Multi-GPU Programming

Paulius Micikevicius
Developer Technology, NVIDIA
Outline

• Usecases and a taxonomy of scenarios
• Inter-GPU communication:
  – Single host, multiple GPUs
  – Multiple hosts
• Case study
• Multiple GPUs, streams, and events
• Additional APIs:
  – GPU-aware MPI, cudalpc*
• NUMA effect on GPU-CPU 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 power among more GPUs
    • Same goes for the cost

• **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

<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>

- **GPUs can communicate via P2P or shared host memory**
- **GPUs communicate via host-side message passing**
Minimal Review of Streams and Async API
Overlap kernel and memory copy

• Requirements:
  – D2H or H2D memcopy from pinned memory
  – Device with compute capability ≥ 1.1 (G84 and later)
  – Kernel and memcopy in different, non-0 streams

• Code:
  cudaMemcpyAsync( dst, src, size, dir, stream1 );
  kernel<<<grid, block, 0, stream2>>>(...);
Streams and Async API

- **Default CUDA API:**
  - Kernel launches are asynchronous with CPU
  - Memcopies (D2H, H2D) block CPU thread until transfer completes
  - CUDA calls are serialized by the driver

- **Streams and async functions provide:**
  - Memcopies (D2H, H2D) asynchronous with CPU and GPU
  - Ability to concurrently execute a kernel, memcopies, and CPU code

- **Stream: sequence of operations that execute in issue-order on GPU**
  - Operations from different streams may be interleaved
  - A kernel and memcopy from different streams can be overlapped
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
  – 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
    - Driver/GPU can determine from an address where data resides
    - An allocation 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:
  – cudaMemcpyPeer( peer_device, 0 )
    • Enables current GPU to access addresses on peer_device GPU
  – cudaMemcpyPeer( &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 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 data is “pushed”: source GPU’s DMA engine carries out the copy
  - 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 $\sim 6.6 \text{ GB/s}$ ($\sim 12 \text{ GB/s}$ for gen3)
    - Duplex transfers achieve $\sim 12.2 \text{ GB/s}$ ($\sim 22 \text{ GB/s}$ for gen3)
      - $\sim 5 \text{ GB/s}$ 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 \((\text{PCIe bandwidth}) \times (\text{number of GPU pairs})\)
    • 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

- 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”

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

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

© 2012, NVIDIA
Example: Even-Odd Stage of Pairwise Approach for 8 GPUs

- **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

```c
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
  • Scaling is linear if compute is longer than exchange
Example: Two Subdomains
Example: Two Subdomains

Phase 1

GPU-0: green subdomain
GPU-1: grey subdomain
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++)
        cudaMemcpyPeerAsync( ..., 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] );
        cudaMemcpyPeerAsync( ..., stream_halo[i] );
        cudaMemcpyPeerAsync( ..., stream_halo[i] );
        cudaMemcpyPeerAsync( ..., stream_halo[i] );
        cudaStreamSynchronize( stream_halo[i] );
        cudaDeviceSynchronize();
        // swap input/output pointers
    }
}
Communication for Multiple 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] );
Overlapping MPI and PCIe Transfers

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

for( int i=0; i<num_gpus; i++ )
    cudaMemcpyPeerAsync( ..., stream_halo[i] );
cudaSetDevice( gpu[0] );
cudaMemcpyAsync( ..., stream_halo[0] );
MPI_Sendrecv( ... );

for( int i=1; i<num_gpus; i++ )
    cudaMemcpyPeerAsync( ..., stream_halo[i] );
cudaSetDevice( gpu[0] );
cudaMemcpyAsync( ..., stream_halo[0] );
MPI_Sendrecv( ... );

cudaSetDevice( gpu[num_gpus-1] );
cudaMemcpyAsync( ..., stream_halo[num_gpus-1] );
Case Study
Case Study: TTI FWM

- **TTI Forward Wave Modeling**
  - Fundamental part of TTI RTM
  - 3DFD, 8\textsuperscript{th} order in space, 2\textsuperscript{nd} 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 computation into halo and internal)
  - 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

• **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
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
cudaStream_t streamA, streamB;
cudaEvent_t eventA, eventB;

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

cudaEventRecord( eventB, streamB );

cudaSetDevice( 0 );
cudaEventSynchronize( eventB );

// streamA and eventA belong to device-0

// streamB and eventB belong to device-1

device-1 is current

device-0 is current
cudaStream_t streamA, streamB;
cudaEvent_t eventA, eventB;

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

// streamA and eventA belong to device-0

// streamB and eventB belong to device-1

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

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 );

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

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 );
kernlelel<<<...>>>( d_A );
}
Additional APIs Useful for Multi-GPU
cudalpc* API

• Processes on the same host can access each others’ GPU memory
  – Example use: bypass communication via host with MPI, use P2P
    • MPI ranks on the same node transfer directly to each other’s GPU
    • As opposed to with MPI copy first, then copying to GPU

• Approach:
  – Process A gets a handle to its pointer, sends it to process B
  – Process B opens the handle: gets a pointer to A’s address
  – Process B (or its GPU kernels) uses the pointer
  – Process B closes the handle

Process A:
cudalpcMemHandle_t  handle_a;
cudalpcGetMemHandle( &handle_a, (void*)d_a );

Process B:
cudalpcOpenMemHandle( (void**)&d_neighbor, neighbors_a, cudalpcMemLazyEnablePeerAccess );
  // use d_neighbor like you would a locally allocated pointer
  cudalpcCloseMemHandle( d_neighbor );
GPU-Aware MPI

• MPI calls can take GPU pointers
  – mvapich, openmpi
  – Works with C/C++, Fortran, CUDA C, CUDA Fortran, directives-based code

• Benefits:
  – Simplifies code (no need to explicitly copy GPU<->CPU)
  – Can pipeline transfers for better performance:
    • Break the transfer into smaller pieces
    • Pipeline the transfer of pieces: overlap PCIe and Netw for all but the first and last piece

• Not yet available:
  – P2P path when MPI ranks are on the same node
Host (CPU) NUMA and CPU/GPU Transfers
Additional System Issues to Consider

• **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
    • 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
“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, PCIe gen2

- **Via PCIe switch:**
  - GPUs attached to the same PCIe switch
  - Simplex: 6.3 GB/s (12 GB/s gen3)
  - Duplex: 12.2 GB/s (22 GB/s gen3)

- **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
Determining Topology/Locality of a System

• Hardware Locality tool:
  – http://www.open-mpi.org/projects/hwloc/
  – Cross-OS, cross-platform
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?