HETEROGENEOUS SYSTEM AND APPLICATION COMMUNICATION MODELING

BY

CARL PEARSON

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

Adviser:

Professor Wen-Mei Hwu

ABSTRACT

With the end of Dennard scaling, high-performance computing increasingly relies on heterogeneous systems with specialized hardware to improve application performance. This trend has driven up the complexity of high-performance software development, as developers must manage multiple programming systems and develop system-tuned code to utilize specialized hardware. In addition, it has exacerbated existing challenges of data placement as the specialized hardware often has local memories to fuel its computational demands. In addition to using appropriate software resources to target application computation at the best hardware for the job, application developers now must manage data movement and placement within their application, which also must be specifically tuned to the target system. Instead of relying on the application developer to have specialized knowledge of system characteristics and specialized expertise in multiple programming systems, this work proposes a heterogeneous system communication library that automatically chooses data location and data movement for high-performance application development and execution on heterogeneous systems. This work presents the foundational components of that library: a systematic approach for characterization of system communication links and application communication demands.

To my family, for their love and support.

ACKNOWLEDGMENTS

I would like to thank my thesis advisor, Professor Wen-Mei Hwu. He allowed me to make this my own work, but his door was always open when I needed guidance.

I would also like to thank the members of the IMPACT research group for their insight and assistance, and for creating such a fruitful research environment.

I would also like to thank Isaac Gelado of Nvidia for his technical insights that contributed to the results described in this thesis.

I would also like to acknowledge Dominic Grande who collaborated extensively in the development of the software tools described in this thesis.

Finally, this work is supported by the following two entities:

- The IBM-Illinois Center for Cognitive Computing Systems Research (C^3SR) - a research collaboration as part of the IBM Cognitive Horizon Network.

- 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.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF CODE LISTINGS

LIST OF ALGORITHMS

LIST OF ABBREVIATIONS

CUDA	Compute Unified Device Architecture
FPGA	field-programmable gate array
GPU	graphics processing unit
NUMA	non-uniform memory access
RAM	random-access memory
SIMD	single-instruction multiple-data
SMP	symmetric multi-processing

CHAPTER 1

INTRODUCTION

With the end of Dennard scaling, computer architects have sought to satisfy demand for increasing performance by providing specialized hardware 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. With the success of GPUs as a template, architects are moving forward with a wide variety of accelerators, such as SIMD extensions [1, 2, 3], AI accelerators (Google tensor processing unit [4], Huawei Neural Processing Unit [5], IBM neuromorphic chips [6], Intel Nervana [7]), motion coprocessors (Apple M-series [8]), field-programmable gate arrays (Xilinx Virtex [9], Intel Stratix [10]), network processors (Netronome Agilio [11]), digital signal processors (Qualcomm Hexagon [12], NXP DSP56xx Family [13]), vision processing units (Eyeriss [14], Movidius VPU [15], Mobileye EyeQ [16], Microsoft Holographic Processing Unit [17]) and many others.

The enormous compute capability of accelerators demands high-bandwidth access to data to "feed the beast." Without this access, the performance potential of the accelerator is largely wasted waiting for data. The trend of *integration* (also motivated by reduction of total system cost) where semiconductor die-size or power limits allow, has provided one approach to solving this problem. By integrating an accelerator onto the same die as the CPU, the accelerator more easily gets high-bandwidth low-power access to data shared with the CPU. For accelerators with high memory demands, however, the system memory DRAM bandwidth may ultimately limit performance.

The second approach is to provide accelerators with their own high-performance memory. Unfortunately, managing this memory then falls upon runtime systems or the application developer, and moving data into accelerator memory to support high-performance execution is a first-order design

consideration for any accelerated application. The data-placement and data-movement challenge is exacerbated by the growing demand for data-driven applications. Analytics and neural-network applications ingest huge amounts of data, and even if the amount of compute per data element is small, the aggregate required computation can be commensurately large. That motivates developers to use accelerators for these applications. To achieve high performance on accelerators, developers must marshal and coordinate their data movement and computation.

This work describes an automated approach to analyzing the performance of data movement in systems that use discrete accelerators with local memories. Broadly, the approach consists of two components: a system characterization tool, which enumerates and characterizes the performance of logical communication paths, and an application characterization tool, which profiles unmodified applications to record how they interact with the system. These tools are examined in the context of heterogeneous systems made of CPUs and Nvidia GPUs and machine-learning workloads due to the maturity of that hardware/software ecosystem. Together, these tools provide a foundation for automating analysis of the relationship between system design and application performance.

In pursuit of that vision, this work makes the following contributions:

- a detailed communication performance characterization of three multi-CPU/multi-GPU systems designed for data-driven applications (Chapters 4 and 5)
- a novel hardware enumeration tool for enumerating undirected graph hardware topologies in multi-CPU/multi-GPU systems (Chapter 3)
- an approach for combining this characterization with an application characterization to understand application performance on modern accelerator-heavy systems (Chapter 6)

The rest of this document is organized as follows: Chapter 2 describes background information on the studied computers, the CUDA programming system, Linux NUMA system, OpenMP, and profiling tools proposed for the application characterization; Chapter 3 describes the hardware system characterization approach; Chapter 4 describes performance characterization of explicit CUDA memory management. Chapter 5 describes performance

characterization of unified memory in CUDA systems. Chapter 6 describes future work of application characterization and combined modeling. Chapter 7 discusses related work; and finally, Chapter 8 concludes.

CHAPTER 2

BACKGROUND

This work examines the relationship between system and application performance in the context of systems comprised of CPUs and Nvidia GPUs. To that end, Section 2.1 describes the relationship between software abstractions and the underlying system hardware, Section 2.2 discusses how Nvidia graphics processing units (GPUs) fit into the computing system architecture, Section 2.3 describes the PCIe and NVLink interconnect systems used to couple GPUs to each other and the rest of the system, Section 2.4 details communication-related components and APIs of CUDA, the Nvidia GPU programming system, Sections 2.5 and 2.6 describe the Linux non-uniform memory access (NUMA) and OpenMP multiprocessing systems, Section 2.7 describes the Linux and CUDA components used in the proposed application profiler, and Section 2.9 documents the heterogeneous systems used as case studies in this work.

2.1 System Communication Abstraction

Application access to computing systems is made through a stack of abstractions. This work considers applications that interface with the system through CUDA, the Linux NUMA abstraction, and OpenMP. Those API calls are implemented in various libraries, such as `libcudart.so` on Linux systems. In turn, those libraries make use of system calls provided by the operating system. The operating system interfaces with various drivers, including the Nvidia GPU drivers and the interconnect drivers, to make use of the underlying hardware. Figure 2.1 shows a schematic of this stack.

Through these layers of abstraction, the underlying communication capabilities are modified. For example, PCIe interconnects transfer data with packets, but the CUDA API does not expose the ability to create custom

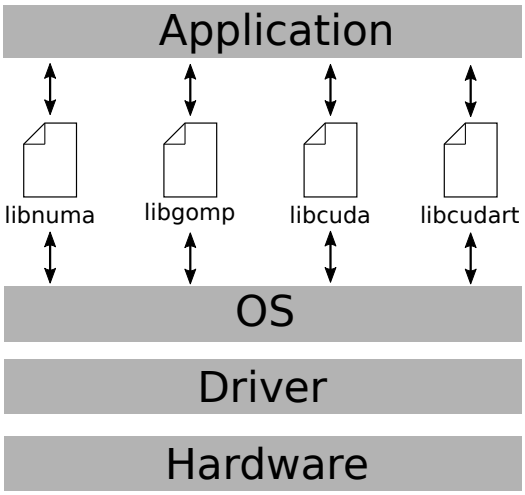


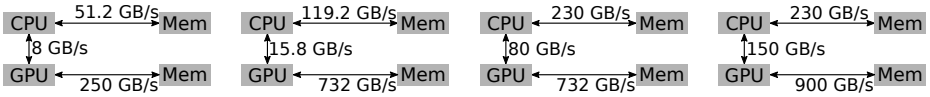
Figure 2.1: The application interacts with the underlying hardware through libraries, the operating system, and drivers.

packets to send to the GPU. Furthermore, the performance of the hardware is modified. For example, when CUDA unified memory is used, some of the link bandwidth may be consumed by control signals that help ensure data coherence. Chapters 4 and 5 show how different uses of the CUDA API can achieve different performance on the same physical hardware.

2.2 GPUs and System Architecture

Nvidia GPUs used in high-performance computing are fully-discrete accelerators. From the software side, they demand explicit management through the Nvidia Compute Unified Device Architecture (CUDA) programming system: a set of C++ language extensions and libraries. In a GPU-accelerated application, the CPU typically acts as a manager and "offloads" specialized compute tasks to the GPU. From the hardware perspective, GPUs are separated from the CPU and memory by an interconnect link. The GPU has its own local memory, which must be populated with data for the GPU to operate on. For the rest of this thesis, we will refer to the system memory associated with that CPU as the "CPU memory" or "system memory", and the GPU's local memory as "GPU memory". Finally, though not covered in this work, the GPU compute cores have dramatically different performance characteristics than CPU cores, and require specialized programming styles.

Figure 2.2 shows some example interconnect bandwidth numbers for some



(a) (b) (c) (d)

Figure 2.2: Representative interconnect bandwidths between CPU and system memory, GPU and GPU memory, and CPU to GPU. (a) shows a c.2011 x86 system with an AMD Operton 6200 CPU [18] system, PCIe 2.0 x16, and Nvidia K20 GPU. (b) shows a c.2017 x86 system with an Intel Xeon Platinum 8180M [19] CPU, PCIe 3.0 x16, and an Nvidia P100 GPU. (c) shows a c.2013 PowerPC system with an IBM Power8 [20] CPU, two-lane NVLink 1.0, and an Nvidia P100 GPU. (d) shows a c.2017 PowerPC system with an IBM Power9 [21] CPU, three-lane NVLink 2.0, and an Nvidia V100 GPU.

hypothetical systems. In all cases, the bandwidth between the CPU or GPU and their respective memories is much higher than the bandwidth between the CPU and GPU. In older systems and current x86 systems, that link may be an order of magnitude slower than the other interconnects. The vastly different link performance makes data placement extremely important for application performance.

Figure 2.2a is representative of a node of a supercomputer designed in 2012 [22]. Figure 2.2b represents a similar system, with components updated to 2017. Figure 2.2c is a c.2013 system with an IBM Power8 CPU and NVLink 1.0. Figure 2.2d represents next-generation interconnect bandwidths present in the Summit [23] and Sierra [24] supercomputers delivered in 2018.

2.3 PCIe and NVLink Interconnects

Modern systems with discrete GPUs feature either NVLink or PCIe accelerator interconnects. These interconnects couple the GPUs to the CPUs and/or other GPUs. Table 2.1 summarizes the theoretical bandwidth of common interconnect configurations. Figure 2.3 shows example PCIe and NVLink topologies. Section 2.4.4 describes how these topologies affect parts of the CUDA peer-communication API.

The Nvidia DGX-1 system (Section 2.9.1) uses PCIe to connect CPUs to GPUs, and single-lane NVLink 1.0 to connect amongst GPUs. The IBM S822LC system (Section 2.9.2) uses two NVLink 1.0 lanes to connect pairs

Table 2.1: Theoretical performance for common interconnect configurations. Only NVLink configurations used in the case studies are shown below. PCIe 3.0 x16 is included for reference, as most PCIe 3.0-attached GPUs use 16 lanes.

Interconnect	Bandwidth	Year	Architecture
PCIe 3.0	15.8 GB/s (16 lanes)	2012	Tree
NVLink 1.0 / NVHS	20 GB/s (1 lanes)	2016	Point-to-Point
NVLink 1.0 / NVHS	40 GB/s (2 lanes)	2016	Point-to-Point
NVLink 2.0 / NVHS	75 GB/s (3 lanes)	2017	Point-to-Point

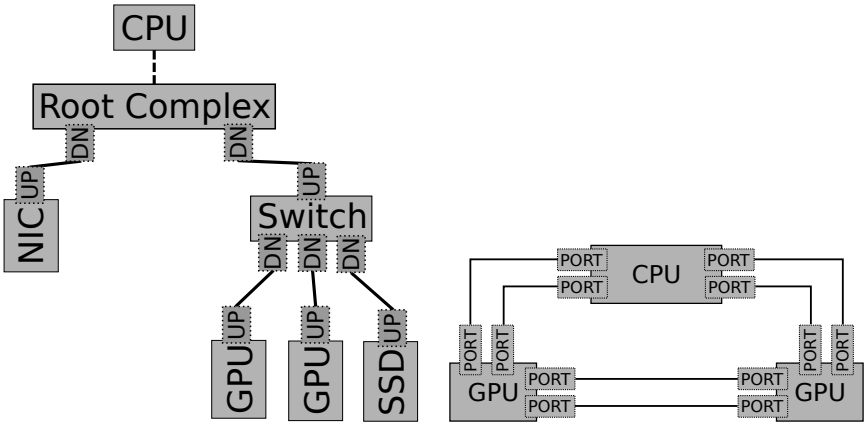


Figure 2.3: (a) An example PCIe topology, showing the root complex at the root of the tree, with endpoints and switches connected by links. (b) An example NVLink topology, with each device supporting four NVLinks, and using two to connect to each of its neighbor devices.

of devices, with each lane operating at 20 GB/s. The IBM AC922 system (Section 2.9.3) uses three NVLink 2.0 lanes to connect pairs of devices, with each lane operating at 25 GB/s.

2.3.1 PCIe

Peripheral Component Interconnect Express (PCIe) is an expansion bus standard [25]. PCIe components form a tree, rooted at the *root complex*. PCIe devices such as GPUs or SSDs are *endpoints*, with a single upstream port and no downstream ports. Data is sent point-to-point between participating endpoints and/or the root complex via unique PCIe addresses. The topology also may include *switches*, components with a single upstream port and

multiple downstream ports. Switches do not receive data packets, and are only used to extend the topology. Components are connected by PCIe *links*, each of which consists of one to 32 lanes. The total bandwidth of the link depends on the PCIe generation and the number of lanes. Figure 2.3a shows an example topology. The PCIe 3.0 specification [26] was finalized in 2010, and the first Nvidia GPUs to support it were some Kepler-architecture products released in 2012 [27].

Table 2.2 shows single-lane PCIe bandwidths from generation 1.0 to 3.0. Physically, each PCIe lane has four wires, divided into two differential signaling pairs. A PCIe 3.0 x16 interconnect therefore has 64 wires. In PCIe 3.0, each signaling pair operates at 8 Gb/s. With a 128b/130b encoding, this gives PCIe 3.0 x16 a 15.8 GB/s unidirectional bandwidth.

Table 2.2: PCIe lane transfer rates

PCIe Revision	Signaling Rate	Encoding	Line Bandwidth
1.0	2.5 GT/s	8b/10b	250 MB/s
2.0	5.0 GT/s	8b/10b	500 MB/s
3.0	8.0 GT/s	128b/130b	984.6 MB/s

2.3.2 NVLink

NVLink is a communication protocol developed by Nvidia. Current NVLink implementations use the proprietary high-speed signaling interconnect NVHS [28]. Like PCIe, each NVLink connects two devices. Unlike PCIe, there is no concept of upstream or downstream ports, and devices may have multiple ports. Multiple links may connect devices, in which case the links are combined to contribute to the available bandwidth between devices. Figure 2.3b shows an example topology. The NVLink 1.0 systems in this work allow each device to have four NVLink lane connections, with each lane running at 20 GB/s [29], [30]. The NVLink 2.0 systems in this work allow each device to have six NVLink lanes at 25 GB/s each [31].

Physically, each bidirectional lane has 32 wires, divided evenly into two unidirectional sublinks of eight differential pairs [28]. Each pair operates at 20 Gb/s for NVLink 1.0 and 25 Gb/s for NVLink 2.0. This means a 2-lane NVLink 1.0 has 64 wires. The improved bandwidth of 2-lane NVLink 1.0 vs. PCIe 3.0 is directly attributable to the signaling rate on the wires.

2.4 CUDA

Nvidia's CUDA (Compute Unified Device Architecture) is a programming system for enabling general-purpose computation on Nvidia GPUs (graphics processing units). CUDA is a set of C extensions and libraries for interfacing with GPUs. Nvidia provides a compiler, `nvcc`, for generating CUDA-enabled binaries.

CUDA provides a set of runtime and driver APIs for the developer to manage the allocation and movement of data between the host and device memory. From its inception, CUDA provided comprehensive APIs for developers to mitigate application performance shortfalls stemming from the relatively limited performance of host-device communication links. As the capabilities of GPUs and host systems have improved, CUDA has provided simpler, higher-level APIs that require less programmer effort. This section describes CUDA memory-management capabilities and the historical context of their introduction.

The CUDA API has an associated version number that defines which CUDA actions are made available by that API. CUDA-capable hardware advertises a specific compute capability (CC) that defines what CUDA actions are supported by that GPU. Although the CUDA API may expose particular capability, the GPU may need a sufficiently high CC to take advantage of it. This section references both these version systems when discussing CUDA features.

This work focuses on the performance of explicit CUDA memory management and the CUDA unified memory system. There is a third set of transfer capabilities that fall under the umbrella of remote-mapping or "zero-copy" memory. These techniques are subsumed by unified ,emory, though making use of them typically requires hints to the CUDA system. Once the flexibility of the unified memory system improves, it will be important to revisit the performance implications of zero-copy memory.

2.4.1 Basic CUDA Memory and System Memory

Prior to the introduction of unified virtual addressing (see Section 2.4.3), the CUDA memory space was composed of multiple address spaces: one for the host, and one for each GPU [32]. Data was explicitly allocated

in those address spaces through `cudaMalloc`. Standard C/C++ memory allocation techniques (`malloc`/`new`, `free`/`new`) are used for managing memory on the host. `cudaMemcpy` is used to move data between address spaces, whether the host and device or between devices. Since each device has a separate address space, the programmer explicitly instructs `cudaMemcpy` how to move data with `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, or `cudaMemcpyDeviceToDevice`. Basic CUDA memory management runtime calls are described in Table 2.3.

Table 2.3: Basic CUDA and C/C++ memory-management APIs.

CUDA API	**Description**
`cudaSetDevice()`	Associate a device with the host thread.
`cudaMalloc()`	Allocate memory on the device.
`cudaFree()`	Free memory on the device.
`cudaMemcpy()`	Copy data between to,from,and between GPUs.
`cudaMemcpyPeer()`	Copy data between GPUs without CPU involved.
C/C++ API	**Description**
`new / malloc()`	Allocate pageable memory on the system heap.

`CudaMemcpy` is only partially asynchronous with respect to the host ([33], ch. 1). It will return once the pageable source allocation is safe to modify, but possibly before the data has finished moving to the device. This impacts the design of the performance characterization routines in Chapter 4. `CudaMemcpyPeer` initiates a DMA copy from one GPU to another, without involving the host. This can improve the bandwidth of the transfer between supported devices. The impact of this peer access is also described in Chapters 4 and 5.

This API definition imposes the following basic structure on all CUDA programs, which remains essentially unchanged through CUDA 9.1.

1. Allocate memory on the host with `new`/`malloc`.
2. Initialize memory on the host.
3. Allocate memory on the device with `cudaMalloc`.
4. Copy initialized data from the host to the device with `cudaMemcpy(..., cudaMemcpyHostToDevice)`.
5. Launch CUDA kernels.

6. Copy results back to the host with `cudaMemcpy(..., cudaMemcpyDeviceToHost)`.

7. Free CUDA allocations with `cudaFree`.

Unified virtual addressing (Section 2.4.3) and unified memory (Section 2.4.5) allow the data transfer steps to happen implicitly on supported systems.

2.4.2 Page-Locked Memory

The GPU uses direct memory access (DMA) to copy data to and from the host. When `cudaMemcpy` is invoked, the CPU instructs the GPU to copy a region of host memory to the device memory (or vise-versa), without the CPU being involved. The host must guarantee that the memory to be accessed by the GPU will not be paged-out during the copy. First, `cudaMemcpy` copies the data from the application address space to a piece of page-locked memory in the system memory managed by the CUDA driver, and then `cudaMemcpy` instructs the GPU to initiate the DMA from that page-locked region to the GPU memory.

