S7300: MANAGED COMMUNICATION FOR MULTI-GPU SYSTEMS

Holger Fröning¹, Benjamin Klenk¹, Hans Eberle² & Larry Dennison²
¹ Ruprecht-Karls University of Heidelberg, Germany
² NVIDIA Research

http://www.ziti.uni-heidelberg.de/compeng
holger.froening@ziti.uni-heidelberg.de

GTC 2017, May 8, 2017
ABOUT US & TODAY

Performance and productivity for future and emerging technologies under hard power and energy constraints

Rather unusual hardware engineers

Sold on BSP styles of computing for data-intensive problems

Strong computer engineering background, focus on low-level software layers

High-performance analytics & high-performance computing

Today’s talk

An update on our work on GPU-centric communication
GPU APPLICATIONS

“Regular” algorithms: scientific/technical, HPC, machine learning

- Mostly dense matrix
- FFT, matrix-matrix multiplication, N-body, convolution, (deep) neural networks, finite-difference codes (PDE solvers)
- Excellent understanding in the community

"Irregular" algorithms: most algorithms outside computational science

- Organized around pointer-based data structures
- Data mining, Bayesian inference, compilers, functional interpreters, Maxflow, n-Body methods (Barnes-Hut, fast multipole), mesh refinement, graphics (ray tracing), event-driven simulation, relational join (databases), ...

Partly by Keshav Pingali et al., Amorphous Data-parallelism, technical report TR-09-05, U. Texas at Austin, 2009
David Kaeli, How Can GPUs Become First-Class Computing Devices?, William & Mary Computer Science Colloquium, October 26th 2016
NOTE ON DEEP LEARNING

Training dataset

shuffle

mini-batch

forward prop

back prop

optimizer

Data parallelism

forward prop

back prop

optimizer

Model parallelism

Sequential dependence

Training: 20 EFLOPs @10TFLOP/s = 23 days
REMINDER: BULK-SYNCHRONOUS PARALLEL

In 1990, Valiant already described GPU computing pretty well
Superstep
  Compute, communicate, synchronize
Parallel slackness: # of virtual processors $v$, physical processors $p$
  $v = 1$: not viable
  $v = p$: unpromising wrt optimality
  $v >> p$: scheduling and pipelining
Extremely scalable
A GPU is a (almost) perfect BSP processor

TRANSITIONING TO MULTI-GPU IS FUNDAMENTAL

Transition from SMP to NUMA

Reasons: multi-GPU systems, multi-chip modules, heterogeneous memory, tiled layout

Beauty of BSP is lost

Kernel launch orchestration
Data movement operations
Naming a physical resource is disgusting

Compute stack lacks NUMA support

Programming models
Abstractions
Consistency model
ADDRESSING NUMA

Analyzing NUMA latency effects

Observations on PCIe

  Huge penalty for local/remote
  Unloaded/loaded penalty

NVLINK changes the regime

Strong and dynamic NUMA effects

  Publicization/privatization concept

=> Managed communication

  Examples: MPI, TCP/IP, active messages, various more ...

<table>
<thead>
<tr>
<th>Pascal-class</th>
<th>Read latency [usec]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>unloaded</td>
</tr>
<tr>
<td>local</td>
<td>0.250</td>
</tr>
<tr>
<td>peer</td>
<td>1.311</td>
</tr>
<tr>
<td>host</td>
<td>0.838</td>
</tr>
<tr>
<td>factor</td>
<td>~3.3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pascal-class</th>
<th>Bandwidth [GB/s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>local</td>
<td>480</td>
</tr>
<tr>
<td>remote</td>
<td>16</td>
</tr>
<tr>
<td>factor</td>
<td>30</td>
</tr>
</tbody>
</table>
Background
Understanding massively-parallel communication
GPU-centric (but unmanaged) communication
Introducing MANTARO
Use cases for work execution
BACKGROUND
COMMUNICATION MODELS

Plain load/store (LD/ST) - de-facto standard in shared memory systems

- Never designed for communication
- Can be fast for SMP, but often unknown costs for NUMA
- Assumption of perfectly timed load seeing a store

Message passing (MP) - de-facto standard in HPC

- Various p2p and collective functions
- Mainly send/recv semantics used - ease-of-use
- Overhead due to functionality & guarantees: copying, matching, progress, ordering

Many more

- Active messages - latency tolerance becomes a programming/compiling concern
- One-sided communication (put/get) - never say receive
GPU COMMUNICATION TODAY

Standard: context switch to CPU
Limited to coarse-grain communication
Kernel-completion boundaries
Related work explores CPU helper threads

