PMPP 2015/16

TECHNISCHE UNIVERSITÄT DARMSTADT

Performance Tuning (3)

Course Schedule

TECHNISCHE UNIVERSITÄT DARMSTADT

12.10.2015	Introduction to PMPP
13.10.2015	Lecture CUDA Programming 1
19.10.2015	Lecture CUDA Programming 2
20.10.2015	Lecture CUDA Programming 3
26.10.2015	Lecture Parallel Basics, Exercise 1 assigned
27.10.2015	Questions and Answers (Q&A), S3|19, Room 2.8
2.11.2015	Intro Final Proj., Ex. 1 due, Ex. 2 assigned, Lecture PRAM
3.11.2015	Lecture PRAM (2)
9.11.2015	Final Projects assigned, L. Parallel Sort., Exercise 2 due
10.11.2015	Questions and Answers (Q&A)
16.11.2015	Questions and Answers (Q&A)
17.11.2015	Questions and Answers (Q&A)
23.11.2015	1st Status Presentation Final Projects
24.11.2015	1st Status Presentation Final Projects (continued)
30.11.2015	Lecture Design Patterns
1.12.2015	Questions and Answers (Q&A)

(Preliminary) Course Schedule

TECHNISCHE UNIVERSITÄT DARMSTADT

	7.12.2015	Lecture Design Patterns (2), Performance Tuning
	8.12.2015	Questions and Answers (Q&A)
	14.12.2015	Performance Tuning (2)
	15.12.2015	Questions and Answers (Q&A)
	11.1.2016	2nd Status Presentation Final Projects
	12.1.2016	2nd Status Presentation Final Projects (continued)
you are here →	18.1.2016	Performance Tuning (3)
	19.1.2016	
	25.1.2016	
	26.1.2016	
	1.2.2016	
	2.2.2016	
	8.2.2016	Final Presentation Final Projects
	9.2.2016	Final Presentation Final Projects (continued)

Overview

TECHNISCHE UNIVERSITÄT DARMSTADT

- Instruction Level Parallelism
- Identifying Performance Limiters
  - Instruction Limited Kernels
  - Latency Limited Kernels
  - Bandwidth Limited Kernels
  - Local Memory Optimizations
- Multi GPU Multi Threading
  - 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

Images and slides partially by developer.nvidia.com

Memory Bandwidth Limited Kernel

TECHNISCHE UNIVERSITÄT DARMSTADT

- We covered global memory, shared memory and access pattern already but there is one item left

- Additional “memories”
  - Local memory
    - Up next but we don’t really want to use it anyways
  - Texture and Constant
    - Read-only
    - Data resides in global memory
    - Read through different caches
      - Additional fast on-chip memory
      - Avoid polluting L1

Local Memory

TECHNISCHE UNIVERSITÄT DARMSTADT

- Name refers to memory where registers and other thread-data is spilled
  - Usually when one runs out of SM resources
  - “Local” because each thread has its own private area
- Details:
  - Not really a “memory” – bytes are stored in global memory
  - Differences from global memory:
    - Addressing is resolved by the compiler
    - Stores are cached in L1

LMEM Access Operation

TECHNISCHE UNIVERSITÄT DARMSTADT

- A store writes a line to L1
  - If evicted, that line is written to L2
  - The line could also be evicted from L2, in which case it's written to DRAM
- A load requests the line from L1
  - If a hit, operation is complete
  - If a miss, then requests the line from L2
    - If a miss, then requests the line from DRAM
- A store always happens before a load
  - Only GPU threads can access LMEM addresses

When is Local Memory Used?

TECHNISCHE UNIVERSITÄT DARMSTADT

- Register spilling
  - Fermi hardware limit is 63 registers per thread
  - Programmer can specify lower registers/thread limits:
    - To increase occupancy (number of concurrently running threads)
    - -maxrregcount option to nvcc, __launch_bounds__() qualifier in the code
  - LMEM is used if the source code exceeds register limit
- Arrays declared inside kernels, if compiler can't resolve indexing
  - Registers aren't indexable, so have to be placed in LMEM

How Does LMEM Affect Performance?

