

# Multi-GPU Programming

Paulius Micikevicius | NVIDIA

Supercomputing 2011

November 14, 2011

# Outline

- **Usecases and a taxonomy of scenarios**
- **Inter-GPU communication:**
  - Single host, multiple GPUs
  - Multiple hosts
- **Case study**
- **Additional considerations:**
  - Multiple GPUs, streams, and events
  - Effect of host IOH chips on inter-GPU communication

- **Why multi-GPU?**

- To further speedup computation
- Working set exceeds a single GPU's memory
- Having multiple GPUs per node improves perf/W
  - Amortize the CPU server cost among more GPUs
  - Same goes for the price

- **Inter-GPU communication may be needed**

- Two general cases:
  - GPUs within a single network node
  - GPUs across network nodes

# Taxonomy of Inter-GPU Communication Cases

		Network nodes	
		Single	Multiple
Single process	Single-threaded		N/A
	Multi-threaded		N/A
Multiple processes			



GPUs can communicate via P2P or shared host memory



GPUs communicate via host-side message passing

# Communication for Single Host, Multiple GPUs

# Managing multiple GPUs from a single CPU thread

- CUDA calls are issued to the current GPU
  - Exception: peer-to-peer memcopies
- `cudaSetDevice()` sets the current GPU
- Current GPU can be changed while async calls (kernels, memcopies) are running
  - It is also OK to queue up a bunch of async calls to a GPU and then switch to another GPU
  - The following code will have both GPUs executing concurrently:

```
cudaSetDevice( 0 );  
kernel<<<...>>>(...);  
cudaMemcpyAsync(...);  
cudaSetDevice( 1 );  
kernel<<<...>>>(...);
```

# Unified Addressing (CUDA 4.0 and later)

- **CPU and GPU allocations use unified virtual address space**
  - Think of each one (CPU, GPU) getting its own range of a single VA space
    - Thus, driver/device can determine from an address where data resides
    - A given allocation still resides on a single device (an array doesn't span several GPUs)
  - 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:**
  - `cudaDeviceEnablePeerAccess( peer_device, 0 )`
    - Enables current GPU to access addresses on `peer_device` GPU
  - `cudaDeviceCanAccessPeer( &accessible, dev_X, dev_Y )`
    - Checks whether `dev_X` can access memory of `dev_Y`
    - Returns 0/1 via the first argument
    - 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



# Peer-to-peer memcopy

- `cudaMemcpyPeerAsync( void* dst_addr, int dst_dev, void* src_addr, int src_dev, size_t num_bytes, cudaStream_t stream )`
  - Copies the bytes between two devices
  - Currently performance is maximized when `stream` belongs to the source GPU
  - There is also a blocking (as opposed to Async) version
- **If peer-access is enabled:**
  - Bytes are transferred along the shortest PCIe path
  - No staging through CPU memory
- **If peer-access is not available**
  - CUDA driver stages the transfer via CPU memory

# How Does P2P Memcopy Help Multi-GPU?

- **Ease of programming**

- No need to manually maintain memory buffers on the host for inter-GPU exchanges

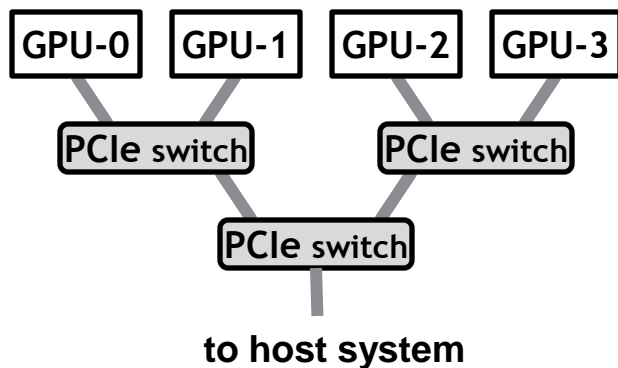
- **Increased throughput**

- Especially when communication path does not include IOH (GPUs connected to a PCIe switch):
  - Single-directional transfers achieve up to **~6.6 GB/s**
  - Duplex transfers achieve **~12.2 GB/s**
    - **4-5 GB/s** if going through the host
- GPU-pairs can communicate concurrently if paths don't overlap

