Evaluating Characteristics of CUDA Communication Primitives on High-Bandwidth Interconnects

Carl Pearson¹, Abdul Dakkak¹, Sarah Hashash¹, Cheng Li¹, I-Hsin Chung², Jinjun Xiong², Wen-Mei Hwu¹

¹ University of Illinois Urbana-Champaign, Urbana, IL
² IBM T. J. Watson Research, Yorktown Heights, NY
Why GPU Interconnect Bandwidth?

Nvidia V100 attached by PCIe 3

**GPU Consumes**

15.7 TFLOP FP32

31.4 x 10^{12} operands/s

**Host Produces**

15.8 GB/s over PCIe

3.95 x 10^{9} operands/s

~8000 FP32 operations per operand transferred

or

~2000 FP32 operations per byte transferred
Challenges and Contributions

**Challenge**
CUDA data transfer bandwidth depends on allocation and transfer method

**Bad Result**
Incomplete System Characterization

**CommScope Solution**
Microbenchmarks for all CUDA communication methods
Avoid synchronization overhead from measurements
Challenges and Contributions

**Challenge**
CUDA data transfer bandwidth depends on allocation and transfer method

Bandwidth influenced by non-CUDA knobs and system topology

**Bad Result**
Incomplete System Characterization

Variability across measurements

**CommScope Solution**
Microbenchmarks for all CUDA communication methods

Avoid synchronization overhead from measurements

Explore effect of system topology

Understand and control non-CUDA system parameters during measurements
Challenges and Contributions

**Challenge**
- CUDA data transfer bandwidth depends on allocation and transfer method
- Bandwidth influenced by non-CUDA knobs and system topology
- Complicated API behavior and system interaction

**Bad Result**
- Incomplete System Characterization
- Variability across measurements
- Reinventing the wheel and repeating mistakes

**CommScope Solution**
- Microbenchmarks for all CUDA communication methods
- Avoid synchronization overhead from measurements
- Explore effect of system topology
- Understand and control non-CUDA system parameters during measurements
- Open-source, cross-platform, error reporting, plotting results
Comprehensive Coverage of CUDA Bulk Transfers

- Explicit transfers
- Peer Access
- “Zero-Copy”

- Unified Memory
- Unidirectional Transfers
- Bidirectional Transfers
Non-CUDA Parameter: NUMA Pinning

- Not all cudaMemcpy created equal on high-bandwidth interconnects

<table>
<thead>
<tr>
<th>Configuration (Limiter)</th>
<th>Theoretical (GB/s)</th>
<th>Observed (GB/s)</th>
<th>1GB pinned host allocation transferred to GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC922 Local (3x NVLink 2)</td>
<td>75</td>
<td>66.6 ± 0.013</td>
<td></td>
</tr>
<tr>
<td>AC922 Remote (X-bus)</td>
<td>64</td>
<td>41.3 ± 0.009</td>
<td></td>
</tr>
<tr>
<td>S822LC Local (2x NVLink 1)</td>
<td>40</td>
<td>31.9 ± 0.008</td>
<td></td>
</tr>
<tr>
<td>S822LC Remote (x-bus)</td>
<td>38.4</td>
<td>29.3 ± 0.013</td>
<td></td>
</tr>
<tr>
<td>4029GP Local (PCIe 3)</td>
<td>15.8</td>
<td>12.4 ± 0.0002</td>
<td></td>
</tr>
<tr>
<td>4029GP Remote (PCIe 3)</td>
<td>15.8</td>
<td>12.4 ± 0.0002</td>
<td></td>
</tr>
</tbody>
</table>
Non-CUDA Parameters

- **Variable CPU Clock Speeds**
  
  ```
  $ cpupower frequency-set --governor performance
  ```

- **CPU Data Caching**
Pinned Allocation and cudaMemcpy

- GPU does DMA to access pinned data on CPU

\[
\text{cudaMemcpy( ... , cudaMemcpyHostToDevice)} \quad \text{cudaMemcpy( ... , cudaMemcpyDeviceToHost)}
\]
CUDA & CPU Cache