The CUDA runtime functions in Table 2.4 are the core functions CUDA provides to make page-locked memory regions directly visible to the application. When the application uses these APIs, the first copy from pageable host memory to page-locked host memory can be elided. Section 4.2.1 demonstrates the performance improvement from skipping this first copy. Overuse of page-locked memory on the host will degrade overall application performance or even impact system stability if the host system is not able to page as needed.

`CudaHostAlloc` allows even more options, including the `cudaHostAllocPortable` and `cudaHostAllocWriteCombined` flags. `CudaHostAllocWriteCombined` causes a pinned allocation to be write-combined. Writes to write-combined memory may be delayed and combined in a buffer to reduce the number of memory accesses. Additionally, the host may not cache this data in L1 or L2 cache, freeing up those resources for other applications. This may prevent unnecessary cache invalidations from occurring during the DMA. Furthermore, coherency is not enforced, so data is not snooped on the PCIe bus, which can increase bandwidth by up to 40% ([35], 3.2.5.2). Due to the possible lack of caching, this type of allocation

Table 2.4: CUDA pinned memory-management APIs.

API	Description	CUDA Version Introduced
`cudaMallocHost()`	allocate page-locked memory on the host	1.0 [34]
`cudaFreeHost()`	free page-locked memory on the host	1.0 [34]
`cudaHostAlloc()`	cudaMallocHost with additional options	3.0 [35]
`cudaHostRegister()`	Page-lock a range of host memory	4.0 [36]

makes sense for data that is not frequently read by the CPU, for example, data written once by the CPU before being sent to a GPU.

`CudaHostAllocPortable` allows all CUDA contexts to treat the memory as pinned, not just the context that performed the allocation. This became the default with the introduction of unified virtual addressing.

2.4.3 Unified Virtual Addressing

Unified virtual addressing was introduced with compute capability 2.0 GPUs and CUDA 4.0 on 64-bit systems. The host memory and the memory of each GPU are mapped into disjoint subsections of a single unified address space.

This enhancement simplifies several of the already-introduced CUDA memory management commands ([36], 3.2.7). The `cudaMemcpyDefault` flag for `cudaMemcpy` instructs the CUDA system to automatically determine how to move data. By examining the address of the pointers passed to `cudaMemcpy`, CUDA determines where which device the memory resides on and moves it accordingly. This simplifies the programmer's use of `cudaMemcpy`, as `cudaMemcpyDefault` may be used everywhere. There is also no need to call `cudaHostGetDevicePointer` for mapped allocations. Furthermore, all mapped allocations are automatically accessible by all GPUs in the system, not restricted to the GPU that was active at the time of the allocation. `CudaMemcpyPeer` is no longer needed for device-to-device memory copies; `cudaMemcpy` may be used instead.

2.4.4 Peer Access and UVA

Peer access was introduced with CUDA 4.0. `CudaDeviceEnablePeerAccess()` allows CC 2.0+ devices to address memory in another device's address space ([36], 3.2.6.4). For example, if GPU1 loads through a pointer to data on GPU0, the data will be directly fetched from GPU0 memory, at the cost of one PCIe transaction and one global memory load [32], and be cached in the L2 of GPU0. Direct peer access requires compute capability 2.0, CUDA 4.0, Fermi+, and 64-bit system. The availability of peer access may rely on a combination of interconnect topology and system hardware, summarized in Table 2.5. When peer access is disabled, data transfers first pass through the host instead of allowing direct DMA between devices.

Table 2.5: GPU-to-GPU connection, and whether peer access is supported for the systems considered in this work.

Between GPUs with...	**System**	**Peer Access**
...a shared PCIe switch	DGX-1	✓
...different PCIe switches	DXG-1	×
...a direct NVLink connection	S822LC	✓
...no direct NVLink connection	S822LC	×
...a direct NVLink connection	AC922	✓
...no direct NVLink connection	AC922	✓

2.4.5 Unified Memory with CC 3.0+ (Kepler+)

The CUDA unified memory system was introduced with CUDA 6.0 and requires a GPU with SM architecture of 3.0 or higher [37]. CUDA unified memory [38] provides a single pool of memory that is accessible from the CPU and GPU by a single pointer. CUDA automatically migrates data between the physically distinct CPU and GPU memory as needed, allowing GPU kernels to access the memory as if it were in the global memory, and CPU functions to access the memory as if it were in the system memory. Like mapped memory, this simplifies programming by removing the need for separate host and device allocations and explicit data transfers. A summary of unified memory APIs is shown in Table 2.6.

The underlying data is only present in one location on the system, and in principle, unified memory allocations are automatically migrated towards

Table 2.6: CUDA unified memory-management APIs. Initial CUDA 6.0 APIs and additional CUDA 8.0 APIs are shown.

CUDA 6 and CC3.0+	**Description**
`__managed__`	Defines a global variable in managed memory
`cudaMallocManaged()`	allocate a unified memory region.
`cudaStreamAttachMemAsync()`	Attach a managed allocation to a stream, instead of globally.
CUDA 8 and CC6.0+	**Description**
`cudaMemPrefetchAsync()`	Hint to prefetch memory to device
`cudaMemAdvise()`	Hint about how memory will be used

their most recent use. When a kernel is launched, all pages attached to that kernel's stream are bulk migrated to the destination GPU. When the host program touches a page, that page is migrated back to the system memory. In multi-GPU systems, data does not migrate between GPUs - all other GPUs receive peer mappings to the data, which is accessed over the PCIe bus ([37], J.1.3).

Unified memory maintains coherence (i.e., all GPUs and the CPU have the same view of unified memory values) by disallowing concurrent accesses to managed data, including concurrent access to distinct managed allocations ([37], J.2.2.1). The CPU may access managed allocations after GPU execution has completed, where "GPU execution" refers to activity in a particular stream for stream-attached memory, or whole-GPU otherwise. For stream-attached memory, completion of GPU execution can be guaranteed by any stream-synchronizing call. For whole-GPU memory, completion is guaranteed by stream synchronization when only one stream is executing on the GPU,[1] or by any call that is fully synchronous with respect to the host.[2] The GPU is considered to be active even if it is not accessing managed data. Concurrent inter-GPU accesses are allowed, as are concurrently-executing kernels on a single GPU ([37], J.2.2.2).

[1] e.g., `cudaStreamSynchronize()`
[2] e.g., `cudaDeviceSynchronize()`

2.4.6 Unified Memory with CC 6.0+ (Pascal+)

With CUDA 8 and for GPUs with CC 6.0+, GPU page faulting provides a more fine-grained data transfer mechanism [39]. Instead of moving all managed allocations to the GPU prior to a kernel launch, the GPU will fault if it accesses a page that is not in its memory. The page is either migrated to the GPU to serve that access, or the page is mapped into the GPU address space to be accessed over the host-device interconnect. Unlike unified memory with CC 3.0, pages can migrate between GPUs on peer accesses ([39], J.1.4). The GPU page faulting mechanism lifts all restrictions on simultaneous access to data ([39], J.2.2.1). However, intensive interleaving of CPU and GPU accesses to a page can cause excessive migrations and result in severe performance degradation.

CC 6.0+ also brings 49-bit virtual addressing to cover the 48-bit virtual addressing of modern CPUs and the GPU memory. This allows CUDA to support managed allocations larger than the GPU memory. The total amount of managed allocations still cannot be larger than the system memory ([39], J.1.3).

`CudaMemPrefetchAsync()` hints to the unified memory system that a particular device will soon be accessing a unified memory allocation. This may cause the system to migrate the specific region of memory over to the referenced device. This hint is used in Chapter 4 in some of the unified memory characterizations.

`CudaMemAdvise()` hints to the unified memory system how a particular region of memory can be used. `CudaMemAdviseSetReadMostly` causes the hinted device to establish a read-only copy of a page, instead of taking complete ownership of a page on access. Any writes to that page become expensive, as all read-only copies must be invalidated. This cost is not evaluated in this work. `CudaMemAdviseSetPreferredLocation` hints to the driver that data migration of the page away from the device should be avoided. This may cause the system to establish a remote mapping for the data, instead of migrating the page. `CudaMemAdviseSetAccessedBy` hints to the driver that the device will access the memory region. It causes the page to be mapped in the device page table for as long as possible, to prevent page faults on access.

2.4.7 Unified Memory with CC 7.0+ (Volta+)

Though not examined in this work, Volta GPUs contain the necessary hardware for more intelligent migration of pages. When support is added in the Nvidia driver, access counters will be used to trigger page migrations of hot pages, instead of on each access. The system will also detect thrashing, and temporarily prevent page faults to allow faster progress on each device. On POWER9 systems, the CPU and GPU have access to each other's page translation hardware, allowing memory accesses on the CPU to be served from the GPU and cached on the CPU. Furthermore, the CPU can execute atomic operations on locations in GPU memory without a page migration [40].

2.5 Linux Non-Uniform Memory Access

Linux includes a system for exposing non-uniform memory access architectures (NUMA) to applications. In NUMA systems, memory is divided into multiple *nodes* [41]. Processors and devices have the same access characteristics when accessing memory in a particular node. Nodes have *affinity* to processors and devices, indicating the processors and devices which can access that node with the best performance.

This is particularly relevant on multi-socket systems, though some single-socket systems also feature NUMA characteristics. For example, every system considered by this work and described in Section 2.9 is a NUMA system. On the AC922 (Figure 2.7), GPU0 is directly connected to CPU0 and GPU2 is directly connected to CPU1. If an allocation on CPU0 were to be copied to GPU2, the data would traverse the CPU-CPU X bus, and then the CPU-to-GPU NVLink. On the other hand, if an allocation on CPU1 were to be copied to GPU2, that data would only have to traverse the NVLink. This can have a substantial effect on available bandwidth, as shown in Figure 2.4. Chapters 4 and 5 show these effects in more detail.

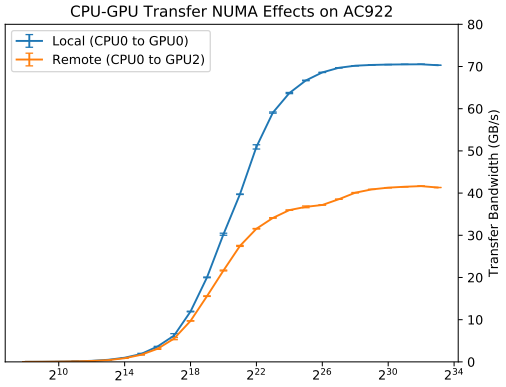


Figure 2.4: Example of NUMA bandwidth effects on AC922

Applications can leverage the numactl [42] library to affect their own NUMA execution policy. This policy controls on which CPUs processes may execute, and on which NUMA nodes memory is allocated. Numactl is used in this work to ensure that data is allocated and programs execute on specific CPUs as needed to exercise different underlying hardware links. Listing 2.1 shows a C++ function that takes the operating system NUMA node ID as an integer argument and binds the executing thread and its allocations to that node. Lines 6 and 7 allocate a `bitmask`, and set the bit corresponding to the NUMA node. `Numa_bind()` in line 8 forces execution and allocations to occur on the nodes in the nodemask.

Listing 2.1: Binding to NUMA nodes.

```
1  static inline void
2  numa_bind_node(const int node) {
3    if (-1 == node) {
4          numa_bind(numa_all_nodes_ptr);
5    } else if (node >= 0) {
6          struct bitmask *nodemask = numa_allocate_nodemask();
7          nodemask = numa_bitmask_setbit(nodemask, node);
8          numa_bind(nodemask);
9          numa_free_nodemask(nodemask);
10   } else {
11         exit(1);
12   }
13 }
```

The benchmarks in this paper also initialize numactl with calls to `numa_set_strict(1)` and `numa_set_bind_policy(1)`, which ensure that NUMA will cause the program to exit on an error, and that if the memory cannot be allocated on

the requested NUMA node, the allocation will fail instead of falling back to a different node.

2.6 OpenMP

OpenMP is a shared-memory processing system that uses compiler directives and library functions to allow applications to implement shared-memory parallel processing techniques [43]. Among other more sophisticated approaches, OpenMP allows hint-assisted parallelization of nested loops through compiler directives. This work uses OpenMP to use multiple CPU threads while transferring data between the CPU and the GPU during the unified memory characterization in Chapter 5. Multiple CPU threads are used to ensure that the CPU is generating sufficient memory traffic to saturate its memory controllers and make the maximal demands of the unified memory system. There is a runtime overhead of entering the parallelized region, which affects the design of the benchmarks.

Many microbenchmarks in this work rely on execution of threads on particular NUMA nodes. The `numa_bind(`*node*`)` function binds the current process and all child processes to the provided node, but OpenMP does not necessarily implement worker threads as children of the current thread. Algorithm 2.1 is used to bind all OpenMP threads to a NUMA node. A full OpenMP thread team is created, and each of those threads individually binds itself to the provided NUMA node.

Algorithm 2.1 Algorithm to bind all OpenMP threads to a NUMA node.

```
1: function OMP_NUMA_BIND_NODE(dev)
2:     bind_cpu(dev)
3:     for all worker threads do
4:         bind_cpu(dev)
5:     end for
6: end function
```

2.7 Profiling Tooling

2.7.1 CUDA Events

An event is a special kind of operation that is performed in a CUDA stream. CUDA events allow the CUDA runtime to set and query noteworthy milestones in the sequence of stream operations without stalling the stream. These events may be used as a lightweight method for measuring how much time CUDA operations take without also measuring the time taken to synchronize the stream. Events are created with `cudaEventCreate()`, destroyed with `cudaEventDestroy()`, and inserted into a stream with `cudaEventRecord()`. `CudaEventSynchronize()` blocks the calling thread until a particular event has been completed in the stream. `CudaEventElapsedTime()` returns the number of milliseconds that have elapsed between two completed events. Listing 2.2 shows an example of measuring time of hypothetical CUDA operation `cudaSomeAsyncOperation()` taking place in the `stream` stream.

Listing 2.2: Measuring time with CUDA events.

```
1  // Declare variables
2  float millis;
3  cudaEvent_t start, stop;
4  cudaStream_t stream;
5
6  // Create events
7  cudaEventCreate(start);
8  cudaEventCreate(stop);
9
10 // Insert events into stream
11 cudaEventRecord(start, stream);
12 cudaSomeAsyncOperation(..., stream);
13 cudaEventRecord(stop, stream);
14
15 // Wait for events to finish before computing time
16 cudaEventSynchronize(stop);
17 cudaEventElapsedTime(&millis, start, stop);
```

Lines 7 and 8 create the events that will wrap the operation to be timed. Lines 11 and 13 insert the `start` and `stop` events around the operation to be timed. Line 16 blocks until the events have finished, then line 17 produces the milliseconds elapsed between the `start` and `stop` events.

2.7.2 CUDA Profiling Tools Interface

The CUDA Profiling Tools Interface [44] (CUPTI) "provides...detailed information about how applications are using the GPU in a system." Users may inject code into the entry and exit point of every CUDA C runtime and CUDA driver API function call. Additionally, users may configure and query hardware and software event counters to get insight into the operation of the GPU and CUDA stack. The event counters include instruction count, instruction throughput, memory loads/stores, memory throughput, cache hits/misses, branches and custom profile triggers. Chapter 6 describes how CUPTI can be used to record memory allocations, kernel arguments, and timestamps to build a model of the application execution.

2.7.3 `LD_PRELOAD`

LD_PRELOAD [45] is a mechanism by which the ld linker will load additional user-specific shared objects before any others. If a function definition is present in a pre-loaded shared object, it will override the implementation present in later objects. When combined with `dlsym()` [46], it can be used to inject code into the entry of library calls in dynamically-linked binaries. Chapter 6 describes how LD_PRELOAD can be used to infer information about application activity based on generic library calls.

2.8 Benchmark Library

This work makes use of the Google Benchmark library [47] as a harness for the developed custom microbenchmarks. Benchmark automatically determines the number of iterations to run based on the number of inputs and the desired CPU time. Benchmark allows repeated runs, and can report individual and aggregated statistics. This work makes uses of repeated runs, and presents results in terms of mean values and standard deviations. Benchmark also supports manual or automatic timing. An example of the basic layout of an automatic timing benchmark is shown in Listing 2.3. Microbenchmarks using Benchmark are written in C++, and have a setup phase, an interaction phase, and a teardown phase.

Listing 2.3: Benchmark with automatic timing.

```
1  #include <benchmark/benchmark.h>
2  static void BM_Foo(benchmark::State& state) {
3    for (auto _ : state) {
4            state.PauseTiming();
5            // Time elapsed here will not be counted
6            state.ResumeTiming();
7            foo();
8          }
9  }
10 BENCHMARK(BM_Foo);
```

Lines 2-9 define a Benchmark function, which is registered to be run in line 10. The loop body in lines 4 through 7 is executed and timed automatically by the framework, and the average time is recorded. Any part of the loop body that should not be timed can be wrapped in `state.PauseTiming()` and `state.ResumeTiming()`. This allows for per-iteration setup code to not be timed. The number of loop iterations is automatically controlled by the benchmark suite based on the number of arguments and desired run time. For the benchmarks presented in this work, there are tens of thousands of iterations for small transfers, and as few as a single transfer test for large iterations. The mean time is reported as a result of the benchmark. This process is then repeated multiple times to determine a standard deviation of the measurement.

Some of the microbenchmarks in this work make use of CUDA events to accurately measure the time taken by various CUDA operations. Benchmark supports this by allowing each microbenchmark to record and report its own iteration time. For example, consider Listing 2.4.

Benchmark provides a `DoNotOptimize(<expr>)` function, which forces the result of `<expr>` to be placed in a register. It does not prevent any optimization of `<expr>`, including replacing it with a statically-known value. Benchmark also provides a `ClobberMemory()` function, which forces all pending global memory writes to be completed. Through these two functions, the microbenchmarks can ensure that specific memory operations are completed and not optimized away.

Listing 2.4: Benchmark with manual timing.

```
1  #include <benchmark/benchmark.h>
2  static void BM_Foo(benchmark::State& state) {
3    cudaEvent_t start, stop;
4    cudaEventCreate(start);
5    cudaEventCreate(stop);
6    float msec;
7
8    for (auto _ : state) {
9      cudaEventRecord(start, NULL);
10     ... // cuda operation to benchmark
11     cudaEventRecord(stop, NULL);
12     cudaEventSynchronize(stop);
13     cudaEventElapsedTime(&msec, start, stop);
14     state.SetIterationTime(msec / 1000);
15   }
16 }
17 BENCHMARK(BM_Foo)->UseManualTime();
```

Now, when the benchmark is registered in line 17, the framework is informed that the benchmark loop iteration will report its own time. Instead of the framework timing the loop body in lines 9-14, CUDA events are used to measure the operation time. The iteration time is manually set in line 14.

Although Benchmark also supports multithreaded benchmarking, the microbenchmarks developed for this work do not use it. Benchmark does not directly support thread synchronization within each benchmark iteration, which is needed to accurately measure the performance of using multiple threads to move memory.

2.9 System Descriptions

Three high-performance heterogeneous systems are used in this work: an IBM S822LC for High Performance Computing [48], an IBM AC922 [49], and an Nvidia DGX-1 [29]. All systems feature multiple GPUs and multiple socketed CPUs. Their key differences are in CPU architecture (64-bit little-endian PowerPC for S822LC and AC922, x86-64 for DGX-1), number of GPUs (4 for the IBM machines, 8 for Nvidia), and GPU connection topology (NVLink 1.0 for S822LC, NVLink 2.0 for AC922, and hybrid PCIe 3.0 / NVLink 1.0

for DGX-1).

2.9.1 Nvidia DGX-1

Table 2.7: Nvidia DGX-1 architecture summary.