# Example: 1D Domain Decomposition and P2P

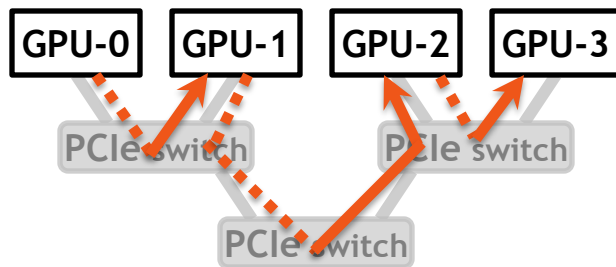
- **Each subdomain has at most two neighbors**
  - “left”/”right”
  - Communication graph = path
- **GPUs are physically arranged into a tree(s)**
  - GPUs can be connected to a PCIe switch
  - PCIe switches can be connected to another switch
- **A path can be efficiently mapped onto a tree**
  - Multiple exchanges can happen without contending for the same PCIe links
  - Aggregate exchange throughput:
    - Approaches (PCIe bandwidth) \* (number of GPU pairs)
    - Typical achieved PCIe gen2 simplex bandwidth on a single link: 6 GB/s

# Example: 4-GPU Topology

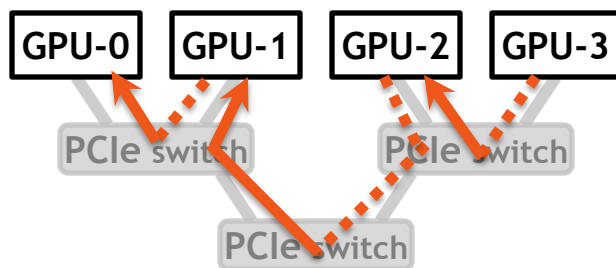


- **Two ways to implement 1D exchange**
  - Left-right approach
  - Pairwise approach
  - Both require two stages

# Example: Left-Right Approach for 4 GPUs



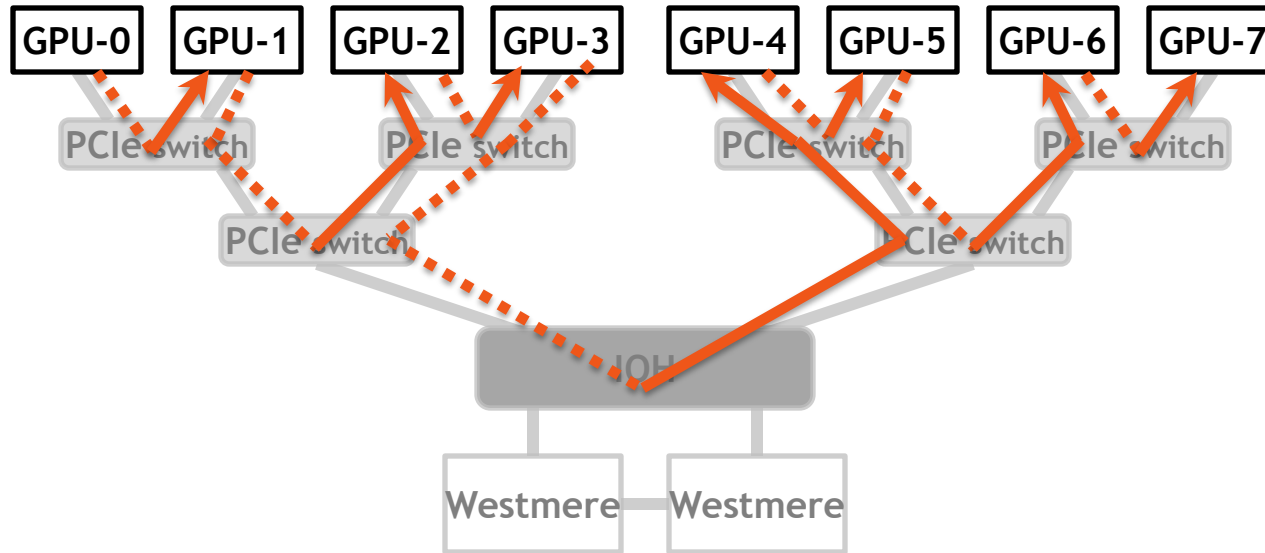
**Stage 1:** send “right” / receive from “left”



**Stage 2:** send “left” / receive from “right”

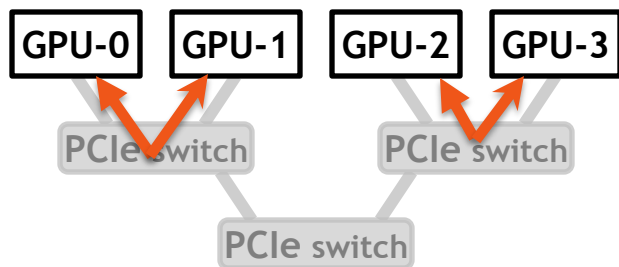
- **The 3 transfers in a stage happen concurrently**
  - Achieved throughput: ~15 GB/s (4-MB messages)
- **No contention for PCIe links**
  - PCIe links are duplex
  - Note that no link has 2 communications in the same “direction”