#GPU entities >> #CPU entities
Applicability depends on communication pattern
[DGCN, dCUDA, ...]
UPSHOT: CPU BYPASS HELPS

GPU-to-GPU streaming

Prototype system consisting of NVIDIA K20c, dual Intel Xeon E5, custom FPGA network
UNDERSTANDING MASSIVELY-PARALLEL COMMUNICATION

Do we need fine-grain privatization?
Characteristics of massively parallel communication
Analyzing large-scale HPC applications

DOE Exascale MPI proxy app traces
~1/2 TB analyzed (25+TB available online)

<table>
<thead>
<tr>
<th>Application</th>
<th>Pattern</th>
<th>Ranks</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOCFE (CESAR)</td>
<td>Nearest Neighbor</td>
<td>64; 256; 1024</td>
</tr>
<tr>
<td>NEKBBONE (CESAR)</td>
<td>Nearest Neighbor</td>
<td>64; 256; 1024</td>
</tr>
<tr>
<td>CNS (EXACT)</td>
<td>Nearest Neighbor</td>
<td>64; 256</td>
</tr>
<tr>
<td>CNS Large (EXACT)</td>
<td>Nearest Neighbor</td>
<td>64; 256; 1024</td>
</tr>
<tr>
<td>MultiGrid (EXACT)</td>
<td>Nearest Neighbor</td>
<td>64; 256</td>
</tr>
<tr>
<td>MultiGrid Large (EXACT)</td>
<td>Nearest Neighbor</td>
<td>64; 256; 1024</td>
</tr>
<tr>
<td>LULESH (EXMATEX)</td>
<td>Nearest Neighbor</td>
<td>64; 512</td>
</tr>
<tr>
<td>CMC 2D (EXMATEX)</td>
<td>Nearest Neighbor</td>
<td>64; 256; 1024</td>
</tr>
<tr>
<td>AMG (DF)</td>
<td>Irregular</td>
<td>64; 1728</td>
</tr>
<tr>
<td>AMG Boxlib (DF)</td>
<td>Nearest Neighbor</td>
<td>64; 256; 1024</td>
</tr>
<tr>
<td>BIGFFT (DF)</td>
<td>Many-to-many</td>
<td>100; 1024; 10000</td>
</tr>
<tr>
<td>BIGFFT Medium (DF)</td>
<td>Many-to-many</td>
<td>100; 1024; 10000</td>
</tr>
<tr>
<td>Crystal Router (DF)</td>
<td>Staged all-to-all</td>
<td>10; 100</td>
</tr>
</tbody>
</table>
APPLICATION CHARACTERISTICS

Observations

Structured patterns

- Neighbor
- Many-to-many
- All-to-all
- Irregular
APPLICATION CHARACTERISTICS

Observations

Structured patterns
Collectives for synchronization, point-to-point for communication
APPLICATION CHARACTERISTICS

Observations

- Structured patterns
- Collectives for synchronization, point-to-point for communication
- Most messages are surprisingly small
APPLICATION CHARACTERISTICS

Observations

Structured patterns
Collectives for synchronization, point-to-point for communication
Most messages are surprisingly small
Few communication peers

Communication peers as percentage of all ranks

<table>
<thead>
<tr>
<th>Job size (ranks)</th>
<th>Min</th>
<th>Median</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0:63]</td>
<td>3.1 %</td>
<td>28.1 %</td>
<td>40.6 %</td>
</tr>
<tr>
<td>[64:127]</td>
<td>6.0 %</td>
<td>12.0 %</td>
<td>15.2 %</td>
</tr>
<tr>
<td>[128:255]</td>
<td>0.6 %</td>
<td>7.8 %</td>
<td>26.4 %</td>
</tr>
<tr>
<td>[256:511]</td>
<td>3.7 %</td>
<td>5.4 %</td>
<td>7.1 %</td>
</tr>
<tr>
<td>[512:1023]</td>
<td>0.4 %</td>
<td>2.0 %</td>
<td>7.0 %</td>
</tr>
<tr>
<td>[1024:2047]</td>
<td>1.3 %</td>
<td>2.0 %</td>
<td>4.6 %</td>
</tr>
<tr>
<td>[8192:16383]</td>
<td>0.1 %</td>
<td>0.2 %</td>
<td>0.7 %</td>
</tr>
</tbody>
</table>
APPLICATION CHARACTERISTICS

Observations

Structured patterns
Collectives for synchronization, point-to-point for communication
Most messages are surprisingly small
Few communication peers

Insights on communication

Selective, structured and fine-grained
Little/no use of advanced MPI features
Irregular applications will further push requirements

