COMP 4901Q: High Performance Computing (HPC)

Lecture 11: Combining Shared-Memory and Distributed-Memory Computing with OpenMP, CUDA, and MPI

Instructor: Shaohuai SHI (shaohuais@cse.ust.hk)
Teaching assistants: Mingkai TANG (mtangag@connect.ust.hk)
Yazhou XING (yxingag@connect.ust.hk)

Course website: https://course.cse.ust.hk/comp4901q/

Outline

- Hybrid OpenMP and MPI
  - Motivation
  - How to do
- Multi-GPU Programming
- Hybrid CUDA and MPI
  - Motivation
  - How to do

Programming Distributed Memory Systems

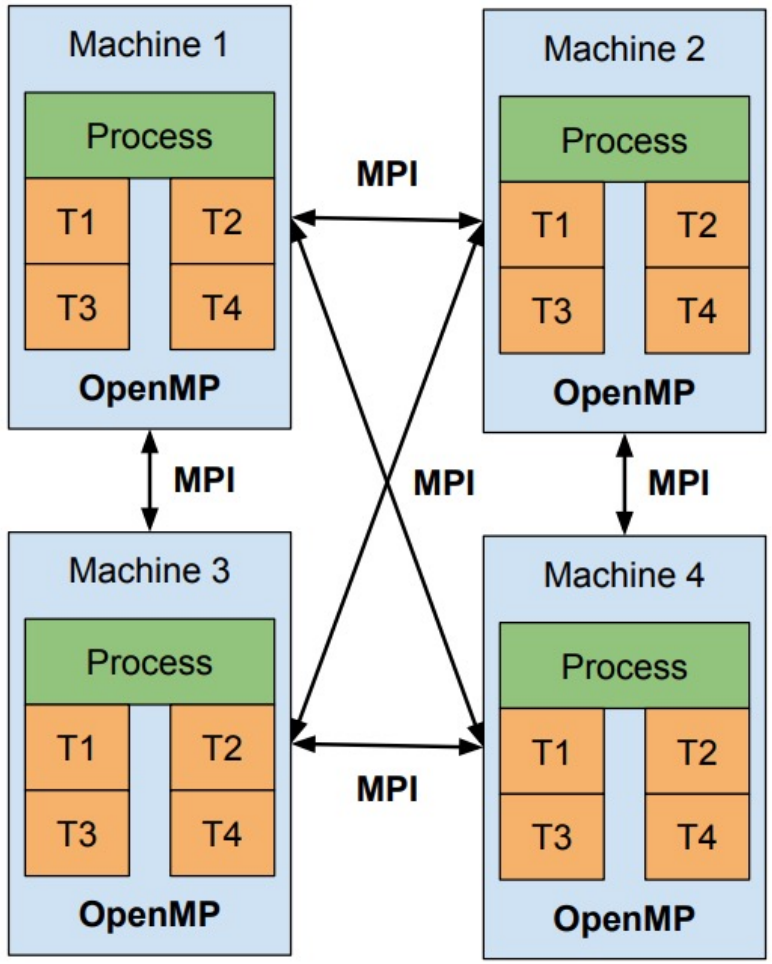
- Pure MPI
  - Pros
    - No modifications on existing MPI code
    - MPI library needs not to support multiple threads
  - Cons
    - Intra-node message passing is typically slower than shared-memory access with multi-threaded processes.
    - Different hardware requires different protocols

- Pure OpenMP
  - Requires distributed virtual shared memory
  - Communication of modified parts of pages at OpenMP flush (part of each OpenMP barrier)

OpenMP+MPI: Motivation

- Two-level Parallelization
  - Mimics hardware layout of cluster
    - Only place this really make sense
  - MPI **between** nodes or CPU sockets
  - OpenMP **within** shared-memory nodes or processors

- Pros
  - No message passing inside of the shared-memory processor (SMP) nodes
  - No topology problem
- Cons
  - Should be careful with sleeping threads



