NUMA-Aware Data-Transfer Measurements for Power/NVLink Multi-GPU Systems

Carl Pearson[1](✉), I-Hsin Chung[2](✉), Zehra Sura[2](✉), Wen-Mei Hwu[1](✉), and Jinjun Xiong[2](✉)

[1] University of Illinois Urbana-Champaign, Urbana, IL 61801, USA
`{pearson, w-hwu}@illinois.edu`

[2] IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598, USA
`{ihchung, zsura, jinjun}@us.ibm.com`

Abstract. High-performance computing increasingly relies on heterogeneous systems with specialized hardware accelerators to improve application performance. For example, NVIDIA's CUDA programming system and general-purpose GPUs have emerged as a widespread accelerator in HPC systems. This trend has exacerbated challenges of data placement as accelerators often have fast local memories to fuel their computational demands, but slower interconnects to feed those memories. Crucially, real-world data-transfer performance is strongly influenced not just by the underlying hardware, but by the capabilities of the programming systems. Understanding how application performance is affected by the logical communication exposed through abstractions, as well as the underlying system topology, is crucial for developing high-performance applications and architectures. This report presents initial data-transfer microbenchmark results from two POWER-based systems obtained during work towards developing an automated system performance characterization tool.

Keywords: CUDA · NVLink · Unified Memory · GPGPU Benchmark

1 Introduction

With the end of Dennard scaling, computer architects have sought to satisfy the demand for increasing performance by providing specialized hardware

This work is supported by IBM-ILLINOIS Center for Cognitive Computing Systems Research (C3SR) - a research collaboration as part of the IBM Cognitive Horizon Network. This work was supported by the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a Semiconductor Research Corporation program co-sponsored by DARPA. 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.

R. Yokota et al. (Eds.): ISC 2018 Workshops, LNCS 11203, pp. 448–454, 2018.
https://doi.org/10.1007/978-3-030-02465-9_32

accelerators tuned to computation with particular characteristics. Perhaps the most successful example of this trend is the widespread adoption of graphics processing units (GPUs) for more general data-parallel compute tasks. Figure 1 shows two such systems with two CPUs and four GPUs; an IBM S822LC for High Performance Computing [4], and an IBM AC922 [3].

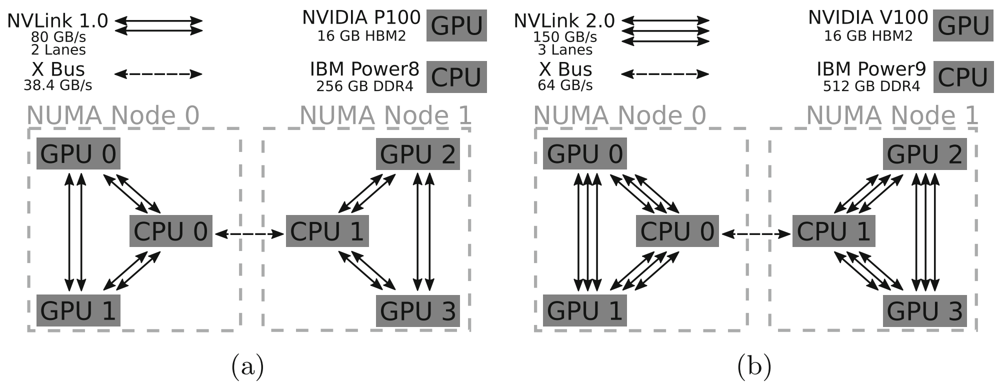

Fig. 1. Summary of the examined systems. (a) shows S822LC, a POWER8-based system with NVLink 1.0 running `nvcc` 9.1.85, CUDA driver 390.31, linux kernel 4.4.0-96. (b) shows AC922, a POWER9-based system with NVLink 2.0 running `nvcc` 9.2.88, CUDA Driver 396.26, linux kernel 4.14.0-49.

The enormous compute capability of GPUs demands high-bandwidth access to data to "feed the beast." The GPUs have high-bandwidth memories (732 GB/s and 900 GB/s on P100 and V100 respectively) to help provide this data, but relatively slow interconnects. A consequence of this architecture is that moving data into accelerator memory to support high-performance execution is a first-order design consideration for any accelerated application. This must be managed by either the application developer explicitly, or by the programming system. In either case, understanding the communication capabilities exposed by the system is foundational to building high-performance applications.

