12-2025

Automated Classification and Verification of Performance Counters

Daniel P. Barry

To the Graduate Council:

I am submitting herewith a dissertation written by Daniel P. Barry entitled "Automated Classification and Verification of Performance Counters." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Data Science and Engineering.

Jack J. Dongarra, Major Professor

We have read this dissertation and recommend its acceptance:

Heike Jagode, Anthony Danalis, Russell L. Zaretzki

Accepted for the Council:

Dr. Marieke Van Puymbroeck

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

Automated Classification and Verification of Performance Counters

A Dissertation Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Daniel Barry

December 2025

Acknowledgments

I would like to thank...

Dr. Anthony Danalis and Dr. Heike Jagode for their dedicated mentorship and advice. Their continual guidance has been instrumental to my PhD journey.

This research was supported in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration; and by the National Science Foundation under awards No. 1450429 "PAPI-EX" and No. 1900888 "ANACIN-X." This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

This research used resources of the Argonne Leadership Computing Facility, a U.S. Department of Energy (DOE) Office of Science user facility at Argonne National Laboratory and is based on research supported by the U.S. DOE Office of Science-Advanced Scientific Computing Research Program, under Contract No. DE-AC02-06CH11357. This work was done on a pre-production supercomputer with early versions of the Aurora software development kit.

This material is based on work supported in part by the National Science Foundation under award number 2311707. A portion of this research was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research (ASCR), Next Generation Scientific Software Technologies (NGSST) Program.

In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of the University of Tennessee's products or services. Internal or personal use of this material is permitted. If interested in

Abstract

Efficient execution of scientific applications on high-performance computing (HPC) systems depends heavily on effective performance analysis. Performance analysis refers to the process of evaluating how well an application performs on an HPC system, including its interaction with underlying hardware components, to understand how well it operates and identify potential areas for improvement. This typically involves the measurement of various performance metrics, such as speed, efficiency, resource utilization, and responsiveness. The insights gained from performance analysis allow the user of an HPC system to optimize hardware usage, pinpoint inefficiencies and bottlenecks, and ensure scalability. Many of the most informative performance metrics can only be obtained by monitoring hardware events (*i.e.*, low-level operations tracked by the hardware itself). Without them, the only available performance metric is execution time.

However, the sheer volume of hardware events in modern HPC systems is overwhelming, making them difficult for users to comprehend and use effectively. The research in this dissertation has focused on mitigating this problem by developing an approach that quantitatively characterizes hardware events, automatically classifies them, and automatically derives meaningful performance metrics from them. To better understand the system behaviors captured by hardware events, this dissertation presents benchmarks consisting of well-defined operations that stress different hardware attributes in isolation. In addition, it elucidates an automated mathematical analysis to identify key hardware events and define useful performance metrics using them. Lastly, it establishes strategies for benchmarking the hardware shared among processor cores and identifying key inter-core events.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Disclosure

This chapter uses content from one of my published papers [16] (© 2024 IEEE as appropriate):

- Barry, D., Danalis, A., and Dongarra, J. (2024). **"Automated Data Analysis for Defining Performance Metrics from Raw Hardware Events."** In *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 716-725, https://doi.org/10.1109/IPDPSW63119.2024.00134.

I am the primary author of this paper. Coauthors for the published paper include Anthony Danalis and Jack Dongarra. Reused coauthor contributions are limited the idea of rounding (shown in Section 4.6), textual improvements, and high-level guidance.

1.2 Motivation

Some of the most emphasized performance metrics of HPC systems are time-based. Execution time and speedup are common metrics and were even utilized as key performance parameters (KPPs) for software applications in the Exascale Computing Project (ECP) [13]. However, execution time alone does not indicate how efficiently an application is performing.

For example, a recent Gordon Bell prize-winning paper [32] presented an innovative methodology to solve the inverse density functional theory (invDFT) problem on the Frontier supercomputer. Prior to these authors' efforts, the execution time for this simulation was seven days; however, they decreased the execution time to three hours. This information about execution time by itself does not explain how well the simulation is performing relative to the hardware capability, nor does it indicate how well the simulation can possibly perform. However, the authors also provided the performance metric of floating-point operations per second (FLOP/s) for this simulation, which is approximately 660 peta-FLOP/s (10^{15} FLOP/s, or PFLOP/s). According to the Top 500 HPC system rankings [102], Frontier is capable of 1,353 PFLOP/s for the HPL benchmark and 14 PFLOP/s for the HPCG benchmark.[1] With this information about the FLOP/s performance metric, it is possible to analyze the simulation's performance in terms of the maximum-attainable FLOP/s. Namely, this simulation performs at approximately 43% of the HPL score. Furthermore, since the simulation outperforms the HPCG benchmark by over an order of magnitude, we know that it is not greatly inhibited by memory bandwidth (memory transferred per second) or latency (responsiveness of memory hardware). This exemplifies how additional performance metrics, such as FLOP/s, offer richer and more nuanced insight into application performance. Due to resource limitations, the simulation from [32] cannot be recreated in this dissertation. However, the following example demonstrates another case where performance metrics provide significantly more insight than execution time alone.

The matrix multiplication (GEMM) operation is a commonplace computational benchmark. The reference implementation of this operation utilizes three nested for-loops, as shown in the kernel (code that performs a specific operation) in Listing 1.1. On a single Frontier CPU (AMD Zen3 architecture) with $N = 2048$, the first loop nest (shown in Listing 1.1) executes in 73.4 seconds. With this information alone, one does not know whether the kernel is performing as well as possible on the hardware. To illustrate how low-level memory access patterns impact execution time, we examine two variants of the GEMM loop nest.

[1] These benchmarks are discussed in more detail in Section 2.4.

```
1 for ( i = 0; i < N; ++i )
2   for ( j = 0; j < N; ++j )
3     for ( k = 0; k < N; ++k )
4       C[i*N+j] += A[i*N+k] * B[k*N+j];
```

Listing 1.1: Loop Nest 1.

```
1 for ( i = 0; i < N; ++i )
2   for ( k = 0; k < N; ++k )
3     for ( j = 0; j < N; ++j )
4       C[i*N+j] += A[i*N+k] * B[k*N+j];
```

Listing 1.2: Loop Nest 2.

To better understand the observed performance, we analyze Level-1 data cache behavior—specifically, accesses and misses—during the GEMM execution. The results are shown in the first row of Table 1.1. The number of cache misses per cache access gives the "Miss Rate," which is 0.48 (or 48%) for Loop Nest 1. In other words, on average, every other data cache access results in a miss. This indicates that the data is not accessed in an efficient manner.

To address this, we can change the data access pattern by swapping the inner two loops (j and k) in Listing 1.1. The resulting code is shown in Listing 1.2. Measuring the cache misses and accesses for Loop Nest 2, the second row in Table 1.1 shows approximately a tenth of the cache misses of Loop Nest 1, even though the total number of accesses is roughly the same. Examining only execution time, we would not have any of the low-level information from the hardware, which enabled us to optimize the GEMM that now runs in a mere 4.1 seconds. However, there are some challenges with using performance metrics:

- A single performance metric may require multiple hardware events, and the specific hardware events that need to be scaled and combined are difficult to determine manually.
- The hardware events needed to form the same metric on separate architectures are often vastly different.

For example, the metric of L1 data cache misses on the AMD Zen3 is the sum of the following hardware events:

- `REQUESTS_TO_L2_GROUP1:RD_BLK_L`
- `REQUESTS_TO_L2_GROUP1:RD_BLK_X`
- `REQUESTS_TO_L2_GROUP1:LS_RD_BLK_C_S`
- `REQUESTS_TO_L2_GROUP1:CHANGE_TO_X`

and L1 data cache accesses is the sum of the hardware events:

- `LS_DISPATCH:LD_ST_DISPATCH`
- `LS_DISPATCH:STORE_DISPATCH`
- `LS_DISPATCH:LD_DISPATCH`

Table 1.1: Avg. Execution Time and Performance Metrics Over Ten Runs

Kernel	**Exec. Time (sec)**	**Cache Misses**	**Cache Accesses**	**Miss Rate (Misses/Access)**
Loop Nest 1	73.4	16,721,129,406	34,745,306,481	0.481
Loop Nest 2	4.1	1,463,854,734	34,454,430,321	0.042

In stark contrast, on the Intel Sapphire Rapids architecture, L1 data cache misses is the lone hardware event `L1D:REPLACEMENT`; furthermore, there do not even exist hardware events to measure L1 data cache accesses.

These issues persist for other prevalent performance metrics of HPC applications, such as FLOP/s and bandwidth. FLOP/s is perhaps one of the most well-known and ubiquitous metrics, being the criterion for ranking the Top 500 HPC systems [102]. More broadly, FLOP/s is a common metric used to gauge the efficiency of basic linear algebra subprogram (BLAS) routines in numerical linear algebra software packages [43, 12, 8]. Bandwidth is a widespread metric of memory hierarchy performance [75, 105, 13]. Hardware events are required to accurately measure both FLOPs/s and bandwidth. Therefore, unmasking which hardware events (and combinations thereof) define these metrics is of interest to the greater HPC community.

Furthermore, modern HPC systems [93, 105, 13, 10] are becoming more and more heterogeneous in their hardware constitution—*e.g.*, deeper and more nuanced memory hierarchies, custom network infrastructures, graphics processing units (GPUs), and accelerated processing units (APUs) that combine CPU and GPU hardware within the same die. As such, they contain on the order of hundreds of thousands of hardware events related to the increasingly diverse array of hardware components. While these hardware events are documented to some extent in vendor-published technical documents [57, 67, 4, 3, 5, 6], they are not always described thoroughly or accurately. This overall lack of clarity makes it challenging for users to comprehend the specific high-level performance metrics—such as bidirectional memory bandwidth—that hardware events actually represent. Even worse, some architectures completely lack hardware events that map to some useful performance metrics. For instance, there are over 427,000 hardware events on a single node of the Summit supercomputer (which contains two IBM POWER9 CPUs and six NVIDIA Tesla V100 GPUs [86]), but not a single one of those hardware events measures all floating-point operations (FLOPs). The contributions of this dissertation are aimed at alleviating these limitations to improve the landscape of performance analysis.

1.3 Contributions

The following items describe the contributions accomplished in this work, ***Automated Classification and Verification of Performance Counters***:

1. Produce a suite of microbenchmarks, which consists of well-defined operations that stress different hardware attributes (floating-point units, branching units, memory hierarchy, *etc.*) in isolation from one another. These benchmarks allow the user to collect hardware event measurements to quantitatively characterize hardware events to discover the true semantic concepts (*e.g.*, FLOPs, memory bandwidth, *etc.*) they measure.

2. *Automatically* classify which hardware events belong to each of the aforementioned hardware attributes. By quantifying measurement variation among repeated benchmark executions, hardware events that are irrelevant to a particular hardware attribute are identified and disregarded.

3. *Automatically* derive meaningful performance metrics using available hardware events. Since the set of all available hardware events contains semantically redundant information (*e.g.*, multiple counters reflecting the same hardware behavior), a unique hardware event combination is not trivially attainable. Therefore, the research in this dissertation establishes data processing methods to filter the data that resulted from the previous contributions.

4. Develop techniques to accurately decode and verify Uncore events (also referred to as Northbridge or Nest events, depending on the vendor).[2] This poses a challenge because, as corresponding to resources shared among the cores, these hardware events are particularly prone to noise incurred by system service tasks running in the background.

[2]These are the Intel-, AMD-, and IBM-specific terms referring to the CPU hardware residing outside of the domain of a specific compute core. The remainder of this document uses the architecture-agnostic term "inter-core events" to refer to events belonging to this category of hardware.

1.4 Dissertation Outline

The work for each chapter of this dissertation is summarized below.

- Chapter 2 discusses the notions of hardware events, performance tools, the taxonomy of benchmarking, and various mathematical tools that serve as foundational knowledge for the contributions in this dissertation. It also provides a literature review of related work in commonplace HPC benchmarks, hardware event noise, matrix decompositions, and regression methods.

- Chapter 3 explains the design and implementation of each of the categories of hardware benchmarks in the Counter Analysis Toolkit (CAT), which systematically collect and characterize hardware events.

- Chapter 4 discusses the automated data analysis methodology to address noise from hardware event measurements, leverage the structure of CAT benchmarks to identify key hardware events, and automatically define performance metrics. This chapter provides results for both a CPU architecture and a GPU architecture, both of which are used in modern exascale supercomputers.

- Chapter 5 introduces strategies for monitoring inter-core events in multicore environments and their significance for understanding the execution profile of HPC applications. In addition, this chapter discusses techniques for amortizing noise present in inter-core event measurements.

- Chapter 6 provides a summary of the results from the research efforts in this dissertation. This chapter also describes some prospective research directions following the contributions of this dissertation.

Chapter 2

Background and Related Work

2.1 Disclosure

This chapter uses content from one of my published papers [16] (© 2024 IEEE as appropriate):

- Barry, D., Danalis, A., and Dongarra, J. (2024). **"Automated Data Analysis for Defining Performance Metrics from Raw Hardware Events."** In *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 716-725, https://doi.org/10.1109/IPDPSW63119.2024.00134.

I am the primary author of this paper. Coauthors for the published paper include Anthony Danalis and Jack Dongarra. Reused coauthor contributions are limited the idea of rounding (shown in Section 4.6), textual improvements, and high-level guidance.

2.2 Overview

This chapter provides an overview of hardware events, performance metrics, event-monitoring tools, categories of benchmarks and benchmarking strategies for HPC systems, and background information of the relevant data analysis techniques. In addition, it describes the limitations present in the state-of-the-art performance-monitoring technology which the efforts in the following chapters mitigate.

2.3 Hardware Events and Performance Tools

Hardware events are low-level operations executed by computational hardware. Special-purpose registers, or *counters*, are programmed to monitor hardware events. Counters reside inside of performance-monitoring units (PMUs), which are found in an extensive, heterogeneous array of HPC hardware components, such as CPUs of various vendors (Intel, AMD, NVIDIA, and IBM—to name a few) GPUs (Intel, AMD, and NVIDIA), interconnects (5D Torus, Aries, Gemini, Infiniband, NVLink, Slingshot, *etc.*), input-output (I/O), power-monitoring components (lm_sensors, NVML, Powercap, and RAPL), *etc.* Access to counters is provided by open-source and vendor-provided application programming interfaces (APIs), such as libpfm4 [41], Performance Co-Pilot (PCP) [92], NVIDIA CUDA Profiling Tools Interface (CUPTI) [84], NVIDIA Management Library (NVML) [85], AMD Radeon Open Compute (ROCm) [7], Intel oneAPI Level Zero [59], among others. CPUs contain various subsystems, such as floating-point units, integer units, branching units, cache hierarchy, *etc.*

Various software tools also provide access to counters: the Performance Application Programming Interface (**PAPI**) [101], **perf** [80], Intel **VTune** [108], and **LIKWID** [103]. **PAPI** provides both command-line utilities and an API. **PAPI** allows users to monitor hardware events from all of the aforementioned hardware components through a unified interface *via* the various open-source APIs. **perf** is a lightweight profiling tool, or *profiler*, built into the Linux kernel. Profilers collect hardware event occurrences throughout the execution of applications. Intel VTune is a profiler that is compatible with Intel hardware. LIKWID is a set of profiling utilities, compatible with primarily CPU architectures, in addition to some GPU architectures. Middleware libraries (*e.g.,* PAPI) provide presets, or definitions for performance metrics, for multiple CPU and GPU architectures using natively available hardware events. Other tools—such as TAU [97], Score-P [96], Vampir [24], Caliper [22], Scalasca [44], and APEX [55]—depend on these libraries to access hardware events.

2.3.1 Complexity of Validating Hardware Events

Hardware event counts often deviate from theoretical expectations due to undocumented micro-architectural behaviors, event aliasing, or non-determinism in execution. [107] While hardware events are documented to some extent in vendors' technical documents [3], they are not always described thoroughly. One reason hardware events do not occur in the expected amounts is due to micro-architectural nuances in hardware, such as writes to memory incurring reads (as observed in Chapter 5). Another reason is that some hardware events are counted twice (*e.g.*, `FP_ARITH_INST_RETIRED` in the Intel Ice Lake architecture). Program variation from run to run is another issue with measuring hardware events. [107, 94] Furthermore, in many CPU architectures, there exist multiple hardware events which measure the same concept, or overlapping concepts. This issue is compounded by the massive amounts of hardware events in modern HPC systems. Even if the above issues were resolvable *via* manual validation, the process would be unfeasible for each and every hardware event that is present.

2.4 Benchmarking Strategies and Taxonomy

As previously stated, hardware events are low-level operations performed by hardware. This means that the hardware event for FLOPs, for instance, should occur an exact number of times during the execution of a FLOPs-heavy kernel, such as a GEMM. There are kernels that exercise other hardware subsystems and the corresponding hardware events. Therefore, the general strategy for decoding what events measure is a two-step process. First, the event is monitored during the execution of a microbenchmark that performs a known, isolated operation at the hardware level. Second, the hardware event's occurrence pattern is compared to the pattern we expect from a performance metric of interest. To address the first step, the main categories of benchmarks are delineated:

- **Low-Level**

  Benchmarks belonging to this category measure low-level hardware characteristics,

such as peak sustainable bandwidth. Examples of this type of benchmark are *STREAM* [75], *CacheBench* [81], and *lmbench* [78].

- **Kernel**

  Benchmarks in this category measure the performance characteristics of software operations. Examples of this type of benchmark are those contained in the *Polybench* suite [90], *Numerical Aerodynamic Simulation* (NAS) parallel benchmarks [15], *High-Performance LINPACK* (HPL) [38], *HPL Mixed Precision Benchmark* (HPL-MxP) [36], *High-Performance Conjugate Gradient* [35], and *SPEC CPU2006* suite. [53, 99]

- **Full-Application**

  These benchmarks gauge the efficiency of full-scale jobs on HPC systems for realistic, scientific use cases. Examples of this type of benchmark are some of those contained in the *Collaboration of Oak Ridge, Argonne, and Livermore 2* (CORAL-2) benchmarks [104, 68] and the United States Department of Defense *Technology Insertion* benchmarks. [23]

The efforts in this dissertation emphasize the **Low-Level** and **Kernel** categories. This is because events are low-level hardware operations, hence they are activated in isolation by low-level microkernels (**Low-Level**) and certain BLAS operations (**Kernel**). More complex benchmarks would utilize multiple hardware subsystems, which would in turn trigger multiple types of hardware events simultaneously. This would complicate the task of determining how specific computational patterns in the code map to the observed hardware event activity. Section 2.4.1 provides an overview of commonplace benchmarking kernels and techniques.

2.4.1 Common Benchmarking Strategies

The design of the CAT benchmarks is based on the techniques introduced in this section, with various adaptations to induce nuanced hardware behavior (*e.g.*, prefetching, varieties of FLOPs such as single- and double-precision, *etc.*). The implementation details of the CAT benchmarks are discussed in detail in Chapter 3.

BLAS microkernels—such as the matrix-matrix multiplication (GEMM), matrix-vector multiplication (GEMV), and vector-vector product (DOT)—are commonplace benchmarks [82, 51, 61, 52, 1, 25] due to (i) their prevalence in scientific applications and other BLAS kernels and (ii) their aggregate range of arithmetic intensity (FLOPs per byte of memory accessed).

STREAM [77, 9, 14, 51, 105] is a well-known benchmark that determines the maximum sustained bandwidth between memory and CPU cores. STREAM defines memory buffers (a, b, and c) and operates on them *via* the following four kernels:

- COPY: $a[i] = b[i]$
- SCALE: $a[i] = \alpha * b[i]$
- SUM: $a[i] = b[i] + c[i]$
- TRIAD: $a[i] = b[i] + \alpha * c[i]$

BabelStream (GPU-STREAM) [33] is a similar benchmark that gauges the bandwidth between memory and GPU cores.

Pointer chasing [79, 109, 28, 47, 78, 81, 95, 100] is another strategy for benchmarking the memory subsystem. This technique entails defining a buffer (pointer chain) such that each element contains the address (pointer) to another element. As opposed to the STREAM kernels, which access each memory element sequentially, the pointer chain is traversed by accessing the element that the current element points to. This is useful for benchmarking the memory hierarchy of both CPUs and GPUs.

2.4.2 Considerations for Inter-core Events

Certain hardware resources are shared among all CPU processor cores. As such, the events belonging to these shared resources (inter-core events) measure the aggregate activity from all cores. This spawns two issues. The first issue stems from the security risk that one core can monitor another core's activity *via* inter-core events. To mitigate the risk, applications typically must run with elevated privileges in order to monitor inter-core events. In the case of the Sierra and Summit supercomputers, elevated privileges were not necessary to access the inter-core events. [42] This was possible *via* a performance-monitoring daemon, which user applications would query. However, this introduces an indirection layer when monitoring inter-core events. This is a potential source of overhead, which could impact the accuracy of event measurements.

The second issue is that the hardware activity induced by processes running on other cores perturb inter-core event measurements. Since there are system services running in the background, even if on dedicated cores [86], they will pollute the inter-core event measurements. Chapter 5 presents experimental benchmark apparatuses that are designed to address these two issues of monitoring inter-core memory traffic in unprivileged HPC environments.