Example: Thread Support within OpenMPI

- In order to enable thread support in Open MPI, configure with
  - configure --enable-mpi-threads

- Progress threads to asynchronously transfer/receive data per network BTL
  - configure --enable-mpi-threads --enable-progress-threads

MPI Rules with OpenMP

- MPI should first be initialized to support multi-threaded MPI processes

```
int MPI_Init_thread(   int * argc, char ** argv[],
                       int thread_level_required,
                       int * thead_level_provided);
int MPI_Query_thread(  int * thread_level_provided);
int MPI_Is_main_thread(int * flag);
```

- thread_level_required
  - MPI_THREAD_SINGLE
    - Only one thread will execute
  - THREAD_MASTERONLY
    - MPI processes may be multi-threaded, (virtual value, but only master thread will make MPI-calls not part of the standard) AND only while other threads are sleeping
  - MPI_THREAD_FUNNELED
    - Only master thread will make MPI-calls
  - MPI_THREAD_SERIALIZED
    - Multiple threads may make MPI-calls, but only one at a time
  - MPI_THREAD_MULTIPLE
    - Multiple threads may call MPI with **no restrictions**

How to: Single-Threaded MPI Calls

- Only the master thread calls MPI

```
 1 #include <mpi.h>
 2
 3 int main(int argc, char **argv)
 4 {
 5     int rank, size, ie, i;
 6     ie = MPI_Init(&argc,&argv[]);
 7     ie = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 8     ie = MPI_Comm_size(MPI_COMM_WORLD, &size);
 9     //Setup shared mem, comp/comm
10 #pragma omp parallel for
11     for(i=0; i<n; i++){
12         // <work>;
13     }
14     // compute & communicate
15     ie = MPI_Finalize();
16     return 0;
17 }
```

Example 1: Estimate pi

```
14      MPI_Init(&argc, &argv);
15      MPI_Comm_size(MPI_COMM_WORLD, &nproc);
16      MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
17      MPI_Get_processor_name(processor_name, &namelen);
18
19      double start = omp_get_wtime();
20      sum = 0.0;
21      printf("h: %lf \n", h);
22  #pragma omp parallel for shared(myrank,nproc),private(i,x),reduction(+:sum)
23      for (i = myrank; i <= N; i = i+nproc) {
24          x = h * (i+0.5);
25          sum += 4.0/(1.0+x*x);
26      }
27      mypi = h * sum;
28      MPI_Reduce(&mypi,&pi,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
29      double end = omp_get_wtime();
30      printf("Result of PI: %.20lf, estimate: %.20lf\n", PI, pi);
31      printf("Running time: %f seconds\n", end - start);
32      MPI_Finalize();
```

How to: Funneled MPI Calls via Master

- Call MPIs inside a parallel region, but force to use the master thread
  - Should use MPI_THREAD_FUNNELED or higher
  - Best to use OMP_BARRIER
    - there is no implicit barrier in the master workshare construct, OMP_MASTER
    - in the example, the master thread will execute a single MPI call within the OMP_MASTER construct
    - All other threads will be sleeping
    - The additional barrier implies also the necessary cache flush!

```
 1 #include <mpi.h>
 2
 3 int main(int argc, char **argv)
 4 {
 5     int rank, size, ie, i;
 6 #pragma omp parallel
 7     {
 8 #pragma omp barrier
 9 #pragma omp master
10         {
11             ie = MPI_XXX(...);
12         }
13 #pragma omp barrier
14     }
15 }
```

Barrier is Necessary

```
 5 #pragma omp parallel
 6 {
 7 #pragma omp for nowait
 8     for (i=0; i<1000; i++)
 9         a[i] = buf[i];
10
11 #pragma omp barrier
12 #pragma omp master
13     MPI_Recv(buf,...);
14 #pragma omp barrier
15
16 #pragma omp for nowait
17     for (i=0; i<1000; i++)
18         c[i] = buf[i];
19 }
20 /* omp end parallel */
```