TECHNISCHE UNIVERSITÄT DARMSTADT

- It could hurt performance in two ways:
  - Increased memory traffic
  - Increased instruction count
- Spilling/LMEM usage isn't always bad
  - LMEM bytes can get contained within L1
    - Avoids memory traffic increase
  - Additional instructions don't matter much if code is not instruction-throughput limited

General Analysis/Optimization Steps

- Check for LMEM usage
  - Compiler output
    - nvcc option: –Xptxas –v,–abi=no
    - Will print the number of lmem bytes for each kernel (only if kernel uses LMEM)
  - Profiler
- Check the impact of LMEM on performance
  - Bandwidth-limited code:
    - Check how much of L2 or DRAM traffic is due to LMEM
  - Arithmetic-limited code:
    - Check what fraction of instructions issued is due to LMEM
- Optimize:
  - Try: increasing register count, increasing L1 size, using non-caching loads

Register Spilling: Analysis

TECHNISCHE UNIVERSITÄT DARMSTADT

- Profiler counters:
  - l1_local_load_hit, l1_local_load_miss, l1_local_store_hit, l1_local_store_miss
  - Counted for a single SM, incremented by 1 for each 128-byte transaction
- Impact on memory
  - Any memory traffic that leaves SMs (goes to L2) is expensive
  - L2 counters of interest: read and write sector queries
    - Actual names are longer, check the profiler documentation
    - Incremented by 1 for each 32-byte transaction
  - Compare:
    - Estimated L2 transactions due to LMEM misses in all the SMs
      - 2*(number of SMs)*4*l1_local_load_miss
        - 2: load miss implies a store happened first
        - Number of SMs: l1_local_load_miss counter is for a single SM
        - 4: local memory transaction is 128-bytes = 4 L2-transactions
    - Sum of L2 read and write queries (not misses)
- Impact on instructions
  - Compare the sum of all LMEM instructions to total instructions issued

When Register Spilling is Problematic

TECHNISCHE UNIVERSITÄT DARMSTADT

- Try increasing the limit of registers per thread
  - Use a higher limit in –maxrregcount, or lower thread count for __launch_bounds__
  - Likely reduces occupancy, potentially reducing execution efficiency
    - may still be an overall win – fewer total bytes being accessed
- Try using non-caching loads for global memory
  - nvcc option: -Xptxas –dlcm=cg
  - Potentially fewer contentions with spilled registers in L1
- Increase L1 size to 48KB
  - Default is 16KB L1, larger L1 increases the chances for LMEM hits
  - Can be done per kernel or per device:
    - cudaFuncSetCacheConfig(), cudaDeviceSetCacheConfig()

Case Study

TECHNISCHE UNIVERSITÄT DARMSTADT

- Time Domain Finite Difference of the 3D Wave Equation
  - Simulates seismic wave propagation through Earth subsurface
  - Largely memory bandwidth-bound
  - Running more threads concurrently helps saturate memory bandwidth
    - Thus, to run 1024 threads per Fermi SM we specify 32 register maximum per thread
- Check for LMEM Use
  - Spills 44 bytes per thread when compiled down to 32 registers per thread

```
$ nvcc -arch=sm_20 -Xptxas -v,-abi=no,-dlcm=cg fwd_o8.cu -maxrregcount=32
ptxas info : Compiling entry function '_Z15fwd_3D_orderX2bILi4ELi9EEvPfS0_S0_iiiii' for 'sm_20‘
ptxas info : Used 32 registers, 44+0 bytes lmem, 6912+0 bytes smem, 76 bytes cmem[0], …
```

Case Study: Impact on Memory

TECHNISCHE UNIVERSITÄT DARMSTADT

- Using profiler counters:
  - SM counters:
    - l1_local_load_miss: 564,332
    - l1_local_load_hit: 91,520
    - l1_local_store_miss: 269,215
    - l1_local_store_hit: 13,477
    - inst_issued: 20,412,251
  - L2 query counts:
    - Read: 99,435,608
    - Write: 33,385,908
    - Total: 132,821,516
- This was on a 16-SM GPU

Case Study: Impact on Memory

TECHNISCHE UNIVERSITÄT DARMSTADT

- Using profiler counters:
  - SM counters:
    - l1_local_load_miss: 564,332
    - l1_local_load_hit: 91,520
    - l1_local_store_miss: 269,215
    - l1_local_store_hit: 13,477
    - inst_issued: 20,412,251
  - L2 query counts:
    - Read: 99,435,608
    - Write: 33,385,908
    - Total: 132,821,516
- This was on a 16-SM GPU