2.5 Pertinent Linear Algebra Methods

For the data analysis discussed in Chapter 4, the output of the microbenchmarks is organized into a matrix and utilize various linear algebra methods to extract information about the performance metrics of interest from the hardware event data in the matrix. This is because performance metrics are often linear combinations of hardware events. Due to the various reasons explained in Section 4.3, the matrix in this linear system is singular. To address this, linear algebra methods are useful in finding the linearly independent columns of a matrix to make the system solvable. This section provides an overview of the linear algebra used in this dissertation.

QR decomposition is an orthogonal matrix factorization, which given a matrix A calculates an orthogonal matrix Q and an upper triangular matrix R, such that $A = QR$. This factorization is used to solve the rank-deficient least squares problem. [98] *Rank deficient* means that there are fewer linearly independent columns of A than the total number of columns in A. The linear least squares gives the "best-fit" solution to the problem $Ax = b$ such that $||Ax - b||_2$ is minimized. The Q and R factors can be used to solve inconsistent systems in the least-squares sense. [64] *Inconsistent* means that $Ax = b$ cannot be solved directly because b is not in A's column space.

The matrix A (composed of event measurements) can be rank deficient or inconsistent with b. If a matrix is rank deficient, then the QR decomposition is not unique [20], and the number of non-zero elements on the R factor's diagonal is not necessarily equal to $rank(A)$. [34] This is a problem because the columns in R which have non-zero values on the diagonal correspond to the linearly independent columns of A, which is what we

seek. This issue is resolved with rank-revealing QR decompositions (RRQR) [26, 29, 19, 48, 27, 54, 46], which is a type of QR with column pivoting (QRCP), also referred to as *GEQP3*. [39] Similar to QR, RRQR produces Q and R matrices; but unlike QR, it also produces a permutation matrix (Π) resulting from the column pivoting, such that $A\Pi = QR$. The reference QRCP [39] chooses the column (in the trailing sub-matrix of a given iteration) with largest 2-norm to be the pivot. In other words, QRCP chooses the largest columns at each step, while enforcing orthogonality among them. This selection criterion does not lead to the optimal subset of columns. Section 4.6 describes the modified pivoting scheme devised in order to obtain the most useful columns.

Singular value decomposition (SVD) is another orthogonal matrix factorization ($A = U\Sigma V^T$). [45] While the SVD offers a robust method for revealing the rank of a matrix, it does not provide the subset of linearly independent columns. Principal Component Analysis (PCA) is a linear dimensionality reduction method that uses SVD to generate a set of vectors that span the largest dimensions of A's column space. [62] However, the generated vectors are not a subset of linearly independent columns of A. Rather, they are linear combinations of the columns of A which maximally span the column space of A in as few dimensions as possible. Because PCA and SVD produce combinations of columns rather than selecting actual columns from A, they are unsuitable when the interpretation of individual hardware events is required.

An alternative to using RRQR is interpolative decomposition (ID) [66, 106, 74], which produces a matrix C (a subset of A's columns) and a matrix V, such that $A \approx CV$. The CUR factorization is similar to ID. [40, 21, 70] CUR produces a matrix C (a subset of A's columns), a matrix R (a subset of A's rows), and a matrix U, such that $A \approx CUR$. Even though these methods produce a linearly independent subset of A's columns, the subset is chosen with priority given to columns that either (i) have the largest 2-norm or (ii) contribute the most to the space spanned by the top singular vectors. [106, 70] Although these methods are structure-preserving, they have the same issue as QRCP: selecting a sub-optimal subset of A's columns.

2.6 Related Works

The *Counter Inspection Toolkit* [30] was a foundational, preliminary work on top of which Chapter 3 is built. It presented portable microbenchmarks, which stressed the memory hierarchy and branching units in CPU cores. It did not include benchmarks for the other hardware attributes elucidated in Chapter 3, such as floating-point units and instruction caches. This work did not include multi-threaded memory hierarchy benchmarks, which is significant for modern many-core architectures. Furthermore, this work did not automatically classify or combine events.

Another related work is *nanoBench* [2], which is a microbenchmarking infrastructure enabling its users to monitor events. This tool takes x86 (Intel and AMD architecture-related) assembly code segments as input, and it outputs counts of user-specified hardware events. This work incurs very low overhead by minimizing extraneous instructions and event occurrences. However, since nanoBench only accepts x86 as the input assembly, it is limited to x86 CPU architectures. This tool is limited because it provides only the infrastructure to monitor user-defined microbenchmark kernels; it does not provide the actual microbenchmarks for various hardware categories.

Benchpark [88] is a continuous benchmarking infrastructure. This work provides a framework for portable benchmarking across various HPC systems. The specific benchmarks used in this work are pre-existing benchmark codes and kernels, such as HPL, saxpy, HPCG, STREAM, among others. [69] These are useful benchmarking codes, however, they vary from the Low-Level to the Full-Application benchmarking categories. For the purposes of hardware event analysis, the efforts of this dissertation focus strictly on the Low-Level and Kernel categories.

BlackjackBench [31] generates a latency profile for the memory hierarchy using a pointer-chasing microbenchmark similar to that included in the Counter Inspection Toolkit. It then employs quality threshold clustering to automatically determine the size of each cache level. The pointer-chasing microbenchmark, however, does not execute across multiple cores.

There has been previous work which discovered such architectural details as cache line sizes, associativity, page sizes, and number of TLB entries using cache hardware events. [37]

This work is limited in scope in that it focuses strictly on the memory hierarchy. Additionally, this work infers memory hardware details by taking measurements of events of which the semantic meanings are known *a priori*; whereas, the efforts in this dissertation determine the semantic meaning of events.

Ritter *et al.* [94] provides an overview of sources of noise in performance metric measurements. This work operates on the known classification of events (by hardware attribute—*e.g.,* floating-point or branch units) *a priori*. However, one of the key contributions of this dissertation determines the classification of events. One of the authors' contributions is a noise generator to inject synthetic memory, network, and I/O noise on top of the already-present system noise. While this does show which events are resilient to such noise, this feature is counterproductive for the purpose of identifying hardware events, which needs to restrict run-to-run variation for events.

Some work has been done in developing models using linear regression with hardware events. [65] This model predicts the execution time of the LULESH benchmark. However, this work is predicated on knowing the semantic meaning of hardware events, such as Instructions and L2 Cache Misses. This dissertation resolves the semantic meaning of hardware events.

Previous efforts [72, 71, 89] have employed PCA to find the set of least-correlated hardware events for performance modelling. The approach in [71] entails signatures produced using randomly selected hardware events; whereas, the efforts in this dissertation is designed to examine all available hardware events. Since the set of hardware event data is collected from logging multiple applications, it does not necessarily yield data for all hardware events if the applications do not exhaustively exercise the hardware subsystems. Additionally, some hardware events could correlate specifically to the applications in use, leading to overfit. That is, one could not generalize performance modelling using the hardware events chosen in these studies due to their application-specific bias. As described in Section 2.5, PCA does not yield a subset of the events, but QRCP—as adapted in Chapter 4—does.

Chapter 3

Benchmarks to Probe Hardware Events

3.1 Disclosure

This chapter uses content from one of my published papers [17] (© 2021 Springer as appropriate):

- Barry, D., Danalis, A., and Jagode, H. (2021). **"Effortless Monitoring of Arithmetic Intensity with PAPI's Counter Analysis Toolkit."** In *Tools for High Performance Computing 2018 / 2019*, pp. 195-218, Cham. Springer International Publishing, https://doi.org/10.1007/978-3-030-66057-4_11.

I am the primary author of this paper. Coauthors for the published paper include Anthony Danalis and Heike Jagode. Reused coauthor contributions are limited to aid in the development of the branching and instruction cache benchmarks, the idea of problem repetitions (shown in Section 3.5.1), textual improvements, and high-level guidance.

3.2 Introduction

Performance metrics in different CPU architectures can be monitored by reading the occurrences of various hardware events. However, from architecture to architecture, it

becomes more and more unclear what specific hardware events actually measure. This chapter presents a suite of microbenchmarks (Counter Analysis Toolkit), which perform well-defined operations that stress different hardware attributes (floating-point units, branching units, memory hierarchy, *etc.*) in isolation from one another. These benchmarks allow the user to collect hardware event measurements to quantitatively characterize hardware events to discover the true semantic concepts they measure. In the era of exascale computing, performance metrics such as memory traffic and arithmetic intensity are increasingly important for codes that heavily utilize numerical kernels. The efforts in this chapter leverage the capabilities of the Counter Analysis Toolkit to identify the hardware events for reading and writing memory traffic in addition to floating-point operations, which are the necessary hardware events to analyze arithmetic intensity.

Most of the major tools that high-performance computing (HPC) application developers use to conduct low-level performance analysis and tuning of their applications typically rely on hardware events to monitor hardware-related activities. The kind of available hardware events is highly dependent on the hardware; even across the CPUs of a single vendor, each CPU generation has its own implementation. The PAPI performance-monitoring library provides a clear, portable interface to the hardware events available on all modern CPUs, as well as GPUs, networks, and I/O systems [101, 73, 76]. Additionally, PAPI supports transparent power monitoring capabilities for various platforms, including GPUs (AMD, NVIDIA) and Intel Xeon Phi [49], enabling PAPI users to monitor power in addition to traditional hardware event data, without modifying their applications or learning a new set of library and instrumentation primitives.

We have witnessed rapid changes and increased complexity in processor and system design, which combines multi-core CPUs and accelerators, shared and distributed memory, PCI-express and other interconnects. These systems require a continuous series of updates and enhancements to PAPI with richer and more capable methods needed to accommodate these new innovations. One such example is the PAPI Performance Co-Pilot (PCP) component, which is discussed in this chapter. Extending PAPI to monitor performance-critical resources that are shared by the cores of multi-core and hybrid processors—including on-chip communication networks, memory hierarchy, I/O interfaces, and power management

logic—will enable tuning for more efficient use of these resources. Failure to manage the usage and, more importantly, contention for these shared hardware resources has already become a major drag on overall application performance. Furthermore, we discuss one of PAPI's features: the Counter Analysis Toolkit (CAT), which is designed to improve the understanding of hardware events, including inter-core events.

We aim to define and verify accurate mappings between particular high-level performance metrics and the underlying low-level hardware events. This extension of PAPI engages novel expertise in low-level and kernel benchmarks for the explicit purpose of collecting meaningful performance data of shared hardware resources.

In this chapter, we outline the PAPI Counter Analysis Toolkit, describe its objective, and then focus on the microkernels that are used to measure and correlate different hardware events needed to define arithmetic intensity on the Intel Broadwell, Intel Skylake, and IBM POWER9 architectures.

3.3 Counter Analysis Toolkit

Hardware events are often appealing to application developers who are interested in understanding and improving the performance of their code. However, in modern architectures it is not uncommon to encounter hardware events whose names and descriptions can mislead users about the meaning of the event. Common misunderstandings can arise due to speculations inside modern CPUs, such as branch prediction and prefetching in the memory hierarchy, or noise in the measurements due to overheads and coarse granularities of measurements when it comes to resources that are shared among the compute cores (*e.g.,* off-chip caches and main memory).

Earlier work [30] explored the use of benchmarks that employ techniques such as pointer chasing [28, 31, 47, 78, 81, 95, 100] to stress the memory hierarchy as well as microbenchmarks with different branching behaviors to test different branch-related hardware events. CAT, which was released with PAPI version 6.0, has built upon these earlier findings by significantly expanding the kinds of tests performed by the microbenchmarks, as well as the parameter

space that is being explored. Also, we continue making our latest benchmarks as well as updates to the basic driver code publicly available through the PAPI project's Git repository.

CAT contains benchmarks for testing four different aspects of CPUs: data caches, instruction caches, branching units, and floating-point operations (FLOPs). CAT also has floating-point benchmarks that encompass GPUs. The microbenchmarks themselves are parameterized and, thus, their behavior can be modified by expert users who desire to focus on particular details of an architecture. The driver uses specific combinations of parameters that we have determined appropriate for revealing important differences between different hardware events. More details on the actual tests are discussed in the following sections.

3.3.1 Data Cache Tests

Figure 3.1 shows a plot of the data generated when the data cache read benchmark is executed. As shown in the figure, there are four regions that correspond to four different parameter sets. In all four regions, the access pattern is random. This choice affects the effectiveness of prefetching, since random jumps are unlikely to be predicted, but sequential accesses are perfectly predictable. The access stride is also varied among regions so that it either matches the size of a cache line on this architecture (64 bytes) or the size of two cache lines (128 bytes). This choice affects the effectiveness of "next-line prefetching," which is common in modern architectures. Another parameter that varies among the four regions is the size of the contiguous block of memory (pages per block, PPB) in which the pointer chaining happens. In effect, this defines the size of the working set of the benchmark, since all the elements of a block will be accessed before the elements of the next block start being accessed. We vary this parameter because in many modern architectures prefetching is automatically disabled as soon as the working set becomes too large.

The X-axis of the graph corresponds to the measurements performed by the benchmark. For each combination of parameters, the code performs a variable number of measurements specified by the user, and within each set of measurements, the X-axis corresponds to the size of the buffer that the benchmark uses. To improve the readability of Figure 3.1, at the bottom of the graph, we have marked the measurement indices within each region that correspond to the sizes of the three caches and main memory (L1, L2, L3, M) of the testbed we used (Intel

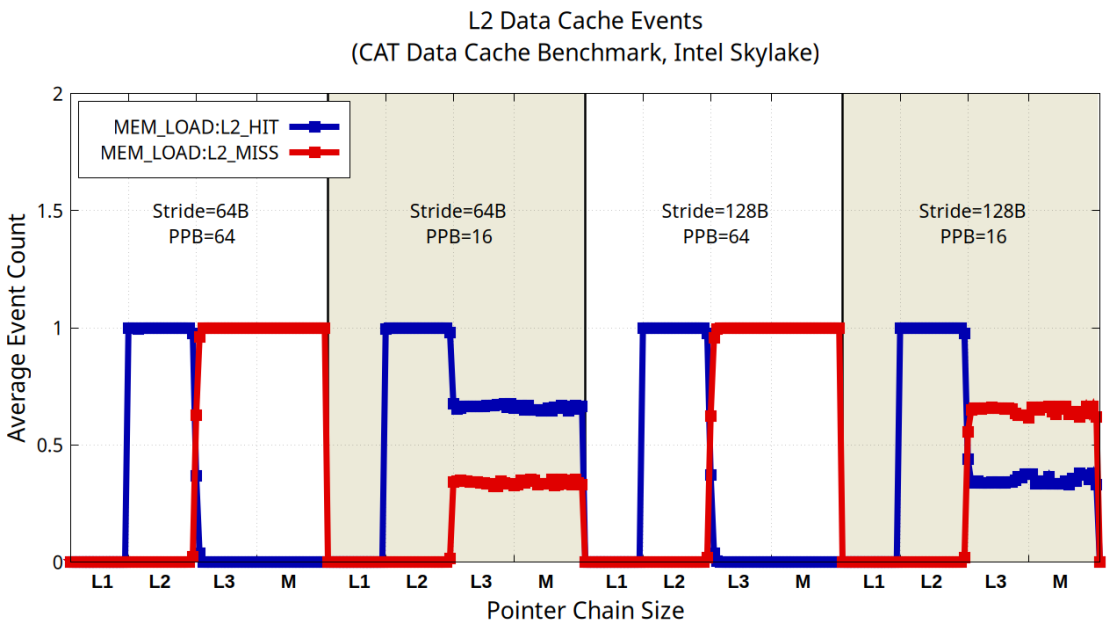


Figure 3.1: L2 Data Cache Hardware Events.

Skylake). For each measurement, the benchmark executes a memory traversal defined by the parameters of the region (*e.g.*, a random pointer chase with a large stride). To amortize the effect of cold cache misses (also known as *compulsory misses*), the benchmark traverses the test buffer in a loop such that the number of memory accesses for each measurement exceeds the size of the buffer by a large factor. As a result, cold cache misses do not have a measurable effect in our results.

The blue curve depicts the number of hits in the L2 cache per memory access (hit rate). In each of the four regions, the L2 hit rate is zero when the buffer size is smaller than the L1 cache (since all accesses are served by the L1 cache). When the buffer is larger than the L1 but smaller than the L2 cache, every access leads to an L2 hit. This can also be observed in each of the four regions, where the blue curve stays at one hit count per memory access between the markers for the L1 and L2 cache sizes (shown at the bottom of the figure).

When the buffer size exceeds the size of the L2, the number of L2 hits per memory access depends on the parameters of our benchmark. Each region uses different parameter settings, and we will discuss the various effects of these parameters on buffer sizes greater than the L2 cache.

PPB=64: Regions one and three illustrate that for large working sets ("PPB=64") prefetching is disabled, which results in a negligible number of hits per access.

PPB=16: For small working sets ("PPB=16"), which are depicted in regions two and four, successful prefetching leads to an L2 hit rate above zero. These two regions, however, exhibit a difference in the hit rate. This is due to varying stride parameter values in our benchmark.

PPB=16, Stride=64B: On a machine with a cache-line size of 64 bytes—as is the case for our testbed—using a stride of 64 bytes means that the data fetched by the "adjacent cache line prefetcher" will contribute to the hit rate.

PPB=16, Stride=128B: However, when the stride of the benchmark is set to 128 bytes, a lower number of prefetched lines is actually accessed, resulting in a lower hit rate compared to the stride=64 bytes setting.

The red curve depicts the miss rate of the L2 cache. As expected, this curve is complementary to the blue curve depicting the hit rate (ignoring some noise in the measurements).

3.3.2 Instruction Cache Tests

Unlike the case we discussed in the previous section—where the same microbenchmark code was used while key parameters were varied to achieve different results—the instruction benchmark consists of a series of automatically generated microbenchmark functions that have a variable number of instructions. In Figure 3.2, we plot the data generated when the instruction cache benchmark is executed. There are two regions in the figure. Within each region, the microbenchmark functions have the same design, but varying numbers of repetitions of their basic block, which are displayed on the X-axis. The difference between the two regions is as follows.

- **1. region** `TRUE_BRANCH`: Each basic block is enclosed in a branch that will always evaluate to "true" (although it is designed such that it cannot be resolved by the compiler).
- **2. region** `FALSE_BRANCH`: Each basic block holds most of the code inside a branch that will always evaluate to "false." This way, only the first instruction in a cache line will be used, as the rest will not be retired, and thus, resulting in a lower hit rate compared to the results from the first region.

Normalization of data for the purpose of readability:[1] In both regions, we normalize the raw hardware event values by dividing them by the number of repetitions of the basic block, which turns these values into rates. In addition, the below function is applied:

$$F(x) = \frac{log(1 + log(1 + 18.8 \times x^4))}{1.15} \tag{3.1}$$

[1]We perform this normalization on the raw data produced by this benchmark only for presentation purposes because we have observed that the measurements are either around 1.5, or extremely large, and thus some of them cannot be visualized in a readable way, not even in a logarithmic graph.

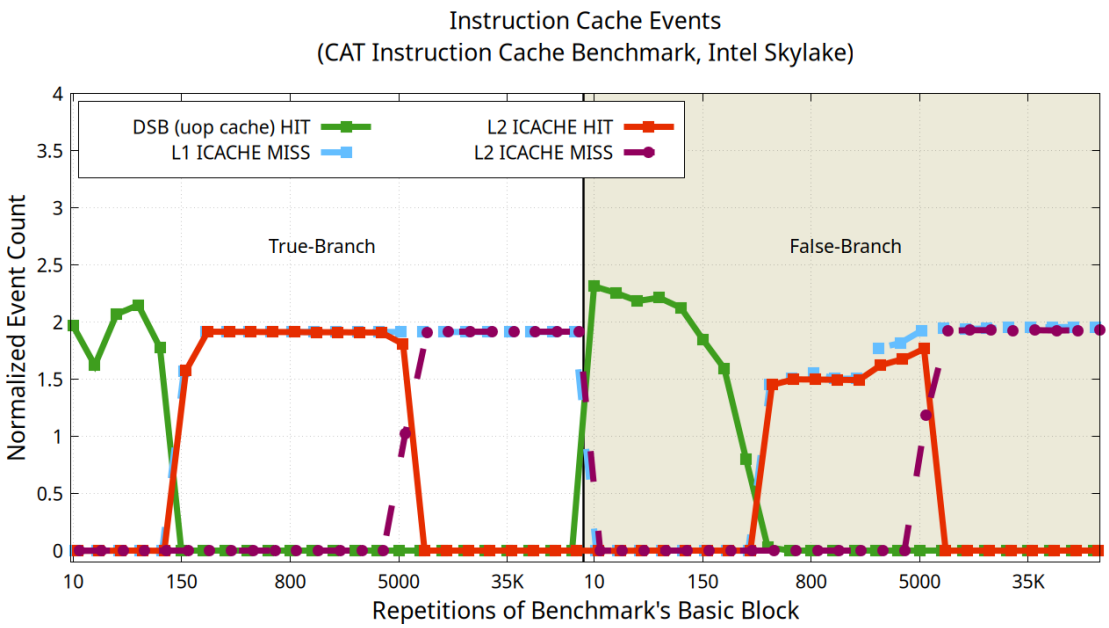


Figure 3.2: Instruction Cache Hardware Events.

This function has the following effects on its input:

- Values lower than 0.5 become smaller.
- Values between 0.5 and 2 are not significantly affected.
- Values larger than 2 grow extremely slowly ($F(10^6) \approx 3.5$).