Component	Specification
CPU	2x Intel Xeon E5-2698 v4 40C / 80T 2.2 GHz
System Memory	512 GB DDR4
GPU	4 Nvidia P100
CPU/GPU Interconnect	NVLink 1.0 (1 lane) / PCIe 3.0
CUDA	8.0, driver 384.125
Kernel	4.4.0-79

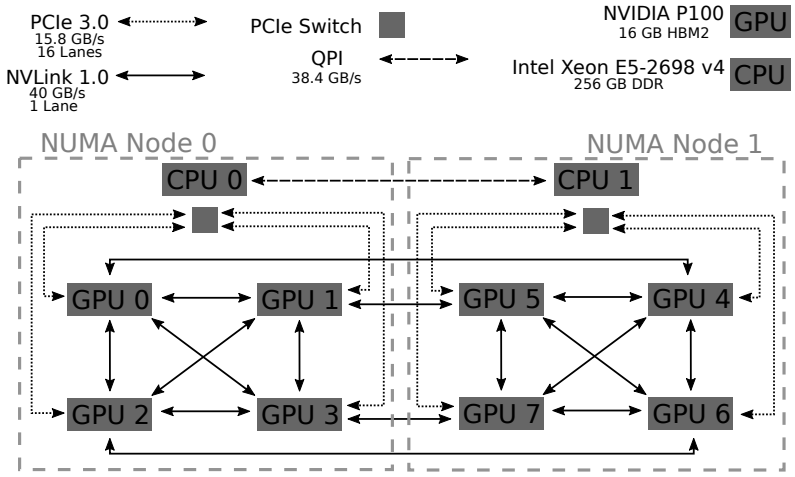


Figure 2.5: Nvidia DGX-1 architecture schematic. Interconnect legends are subtitled with theoretical maximum transfer rates. Each of the four NVLink lanes on a GPU is used to connect it to one other GPU. CPUs are connected to GPUs by PCIe 3.0 x16 interconnects.

Table 2.7 and Figure 2.5 summarize the Nvidia DGX-1 system architecture. The Nvidia DGX-1 machine consists of two symmetric sections [29]. Each section consists of one 20-core Intel Xeon E5-2698v4 CPUs with 2-way SMT. Each CPU is connected to 256GB of DDR4 RAM, and each section makes up a Linux NUMA node. Each section has 4 Nvidia Tesla P100 GPUs coupled by single NVLinks. The sections are connected by an Intel 9.6GT/s QPI bus

between the CPUs providing 38.4 GB/s of bidirectional bandwidth, as well as NVLinks between corresponding GPUs. The first CPU socket hosts the majority of the PCI devices on the system, including the network interfaces and the disks. The CPU/GPU device affinity is relatively simple: CPU0 is directly connected to GPUs 0-3 and CPU1 is directly connected to GPUs 4-7. Every GPU is directly connected to all local GPUs in its cluster, as well as one outside.

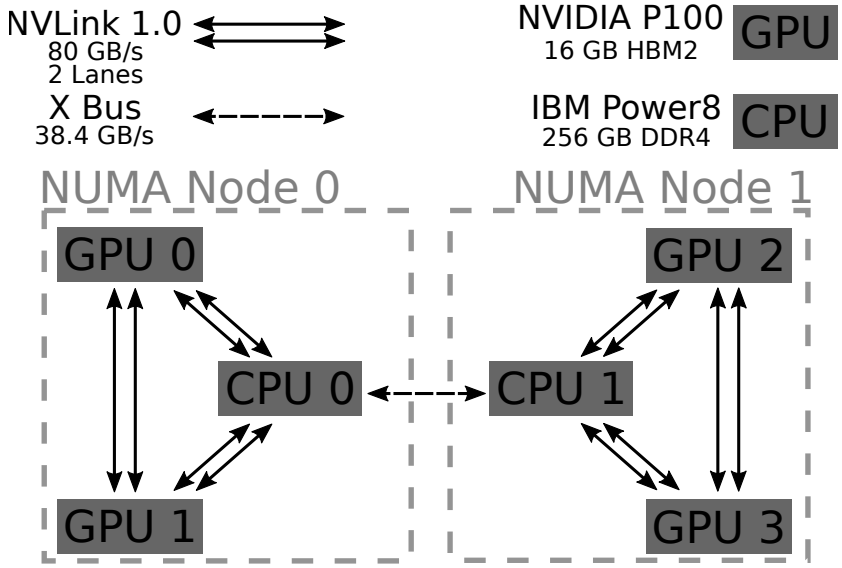
2.9.2 IBM S822LC for High Performance Computing

Table 2.8: IBM S822LC architecture summary.

	Specification
CPU	2x IBM Power8 20C / 80T 4 GHz
System Memory	512 GB DDR4
GPU	4 Nvidia P100
CPU/GPU Interconnect	NVLink 1.0 (2 lanes)
CUDA	9.1.85, driver 390.31
Kernel	4.4.0-96

Table 2.8 and Figure 2.6 summarize the hardware configuration. The IBM S822LC machine features two POWER8 CPUs and four Nvidia P100 GPUs [48]. Each POWER8 CPU has 10 cores, with 8-way simultaneous multithreading, and is attached to 256GB of DDR4 memory for a total of 160 threads and 512 GB of memory. Each POWER8 CPU and associated memory make up a Linux NUMA node. Each POWER8 CPU is part of a fully-connected triad of one POWER8 CPU and two P100 GPUs. Each device in the triad is connected by a gang of two NVLink 1.0 lanes for a total bidirectional bandwidth of 80 GB/s. The two triads are connected at the POWER8 CPUs by an IBM SMP X bus with 38.4 GB/s of bidirectional bandwidth.

Figure 2.6: IBM S822LC architecture schematic. Interconnect legends are subtitled with theoretical maximum transfer rates. The four NVLink lanes on a GPU are bonded into two two-lane pairs to connect it to the neighboring CPU and GPU.



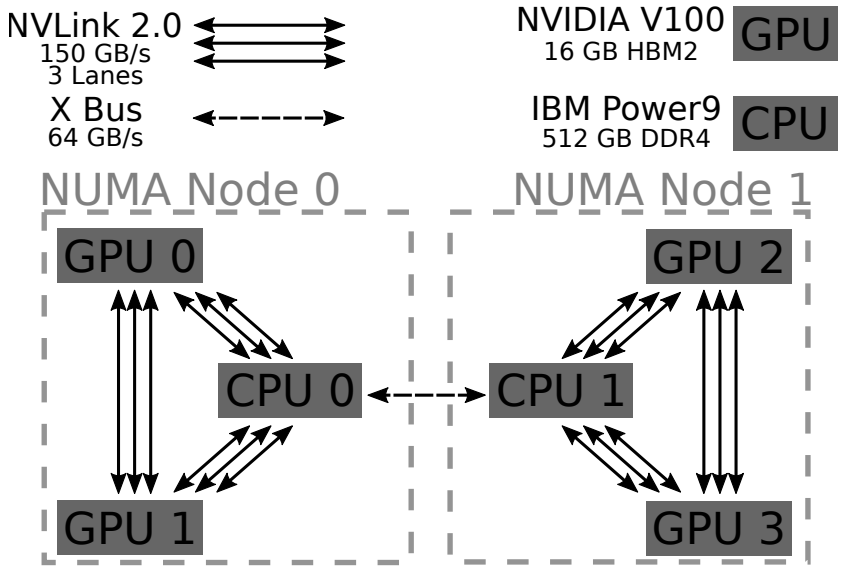
2.9.3 IBM AC922

The IBM AC922 machine features two POWER9 CPUs and four Nvidia V100 GPUs [49]. Each POWER9 CPU has 20 cores, with 4-way simultaneous multithreading, and is attached to 512GB of DDR4 memory for a total of 160 threads and 1TB of memory. Each POWER9 CPU is part of a fully-connected triad of one POWER9 CPU and two V100 GPUs. Each device in the triad is connected by three bonded NVLink 2.0 lanes for bidirectional bandwidth of 150 GB/s. The two triads are connected at the POWER9 CPUs by an IBM SMP X bus with 64 GB/s bandwidth. Table 2.9 and Figure 2.7 summarize the hardware configuration. Like the S822LC, each triad is a Linux NUMA node.

Table 2.9: IBM AC922 architecture summary

	Specification
CPU	2x IBM Power9 40C 160T 3.0 GHz
System Memory	1TB DDR4
GPU	4 Nvidia V100
CPU/GPU Interconnect	NVLink 2.0 (3 lanes)
CUDA	9.2.85, driver 396.15
Kernel	4.14.0-49

Figure 2.7: IBM AC922 architecture schematic. Interconnect legends are subtitled with theoretical maximum transfer rates. The six NVLink lanes on a GPU are bonded into two three-lane pairs to connect it to the neighboring CPU and GPU.



CHAPTER 3

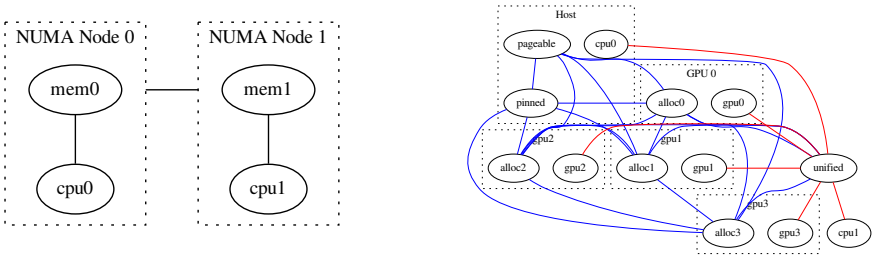
SYSTEM CHARACTERIZATION

This chapter describes and approach to produce an empirical performance model of hardware when an application invokes communication activities through the CUDA API functions. This performance model is needed for understanding the measured performance results to be presented in the rest of this thesis. In particular, this chapter motivates a joint performance model of software abstractions and underlying hardware. It then describes an approach for enumerating hardware components and connections.

3.1 Joint Abstraction and Hardware Model

Figure 3.1 shows two different communication abstractions of S822LC. Figure 3.1a represents the Linux NUMA view of the system. As described in Section 2.5, this view is accessible to the application through the libnuma library. Figure 3.1b represents the connectivity of the components through the CUDA API. Nodes in the graphs represent data storage locations or compute elements, and edges in the graph represent logical communication paths considered in this work. NUMA and CUDA present different abstractions, which are different from the system layout in Section 2.9.2. In practice, the logical communication paths available to the system are the union of these abstractions (and any other abstraction made available by the system).

Despite many of the logical communication paths using the same physical links, they achieve different performance on those links. As demonstrated in Chapters 4 and 5, some aspects of the empirical performance are determined by properties of the hardware links, while others are not. A communication performance model must therefore not only be based on the empirical performance of the logical links, but also incorporate understanding of the underlying hardware, if the model is to be applicable to more than just the



(a) (b)

Figure 3.1: Communication topologies exposed to the application. (a) The abstraction presented by NUMA. (b) The abstraction presented by CUDA.

system it was developed on.

Ideally, while empirical communication performance is characterized, the mapping to underlying hardware should be automatically established. This work takes that mapping as a-priori, but the following section presents initial efforts to automate the process.

3.2 Topology Enumeration

This work proposes a two-step approach to establishing a mapping from logical communication paths to underlying hardware.

1. Generate a graph G_s of the hardware.
2. Observe hardware utilization while exercising logical communication paths.

The hardware is represented by a graph $G_s = \{E, V\}$ where E is a set of edges representing communication links, and V is a set of vertices representing communication endpoints, or data routing elements. Sections 3.2.1 and 3.2.2 describe the specific system components explored. Each vertex in V is a data routing element. These vertices are able to receive and re-transmit data on any of their links. A PCIe switch is an example of a pure data-routing vertex. Optionally, the vertex may serve as a communication endpoint: a source or a sink for data. Processing elements and data storage elements serve as communication endpoints.

`Hwcomm` [50] is an open-source tool developed for automated hardware topology enumeration. This tool can be executed on a target system to generate G_s for that system. Thought `hwcomm` relies largely on exploring the PCIe device tree, it also uses information provided by the Nvidia Management Library [51] (NVML) to discover NVLink devices, and build a graph of hardware components. The first step of generating G_s is to discover the hardware components and connections through a multi-stage process.

Stage 1: Enumerate and Link CPU Sockets:

The Portable Hardware Locality [52] (hwloc) library is used to enumerate the present CPU sockets. As the test systems only have two sockets, all discovered sockets are considered to be directly connected by an SMP bus for the appropriate system type. The sockets and SMP buses are added to G_s.

Stage 2: Enumerate PCI devices:

The hwloc library is used to traverse the PCI device tree. All PCI devices are added to G_s and connected with PCI links of the appropriate type. Most attached storage, networking, and computing components are assigned an address in the PCI system and are discoverable in this step.

Stage 3: Update GPUs to Nvidia GPUs as appropriate:

Next, NVML is used to enumerate all Nvidia GPUs. The GPUs are matched by PCI address with existing PCI devices previously added to G_s, and NVML is used to discover whether NVLink is supported on each GPU and which other devices are connected to the GPU through NVLinks. This information is not provided by hwloc. The edges associated with the NVLinks are added to G_s.

3.2.1 Vertex Types

Table 3.1 summarizes the types of data routers discovered by hwcomm. These make up the vertices of G_s.

3.2.2 Edge Types

In G_s, the vertices are connected by the discoverable edge types shown in Table 3.2.

Table 3.1: A summary of the types of data routers that can be discovered by `hwcomm`. Some components may also serve as data endpoints.

Hardware	Data Router	Data Endpoint
CPU Socket	✓	✓
PCI Device	✓	✓
PCIe Hostbridge	✓	×
PCIe Bridge	✓	×
CUDA GPU	✓	✓
Linux Block Device	✓	✓
Linux Network Interface	✓	✓

Table 3.2: A summary of the types of communication links that can be discovered by `hwcomm`. Some components may also serve as data endpoints.

Edge Type	Description
SMP Bus	A symmetric multiprocessing bus connecting two CPU sockets.
PCIe Bus	A PCIe link connecting a PCIe Bidge and PCIe device or PCIe Hostbridge and PCIe bridge.
NVLink1	A first-generation NVLink connecting two Nvidia GPUs or an Nvidia GPU and CPU
NVLink2	A second-generation NVLink connecting two Nvidia GPUs or an Nvidia GPU and CPU
SATA bus	A Serial AT Attachment link conneting a host bus to a mass storage device.

3.2.3 Discovered Topologies

The topologies of the S822LC, AC922, and DGX-1 systems are show in Appendix 8.

3.2.4 Logical Path to Hardware Link Mapping

Once the system graph is established, the mapping between logical communication paths and system graph vertices and edges can be established by observing performance counters while benchmarking logical paths. For example, NVML provides access to NVLink performance counters. As known quantities of data are moved across the logical connections, the hardware link performance counters can be observed to associate logical transfers with traffic across physical links.

In this work, automatic determination of the mapping is not considered; instead, the mapping for the case study systems is known ahead of time.

CHAPTER 4

EXPLICIT MEMORY PERFORMANCE

This chapter examines the performance of explicit data transfers over logical communication links presented in a system with NUMA and CUDA interfaces. In particular, it focuses on CPU/CPU transfers, CPU/GPU transfers, and GPU/GPU transfers. It highlights cases where the observed logical communication performance deviates significantly from the symmetries present in the CUDA API, numactl API, and hardware. Those deviations take the form of different performance on identical links, anisotropic link performance, or performance affected by device affinity.

The microbenchmarks developed for this section are available in the `microbench` project [53]. That project also includes benchmarks of other aspects of CUDA performance, including CUDA primitives like kernel launches, and CUDA libraries such as cuBLAS and cuDNN.

4.1 CPU / CPU Transfers

This section begins by examining CPU-CPU transfer performance through `cudaMemcpy`. This attempts to provide insight into the CUDA performance when sending data from one CPU socket to another. Such a transfer would occur when data is sent from a CPU A to a GPU attached to another CPU B. The data would traverse the SMP bus between CPU A and CPU B, and the bandwidth of that bus could limit the overall performance of the transfer. Algorithm 4.1 describes the measurement approach. During the setup phase, an allocation is created on the source CPU *src* and destination CPU *dst*. During the benchmark iterations, the *dst* cache is invalidated (if the *src* is different from the *dst*) by accessing that data from the *src*. Then, `cudaMemcpy` is invoked to transfer data between those allocations. CUDA events are used to measure the time of the memory copy.

Algorithm 4.1 Algorithm to measure `cudaMemcpy` CPU-CPU Bandwidth. AllocPinned and AllocPageable are defined in Algorithm 4.2. `numa_bind_node` is defined in Listing 2.1.

```
1: function BANDWIDTH(dst, src, transfer_size)
2:     numa_bind_node(src)
3:     srcPtr ← AllocPageable(transfer_size)
4:     memset(srcPtr, 0, transfer_size)
5:     numa_bind_node(dst)
6:     dstPtr ← AllocPinned(transfer_size)
7:     memset(dstPtr, 0, transfer_size)
8:     start ← cudaEventCreate()
9:     stop ← cudaEventCreate()

10:    for state do                                  ▷ Benchmark library loop
11:        numa_bind_node(src)
12:        memset(srcPtr, 0, transfer_size)          ▷ invalidate dst cache
13:        numa_bind_node(dst)
14:        cudaEventRecord(start)
15:        cudaMemcpy(
               dstPtr, srcPtr, transfer_size,
               cudaMemcpyHostToHost)
16:        cudaEventRecord(stop)
17:        cudaEventSynchronize(stop)
18:        millis ← cudaEventElapsedTime(start, stop)
19:        state.SetIterationTime(millis/1000)
20:    end for
21: end function
```

Algorithm 4.2 shows different kinds of host allocation strategies used in microbenchmarks that need CUDA host allocations on particular NUMA nodes. `AllocPageable` simply defers to `malloc`, which will return memory allocation on a previously-pinned NUMA node. `AllocPinned` defers to `malloc` to get a NUMA allocations, and then uses `cudaHostRegister` to pin that memory. `AllocWriteCombined` uses the `cudaHostAlloc` CUDA library call with the `cudaHostAllocWriteCombined` flag to request that CUDA allocate write-combining memory.

Algorithm 4.2 Pageable, pinned, and write-combining host allocators.

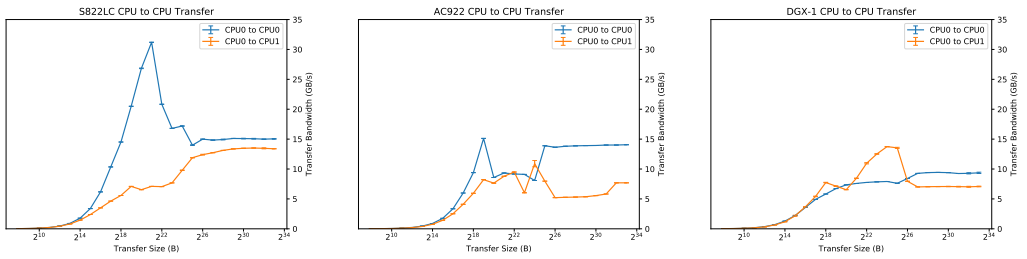
```
1:  function AllocPageable(bytes)
2:      ptr ← 0
3:      malloc(ptr, bytes)
4:      return ptr
5:  end function

6:  function AllocPinned(bytes)
7:      ptr ← 0
8:      malloc(ptr, bytes)
9:      cudaHostRegister(ptr, bytes, cudaHostRegisterPortable)
10:     return ptr
11: end function

12: function AllocWriteCombined(bytes)
13:     ptr ← 0
14:     cudaHostAlloc(ptr, bytes, cudaHostAllocWriteCombined)
15:     return ptr
16: end function
```

Figure 4.1 shows intra- and inter-CPU `cudaMemcpy` performance on S822LC, AC922, and DGX-1. In all cases, for small transfers, the bandwidth is limited by the overhead of invoking the transfer. For intermediate and larger sizes, that overhead ceases to be the performance-limiter. At large sizes, intra-CPU bandwidth is higher, presumably since data transfer over the SMP bus is not required. The bandwidth saturates at the rate that a single thread can generate loads and stores.