# Example: Left-Right Approach for 8 GPUs

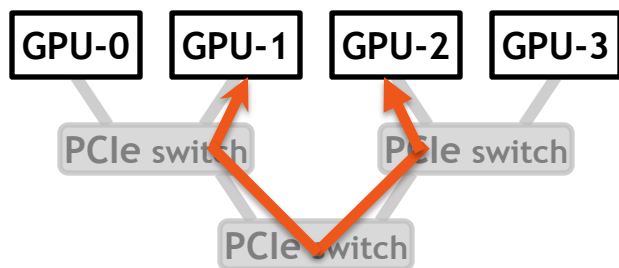


- Stage 1 shown above (Stage 2 is basically the same)
- Achieved aggregate throughput: ~34 GB/s

# Example: Pairwise Approach for 4 GPUs



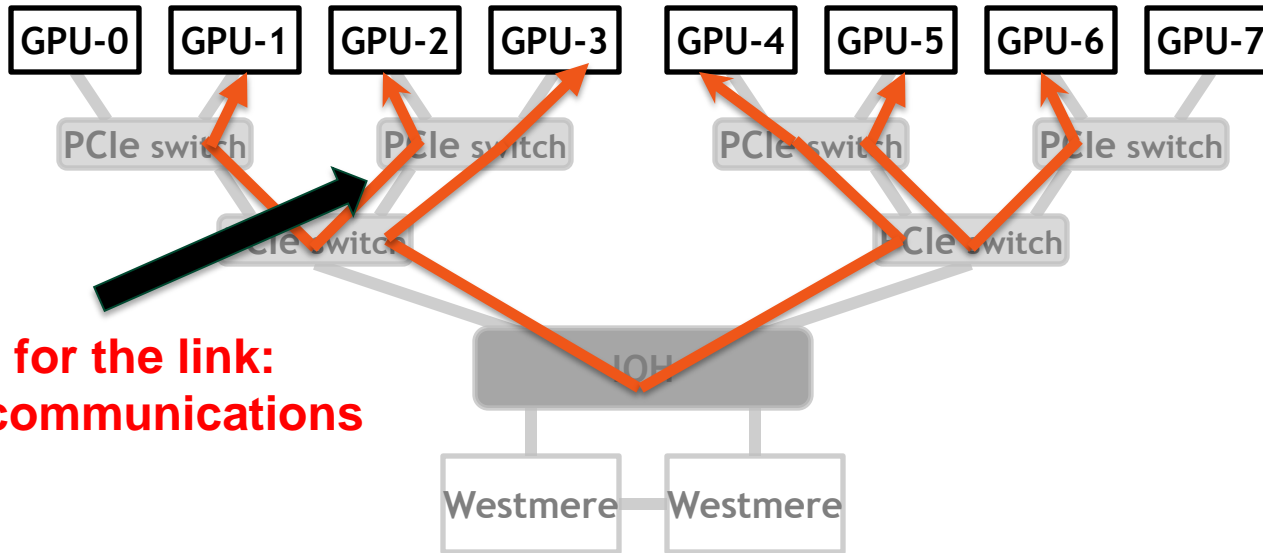
**Stage 1:** even-odd pairs



**Stage 2:** odd-even pairs

- **No contention for PCIe links**
  - All transfers are duplex, PCIe links are duplex
  - Note that no link has more than 1 exchange
    - Not true for 8 or more GPUs

# Example: Even-Odd Stage of Pairwise Approach for 8 GPUs



**Contention for the link:  
2 duplex communications**

- **Odd-even stage:**
  - Will always have contention for 8 or more GPUs
- **Even-odd stage:**
  - Will not have contention



# 1D Communication

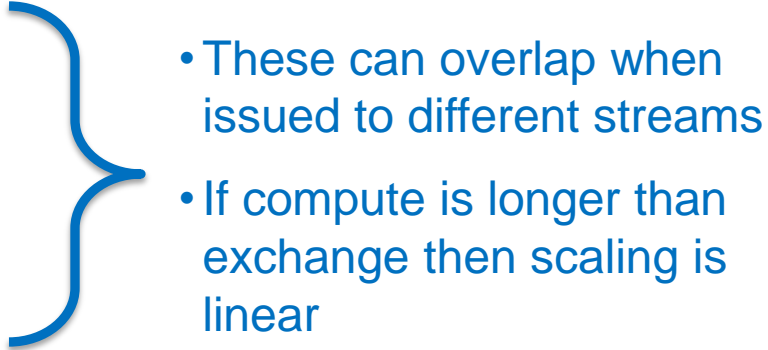
- Pairwise approach slightly better for 2-GPU case
- Left-Right approach better for the other cases

