SGI MPI (1.08)

Debugging

- Commercial tools (DDT, TV, …)
- If possibility to export xterm:
  mpirun –np 2 xterm –e gdb –args <my app args>
- If not, add a sleep (or a loop around a sleep in your applications) and use ”gdb –p <pid>” to attach to your process (once connected to the same node where the application is running)
- gdb can execute GDB commands from a FILE (with --command=FILE, -x)

Profiling

- Non-CUDA application: valgrind (free), or vtune (Intel), Score-P, Tau, Vampir
- CUDA application: nvprof from CUDA