S822LC and AC922 have similar intra-CPU performance except for intermediate sizes, where the S822LC performance peaks (presumably due to transfers happening in cache) and AC922 performance drops. On DGX-1, inter-CPU transfers are actually faster than intra-CPU transfers for intermediate sizes.



(a) (b) (c)

Figure 4.1: CPU-to-CPU transfer bandwidth vs. transfer size. Each transfer is measured using Algorithm 4.1. Whiskers at each point show the standard deviation measured over 5 repetitions.

4.2 CPU / GPU Transfers

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. In this work, the host allocation is created by one of three methods shown in Algorithm 4.2. The device allocation is created by `cudaMalloc`. This section compares bandwidth achievable from pinned, pageable, and write-combining host allocations, with particular emphasis on how device affinity affects transfer-performance and cases where transfers are anisotropic.

Algorithm 4.3 is used to evaluate the achievable bandwidth for `cudaMemcpy` transfers between a GPU allocation and a pageable, pinned, or write-combining host allocation. The same algorithm can be used for these cases, because the same `cudaMemcpy` CUDA API call to transfer data can be used on a pointer pointing to any of the allocation types. Depending on the source and destination types *src* and *dst*, and the desired host allocation type, the corresponding CUDA or numactl APIs are called to bind later activities to the desired GPU or CPU. Then, the CUDA or host allocators are invoked to produce *devPtr* (a pointer to the device allocation) and *hostPtr* (a pointer to the CPU allocation). The main benchmark loop uses the `cudaMemcpy` time as the iteration time that should be reported.

Algorithm 4.3 Measuring CPU/GPU bandwidth with `cudaMemcpy`. Host allocators are described in Algorithm 4.2. `numa_bind_node` is defined in Listing 2.1.

```
1: function BANDWIDTH(dst, src, bytes, num_iters)
2:     if src is GPU then
3:         cudaSetDevice(src)
4:     else                                              ▷ src is CPU
5:         numa_bind_node(src)
6:     end if
7:     if dst is GPU then
8:         cudaSetDevice(dst)
9:     else                                              ▷ dst is CPU
10:        numa_bind_node(dst)
11:    end if
12:    devPtr ← cudaMalloc(bytes)                        ▷ device allocation
13:    hostPtr ← hostAllocate(bytes)                     ▷ appropriate host allocator
14:    if src is GPU then
15:        srcPtr ← devPtr
16:        dstPtr ← hostPtr
17:    else                                              ▷ src is CPU
18:        srcPtr ← hostPtr
19:        dstPtr ← devPtr
20:    end if
21:    start ← cudaEventCreate()
22:    end ← cudaEventCreate()
23:    for state do
24:        cudaEventRecord(start)
25:        cudaMemcpy(dstPtr, srcPtr, bytes, cudaMemcpyDefault)
26:        cudaEventRecord(stop)
27:        millis ← cudaEventElapsedTime(start, stop)
28:        state.SetIterationTime(millis / 1000)
29:    end for
       return elapsed
30: end function
```

4.2.1 Comparison of Pageable, Pinned, and Write-Combining Host Allocations

To contextualize other results presented in this chapter, Figure 4.2 shows the transfer performance between pageable, pinned, and write-combined allocations on CPU0 and a device allocation on GPU0. On all tested systems, this is a local transfer between a directly-connected CPU and GPU. These performance curves exhibit features common throughout this chapter:

- For small transfer sizes, the time is dominated by overhead of invoking the transfer.
- For large transfer sizes, the performance is dominated by the exercised pysical link (in pinned or write-combining transfers) or some part of the abstraction layer (pageable transfers).
- The performance may vary smoothly across intermediate transfer sizes, or exhibit more complicated behavior. For example, in the AC922 transfer shown in Figure 4.2a, bandwidth peaks and then drops for intermediate transfer sizes before recovering for larger transfers.

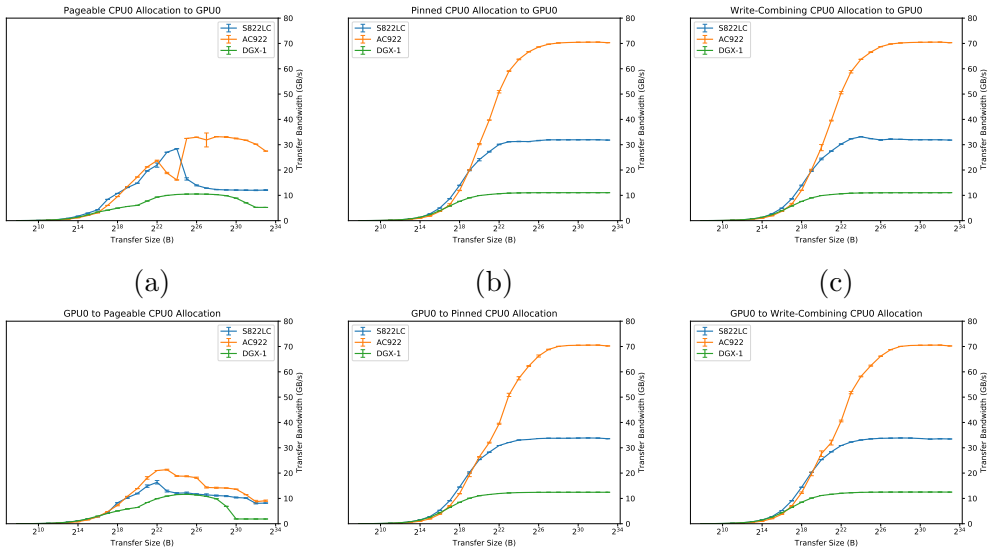


Figure 4.2: `CudaMemcpy` bandwidth vs. transfer size for CPU0 to GPU0 transfers from (a) pageable allocations, (b) pinned allocations, and (c) write-combining allocations and GPU0 to CPU0 transfers of the same kind (d-f). Results for S822LC, AC922, and DGX-1 systems are shown. Whiskers show standard deviations of benchmark measurements over five repetitions.

The curves for transfers involving pinned or write-combined allocations on CPU0 (Figures 4.2(b,c,e,f)) share a similar shape: the transfer bandwidth is low for small sizes, and eventually saturates once transfers become large enough. S822LC and DGX-1 achieve ~75% of the theoretical 40 and 20 GB/s bandwidths of the NVLink 1.0 x2 and PCIe 3.0 x16 links, respectively. AC922 achieves nearly 100% of the theoretical 75 GB/s unidirectional NVLink 2.0 x3 bandwidth.

In contrast, Figures 4.2a and 4.2d show transfers involving pageable allocations. The achievable bandwidth for large transfer sizes on S822LC is reduced to approximately 25% of the theoretical 40 GB/s bandwidth provided by the link. On AC922 the performance is nearly 50% of the theoretical bandwidth. AC922 CPU-to-GPU transfers also show a high bandwidth achieved for large CPU-to-GPU transfers, but not for GPU-to-CPU transfers. DGX-1 bandwidth peaks at 50% of the theoretical 20 GB/s of one-lane NVLink 1.0, but drops substantially for large transfer sizes.

Section 2.4.2 describes how `cudaMemcpy` from a pageable allocation to the GPU actually causes two data copies: one from the pageable allocation application to a pinned buffer, and a second copy, a DMA from the pinned buffer to the GPU. When pinned memory transfers are faster than pageable memory, we can infer that the CPU memory copy from pageable allocation to pinned buffer is limiting the performance. For comparison, consider Figure 4.1, which shows the performance of using `cudaMemcpy` to only do a copy from a pageable allocation to a pinned allocation. For S822LC, the pageable-to-GPU transfer shown in Figure 4.2a is approximately the same performance as the pageable-to-pinned transfer shown in Figure 4.1.

Surprisingly, on AC922, the pageable-to-GPU transfer for large transfer sizes is substantially faster than the pageable-to-pinned transfer that it should be limited by. This suggests that there is a different implementation for the two cases. Bandwidth spikes at intermediate sizes suggest that the GPU DMA may directly access data from the CPU cache when the transfer can fit in the cache. This is further reinforced by the lack of difference between pinned and write-combining transfer bandwidth, which suggests that caching or lack thereof on these systems does not influence the DMA engine.

4.2.2 CPU/GPU Bandwidth Measurements

Figure 4.3 shows CPU/GPU bandwidth on a variety of logical paths for S822LC, AC922, and DGX-1. Transfers involving pinned and pageable allocations are shown. Write-combined results are omitted as they match the pinned performance.

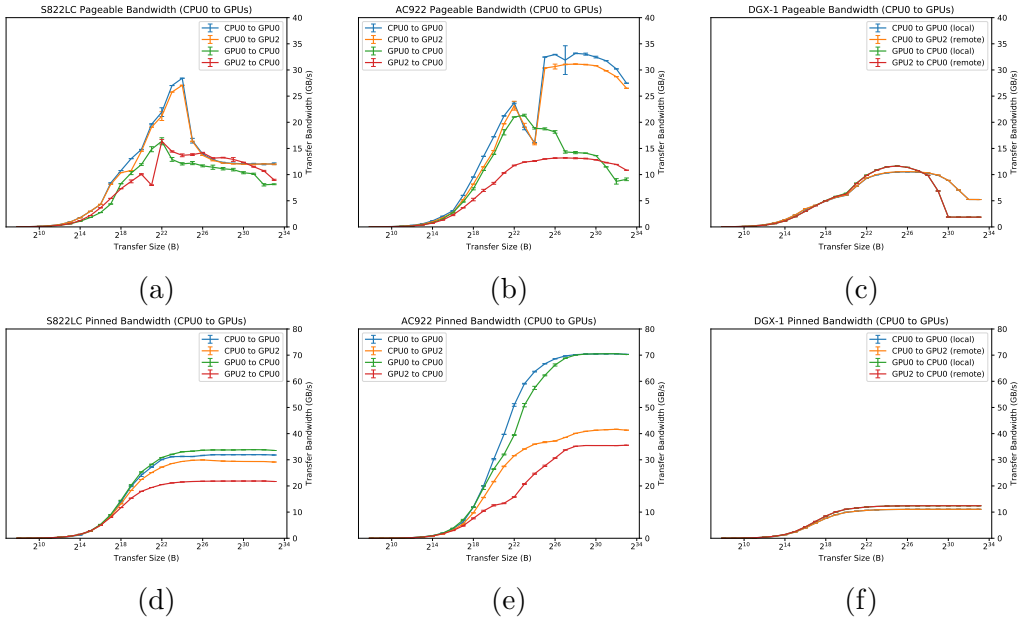


Figure 4.3: Transfer bandwidth vs. transfer size for local and remote transfers from pageable and pinned host allocations on S822LC, AC922, and DGX-1. (a-c) Transfers from pageable allocations to GPUs. (d-f) Transfers from pinned allocations to GPUs. (a) and (d) are for S822LC, (b) and (e) for AC922, and (c) and (f) for DGX-1.

In general, the bandwidth follows the same outline described in Section 4.2.1, with overhead-dominated time for small transfers, bandwidth-dominated time for large transfers, and some other behavior between. There are some distinctive reoccurring patterns in Figures 4.3 (a)-(c).

CPU-to-GPU pageable transfers on the IBM systems exhibit peaks in transfer bandwidth at intermediate transfer sizes. The shape of this curve suggests some insight into the copy implementation. For example, consider the S822LC CPU0 to GPU0 curve. The expected process is that a pageable allocation on CPU0 is copied to one or more pinned allocations on CPU0, which are then accessed by GPU0's DMA engine. The fact that the peak bandwidth at intermediate transfer sizes surpasses the measured bandwidth

for single-threaded inter-CPU transfers (Figure 4.1) suggests that the pageable-to-pinned copy is indeed occurring within a single CPU, and not across CPUs. The same shape is even present in the CPU0 to GPU2 curve, where it would be plausible for the pageable allocation to be on CPU0 and the pinned allocation on CPU1. For S822LC as the transfer grows larger, the bandwidth reaches a steady-state value that is approximately the same as the single-threaded CPU-CPU memory access bandwidth. In AC922, we see a large change in the transfer bandwidth, suggesting that some other implementation is chosen at those sizes. The drop before reaching that bandwidth suggests some system performance bug, like imbalance in the number and size of the pinned transfer buffers that prevents good overlapping of the host-device DMA and the host-host memory copy. Finally, on AC922, the local GPU-to-CPU transfer is actually faster than the corresponding intra-CPU `cudaMemcpy`, again suggesting some difference in implementation when a GPU is involved. On DGX-1, no similar intermediate spike is observed. The total transfer bandwidth is capped by the lower interconnect bandwidth, clipping that shape. Also on DGX-1, the transfer bandwidth at large sizes is seriously degraded, falling well below even the CPU-to-CPU `cudaMemcpy` bandwidth. This also may be a performance bug.

Figures 4.3 (d)-(f) show the same transfers, but from pinned allocations. These transfers are all ultimately limited by the interconnect bandwidth, and do not show the same peaks. On S822LC, local transfers achieve around 75% of the NVLink 1.0 bandwidth, while remote transfers achieve 50-75% of the lower 38.4 GB/s SMP bus bandwidth. On Ac922, local transfers achieve around 95% of the NVLink 2.0 bandwidth, while remote transfers again are limited to around 50-75% of the 64 GB/s SMP bus. On DGX-1, both transfers achieve around 60-70 of the PCIe 3.0 bandwidth.

4.2.3 Affinity

On systems with high-performance interconnects, transfers from GPU allocations to pageable CPU allocations are strongly correlated with device affinity. Transfers involving pinned CPU allocations demonstrate a strong effect in both CPU-to-GPU and GPU-to-CPU directions. The presence of high-performance interconnects further exposes any performance differences,

since bandwidth is less likely to be limited by underlying link bandwidth and more likely to be limited by performance bugs or single-threaded memory copies. Table 4.1 summarizes the effects.

Figure 4.3a shows that affinity has a small effect on bandwidth for pageable transfers on S822LC. For transfers larger than 4 MB, the GPU-to-CPU remote transfer is faster than the GPU-to-CPU local transfer. Figure 4.3b shows local GPU-to-CPU pageable transfers on AC922 are much faster than their remote counterparts, except for large transfers, where remote performance is slightly higher. Figure 4.3c shows that GPU-to-CPU pageable transfers on DGX are not affected by affinity. Figure 4.3d shows on S822LC, pinned bandwidth is correlated with affinity, particularly in the GPU-to-CPU direction. Figure 4.3e shows for pinned transfers on AC922, affinity has an even stronger effect. Figure 4.3f shows no effect from affinity on pinned transfers for DGX-1. Without the additional CPU-CPU copy, pinned bandwidth is highly dependent on the bandwidth of the underlying hardware links. For remote transfers on the IBM machines, the SMP bandwidth is less than the NVLink bandwidth, and limits performance. On the DGX system, remote and local CPU-GPU transfers all must traverse PCIe 3.0 links, so there is no performance effect from affinity.

Table 4.1: Effect of device affinity on logical transfer bandwidth.

Transfer Kind	**S822LC**	**AC922**	**DGX-1**
Pageable → GPU	× (Fig. 4.3a)	× (Fig. 4.3b)	× (Fig. 4.3c)
Pageable ← GPU	✓(Fig. 4.3a)	✓(Fig. 4.3b)	× (Fig. 4.3c)
Pinned → GPU	✓(Fig. 4.3d)	✓(Fig. 4.3e)	× (Fig. 4.3f)
Pinned ← GPU	✓(Fig. 4.3d)	✓(Fig. 4.3e)	× (Fig. 4.3f)

4.2.4 Anisotropy

Anisotropy refers to the property of being directionally-dependent, e.g., CPU/GPU transfer bandwidth is anisotropic if CPU-to-GPU bandwidth is different than GPU-to-CPU bandwidth. Figure 4.3 highlights that link bandwidth exhibits significant anisotropy in systems with high-performance interconnects. Table 4.2 summarizes the effects. Particularly for the pageable transfers shown in Figures 4.3a and 4.3b, corresponding transfers are shown to be highly anisotropic. In the pinned transfers on S822LC (Fig. 4.3d), all

transfers show some degree of anisotropy, with a larger effect over remote transfers. For transfers where there is a difference, CPU → GPU transfers tend to be faster. For AC922 (Fig. 4.3e), remote transfers generally show anisotropy, and local transfers show anisotropy only for intermediate transfer sizes. For DGX-1, pinned transfers all show 2 GB/s of anisotropy (Fig. 4.3f), while the degree of anisotropy for transfers involving pageable allocations depends on the transfer size (Fig. 4.3c). For pageable transfers, CPU-to-GPU transfers tend to be faster than GPU-to-CPU transfers.

Table 4.2: Host-device transfer anisotropy.

Transfer Kind	**S822LC**	**AC922**	**DGX-1**
Pageable ↔ GPU (local)	✓(Fig. 4.3a)	✓(Fig. 4.3b)	✓(Fig. 4.3c)
Pageable ↔ GPU (remote)	✓(Fig. 4.3a)	✓(Fig. 4.3b)	✓(Fig. 4.3c)
Pinned ↔ GPU (local)	✓(Fig. 4.3d)	✓(Fig. 4.3e)	✓(Fig. 4.3f)
Pinned ↔ GPU (remote)	✓(Fig. 4.3d)	✓(Fig. 4.3e)	✓(Fig. 4.3f)

4.2.5 Differences between Identical Transfers

Figure 4.4 shows cases of different performance on two topologically- or logically-identical links. Figures 4.4a and 4.4b show transfer bandwidth between a pageable allocation and local or remote GPU on S822LC, respectively. Figures 4.4c and 4.4d show transfer bandwidth from a local GPU to a pageable allocation, and a pageable allocation to a remote GPU on AC922, respectively. Each of these four scenarios involve identical logical and topological links, and yet a substantial transfer bandwidth difference is observed. Table 4.3 summarizes scenarios where the transfer performance differs on identical links.

These discrepancies only manifest on systems with high-bandwidth interconnects and pageable transfers. This suggests that the lower-performance PCIe 3.0 buses mask any similar effects that appear on that system. Furthermore, it suggests that the causes of these discrepancies are performance bugs in the CUDA system relating to how internal buffers are allocated, or performance bugs in the system firmware related to low-level data transfer.

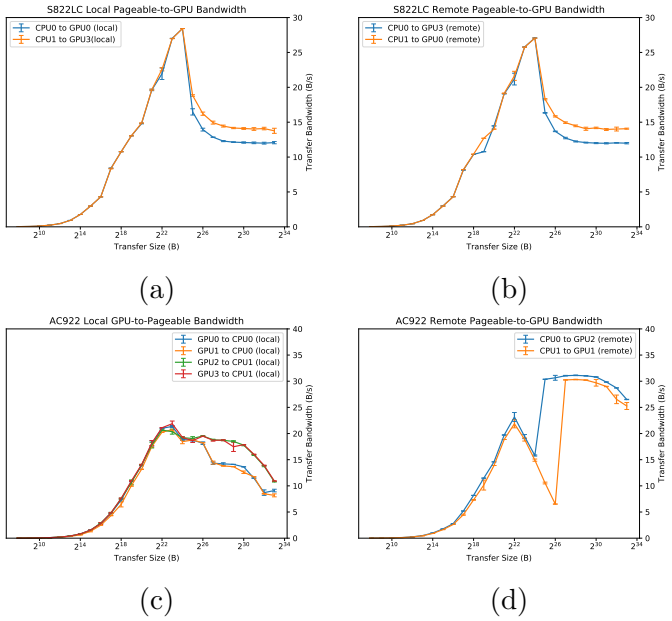


Figure 4.4: Cases of CPU/GPU `cudaMemcpy` bandwidth on identical links. (a-b) Transfer bandwidth from pageable allocations to GPUs on S822lc. (c) Bandwidth from local GPU to pageable CPU allocation on AC922. (d) Bandwidth from pageable CPU allocation to remote GPU on AC922.

Table 4.3: Transfer rate variability on identical CPU-GPU links.