How to: Serialized MPI Calls

- Call MPIs inside a parallel region, but only use one thread (not necessarily the master thread)
  - Should use MPI_THREAD_SERIALIZED or higher
  - Best to use OMP_BARRIER only at beginning, since there is an implicit barrier in the SINGLE workshare construct
    - Example is the simplest one: any thread (not necessarily master) will execute a single MPI call within the OMP_SINGLE construct
    - All other threads will be sleeping

```
#include <mpi.h>

int main(int argc, char **argv)
{
int rank, size, ie, i;
ie= MPI_Init_thread(
MPI_THREAD_SERIALIZED, ...);
#pragma omp parallel
{
#pragma omp barrier
#pragma omp single
{
        ie= MPI_XXX(...);
}
//Don't need omp barrier
}
}
```

How to: Overlapping Communication and Computation

- One core is capable of saturating the lanes of the PCIe/network link
  - Why use all cores to communicate?
  - Instead, communicate using just one or several cores can do work with the rest during communication
- Must have support for MPI_THREAD_FUNNELED or higher to do this
- Can be difficult to manage and load-balance!

```
if (my_thread_rank < …) {
   MPI_Send/Recv….
   // i.e., communicate all halo data
} else {
   Execute those parts of the application
   that do not need halo data
   // (on non-communicating threads)
}
Execute those parts of the application
that need halo data (on all threads)
```

Multi-GPU Programming

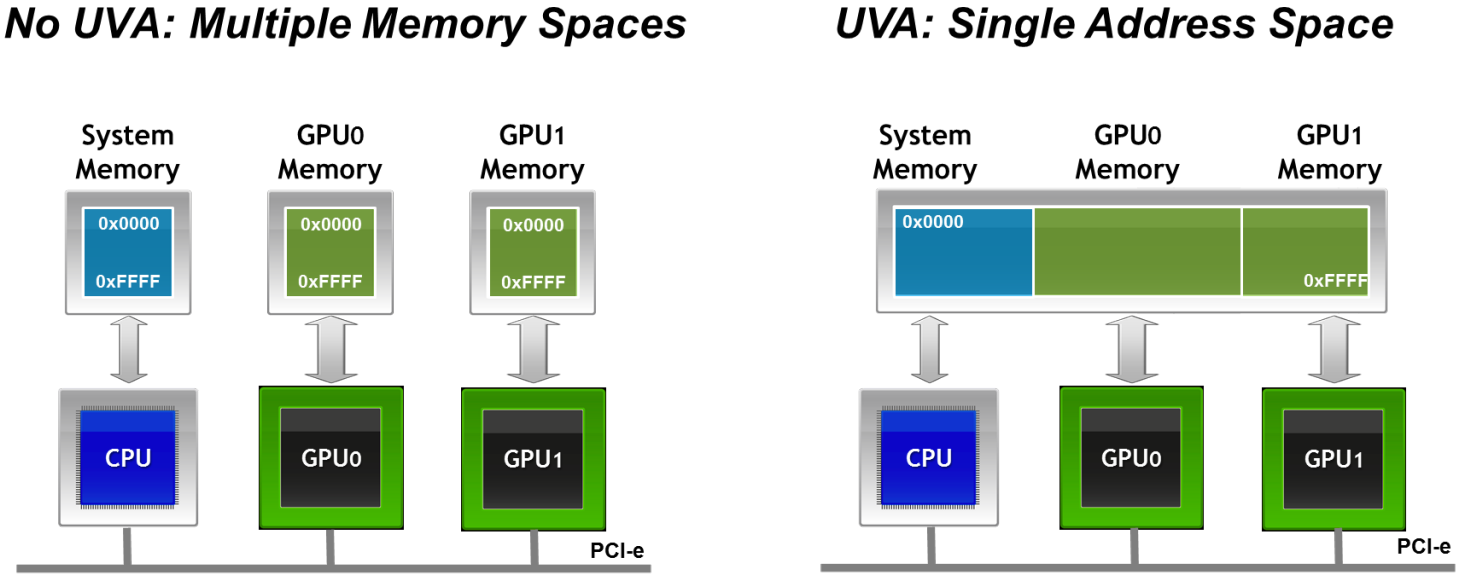
- Multiple GPUs on a single node
  - GPUs have consecutive integer IDs, starting with 0
  - A host thread can maintain more than one GPU context at a time
  - **cudaSetDevice** allows to change the "active" GPU
  - Multiple host threads can establish contexts with the same GPU driver

