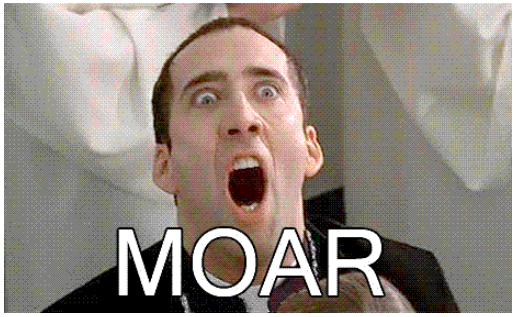
CMSC 691
High Performance Distributed Systems

Multi-GPU Computing

Dr. Alberto Cano
Assistant Professor
Department of Computer Science
acano@vcu.edu

Why multi-GPU?



- Further speedup computation
- Data exceeds single-GPU memory
- Multiple GPUs per node improves perf./watt

Inter GPU-communication

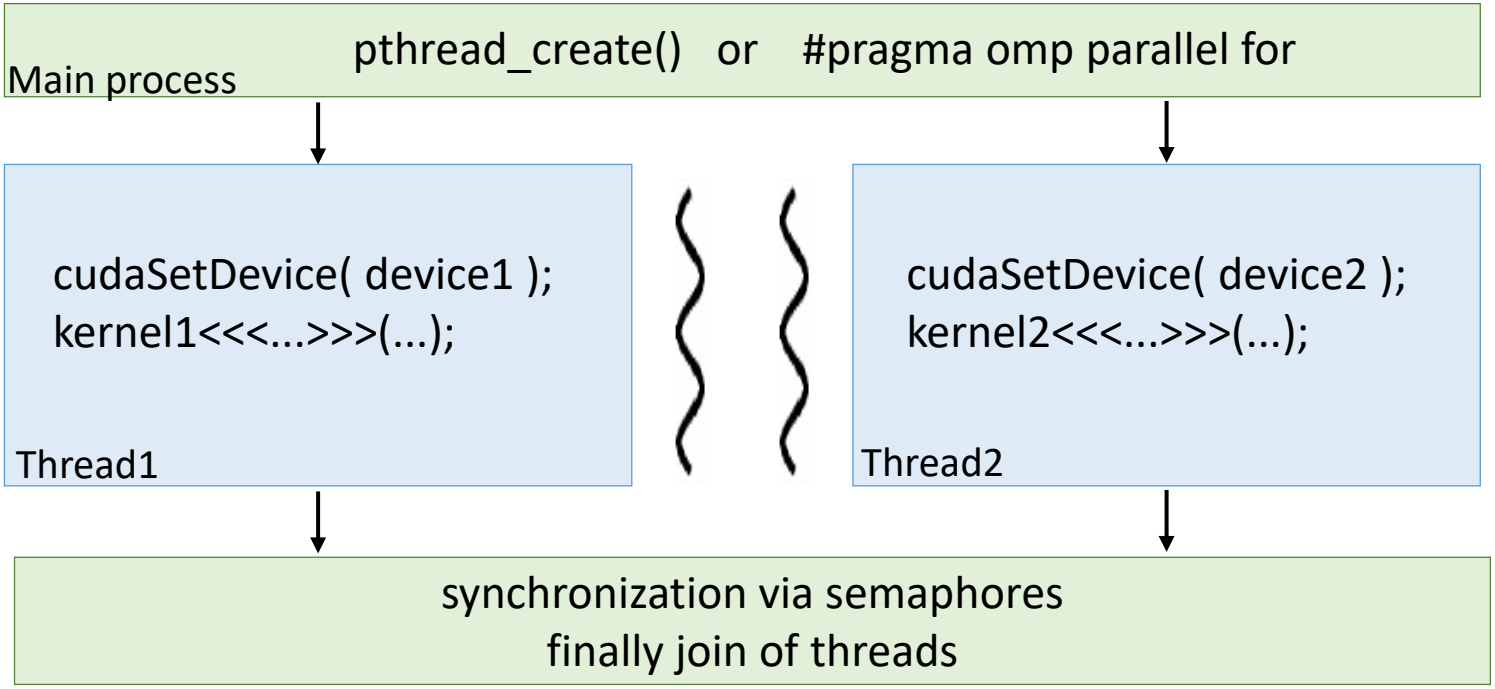
- GPUs in the same node can communicate via P2P addressing or shared host memory
- GPUs in different nodes communicate via host-side message passing (PCIe v3 x16 ~16 GB/s)
- Recently, NVIDIA NVLink provides P2P communication between a GPU and another GPU at a rate up to 80 GB/s.

Managing multiple GPUs in a node from a single CPU thread

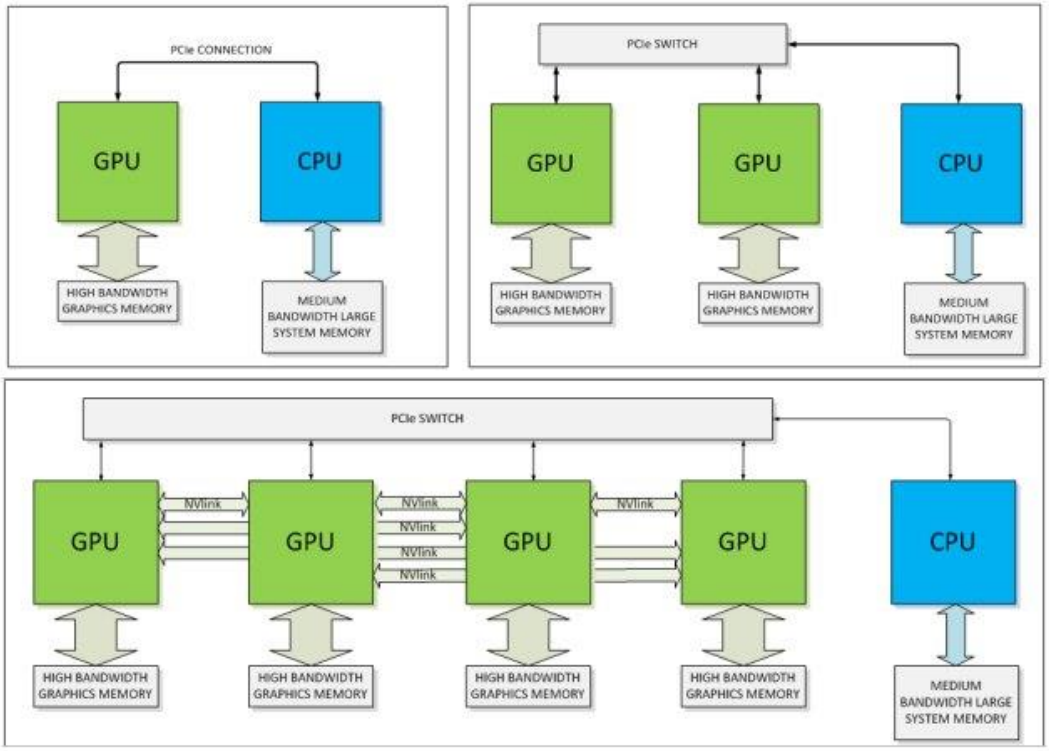
- CUDA call are issued to the current GPU (default GPUID0)
- *cudaSetDevice()* sets the current GPU
- *cudaGetNumDevices(&numDevices)* gets the number of devices
- Current GPU can be changed while async calls (kernels, memcopies) are running:
  - It is also OK to queue up a bunch of async calls to a GPU and then switch to another GPU
  - The following code will have both GPUs executing concurrently:

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

Managing multiple GPUs in a node from multiple CPU threads



Peer-to-peer memory access



- Allows data allocated in one GPU to be accessed from a kernel executing in other GPU
- Very large data latency!!

```
cudaSetDevice( 0 );
…
cudaMalloc(d_ptr, size);
…
cudaSetDevice( 1 );
…
kernel<<<...>>>(d_ptr);
```

Peer-to-peer memory access

- 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 architecture
  - GPUs are connected to different chips on the motherboard
  - QPI and PCIe protocols disagree on P2P

Peer-to-peer memory copy

- *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
- 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 ☹

