Outline

• Brief review of the scenarios
• Single CPU process, multiple GPUs
  – GPU selection, UVA, P2P
• Multiple processes
  – Needs CPU-side message passing
• Dual-IOH CPU systems and NUMA
Several Scenarios

• We assume CUDA 4.0 or later
  – Simplifies multi-GPU programming

• Working set is decomposed across GPUs
  – Reasons:
    • To speedup computation
    • Working set exceeds single GPU’s memory
  – Inter-GPU communication is needed

• Two cases:
  – GPUs within a single network node
  – GPUs across network nodes
Multiple GPUs within a Node

• GPUs can be controlled by:
  – A single CPU thread
  – Multiple CPU threads belonging to the same process
  – Multiple CPU processes

• Definitions used:
  – CPU process has its own address space
  – A process may spawn several threads, which can share address space
Single CPU thread - Multiple GPUs

• All CUDA calls are issued to the current GPU
  – One exception: asynchronous peer-to-peer memcopies

• cudaSetDevice() sets the current GPU

• Asynchronous calls (kernels, memcopies) don’t block switching the GPU
  – The following code will have both GPUs executing concurrently:

```c
    cudaSetDevice( 0 );
    kernel<<<...>>>(...);
    cudaSetDevice( 1 );
    kernel<<<...>>>(...);
```
Devices, 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)

• Using streams and events
  – Calls to a stream can be issued only when its device is current
  – Event can be recorded only to a stream of the same device

• Synchronization/query:
  – It is OK to synchronize with or query any event/stream
    • Even if stream/event belong to one device and a different device is current
Example 1

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 );
cudaEventSynchronize( eventB );

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 ); // streamA and eventA belong to device-0
cudaEventCreate( &eventA );

cudaSetDevice( 1 );
cudaStreamCreate( &streamB ); // streamB and eventB belong to device-1
cudaEventCreate( &eventB );

kernel<<<..., streamA>>>(...);
cudaEventRecord( eventB, streamB );
cudaEventSynchronize( eventB );

**ERROR:**
- device-1 is current
- streamA belongs to device-0
Example 3

cudaStream_t streamA, streamB;
cudaEvent_t eventA, eventB;

cudaSetDevice( 0 );
cudaStreamCreate( &streamA );
cudaEventCreaet( &eventA );

// streamA and eventA belong to device-0

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

// streamB and eventB belong to device-1

kernel<<<..., streamB>>>(...);
cudaEventRecord( eventA, streamB );

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 );  // 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>>>(...);
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);

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
CUDA 4.0 and Unified Addressing

• CPU and GPU allocations use unified virtual address space
  – Think of each one (CPU, GPU) getting its own range of virtual addresses
    • Thus, driver/device can determine from the address where data resides
    • Allocation still resides on a single device (can’t allocate one array across 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)

• More details in the “GPU Direct and UVA” webinar
UVA and Multi-GPU Programming

• Two interesting aspects:
  – Peer-to-peer (P2P) memcopies
  – Accessing another GPU’s addresses

• Both require and peer-access to memory 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
Example 5

```c
int gpu1 = 0;
int gpu2 = 1;

cudaSetDevice( gpu1 );
cudaMalloc( &d_A, num_bytes );

int accessible = 0;
cudaDeviceCanAccessPeer( &accessible, gpu2, gpu1 );
if( accessible )
{
    cudaSetDevice( gpu2 );
    cudaDeviceEnablePeerAccess( gpu1, 0 );
    kernel<<<...>>>( d_A);
}
```

Even though kernel executes on gpu2, it will access (via PCIe) memory allocated on gpu1
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
  - `stream` must belong 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?

