ALL YOU NEED TO KNOW ABOUT PROGRAMMING NVIDIA’S DGX-2

Lars Nyland & Stephen Jones, GTC 2019
DGX-2: FASTEST COMPUTE NODE EVER BUILT

We’re here to tell you about it
NVIDIA® DGX-2™ SERVER AND NVSWITCH™

16 Tesla™ V100 32 GB GPUs
- FP64: 125 TFLOPS
- FP32: 250 TFLOPS
- Tensor: 2000 TFLOPS
- 512 GB of GPU HBM2

Single-Server Chassis
- 10U/19-Inch Rack Mount
- 10 kW Peak TDP
- Dual 24-core Xeon CPUs
- 1.5 TB DDR4 DRAM
- 30 TB NVMe Storage

12 NVSwitch Network
- Full-Bandwidth Fat-Tree Topology
- 2.4 TBps Bisection Bandwidth
- Global Shared Memory
- Repeater-less

New NVSwitch Chip
- 18 2nd Generation NVLink™ Ports
- 25 GBps per Port
- 900 GBps Total Bidirectional Bandwidth
- 450 GBps Total Throughput
USING MULTI-GPU MACHINES

What’s the difference between a mining rig and DGX-2?
SINGLE GPU

- CPU
- PCIe Bus
- PCIe I/O
- NVLink2
- L2
- XBAR
- HBM
- HBM
SINGLE GPU
SINGLE GPU
TWO GPUS

Can read and write each other’s memory over PCIe
TWO GPUS
Can read and write each other’s memory over PCIe
TWO GPUS USING NVLINK
6 Bidirectional Channels Directly Connecting 2 GPUs
TWO GPUS USING NVLINK
6 Bidirectional Channels Directly Connecting 2 GPUs
TWO GPUS USING NVLINK
6 Bidirectional Channels Directly Connecting 2 GPUs
MULTIPLE GPUS

Directly connected using NVLink2

- Requires dedicated connections between GPUs
- Decreases bandwidth between GPUs as more are added
- Not scalable
ADDING A SWITCH
DGX-2 INTERCONNECT INTRO
8 GPUs with 6 NVSwitch Chips
DGX-2 INTERCONNECT INTRO
8 GPUs with 6 NVSwitch Chips
DGX-2 INTERCONNECT INTRO

8 GPUs with 6 NVSwitch Chips
FULL DGX-2 INTERCONNECT

Baseboard to baseboard
MOVING DATA ACROSS THE NVLINK FABRIC

Bulk transfers

- Use copy-engine (DMA) between GPUs to move data
- Available with cudaMemcpy()

Word by word

- Programs running on SMs can access all memory by address
- For LOAD, STORE, and ATOM operations
  - 1 to 16 bytes per thread
- Similar performance guidelines for addressing coalescing apply
HOST MEMORY VIA PCIE

1.5 TB of host memory accessible via four PCIe channels

2 Intel Xeon 8168 CPUs
  ▶ 1.5 Terabytes DRAM

4 PCIe buses
  ▶ 2/CPU
  ▶ 4 GPUs/bus

GPUs can read/write host memory at 50 GB/s
User data moving at 49+ GB/s

1 kernel/GPU reading host memory

- On GPUs 0, 1, ..., 15
- 10 second delay between each launch

GPUs 0-3 share one PCIe bus

- Same for 4-7, 8-11, 12-15
MULTI-GPU PROGRAMMING IN CUDA

Programs control each device independently

- Streams are per-device work queues
- Launch & synchronize on a stream implies device

Inter-stream synchronization uses events

- Events can mark kernel completion
- Kernels queued in a stream on one device can wait for an event from another
// Create as many streams as I have devices
cudaStream_t stream[16];
for(int gpu=0; gpu<numGPUs; gpu++) {
    cudaSetDevice(gpu);
    cudaStreamCreate(&stream[gpu]);
}

// Launch a copy of the first kernel onto each GPU’s stream
for(int gpu=0; gpu<numGPUs; gpu++) {
    cudaSetDevice(gpu);
    firstKernel<<< griddim, blockdim, 0, stream[gpu] >>>( ... );
}

// Wait for the kernel to finish on each stream
for(int gpu=0; gpu<numGPUs; gpu++)
    cudaStreamSynchronize(stream[gpu]);