- CPU writes values to initialize data
- For small allocations, data may reside entirely in cache

```
cudamemcpy( ... , cudaMemcpyHostToDevice)
cudamemcpy( ... , cudaMemcpyDeviceToHost)
```
cudaMemcpy & CPU Cache

- Flushing the cache forces data to start in the DRAM
- Flushing the cache prevents write-back of dirty data

```
cudaMemcpy( ... , cudaMemcpyHostToDevice)
cudaMemcpy( ... , cudaMemcpyDeviceToHost)
```
<table>
<thead>
<tr>
<th></th>
<th><strong>Host to GPU</strong></th>
<th><strong>GPU to Host</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>No Flushing</strong></td>
<td>Flushing forces data to start in DRAM, slowing transfer</td>
<td>Flushing prevents dirty data from being evicted, speeding transfer</td>
</tr>
<tr>
<td></td>
<td><img src="https://via.placeholder.com/150" alt="Diagram" /> 52.14 GB/s</td>
<td><img src="https://via.placeholder.com/150" alt="Diagram" /> 14.91 GB/s</td>
</tr>
<tr>
<td><img src="https://via.placeholder.com/150" alt="Diagram" /> 29.40 GB/s</td>
<td><img src="https://via.placeholder.com/150" alt="Diagram" /> 14.91 GB/s</td>
<td></td>
</tr>
<tr>
<td><strong>Flushing</strong></td>
<td><img src="https://via.placeholder.com/150" alt="Diagram" /> 45.20 GB/s</td>
<td><img src="https://via.placeholder.com/150" alt="Diagram" /> 29.40 GB/s</td>
</tr>
</tbody>
</table>

Dirty data from cache
Benchmark Design

- Using Google Benchmark Support Library
  - Each benchmark run consists of some number of iterations
  - The number of iterations is $1 < n < 1e9$ and total time under measurement $\geq 0.5s$
- Support synchronous and asynchronous operations
- Report variability across runs
  - High variability suggests not all relevant system parameters are fixed
Initialization (as needed)

- Resetting CUDA devices
- NUMA pinning
- Creating allocations
- Creating CUDA streams and events
- Zeroing allocations
- Configure CUDA device peer access
Setup (as needed)

- Move unified memory data to a source device
- Flush caches
- Set CUDA devices
- Adjust NUMA pinning
Timing Strategies

- Timing the data transfer operation
- Different approaches for different transfer types:
  - Synchronous
  - Asynchronous
  - Simultaneous
Asynchronous Operations

- An operation that may complete at any time (from the perspective of the host)
- CUDA API call may return before the operation is complete
Asynchronous Behavior in Synchronous APIs

- cudaMemcpy
  - CUDA Runtime API §2: “for transfers from pageable host memory to device memory…the function will return once the pageable buffer has been copied to the staging memory, but the DMA to final destination may not have completed”

```cpp
// wrong
start = std::chrono::system_clock::now();
cudaMemcpy(..., cudaMemcpyHostToDevice);
end   = std::chrono::system_clock::now();
```
Timing Single Operations

- No spurious synchronization costs!
Timing Simultaneous Sync/Async Operations

- **Synchronous Operation**
  - `cudaStreamSynchronize`
  - `stop wall time`

- **Asynchronous Operation**
  - `unlaunch async`
  - `async operation`
  - `cudaStreamSynchronize`
  - `stop wall time`

**Reported Time**

**Unavoidable stream synchronization is measured**
Timing Simultaneous Asynchronous Operations

Single Device

Device 0 / Stream 0
- "start" event
- operation 0
- "stop" event

Device 0 / Stream 1
- "start" event
- operation 1
- "stop" event

No spurious synchronization costs!