# Code for the Left-Right Approach

```
for( int i=0; i<num_gpus-1; i++ )// “right” stage
    cudaMemcpyPeerAsync( d_a[i+1], gpu[i+1], d_a[i], gpu[i], num_bytes, stream[i] );
for( int i=0; i<num_gpus; i++ )
    cudaStreamSynchronize( stream[i] );
for( int i=1; i<num_gpus; i++ ) // “left” stage
    cudaMemcpyPeerAsync( d_b[i-1], gpu[i-1], d_b[i], gpu[i], num_bytes, stream[i] );
```

- **Code assumes that addresses and GPU IDs are stored in arrays**
- **The middle loop isn't necessary for correctness**
  - Improves performance by preventing the two stages from interfering with each other (15 vs 11 GB/s for the 4-GPU example)

# Possible Pattern for Multi-GPU Code

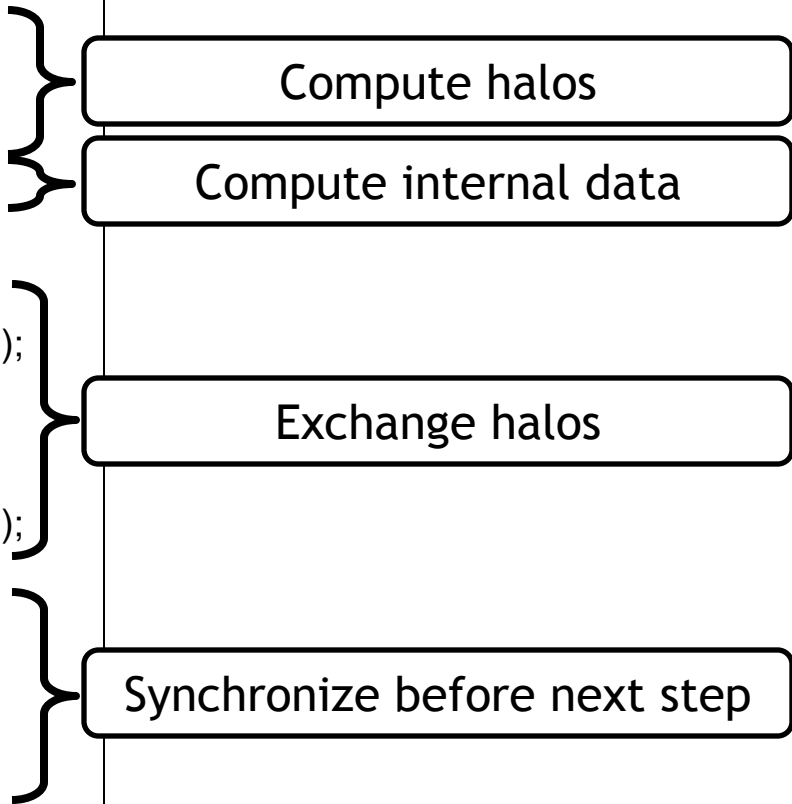
- **Stage 1:**
    - Compute halos (data to be shared with other GPUs)
  - **Stage 2:**
    - Exchange data with other GPUs
      - Use asynchronous copies
    - Compute over internal data
  - **Synchronize**
- 
- These can overlap when issued to different streams
  - If compute is longer than exchange then scaling is linear

# Code Pattern

```
for( int istep=0; istep<nsteps; istep++)
{
    for( int i=0; i<num_gpus; i++ )
    {
        cudaSetDevice( gpu[i] );
        kernel<<<..., stream_halo[i]>>>( ... );
        kernel<<<..., stream_halo[i]>>>( ... );
        cudaStreamQuery( stream_halo[i] );
        kernel<<<..., stream_internal[i]>>>( ... );
    }

    for( int i=0; i<num_gpus-1; i++ )
        cudaMemcpyPeerAsync( ..., stream_halo[i] );
    for( int i=0; i<num_gpus; i++ )
        cudaStreamSynchronize( stream_halo[i] );
    for( int i=1; i<num_gpus; i++ )
        cudaMemcpyPeerAsync( ..., stream_halo[i] );

    for( int i=0; i<num_gpus; i++ )
    {
        cudaSetDevice( gpu[i] );
        cudaDeviceSynchronize();
        // swap input/output pointers
    }
}
```



# Communication for Single Host, Multiple GPUs

# Communication Between GPUs in Different Nodes

- **Requires network communication**
  - Currently requires data to first be transferred to host
- **Steps for an exchange:**
  - GPU->CPU transfer
  - CPU exchanges via network
    - For example, MPI\_Sendrecv
    - Just like you would do for non-GPU code
  - CPU->GPU transfer
- **If each node also has multiple GPUs:**
  - Can continue using P2P within the node, netw outside the node
  - Can overlap some PCIe transfers with network communication
    - In addition to kernel execution