Load L1 hit rate: 13.95%

Estimated L2 queries per SM due to LMEM:

2*4*564,332 = 4,514,656

Estimated L2 queries due to LMEM of all 16 SMs:

16*4,514,656 = 72,234,496

Percentage of all L2 queries due to LMEM:

72,234,496 / 132,821,516 = **53.38%**

Case Study: Impact on Memory

TECHNISCHE UNIVERSITÄT DARMSTADT

- Using profiler counters:
  - SM counters:
    - l1_local_load_miss: 564,332
    - l1_
    - l1_
    - l1_
    - ins
  - L2 que
    - Re
    - W
    - Tot
- This was on a 16-SM GPU

53.38% of memory traffic between the SMs and L2/DRAM is due to LMEM (not useful from the application's point of view).

Since application is bandwidth-limited, reducing spilling could help performance.

Load L1 hit rate: 13.95%

Estimated L2 queries per SM due to LMEM:

2*4*564,332 = 4,514,656

Estimated L2 queries due to LMEM of all 16 SMs:

16*4,514,656 = 72,234,496

Percentage of all L2 queries due to LMEM:

72,234,496 / 132,821,516 = **53.38%**

Case Study: Impact on Instructions

TECHNISCHE UNIVERSITÄT DARMSTADT

- Using profiler counters:
  - SM counters:
    - l1_local_load_miss: 564,332
    - l1_local_load_hit: 91,520
    - l1_local_store_miss: 269,215
    - l1_local_store_hit: 13,477
    - inst_issued: 20,412,251
  - L2 query counts:
    - Read: 99,435,608
    - Write: 33,385,908
    - Total: 132,821,516
- This was on a 16-SM GPU

Total instructions due to LMEM:

938,944

Percentage of instructions due to LMEM:

938,944 / 20,412,251 =

4.60%

Case Study: Impact on Instructions

- Using profiler counters:
  - SM counters:
    - l1_local_load_miss: 564,232
    - l1_
    - l1_
    - l1_
    - ins
  - L2 que
    - Re
    - W
    - Tot
- This was on a 16-SM GPU

4.6% is not significant enough to worry about

(Removing spilling completely cannot improve performance by more than 4.6%, and then only if kernel is instruction-limited)

Total instructions due to LMEM:

938,944

Percentage of instructions due to LMEM:

938,944 / 20,412,251 =

4.60%

Case Study: Optimizations

TECHNISCHE UNIVERSITÄT DARMSTADT

- Try increasing register count
  - Remove the –maxrregcount=32 compiler option
    - 46 registers per thread, no spilling
  - Performance improved by 1.22x
- Increase L1 cache size
  - Keeping the 32 register maximum and spilling 44 bytes
  - Add cudaDeviceSetCacheConfig(cudaFuncCachePreferL1); call
  - L1 LMEM load hit rate improved to 98.32%
  - Estimated 1.63% of all requests to L2 were due to LMEM
    - way too small to worry about
    - 1.63 was computed as on the previous slides (not by 100% - 98.32%)
  - performance improved by 1.45x
- Application was already using non-caching loads for other reasons

Spilling Review

TECHNISCHE UNIVERSITÄT DARMSTADT

- Doesn’t always decrease performance, but when it does it’s because of:
  - Increased pressure on the memory bus
  - Increased instruction count
- Use the profiler to determine:
  - Bandwidth-limited codes: LMEM L1 miss impact on memory bus (to L2) for
  - Arithmetic-limited codes: LMEM instruction count as percentage of all instructions
- Optimize by
  - Increasing register count per thread
  - Incresing L1 size
  - Using non-caching GMEM loads

Overview

TECHNISCHE UNIVERSITÄT DARMSTADT

- Instruction Level Parallelism
- Identifying Performance Limiters
  - Instruction Limited Kernels
  - Latency Limited Kernels
  - Bandwidth Limited Kernels
  - Local Memory Optimizations
- Multi GPU Multi Threading
  - 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

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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:

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

Devices, Streams, and Events

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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