Transfer Kind	**S822LC**	**AC922**	**DGX-1**
Pageable → GPU	✓(Figs. 4.4a and 4.4b)	✓(Figs. 4.4d)	×
Pageable ← GPU	×	✓(Figs. 4.4c)	×
Pinned → GPU	×	×	×
Pinned ← GPU	×	×	×

4.3 GPU / GPU Transfers

Explicit GPU-GPU transfers are caused by the `cudaMemcpy` family of functions being invoked on pointers to device allocations created with `cudaMalloc`. Unlike the different types of host allocations in Section 4.2, this section only refers to a single kind of device allocation. Device allocations come with the concept of peer access, discussed in Section 2.4.4. This section focuses on the effect of peer access on transfer bandwidth, and cases where transfers are anisotropic or have different performance on identical links.

Algorithms 4.4 and 4.5 are used to evaluate the achievable GPU-GPU transfer bandwidth with and without peer access enabled. When peer access is disabled, `numa_bind_node` is used to pin the executing thread to a specific

node. On systems where the CUDA driver does not make NUMA-aware allocations, this may help control for NUMA performance effects. Then, peer access is enabled or disabled depending on the experimental configuration. Then, `cudaMalloc` is used to create allocations of $transfer\_size$ bytes pointed to by $srcPtr$ and $dstPtr$. The achievable bandwidth is measured during the Benchmark loop using `cudaEvent`s and `cudaMemcpy`.

Algorithm 4.4 Measuring GPU-GPU `cudaMemcpy` bandwidth with peer access enabled.

```
1: function BANDWIDTH(dst, src, transfer_size)
2:     cudaSetDevice(src)
3:     srcPtr ← cudaMalloc(transfer_size)              ▷ Source allocation
4:     cudaMemset(srcPtr, transfer_size, 0)
5:     cudaDeviceEnablePeerAccess(dst)
6:     cudaSetDevice(dst)
7:     dstPtr ← cudaMalloc(transfer_size)              ▷ Destination allocation
8:     cudaMemset(dstPtr, transfer_size, 0)
9:     cudaDeviceEnablePeerAccess(src)
10:    start ← cudaEventCreate()
11:    end ← cudaEventCreate()
12:    for state do
13:        cudaEventRecord(start)
14:        cudaMemcpy(dstPtr, srcPtr, bytes, cudaMemcpyDefault)
15:        cudaEventRecord(stop)
16:        millis ← cudaEventElapsedTime(start, stop)
17:        state.SetIterationTime(millis / 1000)
18:    end for
19: end function
```

Algorithm 4.5 Measuring GPU-GPU `cudaMemcpy` bandwidth with peer access disabled. `numa_bind_node` is defined in Listing 2.1.

```
1: function BANDWIDTH(dst, src, numa, transfer_size)
2:     numa_bind_node(numa)
3:     cudaSetDevice(src)
4:     srcPtr ← cudaMalloc(transfer_size)            ▷ Source allocation
5:     cudaMemset(srcPtr, transfer_size, 0)
6:     cudaDeviceDisablePeerAccess(dst)
7:     cudaSetDevice(dst)
8:     dstPtr ← cudaMalloc(transfer_size)            ▷ Destination allocation
9:     cudaMemset(dstPtr, transfer_size, 0)
10:    cudaDeviceDisablePeerAccess(src)
11:    start ← cudaEventCreate()
12:    end ← cudaEventCreate()
13:    for state do
14:        cudaEventRecord(start)
15:        cudaMemcpy(dstPtr, srcPtr, bytes, cudaMemcpyDefault)
16:        cudaEventRecord(stop)
17:        millis ← cudaEventElapsedTime(start, stop)
18:        state.SetIterationTime(millis / 1000)
19:    end for
20: end function
```

4.3.1 Transfer Rate and Peer Access

Figure 4.5 shows the performance of a variety of GPU-GPU transfers with and without peer access. Generally, peer access has a large effect on the bandwidth of local GPU-GPU transfers. With peer access enabled, GPUs may do DMAs directly with their peer memory instead of copying through the host. Without peer access, the execution is pinned to a particular CPU to control the location for the CUDA system making memory allocations.

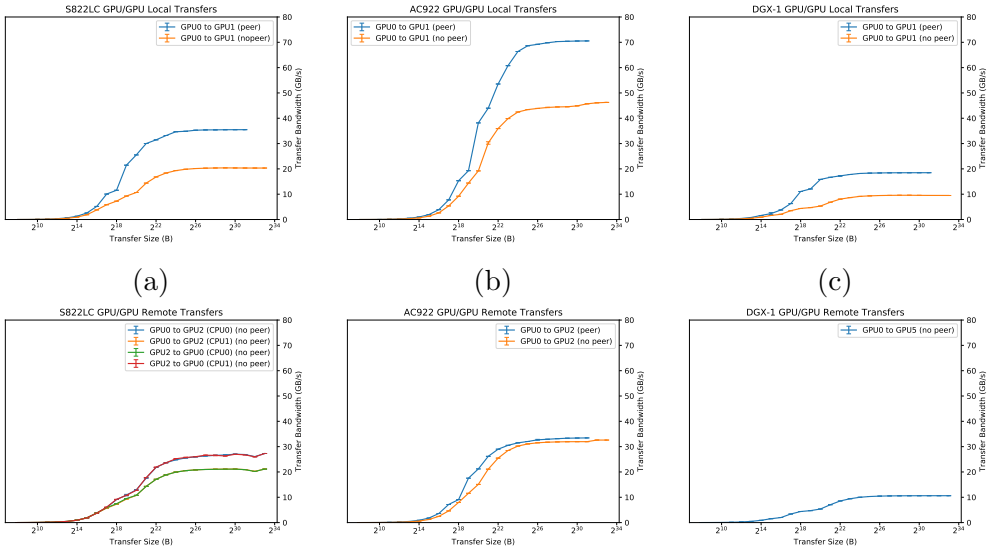


Figure 4.5: GPU-GPU `cudaMemcpy` transfer bandwidth vs. transfer size for various scenarios with peer access enabled or disabled. (a) GPU0 to GPU1 transfer on S822LC with and without peer access. (b) GPU-to-GPU local transfers on AC922. (c) GPU-to-GPU local transfers on DGX-1. (d-f) GPU-to-GPU remote transfers on the same systems. S822LC (d) and DGX-1 (f) do not support peer access for remote GPU-GPU transfers. S822LC (d) is annotated with the CPU that the non-peer benchmark is pinned to. The same is not shown in (e-f) because there is no significant performance effect.

Figure 4.5a shows that enabling peer access on S822LC improves the performance of local GPU-GPU transfers by ~40%, to over 90% of the theoretical link bandwidth. Likewise, Figure 4.5c shows peer access roughly doubling performance on DGX-1 and Figure 4.5b shows similar behavior for AC922, with the much higher performance ceiling due to the increased bandwidth of NVLink 2.0.

On S822LC remote transfers (Figure 4.5d), pinning the benchmark to the NUMA node with affinity to the sending GPU improves the performance by around 25%. On DGX-1 (Figure 4.5f), no similar effect is seen. Any performance differences may be masked by the limited PCIe CPU-GPU bandwidth relative to the CPU-CPU and GPU-GPU bandwidth, so the specific route that the data takes through the system matters less than whether or not the data travels over a slow PCIe bus. The performance difference may be masked by the fact that GPU-CPU interconnects are slower than CPU-CPU interconnects on that system, so when the data transfers

back. On AC922 (Figure 4.5e), there is no substantial effect for large transfers for disabling peer access.

4.3.2 Transfer Rate on Identical Transfers

Different performance is observed on identical GPU-GPU transfers when peer access is disabled. Figure 4.6 shows some example scenarios. In isolation, there is no reason to disable peer access for local transfers, as it always reduces performance. Therefore, it is unlikely that a practical program would ever exercise this scenario. During cases of contention, peer access could be utilized to divert data along a different hardware path, so these results are presented for completeness.

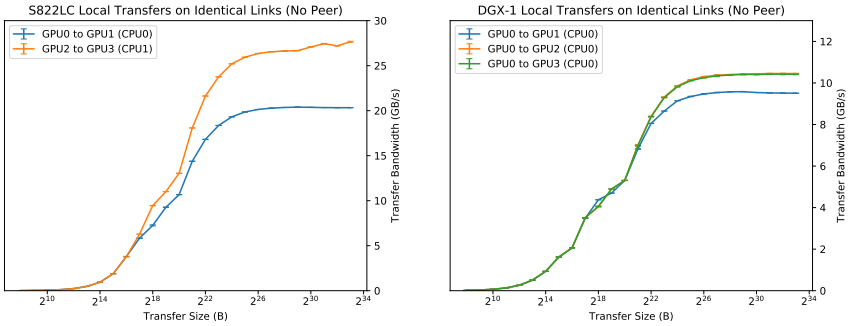
Figure 4.6a compares the bandwidth of transfers from GPU0 to GPU1 (pinned to local CPU0) and GPU2 to GPU3 (pinned to local CPU1) on S822LC. These are identical transfers on different CPU-GPU-GPU triads that make up S822LC (Figure 2.6). The transfer between GPU0 and GPU1 is around 30% slower than the same transfer between GPU2 and GPU3.

Similarly, Figure 4.6b shows variability in transfer bandwidths observed on DGX-1. As shown in Figure A.1, there are three PCIe bridges between GPU0 and GPU1, and there are seven PCIe bridges between GPU2 and GPU3. The bandwidth between GPU0 and GPU2/GPU3 is identical, as are the topologies between them. Although the topology between GPU0 and GPU1 is shorter, the performance is also lower. It is possible that the different bridges have different performance characteristics.

Table 4.3 summarizes the cases where differing performance is observed.

Table 4.4: Transfer rate on identical GPU-GPU links

Transfer Kind	S822LC	AC922	DGX-1
GPU ↔ Local GPU (peer enabled)	×	×	×
GPU ↔ Remote GPU (peer enabled)	N/A	×	N/A
GPU ↔ Local GPU (peer disabled)	✓(Fig. 4.6a)	×	✓(Fig. 4.6b)
GPU ↔ Remote GPU (peer disabled)	×	×	×



(a) (b)

Figure 4.6: Transfer bandwidth vs. transfer size for S822LC and DGX-1. (a) Differing transfer bandwidth on logically-identical local transfers on S822LC. (b) The same for DGX-1. AC922 is omitted as no performance variability is present when controlling for NUMA pinning during non-peer transfers.

4.4 Summary

The performance of `cudaMemcpy` transfers is highly dependent on device affinity, CPU allocation type, transfer direction, and underlying hardware performance. Transfers involving pageable host allocations are particularly unpredictable, probably due to the performance of combined intra-CPU communication, inter-CPU communication, cache effects, and DMA being difficult to tune for all cases. In general, constraining communication to pinned buffers and local devices offers the best performance, though direction of the transfer still has a large impact. Some CPU-GPU pageable transfers exhibit different performance even when the logical communication is the same. This is also present in GPU-GPU transfers, but only for local transfers when peer-access is disabled.

CHAPTER 5

UNIFIED MEMORY PERFORMANCE

The unified memory system (Sections 2.4.5 and 2.4.6) greatly simplifies the programmer interaction with CUDA memories and data transfer. Data tranfers in the unified memory system are created in two ways:

- *coherence* (or *demand*) transfers, where data is migrated to ensure that the CPU and GPU have a consistent view of memory.
- *prefetch* transfers, where data is moved ahead of time, with the purpose of reducing future access times.

This chapter comprises two sections, detailing performance of prefetch and coherence unified memory bandwidth (Section 5.1), and page fault latency (Section 5.2). In each section, transfer bandwidth for coherence and prefetch transfers is examined, as well as page-fault latency for demand transfers, where applicable.

Algorithm 5.1 describes the approach to measure coherence or prefetch bandwidth between two GPUs in the unified memory system. First, a *bytes*-size unified memory allocation at *ptr* is associated with the destination device. As of the time of this writing, this choice of association should not affect the benchmark, but is enforced for consistency. Then, `memset` is used to force pages backing the allocation to be produced. A *start* and *stop* event is created on the destination device, where the kernel will be executed for coherence bandwidth measurements. During the benchmark loop, *ptr* is prefetched to the source device. Then, for coherence bandwidth, `gpu_write` is executed on the destination device, or for prefetch bandwidth, `cudaMemPrefetchAsync` is used to prefetch *ptr* to the destination device. `CudaEventSyncrhonize` is used to ensure the coherence or prefetch workload is complete. The time for the CPU workload or GPU workload and synchronization is recorded as the iteration time. The benchmark is repeated five times to discover outliers and establish a standard deviation of measurement.

Algorithm 5.1 Measuring GPU-GPU unified memory coherence or prefetch bandwidth during a *bytes*-sized transfer between *src* and *dst*. `gpu_write` is defined in Listing 5.2.

```
1: function BANDWIDTH(dst, src, bytes)
2:     pageSize ← sysconf(_SC_PAGESIZE)
3:     cudaSetDevice(dst)
4:     ptr ← cudaMallocManaged(bytes)
5:     memset(ptr, 0, bytes)                    ▷ force pages to be allocated
6:     cudaEventCreate(start)
7:     cudaEventCreate(stop)
8:     for state do
9:         cudaMemPrefetchAsync(ptr, bytes, src)      ▷ move pages to src
10:        cudaSetDevice(src)
11:        cudaDeviceSynchronize()
12:        cudaSetDevice(dst)
13:        cudaDeviceSynchronize()
14:        cudaEventRecord(start)
15:        cudaMemPrefetchAsync(ptr, bytes, src)      ▷ if prefetch, or...
16:        gpu_write<<<256,256>>>(ptr, bytes, pageSize) ▷ if coherence
17:        cudaEeventRecord(stop)
18:        cudaEventSynchronize(stop)
19:        millis ← cudaEventElapsedTime(start, stop)
20:        state.SetIterationTime(millis/1000)
21:    end for
22: end function
```

Algorithm 5.2 describes the approach to measure coherence or prefetch bandwidth from a CPU to a GPU in the unified memory system. First, execution is bound to the source NUMA node. Then, the destination CUDA device is set to be active, a *bytes*-sized unified memory allocation is created, and `cudaMemset` is used to ensure that pages for the allocation are created. During the event loop, `cudaMemPrefetchAsync` followed by `cudaDeviceSynchronize` ensure that unified memory pages are on the source CPU. Then, the `gpu_write` (Listing 5.2) is used to generate coherence requests to move pages to the destination GPU, or `cudaMemPrefetchAsync` is used to prefetch pages to the destination GPU. CUDA events are used to record the elapsed time, and that

is used as the benchmark iteration time. The benchmark is repeated five times to discover outliers and establish a standard deviation of measurement.

Algorithm 5.2 Measuring CPU-GPU unified memory coherence or prefetch bandwidth during a *bytes*-sized transfer between *src* and *dst*. `gpu_write` is defined in Listing 5.2.

```
1: function BANDWIDTH(dst, src, bytes)
2:     numa_bind_node(src)
3:     cudaSetDevice(dst)
4:     ptr ← cudaMallocManaged(bytes)
5:     cudaMemset(ptr, 0, bytes)              ▷ force pages to be allocated
6:     pageSize ← sysconf(_SC_PAGESIZE)
7:     cudaSetDevice(dst)
8:     cudaEventCreate(start)
9:     cudaEventCreate(stop)
10:    for state do
11:        cudaMemPrefetchAsync(ptr, bytes, cudaCpuDeviceId)    ▷ move
   pages to CPU
12:        cudaDeviceSynchronize()
13:        cudaEventRecord(start)
14:        cudaMemPrefetchAsync(ptr, bytes, src)          ▷ if prefetch, or...
15:        gpu_write<<<256,256>>>(ptr, bytes, pageSize)   ▷ if coherence
16:        cudaEventRecord(stop)
17:        cudaEventSynchronize(stop)
18:        millis ← cudaEventElapsedTime(start, stop)
19:        state.SetIterationTime(millis/1000)
20:    end for
21: end function
```

Algorithm 5.3 describes the approach to measure coherence bandwidth from a GPU to a CPU in the unified memory system. First, execution is bound to the destination NUMA node. Then, the source CUDA device is set to be active, a *bytes*-sized unified memory allocation is created, and `cudaMemset` is used to ensure that pages for the allocation are created. During the event loop, `cudaMemPrefetchAsync` followed by `cudaDeviceSynchronize` ensure that unified memory pages are on the source GPU. Then, the `cpu_write`

(Listing 5.2) is used to generate coherence requests to move pages to the destination CPU. The automatic benchmark timing may be used instead of CUDA events, as the CPU accesses are guaranteed to complete from the perspective of the CPU just like normal memory accesses. The benchmark is repeated five times to discover outliers and establish a standard deviation of measurement.

Algorithm 5.3 Measuring GPU-to-CPU unified memory coherence bandwidth during a *bytes*-sized transfer between *src* and *dst*. `cpu_write` is defined in Listing 5.1.

```
1:  function BANDWIDTH(dst, src, bytes)
2:      numa_bind_node(dst)
3:      cudaSetDevice(src)
4:      ptr ← cudaMallocManaged(bytes)
5:      cudaMemset(ptr, 0, bytes)            ▷ force pages to be allocated
6:      pageSize ← sysconf(_SC_PAGESIZE)
7:      cudaSetDevice(dst)
8:      for state do
9:          state.PauseTiming()
10:         cudaMemPrefetchAsync(ptr, bytes, src)
11:         cudaDeviceSynchronize()
12:         state.ResumeTiming()
13:         cpu_write(ptr, bytes, pageSize)
14:     end for
15: end function
```

Algorithm 5.4 describes the approach to measure coherence bandwidth from a GPU to a CPU in the unified memory system. It is the same as Algorithm 5.3, except CUDA events are used to time the asynchronous `cudaMemPrefetchAsync` workload. The benchmark is repeated five times to discover outliers and establish a standard deviation of measurement.

Algorithm 5.4 Measuring GPU-to-CPU unified memory prefetch bandwidth during a *bytes*-sized transfer between *src* and *dst*. `cpu_write` is defined in Listing 5.1.

1: **function** BANDWIDTH(*dst*, *src*, *bytes*)
2: numa_bind_node(*dst*)
3: cudaSetDevice(*src*)
4: *ptr* ← `cudaMallocManaged(`*bytes*`)`
5: `cudaMemset(`*ptr*`, 0,` *bytes*`)` ▷ force pages to be allocated
6: *pageSize* ← sysconf(_SC_PAGESIZE)
7: cudaSetDevice(*dst*)
8: cudaEventCreate(*start*)
9: cudaEventCreate(*stop*)
10: **for** *state* **do**
11: cudaMemPrefetchAsync(*ptr*, *bytes*, *src*)
12: cudaDeviceSynchronize()
13: cudaEventRecord(*start*)
14: cudaMemPrefetchAsync(*ptr*, *bytes*, cudaCpuDeviceId)
15: cudaEventRecord(*stop*)
16: cudaEventSynchronize(*stop*)
17: *millis* ← cudaEventElapsedTime(*start*, *stop*)
18: state.SetIterationTime($\frac{millis}{1000}$)
19: **end for**
20: **end function**

Listing 5.1 shows a simple function to write `sizeof(data_type)` bytes to every `stride` byte in a `count`-byte region starting at `ptr`. When `stride` is the page size, each page is written only once, doing the minimal amount of work to force a page migration.

Listing 5.1: `cpu_write` function.

```
static void
cpu_write(char *ptr, const size_t count, const size_t stride) {
  for (size_t i = 0; i < count; i += stride) {
    benchmark::DoNotOptimize(ptr[i] = 0);
  }
}
```

Listing 5.2 shows CUDA kernel to write `sizeof(data_type)` bytes to every

`stride` byte in a `count`-byte region starting at `ptr`. It assigns consecutive warps in the grid to handle consecutive writes, with a single thread from each warp doing a write. If there are too few warps in the grid to cover all writes, the grid loops over the required writes. Since warps execute in lockstep, this ensures the broadest simultaneous demands on the unified memory system without redundant work within a warp.