<table>
<thead>
<tr>
<th>Application</th>
<th>Pattern</th>
<th>Ranks</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOCFE (CESAR)</td>
<td>Nearest Neighbor</td>
<td>64; 256; 1024</td>
</tr>
<tr>
<td>NEKBONE (CESAR)</td>
<td>Nearest Neighbor</td>
<td>64; 256; 1024</td>
</tr>
<tr>
<td>CNS (EXACT)</td>
<td>Nearest Neighbor</td>
<td>64; 256</td>
</tr>
<tr>
<td>CNS Large (EXACT)</td>
<td>Nearest Neighbor</td>
<td>64; 256; 1024</td>
</tr>
<tr>
<td>MultiGrid (EXACT)</td>
<td>Nearest Neighbor</td>
<td>64; 256</td>
</tr>
<tr>
<td>MultiGrid Large (EXACT)</td>
<td>Nearest Neighbor</td>
<td>64; 256; 1024</td>
</tr>
<tr>
<td>LULESH (EXMATEX)</td>
<td>Nearest Neighbor</td>
<td>64; 512</td>
</tr>
<tr>
<td>CMC 2D (EXMATEX)</td>
<td>Nearest Neighbor</td>
<td>64; 256; 1024</td>
</tr>
<tr>
<td>AMG (DF)</td>
<td>Nearest Neighbor</td>
<td>64; 256; 1024</td>
</tr>
<tr>
<td>AMG Boxlib (DF)</td>
<td>Nearest Neighbor</td>
<td>64; 256; 1024</td>
</tr>
<tr>
<td>BIGFFT (DF)</td>
<td>Irregular</td>
<td>64; 1728</td>
</tr>
<tr>
<td>BIGFFT Medium (DF)</td>
<td>Many-to-many</td>
<td>100; 1024; 10000</td>
</tr>
<tr>
<td>Crystal Router (DF)</td>
<td>Staged all-to-all</td>
<td>10; 100</td>
</tr>
</tbody>
</table>

GPU-CENTRIC (BUT UNMANAGED) COMMUNICATION

Addressing the need for privatization
GPU-CENTRIC TRAFFIC SOURCING & SINKING

**GGAS:** GPU-centric send/receive
- Thread-collective data movement
- Complete CPU bypass

**Cons**
- Special hardware support required
- Reduced overlap

**GRMA:** GPU-centric put/get
- Key is simple descriptor format

**Cons**
- Special hardware support required
- Indirection to issue work

---

Lena Oden and Holger Fröning, **GGAS: Global GPU Address Spaces for Efficient Communication in Heterogeneous Clusters**, IEEE CLUSTER 2013
MICRO-BENCHMARK PERFORMANCE

GPU-to-GPU streaming

Prototype system consisting of NVIDIA K20c, dual Intel Xeon E5, custom network

MPI

CPU-controlled: D2H, MPI send/recv, H2D

Others

GPU-controlled, bypassing CPU

Results do not cover overheads regarding issue & completion
Experiments with applications rewritten for GPU-centric communication

12 nodes (each 2x Intel Ivy Bridge, NVIDIA K20, FPGA network)

Specialized communication always faster than MPI

But can we also get the convenience of managed communication?
INTRODUCING MANTARO

GPU-centric privatization
A MANY-CORE MESSAGE PROCESSOR

Transforming an SM into a message-parallel processor (SOFTNIC)

- Building blocks to support send/recv, put/get, active messages, ...
- Layered on top of LD/ST over global address spaces
- Or interfacing to a NIC/CPU

Managed communication
- Buffer management
- Protocol selection
- Scheduling data transfers
- Choosing communication paths
- Asynchronous communication

Adaptable (reprogrammable) to workload
Scalable with flows and GPUs
FLEXIBLE & COMPOSABLE

Flexible: who sources/sinks traffic?
- Threads, warps, CTAs or kernels

Flexible: what is the model?
- Send/recv, put/get, active messages?

Flexible: which data path?
- LD/ST or DMA engines

Composable using building blocks

Three fundamental tasks
1. Work generation
2. Work execution
3. Work completion
WORK GENERATION

Warp-parallel queue
- Collaborative enqueue of 1-32 elements
- Avoids branch divergence
- Warp-parallel except for pointer update

Building block for various uses
- Entities: warps, CTAs, or kernels
- Shared, global or remote memory

Communication as a sequence of queues
WORK COMPLETION

Notifications have to be found quickly
- Tables are very handy
- Parallel search, low administration overhead
- Messaging operations returns pointer to table entry

Aggregating notifications
- Reducing table contention
- Reducing time to find all notifications

