EMPIRICAL PERFORMANCE ANALYSIS OF HPC APPLICATIONS WITH PORTABLE HARDWARE COUNTER METRICS

by

BRIAN J GRAVELLE

A DISSERTATION

Presented to the Department of Computer and Information Sciences
and the Division of Graduate Studies of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

June 2022

DISSERTATION APPROVAL PAGE

Student: Brian J Gravelle

Title: Empirical Performance Analysis of HPC Applications with Portable Hardware Counter Metrics

This dissertation has been accepted and approved in partial fulfillment of the requirements for the Doctor of Philosophy degree in the Department of Computer and Information Sciences by:

Boyana Norris	Chair
Allen Malony	Core Member
Hank Childs	Core Member
Diego Melgar	Institutional Representative

and

Krista Chronister	Vice Provost for Graduate Studies

Original approval signatures are on file with the University of Oregon Division of Graduate Studies.

Degree awarded June 2022

DISSERTATION ABSTRACT

Brian J Gravelle

Doctor of Philosophy

Department of Computer and Information Sciences

June 2022

Title: Empirical Performance Analysis of HPC Applications with Portable Hardware Counter Metrics

In this dissertation, we demonstrate that it is possible to develop methods of empirical hardware-counter-based performance analysis for scientific applications running on diverse CPUs. Although counters have been used in performance analysis for over 30 years, the methods remain limited to particular vendors or generations of CPUs. Our hypothesis is that counter-based measurements could be developed to provide consistent performance information on diverse CPUs. We prove the hypothesis correct by demonstrating one such set of metrics.

We begin with an introduction and background discussing empirical performance analysis on CPUs. The background includes the Roofline Performance Model which is widely used to visualize the performance of scientific applications relative to the potential system performance. This model uses metrics that are portable to different CPU architectures, making it a useful starting point for efforts to develop portable hardware counter metrics. We contribute to existing roofline literature by presenting a method using counters to measure the required application data on two CPUs and by presenting benchmarks to produce the Roofline Model of the CPU. These contributions are complementary since the benchmarks can be used to validate the hardware counters used to measure the application data.

We present a set of performance metrics derived from Hardware Performance Monitors that we have been able to replicate on CPUs from two vendors. We developed these metrics to focus on information that can inform developers about the performance of algorithms and data structures in applications. This method contrasts with other methods which are aimed at microarchitectural features and allows users to understand application performance from the same perspective on multiple CPUs.

We use a series of case studies to explore the usefulness of our metrics and to validate that the measured values provide the expected information. The first set of studies examines benchmarks and mini-applications with a variety of performance. Finally, we study the performance of several versions of a scientific application using the Roofline Model and the new metrics. These case studies show that our performance metrics can provide performance information on two CPUs, proving our hypothesis by example.

This dissertation includes previously published co-author material.

CURRICULUM VITAE

NAME OF AUTHOR: Brian J Gravelle

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene, OR, USA
Gonzaga University, Spokane, WA, USA

DEGREES AWARDED:

Doctor of Philosophy, Computer Science, 2022, University of Oregon
Bachelor of Science, Computer Engineering, 2015, Gonzaga University

AREAS OF SPECIAL INTEREST:

High Performance Computing
Performance Analysis
Computer Architecture

PROFESSIONAL EXPERIENCE:

Graduate Research Assistant, Los Alamos National Laboratory, 2018-2022
Graduate Employee (Research), Dept. of Computer and Information Science, University of Oregon, 2017-2022
Graduate Employee (Instructor of Record for Computer Architecture), Dept. of Computer and Information Science, University of Oregon, Spring 2021
Graduate Employee (Teaching Assistant for undergraduate courses), Dept. of Computer and Information Science, University of Oregon, 2015-2017
Undergraduate Research Assistant, Gonzaga University, 2013-2015

GRANTS, AWARDS AND HONORS:

Moursund Graduate Teaching Award, University of Oregon, 2021

PUBLICATIONS:

Gravelle, B. J., & Nystrom, W. D., Yokelson, D., & Norris, B. (2021). Enabling cache-aware roofline analysis with portable hardware counter metrics. *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems.* Springer.

Graziano, V., & Nystrom, W. D., & Pritchard, H., & Smith, B., & Gravelle, B. J. (2021). Optimizing a 3d multi-physics continuum mechanics code for the hpe apollo 80 system. *Cray User Group (CUG).*

Gravelle, B. J., & Norris, B. (2019). Performance Analysis of Compressed Batch Matrix Operations on Small Matrices. *The 6th Special Session on High Performance Computing Benchmarking and Optimization.*

Gravelle, B. (2019). "Understanding the Performance of HPC Applications." Technical Report for Departmental Area Exam.

Gravelle, B. (2017). "Performance and Power Impacts of Autotuning of Kalman Filters for Disparate Environments." Technical Report for Departmental Directed Research Project.

ACKNOWLEDGEMENTS

I would like to begin by thanking those organizations that financially supported the research presented in this dissertation. This work was supported by the U.S. Department of Energy through the Los Alamos National Laboratory (LANL). Los Alamos National Laboratory is operated by Triad National Security, LLC, for the National Nuclear Security Administration of U.S. Department of Energy (Contract No. 89233218CNA000001). The work would have been impossible without the Darwin testbed at LANL which is funded by the Computational Systems and Software Environments subprogram of LANL's Advanced Simulation and Computing program (NNSA/DOE). Additional support came from the U.S. National Science Foundation, under grants PHY1520969, PHY1521042, PHY1520942 and PHY1624356, and under Cooperative Agreement OAC1836650, and by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program.

While completing this research I worked primarily in Los Alamos, New Mexico and Eugene, Oregon. The following statements from LANL and the University of Oregon, respectively, recognize the importance of the land to the indigenous populations.

LANL and the communities of Los Alamos and White Rock are located on Indigenous lands, ancestral to the Tanoan and Keresan speaking peoples of northern New Mexico. This landscape, being located in an area of migration and trade along the Rio Grande and Jemez Mountains, also retains ancestral significance to Athabaskan speaking peoples – including the Dinétah and Apache

– the Zuni and the Hopi. We value, respect and honor the traditional peoples and landscapes which comprise the Laboratory, and recognize the multi-generational support that Native American peoples and communities have provided to LANL and the Department of Energy complex across the United States. Our directive is to be responsible stewards of this landscape; to practice environmental responsibility to meet the legal and ethical requirements needed for long-term preservation of these indigenous lands.

The University of Oregon is located on Kalapuya Ilihi, the traditional indigenous homeland of the Kalapuya people. Following treaties between 1851 and 1855, Kalapuya people were dispossessed of their indigenous homeland by the United States government and forcibly removed to the Coast Reservation in Western Oregon. Today, descendants are citizens of the Confederated Tribes of Grand Ronde Community of Oregon and the Confederated Tribes of Siletz Indians of Oregon, and continue to make important contributions in their communities, at UO, and across the land we now refer to as Oregon.

Throughout my time as a graduate student, Boyana Norris has advised, guided, and occasionally cajoled me into completing my research. Her technical and personal advice helped me through the more difficult periods, while her persistent joy and curiosity have been a source of inspiration. I have no doubt that the successes of this dissertation are primarily due to her guiding hand and its limitations due to my obstinacy.

Similarly, Dave Nystrom and been an invaluable mentor in matters of my career and in my research. I have attempted to follow his example of dedication and patience, and I hope that eventually I can pass some of his wisdom to other interns.

At both UO and LANL, I had many colleagues who supported this work. Sam Pollard, Ph.D., Kewen Meng, Ph.D., and Curtis Dlouhy, Ph.D. were colleague and friends throughout my time in Oregon. Cindy Martin, Andrew Montoya, Julie Wiens, Howard Pritchard, and numerous others made my time at LANL enjoyable and productive.

Finally I would like to thank my family for their love and support. They are a constant source of inspiration and encouragement.

Dedicated to Denise and Delia in gratitude for their utmost patience. . . .

TABLE OF CONTENTS

Chapter Page

LIST OF FIGURES

Figure Page

LIST OF TABLES

CHAPTER I

INTRODUCTION

The research presented in this dissertation explores the potential of using Hardware Performance Monitors to analyze application performance consistently across multiple types of CPUs. Hardware Performance Monitors (also called hardware counters) have been used for many years to study the performance of software and systems. The counters offer a close look at how software uses the microarchitecture of a processing unit, which enables users to gain detailed insight into the performance of an application running on a particular system. Unfortunately, the methods based on hardware counters often restrict users to specific systems and only provide limited information about the algorithm targeted by the analysis. This dissertation explores the hypothesis that hardware counter metrics can provide actionable performance information to users on a diverse set of CPU types. We prove the accuracy of our hypothesis by developing a set of such hardware counter metrics, validating those measurements, and applying them to a series of example applications on two CPU types.

This hypothesis is transformed into our central research question: Can hardware counters on different CPU types produce the same set of performance metrics that help the user understand the performance of scientific applications?

We answer this question in four stages

– Can an existing CPU-agnostic performance analysis technique be used in conjunction with hardware counter metrics derived from diverse CPU types?

– Can additional metrics be collected to add further information to the performance analysis?

    – Does this provide useful info in a variety of performance cases?

    – Does this scale to a full application?

For this research, we restrict the scope of hardware and software. Any area of computing can make use of performance analysis to get the most use out of its systems, but we focus on scientific computing which has a long history of using some of the largest Supercomputers available. In general, these applications rely on floating-point arithmetic and have a computational rate defined as the number of floating-point operations executed per second. We limit ourselves to these applications because of the consistency of this definition of performance and the long history of performance analysis in the scientific computing community. Each year a larger portion of scientific computing moves to processing on Graphics Processing Units (GPUs) instead of the more generalizable Central Processing Units (CPUs). We limit the scope of the work to CPUs because there is a sufficient variety of microarchitectures to impede the portability of metrics, justifying the need for application-oriented metrics. In future work, we can extend the scope to GPUs and other accelerators which would add variety to the microarchitectures.

We describe the current state of related research in Chapter II. This chapter discusses historical and modern methods of using hardware counters for performance analysis, with particular focus on the limitations of these methods that our research aims to resolve. In addition, we discuss the Roofline Method of modeling processors and performance. We use hardware counters to gather the metrics required for this method and to validate the benchmarks necessary to produce the architectural models. Our research builds on the work discussed in this chapter.

Throughout the dissertation, we use a consistent set of tools, systems, and benchmarks. These pieces of our methodology are described in Chapter III. In this chapter, we describe the set of tools we use for collecting and processing hardware counter data although users may have other methods that they prefer. Additionally, we describe the two CPU types we use throughout this work and the other aspects of the systems including compilers and operating systems. Finally, we describe several benchmarks that we use to evaluate and then showcase hardware counter metrics. The infrastructure discussed in Chapter III is used repeatedly throughout the dissertation, but additional tools and applications are introduced as needed in later chapters.

After establishing this context for the research, we present our contributions. The central aim of the contributions is to assess the feasibility of using consistent hardware counter metrics across different microarchitectures to provide users with empirical application-oriented performance information. We demonstrate our methods of performance analysis by first introducing a novel method of using hardware counters to conduct Roofline-based performance analysis. Then we develop a larger range of empirical hardware counter metrics that are applicable across different microarchitectures along with a series of analyses that showcase the use of the metrics. Finally, we apply the combined methods to a scientific application, to gain insight into the change in performance over several versions.

In Chapter IV, our contributions begin by building on performance analysis using the Roofline Model. We chose this starting point because the roofline provides a simple model of a generic CPU that is applicable across all modern CPU types. It connects this model to target applications by plotting the application point on the model. This combination provides context for the performance of the application and indicates potential limiting hardware features. We introduce a set of benchmarks that are useful for building empirical Roofline Models. Additionally, we use these benchmarks to validate a set of hardware counter metrics which can be used to plot the application point on the Roofline Model. The result is a new method to empirically use Roofline Modelling to understand the performance of an application on different CPU types.

Starting with roofline measurements, we take a similar approach to defining a set of performance metrics in Chapter V. The Roofline Model uses two metrics that apply to any CPU, and we decided to search for other microarchitecture-agnostic metrics that hardware counters could measure. We sought to define metrics that provided information 1) about common hardware features, 2) related to the algorithm or data structures, and 3) can be measured with hardware counters on our two target systems. Our efforts resulted in a set of metrics that users can apply to their application without detailed knowledge of the microarchitecture. Some of these metrics are novel and some have been used by other researchers. The focus of the metrics is on how data is moved in the system and how computation is performed.

Based on this set of metrics, we conduct several analyses of benchmarks and mini-applications in Chapter VI. Each of the benchmarks and mini-applications we consider is related to one or scientific applications, so the performance features will be relevant to real-world users. We collect data on the two systems discussed in Chapter III and for multiple versions of each target mini-application. This process allows us to demonstrate the applicability of the metrics to a range of performance situations.

Finally, in Chapter VII, we present a case study of analysis of Pagosa, a fluid dynamics application that is in widespread use. We use both the Roofline Model analysis and the hardware counter metrics presented in previous chapters to study the performance of a series of optimizations to key computational kernels in the application. This study addresses why certain optimization efforts worked for the kernels, demonstrating the usefulness in aiding developers' evaluation and optimization of application performance.

The methods presented in this dissertation show that hardware counters can be used to conduct performance analysis focused on the application's features rather than the specifics of a microarchitecture. We relate our method to the commonly used performance analysis method based on the Roofline Model (Chapter IV) and extend it to new areas with the introduction of portable hardware counter metrics (Chapter V). Through a series of case studies we show that the method is scalable to small kernels (Chapter VI) and full applications (Chapter VII). In total, this research demonstrates that a set of hardware counter-based metrics can provide actionable information to the user across a set of CPUs with different ISAs, microarchitectures, and hardware counter interfaces.

CHAPTER II

BACKGROUND

The research presented in this dissertation builds on the long history of node-level performance analysis of scientific applications. Scientific applications are a vital part of research for scientists in many fields who rely on simulation or data processing to conduct their research [2] [3]. Performance analysis and optimization of these applications is a necessary part of the process since time on computing clusters can be expensive or a bottleneck to other research. Of the many aspects of performance in a scientific application, we chose to focus on the performance of computational kernels running on CPUs. At this level, performance analysis considers the use of the processor pipeline, arithmetic units, caches, main memory, and other "node-level" aspects of the system.

There is a great deal of background information available to those interested in High Performance Computing (HPC) in general. We recommend Eijkhout's open-source textbook [4] for an introduction to many of the intersecting fields that an HPC user may encounter. Robey and Zamora [5] is an up-to-date text that covers the systems and software that are found in modern HPC systems. Jeffers and Reinders [6] bring together a set of parallel computing examples with many details and implementations to help readers understand parallel computation. For those interested in the architectural aspects of HPC, Hennessy and Patterson have written six editions of textbooks guiding readers through computer architecture [7]. Also, Dean et al. [8] provides insight on recent trends in computer architecture and the future of the field. Similarly, Jalby et al. give their thoughts on the future of HPC as we enter the Exascale era [9]. There are doubtless many more authors whose work readers may find useful, but we hope these will provide a starting point for the interested reader.

We do not expect that the reader will understand the full scope of HPC research (the author certainly falls short of this goal), but the reader may find a review of performance analysis useful. Performance analysis can be based on data from empirical studies, information gathered statically through the examination of algorithms and hardware, or the results of simulation. Empirical analysis is much like other sciences; measurements are taken and results are examined leading to new hypotheses and experiments. Static analysis or simulation provides information that may be difficult to measure empirically, but these methods are also limited in the amount of detail they can consider. Application developers use the information gathered from any of these methods to understand the performance of the application.

We focus on empirical performance analysis because our background research led us to the conclusion that hardware counter information was not used to its full potential. This conclusion encouraged us to explore our primary hypothesis that hardware counter metrics can provide actionable performance information to users on a diverse set of CPU types.

Performance analysts who use empirical analysis can use a range of metrics from simple timers to tools that record and trace each memory address that an application uses. The methods we use in the following chapters use hardware counters to collect data on events that occur in the CPU or memory subsystem while an application is running. We provide background on this method of analysis in Section 2.1. The Roofline Model is a model of a CPU that can be used in conjunction with measurement or static program analysis to provide the user with an understanding of their application. In Section 2.2, we describe the model and the measurements that are necessary to build and use the model.

2.1 Performance Measurement and Analysis of Applications

Measuring the performance of an application can be as simple as using a clock to check how long an application takes to run or involve complex toolchains to gather detailed information on the hardware. Any measurement of an application running on the target system is empirical analysis.

Modern programming languages offer interfaces to the operating system timers which can be placed at key points in an application. By measuring key points throughout the application, the user can identify which areas are most important. Amdahl's Law [10] tells us that we need to focus our optimization efforts on the most time-consuming portions of the application, so this "hotspots" analysis is a good first step.

The metrics we introduce in this research are all based on hardware counters. Physically, these are registers built into CPUs and GPUs that record events that occur in the processor. Conveniently, libraries such as PAPI [1] and perf [11] provide human-readable names and high-level language interfaces for the counters. The PAPI authors regularly update these interfaces for new CPUs.

The data collection can occur in one of two ways. Sampling is one option, which periodically interrupts the program to record the counter values. This method is less intrusive to the application but also less accurate. It is used by TAU [12, 13] and HPCTookit [14, 15]. Alternatively, the measurements can be inserted into an application at key points. This instrumentation can be done manually as in Caliper [16] and Likwid [17], or a tool can perform automatic instrumentation which is another option in TAU [12, 13]. We find that sampling provides a good overview of the program which can be used to choose particular areas of focus for the instrumentation.

In addition to the open-source tools, many vendors provide tools for users to measure the performance on their systems. Intel offers VTune, ARM has Forge, and HPE provides Craypat. These tools are professionally maintained with nice graphic user interfaces but can limit the user to particular products.

Hardware counters provide unique information about the inner workings of the processor, but the uniqueness causes challenges for validation. Since there is no way to check many of the values measured by counters, validation is a challenge. Carefully made benchmarks [18, 19, 20, 21] are the best way to validate the values provided by counters.

At first hardware, counters were primarily used to characterize the workload of systems. In [22], the authors use counters to identify which instruction types have the largest impact on the performance of a mainframe system. Williams et al. [23] compare two identical systems with different workloads to help guide future system design. Finally, Cvetanovic and Bhandarkar [24] compare technical and commercial workloads to gain insight into the similarities and differences between the two workload types.

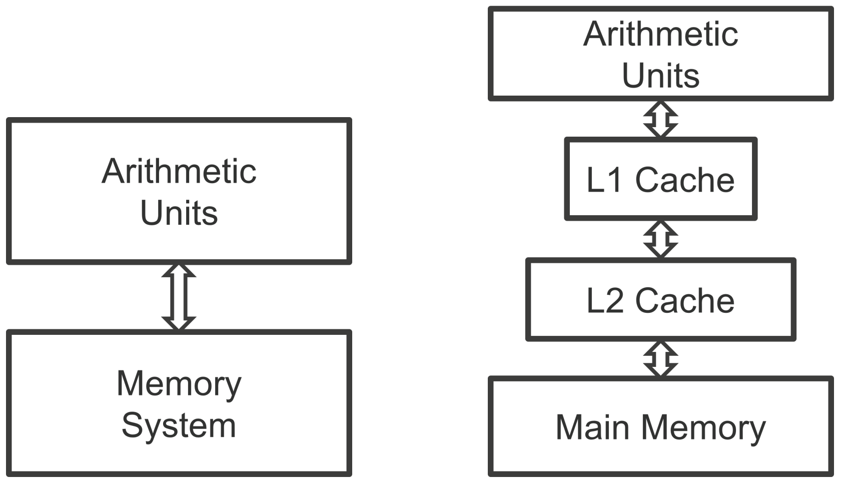
In [25], Zagha et al. conduct analysis and optimization of computational kernels based on the results of hardware counter data. The authors describe the implementation of the hardware required for the counters along with the software that enables the application developer to make use of them. Additionally, the authors use several cache metrics similar to our own to identify performance bottlenecks in their target kernels.

When studying the performance of a particular kernel, the most common method of hardware counter analysis is to identify a base measurement of time or work and then count events that fill up the base value to establish what hardware features are most important to the performance of an application. Eyerman et al. [26, 27] use cycles per instruction (CPI). The authors effectively visualize where the potential CPI is lost which makes the method useful to developers. Similarly, Nowak et al. present the HCA [28], which uses a tree structure to visualize the allocation of CPU cycles guiding users to performance bottlenecks. Yasin introduces the Top-Down Method of Analysis in [29], which uses slots (available spots for instructions to retire in each cycle) as the base metric. The TMA has been extended several times [30, 31] and is widely used on Intel systems, and may soon be used on ARM [32] and AMD [33], as well. These methods are thorough, but require the user to have detailed knowledge of the CPU architecture to be useful. Developing this knowledge is an added burden for many scientific software developers who need to focus their attention on their discipline.

Several authors have worked on identifying particular metrics for performance analysis [34, 35]. Their methods have inspired some of ours, in particular identifying ways to consistently validate counters across systems. We build on this work by adding new metrics beyond Bandwidth and data movement.

2.2 Arithmetic Intensity and the Roofline Model

One of the simplest models of computing is the Von Neumann architecture (Figure 1a). This model presents a computer as having two primary functions; data movement and arithmetic. The model is simple but holds true even for modern computers.



(a) The basic Von Neumann architecure.

(b) A model memory hierarchy with caches.

Figure 1. Comparison between the basic Von Neumann architecture and a more modern model with multiple levels of memory caching.

This model can be used to understand the performance of a computer system by looking at the throughput of operations for arithmetic and data movement parts of the hardware. If the data movement is not operating quickly enough then the arithmetic functions are delayed waiting for the necessary data and vis a versa. The point at which arithmetic and movement match is known as ”machine balance” [36].

2.2.1 Arithmetic Intensity and Data Movement.

In Callahan's work [36], the data movement is simply defined as the amount of data moved between the main memory and the processor. However, in the intervening years, there have been many layers of cache added between the processor and the main memory (see Figure 1b). The amount of data varies at each level, forcing users to question which is most useful. Additionally, there are several techniques for measuring or estimating the data movement each of which has its advantages and disadvantages.

Modern processors usually have 2 or 3 levels of cache and data movement between any of these levels can be used for AI. We refer to the movement between the level 1 cache (closest to the processor) and the processor as $Data_{L1}$. Similarly, movement between the level 2 and level 1 cache is referred to as $Data_{L2}$.

When referring to data movement to or from a particular level of cache, this is subtly different from the hardware-defined bandwidth of the cache. Most existing caches have a single set of ports that moved data from the layer below and layer below, but we only refer to the data moved to or from the next level closest to the CPU. We think of this metric as the "apparent bandwidth" i.e. what the CPU can access when the data is residing in the cache level in question. While there are situations in which the user will want to understand the utilization of the physical bandwidth at a particular cache level, the apparent bandwidth method is easier to relate to an application kernel and therefore more useful for improving the performance.

In the original Roofline [37], data movement was defined as that moved between the main memory and the last level of cache ($AI_{mainmem}$). The main memory bandwidth is often a significant limiting factor to the performance of applications so there is significant motivation to focus on this level. Additionally, users tend to understand the bottleneck posed by the main memory and there are many optimizations (i.e. cache block and data layout optimizations) that aim to alleviate the bandwidth and apply to many scientific kernels.

Unfortunately, $AI_{mainmem}$ has significant disadvantages. In particular, the volume of data moved can be inconsistent for different data sets and many algorithmic changes. For example, a kernel that repeatedly iterates through a data set that is significantly larger than the caches will have more data movement than a kernel operating on a data set that is close to the size of the last level cache. As a result, the $AI_{mainmem}$ will be volatile relative to the data st size. Similarly, if a cache blocking optimization is applied so that the kernel uses the cache's temporal locality, then the $AI_{mainmem}$ will change with the reduced main memory data movement. In the most extreme case, data will fit entirely in a cache level and the $AI_{mainmem}$ will become non-sensical. Such variations in the $AI_{mainmem}$ can be confusing and difficult to connect back to changes in the algorithm.

At the other extreme is the AI_{L1}, which uses the data moved between the first level of cache and the processor. This method was popularized by the Cache Aware Roofline Method [38]which produced Rooflines for each of the cache levels all based on the AI_{L1}. This method provides a consistent understanding of the data moved relative to the amount of computation even with changes to the data size or optimizations designed to better use cache. AI_{L1} is designed to be more closely tied to the algorithm under consideration enabling the user to more easily compare optimizations and predict what changes will best improve application performance. Additionally, it allows bandwidth lines to be plotted fr all the cache levels, so the user can understand which level is having the most impact on performance. The main downside is that new users of the Roofline are often not informed of the differences between AI_{L1} and $AI_{mainmem}$ leading to significant confusion.

For users looking for additional detail, they can combine the AI versions into a single Roofline. This method, the Integrated Roofline, plots each cache line based on its own AI and then plots one application point for each AI. The resulting plots can be quite confusing but offer more information on how the different cache levels are used.

The versions discussed so far are all defined relative to the hardware in use. One final option is a theoretical AI (AI_{theory}). In this case, data movement and floating-point operations can be computed by algorithm analysis and plotted accordingly. This method is most closely related to the formulation of the algorithm in question but can neglect important factors such as compiler optimizations. If these changes aren't understood, then the user may mis-characterize the kernel that is actually run on the system.

While the author's personal preference is the AI_{L1}, there are good reasons for the other options. What is most important is that the user understands which method is in use and takes that into consideration when reasoning about the performance of the application.

2.2.2 Roofline variations. The Roofline model [37] of performance recognizes the computation and data movement as the primary limiting functions of a given processor. The model then compares the floating point operations per second ($flops/s$) to the AI to model the performance potential of the processor. Users can use the AI and $flops/s$ of their application to compare to the potential performance. The position of the application point to the ceilings that represent the performance potential of different hardware features can help the user identify areas for improvement. Roofline Modeling is a useful tool, so researchers have developed many versions.

The peak potential of the processor generally includes all possible hardware optimizations available on the system including, SIMD operations, FMA instructions, threads, and any others. The user can then add roofs where each of those features is removed to measure the reduced potential ceiling. For the data movement part of the Roofline, cache level are often used, but features such as prefetching, unit stride access patterns, and NUMA domains are also possible.

The original version defined the arithmetic intensity in terms of the data moved between main memory and the last level of cache, but as discussed above, this has some drawbacks. In particular, there are no ceilings for the different cache levels. The Integrated Roofline model [39] adjusts the arithmetic intensity to each level of cache. This innovation allows the user to have ceilings of bandwidth for each cache level, but complicates the plotting of application points since there are multiple AI measurements to choose from. The Cache Aware Roofline resolves these issues by using the AI_{L1} as we discussed previously. We choose the Cache Aware version for our work in Chapter IV because of the ease of use.

Many other variations of the Roofline model exist, often targeting particular types of CPU hardware and software. Denoyelle et al. [40] target nodes with heterogeneous memory hierarchies. The authors of [41] modify the Roofline for asynchronous runtimes. Cardwell and Song [42] extend the model for distributed memory systems. Choi et al. modified the model to a greater extent to create a Roofline Model of Energy in [43].

Two separate groups, Marques et al. [44] and Cabezas and Püschel [45], extend the roofline to consider the instruction mixes of the application being studied. This allows the user to have more realistic expectations of the potential and a better understanding of which ceilings are limiting performance.

In addition to many versions of the Roofline, there are several papers about focused on applying the Roofline model in different situations. Lo et al. [46] present a tool to create the architectural model, and Doerfler et al. [47] apply the model to performance analysis of an application on an Intel XEON Phi processor. Norris et al. [48] present a visualization tool for Rooflines.

Most similar to our work, Ofenbeck et al. [49] develop a detailed method of measuring data for application points for the original Roofline. This work is limited to Intel CPUs and single-threaded validation.

2.3 Wrapping up the background

In this chapter, we have provided an overview of empirical performance analysis of scientific applications with particular focus on the use of hardware counters to collect data. Additionally, we discussed the Roofline model of performance and the different ways that a user may measure arithmetic intensity. Lacking in the existing literature are hardware counter metrics that provide actionable performance information to users on a diverse set of CPU types. The remainder of this dissertation shows that such metrics are possible for at least two CPUs with different sets of hardware counters.

CHAPTER III

METHODOLOGY

Throughout this dissertation, we use a consistent set of systems, tools, and benchmarks to develop and demonstrate our analysis methods. The systems we use have several major differences, especially in what types of data are measured in the hardware counters. The tools and benchmarks we try to keep consistent for the whole dissertation. We present the details of our systems, tools, and benchmarks in this Chapter.

3.1 Systems

We use two CPU types to design and test our methods. The first is an Intel Cascade Lake Gold, and the second is the Fujitsu A64FX. While both have 48 cores and 512-bit SIMD Instructions, the most important differences between the two appear in the memory subsystems. Cascade Lake's main memory is DDR with a peak Bandwidth (BW) of approximately 200 GB/s, while A64FX has High Bandwidth Memory (HBM) that measures 800 GB/s. Additionally, the Cascade Lake has three levels of cache while the A64FX has only two levels. Table 1 shows detailed specifications comparing the CPUs and memory subsystems.

Table 1. Architecture details for the systems used in this dissertation.

	A64FX	Cascade Lake Platinum 8260
ISA	ARM with SVE	x86 with AVX512
Sockets	4	2
Cores per socket	12	24
Threads per core	1	2
L1 Cache	64KiB per core	32KiB per core
L2 Cache	32 MiB Total	1 MiB/core (48MiB Total)
L3 Cache	None	35.75 MiB
Main Memory	31 GB HBM	188 GB DRAM
Cache Line size	256 Bytes	64 Bytes

On the Cascade Lake system we compile the applications with the Intel compiler version 19. and OpenMPI 3.1.6 when MPI is used.

On the A64FX we use three different compilers depending on the needs.

3.2 Tools

Measuring hardware counters requires the users to record the counter values and program data then process that data. We discuss some of the tools for empirical measurement of hardware counters in Chapter II Section 2.1. This section covers the methods we use.

PAPI [1] and lilbpfm [11] interface directly with the hardware and provide a consistent interface for accessing hardware counters on a variety of CPU architectures. The developers regularly update the libraries to keep pace with new CPUs. We use these libraries together which allows us to access the counters through the human-readable interface. Counter names used later in the dissertation will use the PAPI names.

We also use the Caliper [16] library to connect the PAPI interface to applications. Although PAPI can be directly called from applications, we use the Caliper library because it handles the interface with PAPI alongside the metadata necessary for understanding OpenMP and MPI results. We configure Caliper to write the results to a set of JSON files, which provide consistent format for processing the results.

As we discuss in Chapters IV and V, we need to collect around 50 counters to compute all the metrics presented. This process takes some time because there are more events counted by hardware counters than there are registers to count them. As a result, we automate the process of repeatedly running the application to collect all of the necessary counters.

Once the data is collected, we use python scripts to automatically compute our metrics.

3.3 Benchmarks

We use a set of benchmarks (small, carefully crafted applications) to help us understand how the CPUs are operating and exactly what the hardware counters are measuring. These benchmarks can serve multiple purposes, including validating hardware counter metrics and developing a deeper understanding of the CPU. We use the benchmarks in this section for both purposes and refer to them throughout the dissertation.

3.3.1 STREAM. STREAM [50] is a widely used benchmark for measuring peak memory bandwidth by element-wise additions and multiplications of large arrays. Conventionally, STREAM includes four kernels: Copy, Add, Scale, and Triad. The data size can be adjusted to fit in each cache to measure the bandwidth at each level of cache. We use the benchmark to measure the bandwidths of the memory subsystem and to validate various hardware counter metrics.

Note that this bandwidth is not that of the physical cache interface; rather, it is the bandwidth between a set of data residing primarily in a particular cache and the CPU. This distinction is subtle, but STREAM cannot be used to measure the physical bandwidth of a cache interface because the timing is performed on the CPU. It is perfect for measuring what we think of as the ”apparent bandwidth” for an application with data residing in some layer of cache. Our research is focused on understanding the performance of applications, not hardware, so we see this distinction as a useful feature of STREAM.

We adjust the STREAM benchmark so that the number of memory and arithmetic operations remain constant as the data size changes. We achieve this consistency by adding an additional repetitions loop and providing an automated tool for computing the sizes and iterations required for each system[1]. It is unlikely that we are the first to use STREAM in this way; however, we are not aware of any similar publications or publically available benchmarks.

We show our modified STREAM Triad in Algorithm 1. N is the number of elements in each array; this value determines the number of load and store operations performed by the benchmark. Originally, N was intended to be larger than the last level of cache to force the benchmark to repeatedly read from the main memory. We adjust N to measure how the apparent bandwidth changes as the data size changes. The first repetition loop (NT_1 in Algorithm 1) is part of the original STREAM benchmark designed to run the kernel multiple times for improved accuracy. We add a second iterations loop (NT_2) which can be adjusted to hold the number of operations constant while N is varied. Keeping the product of N and NT_2 constant ensures that the CPU performs the same amount of work on different sizes of data. This consistency allows the user to adjust the benchmark to measure the bandwidth of each layer of cache or validate the results of hardware counters.

3.3.2 FP Crunch.

We designed the FP Crunch benchmark to complement STREAM by measuring the peak rate of floating-point computation on a system. Algorithm 2 shows the FP Crunch benchmark. At first glance, it appears similar to STREAM, but the trials and the iterations loops are reversed. This change allows the CPU to run the arithmetic operations rapidly without

[1]https://github.com/HPCL/benchmarks/tree/shingles/kernels/shingles/STREAM

Algorithm 1 STREAM Triad

```
$N \leftarrow$ Data size
$NT\_1 \leftarrow$ Number of repetitions
$NT\_2 \leftarrow$ Number of repetitions to match operation count
$a \leftarrow [N]$
$b \leftarrow [N]$
$c \leftarrow [N]$
$scalar \leftarrow$ some scalar floating point
for $k$ in NT_1                                ▷ original repetitions
    for $kk$ in NT_2                 ▷ additional repetitions for our version
        for each $i$ loop nest $N$
            $a[i] = b[i] + scalar * c[i]$
```

waiting on memory operations. Additionally, manual scalar replacement of the array access prevents the compiler from adding unnecessary write operations to the kernel. Finally, the kernel can be easily modified (see Chapter V) to measure different types of arithmetic operations.

Algorithm 2 FP Crunch

```
$N \leftarrow$ Data size
$NT \leftarrow$ Number of trials
$a \leftarrow [N]$
$b \leftarrow [N]$
$c \leftarrow [N]$
for $i$ in N                                   ▷ Iterations loop
    $fa\_sca = fa[i]$
    for $kk$ in NT                             ▷ Trials loop
        $fa\_sca += fc[i] * fb[i]$
    $fa[i] = fa\_sca$
```

3.3.3 Cache Conflict Measurement.

We use a cache conflict benchmark as our last example of the data movement metrics. Similar to STREAM, this benchmark iterates over the elements in an array that is larger than the L1 cache; however the iteration steps by an offset of o. An outer loop is used to ensure the same number of load and store operations occur for all values of o. Algorithm 3 shows the benchmark. When the offset is a multiple of the number and size of cache blocks, then cache conflicts will occur. Our array size and operation count are large enough that only a fully associative cache could handle the number of conflicts.

Algorithm 3 Cache Conflict Benchmark

```
$N \leftarrow$ Data size
$o \leftarrow$ offset; number of elements between accesses
$P \leftarrow$ Number of operations
$L \leftarrow$ Number of trials to run: $P * \frac{o}{N}$
$a \leftarrow [N]$
$scalar \leftarrow$ some scalar floating point
**for** $k$ in L                          ▷ repetitions to keep operations constant
    **for** $i$ in $0, \dots, i+o, \dots, N$              ▷ iterate from 0 to $N$ by $o$
        $a[i] = scalar * a[i]$
```

Compared to the other benchmarks in this chapter, the cache conflict kernel is more specialized. We designed this kernel to identify a particular performance issue that can be difficult to identify in applications. In Chapter V, we use the benchmark to validate some of our hardware counter metrics for caches.

To our knowledge, this kernel is the first benchmark developed and publicly released which can reproduce the problem of cache conflicts on modern CPUs. In future work, it may be possible to use this benchmark along with other data to develop an exact method of diagnosing cache conflicts in real-world applications.

3.3.4 Matrix Multiplication. Matrix Multiplication is a vital part of many applications and an excellent benchmark kernel for performance analysis. The computation is well studied which allows performance analysts to try new techniques on a familiar subject and easily reproduce a variety of variations to compare.

Our Default version of matrix multiplication (Algorithm 4) is a triple nested loop that loops over the three matrices. While we will not review the process of matrix multiplication here, we will draw the reader's attention to two points. The entire B matrix is reused once for each row of the C matrix and each row of the A matrix is reused once for each column of the C matrix. This repetition provides an opportunity for the kernel to make use of CPU caches for improved data access. Secondly, if we assume column-major ordering, the kernel will skip over entire rows of the B matrix with each iteration preventing the CPU from using all of the data in the cache lines.

Algorithm 4 Default matrix multiplication

```
Let A be an $n \times m$ matrix
Let B be an $m \times o$ matrix
Let C be an $n \times o$ matrix
for $i \in \{0, \ldots, n-1\}$
    for $j \in \{0, \ldots, m-1\}$
        for $k \in \{0, \ldots, o-1\}$
            $C[i,j] += A[i,k] \times B[k,j]$
```

The first optimization of matrix multiplication is to transpose the B matrix which is shown in Algorithm 5. We make this adjustment intending to improve the use of data loaded into cache in cache lines. If this optimization were included in an actual application, rather than a benchmark, the users would need to assess the overhead required to transpose B. For our studies, we ignore that overhead since our goal is to use this benchmark to compare measurements of cache efficiency. Measuring the transposition along with the matrix multiplication would only obfuscate the results.

Next, we manually unroll the j loop and jam the resulting loops back together (Algorithm 6). Known as Unrolljam, this optimization can improve the

Algorithm 5 Transpose Matrix Multiplication

```
1: Let A be an $n \times m$ matrix
2: Let B be an $m \times o$ matrix
3: Let C be an $n \times o$ matrix
4: Let BT be the Transposition of the B matrix
5: for $i \in \{0, \ldots, n-1\}$
6:     for $j \in \{0, \ldots, m-1\}$
7:         for $k \in \{0, \ldots, o-1\}$
8:             $C[i, j] += A[i, k] \times BT[j, k]$
```

use of SIMD operations and the pipeline efficiency of the kernel. We apply this optimization on top of the Modern compilers can also unroll loops automatically, but we apply the optimization manually for another opportunity to study the results of our metrics.

Algorithm 6 Unroll-jam matrix multiplication

```
1: Let A be an $n \times m$ matrix
2: Let B be an $m \times o$ matrix
3: Let C be an $n \times o$ matrix
4: Let BT be the Transposition of the B matrix
5: for $i \in \{0, \ldots, n-1\}$
6:     for $j \in \{0, 8, 16, \ldots, m-8\}$
7:         for $k \in \{0, \ldots, o-1\}$
8:             $C[i, j] += A[i, k] \times B[j, k]$
9:             $C[i, j+1] += A[i, k] \times B[j+1, k]$
10:             $\ldots$
11:             $C[i, j+7] += A[i, k] \times B[j+7, k]$
12:     for $j \in \{j, \ldots, m-1\}$
13:         for $k \in \{0, \ldots, o-1\}$
14:             $C[i, j] += A[i, k] \times B[j, k]$
```

Our final optimization, shown in Algorithm 7, is to block the loops so that operations that reuse matrix elements occur shortly after each other. This optimization is applied after transposing B but without the unrolling. Unrolling and Blocking can be combined, but this results in much more complex code without a significant change in performance. The combination does not add to our understanding of performance analysis, so we do not present it here.

Blocking is a common optimization for matrix codes since it can significantly improve the cache efficiency of some applications. We collect data on a range of block sizes for our study. Finding the best block size is a tedious process that can be performed using autotuners such as [51] and [52]. We explore a range of block sizes with our metrics in Chapter V which allows us to develop a better understanding of the measurements we make and the impacts of blocking on the kernels.

