S6357
Towards Efficient Communication Methods and Models for Scalable GPU-Centric Computing Systems

Holger Fröning
Computer Engineering Group
Ruprecht-Karls University of Heidelberg

GPU Technology Conference 2016
About us

• JProf. Dr. Holger Fröning
  • PI of Computer Engineering Group, ZITI, Ruprecht-Karls University of Heidelberg
  • http://www.ziti.uni-heidelberg.de/compeng
• Research: Application-specific computing under hard power and energy constraints (HW/SW), future emerging technologies
  • High-performance computing (traditional and emerging)
  • GPU computing (heterogeneity & massive concurrency)
  • High-performance analytics (scalable graph computations)
  • Emerging technologies (approximate computing and stacked memory)
  • Reconfigurable logic (digital design and high-level synthesis)
• Current collaborations
  • Nvidia Research, US; University of Castilla-La Mancha, Albacete, Spain; CERN, Switzerland; SAP, Germany; Georgia Institute of Technology, US; Technical University of Valencia, Spain; TU Graz, Austria; various companies
The Problem

- GPU are powerful high-core count devices, but only for in-core computations
  - But many workloads cannot be satisfied by a single GPU
  - Technical computing, graph computations, data-warehousing, molecular dynamics, quantum chemistry, particle physics, deep learning, spiking neural networks
- => Multi-GPU, at node level and at cluster level
  - Hybrid programming models
  - While single GPUs are rather simple to program, interactions between multiple GPUs dramatically push complexity!
- This talk: how good are GPUs in sourcing/sinking network traffic, how should one orchestrate communication, what do we need for best performance/energy efficiency
Review: Messaging-based Communication

- Usually Send/Receive or Put/Get
  - MPI as de-facto standard
- Work requests descriptors
  - Issued to the network device
  - Target node, source pointer, length, tag, communication method, ...
  - Irregular accesses, little concurrency
- Memory registration
  - OS & driver interactions
- Consistency by polling on completion notifications
Beyond CPU-centric communication

... a bad semantic match between communication primitives required by the application and those provided by the network." - DOE Subcommittee Report, Top Ten Exascale Research Challenges. 02/10/2014

Communication orchestration - how to source and sink network traffic
Example application: 3D stencil code

• Himeno 3D stencil code
  • Solving a Poisson equation using 2D CTAs (marching planes)
  • Multiple iterations using iterative kernel launches
  • Multi-GPU: inter-block and inter-GPU dependencies

• Dependencies => communication
  • Inter-block: device synchronization required among adjacent CTAs
  • Inter-GPU: all CTAs participate communications (sourcing and sinking) => device synchronization required

3D Himeno stencil code with 2D CTAs

Control flow using CPU-controlled communication
Different forms of communication control for an example stencil code

Control flow using in-kernel synchronization

Control flow using stream synchronization (with/without nested parallelism)
Performance comparison - execution time

- CPU-controlled still fastest
  - Backed up by previous experiments
- In-kernel synchronization slowest
  - Communication overhead increases with problem size: more CTAs, more device synchronization
  - ~28% of all instructions have to be replayed, likely due to serialization (use of atomics)
- Stream synchronization a good option
  - Difference to device synchronization is overhead of nested parallelism
- Device synchronization most flexible regarding control flow
  - Communication as device function or as independent kernel
  - Flexibility in kernel launch configuration

Performance comparison - energy consumption

- Benefits for stream/device synchronization as the CPU is put into sleep mode
  - 10% less energy consumption
  - CPU: 20-25W saved
- In-kernel synchronization saves much more total power, but execution time increase results in a higher energy consumption
  - Likely bad GPU utilization

• CPU-controlled communication is still fastest - independent of different orchestration optimizations
• GPU-controlled communication: intra-GPU synchronization between the individual CTAs is most important for performance
  • Stream synchronization most promising
  • Otherwise reply overhead due to serialization
• Dedicated communication kernels or functions are highly recommended
  • Either device functions for master kernel (nested parallelism), or communication kernels in the same stream (issued by CPU)
• Bypassing CPUs has substantial energy advantages
  • Decrease polling rates, or use interrupt-based CUDA events!
  • More room for optimizations left

```c
while( cudaStreamQuery(stream) == cudaErrorNotReady )
    usleep(sleeptime);
```
GGAS: Fast GPU-controlled traffic sourcing and sinking
GGAS – Global GPU Address Spaces