The application does not interface directly with the hardware, but through software abstractions such as the Linux Non-Uniform Memory Access [1] (NUMA) system or the NVIDIA Compute Unified Device Architecture [2] (CUDA) programming system. These software abstractions expose a set of logical communication capabilities on top of the hardware. As this report demonstrates, these capabilities are substantially affected by the underlying hardware, but have distinct performance profiles in their own right. This report describes some initial results obtained while developing an automated approach for understanding how applications use these systems and the capabilities the systems provide.

The rest of this report is organized as follows: Sect. 2 reports initial measurements of data transfer using `cudaMemcpy`. Section 3 reports initial measurements of data transfer bandwidth using CUDA managed memory. Section 4 discusses future work and concludes.

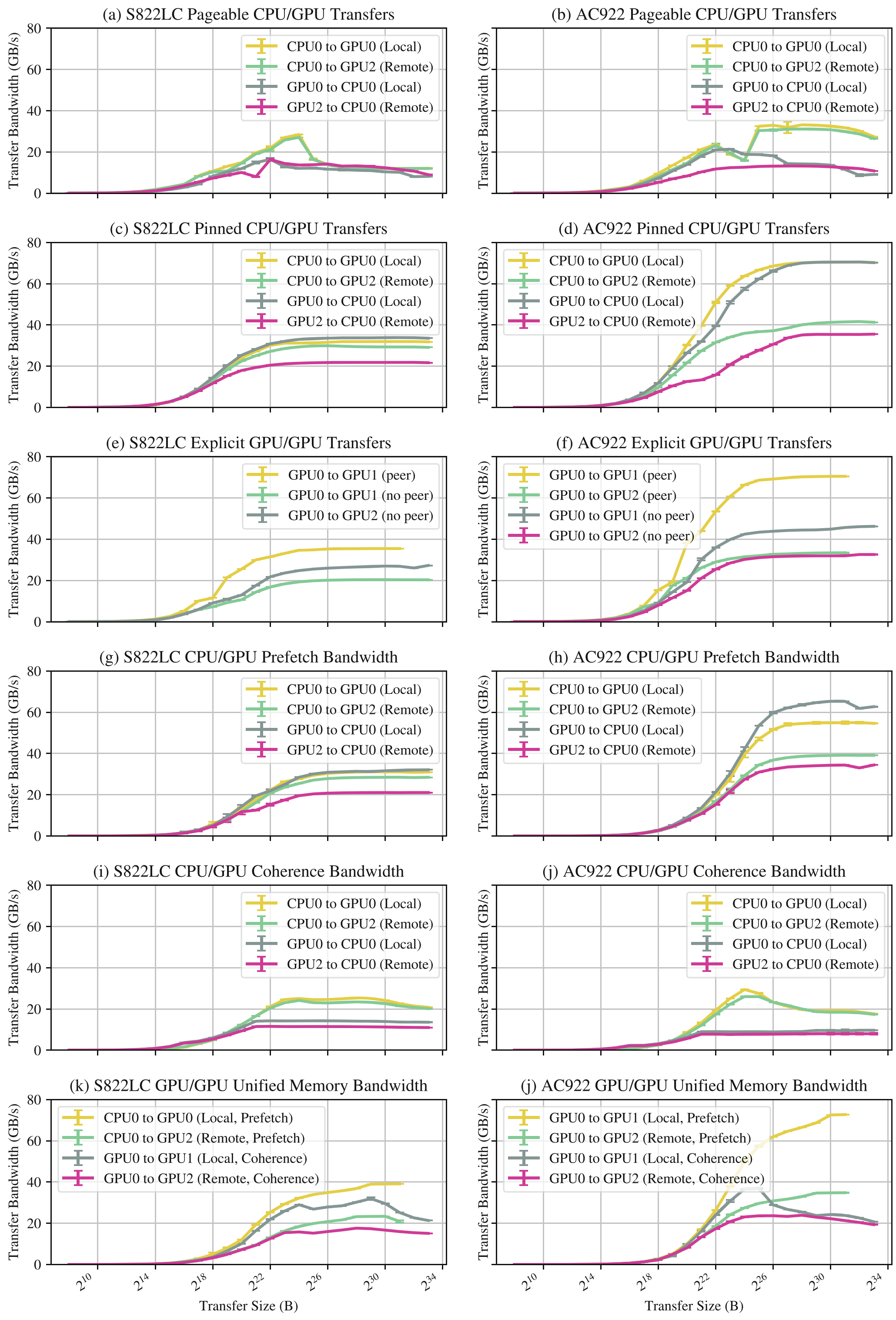

Fig. 2. Measured transfer bandwidths for explicit and unified-memory transfers for S822LC and AC922. Transfer bandwidth vs. transfer size is shown. Each measurement point is the average of at least five iterations, repeated five times. Error bars show the standard deviation of the five average measures.

2 Explicit Data Transfer

The microbenchmark results presented in this report are available at `microbench` [7], a set of microbenchmarks developed using Google's Benchmark support library [5].

Explicit CPU-GPU transfers are caused by the `cudaMemcpy` family of functions being invoked on one pointer to a host allocation and one pointer to a device allocation. The host allocation may be pageable (created by `malloc` or `new`), or pinned (created by `cudaMallocHost` or `cudaHostAlloc`). The device allocation is created by `cudaMalloc`. Algorithm 1 demonstrates a Google Benchmark loop for CUDA operations. The explicit transfers are all asynchronous `cudaMemcpAsync` operations. In the setup phase of the benchmark, a source and destination allocation are created on the host and device. Execution is pinned to the correct host and device through `libnuma` [1,8] and/or `cudaSetDevice` APIs, respectively. For CUDA operations that are asynchronous with respect to the host, CUDA events are used to accurately measure the operation time without measuring other system overhead. For synchronous operations, the operating system wall time is used to measure the operation. In the *teardown* phase, resources are freed.

