# NUMA Data-Access Bandwidth Characterization and Modeling

Ryan Karl Braithwaite

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science and Applications

Wu-chun Feng, Chair

Calvin J. Ribbens

Patrick S. McCormick

23 January 2012

Blacksburg, Virginia

Keywords: Benchmarking, NUMA, Performance Modeling

# NUMA Data-Access Bandwidth Characterization and Modeling

Ryan Karl Braithwaite

Clusters of seemingly homogeneous compute nodes are increasingly heterogeneous within each node due to replication and distribution of node-level subsystems. This intra-node heterogeneity can adversely affect program execution performance by inflicting additional data-access performance penalties when accessing non-local data. In many modern NUMA architectures, both memory and I/O controllers are distributed within a node and CPU cores are logically divided into “local” and “remote” data-accesses within the system. In this thesis a method for analyzing main memory and PCIe data-access characteristics of modern AMD and Intel NUMA architectures is presented. Also presented here is the synthesis of data-access performance models designed to quantify the effects of these architectural characteristics on data-access bandwidth. Such performance models provide an analytical tool for determining the performance impact of remote data-accesses for a program or access pattern running in a given system. Data-access performance models also provide a means for comparing the data-access bandwidth and attributes of NUMA architectures, for improving application performance when running on these architectures, and for improving process/thread mapping onto CPU cores in these architectures. Preliminary examples of how programs respond to these data-access bandwidth characteristics are also presented as motivation for future work.

GRANT INFORMATION

I am grateful to the NSF Center for High Performance Reconfigurable Computing (CHREC) and Los Alamos National Laboratory for their support through NSF I/UCRC Grant IIP-0804155.

# Acknowledgments

I am grateful for the encouragement, support, understanding, and patience of my wife, Kylie. She is as excellent a companion as one could hope for, and her assistance with all phases of my degree has been invaluable.

To my parents I offer thanks for all that I am and for the education that they provided for me.

I also appreciate the infrastructure support and technical feedback from Marcus Epperson, Jeffry Ogden at Sandia National Labs, and Scott Pakin, Marcus Daniels, and Kei Davis at Los Alamos National Laboratory. I also appreciate Tom Scogland and other members of the Synergy Laboratory for their technical support and feedback during my time at Virginia Tech.

Finally, I sincerely appreciate the time and effort put forth by the members of my committee to help me through each step of the process toward the completion of my thesis.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Motivation

Data-access latency and bandwidth performance in modern Non-Uniform Memory Access (NUMA) architectures can significantly affect program performance when non-local resources are accessed frequently. For applications that are sensitive to data-access performance, it may be critical for a system designer or application developer to understand the architectural layout of the system in which the program is executed so as to avoid remote data accesses as much as possible. If a developer is given the layout of the memory and I/O subsystems and the data-access characteristics for a system, he or she may make informed decisions regarding process and memory placement within a program. This improves memory-access performance and, consequently, overall application performance. Likewise, if a system designer is given the data-access characteristics of a given system,

he or she may configure a system's components such that data-access performance is maximized for that particular system. Furthermore, a run-time scheduler may make use of data-access characteristics in scheduling tasks within the system, potentially improving both individual application performance as well as overall system efficiency.

We propose the use of data-access performance models based on empirical micro-benchmark data to provide data-access *bandwidth* information in a concise, usable form. The requisite benchmark data comes from characterization and analysis of NUMA systems using the Cbench testing framework's node-level NUMA data-access tests. These data-access performance models may then be used to analyze and compare different architectures and systems, to improve application memory behavior, and to assist run-time schedulers with running tasks such that data-access performance is maximized.

Characterizing and analyzing modern NUMA systems requires gathering information about data-access characteristics for critical subsystems within a server. These subsystems include memory subsystems consisting of memory dual in-line memory modules (DIMMs), memory controllers, and CPU interconnection buses, and I/O subsystems consisting of I/O controllers, system interconnects such as PCIe (Peripheral Component Interconnect Express), and various I/O devices such as graphics processing units (GPUs), network interface cards (NICs), and storage devices. The specific subsystem components that are primarily responsible for data-access heterogeneity typically include the location of memory banks within the system, attributes of the interconnection bus links between CPU cores and subsystems, and the location of I/O controllers in the system.

In *uniform memory access* (UMA) architectures of the past, all memory accesses in a system were handled by a single memory controller and every CPU core in the system experienced uniform memory bandwidth and latency. Likewise, with a single I/O controller, accesses to the PCI interconnect were uniform for all CPU cores. Modern NUMA architectures, however, contain multiple memory controllers and multiple banks of memory DIMMs. Each memory bank is connected directly to its own memory controller, relieving some of the memory-access bottleneck that is seen in UMA designs. The drawback of distributing memory to increase overall memory bandwidth is the introduction of non-local (or "remote") memory accesses. A processor in a NUMA system may request data that is located in its locally-connected memory bank, or it may request data that is located in a remote memory bank. Remote accesses require transferring data between memory controllers over the CPU interconnect bus, a process that usually decreases bandwidth and increases latency compared to local-memory accesses.

Similar to memory controllers in NUMA systems, I/O controllers are now replicated to provide higher overall bandwidth between CPU cores and I/O devices at the expense of introducing local and remote I/O accesses. The result of the distribution of resources and the resultant local/non-local data accesses in NUMA architectures is significant variation in the latency and bandwidth of data-accesses depending on system implementation-specific design parameters. Ultimately, this means that application performance may also vary significantly depending on memory and process mapping within the machine and the associated data-movement performance. In NUMA systems, intelligent mapping of processes onto CPU cores while considering data location and data-access performance can mitigate this problem and improve program performance. Reducing remote data

movement is a key to application execution efficiency in NUMA systems.

Data-movement operations may be divided into memory accesses and I/O accesses. I/O accesses usually involve memory accesses as part of the I/O data-movement operation (e.g., using DMA (Direct Memory Access) when transferring data to or from a CPU core and a device), but it is useful to separate the I/O access characteristics from the memory-access characteristics to give an accurate view of the system. A memory access may include any combination of accesses to memory system components in order to access the desired data, e.g., L1, L2, or L3 cache, TLB (Translation Lookaside Buffer), or main memory controller, all of which occur using direct memory channels or the CPU interconnect bus. An I/O access may consist of a CPU core accessing an I/O controller using the CPU interconnect bus, DMA communication between the I/O controller and the memory controller over the system bus and the CPU interconnect bus, or an access from the I/O controller to the I/O device over the system bus. In either case, when a data access is remote, the latency increases and bandwidth usually decreases for that data transfer operation (relative to a local data transfer).

Data-access performance variation in a NUMA system is the result of the bandwidth and latency effects for accessing data that are either *local* or *remote* to the processor requesting the data. A processor achieves maximum data-access performance when accessing data resources that are local to it, i.e., those resources that are lowest latency and highest bandwidth for data transfers. Expressing how strongly local resources are preferred is the main goal of the data-access performance models presented in this thesis. Information about system attributes may be condensed into a straightforward performance model by utilizing micro-benchmarks to evaluate data-access performance and

normalizing the benchmark results against the technology specification's theoretical maximum.

This data-access performance variation is a function of where the CPU core is located in relation to the data being accessed and the attributes of the bus links connecting the cores and the data. CPU interconnect buses are becoming heterogeneous in their own right as engineers continue to scale the number of CPU cores in a system and further stretch the already strained interconnect bus resources. As the number of cores, sockets, and memory controllers in a system increases, system designers are splitting bus lanes into variable-width links to provide increased subsystem connectedness at the expense of uniformity.

The exact design of a NUMA system depends on choices made by system designers and is therefore implementation-specific, even for the same technology specification. Comparing servers based on these architectures is an ever more daunting task, and predicting performance of a data-bandwidth sensitive program running in a modern NUMA system is impractical without empirical characterization and analysis of the various levels of data-access performance that a program may incur during execution in the system.

The purpose of this thesis is to present the characterization and modelling of data-access bandwidth in modern AMD and Intel symmetric multi-processing (SMP) NUMA server architectures. Micro-benchmarks are used to characterize data-access performance and evaluate the heterogeneity presented by the design of both Intel and AMD NUMA architectures. Empirical tests are quick and effective at providing a real-world view of an architecture. Using precise micro-benchmarks within the Cbench framework to analyze a few key metrics for data-access performance allows for

the design attributes of the architecture to be extrapolated and how these attributes affect program performance to be inferred. This knowledge, expressed in data-access performance models, provides a generalized interface for comparing data-access attributes in NUMA systems. This data may then be used to improve application behavior in NUMA systems, to improve configurations of such systems, and may eventually be utilized by run-time schedulers to improve program execution efficiency.

## 1.2 Related Work

The heterogeneity of data-access performance in NUMA machines may be handled at various layers within a computer system. We present the related work done in the system, run-time, and application layers as these are the layers treated in this thesis. Investigating data-access performance is interesting from a system configuration and performance perspective (system-layer), but to actually impact program execution efficiency it is necessary to either manage memory explicitly within programs (application-layer) or to have an intelligent run-time scheduler to automatically manage process-data mappings (run-time-layer).

#### Application Layer Related Work

Managing memory under the asymmetric memory-access models present in NUMA symmetric multiprocessor (SMP) architectures has been an issue for decades [12][11]. Many developers and researchers implement performance models, libraries, and tools for assisting programmers with

data management in NUMA architectures. This thesis builds upon these projects by analyzing the data-access bandwidth characteristics that were not prevalent until recent multi- and many-core architecture designs.

Supercomputer applications have been shown to largely exhibit latency sensitivity [30], although new areas of research using techniques such as graph traversal, for example, are more sensitive to data-access bandwidth than latency. This is also the primary motivation behind the recently released Graph 500 benchmark [8]. Our justification for considering only data-access bandwidth in this thesis is two-fold: 1) it is data-access bandwidth that varies most significantly in modern server architectures and 2) historically, NUMA research has focused on data-access latency since this metric was the primary factor affecting program performance. Furthermore, this emerging class of data-bandwidth sensitive applications is particularly susceptible to the NUMA effects described in this thesis and we feel it necessary to present data-access bandwidth performance models to assist with this issue.

Application-layer NUMA mitigation is predicated on the programmer having enough information about a system's NUMA characteristics to make wise programming decisions. Provided that a programmer gathers sufficient information about the memory subsystem layout and characteristics using current tools and libraries, numerous performance models may be used by the programmer to determine the expected performance of a program for a given architecture [20][17]. Many of these performance models rely on having values for memory-access latency and bandwidth to estimate the performance on a given architecture. The models are therefore sensitive to the irregular memory accesses in NUMA systems and may not have considered such irregularities in

their models. For example, the PRAM model [20] expects that any processor is able to access any data in global memory in unit time, something that is not true for all processors in a NUMA system. Fortunately, the PRAM model has been extended [19] to handle any NUMA system by accounting for a range of possible memory-access latencies in the system. Similarly, the LogP model [17] expects a latency component to describe the access latency between memory nodes and has also been extended to NUMA architectures [15]. The $\kappa$NUMA [35] project strives to analyze algorithms when running on "clusters" of SMP cores. This model also requires latency values for accesses between memory nodes and within memory nodes. The work presented in this thesis is more focused on modeling program performance directly according to system characteristics. The characterization work presented in this thesis uncovers architectural details that are either assumed by the models mentioned above or are captured in a latency or bandwidth term that must be supplied by the programmer.

**Run-time Layer Related Work**

Run-time scheduling and dynamic process mapping are increasingly popular methods of managing NUMA data-access performance issues. The Minas [33][34] framework uses its `numarch` module to gather system information and then provides a library for allocating data that will be automatically managed by the Minas framework. StarPU [7] is a framework for run-time scheduling of heterogeneous tasks. The StarPU run-time discovers the latency and bandwidth between memory nodes both experimentally and from online regression analysis. However, this information is only used by the StarPU run-time for running StarPU codelets and may not be useful for

general-purpose programming. ForestGOMP [14] is a project that extends the GNU OpenMP thread scheduler to utilize a NUMA-aware memory-management system. This allows for fine-grained, NUMA-aware thread scheduling within the pragma-directed code of OpenMP to avoid remote memory accesses when possible. The performance models presented in this thesis may be of use to such a run-time scheduler, though currently the ForestGOMP system relies on its own NUMA architecture discovery mechanisms.

**System Layer Related Work**

Most modern operating systems incorporate some form of NUMA support into their schedulers. The default memory policy for NUMA in Linux is to allocate memory on the memory node that is local to the core where the process is currently running [22]. This works well until the process is rescheduled to run on a core that is not local to the original memory node, potentially degrading performance significantly if the data now requires a remote memory-access. Though some Linux kernels feature page migration [21][23], short-running tasks may not be able to amortize the overhead of page migration operations. This issue led to the development of tools and libraries, such as `libnuma` [22], which provide the programmer with the ability to directly manage CPU scheduler and memory policies in the program code or at run time.

The issue with requiring programmers to directly use a library such as `libnuma` is that the programmer must understand the system layout to properly manage the scheduler CPU and memory policies. In addition, `libnuma` relies on the BIOS ACPI table to determine NUMA distances and

this BIOS information may be incorrect or oversimplified, leaving the programmer to dig through system files in order to attempt to understand the system architecture. The `hwloc` [13] project is designed to interrogate the Linux kernel filesystems in order to provide the programmer with a complete view of the system without completely relying on ACPI information; instead, `hwloc` parses and analyzes the `/proc` and `/sys` filesystems presented by the Linux kernel. This approach provides a more detailed view of the system, but it relies on data provided by the operating system which may or may not reflect the influence of architectural characteristics on program execution.

The work presented in this thesis shares some benchmarking methodology with [6], in which the authors explore the characteristics of cache-coherent NUMA (ccNUMA) systems using the `lmbench` [29] and `STREAM` [28] [27] benchmarks to determine the characteristics of the memory system. The authors then use Basic Linear Algebra Subprograms (BLAS) to validate their benchmark findings and attempt to determine the probability of remote memory accesses for processes scheduled in a system. Our work presents a more detailed characterization of the NUMA attributes in a system as well as new and novel data-access performance models.

## 1.3 Contributions

The goal of this thesis is to present methods for characterizing and modelling data-access bandwidth in NUMA architectures. Future projects may leverage this method of analysis and utilize the performance models to improve application performance in NUMA systems. Below is a descrip-

tion of each of the major areas of contribution for this thesis.

**NUMA System Characterization:** We present methods for data-access bandwidth characterization in modern NUMA architectures using open-source micro-benchmarks. This characterization thoroughly analyzes CPU interconnect links and PCIe interconnect lanes to describe the bandwidth capabilities of a NUMA system.

**Data-Access Performance Modelling:** The characterization results are useful in and of themselves, but to assist with utilizing the benchmark results we present performance models for data-access bandwidth in NUMA systems. These performance models are normalized against the relevant technology specifications so that absolute performance may be compared within a machine and between machines.

**Cbench data-access bandwidth benchmarking tool:** To assist with the analysis of NUMA architectures and with the generation of performance models based on benchmark data, we present our addition of the NUMA system characterization tests to the Cbench cluster testing framework. This provides a straightforward, deterministic tool for analyzing and comparing data-access bandwidth across thousands of nodes in a cluster or single stand-alone nodes, as necessary.

The purpose of presenting this characterization, analysis, and performance modeling of modern NUMA architectures is to provide an empirical and theoretical foundation on which system designers, program developers, and system software engineers may build to more efficiently utilize

the increasingly heterogeneous multi-core servers in use today. We provide a useful and scalable tool for testing and analyzing NUMA systems, as well as generalized performance models for using data-access bandwidth in run-time systems or application development.

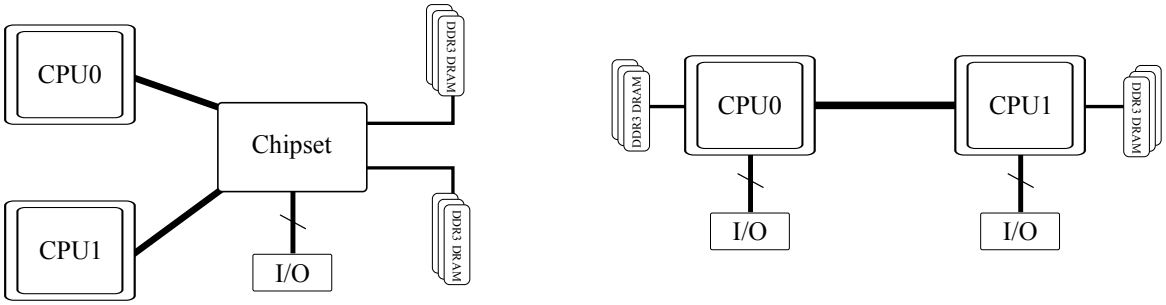
## 1.4 Document Overview

The rest of this thesis is organized as follows. Chapter 2 provides background information for the NUMA architectures studied in this thesis, as well as a survey of previous research in the analysis and efficient use of NUMA architectures. Chapter 3 is a presentation of the characterization and analysis of two-socket, or 2P, AMD and Intel NUMA architectures memory bandwidth and PCIe bandwidth micro-benchmarks. Chapter 4 discusses the data-access performance models derived from the benchmark data. Chapter 5 details the integration of the data-access characterization methods presented in this thesis into the Cbench testing framework. Chapter 6 presents results of macro-benchmarks showing sensitivity to data-access bandwidth. Conclusions and future work are presented in Chapter 7.

# Chapter 2

# NUMA Architectures

In this chapter we present an overview of modern NUMA architectures as well as details about current AMD and Intel NUMA system designs. The material presented in this chapter shows that remote data-access performance is the product of controller overhead, bus link characteristics, and end-device characteristics that affect available data-transfer bandwidth. In addition to the many features shared by modern Intel and AMD architectures, we describe the differences between their designs that lead to significantly varying NUMA data-access performance, which we present in Chapter 3.




(a) UMA System Block Diagram

(b) NUMA System Block Diagram

Figure 2.1: Uniform vs. Non-Uniform Memory Access Architectures

## 2.1 NUMA Overview

NUMA architectures are the de facto standard for servers today, supplanting UMA architectures of the past. The block diagrams shown in Figure 2.1 illustrate the architectural shift from UMA to NUMA. The wholesale shift toward the replication and distribution of system resources discussed in Chapter 1 is driven by the trend toward multi- and many-core processor designs and the resultant need to increase memory and I/O bandwidth to satisfy more cores simultaneously accessing data. In NUMA systems, resources that are local to a processor exhibit the uniform access bandwidth and latency to which programs that are designed for UMA systems are accustomed. However, resources that are remote to a processor may be subject to significantly worse bandwidth and latency data-transfer characteristics. NUMA systems designed by Intel and AMD are similar in many ways, but their system-level interconnect networks are substantially different from each other. These differences result in substantially different data-access performance for each architecture. In the following sections we present architectural details that are necessary for understanding

the *how* and *why* of NUMA data-access characteristics.

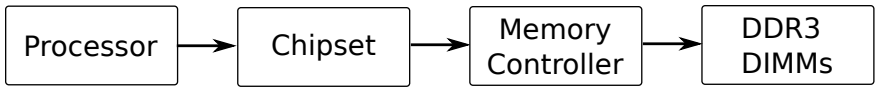
## 2.2 NUMA Data Transfers

For a processor to access data that is located in a memory bank or PCIe peripheral that is not local to that processor, a remote data-access must be made. The differing latency and bandwidth characteristics of local and remote data accesses is the cause of data-transfer asymmetry that may lead to inefficient scenarios for program execution. We characterize each type of data transfer in a system in order to fully understand the potential data-transfer pitfalls of a given NUMA architecture. In this section we look at the possible memory and PCIe data-transfer scenarios found in modern NUMA systems.