Listing 5.2: `gpu_write` function.

```
template <typename data_type>
__global__ void gpu_write(data_type *ptr,
                          const size_t count,
                          const size_t stride)
{

  size_t gx =
    blockIdx.x * blockDim.x + threadIdx.x;
  size_t lx = gx & 31;
  size_t wx = gx / 32;
  size_t numWarps =
    (gridDim.x * blockDim.x + 32 - 1) / 32;
  size_t numStrides = count / stride;
  size_t numData = count / sizeof(data_type);
  size_t dataPerStride =
    stride / sizeof(data_type);

  if (0 == lx)
  {
    for (; wx < numStrides; wx += numWarps)
    {
      const size_t id = wx * dataPerStride;
      if (id < numData)
      {
        ptr[id] = 0;
      }
    }
  }
}
```

5.1 Coherence vs. Prefetch Bandwidth

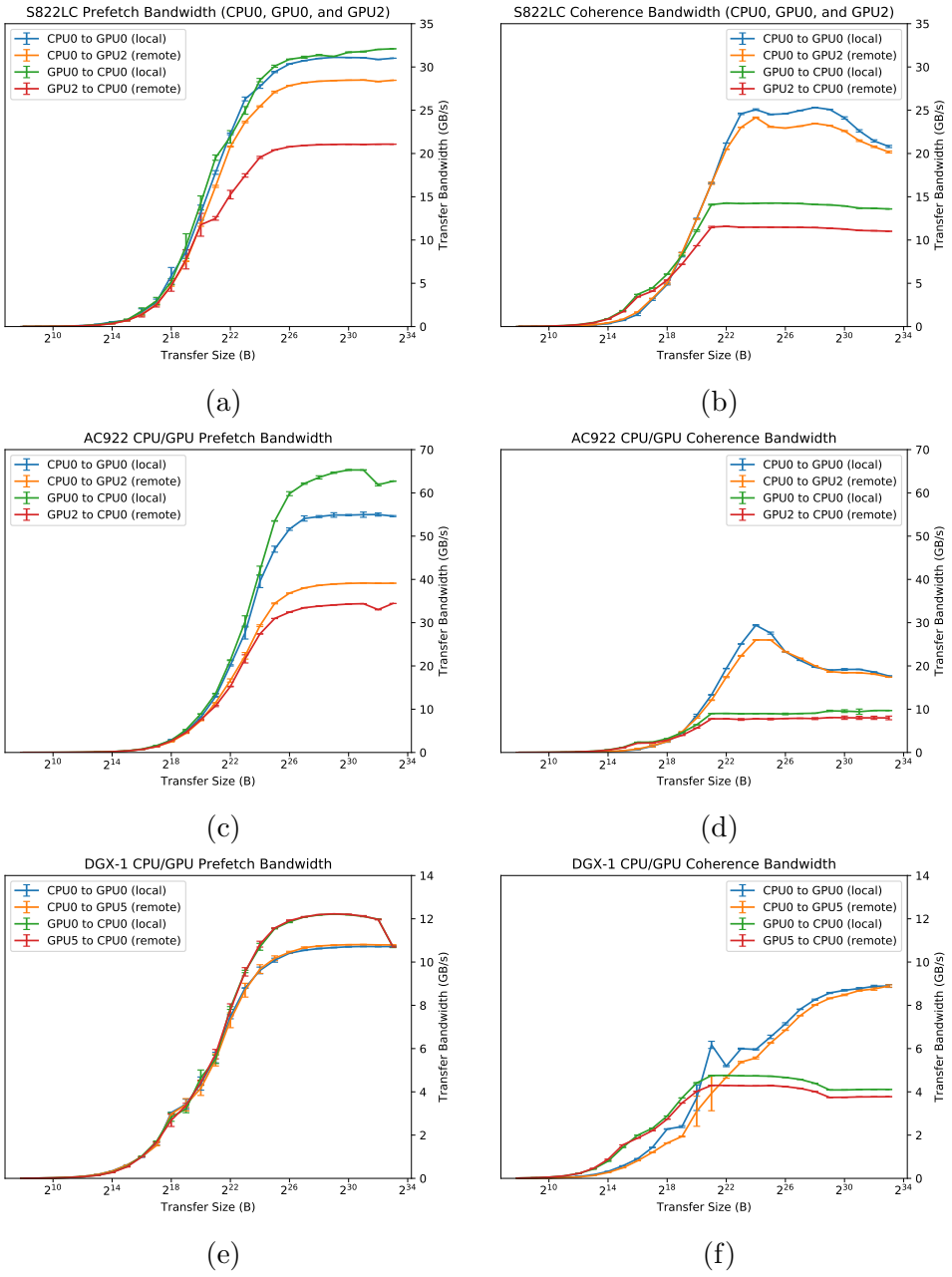


Figure 5.1: Measured CPU-GPU coherence and prefetch bandwidth vs. transfer size for S822LC, AC922, and DGX-1. Local and remote transfers in the CPU-to-GPU and GPU-to-CPU direction are shown.

Figure 5.1 compares CPU/GPU prefetch and coherence bandwidth on S822LC, AC922, and DGX-1. Prefetch provides substantially higher bandwidth than coherence demands, especially for large transfers. This is likely because the

DMA associated with a prefetch is a simpler and higher-performance operation than the on-demand migration of pages. Prefetch is still lower performance than CPU-to-GPU transfers from pinned memory. Prefetch must still obey consistency and coherency requirements of the unified memory system, which is a cost not present in explicit data transfer.

On the IBM systems, prefetch bandwidth between local devices is higher than between remote devices (Figures 5.1a and 5.1c). The hardware/software overhead of the unified memory prefetch is low enough to exercise a majority of the available underlying hardware links, so the more limited performance of the X bus in the remote transfers is likely the cause. The situation is reversed on DGX-1 (Figure 5.1e), where GPU-to-CPU transfers manage to match the performance of pinned explicit transfers, and CPU-to-GPU transfers do not. Overall, the prefetch bandwidth is closely correlated with the underlying hardware link performance, with AC922 providing much more bandwidth than the other two systems.

All three systems share behavior for coherence bandwidth (Figures 5.1b, 5.1d, and 5.1f). In these measurements, CPU-to-GPU performance exceeds GPU-to-CPU performance due to only using a single CPU thread to generate coherence requests. On S822LC and AC922, there is a peak in coherence bandwidth at intermediate transfer sizes. On DGX-1, coherence requests actually provide higher bandwidth at small transfer sizes. This may be due to reduced overhead for small number of coherence requests compared to setting up a bulk prefetch. Investigating the details of unified memory performance for different thread counts is under consideration as future work.

5.1.1 Device Affinity and Coherence Bandwidth

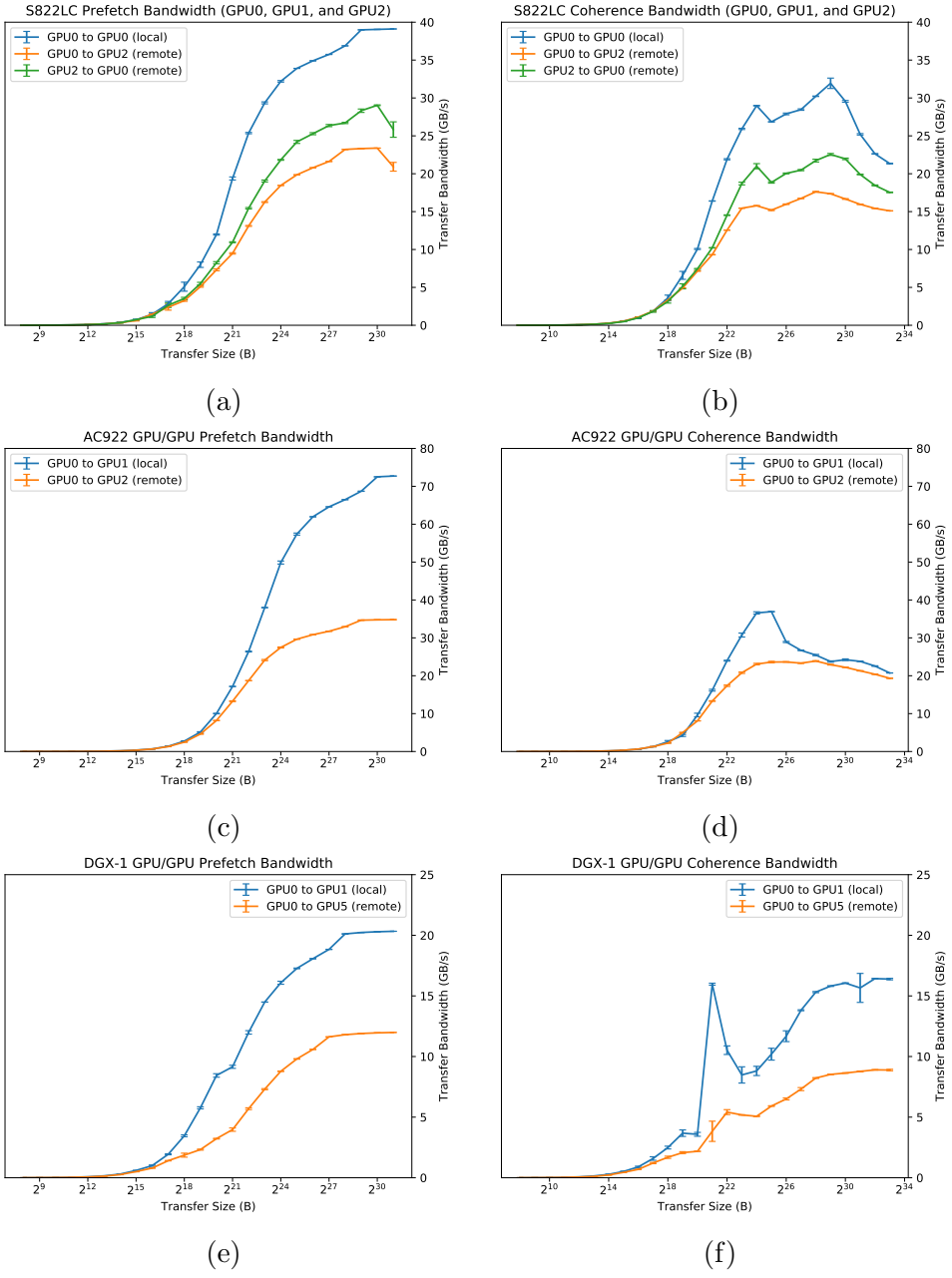


Figure 5.2: Measured GPU/GPU coherence and prefetch bandwidth vs. transfer size for S822LC, AC922, and DGX-1. Local and remote transfers in both directions for each pair of GPUs are shown. For S822LC, anisotropy on GPU-GPU transfers is also shown. Similar anisotropy was not observed on AC922 or DGX-1.

Observed coherence transfer bandwidth is correlated with device affinity; Table 5.1 summarizes the observed effects. The effect for CPU-to-GPU transfers is small but measurable on all three systems, as shown in Figures 5.1b, 5.1d, and 5.1f. There is also a small, measurable effect for GPU-to-CPU transfers, though using a single CPU thread may be obfuscating a larger difference. This small effect is probably due to overhead of ensuring coherence, so reduced link bandwidth in remote transfers is not a highly influential effect.

Figure 5.2 compares GPU/GPU prefetch and coherence bandwidth on S822LC, AC922, and DGX-1. For GPU-GPU transfers, the effect of device affinity on coherence bandwidth is much stronger. On S822LC, local transfers achieve around 30% higher bandwidth. The local bandwidth is nearly 100% higher on DGX, and also on AC922 for intermediate transfer sizes.

Table 5.1: Observed cases of device affinity affecting coherence bandwidth.

Transfer Kind	**S822LC**	**AC922**	**DGX-1**
CPU → GPU	small (Fig. 5.1b)	× (Fig. 5.1d)	small (Fig. 5.1f)
CPU ← GPU	✓(Fig. 5.1b)	small (Fig. 5.1d)	small (Fig. 5.1f)
GPU ↔ GPU	✓(Fig. 5.2b)	✓(Fig. 5.2d)	✓(Fig. 5.2f)

5.1.2 Device Affinity and on Prefetch Bandwidth

Device affinity can affect the observed prefetch bandwidth. Table 5.2 shows some cases where affinity affects prefetch transfer bandwidth. Figures 5.1a and 5.1c show that CPU/GPU prefetch bandwidth is strongly correlated with device affinity on S822LC and AC922. The overhead of prefetch transfers is lower than that of coherence transfers, and the availability of bandwidth on the underlying links has a large impact. On DGX-1 (Figure 5.1e), bandwidth is not at all correlated with device affinity; in fact, for GPU-to-CPU transfers, remote bandwidth is higher than local bandwidth. The hardware bandwidth on DGX-1 is much lower, and other implementation details control the performance.

Figures 5.2a, 5.2c, and 5.2e show that, like coherence bandwidth, GPU-GPU affinity is strongly correlated with prefetch bandwidth. On all systems, local GPUs can prefetch data much faster than remote GPUs. On S822LC, local GPUs enjoy 130% of the transfer bandwidth of their remote companions. On DGX-1, that number is 170%, and on AC922, it balloons to 230%. Generally,

local GPU-GPU transfers are able to saturate around 90% of the theoretical underlying link bandwidth, just like pinned transfers.

Table 5.2: Observed cases of device affinity affecting prefetch bandwidth.

Transfer Kind	**S822LC**	**AC922**	**DGX-1**
CPU → GPU	✓(Fig. 5.1a)	✓(Fig. 5.1c)	× (Fig. 5.1e)
CPU ← GPU	✓(Fig. 5.1a)	✓(Fig. 5.1c)	× (Fig. 5.1e)
GPU ↔ GPU	✓(Fig. 5.2a)	✓(Fig. 5.2c)	✓(Fig. 5.2e)

5.1.3 Observed Anisotropy in Coherence Bandwidth

Table 5.3 describes instances of observed anisotropy in coherence bandwidth. All CPU/GPU coherence transfers show anisotropy due to the single CPU thread vs. multiple GPU threads making accesses. On all systems, GPU-to-CPU transfers can be around twice the performance for small transfers, but for larger sizes the bandwidth saturates at the rate a single CPU thread can generate requests. On S822LC, Figure 5.2f, there is anisotropy present in remote GPU-to-GPU transfers. Similar GPU-GPU bandwidth anisotropy was not observed in any other cases.

Table 5.3: Cases where anisotropy is observed in coherence bandwidth.

Transfer Kind	**S822LC**	**AC922**	**DGX-1**
CPU ↔ GPU (local)	✓(Fig. 5.1b)	✓(Fig. 5.1d)	✓(Fig. 5.1f)
CPU ↔ GPU (remote)	✓(Fig. 5.1b)	✓(Fig. 5.1d)	✓(Fig. 5.1f)
GPU ↔ GPU (local)	× (Fig. 5.2b)	× (Fig. 5.2d)	× (Fig. 5.2f)
GPU ↔ GPU (remote)	✓(Fig. 5.2b)	× (Fig. 5.2d)	× (Fig. 5.2f)

5.1.4 Observed Anisotropy in Prefetch Bandwidth

Table 5.4 describes instances of observed anisotropy in prefetch bandwidth.

Figures 5.1a shows limited anisotropy on local CPU/GPU transfers on S822LC, though there is substantial anisotropy for remote CPU/GPU transfers. For S822LC remote transfers, as well as on AC922 and DGX-1, there is significant CPU/GPU coherence transfer anisotropy. On DGX-1, GPU-to-CPU transfers are always faster. On AC922 and S822LC, GPU-to-CPU

transfers are faster for local transfers and CPU-to-GPU transfers are faster for remote devices.

Table 5.4: Cases where anisotropy is observed in prefetch bandwidth.

Transfer Kind	**S822LC**	**AC922**	**DGX-1**
CPU ↔ GPU (local)	× (Fig. 5.1a)	✓(Fig. 5.1c)	✓(Fig. 5.1e)
CPU ↔ GPU (remote)	✓(Fig. 5.1a)	✓(Fig. 5.1c)	✓(Fig. 5.1e)
GPU ↔ GPU (local)	× (Fig. 5.2a)	× (Fig. 5.2c)	× (Fig. 5.2e)
GPU ↔ GPU (remote)	✓(Fig. 5.2a)	× (Fig. 5.2c)	× (Fig. 5.2e)

5.2 Page Fault Latency

Unified memory page fault latency is estimated by constructing a linked list in managed memory and traversing it. The list is realized as a unified memory array, where each element of the list (starting with the 0th offset of the array) contains the array offset that is the next element of the list. Algorithms 5.5 and 5.6 summarize the measurement routine for a transfer to a CPU and to a GPU, respectively. The main difference between the benchmarks is the timing method in the main benchmark loop. For CPU destinations, the automatic Google Benchmark timing is used, as the page will be resident on the CPU when the CPU load returns. For GPU destinations, CUDA events are used to measure the execution time of the CUDA kernel. The stride between linked list elements is a large number, to avoid prefetching effects on page faults. First, execution is bound to the relevant CPU and the relevant GPU is set as active. Then, a unified memory allocation is created and zeroed, to force it to be backed by pages. Then, the linked list is initialized in the unified memory allocation. In the benchmark loop, a destination-dependent list traversal function (shown in Listings 5.3 and 5.4) is executed on the destination device. 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.

Listings 5.3 and 5.4 show functions for traversing the linked list. Each function starts at the beginning of the array, and reads the offset of the next element to read from the current offset. Finally, a value is written to the final element to introduce a side-effect and prevent the otherwise read-only function

Algorithm 5.5 Measuring unified memory page fault latency page fault latency with a CPU destination. `cpu_traverse` is defined in Listing 5.4.

1: **function** LATENCY(*dst*, *src*, *ptr*, *stride*)
2: numa_bind_node(*dst*)
3: cudaSetDevice(*src*)
4: *stride* ← PAGE_SIZE ×2
5: *bytes* ← sizeof(size_t) ×(*steps* + 1) × *stride*
6: *ptr* ← cudaMallocManaged(*bytes*))
7: cudaMemset(*ptr*, 0, *bytes*)
8: **for** *i* in *steps* **do** ▷ set up stride pattern
9: ptr[*i*] ← (*i* + 1) × *stride*
10: **end for**
11: cudaDeviceSynchronize()
12: **for** state **do**
13: state.PauseTiming()
14: cudaMemPrefetchAsync(*ptr*, *bytes*, *src*)
15: cudaDeviceSynchronize()
16: state.ResumeTiming()
17: cpu_traverse(*ptr*, *steps*)
18: **end for**
19: **end function**

from being optimized away. If the stride between elements is constructed appropriately, and pages are not present on the executing device, each access will incur a page fault.

Listing 5.3: GPU linked list traversal kernel for Algorithm 5.6.

```
__global__ void gpu_traverse(size_t *ptr,
                             const size_t steps)
{
  size_t next = 0;
  for (int i = 0; i < steps; ++i)
  {
        next = ptr[next];
  }
  ptr[next] = 1;
}
```

Algorithm 5.6 Measuring unified memory page fault latency page fault latency with a GPU destination. `gpu_traverse` is defined in Listing 5.3.

```
1: function LATENCY(dst, src, ptr, stride)
2:     if src is CPU then
3:         numa_bind_node(dst)
4:     end if
5:     cudaSetDevice(dst)
6:     stride ← PAGE_SIZE ×2
7:     bytes ← sizeof(size_t) ×(steps + 1) × stride
8:     ptr ← cudaMallocManaged(bytes))
9:     cudaMemset(ptr, 0, bytes)
10:    for i in steps do                          ▷ set up stride pattern
11:        ptr[i] ← (i + 1) × stride
12:    end for
13:    cudaDeviceSynchronize()
14:    cudaEventCreate(start)
15:    cudaEventCreate(stop)
16:    for state do
17:        if src is CPU then
18:            cudaMemPrefetchAsync(ptr, bytes, cudaCpuDeviceId)
19:        else
20:            cudaMemPrefetchAsync(ptr, bytes, src)      ▷ ptr to src GPU
21:            cudaSetDevice(src)
22:            cudaDeviceSynchronize()
23:            cudaSetDevice(dst)
24:        end if
25:        cudaDeviceSynchronize()
26:        cudaEventRecord(start)
27:        gpu_traverse(ptr, steps)
28:        cudaEventRecord(stop)
29:        cudaEventSynchronize(stop)
30:        millis ← cudaEventElapsedTime(start, stop)
31:    end for
32:    state.SetIterationTime(millis/1000)
33: end function
```

Listing 5.4: CPU linked list traversal function for Algorithm 5.5.