Algorithm 1. Measuring transfer time of *bytes* between *src* and *dst* for asynchronous and synchronous CUDA operations using Google Benchmark support library.

```
1:  function ASYNC_BANDWIDTH($dst$, $src$, $bytes$)
2:      ...                                              ▷ benchmark setup
3:      for state do                                     ▷ Google Benchmark loop
4:          cudaEventRecord($start$)
5:          asynchronousTransfer($dst$, $src$, $bytes$)      ▷ timed asynchronous operation
6:          cudaEventRecord($stop$)
7:          $millis \leftarrow$ cudaEventElapsedTime($start$, $stop$)
8:          state.SetIterationTime($\frac{millis}{1000}$)         ▷ record time
9:      end for
10:                                                      ▷ benchmark teardown
11: end function

12: function SYNC_BANDWIDTH($dst$, $src$, $bytes$)
13:     ...                                              ▷ benchmark setup
14:     for state do                                     ▷ Google Benchmark loop
15:         state.PauseTiming()
16:         ...                                          ▷ per-iteration setup
17:         state.ResumeTiming()
18:         synchronousTransfer($dst$, $src$, $bytes$)       ▷ timed synchronous operation
19:     end for
20:     ...                                              ▷ benchmark teardown
21: end function
```

Figures 2a–f shows measured transfer rates. The NVLink 2.0 bandwidth of 150GB/s on AC922 compared to the 80 GB/s NVLink 1.0 bandwidth on S822LC is reflected in higher transfer rates on AC922 in most cases. This is especially evident in pinned local transfers or local GPU-GPU transfers. In the GPU-GPU transfers, peer access refers to whether the GPUs are configured to support direct DMA between GPUs. Without peer access, a GPU-to-GPU transfer is implemented with a GPU-to-CPU transfer followed by a CPU-to-GPU transfer, effectively halving the bandwidth observed by the application. On S822LC, remote GPUs do not support peer access (Fig. 2e). Underscoring the performance complexities in these systems are the direction- and affinity-dependent performance seen in almost all CPU/GPU transfers involving pageable and pinned allocations. There is also a strong affinity effect in GPU-GPU transfers when peer access is enabled.