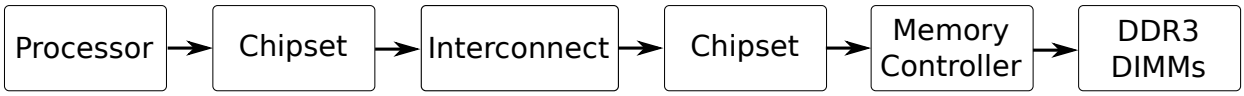
### 2.2.1 Memory Transfers

Memory containing data requested by a processor in a NUMA system is either local or remote to that processor. Figure 2.2a shows the resources involved in a local-memory data-access in a NUMA system. A local memory-access is subject to the overhead and constraints of the chipset, memory controller, and the memory technology (e.g., DDR3) used in that system. Figure 2.2b, on the other hand, shows the resources involved in a first-level *remote*-memory data-access. In the remote-access case, the interconnect link and an additional chipset are present, increasing overhead and decreasing data-access bandwidth according to the constraints of those intermediary resources.

Additional levels of remote data-access may be possible, depending on the architecture, with each level adding another "$Interconnect \rightarrow Chipset$" segment to the resource list and, subsequently, increasing overhead and potentially decreasing bandwidth for each type of data-access.




(a) Local Memory Access



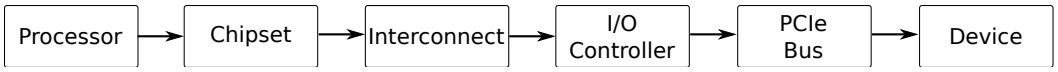

(b) First-Level Remote Memory Access

Figure 2.2: Memory-Access Processes

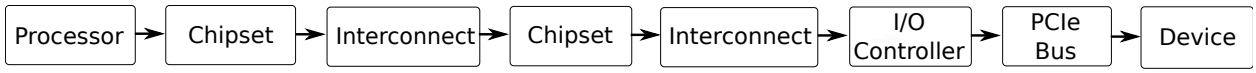
### 2.2.2 I/O Transfers

In a similar manner to memory data-transfers, processors conducting I/O data-transfers accessing PCIe through a local I/O controller are more efficient than transfers using a remote I/O controller. However, even a local I/O controller is accessed by using at least one interconnect link, a chipset, an I/O controller, and the PCIe interconnect itself, as shown in Figure 2.3a.

A remote device requires access to another node's chipset and another segment of the interconnect, as shown in Figure 2.3b. Traversing these additional resources during a data transfer will result in increased overhead and decreased bandwidth.




(a) Local PCIe Access




(b) First-Level Remote PCIe Access

Figure 2.3: PCIe-Access Processes

## 2.3 AMD & Intel 2P Architecture Details

In the following sections we present relevant details of modern NUMA architectures designed by AMD and Intel.

### 2.3.1 Intel Nehalem + QuickPath Interconnect

The Intel Nehalem 2P (two-socket) architecture shown in Figure 2.4 is a straightforward NUMA design. Each socket contains one CPU package, and each CPU package is a single NUMA memory node. The two sockets are connected to each other and to their I/O controllers by full-width, 20-lane QuickPath Interconnect (QPI) links.

QPI links are capable of bandwidth between $4.8\ {}^{GB}/_{s}$ and $25.6\ {}^{GB}/_{s}$, depending on bus clock frequency and the number of lanes (each lane corresponding to one bit that can be sent in parallel in a single transfer) used in each link[1]. A QPI link running at its current full clock speed of 3.2 GHz with all 20 lanes provides up to 25.6 GB/s of bandwidth, more than the DDR3 specification

[1]QPI links may degrade from 20 data lanes to 10 or 5 lanes depending on link-level events

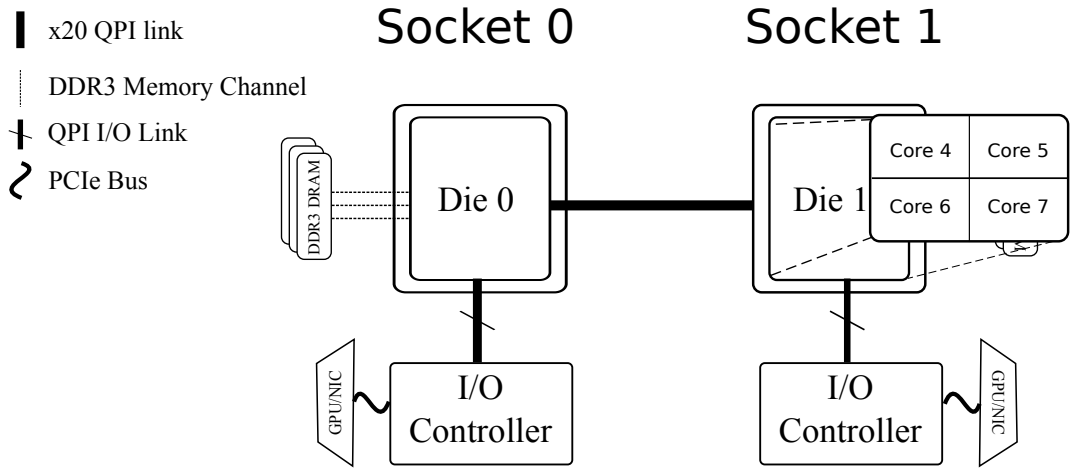



Figure 2.4: Intel Nehalem 2P Architecture

of 17.066 GB/s. In such a case, only QPI and chipset overheads are to blame for bandwidth degradation when accessing remote memory.

The possible memory-access bandwidth classes for this architecture are

1. **Local Memory-Access**: DDR3, $6.4\ {}^{GB}/_{s}$ to $17.066\ {}^{GB}/_{s}$ depending on memory frequency

2. **Remote Memory-Access**: One QPI link at $4.8\ {}^{GB}/_{s}$ to $25.6\ {}^{GB}/_{s}$ + DDR3

Likewise, the possible PCIe-access bandwidth classes for this architecture are

1. **Local PCIe-Access**: One QPI link ($4.8$-$25.6\ {}^{GB}/_{s}$) + PCIe interconnect at $16\ {}^{GB}/_{s}$ bidirectional

2. **Remote PCIe-Access**: Two QPI links ($4.8$-$25.6{}^{GB}/_{s}$) + PCIe interconnect ($16\ {}^{GB}/_{s}$)

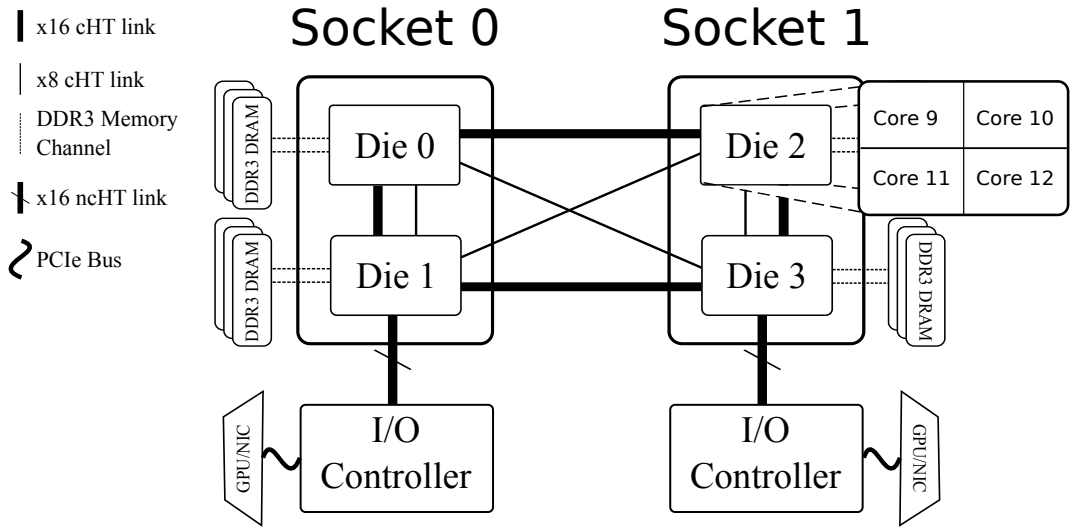



Figure 2.5: AMD Magny-Cours 2P Architecture

### 2.3.2 AMD Magny-Cours + HyperTransport 3.0

The AMD Magny-Cours 2P (two-socket) architecture shown in Figure 2.5 is somewhat more complex than the Intel architecture shown in Figure 2.4. The complexity has two sources: the CPU package, consisting of the CPU dies, caches, and wires in a socket; and the HyperTransport (HT) interconnect links between CPUs and components outside their socket packages. As a result of this increased complexity, the amount of data-access heterogeneity is much greater in the AMD system despite the fact that it is also a two-socket design.

We consider first the G34 CPU package present in Magny-Cours architectures. Figure 2.6 is adapted from the *AMD Family 10h Processor BIOS and Kernel Developer's Guide* (BKDG) [1] and shows the detailed block diagram of a CPU package (one package per socket). This figure is in reference to the G34 processor package, so the details presented in this section apply to the G34 design specification. We see in Figure 2.6 a few key features that set the AMD architecture apart

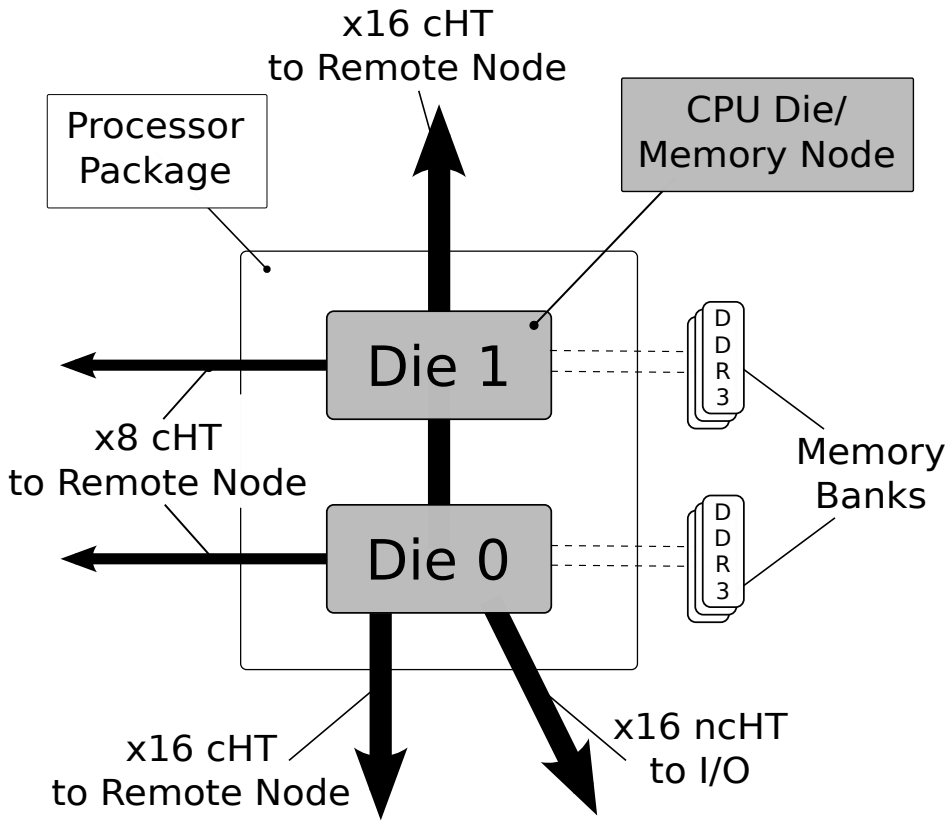



Figure 2.6: AMD G34 Processor Package Detail [1]

from the Intel architecture presented in 2.3.1. First, there are two memory nodes contained in one processor package (socket). Second, the two memory nodes within a CPU package are connected by HyperTransport (HT) links even though they are within the same processor package and share an L3 cache (not shown in Figure 2.6). Third, only one of the two nodes in a package is connected with a non-coherent HyperTransport (ncHT) I/O link. Finally, the width of the coherent HyperTransport (cHT) links between memory nodes are not uniform; some are eight lanes (x8) and some are sixteen lanes (x16). This has strong implications for the data-access bandwidth that we report in Chapter 3.

The HT links connecting memory nodes in the AMD system are of variable width in order to accommodate an all-to-all network connecting memory nodes. This design improves data-access latency between memory nodes, as each node can directly access every other node. However, the use of variable-width HT links causes an additional level of data-access bandwidth asymmetry beyond that which we see in basic NUMA architectures. A processor may request data from local memory, remote memory within the same processor package, or remote memory outside of the processor package, each of which may be connected by a different link type or width. When accessing I/O, a processor may request data from a local or remote I/O controller. As mentioned previously, only one node within each CPU package is connected to the I/O HT link, meaning that each socket has one node with a direct connection to an I/O controller and one node that must traverse an extra HT link to access its nearest I/O controller.

Data-access bandwidth in the AMD architecture is a function of all of the components along the data transfer's path. However, it is the HT links that are the greatest source of heterogeneity in this architecture. The bandwidth of a HT link is a function of the clock rate of the link, the number of lanes in the link, and the data rate per clock cycle (single or double). It is the number of lanes per link that changes depending on the path of the data transfer, and therefore it is the width of the links that is the cause of increased data-access heterogeneity in the AMD architecture. Naturally, there are many more possibilities for memory-access bandwidth in the AMD architecture than the Intel architecture due to the possible link configurations in the AMD system.

As shown in Figure 2.5, according to the specification for Socket G34 architectures [16] there are x8 and x16 links between memory nodes within a socket and either a x8 or a x16 link connecting

the other nodes in a 2P system. However, according to the AMD BKDG [1], the x8 connection between dies within a socket is not functional. Furthermore, it is left up to the system designer whether x8 or x16 links are used to connect nodes. In any case, the following classes of memory-access bandwidth are expected in the AMD architecture:

1. **Local Memory-Access**: DDR3, $6.4\ {}^{GB}/_{s}$ to $17.066\ {}^{GB}/_{s}$, depending on the memory frequency

2. **Remote Memory-Access #1**: x16 HT link at $1.6\ {}^{GB}/_{s}$ to $25.6\ {}^{GB}/_{s}$, depending on HT clock frequency + DDR3

3. **Remote Memory-Access #2**: x8 HT link at $0.8\ {}^{GB}/_{s}$ to $12.8\ {}^{GB}/_{s}$, depending on HT clock frequency + DDR3

In the next chapter we show the empirical data-access bandwidth results for systems based on these Intel and AMD architectures. As predicted in this chapter, distinct classes of data-access bandwidth are seen for both architectures.

# Chapter 3

# Data-Access Performance of Modern NUMA Architectures

We present in this chapter our data-access bandwidth characterization methodology, details of our data-access bandwidth benchmark tests, and analysis of data-access bandwidth benchmark results from AMD and Intel 2P (two-socket) NUMA systems.

## 3.1 Methodology for Data-Access Bandwidth Characterization

Data accesses in modern computer systems are complex operations with many levels of hierarchy and redirection. Rather than individually benchmark every subsystem involved in a memory or PCIe access, we present a method for using benchmark applications to characterize a system as a

user's application sees it. As shown in Chapter 2, memory and PCIe data-access performance is related but differs in the resources used in each data-transfer process. Since each type of data-access uses a different set of resources during the data-transfer process, we characterize the bandwidth of memory and PCIe data accesses separately.

As discussed in Chapter 1, latency of data accesses is critical to the efficiency of many applications. Data-access bandwidth, which is the focus of this thesis, is critical only to certain programs with attributes such as high Last-Level of Cache (LLC) miss rates or poor data reuse. A program that is compute-bound will not see much effect from changes to the data-bandwidth characteristics of a machine, whereas a memory-bound application almost certainly will. It is the bandwidth of data-accesses in modern server architectures that is most quickly becoming heterogeneous, and therefore our analysis focuses on the bandwidth attribute of data-accesses in these systems. Emerging data-intensive applications [5][8][10] are sensitive to memory-bandwidth performance and are therefore susceptible to the remote-access data-transfer performance loss of these NUMA systems, further justifying our bandwidth-focused approach.

The bandwidth of a data-access operation is the number of bytes of data that are transferred between a device's memory and the requesting processor in a given amount of time. This metric varies not only based on the distance (in terms of bus segments, intermediary controllers, etc.) between device and processor, but on interconnection link characteristics such as the number of lanes, clock frequency, etc. of the bus connecting the sockets, as discussed in Chapter 2.

There are many factors to consider when attempting to characterize a computer system. To focus

on NUMA data-access bandwidth, we consider only those factors that directly affect system-level data-access bandwidth. The key elements of this analysis are

1. Discovering which resources are local and which are remote to a given processor core
2. Measuring the difference that remote data accesses make on data-transfer performance

We consider these elements as we characterize data-access bandwidth in the following sections of this chapter.

## 3.2 Characterization Details

Our method for characterizing data-access bandwidth is to run benchmark programs using the various data-access scenarios for a given system and partition the results into "bandwidth classes", or equivalence classes of data-access bandwidth, based on architectural information and empirical results. As the benchmarks used for measuring memory- and PCIe-access bandwidth differ, we consider each type of data-access characterization separately.

### 3.2.1 Memory-Access Bandwidth Characterization

#### Memory Bandwidth Benchmark Details

We use the `STREAM` [27][28] benchmark for determining memory-access bandwidth between CPU cores and memory nodes in a system. The `STREAM` benchmark executes four types of memory-

access operations on a large data array. The four components of the `STREAM` benchmark are

**`Add`** measures transfer rates in the absence of arithmetic

**`Copy`** adds a simple arithmetic operation

**`Scale`** adds a third operand to allow multiple load/store ports on vector machines to be tested

**`Triad`** allows chained/overlapped/fused multiply/add operations

Due to `Triad`'s similarity to operations found in many scientific applications (`Triad` is similar to the level 1 BLAS[1] SAXPY[2] operation), we report only the `Triad` results for `STREAM`.

The memory-access behavior of a program is immensely affected by the compiler used to produce the program's executable code, and `STREAM` is a good example of the effect that a compiler may have on a program's memory-access characteristics. The memory bandwidth reported by the `STREAM` benchmark varies substantially depending on the compiler used to compile it. The effects of compiler optimizations on benchmark performance are considered and compared in the course of our characterization effort by running versions of the `STREAM` benchmark that are compiled using the GNU, Open64, and Intel compilers. Testing multiple versions of the `STREAM` benchmark provides multiple bandwidth values for a given system which, in addition to helping to determine the best memory bandwidth for a given access scenario, may be useful for defining a range of expected performance in a system.

Our memory-access bandwidth characterization method is to gather the memory-bandwidth results for memory-node-to-memory-node memory-transfers, both single-threaded and multi-threaded,

[1] Basic Linear Algebra Subprograms

[2] Single-precision Alpha-X Plus Y

for all memory node combinations in a system. To test single-threaded memory-access bandwidth, we ran each of the GNU, Intel, and Open64 `STREAM` binaries through all combinations of memory-node-to-memory-node mappings in the system. The results reported by the `STREAM` benchmark vary substantially depending on which version of `STREAM` is run on which architecture, as we will show in Section 3.3. Table 3.1 summarizes the compilers and flags used for the tests reported in this thesis.