// Now launch a copy of another kernel onto each GPU’s stream
for(int gpu=0; gpu<numGPUs; gpu++) {
    cudaSetDevice(gpu);
    secondKernel<<< griddim2, blockdim2, 0, stream[gpu] >>>( ... );
}
BETTER ASYNCHRONOUS MULTI-GPU LAUNCH

Launch kernel on each GPU

GPU 0 waits for all kernels on all GPUs

Other GPUs wait for GPU 0

Synchronize all GPUs at end

// Launch the first kernel and an event to mark its completion
for(int gpu=0; gpu<numGPUs; gpu++) {
    cudaSetDevice(gpu);
    firstKernel<<<griddim, blockdim, 0, stream[gpu]>>>(...);
    cudaEventRecord(event[gpu], stream[gpu]);
}

// Make GPU 0 sync with other GPUs to know when all are done
for(int gpu=1; gpu<numGPUs; gpu++)
    cudaStreamWaitEvent(stream[0], event[gpu], 0);

// Then make other GPUs sync with GPU 0 for a full handshake
cudaEventRecord(event[0], stream[0]);
for(int gpu=1; gpu<numGPUs; gpu++)
    cudaStreamWaitEvent(stream[gpu], event[0], 0);

// Now launch the next kernel with an event... and so on
for(int gpu=0; gpu<numGPUs; gpu++) {
    cudaSetDevice(gpu);
    secondKernel<<<griddim, blockdim, 0, stream[gpu]>>>(...);
    cudaEventRecord(event[gpu], stream[gpu]);
}
MUCH SIMPLER: COOPERATIVE LAUNCH

Launch cooperative kernel

Program runs across all GPUs

Synchronize within GPU code

Exit when done

// Cooperative launch provides easy inter-grid sync.
// Kernels and per-GPU streams are in “launchParams”
cudaLaunchCooperativeKernelMultiDevice(launchParams, numGPUs);

// Now just synchronize to wait for the work to finish.
cudaStreamSynchronize(stream[0]);

// Inside the kernel, instead of kernels make function calls
غلل void masterKernel( ... ) {
  firstKernelAsFunction( ... );  // All threads on all GPUs run
  this_multi_grid().sync();      // Sync all threads on all GPUs

  secondKernelAsFunction( ... )
  this_multi_grid().sync();

  ...}

MULTI-GPU MEMORY MANAGEMENT

Unified Memory
Program spans all GPUs + CPUs

Individual memory
Independent instances read from neighbours explicitly
16 GPUs WITH 32GB MEMORY EACH

NVSWITCH PROVIDES

All-to-all high-bandwidth peer mapping between GPUs

Full inter-GPU memory interconnect (incl. Atomics)
UNIFIED MEMORY PROVIDES

- Single memory view shared by all GPUs
- User control of data locality
- Automatic migration of data between GPUs
WE WANT LINEAR MEMORY ACCESS
But cudaMalloc creates a partitioned global address space

4 GPUs require 4 allocations giving 4 regions of memory

Problem: Program must now be aware of data & compute layout across GPUs
UNIFIED MEMORY

CUDA’s Unified Memory Allows One Allocation To Span Multiple GPUs

Normal pointer arithmetic just works
SETTING UP UNIFIED MEMORY

```c
// Allocate data for cube of side “N”
float *data;
size_t size = N*N*N * sizeof(float);
cudaMallocManaged(&data, size);

// Make whole allocation visible to all GPUs
for(int gpu=0; gpu<numGPUs; gpu++) {
cudaMemAdvise(data, size,
cudaMemAdviseSetAccessedBy, gpu);
}

// Now place chunks on each GPU in a striped layout
float *start = ptr;
for(int gpu=0; gpu<numGPUs; gpu++) {
cudaMemAdvise(start, size / numGpus,
cudaMemAdviseSetPreferredLocation, gpu);
cudaMemPrefetchAsync(start, size / numGpus, gpu);
start += size / numGpus;
}
```
WEAK SCALING
Problem Grows As Processor Count Grows - Constant Work Per GPU
IDEAL WEAK SCALING

Weak Scaling solution time

Time to solution

Node Count

0 200 400 600 800 1000
0 20 40 60 80 100 120
STRONG SCALING
Problem Stays Same As Processor Count Grows - Less Work Per GPU
IDEAL STRONG SCALING