Data Communication between GPUs

- Explicit copies via host
- Zero-copy shared host array
  - Unified virtual addressing
- Per-device arrays with peer-to-peer exchange transfers
  - Transfer data between GPUs using PCIe P2P support
  - Done by GPU DMA hardware – host CPU is not involved
    - Data traverses PCIe links, without touching CPU memory
- GPUDirect makes data communication easier

Unified Virtual Addressing (UVA)

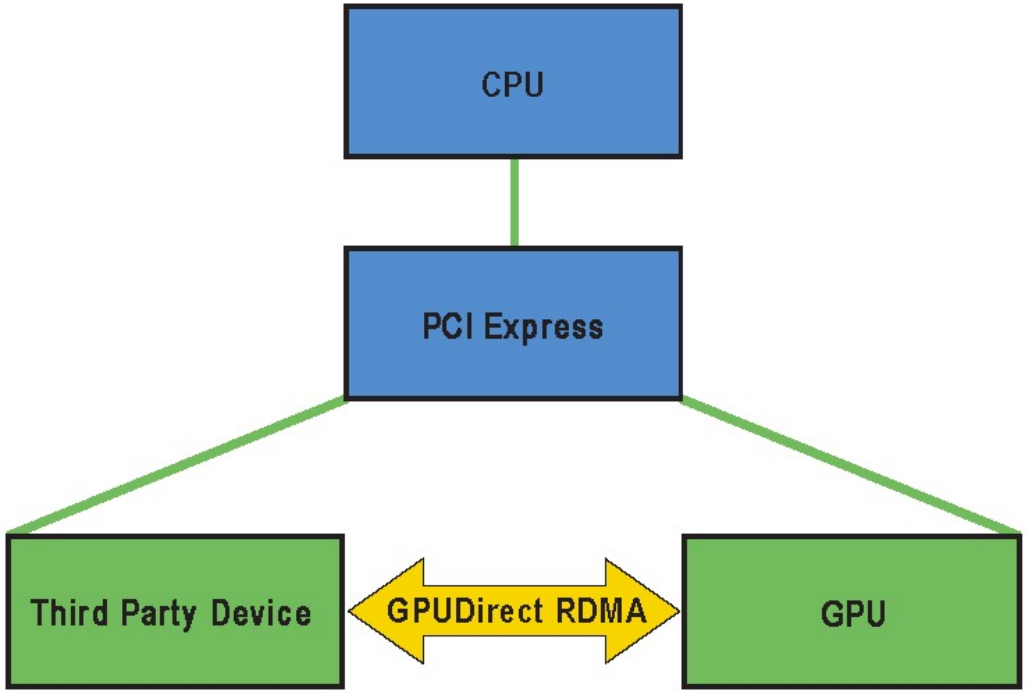
- No UVA: Separate Address Spaces vs. UVA



- UVA: One address space for all CPU and GPU memory
  - Determine physical memory location from a pointer value
  - Enable libraries to simplify their interfaces (e.g. MPI and cudaMemcpy)
  - Supported on devices with compute capability 2.0