Table 3.1: Compilers Used for Benchmarks

| COMPILER | VERSION | OPTIMIZATION FLAGS |
|---|---|---|
| GNU | 4.1.2 | -O3 -march,mtune=amdfam10,core2 |
| Open64 | 4.2.3 | -Ofast |
| Intel | 11.1 20091130 | -O3 -xsse4.2 |

**Memory Benchmark Run-time Details**

Benchmarking every data-access scenario in a NUMA system entails mapping benchmark processes and their memory in every possible configuration that a program might encounter during execution. For modern many-core systems, mapping processes to individual cores greatly expands the set of core-to-node combinations that must be tested. To test all core-to-memory-node combinations, $(\#\ of\ cores) \times (\#\ of\ nodes)$ tests must be executed. Cores that are attached to the same memory node show virtually identical memory bandwidth when accessing data on a given memory node (see Table 3.2), so core-to-node testing is not necessary.

Table 3.2: Core-to-Memory-Node Bandwidth Distribution for AMD 2-socket System

| **STREAM TRIAD ON CORES 0-3** | **MEAN BANDWIDTH, 30 RUNS PER CORE** | **STD. DEV.** | **95% CONF. INTERVAL** |
|---|---|---|---|
| **Data on Mem. Node 0** | 8127 $^{MB}/_{s}$ | 20.367 (0.25% of mean) | (8106.338;8147.072) |
| **Data on Mem. Node 1** | 5046 | 12.808 (0.25% of mean) | (5033.103;5058.719) |
| **Data on Mem. Node 2** | 3045 | 22.587 (0.75% of mean) | (3022.432;3067.606) |

Allowing the process scheduler to choose to schedule a process on any core attached to a memory node allows for a greatly reduced testing set as we characterize a system. Such a process-to-node mapping simplifies the number of tests to $(\#\ of\ nodes) \times (\#\ of\ nodes)$. We therefore run the single-threaded `STREAM` benchmark tests with the `STREAM` process pinned to a memory node rather than to a specific core and restrict its memory allocation to each memory node in the system in turn. This pinning of processes and memory is done at run-time using the `numactl` tool.

`numactl` utilizes `libnuma` [22] to set the Linux scheduler's CPU and memory policies for a program, giving the user command-line control over process and memory location during run-time. For example, to run GNU-compiled `STREAM` from memory node 1 to all four memory nodes in a system, we issue the following commands:

1. `numactl --cpunodebind=1 --membind=0 stream_gcc.exe`
2. `numactl --cpunodebind=1 --membind=1 stream_gcc.exe`

3. `numactl --cpunodebind=1 --membind=2 stream_gcc.exe`
4. `numactl --cpunodebind=1 --membind=3 stream_gcc.exe`

The `--cpunodebind=1` parameter sets the scheduler CPU policy to only allow scheduling of that process on any core attached to memory node 1. The `--membind` parameter sets the `STREAM` process memory policy to allocate memory only on the given memory node. These parameters ensure that the `STREAM` process is run only on a core attached to memory node 1 and that its memory is allocated only on the memory node given with `--membind`.

Multi-threaded, MPI-based `STREAM` is used to measure the maximum aggregate bandwidth between memory nodes in a system. We selected the MPI version of `STREAM` since it is the multi-threaded version of `STREAM` provided by the original author of the benchmark. Multi-threaded MPI-`STREAM` tests are also run using `numactl` to manage scheduler CPU and memory policies, but in this case the `numactl` command is given after the `mpirun` command. For multi-threaded tests we increase the number of threads for each run of the benchmark until the bandwidth reported by `STREAM` decreases and take the maximum bandwidth observed between the memory nodes as the final result. For example, to test the multi-threaded bandwidth between memory node 1 and memory node 0, we run:

1. `mpirun -np 1 numactl --cpunodebind=1 --membind=0 stream_gcc.exe`
2. `mpirun -np 2 numactl --cpunodebind=1 --membind=0 stream_gcc.exe`
3. `mpirun -np 3 numactl --cpunodebind=1 --membind=0 stream_gcc.exe`
4. `mpirun -np 4 numactl --cpunodebind=1 --membind=0 stream_gcc.exe`

Presuming that for this example using four threads results in lower reported bandwidth than the

previous observation, these four commands are sufficient for testing aggregate memory-access bandwidth for processes running on memory node 1 with their data on memory node 0.

### 3.2.2 PCIe-Access Bandwidth Characterization

#### PCIe Bandwidth Benchmark Details

Similar to our use of the `STREAM` benchmark for characterizing memory data-access bandwidth, we employ the user-level Scalable HeterOgeneous Computing (`SHOC`) benchmark suite [18] to characterize PCIe data-access bandwidth. PCIe data-access bandwidth is less straightforward to measure than memory data-access bandwidth bandwidth since the data-access bandwidth of a device connected by PCIe I/O interconnect links is a function of traversing additional resources when transferring data between the PCIe device and the processor. These additional resources include the PCIe interconnect links, the CPU interconnect links (i.e., HT or QPI) from CPUs to the PCIe devices, various controllers and chipsets along the CPU-to-PCIe path, and the specific PCIe device that contains the requested data. Whereas all memory controllers (which are conceptually equivalent to a PCIe "device") in a system are identical, PCIe devices may range from high-performance GPUs to simple network interface cards, with each device having its own data-transfer capabilities. We present the analysis of high-performance GPUs transferring data across the PCIe interconnect to requesting processors in a NUMA system. The CPU core-to-I/O controller relationship shown by these tests also applies to other devices transferring data using PCIe, though the reported absolute bandwidth for other PCIe devices may change.

GPU programs are typically written using the Compute Unified Device Architecture (CUDA) [26] framework or the Open Computing Language (OpenCL) [37] framework. In both frameworks, a host CPU process handles the execution of "kernels" on GPU devices, which includes transferring data between main system memory and the onboard device memory of the GPU. Below is a high-level example of launching a GPU kernel and the data-access steps involved:

1. Host process allocates buffers
2. Host process downloads buffers to GPU device (system memory $\rightarrow$ GPU device memory)
3. GPU executes kernel
4. Host process reads data back from GPU device (GPU device memory $\rightarrow$ system memory)

Analysis of data transfers for a GPU connected by PCIe involves measuring the download rate (writing to the GPU memory) and the readback rate (reading from the GPU memory) when transferring data between the host CPU core and the GPU device. To measure these rates, we utilize the `BusSpeedDownload` and `BusSpeedReadback` tests in the `SHOC` benchmark suite. These tests measure the bandwidth of the link(s) between host processor and GPU device by transferring data payloads of varying size to (download) and from (readback) the GPU device.

Systems configured with NVIDIA GPUs can run both CUDA and OpenCL code, and in some instances the PCIe bandwidth reported by the CUDA and OpenCL versions of `SHOC` vary. We execute both the CUDA and OpenCL versions of the `SHOC` PCIe bandwidth tests to compare possible performance differences between the frameworks on a given system, similar to the compiler comparison we perform with the `STREAM` benchmark.

**PCIe Benchmark Run-time Details**

We run the PCIe data-access bandwidth tests in a manner very similar to the memory-access bandwidth tests, simply replacing `STREAM` with the appropriate `SHOC` benchmark. To run the `SHOC` `BusSpeedDownload` benchmark with the CPU host process and memory pinned to memory node 0, for example, the command is

```
numactl --cpunodebind=0 --membind=0 BusSpeedDownload
```

The download and readback rates for each of the node-to-device combinations in a system are gathered and organized into bandwidth equivalence classes, similar to the results of the `STREAM` benchmarks.

## 3.3 Results of Data-Access Bandwidth Characterization

Data-access bandwidth characterization tests were performed on the two machines described in Table 3.3. The `STREAM` and `SHOC` bandwidth results are presented in Table 3.4.

The benchmark tests used to quantify bandwidth performance were run using commands described in previous sections. The `STREAM` tests were run at least five times for each node-to-node combination and the maximum observed value for each type of data-access is reported here. Results within each bandwidth class vary; the results presented here have bandwidth classes with values within ten percent of the maximum value. We select the maximum value instead of the mean value so that we may analyze the maximum bandwidth characteristics of each bandwidth class.

Table 3.3: Attributes of Test Systems

| ATTRIBUTE | AMD 2P SYSTEM | INTEL 2P SYSTEM |
|---|---|---|
| **CPU Model** | Magny-Cours 6134 | Nehalem E5645 |
| **Cores** | 16 | 12 |
| **Memory Nodes** | 4 | 2 |
| **Motherboard** | Supermicro H8DGG | Supermicro X8DTG-QF+ |
| **GPUs (#)** | Tesla C2050 (2) | Tesla M2050 (2) |
| **OS** | CentOS 5.5 | CentOS 5.6 |
| **Linux Kernel** | 2.6.18-194.17.4.el5 | 2.6.18-238.12.1.el5 |

## 3.4 Analysis of Data-Access Performance in AMD & Intel 2P NUMA Systems

The results in Table 3.4 describe well the architectural differences between the AMD and Intel system designs that were discussed in Chapter 2. The AMD system has four memory nodes and three levels of data-access bandwidth for both memory and PCIe devices, as discussed in Section 2.3.2. The Intel has two memory nodes and two levels of data-access bandwidth. We delve into the memory and PCIe data-access bandwidth performance in the following sections[3].

[3]Results for a 4P AMD system are presented in Appendix A

Table 3.4: Microbenchmark Characterization Results

Classes of data-access bandwidth

| | AMD | | | INTEL | | |
|---|---|---|---|---|---|---|
| | **STREAM SINGLE-THREADED – GB/S (% OF CLASS 0)** | | | | | |
| | **GNU** | **Intel** | **Open64** | **GNU** | **Intel** | **Open64** |
| **Class 0** | 5.2 (100%) | 7.8 (100%) | 9.3 (100%) | 9.2 (100%) | 10.5 (100%) | 9.0 (100%) |
| **Class 1** | 3.3 (63%) | 5.3 (68%) | 5.6 (60%) | 6.5 (71%) | 7.1 (68%) | 6.1 (68%) |
| **Class 2** | 2.1 (40%) | 3.1 (40%) | 3.1 (33%) | – | – | – |
| | **STREAM MULTI-THREADED – GB/S (% OF CLASS 0)** | | | | | |
| **Class 0** | 7.9 (100%) | 9.8 (100%) | 11.4 (100%) | 14.1 (100%) | 20.1 (100%) | 19.5 (100%) |
| **Class 1** | 5.0 (63%) | 10.2 (104%) | 10.9 (96%) | 14.1 (100%) | 20.1 (100%) | 19.5 (100%) |
| **Class 2** | 2.2 (28%) | 10.0 (102%) | 11.4 (100%) | – | – | – |
| | **SHOC BUSDOWNLOAD – GB/S (% OF CLASS 0)** | | | | | |
| | **CUDA** | **OpenCL** | | **CUDA** | **OpenCL** | |
| **Class 0** | 5.8 (100%) | 5.8 (100%) | | 6.1 (100%) | 6.1 (100%) | |
| **Class 1** | 5.3 (91%) | 5.3 (91%) | | 4.7 (77%) | 4.7 (77%) | |
| **Class 2** | 2.3 (40%) | 2.3 (40%) | | – | – | |
| | **SHOC BUSREADBACK – GB/S (% OF CLASS 0)** | | | | | |
| **Class 0** | 6.6 (100%) | 6.6 (100%) | | 6.1 (100%) | 6.1 (100%) | |
| **Class 1** | 5.3 (80%) | 5.3 (80%) | | 4.1 (67%) | 4.1 (67%) | |
| **Class 2** | 2.3 (35%) | 2.3 (35%) | | – | – | |

### 3.4.1 Memory Data-Access Bandwidth

The effect of the compiler/architecture relationship on STREAM results is clearly seen in the benchmark results. What is interesting to note, however, is that the relative performance between data-access bandwidth classes (e.g., comparing local-access bandwidth vs. remote-access bandwidth) is often similar regardless of the compiler used. For example, the three data-access bandwidth classes in the AMD system for single-threaded STREAM using the GNU compilers are 100%, 63%, and 40% of local data-access bandwidth. The three bandwidth classes for the same test using the top-performing Open64 compilers is 100%, 60%, and 33% of local data-access bandwidth, a relative difference of 7% or less for each class. This similarity may be attributed to the fact that no matter what compiler-level optimizations might do to improve cache utilization, once a remote memory access is made a similar data-access bandwidth penalty is experienced by all programs.

The explanation for why an executable generated by a given compiler performs better than one produced by another compiler comes down to architecture-level optimizations. The Open64 compilers use architecture-specific optimizations such as non-temporal prefetching [2], which significantly improves STREAM bandwidth in AMD systems that are tuned for this type of caching. The Intel compilers achieve the best performance on Intel architectures due to their architecture-specific optimizations. The GNU compilers generally provide a minimum-performance baseline for the STREAM benchmark, as the GNU compiler optimizations are not as well-tuned as those of Intel or Open64.

The multi-threaded STREAM results are somewhat more ambiguous. It is unclear whether it is

the fault of the compiler or the MPI library that the performance of remote-memory accesses is better than local accesses in the AMD machine for the Intel and Open64-compiled versions of `STREAM`. Further conclusions based on the data presented here are not warranted given the data, so we postpone further analysis of the multi-threaded results to our Future Work.

As mentioned in Chapter 2, it is possible to use the memory data-access bandwidth results to reverse-engineer certain aspects of a system's architecture. The ability to determine a system's design using user-level benchmarks is particularly helpful for systems such as those based on the AMD specifications discussed in Chapter 2 since the precise HyperTransport network configuration is implementation-specific.

Using the results presented in Table 3.4, we may determine the width of HT links connecting memory nodes in the AMD 2P system that we tested. A x16 HT 3.0 link operating at full clock rate may achieve 25.6 GB/s of bi-directional bandwidth. A x8 link with the same operating parameters may achieve 12.8 GB/s of bi-directional bandwidth. The results produced by even the best-performing Open64 `STREAM` benchmark are below 12.8 GB/s, even for local memory access. Therefore, we must look at other factors to determine anything useful about the HyperTransport network in this machine.

The astute reader will recognize that there are only three classes of memory data-access bandwidth even though there are four memory nodes in the system. This indicates that two memory nodes must fall within the same bandwidth class, and this is indeed the case, as shown in Figure 3.1. This figure shows data-access bandwidth from cores on all four memory nodes in the system accessing

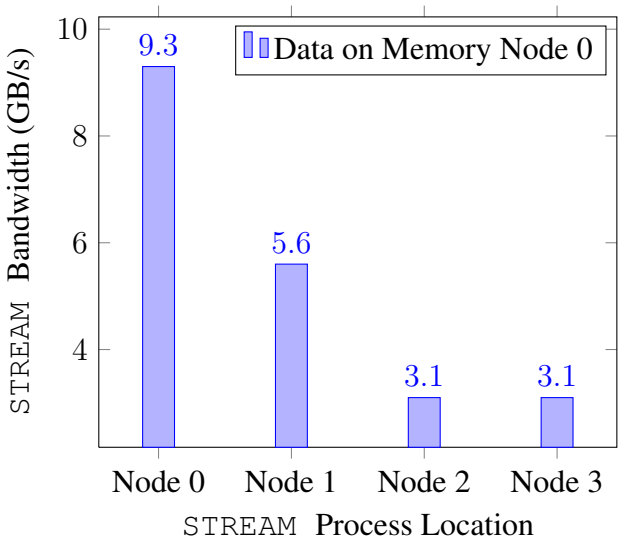



Figure 3.1: Memory Data-Access Bandwidth in AMD 2P System

data on memory node 0. This means that a STREAM process running on node 0 will access *Local* memory, running on node 1 will access *Class 1* memory, and running on nodes 2 or 3 will access *Class 2* memory. The two *Class 2* memory nodes have identical memory-access bandwidth, and this bandwidth is roughly 55% of the next bandwidth class for the system. Based on the empirical evidence, we infer that memory node 1 is connected by a x16 link and the two remote nodes (2 & 3) are connected by x8 links. This is confirmed in Figure 2.6, which shows that each node has only one x16 and one x8 link to connect to other nodes, and the AMD *BKDG*, which mentions that x16 links may be split into two x8 links. Splitting the x16 HT links into x8 links is clearly what the designers have done for this system. The reduced bandwidth of a x8 link is amortized when multiple threads are accessing memory over the same HT link, allowing bandwidth for the MPI-STREAM tests to approach the x8 link's limits. However, for the memory data-access bandwidth of single-threaded programs the implications of HT link width are clear: performance drops severely

when link width decreases.

### 3.4.2 PCIe Data-Access Bandwidth

The `SHOC` PCIe bandwidth benchmark results show the same number of data-access bandwidth classes as the `STREAM` results do, though the relative performance between bandwidth classes is somewhat different. The AMD architecture's first two PCIe data-access bandwidth classes are for the two memory nodes within the package that is local to the I/O controller. One of the nodes in the package is directly connected to the I/O controller via a x16 ncHT link and constitutes the *Local* bandwidth class. The other node in the package must traverse the x16 cHT link to reach the I/O link via the other node and constitutes the *Class 1* bandwidth class. The third class of PCIe data-access bandwidth is for the other CPU package in the system, as both nodes in the remote package must traverse a x8 HT link to reach the I/O controller. Transferring data between the host processor and the GPU is often a bottleneck when running GPGPU applications, so the PCIe data-access bandwidth classes shown for the two test systems should be taken into account when launching GPU kernels in these systems.

The question of whether to use absolute (e.g., 6.5 GB/s) or relative (e.g., 71% of local) data-access bandwidth figures depends on what the data are being used for. Architectural inferences require absolute benchmark results to approximate the actual hardware characteristics of a system. Task scheduling, on the other hand, may be more amenable to using relative benchmark results to determine where a process will experience the lowest data-access bandwidth performance relative

to other locations in the system.

To assist with analysis and comparison of these NUMA systems, we next present our data-access performance models which are built upon the benchmark data shown here.

# Chapter 4

# Data-Access Performance Models

In this chapter we present data-access performance models that are based on the system characterization data discussed in Chapter 3. We discuss our method of performance model synthesis using benchmark data and present examples and use cases for the performance models.

## 4.1 Building Data-Access Performance Models

A program running on a given CPU core in a NUMA system sees a hierarchy of data-access performance, as shown in Table 3.4 and discussed in Chapter 3. To present these classes of data-access performance in a more accessible and usable form, we develop models based on the measured data-access bandwidth of a system. We create data-access performance models using the benchmark data for each type of data access: `STREAM Triad` data for data-access memory bandwidth

and `SHOC` PCIe Bandwidth Test data for data-access PCIe bandwidth. These benchmark results provide a baseline with which the effect of data-access bandwidth on program execution performance may be approximated. It is expected that actual programs will express some fraction of the bandwidth sensitivity shown by the performance models presented in this chapter since benchmarks are designed to be more sensitive to bandwidth than applications will be. Nevertheless, the data-access performance models presented in this chapter provide a means by which data-access bandwidth characteristics may be compared between systems and by which the effect the of data-access bandwidth characteristics of a system on program's performance may be estimated.

The data-access performance models compare the measured performance of data accesses against the standards-based specifications for the appropriate data-access technology. To calculate data-access performance we scale the bandwidth of accessing data located on a device in each bandwidth class by the percentage of a program's data accesses that are made to a device belonging to that class and take the sum of the performance for all bandwidth classes:

$$\begin{aligned} \textit{Data-Access Performance} = & \left(\frac{\textit{class 0 bandwidth}}{\textit{theoretical bandwidth}}\right) \times (\textit{frac. of accesses to bw class 0}) + \\ & \left(\frac{\textit{class 1 bandwidth}}{\textit{theoretical bandwidth}}\right) \times (\textit{frac. of accesses to bw class 1}) + \\ & \vdots \\ & \left(\frac{\textit{class } (N-1) \textit{ bandwidth}}{\textit{theoretical bandwidth}}\right) \times (\textit{frac. of accesses to bw class } (N-1)) \end{aligned} \tag{4.1}$$

$$\textit{Data-Access Performance} = \sum_{i=0}^{N-1} \left(\frac{\textit{class } i \textit{ bandwidth}}{\textit{theoretical bandwidth}}\right) \times (\textit{frac. of accesses to bandwidth class } i) \tag{4.2}$$

**The Bandwidth-Class Performance Parameter $\alpha$**

Let $\alpha_{\{m,p\}_i}$ be the ratio of *data-access bandwidth* for accessing data on a memory node $(\alpha_{m_i})$ or PCIe device $(\alpha_{p_i})$ in bandwidth class $i$ to the *theoretical maximum bandwidth* for the device transferring data, where $i \in 0, ..., N-1$ and $N$ is the number of data-access bandwidth classes in a system.

For the *theoretical maximum* ***memory-access*** *bandwidth* term we use the JEDEC[1] DDR3-2133 specification of $17.066\ {}^{GB}/_{s}$. For the *theoretical maximum* ***PCIe-access*** *bandwidth* we use the PCIe 2.0 specification of $12.8\ {}^{GB}/_{s}$ [2] since it is the standard used in most modern systems. Dividing the measured data-access bandwidth by the maximum theoretical bandwidth for the device performing the data transfer normalizes the results. Normalization facilitates the use of the performance models to compare machines against a common standard. It also presents the absolute performance of the machine while also simplifying the task of comparing the data-access characteristics of different architectures by presenting a single number that represents data-access bandwidth.

Once the *data-access bandwidth* values for each data-access bandwidth class are known using the characterization methods presented in Chapter 3, the $\alpha_{\{m,p\}_i}$ values for a system can be calculated and stored since these values are static for a given system configuration. This allows for efficient use of performance model parameters in lookup-table fashion by programmers and run-time schedulers.

[1] JEDEC Solid State Technology Association, formerly known as the Joint Electron Devices Engineering Council

[2] 8/10 encoding of the $16\ {}^{GB}/_{s}$ specification for a x16 PCIe slot

In summary, the $\alpha$ parameter describes the data-access performance of a given bandwidth class:

$$\alpha_{\{m,p\}_i} = \frac{data\text{-}access\ bandwidth}{theoretical\ maximum\ bandwidth} \tag{4.3}$$

For the $\alpha_{m_i}$ parameters of any machine, the data locations ($i$) are explicitly ordered in terms of degrading performance:

$$\begin{aligned} \alpha_{\{m,p\}_0} &= Class\ 0\ Bandwidth \\ \alpha_{\{m,p\}_1} &= Class\ 1\ Bandwidth \\ &\vdots \\ \alpha_{\{m,p\}_{N-1}} &= Class\ (N-1)\ Bandwidth \end{aligned} \tag{4.4}$$

where $N$ is the number of bandwidth classes in a system.

**The *Fraction of Accesses* Parameter $\theta$**

Let $\theta$ be the fraction of accesses made by a program to resources belonging to a given bandwidth class in a system. The $\theta$ component of the performance model gives the appropriate weight to each bandwidth class for a given application's data-access profile. It is the frequency of remote data accesses in NUMA systems that is the heart of modelling or estimating the effects of NUMA data-access characteristics. By properly weighting the bandwidth classes we relay the frequency of accesses to each bandwidth class for a given application. That said, calculating the $\theta$ parameter of the performance model is a non-trivial task.

To calculate the frequency of data accesses to a given resource, a programmer or run-time system must to determine to what bandwidth class a data-access belongs. System hardware event

counters [1][4][22][24] may be of use to determine this relationship, but such profiling is labor-intensive, application-specific, and potentially difficult to analyze. Determining on which memory node a page of memory in a Linux system is allocated is possible using the `/proc` kernel interface, but this approach is tedious and requires knowledge of the memory addresses in use by a program. A better solution is to explicitly specify the location of memory allocations and CPU scheduler policy using `libnuma`; that way, a programmer or run-time system is aware of what data is located on which device.

A third solution is to use a compiler that can provide data-access information for a program at compile-time, thereby providing hints to the run-time scheduler as to what data a task or function may need during execution. It is with this "compiler-hints" idea that the performance models presented here were developed; some layer of software or the programmer must know where the data are located in the system in order to determine which data accesses are going to be local and which will be remote. The calculation of remote-access probability presented in [6] may also be of use to determine the probability of a given remote-data-access and the bandwidth associated with that probability, but we have not explored this potential solution.

Using the $\alpha_{\{m,p\}_i}$ terms and knowledge of a program's data-access characteristics, we parameterize the original bandwidth model statement in Equation 4.1 to produce a parameterized perfor-

mance model:

$$D_{\{m,p\}} = \alpha_{\{m,p\}_0} \times \theta_0 + \tag{4.5}$$

$$\alpha_{\{m,p\}_1} \times \theta_1 +$$

$$\vdots$$

$$\alpha_{\{m,p\}_{N-1}} \times \theta_{N-1}$$

$$D_{\{m,p\}} = \sum_{i=0}^{N-1} \alpha_{\{m,p\}_i} \times \theta_i \tag{4.6}$$

### 4.1.1 Meaning of Performance Model Values

The performance model function presented above produces a decimal value $0 < D_{\{m,p\}} \leq 1$, with a value of $D_{\{m,p\}} = 1$ indicating that a system achieves perfect theoretical data-access performance. Performance model results may be compared to determine the efficiency of scheduling a task in a given location (i.e., running a thread on a specific core or memory node) within a system, or they may be used to analyze architectures by modelling identical applications using identical data-access scenarios in different systems. In any case, the normalized performance model result provides an indication of the percent of total theoretical memory and/or PCIe bandwidth that a program may achieve in a machine. Given the importance of considering data-access constraints in current and future systems, the performance models presented here are useful tools for analyzing the data-access performance of current and future NUMA systems.

We discuss details and examples of the memory-access and PCIe-access performance models in

the following sections.

## 4.2 Memory-Access Performance Model

Let $\alpha_{m_i}$ be the ratio of data-access bandwidth when accessing data on memory node $i$ to the maximum theoretical bandwidth for the DDR3-2133 specification[3] of 17.066 $^{GB}/_{s}$ and $\theta_{m_i}$ is the fraction of a program's memory accesses made to data in bandwidth class $i$, where $i \in [0, ..., N-1]$ and $N$ is the number of memory-access bandwidth classes in a system. The memory data-access performance model, $D_{m_{bw}}$, is

$$\begin{aligned} D_{m_{bw}} = \alpha_{m_0} \times \theta_{m_0} + \\ \alpha_{m_1} \times \theta_{m_1} + \\ \vdots \\ \alpha_{m_{(N-1)}} \times \theta_{m_{(N-1)}} \end{aligned} \tag{4.7}$$

To synthesize the memory-access performance model we first consider a single process running in the two-socket, four-memory-node AMD 2P system described in Table 3.3. Considering the single-threaded `STREAM` benchmark results in Table 3.4, we see that the performance model for the AMD 2P system must have three $\alpha_{m_i}$ parameters corresponding to the *Class 0*, *Class 1*, and *Class 2* memory-access bandwidth classes present in that system.

This memory-access bandwidth class hierarchy provides a relative ordering of data locations for

[3]The maximum DDR3 clockrate at the time of publication is 2133 MHz

the cores of each memory node in a system. It is possible to use this model and data-access information gathered from a program to approximate the impact of data-access bandwidth when running a task on a given CPU core.

For the following examples we use the data-access memory performance model for the AMD 2P system:

$$D_{m_{bw}} = \alpha_{m_0} \times \theta_{m_0} + \alpha_{m_1} \times \theta_{m_1} + \alpha_{m_2} \times \theta_{m_2} \tag{4.8}$$

As mentioned in Chapter 3, we consider memory nodes as the locations where programs may be executed since modern architectures show little variation of data-access bandwidth between the cores attached to a memory node. Therefore, all cores directly attached to a memory node will share a single set of $\alpha_{m_i}$'s to describe their possible memory-access bandwidth values, and it is on which memory node a process is running that determines its memory-access bandwidth values throughout the system.

To use the memory-access performance model, each $\alpha_{m_i}$ for a given system must be defined. For convenience we define the parameter $\Gamma$ as the JEDEC DDR3-2133 standard: $\Gamma = 17.066\ {}^{GB}/_{s}$. The memory-access bandwidth parameters for the system are reported as a fraction of $\Gamma$. To calculate the value of each $\alpha_{m_i}$ parameter we use the best result for each bandwidth class shown in Table 3.4 and normalize it by $\Gamma$.

The AMD 2P architecture has the following memory-access bandwidth classes as determined from the results of the Open64-compiled, single-threaded `STREAM Triad` benchmark from Table 3.4:

- Class 0 Memory Access Bandwidth: $9.3\ {}^{GB}/_{s}$

- Class 1: $5.6\ ^{GB}/_{s}$
- Class 2: $3.1\ ^{GB}/_{s}$

Using the benchmark data above and the previously-defined $\Gamma$ parameter, we calculate the three $\alpha_{m_i}$ parameters for the AMD 2P system:

$$\alpha_{m_0} = \frac{9.3\ GB/s}{\Gamma} = 0.5449 \tag{4.9}$$

$$\alpha_{m_1} = \frac{5.6\ GB/s}{\Gamma} = 0.3281$$

$$\alpha_{m_2} = \frac{3.1\ GB/s}{\Gamma} = 0.1816$$

By defining the $\alpha_{m_i}$'s for a system it is possible to analyze the memory-access performance for programs using knowledge of a program's memory-access behavior.

### 4.2.1 Memory-Access Performance Model Examples

We consider two hypothetical program memory-access profiles to illustrate the utility of the data-access memory performance model. Consider a single-threaded program with the following memory-access characteristics running in the AMD 2P system:

- 50% of memory accesses are to memory allocated on the Class 0 memory node
- 50% of memory access are to memory allocated on the Class 1 memory node

Using the characteristics listed above and the performance model parameters for the AMD 2P system, the data-access memory performance model, $D_{m_{bw}}$, is

$$\begin{aligned} D_{m_{bw}} &= 0.5 \times \alpha_{m0} + 0.5 \times \alpha_{m1} \\ &= 0.5 \times 0.5449 + 0.5 \times 0.3281 = 0.4365 \end{aligned} \tag{4.10}$$

This results indicates that a program using this memory-access pattern may achieve 43.65% of theoretical bandwidth if the program is extremely memory-bandwidth sensitive.

Consider another example where the program's memory layout changes and 50% of accesses are Class 0 25% are Class 1, and 25% are Class 2. In this case $D_{m_{bw}}$ is

$$D_{m_{bw}} = 0.5 \times \alpha_{m0} + 0.25 \times \alpha_{m1} + 0.25 \times \alpha_{m2} = 0.4000 \tag{4.11}$$

As expected, the addition of Class 2 accesses decreases the value of $D_{m_{bw}}$, indicating an allocation scheme that is up to 3% worse than the previous configuration. Compared to the maximum memory-access bandwidth available in a system, $\alpha_{m_0}$, the application having a high proportion of either Class 1 or Class 2 memory accesses results in a potential loss of nearly 11% of memory-access bandwidth, as seen when $D_{m_{bw}}$ in Equation 4.10 is compared to $\alpha_{m_0}$:

$$D_{m_{bw}} - \alpha_{m_0} = 0.4365 - 0.5449 = -0.1084 \tag{4.12}$$

For an application that is sensitive to data-access bandwidth, this loss of memory bandwidth will most likely correlate to worse overall program performance; we investigate this model-to-application-performance conversion in Chapter 6.

## 4.3 PCIe Access Performance Model

Let $\alpha_{p_i}$ be the ratio of data-access bandwidth when a processor accesses data on a PCIe device belonging to bandwidth class $i$ to the maximum theoretical bandwidth for the PCIe 2.0 specification of $12.8\,{}^{GB}/_{s}$ and $\theta_{p_i}$ is the fraction of a program's PCIe data-accesses made to a device in bandwidth class $i$, where $i \in [0, ..., N-1]$ and $N$ is the number of PCIe-access bandwidth classes in a system. The PCIe data-access performance model, $D_{p_{bw}}$, is

$$\begin{aligned} D_{p_{bw}} = {} & \alpha_{p_0} \times (\%\ p_0\ accesses) + \\ & \alpha_{p_1} \times (\%\ p_1\ accesses) + \\ & \vdots \\ & \alpha_{p_{(N-1)}} \times (\%\ p_{(N-1)}\ accesses) \end{aligned} \tag{4.13}$$

The data-access bandwidth seen by a program accessing a PCIe device is determined by the various system characteristics discussed in Chapter 2. Whereas the memory-access performance model is a function of memory nodes accessing other memory nodes over DDR3 channels and CPU interconnect links, the PCIe-access performance model is a function of memory nodes accessing devices through I/O controllers over CPU interconnect links and the PCIe interconnect. To synthesize the PCIe-access performance model we consider again a single process running in the two-socket, four-memory-node, two-I/O-controller AMD 2P system. The `SHOC` benchmark results in Table 3.4 show that the PCIe-access performance model for the AMD 2P system must have three $\alpha_{m_i}$ parameters corresponding to the *Class 0*, *Class 1*, and *Class 2* PCIe-access bandwidth classes present in that system. Although there are only two I/O controllers in the AMD 2P system, there are

still three data-access bandwidth classes due to the intra-socket HyperTransport link heterogeneity discussed in Section 2.3.2.

Similar to our method for synthesizing the memory-access performance model, we present examples of the PCIe-access performance model using the AMD 2P system's benchmark results. The PCIe-access performance model for this system is

$$D_{p_{bw}} = \alpha_{p_0} \times (\%\ p_0\ accesses) + \alpha_{p_1} \times (\%\ p_1\ accesses) + \alpha_{p_2} \times (\%\ p_2\ accesses) \quad (4.14)$$

We again consider memory nodes as the locations where programs may be executed because PCIe data-access performance is a factor of the memory-node-to-memory-node CPU interconnect link characteristics with respect to the I/O controllers in the system. Therefore, all cores directly attached to a memory node will share a single set of $\alpha_{p_i}$ parameters to describe their possible PCIe-access bandwidth values.

To use the PCIe-access performance model, each $\alpha_{p_i}$ for a given system must be defined. For convenience we define the parameter $\Phi$ as the PCIe 2.0 bandwidth standard: $\Phi = 12.8\ {}^{GB}/_{s}$. The PCIe-access bandwidth parameters for the system are reported as a fraction of $\Phi$. To calculate each $\alpha_{p_i}$'s value we use the best result for each PCIe bandwidth class shown in in Table 3.4 and normalize it by $\Phi$.

The AMD 2P architecture has the following PCIe-access bandwidth classes as determined from the results of the OpenCL `SHOC BusSpeedDownload` and `BusSpeedReadback` benchmarks from Table 3.4:

- Writing Data to Device (Download)

  - Class 0 PCIe Access Bandwidth: $5.8\ ^{GB}/_{s}$
  - Class 1: $5.3\ ^{GB}/_{s}$
  - Class 2: $2.3\ ^{GB}/_{s}$

- Reading Data from Device (Readback)
  - Class 0 PCIe Access Bandwidth: $6.6\ ^{GB}/_{s}$
  - Class 1: $5.3\ ^{GB}/_{s}$
  - Class 2: $2.3\ ^{GB}/_{s}$

For the PCIe-access performance model we add a third variable to the $\alpha$ parameter, $r$ or $w$, denoting reading from the device (readback) and writing to the device (download), respectively.

Using the benchmark data above and the previously-defined $\Phi$ parameter we calculate the six $\alpha_{p_{\{r,w\}i}}$ parameters for the AMD 2P system:

$$\alpha_{p_{r0}} = \frac{6.6\ ^{GB}/_{s}}{\Phi} = 0.5156 \tag{4.15}$$

$$\alpha_{p_{r1}} = \frac{5.3\ ^{GB}/_{s}}{\Phi} = 0.4141$$

$$\alpha_{p_{r2}} = \frac{2.3\ ^{GB}/_{s}}{\Phi} = 0.1797$$

$$\alpha_{p_{w0}} = \frac{5.8\ ^{GB}/_{s}}{\Phi} = 0.4531 \tag{4.16}$$

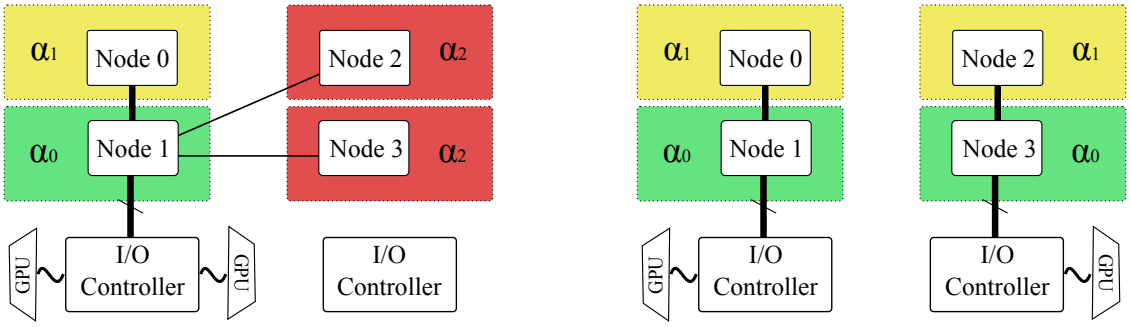
$$\alpha_{p_{w1}} = \frac{5.3\ ^{GB}/_{s}}{\Phi} = 0.4141$$

$$\alpha_{p_{w2}} = \frac{2.3\ ^{GB}/_{s}}{\Phi} = 0.1797$$

By defining the $\alpha_{p_i}$ parameters for a system it is possible to analyze the PCIe-access performance for programs using knowledge of a program's PCIe-access behavior.

### 4.3.1 PCIe-Access Bandwidth Performance Model Examples

Consider two GPU programs running concurrently in the AMD 2P system. This machine has two GPUs installed, which allows for kernels of code to be executed simultaneously on both GPU devices. The optimal configuration for such a system is to have each GPU on its own I/O controller so that the two kernels may each transfer data using the full bandwidth of the interconnect lanes connecting the GPU device to the host CPU core. However, in this machine both GPUs are installed on the same I/O controller. The result of this configuration is that one CPU socket will never realize the $\alpha_{p_0}$ bandwidth when accessing the device due to I/O controller overhead. In fact, GPU-access bandwidth will be limited to $\alpha_{p_2}$ for this socket since $\alpha_{p_0}$ and $\alpha_{p_1}$ are both in the socket that is local to the GPUs, as shown in Figure 4.1. To show the effect of these configuration choices on




(a) Current (b) Optimal

Figure 4.1: AMD 2P System GPU Configurations

peak theoretical bandwidth we compare the performance of reading back data from both of the GPU devices in the current configuration and in the optimal configuration, with each host CPU process having exclusive access to a memory node (one host process per memory node). This

produces three possible execution scenarios for each configuration, as shown below.

Two GPUs on one I/O controller (current configuration):

$$1. \quad D_{p_{r_{bw}}} = 0.5 \times \alpha_{p_{r0}} + 0.5 \times \alpha_{p_{r2}} = 0.3477 \tag{4.17}$$

$$2. \quad D_{p_{r_{bw}}} = 0.5 \times \alpha_{p_{r1}} + 0.5 \times \alpha_{p_{r2}} = 0.2969$$

$$3. \quad D_{p_{r_{bw}}} = 0.5 \times \alpha_{p_{r2}} + 0.5 \times \alpha_{p_{r2}} = 0.1797$$

Each GPU on its own I/O controller (optimal configuration):

$$1. \quad D_{p_{r_{bw}}} = 0.5 \times \alpha_{p_{r0}} + 0.5 \times \alpha_{p_{r0}} = 0.5156 \tag{4.18}$$

$$2. \quad D_{p_{r_{bw}}} = 0.5 \times \alpha_{p_{r0}} + 0.5 \times \alpha_{p_{r1}} = 0.4648$$

$$3. \quad D_{p_{r_{bw}}} = 0.5 \times \alpha_{p_{r1}} + 0.5 \times \alpha_{p_{r1}} = 0.4141$$

The current configuration has a worst-case mapping of GPU host processes to memory nodes that achieves 33% less PCIe-access bandwidth than the optimal configuration's best-case mapping; the best-case mapping of the current configuration still achieves 7% less PCIe-access bandwidth than the worst-case mapping in the optimal configuration. These examples indicate that the current GPU configuration may achieve between 33% and 7% less PCIe-access bandwidth than an optimal configuration would achieve when reading from the GPU devices.

The results presented in Chapter 3 may be used to generate $\alpha$ values for the test AMD and Intel architectures. These $\alpha$ values are shown in Table 4.1.

We have presented and discussed the method for modelling data-access performance using the benchmark results presented in Chapter 3. These performance models clearly show the percentage

Table 4.1: $\alpha$ Values for Data-Access Costs

AMD and Intel Machine Attributes listed in Table 3.3

| | AMD | | | INTEL | | |
|---|---|---|---|---|---|---|
| | **MEMORY BANDWIDTH COSTS (GB/S)** $\Gamma = 17.066\ ^{GB}/_{s}$ | | | | | |
| | **GNU** | **Intel** | **Open64** | **GNU** | **Intel** | **Open64** |
| $\alpha_{m_0}$ | 0.3047 | 0.4570 | 0.5449 | 0.5391 | 0.6153 | 0.5274 |
| $\alpha_{m_1}$ | 0.1934 | 0.3106 | 0.3281 | 0.3809 | 0.4160 | 0.3574 |
| $\alpha_{m_2}$ | 0.1231 | 0.1816 | 0.1816 | – | – | – |
| | **PCIE BANDWIDTH COSTS (GB/S)** $\Phi = 12.8\ ^{GB}/_{s}$ | | | | | |
| | **OpenCL (CUDA is Identical)** | | | **OpenCL (CUDA is Identical)** | | |
| $\alpha_{p_{r0}}$ | 0.5156 | | | 0.4766 | | |
| $\alpha_{p_{w0}}$ | 0.4531 | | | 0.4766 | | |
| $\alpha_{p_{r1}}$ | 0.4141 | | | 0.3203 | | |
| $\alpha_{p_{w1}}$ | 0.4141 | | | 0.3672 | | |
| $\alpha_{p_{r2}}$ | 0.1797 | | | – | | |
| $\alpha_{p_{w2}}$ | 0.1797 | | | – | | |

of theoretical data-access bandwidth that a system can achieve and may be combined with the data-access information of a program to estimate the impact of data-access performance on a program executed in a given machine with a given process-to-core mapping.

# Chapter 5

# Cbench as a Data-Access Analysis Tool

The NUMA data-access characterization and modelling discussed in Chapters 3 and 4 are effective methods for analyzing data-access performance in modern NUMA systems. To make such analysis more robust, automated, flexible, and convenient, we present our modifications to the Cbench Scalable Testing Framework [3][32] developed at Sandia National Laboratories. In this chapter we present an overview of the Cbench testing framework, its applicability to the characterization work presented in this thesis, and details of our automation of the NUMA data-access bandwidth analysis using Cbench.

## 5.1 Overview of Cbench

### 5.1.1 Design & Purpose

Cbench is a benchmark utility framework designed for HPC system integration engineers and analysts. The Cbench framework consists of a toolkit of Perl and Bash scripts that manage the building of supported open-source benchmarks from source code and the creation, submission, and analysis of benchmark jobs on Linux HPC clusters. Figure 5.1 illustrates the four phases of the Cbench benchmarking workflow and the core Cbench files associated with each phase.

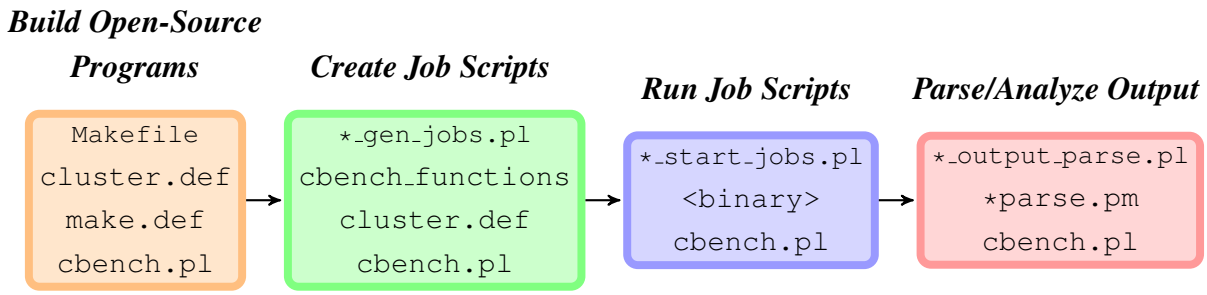



Figure 5.1: Cbench Benchmarking Workflow

Cbench also provides basic statistical analysis in the "Parse/Analyze Output" phase to compare benchmark results for nodes in a cluster so that machines with sub-standard performance may be automatically identified. Running benchmark programs in Cbench is also a robust and straightforward method for executing identical tests on evaluation machines in order to compare performance.

Table 5.1 summarizes our contributions to Cbench. The primary focus of the Cbench framework is to run benchmark jobs across a large number of nodes in a Linux cluster to test various attributes

Table 5.1: Cbench Capabilities

| | Old Cbench | NUMA Cbench |
|---|---|---|
| **MPI-level** | ✓ | ✓ |
| **Node-level** | ✓ | ✓ |
| **NUMA Memory** | | ✓ |
| **GPU** | | ✓ |
| **PCIe Bandwidth** | | ✓ |
| **NUMA BW Class** | | ✓ |

of the system. Benchmark tests run within Cbench are generally divided into cluster-level MPI-based tests (e.g., `Linpack`, OSU Bandwidth Benchmarks, NAS Parallel Benchmarks, etc.) and node-level serial or parallel tests (e.g., `STREAM`, `memtester`, node-level `Linpack`, etc.). As the NUMA data-access characterization effort presented in this thesis is focused on node-level data-access bandwidth, it is the node-level portion of Cbench that is our focus.

### 5.1.2 Node-level System Analysis

The core set of Cbench node-level tests typically used to characterize individual compute nodes includes

**`STREAM`** – Measures memory bandwidth.

**`MPI-STREAM`** – Measures multi-threaded memory bandwidth.

**`Cachebench`** – Characterizes the memory subsystem.

**`DGEMM`** – Performs the DGEMM BLAS matrix operation to evaluate computational capability.

**`Linpack`** – Performs scalable, multi-threaded Linear Algebra calculations.

**`NPB`** – The NAS Parallel Benchmarks; tests the parallel capabilities of machines.

This suite of node-level benchmarks is used in Cbench to analyze the performance of various subsystems in each node and compare the performance distribution of a large sampling of nodes. Comparing system performance is particularly useful during the integration of a new cluster, as finding and replacing poorly-performing hardware is a priority as nodes are brought online.

Another use for the Cbench node-level tests is evaluating new systems and architectures. The Cbench Single-Node-Benchmark (SNB) script runs the tests listed above and produces a LaTeX report showing the baseline single-threaded and multi-threaded scaling performance of a given architecture. It is to the SNB script that the NUMA data-access characterization tests and analysis are added.

## 5.2 Data-Access Characterization Tests using the Cbench Single-Node-Benchmark Tool

The Cbench Single-Node-Benchmark (SNB) workflow is similar to the overall Cbench workflow shown in Figure 5.1. Instead of generating and running jobs through a batch scheduler, however, the SNB script directly invokes the benchmarks on the system where the script is executed.

Adding NUMA data-access tests to the SNB script required the modification of the last three phases of the SNB workflow that are shown in Figure 5.2. To fully automate the data-access characterization tests presented in this thesis we added to Cbench the capability to run the necessary NUMA data-access benchmarks using the SNB script. These additions included the ability to generate bandwidth class and performance model parameters from the data produced by these NUMA benchmark tests. In this section we present the three types of data-access analysis and the automatic bandwidth class/performance model parameter analysis that we added to Cbench.

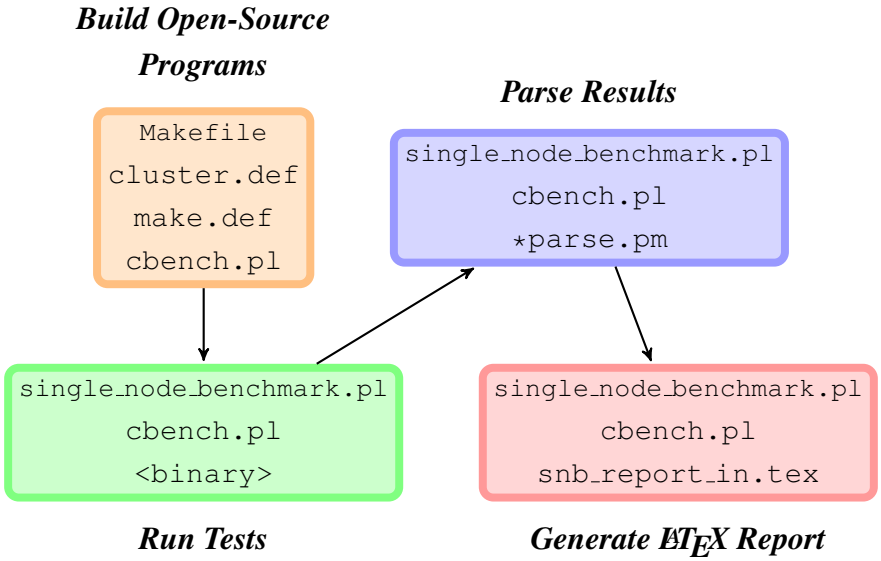



Figure 5.2: Cbench Single-Node-Benchmark Workflow

### 5.2.1 Single-Threaded Memory Bandwidth Testing

The method for running core- or memory-node-to-memory-node single-threaded `STREAM` tests to characterize NUMA data-access memory bandwidth using the Single-Node-Benchmark script

is very similar to the method described in Section 3.2.1. Cbench builds many versions of the `STREAM` benchmark by default, and incorporating the different compiler versions is only a matter of instructing Cbench to build the benchmarks with the appropriate compiler. The SNB script runs any `STREAM` binary it finds in its `CBENCHTEST` directory, allowing for an even greater degree of comparison than that provided by the original hand-executed method described in Section 3.2.1.

We extended the SNB script to characterize data accesses in NUMA systems by running a given program from all cores to all memory nodes or from all memory nodes to all memory nodes and binding CPU and memory locations as described in Chapter 3. This solution is not `STREAM`-specific, and thereby has the benefit of providing NUMA characterization capability to any of the node-level tests already supported by Cbench. Algorithm 1 summarizes the design of the Single-Node-Benchmark NUMA single-threaded test capability.

**Algorithm 1** Single-Threaded Cbench NUMA Test Pseudocode

**for all** $cpu \in \{$all cores or memory nodes$\}$ **do**
  **for all** $mem \in \{$all memory nodes$\}$ **do**
    **for all** $binary$ **do**
      `numactl --{cpunodebind,physcpubind}=`*cpu* `--membind=`*mem* *binary*`;`
    **end for**
  **end for**
**end for**

### 5.2.2 Multi-Threaded Memory Bandwidth Testing

The `MPI-STREAM` tests that were already in the Single-Node-Benchmark script were designed to test the maximum aggregate bandwidth within a compute node. We extend the use of `MPI-STREAM`

in the SNB script to evaluate memory-node-to-memory-node CPU interconnect link characteristics, as discussed in Section 3.2.1. The pseudocode for the `MPI-STREAM` tests is given in Algorithm 2.

**Algorithm 2** Multi-Threaded `STREAM` NUMA Test Pseudocode

**for all** $mem \in \{$all memory nodes$\}$ **do**
  **for all** $MPI\text{-}binary$ **do**
    $i = 1$;
    **while** $\Delta > 0$ **do**
      `mpirun -np` $i$ `numactl --cpunodebind=`$cpu$ `--membind=`$mem$ $MPI\text{-}binary$;
      $i$++;
      $\Delta = current\_result - previous\_result$;
    **end while**
  **end for**
**end for**

### 5.2.3 GPU PCIe Bandwidth Testing

GPU benchmark support in Cbench is in the early stages of development, and the addition of the `SHOC` PCIe bandwidth tests to the Single-Node-Benchmark script provides the first complete GPU testing capability for Cbench. These `SHOC` tests facilitate PCIe data-access characterization similar to the `STREAM` tests discussed above. The algorithm for these tests is shown in Algorithm 3.

### 5.2.4 Automatic Performance Model Generation

Generation of the data-access performance model parameter $\alpha$ presented in Chapter 4 requires the determination of bandwidth classes using the appropriate benchmark results. We use the K-

**Algorithm 3** `SHOC` Cbench NUMA Test Pseudocode

**for all** $mode \in \{CUDA, OpenCL\}$ **do**
  **for all** $device \in \{available\ GPU\ devices\}$ **do**
    **for all** $mem \in \{$all memory nodes$\}$ **do**
      `numactl --cpunodebind=`$mem$ `--membind=`$mem$ `\`
        $mode$`/Serial/BusSpeedDownload -d` $device$`;`
      `numactl --cpunodebind=`$mem$ `--membind=`$mem$ `\`
        $mode$`/Serial/BusSpeedReadback -d` $device$`;`
    **end for**
  **end for**
**end for**

Means clustering algorithm [25] via the Perl `Algorithm::KMeans` module to find clusters representing bandwidth classes in the benchmark data. To prepare the data for K-Means cluster analysis we process the data and find the best benchmark values for each type of data-access, e.g., the best single-threaded `STREAM Triad` result for all single-threaded `Triad` runs with the process running on node 0 and its data allocated on node 3. We compare every result for a given data-access scenario against the best result for that scenario and select only the results of the best-performing executable to use for classification. This pruning assists the K-Means algorithm with determining the true bandwidth classes by preventing cluster detection for results that are produced by poor-performing benchmark executables and not actual bandwidth classes.

Once the data are pruned, the K-Means algorithm is used to determine the best clusters for the benchmark dataset. The pruning and clustering process is shown in Figure 5.3.

Using the K-Means algorithm along with these heuristics, the Cbench Single-Node-Benchmark script successfully detects the data-access bandwidth classes present in the benchmark data for either memory or PCIe data-access benchmarks. It should be noted that the K-Means algorithm uses

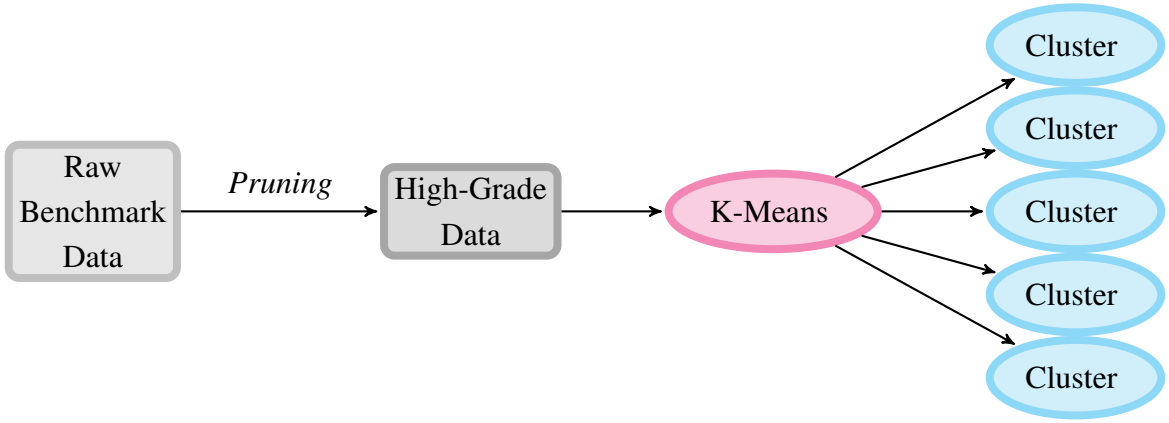



Figure 5.3: Clustering using K-Means

random initial cluster values and may therefore fail to detect the bandwidth clusters appropriately. This is inherent in the K-Means algorithm, but we intend to work to improve the accuracy of the automatic bandwidth class detection as part of our future work.

With the data-access bandwidth class information in-hand, Cbench also automatically generates the performance model $\alpha$ value associated with each type of data-access. This provides analysts with both raw and normalized data-access bandwidth information.

### 5.2.5 Cbench Single-Node-Benchmark Report

Tables 5.2 and 5.3 show the LaTeX report for memory data-access bandwidth results produced by the Cbench Single-Node-Benchmark script in the AMD 2P machine[1]. Similarly, Tables 5.4 and 5.5 show the SNB LaTeX report for PCIe data-access bandwidth results.

[1]Results for Cbench NUMA characterization tests run in an AMD 4P machine are shown in Appendix A

Table 5.2: **Cbench Memory Data-Access Bandwidth Classes**

| $\Gamma = 17066MB/s$ | `STREAM Triad` |
|---|---|
| **Class 0** $(\alpha_{m_0})$ | $\leq$ 6441 MB/s (0.3774) |
| **Class 1** $(\alpha_{m_1})$ | $\leq$ 3915 MB/s (0.2294) |
| **Class 2** $(\alpha_{m_2})$ | $\leq$ 2181 MB/s (0.1278) |

Table 5.3: **Cbench NUMA `STREAM` Bandwidth Test Results** — the best value in each column is highlighted

| **STREAM Triad (MB/s)** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Node 0 to Node 0 | Node 0 to Node 1 | Node 0 to Node 2 | Node 0 to Node 3 | ··· | Node 3 to Node 0 | Node 3 to Node 1 | Node 3 to Node 2 | Node 3 to Node 3 |
| stream-big-c | 6395 | 3905 | 2162 | 2146 | ··· | 2127 | 2141 | 3915 | 6323 |
| stream-big-f | 6114 | 3779 | 2135 | 2109 | ··· | 2107 | 2102 | 3786 | 6109 |
| stream-c | 5316 | 3347 | 2181 | 2147 | ··· | 2147 | 2130 | 3875 | 5341 |
| stream-f | 5351 | 3350 | 2166 | 2130 | ··· | 2154 | 2120 | 3897 | 5365 |
| stream-gcc-c | 6121 | 3875 | 2145 | 2141 | ··· | 2136 | 2147 | 3891 | 6284 |
| stream-gcc2-c | 6383 | 3907 | 2150 | 2122 | ··· | 2122 | 2118 | 3913 | 6141 |
| stream-gcc5-c | 6147 | 3789 | 2142 | 2132 | ··· | 2129 | 2138 | 3808 | 6137 |
| stream-gcc6-c | 6153 | 3792 | 2135 | 2109 | ··· | 2104 | 2095 | 3804 | 6120 |
| stream-gcc7-c | 6364 | 3874 | 2135 | 2132 | ··· | 2129 | 2128 | 3890 | 6316 |

Table 5.4: **Cbench GPU Data-Access Bandwidth Classes**

| $\Phi = 12.8GB/s$ | BusSpeedDownload | BusSpeedReadback |
|---|---|---|
| **Class 0** ($\alpha_{p_0}$) | $\leq$ 5.85 GB/s (0.4571) | $\leq$ 6.58 GB/s (0.5141) |
| **Class 1** ($\alpha_{p_1}$) | $\leq$ 5.32 GB/s (0.4157) | $\leq$ 5.33 GB/s (0.4161) |
| **Class 2** ($\alpha_{p_2}$) | $\leq$ 2.35 GB/s (0.1837) | $\leq$ 2.32 GB/s (0.1813) |

Table 5.5: **Cbench NUMA `SHOC` GPU Data-Access Bandwidth Test Results** — the best value in each column is highlighted

| **`BusSpeedDownload (GB/s)`** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **CUDA** | | | | | | | | | |
| | Node 0 to Node 0 | Node 0 to Node 1 | Node 0 to Node 2 | Node 0 to Node 3 | ··· | Node 3 to Node 0 | Node 3 to Node 1 | Node 3 to Node 2 | Node 3 to Node 3 |
| Device 0 | 5.8503 | 5.3153 | 2.3481 | 2.3192 | ··· | 5.8114 | 5.3190 | 2.3458 | 2.3026 |
| Device 1 | 5.8104 | 5.3164 | 2.3455 | 2.3159 | ··· | 5.8101 | 5.3202 | 2.3451 | 2.3026 |

| **`BusSpeedReadback (GB/s)`** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **CUDA** | | | | | | | | | |
| | Node 0 to Node 0 | Node 0 to Node 1 | Node 0 to Node 2 | Node 0 to Node 3 | ··· | Node 3 to Node 0 | Node 3 to Node 1 | Node 3 to Node 2 | Node 3 to Node 3 |
| Device 0 | 6.5800 | 5.3182 | 2.3089 | 2.2828 | ··· | 6.5760 | 5.2802 | 2.3109 | 2.2813 |
| Device 1 | 6.5781 | 5.3018 | 2.3083 | 2.2942 | ··· | 6.5756 | 5.2707 | 2.3104 | 2.2812 |

### 5.2.6 Summary

With the addition of NUMA data-access bandwidth characterization, the Cbench Single-Node-Benchmark script is a useful tool for analyzing and evaluating NUMA architectures at the node-level. The modular design of Cbench allows for additional NUMA characterization tests to be added to the framework as we look at other data-access performance issues. Furthermore, the automatic benchmarking and report generation of NUMA data-access bandwidth provides useful and easy-to-access data for system engineers and analysts.