Issues with current GPUs
- Preemption & scheduling

```c
compute_stencil (...) {
    ...  
    exchange_halo(top, noti);
    exchange_halo(bot, noti);
    exchange_halo(left, noti);
    exchange_halo(right, noti);
    ...  
    wait( noti == 4 );
}```
USE CASES FOR WORK EXECUTION

MPI-like send/recv
Active messages
REMINDER: MESSAGE MATCHING USING MPI

Match one send to one receive based on \{\text{communicator, sender, tag}\}-tuple

- Wildcards on sender and tag possible
- Messages can arrive unexpectedly
- Messages stay in-order

MPI internally maintains lists for pre-posted receives and unexpected messages

Queue length and search depth of importance

\[
\text{MPI\_I\_Send(} \text{<buffer>, <count>, <type>, <dest.>, <tag>, <comm>, ...})
\]

\[
=\]

\[
\text{MPI\_I\_Recv(} \text{<buffer>, <count>, <type>, <source>, <tag>, <comm>, ...})
\]
CPU MATCHING PERFORMANCE

Best case (forward)  Average case (random)
MASSIVELY-PARALLEL TAG MATCHING

Parallelization

Vote matrix
Shared memory
Hierarchical approach

1. Multi-warp scan
   Based on __ballot
2. Single-warp reduction of column vector to single vote
   Based on __ballot, __ffs and bit masking
RELAXING MATCHING SEMANTICS

No unexpected messages
  No compaction (10% perf.)
  No unnecessary propagation of unmatched elements

No source wildcards
  Rank partitioning
  Multiple matrixes

No ordering
  Hash tables
  Constant insert and search time complexity

Two orders of magnitude

<table>
<thead>
<tr>
<th>Wildcards</th>
<th>Ordering</th>
<th>Unexpected messages</th>
<th>Partitioning</th>
<th>Data structure</th>
<th>Performance [matches/s]</th>
<th>User implications</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>matrix</td>
<td>&lt; 6M</td>
<td>none (MPI-like)</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>matrix</td>
<td>~ 6M</td>
<td>medium</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>matrix</td>
<td>&lt; 60M</td>
<td>low</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>matrix</td>
<td>~ 60M</td>
<td>medium</td>
</tr>
<tr>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>hash table</td>
<td>&lt; 500M</td>
<td>high</td>
</tr>
<tr>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>hash table</td>
<td>~ 500M</td>
<td>high</td>
</tr>
</tbody>
</table>
ACTIVE MESSAGES

Heavily used in task-based programming models

Map nicely to irregular applications

Work lists

Coalescing/aggregation

Possibly sorting for locality maximization

Different forms of execution in Mantaro

Inline (thread warp) - limited to max. 32 threads

Inline (complete CTA) - stalls communication

Kernel launch - high costs (NVIDIA’s Dynamic Parallelism feature)

Registered and pre-launched kernel (persistent threads)
RANDOM-ACCESS BENCHMARK

Part of HPCC benchmark suite (CPU version)

http://icl.cs.utk.edu/hpcc/

Ported to a GPU version

- Data-driven memory accesses distributed over multiple GPUs
- Many fine-grain interactions
- Buckets aggregate update operations

Performance

- PCIe-connected K80 GPUs
- Up to 1 GUPS, good scalability
- Similar to equivalent CPU system (192 MPI ranks, 104 SMs total)
WRAPPING UP
COMMUNICATION MODEL PROPOSALS

Send/recv communication
No ordering guarantees, limited use of wildcards
Asynchronous communication
Consistency control: blocking wait, or pre-registered, deferred actions depending on completion

Active messages
Pre-registered functions
Different forms of execution, possibly dynamically determined
Data placement determines place of execution

Point-to-point source
`mantaro_send (dest, &buf, tag, &handle, ...);`

Point-to-point sink
`mantaro_recv (src, &buf, tag, &handle, ...);`

Collective ops
`mantaro_barrier (group, &handle, ...);
mantaro_all2all (tag, &handle, ...);`

Synchronization
`mantaro_wait (&handle); /* blocking wait */
mantaro_defer (&handle, &action);`

Function registration
// base handler class w/t virtual functions
class AMUpdate : public Mantaro::AMBase {...}

AM send
`mantaro_am_send (AMUpdate_msg, &buf, ...);`
SUMMARY

Managed communication addresses

- Strong NUMA effects of multi-GPU systems
- Needs of fine-grain, selective communication

Mantaro: a many-core message-parallel processor

- Capable of handling massively parallel communication
- Flexible and adaptable
- Tool to explore GPU communication

Issues/limitations

- Inter-CTA latency, progress guarantees, preemption & memory management, execution launch costs
- We believe GPUs will continue to evolve