In Figure 3.2, the green line with the square points depicts the (normalized) hit rate in the Decoded Stream Buffer (DSB), also known as μOP cache. The DSB functions as a level-0 instruction cache, as it is the unit inside each core that caches μOPs after they have been decoded by the Micro Instruction Translation Engine (MITE)—which is the unit that decodes instructions into μOPs. On Skylake, the DSB can hold up to 1,536μOPs. In both regions of the graph, the green line reveals, for small benchmark codes (fewer than 150 repetitions of the basic block), most instructions are delivered to the back-end from the DSB.

The dashed light-blue line with square points depicts the (normalized) miss rate of the L1 instruction cache. In both regions of the graph, we see that the L1 only experiences misses when the code becomes large.

Likewise, the (normalized) L2 miss rate, displayed by the purple curve, follows a similar pattern as the L1 miss rate. The (normalized) L2 hit rate, depicted by the red curve with square points, shows a peak for moderately sized codes, and zero for smaller and larger codes.

In summary, the goal of this work is to generate benchmarks that make these curves different from one another, so we can distinguish between hardware events that have semantic differences. While Figure 3.2 holds a significant amount of data, the curves shown are notably distinct from each other, which substantiates the validity of this effort.

3.3.3 Branch Tests

Figure 3.3 shows a plot of the data generated when the branch benchmark is executed. This test consists of a series of different hand-crafted microbenchmarks, each of which exhibits different behavior from the others with respect to one or more branch instructions. Consequently, when all microbenchmarks are used, each type of branch event produces a unique signature, as can be seen in the figure.

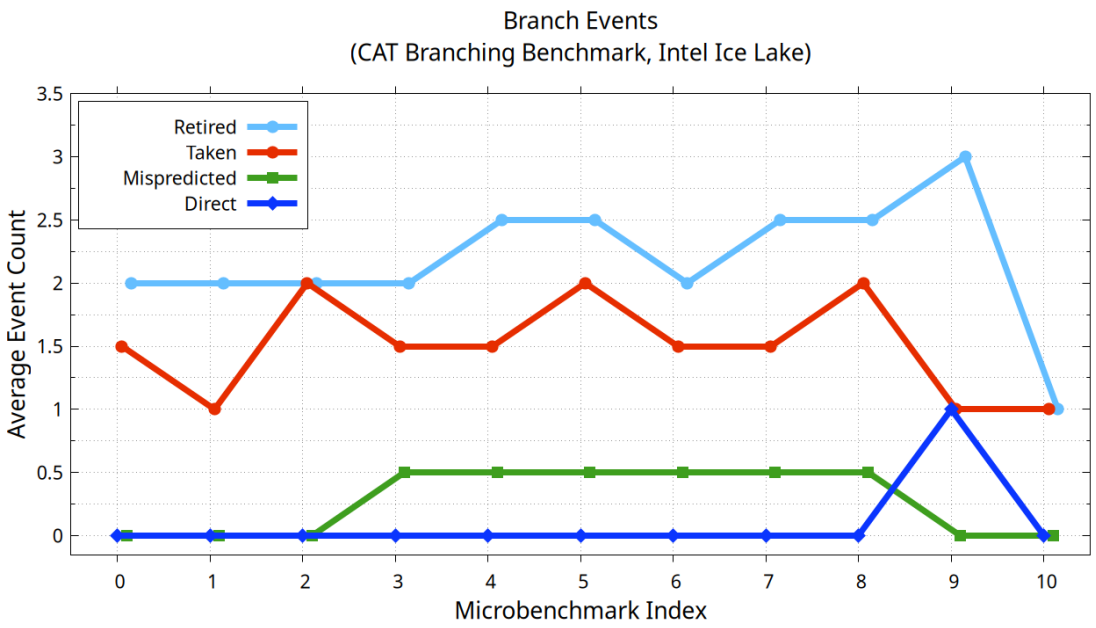


Figure 3.3: Branch Hardware Events.

Listing 3.1: Branch benchmark #5.

```
do{
    iter_count++;
    BUSY_WORK();
    result = pseudo_random();
    if ( (result % 2) == 0 ){
        BUSY_WORK();
        if((global_var1%2) != 0){
            global_var2++;
        }
        global_var1+=2;
    }
    BUSY_WORK();
}while(iter_count<size);
```

Listing 3.2: Branch benchmark #9.

```
global_var2 = 1;
do{
    BUSY_WORK();
    global_var2+=2;
    if(iter_count < global_var2){
        global_var1+=2;
        goto lbl;
    }
    BUSY_WORK();

lbl: iter_count++;
    BUSY_WORK();
}while(iter_count<size);
```

To illustrate the workings of these microbenchmarks, we show the key loop of two of them in the code Listings 3.1 and 3.2, with modifications for readability. These two codes correspond to the measurements shown in the graph at indices 5 and 9, respectively.

Looking at the dark blue curve with the diamond points, we see that at index 5 the value is zero, which means that benchmark #5 does not trigger any direct branch events (`BR_INST_RETIRED:ALL_BRANCHES - BR_INST_RETIRED:COND`). On the other hand, at index 9 the dark blue curve shows a value of one, indicating that benchmark #9 does execute one direct branch per iteration. Looking at the code snippets, we can verify that benchmark #5 does not contain any direct branches, but benchmark #9 includes a `goto` instruction which will execute in every iteration (the enclosing `if` statement is always true).

The green curve with square points indicates that benchmark #5 will experience branch mispredictions (`BR_MISP_RETIRED:COND`) with a rate of 50% per iteration, while benchmark #9 will not experience any mispredictions. This again becomes evident in the code, since benchmark #5 executes a branch that checks the last bit of a randomly generated variable (`result`), and therefore it will be mispredicted 50% of the time, while benchmark #9 does not execute any non-deterministic branches.

The red curve with round points indicates that, in both benchmarks #5 and #9, two conditional branches are taken at each iteration (`BR_INST_RETIRED:COND_TAKEN`). Depending on the compiler, benchmark #9 might only record one taken conditional branch, although the code has two conditional branches—one for the `if` statement and a second one

for the back-edge of the `while` statement. This happens because the compiler generates a jump that is taken when the condition of the `if` statement is false (*i.e.*, it jumps for the `else` case, not for the `if` case), and in the case of benchmark #9 the `if` statement is never false, thus, the branch for the `if` statement is never taken. This explanation is easy to verify by examining the generated assembler code.

The light blue curve with round points (`BR_INST_RETIRED:ALL_BRANCHES`) indicates that benchmark #5 executes two and a half branches per iteration and all of them are retired (*i.e.*, they are not discarded from speculative execution). Benchmark #9 executes three branches per iteration, and all three are retired as well. Examining the code of benchmark #5 reveals that the branch, due to the statement `if((global_var1%2)!=0)`, will only execute for half the iterations (only when the enclosing `if` turns out to be true); and the two branches, due to the enclosing `if` and the `while` statement, will execute once in every iteration. In the case of benchmark #9, the statement `if(iter_count<global_var2)` will be true for every iteration, therefore the direct branch contained in it (`goto`) will execute for every iteration as well, and so will the `while` statement.

Once again, the detailed explanation of each data point in this graph can be complicated by micro-architecture and compiler optimizations, but the difference between the different curves is evident, and thus using these benchmarks helps distinguish between events with different semantics. An additional discussion on the design of our branch benchmarks can be found in [30].

3.3.4 Floating-Point Tests

FLOPs are traditionally separated into the single- and double-precision categories. On IBM's POWER9 architecture, there is additional native hardware support for quad-precision FLOPs [58, 57]. For the sake of consistency across architectures, we closely examine the double-precision FLOPs in the following discussion.

Figure 3.4 shows a plot of the data generated when the floating-point benchmark is executed. As shown in the figure, there are six regions, each of which corresponds to a different Basic Linear Algebra Subprograms (BLAS) kernel being executed. The first three regions (from left to right) correspond to the single-precision ("SP") implementations of the

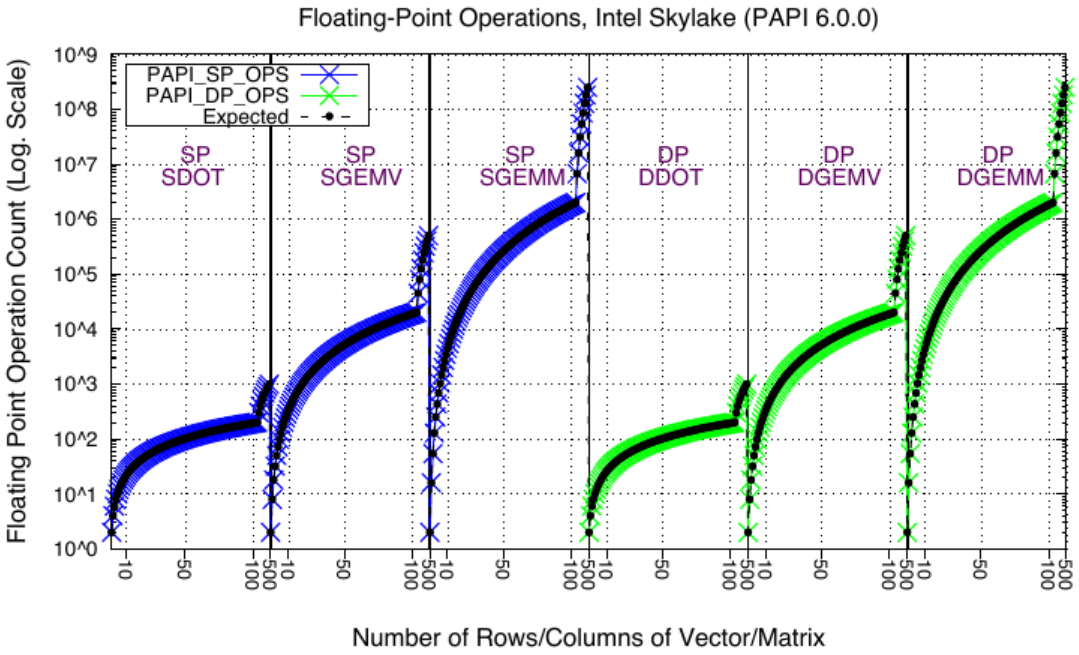


Figure 3.4: Single-Precision and Double-Precision Floating-Point Hardware Events.

Level-1 ("DOT"), Level-2 ("GEMV"), and Level-3 ("GEMM") BLAS kernels (one level per region). The latter three regions correspond to the double-precision ("DP") implementations of the three respective BLAS kernels. More details about the chosen BLAS routines are discussed in Section 3.4.

Within each region in Figure 3.4, the X-axis denotes the number of rows and columns N of the matrix (or vector) being used in the kernel and will hereafter be referred to as the *dimension*. The dimension is incremented per the following piecewise linear progression. For $1 \leq N \leq 100$, N is incremented by 1. For $100 < N \leq 500$, it is incremented by 50. This choice allows us to observe the FLOPs from a larger domain of N while not proportionally increasing the runtime of the range of problem sizes. For each N, the benchmark executes the BLAS kernel of the floating-point precision corresponding to the region. In Figure 3.4, there is a jump in each of the six regions at $N = 100$, resulting from the increment changing from 1 to 50.

In the first region, the blue curve shows the single-precision FLOPs observed during the execution of the DOT kernel for vectors of dimension ranging from 1 to 500. The black curve shows the number of FLOPs that are expected to occur during the DOT kernel, which is $2N$ FLOPs. The second region shows a similar progression for the GEMV kernel. However, the blue curve in this region grows more rapidly than in the first region, as the GEMV kernel invokes $2N^2$ FLOPs. The third region shows that the single-precision FLOPs occur per the $2N^3$ expectation of the GEMM kernel. For the next three regions, the blue curve is constantly zero, corresponding to no single-precision FLOPs being invoked by the double-precision BLAS kernels. The green curve in the next three regions shows that the double-precision FLOPs observed during the double-precision DOT, GEMV, and GEMM kernels perfectly agree with the expectation. The green curve is constantly zero in the first three regions because the single-precision BLAS kernels do not invoke double-precision FLOPs.

3.4 Computation of Arithmetic Intensity for BLAS Kernels

For the study of more precise monitoring of metrics, such as memory traffic and arithmetic intensity, we have chosen different linear algebra routines that are representative of many techniques used in scientific applications, such as computational chemistry, climate modeling, and material science simulations, to name but a few. Dense linear algebra is well represented on most architectures in highly optimized libraries implementing the BLAS API. We present the analysis and study for the DDOT, DGEMV, and DGEMM routines, as they demonstrate a wide range of computational intensities. Our goal is to find answers to the following questions:

1. What is the performance and computational intensity that can be attained on different architectures?

2. Can PAPI's monitoring features for memory traffic and arithmetic intensity help to make meaningful predictions for a real application?

3. How reliable are FLOP and memory traffic hardware events on the different architectures?

BLAS operations are categorized into three levels by the type of operation. Level 1 addresses scalar and vector operations, Level 2 addresses matrix-vector operations, and Level 3 addresses matrix-matrix operations. The BLAS routines provide an excellent means of examining arithmetic intensity and performance characteristics given that they are of high importance to scientific computations and are well-defined and well-understood operations; their implementations are highly optimized by vendor libraries, and the three levels of the BLAS routines have different memory, performance, and computational characteristics.

We examine the Level-1 BLAS routine, DDOT, in greater detail. This is a double-precision operation that multiplies two vectors such that $\alpha = x^T \cdot y$. For the $2N$ FLOPs (multiply and add), DDOT reads $2N$ doubles (assuming $x \neq y$) and writes one double back. Because there is no data reuse, the routine requires $(2N * 8 \text{ bytes})/2N = 8$ bytes per FLOP.

On modern architectures, such an operation is bandwidth limited and will reach about 5-10% of the theoretical peak performance of the machine. The hardware bandwidth will not be able to supply the computational cores with data at a high enough rate to feed the floating-point units.

The Level-2 BLAS routine, DGEMV, is a matrix-vector operation that computes $y = \alpha Ax + \beta y$ where A is a matrix, x, y are vectors and α, β are scalar values. This routine performs $2N^2$ floating-point operations on $(N^2+3N)*8$ bytes for read and write operations, resulting in a data movement of approximately $(8N^2 + 24N)/2N^2 = 4 + 12/N$ bytes per FLOP. When doing a DGEMV on matrices of size N, each FLOP uses $4 + 12/N$ bytes of data. With an increasing matrix size, the number of bytes required per flop stalls at 4, resulting in bandwidth-bound operations.

The Level-3 BLAS routine, DGEMM, performs a matrix-matrix multiplication computing $C = \alpha AB + \beta C$ where A, B, C are all matrices and α, β are scalar values. This operation performs $2N^3$ floating-point operations (multiply and add) for $4N^2$ data movements, reading the A, B, C matrices and writing the results back to C. This means that DGEMM has a bytes/FLOP ratio of $(4N^2 * 8)/2N^3 = 16/N$. When doing a DGEMM on matrices of size N, each FLOP uses $16/N$ bytes of data. As the size of the matrix increases, the number of bytes required per FLOP decreases, until other limits of the processor are reached. The DGEMM has a high data reuse allowing it to scale with the problem size until the performance is near the machine peak.

3.4.1 Results

Our implementations of the BLAS-based benchmarks access a buffer larger than the largest cache after the initialization of the arrays that hold the vectors and matrices, but before the actual numerical operations occur. This is done to ensure the vectors and matrices used in the operations are not present in the cache, but they reside strictly in memory at the start of each BLAS operation. As such, the following implementations differ from the floating-point test of CAT. CAT does not require such a mechanism to be in place since its test only gauges FLOP occurrences and is agnostic to memory traffic. This mechanism does not affect the actual number of FLOPs executed.

The FLOPs hardware events we measure using PAPI are defined by the following PAPI preset on each of the Intel Broadwell, Intel Skylake, and IBM POWER9 architectures: `PAPI_DP_OPS`. This preset event is specifically optimized to count scaled double-precision vector operations. For the sake of completion, it is worth mentioning a second PAPI FLOPs preset event, namely `PAPI_SP_OPS`, which is optimized to count scaled single-precision vector operations. Table 3.1 shows how the two PAPI FLOPs presets are derived from the hardware events as they are available on our three chosen architectures.

For the experiments in this chapter, we exclusively focus on double-precision arithmetic, and thus we will not include `PAPI_SP_OPS` measurements in our analyses.

Figure 3.5 shows the double-precision floating-point operation counts for each of the three levels of BLAS operations for each of the Intel Broadwell, Intel Skylake, and IBM POWER9 CPU architectures. The dimension of the vectors and matrices used in the BLAS operations follows the same piecewise linear progression as in CAT's floating-point tests.

For each of the three BLAS kernels, the expected number of floating-point operations—as calculated and discussed in Section 3.4—matches perfectly the measurements from `PAPI_DP_OPS`. This demonstrates that for the Intel Broadwell, Intel Skylake, and IBM POWER9 architectures, the definitions for the PAPI preset `PAPI_DP_OPS` (as listed in Table 3.1) reliably measure double-precision floating-point operations for various kernels with different computational characteristics.

In Figures 3.6 and 3.7, we plot the statistical minimum and median of the measured memory accesses, taken from 20 executions of the DDOT BLAS operation using the Intel Broadwell and Skylake architectures, respectively. The minimum and median measurements are shown because noise in the measurement can only be positive, so the minimum is the closest to a noise-free measurement, the median provides a sense of the variance, and the maximum can be arbitrarily noisy, so we omit it. We also show the expected number of memory accesses per the following formulation. There are two vectors of N double-precision floating-point elements (each of which is 8 bytes). Thus, a DDOT operation using vectors of length N consumes $2 * 8 * N$ bytes of memory since each element of each vector must be read. There is no expected, systematic pattern of memory writing traffic for the DDOT operation. The memory events we measure count memory traffic in sizes of entire cache lines

Table 3.1: PAPI's Double- and Single-Precision FLOPs Preset Definitions

Architecture	PAPI_DP_OPS	PAPI_SP_OPS
Skylake	FP_ARITH:SCALAR_DOUBLE + 2×FP_ARITH:128B_PACKED_DOUBLE + 4×FP_ARITH:256B_PACKED_DOUBLE + 8×FP_ARITH:512B_PACKED_DOUBLE	FP_ARITH:SCALAR_SINGLE + 4×FP_ARITH:128B_PACKED_SINGLE + 8×FP_ARITH:256B_PACKED_SINGLE + 16×FP_ARITH:512B_PACKED_SINGLE
Broadwell	FP_ARITH:SCALAR_DOUBLE + 2×FP_ARITH:128B_PACKED_DOUBLE + 4×FP_ARITH:256B_PACKED_DOUBLE	FP_ARITH:SCALAR_SINGLE + 4×FP_ARITH:128B_PACKED_SINGLE + 8×FP_ARITH:256B_PACKED_SINGLE
POWER9	PM_DP_QP_FLOP_CMPL	PM_SP_FLOP_CMPL

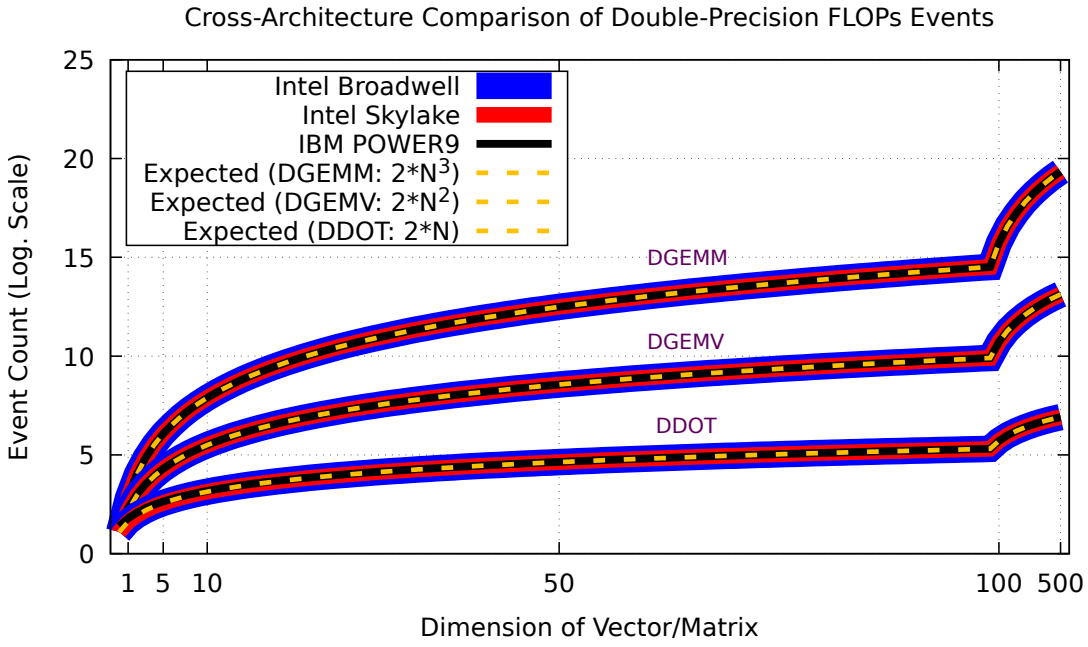


Figure 3.5: BLAS FLOPs on the Broadwell, Skylake, and POWER9 Architectures.