Local Strong Scaling solution time

Node Count

Time to solution
EXAMPLE: MERGESORT

Break list into 2 equal parts, recursively

- Until just 1 element per list

Merge pairs of lists

- Keeping them sorted
- Until just 1 list remains
PARALLELIZING MERGESORT

Parallelize merging of lists

Compute where elements are stored in result independently

For all items in list A
  ▶ Find lowest $j$ such that $A[i] < B[j]$
  ▶ Store $A[i]$ at $D[i+j]$

Binary search step impacts $O(\text{parallelism})$
  ▶ Extra $\log(n)$ work to perform binary search
OUTLINE OF PARALLEL MERGESORT

The “merge” step

1. Read by thread-id
2. Compute location in write-buffer
3. Write
4. Sync
5. Flip read, write buffers
6. Repeat, doubling list lengths
OUTLINE OF PARALLEL MERGESORT

The “merge” step

1. Read by thread-id
2. Compute location in write-buffer
3. Write
4. Sync
5. Flip read, write buffers
6. Repeat, doubling list lengths
OUTLINE OF PARALLEL MERGESORT

The “merge” step

1. Read by thread-id
2. Compute location in write-buffer
3. Write
4. Sync
5. Flip read, write buffers
6. Repeat, doubling list lengths
OUTLINE OF PARALLEL MERGESORT

The “merge” step

1. Read by thread-id
2. Compute location in write-buffer
3. Write
4. Sync
5. Flip read, write buffers
6. Repeat, doubling list lengths
ALIGNMENT OF THREADS & DATA

Memory pages are 2 MB

- Pinned to GPUs in round-robin fashion

Run $80 \times 1024 = 81920$ threads on each GPU

- One 8-byte read covers 655,360 bytes
- 16 GPUs cover 10,485,760 bytes

No optimized alignment between threads and memory

- Possible to do better (and worse)
ACHIEVED BANDWIDTH

Aggregate bandwidth for all loads and stores in mergesort

16 GPUs read & write data at 6 TB/s

Adding more GPUs
  ▶ Adds more accessible bandwidth
  ▶ Adds memory capacity

Is 6 TB/s fast enough?
  ▶ Speed of light is 2 TB/s for DGX-2
  ▶ Caching gives a performance boost
STRONG SCALING
Sorting 8 billion values on 4-16 GPUs
COMMUNICATING ALGORITHMS

3x3 Convolution
COMMUNICATING ALGORITHMS

3x3 Convolution
COMMUNICATION IS EVERYTHING
COMMUNICATION IS EVERYTHING
COMMUNICATION IS EVERYTHING
Halo Cells Keep Computation Local and Communication Asynchronous

Copy remote node boundary-cell data into local halo cells
STRONG SCALING: DIMINISHING RETURNS
LEAVING COMMUNICATION TO NVLINK
Pretending That Memory Over NVLink Is Local Memory

1. Eliminate halo cells
LEAVING COMMUNICATION TO NVLINK
Pretending That Memory Over NVLink Is Local Memory

1. Eliminate halo cells
2. Read directly from neighbor GPUs as if all memory were local
LEAVING COMMUNICATION TO NVLINK
Pretending That Memory Over NVLink Is Local Memory

1. Eliminate halo cells
2. Read directly from neighbor GPUs as if all memory were local
3. NVLink takes care of fast communication

How far can we push this?
LEAVING COMMUNICATION TO NVLINK

Pretending That Memory Over NVLink Is Local Memory

As GPU count increases

- Halo-to-core ratio \textit{increases}
- Off-chip accesses \textit{increase}
- On-chip accesses \textit{decrease}
- Proportions depend on algorithm

\textit{We reach NVLink bandwidth limit}
LEAVING COMMUNICATION TO NVLINK
Pretending That Memory Over NVLink Is Local Memory

As GPU count increases

- Halo-to-core ratio increases
- Off-chip accesses increase
- On-chip accesses decrease
- Proportions depend on algorithm

We reach NVLink bandwidth limit

BUT: Communication is many-to-many so full aggregate bandwidth is available
LOOKING FOR BANDWIDTH LIMITS

Hypothesis: Local-to-Remote Ratio Determines Performance

Stencil codes are memory bandwidth limited

Limited by **HBM2** bandwidth for **on-chip** reads

Limited by **NVLink** bandwidth for **off-chip** reads

NVLink = 120 GB/sec
HBM2 = 880 GB/sec
Ratio = 18.4%
NAÏVE 3D STENCIL PROGRAM
How Well Does The Simplest Possible Stencil Perform?