# Code Pattern

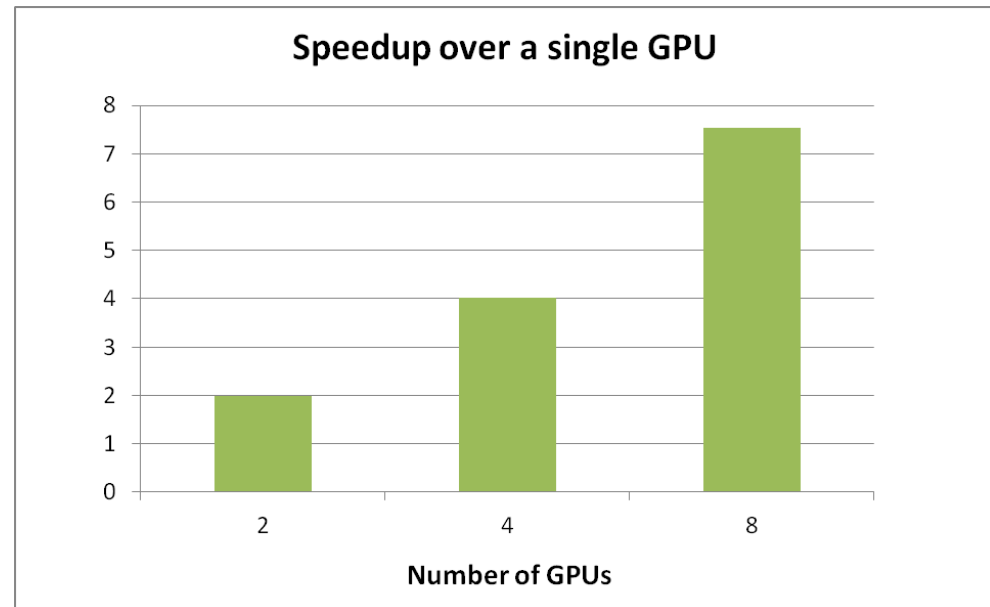
```
cudaMemcpyAsync( ..., stream_halo[i] );  
cudaStreamSynchronize( stream_halo[i] );  
MPI_Sendrecv( ... );  
cudaMemcpyAsync( ..., stream_halo[i] );
```

# Case Study



# Case Study: TTI RTM

- **TTI RTM**
  - One of the applications in seismic processing
  - 3DFD, 8<sup>th</sup> order in space, 2<sup>nd</sup> order in time
  - Regular grid
  - 1D domain decomposition
- **Data set:**
  - 512x512x512 cube
  - Requires ~7 GB working set
- **Experiments:**
  - Throughput increase over 1 GPU
  - Single node, 4-GPU “tree”



# Case Study: Time Breakdown

- **Single step (single 8-GPU node):**

- Halo computation: 1.1 ms
- Internal computation: 12.7 ms
- Halo-exchange: 5.9 ms
- Total: **13.8 ms**

- **Communication is completely hidden**

- 12.7 ms for internal computation, 5.9 ms for communication
  - ~95% scaling: halo+internal: 13.8 ms (13.0 ms if done without splitting)
- Thus, plenty of time for slower communication (network)

# Case Study: Multiple Nodes

- **Test system:**

- 3 servers, each with 2 M2090 GPUs, Infiniband DDR interconnect

- **Performance:**

- 512x512x512 domain:

- 1 node x 2 GPUs: 1.98x
- 2 nodes x 1 GPU: 1.97x
- 2 nodes x 2 GPUs: 3.98x
- 3 nodes x 2 GPUs: 4.50x

- 768x768x768 domain:

- 3 nodes x 2 GPUs: 5.60x



**Communication takes longer than internal computation**

- **Test system:**

- Communication (PCIe and IB DDR2) is hidden when each GPU gets ~100 slices
  - Network is ~68% of all communication time
- IB QDR hides communication when each GPU gets ~70 slices