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

• **Performance**
  – Especially when communication path does not include IOH (GPUs connected to a PCIe switch):
    • Single-directional transfers achieve up to $\sim 6.6 \text{ GB/s}$
    • Duplex transfers achieve $\sim 12.2 \text{ GB/s}$
      – 4-5 GB/s if going through the host
  – Disjoint GPU-pairs can communicate without competing for bandwidth
Example 6

• 1D decomposition of data set, along the slowest varying dimension ($z$)

• GPUs have to exchange halos with their left/right neighbors

• 2-phase approach:
  – Each GPU sends data to the “right”
  – Each GPU sends data to the “left”
Example 6: one 8-GPU node configuration
Example 6: “Right” phase
Example 6: “Right” phase

Dashed lines: “down” direction of transfer on a PCIe link
Solid lines: “up” direction of transfer on a PCIe link
Example 6: “Right” phase

Dashed lines: “down” direction of transfer on a PCIe link
Solid lines: “up” direction of transfer on a PCIe link
There are no conflicts on the links - PCIe is duplex
All transfers happen simultaneously
Aggregate throughput: ~42 GB/s
Example 6: Code Snippet

```c
for( int i=0; i<num_gpus-1; i++ ) // “right” phase
cudaMemcpyPeerAsync( d_a[i+1], device[i+1], d_a[i], device[i], num_bytes, stream[i] );

for( int i=1; i<num_gpus; i++ )    // “left” phase
cudaMemcpyPeerAsync( d_b[i-1], device[i-1], d_b[i], device[i], num_bytes, stream[i] );
```
Example 6: Code Snippet

for( int i=0; i<num_gpus-1; i++ ) // “right” phase
cudaMemcpyPeerAsync( d_a[i+1], device[i+1], d_a[i], device[i], num_bytes, stream[i] );

for( int i=1; i<num_gpus; i++ ) // “left” phase
cudaMemcpyPeerAsync( d_b[i-1], device[i-1], d_b[i], device[i], num_bytes, stream[i] );

• Note that a device isn’t set prior to each copy
  – No need as async P2P memcopies use the source device

You may have to insert a device-synchronization between the phases:
• prevents the “last” device from sending in the “right” phase, which would cause link-contention
  results is correct, some performance is lost
• this can happen because all the calls above are asynchronous
Typical Pattern for Multi-GPU Code

• **Stage 1:**
  – Compute halos (data to be sent to other GPUs)

• **Stage 2:**
  – Exchange data with other GPUs
    • Use asynchronous copies
  – Compute over internal data

• **Synchronize**

• These will overlap when issued to different streams
• If compute is longer than exchange then scaling is linear
Summary for Single CPU-thread/multiple-GPUs

• CUDA calls are issued to the current GPU
  – Pay attention to which GPUs streams and events belong
• GPUs can access each other’s memory
  – Keep in mind that still at PCIe latency/bandwidth
• P2P memcopies between GPUs enable high aggregate throughputs
  – P2P not possible for GPUs connected to different IOH chips
• Try to overlap communication and computation
  – Issue to different streams
Multiple threads/processes

• **Multiple threads of the same process**
  – Communication is same as single-thread/multiple-GPUs

• **Multiple processes**
  – Processes have their own address spaces
    • No matter if they’re on the same or different nodes
  – Thus, some type of CPU-side message passing (MPI, ...) will be needed
    • Exactly the same as you would use on non-GPU code
Multiple-Processes

• Inter-GPU transfer pattern:
  – D2H memcopy
  – CPU-GPU message passing
  – H2D memcopy

• Pinned memory:
  – Both GPU and network transfers are fastest when operating with pinned CPU memory
    • Pinning prevents memory pages from being swapped out to disk
    • Enables DMA transfers by GPU or network card

• GPU direct:
  – Enables both NVIDIA GPUs and Infiniband devices to share pinned memory
    • Either can DMA from the same pinned memory region
    • Eliminates redundant CPU-CPU copies
  – More details in the “GPU Direct and UVA” webinar
# Summary of Cases

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

• Number of PCIe hops doesn’t seem to affect throughput
“Local” H2D Copy: 5.7 GB/s
“Remote” H2D Copy: 4.9 GB/s
“Local” D2H Copy: 6.3 GB/s
“Remote” H2D Copy: 4.3 GB/s
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 processes to control GPUs
• Keep NUMA in mind on multi-IOH systems
Questions?