TECHNISCHE UNIVERSITÄT DARMSTADT

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

cudaEventSynchronize( eventB );
```

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

Example 2

TECHNISCHE UNIVERSITÄT DARMSTADT

```
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

TECHNISCHE UNIVERSITÄT DARMSTADT

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

cudaEventSynchronize( eventB );
```

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

Example 4

TECHNISCHE UNIVERSITÄT DARMSTADT

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

cudaSetDevice( 0 );
cudaStreamCreate( &streamA );          // streamA and eventA belong to device-0
cudaEventCreaet( &eventA );

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

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

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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)

UVA and Multi-GPU Programming

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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

TECHNISCHE UNIVERSITÄT DARMSTADT

```
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

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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?

TECHNISCHE UNIVERSITÄT DARMSTADT

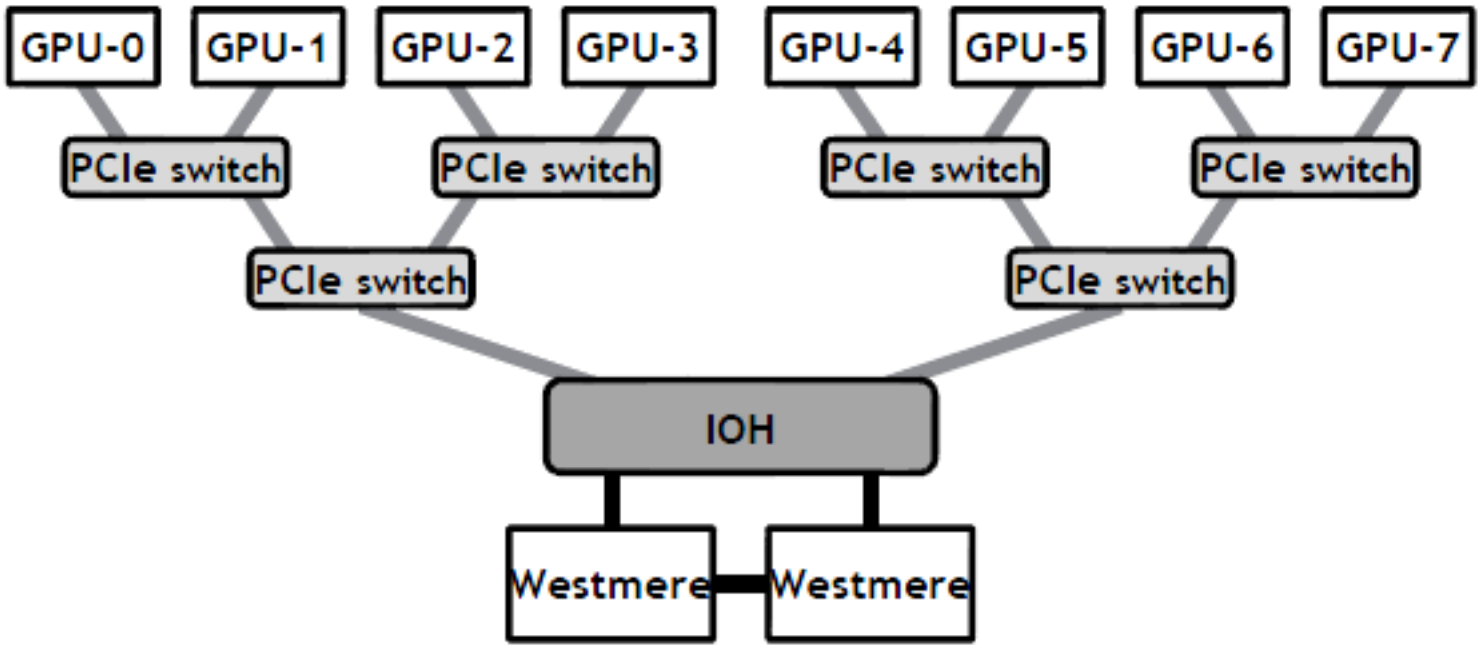
- Ease of programming
  - No need to manually maintain memory buffers on the host for inter-GPU exchanges
- Performance
  - Especially when communication path does not include IOH (GPUs connected to a PCIe switch):
    - Single-directional transfers achieve up to ~6.6 GB/s
    - Duplex transfers achieve ~12.2 GB/s
      - 4-5 GB/s if going through the host
  - Disjoint GPU-pairs can communicate without competing for bandwidth

