COMP4007: 并行处理和体系结构

第七章：并行编程高级主题I

授课老师：王强、施少怀
助　　教：林稳翔、刘虎成

哈尔滨工业大学（深圳）

要点

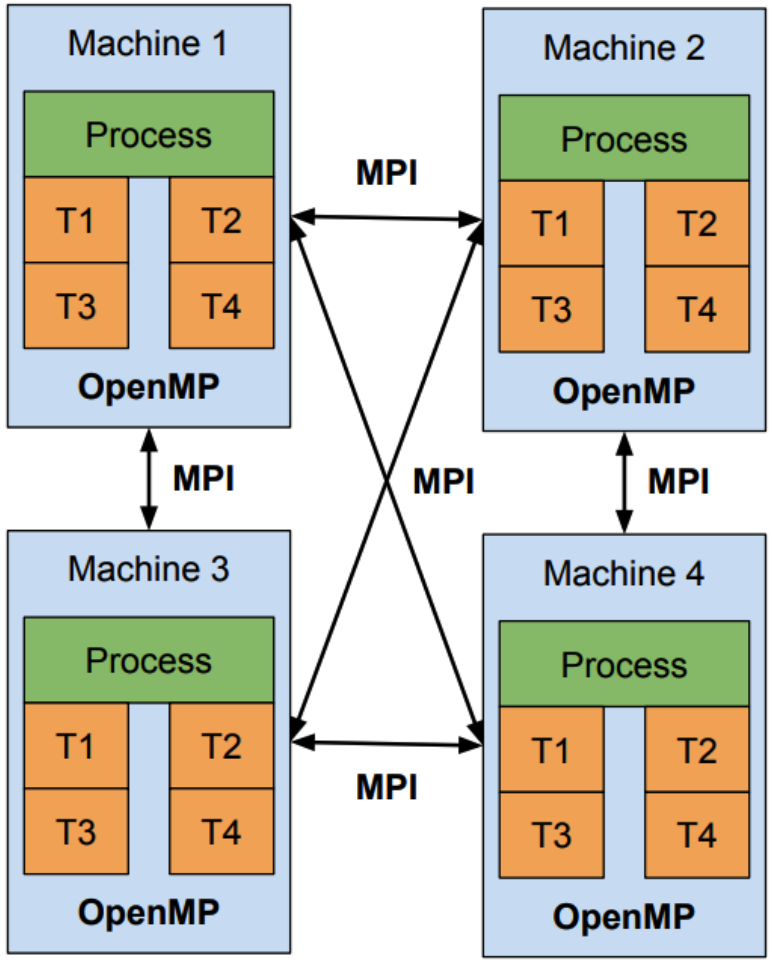
- 融合OpenMP与MPI
  - 动机
  - 方法
- 多GPU编程
- 融合CUDA与MPI
  - 动机
  - 方法

分布式内存系统编程

- 纯粹的MPI
  - 好处
    - 不需要对现有的MPI代码进行修改
    - MPI库不需要支持多线程
  - 坏处
    - 节点内消息传递通常比多线程处理的共享内存访问要慢
    - 不同的硬件需要不同的协议
- 纯粹的OpenMP
  - 需要分布式虚拟共享内存

OpenMP+MPI：动机

- 两级并行
  - 模拟集群的硬件布局
  - 跨节点或跨CPU使用MPI
  - 在共享内存的节点或处理器中使用OpenMP
- 好处
  - 在共享内存处理器的节点中不需要消息的传递
  - 没有拓扑问题
- 坏处
  - 需要注意休眠进程



示例：OpenMPI中的线程支持

- 要启用OpenMPI中的线程支持，如下进行配置
  - configure --enable-mpi-threads

- 进度线程异步传输/接收数据
  - configure --enable-mpi-threads --enable-progress-threads

包含OpenMP的MPI规则

- MPI必须首先被初始化来支持多线程MPI进程

```
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
    - 只有一个线程会执行
  - THREAD_MASTERONLY
    - MPI进程可以是多线程的, 且仅当其他线程在休眠时
  - MPI_THREAD_FUNNELED
    - 仅主线程会进行MPI调用
  - MPI_THREAD_SERIALIZED
    - 多线程可能进行MPI调用，但只会进行一次
  - MPI_THREAD_MULTIPLE
    - 多线程可能会无限制调用MPI

方法：单线程MPI调用

- 仅主线程调用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 }
```