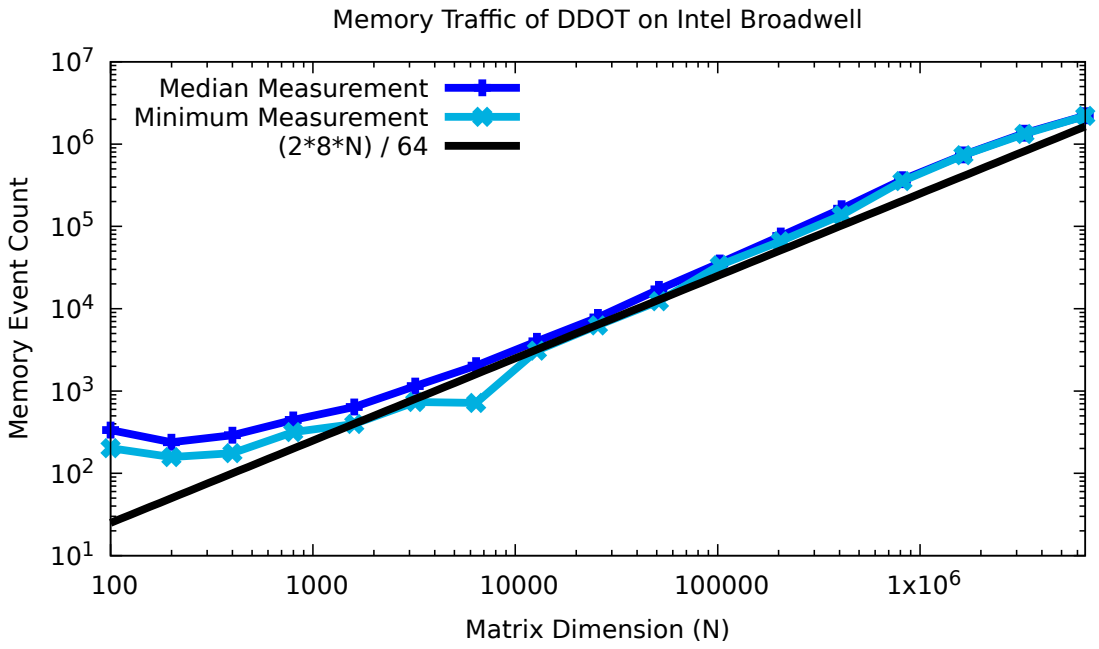


Figure 3.6: DDOT Memory Accesses on the Intel Broadwell Architecture.

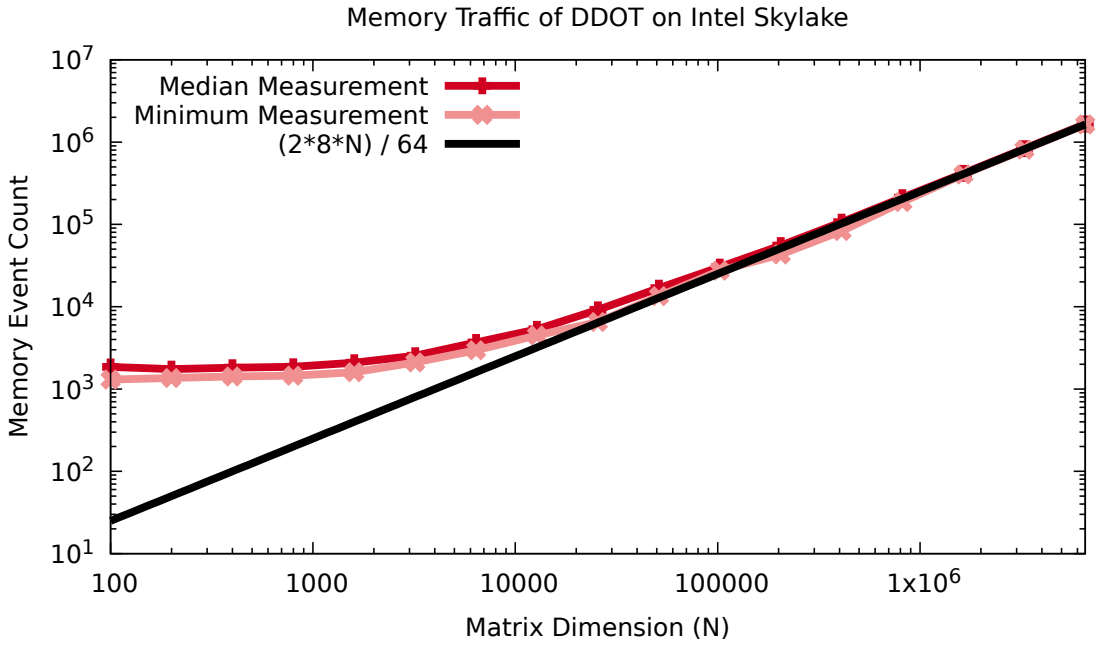


Figure 3.7: DDOT Memory Accesses on the Intel Skylake Architecture.

of memory, and each cache line is 64 bytes. Therefore, the amount of memory traffic we observe by measuring the events is $\frac{(2*8*N)}{64}$. Figures 3.6 and 3.7 show that the measurements of DDOT operations for smaller vector dimensions exhibit background memory accesses from the system on the order of 10^2 and 10^3, respectively. As N increases, the minimum and median measurements very closely agree with the expectation. Note that since the DDOT operation streams through the vectors, there is no data reuse. Thus, DDOT is agnostic to the size of the CPUs' caches. Because of this, when N is large enough such that the memory required to store the two vectors is greater than the size of the cache, the measured behavior should remain close to the expected behavior shown. Figures 3.6 and 3.7 show that the PAPI performance metrics on both the Intel Broadwell and Skylake architectures measure the correct memory traffic for the DDOT operation. In Section 3.5, we elaborate further on the actual PAPI events that we used for measuring memory traffic.

Figure 3.8 and Figure 3.9 show the minimum and median memory access measurements during the DGEMV BLAS operation on the Intel Broadwell and Skylake architectures, respectively. We show the expected number of memory accesses per the following formulation. There are two vectors of N double-precision floating-point elements (each of which is 8 bytes). In addition, there is a matrix of double-precision floating-point elements, of which there are N^2. The DGEMV operation incurs a read for each of the elements of the operand matrix, operand vector, and result vector, totaling $8 * (N^2 + 2 * N)$ bytes read. It incurs a write for each of the elements in the result vector, which would total $8 * N$ bytes written. But other microbenchmarks indicate that the cache writes back to memory in whole counts of a cache line. To account for this, we instead include the term $8 * 8 * N$ ($8 * 8$ bytes $= 64$ bytes, which is the size of a cache line) in the expectation formula shown in Figures 3.8 and 3.9. This term would theoretically have more influence on the total expectation for memory traffic than $8 * N$, but since the bytes read include a term which is quadratic with N, neither $8 * 8 * N$ nor $8 * N$ has a significant numerical impact on the total expectation. Furthermore, since two expectations, including one for each of $8 * 8 * N$ and $8 * N$ bytes written, are visually indistinguishable, we include $8 * 8 * N$. Thus, the total expectation for the memory traffic of the DGEMV operation is the number of bytes read plus the number of bytes written divided by 64, $\frac{(8*(N^2+2*N)+8*8*N)}{64}$, by virtue of the memory traffic events we

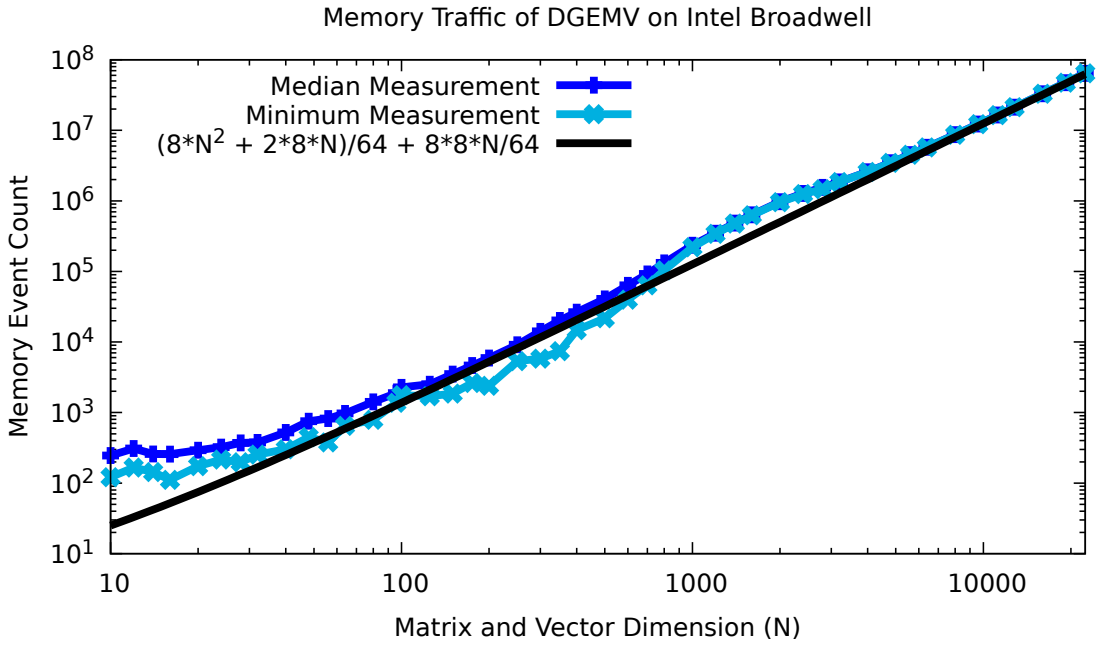


Figure 3.8: DGEMV Memory Accesses on the Intel Broadwell Architecture.

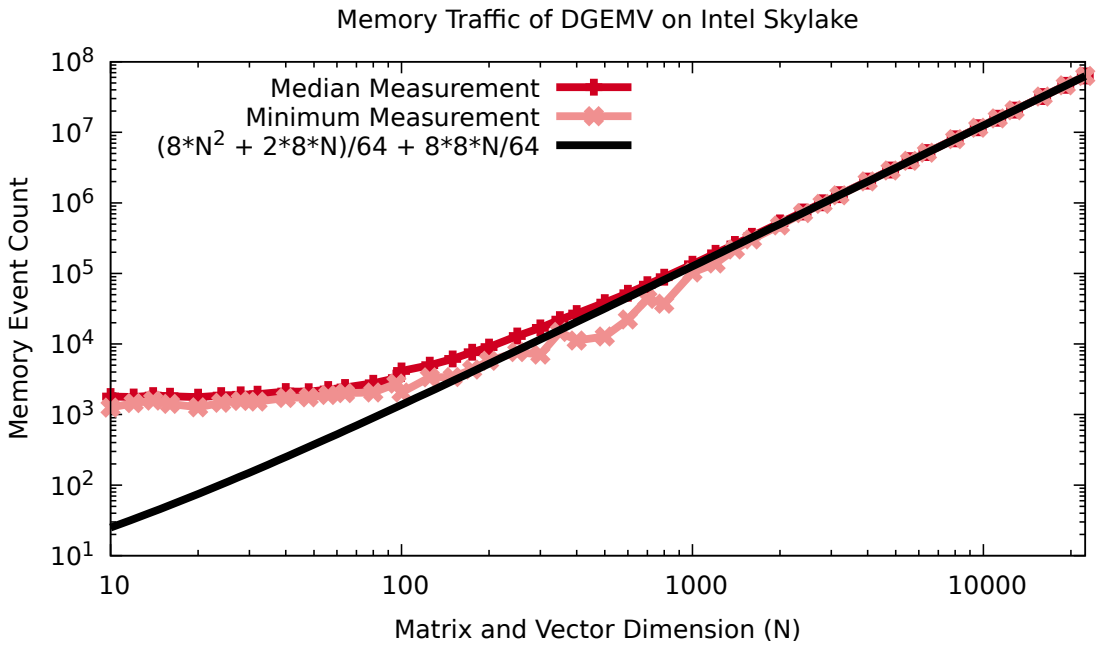


Figure 3.9: DGEMV Memory Accesses on the Intel Skylake Architecture.

measure counting traffic in sizes of entire cache lines. DGEMV has little data reuse since it streams through the operand matrix and result vector. Only the operand vector's data is reused. As such, DGEMV is not sensitive to the size of the cache until the memory required to store the single operand vector of N elements requires enough memory to exceed the size of the cache. As in the case of the DDOT, we see that there is background memory traffic from the system, on the order of 10^2 for Broadwell and 10^3 for Skylake, for small values of N. We observe that as N increases, the measured memory traffic closely agrees with the expectation. Therefore, Figures 3.8 and 3.9 show that the selected hardware events on both the Intel Broadwell and Skylake architectures measure the correct memory traffic for the DGEMV operation.

Figures 3.10 and 3.11 show the minimum and median memory access measurements for the DGEMM BLAS operation (also on the Intel Broadwell and Skylake architectures). There are three matrices (two operand matrices and one result matrix) of N^2 double-precision floating-point elements (each of which is 8 bytes), each of which must be read, resulting in $8 * 3 * N^2$ bytes read. It incurs a write for each of the elements of the result matrix, totaling either $8 * 8 * N^2$ or $8 * N^2$ bytes written, depending on whether the writebacks to memory occur per cache lines written or per elements written, respectively. Since the bytes written for the DGEMM are quadratic in N, there is a significant difference between these two potential memory writing terms with respect to their impact on the total expectation. Thus, we have two expectations, $\frac{(8*(3*N^2+8*N^2))}{64}$ and $\frac{(8*(3*N^2+N^2))}{64}$. We once again divide by 64 here since the events we measure account for memory traffic in the amount of entire cache lines. As such, we show both expectations in Figures 3.10 and 3.11. Unlike the DDOT and DGEMV operations, the DGEMM operation is sensitive to the size of the cache of the CPU on which it is executed because the second operand matrix (which contains a number of elements quadratic with N) is reused for every row of the result matrix which is computed. Depending on how the hardware prefetches and caches data for the DGEMM operation, we establish two bounds for the maximum dimension of matrices which fit within the cache. The sizes of the caches in the Broadwell and Skylake architectures are 35.84 and 25.344 MB, respectively. If the during the DGEMM operation, the hardware caches the entire first and second operand matrices, then we establish a lower bound on the maximum dimension of the

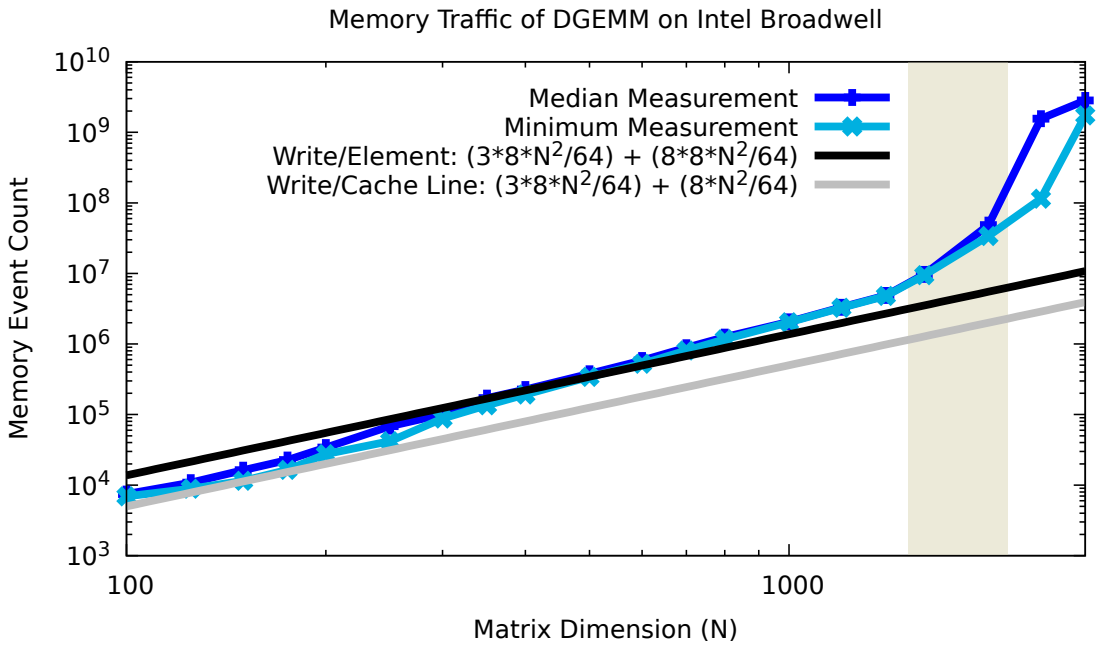


Figure 3.10: DGEMM Memory Accesses on the Intel Broadwell Architecture.

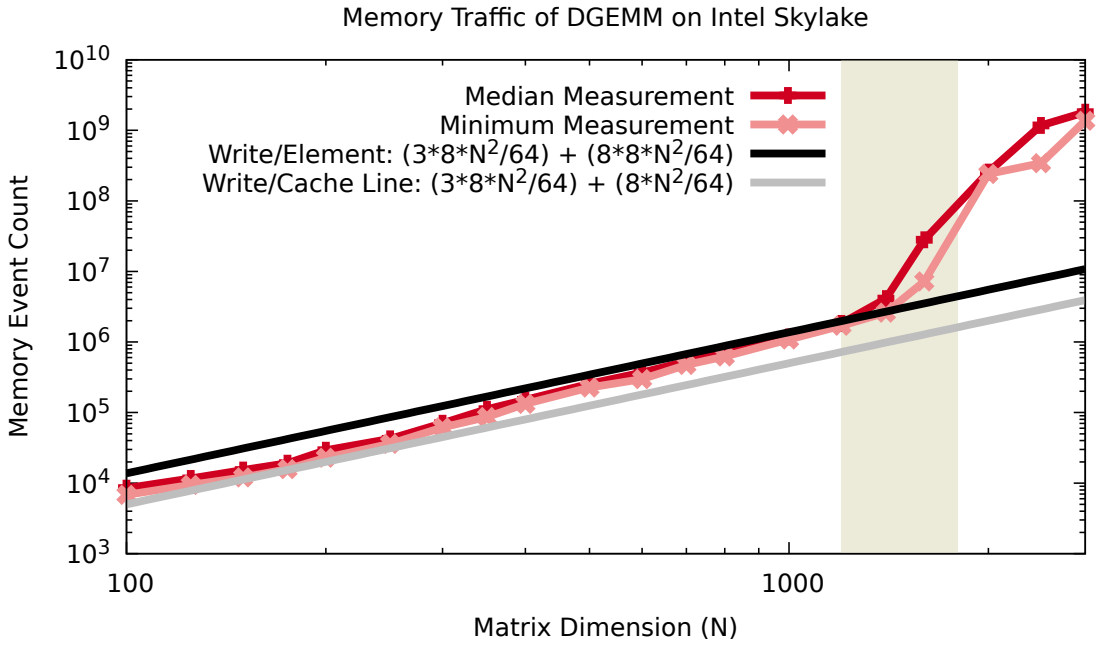


Figure 3.11: DGEMM Memory Accesses on the Intel Skylake Architecture.

matrices which fit within the cache per the following equations (in which we use the cache sizes of the two architectures).

Broadwell: $35840 * 1024 = 2 * 8 * N^2 \implies N = 1514$

Skylake: $25344 * 1024 = 2 * 8 * N^2 \implies N = 1273$

If the hardware caches the entire second operand matrix but only a row of the first operand matrix, we establish an upper bound on the maximum dimension of the matrices which fit within the cache per the following equations.

Broadwell: $35840 * 1024 = 8 * (N^2 + N) \implies N = 2141$

Skylake: $25344 * 1024 = 8 * (N^2 + N) \implies N = 1800$

For each of the above equations, the negative solutions for N are disregarded. The region between these bounds is shaded in each of Figures 3.10 and 3.11. We observe that while N fits well within the size of the caches, the measured memory traffic closely agrees with the expectation. We also observe that for relatively small values of N, the memory writing behavior tends to occur per cache line. However, as N increases, the writing tends to occur per element. Background memory traffic is not prevalent, even for relatively small values of N, due to the large amount of memory accesses incurred relative to the DDOT and DGEMV. Thus, Figures 3.10 and 3.11 show that we obtain the correct measurements for memory traffic for the DGEMM operation utilizing the selected hardware events on the Intel Broadwell and Skylake architectures.