Example 6

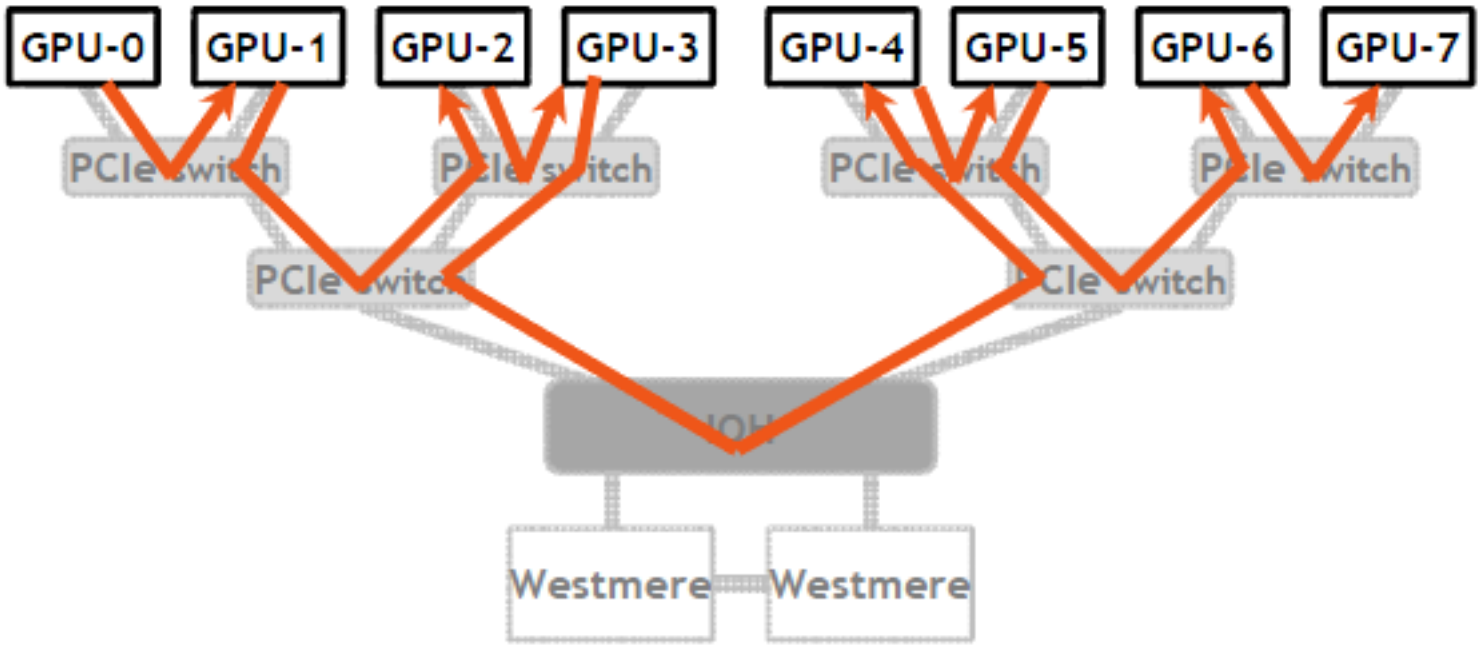
TECHNISCHE UNIVERSITÄT DARMSTADT

- 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”