Algorithm 7 Blocked matrix multiplication

```
1: Let A be an $n \times m$ matrix
2: Let B be an $m \times o$ matrix
3: Let C be an $n \times o$ matrix
4: Let BT be the Transposition of the B matrix
5: for $ii \in \{0, \ldots, BlockSize, \ldots, n-1\}$
6:     for $jj \in \{0, \ldots, BlockSize, \ldots, m-1\}$
7:         for $kk \in \{0, \ldots, BlockSize, \ldots, o-1\}$
8:             for $i \in \{ii, \ldots, ii + BlockSize - 1\}$
9:                 for $j \in \{jj, \ldots, jj + BlockSize - 1\}$
10:                    for $k \in \{kk, \ldots, kk + BlockSize - 1\}$
11:                        $C[i,j] += A[i,k] \times B[j,k]$
```

These four variations of matrix multiplication provide a useful tool for our study of performance measurement. The computation is familiar to readers in many areas of computational science, so it is a useful example for reaching a wide audience. Similarly, the code needed to run a matrix multiplication is smaller, but the size of the data can be scaled to fill large caches or even multiple nodes of a cluster. Despite the simplicity of the kernel, we have four variations with different performance features, allowing us to examine and then showcase many hardware counter metrics in Chapter V.

3.3.5 N-Body Simulation.

N-Body problems arise in science when a set of objects each interact continuously with each other object in the set. These systems can be modeled by computing the pairwise interactions for all the objects in each timestep. For example, this method can model the gravitational interactions of celestial bodies. Each body in the 3-dimensional space has a position, velocity, and mass. The position and velocities are represented by 3 element vectors in a Cartesian space. The equation to calculate the force applied by two objects on each other is:

$$\mathbf{f}_{ij} = G * \frac{m_i * m_j * \mathbf{r}_{ij}}{||\mathbf{r}_{ij}||^3} \tag{3.1}$$

Our N Body benchmark computes this force between each pair of bodies and then uses it to update the positions at each timestep. In the above equation the force ($\mathbf{f}_{ij}$), and distance between objects ($\mathbf{r}_{ij}$) are each 3D vectors. The mass of the two objects (m_i and m_j) are scalars. The algorithm used to simulate the system of celestial bodies is shown in Algorithm 8.

N-Body is similar to matrix multiplication in the sense that it is a simple kernel that is particularly useful for the study of performance analysis. The problem is widely known and implementations do not require a significant coding effort, but there are numerous optimization options available for those interested in applying them. For example, the data structures are based on one-dimensional arrays which can be organized as arrays of structures or structures of arrays. Some of the optimizations are not useful in the matrix multiplication algorithm, so the two kernels complement each other in that way.

Algorithm 8 N Body Problem

```
1: Let $T$ be the number of timesteps to complete
2: Let $dt$ be the change in time with each timestep
3: Let $N$ be the number of particles
4: Let $M$ be an array of size $N$ with the mass of each particle
5: Let $PX$ be an array of size $N$ with the x position of each particle
6: Let $VX$ be an array of size $N$ with the x velocity of each particle
7: Let $PY$ be an array of size $N$ with the y position of each particle
8: Let $VY$ be an array of size $N$ with the y velocity of each particle
9: Let $PZ$ be an array of size $N$ with the z position of each particle
10: Let $VZ$ be an array of size $N$ with the z velocity of each particle
11: for $t \in \{0, \dots, T-1\}$
12:     for $j \in \{0, \dots, N-1\}$
13:         $a_x = 0.0$
14:         $a_y = 0.0$
15:         $a_z = 0.0$
16:         for $k \in \{0, \dots, N-1\}$
17:             $dx = PX[k] - PX[j]$
18:             $dy = PY[k] - PY[j]$
19:             $dz = PZ[k] - PZ[j]$
20:             $d = \sqrt{dx * dx + dy * dy + dz * dz}$        ▷ distance between the objects
21:             $w = \frac{M[k]}{d^3}$
22:             $a_x = a_x + w * dx$        ▷ Acceleration caused by the interaction
23:             $a_y = a_y + w * dy$
24:             $a_z = a_z + w * dz$
25:         $VX[j] = VX[j] + a_x * dt$        ▷ Update speed
26:         $VY[j] = VY[j] + a_y * dt$
27:         $VZ[j] = VZ[j] + a_z * dt$
28:     for $j \in \{0, \dots, N-1\}$        ▷ Update positions
29:         $PX[j] += VX[j] * dt$
30:         $PY[j] += VY[j] * dt$
31:         $PZ[j] += VZ[j] * dt$
```

3.4 Summary

In this chapter, we presented the experimental environment that we use to develop and validate our methods of empirical hardware counter performance analysis. The systems uses different microarchitectures, ISAs, and hardware counter architectures. These differences ensure that our metrics are applicable to at least some variations of CPUs and can be measured with at least some variation of hardware counter designs. For readers who wish to replicate our work, we include an explanation of the measurement tools we used. These tools are one of many options that a user could chose to collect the hardware counter data needed for the metrics discussed later in this dissertation. Finally the bulk of the chapter focused on a series of benchmarks that we will use in subsequent chapters. These benchmarks have known performance characteristics which allow us to validate and demonstrate hardware counter measurements. We refer back to this chapter as we use the systems, tools, and benchmarks discussed here.

CHAPTER IV

EMPIRICAL ROOFLINES WITH HARDWARE COUNTERS

As we discussed in Chapter II, the Roofline Method [37] is a common technique for understanding the node-level performance of HPC systems and applications. The Roofline Model of a processor separates performance into two aspects: the data movement and the computation. These aspects are measured as bytes of data moved between the L1 Cache and the CPU (for the cache aware version), floating-point operations, and execution time. Despite differences in architectures, the Roofline Model is a widely used tool for comparing the performance of an application to the potential performance of the system.

In this chapter, we identify hardware counters which can be used to measure the metrics necessary to plot application points on architectural rooflines. Empirical methods for obtaining this information have long been lacking in the literature, so we fill the gap with hardware counter metrics across multiple system types. By doing so, we demonstrate that an existing CPU-agnostic performance analysis technique can be used in conjunction with hardware counter metrics derived from diverse CPU types.

For the purposes of this research, we use the Cache Aware Roofline [38] based on the data movement and bandwidth between the L1 cache and the CPU based (BW_{L1}). First (Section 4.1), we develop a set of benchmarks to measure data movement and computation on CPUs. These benchmarks allow us to produce architectural rooflines and to validate the hardware counter metrics. These metrics are presented and validated in Section 4.2 which explains how they can be used to place application points on an architectural roofline. We show examples of the combination of architectural rooflines and application points in Section 4.3.

This chapter includes previously published [53] co-authored work. Contributions to the paper are as follows: Brian Gravelle, the author of this dissertation, was first author in [53]. He developed the hardware counter metrics to enable Roofline analysis with hardware counters and modified existing. Additionally, he collected the hardware counter data for validation and case studies. William Nystrom assisted with the benchmark design, and the writing of the paper. Dewi Yokelson contributed the software needed to produce the plots of the Roofline models. Boyana Norris gave advice on many aspects of the work especially identifying research aims and presenting the results effectively to the reader in the resulting paper.

4.1 Architectural Rooflines

Producing accurate Rooflines can be a challenge thanks to ever-expanding hardware and compiler features that make achieving peak floating-point performance elusive to the casual user. Some options exist to handle this challenge, but these are limited to specific vendor systems or require too much user input to be reliable for some users.

To produce our architectural Rooflines, we use two benchmarks, one for the floating-point peak and one for the peaks in the memory subsystem. Using two separate kernels lets us focus each on the details of the hardware that is being exercised. Our first benchmark can consistently measure the performance of each level of the memory subsystem, as it appears to the CPU when running a kernel out of that level of memory. The second is designed to flexibly exercise the arithmetic features of the CPU, providing an ideal maximum performance of the computational hardware. The results can then be combined to respectively draw the angled and horizontal parts of the Roofline.

4.1.1 STREAM Modification.

We use our modified STREAM benchmark from Chapter III to measure the bandwidth of each level of the memory hierarchy for the systems in question. This benchmark is designed to consistently execute the same number of operations for each level of cache which simplifies the validation of the benchmark and the measurements. Because we know the amount of data moved for the kernel, we can measure the time it takes to complete the computation and compute the bandwidth (BW) from the data size (D) and time (t) as follows:

$$BW = \frac{D}{t} \tag{4.1}$$

Note that this bandwidth is not the bandwidth of the physical hardware connected to a level of the memory hierarchy. Rather, we are measuring the bandwidth between the CPU and the memory level including the systems in between. We notate this bandwidth as BW_{cpu-X}, where X is the level of memory. The measurement takes many aspects of the microarchitecture into consideration. Delays, such as those caused by caches in between the CPU and target memory level, are part of the measurement as well as cache optimizations such as cache lines that move more data than requested. However, the data is large enough to prevent the kernel from reusing data in any cache closer to the CPU than the target cache. For example, our L2 bandwidth is measured with a data size larger than the L1 cache, but smaller than the L2 cache. Therefore, the benchmark includes microarchitectural features that an application can take advantage of if its data is similarly sized, but not those features that would only be useful to computation on data of a smaller size.

Consider a kernel that operates on a data set that is larger than the L1 cache, but smaller than the L2 cache. If it performs computation that iterates through the entire data set multiple times, then it is unlikely that the kernel will reuse data in the L1 cache. The cache line, which is larger than a double-precision floating-point will load more data than requested into the L1 cache. The kernel can take advantage of this optimization if it loads data sequentially in memory. Optimizations that take advantage of the cache line could bring the bandwidth of this example kernel up to the measured bandwidth. If the user adjusted the kernel to reuse data in the L1 cache (i.e., by blocking the loops), then this optimization could raise the bandwidth above our L2 bandwidth. We would consider the optimized kernel to be operating out of the L1 cache.

Table 2. Cache details of the two systems.

	A64FX	Cascade Lake 8260
Cores	48	48 (96 with SMT)
Threads per core	1	2
L1 Cache Size	64KiB per core	32KiB per core
L1 Cache Type	Private to core	Private to core
L2 Cache Size	0.68 MiB per core (32 MiB Total)	1 MiB/core (48MiB Total)
L2 Cache Type	Shared by CMG (12 cores)	Private to core
L3 Cache	None None	35.75 MiB Total (0.763 MiB per core)
L3 Cache Type L3 Cache Type		Logically shared physically distributed
Main Memory	31 GB HBM	188 GB DRAM
Cache Line size	256 Bytes	64 Bytes

The A64FX has two caches and the main memory, so we could choose three data sizes to measure the bandwidth for each cache. The user can use the documentation to determine the size of the caches (see Table 2) and set STREAM to run with data smaller than the size of the cache of interest. For example, on A64FX, we could run with arrays of 20 KiB (2560 double-precision floats). STREAM Triad uses three arrays, so the total data size will be 60 KiB. STREAM will then run Triad with this data size on each of the cores.

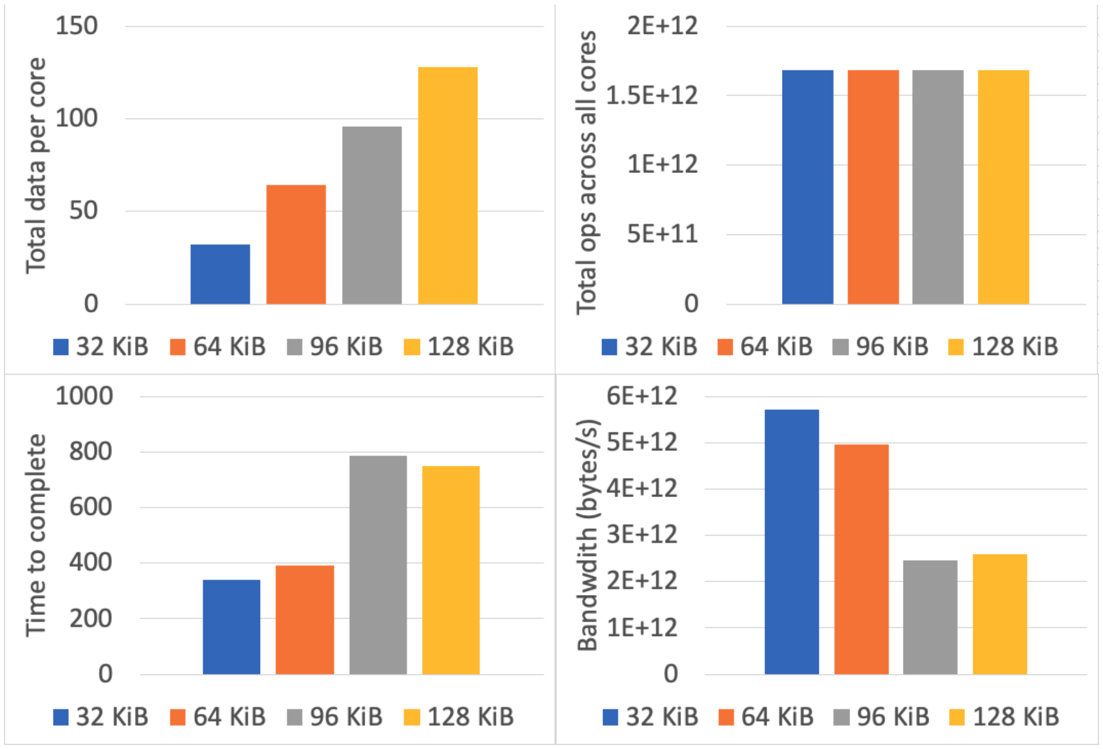


Figure 2. Results running the modified STREAM benchmark on A64FX with sizes slightly smaller and larger than the L1 cache. The upper left figure shows the bytes allocated per core. The upper right plot shows the number of operations executed. Bottom Left shows the resulting time. The bottom right plot shows the measured Bandwidth as seen by the CPU.

For example runs, we ran STREAM with a series of sizes ranging from slightly smaller to slightly larger than each cache. Figure 2 shows the results on A64FX for sizes around the L1 cache size. The upper left of the plot shows the size of data allocated per core for each of the four trials. Our STREAM benchmark computes the number of repetitions to perform so that it completes the same number of operations (upper right figure) for any size. There is a slight (less than 0.01%) variation in the number of operations due to rounding involved in integer division. Since the number of load and store operations are constant, the time measurement can demonstrate the change in cache level as the data increases in size (see the bottom left of the plot). We then compute the number of bytes from the number of operations and divide by the time to get the bandwidth (bottom right) for each of the sizes.

Figure 2 shows the bandwidth results for the four data sizes we selected. Checking with our documentation we know that either the 64 KiB or 32 KiB result will apply to our L1 cache. We take the highest of the values because the goal of the benchmark is to measure peak performance. Therefore, we conclude that:

$$BW_{cpu-L1} = 5,703.5GB/s \tag{4.2}$$

We repeat the experiment on the Cascade Lake system with smaller sizes because of the smaller L1 cache. Figure 3 shows the results of that experiment. These results suggest that:

$$BW_{cpu-L1} = 12,433.5GB/s \tag{4.3}$$

The results in Figures 2 and 3 include data sizes that are larger than the respective L1 caches. The measurements suggest that:

On the A64FX

$$BW_{cpu-L2} = 2,593.2GB/s \tag{4.4}$$

And on the Cascade Lake:

$$BW_{cpu-L2} = 6,762.7GB/s \tag{4.5}$$

However, the experiments that produced these BW_{cpu-L2} measurements use data sizes that are only slightly larger than the L1 caches. We rerun the

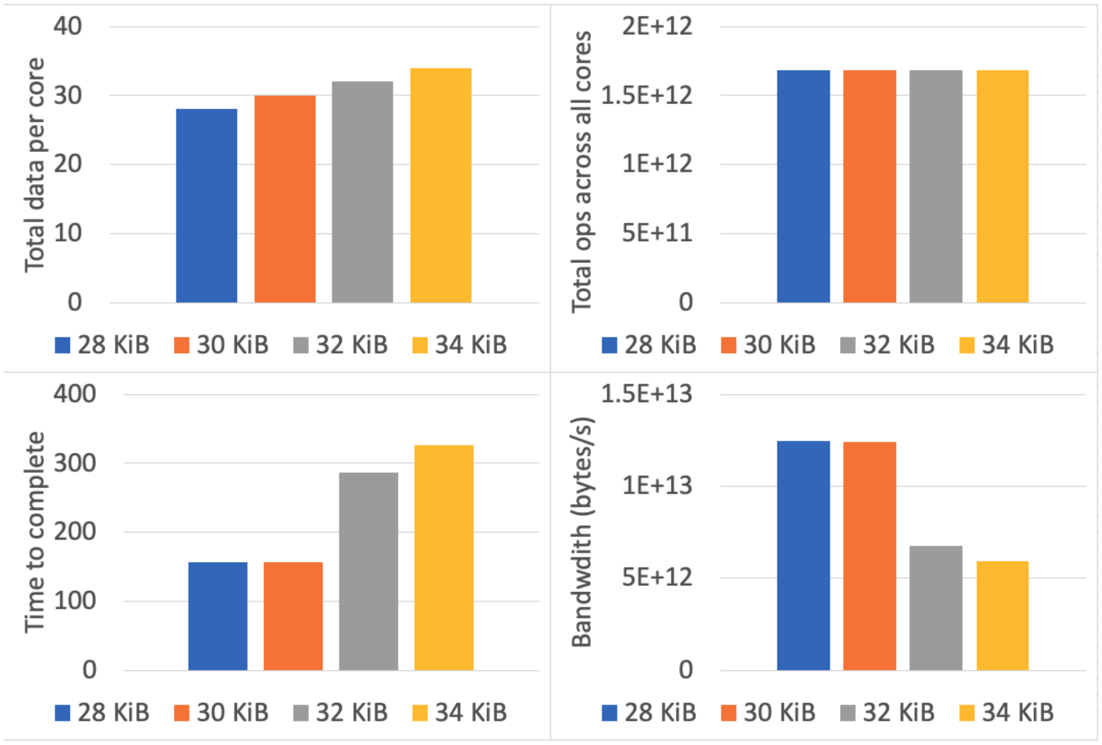


Figure 3. Results running the modified STREAM benchmark on Cascade Lake with sizes slightly smaller and larger than the L1 cache. The upper left figure shows the bytes allocated per core. The upper right plot shows the number of operations executed. The bottom left shows the resulting time. The bottom right shows the measured bandwidth as seen by the CPU.

experiments with data around the size of the L2 cache on both systems to ensure that a small dataset relative to the total L2 size does not produce unrealistically high expectations of the BW_{cpu-L2}. We suspect that if the data is not more than twice as large as the L1 cache then the measurement may include some unintended reuse in the L1 cache. That reuse would mean we are not accurately measuring BW_{cpu-L2} as defined above.

Figures 4 and 5 show the results of running the experiments on the A64FX and the Cascade Lake systems with data sizes that range from fitting into the L1 cache to larger than the last level of cache. The Xs mark the sizes of each layer of cache on the systems. In both cases, the plots show how the measured BW can vary even for sizes within a cache level. When the data size is only slightly larger than the next smaller level of cache, then the measured bandwidth tends to be higher. Similarly, when the data size is close to the maximum for some level of cache, then the bandwidth tends to be lower.

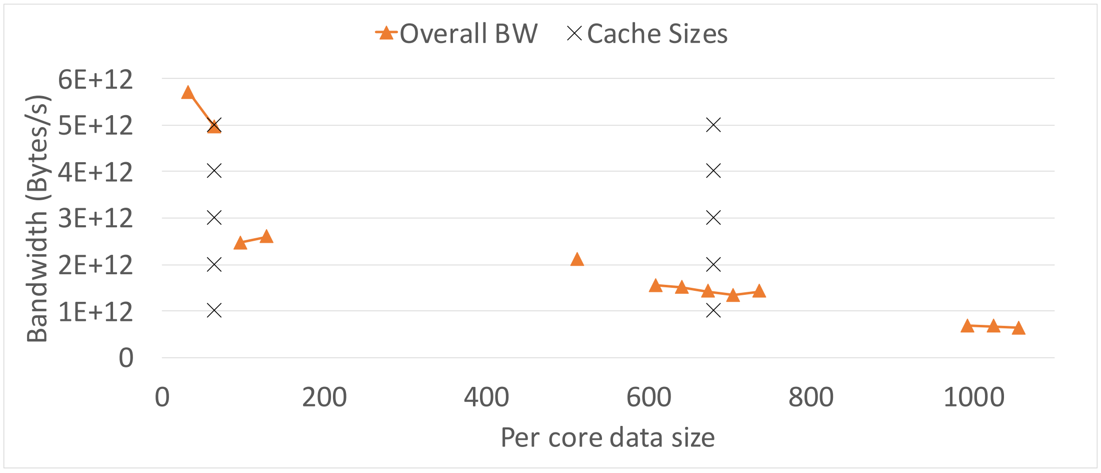


Figure 4. STREAM Bandwidth relative to per core data size on the A64FX. To help understand the bandwidth relative to the caches, we use Xs to indicate the per core caches sizes for the L1 (64 KiB) cache.

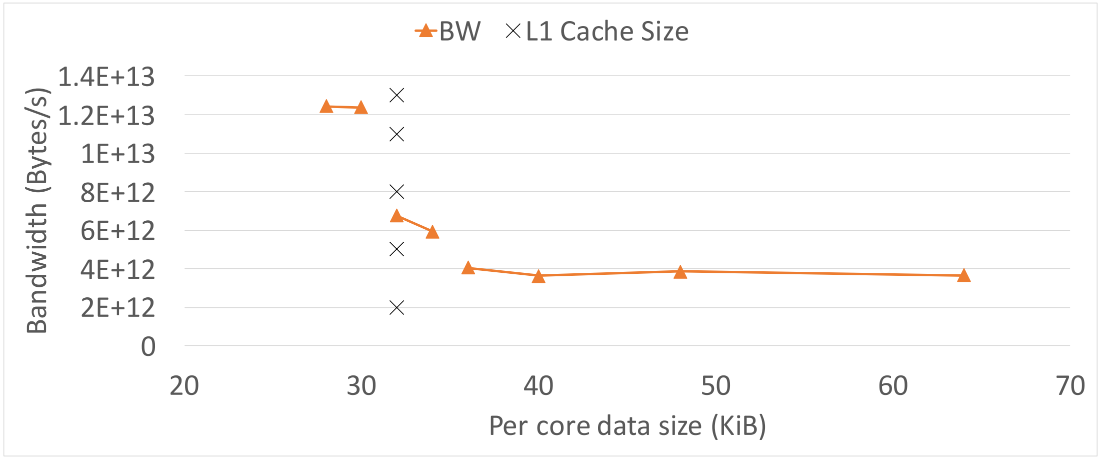


Figure 5. STREAM Bandwidth relative to per core data size on the Cascade Lake. To help understand the bandwidth relative to the caches, we use Xs to indicate the per core cache size for the L1 (32 KiB) cache.

We extend the results on the Cascade Lake in Figure 6 to show bandwidth results for STREAM sizes close to the L2 and L3 capacities, and near the middle of those capacities. Additionally, we include the bandwidth from a size of 4096 KiB per core which is large enough to provide us with the main memory bandwidth. From L1 to L2 and from L2 to L3 there is a noticeable drop in bandwidth as the array size increases beyond the cache capacity. There is not a similar step-change in the bandwidth as the array size increases beyond the size of the L3 cache. This smooth transition seen in Figure 6 is caused by the hardware implementation of the cache and main memory including the prefetching hardware.

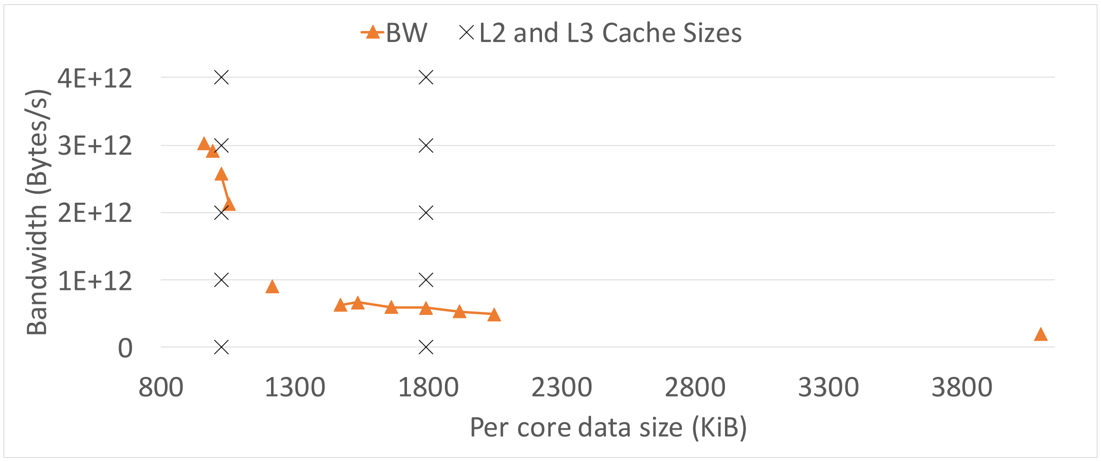


Figure 6. STREAM Bandwidth relative to per core data size on the Cascade Lake. To help understand the bandwidth relative to the caches, we use Xs to indicate the per core caches sizes for the L2 (1024 KiB), and L3 (1787 KiB) caches. Note that the L3 cache does not include data in the L2 cache, so that size is the sum of the L2 and L3 caches.

We prefer to use bandwidths measured from approximately half of the cache size, but other users may have different preferences. The data sizes close to the cache sizes may be capturing effects of the other caches improving or degrading performance, so we conclude that the middle ground will be most representative. If an application has data that fits closer to the edge cases, then the user may prefer to choose another size. Based on this reasoning, we use the following:

On the A64FX

$$BW_{cpu-L1} = 5,703.5GB/s \tag{4.6}$$

$$BW_{cpu-L2} = 2,096.9GB/s \tag{4.7}$$

$$BW_{cpu-memory} = 669.8GB/s \tag{4.8}$$

And on the Cascade Lake:

$$BW_{cpu-L1} = 12,433.5GB/s \tag{4.9}$$

$$BW_{cpu-L2} = 3,651.2GB/s \tag{4.10}$$

$$BW_{cpu-L3} = 910.2GB/s \tag{4.11}$$

$$BW_{cpu-memory} = 204.9GB/s \tag{4.12}$$

4.1.2 FP Crunch. For the floating-point peaks in our architectural Roofline, we developed an FP Crunch benchmark (see Chapter III). This benchmark is similar to the STREAM benchmark but with two major changes. First, the data size is much smaller than the L1 cache on the target system. Second, the loop over the array elements is moved outside the iterations loop. These two changes produce a benchmark that repeatedly performs floating-point operations on data that resides in CPU registers which allows the user to measure the computational potential of the CPU unhindered by the latency of the memory subsystem.

The FP Crunch benchmark executes a specific number of operations and reports this count to the user along with the time required to complete the operations and the rate of computation. We compute the rate of computation (R) as the ratio of the number of floating-point operations (F) and the time (t) that is required to complete those operations:

$$R = \frac{F}{t} \tag{4.13}$$

The user can use compiler options to guide the build process to measure different aspects of the computational hardware. For example, most compilers have options to exclude SIMD operations. Similarly, we enable the user to select the precision (double or single) at compile time. These options allow the user to place ceilings on the Roofline for features such as SIMD operations and to produce Rooflines for both single and double precision computation.

Table 3 shows the results of running four versions of FP Crunch on the A64FX. We use compiler options to run with and without SIMD operations for both the multiplication and addition kernels. The multiply scalar version is able to use FMA operations, so it is twice as fast as the addition scalar version. Unfortunately, the GCC compiler was unable to combine SIMD and FMA operations for this kernel, so the multiply SIMD and addition SIMD versions have the same rate of computation. Therefore we double the computational rate of multiply SIMD version to get our peak potential computation rate.

Table 3. Results from the FP Crunch benchmark on A64FX

	Time (s)	Speedup (scalar / simd)	GFlops	GFlops/s
mult scalar	42.54	1.00	6,390	150
mult simd	6.17	6.89	6,390	1,040
add scalar	84.17	1.00	6,390	75.9
add simd	6.17	13.64	6,390	1,040

We show the Cascade Lake results in Table 4. In this case, we include runs with 48 threads and with 96 threads to take SMT into consideration. For scalar and SIMD versions, the multiplication kernel takes is faster than the addition kernel due to the FMA operations. The SIMD versions are only 1.6× to 1.9× faster than the scalar versions. We conclude that the scalar versions are using some SIMD operations.

Table 4. Results from the FP Crunch benchmark on Cascade Lake

	Time (s)	Speedup (scalar / simd)	GFlops	GFlops/s
mult scalar 48 thr	3.59	1.0	6,390	1,780
mult simd 48 thr	2.15	1.7	6,390	2,973
mult scalar 96 thr	6.86	1.0	12,780	1,863
mult simd 96 thr	4.25	1.6	12,780	3,008
add scalar 48 thr	7.03	1.0	6,390	909
add simd 48 thr	3.72	1.9	6,390	1,718
add scalar 96 thr	12.9	1.0	12,780	991
add simd 96 thr	7.52	1.7	12,780	1,700

4.2 Application Points

To plot the application points we need to measure three values for a kernel: the time, the number of floating-point operations, and the amount of data moved (between L1 cache and CPU in this case). If we are to use Rooflines across a variety of different architectures, then we must be clear about what we are measuring and certain that those measurements are accurate. The benchmarks used for the architectural lines (Section 4.1) can be useful in validating the counters on different systems. Most of these results were previously published at the PMBS workshop [53].

4.2.1 Measuring Time. Measuring time can be as simple as watching a clock on the wall, but for most kernels, we prefer more exact methods. Generally, this means using timers that are built into the language which access the operating system clock. We place these timers around the function or the kernel of interest to avoid measuring the overhead that we do not want to consider in the analysis. For parallel applications we measure the These timers are the only metric we use that is not from hardware counters.

4.2.2 Measuring Data Movement. Here we present methods that can effectively provide empirical estimates for the amount of data moved between the CPU and the closest (L1) level of cache. This metric is useful because it can be readily understood by users, is closely related to the algorithm implementation, and can be measured on all the processors of interest to us. We intend that detailing our methods and reasoning will enable other researchers to extend our work to additional systems as they are released.

The systems in this work provide counters for load and store operations which provide a close approximation of the amount of data moved for a kernel of computation. Bytes of data moved can be computed as:

$$Bytes = [Load\ Store\ operations] * [Bytes\ per\ operation] \tag{4.14}$$

However, thanks to features such as SIMD operations, the number of bytes moved by a single instruction is variable. How does the user know what fraction of data movement operations are for moving larger sets of data to SIMD registers, which is moving double, single, or half-precision data, and what is moving integers or program instructions? The answer to this question depends on how the vendor implements the hardware counters. Our two CPUs implement the counters in different ways, so we need to establish two methods of measuring data movement. Bridging this gap between hardware counter architectures has not been published prior to our work in [53] and will allow performance analysts to use the same metrics on different CPU types.

Table and 5 list the counters used to compute data movement on the A64FX while Algorithm 9 show how to compute the **Bytes**. The A64FX measures three types of load and store instructions. There is the total count of loads and stores and then three different types of floating point data that can be used. We compute the number of load and store bytes by identifying the number of bytes that each type of data requires and then isolating that type from the other kinds of loads and stores. Once each type of instruction is counted correctly, we multiply that count by the number of bytes it moves and then sum the total back together.

Table 5. Counters used to calculate the Bytes on A64FX

[LD,ST]_SPEC	Scalar floating point instructions
ASE_SVE_[LD,ST]_SPEC	Executed Loads and Stores into FP registers
FP_[LD,ST]_SPEC	Executed Loads and Stores into scalar FP registers

For Intel systems, the load and store instruction counters do not differentiate between the different data sizes loaded, but there are several counters for counting different types of computation such as precision and SIMD vector sizes. Using this data, we assume that the ratios of data movement operation types roughly correspond to the ratios of floating point operation types. The ratios of floating point operation types are then used to weight the required bytes per operation

Algorithm 9 *Bytes* moved on A64FX

```
$precision$ = [SP, DP]
$[LD, ST]\_ins\_cnt$=[LD,ST]_SPEC
$[LD, ST]\_fp\_cnt$=ASE_SVE_[LD,ST]_SPEC
$[LD, ST]\_sclr\_cnt$ = FP_[LD,ST]_SPEC
$Bytes = 4 * (LD\_sclr\_cnt + ST\_sclr\_cnt)$
if $precision$ == SINGLE
    $Bytes$ = 2*$Bytes$
$Bytes$ += $4 * (LD\_ins\_cnt + ST\_ins\_cnt$

$-LD\_fp\_cnt - ST\_fp\_cnt)$
$Bytes$ += $64 * (LD\_fp\_cnt + ST\_fp\_cnt$

$-LD\_sclr\_cnt - ST\_sclr\_cnt)$
```

for the total byte count. Tables 6 list the counters used on Cascade Lake while Algorithm 10 show how to compute the **Bytes**.

A significant downside to this work is that, to date, we have not managed to develop a method of checking how close such estimation is for non-trivial codes. In particular, we would like to be able to check for edge cases where computation is run largely out of registers, or floating-point operations are not a significant driver of the load and store operations.

Table 6. Counters used to calculate the flops and bytes on Cascade Lake

PAPI_[LD,SR]_INS	Load and Store Instructions
FP_ARITH_INST_RETIRED… _SCALAR_[SINGLE,DOUBLE]	Scalar floating point instructions
FP_ARITH_INST_RETIRED… _128B_PACKED_[SINGLE,DOUBLE]	128bit SIMD floating point instructions
FP_ARITH_INST_RETIRED… _256B_PACKED_[SINGLE,DOUBLE]	256bit SIMD floating point instructions
FP_ARITH_INST_RETIRED… _512B_PACKED_[SINGLE,DOUBLE]	512bit SIMD floating point instructions

We use our STREAM modification to validate the hardware counter metrics. Figures 7 and 8 show how the number of bytes which we expect STREAM to move

Algorithm 10 *Bytes* moved on Cascade Lake

```
$prec$ = [SINGLE, DOUBLE]
$prefix$ = FP_ARITH_INST_RETIRED:
$FP\_scalar$=prefix_SCALAR_$prec$
$FP\_128b$=prefix_128B_PACKED_$prec$
$FP\_256b$=prefix_256B_PACKED_$prec$
$FP\_512b$=prefix_512B_PACKED_$prec$
$LS\_INS$ = PAPI_SR_INS + PAPI_LD_INS
$FP\_tot = FP\_scalar\_cnt$
$+ FP\_128b\_cnt$
$+ FP\_256b\_cnt$
$+ FP\_512b\_cnt$
$S\_BYTE = 4 * \frac{FP\_scalar}{FP\_tot} * LS\_INS$
**if** $precision$ == DOUBLE  $S\_BYTE = S\_BYTE$ *2
$128\_BYTE = 16 * FP\_128b/FP\_tot$

$256\_BYTE = 32 * FP\_256b/FP\_tot$

$512\_BYTE = 64 * FP\_512b/FP\_tot$

$LS\_BYTES = S\_BYTE + 128\_BYTE + 256\_BYTE + 512\_BYTE$
```

compares to the number of bytes that we measure with the hardware counters discussed above.

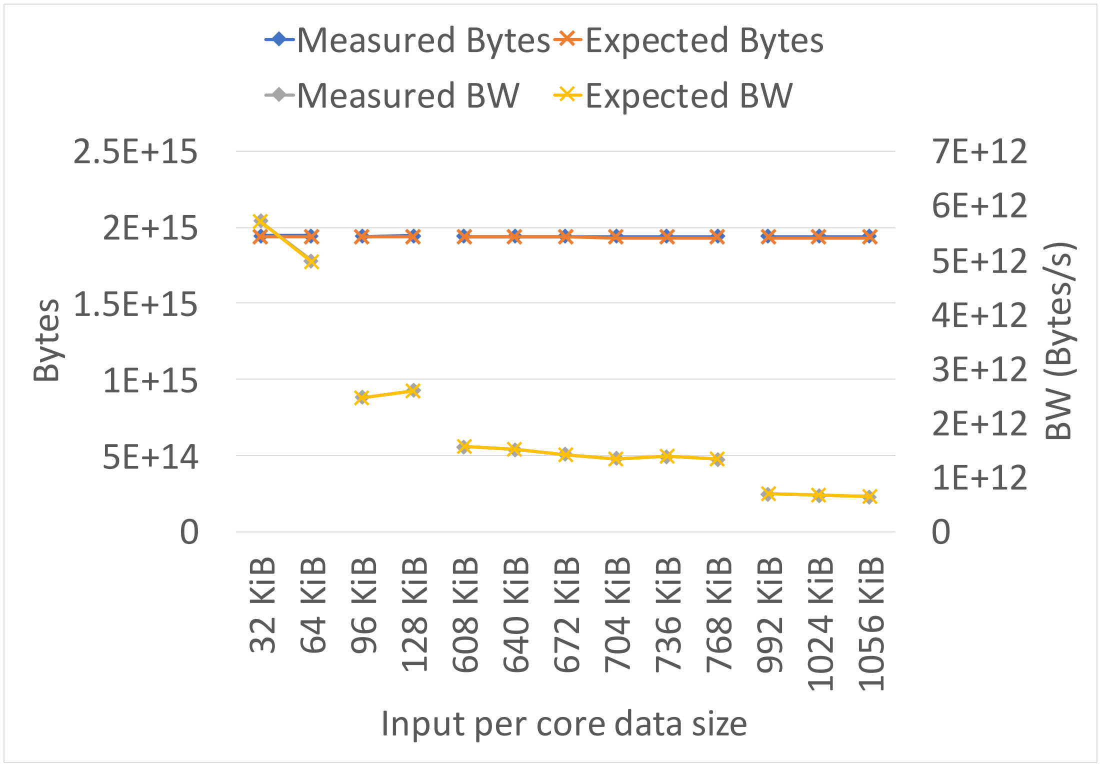


Figure 7. Comparison of the bytes and bandwidth of data moved between the CPU and L1 cache measured with hardware counters and those expected from the STREAM benchmark running on the A64FX.

4.2.3 Measuring Floating Point Operations.

We need to address two issues to be able to count floating-point operations. First, we need to define a single floating point operation. Second, we identify different ways that they can be executed on current CPUs. This understanding will then allow us to approach a set of relevant hardware counters and determine the best way to collect the count of floating-point operations.

In this dissertation, we define floating point operations (**flops**) as arithmetic operations on floating point operands. These operations consist of the following:

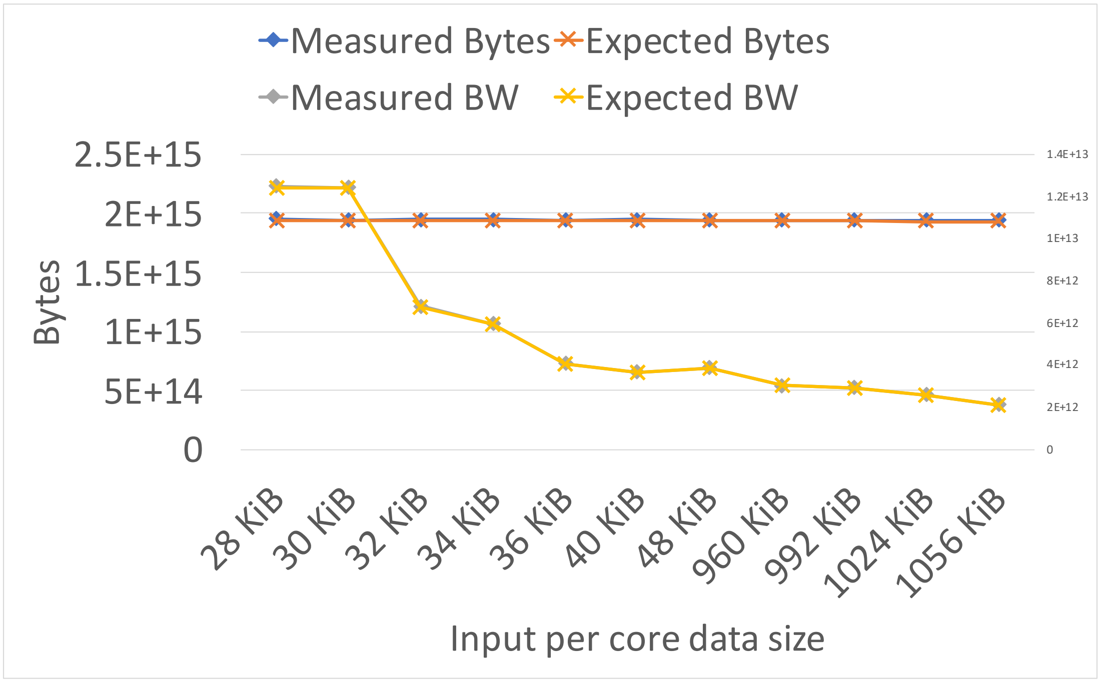


Figure 8. Comparison of the bytes and bandwidth of data moved between the CPU and L1 cache measured with hardware counters and those expected from the STREAM benchmark running on the Cascade Lake.