# Chapter 6

# Data-Access Sensitivity of Scientific Algorithms

The data-access bandwidth characterization results and performance models presented in previous chapters provide details about the effects of NUMA architectural designs on a system's data-access bandwidth performance. However, knowledge of a system's data-access bandwidth capabilities alone does not directly indicate the effect of a system's NUMA characteristics on an actual user-space application. Factors such as data-access latency, data-access patterns, CPU performance, etc. also significantly impact program execution performance. In this chapter we observe the impact of data-access performance in NUMA systems on the execution performance of scientific algorithms by testing the impact of remote data-access bandwidth on the Graph 500 [8] and NAS Parallel Benchmarks [9]. Some of the algorithms in these benchmarks show a correlation to the data-

access bandwidth discussed in Chapter 4, providing insight into the impact that the data-access bandwidth of remote data accesses may have on the overall performance of a given algorithm.

The Graph 500 benchmark performs various types of breadth-first searches that are intended to test the performance of a machine or cluster when solving data-intensive problems. One of the primary metrics measured by Graph 500 is Traversed Edges Per Second (TEPS, or the rate at which edges in a graph are traversed), and this is the metric we consider when analyzing the data-access bandwidth sensitivity of Graph 500.

We use the following eight of the NAS Parallel Benchmarks to test a variety of scientific algorithms:

**MG:** `MultiGrid` - solves a 3D Poisson equation

**CG:** `Conjugate Gradient` - subroutine for solving systems of linear equations

**FT:** `Fast Fourier Transform` - solves a 3D Partial Differential Equation (PDE) using FFT

**IS:** `Integer Sort` - small-integer bucket sort

**EP:** `Embarrassingly Parallel` - generates independent Gaussian random variables

**BT:** `Block Tridiagonal` - non-linear PDE solver

**SP:** `Scalar Pentadiagonal` - non-linear PDE solver

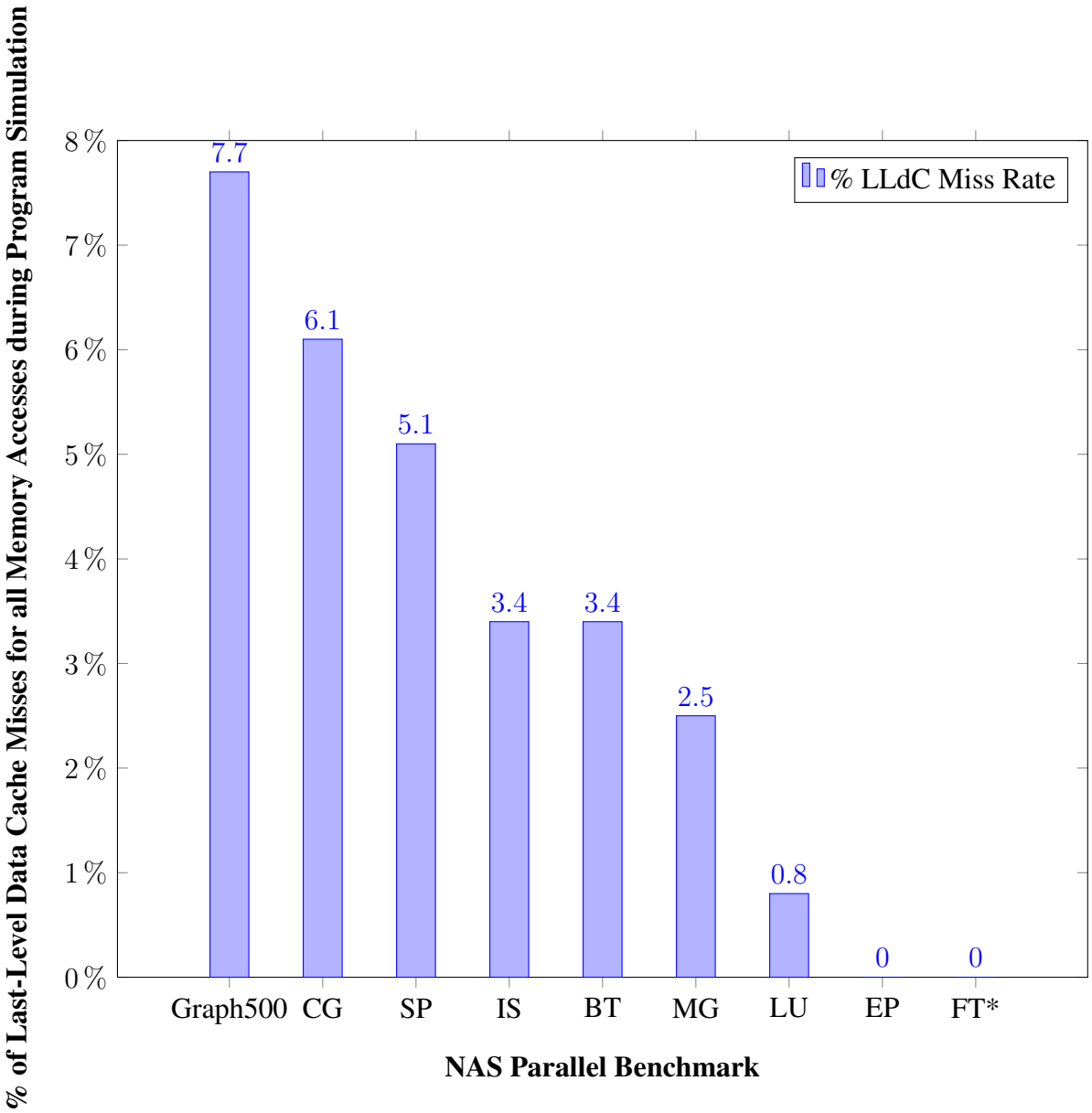
**LU:** `Lower-Upper symmetric Gauss-Seidel` - non-linear PDE solver

The metric reported by each of the `NPB` is million operations per second (Mop/s). Although the algorithms represented by these eight benchmarks are diverse, we expect that certain of the algorithms that are known to be CPU- or communication-intensive, such as the PDE solvers and

the `Embarrassingly Parallel` benchmark, will exhibit a low degree of sensitivity to data-access performance, whereas those algorithms that are known to be memory-access-intensive, such as `Integer Sort`, `Conjugate Gradient`, and `MultiGrid` may be more sensitive to data-access performance.

To assist with our analysis of these algorithms, we first look at the last-level-cache (LLC) miss rate for the execution of each program. The LLC miss rate indicates how often a memory-access in a program misses all levels of cache and must access main memory. Since the bandwidth of main memory data accesses suffers when remote memory is accessed, high LLC miss rates should generally correspond to higher sensitivity to memory-access performance. We use the `Valgrind` [31] program's `cachegrind` tool to analyze the Last-Level data Cache (LLdC) miss rate for each of the Graph 500 and NPB benchmarks. The results of these `cachegrind` simulations are presented in Figure 6.1.

The LLdC rates shown in Figure 6.1 indicate that algorithms with high LLdC miss rates are designed in such a way that data-access bandwidth may affect their performance since they access memory more frequently. As we will see from the benchmark results, however, LLdC miss rate does not always correlate to program performance, but simulation LLdC misses for a program does provide a starting point for analysis of the sensitivity of applications to data-access bandwidth.




**The FT benchmark caused Cachegrind to abort, so no results for FT are presented*

Figure 6.1: Cachegrind Simulation of `Benchmarks` Single-Threaded (`-np 1`) Benchmark Last-Level Data Cache Miss Rates

## 6.1 Memory-Access Benchmark Results

The Graph 500 and NAS Parallel Benchmarks were run using `numactl` to bind the benchmark process and its data to the desired memory nodes, similar to the benchmarking methods described in Chapter 3. We used the MPI-based `graph500_mpi_replicated` Graph 500 reference benchmark and ran it with a single MPI process to validate the single-threaded `STREAM` results presented in Chapter 3. The `NPB` were executed using a single MPI process in the same fashion.

The results for running the `graph500_mpi_replicated` benchmark in each of the AMD and Intel 2P machines are normalized by each machine's *Class 0* (local NUMA data access) results in Figure 6.2. We normalized the results by the *Class 0* results so that relative performance between data-access classes may be compared.

Both systems have a significant increase in bandwidth between $\alpha_0$ and $\alpha_1$ and we expect to see a significant performance hit for algorithms that are sensitive to data-access bandwidth. We revisit the $\alpha_i$ memory-access bandwidth classes presented in Chapter 4 to better understand the impact of memory-access bandwidth on these benchmarks. The data-access performance predicted by $\alpha_i$ for each machine and each bandwidth class is presented in Table 6.1 for reference.

We see from the Graph 500 results in Figure 6.2 that the performance loss when Graph 500 uses only remote memory (i.e. results in "Class 1" or "Class 2" in Figure 6.2) is not as severe as the performance model predicted for the AMD architecture and is more severe than the performance model predicted for the Intel architecture. As mentioned previously, the execution performance

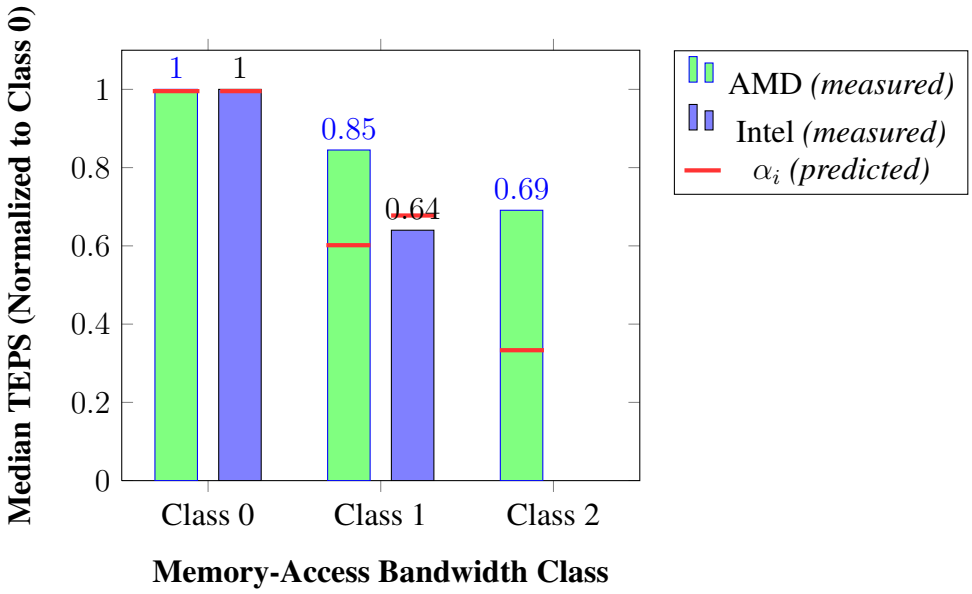
Table 6.1: $\alpha$ Values for Data-Access Latency

| | AMD | INTEL |
|---|---|---|
| | **Memory-Access Bandwidth (ratio to $\alpha_0$)** | |
| $\alpha_0$ | 1.00 | 1.00 |
| $\alpha_1$ | 0.6021 | 0.6777 |
| $\alpha_2$ | 0.3333 | – |

of Graph 500 depends not only on data-access bandwidth, but also data-access latency, processor performance, etc. It is these additional factors that are responsible for the misprediction; however, these additional factors are not the focus of this thesis. Nevertheless, this data shows that Graph 500 is sensitive to data-access bandwidth and would benefit from data-access performance-aware scheduling.

The NAS Parallel Benchmarks' results, shown in Figures 6.3 and 6.4, present a slightly more interesting case study. Some of the `NPB`, such as `SP`, `MG`, and `LU` have substantial sensitivity to data-access bandwidth, even though their LLdC rates vary from 5.1% (`SP`) to 0.8% (`LU`). Some of the anomalies in the data, such as the sensitivity of `CG` to data-access bandwidth in the Intel system but not in the AMD system require further profiling and analysis to properly explain. From these results it is apparent that considering Last-Level data Cache misses is not sufficient for accurately predicting sensitivity to data-access performance. Nevertheless, task scheduling using the performance model's bandwidth classes may substantially help those Graph 500 and `NPB` algorithms that

Figure 6.2: `graph500_mpi_replicated` Single-Threaded (`-np 1`) Benchmark Results




show sensitivity to data-access performance, while those that do not show such sensitivity will not be harmed by such scheduling.

## 6.2 PCIe-Access Benchmark Results

We next consider the `SHOC` GPU benchmarks to examine the effect of remote data-access performance on GPU computational algorithms We analyze the sensitivity of each benchmark to remote data-access performance using the CUDA `computeprof` tool's *% GPU time doing memory copy* metric. The `computeprof` results for the `SHOC` benchmarks are shown in Figure 6.5.

The *% GPU time doing memory copy* metric is suitable for determining a GPU program's sensitivity to data-access performance because it indicates whether a program spends a significant

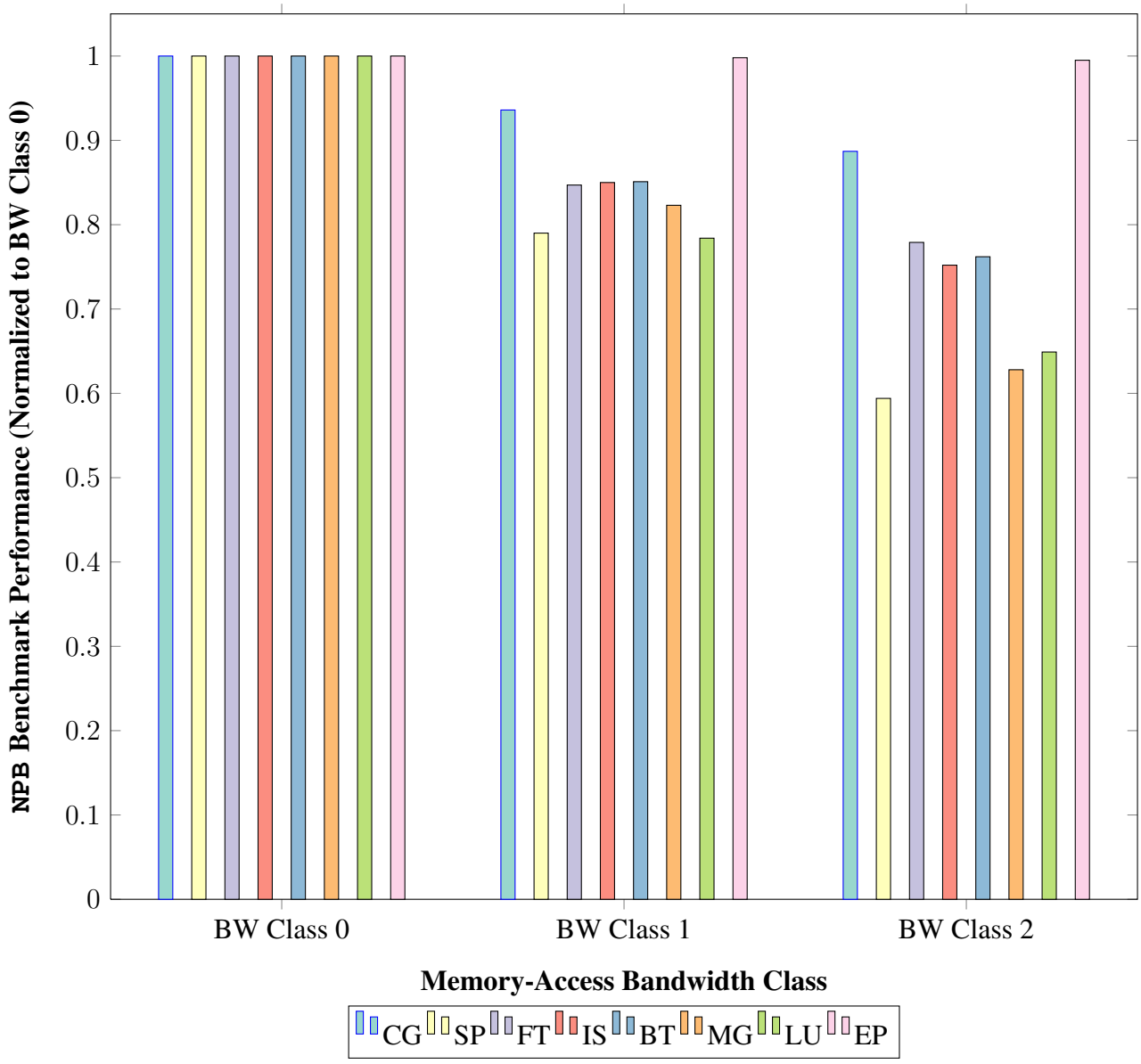



Figure 6.3: NAS Parallel Benchmarks Single-Threaded (-np 1) Benchmark Results in the AMD 2P System. Results within each class are presented in descending order by LLdC miss rate.