- Forwarding load/store operations to global addresses
  - Address translation and target identification
  - Special hardware support required (NIC)
- Severe limitations for full coherence and strong consistency
  - Well known for CPU-based distributed shared memory
- Reverting to highly relaxed consistency models can be a solution

Holger Fröning and Heiner Litz, Efficient Hardware Support for the Partitioned Global Address Space, *10th Workshop on Communication Architecture for Clusters (CAC2010)*, co-located with 24th International Parallel and Distributed Processing Symposium (IPDPS 2010), April 19, 2010, Atlanta, Georgia.
remMailbox[getProcess(index)][tid] = data[tid];
__threadfence_system(); // memory fence
remoteEndFlag[getProcess(index)][0] = 1;
__ggas_barrier();
...
GGAS Prototype

- FPGA-based network prototype
  - Xilinx Virtex-6
  - 64bit data paths, 156MHz = 1.248GB/s (theoretical peak)
  - PCIe G1/G2
  - 4 network links (torus topology)

Remote load latency
Virtex-6: 1.44 – 1.9 usec (CPU/GPU)

Node #0 (Source)
Issuing loads/stores

Node #1 (Target)
Memory host

Source-local address
Target node determination
Address calculation

Global address
Loss-less and in-order packet forwarding

Target-local address
Source tag management
Address calculation
Return route
• GPU-to-GPU streaming
  • Prototype system consisting of Nvidia K20c & dual Intel Xeon E5
  • Relative results applicable to technology-related performance improvements
• MPI
  • CPU-controlled: D2H, MPI send/recv, H2D
• GGAS
  • GPU-controlled: GDDR to GDDR, remote stores
• RMA: Remote Memory Access
  • Put/Get-based, CPU-to-CPU (host) resp. GPU-to-GPU (direct)

GGAS latency starting at 1.9usec
Analyzing Communication Models for Thread-parallel Processors
• MPI
  • CPU-controlled
  • De-facto standard, widely used, heavily optimized (for CPUs)
• GGAS (using mailboxes with send/receive semantics)
  • GPU-controlled
  • Communication by forwarding load/store operations using global address spaces
  • Completely in-line with the GPU execution model => highly thread parallel
  • Main drawback is reduced overlap
• RMA: Remote Memory Access
  • GPU-controlled
  • Put/Get operations of the custom interconnect
  • Communication engine designed for HPC
  • GPUs have to construct/interpret descriptors (which was very crucial for the IBVERBS experiment)
Completing GGAS with Put/Get operations

- Descriptor-based Put/Get operations
  - Completely asynchronous
- Work request with
  - Type
  - Local/Remote pointers
  - Credentials
  - Notification requests
- Notification
  - Completion information with reference to work request
- Key is a simple descriptor format
Put/Get: Different Work Request Queue Implementations

- Descriptor format and queue organization matters
- Explicit/implicit trigger, conversion effort, descriptor complexity
- FPGA implementation: we could even change the descriptor format

IBVERBS descriptor format

![IBVERBS descriptor format]

Custom descriptor format

![Custom descriptor format]
Example applications

- Testing is hard as applications have to be re-written for GPU-centric communication
- Set of 4 workloads implemented:

<table>
<thead>
<tr>
<th></th>
<th>N-Body</th>
<th>Himeno</th>
<th>Global sum</th>
<th>Rnd. Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computation</td>
<td>$\mathcal{O}(n^2)$</td>
<td>$\mathcal{O}(n)$</td>
<td>$\mathcal{O}(n)$</td>
<td>$\mathcal{O}(n)$</td>
</tr>
<tr>
<td>Memory</td>
<td>$\mathcal{O}(n)$</td>
<td>$\mathcal{O}(n)$</td>
<td>$\mathcal{O}(n)$</td>
<td>$\mathcal{O}(n)$</td>
</tr>
<tr>
<td>Communication</td>
<td>$\mathcal{O}(n)$</td>
<td>$\mathcal{O}(\sqrt[3]{n^2})$</td>
<td>$\mathcal{O}(\log n)$</td>
<td>$\mathcal{O}(n)$</td>
</tr>
<tr>
<td>Communication Pattern</td>
<td>Ring</td>
<td>Neighbor</td>
<td>All-gather</td>
<td>Rnd. Unif.</td>
</tr>
<tr>
<td>Avg. Payload</td>
<td>$\frac{n}{N}$</td>
<td>$\frac{3\sqrt{n^2}}{\sqrt{2}}$</td>
<td>1 val.</td>
<td>0-1024 vals.</td>
</tr>
<tr>
<td>Overlap</td>
<td>+</td>
<td>++</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Performance comparison - execution time