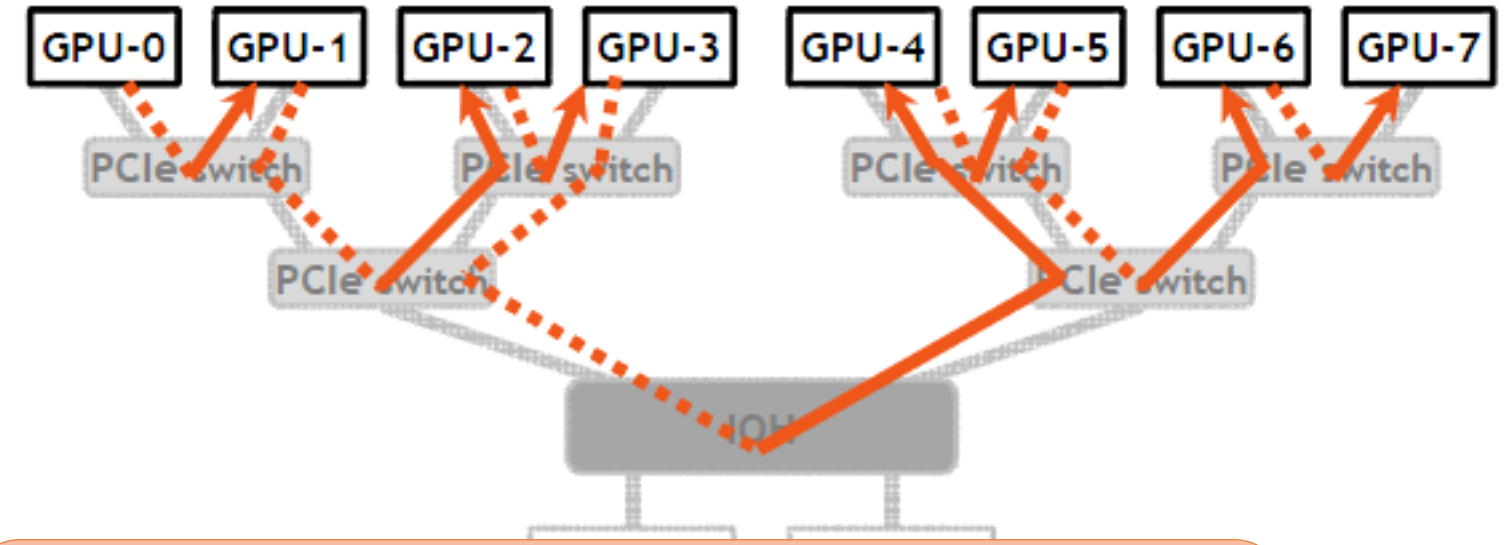
Example 6:
One 8-GPU Node Configuration



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

There are no conflicts on the links – PCIe is duplex

All transfers happen simultaneously

Aggregate throughput: ~42 GB/s

Example 6: Code Snippet

TECHNISCHE UNIVERSITÄT DARMSTADT

```
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

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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

Review Single CPU-thread/multiple-GPUs