Aggregate B/W Achieved (GB/sec) over 16 GPUs
(512-cube volume, 3D "plus" stencil)

Slope = 25%
BANDWIDTH & OVERHEAD LIMITS

Example: LULESH stencil CFD code

Expect to lose performance when off-chip bandwidth exceeds NVLink bandwidth:

NVLink = 120 GB/sec
HBM2 = 880 GB/sec
Ratio = 18.4%
BONUS FOR NINJAS: ONE-SIDED COMMUNICATION

NVLink achieves higher bandwidth for writes than for reads:

Read requests consume inbound bandwidth at remote node

Difference is ~9%
WORK STEALING

Weak Scaling Mechanism: GPUs Compete To Process Data

Producers feed work into FIFO queue

Any consumer can pop head of queue whenever it needs more work
DESIGN OF A FAST FIFO QUEUE
Basic Building Block For Many Dynamic Scheduling Applications

Push Operation: Adds data to head of queue
DESIGN OF A FAST FIFO QUEUE
Basic Building Block For Many Dynamic Scheduling Applications

Push Operation: Adds data to head of queue

1. Advance head pointer to claim more space
**DESIGN OF A FAST FIFO QUEUE**

Basic Building Block For Many Dynamic Scheduling Applications

- **Push Operation**: Adds data to head of queue
  1. Advance head pointer to claim more space
  2. Write new data into space

Diagram showing a queue with a head pointer and a tail pointer.
DESIGN OF A FAST FIFO QUEUE

Basic Building Block For Many Dynamic Scheduling Applications

Pop Operation: Extracts data from tail of queue
DESIGN OF A FAST FIFO QUEUE
Basic Building Block For Many Dynamic Scheduling Applications

Pop Operation: Extracts data from tail of queue
1. Advance tail pointer to next item
DESIGN OF A FAST FIFO QUEUE

Basic Building Block For Many Dynamic Scheduling Applications

Pop Operation: Extracts data from tail of queue

1. Advance tail pointer to next item
2. Read data from new location
DESIGN OF A FAST FIFO QUEUE

Basic Building Block For Many Dynamic Scheduling Applications

Push Operation: Adds data to head of queue
1. Advance head pointer to claim more space
2. Write new data into space

Pop Operation: Extracts data from tail of queue
1. Advance tail pointer to next item
2. Read data from new location
DESIGN OF A FAST FIFO QUEUE

Problem: Concurrent Access Of Empty Queue

Empty Queue: When Head == Tail
DESIGN OF A FAST FIFO QUEUE

Problem: Concurrent Access Of Empty Queue

Push Operation: Adds data to head of queue

1. Advance head pointer to claim more space
DESIGN OF A FAST FIFO QUEUE

Problem: Concurrent Access Of Empty Queue

**Push Operation:** Adds data to head of queue

1. Advance head pointer to claim more space

**Pop Operation:** Extracts data from tail of queue

1. Advance tail pointer to next item
DESIGN OF A FAST FIFO QUEUE

Problem: Concurrent Access Of Empty Queue

**Push Operation:** Adds data to head of queue

1. Advance head pointer to claim more space

**Pop Operation:** Extracts data from tail of queue

1. Advance tail pointer to next item
2. Read data from new location
DESIGN OF A FAST FIFO QUEUE

Solution: Two Head (& Tail) Pointers For Thread-Safety
DESIGN OF A FAST FIFO QUEUE

Solution: Two Head (& Tail) Pointers For Thread-Safety

Push Operation: Adds data to head of queue

1. Advance outer head pointer
DESIGN OF A FAST FIFO QUEUE
Solution: Two Head (& Tail) Pointers For Thread-Safety

Push Operation: Adds data to head of queue
① Advance outer head pointer

Pop Operation: Extracts data from tail of queue
① While “tail” == “inner head”, do nothing
DESIGN OF A FAST FIFO QUEUE
Solution: Two Head (& Tail) Pointers For Thread-Safety