- 2-12 nodes (each 2x Intel Ivy Bridge, Nvidia K20, FPGA network)
- Normalized to MPI: >1 = better performance, <1 = worse performance
12 nodes (each 2x Intel Ivy Bridge, Nvidia K20, Extoll FPGA)

Normalized to MPI: <1 = better energy consumption, >1 = worse energy consumption
### Performance comparison - observations

<table>
<thead>
<tr>
<th>Observation</th>
<th>N-Body</th>
<th>Himeno Stencil</th>
<th>Global Sum</th>
<th>Random Access</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RMA and MPI offer a better exploitation of overlap possibilities</strong></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>GGAS performs outstanding for small payloads, as no indirections are required like context switches to the CPU or work request issues to the NIC</strong></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>The PCIe peer-to-peer read problem results in MPI performing better than RMA or GGAS for large payload sizes</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td><strong>GGAS in combination with RMA outperform MPI substantially (without the PCIe peer-to-peer read limitation)</strong></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td><strong>In practice, the execution time has an essential influence on energy consumption</strong></td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td><strong>Accesses to host memory contribute significantly to DRAM and CPU socket power</strong></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td><strong>Bypassing a component like a CPU can save enough power to compensate a longer execution time, resulting in energy savings</strong></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td><strong>Staging copies contribute significantly to both CPU and GPU power, due to involved software stacks respectively active DMA controllers</strong></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td><strong>For irregular communication patterns or small payloads, GGAS saves both time and energy</strong></td>
<td></td>
<td></td>
<td></td>
<td>X (X)</td>
</tr>
</tbody>
</table>
Related effort: Simplified multi-GPU programming
• Single-GPU programming based on CUDA/OpenCL is a prime example for a BSP execution model
  • Exposes large amounts of structured parallelism, rather easy to use
  • Multi-GPU programming becoming more important, but adds huge amounts of complexity
    • Processor aggregation easy
    • Memory aggregation challenging
      • Local/remote bandwidth disparity
      • UVA (Unified Virtual Addressing) works out-of-the-box
        -> poor performance
  • Solution: Use a compiler-based approach to automatically partition regular GPU code
• Tool stack based on LLVM tool chain
  • Code analysis
  • Automated decision making
  • Code transformations for automated partitioning and data movements

GPU Mekong
• Funded by a Google Research Award
• Under heavy development
• [https://sites.google.com/site/gpumekong](https://sites.google.com/site/gpumekong)
• Initial results for a 16-GPU matrix multiply
Wrapping up
Summary

• GPUs as first-class citizens in a peer networking environment, capable of sourcing and sinking traffic
  • Traditional messaging libraries and models poorly match the GPU’s thread-collaborative execution model
  • Also GPUs require fast communication paths with minimal latency/maximum message rate, combined with high-overlap paths like Put/Get

Specialized processors like GPUs require specialized communication models/methods

• However: the semantic gap between architecture and user is growing
  • We need automated tooling to close this gap

• Unified communication models
  • Automatically selecting the right communication method and path between heterogeneous computing units => flexibility of scheduling kernels
  • Multiple communication models will dramatically push complexity, too

• CACM 03/2015, John Hennessy: it’s the era of software
Credits
Contributions: Lena Oden (former PhD student), Benjamin Klenk (PhD student), Daniel Schlegel (graduate student), Günther Schindler (graduate student)

Discussions: Sudha Yalamanchili (Georgia Tech), Jeff Young (Georgia Tech), Larry Dennison (Nvidia), Hans Eberle (Nvidia)

Sponsoring: Nvidia, Xilinx, German Excellence Initiative, Google

Current main interactions

Thank you!