Multiple Device

Device 0 / Stream 0
- "start" event
- operation 0
- wait
- "stop" event

Device 1 / Stream 1
- "other" event
- operation 1

Streams synchronization event measured
# IBM S822LC and IBM AC922

<table>
<thead>
<tr>
<th>Spec</th>
<th>S822LC</th>
<th>AC922</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>2x IBM POWER 8</td>
<td>2x IBM POWER 9</td>
</tr>
<tr>
<td>GPU</td>
<td>4x Nvidia P100 (Pascal)</td>
<td>4x Nvidia V100 (Volta)</td>
</tr>
<tr>
<td>CPU ↔ CPU</td>
<td>X-bus (38.4 GB/s)</td>
<td>X-bus (64 GB/s)</td>
</tr>
<tr>
<td>CPU ↔ GPU</td>
<td>2x NVLink 1 (80 GB/s)</td>
<td>3x NVLink 2 (150 GB/s)</td>
</tr>
<tr>
<td>GPU ↔ GPU</td>
<td>2x NVLink 1 (80 GB/s)</td>
<td>3x NVLink 2 (150 GB/s)</td>
</tr>
</tbody>
</table>

![Diagram of CPU and GPU connectivity](image.png)
## SuperMicro 4029GP-TVRT

<table>
<thead>
<tr>
<th>Spec</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>2x Intel Xeon Gold 6148</td>
</tr>
<tr>
<td>GPU</td>
<td>8x Nvidia V100 (Volta)</td>
</tr>
<tr>
<td>CPU ↔ CPU</td>
<td>Intel UPI (62.4 GB/s)</td>
</tr>
<tr>
<td>CPU ↔ GPU</td>
<td>PCIe 3.0 x16 (31.6 GB/s)</td>
</tr>
<tr>
<td>GPU ↔ GPU</td>
<td>1x/2x NVLink 2 (25-50 GB/s)</td>
</tr>
</tbody>
</table>
No Locality or Anisotropy on PCIe

- cudaMemcpyAsync vs zero-copy CPU/GPU
- explicit vs zero-copy CPU/GPU

Unified memory demand transfers
- Low bandwidth PCIe 3.0 on 4029GP hides interesting behavior

cudaMemcpyAsync
## Pageable Host Allocations and Fast Interconnects

- The implicit pageable-to-pinned copy prevents exploiting fast interconnects
- Multiple threads should speed up pageable-pinned copy
  - Application could use simultaneous transfers
  - CUDA runtime could use multiple worker threads
Strong Locality with High Bandwidth Configurations

Transfers across NVLink 2 show strong locality effects.

cudaMemcpyAsync CPU-GPU  
cudaMemcpyAsync GPU-GPU
Demand Page Migration

- CUDA system software limits performance available in hardware
  - Page faults
  - Per-page driver heuristics
- Underlying interconnect performance not so important
Multiple host threads are needed to make UM faster
Zero-Copy

- Implicit, like unified memory
- Unlike unified memory, can achieve near interconnect theoretical bandwidth
Unified Memory Prefetch vs Explicit

- Unified memory prefetch is slow at intermediate sizes
Open-source & Docker

- v0.7.2 released April 8th
- Github: c3sr/comm_scope
- Docker: c3sr/comm_scope

- CUDA 8.0+, CMake 3.12+
- x86 and POWER
- Apache 2.0 license
- Python scope_plot package for plotting results
Future Work

- Unified Memory Microbenchmarks
  - Access patterns & driver heuristics

- System-aware CPU/GPU and GPU/GPU data structures
  - How to allocate and move data depending on who produces and who consumes
    - Hints from application or records from previous executions

- System health status
  - Sanity check during system firmware development or system bring-up
Conclusion

- Comprehensive coverage of CUDA communication methods
- Bandwidth affected by CUDA APIs, non-CUDA system knobs, system topology
- High-bandwidth interconnects expose idiosyncracies of hardware/software system
- Open-source, cross-platform, artifact evaluation stamp
Thank you / Questions

pearson@illinois.edu
https://cwpearson.github.io

Other C3SR System Performance Research Projects

System microbenchmarks: https://scope.c3sr.com
Full-stack machine learning with tracing: https://mlmodelscope.org

This work is supported by IBM-ILLINOIS Center for Cognitive Computing Systems Research (C3SR) - a research collaboration as part of the IBM AI Horizon Network.

This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation award OCI-0725070 and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications.