Scalable Cluster Computing with NVIDIA GPUs
Axel Koehler - NVIDIA
Outline

- Introduction to Multi-GPU Programming
- Communication for Single Host, Multiple GPUs
- Communication for Multiple Hosts, Multiple GPUs
- GPU-aware MPI implementations
- NUMA Considerations
- Summary
Multi-GPU Programming

Applications are using multiple GPUs in single/multiple nodes

Advantages of the Multi-GPU usage
- More memory $\rightarrow$ larger problem sizes
- More processing power $\rightarrow$ faster runtimes
- Multiple GPUs per node improve perf/Watt ratio

Taxonomy of Inter-GPU Communication Cases

<table>
<thead>
<tr>
<th></th>
<th>Network nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Single</td>
</tr>
<tr>
<td>Single process</td>
<td>Single-threaded</td>
</tr>
<tr>
<td></td>
<td>Multi-threaded</td>
</tr>
<tr>
<td>Multiple processes</td>
<td></td>
</tr>
</tbody>
</table>

GPUs can communicate via P2P or via host memory
GPUs communicate via host-side message passing
• Model geometry is decomposed; partitions are sent to independent compute nodes on a cluster
• Compute nodes operate distributed parallel using MPI communication to complete a solution per time step
• In a hybrid model a partition will be also mapped to GPUs and supports the CPU computation
Communication for Single Host, Multiple GPUs
Managing multiple GPUs from a single CPU thread

- CUDA 4.0 makes it easy to work with multiple GPUs
- Easy to coordinate work among multiple GPUs for instance halo exchange (no need for CPU multithreading to handle it)
- Any host thread can access all GPUs in the system – just cudaSetDevice(id)
- Current GPU can be changed while async calls (kernels, memcopies) are running
  - Possible to queue up a bunch of async calls to a GPU and then switch to another GPU

```c
cudaSetDevice( 0 );
kernel<<<...>>>(...);
cudaMemcpyAsync(...);
cudaSetDevice( 1 );
kernell<<<...>>>(...);
```
Unified Virtual Addressing (UVA)

- CPU and GPU allocations use unified virtual address space
  - Think of each one (CPU, GPU) getting its own range of a single VA space
  - Driver/device can determine from an address where data resides
- Requires:
  - 64-bit Linux or 64-bit Windows with TCC driver
  - Fermi or later architecture GPUs (compute capability 2.0 or higher)
  - CUDA 4.0 or later
- A GPU can dereference a pointer that is:
  - an address on another GPU
  - an address on the host (CPU)
UVA and Multi-GPU Programming

- Two interesting aspects
  - Peer-to-peer (P2P) memcopies
  - Accessing another GPU’s addresses
- Both require peer-access to be enabled
- Peer-access is not available if
  - One of the GPUs is pre-Fermi
  - GPUs are connected to different Intel IOH chips on the motherboard (QPI and PCIe protocols disagree on P2P)
Topology Matters for P2P Communication

P2P Communication is **Not Supported** Between Bridges (*)

(*) “The IOH does not support non-contiguous byte enables from PCI Express for remote peer-to-peer MMIO transactions. This is an additional restriction over the PCI Express standard requirements to prevent incompatibility with Intel QuickPath Interconnect.“

(http://www.intel.com/Assets/PDF/datasheet/321328.pdf)
Topology Matters

PCI-e Switches: Fully Supported

Best P2P Performance Between GPUs on the Same PCIe Switch

P2P Communication Supported Between GPUs on the Same IOH
How Does P2P Memcopy Help Multi-GPU?

- **Ease of programming**
  - No need to manually maintain memory buffers on the host for inter-GPU exchanges

- **Increased throughput**
  - Especially when communication path does not include IOH (GPUs connected to a PCIe switch):
    - Single-directional transfers achieve up to ~6.3 GB/s
    - Duplex transfers achieve ~12.2 GB/s

- GPU-pairs can communicate concurrently if paths don’t overlap
Peer-to-Peer Throughputs

Via PCIe switch:
- GPUs attached to the same PCIe switch
  - Simplex: 6.3 GB/s
  - Duplex: 12.2 GB/s

Via IOH chip:
- GPUs attached to the same IOH chip
  - Simplex: 5.3 GB/s
  - Duplex: 9.0 GB/s

Via host:
- GPUs attached to different IOH chips
  - Simplex: 2.2 GB/s
  - Duplex: 3.9 GB/s
P2P transfer between multiple CPU processes

CUDA 4.1 introduces a new family of functions called cudalpc*:
- Create a handle to a GPU device memory segment which can be exported to other processes within a node
- API functions also provide an Ipc mechanism for passing events between processes

MVAPICH2-1.8a2 is already supporting this efficient GPU-GPU transfer within a node

IPC functionality is restricted to devices with support for unified addressing on Linux operating systems
Communication for Multiple Hosts, Multiple GPUs
Communication Between GPUs in Different Nodes

- GPUs in different network nodes
  - Require network communication
  - Currently require transferring GPU data to/from host CPU memory
  - Work underway to use P2P path for internode communications
  - Efforts underway to make MPI aware of GPUs (see later slides)

- With CUDA 4.0, transparent interoperability between CUDA pinned memory and Infiniband
  - export CUDA_NIC_INTEROP=1

- With CUDA 4.1 default behavior

- Do NOT use old GPU Direct kernel patch
Communication Between GPUs in Different Nodes