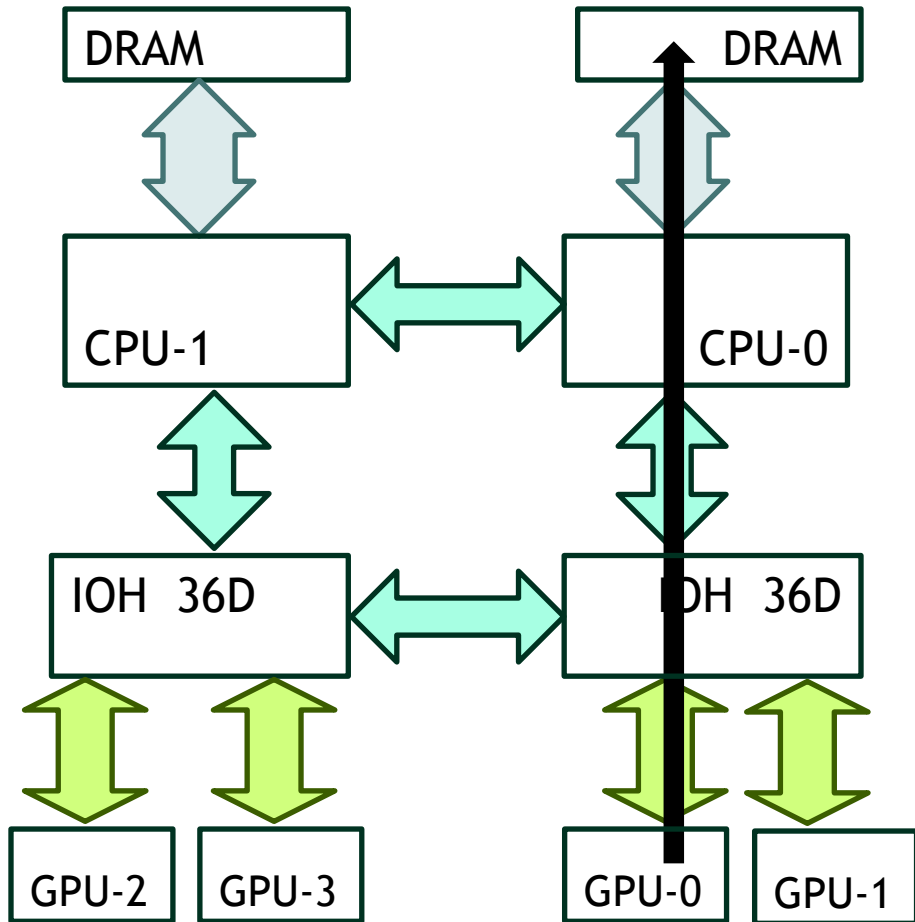
# Additional Considerations

CPU-GPU and GPU-GPU transfers in NUMA hosts  
Multi-GPU with Streams and Events

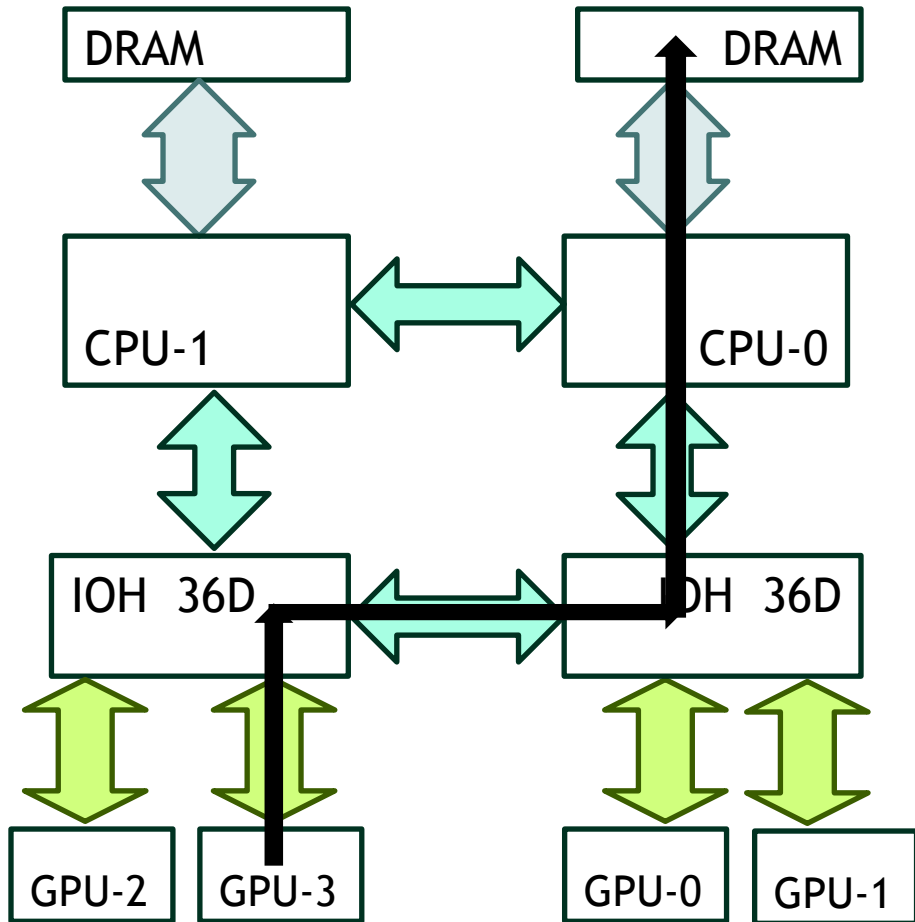
# Additional System Issues to Consider

- **Host (CPU) NUMA affects PCIe transfer throughput in dual-IOH systems**
  - Transfers to “remote” GPUs achieve lower throughput
  - One additional QPI hop
  - This affects any PCIe device, not just GPUs
    - Network cards, for example
  - When possible, lock CPU threads to a socket that’s closest to the GPU’s IOH chip
    - For example, by using numactl, GOMP\_CPU\_AFFINITY, KMP\_AFFINITY, etc.
- **Dual-IOH systems prevent PCIe P2P across the IOH chips**
  - QPI link between the IOH chips isn’t compatible with PCIe P2P
  - P2P copies will still work, but will get staged via host memory
    - Lower throughput