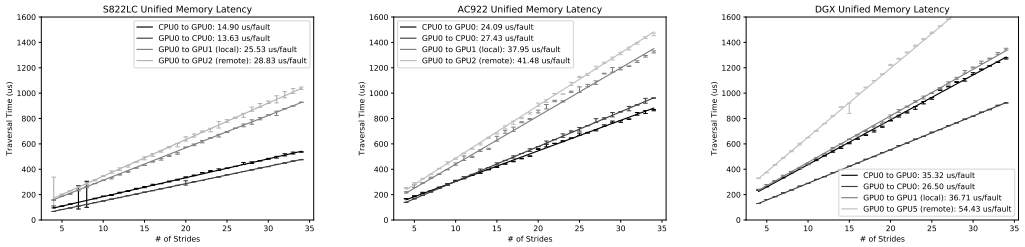
```
void cpu_traverse(size_t *ptr, const size_t steps)
{
  size_t next = 0;
  for (size_t i = 0; i < steps; ++i)
  {
        next = ptr[next];
  }
  ptr[next] = 1;
}
```

Table 5.5 summarizes the estimated page fault latencies. Figure 5.3 shows the raw traversal times. There is no substantial difference in page fault latencies for different CPUs, so values for transfers to CPU0 are shown. Each traversal is run at least 200 times, and the average value is reported. Standard deviations are computed from five repetitions of the entire benchmark procedure.

All three systems follow the same behavior, with GPU/GPU page fault latencies being higher than CPU/GPU page fault latencies. The bottom section of Table 5.5 shows that the cost of moving the page is a small portion of the observed transfer time. These calculated values do not include link latencies, and represent a lower bound. On S822LC and AC922, the CPU/GPU page fault latency is nearly identical in both directions.

Table 5.5: Measured page-fault latencies on S822LC, AC922, and DGX-1. Above the double-line are empirically-measured values. Below the double-line are computed values, for the system page size and system configuration. The value is computed by taking the link bandwidth and dividing it by the page size; e.g., NVLink for S822LC refers to two-lane NVLink 1.0.

Page Fault	**Latency (μs)**		
Type	**S822LC**	**AC922**	**DGX-1**
CPU $\rightarrow$ GPU	14.9	24.1	35.3
CPU $\leftarrow$ GPU	13.6	27.4	26.5
GPU0 $\leftrightarrow$ GPU1 (local)	25.5	38.0	36.7
GPU0 $\leftrightarrow$ GPU2 (remote)	28.8	41.5	54.4
One Page, PCIe 3.0 x16	N/A	N/A	0.25
One Page, NVLink	1.6	0.9	0.4



(a) (b) (c)

Figure 5.3: Linked-list traversal time vs. number of strides for S822LC, AC922, and DGX-1. For each system, CPU-to-GPU, GPU-to-CPU, and remote/local GPU-to-GPU times are shown. Each stride incurs a page fault, so the slopes of these lines estimate the page fault cost. Whiskers at each point are the standard deviation over five benchmark repetitions.

5.3 Summary

The unified memory system comes at a significant performance penalty compared to explicit memory management, and suffers from similar topological performance effects. Unsurprisingly, performance is strongly correlated with device affinity, particularly for prefetch bandwidth. Performance anisotropy is observed in nearly all prefetch benchmarks and many coherence benchmarks. Achievable coherence performance ranges from around 30%-90% of the best possible explicit performance, with more degradation on higher-performance links. With the additional programmer effort of including prefetching hints, performance nearly equal to the explicit data transfer can be achieved.

CHAPTER 6

FUTURE WORK: APPLICATION CHARACTERIZATION AND COMBINED MODELING

6.1 Measuring Additional Communication Capabilities

The microbenchmarks presented in Chapters 4 and 5 cover CUDA transfers with the unified memory system and `cudaMemcpy`. There are two additional CUDA communication capabilities that are not measured in this thesis: direct peer access and system atomics. Expanding the benchmarks to measure those methods would provide a more complete view of CUDA communication performance. The benchmarks could also be expanded to examine interaction between CUDA and other system components. One such opportunity is the GPUDirect capability, where supported non-GPU devices can communicate with CUDA GPUs through DMA. Another opportunity is to investigate the effect of contention on the communication performance. Such contention could occur in bi-directional data transfers, or when multiple devices are simultaneously communicating.

6.2 Mapping Logical Communication to Underlying Links

Previous sections described the approach and results of measuring the data transfer performance of the logical communication paths. Those models took some knowledge of the underlying system configuration as *a priori* knowledge, but in general this process should be automated. Application developers who want to understand the performance of the system may not have the architecture background to select appropriate performance models. System architects who develop system capabilities may not understand the implications their choices have on performance at the top abstraction layer. This work presents

an initial step towards automation with automatic hardware enumeration described in Section 3.2.

The next step is to systematically instrument all possible links for observation during the execution of applications. If the instrumentation is low-overhead, this could be done jointly with logical performance measurements. If not, known, separate workloads could be sent over logical communication paths to observe hardware links. For NVLink monitoring, NVML provides the ability to query the link utilization counters [54]. For PCIe monitoring, the Performance Counter Monitor [55] project provides the ability to query PCIe link counters.

For each enumerated link, the utilized hardware components could be associated and used to select the appropriate performance model. Follow-up work investigating the feasibility of using a single parameterized model per combination of logical communication path and hardware link. For example, consider a simple model to compute the transfer time $t_{transfer}$ for moving *bytes* bytes between some CPU and GPU:

$$t_{transfer} = c + t_{bytes}$$

where t_{bytes} is defined as

$$\begin{cases} max(BW_{link}, \frac{bytes}{BW_{L1}}) & 0 \leq bytes < L1_{size} \\ max(BW_{link}, \frac{bytes}{BW_{L2}}) & L1_{size} \leq bytes < L2_{size} \\ max(BW_{link}, \frac{bytes}{BW_{L3}}) & L2_{size} \leq bytes < L3_{size} \\ max(BW_{link}, \frac{bytes}{BW_{mem}}) & L3_{size} \leq bytes \end{cases}$$

c is some constant overhead, and BW_{link}, BW_{L1}, BW_{L2}, BW_{L3}, and BW_{mem} are the bandwidths of the CPU-GPU link, L1 cache, L2 cache, L3 cache, and main memory.

In this model, the bandwidth is a piecewise function that depends on the transfer size - if a transfer fits within a cache, it happens at the speed of that cache. In all cases, the transfer never happens faster than the underlying CPU-GPU link bandwidth. The empirical results from the benchmark can be used to determine the constants in this model.

6.3 Application Model

After a model of system communication performance is established, the next step is to establish an understanding of how applications use the logical communication paths. This would encapsulate information about how an application produces, moves, and consumes data. The application can be modeled as a dynamic value dependence graph (DVDG) $G_a = \{E_a, V_a\}$, where E_a is a set of edges representing data transfer, and V_a is a set of vertices representing data values. Each value represents a contiguous region of memory that the program interacts with. Each edge is either an explicit memory copy or a CUDA kernel launch. The edges can be tagged with observed timestamps to facilitate program performance analysis.

Figure 6.1 shows an example code and corresponding DVDG. Each value, represented by square boxes, is tagged with the position of the value, and size of the value. For simplicity's sake, it is also tagged with the line of code that produced it. Each rounded-box edge label corresponds to the `cudaMemcpy` or kernel execution. In practice, each node and edge would have much more detailed information.

Though it is generally possible to generate these dependence graphs by hand, it is not feasible for complicated applications. Furthermore, as applications are updated, new models would have to be manually generated. The proposed method described in this section would automate that process.

6.4 Constructing the Dynamic Value Dependence Graph for Unmodified Applications

Ideally, the DVDG would be generated from an unmodified application execution. This ensures that the tool is as accessible to users as possible, and that it can work on closed-source applications. Unfortunately, the DVDG cannot rely on the application to advertise any helpful information about its behavior, and any relevant application state must be observed through its interaction with the operating system. The proposed system ("apptracer") leverages two tools available on the Linux platform: the CUDA Profiling Tools Interface (Section 2.7.2 (CUPTI) and the Linux `LD_PRELOAD` (Section 2.7.3) mechanism.

```
 1. float *a_h, *b_h, *s_h, *a_d, *b_d, *s_d;
 2. malloc(a_h, 1024);       // a_h = 0x0000
 3. malloc(b_h, 1024);       // b_h = 0x0400
 4. malloc(s_h, 1024);       // s_h = 0x0800
 5. cudaMalloc(&a_d, 1024); // a_d = 0x0c00
 6. cudaMalloc(&b_d, 1024); // b_d = 0x1000
 7. cudaMalloc(&s_d, 1024); // s_d = 0x1400
 8. cudaMemcpy(a_d, a_h, 1024, cudaMemcpyDefault);
 9. cudaMemcpy(b_d, b_h, 1024, cudaMemcpyDefault);
10. vector_add<<<gd, bb>>>(s_d, a_d, b_d, 1024);
11. cudaMemcpy(s_h, s_d, 1024, cudaMemcpyDefault);
```

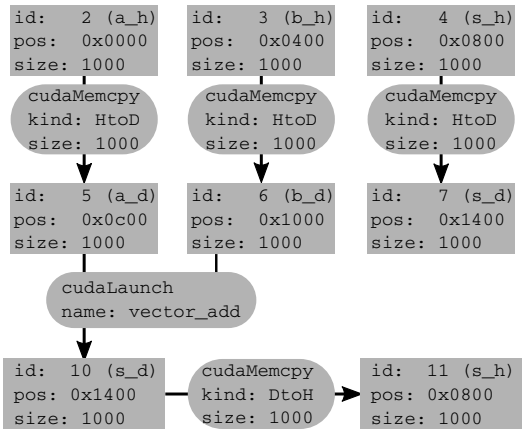


Figure 6.1: Example of the dynamic value dependence graph for a simple vector add. Allocations in the code are commented with a hypothetical position of the allocation. Square blocks represent values, and rounded boxes represent node labels.

`Apptracer` would use CUPTI to capture most CUDA-related information, and LD_PRELOAD for everything else. CUPTI allows a tool to provide a callback function that is invoked at every CUDA runtime or driver call, and also allows `apptracer` to collect any performance metrics the GPU exposes. The callback function records relevant information, including the wall time when the CUDA runtime function is invoked, its arguments, and the device and stream associated with the call. In this way, detailed information about data transfers from runtime functions can be reconstructed. For example, allocations from `cudaMalloc` can be associated with pointers passed to `cudaMemcpy` to discover data transfers from host to device.

`Apptracer` would use `LD_PRELOAD` (Section 2.7.3) to intercept known API calls made by the application to shared libraries. For example, `LD_PRELOAD` could be used to observe file access, calls to CUDA libraries such as cuDNN or cuBLAS, network access, and system memory allocations. The various

kernel launches and allocations used by cuDNN and cuBLAS are already visible through CUPTI, but the known semantics of the higher-level cuBLAS and cuDNN calls allow for detailed edges to be added.

One challenge of `apptracer` is handling implicit data movement from three scenarios: data moved from GPU global memory to arbitrary GPU kernels, implicit data movement between remote mappings, and implicit data movement through the unified memory system. On supported systems, GPUs can directly access data that is on the host or other GPUs without making any CUDA runtime calls. CUPTI allows the GPU to record detailed profiling information, but this affects program execution time and distorts the timeline. A two-pass approach, once to collect accurate timeline information and another to capture more detailed information, may be a solution.

Another challenge is for `apptracer` to infer which kernel arguments are pointers to allocations. It may be possible for `apptracer` to examine the intermediate PTX representation of CUDA kernel code embedded in most CUDA binaries, and deduce some information about function signatures.

Once the application's use of CUDA is recorded, the next step would be to extend the graph to some view of activity on the CPU as well. This could be accomplished through a variety of techniques. The `LD_PRELOAD` mechanism could be used to instrument popular libraries such as BLAS or MPI. The operating system trace facilities strace [56] for Linux, DTrace [57] for MacOS, NtTrace [58], or Dr. Memory [59], [60] for Windows could be used to track system calls and profile things such as file I/O or network interaction. Tracking arbitrary function calls within an unmodified application may not be possible in the case of a binary without debug symbols. Dynamic tracing tools like Intel's PIN [61] can insert instrumentation code, but further work would be needed to determine the number of function arguments and their locations to recover their values. For a cluster environment, it may be possible to generate distributed dependence graphs at each node, and then join them together by linking together information recorded about MPI calls.

6.5 Combined Modeling

Finally, once system performance modeling is established and application demands are recorded, joint performance modeling is possible to tackle questions

like

- For a particular DVDG, what performance could we expect on a particular system?
- For a particular DVDG and a particular budget, what system configuration would perform best?
- For a particular DVDG, can the execution be rescheduled on a system to improve performance?
- How would changing link parameters or topology on a particular system affect application performance?

Answering these questions may require additional effort in open research challenges, such as task scheduling with placement-dependent communication costs, compute kernel performance estimation, and design space exploration.

CHAPTER 7

RELATED WORK

7.1 System Topology Enumeration / Hardware Models

This work relies on and enhances an existing system topology enumeration tool `hwloc` [52], which is designed around the expectation that current and next-generation systems are hierarchical. This work uses `hwloc` to enumerate topology, but embeds the devices that `hwloc` discovers in a graph (Section 3.2), which is a more general model of modern hardware systems.

Amaral et al. [62] use a similar hardware graph, but their path costs are defined qualitatively, whereas this work proposes automating a quantitative cost that depends on the communication method used. They also describe a topology-aware job placement strategy, but the problem considered is jobs on nodes in a cluster environment instead of computation tasks and data placement on GPUs in a multi-GPU node.

7.2 System Characterization

Several prior benchmark suites strive to determine system parameters through microbenchmarking. LMBench [63] is a benchmark designed to determine memory hierarchy parameters. It includes a single-threaded memory bandwidth benchmark similar to the one included in this work. It also includes cached I/O bandwidth measurements, a logical communication path that this work will be extended to explore. P-Ray [64] is a benchmark suite designed to help guide performance autotuners. It is designed to discover hardware parameters, such as L2 cache size. Among other things, it extends LMBench's memory bandwidth microbenchmark to include multithreaded transfers. Servet [65] goes a step further and includes a benchmark to deter-

mine communication costs between pairs of cores in the context of MPI. It also attempts to analyze the results to establish which cores are equivalent from a communication perspective to simplify the benchmarking process. The BlackjackBench [66] benchmark suite is designed to measure the observable performance parameters of a system as opposed to hardware parameters. This work also takes the view that observable system parameters are what matters to application performance. It focuses on memory hierarchy performance, but also includes a workload to measure communication bandwidth between pairs of CPU cores. Liu et al. [67] take a microbenchmarking approach to evaluating high-speed cluster interconnects, including latency, uni- and bi-directional bandwidth, and host latency. This work takes a similar approach or subsumes the communication microbenchmarking of this previous work, but extends it to many kinds of CUDA communication.

This work also proposes correlating microbenchmark performance directly with underlying hardware. McCurdy and Vetter [68] describe using performance counters to examine the NUMA abstraction and determine its mapping to the underlying hardware. This work proposes to automate that analysis in Section 3.2.4.

Some GPGPU benchmark suites also make an effort to characterize certain aspects of data-transfer performance. The Scalable Heterogeneous Computing (SHOC) benchmark suite [69] examines host-to-device and device-to-host data transfer on some PCIe-based systems. This work examines a much broader set of CUDA communication capabilities such as prefetch and coherence transfers in unified-memory systems. SHOC also examines the latency effect of data transfers conflicting with MPI message sending on the PCIe bus. This contention effect is similar to some future contention characterization this work will be extended to.

Prior work has also specifically examined the communication performance effect of incorrect NUMA pinnings in multi-GPU systems. Spafford, Meredith, and Vetter [70] show significant anisotropy and bandwidth degradation in PCIe bandwidth for incorrect NUMA pinnings. They also discuss how application performance can degrade under PCIe bus contention. Expansion of the benchmarks in this work will most likely include bus contention, as multi-GPU applications will likely consist of phases with multi-device communication.

Limited prior work has examined performance of CUDA transfers specifically. Landaverde et al. [71] develop a set of microbenchmarks to measure

performance of unified memory accesses for particular access patterns. Their work focuses on the performance effect of unified memory on particular computation patterns, while this work isolates NUMA effects and data transfer performance alone. Li et al. [72] evaluate the unified memory system on several platforms, including a multi-CPU PCIe platform, and show around a 10% performance penalty on some applications for unified memory. They do not do any microbenchmarking, but observe that the unified memory system in CUDA 6.0 produced redundant transfers that were avoiding in the explicitly-managed code. MGBench, a multi-GPU communication benchmark [73], contains multi-GU microbenchmarks including scatter, direct access, and ring broadcast messages. Ben-Nun et al. [74] examine direct-access transfers between GPUs. They show the transfer rate of direct-access transfers between local GPUs, remote GPUs, and CPU/GPU transfers with various access patterns. They observe that the performance is highly dependent on the access pattern. Like this work, they discover that the transfer rate is highly correlated with the topological proximity of the devices. Github user `woodun` has a set of open-source, unpublished microbenchmarks under the name `9_Microbenchmarks` [75]. These benchmarks largely focus on details of Nvidia Pascal and Volta memory hierarchy, including cache lines sizes, and costs of various operations, including TLB misses and page faults under different CUDA device configurations and access patterns. As of the time of this writing, no results are presented and there is no discussion of the motivation of the benchmarks. Some of these workloads seem to attempt to measure some of the same operations in this work, such as page fault latency.

7.3 Using Communication Models

Related works make use of the communication costs to make scheduling decisions. MPIPP [76] uses communication parameters in its process placement routine, but it gets them from the technical specification of the machine instead of measuring it. Mercier and Clet-Ortega [77] also use communication parameters in their placement policy. They determine the topology of the machine from the specification and estimate the communication costs from that topology.

7.4 NUMA / Multi-GPU APIs

Prior work has made an effort to create NUMA-aware APIs, a possible future direction of this work. Ben-Nun et al. [78] describe a multi-GPU partitioning framework for distributing parallel workloads on multi-GPU nodes according to their access patterns. It provides a set of host and device APIs that describe containers and allow the framework to analyze kernels to determine access patters, to decide how to schedule underlying operations onto multiple GPUs. Groute [74] makes use of parallel constructs for asynchronous multi-GPU programming. The programming model involves describing the application communication pattern as a graph of communicating links and endpoints. Section 6.3 of this work describes the DVDG, which would be used to construct a similar application model for an existing application.

Umpire [79] is a library that facilitates the discovery and provision of memory on next-generation architectures. Umpire aims to provide NUMA- and GPU-aware allocators and allocation strategies. At present, it seems to largely wrap existing host and device allocators, but also provides three different pool allocation strategies. Umpire may eventually function as a foundation for a high-performance memory abstraction for heterogeneous systems.

CHAPTER 8

CONCLUSION

This thesis examines the data-movement performance available to applications running on multi-GPU non-uniform memory access (NUMA) systems through a comprehensive series of microbenchmarks. Three different systems are used as case studies: an IBM S822LC with two POWER8 CPUs and four P100 GPUs (Section 2.9.2), an IBM AC922 with two POWER9 CPUs, four V100 GPUs, and NVLink 2.0 (Section 2.9.3), and an Nvidia DGX-1 with two Intel (Section 2.9.1), eight P100 GPUs, and hybrid PCIe and NVLink 1.0. These three systems cover the common component for high-performance multi-GPU NUMA systems, with multiple CPUs, NVLink and PCIe 3.0 x16, and Pascal- or Volta-architecture GPUs. The performance of the underlying hardware (Section 2.3) in these systems is made available to applications through software abstractions (Section 2.1) like CUDA and numactl. CUDA provides a variety of methods for moving data between system components (Section 2.4), where different choices have different performance effects due to different uses of the underlying hardware.