3.5 Benchmarks for Memory Traffic

There are two crucial categories of events to define arithmetic intensity: memory traffic and FLOPs. Memory traffic is further categorized as reading or writing. For the purposes of our benchmarks, memory *reading* traffic entails the amount of data read from memory to the CPU cache, and memory *writing* is the amount of data written to memory from the cache. Among the CAT benchmarks that we have publicly released, the codes for testing the data caches can also be used to test traffic to main memory. This is the case when the buffer size exceeds the size of the last level cache. The known hardware events that we utilize to

measure memory traffic on the Intel Broadwell and Skylake architectures are as follows: Intel Broadwell (One-Socket Node):

```
bdx_unc_imc[0|1|4|5]::UNC_M_CAS_COUNT:[RD|WR]:cpu=0
```

Intel Skylake (Two-Socket Node):

```
skx_unc_imc[0-5]::UNC_M_CAS_COUNT:[RD|WR]:cpu=[0|18]
```

By measuring these events using the CAT data cache reading benchmark, we obtain the plots that follow. We used the same CAT benchmarks to classify the available inter-core events on the IBM POWER9 architecture which correlate with the observed behavior of the memory-reading events on the Intel Broadwell and Skylake architectures shown in Figures 3.12 and 3.13, respectively. The events measured in Figure 3.14 exhibit similar behavior to those of memory reading events measured in Figures 3.12 and 3.13. Note that the expectation in the third and fourth regions in Figure 3.14 varies from those in Figures 3.12 and 3.13 since the size of a cache line on the IBM POWER9 architecture is 128 bytes [58]. Subsequent cross-referencing of [57] verified these events indeed measure the memory reading traffic. Hence, we obtained the following names of the memory traffic events on the IBM POWER9 architecture, which we use to measure memory reading during the execution of the BLAS operations on the IBM POWER9 architecture. IBM POWER9 (Two-Socket Node):

```
pcp:::perfevent.hwcounters.nest_mba[0-7]_imc.
PM_MBA[0-7]_[READ|WRITE]_BYTES.value:cpu[84|172]
```

3.5.1 IBM POWER9 Measurements *via* PCP

Measuring the traffic to main memory requires access to the hardware shared among CPU cores, which measure inter-core events. Therefore, elevated privileges—or very permissive system settings—are required in order to read them. To work around this limitation, IBM also made their inter-core events available through the PCP interface, which can be accessed by any user. To take advantage of this feature, PAPI included a component for interfacing with PCP. As a result, the hardware events for measuring memory traffic on IBM systems can be read using PAPI without the need for elevated privileges. The downside of making

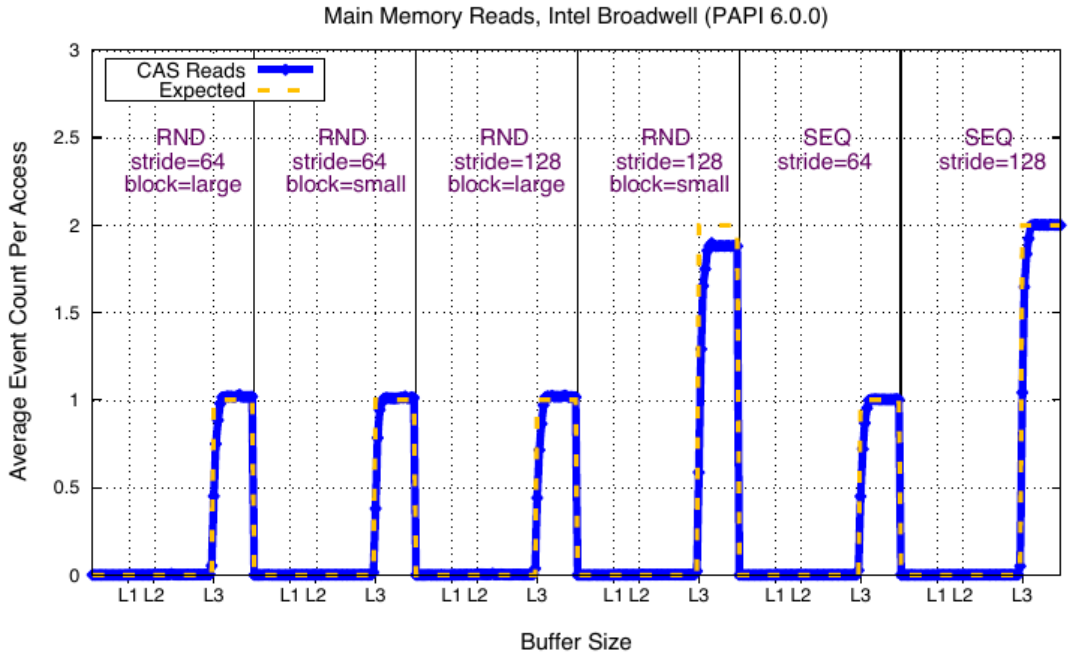


Figure 3.12: Memory Reading Traffic on the Broadwell Architecture.

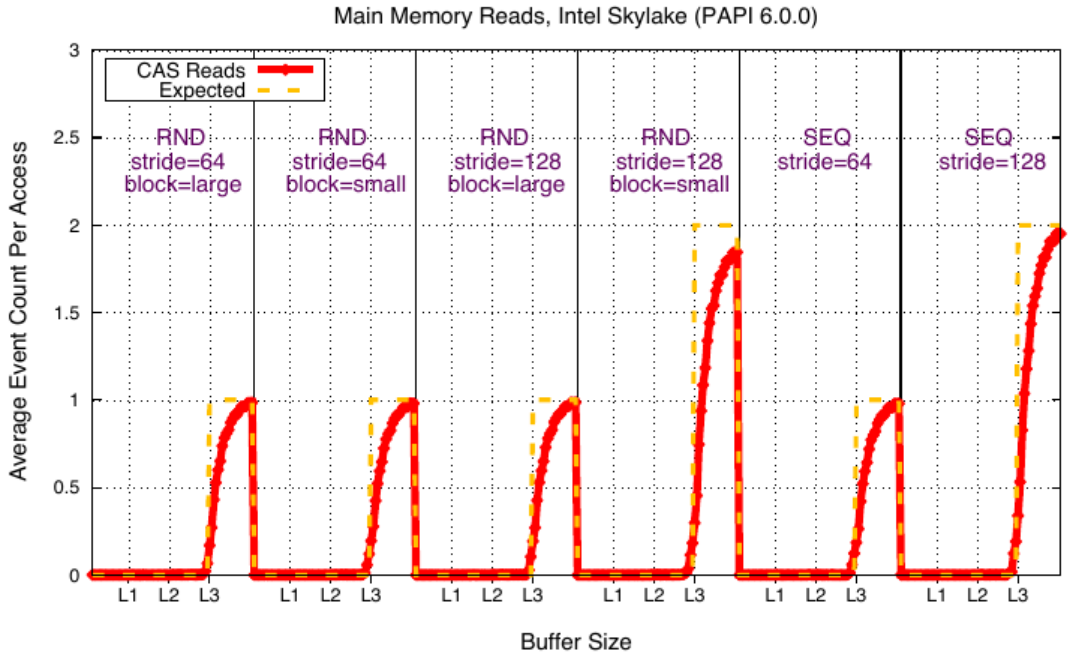


Figure 3.13: Memory Reading Traffic on the Skylake Architecture.

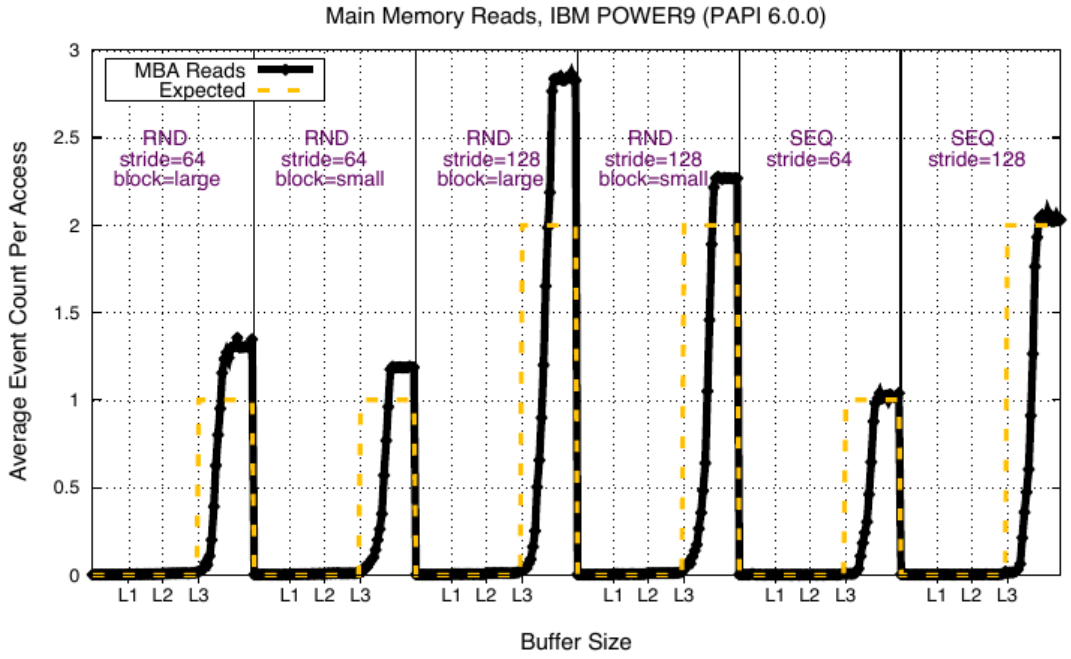


Figure 3.14: Memory Reading Traffic on the POWER9 Architecture.

measurements through PCP is the coarseness of the measurements and the overhead incurred by the PCP daemon. In the rest of this section, we describe our effort to amortize the overheads of PCP in our measurements and give a quantitative analysis of the results. The discussion that follows is focused on the vector dot-product operation (DDOT), but all the techniques we will discuss apply directly to all other kernels we used as benchmarks.

If a measurement infrastructure—*e.g.*, PCP—is susceptible to noise, it is usually beneficial to take measurements of operations that take longer to complete and result in larger measurements in order to amortize the noise. This approach, however, would limit the size of the vectors that we use to very large numbers. Since we aim to correlate the memory traffic measurements with the theoretical expectation for the known linear algebra operations, this limitation is not ideal. To work around this problem, and study the noise in PCP, we used the approach that is shown in Listing 3.3.

As can be seen in the code listing, the actual operation is executed in Line 20. However, instead of simply executing the operation once and measuring it with PAPI, we execute multiple iterations of it. However, simply executing the exact same operation multiple times would skew the memory traffic measurements, since the caches would filter some of the memory requests. To avoid this problem, we allocate memory for multiple copies of the vectors (Lines 1 and 2), and every time we execute the operation, we provide it with a different memory region (*e.g.*, `&v_a[offset]`). Furthermore, we do not just execute the operation a fixed number of times, but rather we vary the number of repetitions (Line 10) in order to study the effect of repetition on noise suppression. Finally, to avoid cache reuse between iterations of the outer loop, we access (in every iteration) a buffer that exceeds all cache sizes (Lines 12-14). We should also note that the actual benchmark contains additional code (not shown to improve readability) that prevents compilers from labeling parts of our code as dead, which would lead to optimizing those parts away.

The results of this study can be seen in Figure 3.15. In these graphs, for any given vector size N the expected number of reads is given by the equation:

$$Reads = \frac{2 \times 8 \times N}{64}$$

Listing 3.3: Benchmark code for amortizing and studying PCP noise.

```
1  v_a = malloc( v_size * max_reps * sizeof(double) );
2  v_b = malloc( v_size * max_reps * sizeof(double) );
3  junk = malloc( LARGE_BUF_SIZE * sizeof(double) );
4
5  for ( i = 0; i <= v_size*max_reps; i++ ) {
6      v_a[i] = ...
7      v_b[i] = ...
8  }
9
10 for ( reps = 1; reps <= max_reps; reps *= 2 ) {
11
12     for( i = 0; i < LARGE_BUF_SIZE; i++ ){
13         junk[i] = ...
14     }
15
16     PAPI_start( EventSetBW );
17
18     for ( iter = 0; iter < reps; iter++ ) {
19         offset = iter * v_size;
20         ddot(v_size, &v_a[offset], &v_b[offset]);
21     }
22
23     PAPI_stop(EventSetBW, &value);
24     printf("%.0lf:", (double)value/(double)reps);
25 }
```

since DDOT reads two vectors with double-precision elements (which use 8 bytes each), and the cache of the target machine (POWER9) implements a memory controller with the "capability to fetch only 64 bytes of data (half cache lines), instead of the normal full cache-line size of 128 bytes of data from the memory when memory bandwidth utilization is very high" [58] (Page 350). The expected number of write operations should be constant, and close to zero, since the DDOT operation does not write anything back into the memory, but rather accumulates the result into a register. Since the DDOT does not write back to

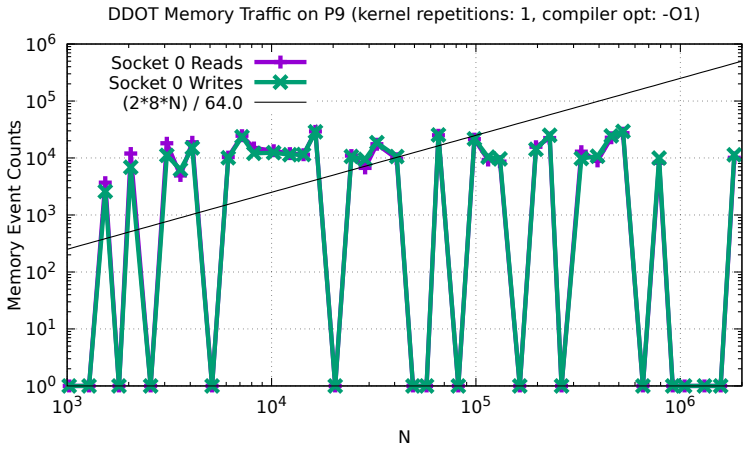
memory, and the measured reads in Figure 3.15 correlate to the measured writes for small N, these reads are regarded as noise.

The graph shown in 3.15a shows the data measured when the operation was repeated only once. Clearly, the measurements do not correlate with the expectation (plotted as a solid black line) due to very heavy noise, for all vector sizes. In 3.15b, we show the measured data when eight repetitions of the operation were used, and as can be seen in the plot, for very large vector sizes the measurements start converging to the expected values. In 3.15c, we used 64 repetitions of the operation and the measurements start converging to the expected values much earlier. Finally, in 3.15d, our benchmark repeats the operation 512 times, and the measurements converge to the expected values very early, and remain close to the expectation.

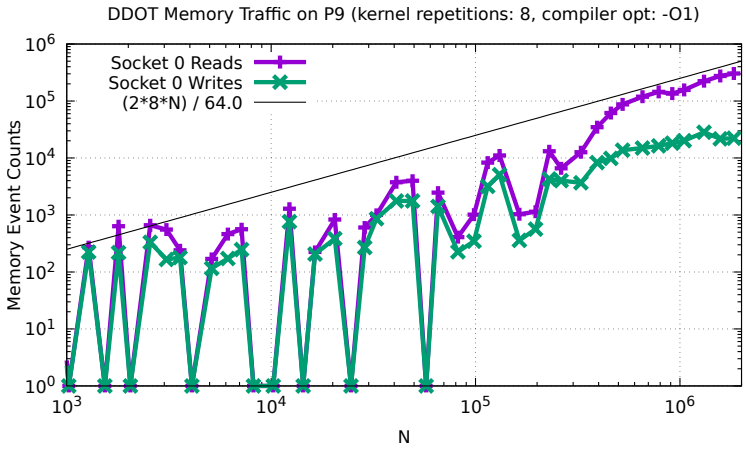
These results are encouraging but at the same time they represent a cautionary tale. On one hand, they show that the experiments we performed on the IBM POWER9 architecture for the purpose of this study were successful in amortizing the overhead and the noise caused by PCP. On the other hand, they highlight the coarseness of the measurements offered by PCP and the limited usability when studying short kernels. In other words, our findings suggest that application developers who wish to study the memory traffic of their applications in coarse intervals can acquire useful measurements without the need for elevated privileges by using PCP. However, library developers who wish to study the behavior of fast kernels need to resort to techniques similar to the one outlined in this section in order to amortize the high overhead and noise of PCP.

3.6 Conclusions and Future Work

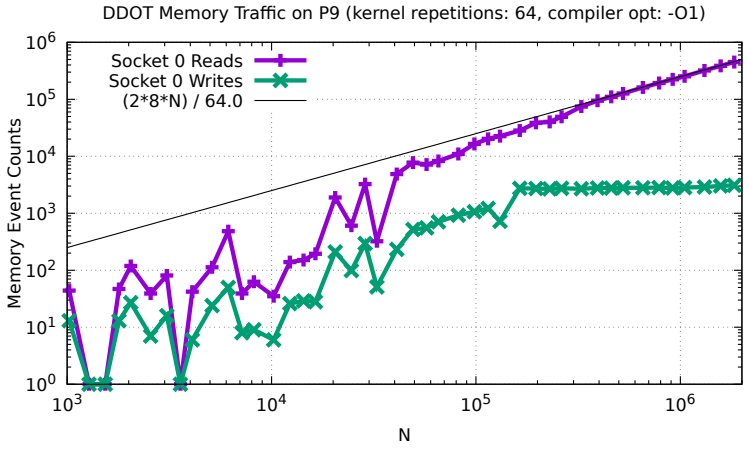
Computing the Arithmetic Intensity of an application or a kernel is essential for understanding its performance, and whether there is room for improvement. However, measuring the quantities necessary to compute the arithmetic intensity—namely floating-point operations and traffic to memory—often entails access to hardware events that may require elevated privileges, or have cryptic names.



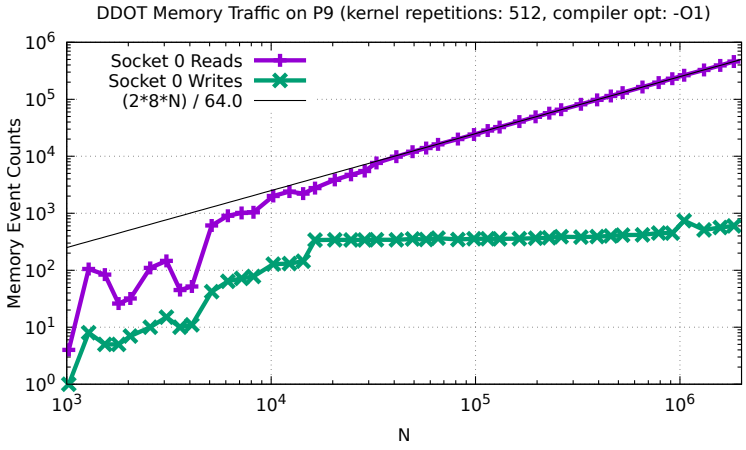
(a) 1 Repetition.



(b) 8 Repetitions.



(c) 64 Repetitions.



(d) 512 Repetitions.

Figure 3.15: POWER9 Measurements of Memory Traffic Events *via* PCP for DDOT Benchmark.

In this chapter, we discussed our effort to simplify the effort of measuring these hardware events and quantifying their reliability through PAPI. In particular, we outlined CAT, a tool that was released with PAPI 6.0, and we showed how it can be used to identify which hardware events are best suited for measuring traffic to main memory. We demonstrated that the arithmetic intensity of three important BLAS operations (DOT, GEMV, GEMM) can be computed on three different architectures (Intel Broadwell, Intel Skylake, IBM POWER9) and explained how PAPI's PCP component can be used on the POWER9 system to sidestep the requirement for elevated privileges. Finally, we performed a study on the reliability of the PCP measurements and explained how the noise and overhead in the measurements can be mitigated, even for small kernels that do not perform enough operations to amortize the noise on their own.

In addition, this chapter addresses the following questions:

1. What is the performance and computational intensity that can be attained on different architectures? On the Intel Broadwell, Intel Skylake, and IBM POWER9 architectures, such performance metrics as FLOPs and main memory traffic are gauged *via* hardware. We have shown that the FLOPs and memory traffic—which occur during the execution of the DDOT, DGEMV, and DGEMM operations—match the expectations for each respective operation.

2. Can PAPI's monitoring features for memory traffic and arithmetic intensity help to make meaningful predictions for a real application? As we have shown, the PCP component in PAPI allows the user to measure the inter-core events for memory traffic for the DDOT, which is a common dense linear algebra operation. The results we have presented indicate that relatively fast kernels, such as DDOT, require multiple repetitions to provide meaningful, expected performance measurements to application developers and performance analysts.

3. How reliable are FLOP and memory traffic events on the different architectures? Per our experiments, the PAPI performance metrics report the expected FLOPs for the three BLAS operations on the Intel Broadwell, Intel Skylake, and IBM POWER9 architectures. The inter-core events also report the expected memory traffic for each

BLAS operation on the Intel Broadwell and Skylake architectures. On the IBM POWER9 architecture, repetitions of the DDOT operation yield the expected amount of memory traffic by amortizing the noise in PCP measurements. Hence, the utilized hardware events provide reliable FLOP and memory traffic measurements across the three architectures we have examined.

The future work related to this chapter includes monitoring multiple events simultaneously (fully utilizing the available hardware counters) instead of monitoring one event per benchmark execution. This would decrease the time needed to measure large numbers of events. This is a challenge because different hardware events occupy different numbers of registers, making it difficult to determine which events can be monitored together.

The benchmarking process can also be improved by monitoring a batch of hardware events using all available compute cores simultaneously. This would require forming batches using only events that are strictly related to compute core resources (as opposed to shared resources), which are not affected by hardware activity from other cores. Other future work will include developing benchmarks for the shared hardware subsystems in GPUs.

Chapter 4

Automatically Defining Performance Metrics from Hardware Events

4.1 Disclosure

This chapter uses content from one of my published papers [16] (© 2024 IEEE as appropriate):

- Barry, D., Danalis, A., and Dongarra, J. (2024). **"Automated Data Analysis for Defining Performance Metrics from Raw Hardware Events."** In *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 716-725, https://doi.org/10.1109/IPDPSW63119.2024.00134.