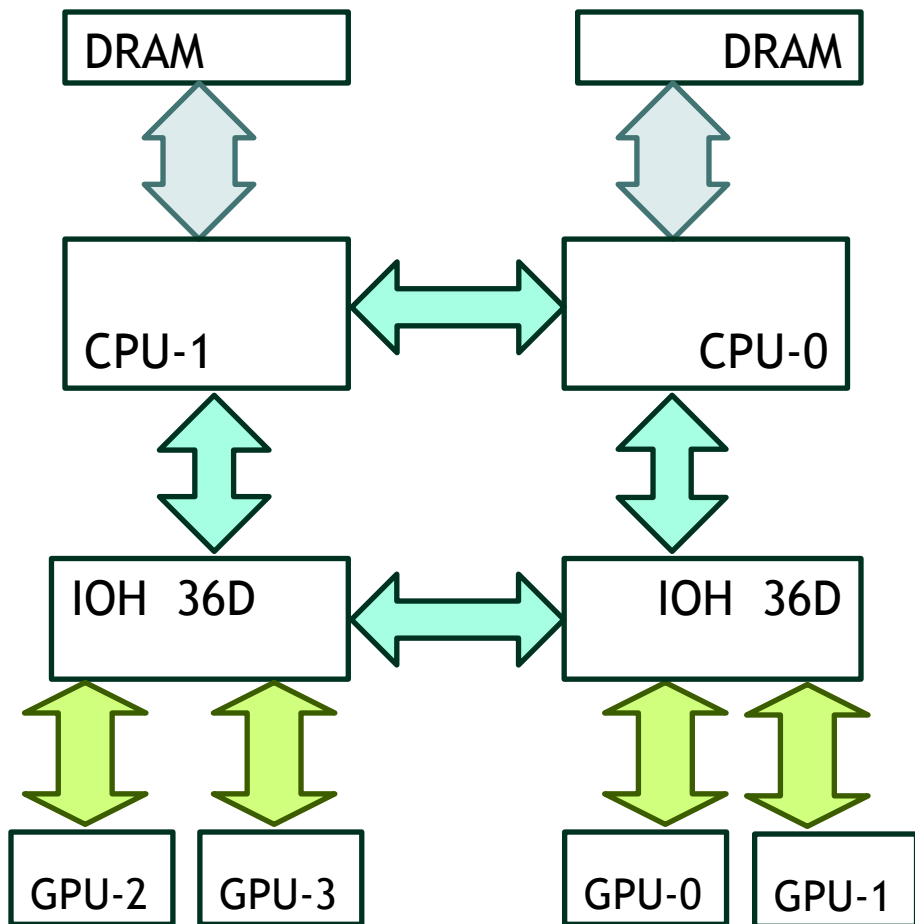
# “Local” D2H Copy: 6.3 GB/s



# “Remote” D2H Copy: 4.3 GB/s



# Summary of CPU-GPU Copy Throughputs on One System



- **Note that these vary among different systems**
  - Different BIOS settings
  - Different IOH chips
- **Local:**
  - D2H: 6.3 GB/s
  - H2D: 5.7 GB/s
- **Remote:**
  - D2H: 4.3 GB/s
  - H2D: 4.9 GB/s



# Summary of P2P 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

# Multi-GPU, Streams, and Events

# Multi-GPU, Streams, and Events

- **CUDA streams and events are *per device* (GPU)**
  - Determined by the GPU that's current at the time of their creation
  - Each device has its own *default* stream (aka 0- or NULL-stream)
- **Streams and:**
  - **Kernels:** can be launched to a stream only if the stream's GPU is current
  - **Memcopies:** can be issued to any stream
    - even if the stream doesn't belong to the current GPU
    - Driver will ensure that all calls to that stream complete before bytes are transferred
  - **Events:** can be recorded only to a stream if the stream's GPU is current
- **Synchronization/query:**
  - It is OK to query or synchronize with any event/stream
    - Even if stream/event does not belong to the current GPU

# Example 1

```
cudaStream_t streamA, streamB;  
cudaEvent_t eventA, eventB;  
  
cudaSetDevice( 0 );  
cudaStreamCreate( &streamA );  
cudaEventCreate( &eventA );  
  
cudaSetDevice( 1 );  
cudaStreamCreate( &streamB );  
cudaEventCreate( &eventB );  
  
kernel<<<..., streamB>>>(...);  
cudaEventRecord( eventB, streamB );  
  
cudaEventSynchronize( eventB );
```