– Addition and Subtraction

– Multiplication

– Division

– Logarithms and Trigonometry in some cases

The first two categories of operations can all be performed with a single instruction on most or all CPUs used in modern HPC systems. Divisions, logarithms, and trigonometric operations can be performed as single instructions or multiple instructions depending on the microarchitecture and the compiler options selected. Since we cannot modify the hardware counters on our systems, we must simply accept whatever data is provided to us.

The A64FX and the Cascade Lake CPUs both provide counters for floating-point instructions which are more alike than the data movement counters. Tables 7 and 12 list the required counters used on the A64FX and Cascade Lake while Algorithms 11 and 12 show how to compute the **FLOPs** on the two CPUs. For both systems, the process requires the user to multiply each counter by the appropriate scale and then sum the results to get a total count of operations.

Table 7. Counters used to calculate the FLOPs on the A64FX

FP_[SP,DP]_SCALE_OPS_SPEC	Scalable floating point operations (assumes 128 SIMD length)
FP_[SP,DP]_FIXED_OPS_SPEC	Fixed floating point operations (correctly counts SIMD lengths)

Algorithm 11 flops on A64FX. RVL is the relative vector length compared to the assumed 128 bits. For our work, we have the compiler use 512 bit SIMD operations, RVL is 4.

```
$precision$ = [SP, DP]
$FP\_sve\_cnt$ = FP_[SP,DP]_SCALE_OPS_SPEC
$FP\_fix\_cnt$ = FP_[SP,DP]_FIXED_OPS_SPEC
$FLOPs = FP\_fix\_cnt$ + RVL*$FP\_sve\_cnt$
```

Algorithm 12 flops on Cascade Lake

```
$precision$ = [SINGLE, DOUBLE]
$prefix$ = FP_ARITH_INST_RETIRED:
$FP\_scalar\_cnt$=prefix_SCALAR_$prec$
$FP\_128b\_cnt$=prefix_128B_PACKED_$prec$
$FP\_256b\_cnt$=prefix_256B_PACKED_$prec$
$FP\_512b\_cnt$=prefix_512B_PACKED_$prec$
$FLOPs = FP\_scalar\_cnt$
+ 2*$FP\_128b\_cnt$
+ 4*$FP\_256b\_cnt$
+ 8*$FP\_512b\_cnt$
if $precision$ == SINGLE
    $FLOPs$ = 2*$FLOPs$
```

We use our Fp Crunch benchmark to validate the hardware counter floating-point operation measurements. Figures 9 and 10 that for additions, multiplications, and divisions, the hardware counters and expected flop counts agree. However, the cosine and logarithms do not. Although cosine (and other trigonometric operations) and logarithms are mathematically single operations, most computers estimate the results with a large number of other floating-point operations. We do not have a method to consistently estimate the cost of performing such operations in scientific applications, but users can make use of our fp crunch benchmark to better understand their impact.

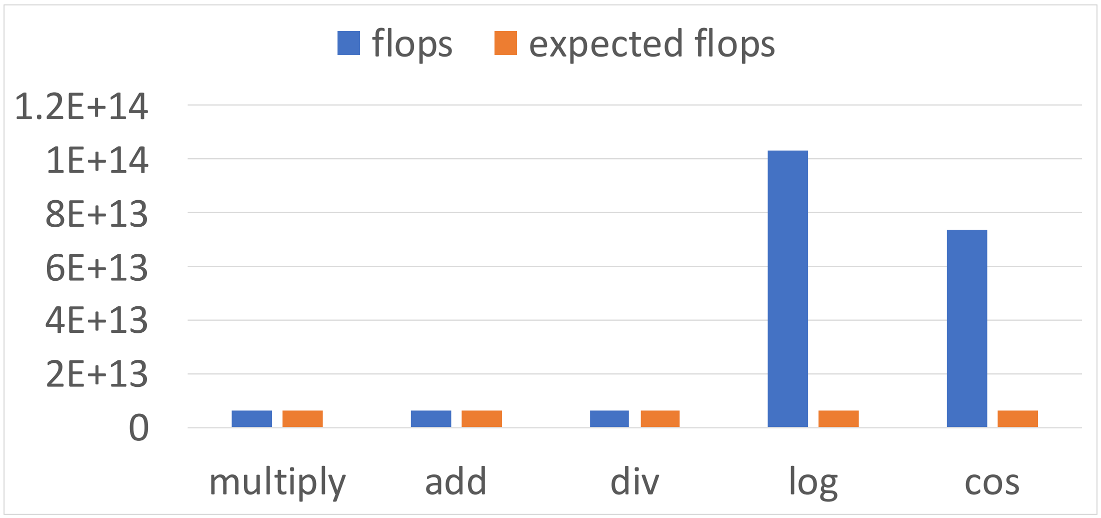


Figure 9. Comparison of expected and measured floating point operations for different versions of the FP Crunch benchmark running on the A64FX.

4.3 Examples

In this section, we combine the architectural rooflines from Section 4.1 and the application points from Section 4.2 into examples of how a user can make full use of the Roofline model for performance analysis.

4.3.1 N Body Example. Consider the N Body kernel that we introduced in Chapter III. For this example, we build and run the application

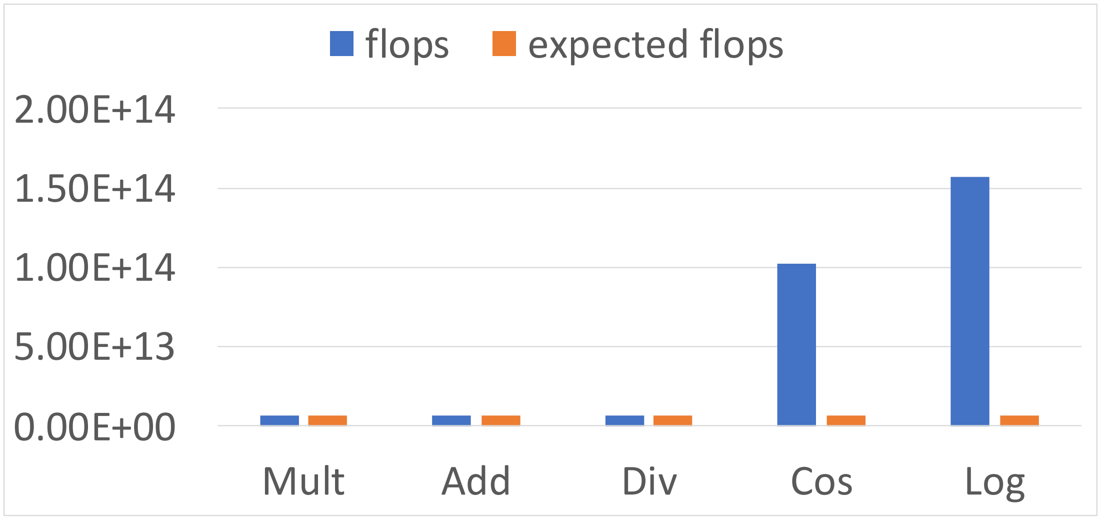


Figure 10. Comparison of expected and measured floating point operations for different versions of the FP Crunch benchmark running on the Cascade Lake.

without compiler optimizations and then with compiler optimizations. The optimizations we used aggressively target SIMD operations on the two systems since those operations are vital to reach peak floating-point performance.

Figure 11 shows the A64FX Roofline derived from our benchmarks and application points that we measured with hardware counters on the same CPU. As expected, the *N Body simd* version of the kernel performs much better than the *N Body scalar* version. We see the same pattern in Figure 12.

In addition to the higher computation rate, the *N Body simd* also has a higher **AI** on both of the processors. We did not expect the **AI** to change with the optimizations because the number of floating-point and data movement operations in the source code remain unchanged.

Table 8 shows the data that we used to produce the application points in Figures 11 and 12. We saw in the rooflines that the rate of computation improved with the compiler optimizations on both systems, this table shows that the **Time** does improve in both cases as well. On the A64FX we see that the change in **AI**

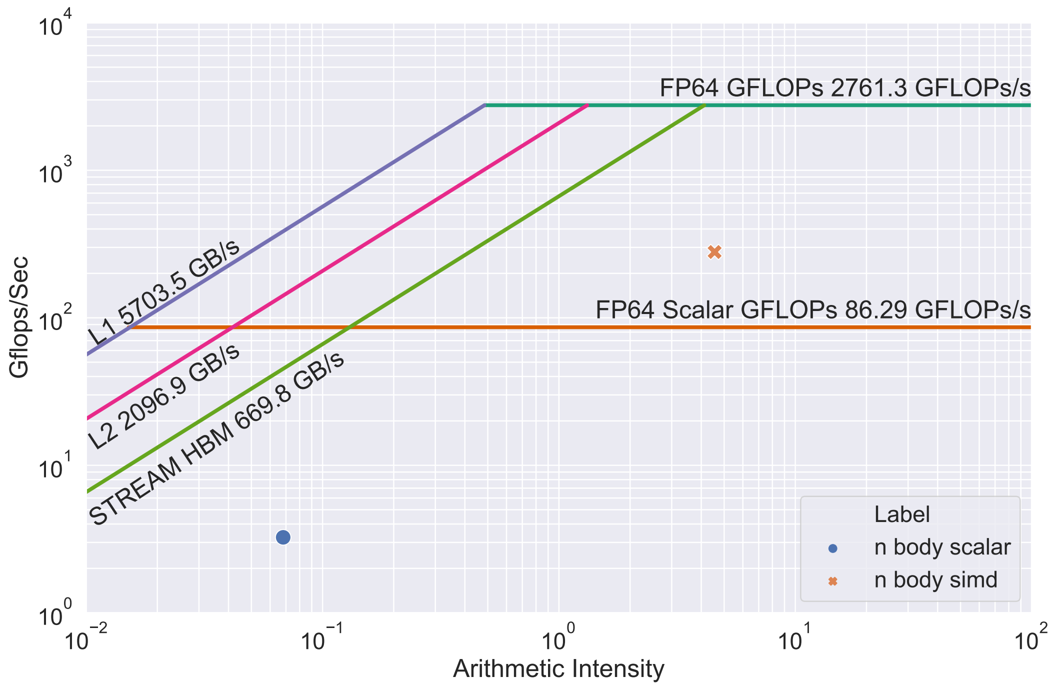


Figure 11. A64FX architectural rooflines with the application points from two versions of N Body run on that CPU.

Table 8. Roofline application point data for N Body on two systems.

	A64FX scalar	A64FX simd	Clake scalar	Clake simd
Time (s)	421.7	35.8	11.2	2.98
flops	1.37E+12	10.0E+12	1.37E+12	1.37E+12
LS Bytes	20.2E+12	2.20E+12	16.2E+12	2.58E+12
AI	0.07	4.56	0.08	0.53

is caused primarily by a combination of a 9× reduction in **LS Bytes** and a 7.3× increase in the **flops** when the compiler optimizations are applied. On the Cascade Lake the change in **AI** is smaller because the **LS Bytes** is only 6.3× lower with the optimizations and **flops** remains the same. These results are unexpected because the **AI** changed without the algorithm changing.

4.3.2 Matrix Multiplication Example. As a second example, we select the matrix multiplication kernel from Ch. III and consider four versions

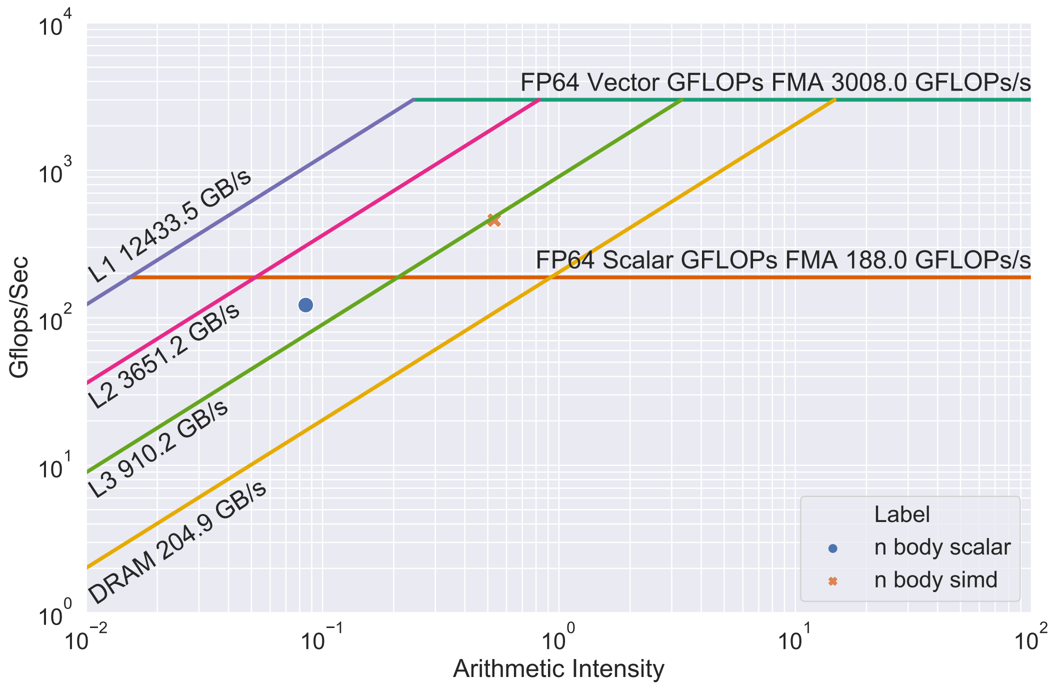


Figure 12. Cascade Lake architectural rooflines with the application points from two versions of N Body run on that CPU.

of the kernel. We look at the default, transpose, unroll, and block versions. but only use the best performing block size for each system. With this example, we can demonstrate how different optimizations impact the performance relative to the potential of the system.

Figure 13 shows the A64FX Roofline derived from our benchmarks and application points that we measured with hardware counters on the same CPU. The *default* version performs far below any of the architectural lines. Based on this result and our knowledge of the application, we expect that this version is not effectively using the memory subsystem. The *transpose* and *unrolljam* version achieve performance close to the main memory roofline, while the *block 64* version has a large improvement to the arithmetic intensity. It is likely that the blocking enables the application to perform more operations without loading new data. More analysis can be seen in Ch V with our other hardware counter metrics.

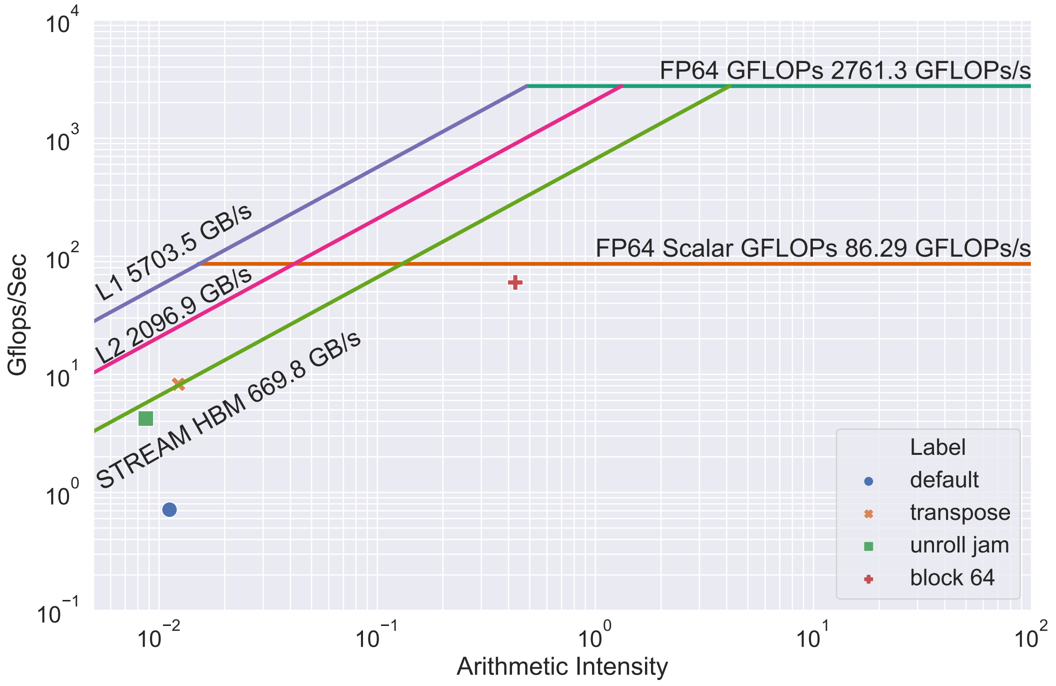


Figure 13. A64FX architectural rooflines with the application points from four versions of the Matrix Multiplication run on that CPU.

The Cascade Lake results for the same set of matrix multiplication versions are shown in Figure 14. One major difference is that the *transpose* version outperforms the *block 64* version and has a higher **Arithmetic Intensity**. We suspect that the difference is caused by how effective the compilers are at optimization on each system. This difference underscores the importance of trying different optimizations with different compilers to ensure that the best result is achieved.

4.4 Summary

This chapter describes how we use hardware counters to enable empirical performance analysis with the Roofline Model. This model can describe the

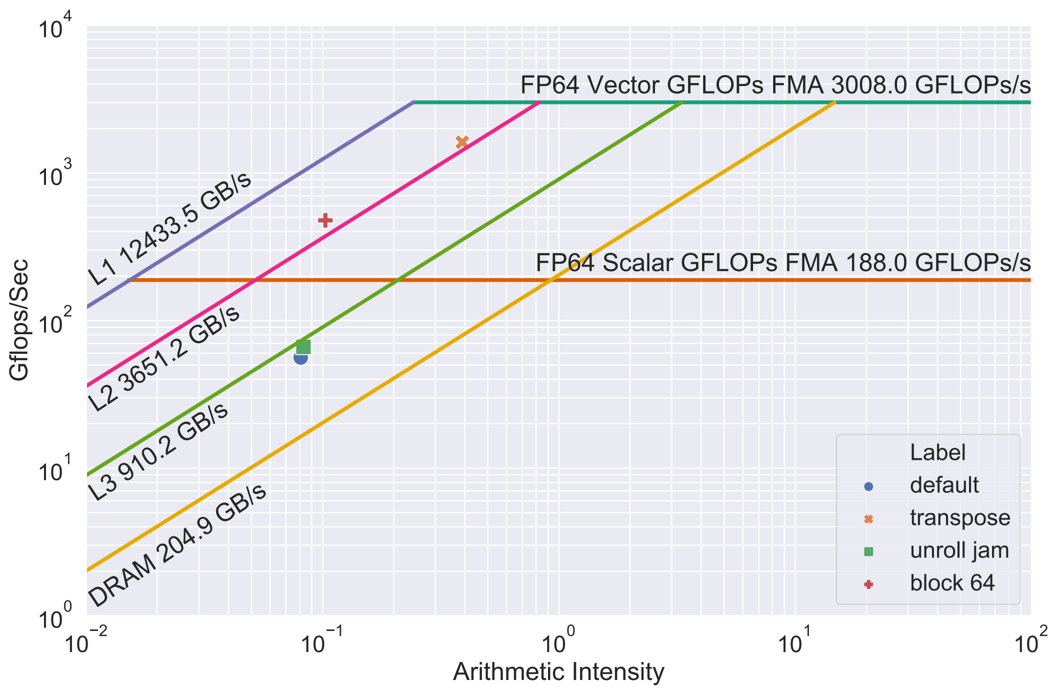


Figure 14. Cascade Lake architectural rooflines with the application points from four versions of the Matrix Multiplication run on that CPU.

potential performance of any modern CPU based on two metrics; the **Arithmetic Intensity** and **flops/s**. Since these metrics apply to all modern CPUs, performance analysis with the Roofline Model fits the portability requirement from our hypothesis. We use empirical hardware counter measurements to support the roofline analysis in two ways. First, the hardware counters can validate our benchmarks which produce the architectural rooflines. Second, the counters measure the **Arithmetic Intensity** and **flops/s** in applications which allow the user to plot application points on the architectural roofline. In the next chapter, we will build on this work to identify measurements that are portable and provide additional performance information.

CHAPTER V

A PROPOSED SET OF HARDWARE COUNTER METRICS

In this Chapter, we present additional performance metrics that inform users about common CPU features. We also present hardware counter techniques for measuring each of these metrics. As a result we demonstrate that additional CPU-agnostic metrics be collected which add performance information in addition to what is provided by the Rooflines.

To use the Roofline Model, we defined a set of hardware counter metrics based on the information we needed about an application. We then found hardware counters that fit those needs. We expand this approach to gain additional performance information. Specifically, we focus on how applications are impacted by hardware features common to most modern CPUs. By focusing on these common features we can develop hardware counter metrics that are portable to different architectures. This chapter describes the metrics we found, which are able to provide actionable information about the application and are not tied to specific hardware counter implementations.

Hardware Performance Monitors (often called counters) can provide detailed insight into how software is using the hardware but can be difficult to relate back to the application in question. Additionally, comparing counters and metrics derived from them can be difficult between systems. Vendors such as Intel, AMD, and ARM all use different counter organizations. These different designs can focus on different aspects of execution, such as instructions, cycles, or stalls. Each is useful in its own way but learning each new system and making comparisons between systems can be difficult.

These differences pose a challenge to users who are focused on software development or scientific results and do not have the time to study the differences in systems and counter definitions. Our research has been aimed at finding useful commonalities between these different counter systems and developing metrics that overlap despite the differences in design. This section presents and demonstrates a series of metrics that can be used across Intel and ARM systems. We also discuss the limitations of the metrics and areas where they can be extended.

5.1 Defining Metrics

The metrics we settled on (Table 9) are a combination of new and existing metrics that can be used to conduct performance analysis. Each of the metrics is chosen because it provides information that can be impacted by changes to the kernel or compiler options.

5.2 Measuring the Metrics

We derive the metrics in Section 5.1 on both systems from hardware counters. Chapter III contains a detailed description of the collection method, and this section describes the counters required and the process for computing the metrics. Identifying and validating these counter sets is a time consuming process. Each system has a different hardware counter architecture design and a different microarchitecture. The derived metrics need to be flexible enough to describe the different microarchitectures and be derived from different sets of counters.

From a measurement and analysis perspective, the metrics in Table 9 fall into three categories. First, there are metrics that measure the data movement through the memory hierarchy, such as bytes moved to or from different levels of the cache and main memory. This category includes the metrics that measure the amount of data and the cache efficiency. Second, there are metrics that measure the

Table 9. The Proposed set of existing and new metrics.

Metric	Definition
Overall Performance	
Time	Wall clock time for the region of interest, not a hardware counter
flops/s	floating point operations per second; a rate of computation
IPC	Instructions per cycle, measuring how well the application uses instruction level parallelism
Data Movement	
LS Bytes	Number of bytes moved between the CPU and L1 Cache as measured by load-store (LS) instructions
[L2, L3] Bytes	Number of bytes requested by the L1 cache based on counts of missed accesses to the L1 or L2 Cache.
Mem bytes	Number of bytes moved to and from main memory.
Cache Efficiency	
[L1,L2,L3] MR	Ratio of missed to total accesses for each Cache.
[L2,L3] / LS	Ratio of **L2 Bytes** or **L3 Bytes** to **LS bytes**; to help understand how much the kernel relies on the the L2 or L3 cache for data movement.
MEM / LS	Ratio of **Mem bytes** to **LS bytes**; to help understand how much the kernel relies on the main memory for data movement.
Computation	
flops	Number of floating-point operations; an amount of computation
flops/fpins	Ratio of FP operations to FP instructions; indicating how well SIMD instructions are used by the kernel.
Computation Data Rates	
AI	Arithmetic Intensity; the ratio of flops to **LS bytes**
LD ins/ST ins	Ratio of Load to Store instructions.
flops/[LD, ST] ins	Ratio of floating-point operations to load or store instructions; indicates how much computation happens per load or store operation.
flops/[LD, ST] bytes	Ratio of floating-point operations to bytes loaded into or stored from the CPU; how much computation happens per volume of data loaded or stored.

computation, total instructions, FP instructions, and FP operations. These two can be combined into the third category, which we call data computation ratios. Data movement, computation, and their relationship are the most prominent factors determining the performance of computationally intensive kernels. The required hardware counters and formulas for computing our metrics are included below.

Where possible, we have tried to make the metric derivation intuitive based on the metric names. For example, the miss rates are ratios of the misses to a cache (counters ending in DCM or TCM) to the total accesses to that layer. Some metrics are also named based on the derivation (flops/LD bytes) to guide computation based on previously computed metrics. The remainder of this section explains the metrics and derivations necessary to measure our metrics.

The reader may notice that some of the information overlaps with Chapter IV. We include the metrics needed for the Roofline in the Table 9 metrics, so we also repeat the description of the computation for the sake of completion.

5.2.1 Counter Collection and Analysis for Data Movement Metircs. Our memory subsystem metrics can be computed by measuring the amount of data moved between the CPU and L1 cache, between adjacent levels of cache, and between the last level of cache and main memory. For some of the measurements, we also use computation-related hardware counters to determine the size of the data elements moved. Once the amount of data moved at each level of the memory hierarchy is computed, we can calculate the cache efficiency metrics. Tables 10 and 11 list the hardware counters used to collect data for all the data movement metrics in this paper.

The data movement between the L1 cache and the CPU (**LS bytes**) is vital for understanding an application, but it can be challenging to compute.

Table 10. PAPI [1] Counter names used to calculate the data movement on Cascade Lake ([prefix] is FP_ARITH_INST_RETIRED:)

PAPI_[LD,SR]_INS	Executed load and stores
PAPI_[L1,L2,L3]_TCM	Total cache misses for each cache
PAPI_L1_DCM	L1 Data cache misses
[prefix]SCALAR_[SINGLE,DOUBLE]	Scalar FP instructions
[prefix]128B_PACKED_[SINGLE,DOUBLE]	128 bit SIMD FP instructions
[prefix]256B_PACKED_[SINGLE,DOUBLE]	256 bit SIMD FP instructions
[prefix]512B_PACKED_[SINGLE,DOUBLE]	512 bit SIMD FP instructions
skx_unc_imc[0,1,2,3,4,5]:: UNC_M_CAS_COUNT:[WR,RD]:cpu=[0,24]	reads and writes to main memory

Table 11. Counters used to calculate data movement on A64FX

[LD,ST]_SPEC	Architecturally executed loads and stores
ASE_SVE_[LD,ST]_SPEC	Executed loads and stores into FP registers
FP_[LD,ST]_SPEC	Executed loads and stores into scalar FP registers
L1D_CACHE	L1 Data cache accesses
PAPI_L1_DCM	L1 Data cache misses
L2D_CACHE	L2 Data cache accesses
PAPI_L2_DCM	L2 Data cache misses
PAPI_L2_DCH	L2 Data cache hits
L2D_CACHE_REFILL	Loads from main memory
L2D_CACHE_WB	Stores to main memory

Cascade Lake counts load-store instructions together regardless of the size of data moved. We estimate the amount of data movement by computing the ratios of FP instructions and assuming that the data movement operates with similar ratios. See Algorithm 13 for a mathematical representation. So far, the assumptions made have produced results that match the analytical analysis of the data movement, but more work is necessary to identify cases where the assumption breaks down and address them appropriately. The A64FX is far simpler because it provides counters for loads and stores that differentiate the size of data being moved (see Algorithm 14).

(L2, L3) bytes are computed as the cache line size (64 bytes on Cascade Lake and 256 bytes on A64FX) multiplied by the number of misses from the level above. This metric is an imprecise measurement of the bandwidth used, but it shows how much an application relies on a particular cache level, especially when compared across different kernel versions. Data movement to and from main memory (**Mem bytes**) is more precise, found by multiplying the cache line size by the number of operations that touch main memory. Main memory operations are the sum of the main memory counters in Tables 10 and 11.

Algorithm 13 LS bytes on Cascade Lake

```
1: $prec$ = [SINGLE, DOUBLE]
2: $FP\_scalar$=FP_ARITH:INST_RETIRED_SCALAR_$prec$
3: $FP\_128b$=FP_ARITH:INST_RETIRED_128B_PACKED_$prec$
4: $FP\_256b$=FP_ARITH:INST_RETIRED_256B_PACKED_$prec$
5: $FP\_512b$=FP_ARITH:INST_RETIRED_512B_PACKED_$prec$
6: $LS\_INS$ = PAPI_SR_INS + PAPI_LD_INS
7: $FP\_tot$=$FP\_scalar\_cnt$+$FP\_128b\_cnt$ +$FP\_256b\_cnt$+$FP\_512b\_cnt$
8: $SCLR\_BYTE$ = 4 * $FP\_scalar/FP\_tot$ * $LS\_INS$
9: if $prec$ == SINGLE
10:     $SCLR\_BYTE$ = $SCLR\_BYTE$ *2
11: 128_$BYTE$ = 16 * $FP\_128b/FP\_tot$
12: 256_$BYTE$ = 32 * $FP\_256b/FP\_tot$
13: 512_$BYTE$ = 64 * $FP\_512b/FP\_tot$
14: $LS\_BYTES$ = $SCLR\_BYTE$ + 128_$BYTE$ + 256_$BYTE$ + 512_$BYTE$
```

Having computed the amount of data moved, we use these data movement metrics to compute several data efficiency metrics. These ratios offer an important perspective on the performance because they show the efficiency of the cache system. The data sizes can be informative, but are directly related to the inputs and algorithms. For this reason, cache efficiency are necessary to understand some aspect of the performance.

Algorithm 14 LS bytes on A64FX

```
1: $precision$ = [SP, DP]
2:
3: $[LD,ST]\_ins$=[LD,ST]_SPEC
4:
5: $[LD,ST]\_fp$=ASE_SVE_[LD,ST]_SPEC
6:
7: $[LD,ST]\_sclr$ = FP_[LD,ST]_SPEC
8:
9: $Scalarbytes = 4 * (LD\_sclr + ST\_sclr)$
10: $bytes = Scalarbytes$
11: $bytes \mathrel{+}= 4 * (LD\_ins + ST\_ins - LD\_fp - ST\_fp)$
12: $bytes \mathrel{+}= 64 * (LD\_fp + ST\_fp - LD\_sclr - ST\_sclr)$
13:
14: if $precision$ == SP
15:     $bytes = bytes$ *2
```

Cache miss rates are a long-standing metric of cache efficiency, calculated as the number of misses per total accesses for a given layer of cache. The count of accesses is sometimes directly available ($L2D\_CACHE$ on A64FX), but sometimes must be assumed to be the number of Loads and Stores to the L1 cache or the number of misses from the higher level (**L2 MR** $= \frac{\textbf{PAPI\_L2\_DCM}}{\textbf{PAPI\_L1\_DCM}}$). All necessary counters are listed in Tables 10 and 11.

Once the various byte metrics (**LS bytes**,**(L2,L3) bytes**,**Mem bytes** have been computed, the ratio metrics (i.e. **L2 bytes** per**LS bytes**) can be easily computed. With these and other metrics, we aimed to make the derivation implied in the metric name.

5.2.2 Counter Collection and Analysis for Computation Metrics.

The second set of counters is those related to floating-point computation (as opposed to data movement). Tables 12 and 13 show the counters required to derive the computation related metrics along with **IPC**. Notably, the two architectures present these measurements in very different ways. The Intel

Cascade Lake offers the user a set of counters that measure the number of FP instructions, divided between the different vector lengths. In contrast, the A64FX provides a set of counters that measures **flops** directly. This difference is part of the motivation for using **flops/fpins** for measuring vectorization since it can be computed in both cases.

Table 12. PAPI [1] Counter names used to calculate the data movement on Cascade Lake ([prefix] is FP_ARITH_INST_RETIRED:)

[prefix]SCALAR_[SINGLE,DOUBLE]	Scalar FP instructions
[prefix]128B_PACKED_[SINGLE,DOUBLE]	128 bit SIMD FP instructions
[prefix]256B_PACKED_[SINGLE,DOUBLE]	256 bit SIMD FP instructions
[prefix]512B_PACKED_[SINGLE,DOUBLE]	512 bit SIMD FP instructions
INST_RETIRED:PREC_DIST	Instructions retired

Table 13. Counters used to calculate the **flops** and IPC on A64FX

CPU_CYCLES	number of cycles to complete the measured region
INST_RETIRED	Instructions retired
INST_SPEC	Instructions executed including those not retired due to mis-speculation
FP_SPEC	FP instructions executed including those not retired due to mis-speculation
FP_[SP,DP]_SCALE _OPS_SPEC	Scalable floating-point operations (assumes 128 SIMD length)
FP_[SP,DP]_FIXED _OPS_SPEC	Fixed floating-point operations (correctly counts SIMD lengths)

The computation for flops on the Cascade Lake can be seen in Algorithm 15 and consists mainly of multiplying the instruction counts by the correct number of operations. On the A64FX (Algorithm 16) the scalable SIMD counters assume a vector length of 128 bits, which must be corrected for the A64FX with 512-bit vector widths. In this case, the assumption is easy to make since all scalable SIMD on A64FX uses 512-bit registers, but that may not remain true for all architectures.

Algorithm 15 flops on Cascade Lake.

```
1: $prec$ = [SINGLE, DOUBLE]
2: $FP\_scalar$=FP_ARITH:INST_RETIRED _SCALAR_$prec$
3: $FP\_128b$=FP_ARITH:INST_RETIRED _128B_PACKED_$prec$
4: $FP\_256b$=FP_ARITH:INST_RETIRED _256B_PACKED_$prec$
5: $FP\_512b$=FP_ARITH:INST_RETIRED _512B_PACKED_$prec$
6: if $prec$ == DOUBLE
7:     $FLOPs = FP\_scalar + 2*FP\_128b + 4*FP\_256b + 8*FP\_512b$
8: else
9:     $FLOPs = FP\_scalar + 4*FP\_128b + 8*FP\_256b + 16*FP\_512b$
```

Algorithm 16 flops on A64FX

```
1: $precision$ = [SP, DP]
2:
3: $FP\_sve\_count$ = FP_[SP,DP]_SCALE_OPS_SPEC
4:
5: $FP\_fixed\_count$ = FP_[SP,DP]_FIXED_OPS_SPEC
6:
7: $FLOPs = FP\_fixed\_count + 4*FP\_sve\_count$
8:
```

The other computational metrics have straightforward derivations. **IPC** is the ratio of instruction to cycles. For instructions, A64FX primarily counts speculatively executed instructions, while Cascade Lake primarily counts retired instructions. We keep this difference in mind when comparing counters across architectures, but have not found a computationally intensive kernel where the difference has a significant impact. **Flops/fpins** can be computed with the **flops** from above and the sum of the FP instruction counters on Cascade Lake and the FP_SPEC counter on A64FX. Similarly, **flops/second** uses **flops** and wall clock time for the region of interest.

On Cascade Lake the **flops/fpins** metric is limited by how the counters handle fused multiply-add (FMA) instructions. Each FMA instruction counts as two instructions, allowing for correct counting of **flops**, but the **fpins** will be overcounted if FMAs are involved. For example, a double-precision application performing a series of fusible multiplications and additions with AVX512 will have an **flops/fpins** of 8 whether or not FMAs are used. Users can resolve this by looking at the optimization reports to determine if FMAs are in use, or they can disregard FMAs and focus on how **flops/fpins** show the use of SIMD instructions.

5.2.3 Counter Collection and Analysis for Computation Data Rate Metrics.

The Computation Data Rates are all ratios of **flops** to various measures of the data movement. We measure Arithmetic Intensity (**AI**) as the ratio of **flops** and **LS bytes**. Inspired by **AI** we derive **flops/[LD,ST] ins** and **flops/[LD,ST] bytes**. Both processors have load and store instruction counters (see the data movement derivations) for the denominator of **flops/[LD,ST] ins**. The **LS Bytes** derivation can be modified by leaving out either the **LD** or **ST** counters to derive the separate **LD bytes** and **ST bytes** results.

5.3 Detailed Metric Discussion

The above metrics (Table 9) can be divided into five types for the purposes of analysis. We consider the Overall Performance, amount of Data Movement, Cache Efficiency, Computation, and lastly the Ratio between the Data movement and Computation. Together the metrics give us insight into the application and apply it to most modern CPUs.

5.3.1 Overall Performance.

We use three metrics for measuring overall performance; the **Time**, **flops/s**, **IPC**. Many applications also have metrics of interest which can provide an application-specific performance metric as well.

The primary measure is almost always the amount of time it takes for an application to run. Simply, the **Time** of an application determines how soon the users can get results and move forward in their work. Also, users are generally changed for the execution **Time** of their applications on systems. Therefore, wall clock time is the main measure of performance in most cases of applications.

In addition to the time, the rate of computation can also be informative. We use **flops/s** and **IPC** to measure the rate of computation. The **flops/s** is the number of floating-point operations executed per second. **flops** are the traditional measure of work in scientific computing, so we use it here. In some cases, **IPC**, or instructions per cycle, is a better measure of the rate of computation, especially if most of the computation is not performed by floating-point operations. **IPC** is also a good measure of how well the CPU pipeline is used. These rates are good at identifying how well an application is using the hardware, but many algorithmic improvements can reduce the amount of work being performed. Therefore, both the total time and the rate of computation are vital for understanding the performance of an application.

5.3.1.1 Demonstration with STREAM. Consider this example of using the overall performance metrics with our modified STREAM benchmark. Our version of the STREAM benchmark (discussed in Chapter III) iterates through an array sized to fit within a particular level of the memory, the computation is repeated to ensure that each data size performs the same number of memory and floating-point operations. In this example, we run the STREAM benchmark on each layer of the memory hierarchy of both the A64FX and the Cascade Lake CPUs.

We expect that as the data size grows, from the L1 cache to the L2 cache to the L3 cache, and finally large enough to only be able to fit into the main memory, the computation will take longer and longer to complete. Our expectations are met as shown in Figures 15 (A64FX) and 16 (Cascade Lake). The leftmost plot shows that the amount of **time** required for each version increases as we progress through the levels of the memory hierarchy.

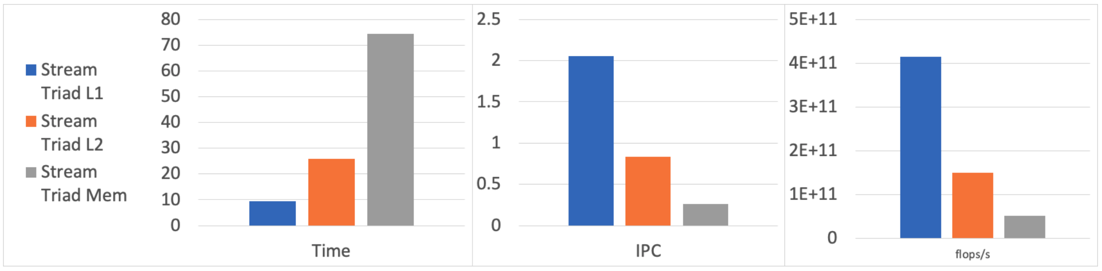


Figure 15. Overall metrics for the STREAM benchmark run at each level of cache on the A64FX.