I am the primary author of this paper. Coauthors for the published paper include Anthony Danalis and Jack Dongarra. Reused coauthor contributions are limited the idea of rounding (shown in Section 4.6), textual improvements, and high-level guidance.

4.2 Introduction

This chapter presents a methodology to (i) automatically classify which hardware events belong to which hardware attributes and (ii) automatically derive meaningful performance metrics using available hardware events.

Hardware events are at the center of application performance analysis. However, the sheer volume of low-level hardware events in modern HPC systems is overwhelming, making them difficult for users to comprehend. Understanding which concepts are monitored by hardware events can be achieved using a two-step process. The first step is the execution of benchmarks designed to stress different hardware attributes in isolation. For every hardware event we wish to understand, we execute the benchmarks while measuring the hardware event. This is achieved by the contributions of Chapter 3. In the second step, the data produced by executing the benchmarks is analyzed to identify what each hardware event actually measures. This chapter presents the methodology for analyzing the data from four previously developed benchmarks that stress key hardware attributes—CPU and GPU floating-point units, branching units, and data caches—to map low-level hardware events to high-level performance metrics. This chapter establishes an automated methodology to express the hardware event data in a well-understood, conceptual basis. This chapter implements a specialized pivoting scheme for QR factorization to identify hardware events that provide distinct information from each other, and techniques for addressing noise in hardware event measurements. Lastly, this chapter utilizes least-squares regression to combine the chosen events to define particular performance metrics of interest.

Performance metrics in HPC systems are monitored by reading the occurrences of various hardware events. However, as a user transitions from one architecture to another, the mapping between hardware events and the concepts they measure becomes increasingly ambiguous due to (i) different architectures containing differing sets of hardware events and (ii) the vast amounts of hardware events present in newer machines.

Modern HPC systems [13, 105, 10] are becoming more and more heterogeneous in their hardware constitution—*e.g.*, deeper and more nuanced memory hierarchies, custom network infrastructures, and GPU accelerators. As such, they contain on the order of hundreds of thousands of hardware events related to the increasingly diverse array of hardware attributes. While these hardware events are documented to some extent in vendors' technical documents [3], they are not always described thoroughly.

This overall lack of clarity makes it challenging for users to comprehend the specific high-level programming concepts—such as total floating-point operations (FLOPs), bidirectional

memory bandwidth, *etc.*—that hardware events actually represent. Further exacerbating this issue, there are far fewer physical counters available than there are hardware events, by several orders of magnitude. Therefore, measuring all hardware events at once is not possible. Even if it was possible, the problem of determining meaningful combinations of the vast amounts of available hardware events to define specific performance metrics is an intractable task to accomplish manually.

Middleware libraries, such as PAPI [101], serve as portability layers that provide definitions for performance-metric presets across diverse architectures using hardware events. Since third-party performance tools—TAU [97], Score-P [96], Vampir [24], Caliper [22], *etc.*—utilize middleware layers to access hardware events, being able to automatically define these performance metrics using available hardware events can have a significant impact on the community. While the Counter Analysis Toolkit (CAT) [17] consists of benchmarks to discover the true concepts measured by hardware events, it has still been necessary to manually parse its output.

The following contributions are achieved in this work.

- We express hardware event data (from CAT benchmarks) as vectors and form them into data matrices both as-is and in hardware-attribute specific bases.
- We implement a special-purpose column-pivoted QR factorization (QRCP) for our data matrices to identify which hardware events represent independent concepts from each other.
- We develop filtering mechanisms to suppress the noise present in event measurements to eliminate bogus outcomes from the linear algebra operations.
- We utilize least-squares regression to identify the best-fit combination of hardware events needed to define high-level performance metrics of interest.
- We showcase how the analysis presented in this chapter can be used to automatically define useful performance metrics for recent x86 CPUs and AMD GPUs.

We conduct experiments on two systems:

- **Aurora at the Argonne National Laboratory:** compute nodes contain Intel Sapphire Rapids CPUs. [11]
- **Frontier at the Oak Ridge National Laboratory:** compute nodes contain AMD MI250X GPUs. [13, 87]

We use the Aurora supercomputer to collect event measurements from the CAT CPU-FLOPs, branching, and data cache benchmarks. To verify that our methods hold for an entirely different category of compute hardware, we collect event measurements by running a new GPU-FLOPs benchmark on the Frontier supercomputer.

4.3 Hardware Event Analysis Methodology

Suppose a programmer is interested in monitoring the number of double-precision floating-point operations performed by their code. For brevity, we will refer to these operations as DP FLOPs. However, many architectures—such as Intel Sapphire Rapids—do not include a hardware event that measures DP FLOPs. Therefore, this quantity has to be constructed by combining measurements from existing hardware events that measure more detailed concepts, such as the floating-point instructions of a particular AVX type (*e.g.,* 256-bit AVX). Combining such hardware events requires that we identify all the relevant hardware events, but it also involves an additional problem that must be addressed. Specifically, these hardware events measure *instructions*, not *operations*. In some cases, such as scalar addition and subtraction, instructions and operations have the same count, but instructions such as fused multiply-add (FMA) perform two operations for every one instruction, and the aforementioned 256-bit AVX instructions perform four FLOPs for every instruction. Therefore, to form the concept of operations, the existing hardware events first have to be scaled and then added together.

The challenge now becomes identifying all the hardware events that measure the independent floating-point instruction concepts. We do this by running a series of microkernels wherein each microkernel performs a known, expected number of a specific type of floating-point instructions, such as single-precision scalar, or double-precision AVX256

FMA, *etc.* For every such microkernel execution, we measure the response of all hardware events found on the target hardware.

Let us assume that the benchmark contains ten kernels. Then for every hardware event e_i, we will get a vector of length ten with values that correspond to the measurements of e_i for the ten kernels. Concatenating the vectors for all hardware events would produce a matrix `A`. Ideally, we could turn our search for a DP FLOPs performance metric into a linear algebra problem. To do so, we first have to handcraft the vector that DP FLOPs would measure for our ten kernels, if such an event existed on the hardware. We will refer to this handcrafted vector as the *signature* of this performance metric. Now, we can use the matrix `A` which contains the real measurements of the hardware events and solve the problem:

$$A \cdot x = b$$

where b is the signature of the performance metric we are interested in. Solving this problem would result in a vector of coefficients, x, that would tell us which columns of `A` need to be combined to form the signature of the performance metric we seek, and by what factor they need to be scaled. Since the columns of `A` directly correspond to hardware events found on the target hardware, solving this problem would tell us how to compose the performance metric of interest using existing hardware events.

However, this problem cannot be solved as such, for multiple reasons. First of all, the matrix `A` is singular. That is, it contains multiple columns of only zeros, since multiple hardware events will not measure anything that relates to kernels with floating-point operations (*e.g.,* hardware events that measure TLB misses). Even if we removed those columns in a filtering step, there will be other non-zero columns that will appear multiple times, columns that are scaled versions of other columns, and columns that are linear combinations of other columns. For example, hardware events that measure integer operations or branch operations would cause this for the floating-point kernels, since the headers of the loops will contain integer operations and branches. In theory, such columns could also be filtered out of `A` using a matrix orthogonalization algorithm, such as QR. However, noise in the measurements can hide linear dependencies between columns, or create

bogus dependencies between columns that ought to be independent. For example, the vectors $(1, 1)$ and $(0.99, 1.01)$ are numerically linearly independent (since one cannot be expressed as a scaled version of the other) but semantically they are the same vector if their difference is solely due to noise. Besides noise, the vast divergence of scale between different columns would cause QR to pick irrelevant hardware events in the resulting matrix. For example, hardware events measuring cycles would lead to columns that have a significantly larger norm than the column resulting from floating-point hardware events. Since the norm of a vector is the criterion that QR typically uses to select vectors, the resulting matrix would contain many irrelevant hardware events.

In the following sections, we describe the analysis and data manipulation we performed on our measurements in order to overcome all of these difficulties and produce meaningful combinations of hardware events that define the performance metrics in which programmers are interested.

4.4 Structure of CAT Benchmark Data

The first microkernel in the CAT CPU-FLOPs benchmark contains three loops as shown in Figure 4.1. The loops contain 24, 48, and 96 double-precision scalar instructions respectively. Let us refer to this kernel as K^{SCAL}. A second microkernel has the same structure, but contains loops with 12, 24, and 48 AVX256 fused multiply-add instructions. Let us refer to this one as K^{256}_{FMA}.

In the interest of readability, in the following discussion we will assume that the target platform only has two types of floating-point instructions, double-precision scalar non-FMA, and double-precision AVX256 FMA.

4.4.1 Performance Metric Signatures

A performance metric that performance analysts are often interested in is double-precision floating-point operations, DP FLOPs. However, this performance metric does not correspond to any hardware event in most existing hardware. Therefore, we must compose it by combining existing hardware events. The challenge is to identify which existing hardware

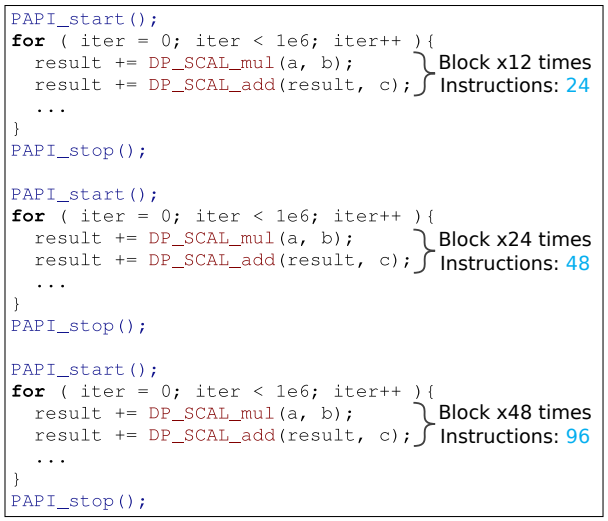


Figure 4.1: Double-Precision Scalar Floating-Point Kernel, K^{SCAL}.

events we need to use and how they need to be scaled in order to properly compose the performance metric in which we are interested.

On the simplified target hardware that we mentioned above, the values of this performance metric during the execution of the K^{SCAL} kernel would be (24,48,96) per iteration, for the three loops of the kernel, since each DP instruction in the kernel performs a DP operation. In contrast, when executing the K^{256}_{FMA} kernel, we would expect the FLOP counts for the three loops to be (96,192,384) since each AVX256 FMA instruction performs eight FLOPs. The measurements from these two kernels concatenated together would form the vector (24,48,96,96,192,384), which we will refer to as the *signature* for DP FLOPs.

Since this event does not exist on the target hardware though, the goal now is to form this signature by adding together scaled measurement vectors of hardware events that actually exist on the target architecture. Let us consider that the target hardware has a hardware event that measures only DP scalar non-FMA instructions, and another that only measures DP AVX256 FMA instructions (both of which are common in real hardware). If we monitor the scalar event while executing kernel K^{SCAL} followed by kernel K^{256}_{FMA}, we will obtain as output the vector (24,48,96,0,0,0), denoted by D^{SCAL}. If we monitor the AVX event while running the same two kernels, we will obtain as output the vector (0,0,0,12,24,48), denoted by D^{256}_{FMA}. Each AVX256 FMA instruction performs eight floating-point operations, so since we are interested in composing a FLOPs performance metric, we need to scale D^{256}_{FMA} by a factor of eight. Scaling and adding these measurement vectors, as shown in Equation 4.1, produces the desired signature for all DP FLOPs from these two instruction-counting hardware events.

$$\text{All DP FLOPs} = D^{SCAL} + 8 \cdot D^{256}_{FMA}$$

$$\begin{pmatrix} 24 \\ 48 \\ 96 \\ 96 \\ 192 \\ 384 \end{pmatrix} = \begin{pmatrix} 24 \\ 48 \\ 96 \\ 0 \\ 0 \\ 0 \end{pmatrix} + 8 \cdot \begin{pmatrix} 0 \\ 0 \\ 0 \\ 12 \\ 24 \\ 48 \end{pmatrix} = \begin{pmatrix} 24 \\ 48 \\ 96 \\ 0 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 96 \\ 192 \\ 384 \end{pmatrix} \tag{4.1}$$

In this simplified example, we demonstrated how we can use the measurements we obtained from monitoring existing hardware events while executing kernels to compose a performance metric based on its "signature." Essentially, the signature describes what we expect the composed performance metric to measure when executing a specific set of kernels.

However, the aforementioned description was an oversimplification of the floating-point hardware. In reality, there are more types of floating-point instructions than strictly scalar and AVX256 FMA in double-precision. To capture this complexity, we use more than the two kernels we discussed previously. Namely, we use kernels for scalar and all AVX vector widths, both FMA and non-FMA, and in both single- and double-precision: Space={scalar, 128, 256, 512}×{FMA,non-FMA}×{SP,DP}

4.4.2 Hardware Event Measurement Normalization

We use the term *expectation* to refer to the vector of expected measurements when monitoring an ideal event while executing all those kernels in sequence. In the above example, the expectation vectors were D^{SCAL} and D^{256}_{FMA}. Note that we used the term "ideal event," as opposed to "hardware event," because some of the expectation vectors might refer to concepts that do not exist in a target architecture. For example, several Intel processors only offer hardware events that count both FMA and non-FMA instructions, instead of hardware events that count strictly one or the other. Similarly, several AMD processors do not offer different events for strictly single-precision, or strictly double-precision instructions. In contrast, our expectation vectors span all of these ideal performance concepts.

Scaling and combining *all* the relevant expectations results in the correct DP FLOPs signature as follows:

$$1 \cdot \mathrm{D}^{\mathrm{SCAL}} + 2 \cdot \mathrm{D}^{128} + 4 \cdot \mathrm{D}^{512} + 8 \cdot \mathrm{D}^{512} + 2 \cdot \mathrm{D}^{\mathrm{SCAL}}_{\mathrm{FMA}} + 4 \cdot \mathrm{D}^{128}_{\mathrm{FMA}} + 8 \cdot \mathrm{D}^{256}_{\mathrm{FMA}} + 16 \cdot \mathrm{D}^{512}_{\mathrm{FMA}}$$

A different way to view this is that we use the expectation vectors to project the signature of the performance metric we are trying to compose (*e.g.,* DP FLOPs) onto a set of "ideal hardware dimensions" defined by the ideal events.

Combining the entire set of expectation vectors into a matrix forms an *expectation basis*, `E`, which we can use as a coordinate system to represent different hardware events. For example, in the coordinate system formed by the expectation basis:

$$E = \begin{pmatrix} | & | & | & | & | & \cdots & | & | & \cdots & | & | & \cdots & | \\ \mathrm{S}^{\mathrm{SCAL}} & \mathrm{S}^{128} & \mathrm{S}^{256} & \mathrm{S}^{512} & \mathrm{D}^{\mathrm{SCAL}} & \cdots & \mathrm{D}^{512} & \mathrm{S}^{\mathrm{SCAL}}_{\mathrm{FMA}} & \cdots & \mathrm{S}^{512}_{\mathrm{FMA}} & \mathrm{D}^{\mathrm{SCAL}}_{\mathrm{FMA}} & \cdots & \mathrm{D}^{512}_{\mathrm{FMA}} \\ | & | & | & | & | & \cdots & | & | & \cdots & | & | & \cdots & | \end{pmatrix}$$

the DP FLOPs signature has the representation:

$$(0, 0, 0, 0, 1, 2, 4, 8, 0, 0, 0, 0, 2, 4, 8, 16)$$

Note that half the values are zeros because they correspond to single-precision (SP) expectations, which do not contribute to the signature of DP FLOPs.

After we have established our expectation basis `E` for the FLOPs benchmark, we can obtain the representation for the measurement vector m_e of a hardware event e by solving the linear algebra problem:

$$E \cdot x_e = m_e$$

where `E` is the expectation basis, and x_e is the resulting representation of m_e.

However, because this linear system is rectangular for some expectation bases, we solve it using least squares. If the least-squares error is too large, then a hardware event cannot be sufficiently represented in the expectation space and therefore is disregarded from further analysis.

After using this process to produce the x_e vector for each hardware event e, we concatenate all such vectors into a matrix `X`. Note that there is a distinct matrix `X` for each set of benchmark kernels (*i.e.,* ideal events for FLOPs, branches, caches, *etc.* are handled independently). In Section 4.5, we describe how we suppress the noise of the vectors that we add into matrix `X`, and in Section 4.6, we explain how we generate a new matrix $\hat{\mathtt{X}}$ that contains linearly independent columns of `X` so that we can use it to define useful performance metrics.

4.4.3 GPU FLOPs

The analysis we are describing in this chapter is not limited to one type of hardware events, nor only hardware events that originate on the CPU, nor any other hardware-event specific limitation. In this section, we utilize a GPU benchmark that stresses the floating-point units. This benchmark contains kernels that do one of addition, subtraction, multiplication, square root, and fused multiply-add. We tested these kernels on Frontier's AMD MI250X GPUs. The symbols we use in this expectation basis are denoted by T_P, where T is the type of operation—A, S, M, SQ, or F—standing for the aforementioned operations. P is the precision of the operation—H, S, or D—denoting half-, single-, or double-precision. Table 4.2 lists the signatures for floating-point performance metrics on the GPU. The portions of signatures corresponding to the FMA kernels are scaled by two because the kernels issue instructions, but an FMA is two arithmetic operations per instruction.

$$E_{\text{GPU\_FLOP}} = \begin{pmatrix} | & | & | & | & \cdots & | & \cdots & | & \cdots & | \\ \text{A}_\text{H} & \text{A}_\text{S} & \text{A}_\text{D} & \text{S}_\text{H} & \cdots & \text{M}_\text{H} & \cdots & \text{SQ}_\text{H} & \cdots & \text{F}_\text{D} \\ | & | & | & | & \cdots & | & \cdots & | & \cdots & | \end{pmatrix} \tag{4.2}$$

4.4.4 Branching

We use the same notation as [30] for the branching basis. The symbols CE, CR, T, D, and M denote *Conditional Branches Executed, Conditional Branches Retired, Conditional Branches Taken, Unconditional (Direct) Branches, and Mispredicted Branches*, respectively.

Table 4.3 lists the signatures for branching performance metrics. *Conditional Branches Not Taken* is equivalent to *Conditional Branches Retired* minus *Conditional Branches Taken*. *Correctly Predicted Branches* is equivalent to *Conditional Branches Retired* minus *Mispredicted Branches*. All other signatures have a one-to-one mapping with the expectations.

$$E_{\text{branch}} = \begin{pmatrix} | & | & | & | & | \\ \text{CE} & \text{CR} & \text{T} & \text{D} & \text{M} \\ | & | & | & | & | \end{pmatrix} = \begin{pmatrix} 2 & 2 & 1.5 & 0 & 0 \\ 2 & 2 & 1 & 0 & 0 \\ 2 & 2 & 2 & 0 & 0 \\ 2 & 2 & 1.5 & 0 & 0.5 \\ 2.5 & 2.5 & 1.5 & 0 & 0.5 \\ 2.5 & 2.5 & 2 & 0 & 0.5 \\ 2.5 & 2 & 1.5 & 0 & 0.5 \\ 3 & 2.5 & 1.5 & 0 & 0.5 \\ 3 & 2.5 & 2 & 0 & 0.5 \\ 2 & 2 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 \end{pmatrix} \tag{4.3}$$

4.4.5 Data Caches

The CAT data cache benchmark performs a pointer chase on a buffer. By controlling the size of the buffer, it can incur hits or misses on the different levels of the cache hierarchy. This benchmark uses multiple concurrent threads working independently on disjoint buffers to add more pressure on the memory subsystem than a single thread would impose. The following symbols are used to denote the expectations for the data caches: L1_{DM}, L1_{DH}, L2_{DH}, and L3_{DH}. In this basis, the symbols DM and DH denote *Demand Misses* and *Demand Hits*. The signatures for various data cache performance metrics are listed in Table 4.4.

4.5 Noise Analysis

As mentioned in Section 4.3, variability in hardware event measurements due to noise is problematic. Thus, we filter such noisy hardware events out of our analysis pipeline. To quantify the variability of a hardware event, we collect the hardware event's measurement vector from multiple repetitions of a CAT benchmark. We compute the root normalized mean-square error (RNMSE) between each pair of measurement vectors (m_e^i and m_e^j) and keep the maximum value. The maximum RNMSE is given by Equation 4.4. In this formula, m_e^i and m_e^j are measurement vectors, and N is the number of elements in each vector. When the denominator in this formula is zero, it means that the average value of a hardware event's measurement vector ($\bar{m}_e^i$ or $\bar{m}_e^j$) is zero. If one of these two quantities is zero, then

we define the variability to be one, corresponding to a 100 percent error.[1] We introduce a noise threshold, τ; if a hardware event has a variability measure greater than the noise threshold τ, it is regarded as being too noisy to be reliably controlled by CAT benchmarks, and the hardware event is not considered for further analysis.