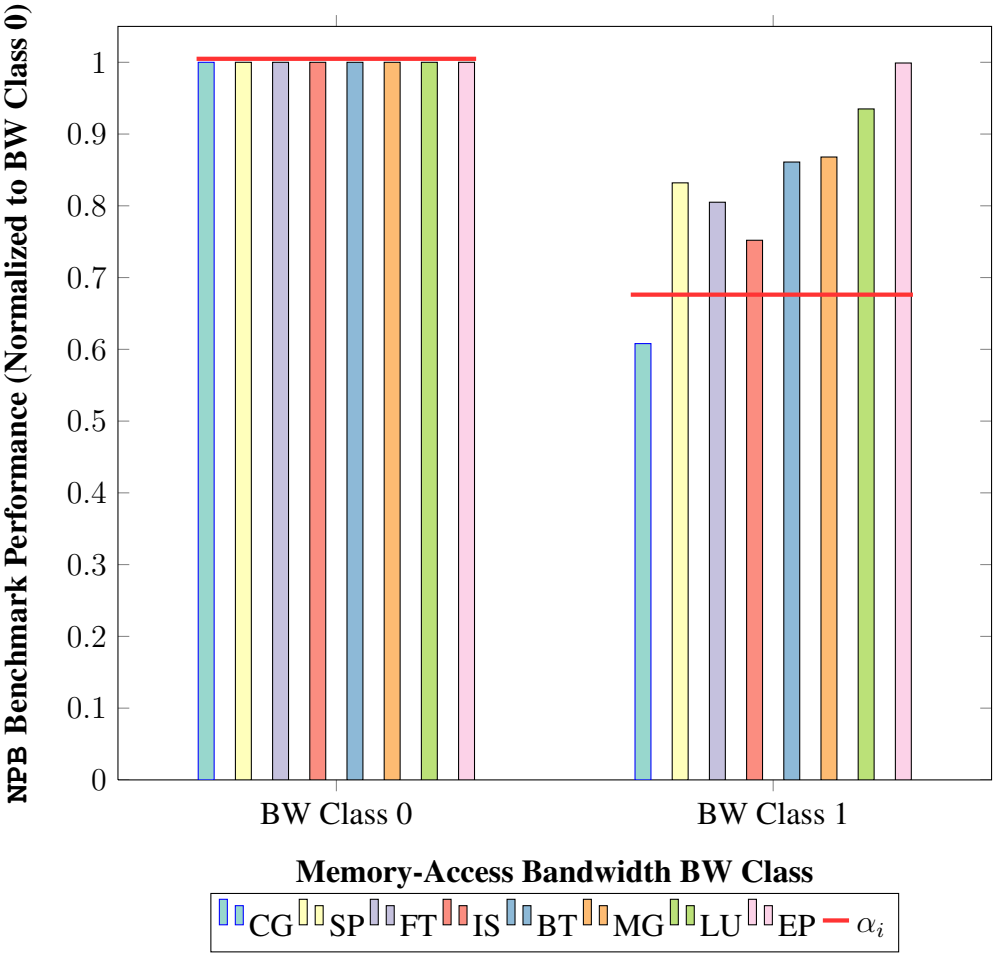



Figure 6.4: `NAS Parallel Benchmarks` Single-Threaded (`-np 1`) Benchmark Results in the Intel 2P System. Results within each class are presented in descending order by LLdC miss rate.

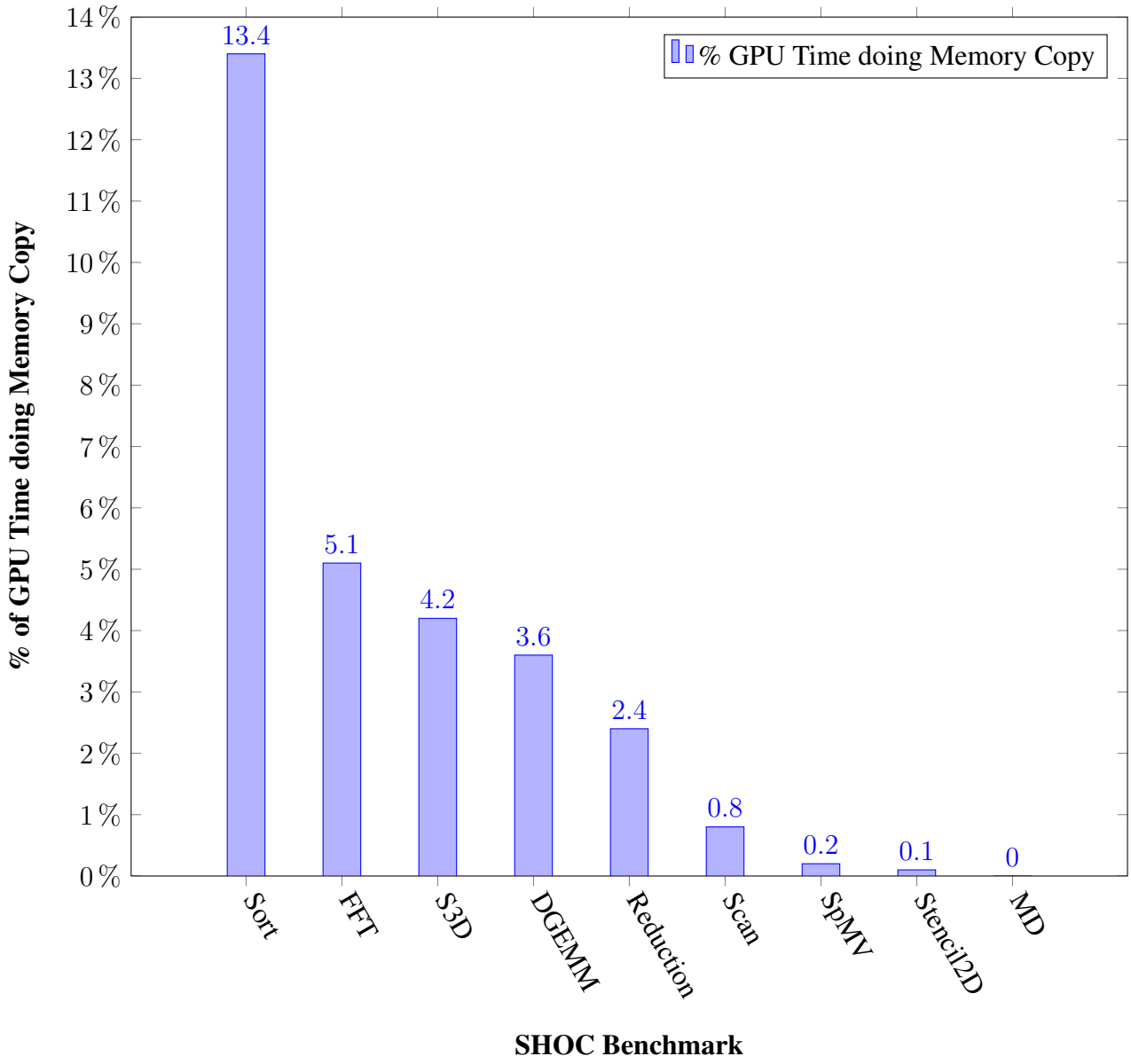



Figure 6.5: CUDA `computeprof` Profile of `SHOC` GPU Benchmarks' % of GPU time spent in Memory Copy

amount of time doing memory copies between CPU memory and GPU device memory over PCIe. As the data presented in Figures 6.6 and 6.7 show, however, high *% GPU time doing memory copy* does not necessarily indicate high sensitivity to PCIe-access performance. For example, although the `Sort` benchmark spends over 2x as much time doing memory copies as the next-highest benchmark, `Sort` is not as sensitive to remote PCIe-access bandwidth as are `FFT`, `DGEMM`, `Reduction`, and `SpMV`. The performance of these benchmarks tracks closely with data-access performance as predicted using our performance models, indicating that bandwidth class-based scheduling may significantly improve the performance of nearly all of the `SHOC` algorithms.

## 6.3 Summary

The experimental results in this chapter provide a survey of CPU and GPU scientific algorithms and their sensitivity to data-access bandwidth. While metrics such as Last-Level Data Cache and *% of GPU time doing memory copy* are useful for some analysis of the impact of data-access bandwidth on a program, further profiling and analysis of individual algorithms is needed to more accurately predict the performance of a given algorithm in a given NUMA data-access scenario.

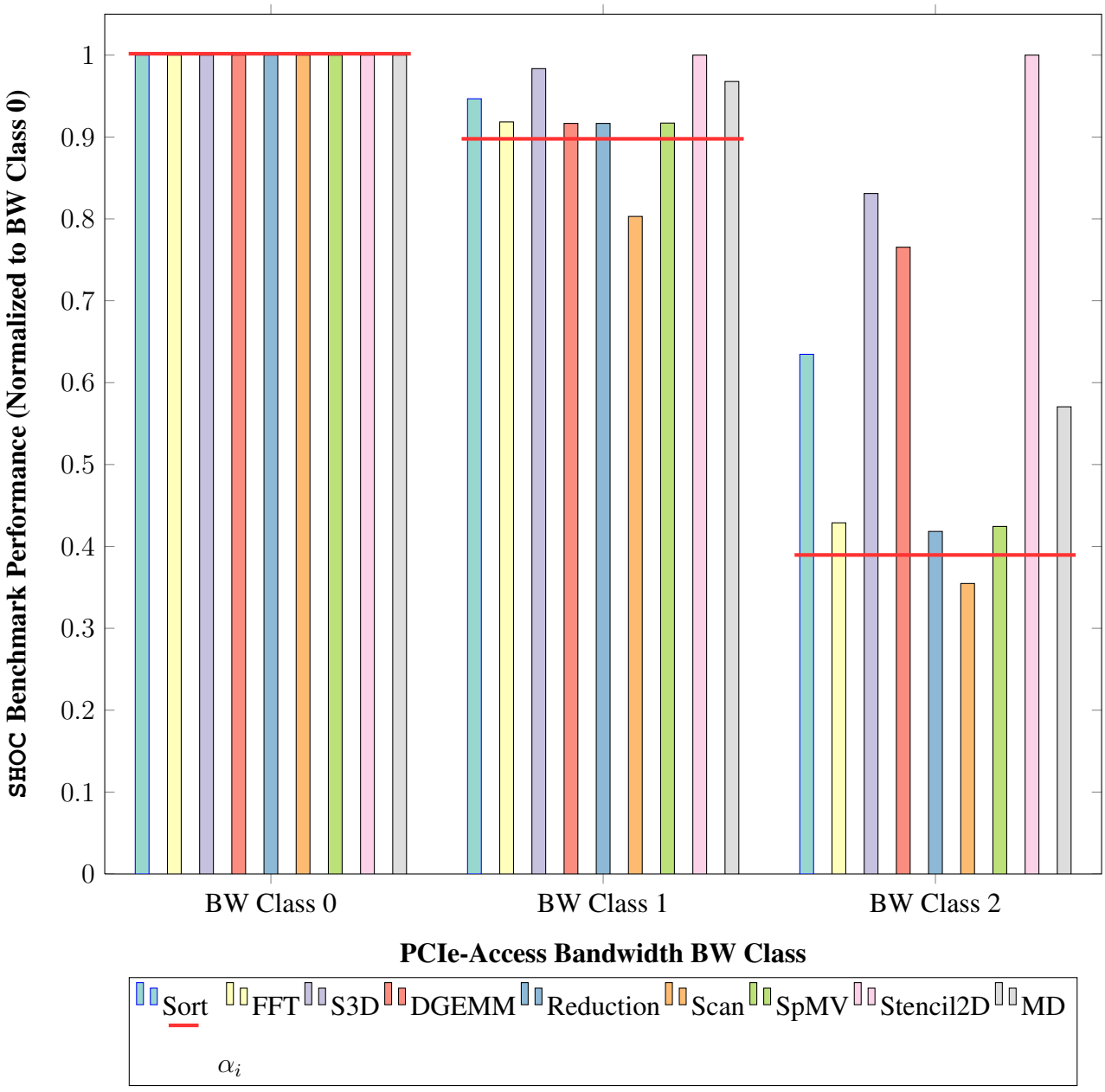



Figure 6.6: SHOC GPU Benchmark Results in the AMD 2P System.

Results within each class are presented in descending order by % GPU time doing memory copy.

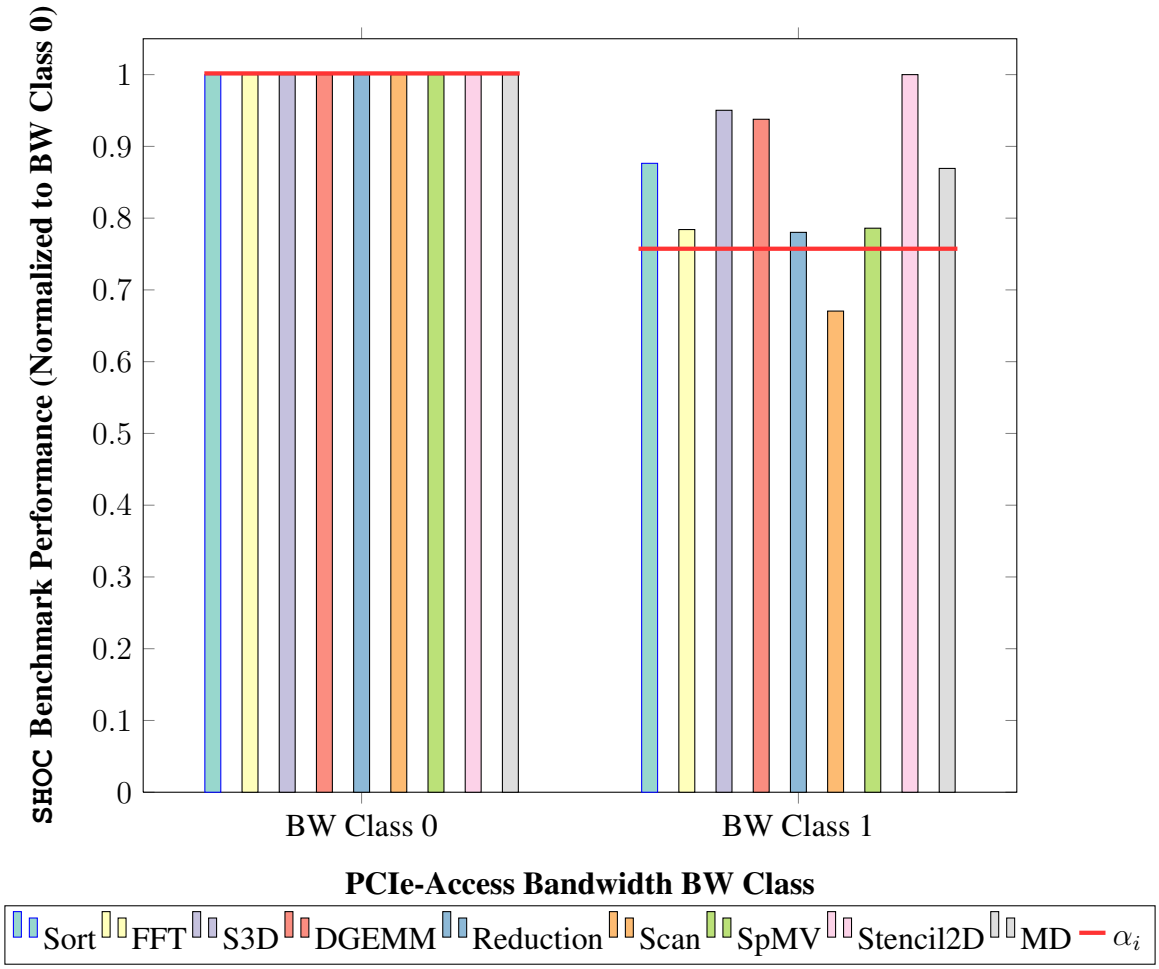



Figure 6.7: SHOC GPU Benchmark Results in the Intel 2P System.

Results within each class are presented in descending order by % GPU time doing memory copy.

# Chapter 7

# Conclusion

## 7.1 Summary

Heterogeneity within modern server architectures is a serious concern for system designers and application developers. As discussed in Chapters 1 and 2, Non-Uniform Memory Access (NUMA) architectures present the possibility for significantly reduced data-access bandwidth and increased data-access latency, thereby adversely affecting application execution performance.

To assist with the analysis and maximization of data-access performance in NUMA systems we presented in Chapter 3 a method for the characterization of data access bandwidth in modern NUMA architectures. The results of this characterization describe the fundamental data-access bandwidth characteristics of a given system; benchmark results were used to characterize and analyze modern AMD and Intel two-socket systems. Benchmark-based analysis of the data-access

bandwidth of these NUMA architectures provides the necessary data to create the data-access performance models discussed in Chapter 4. These performance models enable a concise comparison of architectures and give guidance to developers and system engineers regarding system configuration and application behavior. Our addition of these data-access bandwidth characterization and modeling tests to the Cbench testing framework in Chapter 5 allows for scalable, repeatable, and convenient NUMA data-access bandwidth testing of Linux systems.

With the necessary data-access profile information for an application, these performance models may even be used to assist with scheduling and application performance prediction. The preliminary results presented in Chapter 6 show sensitivity to data-access bandwidth for certain CPU and GPU programs. Programmers and system administrators may use these performance models and results to estimate the impact of certain NUMA designs on given algorithms and systems.

## 7.2 Future Work

The material presented in this thesis only begins the investigation into data-access performance modeling in NUMA systems. We anticipate further research in each of the following areas as we utilize and expand the characterization and performance modeling methods presented here.

**Intelligent Task Scheduling**

The data-access performance information presented in this thesis may be of value to a run-time scheduling system if sufficient information about a task's data-access requirements is known. Assuming knowledge of a task's data-access needs, the run-time scheduler may associate metadata with certain data structures so that an efficient determination of data-access performance may be made during the scheduler's decision-making process by examining the locations of data accessed by the task. Such an example of metadata attached to a data array is shown in Figure 7.1.

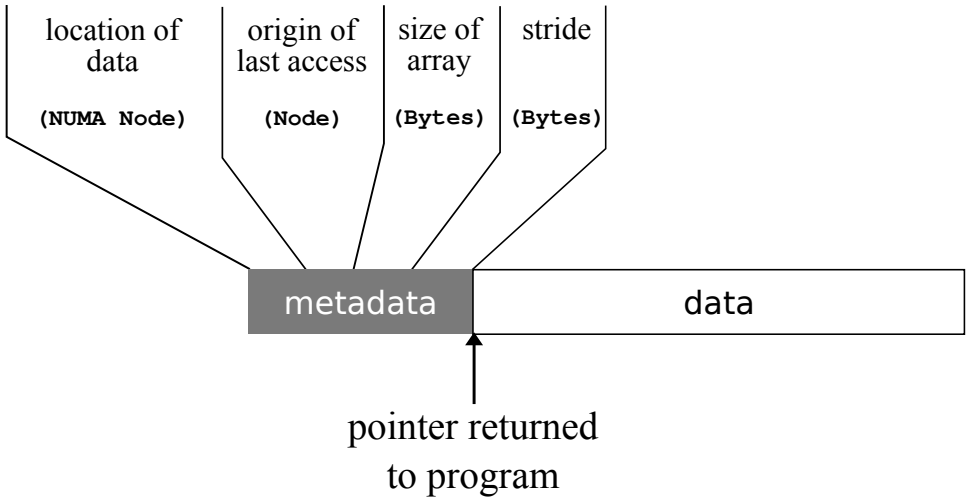



Figure 7.1: Example of a Dope Vector

Using the information encoded in the metadata header, the scheduler may consider which of the available cores or devices in a system has best data-access performance when accessing the task's required data. Such a run-time scheduler may be created using a current project such as Forest-GOMP [14] or built from scratch. Regardless of the implementation details, having the ability to schedule tasks in a manner that will maximize data-access performance is something that may be a

reasonable solution for handling the NUMA data-access performance trade-offs of modern server architectures.

**Data-Access-Performance-Based Application Performance Prediction**

Program performance prediction is a natural extension of the work presented in this thesis. With the necessary data-access information for a program and with the data-access bandwidth *and* latency performance for the system, program performance prediction for the data-access scenarios that are possible in a system will give programmers and system administrators the ability to analyze and predict program performance for a range of system configurations and program designs.