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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

TECHNISCHE UNIVERSITÄT DARMSTADT

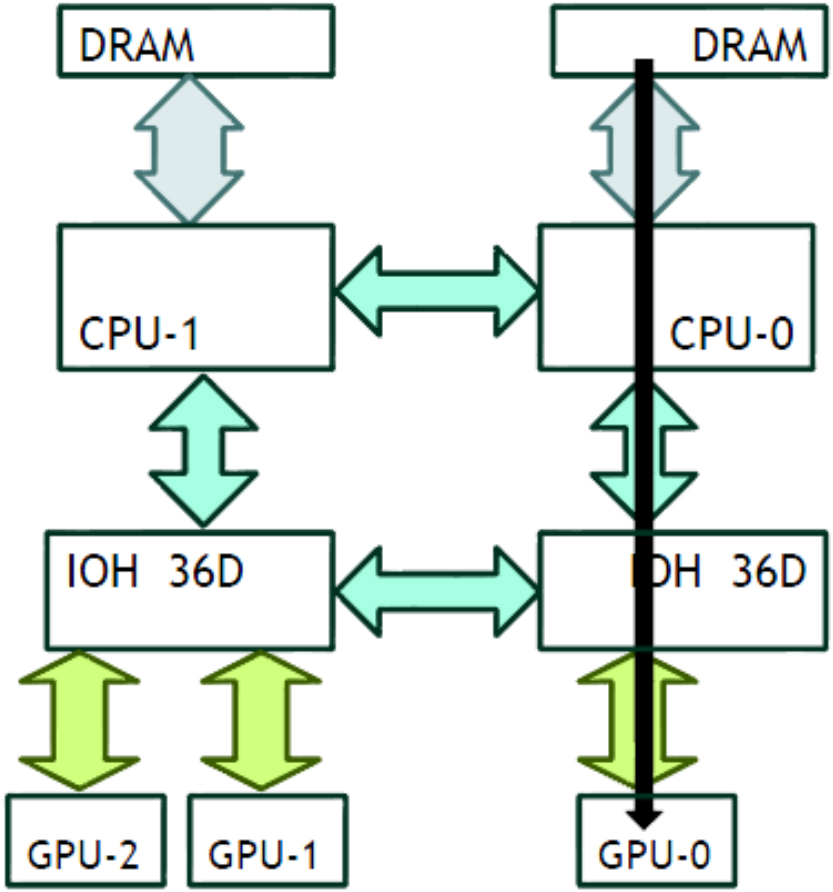
- Inter-GPU transfer pattern:
  - D2H memcopy
  - CPU-CPU message passing
  - H2D memcopy
- Pinned memory:
  - Both GPU and network transfers are fastest when operating with pinned CPU memory
    - Pinning prevents memory pages from being swapped out to disk
    - Enables DMA transfers by GPU or network 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