示例1：估计π


```
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();
```

方法：通过主线程进行MPI调用

- 在并行区内调用MPI，但强制使用主线程
  - 必须指定 MPI_THREAD_FUNNELED 及以上级别
  - 最好使用 OMP_BARRIER
    - 在主工作共享结构体中没有显式的屏障，OMP_MASTER
    - 在例子中，主线程会在OMP_MASTER结构体中执行一个单独的MPI调用
    - 所有其他线程将休眠
    - 附加屏障同样意味着必要的缓存刷新

```
 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 }
```

屏障是必须的


```
 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 */
```

方法：序列化MPI调用

- 在并行区内调用MPI，但仅使用单线程（不一定是主线程）
  - 必须指定MPI_THREAD_SERIALIZED及以上级别
  - 最好只在开始时使用OMP_BARRIER，因为在SINGLE工作共享结构体中存在隐式的屏障
    - 最简单的例子：任何线程（未必是主线程）将会在OMP_SINGLE 结构体中执行一个单一的MPI调用
    - 所有其他线程将休眠

```
#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
}
}
```

方法：重叠通信与计算

- 一个核心就可跑满PCIe与网络的所有通道
  - 为什么要用所有核心进行通讯？
  - 相反的，仅使用一个或少数核心进行通讯，还可以在通信期间与其他核心一起完成工作
- 必须指定MPI_THREAD_FUNNELED及更高级别来实现
- 可能会增加管理与负载均衡的难度

```
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)
```

多GPU编程

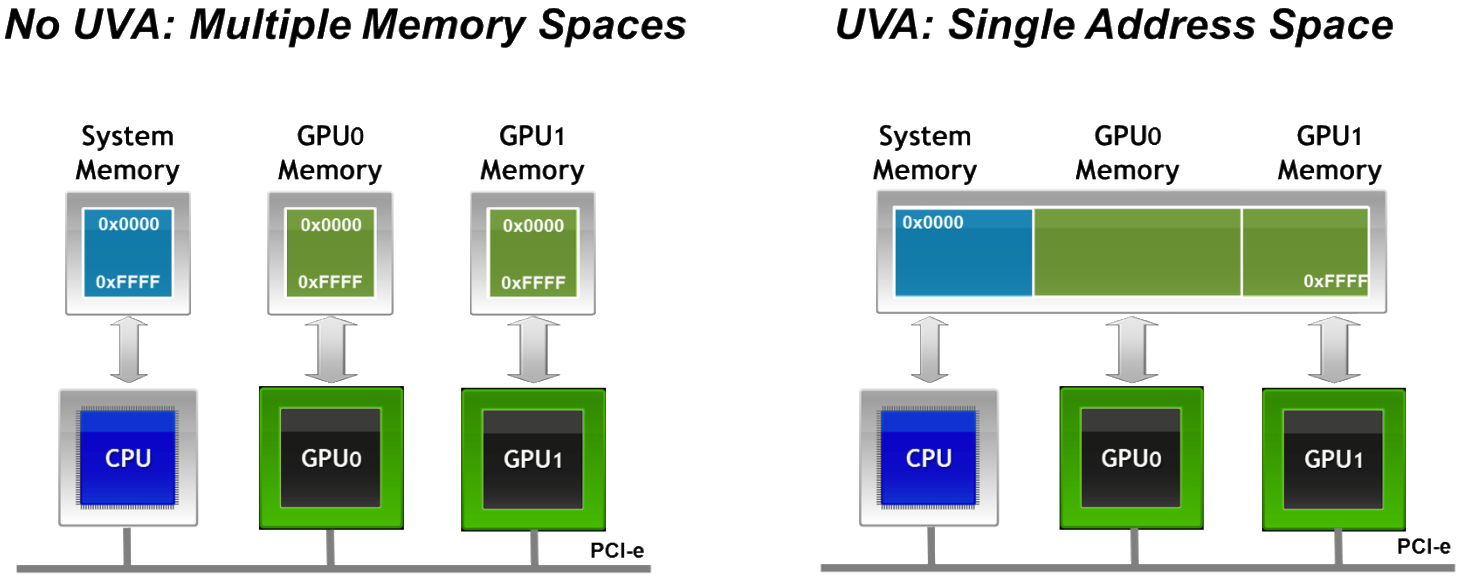
- 单节点多GPU
  - GPU设备拥有连续的整数编号，从0开始
  - 一个主机线程可以同时维护不止一个GPU上下文
  - **cudaSetDevice** 允许改变“激活的”GPU
  - 多个主机线程可以利用同一个GPU驱动器建立上下文