**System Analysis**

The characterization and modeling work presented in this thesis focused on two-socket NUMA systems from Intel and AMD. Much more heterogeneous systems are available today, and analyzing these systems should be an extension of the work presented here. Furthermore, analysis and modeling of cluster interconnect communication performance should be possible using the methods described in Chapters 3 and 4 by benchmarking the network access bandwidth of a system's interconnect adapter. This analysis and modeling would provide data-access performance information of use for distributed programs and how their communicating processes' data-access sensitivity in NUMA systems affects overall application performance. For example, by expanding the *dope vector* shown in Figure 7.1 to include the bandwidth of accessing data that is located in

a distributed system, an MPI task scheduler could more efficiently allocate processes and decrease remote and out-of-node memory accesses. Combined with MPI process migration, such as that presented in the SyMMer project [36], such data-access performance information could also allow for more efficient distribution of MPI processes within and across nodes.

NUMA data-access performance is a significant concern when programming or configuring modern server architectures. It is our hope that the material presented in this thesis is useful for assisting with the management of NUMA data-access performance and for creating future solutions to this problem.

# Appendix A

# AMD 4P Characterization Results

As an example of the heterogeneity present in modern NUMA systems, we present additional characterization data for a four-socket (4P) AMD system. The configuration of this system is shown in Table A.1 and its architecture is shown in Figure A.1.

Table A.1: Configuration of the AMD 4P Test System

| | |
|---|---|
| **CPU Model** | Magny-Cours 6168 |
| **CPU Cores / Mem. Nodes** | 48 / 8 |
| **Motherboard** | HP ProLiant DL585 G7 |
| **GPUs (#)** | Tesla C2050, Quadro 5000 |
| **Linux Kernel** | 2.6.35.10-74.fc14.x86_64 |

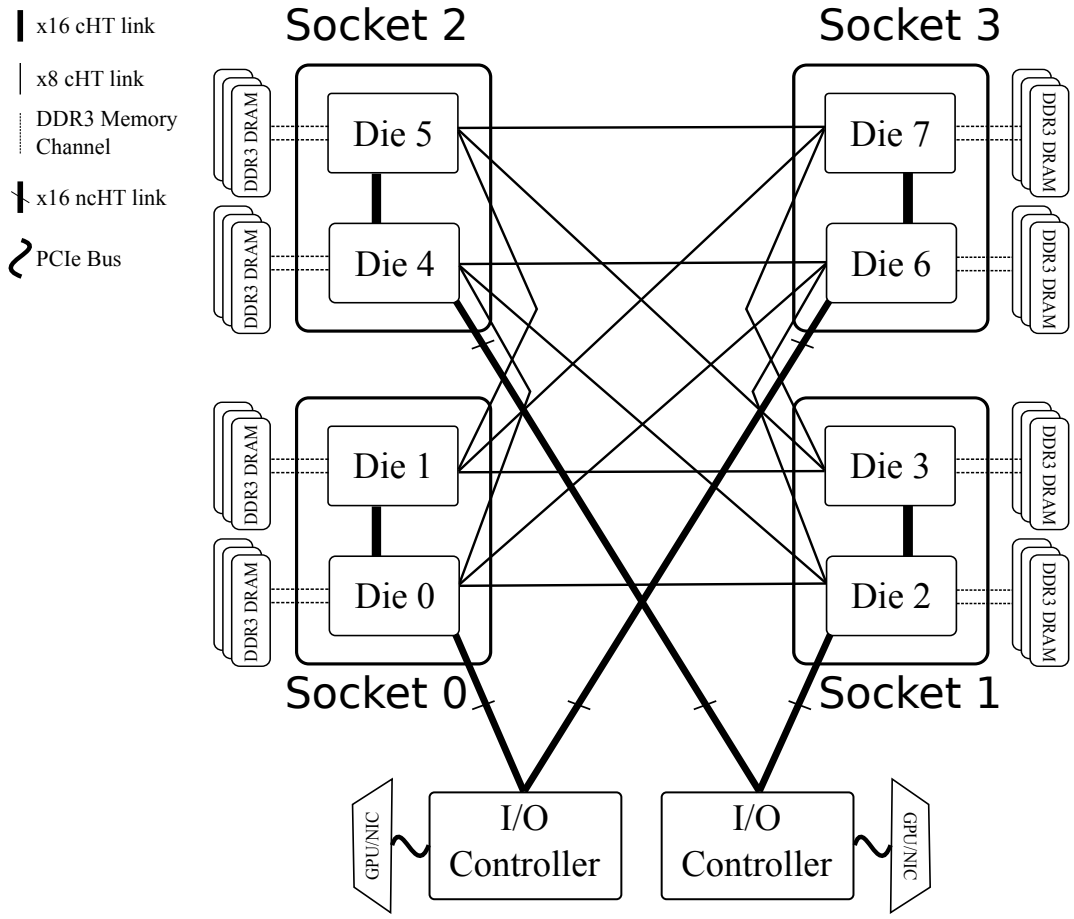



Figure A.1: AMD Magny-Cours 4P Architecture[1]

## A.1 Manual Benchmark Execution

We executed benchmark runs in an identical manner as that which we discussed in Chapter 3. The results for the manually-executed benchmark runs are shown in Table A.2.

This machine has eight memory nodes and the results show that it has four bandwidth classes. Most of the node-to-node combinations in the system fall into *Class 1* and *Class 2*, which is not surprising since the motherboard designers likely used x8 HT links between most memory nodes.

Table A.2: Microbenchmark Characterization Results for AMD 4P System

Classes of data-access bandwidth

| | AMD 4P | | |
|---|---|---|---|
| | **STREAM SINGLE-THREADED – GB/S (% OF CLASS 0)** | | |
| | **GNU** | **Intel** | **Open64** |
| **Class 0** | 5.8 (100%) | 7.2 (100%) | 8.0 (100%) |
| **Class 1** | 3.3 (58%) | 4.9 (68%) | 5.5 (69%) |
| **Class 2** | 2.7 (47%) | 3.7 (51%) | 4.0 (50%) |
| **Class 3** | 2.2 (37%) | 3.1 (44%) | 3.2 (40%) |
| | **SHOC BUSDOWNLOAD – GB/S (% OF CLASS 0)** | | |
| | **CUDA** | **OpenCL** | |
| **Class 0** | 2.9 (100%) | 2.9 (100%) | |
| **Class 1** | 2.3 (79%) | 2.3 (79%) | |
| | **SHOC BUSREADBACK – GB/S (% OF CLASS 0)** | | |
| **Class 0** | 3.3 (100%) | 3.3 (100%) | |
| **Class 1** | 2.3 (70%) | 2.3 (70%) | |
| **Class 2** | 1.9 (58%) | 1.9 (58%) | |

This machine also exhibits some asymmetry in data-access performance. For example, our tests showed that between `STREAM` running on Node 4 and accessing data on Node 7 achieved 4.2 GB/s of bandwidth, but with the benchmark running on Node 7 and accessing data on Node 4 only 3.3 GB/s of bandwidth was achieved. Although this asymmetry is worthy of investigation from an architectural standpoint, such asymmetry is handled by our bandwidth-class analysis since each type of access will fall into its own bandwidth class. All bandwidth class results are only valid from the perspective of running in a certain location within the system, and all data-access scenarios for each process location are assigned to bandwidth classes appropriately.

## A.2 Cbench Benchmark Execution

The results from the Cbench tool shows a slightly different breakdown of the results for this machine. Excerpts of the Cbench-produced report are shown in Tables A.3, A.4, A.5, and A.6.

The methods and tools we presented are applicable to any modern NUMA architecture. The data-access performance analysis provided by these tests provides insight into what a program may expect when executing in such a machine.

Table A.3: **AMD 4P Cbench Memory Data-Access Bandwidth Classes**

| $\Gamma = 17066MB/s$ | STREAM Triad |
|---|---|
| **Class 0** $(\alpha_{m_0})$ | $\leq$ 8778 MB/s (0.5144) |
| **Class 1** $(\alpha_{m_1})$ | $\leq$ 8109 MB/s (0.4752) |
| **Class 2** $(\alpha_{m_2})$ | $\leq$ 5583 MB/s (0.3271) |
| **Class 3** $(\alpha_{m_2})$ | $\leq$ 4002 MB/s (0.2345) |

Table A.4: **AMD 4P Cbench NUMA STREAM Bandwidth Test Results** — the best value in each column is highlighted

| **STREAM Triad (MB/s)** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Node 0 to Node 0 | Node 0 to Node 1 | Node 0 to Node 2 | Node 0 to Node 3 | Node 0 to Node 4 | Node 0 to Node 5 | Node 0 to Node 6 | Node 0 to Node 7 | ··· |
| stream-big-c-intel-12.0.2.137 | 7676 | 5161 | 3035 | 2815 | 3126 | 3116 | 3909 | 3303 | ··· |
| stream-big-c-open64-5.0 | 7767 | 4880 | 3049 | 3024 | 3210 | 3123 | 3647 | 3535 | ··· |
| stream-big-f-gcc-4.4.6 | 5785 | 3148 | 2468 | 2005 | 2127 | 2066 | 2723 | 2287 | ··· |
| stream-big-f-intel-12.0.2.137 | 5599 | 3337 | 2339 | 2001 | 2080 | 2020 | 2540 | 2487 | ··· |
| stream-big-f-open64-5.0 | 7569 | 5410 | 3048 | 2925 | 3204 | 3146 | 3652 | 3485 | ··· |
| stream-c-intel-12.0.2.137 | 6630 | 5199 | 3039 | 2842 | 3122 | 3039 | 3454 | 3326 | ··· |
| stream-c-open64-5.0 | 7790 | 5549 | 3060 | 2923 | 3240 | 3176 | 3678 | 3526 | ··· |
| stream-f-gcc-4.4.6 | 5043 | 3811 | 2469 | 2090 | 2185 | 2128 | 2470 | 2406 | ··· |
| stream-f-intel-12.0.2.137 | 6991 | 5257 | 3097 | 2969 | 3142 | 3068 | 3480 | 3624 | ··· |
| stream-f-open64-5.0 | 6462 | 5507 | 3097 | 3016 | 3155 | 3165 | 3634 | 3761 | ··· |
| stream-intel2-c-intel-12.0.2.137 | 7686 | 4467 | 2995 | 2758 | 3115 | 3093 | 3407 | 3684 | ··· |
| stream-intel2-f-intel-12.0.2.137 | 7796 | 5097 | 3054 | 2791 | 3124 | 3041 | 3908 | 3335 | ··· |
| stream-intel3-f-intel-12.0.2.137 | 7695 | 4988 | 3056 | 2789 | 3133 | 3128 | 3916 | 3509 | ··· |

Table A.5: **AMD 4P Cbench GPU Data-Access Bandwidth Classes**

| $\Phi = 12.8GB/s$ | BusSpeedDownload | BusSpeedReadback |
|---|---|---|
| **Class 0** $(\alpha_{p_0})$ | $\leq$ 2.89 GB/s (0.2260) | $\leq$ 3.35 GB/s (0.2621) |
| **Class 1** $(\alpha_{p_1})$ | $\leq$ 2.88 GB/s (0.2252) | $\leq$ 3.34 GB/s (0.2609) |
| **Class 2** $(\alpha_{p_2})$ | $\leq$ 2.31 GB/s (0.1808) | $\leq$ 2.29 GB/s (0.1792) |

Table A.6: **AMD 4P Cbench NUMA `SHOC` GPU Data-Access Bandwidth Test Results** — the best value in each column is highlighted

| **`BusSpeedDownload (GB/s)`** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **CUDA** | | | | | | | | | |
| | Node 0 to Node 0 | Node 0 to Node 1 | Node 0 to Node 2 | Node 0 to Node 3 | Node 0 to Node 4 | Node 0 to Node 5 | Node 0 to Node 6 | Node 0 to Node 7 | ··· |
| Device 0 | 2.8824 | 2.8823 | 2.8823 | 2.3139 | 2.3138 | 2.3104 | 2.8821 | 2.8822 | ··· |
| Device 1 | 2.8925 | 2.3300 | 1.9386 | 1.9354 | 2.8925 | 2.8925 | 2.3232 | 2.3322 | ··· |

| **`BusSpeedReadback (GB/s)`** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **CUDA** | | | | | | | | | |
| | Node 0 to Node 0 | Node 0 to Node 1 | Node 0 to Node 2 | Node 0 to Node 3 | Node 0 to Node 4 | Node 0 to Node 5 | Node 0 to Node 6 | Node 0 to Node 7 | ··· |
| Device 0 | 3.3387 | 3.3388 | 2.2944 | 1.9148 | 3.3388 | 3.3389 | 3.3388 | 3.3387 | ··· |
| Device 1 | 2.3064 | 1.9253 | 1.8993 | 3.3547 | 3.3547 | 3.3547 | 3.3546 | 3.3547 | ··· |

# Bibliography

[1] Bios and kernel developer's guide (bkdg) for amd family 10h processors. Technical Report 3.48, Advanced Micro Devices, Inc., April 2010.

[2] Using the x86 open64 compiler suite. *URL: http://developer.amd.com/tools/open64/onlinehelp/pages/x86_open64_help.htm*, 2010.

[3] Cbench - scalable cluster benchmarking. `http://sourceforge.net/projects/cbench/`, August 2011.

[4] Intel 64 and ia-32 architectures software developer's manual volume 3b: System programming guide, part 2. Technical report, Intel Corporation, 2011.

[5] M. Anderson. Better benchmarking for supercomputers. *Spectrum, IEEE*, 48(1):12 –14, 2011.

[6] Joseph Antony, Pete Janes, and Alistair Rendell. Exploring thread and memory placement on numa architectures: Solaris and linux, ultrasparc/fireplane and opteron/hypertransport. In Yves Robert, Manish Parashar, Ramamurthy Badrinath, and Viktor Prasanna, editors, *High*

*Performance Computing - HiPC 2006*, volume 4297 of *Lecture Notes in Computer Science*, pages 338–352. Springer Berlin / Heidelberg, 2006.

[7] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 863–874, Berlin, Heidelberg, 2009. Springer-Verlag.

[8] David A. Bader, Jonathan Berry, Simon Kahan, Richard Murphy, E. Jason Riedy, and Jeremiah Willcock. Graph 500 benchmark 1 ("search"). Technical report, Graph500.org, October 2010.

[9] D. Bailey, T. Harris, W. Saphir, R. Van der Wijngaart, A. Woo, and M. Yarrow. The nas parallel benchmarks 2.0. Technical report, NASA Ames Research Center, 1995.

[10] Christopher L. Barrett, Keith R. Bisset, Stephen G. Eubank, Xizhou Feng, and Madhav V. Marathe. Episimdemics: an efficient algorithm for simulating the spread of infectious disease over large realistic social networks. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 37:1–37:12, Piscataway, NJ, USA, 2008. IEEE Press.

[11] W. Bolosky, R. Fitzgerald, and M. Scott. Simple but effective techniques for numa memory management. In *Proceedings of the twelfth ACM symposium on Operating systems principles*, SOSP '89, pages 19–31, New York, NY, USA, 1989. ACM.

[12] William J. Bolosky, Michael L. Scott, Robert P. Fitzgerald, Robert J. Fowler, and Alan L. Cox. Numa policies and their relation to memory architecture. In *Proceedings of the fourth*

*international conference on Architectural support for programming languages and operating systems*, ASPLOS-IV, pages 212–221, New York, NY, USA, 1991. ACM.

[13] 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*, pages 180 –186, 2010.

[14] Franois Broquedis, Nathalie Furmento, Brice Goglin, Pierre-Andr Wacrenier, and Raymond Namyst. Forestgomp: An efficient openmp environment for numa architectures. *International Journal of Parallel Programming*, 38:418–439, 2010.

[15] K.W. Cameron and Xian-He Sun. Quantifying locality effect in data access delay: memory logp. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, page 8 pp., 2003.

[16] Tracy Carver. Magny-cours and direct connect architecture 2.0. *URL: http://developer.amd.com/documentation/articles/pages/magny-cours-direct-connect-architecture-2.0.aspx*, March 2010.

[17] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. Logp: towards a realistic model of parallel computation. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '93, pages 1–12, New York, NY, USA, 1993. ACM.

[18] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. The scalable heterogeneous computing (shoc) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 63–74, New York, NY, USA, 2010. ACM.

[19] M. Forsell. A pram-numa model of computation for addressing low-tlp workloads. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1 –8, 2010.

[20] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, STOC '78, pages 114–118, New York, NY, USA, 1978. ACM.

[21] Brice Goglin and Nathalie Furmento. Enabling high-performance memory migration for multithreaded applications on linux. *Parallel and Distributed Processing Symposium, International*, 0:1–9, 2009.

[22] Andreas Kleen. A numa api for linux. Technical report, SUSE Labs, April 2005.

[23] Stefan Lankes, Boris Bierbaum, and Thomas Bemmerl. Affinity-on-next-touch: an extension to the linux kernel for numa architectures. In *Proceedings of the 8th international conference on Parallel processing and applied mathematics: Part I*, PPAM'09, pages 576–585, Berlin, Heidelberg, 2010. Springer-Verlag.

[24] David Levinthal. Performance analysis guide for intel core i7 processor and intel xeon 5500 processors. Technical report, Intel Corporation, 2009.

[25] S. Lloyd. Least squares quantization in pcm. *Information Theory, IEEE Transactions on*, 28(2):129 – 137, mar 1982.

[26] D. Luebke. Cuda: Scalable parallel programming for high-performance scientific computing. In *Biomedical Imaging: From Nano to Macro, 2008. ISBI 2008. 5th IEEE International Symposium on*, pages 836 –838, may 2008.

[27] John D. McCalpin. Stream: Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, Charlottesville, Virginia, 1991-2007. A continually updated technical report. http://www.cs.virginia.edu/stream/.

[28] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[29] Larry McVoy and Carl Staelin. lmbench: portable tools for performance analysis. In *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, pages 23–23, Berkeley, CA, USA, 1996. USENIX Association.

[30] Richard Murphy. On the effects of memory latency and bandwidth on supercomputer application performance. *IEEE Workload Characterization Symposium*, 0:35–43, 2007.

[31] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 89(2):44 – 66, 2003. RV '2003, Run-time Verification (Satellite Workshop of CAV '03).

[32] Jeffry Ogden. Cbench: A software toolkit for testing, benchmarking, and qualifying hptc linux clusters. Technical report, Sandia National Laboratories, Accessed August, 2011.

[33] Christiane Pousa Ribeiro and Jean-François Méhaut. Minas: Memory Affinity Management Framework. Research Report RR-7051, INRIA, 2009.

[34] C.P. Ribeiro, J.-F. Mehaut, A. Carissimi, M. Castro, and L.G. Fernandes. Memory affinity for hierarchical shared memory multiprocessors. In *Computer Architecture and High Performance Computing, 2009. SBAC-PAD '09. 21st International Symposium on*, pages 59 –66, 2009.

[35] Martin Schmollinger and Michael Kaufmann. $\kappa$numa: A model for clusters of smp-machines. In Roman Wyrzykowski, Jack Dongarra, Marcin Paprzycki, and Jerzy Wasniewski, editors, *Parallel Processing and Applied Mathematics*, volume 2328 of *Lecture Notes in Computer Science*, pages 42–50. Springer Berlin / Heidelberg, 2006.

[36] T. Scogland, P. Balaji, W. Feng, and G. Narayanaswamy. Asymmetric interactions in symmetric multi-core systems: Analysis, enhancements and evaluation. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1 –12, nov. 2008.

[37] John E. Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science and Engineering*, 12:66–73, 2010.