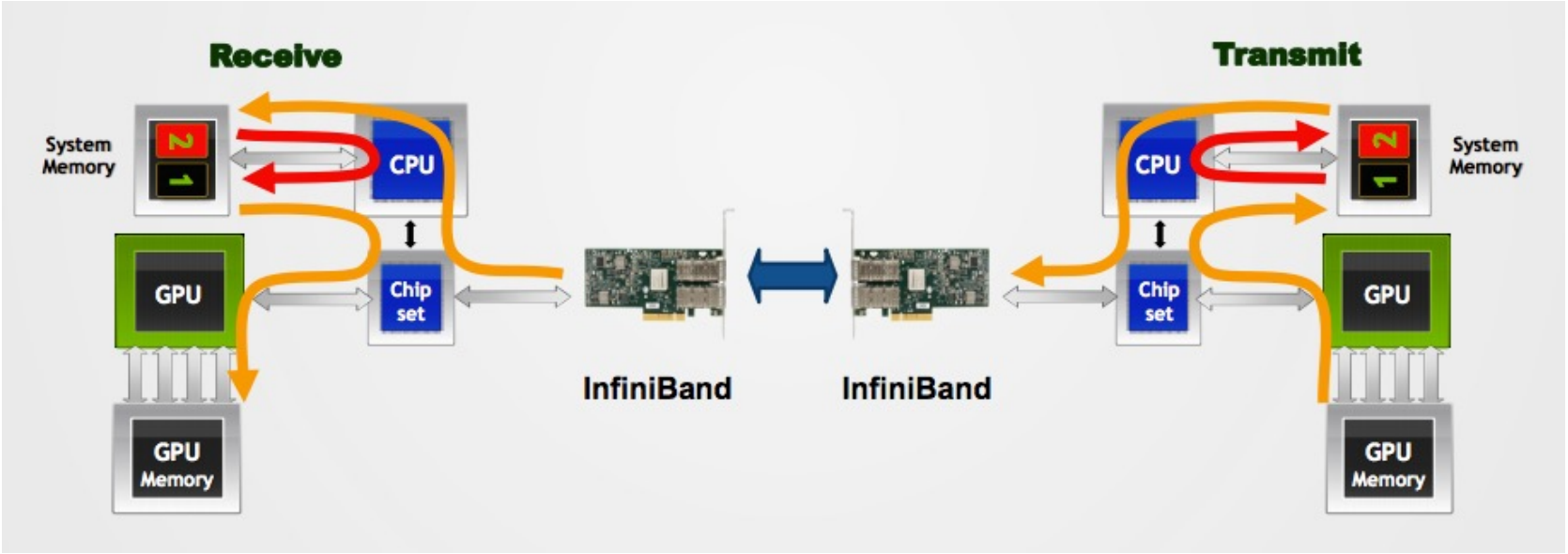
GPUDirect

- Accelerated communication across multiple GPUs
  - Peer-to-peer access
    - Multiple GPUs within a single GPU card (e.g., Nvidia Tesla K80)
    - NVLink
  - Remote access through RMDA