GPU间的数据通信

- 通过主机显式复制
- 0-复制共享的主机数组
  - 统一虚拟寻址
- 单设备数组通过点对点(P2P)传输交换数据
  - 利用PCIe的P2P传输支持在GPU间传输数据
  - 通过GPU直接内存访问硬件实现——不通过主机CPU
    - 数据以不涉及CPU内存的方式横贯PCIe连接
- GPU直连使数据通信更加简单

统一虚拟寻址(UVA)

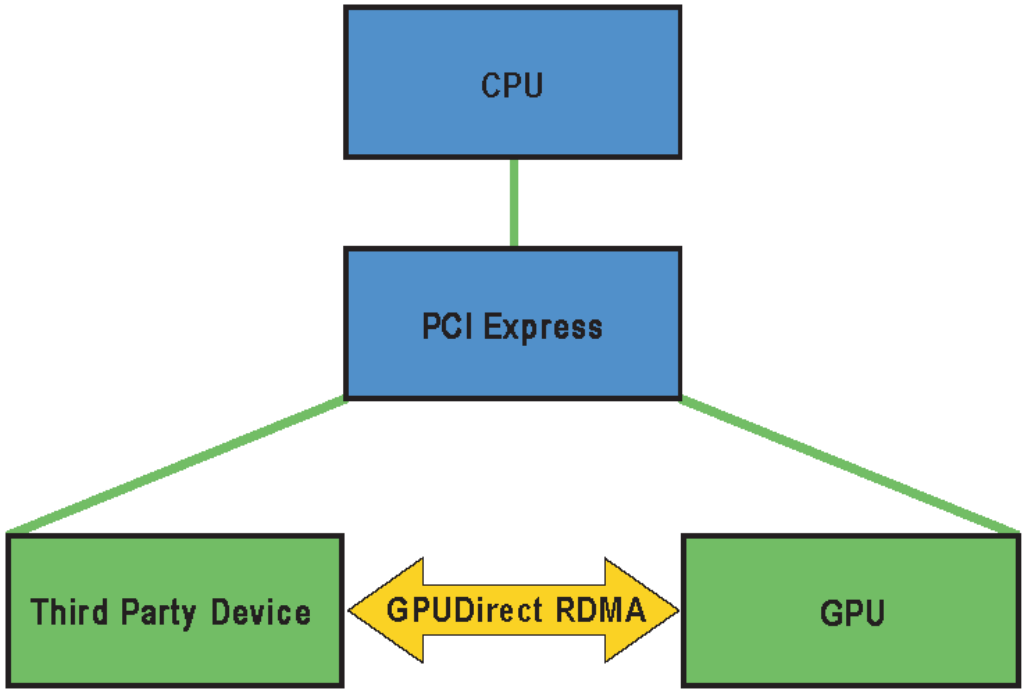
- 非UVA：独立地址空间 vs. UVA



- UVA：面向所有CPU与GPU存储的统一地址空间
  - 根据一个指针确定物理存储位置
  - 利用对应库简化接口（如 MPI 和 cudaMemcpy）
  - 只在计算能力高于2.0的设备上支持