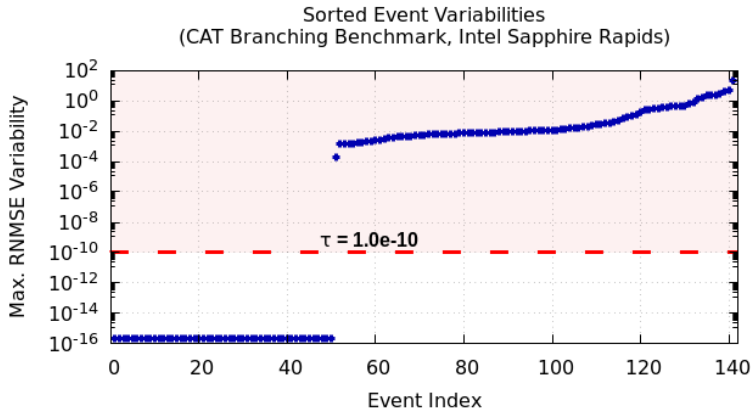
$$\text{Max. RNMSE}(m_e) = \max_{i \neq j} \frac{\|m_e^i - m_e^j\|_2}{\sqrt{N * \bar{m}_e^i * \bar{m}_e^j}} \tag{4.4}$$

In Figure 4.2a, we show the max-RNMSE value for each hardware event measurement from the CAT branching benchmark, sorted in increasing order. There is a cluster of hardware events with a zero variability (these are plotted as machine epsilon on the y-axis for the sake of visualization on a logarithmic scale). From these results, we can see that setting τ to any value from 10^{-4} to 10^{-15} unambiguously divides the zero-noise hardware events from the noisy hardware events. For the experiments described in this chapter, we chose the value 10^{-10} for τ. We discard hardware events with noise above this threshold (indicated by the shaded regions in Figure 4.2). For hardware events with noise below this threshold, we can keep the average measurement vector from all repetitions, or any one of the vectors since all vectors are identical.

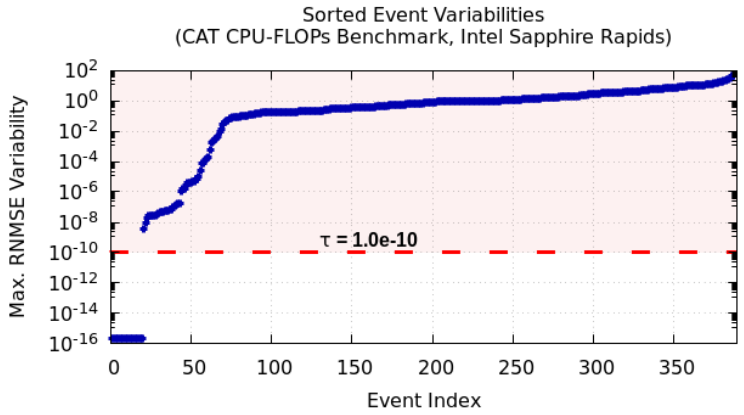
We repeat this analysis for the hardware event measurements from the other CAT benchmarks, as shown in Figures 4.2b-4.2d. The hardware event measurements from the CPU- and GPU-FLOPs benchmarks have a cluster of zero-noise hardware events, similarly to those from the branching benchmark. Therefore, for these two benchmarks, we also set τ to 10^{-10}. For the data cache benchmark measurements, the value of τ is not as clear as it is for the other benchmarks, due to the unpredictability and complex behavior of the cache hierarchy. However, from empirical observations, we found that setting τ to 10^{-1} is sufficient. This is true because this filtering step is only meant to reduce the amount of noisy, irrelevant measurements that will proceed to the next stage of the analysis. Choosing a lenient filtering threshold leading to false positives is better than not applying this step at all.

[1] If all measurements of a hardware event are zero, then the hardware event is discarded as irrelevant.

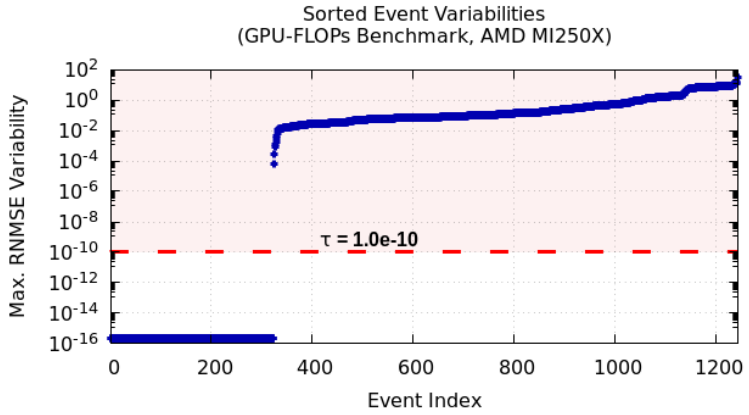
These results show that τ can vary depending on the hardware attribute to which a class of hardware events relates, because different hardware attributes exhibit different levels of noise. Other studies [94] have also found that certain hardware attributes are less prone to



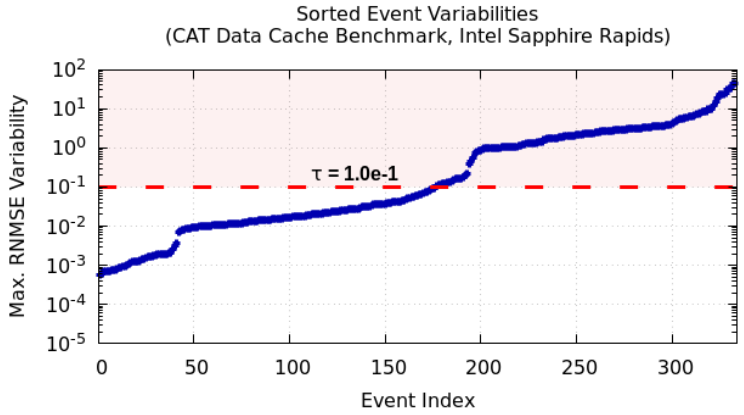
(a) CAT Branching Benchmark.



(b) CAT CPU-FLOPs Benchmark.



(c) GPU-FLOPs Benchmark.



(d) CAT Data Cache Benchmark.

Figure 4.2: Hardware Event Variabilities in CAT Benchmarks.

noise than others. For most classes of hardware events, the separation between noisy and noise-free hardware events is clear, and choosing a cutoff threshold does not require a very careful selection process. For the cache hardware events, where noise is more prominent, we choose a lenient cutoff threshold to only filter out the noisiest hardware events, because in a subsequent step we will use multiple measuring threads and select the median measurement among them to further suppress noise.

4.6 Specialized QRCP Factorization

As we mentioned before, in order to define useful performance metrics from the hardware events, we need the columns of matrix `X` to be linearly independent. The mathematical reason behind this requirement is that if there are linearly dependent columns in a matrix `A`, then when trying to solve the system $A \cdot x = b$, there can be infinite solutions. However, for the purposes of defining a performance metric, there is only one solution (*i.e.,* one combination of hardware events) that is the most meaningful. Another way to view this is that linearly independent hardware event representations are semantically equivalent to hardware events that provide distinct information from each other, and therefore are best equipped to construct higher-level performance metrics.

QRCP is an orthogonal matrix factorization that provides a linearly independent subset of a matrix's columns. Algorithm 4.1 outlines the basic steps in computing the QRCP, where matrix `X` is the input matrix. As can be seen in the listing, π is the array containing the permuted column-indices of `X`. We say that a matrix has rank k if it has k linearly independent columns. In this case the first k entries of π will correspond to the linearly independent columns, and the others will correspond to the remaining columns (which are linearly dependent on the first k).

Algorithm 4.2 is our modified version of the QRCP. Our modifications are focused on the pivot step, so that the linearly independent columns that are chosen by the algorithm best fit the needs of our analysis. Our modified algorithm takes matrix `X` as input and provides the permutation array as output; however, it also takes a tolerance α as input. The primary difference between Algorithms 4.1 and 4.2 is that Algorithm 4.2 uses a special

pivoting scheme (Line 3) that prioritizes columns that are closer to the expectations in the basis. In other words, columns that contain a few values of one and multiple values of zero are prioritized over columns that contain an assortment of values that differ from one and zero. In contrast, in the standard QRCP, the pivot is chosen as the column with the largest norm [98], which is the opposite of what we want for our analysis. Regardless of the pivoting scheme, linear independence of the resulting hardware events is guaranteed by the orthogonalization inherent to QR.

For this scheme to be practical, we introduce the tolerance α to account for noise in hardware event measurements. Each element u of `X` ($u := X_{ij}$) is rounded to the closest integer within a tolerance of α using the formula:

$$R(u) = \alpha \cdot \lfloor \frac{u}{\alpha} + 0.5 \rfloor$$

After rounding the values in `X`, the pivoting scheme scores each column in `X` as follows. Every element $v := |X_{ij}|$ of a column j contributes to the pivoting score of the column based on the following formula:

$$Sc(v) = \begin{cases} v, & \text{if } v \geq 1 \\ 1/v, & \text{if } 0 < v < 1 \\ 0, & \text{if } v = 0 \end{cases}$$

The global minimum score is tracked along the way. If multiple columns have this minimum score, then the tie is broken by choosing the column in `X` with the smallest norm. However, if the norm of a column is smaller than a threshold β (where we define β to be the norm of the vector in which each element is α), then this column is disregarded. This ensures that columns close to the zero-vector are not chosen as a pivot. If all pivot candidates have a norm that is smaller than β, then the algorithm terminates (in Algorithm 4.2, we show this as setting the pivot to -1).

To give an example of the two operations of the two previous formulas, for $\alpha = 0.01$, the vector: $(1.002, 0.001, \text{-}0.5, 1.5)$ would have the score:

$$1 + 0 + \frac{1}{0.5} + 1.5 = 4.5$$

Using the rounding and scoring formulas on the matrix $\mathtt{X}$, followed by the modified QR that we described, results in a matrix $\hat{\mathtt{X}}$ which is either square or overdetermined[2], and has linearly independent columns.

4.6.1 CPU FLOPs

Setting $\alpha = 5 \times 10^{-4}$, Algorithm 4.2 resulted in an $\hat{\mathtt{X}}$ whose columns correspond to the following hardware events:

`FP_ARITH_INST_RETIRED:[128|256|512]B_PACKED_[SINGLE|DOUBLE]` and `FP_ARITH_INST_RETIRED:SCALAR_[SINGLE|DOUBLE]`.

These hardware events chosen by the QR are "good" in the sense that they closely correspond with the expected occurrence patterns and therefore the intended, targeted attributes of the floating-point units.

4.6.2 GPU FLOPs

By setting $\alpha = 5 \times 10^{-4}$, Algorithm 4.2 identified the following key FLOPs hardware events for the GPU:

`SQ_INSTS_VALU_[ADD|MUL|TRANS|FMA]_F[16|32|64]`.[3]

The hardware events `SQ_INSTS_VALU_ADD_F[16|32|64]` occur in equivalent amounts for addition and subtraction kernels, indicating that it counts both types of operations.

4.6.3 Branching

After setting $\alpha = 5 \times 10^{-4}$, Algorithm 4.2 found the hardware events:

[2]The matrix has at least as many rows as it has columns.

[3]These hardware events are prefixed with 'rocm:::' and suffixed with 'device=0' in PAPI; the 'device' qualifier can assume the value of 0-7 on Frontier since there are 8 GPU devices per node, but we need only define performance metrics for a single device. These are excluded from the above text for the sake of readability.

Algorithm 4.1 QRCP.

Input $A \in \mathbb{R}^{m \times n}$
Output $\pi \in \mathbb{N}^n$

1: $\pi \leftarrow [1, \ldots, n]$
2: **for** $i = 1, \ldots, n$ **do**
3: pivot $\leftarrow \text{argmax}_{i \leq j \leq n} \|A_{i:m,j}\|_2$
4: $A_{:,\text{pivot}} \leftrightarrow A_{:,i}$
5: $\pi_i \leftrightarrow \pi_{\text{pivot}}$
6: Update A using column *pivot*.

Algorithm 4.2 QRCP with Special Pivoting.

Input $A \in \mathbb{R}^{m \times n}, \alpha \in \mathbb{R}$
Output $\pi \in \mathbb{N}^n$

1: $\pi \leftarrow [1, \ldots, n]$
2: **for** $i = 1, \ldots, n$ **do**
3: pivot $\leftarrow$ get_pivot(A, π, i, α)
4: **if** pivot == - 1 **then**
5: BREAK
6: $A_{:,\text{pivot}} \leftrightarrow A_{:,i}$
7: $\pi_i \leftrightarrow \pi_{\text{pivot}}$
8: Update A using column *pivot*.

- BR_MISP_RETIRED,
- BR_INST_RETIRED:COND,
- BR_INST_RETIRED:COND_TAKEN,
- BR_INST_RETIRED:ALL_BRANCHES.

4.6.4 Data Caches

Setting $\alpha = 5 \times 10^{-2}$, Algorithm 4.2 chose the hardware events:

- MEM_LOAD_RETIRED:L3_HIT,
- L2_RQSTS:DEMAND_DATA_RD_HIT,
- MEM_LOAD_RETIRED:L1_MISS,
- MEM_LOAD_RETIRED:L1_HIT.

4.6.5 Threshold Sensitivity

The threshold α that we used for the data cache hardware events was chosen to be higher than the other hardware event categories because α is a noise tolerance threshold. As we discussed in Section 4.5, the cache hardware events exhibit higher levels of noise than all other hardware events, which affects this discussion, too. The actual value of the threshold is chosen empirically, but it does not have to be a perfect "magic" value. A wide range of values for α lead to the creation of a matrix $\hat{\mathtt{X}}$ that contains hardware events that properly capture the behavior of the hardware attribute that is tested by the corresponding kernels.

4.7 Defining Useful Metrics

Since we have the linearly independent hardware events from the analysis performed in Section 4.6, we are able to meaningfully solve the system of the form $\hat{X}y = s$, where $\hat{\mathtt{X}}$ is the matrix of hardware events chosen by the QR, and s is the signature for a performance metric that we want to compose, such as those in Tables 4.1-4.4.

We solve this system in the following subsections to define performance metrics as combinations of hardware events. If the QR detected fewer linearly independent hardware events than there are expectations in the basis, then $\hat{\mathtt{X}}$ will have more rows than columns. Since the system is rectangular in this case, we solve it using least squares.

Table 4.1: CPU FLOPs Metric Signatures

Performance Metric	**Signature ($\mathbf{S^{SCAL}}$,...,$\mathbf{D^{512}_{FMA}}$)**
SP Instrs.	$(1, 1, 1, 1, 0, 0, 0, 0, 2, 2, 2, 2, 0, 0, 0, 0)$
SP Ops.	$(1, 4, 8, 16, 0, 0, 0, 0, 2, 8, 16, 32, 0, 0, 0, 0)$
SP FMA Instrs.	$(0, 0, 0, 0, 0, 0, 0, 0, 2, 2, 2, 2, 0, 0, 0, 0)$
DP Instrs.	$(0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 2, 2, 2, 2)$
DP Ops.	$(0, 0, 0, 0, 1, 2, 4, 8, 0, 0, 0, 0, 2, 4, 8, 16)$
DP FMA Instrs.	$(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 2, 2, 2)$

Table 4.2: GPU FLOPs Metric Signatures

Performance Metric	**Signature ($\mathbf{A_H}$,...,$\mathbf{F_D}$)**
HP Add Ops.	$(1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)$
HP Sub Ops.	$(0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0)$
HP Add and Sub Ops.	$(1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0)$
All HP Ops.	$(1,0,0,1,0,0,1,0,0,1,0,0,2,0,0)$
All SP Ops.	$(0,1,0,0,1,0,0,1,0,0,1,0,0,2,0)$
All DP Ops.	$(0,0,1,0,0,1,0,0,1,0,0,1,0,0,2)$

Table 4.3: Branching Metric Signatures

Performance Metric	**Signature (CE,CR,T,D,M)**
Unconditional Branches.	$(0, 0, 0, 1, 0)$
Conditional Branches Taken.	$(0, 0, 1, 0, 0)$
Conditional Branches Not Taken.	$(0, 1, -1, 0, 0)$
Mispredicted Branches.	$(0, 0, 0, 0, 1)$
Correctly Predicted Branches.	$(0, 1, 0, 0, -1)$
Conditional Branches Retired.	$(0, 1, 0, 0, 0)$
Conditional Branches Executed.	$(1, 0, 0, 0, 0)$

Table 4.4: Data Cache Metric Signatures

Performance Metric	**Signature** ($L1_{DM}$, $\cdots$, $L3_{DH}$)
L1 Misses.	$(1, 0, 0, 0)$
L1 Hits.	$(0, 1, 0, 0)$
L1 Reads.	$(1, 1, 0, 0)$
L2 Hits.	$(0, 0, 1, 0)$
L2 Misses.	$(1, 0, -1, 0)$
L3 Hits.	$(0, 0, 0, 1)$

Table 4.5: CPU Floating-Point Metrics

Metric	Combination of Hardware Events	Error
SP Instrs.	+ 1×FP_ARITH_INST_RETIRED:128B_PACKED_SINGLE + 1×FP_ARITH_INST_RETIRED:256B_PACKED_SINGLE + 1×FP_ARITH_INST_RETIRED:512B_PACKED_SINGLE + 1×FP_ARITH_INST_RETIRED:SCALAR_SINGLE	1.67e-16
SP Ops.	+ 4×FP_ARITH_INST_RETIRED:128B_PACKED_SINGLE + 8×FP_ARITH_INST_RETIRED:256B_PACKED_SINGLE + 16×FP_ARITH_INST_RETIRED:512B_PACKED_SINGLE + 1×FP_ARITH_INST_RETIRED:SCALAR_SINGLE	6.05e-18
SP FMA Instrs.	+ 0.8×FP_ARITH_INST_RETIRED:128B_PACKED_SINGLE + 0.8×FP_ARITH_INST_RETIRED:256B_PACKED_SINGLE + 0.8×FP_ARITH_INST_RETIRED:512B_PACKED_SINGLE + 0.8×FP_ARITH_INST_RETIRED:SCALAR_SINGLE	2.36e-1
DP Instrs.	1×FP_ARITH_INST_RETIRED:128B_PACKED_DOUBLE + 1×FP_ARITH_INST_RETIRED:256B_PACKED_DOUBLE + 1×FP_ARITH_INST_RETIRED:512B_PACKED_DOUBLE + 1×FP_ARITH_INST_RETIRED:SCALAR_DOUBLE	5.55e-17
DP Ops.	2×FP_ARITH_INST_RETIRED:128B_PACKED_DOUBLE + 4×FP_ARITH_INST_RETIRED:256B_PACKED_DOUBLE + 8×FP_ARITH_INST_RETIRED:512B_PACKED_DOUBLE + 1×FP_ARITH_INST_RETIRED:SCALAR_DOUBLE	1.69e-19
DP FMA Instrs.	0.8×FP_ARITH_INST_RETIRED:128B_PACKED_DOUBLE + 0.8×FP_ARITH_INST_RETIRED:256B_PACKED_DOUBLE + 0.8×FP_ARITH_INST_RETIRED:512B_PACKED_DOUBLE + 0.8×FP_ARITH_INST_RETIRED:SCALAR_DOUBLE	2.36e-1

$$\text{Backward Error} = \frac{\|\hat{E}y - s\|_2}{\|\hat{E}\|_2 \cdot \|y\|_2 + \|s\|_2} \tag{4.5}$$

The fitness of a least-squares solution is given by the backward error, provided in Equation 4.5. [98] This error is listed in Tables 4.5-4.7 for each signature's least-squares approximation.

4.7.1 CPU FLOPs

Using the hardware events chosen by the QR from Section 4.6 and the signatures from Table 4.1, we utilize least squares to define the floating-point performance metrics in Table 4.5. The number of instructions for both single- and double-precision are simply the sum of the scalar and vector hardware events for the respective precision. The number of operations is a weighted sum of the same hardware events. For the operations performance metrics, each hardware event is scaled by the number of floating-point operands. The least-squares error for most of these performance metrics is extremely small, indicating that these are indeed good performance metric definitions. However, the error is relatively large for the FMA instructions performance metrics. Furthermore, the least squares gives unintuitive linear combination of hardware events for these performance metrics. These results suggest that there do not exist dedicated FMA-counting hardware events in this architecture, which after inspecting the list of hardware events, we verified to be true. Therefore, our analysis correctly identifies both the existence and the absence of hardware events that can compose desired performance metrics.

4.7.2 GPU FLOPs

We repeat the least-squares analysis to define the GPU floating-point performance metrics in Table 4.6. The error for each of the *HP Add* and *HP Sub* performance metrics is relatively large. This again suggests that these performance metrics cannot be defined in isolation on this architecture; however, the performance metric of *HP Adds and Subs* is indeed defined.

Table 4.6: GPU Floating-Point Metrics

Metric	Combination of Hardware Events	Error
HP Add.	0.5×SQ_INSTS_VALU_ADD_F16	4.14e-1
HP Sub.	0.5×SQ_INSTS_VALU_ADD_F16	4.14e-1
HP Add. and Sub.	+ 1×SQ_INSTS_VALU_ADD_F16	5.55e-17
All HP Ops.	2×SQ_INSTS_VALU_FMA_F16 + 1×SQ_INSTS_VALU_MUL_F16 + 1×SQ_INSTS_VALU_TRANS_F16 + 1×SQ_INSTS_VALU_ADD_F16	2.39-17
All SP Ops.	2×SQ_INSTS_VALU_FMA_F32 + 1×SQ_INSTS_VALU_MUL_F32 + 1×SQ_INSTS_VALU_TRANS_F32 + 1×SQ_INSTS_VALU_ADD_F32	2.39e-17
All DP Ops.	+ 2×SQ_INSTS_VALU_FMA_F64 + 1×SQ_INSTS_VALU_MUL_F64 + 1×SQ_INSTS_VALU_TRANS_F64 + 1×SQ_INSTS_VALU_ADD_F64	2.39e-17

The performance metrics for *All Operations* for each precision are also defined using the results from the least squares. These performance metric definitions have very small errors.