Push Operation: Adds data to head of queue
1. Advance outer head pointer
2. Write new data into space

Pop Operation: Extracts data from tail of queue
1. While “tail” == “inner head”, do nothing
DESIGN OF A FAST FIFO QUEUE
Solution: Two Head (＆ Tail) Pointers For Thread-Safety

Push Operation: Adds data to head of queue
1. Advance outer head pointer
2. Write new data into space
3. Advance inner head pointer

Pop Operation: Extracts data from tail of queue
1. While “tail” == “inner head”, do nothing
DESIGN OF A FAST FIFO QUEUE
Solution: Two Head (\& Tail) Pointers For Thread-Safety

Push Operation: Adds data to head of queue
1. Advance outer head pointer
2. Write new data into space
3. Advance inner head pointer

Pop Operation: Extracts data from tail of queue
1. While “tail” == “inner head”, do nothing
2. Advance tail pointer to next item
DESIGN OF A FAST FIFO QUEUE

Solution: Two Head (\& Tail) Pointers For Thread-Safety

Push Operation: Adds data to head of queue
1. Advance outer head pointer
2. Write new data into space
3. Advance inner head pointer

Pop Operation: Extracts data from tail of queue
1. While “tail” == “inner head”, do nothing
2. Advance tail pointer to next item
3. Read data from new location
DESIGN OF A FAST FIFO QUEUE
Thread-Safe Multi-Producer / Multi-Consumer Operation

Basic Rules

1. Use inner/outer head to avoid underflow
2. Use inner/outer tail to avoid overflow
3. Access all pointers atomically to allow multiple push/pop operations at once
4. NVLink carries atomics - this would be much, much harder over PCIe
SCALE EASILY BY ADDING MORE CONSUMERS

Limit Is Memory Bandwidth Between Queue & Consumers
SCALING OUT TO LOTS OF GPUS
It Works! Here’s an Incredibly Boring Graph To Prove It

Throughput vs. Number of GPUs
(2ms Packet Processing Time, DGX-2)
Why are more consumers worse?

- Large number of consumers accessing a single queue
- Saturates memory system at queue head
- Loss of throughput even though bandwidth is available
MEMORY CONTENTION LIMITATION
Throttling requests restores throughput, but costs latency

Mitigations

- Contention arises when many consumers are available for work
- Apply backoff delay between queue requests
- BUT: Unnecessary backoff increases latency
- Full-queue management still adds overhead
GUIDANCE AND GOTCHAS

NVSwitch Is The Secret Sauce

You can mostly ignore NVLink - it’s just like memory thanks to NVSwitch

BUT

2TB/sec is combined bandwidth, for many-to-many access patterns

If everyone accesses a single GPU, they share 137GB/sec
GUIDANCE AND GOTCHAS

NVSwitch Is The Secret Sauce

You can mostly ignore NVLink - it’s just like memory thanks to NVSwitch

BUT

2TB/sec is combined bandwidth, for many-to-many access patterns

If everyone accesses a single GPU, they share 137GB/sec

ALSO

Sometimes NVLink bandwidth is not the limiting factor

High contention at a single memory location hurt you

You have 2.6 million threads - contention can get very high
KEEPING PERFORMANCE HIGH

Spread threads and data across all GPUs to use the most hardware

Spread your data across all GPUs

- To avoid colliding memory requests at the “storage” GPU

Spread your threads across all GPUs

- To avoid all traffic congesting at the “computing” GPU

“All the wires, all the time”
GUIDANCE: RELY ON DGX-2 HARDWARE

How much tuning is needed?

Unified virtual memory gives you a simple memory model for spanning multiple GPUs

- At maximum performance

Remote memory traffic uses L1 cache

- Volta has a 128 KB cache for each SM
- Not coherent, use fences to ensure consistency after writes

Explicit management of local & remote memory accesses may improve performance
CONCLUSIONS

The fabric makes DGX-2 more than just 16 GPUs - it’s not just a mining rig

DGX-2 is a superb strong scaling machine in a time-to-solution sense

Overhead of Multi-GPU programming is low

Naïve code behaves well - great effort/reward ratio

Linearization of addresses through UVM provides a familiar model for free