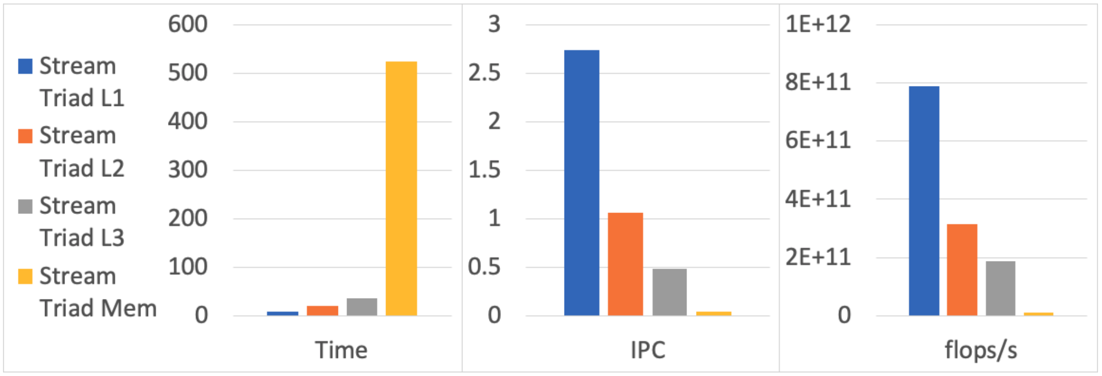


Figure 16. Overall metrics for the STREAM benchmark run at each level of cache on the Cascade Lake.

The center plots in Figures 15 and 16 show the **IPC** for each system. When data requests are filled from caches that are further from the CPU, the latency required to meet those requests increases. Therefore each load instruction must wait for more cycles, and we see the expected decrease in **IPC** from the L1 cache through the main memory.

Much like the **IPC**, the **flops/s** (rightmost of the charts in Figures 15 and 16) decreases with each layer of the cache that the experiment progresses through. For this example, the three overall performance metrics move with each other consistently because the amount of work and types of operations are held constant.

If an optimization to an application reduces the amount of work or changes the number of instructions required to complete some work (i.e. using SIMD instead of scalar operations) then these metrics may not remain correlated.

5.3.1.2 Demonstration with Matrix Multiplication. We use the matrix multiplication benchmark (details in Chapter III) to illustrate some less expected results in our overall performance metrics. We look at just the first three versions of the matrix multiplication kernel: the default, with a transposed second matrix, and with one of the loops unrolled. The transposition is expected to significantly improve the cache performance of the kernel. Unrolling can improve how the kernel uses the pipeline, but it can also interfere with compiler optimizations due to increased code complexity.

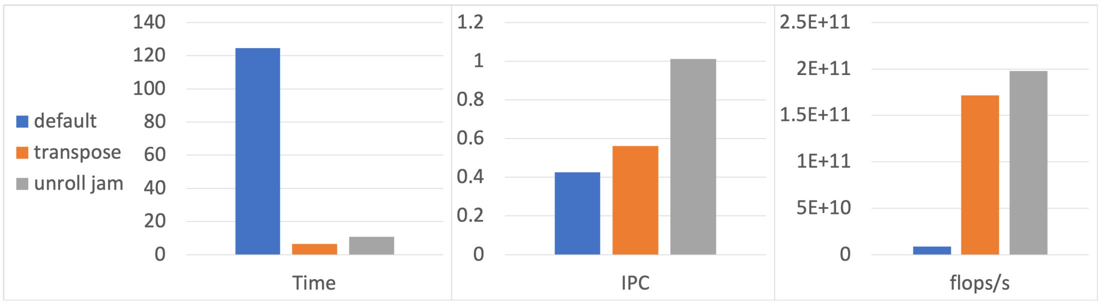


Figure 17. Overall metrics for three versions of the Matrix Multiplication run on the A64FX and using the Fujitsu compiler of computation.

Figure 17 shows the overall metrics from running these versions of matrix multiplication on the A64FX. In this matrix multiplication example, we use the Fujitsu compiler. As expected the default version performs the worst in all categories. The unrolled version also takes longer (about 1.7×) than the plain transpose version. Interestingly, the unrolled version also has higher **IPC** and **flops/s** than the transposed version. Normally **IPC** and **flops/s** correspond to better performance because the computation is faster, but in this case, the **time** is longer as well. Presumably since unrolling increases all three of the metrics it is increasing the total number of operations performed to complete the kernel. We can explore this hypothesis further using the other metrics presented in this chapter.

In contrast, the Cascade Lake (Figure 16) results show that the three metrics correlate with each other for the three variations. Since we are using the same set of metrics on both systems, we can compare the results and speculate as to a reason for the difference. One hypothesis is that the compilers handle unrolling differently. For these results, we used Fujitsu's compiler on the A64FX and Intel's on the Cascade Lake.

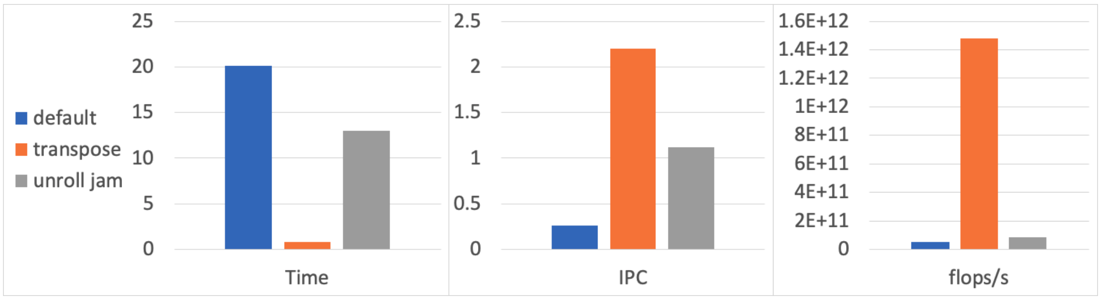


Figure 18. Overall metrics for three. versions of the Matrix Multiplication benchmark run on the Cascade Lake.

For comparison, we rerun the experiment on the A64FX with ARM's Clang-based compiler. Figure 19 shows the **Time**, **IPC**, and **flops/s** for the same Matrix multiplication version as seen in Figures 17 and 18. When using the new compiler, the overall performance metrics correlate as expected as we saw on the Cascade Lake system.

5.3.2 Data Movement. We measure data movement at each layer of cache to inform the user about the amount of data that each level supplies to the CPU for the application.

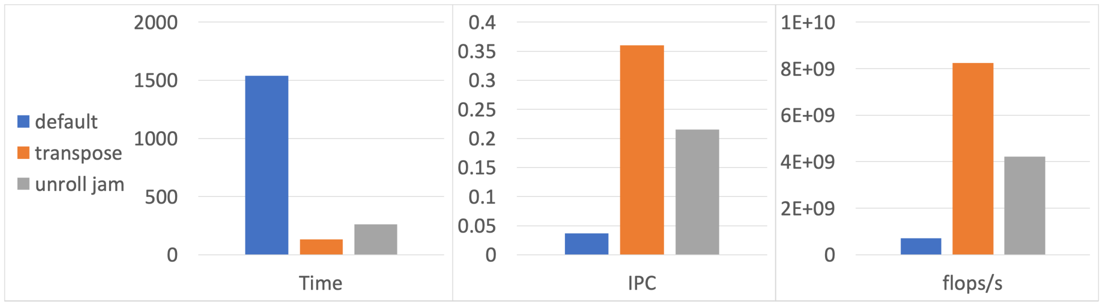


Figure 19. Overall metrics for three versions of the Matrix Multiplication run on the A64FX and using ARM's Clang-based compiler of computation.

The **LS Bytes** metric is based on our Roofline work (Chapter IV). The metric measures the bytes requested by load and store instructions which we use as a proxy for the amount of data required by the algorithm. The metric is different from the amount of data movement a user may calculate by algorithm analysis since some data may be kept in registers and additional loads and stores are necessary for array addresses and indexing. With these limitations in mind, **LS Bytes** still serves as a close proxy to the data movement needed by the algorithm under analysis.

At the other end of the memory hierarchy from the CPU is the main memory. Main memory sits off of the CPU chip and therefore has a high latency for getting data to the CPU. This latency is too high to keep the CPU working at full speed without the help of the caches. Hence, the data moving to and from the main memory can be a major limiting factor of the performance, and many optimization efforts are aimed at reducing the use of the main memory. Our precise measurements of the data movement help identify when optimizations achieve this aim.

Between the main memory and the CPU, there are caches, usually two or three on current CPUs. As noted in Ch. IV, it is difficult to define and measure the data movement of the caches. The buses that move the data into and out of each cache work for both directions, so the physical bandwidth for a cache is used for data moved between it and both adjacent caches. The total data moved on the bus includes requests and stores going in both directions for each cache. The combination of misses and hits makes such a measurement difficult to reason about, particularly in reference to the **LS Bytes** and **Mem Bytes** which each have one adjacent level of cache. Instead, we use proxy measurements to estimate the volume of data that each layer of cache supplies to the algorithm.

For our proxy measurements of **L2 Bytes** and **L3 Bytes**, we collect the number of cache misses from the next layer closer to the CPU. We assume that the data moved by cache misses will be more important to the algorithm than data moved by other features such as prefetching or cache coherency. In our view, this proxy provides sufficiently detailed information about the cache to guide a user's understanding while remaining microarchitecture independent.

5.3.2.1 Demonstration with STREAM data. For an example of the data volume measurements in practice, we use the modified STREAM benchmark from Ch. IV. The results for the A64FX are in Figure 20 and the Cascade Lake results are in Figure 21. In both cases, the data show that the **LS Bytes** are consistent across the different STREAM sizes. As we look past **LS Bytes** to **L2, L3, and Mem Bytes**, the number of bytes measured in the L1, L2, and L3 fall to near zero one after the other. We note that there is some variance in the exact number of bytes at the lower levels.

The observant reader will note that there is no "known value" for these measurements. We have not attempted to validate the measurements[1] presented

[1]The exception being **LS Bytes** which has validation in Ch. IV.

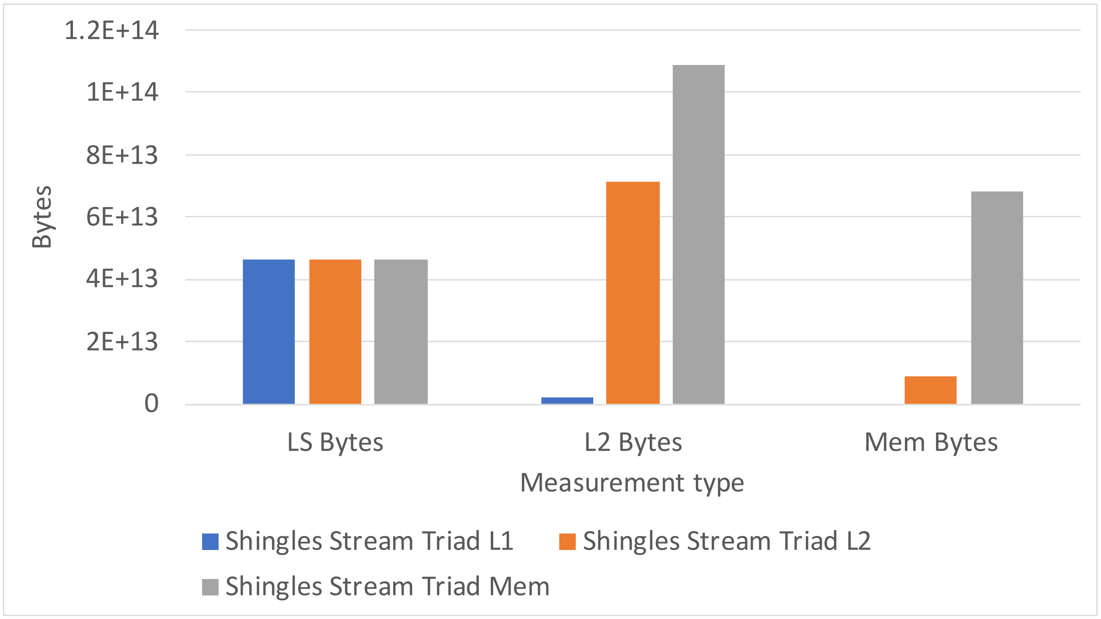


Figure 20. A64 bytes at each cache level for each STREAM size.

in Figures 20 and 21. We think such validation would give create the wrong impression for the metrics. Each byte measurement is a proxy for the amount of data moved at that level of cache and should be treated as a broad estimate. Users should consider how trends in data movement change with changes in the application or its inputs. This example improves confidence in a vague analysis method by demonstrating one case in which the measurements respond as expected.

5.3.2.2 Demonstration with Loop Blocking. The matrix multiplication kernel gives us a good opportunity to demonstrate the Data Movement metrics because the blocking optimization is designed to improve the usage of the memory hierarchy. When a nested loop is blocked it focuses on smaller sections of the data at a time so that data reuse occurs before it is ejected from the cache. We describe the optimization in more detail in Chapter III.

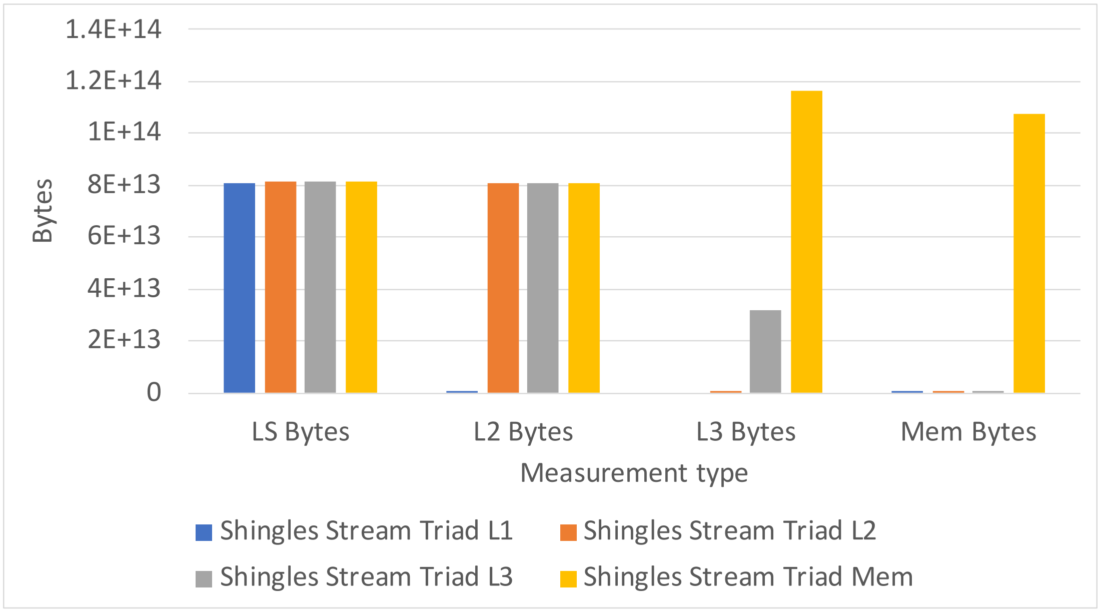


Figure 21. Cascade Lake bytes at each cache level for each STREAM size.

We use the ARM Clang compiler to build the Matrix Multiplication kernel with square matrices that have 8192 rows and columns. With double-precision floating-point operations, each row is approximately the same size as the L1 cache on the A64FX. We chose this size so that we could effectively exercise the cache. The loop blocking can be scaled to different sizes depending on the cache sizes in the system. We explore block sizes of 16, 32, 64, 128, 256, 512, and 1024.

The **LS Bytes** results are pictured in Figure 22. As the block sizes increase, the data fits better into the caches so that the kernel requires fewer operations to complete. Of particular note, the **L2 Bytes** reduce slowly from block sizes 16 to 64. At block size 256, the number of **L2 Bytes** increases sharply because the blocks are now larger than the L1 cache. To help understand these results, we include the timing results for the same experiments in Figure 23.

We repeat the experiment on the Cascade Lake system with similar results. Figure 24 show the **Byte** counts for the four levels of memory hierarchy relative to

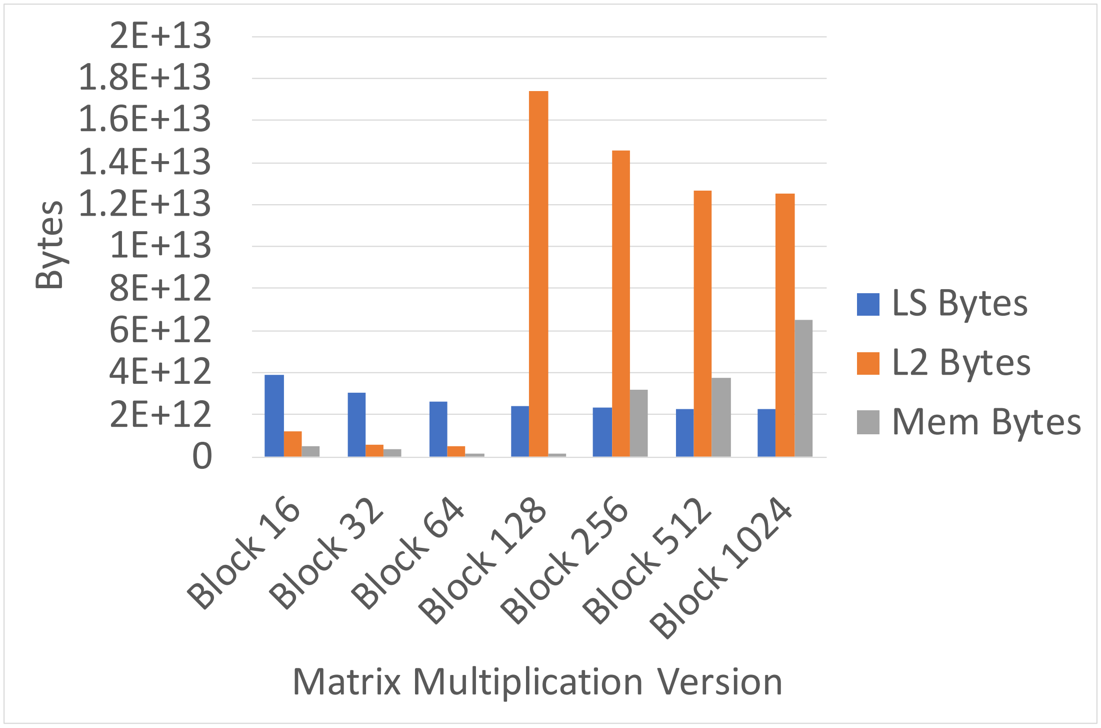


Figure 22. Data movement relative to block size for the matrix multiplication kernel on the A64FX.

the tested block sizes. The execution **Time** relative to the block sizes is pictured in Figure 25. Similar to the A64FX, the **L2, L3, and Mem Bytes** on the Cascade Lake are relatively small at each level until the blocks are large enough to fill that cache level. At that point, the number of bytes moved to the next level of cache increases rapidly.

The execution time of on the Cascade Lake (Figure 25) improves with the increased block sizes until block size 256. This point corresponds with the increase in **L3 Bytes**.

Automatic performance tuners, such as Orio [54], can test a large number of block sizes and similar optimizations. Such tests are tedious and time consuming, so the automated method of testing is a preferred approach.

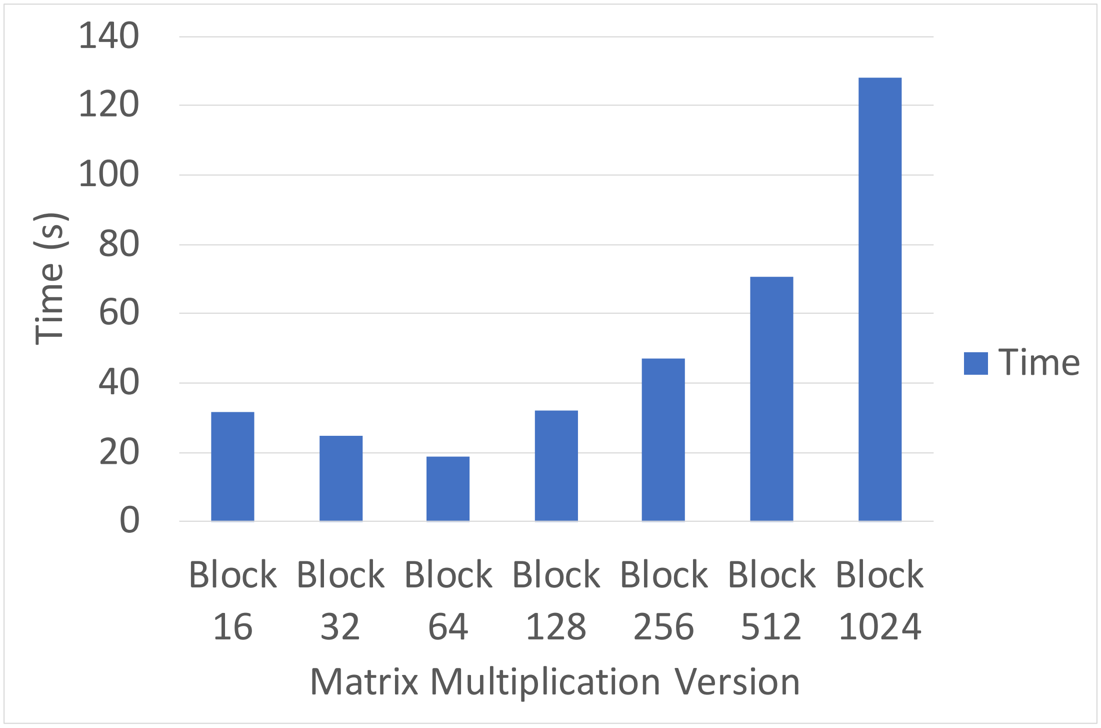


Figure 23. Execution time relative to block size for the matrix multiplication kernel on the A64FX.

5.3.2.3 Demonstration with Cache Conflicts. We gathered our data movement measurements for this cache conflict benchmark (see Chapter III) on the A64FX (Figure 26) and the Cascade Lake (Figure 27). This benchmark performs a series of computations on data elements that are separated by some offset. When this offset is a multiple of the cache block size, then cache conflicts will prevent the L1 cache from being used effectively. In both cases, we can see two spikes where the offset is aligned with the number and size of the cache blocks, causing a dramatic increase in the number of L1 cache misses.

Once again we avoid detailed validation of the results. We could present an analysis of the cache that shows where the spikes are expected, or we could run simulations of our cache hierarchy to check that the increased misses are due to conflicts and not capacity. Unfortunately, the complexities of applications make

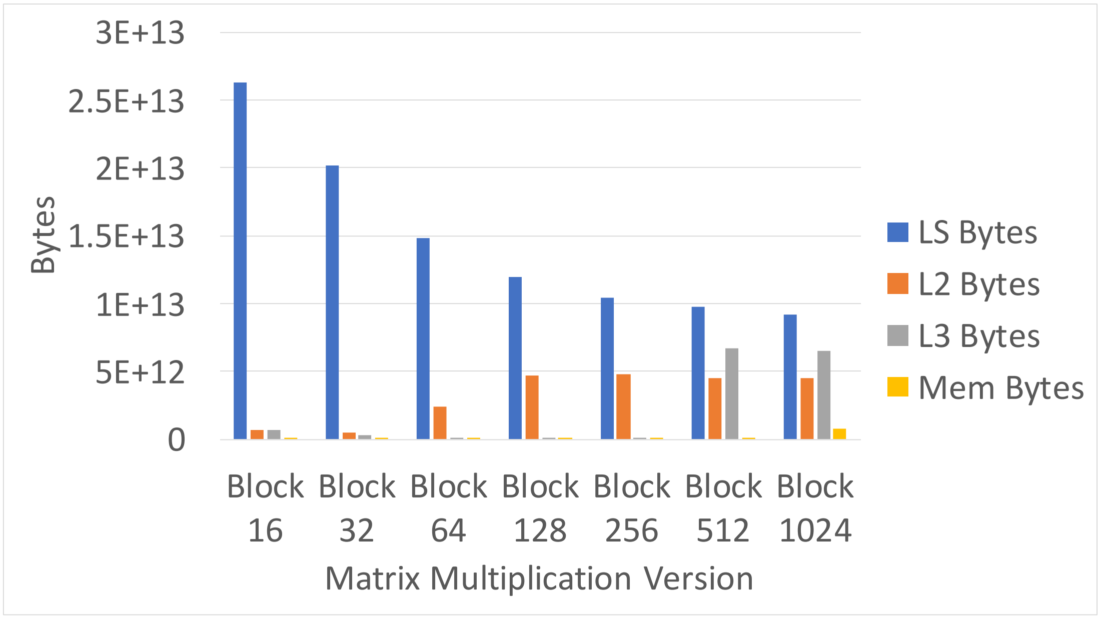


Figure 24. Cascade Lake bytes relative to block size for the matrix multiplication kernel.

such analysis prohibitive, so we strive to create metrics that do not rely on such validations.

In this case, we observe that the **LS Bytes** have almost no variation (about 10%) variation across the offsets. Further, the input size fits into the L2 but not the L1 cache on both systems. These features suggest that the cache capacity is not impacting the number of misses. Lastly, there is no change to the initialization process which minimizes the possibility that compulsory misses are impacting the data. Therefore, we conclude that it is likely that the spikes in L2 data movement occur due to conflict misses.

We find it straightforward to build experiments of this sort in the fully controlled environment of a new benchmark. The challenge is greater when examining kernels within full applications, but doing so will enable hardware counter analysis.

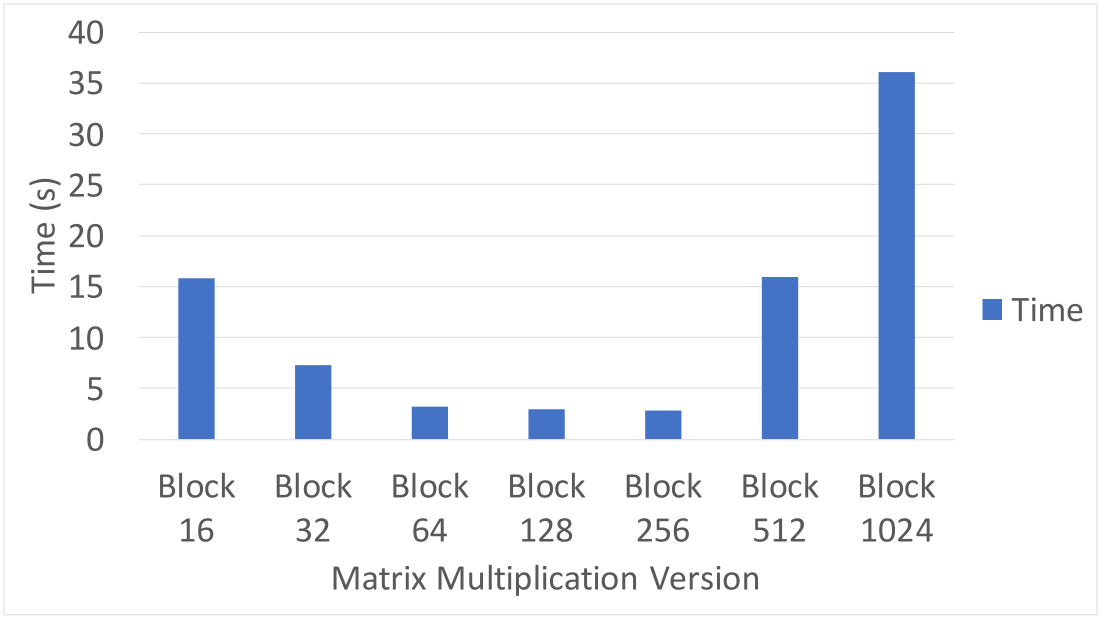


Figure 25. Cascade Lake time relative to block size for the matrix multiplication kernel.

5.3.3 Cache Efficiency.

We use two types of similar metrics to measure cache efficiency: **Miss Rates** and **Byte Ratios**. These metrics complement the volume of data moved, by showing how effectively each cache layer is used and how much the application relies on it.

Miss Rates is the ratio of misses to accesses for a particular level of cache. We discuss the details of these measurements above. Cache misses arise in three ways [55]: compulsory misses from new data, capacity misses when the data had been evicted due to a lack of space, and conflict misses when the data was evicted as the result of associativity conflicts. A low miss rate indicates that the application is reusing data repeatedly and making the most of the compulsory misses.

Distinguishing between the capacity and conflict misses is challenging to do without memory tracing, simulation, or other methods which are beyond the scope

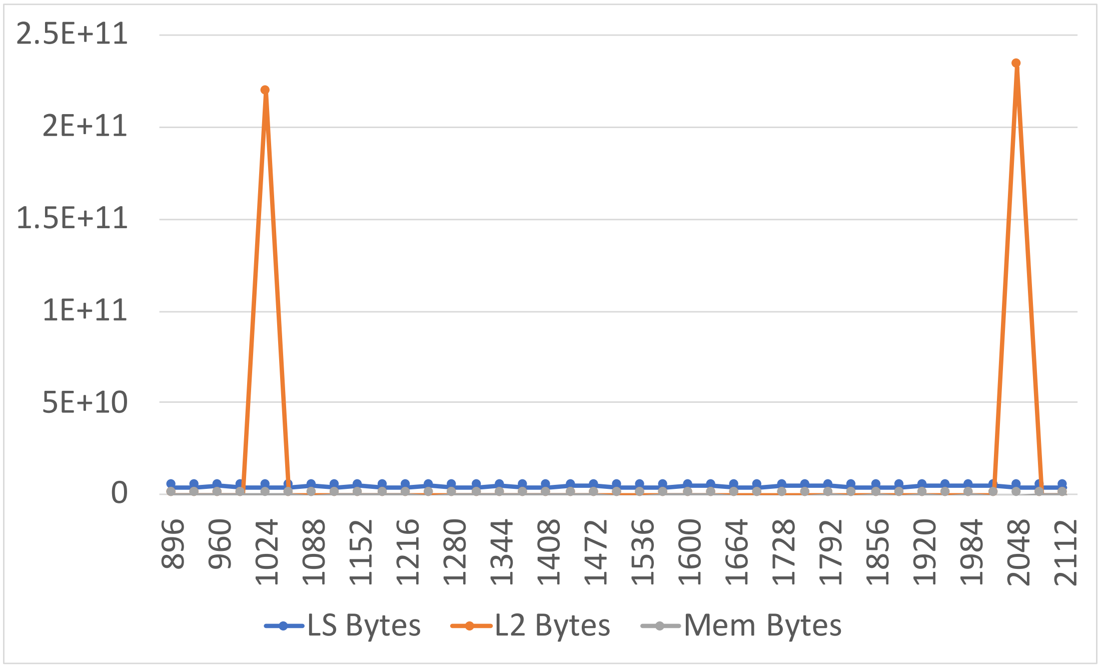


Figure 26. A64FX bytes relative to the offset in our associativity example.

of this work. The user can use their knowledge of the program to reason about the type of misses, and design experiments adjusting the inputs or data structures to see how the miss rates are impacted. Such experiments are application-specific but can provide a deeper understanding of the performance of an application. We offer some examples in Section 6.1.

The **Byte Ratio** is the ratio of data supplied by a particular level of cache to the data requested by the CPU. This metric is similar to **Miss Rate** but uses the **LS Bytes** as the denominator for all the levels of cache. In [44], the authors present a similar metric based on simulation data.

The **Byte Ratios** can be used to determine which caches are most heavily used by the application. A higher ratio indicates more reliance on the cache level, knowing how much each cache level contributes to the application is vital to know how to focus the attention of the optimizations. For example, if cache blocking is

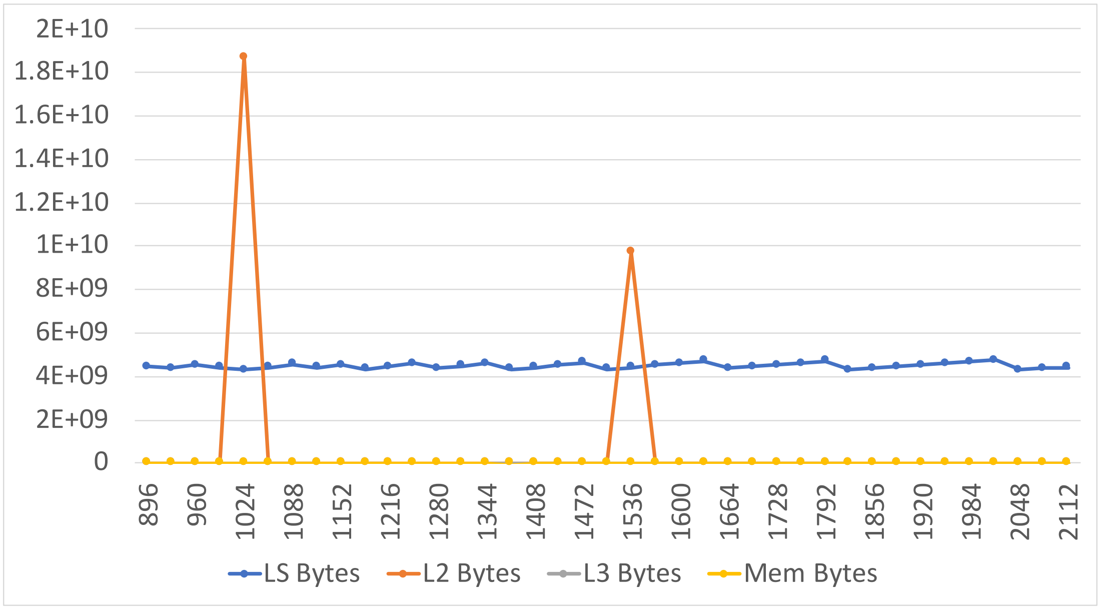


Figure 27. Cascade Lake bytes relative to the offset in our associativity example.

used to improve cache efficiency, then the metrics can be used to verify the success of the blocking and pinpoint a particular block size. We include examples of this type in Section 6.1.

5.3.3.1 Demonstration with Cache Conflicts. To demonstrate the cache efficiency metrics, we repeat the cache coherency experiment from the Data Movement metrics. Once again we ran the benchmark with a series of offsets between data accesses. The data size is close to the L1 cache, so the computation can be performed very efficiently except when cache conflicts occur because the offset is a multiple of the block size.

Figure 28 shows the byte ratios on the A64FX and Figure 29 shows the same ratios on the Cascade Lake. The higher associativity of the L2 (and Cascade Lake's L3) caches prevents the cache conflicts from impacting those caches as well, so we see spikes in **L2 Bytes / LS Bytes** but not in **Mem Bytes / LS Bytes** or **L3 Bytes / LS Bytes**. These peaks indicate that the L2 cache is used to complete a larger fraction of the total load and store operations for the kernel in those instances.

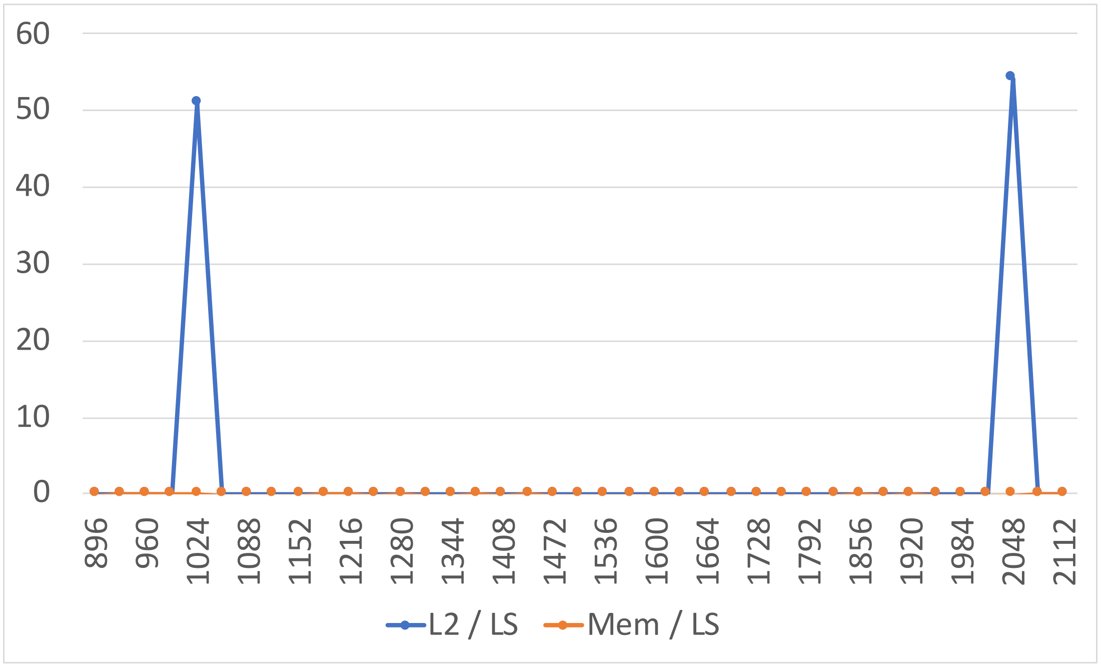


Figure 28. A64FX byte ratios relative to the offset in our associativity example.

We show the miss rates for A64FX in Figure 30) and for the Cascade Lake in Figure 31). For both systems, we see a low **L1 Miss Rate** for most of the offsets except those which correspond to the cache conflicts. Once again the cache metrics are performing as expected for this benchmark.

In this case, we also see a high **L2 Miss Rate** on both systems. A miss rate of 30% or 40% could be the indication of poor cache performance, but from our **L2 Bytes / LS Bytes** metric, we know that the L2 cache is not used for very many of the load and store operations. Therefore, we hypothesize that the high miss rate is probably caused more by the low number of accesses to that cache level than a high number of misses. Notably, at the cache conflict points, the **L2**

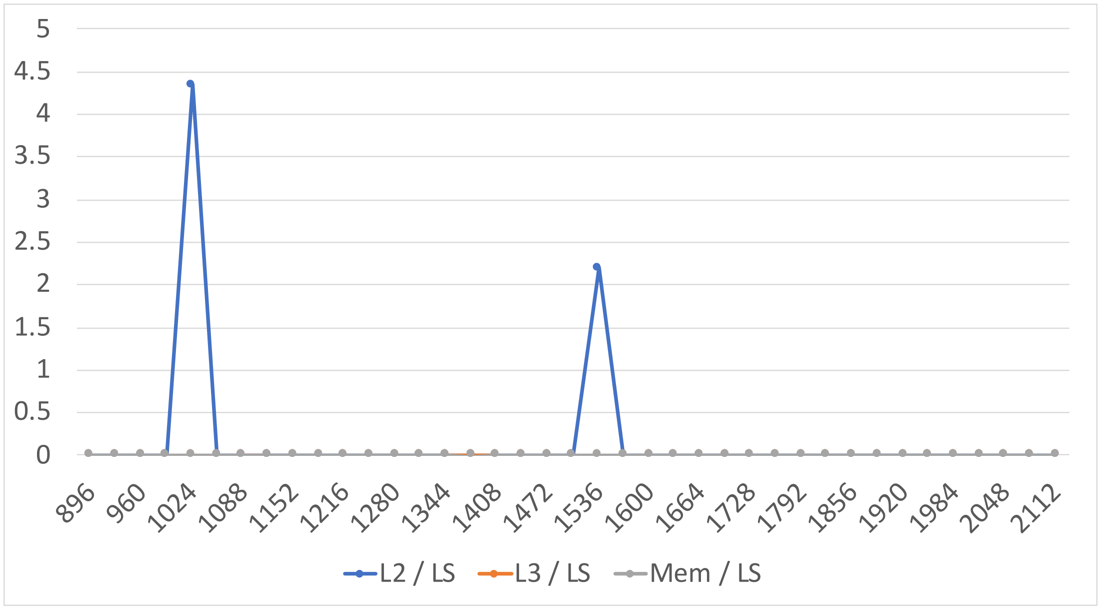


Figure 29. Cascade Lake byte ratios relative to the offset in our associativity example.

Miss Rate drops close to zero. This fact corroborates the hypothesis since the drop corresponds with a large increase in accesses to the L2 cache.