Pre-GPUDirect

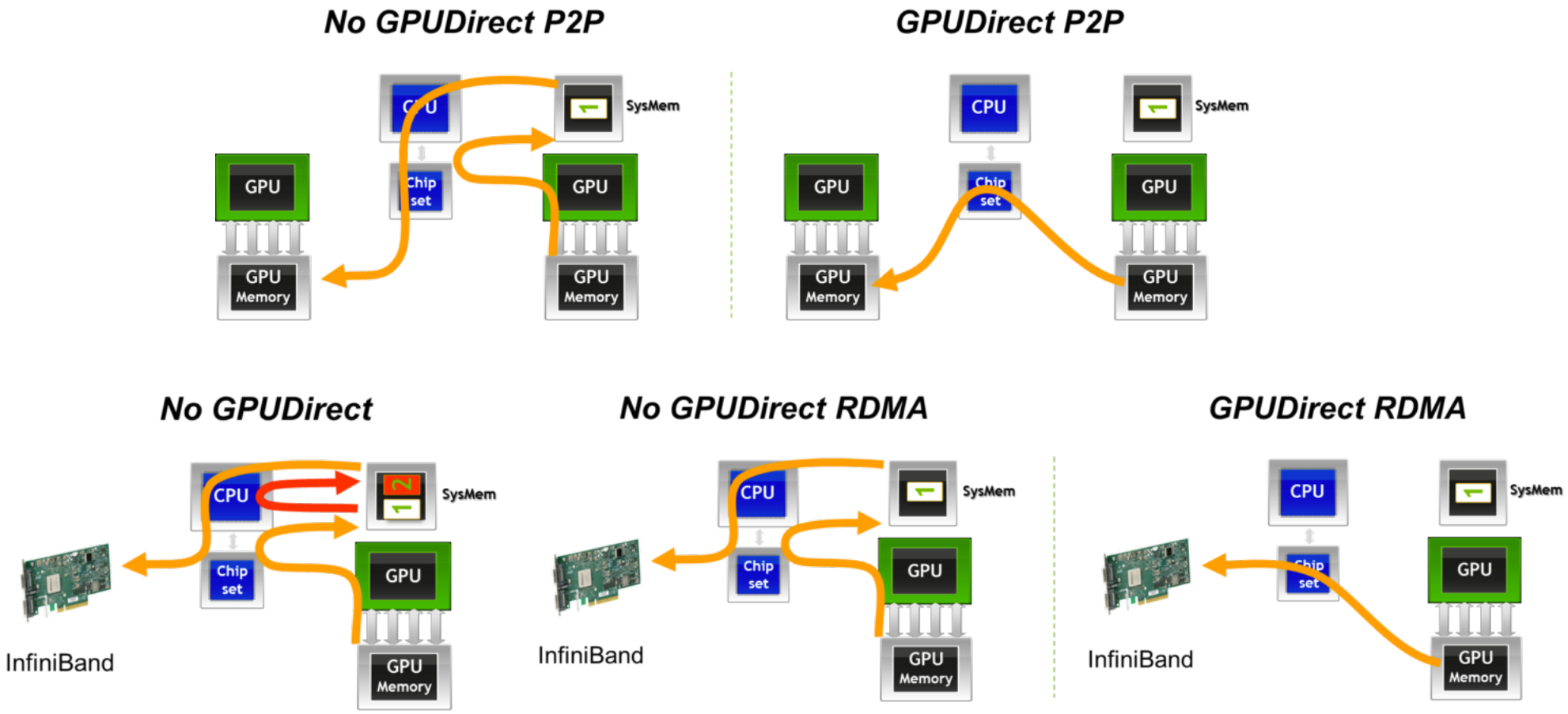
- Pre-GPUDirect, GPU communications required CPU involvement in the data path
  - Memory copies between the different "pinned buffers"
  - Slow down the GPU communications and creates communication bottleneck