The core of this thesis consists of performance measurements of explicit (Chapter 4) and unified-memory (Chapter 5) data-movement systems in CUDA. These measurements are generated using a new series of microbenchmarks available at `https://github.com/rai-project/microbench`. Generally, these benchmarks reveal that the CUDA method used to move data has a substantial impact on the actual performance available to the application, in some cases up to a 3x difference, as in the case of pinned transfers and coherence transfers. When the data transfer method is simpler (pinned or prefetch transfers), performance is highly correlated with device affinity, but typically presents less anisotropic behavior. This is because the underlying hardware link performance limits the overall throughput. For more complicated transfer methods such as coherence, pageable transfers, or non-peer GPU-GPU transfers, performance tends to be more anisotropic but less corre-

lated with device affinity. The software overhead of managing these transfers tends to limit the performance more than the underlying links. These results can be used to inform communication and allocation choices during application development, or allow an automated system to make the appropriate data-movement decision to provide the best performance.

In addition to the core performance measurements, this thesis also introduced a tool for enumerating the different communication hardware present in the system. This tool is meant to serve as a foundational component of future systems research. Any system that needs to make communication or scheduling performance decisions will need information about the underlying hardware that can be provided by this tool. The system abstraction available through CUDA, the performance measurements, and the underlying hardware topology together provide the raw information for a detailed system model.

To complete the model, it will be necessary to automatically correlate communication abstractions to the underlying hardware (Section 6.2). This thesis also describes a corresponding application model (Section 6.3) that can be coupled with the system model. Together these models should provide enough information to create an automatically-tuned high-performance heterogeneous memory management capability for multi-GPU NUMA systems.

REFERENCES

[1] Intel, "Intel AVX," 2017. [Online]. Available: https://software.intel.com/en-us/isa-extensions/intel-avx

[2] *System V Application Binary Interface AMD64 Architecture Processor Supplement Draft Version 0.99.4*, Advanced Micro Devices Std. 0.99.4, 2010.

[3] ARM, "NEON," 2017. [Online]. Available: https://developer.arm.com/technologies/neon

[4] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers et al., "In-datacenter performance analysis of a tensor processing unit," *arXiv preprint arXiv:1704.04760*, 2017.

[5] Huawei, "Kirin 970," 2017. [Online]. Available: http://consumer.huawei.com/en/press/news/2017/ifa2017-kirin970/

[6] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura et al., "A million spiking-neuron integrated circuit with a scalable communication network and interface," *Science*, vol. 345, no. 6197, pp. 668–673, 2014.

[7] Intel, "Intel Nervana hardware," 2017. [Online]. Available: https://www.intelnervana.com/intel-nervana-hardware/

[8] A. Kandangath and G. Badie, "What's new in core motion," 2015, Apple. [Online]. Available: http://devstreaming.apple.com/videos/wwdc/2015/705qrxhfxo0/705/705_whats_new_in_core_motion.pdf

[9] "Virtex UltraSCALE+ product table," 2018, Xilinx. [Online]. Available: https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus.html\#productTable

[10] "Overview," 2018, Intel. [Online]. Available: https://www.altera.com/products/fpga/stratix-series/stratix-10/overview.html

[11] "About agilio SmartNICs," 2018, Netronome. [Online]. Available: https://www.netronome.com/products/smartnic/overview/

[12] L. Codrescu, "Qualcomm Hexagon DSP," 2013, Qualcomm. [Online]. Available: https://developer.qualcomm.com/qfile/27696/qualcomm-hexagon-architecture.pdf

[13] "Digital signal processors," 2018, NXP. [Online]. Available: https://www.nxp.com/products/processors-and-microcontrollers/additional-processors-and-mcus/digital-signal-processors:Digital-Signal-Processors

[14] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017.

[15] "Vision processing unit," 2018, Movidius. [Online]. Available: https://www.movidius.com/solutions/vision-processing-unit

[16] "The evolution of EyeQ," 2018, Mobileye. [Online]. Available: https://www.mobileye.com/our-technology/evolution-eyeq-chip/

[17] N. Baker, "Mixed reality," Youtube. [Online]. Available: https://www.youtube.com/watch?v=u0eBd2m_wEs\#t==27m16s

[18] A. M. Devices, "AMD Opteron 6200 series processor quick reference guide," 2012. [Online]. Available: https://www.amd.com/Documents/Opteron_6000_QRG.pdf

[19] *Intel Xeon Platinum 8180M Processor*, Intel, 2018. [Online]. Available: https://ark.intel.com/products/120498/Intel-Xeon-Platinum-8180M-Processor-38_5M-Cache-2_50-GHz

[20] J. Stuecheli, "An introduction to POWER8 processor," 2013. [Online]. Available: http://www.idh.ch/IBM_TU_2013/Power8.pdf

[21] B. Thompto, "POWER9: Processor for the cognitive era," in *Hot Chips 28 Symposium (HCS), 2016 IEEE*. IEEE, 2016, pp. 1–19.

[22] "Hardware summary," 2018, National Center for Supercomputing Applications. [Online]. Available: https://bluewaters.ncsa.illinois.edu/hardware-summary

[23] "Summit," 2018, Oak Ridge National Laboratory. [Online]. Available: https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/

[24] "Sierra," 2018, Lawrence Livermore National Laboratory. [Online]. Available: https://computation.llnl.gov/computers/sierra

[25] *PCI Express Base Specification*, PCI SIG Std. 1.0, 2002.

[26] *PCI Express Base Specification*, PCI SIG Std. 3.0, 2010.

[27] L. Nyland, "Inside Kepler," 2012, Nvidia. [Online]. Available: http://on-demand.gputechconf.com/gtc/2012/presentations/S0642-GTC2012-Inside-Kepler.pdf

[28] M. Harris, "Inside Pascal: NVIDIA's newest computing platform," 2016. [Online]. Available: https://devblogs.nvidia.com/inside-pascal/

[29] *NVIDIA DGX-1*, Nvidia, 2017.

[30] "NVIDIA Tesla P100," 2016, Nvidia. [Online]. Available: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

[31] "NVIDIA Tesla V100 GPU architecture," 2017, Nvidia. [Online]. Available: http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[32] T. C. Schroeder, "Peer-to-peer and unified virtual addressing," in *GPU Technology Conference, NVIDIA*, 2011.

[33] *CUDA Runtime API*, Nvidia, Feb 2014.

[34] *NVIDIA CUDA Compute Unified Device Architecture*, Nvidia Std. 1.0, Jun 2007.

[35] *NVIDIA CUDA Reference Manual*, Nvidia Std. 3.0, Feb 2010.

[36] *NVIDIA CUDA C Programming Guide*, Nvidia Std. 4.0, May 2011.

[37] *CUDA C Programming Guide*, Nvidia Std. 6.0, Feb 2014.

[38] M. Harris, "Unified memory in CUDA 6," 2013. [Online]. Available: https://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/

[39] *CUDA C Programming Guide*, Nvidia Std. 8.0, Jan 2017.

[40] N. Sakharnykh, "Unified memory on Pascal and Volta," in *GPU Technology Conference, NVIDIA*, 2017.

[41] *numa(3) Linux Programmer's Manual*, August 2007.

[42] C. Wickman, C. Lameter, and L. Schermerhorn, "numactl v2.0.11," https://github.com/numactl/numactl, 2015.

[43] *OpenMP Application Program Interface*, OpenMP Architecture Review Board Std. 4.0, 2013.

[44] Nvidia, "CUDA profiling tools interface," 2017. [Online]. Available: https://developer.nvidia.com/cuda-profiling-tools-interface/

[45] M. Kerrisk, "Linux programmer's manual, ld.so," 2017. [Online]. Available: http://man7.org/linux/man-pages/man8/ld.so.8.html/

[46] M. Kerrisk, "Linux programmer's manual, DLSYM(3)," 2017. [Online]. Available: http://man7.org/linux/man-pages/man3/dlsym.3.html

[47] "Benchmark," 2018, Google Inc. [Online]. Available: https://github.com/google/benchmark

[48] A. B. Caldeira, V. Haug, and S. Vetter, "IBM power system 822LC for High Performance Computing introduction and technical overview," *IBM Redbooks*, 2016.

[49] A. B. Caldeira, "IBM power system AC922 introduction and technical overview," *IBM Redbooks*, 2018.

[50] C. Pearson, "hwcomm," 2018. [Online]. Available: https://github.com/illinois-impact/hwcomm

[51] Nvidia, "NVIDIA management library (nvml)," 2017. [Online]. Available: https://developer.nvidia.com/nvidia-management-library-nvml/

[52] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "hwloc: A generic framework for managing hardware affinities in hpc applications," in *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*. IEEE, 2010, pp. 180–186.

[53] C. Pearson, A. Dakkak, and C. Li, "microbench," 2018. [Online]. Available: https://github.com/rai-project/microbench

[54] *NVML Reference Manual*, Nvidia, Oct 2017.

[55] "Processor counter monitor." [Online]. Available: https://github.com/opcm/pcm

[56] *strace(1) - Linux man page*, 2018. [Online]. Available: https://linux.die.net/man/1/strace

[57] *Dynamic Tracing Guide*, Sun Microsystems, 2008. [Online]. Available: http://dtrace.org/guide/preface.html\#preface

[58] R. Orr, "NtTrace - native API tracing for Windows," 2014, OR/2. [Online]. Available: http://www.howzatt.demon.co.uk/NtTrace/

[59] D. Bruening, E. Duesterwald, and S. Amarasinghe, "Design and implementation of a dynamic optimization framework for Windows," in *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, 2001.

[60] "Dr. Memory," 2018, Dr. Memory. [Online]. Available: http://drmemory.org/

[61] N. S., "PIN - a dynamic binary instrumentation tool," 2012, Intel. [Online]. Available: https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

[62] M. Amaral, J. Polo, D. Carrera, S. Seelam, and M. Steinder, "Topology-aware GPU scheduling for learning workloads in cloud environments," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2017, p. 17.

[63] L. W. McVoy, C. Staelin et al., "lmbench: Portable tools for performance analysis." in *USENIX annual technical conference*. San Diego, CA, USA, 1996, pp. 279–294.

[64] A. X. Duchateau, A. Sidelnik, M. J. Garzarán, and D. A. Padua, "P-ray: A suite of micro-benchmarks for multi-core architectures," in *Proc. 21st Intl. Workshop on Languages and Compilers for Parallel Computing (LCPC08)*, vol. 5335. Citeseer, 2008, pp. 187–201.

[65] J. González-Domínguez, G. L. Taboada, B. B. Fragüela, M. J. Martín, and J. Tourino, "Servet: A benchmark suite for autotuning on multicore clusters," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–9.

[66] A. Danalis, P. Luszczek, G. Marin, J. S. Vetter, and J. Dongarra, "BlackjackBench: portable hardware characterization," *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 2, pp. 74–79, 2012.

[67] J. Liu, B. Chandrasekaran, W. Yu, J. Wu, D. Buntinas, S. Kini, D. K. Panda, and P. Wyckoff, "Microbenchmark performance comparison of high-speed cluster interconnects," *IEEE Micro*, vol. 24, no. 1, pp. 42–51, 2004.

[68] C. McCurdy and J. Vetter, "Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms," in *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 87–96.

[69] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (SHOC) benchmark suite," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. ACM, 2010, pp. 63–74.

[70] K. Spafford, J. S. Meredith, and J. S. Vetter, "Quantifying NUMA and contention effects in multi-GPU systems," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*. ACM, 2011, p. 11.

[71] R. Landaverde, T. Zhang, A. K. Coskun, and M. Herbordt, "An investigation of unified memory access performance in CUDA," in *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*. IEEE, 2014, pp. 1–6.

[72] W. Li, G. Jin, X. Cui, and S. See, "An evaluation of unified memory technology on Nvidia GPUs," in *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*. IEEE, 2015, pp. 1092–1098.

[73] T. Ben-Nun, "MGBench: Multi-GPU computing benchmark suite," 2016 (Accessed March 14, 2018). [Online]. Available: https://github.com/tbennun/mgbench

[74] T. Ben-Nun, M. Sutton, S. Pai, and K. Pingali, "Groute: An asynchronous multi-GPU programming model for irregular computations," in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2017, pp. 235–248.

[75] woodun, "9_Microbenchmarks," https://github.com/woodun/9_Microbenchmarks/, 2018.

[76] H. Chen, W. Chen, J. Huang, B. Robert, and H. Kuhn, "MPIPP: an automatic profile-guided parallel process placement toolset for SMP clusters and multiclusters," in *Proceedings of the 20th annual international conference on Supercomputing*. ACM, 2006, pp. 353–360.

[77] G. Mercier and J. Clet-Ortega, "Towards an efficient process placement policy for MPI applications in multicore environments," in *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*. Springer, 2009, pp. 104–115.

[78] T. Ben-Nun, E. Levy, A. Barak, and E. Rubin, "Memory access patterns: the missing piece of the multi-GPU puzzle," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2015, p. 19.

[79] D. Beckingsale, "Umpire release/0.1.3," https://github.com/LLNL/Umpire, 2018.

APPENDIX A: FULL TOPOLOGIES

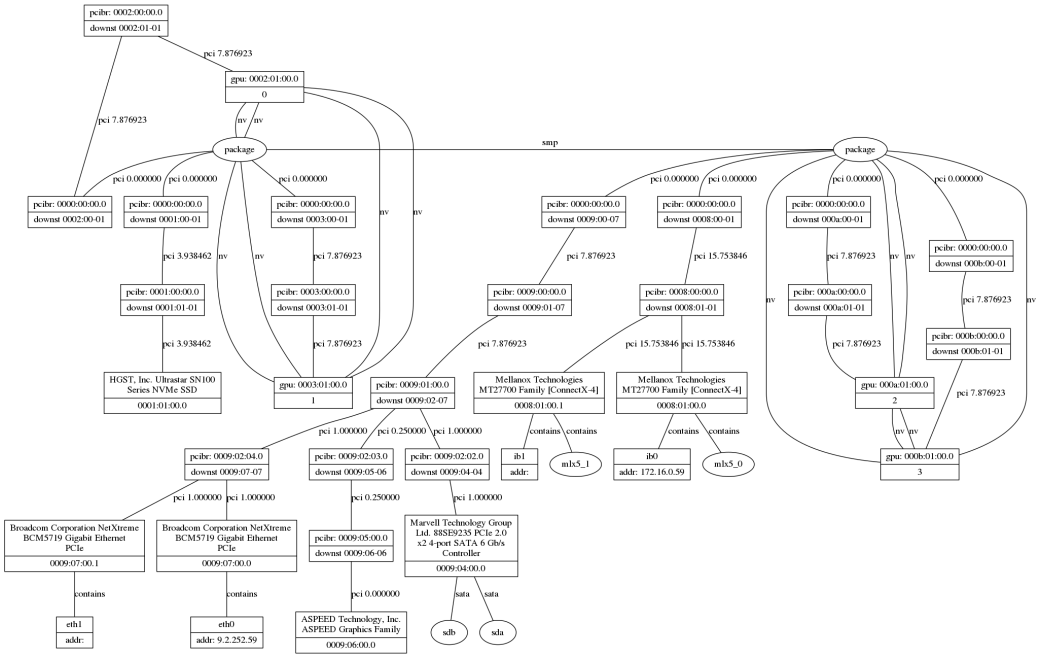


Figure A.1: S822LC discovered topology.

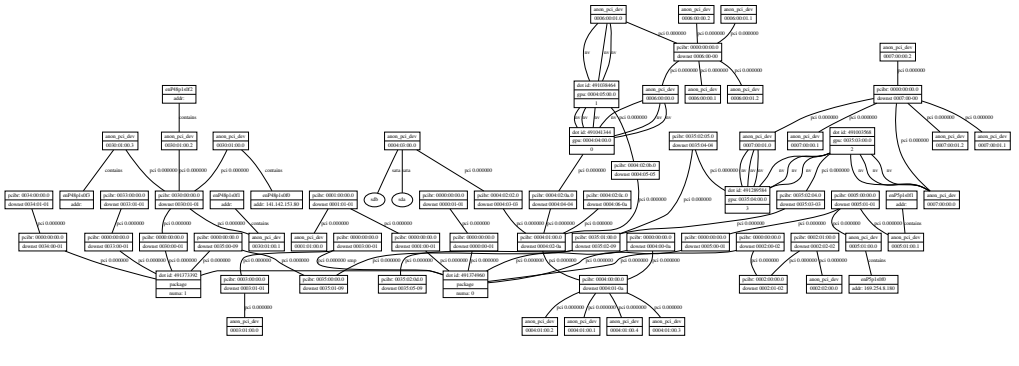


Figure A.2: AC922 discovered topology.

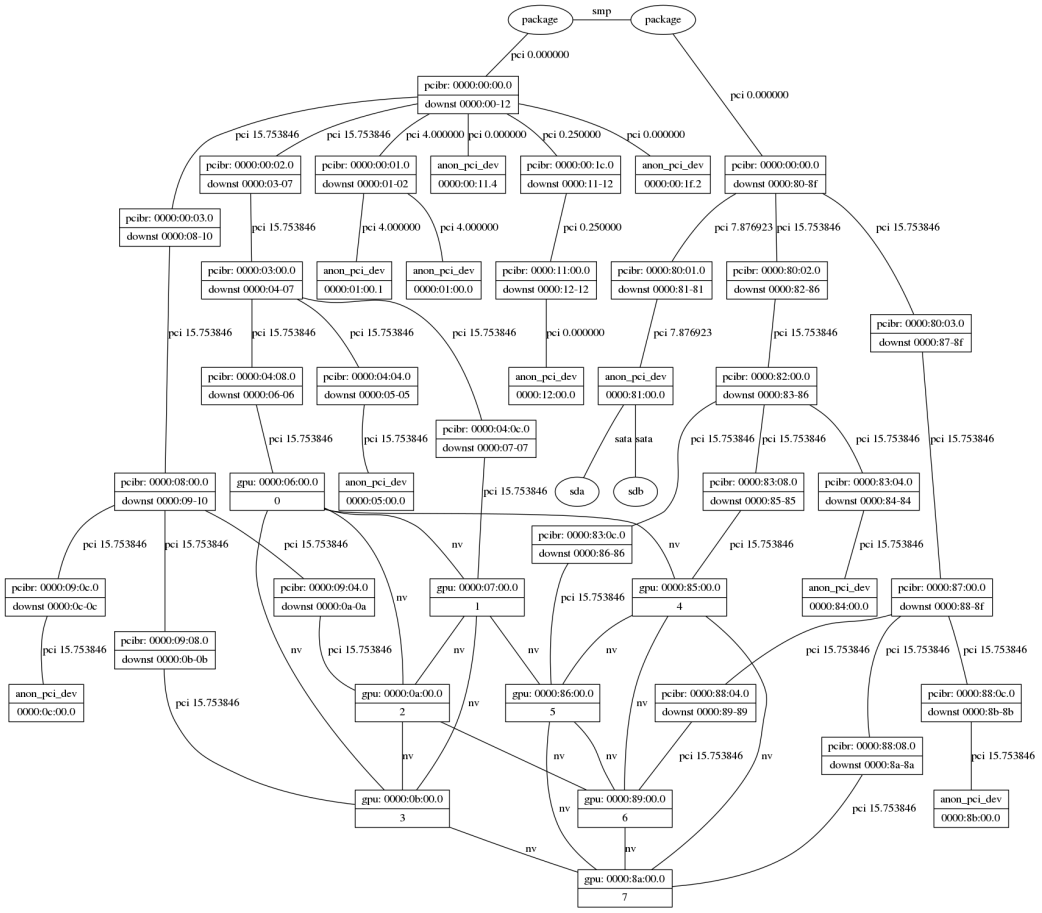


Figure A.3: DGX-1 discovered topology.