5.3.3.2 Demonstration with Loop Blocking. As with the Data Movement metrics, the Matrix Multiplication benchmark provides a good example of how the Cache Efficiency metrics respond to changes that improve the efficiency of the cache use. We run the matrix multiplication with four types of kernels: default, transpose, unroll, and blocked. The blocking version has seven block sizes which show how the performance changes as the block size varies. Details on the variations are available in Chapter III.

First, we look at the Cache Ratio metrics in Figures 32 and 33. On both of the systems, the **L2/LS Ratio** is low for the small block sizes, then rises quickly. Our hypothesis is that this rise corresponds to the block size becoming larger than the L1 Cache which causes the kernel to have more data-filled from the larger caches instead of the smaller and faster L1 cache. This result re-enforces our

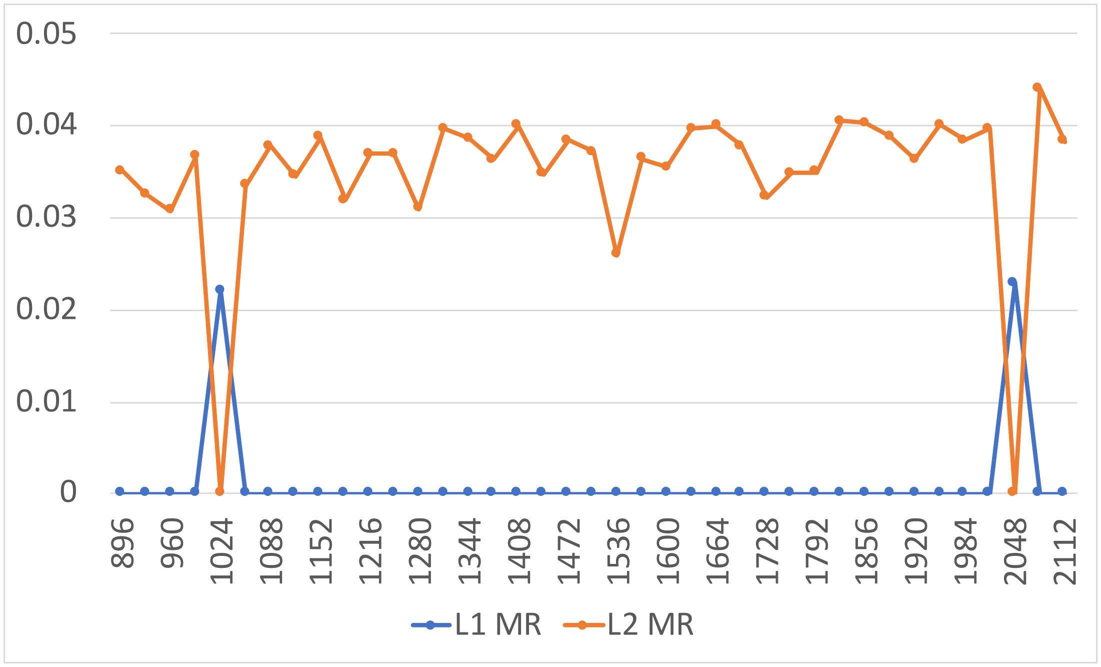


Figure 30. A64FX miss rates relative to the offset in our associativity example.

confidence that the blocking is working as intended since the smaller block sizes can fill a larger number of requests from the L1 Cache than the larger block sizes.

On the Cascade Lake system (Figure 33), we notice that the **L3/LS Ratio** and the **Mem/LS Ratio** rise significantly as the size increases further. The **Mem/LS Ratio** on the A64FX similarly rises at a larger block size than the **L2/LS Ratio**. A more fine-grained set of block sizes could help the user establish the exact point at which the block size becomes too large for each cache.

The results have two notable differences between the two systems. First, the A64FX has much higher ratios overall. We suspect that the higher ratios are the result of the larger cache line size (256 bytes on A64FX compared to 64 bytes on Cascade Lake). Inherently, any access to a lower cache is therefore moving more data. Secondly, the Default version of matrix multiplication has a higher **L2/LS Ratio** than the transpose and unroll versions on A64FX, but the three versions

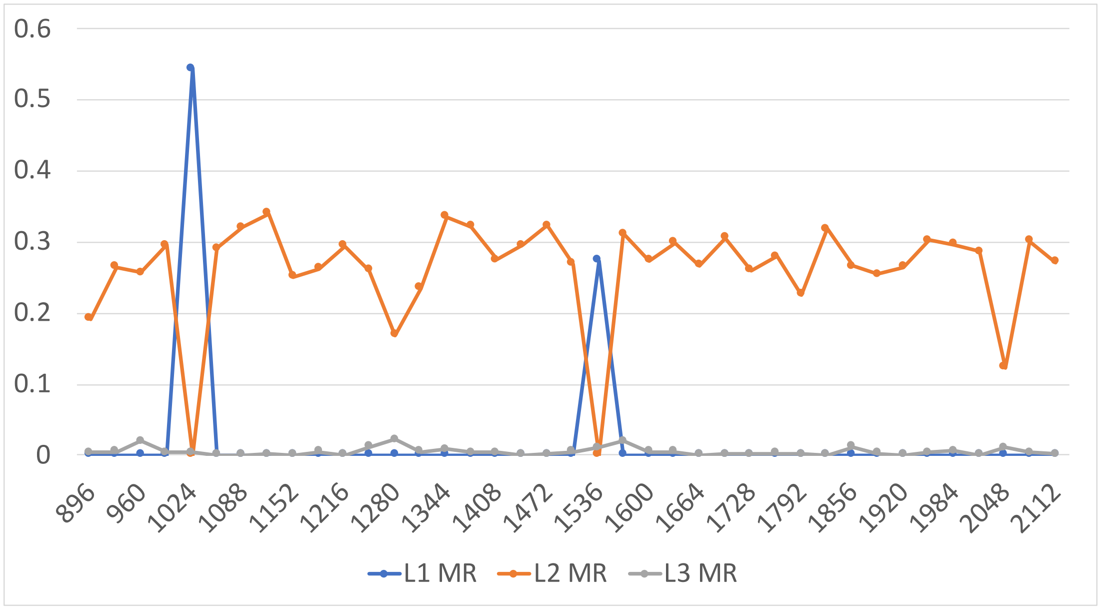


Figure 31. Cascade Lake miss rates relative to the offset in our associativity example.

have similar **L2/LS Ratios** on the Cascade Lake. Once again the cache line size is the likely culprit. The default version wastes all but one element in the cache line when loading data for the B matrix; therefore the larger cache line size on the A64FX will waste more data.

We repeat the experiment with the **L1 L2 and L3 Miss Rates**, but understanding these results requires some information from the **Byte Ratios** as well. Figures 34 and 35 present the **Miss Rates** on the A64FX and the Cascade Lake, respectively. Unsurprisingly, the **L1 Miss Rates** follow a pattern similar to the data movement and **Byte Ratio** metrics, which show that transposition, unrolling, and blocking each lower the missrate. The **L1 Miss Rates** then rise as the Block Size increases. These results add to our confidence that our blocking optimization improves the efficiency of the L1 cache.

The unrolled version of the Matrix Multiplication varies somewhat between the two systems. On the A64FX, the **L1 Miss Rate** falls from 5.1% to 0.7%

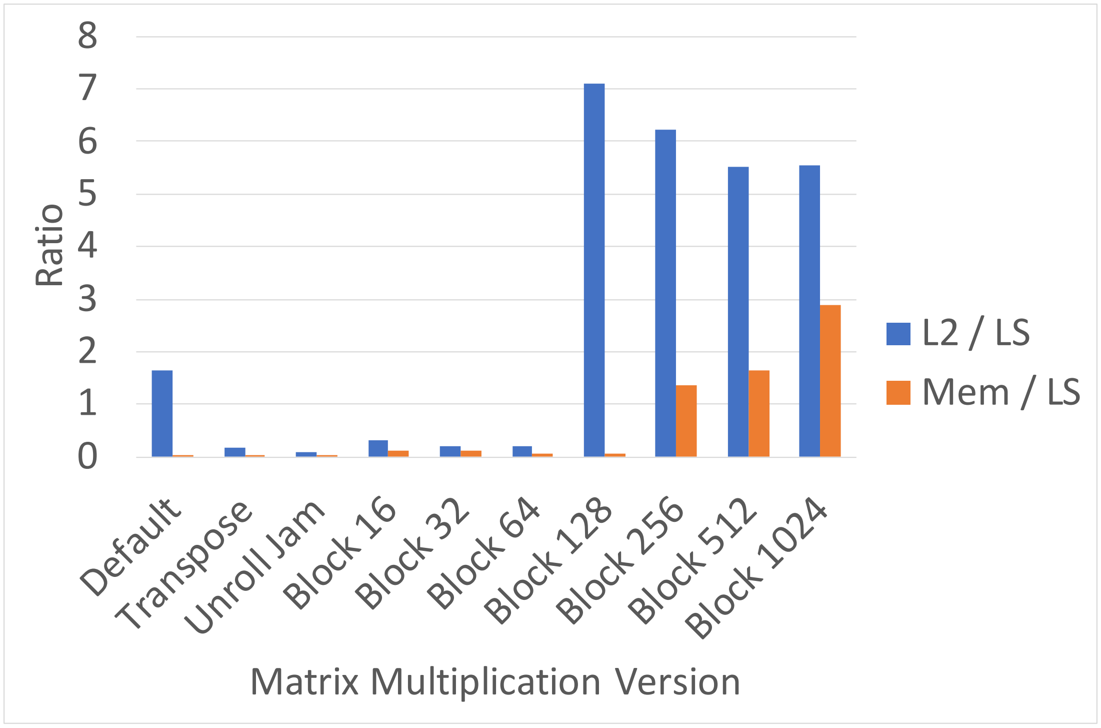


Figure 32. A64FX byte ratios for the matrix multiplication variations.

from the default to the transpose version, then to 0.5% for the unrolled version. In contrast, the Cascade Lake **L1 Miss Rate** falls from 65% (default) to 48% (transpose) to 4.8% (unroll). Although this pattern suggests that the loop unrolling is highly effective on Cascade Lake, that is a misleading interpretation. The transpose version is over 27× faster than the default version while the unroll version is only 1.2× faster. The improved miss rate is the result of a combination of factors; the Cascade Lake transpose version has a lower **LS Bytes** than either the default or the unroll version, and the use of SIMD operations changes in the unroll version.

Without examining the computation metrics out of turn, we recognize that we avoided incorrect assumptions about the **L1 Miss Rate** by considering it in conjunction with the **LS Bytes** and the **flops/fp ins**. A vital aspect of our

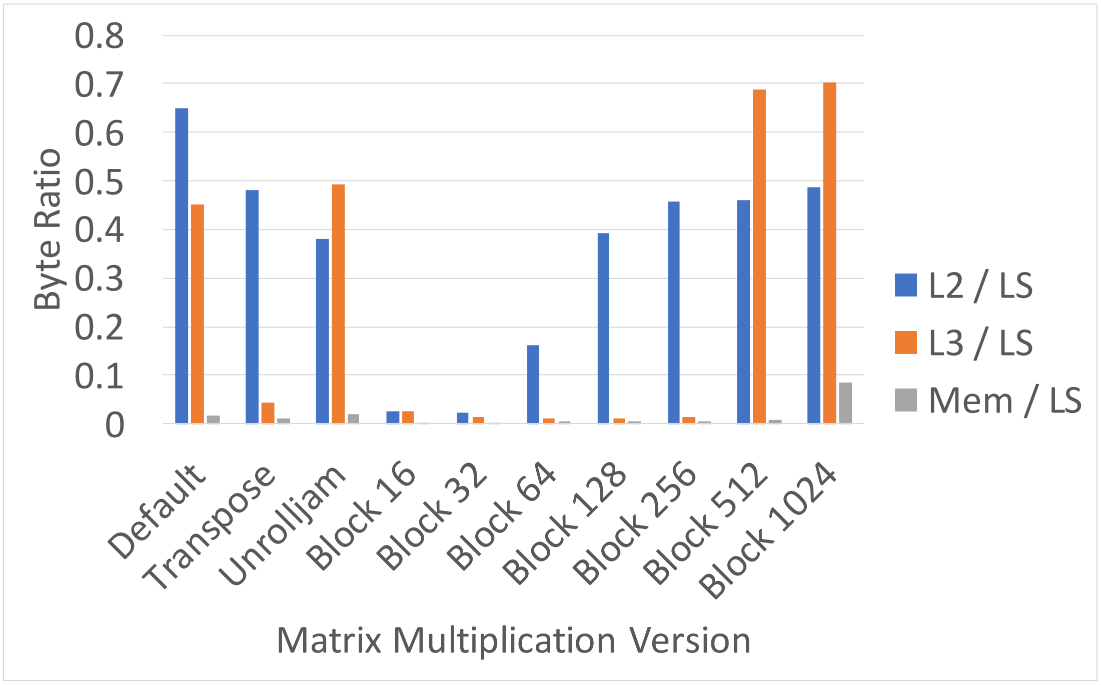


Figure 33. Cascade Lake byte ratios for the matrix multiplication variations.

metrics is that they each address issues that are left open by the other metrics. For example, the **LS Bytes** metric can inform us that the Cascade Lake unroll and transpose matrix multiplications use the L1 Cache in dramatically different ways, so we should not directly compare the **LS Bytes** for those versions. Additionally, the **flops/fp ins** informs us that the two versions have different SIMD operations usage. The **L1 Miss Rate** is based on the number of instructions that access the L1 cache, so this change has a large impact on how we interpret the **L1 Miss Rate**. We will not pretend to know all of the interactions between our metrics, but we use the example mini-applications in Chapter VI to describe our experience with some of these interactions.

Both systems have disturbingly high **L2 Miss Rates** for many of the versions. In particular, the smaller sizes of blocks have Miss Rates over 30% on A64FX and close to 100% on the Cascade Lake. However, the **L2/LS Ratio** informs us that the L2 Cache is largely unused, so these misses are probably compulsory misses required when data is touched for the first time. Additionally, we used SIMD operations, so each load operation requests 64 bytes of data. This size is the same as the cache line n the Cascade Lake, so each compulsory load from the L2 into the L1 is fully utilized by the load operation. Therefore the **L2 Miss Rate** is close to 100% on the Cascade Lake. The A64FX has a larger cache line, so more data is brought in with each of the compulsory misses which improves the efficiency of those L2 Accesses, so the A64FX **L2 Miss Rate** is closer to 30%.

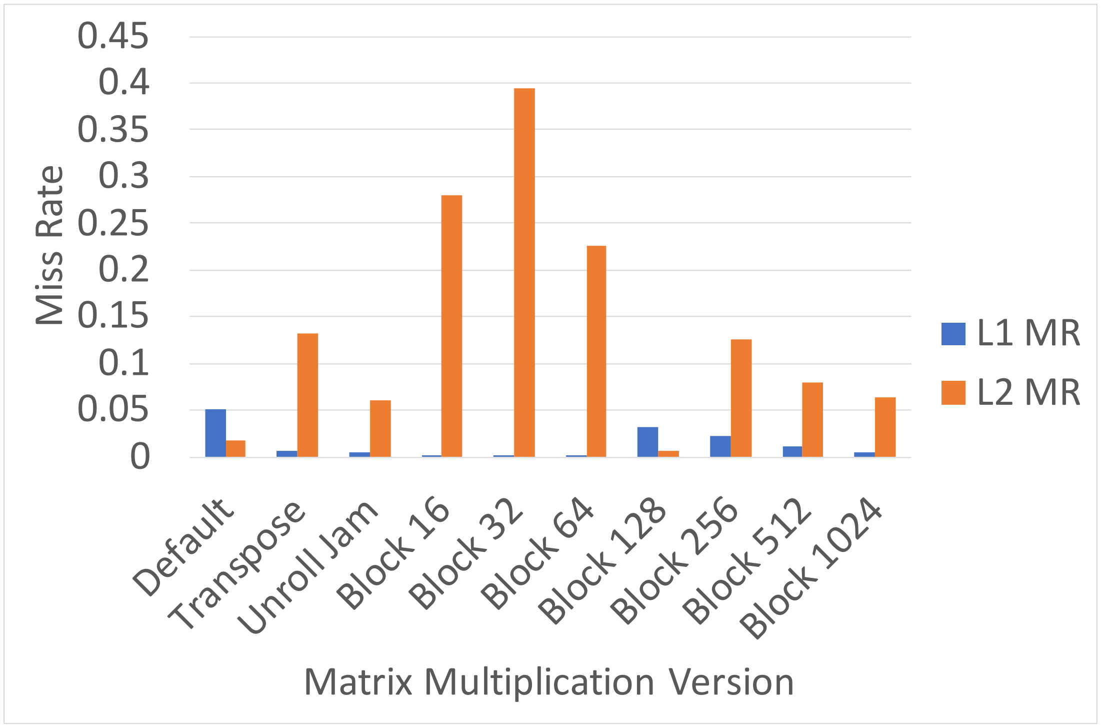


Figure 34. A64FX miss rates for the matrix multiplication variations.

5.3.4 Computation. Our research is focused on scientific computing applications where the primary measure of work is floating-point computation. Much like the data movement measurements, we provide two types of computational metrics to the user. First we look at total computation as **flops**

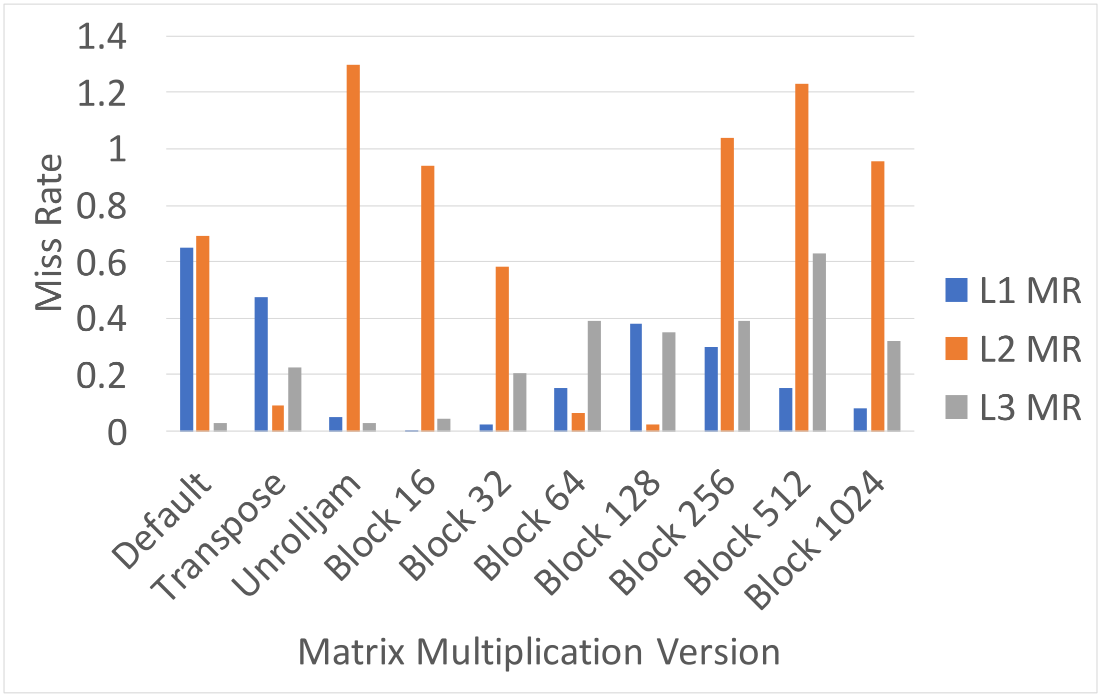


Figure 35. Cascade Lake miss rates for the matrix multiplication variations.

an efficiency measurement as **flops/ fp instruction**. These metrics help the user understand how much work the application is performing and if it is using the most efficient instructions available.

As discussed in Section 5.1, we define **flops** as the number of floating-point operations executed by the hardware. We distinguished this term from **flops/s** which is the rate of computation. Note that many authors differ on the notation. The total computation can be useful for checking how optimizations change the amount of work performed and is particularly useful when used to compare with other metrics such as time (above) and data movement (below) to measure the efficiency of the application.

Proper use of Single Instruction Multiple Data (SIMD or vector) instructions is vital to maximizing the computational speed on modern CPUs. These instructions allow the CPU to execute several identical operations at once on adjacent data. The user can check for SIMD use different ways,including our **flops/fp ins** (**flops** per floating-point instruction) metric. This metric provides an empirical method to check how well an application is using SIMD operations.

5.3.4.1 Demonstration with STREAM. Our modified STREAM benchmark is a good tool to help validate the count of floating-point operations. It is carefully designed to run a known number of operations in each experiment, which can then be compared to the measured amount from the hardware counters. In these examples, we reuse the experimental data from Chapter IV. These results provide a wide range of sizes on both systems and include data sizes that are larger and smaller than each cache.

Figure 36 shows the results from various data sizes of STREAM run on all 48 cores of the A64FX. The results show that for all of the sizes there is less than a 0.5% difference between the expected **flops** and the measured value. The error could arise from many factors including the overhead of Caliper or Papi, variations in how the overflow of counters is handled, or how the compiler handles the potential remainder loops required by vectorization of the computation. We are satisfied with this error level because it is significantly more precise than the cache measurements which are based primarily on proxy measurements.

We present similar results for the Cascade Lake CPU in Figure 37. Once again these data demonstrate the percent differences are less than 0.5% for all of the data sizes.

5.3.4.2 Demonstration with Matrix Multiplication. We use the matrix multiplication benchmark to study what our computation metrics tell us about a kernel with more complex loop nesting and data structures. The kernel is simple in comparison to many other computation types, but the three nested

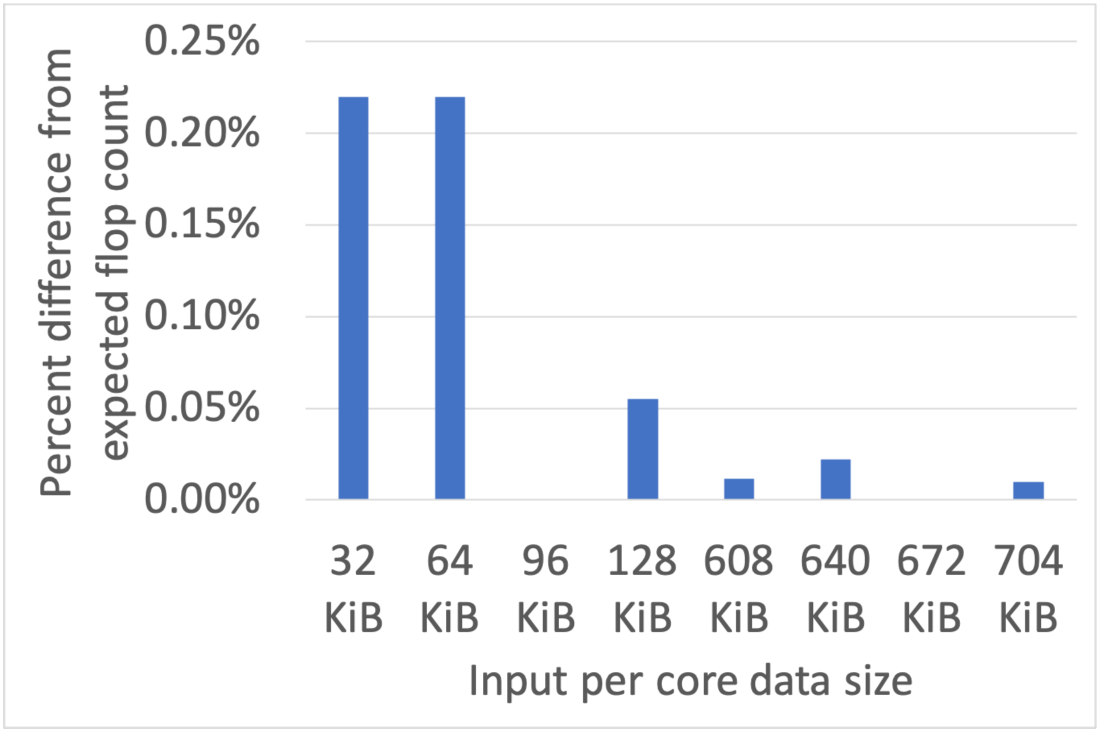


Figure 36. Percent difference between the expected and measured floating point operations for different sizes of STREAM run on the A64FX CPU.

loops, 2D matrix data structures, and many optimization options represent a major increase in complexity from the STREAM benchmark.

In principle, the number of floating-point operations in the matrix multiplication kernel should not need to change with any of our operations. The transpose, unrolling, and blocking, versions of the kernel all use the same number of operations if the user were to count operations per data element in the source code representation. We will see in this example that compilers can make unexpected changes to the amount of work performed by a kernel, further motivating the need for empirical and static analysis.

We show the count of Floating Point operations (**flops**) in Figure 38 and 38. On both systems, the number of **flops** is constant for the non-blocked versions of matrix multiplication. However, once blocking is applied, the number of floating-

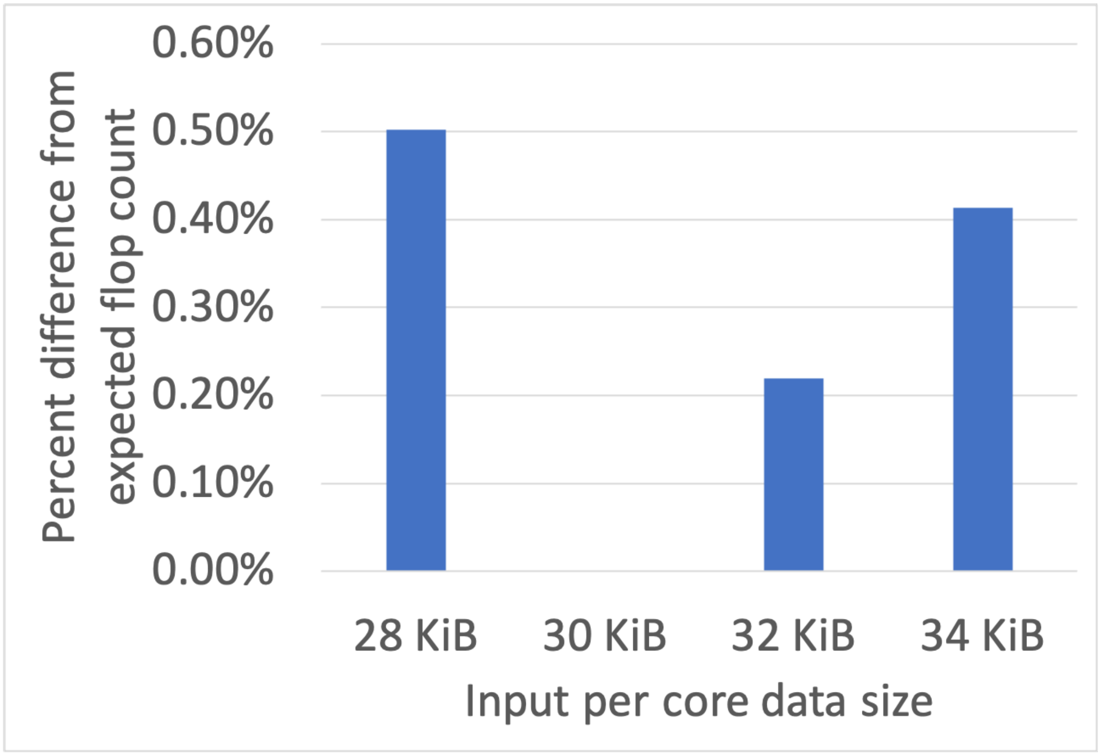


Figure 37. Percent difference between the expected and measured floating point operations for different sizes of STREAM run on the Cascade Lake CPU.

point operations increases by 25% on the A64FX and by close to 100% on the Casacdelake. These increase the trend downward as the block size increases, and the **flops** appears to be close to the same as the default version for the largest block sizes.

At first, we suspected that this change in **flops** indicated that we were not measuring the floating-point operations correctly. Further examination of the assembly language revealed that the compiler modifies the algorithm slightly to perform multiple operations that need to be summed into one of the elements of the C matrix. Once these operations are separated, additional reductions are needed to sum the temporary C elements into the final C element. The reduction occurs at the end of one of the block loops, which means the number of added reduction operations is inversely proportional to the block size in this kernel.

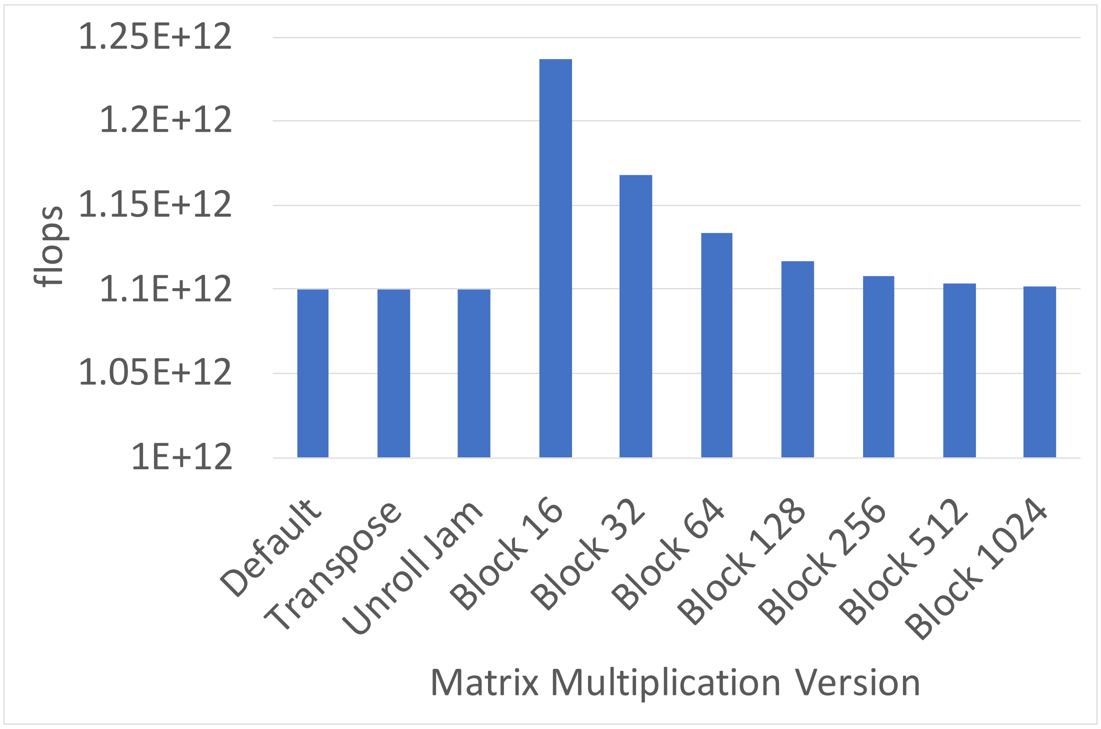


Figure 38. A64FX **flops** for the matrix multiplication variations.

The results for our other computational metric, **flops / fp ins**, are shown in Table 14 We chose this format because the results lack variation between the kernels. Most importantly, the A64FX version is not vectorized well. The compilers can make use of the FMA instructions, but not SIMD operations. The Cascade Lake compiler does vectorize well and reaches close to 8 **flops / fp ins** on most versions. The unrolling seems to obscure the kernel from the compiler, so it is prevented from using SIMD operations. The reductions added by the blocking of the loops cannot be performed as SIMD operations, so the blocked version has a **flops / fp ins** that increases, presumably asymptotically, towards 8 as the block size increases.

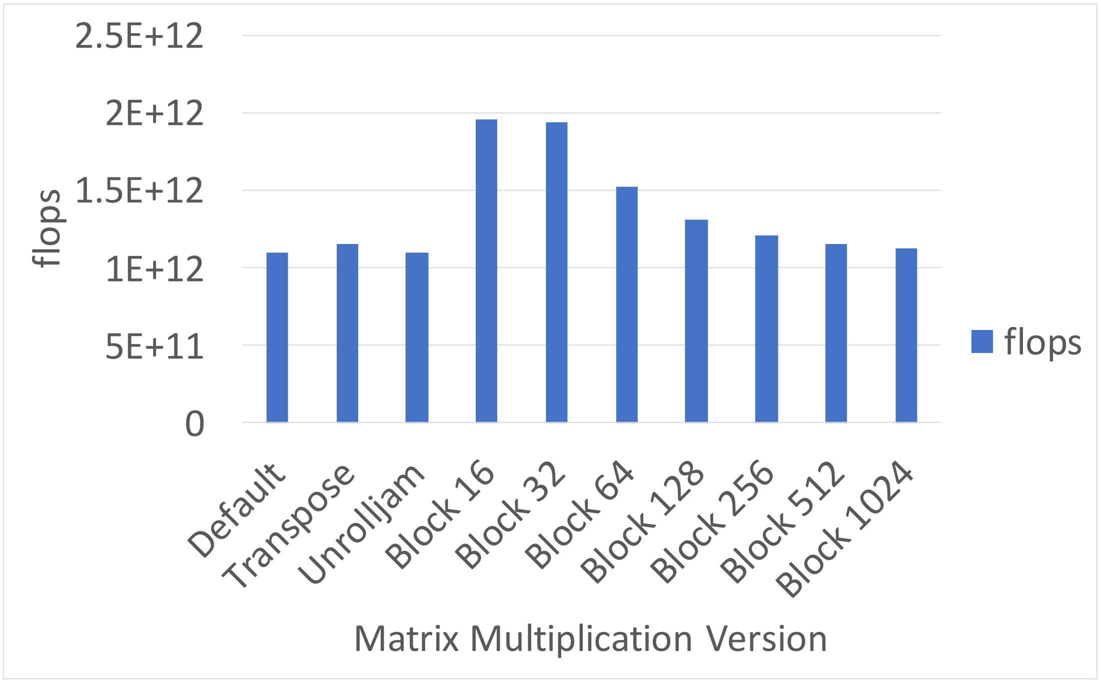


Figure 39. Cascade Lake **flops** for the matrix multiplication variations.

From the computation metrics, we see two clear avenues to improve the kernel. First, we will want to identify a method to perform blocking without adding additional floating-point operations. Then we can rerun the experiments to validate that method. Second, on the A64FX, e need to find a way to make use of the SIMD instructions. These instructions are a vital part of the performance of modern CPUs and matrix multiplication is a well-known code that should be able to use SIMD operations. Addressing these issues may help us further improve the performance of our kernel.

5.3.5 Computation Data Rates. Our final set of metrics is meant to unite the data movement and computation measurements to help the user understand them in context of each other. We start with Arithmetic Intensity (**AI**) and add several similar ratios of **flops** per data loads and stores.

Table 14. **flops / fp ins** for each version of matrix multiplication on the A64FX and Cascade Lake systems.

	A64FX	Cascade Lake
Default	2	8.00
Transpose	2	7.90
Unroll	2	1.00
Block 16	2	1.58
Block 32	2	7.53
Block 64	2	7.70
Block 128	2	7.82
Block 256	2	7.90
Block 512	2	7.95
Block 1024	2	7.97

AI has been used since at least [36] described the balance point of a machine when neither the memory nor computational hardware was stalled for the other. The Roofline Model [37] uses **AI** as the independent variable for modeling the application and the hardware. In Chapter IV, we discuss **AI** and the Roofline model in depth.

Additionally, we use **flops/LD ins** and **flops/ST ins** to compare the computational work to the number of load and store instructions executed. Distinguishing between the loads and stores is necessary because they impact application performance in different ways. Computation (**flops**) moves the application forward enabled by the loading of new data. The time to store data can be hidden in hardware with store buffers or algorithmically by data reuse, so computation does not depend on data writes as much as it depends on the data loads. However, the store operations can limit the bandwidth available for load operations, which then limits the rate of computation. Understanding the differences in how the application uses the two types of operations can help identify which optimizations to use.

Our last adjustment is to use **flops/LD bytes** and **flops/ST bytes** which uses the bytes moved rather than the number of instructions. SIMD operations also apply to the data movement operations so this can provide a slightly different view than the instruction variations. Subtle differences in the metrics can occasionally reveal important information, so extending the set of metrics to include more variations may be a useful exercise if the user is dissatisfied with the results.

5.3.5.1 Demonstration with STREAM. Our STREAM benchmark (see Chapter III) is designed to perform a consistent number of floating-point and data movement operations. We run the benchmark with a range of data sizes covering two of the caches on each system. We expect that the ratio of floating-point operations to data movement instructions and the number of bytes moved will remain consistent for all of the measured sizes.

Figures 40 and 41 show the results for the A64FX and the Casadelake CPUs respectively. In both cases, we see that all of the Computation Data Rate metrics are consistent across all of the measured data sizes. Additionally, we see that **flops/LD ins** is approximately 64$\times$ the value for **flops/LD byte** and the store operations have a similar ratio. The STREAM benchmark uses 512-bit SIMD operations so each data movement instruction moved 64 bytes of data. These results give us confidence that the metrics are performing as designed but do not demonstrate the metrics' usefulness.

5.3.5.2 Demonstration with Matrix Multiplication. We collected the computation data rates for the variations of the matrix multiplication that we discussed in Chapter III. The results show how the relationship between computation and data movement impacts the performance of the application. As

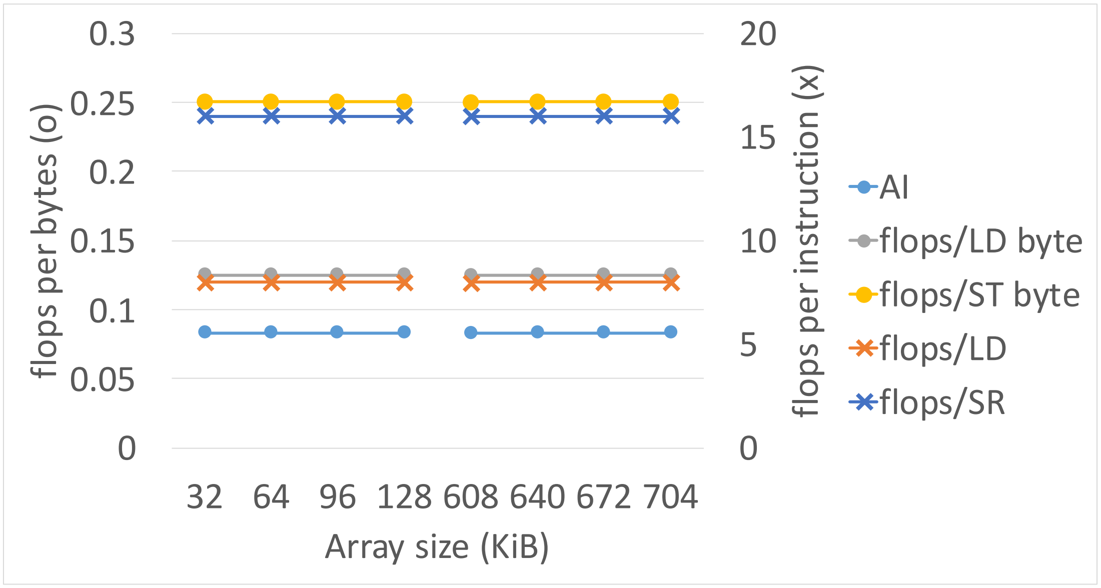


Figure 40. A64FX computation data rates for STREAM with several data sizes.

shown in the Roofline Models, applications that have a higher ratio of floating-point operations per amount of data moved to have a higher potential performance.

Figure 42 shows the **flops/LD ins** and **flops/ST ins** for the matrix multiplication versions on the A64FX and Figure 43 shows the same data on the Cascade Lake. On A64FX the **flops/LD ins** is less than one for the default, transpose, and unroll versions. It rises to two and three for the blocked verison, but remains constant. The Cascade Lake **flops/LD ins** consistently matches the **flops/FP ins** for all of the versions other than the transpose version which performs 26.2 **flops** per **LD instruction**.

Based on these results for **flops/LD ins**, we conclude that the kernel is loading data from the cache for each of the array accesses in most of the Matrix Multiplication versions on both processors. The exception is the transpose version on the Cascade Lake which appears to be able to reuse some of the data within the CPU to increase the amount of computation performed for each load operation.