3 Unified Memory

CUDA Unified Memory allows allocations to be transparently referenced by the CPUs and GPUs [6]. Instead of the programmer explicitly moving data, the CUDA system is responsible for ensuring accessed data is present on the correct device. On both these systems, the unified memory allocations operate at page granularity. A "coherence" or "demand" migration occurs when an accessing device does not have the page in its local memory, and that page must be moved from the currently owning device. A "prefetch" migration occurs when pages are bulk-migrated to a device ahead of their use.

Algorithm 1 shows the asynchronous and synchronous measurements used for unified memory benchmarks. In the *setup* phase, a single unified memory allocation is created, which is accessible from any device. `cudaMemPrefetchAsync` is used to move the allocation's backing pages to the source or destination device in the *per-iteration setup* phase. The timed operation is `cudaMemPrefetchAsync` to generate prefetch transfers, or CPU/GPU kernels to generate demand transfers.

Figures 2g–l show example results for CPU/GPU and GPU/GPU transfers, for both prefetch and demand migrations. Prefetch bandwidth is capable of achieving nearly the same performance as pinned memory transfers; demand migration bandwidth is substantially lower for large transfers. The GPU-to-CPU demand transfer rate is limited by the rate that a single CPU thread can generate loads. Prefetch transfers exhibit some performance variation based on device affinity and direction of transfer. Coherence overhead limits CPU/GPU demand transfer bandwidth and these transfers do not show strong correlation with device affinity.

3.1 Page Fault Latency

Unified memory page fault latency is estimated by constructing a linked list in managed memory, forcing it to be migrated to the source device, and executing a single-threaded CPU or GPU kernel on the destination device to traverse it.

The stride between linked-list elements is large enough to avoid the effects of prefetching. Each access to the list incurs a page fault. The incremental change in function execution time as the number of strides increases is therefore an approximate measure of the page fault latency. Table 1 summarizes the estimated page fault latencies. There is no substantial difference in page fault latencies for different CPUs in the same system, so values for faults involving CPU0 are shown. AC922 is slower than S822LC in all categories.

Table 1. Measured page-fault latencies.

Page Fault	Latency (μs)	
Type	S822LC	AC922
CPU $\rightarrow$ GPU	14.9	24.1
CPU $\leftarrow$ GPU	13.6	27.4
GPU0 $\leftrightarrow$ GPU1 (local)	25.5	38.0
GPU0 $\leftrightarrow$ GPU2 (remote)	28.8	41.5

4 Conclusion

This report presents initial results of a system interconnect characterization effort. Other relevant transfer bandwidths under investigation include CPU-to-CPU, CUDA remote mappings, system atomics, GPU-direct I/O and network data transfers, and more detailed characterizations of the communication methods discussed in this report. A basis set of these algorithms will be used to create a microbenchmark suite. This suite could be leveraged to sanity-check system configuration during firmware development and help eliminate performance bottlenecks caused by incorrect parameters or implementation heuristics.

Furthermore, the NUMA and CUDA libraries present a logical abstraction of the system communication. To create accurate performance models, a relationship between the logical communication paths and underlying hardware should be established. This work will be expanded to enumerate the underlying hardware through the operating system, and observe traffic on that hardware to establish the logical-to-physical mapping. Tying the underlying hardware to the observed bandwidths allows the appropriate performance models to be automatically generated. Those models may then be utilized to create NUMA and multi-GPU topology-aware communication runtimes and allocators. Those models may also be used for architecture studies to understand the high-level architectural tradeoffs for affecting application performance.

References

1. NUMA(3) Linux Programmer's Manual (August 2007)
2. Cuda c programming guide (Nov 2017)
3. Caldeira, A.B.: Ibm power system ac922 introduction and technical overview. IBM Redbooks (2018)
4. Caldeira, A.B., Haug, V., Vetter, S.: Ibm power system 822lc for high performance computing introduction and technical overview. IBM Redbooks (2016)
5. Google: Benchmark. https://github.com/google/benchmark (2018)
6. Harris, M.: Unified memory in cuda 6 (2013), https://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/
7. Pearson, C., Dakkak, A., Li, C.: microbench. https://github.com/rai-project/microbench (2018)
8. Wickman, C., Lameter, C., Schermerhorn, L.: numactl v2.0.11. https://github.com/numactl/numactl (2015)