Communication between GPUs in different nodes

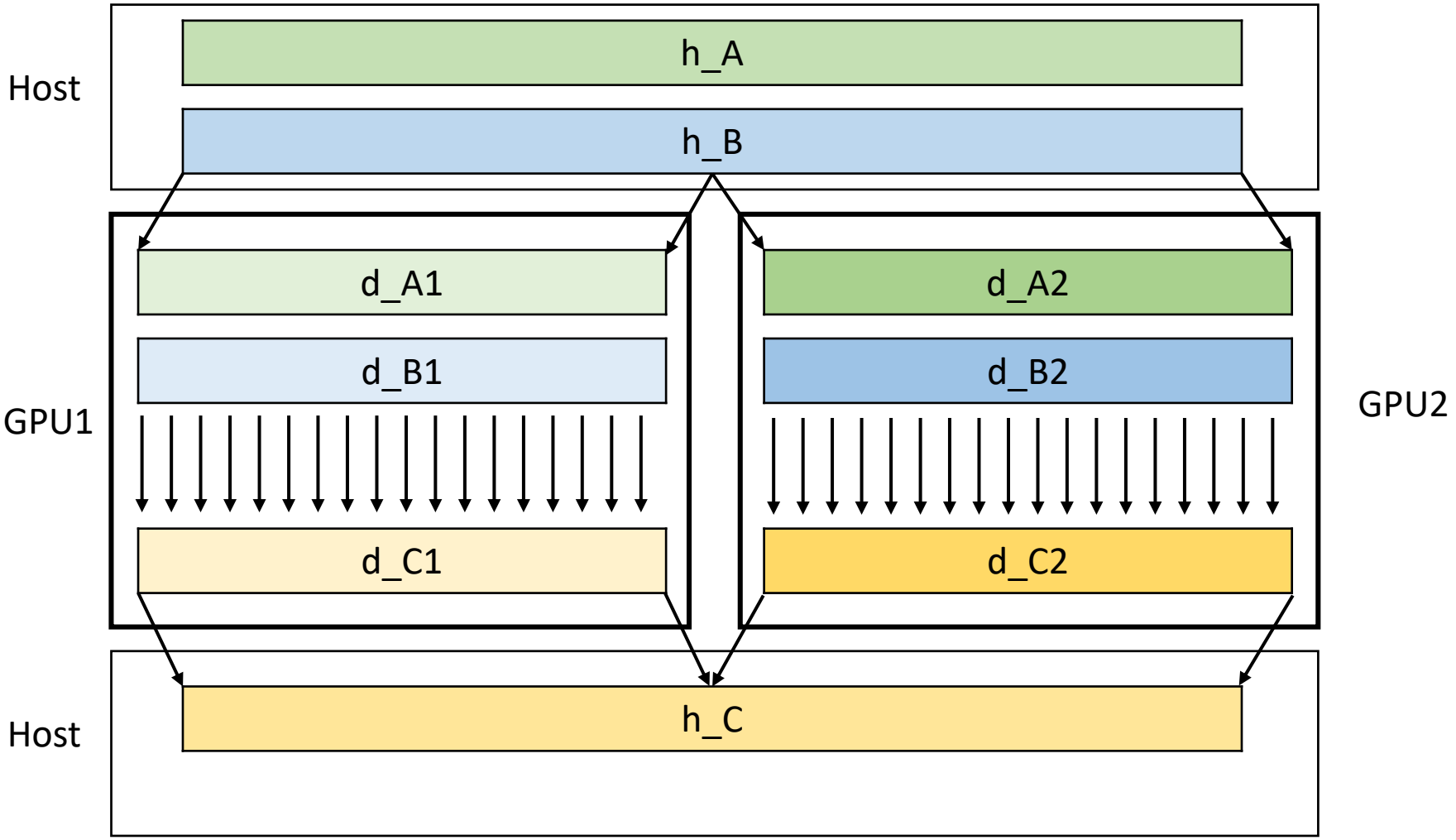
- Requires low-latency high-bandwidth network communication
- Steps for an exchange:
  1. GPU1->CPU1 transfer (PCIe limited)
  2. CPU2 exchanges via network with CPU2 (MPI?) (netw limited)
  3. CPU2->GPU2 transfer (PCIe limited)
- If each node also has multiple GPUs:
  - Can continue using P2P within the node
  - Can overlap some PCIe transfers with network communication
  - In addition to concurrent kernel execution

Using MPI for nodes communication

- Computational task is divided into many nodes
- Workload within a node is distributed among its GPUs
- Synchronization among the streams in a node
- Synchronization among the nodes in the cluster
- Use asynchronous non-blocking instructions when possible
- Code pattern:

```
cudaMemcpyAsync( ..., stream[i] );
cudaStreamSynchronize( stream[i] );
MPI_Send/recv( ... );
cudaMemcpyAsync( ..., stream[i] );
```

Example: vectorAdd using 2 GPUs



Exercises

- Multiple streams per GPU for the VectorAdd
- Multiple GPU contexts per physical GPU
- Decompose the matrix transpose into two GPUs
- Decompose the matrix multiplication into two GPUs
- Evaluate the performance penalty of P2P memory access

CMSC 691
High Performance Distributed Systems

Multi-GPU Computing

Dr. Alberto Cano
Assistant Professor
Department of Computer Science
acano@vcu.edu