Source: https://developer.nvidia.com/gpudirect

GPUDirect

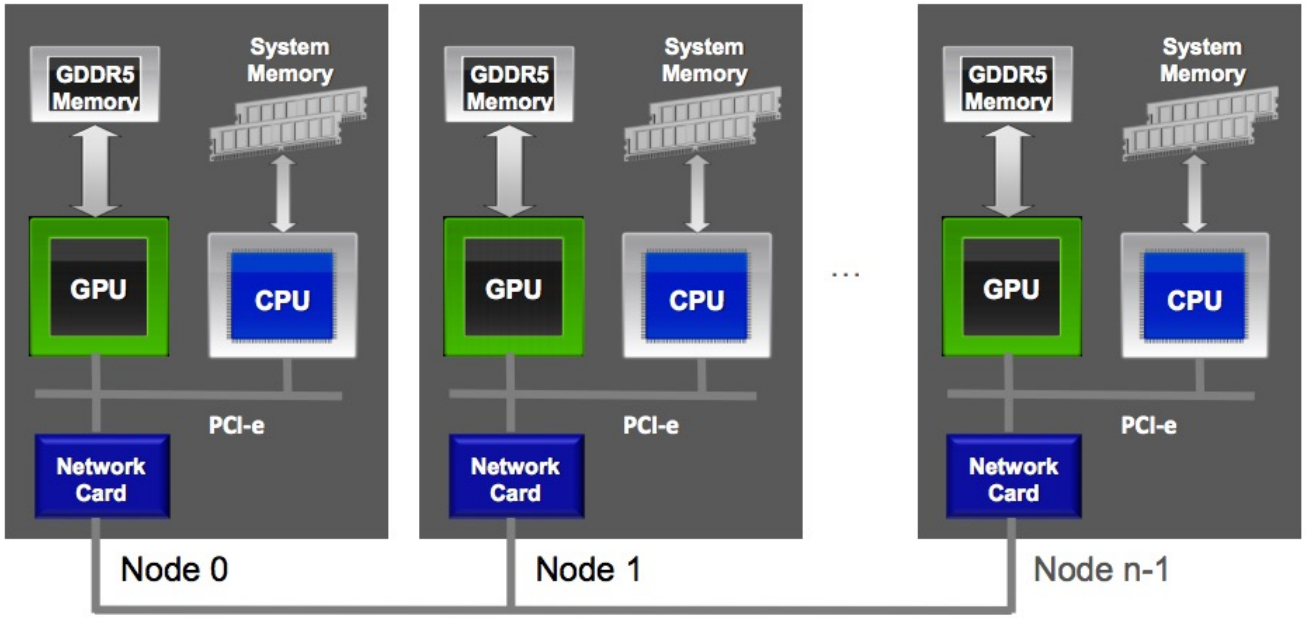
- NVIDIA GPUDirect technologies provide high-bandwidth, low-latency communications with NVIDIA GPUs



Source: https://developer.nvidia.com/blog/introduction-cuda-aware-mpi/

Hybrid CUDA and MPI: Motivation

- MPI is easy to exchange data located at different processors
  - CPU <-> CPU: Traditional MPI
  - GPU <-> GPU: CUDA-Aware MPI
- MPI+CUDA makes the application run more efficiently
  - All operations that are required to carry out the message transfer can be pipelined
  - Acceleration technologies like GPUDirect can be utilized by the MPI library transparently to the user.



UVA Data Exchange with MPI

UVA: Unified Virtual Addressing

UVA

```
//MPI Rank 0
MPI_Send(s_buf_d, size, …);

//MPI Rank n-1
MPI_Recv(r_buf_d, size, …);
```

CUDA-aware MPI is required!

Non-UVA

```
//MPI Rank 0
cudaMemcpy(s_buf_h, s_buf_d, size,…);
MPI_Send(s_buf_h,size,…);

//MPI Rank n-1
MPI_Recv(r_buf_h, size, …);
cudaMemcpy(r_buf_d, r_buf_h, size,…);
```

GPU Collectives with NCCL

- NCCL (pronounced "Nickel") is a library of multi-GPU collective communication primitives
  - https://developer.nvidia.com/nccl

- Features
  - Good performance
  - Ease of programming
  - Compatibility
    - Easily integrated with MPI

GPU Collectives with NCCL

- Supported collective communications
  - ncclAllReduce
  - ncclBroadcast
  - ncclReduce
  - ncclAllGather
  - ncclReduceScatter

Reading List

- https://developer.nvidia.com/blog/introduction-cuda-aware-mpi/
- https://developer.nvidia.com/blog/fast-multi-gpu-collectives-nccl/
- Chu, Ching-Hsiang, et al. "Exploiting hardware multicast and GPUDirect RDMA for efficient broadcast." *IEEE Transactions on Parallel and Distributed Systems* 30.3 (2018): 575-588.