// streamA and eventA belong to device-0

// streamB and eventB belong to device-1

**OK:**

- device 1 is current
- eventB and streamB belong to device 1

# Example 2

```
cudaStream_t streamA, streamB;  
cudaEvent_t eventA, eventB;  
  
cudaSetDevice( 0 );  
cudaStreamCreate( &streamA );  
cudaEventCreat( &eventA );  
  
cudaSetDevice( 1 );  
cudaStreamCreate( &streamB );  
cudaEventCreate( &eventB );  
  
kernel<<<..., streamA>>>(...);  
cudaEventRecord( eventB, streamB );  
  
cudaEventSynchronize( eventB );
```

// streamA and eventA belong to device-0

// streamB and eventB belong to device-1

## ERROR:

- device 1 is current
- streamA belongs to device 0

# Example 3

```
cudaStream_t streamA, streamB;  
cudaEvent_t eventA, eventB;  
  
cudaSetDevice( 0 );  
cudaStreamCreate( &streamA );  
cudaEventCreat( &eventA );  
  
cudaSetDevice( 1 );  
cudaStreamCreate( &streamB );  
cudaEventCreate( &eventB );  
  
kernel<<<..., streamB>>>(...);  
cudaEventRecord( eventA, streamB );
```

// streamA and eventA belong to device-0

// streamB and eventB belong to device-1

## ERROR:

- eventA belongs to device 0
- streamB belongs to device 1

# Example 4

```
cudaStream_t streamA, streamB;  
cudaEvent_t eventA, eventB;
```

```
cudaSetDevice( 0 );  
cudaStreamCreate( &streamA );  
cudaEventCreate( &eventA );
```

```
cudaSetDevice( 1 );  
cudaStreamCreate( &streamB );  
cudaEventCreate( &eventB );
```

```
kernel<<<..., streamB>>>( ... );  
cudaEventRecord( eventB, streamB );
```

```
cudaSetDevice( 0 );  
cudaEventSynchronize( eventB );  
kernel<<<..., streamA>>>( ... );
```

// streamA and eventA belong to device-0

// streamB and eventB belong to device-1

**device-1 is current**

**device-0 is current**

# Example 4

```
cudaStream_t streamA, streamB;  
cudaEvent_t eventA, eventB;
```

```
cudaSetDevice( 0 );  
cudaStreamCreate( &streamA );  
cudaEventCreate( &eventA );
```

// streamA and eventA belong to device-0

```
cudaSetDevice( 1 );  
cudaStreamCreate( &streamB );  
cudaEventCreate( &eventB );
```

// streamB and eventB belong to device-1

```
kernel<<<..., streamB>>>( ... );  
cudaEventRecord( eventB, streamB );
```

```
cudaSetDevice( 0 );  
cudaEventSynchronize( eventB );  
kernel<<<..., streamA>>>( ... );
```

## OK:

- device-0 is current
- synchronizing/querying events/streams of other devices is allowed



# Example 4

```
cudaStream_t streamA, streamB;  
cudaEvent_t eventA, eventB;
```

```
cudaSetDevice( 0 );  
cudaStreamCreate( &streamA );  
cudaEventCreate( &eventA );
```

// streamA and eventA belong to device-0

```
cudaSetDevice( 1 );  
cudaStreamCreate( &streamB );  
cudaEventCreate( &eventB );
```

// streamB and eventB belong to device-1

```
kernel<<<..., streamB>>>( ... );  
cudaEventRecord( eventB, streamB );
```

```
cudaSetDevice( 0 );  
cudaEventSynchronize( eventB );  
kernel<<<..., streamA>>>( ... );
```

## OK:

- device-0 is current
- synchronizing/querying events/streams of other devices is allowed
- here, device 0 won't start executing the kernel until device 1 finishes its kernel

# Example 5

```
int gpu_A = 0;
```

```
int gpu_B = 1;
```

```
cudaSetDevice( gpu_A );
```

```
cudaMalloc( &d_A, num_bytes );
```

```
int accessible = 0;
```

```
cudaDeviceCanAccessPeer( &accessible, gpu_B, gpu_A );
```

```
if( accessible )
```

```
{
```

```
    cudaSetDevice(gpu_B );
```

```
    cudaDeviceEnablePeerAccess( gpu_A, 0 );
```

```
    kernel<<<...>>>( d_A);
```

```
}
```

Even though kernel executes on gpu2, it will access (via PCIe) memory allocated on gpu1

# 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
  - Applies whether using single- or multiple threads to control GPUs
- **Keep NUMA in mind on multi-IOH systems**

# Questions?