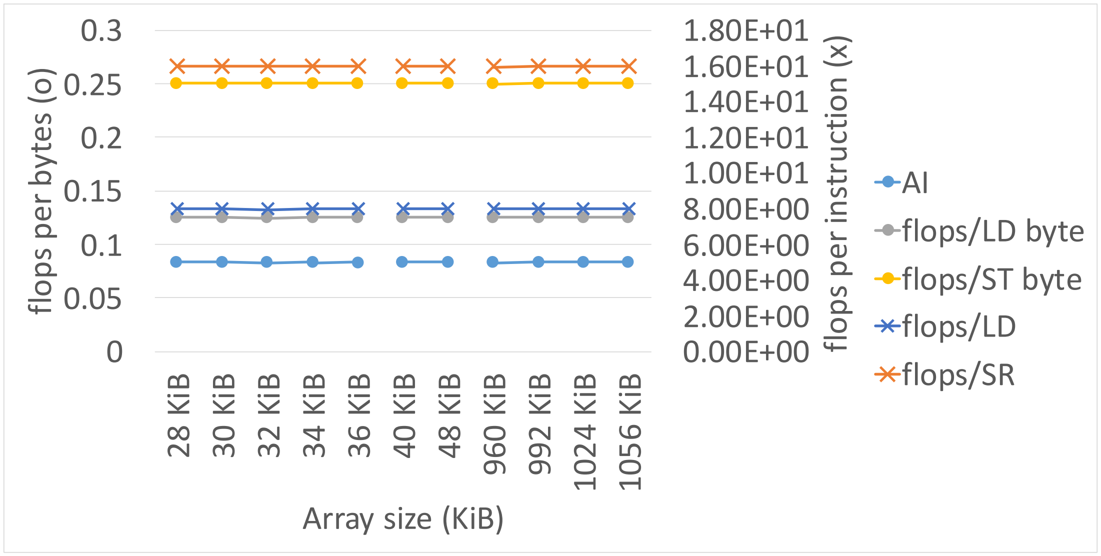


Figure 41. Cascade Lake computation data rates for STREAM with several data sizes.

The fact that the Cascade Lake transpose version is an outlier in this way is significant because it is also the most performant variation of the kernel (see Figures 44 and 45). Notably, transpose outperforms the blocked versions on Cascade Lake, but not on the A64FX where the **flops/LD ins** is more consistent with the other versions. We conclude that finding ways to increase the **flops/LD ins** on both systems could lead to improved performance.

The **flops/ST ins** has a slightly different performance pattern than the **flops/LD ins**. On the A64FX, it is about 4 **flops/ST ins** for the default, transpose, and unroll versions. This result matches what a user could count from the source code. When blocking is applied, the metric increases with the block size up to nearly 900 **flops/ST ins**. The Cascade Lake results are similar, but with a higher **flops/ST ins** for the transpose version.

The **flops/ST ins** results indicate that the compiler can limit stores in the blocked version to the end of the loop blocks which results in a significant reduction

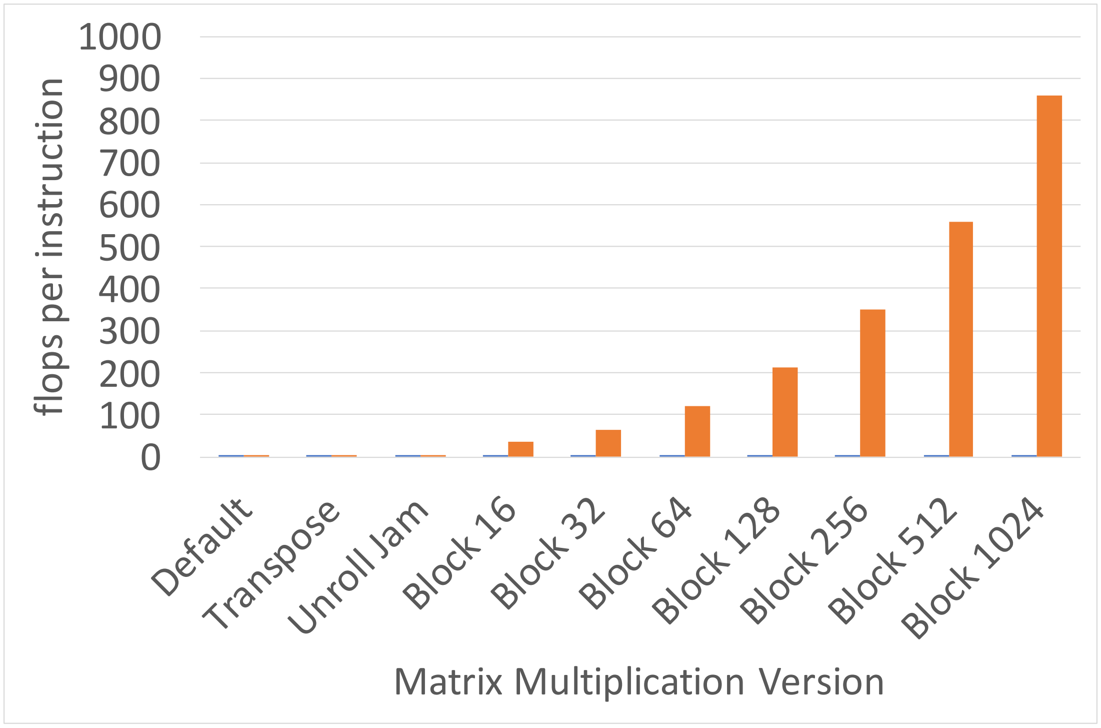


Figure 42. A64FX **flops / LD** and **flops / ST** for the matrix multiplication variations.

in the number of store operations required. However, Figures 46 and 47 show that the ratio of load operations to store operations is quite high. This result, combined with the **flops/ST ins** metrics does not correlate with the execution time, leads us to believe that the stores are not a determining factor in the performance of the kernel.

An alternative measurement to the **flops/LD ins** and **flops/ST ins**, is the **flops/LD Bytes**, **flops/ST Bytes**, and the **AI**. We include these results in Figures 48 and 49. Since the **flops/ST Bytes** is much larger than the other metrics, we include just **flops/LD ins** and **AI** in Figures 48 and 49.

Depending on the situation, the change in the denominator can help improve understanding. For this example, the results are largely the same as the instruction-based results above. The one exception is that **flops/LD ins** for the

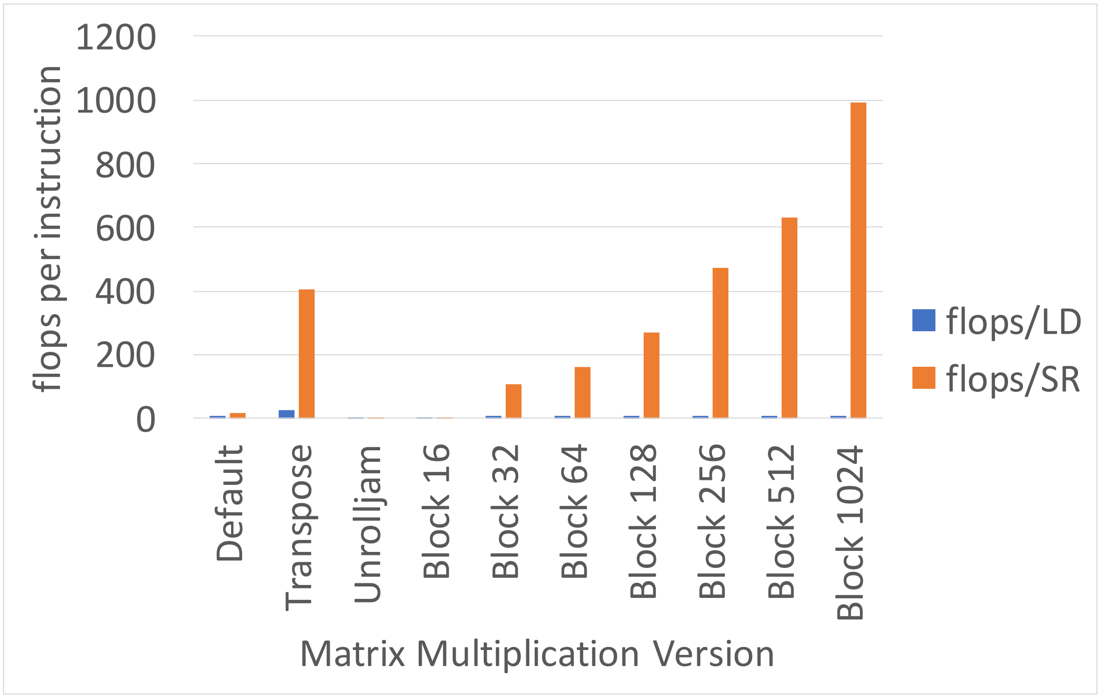


Figure 43. Cascade Lake **flops / LD** and **flops / ST** for the matrix multiplication variations.

default and unroll versions on the Cascade Lake. The **flops/LD ins** is higher for the default version, but the **flops/LD Bytes** are identical for the two. This difference occurs because the default version uses SIMD operations, but the unroll version does not.

CPUs have a limited number of ports that can execute memory operations, so SIMD instructions are necessary to achieve the highest possible bandwidth. Analyzing the computation data rates based on the number of memory instructions and the number of bytes moved will help the users make sense of the relationships better than one of the metrics alone.

5.4 Summary

This chapter presents and demonstrates our set of hardware counter based performance metrics. We define these metrics and show how they can be measured

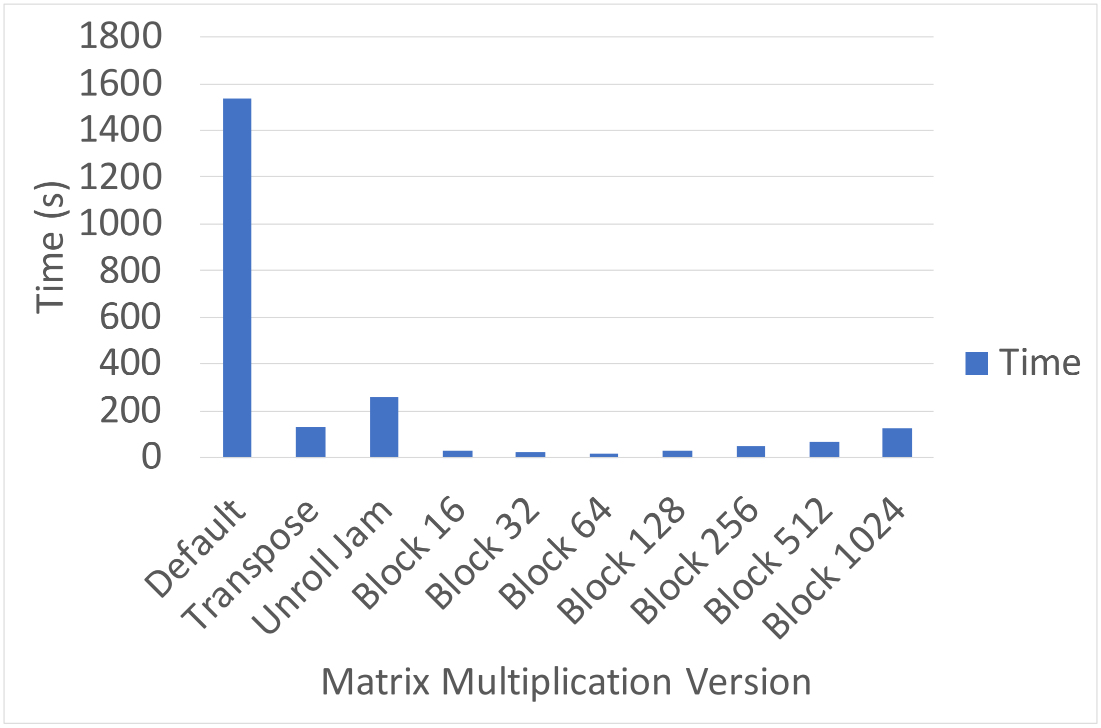


Figure 44. A64FX **Time** for the matrix multiplication variations.

on our two systems. After that, we provide several examples of using those metrics to study the benchmarks discussed in Chapter III. We use these examples to show the accuracy and usefulness of our metrics on both of the CPUs. The hardware counter metrics in this chapter can provide performance information to users on our two different CPUs.

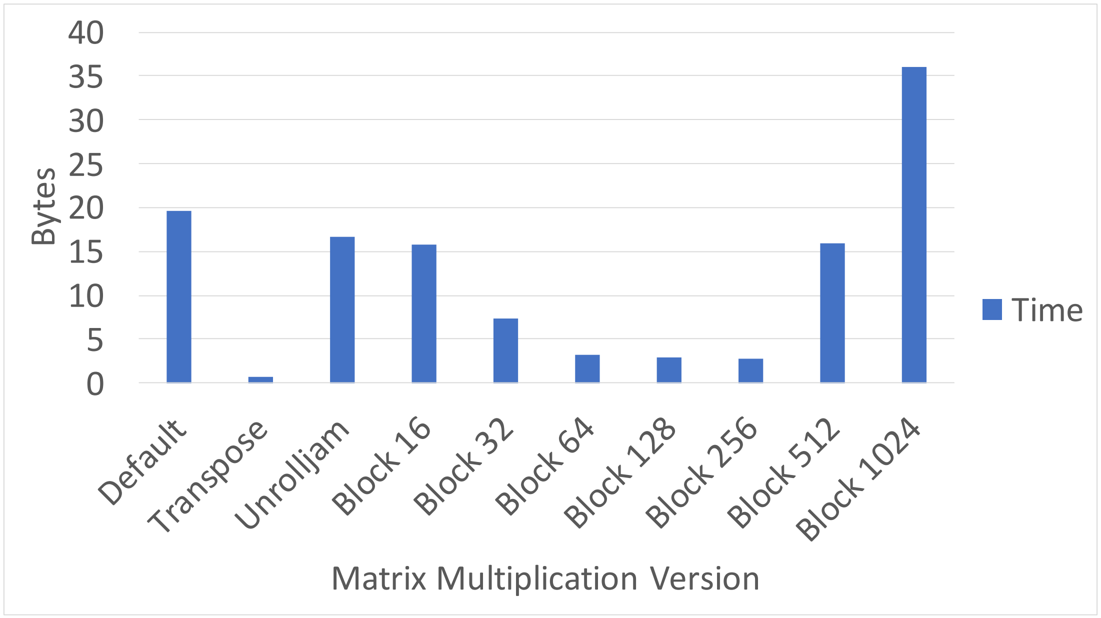


Figure 45. Cascade Lake **Time** for the matrix multiplication variations.

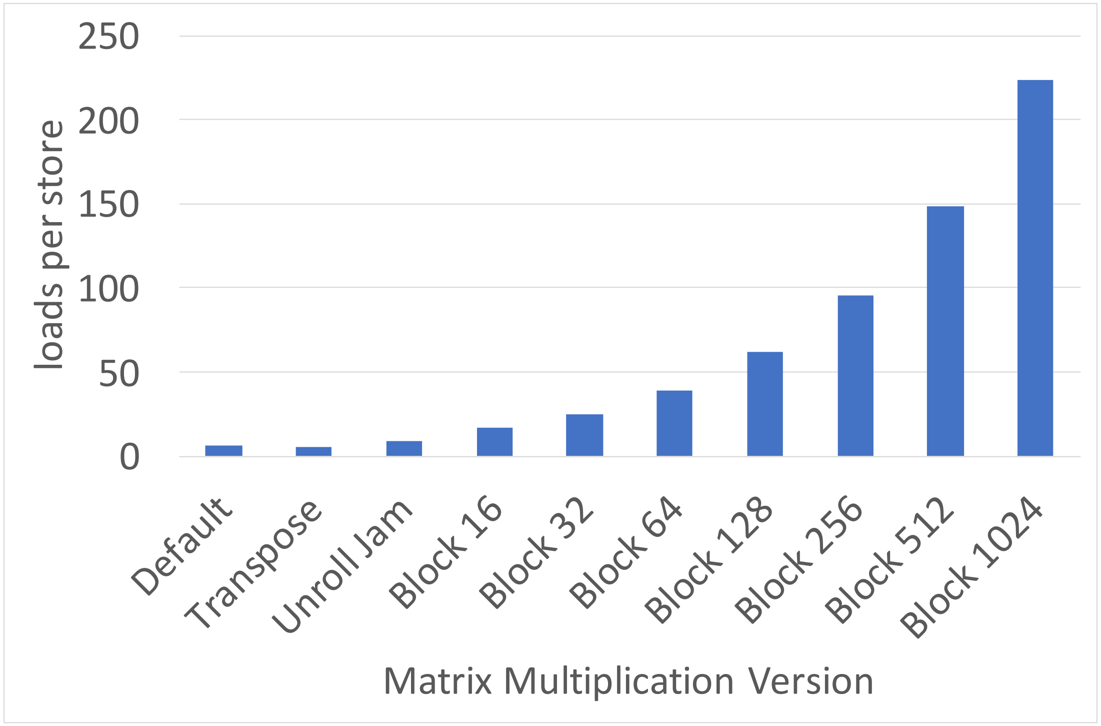


Figure 46. A64FX **LD ins/ST ins** for the matrix multiplication variations.

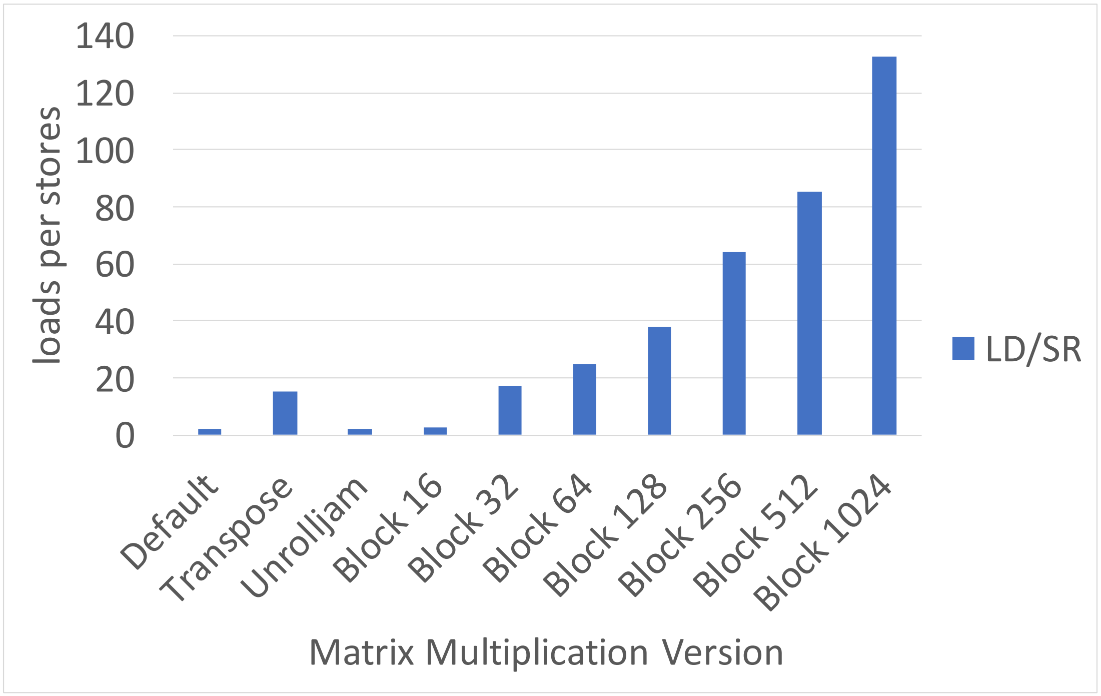


Figure 47. Cascade Lake **LD ins/ST ins** for the matrix multiplication variations.

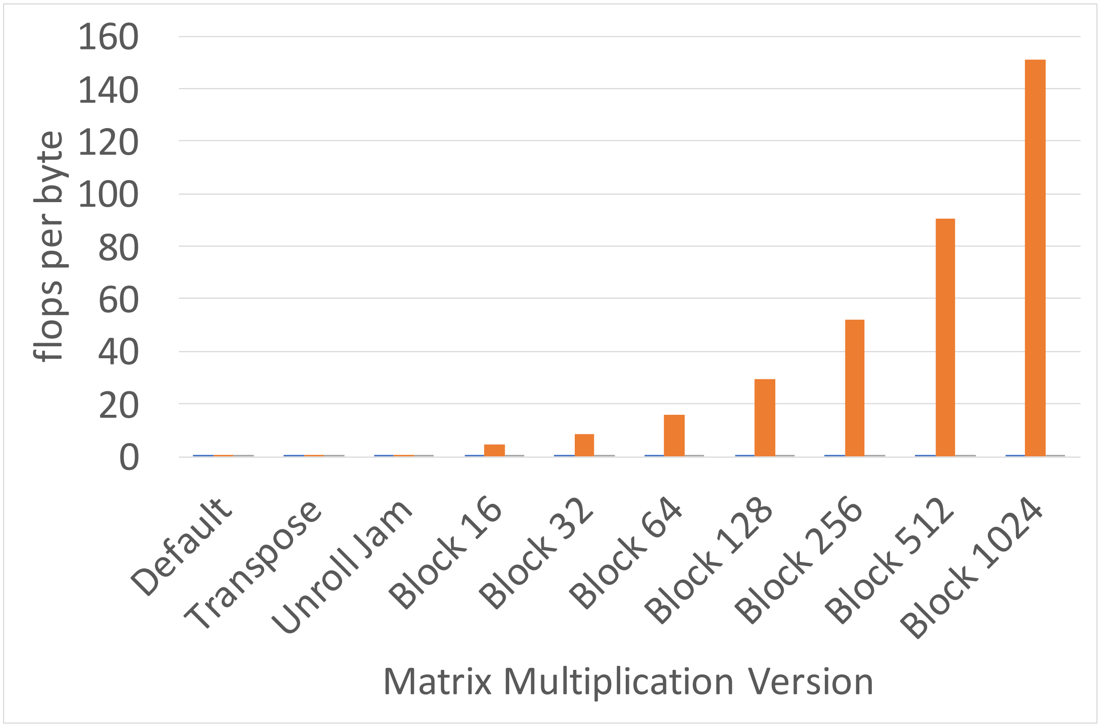


Figure 48. A64FX **AI flops / LD bytes** and **flops / ST bytes** for the matrix multiplication variations.

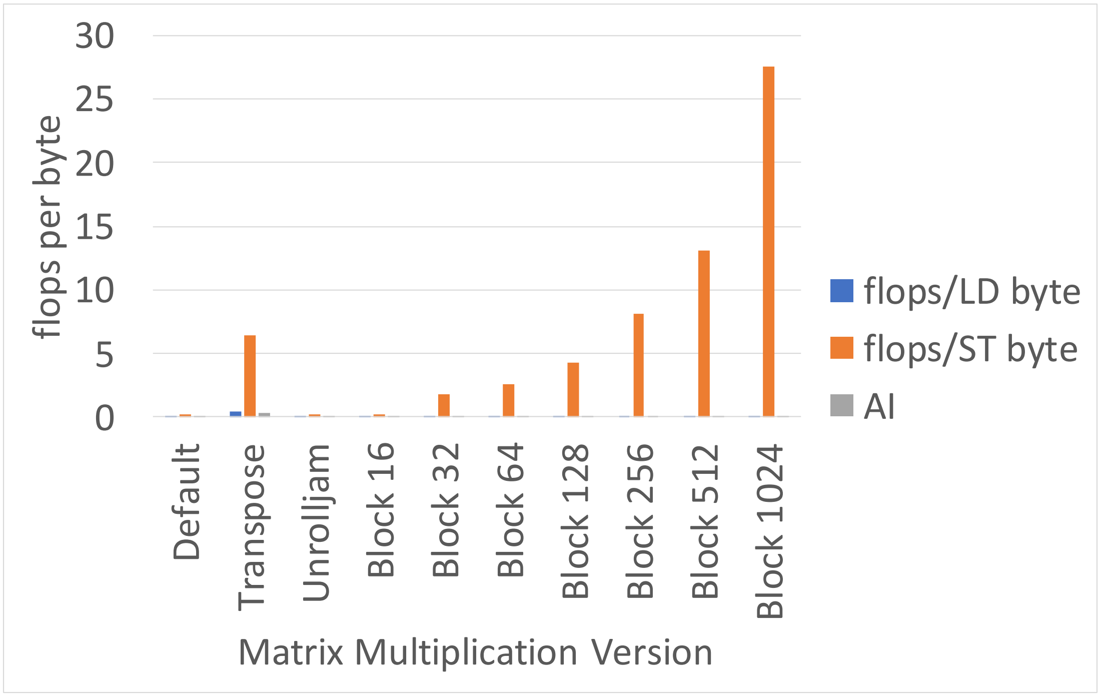


Figure 49. Cascade Lake **AI flops / LD bytes** and **flops / ST bytes** for the matrix multiplication variations.

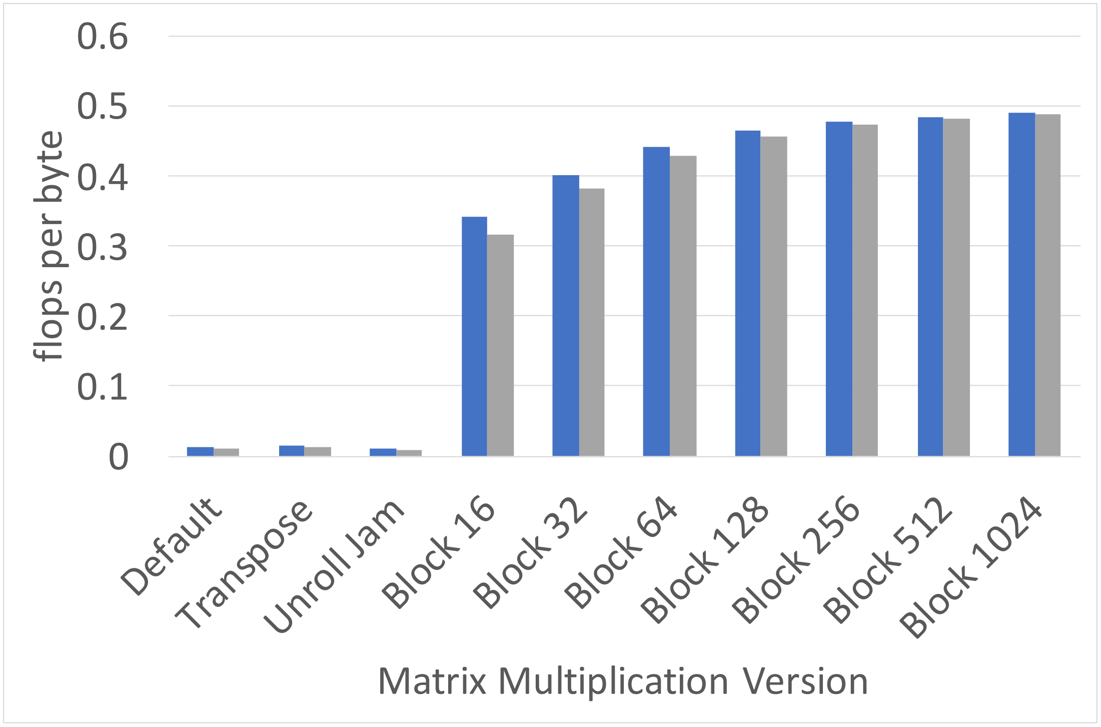


Figure 50. A64FX **AI** and **flops / LD bytes** for the matrix multiplication variations.

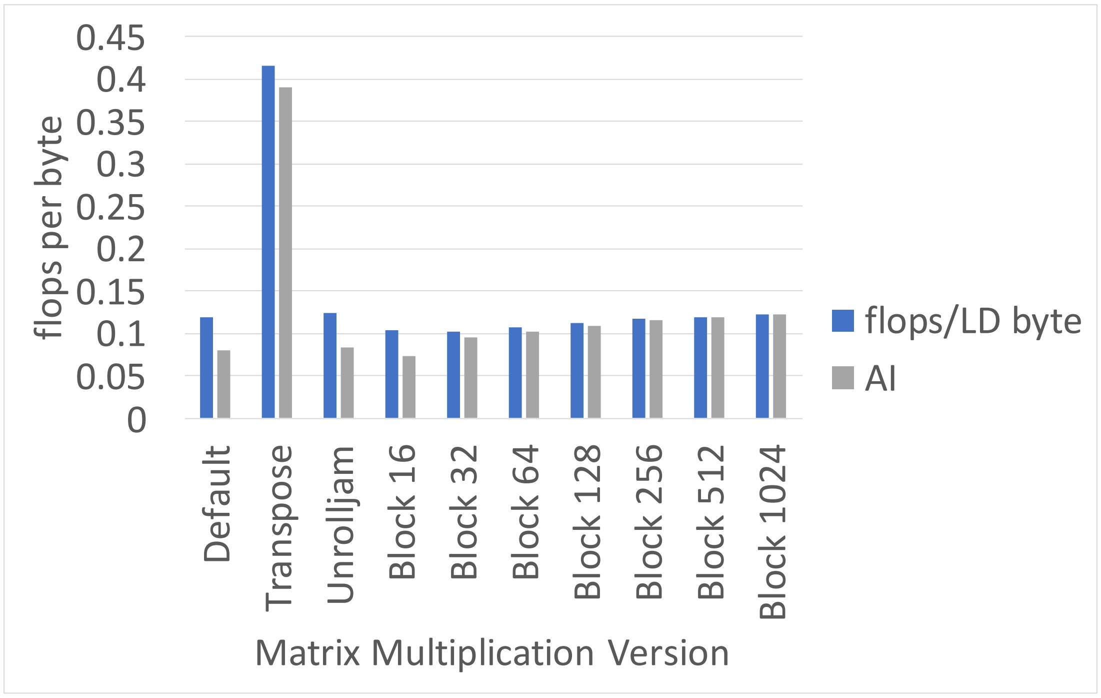


Figure 51. Cascade Lake **AI** and **flops / LD bytes** for the matrix multiplication variations.

CHAPTER VI

CASE STUDIES OF MINI-APPLICATION PERFORMANCE USING THE HARDWARE COUNTER METRICS

In Chapter V, we presented a set of performance metrics from hardware that we can measure on our two different systems. Here we analyze several example mini-applications with our performance metrics. These examples show that the metrics used can improve the user's understanding of the performance of their application.

6.1 Kernel Examples

To illustrate the usefulness and differences between the kernels, we analyze several kernels designed to demonstrate the metrics. For each kernel, we include multiple versions so the user can gain a full context for how the metrics can be used to analyze optimizations to an application. Some of these changes are minimal, such as changes to the compiler options, while others are significant, such as changes to data structures. These analyses offer a demonstration of both the individual metrics as well as the process of analysis.

The process of evaluating performance based on hardware counters can be tedious and confounding. Numerous research efforts (see Ch. II) attempt to clarify this process, but the efforts generally rely on extensive benchmarking of the hardware or micro-architecture specific measurements. Both options can be prohibitively difficult to apply. Our approach is focused on the application, so we find that patient study of multiple versions of an application can provide useful context to the measurements. This section illustrates that process.

6.1.1 N-Body Simulation. The N-Body problem is a classic mechanics problem computing the movement of N celestial bodies as their gravitational forces interact. This computational kernel is well-studied and a common introduction to scientific parallel computing. In this paper, we use it to demonstrate how the use of vector operations can have surprising effects on the performance of an application and how our proposed metrics reveal those changes. The N-Body results (Table 16) show results for the kernel compiled with and without the use of SIMD instructions on each of our systems. On both systems a few changes to compiler flags (Table 15) allow us to get a speedup (11.8× on A64FX and 3.77× on Cascade Lake) relative to the respective scalar versions.

Table 15. Compilers and options for the N-Body application

CPU and compiler	Scalar options	SIMD options
Cascade Lake, Intel	-O1 -g - -march=cascadelake -no-simd	-O3 -g -march=cascadelake -qopt-zmm-usage=high
A64FX, Fujitsu	-O0 -g	-O3 -g -march=armv8.2-a+sve -mcpu=a64fx -Kfast,openmp,ocl -Ksimd_reg_size=512

We begin by looking at the overall metrics of performance; **time**, **IPC**, and **flops/s**. For N-Body, we see that the **time** is significantly reduced with the addition of SIMD-oriented compiler options. Speedup is about 12× on the A64FX and 4× on the Cascade Lake. The rate of computation also increases for both systems. However, the IPC does not increase on the Cascade Lake. From this we learn that the improvement comes in different ways on the two systems and, for the Cascade Lake, it likely comes from less computation being performed.

Table 16. N-Body data from unoptimized and optimized versions

	A64FX		CAS	
	scalar	simd	scalar	simd
Overall Performance				
Time (s)	421.72	35.80	11.23	2.98
IPC	0.38	0.80	2.48	0.55
flops/s (Gflops/s)	3.26	280.3	122.4	460.8
Data Movement				
LS Bytes (GB)	20,170	2,202	16,191	2,581
L2 Bytes (GB)	11,101	2,440	1,272	1,283
L3 Bytes (GB)	NA	NA	1,661	1,678
Mem Bytes (GB)	0.26	0.12	0.78	0.33
Cache Efficiency				
L1 MR	1.59E-03	31.2E-03	9.80E-03	0.50
L2 MR	12.0E-06	43.0E-06	1.31	1.31
L3 MR	NA	NA	21.0E-06	25.0E-06
L2 / LS	0.55	1.11	0.079	0.50
L3 / LS	NA	NA	0.10	0.65
Mem / LS	12.8E-06	55.8E-06	47.9E-06	129.0E-06
LD ins / ST ins	5.65	3.75E3	2.26	80.80
Computation				
flops	1.37E12	10.0E12	1.37E12	1.37E12
flops/fpins	0.95	6.08	1	7.9997
Computation Data Ratios				
AI	0.068	4.56	0.085	0.53
flops/LD ins	0.32	36.41	0.98	34.50
flops/ST ins	1.81	136.5E3	2.21	2.8E3
flops/LD Bytes	0.08	4.56	0.12	0.54
flops/ST Bytes	0.45	32.7E3	0.28	43.56

Looking at the data movement in Table 16, we note at least two ways the optimizations improve memory performance. Most obviously, the **LS bytes** are significantly reduced in the optimized version. This often happens when the unoptimized version repeatedly reloads the memory address of an array in a loop rather than calculating the next address. Without the repetitive address loads, we see that a significant portion of the data comes from the L2 cache. Further, the optimizations reduce the amount of data stored which improves the **LD ins**

/ ST ins ratio. Since loading data is often a limiting factor for performance, it is important not to delay loads with unnecessary stores. The reduction in overall data movement and stores, in particular, is a likely contributor to the improved overall performance of the kernel.

The computation-focused metrics reveal a murkier picture. We see the SIMD version **flops** are nearly 10× higher than the scalar version. Some of these added **flops** are likely due to reductions that are sometimes performed to sum SIMD registers into a single scalar value. Others may have unused results that are excluded with masking. Details may be available in optimization reports or analysis of the assembly but not through empirical analysis of hardware counters. Notably, it is most likely that these additional **flops** are performed entirely in registers without movement to cache or memory since the **AI** for A64FX increases much more than that measured for the Cascade Lake. The vectorization metric, **flops/fp ins**, indicates that A64FX is not vectorizing to its full potential (maximum 16 **flops/fp ins** with FMAs), but that Cascade Lake gets quite close (maximum of 8 **flops/fp ins** since FMA are not measured). We could use compiler reports to check on the use of FMA operations, but want to keep the focus of the paper on hardware counter analysis. In total, computation shows some improvement with the SIMD options, but may be able to benefit from additional SIMD operations and changes limit extraneous **flops** on A64FX.

The impact of the memory and floating-point improvements can be seen in the comparative ratios: **AI**, **flops / LD ins**, **flops / ST ins**, **flops / LD bytes**. Improving the amount of computation relative to the data movement is an important step in using most modern systems, so it is encouraging to see increases in all these measurements. High **AI** would indicate the algorithm is a good candidate for further vectorization or porting to accelerators. On the A64FX, the compiler can hold enough data in registers so that stores are nearly eliminated. The N-Body algorithm updates each body based on data from all the others, so the minimal stores make sense if there are enough registers to hold all the data for two bodies.

Based on this analysis, we conclude that further exploration of vectorization on A64FX could be helpful, while the Cascade Lake version could use additional work on its cache performance. The user can modify the code based on these results, form hypotheses about how performance will be impacted, and repeat the hardware counter analysis to verify how the performance changes.

6.1.2 XSBench. The first mini-application we consider is XSBench [56], which represents a key kernel of the Monte Carlo neutron transport algorithm. The kernel executes a large number of lookups into a table of data based on material properties and the energy of neutrons moving through those materials. Functionally, the kernel is executing a series of randomized memory accesses into a data table larger than most caches. XSBench is useful for studying full neutron transport applications like OpenMC [57].

We consider the *event* based parallelization of XS Bench which has three variations: a default random access version (called *event*), optimization 1 which sorts based on energy and materials (*event opt 1*), and optimization 2 which sorts solely based on energy (*event opt 2*). We profile and compare all three versions with the metrics presented in this paper.

Table 17. XSBench event kernel data

	A64FX			CAS		
	Original	*opt 1*	*opt 2*	*Original*	*opt 1*	*opt 2*
Overall Performance						
Time (s)	4.17	3.51	2.37	1.46	0.80	0.58
flops/s (Gflops/s)	9.46	10.22	15.51	18.42	32.27	45.32
IPC	0.22	0.75	0.76	0.27	1.43	1.79
Data Movement						
LS Bytes (GB)	201	190	194	234	226	230
L2 Bytes (GB)	913	624	699	214	137	151
L3 Bytes (GB)	NA	NA	NA	188	12.3	13.8
Mem Bytes (GB)	702	12.5	6.89	129	112	7.42
Cache Efficiency						
L1 MR	0.017	0.018	0.019	0.241	0.16	0.18
L2 MR	0.43	0.020	0.0098	0.88	0.090	0.091
L3 MR	NA	NA	NA	0.63	0.60	0.35
L2 / LS	4.54	3.29	3.61	0.92	0.61	0.66
L3 / LS	NA	NA	NA	0.80	0.055	0.060
Mem / LS	3.49	0.066	0.036	0.55	0.050	0.032
LD ins / ST ins	3.73	3.91	3.91	5.23	5.36	5.361
Computation						
flops	39.45E9	35.91E9	36.71E9	26.87E9	25.88E9	26.42E9
flops/fpins	1.59	1.51	1.51	2.11	2.15	2.15
Computation Data Ratios						
AI	0.20	0.19	0.19	0.11	0.11	0.11
flops/LD ins	1.86	1.75	1.76	2.31	2.34	2.34
flops/ST ins	6.96	6.86	6.87	12.06	12.52	12.53
flops/LD Bytes	0.25	0.24	0.24	0.14	0.14	0.14
flops/ST Bytes	0.95	0.91	0.91	0.72	0.73	0.73

Our XS Bench optimizations improve on each other for all three of the overall performance metrics (Table 17). This improvement is seen in both **IPC** and **flops/s**, so we suspect that the improvement comes from more efficient processing rather than a reduction in work. We will learn how from the other metrics.

Starting with the data movement, we see that the **LS Bytes** remains fairly consistent, but the other data volumes change considerably. On both systems, from the *event* version to *event opt 1*, the **L2 Bytes** and **Mem Bytes** both drop by large margins (**L3 Bytes** too on Cacade Lake). Looking at the change from **opt 1** to **opt 2**, we see that the **L2 Bytes** goes back up a small amount, while **Mem Bytes** is reduced further. The efficiency metrics (miss rates and cache ratios) reinforce the impression that the optimizations both make the cache use more efficient.

Consider the case of comparing the two systems to port an application from one to the other. With the help of the metrics here we can see where the performance on the A64FX falls short of the Cascade Lake. The most striking difference arises in the measurement of **L2 Bytes** which is the number of bytes moved from the L2 cache to the L1 and then CPU due to L1 cache misses. The A64FX has a 256 byte cache line, while that of the Cascade Lake is only 64 bytes. Based on this information, we can hypothesize that the additional cache line space is being wasted moving unnecessary data, resulting in more data being moved on the A64FX than on the Cascade Lake.