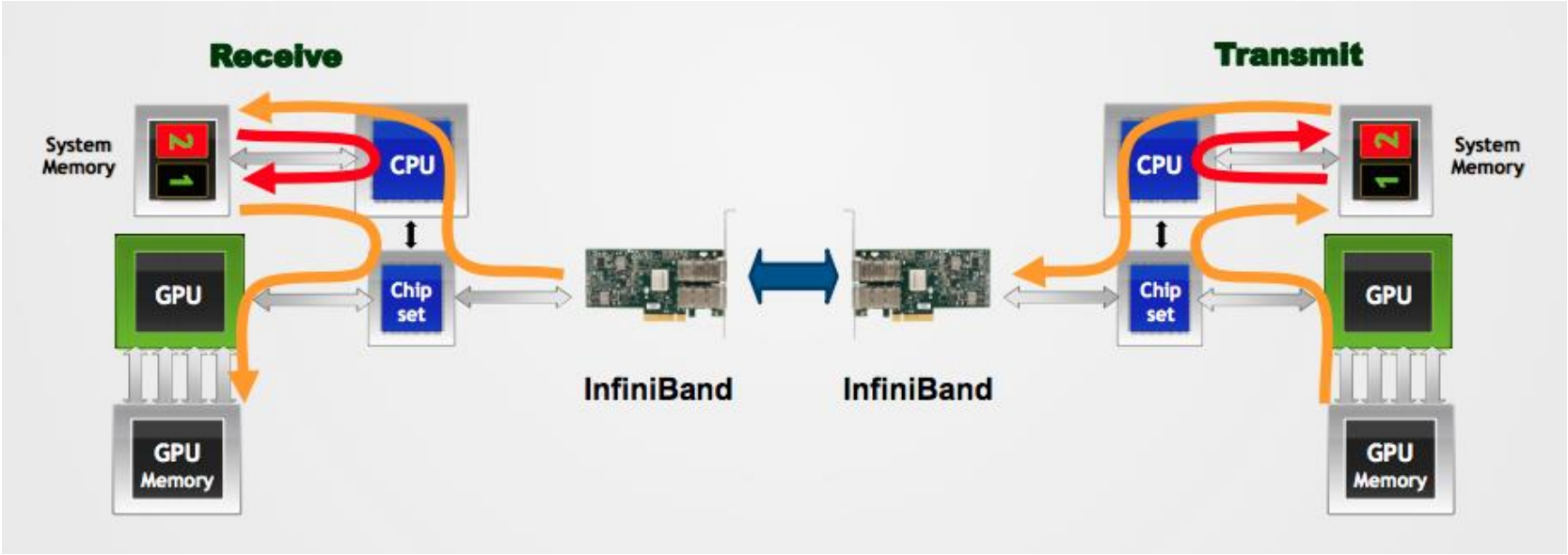
GPUDirect

- 加速多GPU之间的通信
  - 点对点(P2P)访问
    - 多芯计算卡（如Nvidia Tesla K80）
    - NVLink
  - 通过RDMA (Remote Direct Memory Access) 的远程访问



GPUDirect之前

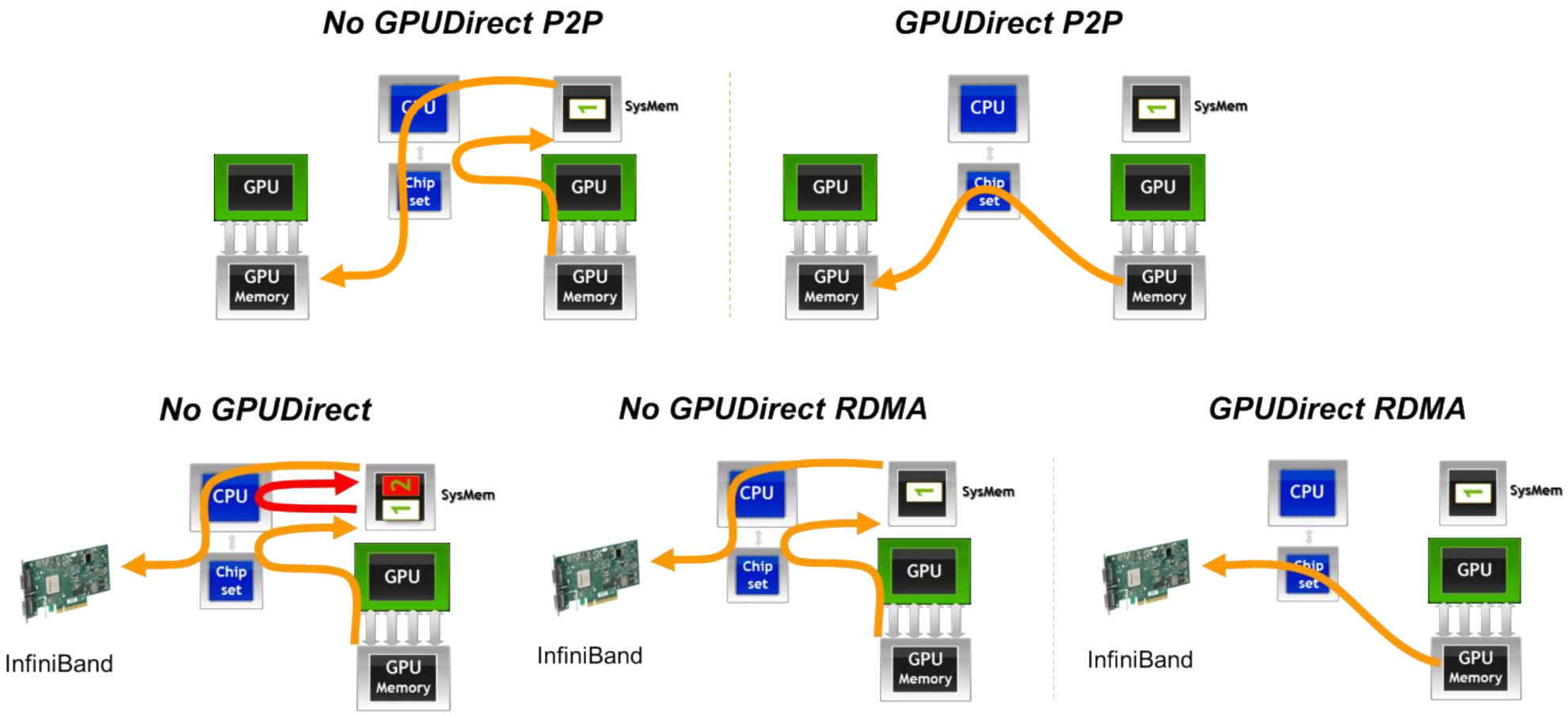
- 在GPUDirect之前，GPU通信需要CPU参与数据的传递
  - 数据在内存中不同“锁页缓存”间拷贝
  - 降低了GPU通信速度，造成了通信瓶颈