If each node also has multiple GPUs:
- Can continue using P2P within the node
- Can overlap some PCIe transfers with network communication (in addition to kernel execution)
GPU aware MPI Implementations
GPU-aware MPI

Support GPU to GPU communication through standard MPI interfaces without exposing low level details to the programmer

- e.g. enable MPI_Send, MPI_Recv from/to GPU memory
- Made possible by Unified Virtual Addressing (UVA) in CUDA 4.0
- MVAPICH2, OpenMPI, Platform MPI (Beta)

**Code without MPI integration**

At Sender:
```
cudaMemcpy(s_buf, s_device, size, cudaMemcpyDeviceToHost);
MPI_Send(s_buf, size, MPI_CHAR, 1, 1, MPI_COMM_WORLD);
```

At Receiver:
```
MPI_Recv(r_buf, size, MPI_CHAR, 0, 1, MPI_COMM_WORLD, &req);
cudaMemcpy(r_device, r_buf, size, cudaMemcpyHostToDevice);
```

**Code with MPI integration**

At Sender:
```
MPI_Send(s_device, size, …);
```

At Receiver:
```
MPI_Recv(r_device, size, …);
```
GPU-aware MVAPICH2

MVAPICH2 (1.8a2) provides optimized support for GPU to GPU communication through standard MPI interface
- Supports point to point and collective operations
- Pipelined data transfer which *automatically* provides optimizations
- Overlap CUDA copy and RDMA transfer
- Supports GPU Direct (P2P) and CUDA IPC

All possible combinations of source/target available:
- D2D, D2H, H2D
GPU-aware OpenMPI

- Transfer data directly to/from GPU memory via MPI calls
- Use of device pointers is supported in all of the send and receive APIs and most of the collective APIs
- Neither the collective reduction APIs nor the one-sided APIs are currently supported
- Code is currently available in the Open MPI trunk, available at:
  - http://www.open-mpi.org/nightly/trunk (contributed by NVIDIA)
- More details in the Open MPI FAQ
  - Features: http://www.open-mpi.org/faq/?category=running#mpi-cuda-support
  - Build Instructions: http://www.open-mpi.org/faq/?category=building#build-cuda
Compiling CUDA + MPI

* .cu file
  - Kernels / wrappers for kernels
    nvcc -O3 -c app_cuda.cu

* .c file
  - Main program with MPI calls and CUDA runtime calls
    #include "cudaruntime.h"
    mpicc -O3 -c app.c -l/usr/local/cuda/include

Link
  mpicc -o app app.o app_cuda.o -L/usr/local/cuda/lib64 -lcudart
NUMA Considerations
NUMA and GPUs

- Host (CPU) NUMA affects PCIe transfer throughput in dual-IOH systems

- Transfers to “remote” GPUs achieve lower throughput
  - Additional QPI hops (This affects any PCIe device, not just GPUs (eg. network cards))

- When possible, lock CPU threads to a socket that’s closest to the GPU’s IOH chip
  - For example, by using taskset, numactl, GOMP_CPU_AFFINITY, KMP_AFFINITY, etc.
    - taskset -c 2 gpuapp --device=0
    - numactl --physcpubind=2 yourapp --device=0
NUMA and GPUs

**C-function which uses sched_setaffinity() system function**

```c
#include <sched.h>
void set_cpu_affinity(int id)
{
    cpu_set_t mask;
    int gpu_table[3] = {2, 3, 5};
    /* Set the affinity for the GPU specified by id. */
    CPU_ZERO(&mask);
    CPU_SET(gpu_table[id], &mask);
    sched_setaffinity(0, sizeof(mask), &mask);
}
```

A sample usage:

```c
/* Select the CUDA GPU and set the CPU processor affinity. */
cudaSetDevice(dev);
set_cpu_affinity(dev);
```

- For some applications, it is not practical to restrict GPU to memory transfers to the memory of a single CPU socket
- Use alternatives like “numactl –interleave=all yourapp” to minimize the difference between using near memory and far memory

NUMA and GPUs

Developers should use hwloc when concerned about NUMA topologies

- Used by Open MPI, MVAPICH2, others
- Handles other PCIe devices too
- Built-in support to get topology information for CUDA devices
  - Interaction between hwloc and CUDA driver to get for instance a list of processors near NVIDIA GPUs
Cluster Solutions

Allinea and TotalView Cluster Debuggers

- Multi-GPU debugging support
- CUDA-MEMCHECK support for memory errors
- MPI and CUDA support for GPU clusters
- Breakpoints, thread control, and data evaluation

VAMPIR Cluster Profiler

- Visualization and Analysis of MPI + CUDA code.

Other profiler partners

TAU, PAPI, HPC-Toolkit
Summary

CUDA provides a number of features to facilitate multi-GPU programming

Single-process / multiple GPUs:
- Unified virtual address space
- Ability to directly access peer GPU’s data
- Ability to issue P2P memcopies
  - No staging via CPU memory
  - High aggregate throughput for many-GPU nodes

Multiple-processes:
- GPU Direct to maximize performance when both PCIe and IB transfers are needed

Streams and asynchronous kernel/copies
- Allow overlapping of communication and execution

Keep NUMA in mind on multi-IOH systems
Scalable Cluster Computing with NVIDIA GPUs
Axel Koehler - NVIDIA (akoehler@nvidia.com)