Moving on to the floating-point metrics, we note that the **flops/s** improves consistently, but the **flops/fp ins** does not. It is not surprising that random memory accesses are not easily transformed into SIMD operations, but if the user succeeds there will likely be large benefits. The limited number of load and store ports on CPUs means that SIMD instructions are needed to achieve peak memory bandwidth on many systems, so even memory-focused applications can benefit from vectorization.

Due to the consistency of both **flops** and **LS Bytes**, the various comparative metrics do not change significantly across the variations. However, there is a difference, between the two systems. The difference comes from a combination of higher **flops** and lower **LS Bytes** on the A64FX relative to the

Cascade Lake. Identifying experiments to explore these differences may be a good next step in the optimization process since understanding why the runtime is smaller on Cascade Lake and the data movement is reduced on A64FX may help the user port those improvements to the other system.

6.1.3 Cloverleaf. CloverLeaf [58] is a mini-app that solves the compressible Euler equations on a Cartesian grid, using an explicit, second-order accurate method. Each cell stores three values: energy, density, and pressure. A velocity vector is stored at each cell corner. This arrangement of data, with some quantities at cell centers, and others at cell corners is known as a staggered grid. CloverLeaf currently solves the equations in two dimensions, but a 3D implementation has been started in CloverLeaf3D.

One of the key kernels of Cloverleaf is PdV which computes changes in energy and density in each cell. The kernel consists of a double nested loop iterating over some subset of the volume. We built our baseline, $PdVSIMD$, with compiler options encouraging vectorization. Next, we applied a blocking optimization to try to improve cache performance ($PdVBlockSize4$). Table 18 show the results from these experiments.

Table 18. Cloverleaf PdV kernel data

	A64FX		CAS	
	Original	Block Size 4	Original	Block Size 4
Overall Performance				
Time (s)	33.12	75.46	14.67	16.72
flops/s (Gflops/s)	24.17	16.57	45.49	42.69
IPC	0.56	0.52	0.10	0.28
Data Movement				
LS Bytes (GB)	2,600	5,350	6,250	8,830
L2 Bytes (GB)	9,300	2,620	2,810	4,080
L3 Bytes (GB)	NA	NA	3,790	2.28E+12
Mem Bytes (GB)	3,490	3,540	2,870	2,790
Cache Efficiency				
L1 MR	0.040	0.039	0.45	0.18
L2 MR	0.25	0.10	1.35	0.56
L3 MR	NA	NA	0.61	0.998
L2 / LS	3.58	4.90	0.45	0.46
L3 / LS	NA	NA	0.61	0.26
Mem / LS	1.34	0.66	0.46	0.322
LD ins / ST ins	8.60	6.80	11.03	6.17
Computation				
flops	0.80E12	1.25E12	0.67E12	0.71E12
flops/fpins	2.00	1.67	7.99...	3.19
Computation Data Ratios				
AI	0.31	0.23	0.11	0.081
flops/LD ins	2.47	1.80	7.45	2.40
flops/ST ins	21.23	12.21	82.20	14.81
flops/LD Bytes	0.34	0.27	0.12	0.094
flops/ST Bytes	2.92	1.95	1.28	0.58

In this case, our attempt at optimization was unsuccessful; the overall metrics clearly show that the performance slows with the blocking we did here. Note that we tried several block sizes, and all performed worse than the baseline case. The data can help us understand the performance of the baseline case better by revealing which aspects are most important to the performance. Often we will achieve a performance goal with optimization (such as a lower miss rate), but cause different problems creating a negative overall impact.

Considering the data movement metrics, we note that the **LS Bytes** and **L2 Bytes** for both systems increase with the change, while the **Mem Bytes** remain steady. The **L3 Bytes** is reduced on the Cascade Lake. Increased data traffic between the L1 caches and the CPU is a likely culprit for performance degradation, but there may be other impacts to consider.

At the L1 level, the **L1 MR** is far higher on the Cascade Lake than on the A64FX. We suspect this is the product of increased memory movement between L1 and CPU since both **LS Bytes** and **L2 Bytes** rise. The other miss rates also go down for both applications. Improved cache efficiency is a goal of blocking, but the increase in total data movement suggests that the improved efficiency is misleading.

The total computation (**flops**) increases for both systems, but more so for A64FX. The user could look to assembly language or compiler reports to attempt to diagnose the source of new operations. Notably, there is a dramatic reduction in SIMD instruction use for the Cascade Lake. The lack of SIMD is partially due to the size of the blocks (i.e. smaller than the SIMD registers), but the additional complexity of the blocked loops also hinders the compiler's use of SIMD operations. Overall, the efforts at blocking have resulted in far less efficient computation on both systems.

Finally, the comparative rates show that less computation is performed relative to data moved for both systems. These results are confirmation of the others indicating less efficiency all around.

Finding a path forward from these results can be frustrating. These results give us the impression that we need to approach the systems completely differently. First, the A64FX has relatively good cache performance with $PdVSIMD$, but poor SIMD performance. In contrast, the Cascade Lake vectorizes fairly well but misses half of the L1 cache accesses. In our next experiments, we would explore optimizations that can improve vectorization on the A64FX while we seek methods of improving cache use without sacrificing SIMD operations on the Cascade Lake.

6.1.4 PENNANT.

PENNANT [59] is a physics mini-app based on unstructured mesh data structures and physics algorithms adapted from the LANL rad-hydro code FLAG. It is useful for studying the performance of similar algorithms on new systems without the burden of running FLAG as a whole.

One simple, but effective method of performance improvement is to explore the numerous settings available in an application. In this case, PENNANT is using shared memory threads with OpenMP which has a variety of options for the arrangement and scheduling of the threads. For example, chunk size determines how many sequential iterations are allocated to each thread. We compare a larger chunk size of 64 and a smaller size of 4.

Looking at the overall performance metrics, we see that the smaller chunk size is faster on both systems and that is reflected in the **flops/s** as well as the **time**. The **IPC** falls with the improved version, which we find surprising.

LS Bytes on A64FX is reduced by about half when the chunk size is reduced. Cascade Lake **LS Bytes** not reduced significantly. Otherwise, the data movement is not impacted very much by the change in chunk size.

Table 19. PENNANT doCycle kernel data with different OpenMP chuck sizes

	A64FX		CAS	
	chunk 64	chunk 4	chunk 64	chunk 4
Overall Performance				
Time (s)	71.6	44.8	10.57	7.65
flops/s (Gflops/s)	10.7	17.1	119	165
IPC	1.29	0.60	1.05	0.93
Data Movement				
LS Bytes (GB)	10,700	5,420	13,300	13,200
L2 Bytes (GB)	8,170	8,130	2,590	2,590
L3 Bytes (GB)	NA	NA	915	1,010
Mem Bytes (GB)	1,440	1,320	961	1,120
Cache Efficiency				
L1 MR	0.011	0.016	0.075	0.13
L2 MR	0.086	0.096	0.41	0.39
L3 MR	NA	NA	0.37	0.53
L2 / LS	0.76	1.50	0.19	0.20
L3 / LS	NA	NA	0.069	0.076
Mem / LS	0.13	0.24	0.072	0.085
LD ins / ST ins	7.39	4.926	6.30	4.90
Computation				
flops	0.77E12	0.77E12	1.26E12	1.26E12
flops/fpins	1.06	1.06	5.11	5.11
Computation Data Ratios				
AI	0.072	0.14	0.095	0.095
flops/LD ins	0.42	1.25	2.99	4.69
flops/ST ins	3.13	6.16	18.84	23.02
flops/LD Bytes	0.084	0.18	0.11	0.11
flops/ST Bytes	0.50	0.74	0.56	0.56

Similarly, the cache efficiency metrics show only minimal change. The one exception is the **L1 MR** on the Cascade Lake which rises despite the improved overall performance. Notably, **L2/LS** does not increase with the miss rate. This discrepancy could be explained because the miss rates are based on instruction counts while the **L2/LS** is based on bytes. The cache metrics show us that the memory performance may be the cause of the change, but are not conclusive.

For the computation, both **flops** and **flops/s** show no difference between the two versions. We note that the Cascade Lake does vectorize partially while A64FX has essentially no SIMD operations. Knowing that SIMD use is possible, we could confidently target those operations on A64FX in future optimizations.

Finally, understanding the computation data ratios helps us develop a convincing hypothesis about the improved runtime of the smaller chunk size. The A64FX has increased ratios in all cases, as expected with the constant **flops** and reduced **LS bytes**. However the **flops/[LD,ST] bytes** both increase slower than the **flops/[LD,ST] ins**. On Cascade Lake, the **flops/[LD,ST] ins** is higher with the smaller chunks, but the **flops/[LD,ST] bytes** remain constant. Based on these ratios, the lower **IPC**, and the cache metrics, we expect that the load-store operations are using SIMD operations which load multiple elements at a time when the chunk size is reduced. This change will allow the CPU to more efficiently use the bandwidth between the CPU and L1 cache.

We are not entirely satisfied with our theories on PENNANT performance. More information, such as comparisons of instruction mixes, thread coordination, and stalls could all be useful. We will explore such metrics in future work.

6.1.5 VPIC. VPIC [60, 61, 62, 63] is a 3D relativistic, electromagnetic particle-in-cell plasma simulation code. The 3D grid is a structured Cartesian mesh with uniform grid spacing. Most of the computational work in a time step is done in a series of loops over either particles or grid cells. VPIC uses single-precision floating-point arithmetic to optimize the use of the available memory bandwidth. Our inputs are for the case of a few hundred particles per cell, but they can range from a few tens to a few thousand.

VPIC uses asynchronous MPI as the top level of parallelism, pthreads or OpenMP as the middle level of parallelism, and vectorization at the lowest level. The granularity of work assigned to a thread is relatively large. The computation is vectorized in a lightweight C++ vector wrapper class that wraps intrinsic function implementations of basic math operations. We use 512-bit SIMD operations on both the Cascade Lake and A64FX nodes. There is also a portable reference implementation that does no explicit vectorization but instead leaves the vectorization task to the compiler.

In the base version of VPIC (which we call *AoS*), both the particle data and the grid data are organized in an Array of Structures format that is aligned along the appropriate word boundaries. Data is read and stored using SIMD loads and stores and is then transposed on the fly so it can be used in SIMD operations. The original particle advance iterates over all of the particles, calculating their next time step and storing the particle back to the appropriate cell. The second version (*AoSoA cell*) uses an Array of Structures of Arrays data format to simplify the vectorization. Additionally, *AoSoA cell* advances all the particles within one cell before moving onto the next cell. This optimization is designed to improve the cache use because the particles within a cell need similar data from the particle advance calculation. The focus on cells also limits the number of atomic operations which must be used when the new positions are written and helps keep the particle lists sorted from one time step to another. Table 20 shows the results from both versions.

Table 20. VPIC Data for the *center_p* kernel

	A64FX		CAS	
	AoS	AoSoA cell	AoS	AoSoA cell
Overall Performance				
Time (s)	23.50	9.55	21.12	15.25
flops/s (Gflops/s)	275	684	289	404
IPC	0.71	0.55	0.43	0.20
Data Movement				
LS Bytes (GB)	13,800	5,220	22,400	3,900
L2 Bytes (GB)	6,770	9,600	2,050	1,590
L3 Bytes (GB)	NA	NA	2,750	2,630
Mem Bytes (GB)	4,590	5,890	4,090	2,930
Cache Efficiency				
L1 MR	0.041	0.097	0.091	0.41
L2 MR	0.26	0.27	1.34	1.65
L3 MR	NA	NA	0.74	0.81
L2 / LS	0.49	1.84	0.091	0.41
L3 / LS	NA	NA	0.12	0.69
Mem / LS	0.33	1.13	0.18	0.75
LD ins / ST ins	5.12	2.06	8.26	3.91
Computation				
flops	6.46E+12	6.53E+12	6.11E+12	6.16E+12
flops/fpins	23.37	23.37	15.99...	15.99...
Computation Data Ratios				
AI	0.47	1.25	0.27	1.58
flops/LD ins	35.90	117.66	19.57	126.95
flops/ST ins	183.99	242.66	161.68	496.31
flops/LD Bytes	0.56	1.87	0.31	1.98
flops/ST Bytes	2.87	3.80	2.53	7.75

Our VPIC results (Table 20) clearly show that the *AoSoA cell* version performs better on both systems than the *AoS* version. Similarly to PENNANT, the **time** and **flops/s** both improve, but IPC does not.

The *AoSoA cell* version uses about half of the **LS bytes** relative to the *AoS* version. If there is a corresponding reduction in the load and store instructions, then this could be a good explanation for both the improved **time** and the reduced **IPC**. In contrast to the reduced **LS bytes**, we note that the **L2 bytes** and **Mem**

bytes increase from *AoS* to *AoSoA cell* on the A64FX. The **LS byte** reduction is a good start; more improvement may be found by reducing the bytes moved at lower levels as well.

The cache efficiency metrics show less efficiency at every level. The reduction in data needed is the main culprit since it lowers the denominator of the efficiency metrics. The **LD ins/ST ins** ratio also falls. We suspect the change comes from fewer loads, perhaps more reuse of data in registers, but we have yet to develop metrics to verify the theory. Now that the overall data movement is reduced, looking to improve cache efficiency could be an effective optimization for this part of VPIC.

The **L2 MR** and **L2/LS** for Cascade Lake are identical. The load and store instructions for the kernel are loading the same number of bytes as the cache line moves, so the **LS bytes** and **L2 Bytes** have the same ratio as the L1 accesses and misses. This only occurs when nearly all the loads and stores are full SIMD operations. A64FX has a different cache line size than its SIMD register size, so the same phenomenon is not observed on that system.

The computation metrics are hard to improve upon. Since the application is in single precision, there can be up to 16 elements per 512 bit register. Therefore, with FMAs, **flops/fp ins** can be as high as 32 for A64FX. VPIC falls short of that because not all operations can be paired to use FMAs. The Cascade Lake manages to hit the perfect 16 **flops/fpins** since the counters on that system doesn't differentiate FMAs the same way as A64FX. The computational performance can only be improved by reducing the amount of work performed.

6.2 Summary

In this chapter, we apply the metrics presented in Chapter V to a range of mini-applications and computational kernels. For each of the examples, we explore several optimizations allowing us to show that the metrics are able to guide and evaluate the performance of computational kernels on our two architectures. In doing so, we show that our set of hardware counter metrics can provide useful performance information on different CPU types. We will not claim that our set of metrics is complete, but it represents a proof, by example, that empirical hardware counter performance metrics can be unified across different micro-architectures while providing actionable information about the target applications.

CHAPTER VII

A CASE STUDY OF THE PAGOSA MULTI-PHYSICS APPLICATION

This chapter provides a detailed performance case study of the Pagosa hydrodynamics application on two systems [64]. We use Pagosa as a case study to demonstrate how our methods and metrics can be applied to large applications. Applying the hardware counter-based analysis to Pagosa gives a better understanding of how optimizations impact the performance of the application. This case study uses empirically measured rooflines (see Chapter IV) to provide context for the performance, while looking at the detailed hardware use through our hardware counter metrics (Chapter V). This case study brings together the work from all the other chapters to show that our performance method can be applicable to large scientific applications.

7.1 Target Application

Pagosa is a 3D multi-physics continuum mechanics simulation application. It uses a 3D structured Cartesian mesh which is distributed evenly across MPI ranks. This distribution facilitates load balancing. Pagosa solves the physics model equations using explicit time integration which avoids more complex matrix solving methods. The computation is dominated by loops iterating over the three dimensions of the mesh which results in contiguous memory operations. These features make Pagosa a useful case study of our methods.

We use a "hotspots" analysis (Fig. 52) of Pagosa to compare the execution time of key functions in Pagosa. This analysis does not show a single function that takes up most of the execution time. Many scientific applications, and especially mini-applications, are dominated by one or two kernels of computation that take up most of the runtime. In contrast, Pagosa has several kernels that each take around

5% of the runtime and one that takes approximately 22%. This application profile forces us to consider multiple kernels in our analysis since an improvement to an individual kernel will have only a limited impact on the overall application.

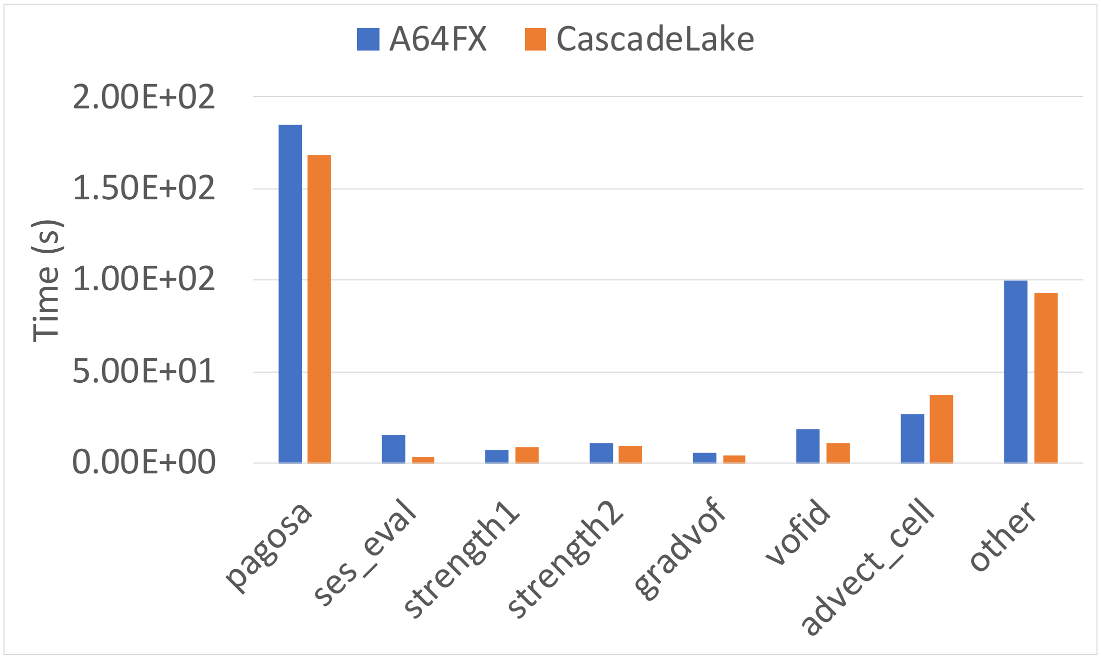


Figure 52. Hotspots of Pagosa. Compares the execution time of key functions in the application to identify which are most time-consuming.

For the case study, we select three functions that make up approximately 35% of the runtime for a given simulation timestep. The first function is Vofid (Section 7.2) which previous work has 2.7× and 7.7× speedup on the A64FX and Cascade Lake. The second two, Strength 1 and Strength 2 (Section 7.3), are very similar to each other. From the original version to the final optimization we see approximately 2.5× speedup on A64FX and an average of 3.7× speedup on the Cascade Lake. For both functions, we discuss the progression of optimizations, what impacts are expected from those optimizations, and then the results that we see from the hardware counters and how those compare to the expectations.

7.2 Vofid

Our first optimization target is the Vofid function which contains multiple kernels each of which iterates over the entire mesh to perform computations. These computations are confined to individual mesh points, so MPI communication can be avoided. In total there are four groups of operations (see Alg. 17), three of which are contained in separate function calls. The optimizations presented here center around the manipulation of these kernels to improve the cache use and vectorization of the function.

Algorithm 17 Vofid Algorithm

```
3D mesh operations group A (as subroutine call)
3D mesh operations group B (as subroutine call)
3D mesh operations group C (as array syntax)
3D mesh operations group D (as subroutine call)
```

7.2.1 Versions.

We look at the original version and five optimizations of the Vofid function, to explore how the changes impact the application performance on our two architectures. Note that these optimizations were implemented by various collaborators and some of the work is explained in [65], but the analysis of the work is presented here for the first time.

The *Original* version of the kernel was written in Fortran array syntax. This syntax allows Fortran developers to write operations that occur on each array element as a single line of code. For Vofid, this effectively means writing a series of operations that are applied to each mesh point. Array Syntax is relatively easy to use and understand but is compiled as a long series of 3D loop nests (see Algorithm 18 for an example loop nest), which can be challenging for the compiler to improve. In particular, the array syntax causes the kernel to iterate over the mesh many times, undermining the ability of the cache to reuse data effectively.

Algorithm 18 A loop nest over a 3D mesh.

```
for $x$ dimension
    for $y$ dimension
        for $z$ dimension
            computations on data at a single mesh point
```

The first step in improving the performance, *OPT 1A*, is to remove the array syntax. Array syntax allows developers to easily manipulate Fortran Arrays but effectively separates each operation into its own loop nest which prevents data reuse in the caches [66]. Recognizing the limits of array syntax, the developers removed it from Vofid and the subroutines called from it.

Removing array syntax is a three-step process. First, the array syntax is transformed into a series of nested loops. This step has limited impact on the performance but is necessary to enable the next two. Second, the loop nests are fused into a single nest which can reuse the data in the caches. Finally, array temporaries which are necessary for array syntax can be converted into scalar temporaries. The fusion of loop nests allows the application to make better use of the cache and the transformation of array temporaries into scalar temporaries reduces the total amount of data required.

In the case of Vofid and its subroutines, the arrays are 3 dimensional, so the loop nests that result have three for loops each and the temporary arrays that are removed are 3D as well. After the three steps are followed to remove the array syntax, Vofid has four loop nests which correspond to the four groups of computation in Algorithm 17. This aim of removing array syntax for *OPT 1A* is to reduce the total amount of data movement and improve the cache efficiency of Vofid.

In optimization *OPT 1B*, we take an incremental step to reduce the depth and complexity of callpaths. In the *Original* version, Vofid calls three subroutines each of which select from a set of other subroutines to complete the operation. However, Vofid only uses one of the possible options for each of the calls. Identifying this limited choice, the developers adjusted Vofid to directly call the subroutines which complete the operations required. Removing unnecessary potential paths can be an effective way to assist compilers' optimization efforts.

Additionally, some of the array temporaries in Vofid were not removed in version *OPT 1A*, so version *OPT 1B* completes the process.

Table 21. Versions of the Vofid kernel

Version	Description
Original	Vofid before changes
Production	The production version of the code, which is similar to the original
OPT 1A	Remove array syntax, fuse loops, eliminate array temporaries
OPT 1B	additional array temps removed, direct calls to functions
OPT 1C	Manual inlining of functions
OPT 2A	Fusing of loops
OPT 2B	Collapse Triple loop into a single loop

OPT 1C is a second incremental step in the process of optimization. The developers manually inlined the three subroutines into Vofid. This step turns the Vofid kernels into a set of four 3D loop nests without costly function calls. Removing function calls can reduce overheads, but the main goal of the change is to enable the changes that follow.

Once the subroutines are inlined, Vofid is a sequence of 3D loops each of which iterates over the whole of the mesh. The primary change in *OPT2A* is to fuse the four loop nests into a single loop nest. Previously, the data for each mesh point would need to be reloaded once for each of the loop nests. By fusing the loops in *OPT2A*, we can eliminate more temporary arrays to reduce memory footprint size. This smaller data volume is designed to improve performance by reducing the amount of time spent waiting on data being loaded. These changes are similar to the process of removing array syntax, so we expect that they will reduce the amount of data movement necessary and improve cache efficiency.

After fusing the loops, the developers apply an optimization known as scalar replacement (Algorithm 19). This optimization replaces repeated array accesses to the same element with a single access and a temporary scalar variable. It sounds similar to removing the temporary arrays, but it replaces single accesses. The purpose of the change is to help the compiler keep important variables in registers which reduces unnecessary load and store operations.

Algorithm 19 An example of a loop without and with scalar replacement. The optimization is more useful when data elements are used several times, but this example illustrates the concept.

```
for some $i$                                   ▷ without scalar replacement
    $A(i) = B(i) * C(i)$
for some $i$                                   ▷ with scalar replacement
    $A_i = A(i)$
    $B_i = B(i)$
    $C_i = C(i)$
    $A_i = B_i * C_i$
    $A(i) = A_i$
```

The most recent modification to the Vofid kernel, *OPT 2B*, collapses the 3D triple loop into a single loop that iterates over the total number of mesh points. It also Fortran's default memory layout allows this change to be made to loops without modification to the data structure. In theory, such a modification should simplify the memory access pattern to improve performance.

7.2.2 Results. For each of the versions of the Vofid Kernel, we collect hardware counter data to produce the metrics presented in Chapter V. The results are displayed in Tables 22 and 23. For an overall picture of performance, we show the roofline and application points of the kernel (Figures 53 and 54), as discussed in Chapter IV. The combination of roofline and Hardware Counter Metrics provides an overview of the performance of the kernel and detailed information on the hardware use.

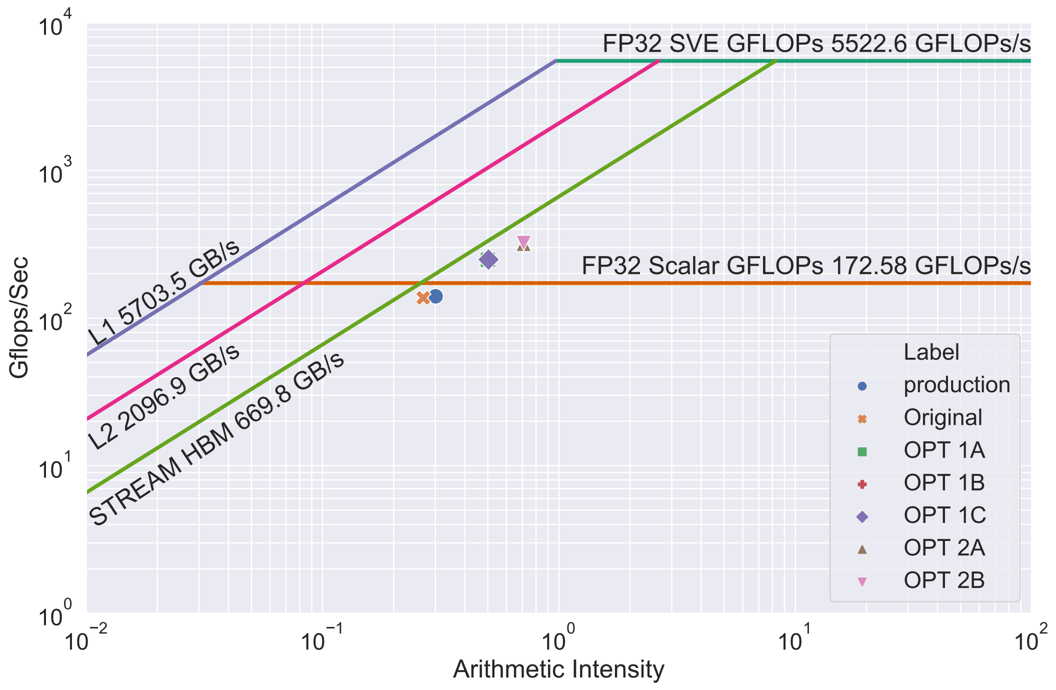


Figure 53. Vofid roofline on the A64FX. The Original and Production versions have similar floating point rates. OPT 1A, 1B, and 1C form another performance grouping, while OPT 2A and 2B form a third grouping. All of the points sit just below the main memory ceiling which implies that memory bandwidth may be a limiting factor in the performance.

Starting with the A64FX roofline (Figure 53), we note that the rate of computation increases with each level of optimization. This trend is confirmed by looking at the Overall metrics in Table 22. Optimizations *OPT 1A*, *OPT 1B*, and *OPT 1C* all complete in about 8 seconds, while *OPT2A* and *OPT 2B* complete in about 6.25 seconds.

The application points on the roofline run along the Main Memory line which implies that the main memory bandwidth may be a limiting factor of the performance. The optimizations can move the kernel above one of the floating-point ceilings, so it may be limited by floating-point operations as well as memory operations. The hardware counter metrics (Table 22) can help us make sense of the placement of the application points on the roofline.

Our hardware counter metrics on the A64FX (Table 22) confirm the expectations we have from our knowledge of the optimizations and the roofline analysis. Removing array syntax and loop fusion is similar in the sense that both enable the kernel to perform all the operations on a mesh point before moving to the next point. The impact is seen in all of the data movement metrics. *LS, L2, and Mem Bytes* all decrease with the changes, which shows that the kernel is reusing more data in the CPU and at each level of cache. Additionally, the *L1 and L2 MR* and the *L2 and Mem / LS* measurements show improved efficiency. This combination of impacts shows that the optimizations are both using fewer data and using the caches more efficiently.

The floating-point metrics show less improvement than the data movement metrics. The *Production* version currently makes less use of SIMD operations than the other variations, but the original is as well vectorized as the optimized versions. The *flops / fp ins* metrics hovers around 14 for each variation. A64FX with single precision can have a ratio as high as 32 if 512-bit SVE and fused-multiply-add operations are fully utilized. Analysis of the assembly language or the optimization reports would help the developers identify if these operations are being left out unnecessarily.

We note that there is almost no change in the *IPC* for any of the versions on A64FX. This metric implies that the pipeline efficiency is not improved by the optimizations. On benchmarks (see Chapter VI), the *IPC* regularly rises above 2, so there is plenty of space for improvement. The low *IPC* may be the result of numerous conditional statements in the innermost loop, but more experimentation would be required to verify this hypothesis.

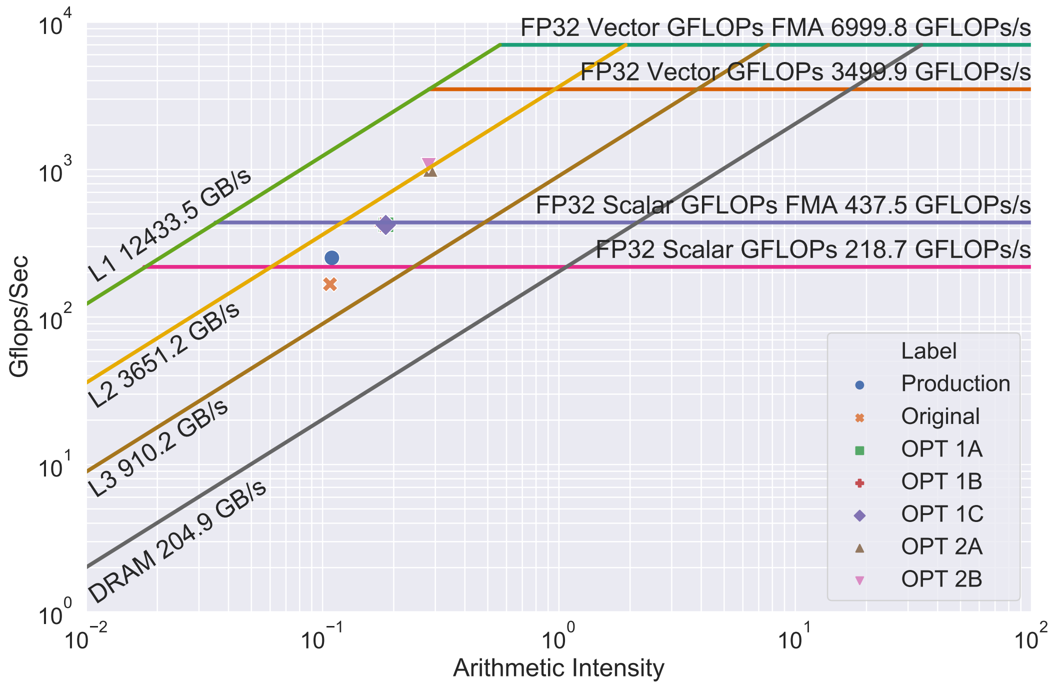


Figure 54. Vofid roofline on the Cascade Lake. The Original is slightly worse than Production on the Cascade Lake. OPT 1A, 1B, and 1C form another performance grouping, while OPT 2A and 2B form a third grouping. All of the points sit just below the L2 Cache bandwidth ceiling which implies that memory bandwidth may be a limiting factor in the performance.

Figure 54 shows the Vofid application points on the Cascade Lake roofline. This plot along with the overall performance metrics in Table 23 show that the performance trend on Cascade Lake are similar to what we've seen on A64FX.

Optimizations *OPT 1A*, *OPT 1B*, and *OPT 1C* complete in 6.6 seconds while optimizations *OPT2A* and *OPT 2B* complete in between 2 and 3 seconds.

There are some noticeable differences in the performance of the A64FX. *OPT 2B* has negligible impact on A64FX, but it shows a 1.27× speedup on Cascade Lake when compared to *OPT 2B*. This speedup is at least in part due to the reduction in **LS Bytes** and **flops** seen between *OPT2A* and *OPT 2B*. These work metrics are reduced by the flattening of the loop nest into a single loop, but most of the ratio metrics remain unchanged.

Additionally, the Cascade Lake application points reside along the L2 Cache ceiling rather than Main Memory Ceilings as on the A64FX. This difference is likely related to the cache size; A64FX has a 32 MiB L2 cache while Cascade Lake has a 48 MiB L2 and 35.75 MiB L3 cache. Experimentation with different input decks may provide additional evidence of the benefits of larger caches.

A final difference from the A64FX results is that the optimizations improve the **IPC** on the Cascade Lake system. This metric implies that the kernel is making better use of the CPU pipeline with each change.

7.2.3 Takeaways. Our analysis method reveals how the optimizations applied to the kernel have improved memory performance and how they have been unsuccessful in improving floating-point performance. This information is vital for understanding the performance of kernels but can be expanded further. Careful examination of the instruction mix or stall counts may provide some insight into the lack of **IPC** improvement. Comparing measurements of this kernel on the new systems could reveal which new metrics are related to the **IPC** and which are not. Alternatively, a benchmark could be based on the kernel to produce similar performance patterns. Such work is a target for future exploration.

Table 22. Counter results for the Vofid kernel on A64FX

	Production	*Original*	*OPT 1A*	*OPT 1B*	*OPT 1C*	*OPT2A*	*OPT 2B*
Overall Performance							
Time (s)	18.3	16.6	8.09	8.08	8.07	6.28	6.24
flops/s (Gflops/s)	140.27	136.70	248.69	249.15	249.21	319.46	321.54
IPC	0.65	0.53	0.57	0.57	0.56	0.64	0.65
Data Movement							
LS Bytes (GB)	8,538.04	8,497.08	4,018.64	3,997.47	3,993.49	2,828.07	2,828.22
L2 Bytes (GB)	2,648.85	10,664.80	2,643.57	2,640.66	2,639.27	584.78	581.64
Mem Bytes (GB)	1,652.74	6,739.96	1,652.45	1,652.82	1,651.90	361.41	360.41
Cache Efficiency							
L1 MR	0.0065	0.033	0.022	0.023	0.022	0.0078	0.0073
L2 MR	0.28	0.30	0.28	0.28	0.28	0.28	0.28
L2 / LS	0.31	1.26	0.66	0.66	0.66	0.21	0.21
Mem / LS	0.19	0.79	0.41	0.41	0.41	0.13	0.13
LD ins / ST ins	1.76	2.90	2.00	1.89	1.95	2.50	2.58
Computation							
flops (Gflops)	2,568.02	2,266.14	2,012.72	2,012.73	2,011.61	2,004.7	2,005.81
flops/fpins	10.80	14.38	13.99	13.99	14.08	14.29	14.20
Computation Data Ratios							
AI	0.30	0.267	0.50	0.50	0.50	0.71	0.71
flops/LD ins	13.69	14.26	33.48	36.26	34.53	40.34	39.34
flops/ST ins	24.03	41.35	66.90	68.41	67.45	100.84	101.52
flops/LD Bytes	0.47	0.40	0.82	0.82	0.83	1.05	1.02
flops/ST Bytes	0.84	0.81	1.30	1.30	1.29	2.18	2.19

Table 23. Counter results for the Vofid kernel on Cascade Lake

	Production	*Original*	*OPT 1A*	*OPT 1B*	*OPT 1C*	*OPT2A*	*OPT 2B*
Overall Performance							
Time (s)	11.20	16.90	6.61	6.61	6.63	2.80	2.19
flops/s (Gflops/s)	251.00	167.00	422.00	421.00	420.00	998.00	1,070.00
IPC	0.59	0.54	0.74	0.75	0.75	1.33	1.39
Data Movement							
LS Bytes (GB)	25,712.60	26,210.50	15,021.70	15,595.20	15,049.50	9,792.19	8,315.37
L2 Bytes (GB)	1,591.43	3,798.14	1,129.24	1,127.93	1,146.16	221.44	215.34
L3 Bytes (GB)	1,555.57	4,645.86	1,418.29	1,419.86	1,419.93	275.34	275.46
Mem Bytes (GB)	1,621.59	3,168.31	1,093.90	1,096.34	1,098.92	232.09	231.39
Cache Efficiency							
L1 MR	0.062	0.15	0.075	0.073	0.076	0.023	0.026
L2 MR	0.98	1.22	1.26	1.26	1.24	1.24	1.28
L3 MR	0.61	0.41	0.44	0.45	0.44	0.54	0.54
L2 / LS	0.062	0.14	0.075	0.072	0.076	0.023	0.026
L3 / LS	0.060	0.18	0.094	0.091	0.094	0.028	0.033
Mem / LS	0.063	0.12	0.073	0.070	0.073	0.024	0.028
LD ins / ST ins	2.83	2.39	2.89	3.02	2.84	3.92	3.97
Computation							
flops (Gflops)	2,815.58	2,818.34	2,785.22	2,785.22	2,785.22	2,799.02	2,339.42
flops/fpins	16	16	16	16	16	16	16
Computation Data Ratios							
AI	0.11	0.11	0.19	0.18	0.19	0.29	0.28
flops/LD ins	9.49	9.76	15.97	15.22	16.02	22.96	22.55
flops/ST ins	26.83	23.33	46.21	45.90	45.48	90.03	89.42
flops/LD Bytes	0.15	0.15	0.25	0.24	0.25	0.36	0.35
flops/ST Bytes	0.42	0.36	0.72	0.72	0.71	1.41	1.40

7.3 Strength 1 and Strength 2

The Strength 1 function computes the impact of stresses, shears, and other factors on the materials in the simulation. This function advances these factors through the first half of the time step. The Strength 2 function completes the computation of stresses, shears, and other factors that impact the materials in the simulation. In both functions, the computation is focused on individual materials and at each point in the mesh. Therefore, like vofid, the kernel does not require MPI communication between ranks. In contrast to Vofid, there is an extra loop required to iterate over the materials in the simulation.

7.3.1 Versions. In this section, we analyze four versions of the Strength functions. The optimizations are similar to Vofid, but with some variations specific to these functions. A major difference from Vofid is that Strength 1 and 2 use different computation types with different materials. For this reason, there are many instances of conditional logic which determine how the function proceeds. Several optimizations focus on more efficient handling of the conditional logic in Strength 1, Strength 2, and the subroutine.

In the *Original* version of both Strength 1 and Strength 2, the kernel uses array syntax to process the 3D arrays similar to Vofid. The functions also make several calls to other subroutines to handle different material cases and aspects of the computation. Those subroutines have additional array syntax and conditional logic to handle particular materials.

The first step in optimization is now included in the *Production* version of Pagosa. The developers removed the array syntax in the Strength1, but not the subroutines called from it. The *Optimization 0* version is similar to the *Production* version, but with compiler directives that collapse the nested loops. This directive only works with the compilers on the A64FX, so *Production* and *Optimization 0* are equivalent on the Cascade Lake.

Optimization 1 builds on these changes by fusing additional loops within the Strength functions function which enables removing temporary arrays. The loop fusing is aimed at using caches more efficiently, while fewer temporary arrays reduce the total amount of memory needed.

Optimization 2 applies the optimizations from the Strength functions to two of the most influential subroutines called from those functions. These subroutines contribute a significant amount of the computation time, so applying the same optimizations to them is a necessary step to maximize the performance of the Strength functions.

Finally, in *Optimization 3* the developers inline the two subroutines that were improved in *Optimization 2*. Inlining can help some compilers improve the optimization and reduce the overhead from numerous calls to subroutines. In this case, the conditional logic in the Strength functions and the subroutines complicate the process of inlining by requiring the developers to handle various special cases. Such changes are currently intractable for compilers to perform automatically and harm the maintainability of the application for human developers. Conducting performance evaluations on these changes informs developers of this, and other, applications about which optimization is sufficiently performant to justify increased code complexity. Compiler teams can also use these evaluations to identify and prioritize areas of improvement for their compilers.