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

GPUDirect

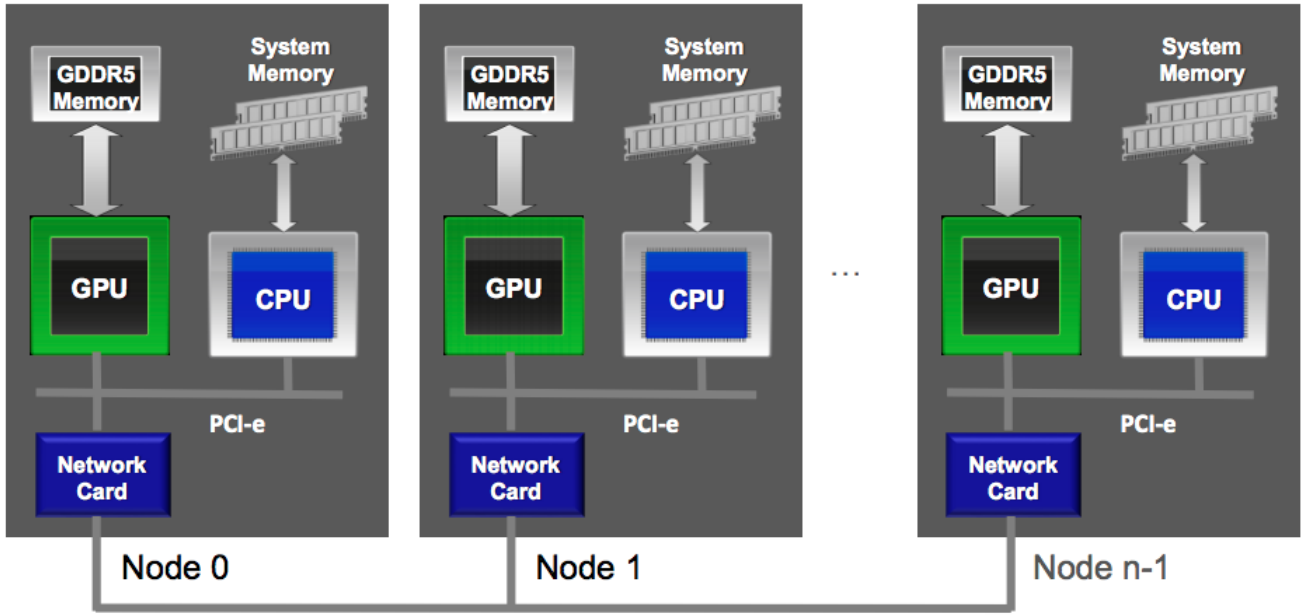
- NVIDIA GPU直连技术提供高带宽、低延迟的GPU间通信



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

混合CUDA与MPI：动机

- MPI易于交换位于不同处理器上的的数据
  - CPU <-> CPU：传统MPI
  - GPU <-> GPU：CUDA-Aware MPI
- MPI+CUDA使应用更加高效地运行
  - 被要求执行消息传递的所有操作都可以被流水线化
  - 像GPU直连这样的加速技术可以被MPI库显式的利用



MPI中的UVA数据交换

UVA：统一虚拟寻址(Unified Virtual Addressing)

UVA

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

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

需要 CUDA-aware MPI！

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,…);
```

利用NCCL进行GPU聚合

- NCCL（读作“Nickel”）是一个多GPU聚合通信库
  - https://developer.nvidia.com/nccl

- 特性
  - 高性能
  - 易编程
  - 高兼容性
    - 易与MPI集成

利用NCCL进行GPU集合通信

- 支持的集合通信操作
  - ncclAllReduce
  - ncclBroadcast
  - ncclReduce
  - ncclAllGather
  - ncclReduceScatter

阅读列表

- 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.