Additional System Issues to Consider

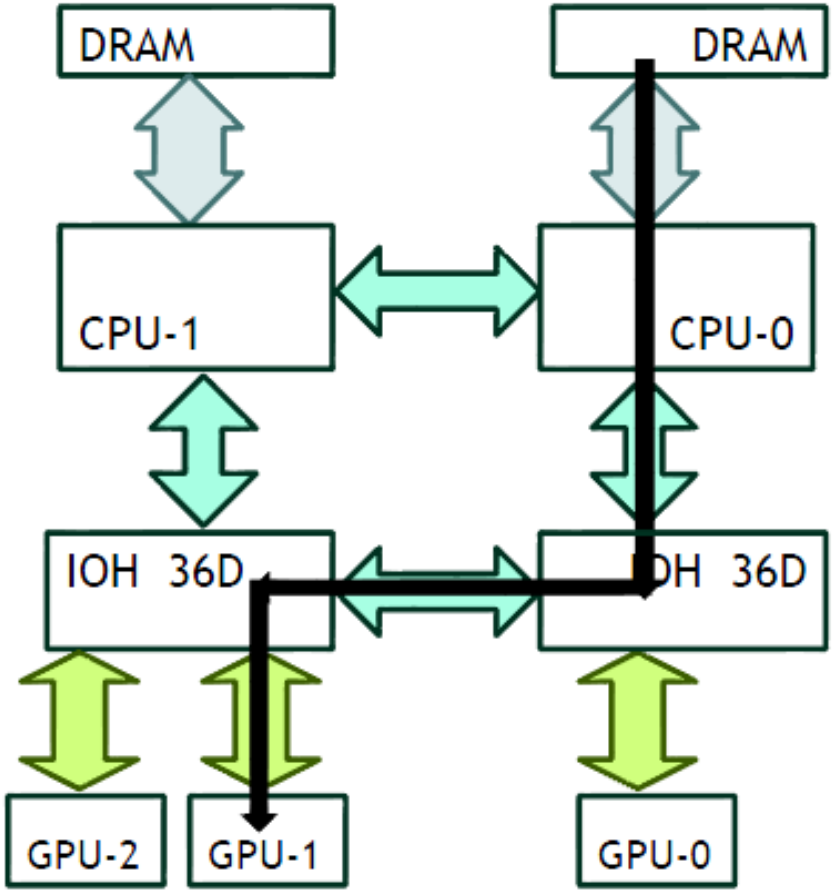
TECHNISCHE UNIVERSITÄT DARMSTADT

- 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

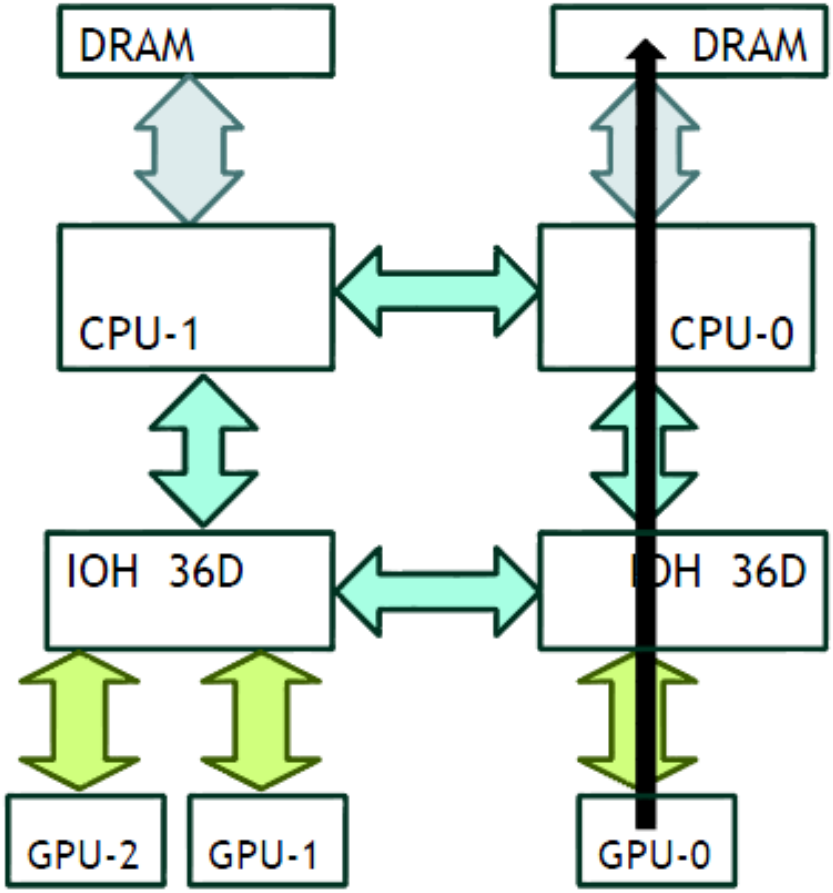
“Local” H2D Copy: 5.7 GB/s



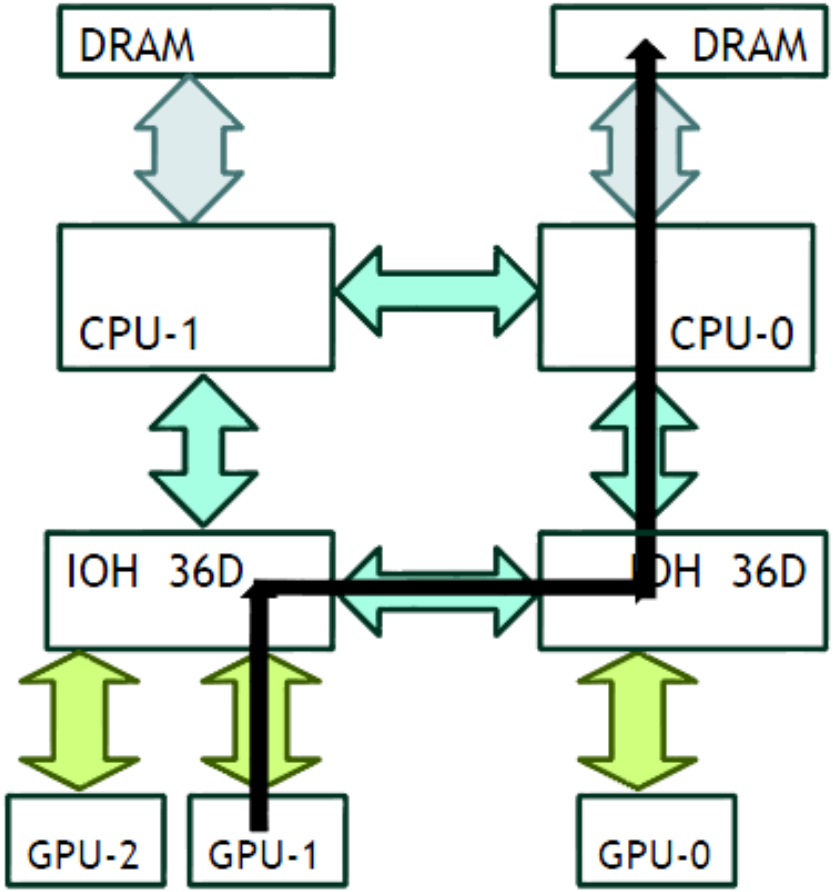
“Remote” H2D Copy: 4.9 GB/s



“Local” D2H Copy: 6.3 GB/s



“Remote” H2D Copy: 4.3 GB/s



Review Multi GPU

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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