7.3.2 Results. The *Original* version of Strength 1 performs most of the computation in array syntax and several subroutines that are called for each of the materials in use (25 for our input deck). We plot the Strength 1 variations

Table 24. Versions of the Strength functions.

Version	Description
Production	Removes array syntax
Original	Strength functions before changes
Optimization 0	Similar to the production version, but with additional compiler pragmas
Optimization 1	Additional loop fusion and removal of temporary arrays
Optimization 2	Subroutines have array syntax removed and resulting loops fused together
Optimization 3	Subroutines are inlined

on rooflines in Figures 55 and 56 and the Strength 2 application points in Figures 55 and 56. On both systems, there is limited change in the application point placement for the versions except *Optimization 3* which outperforms the others.

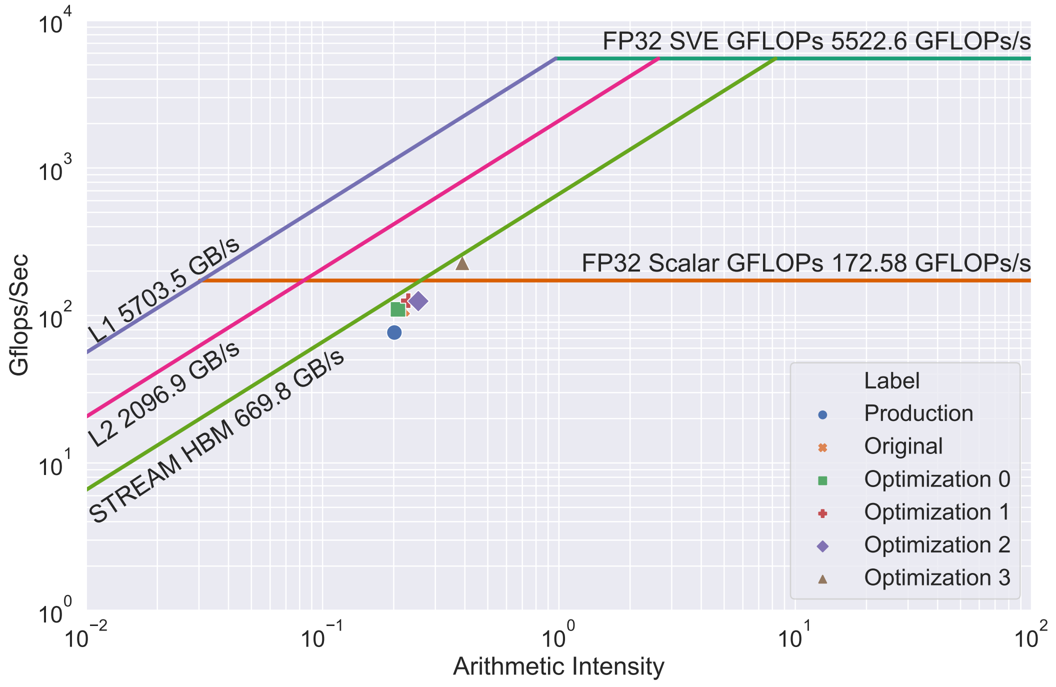


Figure 55. Strength 1 roofline on the A64FX. The AI improves consistently along with the computational rate for each optimization. The consistent placement of the application points below the main memory ceiling indicates that improving cache efficiency could improve the overall performance.

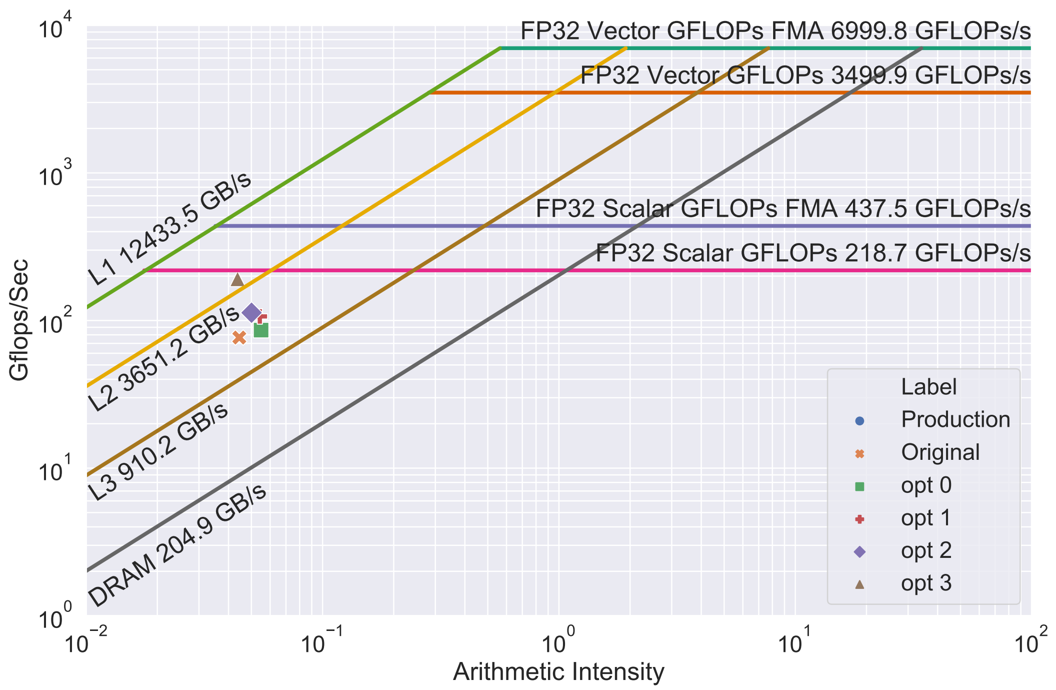


Figure 56. Strength 1 roofline on the Cascade Lake. There is limited change in the AI from the *Original* version, but consistent improvement in the rate of computation with each optimization. *Optimization 3* break above the L2 cache bandwidth line and could be limited by data movement or computation bottlenecks.

We show the hardware counter results for Strength 1 in Table 25 (A64FX) and Table 26 (Cascade Lake). The overall metrics show similar results to the roofline plots; the optimizations slowly improve performance until a larger increase with version *Optimization 3*. Much like Vofid, the improvement is seen in the execution time and the **flops/s**, but generally not in the **IPC**.

One unique point on the A64FX is that WDN V02 has a 1.13× speedup compared to WDN V00, but the **flops/s** is less than 1% different between the two. In this case, the optimizations reduce the number of **flops**, so the overall performance is improved without impacting the rate of computation.

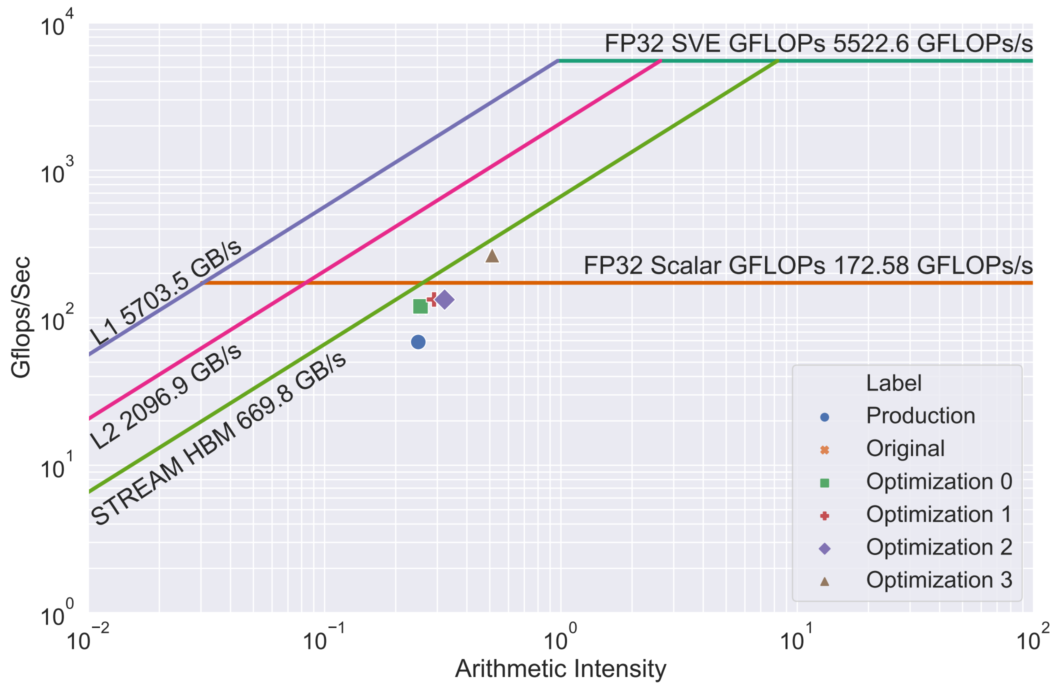


Figure 57. Strength 2 roofline on the A64FX. The AI improves consistently along with the computational rate for each optimization. The consistent placement of the application points below the main memory ceiling indicates that improving cache efficiency could improve the overall performance.

The data movement results show some similarities and differences between the two systems. Overall the optimizations result in less data movement at each level of the memory and more efficient use of the caches. The details of how much change occurs at each level differ between the systems.

The **LS Bytes** of the two systems differ dramatically. In the *Production* version, the A64FX load and store instructions move approximately 2,663.32 GB of data while the Cascade Lake moves 13,578.50. On both systems, the optimizations reduce the amount of data moved by a consistent amount, so the difference consistently remains around 10,000 GB. A similar difference in the data volume is

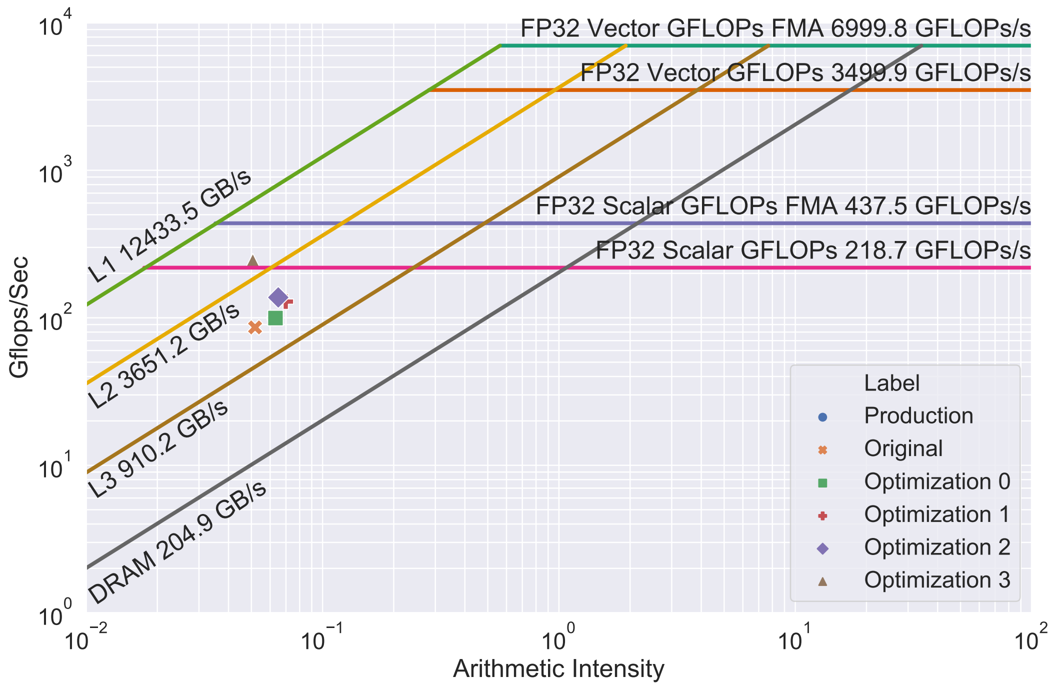


Figure 58. Strength 2 roofline on the Cascade Lake. There is limited change in the AI from the *Original* version, but consistent improvement in the rate of computation with each optimization. *Optimization 3* breaks above the L2 cache line and may be limited by data movement or computation bottlenecks.

observed in the Vofid functions and the Strength 2 function, but not in the kernels and mini-apps that we evaluated in Chapter VI.

Having identified the discrepancy between the two systems, we have two questions. First, why does the Cascade Lake load so much additional data? It could be that there are more register spills in the Cascade Lake version of the application, or the compiler may be repeatedly re-loading the array indexes. The second question is; do we think this is a major factor in the performance of the kernel? Compared to the Cascade Lake, Strength 1 has a faster execution time on A64FX, but Vofid has a slower execution. Both kernels move far more data when running on the Cascade Lake, but Strength 1 uses vector operations less efficiently on Cascade Lake than it does on A64FX. These trends are not proof that the use of SIMD instructions is more important than the number of load and store operations, but it does offer another direction of study.

Additional experiments could be designed using different data sizes and compilers on both systems. These experiments could assess the performance impact of register spills compared to caches and of the data movement compared to floating-point operations.

At the cache level, we observe that the **L2 Bytes** is lower for the Cascade Lake than for the A64FX. This trend holds across all of the optimization versions. We find this result surprising because the L1 cache on Cascade Lake is smaller than in A64FX and because it contrasts with the difference in L1 cache accesses mentioned above. Based on this observation and the **L2 Bytes/LS Bytes** metric, we can conclude that the Cascade Lake is using the L1 cache very efficiently despite the numerous extra accesses to it.

For both systems, the optimizations have two major successes. Removing array syntax, fusing loops, and transforming temporary arrays into scalars reduced data use considerably. The **L2 Bytes** is reduced by a factor of 6.3× on A64FX and 6.9× on Cascade Lake from the original code version to the final version in this study. Additionally, the optimizations reduce the number of floating-point operations required by factors of 1.2× on A64FX and 1.4× on Cascade Lake.

Strength 1 on the A64FX also sees a significant improvement in the **AI** since more data movement is eliminated than floating-point operations. The rise in **AI** could allow the kernel to utilize more of the computational potential of the CPU and SIMD instructions if it can keep those resources supplied with data to be processed. Due to the extra memory operations on the Cascade Lake, the same improvement to the **AI** does not occur on that system. Therefore, the kernel may be prevented from achieving higher performance by the L1 cache rather than the lower levels of the memory hierarchy as on the A64FX.

7.3.3 Take Aways. For both systems, SIMD operations offer an area for potential improvement. Both systems fall considerably short of the potential **flops/fp ins**, and the optimizations have had only a limited impact on that metric so far. The bandwidth of data between the CPU and the L1 cache is also improved with the use of SIMD instructions. Compiler options, pragmas, or the simplification of conditional logic could all have an impact on the vectorization.

Unfortunately, Strength 1 cannot fully utilize the SIMD operations if the data cannot be loaded in quickly enough. The cache efficiency metrics show that on A64FX the kernel fills a significant fraction of the loads and stores from the main memory or the L2 cache which prevents full utilization of the computational resources. In contrast, the kernel's loads and stores are nearly all filled from the L1 cache on the Cascade Lake, but there is far too much data moved for each floating-point operation to utilize the computational resources effectively.

Therefore, on Cascade Lake, identifying and eliminating the extra loads and stores should be the next priority. On A64FX, the lack of cache efficiency needs to be addressed so that more data is available in the L1 cache when it is needed.

Lastly, the kernel contains a significant amount of conditional logic that can be detrimental to performance. In particular, conditionals can prevent long latency memory operations from being properly overlapped with other operations which increases the impact of cache misses. We currently do not have a metric to assess the frequency of conditional operations. In future work, we aim to develop such a metric and apply it to these kernels.

Table 25. Counter results for the Strength 1 kernel on A64FX

	Production	*Original*	*Optimization 0*	*Optimization 1*	*Optimization 2*	*Optimization 3*
Overall Performance						
Time (s)	7.20	5.33	5.00	4.23	3.76	2.10
flops/s (Gflops/s)	76.63	109.41	110.04	126.12	125.15	227.26
IPC	0.37	0.38	0.40	0.40	0.37	0.44
Data Movement						
LS Bytes (GB)	2,741.41	2,663.32	2,638.85	2,313.23	1,850.05	1,221.3
L2 Bytes (GB)	6,162.85	4,579.48	4,154.09	3,352.47	2,832.36	720.49
Mem Bytes (GB)	1,798.96	2,034.39	1,877.17	1,366.74	1,072.55	346.38
Cache Efficiency						
L1 MR	0.029	0.052	0.047	0.047	0.047	0.026
L2 MR	0.18	0.25	0.25	0.23	0.22	0.25
L2 / LS	2.25	1.72	1.57	1.45	1.53	0.59
Mem / LS	0.66	0.76	0.71	0.59	0.58	0.28
LD ins / ST ins	2.80	4.10	3.63	3.59	3.81	5.03
Computation						
flops (Gflops)	551.92	583.26	550.24	533.20	470.20	476.58
flops/fpins	13.11	19.06	18.95	19.12	19.36	19.25
Computation Data Ratios						
AI	0.20	0.22	0.21	0.23	0.25	0.39
flops/LD ins	7.69	10.71	11.25	14.05	14.13	21.03
flops/ST ins	21.53	43.93	40.85	50.50	53.85	105.76
flops/LD Bytes	0.27	0.29	0.28	0.31	0.34	0.48
flops/ST Bytes	0.77	0.94	0.80	0.91	0.992	2.08

Table 26. Strength 1 counter results on Cascade Lake

	Production	*Original*	*Optimization 0*	*Optimization 1*	*Optimization 2*	*Optimization 3*
Overall Performance						
Time (s)	8.64	9.96	8.65	6.78	5.70	2.74
flops/s (Gflops/s)	86.20	76.70	86.10	107.00	113.00	192.00
IPC	0.67	0.60	0.67	0.63	0.72	1.46
Data Movement						
LS Bytes (GB)	13,578.50	17,203.80	13,555.10	13,318.30	12,867.90	12,006.00
L2 Bytes (GB)	1,657.35	1,989.45	1,663.31	1,381.90	1,133.14	288.39
L3 Bytes (GB)	1,826.36	2,524.23	1,826.93	1,240.71	957.34	217.58
Mem Bytes (GB)	1,644.71	1,962.94	1,645.05	1,211.23	980.914	293.125
Cache Efficiency						
L1 MR	0.083	0.11	0.084	0.095	0.081	0.022
L2 MR	1.10	1.27	1.10	0.90	0.85	0.77
L3 MR	0.62	0.55	0.62	0.67	0.72	0.91
L2 / LS	0.12	0.12	0.12	0.10	0.088	0.024
L3 / LS	0.13	0.15	0.13	0.093	0.074	0.018
Mem / LS	0.12	0.11	0.12	0.091	0.076	0.024
LD ins / ST ins	2.34	2.59	2.28	2.10	2.15	2.07
Computation						
flops (Gflops)	744.76	764.39	744.76	722.76	644.23	524.64
flops/fpins	10.83	14.57	10.83	14.51	14.35	14.02
Computation Data Ratios						
AI	0.055	0.044	0.055	0.054	0.050	0.044
flops/LD ins	3.40	3.61	3.43	4.67	4.25	3.69
flops/ST ins	7.95	9.33	7.82	9.83	9.11	7.64
flops/LD Bytes	0.078	0.062	0.079	0.080	0.073	0.065
flops/ST Bytes	0.18	0.16	0.18	0.17	0.16	0.14

Table 27. Strength 2 counter results on A64FX

	Production	*Original*	*Optimization 0*	*Optimization 1*	*Optimization 2*	*Optimization 3*
Overall Performance						
Time (s)	10.70	6.33	5.94	5.43	4.96	2.49
flops/s (Gflops/s)	68.51	117.64	119.54	133.29	132.87	266.36
IPC	0.31	0.36	0.37	0.38	0.33	0.45
Data Movement						
LS Bytes (GB)	2,942.87	2,885.69	2,789.31	2,488.92	2,045.61	1,292.96
L2 Bytes (GB)	11,725.80	5,384.28	5,030.79	3,842.94	3,362.47	935.58
Mem Bytes (GB)	1,986.97	2,328.44	2,122.05	1,469.80	1,176.07	354.69
Cache Efficiency						
L1 MR	0.043	0.056	0.056	0.045	0.045	0.034
L2 MR	0.11	0.25	0.24	0.22	0.20	0.21
L2 / LS	3.98	1.87	1.80	1.54	1.64	0.72
Mem / LS	0.68	0.81	0.76	0.59	0.57	0.27
LD ins / ST ins	3.02	3.87	3.67	3.63	3.93	4.51
Computation						
flops (Gflops)	733.33	744.55	710.18	723.10	658.76	662.02
flops/fpins	12.84	19.34	19.32	19.64	19.88	19.76
Computation Data Ratios						
AI	0.25	0.26	0.25	0.29	0.32	0.51
flops/LD ins	8.20	12.87	13.35	16.01	15.84	28.76
flops/ST ins	24.74	49.79	49.00	58.07	62.29	129.58
flops/LD Bytes	0.34	0.34	0.35	0.40	0.45	0.65
flops/ST Bytes	0.94	1.05	0.95	1.05	1.14	2.44

Table 28. Counter results for the Strength 2 kernel on Cascade Lake

	Production	*Original*	*Optimization 0*	*Optimization 1*	*Optimization 2*	*Optimization 3*
Overall Performance						
Time (s)	9.61	11.40	9.61	7.29	6.26	2.95
flops/s (Gflops/s)	99.60	86.20	99.60	129.00	137.00	242.00
IPC	0.63	0.57	0.63	0.59	0.68	1.37
Data Movement						
LS Bytes (GB)	15,256.10	18,936.50	15,145.20	13,437.70	13,245.20	14,087.80
L2 Bytes (GB)	1,922.95	2,556.05	1,920.29	1,491.70	1,241.09	300.834
L3 Bytes (GB)	2,034.08	2,912.70	2,034.15	1,252.41	969.33	223.46
Mem Bytes (GB)	1,817.53	2,242.81	1,817.45	1,270.96	1,038.47	310.47
Cache Efficiency						
L1 MR	0.093	0.13	0.093	0.10	0.088	0.021
L2 MR	1.06	1.14	1.06	0.84	0.78	0.76
L3 MR	0.61	0.54	0.61	0.68	0.73	0.92
L2 / LS	0.13	0.13	0.13	0.11	0.094	0.021
L3 / LS	0.13	0.15	0.13	0.093	0.073	0.016
Mem / LS	0.12	0.12	0.12	0.095	0.078	0.022
LD ins / ST ins	2.32	2.67	2.27	2.13	2.11	2.33
Computation						
flops (Gflops)	957.24	981.16	957.24	938.99	860.46	714.16
flops/fpins	11.66	14.87	11.66	14.82	14.72	14.49
Computation Data Ratios						
AI	0.063	0.052	0.063	0.070	0.065	0.051
flops/LD ins	4.19	4.25	4.26	6.11	5.67	4.25
flops/ST ins	9.75	11.33	9.66	13.01	11.98	9.90
flops/LD Bytes	0.090	0.071	0.091	0.10	0.096	0.072
flops/ST Bytes	0.21	0.19	0.21	0.22	0.20	0.17

7.4 Summary

This chapter presents a final case study that demonstrates how our metrics may be used in practice. We examined three functions which contribute a significant portion of the execution time of the Pagosa application. Each of the functions has several versions that the developers have optimized to improve runtime. We the roofline and other hardware counter data to conduct an analysis of these optimization attempts and compare the results to the expectations of the developers. This analysis shows how the Roofline Model and our metrics can be applied to a full application to improve the understanding of its performance.

CHAPTER VIII

CONCLUSION

For many years, Hardware Performance Monitors have offered the possibility of deep insight into the performance of applications. Performance analysis using hardware counters is often limited to a particular CPU. This lack of portability comes from the counters that are available and the microarchitectural differences in CPUs. Our research shows that hardware counters can be used to measure performance metrics portably across different CPU types.

We began the dissertation with a central hypothesis: that hardware counter metrics can provide actionable performance information to users on a diverse set of CPU types. In this dissertation we have shown that hypothesis to be accurate in four steps. First (Chapter IV), we used hardware counters to support performance analysis based on the Roofline Model. The metrics involved were common to all modern CPUs, so we contributed a novel method of measurement using hardware counters on two distinct CPU types. In Chapter V, we presented a set of additional hardware counter performance metrics. We show that these metrics are measurable on both CPUs and demonstrate those measurements with benchmark examples. Next we apply the set of performance metrics to a variety of mini-applications in Chapter VI. These demonstrations show that the performance metrics we developed are useful for performance analysis of computationally intensive scientific applications. Finally, we use the combination of roofline-based performance analysis and the additional metrics to examine the performance of the Pagosa multi-physics application. This case study shows that the method is applicable to full applications.

Going forward, we would like to continue work on both VPIC and Pagosa. These application have many more kernels than those analyzed in this dissertation, and our metrics may be able to provide additional insight into those algorithms as well. Both applications are also under active development, so there are always new variations to consider. One important study to consider is the impact of input types on the performance metrics. Some types of input may have different performance characterizations than others, so we could explore such possibilities with a selection of mini-application and the two full applications.

Our metrics could be applied automatically through an autotuner. For example, loop blocking is designed to improve the cache efficiency of a kernel and can be applied automatically [54]. An autotuner could collect our metrics and use the resulting information to prune the search space. Such pruning may reduce the amount of time required for autotuning by identifying block size more rapidly than naive search methods.

Despite the success of our existing metrics, there are many other possibilities to consider. We have no doubt that additional performance metrics could be identified. For example, branching and control logic has a significant impact on many applications, so we would like to add metrics that measure the impact of branching on performance. Although we focus on floating point arithmetic, integer operations can impact performance as well. These operations are often used to compute memory addresses which can impact the ability of the application to load data in a timely manner. Metrics based on the mix of instructions in an application to give the user an measure of the relative impact of memory, floating point, control, and integer operations. We hope to identify such metrics in the near future.

In this dissertation, we only considered the Cache Aware Roofline Model. Currently, we are collaborating with the developers of the Adaptive Cache Aware Roofline Method (adCARM) [44] to use hardware counters to support their method of tailoring rooflines to the specific application. This method provides the user with more realistic projections of potential performance, given the types of operations used by the kernel.

So far we have demonstrated portability between only two systems. Ideally, we could expand to many other types of CPU; however, there is no guarantee that portability between the CPUs discussed in this dissertation implies portability to others. This work would proceed in stages. First, we would identify a method to count **flops** and validate the counters with our benchmarks. Then we would identify and validate the counters for **LS Bytes**. Once these initial measurements are established, we can move through each metric adding counters as necessary. Finally, we could repeat the evaluation of our benchmarks and mini-applications on the new CPU type.

Finally, we would like to extend the methodology to GPUs and other accelerators which are becoming vital for HPC. These types of processing units often are paired with a CPU and have a memory subsystem that is a combination of memory local to the processor and memory accessed through the CPU. This arrangement is similar to the cache levels that we based these metrics on, but it may require significant alteration to the data movement metrics to effectively inform users about performance. For computational metrics, GPU developers use **occupancy** to assess how efficient the application uses the available threads. We would seek to measure this and other GPU metrics using hardware counters.

For other types of future accelerators, there are so many possibilities that it is impossible to know just how these will be implemented and what types of hardware counters (if any) will be available. Experiments on GPUs and NEC Vector engines could offer some insight into the potential for portability of accelerator measurements or into the limitations.

Current trends [8] suggest a proliferation of accelerators and unconventional processors. Scientists, researchers, and developers working in HPC will need to build more flexibility and portability into the tools and applications that they develop. We hope that our work can serve as an aid to others who are seeking to instill these features into performance analysis methodology.

REFERENCES CITED

[1] D. Terpstra, H. Jagode, H. You, and J. Dongarra, “Collecting performance data with PAPI-C,” in *Tools for High Performance Computing 2009*. Springer, 2010, pp. 157–173.

[2] M. A. Heroux, P. Raghavan, and H. D. Simon, *Parallel processing for scientific computing*. SIAM, 2006.

[3] T. P. Straatsma, K. B. Antypas, and T. J. Williams, *Exascale scientific applications: Scalability and performance portability*. CRC Press, 2017.

[4] V. Eijkhout, *Introduction to high performance scientific computing*. Lulu. com, 2010.

[5] R. Robey and Y. Zamora, *Parallel and High Performance Computing*. Simon and Schuster, 2021.

[6] J. Jeffers and J. Reinders, *High performance parallelism pearls volume two: multicore and many-core programming approaches*. Morgan Kaufmann, 2015.

[7] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2019.

[8] J. Dean, D. Patterson, and C. Young, “A new golden age in computer architecture: Empowering the machine-learning revolution,” *IEEE Micro*, vol. 38, no. 2, pp. 21–29, 2018.

[9] W. Jalby, D. Kuck, A. D. Malony, M. Masella, A. Mazouz, and M. Popov, “The long and winding road toward efficient high-performance computing,” *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1985–2003, 2018.

[10] G. M. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” in *Proceedings of the April 18-20, 1967, spring joint computer conference*, 1967, pp. 483–485.

[11] A. C. De Melo, “The new linux’perf’tools,” in *Slides from Linux Kongress*, vol. 18, 2010.

[12] S. S. Shende and A. D. Malony, “The TAU parallel performance system,” *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

[13] J. C. Linford, S. S. Shende, and A. D. Malony, "TAU Commander: An intuitive interface for the TAU performance system," 11 2013.

[14] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "HPCToolkit: Tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010.

[15] X. Liu and J. Mellor-Crummey, "A data-centric profiler for parallel programs," in *High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for*. IEEE, 2013, pp. 1–12.

[16] D. Boehme, T. Gamblin, D. Beckingsale, P.-T. Bremer, A. Gimenez, M. LeGendre, O. Pearce, and M. Schulz, "Caliper: Performance introspection for HPC software stacks," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2016, p. 47.

[17] T. Röhl, J. Eitzinger, G. Hager, and G. Wellein, "Likwid monitoring stack: A flexible framework enabling job specific performance monitoring for the masses," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2017, pp. 781–784.

[18] V. M. Weaver and S. A. McKee, "Can hardware performance counters be trusted?" in *2008 IEEE International Symposium on Workload Characterization*. IEEE, 2008, pp. 141–150.

[19] D. Chen, N. Vachharajani, R. Hundt, S.-w. Liao, V. Ramasamy, P. Yuan, W. Chen, and W. Zheng, "Taming hardware event samples for FDO compilation," in *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. ACM, 2010, pp. 42–52.

[20] A. Nowak, A. Yasin, A. Mendelson, and W. Zwaenepoel, "Establishing a base of trust with performance counters for enterprise workloads," in *2015 {USENIX} Annual Technical Conference ({USENIX} {ATC} 15)*, 2015, pp. 541–548.

[21] V. M. Weaver, D. Terpstra, and S. Moore, "Non-determinism and overcount on modern hardware performance counter implementations," in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2013, pp. 215–224.

[22] O. R. LaMaire and W. W. White, "The contribution to performance of instruction set usage in system/370," in *Proceedings of 1986 ACM Fall joint computer conference*, 1986, pp. 665–674.

[23] E. Williams, C. T. Myers, and R. Koskela, "The characterization of two scientific workloads using the cray x-mp performance monitor," in *Conference on High Performance Networking and Computing: Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, vol. 12, no. 16, 1990, pp. 142–152.

[24] Z. Cvetanovic and D. Bhandarkar, "Characterization of alpha axp performance using tp and spec workloads," in *Proceedings of 21 International Symposium on Computer Architecture.* IEEE, 1994, pp. 60–70.

[25] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz, "Performance analysis using the mips r10000 performance counters," in *Supercomputing'96: Proceedings of the 1996 ACM/IEEE Conference on Supercomputing.* IEEE, 1996, pp. 16–16.

[26] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A performance counter architecture for computing accurate CPI components," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 5, pp. 175–184, 2006.

[27] ——, "A top-down approach to architecting CPI component performance counters," *IEEE micro*, vol. 27, no. 1, pp. 84–93, 2007.

[28] A. Nowak, D. Levinthal, and W. Zwaenepoel, "Hierarchical cycle accounting: A new method for application performance tuning," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS).* IEEE, 2015, pp. 112–123.

[29] A. Yasin, "A top-down method for performance analysis and counters architecture," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS).* IEEE, 2014, pp. 35–44.

[30] A. Yasin, A. Mendelson, and Y. Ben-Asher, "Tuning performance via metrics with expectations," *IEEE Computer Architecture Letters*, 2019.

[31] A. Yasin, J. Haj-Yahya, Y. Ben-Asher, and A. Mendelson, "A metric-guided method for discovering impactful features and architectural insights for Skylake-based processors," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 16, no. 4, pp. 1–25, 2019.

[32] F. Lebeau, "Top-down performance analysis webinar," https://www.brighttalk.com/webcast/17792/384060, accessed: 2020-05-03.

[33] M. Jarus and A. Oleksiak, "Top-down characterization approximation based on performance counters architecture for amd processors," *Simulation Modelling Practice and Theory*, vol. 68, pp. 146–162, 2016.

[34] S. Manakkadu and S. Dutta, "Bandwidth based performance optimization of multi-threaded applications," in *2014 Sixth International Symposium on Parallel Architectures, Algorithms and Programming.* IEEE, 2014, pp. 118–122.

[35] D. Molka, R. Schöne, D. Hackenberg, and W. E. Nagel, "Detecting memory-boundedness with hardware performance counters," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering.* ACM, 2017, pp. 27–38.

[36] D. Callahan, J. Cocke, and K. Kennedy, "Estimating interlock and improving balance for pipelined architectures," *Journal of Parallel and Distributed Computing*, vol. 5, no. 4, pp. 334–358, 1988.

[37] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[38] A. Ilic, F. Pratas, and L. Sousa, "Cache-aware Roofline model: Upgrading the loft," *IEEE Computer Architecture Letters*, vol. 13, no. 1, pp. 21–24, 2013.

[39] T. Koskela, Z. Matveev, C. Yang, A. Adedoyin, R. Belenov, P. Thierry, Z. Zhao, R. Gayatri, H. Shan, L. Oliker *et al.*, "A novel multi-level integrated roofline model approach for performance characterization," in *International Conference on High Performance Computing.* Springer, 2018, pp. 226–245.

[40] N. Denoyelle, A. Ilic, B. Goglin, L. Sousa, and E. Jeannot, "Automatic cache aware roofline model building and validation using topology detection," in *NESUS Third Action Workshop and Sixth Management Committee Meeting*, vol. 1, 2016.

[41] J. D. Suetterlein, J. Landwehr, A. Marquez, J. Manzano, and G. R. Gao, "Extending the Roofline model for asynchronous many-task runtimes," in *Cluster Computing (CLUSTER), 2016 IEEE International Conference on.* IEEE, 2016, pp. 493–496.

[42] D. Cardwell and F. Song, "An extended roofline model with communication-awareness for distributed-memory hpc systems," in *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, 2019, pp. 26–35.

[43] J. W. Choi, D. Bedard, R. Fowler, and R. Vuduc, "A Roofline model of energy," in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on.* IEEE, 2013, pp. 661–672.

[44] D. Marques, A. Ilic, Z. A. Matveev, and L. Sousa, “Application-driven cache-aware roofline model,” *Future Generation Computer Systems*, vol. 107, pp. 257–273, 2020.

[45] V. C. Cabezas and M. Püschel, “Extending the roofline model: Bottleneck analysis with microarchitectural constraints,” in *2014 IEEE International Symposium on Workload Characterization (IISWC).* IEEE, 2014, pp. 222–231.

[46] Y. J. Lo, S. Williams, B. Van Straalen, T. J. Ligocki, M. J. Cordery, N. J. Wright, M. W. Hall, and L. Oliker, “Roofline model toolkit: A practical tool for architectural and program analysis,” in *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems.* Springer, 2014, pp. 129–148.

[47] D. Doerfler, J. Deslippe, S. Williams, L. Oliker, B. Cook, T. Kurth, M. Lobet, T. Malas, J.-L. Vay, and H. Vincenti, “Applying the Roofline performance model to the Intel Xeon Phi Knights Landing processor,” in *International Conference on High Performance Computing.* Springer, 2016, pp. 339–353.

[48] B. Norris, W. Spear, and A. Malony, “Performance analysis of applications in the context of architectural rooflines,” in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, 2017, pp. 345–348.

[49] G. Ofenbeck, R. Steinmann, V. Caparros, D. G. Spampinato, and M. Püschel, “Applying the roofline model,” in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS).* IEEE, 2014, pp. 76–85.

[50] J. D. McCalpin, “Memory bandwidth and machine balance in current high performance computers,” *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.

[51] A. Hartono, B. Norris, and P. Sadayappan, “Annotation-based empirical performance tuning using orio,” in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on.* IEEE, 2009, pp. 1–11.

[52] C. Chen, J. Chame, and M. Hall, “Chill: A framework for composing high-level loop transformations,” Technical Report 08-897, U. of Southern California, Tech. Rep., 2008.

[53] B. J. Gravelle, W. D. Nystrom, D. Yokelson, and B. Norris, “Enabling cache-aware roofline analysis with portable hardware counter metrics,” in *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems.* Springer, 2021.

[54] B. Norris, A. Hartono, and W. Gropp, "Annotations for productivity and performance portability," in *Petascale Computing: Algorithms and Applications*, ser. Computational Science. Chapman & Hall / CRC Press, Taylor and Francis Group, 2007, pp. 443–462, also available as Preprint ANL/MCS-P1392-0107. [Online]. Available: http://www.mcs.anl.gov/uploads/cels/papers/P1392.pdf

[55] M. D. Hill and A. J. Smith, "Evaluating associativity in cpu caches," *IEEE Transactions on Computers*, vol. 38, no. 12, pp. 1612–1630, 1989.

[56] J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz, "XSBench - the development and verification of a performance abstraction for Monte Carlo reactor analysis," in *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*, Kyoto, 2014. [Online]. Available: https://www.mcs.anl.gov/papers/P5064-0114.pdf

[57] P. K. Romano, N. E. Horelik, B. R. Herman, A. G. Nelson, B. Forget, and K. Smith, "Openmc: A state-of-the-art monte carlo code for research and development," *Annals of Nuclear Energy*, vol. 82, pp. 90–97, 2015.

[58] A. Mallinson, D. A. Beckingsale, W. Gaudin, J. Herdman, J. Levesque, and S. A. Jarvis, "Cloverleaf: Preparing hydrodynamics codes for exascale," *The Cray User Group*, vol. 2013, 2013.

[59] "The PENNANT mini-app," https://github.com/lanl/PENNANT, accessed: 2018-06-25.

[60] K. J. Bowers, B. J. Albright, B. Bergen, L. Yin, K. J. Barker, and D. J. Kerbyson, "0.374 pflop/s trillion-particle kinetic modeling of laser plasma interaction on Roadrunner," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, ser. SC '08. IEEE Press, 2008.

[61] K. J. Bowers, B. Albright, L. Yin, B. Bergen, and T. Kwan, "Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulation," *Physics of Plasmas*, vol. 15, no. 5, p. 055703, 2008.

[62] K. J. Bowers, B. J. Albright, L. Yin, W. Daughton, V. Roytershteyn, B. Bergen, and T. Kwan, "Advances in petascale kinetic plasma simulation with VPIC and Roadrunner," in *Journal of Physics: Conference Series*, vol. 180, no. 1. IOP Publishing, 2009, p. 012055.

[63] "Vector particle-in-cell (VPIC) project," https://github.com/lanl/vpic, accessed: 2019-11-05.

[64] "The 17.4 theory manual (pagosa theory manual. la-ur-20-29881)."

[65] V. Graziano, D. Nystrom, H. Pritchard, B. Smith, and B. Gravelle, “Optimizing a 3d multi-physics continuum mechanics code for the hpe apollo 80 system,” *Cray User Group (CUG)*, 2021.

[66] J. Levesque and A. Vose, *Programming for Hybrid Multi/Manycore MPP Systems.* Chapman and Hall/CRC, 2017.