4.7.3 Branching

We perform the least-squares analysis for the performance metrics listed in Table 4.7. All of these performance metrics, with the exception of *All Branches Executed*, can be defined for this architecture. The small errors and reasonable linear combinations of hardware events produced by least-squares verify that these performance metrics are well defined; whereas, the near-zero coefficient in the last linear combination, along with the error having the maximum possible value (1) indicate the lack of hardware events that can compose an *All Branches Executed* performance metric.

4.7.4 Data Caches

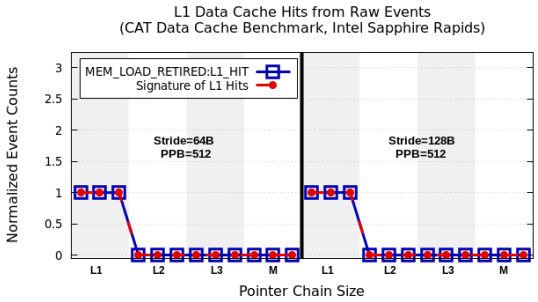
Table 4.8 shows the least-squares results for the data cache performance metrics. For all of these performance metrics, there is very little least-squares error. The coefficients are not exactly zero or one in the hardware event combinations in Table 4.8; however, this can be attributed to the noise from the data cache benchmark. Notice that the coefficients are either within 2% of one, or smaller than 5.87×10^{-3}. Therefore, we can easily round them to one or zero. If we round the coefficients to zero or one to form a new combination, then we can evaluate how well the combination compares to the signature. Figure 4.3 shows that rounding the coefficients from least squares provides an exact match for the signatures. This means that the least squares yields accurate hardware event combinations, even for the noisy data cache hardware events. These results show that we must account for architectural nuance when forming signatures; we are able to discover valuable combinations of hardware events to the extent that we understand the noise present in the hardware attributes of a given architecture.

Table 4.7: Branching Metrics

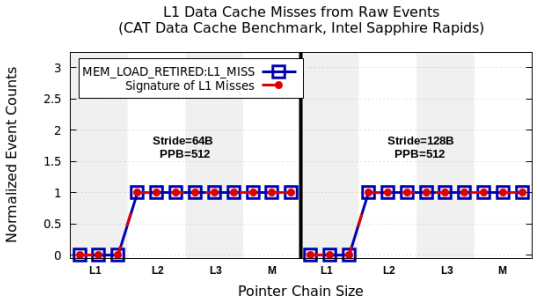
Metric	Combination of Hardware Events	Error
Unconditional Branches.	-1×BR_INST_RETIRED:COND + 1×BR_INST_RETIRED:ALL_BRANCHES	4.03e-16
Conditional Branches Taken.	1×BR_INST_RETIRED:COND_TAKEN	2.15e-16
Conditional Branches Not Taken.	1×BR_INST_RETIRED:COND - 1×BR_INST_RETIRED:COND_TAKEN	2.35e-16
Mispredicted Branches.	1×BR_MISP_RETIRED	9.26e-17
Correctly Predicted Branches.	-1×BR_MISP_RETIRED + 1×BR_INST_RETIRED:COND	2.80e-16
Conditional Branches Retired.	1×BR_INST_RETIRED:COND	4.93e-16
Conditional Branches Executed.	2.22e-16×BR_MISP_RETIRED + 5.62e-16×BR_INST_RETIRED:COND + 1.53e-16×BR_INST_RETIRED:COND_TAKEN + 9.86e-32×BR_INST_RETIRED:ALL_BRANCHES	1.0

Table 4.8: Data Cache Metrics

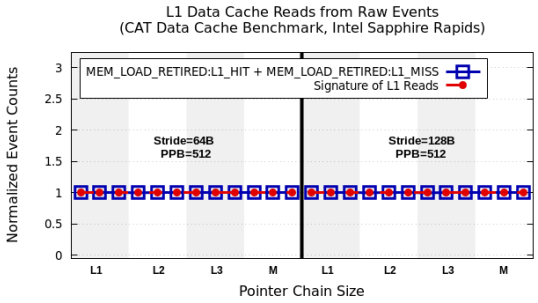
Metric	Combination of Hardware Events	Error
L1 Misses.	2.56e-3×MEM_LOAD_RETIRED:L3_HIT + 3.50e-4×L2_RQSTS:DEMAND_DATA_RD_HIT + 1.00001×MEM_LOAD_RETIRED:L1_MISS - 3.05e-4×MEM_LOAD_RETIRED:L1_HIT	4.07e-16
L1 Hits.	-5.69e-6×MEM_LOAD_RETIRED:L3_HIT - 4.21e-4×L2_RQSTS:DEMAND_DATA_RD_HIT - 4.19e-6×MEM_LOAD_RETIRED:L1_MISS + 0.9996×MEM_LOAD_RETIRED:L1_HIT	9.64e-17
L1 Reads.	2.55e-3×MEM_LOAD_RETIRED:L3_HIT - 7.14e-5×L2_RQSTS:DEMAND_DATA_RD_HIT + 1.00001×MEM_LOAD_RETIRED:L1_MISS + 0.9993×MEM_LOAD_RETIRED:L1_HIT	1.70e-16
L2 Hits.	-5.87e-3×MEM_LOAD_RETIRED:L3_HIT + 1.003×L2_RQSTS:DEMAND_DATA_RD_HIT - 2.39e-3×MEM_LOAD_RETIRED:L1_MISS - 3.11e-4×MEM_LOAD_RETIRED:L1_HIT	2.51e-16
L2 Misses.	8.43e-3×MEM_LOAD_RETIRED:L3_HIT - 1.002×L2_RQSTS:DEMAND_DATA_RD_HIT + 1.002×MEM_LOAD_RETIRED:L1_MISS + 5.74e-6×MEM_LOAD_RETIRED:L1_HIT	2.33e-16
L3 Hits.	1.02×MEM_LOAD_RETIRED:L3_HIT + 5.22e-3×L2_RQSTS:DEMAND_DATA_RD_HIT - 5.26e-3×MEM_LOAD_RETIRED:L1_MISS - 5.88e-6×MEM_LOAD_RETIRED:L1_HIT	2.54e-16



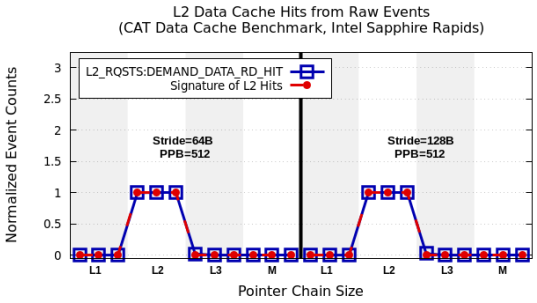
(a) L1 Hits.



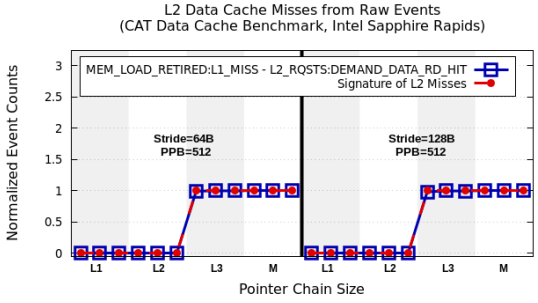
(b) L1 Misses.



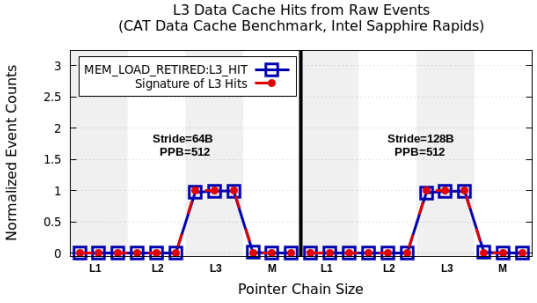
(c) L1 Reads.



(d) L2 Hits.



(e) L2 Misses.



(f) L3 Hits.

Figure 4.3: Various Data Cache Performance Metric Approximations from Least Squares.

4.8 Conclusions and Future Work

In this chapter, we have introduced an automated mathematical analysis to parse through thousands of hardware events and identify those most pertinent to specific hardware attributes. We have demonstrated the use of this analysis with both CPU and GPU floating-point units, branching units, and the memory subsystem, by coupling the analysis with the CAT benchmarks on the Intel Sapphire Rapids CPU and the AMD MI250X GPU architectures.

We are able to account for high levels of noise by collecting hardware event measurements multiple times and excluding hardware events that have high run-to-run variability across measurements. We quantify this noise using the maximum RNMSE between two measurements. For the data cache benchmark, we use multiple threads for our experiments, and minimize the noise by keeping the median reading across all threads.

These strategies sufficiently precondition measurement data prior to the QR factorization. Furthermore, our specialized QR factorization allows us to account for small noise, making it useful for data analysis of practical counter readings. We observed that branching and FLOPs (both CPU and GPU) hardware events exhibit relatively low amounts of noise when executing the CAT benchmarks; however, the measurements of hardware events corresponding to the memory subsystem are noisier.

We implemented a specialized QR pivoting strategy that chooses the best sets of hardware events representing our hardware expectation bases. It accomplishes this by prioritizing hardware events which are closest to the individual dimensions of the expectation basis.

After producing the set of linearly independent hardware events from the QR, performing least squares on the resulting matrix successfully provided linear combinations of hardware events for the desired performance metrics, and it correctly indicated through the resulting error the cases where a performance metric cannot be composed from hardware events on a given architecture. This analysis provides a numerical notion of fitness, which aids in validating hardware event combinations. Our analysis defined architecturally available branching, floating-point, and cache performance metrics.

Even in the case of the cache performance metrics, where noise is the most prevalent, we demonstrated that rounding the resulting coefficients by just a few percent results in simple combinations of hardware events that behave in ways that perfectly match the desired signatures of the performance metrics.

Future work will entail methods to develop different measures to quantify hardware event noise and more rigorously select noise suppression thresholds and pivoting criteria.

Chapter 5

Benchmarking Strategies for Inter-core Events

5.1 Disclosure

This chapter uses content from one of my published papers [18] (© 2023 IEEE as appropriate):

- Barry, D., Jagode, H., Danalis, A., and Dongarra, J. (2023). **"Memory Traffic and Complete Application Profiling with PAPI Multi-Component Measurements."** In *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 393-402, https://doi.org/10.1109/IPDPSW59300.2023.00070.

I am the primary author of this paper. Coauthors for the published paper include Heike Jagode, Anthony Danalis, and Jack Dongarra. Reused coauthor contributions are limited to the idea of the capped GEMV (shown in Section 5.3.1), textual improvements, and high-level guidance.

5.2 Introduction

This chapter introduces techniques to accurately decode and verify inter-core events. Some of the most important categories of hardware events count the data traffic between the processing cores and the main memory. However, since these hardware events are not core-private, applications require elevated privileges to access them. PAPI offers a component that can access this information on IBM systems through the Performance Co-Pilot (PCP); however, doing so adds an indirection layer that involves querying the PCP daemon. This chapter performs a quantitative study of the accuracy of the measurements obtained through this component on the Summit supercomputer. We use two linear algebra kernels—a generalized matrix multiply, and a modified matrix-vector multiply—as benchmarks and a distributed, GPU-accelerated 3D-FFT mini-app (using cuFFT) to compare the measurements obtained through the PAPI PCP component against the expected values across different problem sizes. We also compare our measurements against an in-house machine with a very similar architecture to Summit, where elevated privileges allow PAPI to access the hardware events directly (without using PCP) to show that measurements taken *via* PCP are as accurate as the those taken directly. Finally, using both QMCPACK and the 3D-FFT, we demonstrate the diverse hardware activities that can be monitored simultaneously *via* PAPI hardware components.

The amount of data that is moved between the processor and the main memory of a computer often has a higher impact on the performance of an application than any other aspect of the application. Typically, application developers, as well as compiler optimizations, try to structure data accesses so that the cache hierarchy is utilized in order to minimize traffic to the main memory. However, data still needs to be transferred to and from the main memory for all but the simplest codes.

Assessing the amount of memory traffic during the execution of a program relies on hardware events that are not private to the core in which the program is running, and thus are referred to as Uncore, Northbridge, or Nest, depending on the hardware vendor.[1] However, since the main memory is a shared resource among all processors, applications

[1] In the rest of this chapter, we will use the term "Nest" since these studies are focused on IBM systems.

must run with elevated privileges in order to access these hardware events. Unfortunately, the typical user of high-performance systems, such as the Summit supercomputer, does not usually have the privileges needed to query the Nest counters. To circumvent this problem, IBM chose to utilize the Performance Co-Pilot [92] to export Nest-related information to ordinary users.

The middleware library PAPI [101] offers a component that interfaces with PCP. Through this component, third-party performance tools that depend on PAPI to access hardware events, such as TAU [97], Score-P [96], Vampir [24], Caliper [22], *etc.*, can acquire information regarding the memory traffic of applications without the need for elevated privileges. The PCP component of PAPI operates by communicating with the Performance Metrics Collector Daemon (PMCD) running on a given system. The PMCD runs with the special privileges needed to query the Nest counters. PAPI then queries the PMCD *via* the PCP component without the user requiring any special permissions. This enables all PAPI users to monitor Nest hardware events from user space without elevated privileges and without using multiple APIs to access these hardware events from across the system. Also, one of PAPI's commitments as a portability layer is the thorough validation of the hardware events exposed to the user to account for unreliable hardware events, especially when there are multiple sources of hardware events.

Another burden in assessing whether an application uses the available resources efficiently is the heterogeneous nature of modern machines. This has resulted in modern applications that have a hierarchical structure in order to utilize multi-core CPUs, GPUs, and distributed-memory execution all at the same time. However, assessing whether all the parts of an application use the corresponding hardware components efficiently and timely requires collecting information from multiple diverse sources at the same time. The multi-component structure of PAPI allows this diverse collection of data, and as we demonstrate in this chapter, enables the user to assess the efficiency of the whole execution.

The following contributions are achieved in this work.

- We showcase monitoring of hardware events that measure memory traffic using the PAPI PCP component. We perform a series of experiments with various computational kernels—DGEMM, a modified DGEMV, and data re-sorting subroutines of a

distributed-memory 3D-FFT—to quantify the accuracy of the measurements taken *via* PCP.

- We evaluate the accuracy of measurements taken *via* PCP and compare them with those taken directly from the Nest counters.
- We elucidate micro-architecturally driven nuance in memory traffic incurred by streaming and strided data access patterns. Accessing data in strides has a similar impact on memory traffic as utilizing software prefetching. Both incur a read-per-write to memory.
- We use a hybrid 3D-FFT mini-app and the QMCPACK application to demonstrate how PAPI can be used to monitor and correlate the activity of multiple hardware components of a heterogeneous, distributed-memory system.

We conduct experiments on two systems:

- **Summit at the Oak Ridge National Laboratory:** each compute node has two sockets containing 22-core IBM POWER9 CPUs and NVIDIA Tesla V100 GPUs. We do not have elevated privileges on this system; therefore, we use the PAPI PCP component to measure memory traffic.
- **Tellico at the University of Tennessee Knoxville:** a two-socket testbed containing 16-core IBM POWER9 CPUs in which we do have elevated privileges, so we measure Nest events without the use of PCP. We define the perf_uncore events using the Nest IMC Memory Offsets [57]. This serves as a basis for comparison of the fidelity of measurements from Summit using PCP.

The memory traffic hardware events we measure on these systems are listed in Table 5.1. In our experiments, we pin only one thread to each physical core. Although there are 22 cores per socket, one of these cannot be accessed by the user because it is "set aside for system service tasks" [86].

Table 5.1: Architectures and Hardware Events

System	Arch.	Hardware Events
Summit	IBM POWER9	pcp ::: perfevent . hwcounters.nest_mba[0−7]_imc :PM_MBA[0−7]_[READ|WRITE]_BYTES :value :cpu[87|175]
Tellico	IBM POWER9	power9_nest_mba[0−7]::PM_MBA[0−7]_[READ|WRITE]_BYTES :cpu=0

5.3 BLAS Benchmarks for Memory Traffic

There are two prevalent Basic Linear Algebra Subprograms (BLAS) operations we use to evaluate the accuracy of memory traffic measurements from the PCP component: the matrix-vector (GEMV) and matrix-matrix multiplications (GEMM). Validating PAPI's capabilities to monitor memory traffic contributes to an ongoing effort to design scalable math library routines. However, we cannot inspect proprietary vendor library kernel implementations at a sufficient fine-grain granularity in order to use them to verify the identities of performance hardware events. Hence, we examine reference implementations of these two BLAS operations. Note that the absolute performance achieved by these kernels is not relevant to this work. We only use them to evaluate the accuracy of the measurements of the hardware counters against the expected behavior of these kernels, and therefore the reference implementations are entirely sufficient for this study. We vary the size of these operations and monitor and analyze their memory reading and writing traffic.

In previous work [17], we accounted for noise in memory-related measurements on Intel architectures by taking the minimum or median counter reading of multiple executions, or *repetitions*, of BLAS operations. On IBM POWER9, we used the average over 512 repetitions of the DOT operation, executing a different problem size each time to prevent data reuse. Since the work presented here focuses exclusively on POWER9, we use average readings for GEMV and GEMM. Also, we vary the number of repetitions with problem size and contrast batched and serial kernels, while in the previous work, we only used serial kernels.

5.3.1 GEMV

Suppose we have the vectors $x \in \mathbb{R}^N, y \in \mathbb{R}^M$ and a matrix $A \in \mathbb{R}^{M \times N}$, with each element being a double-precision, floating-point number. The GEMV operation is defined as `y = Ax`, where the ith element of vector `y` is the dot product of the vector `x` and the ith row of matrix `A`.

```
1 for ( i = 0; i < M; i++ ){
2     /* Dot product of row of A and x. */
3     sum = 0.0;
4     for ( k = 0; k < N; k++ )
5         sum += A[i][k] * x[k];
6     /* Store result in memory. */
7     y[i] = sum;
8 }
```

Listing 5.1: Reference GEMV Kernel.

Per line 6 of Listing 5.1, the reference GEMV incurs one read from memory to retrieve the kth element of a row of matrix `A` and one read to retrieve the kth element of input vector `x`. The vector `x` gets reused for every successive iteration of the outer for-loop, so if it fits in the cache it will only be read from memory once, in the first iteration of the outer for-loop. Therefore, the entire kernel causes `N` reads for vector `x`, `M`×`N` reads for matrix `A` (which is only accessed once, so no reuse is possible), and `M` reads are incurred by the hardware when writing into the vector `y`. Therefore, a total of `M`×`N`+`M`+`N` elements are read from memory for GEMV.

In order to observe a very large amount of memory writing traffic, the resulting vector `y` must be large, since only `y` is being written. However, to produce a vector `y` of size `M` we need a source matrix of size `M`×`N`. Storing this matrix in memory limits the maximum output size `M` that we can use in our experiments. Making things even worse, in our actual experiments we store a different matrix `A` for each thread. The reason this is necessary is because if all threads shared a common matrix `A`, then one of the threads would fetch a portion of the matrix into the shared L3 cache, and the other threads would read it directly from there, resulting in a complicated data transfer pattern that is difficult to predict and analyze. Furthermore, to ensure no data is cached between different repetitions of our experiment we use a different matrix `A` for each repetition. Therefore, when using a large size `M` the memory requirements for allocating all the necessary data would exceed the practical memory limitations of the systems we used. One option is to use a matrix with a small width, `N`, and only vary the height, `M`. Even with this approach, we would still encounter the limitation of how big of a problem size we can use, as we only limit the width of matrix `A`.

As such, we used a modified operation to limit both the width and height of matrix `A` *without restricting the size of the vector* `y`. This allows us to simulate the memory traffic of a GEMV that computes a very large vector `y` (thus increasing memory write traffic) while not consuming as much memory for allocating matrix `A`. The modified operation is defined as follows. We introduce the new variable `P` $= min${`M,N`}$$ and cap the size of matrix `A` to `P`×`N`. Then we access the rows of matrix `A` multiple times using the index $i_p = i\%P$.

$$y_i = \sum_{k=1}^{N} a_{i_p,k} \cdot x_k \qquad (1 \leq i \leq M) \tag{5.1}$$

We refer to this as the capped GEMV because it caps the size of matrix `A`, regardless of the size of the output vector `y`.

This capping of memory allocation is shown visually in Figure 5.1 with the shaded area of size `P`×`N` on the top of the matrix depicting the *capped* amount of memory that is needed, and the area with the diagonal pattern in the bottom depicting the amount of memory that is not required. Factoring in the multiple copies of matrix `A` that are needed to facilitate multiple threads and multiple repetitions, as we explained earlier, one can easily see that this modified kernel allows for a much larger vector `y` and thus much higher writing traffic to memory than the unmodified GEMV.

```
1  P = min(N,M);
2  #pragma omp parallel for schedule(static)
3  for ( idx = 0; idx < numThreads; idx++ )
4    for ( i = 0; i < M; i++ ){
5      /* Dot product row of A and x. */
6      sum = 0.0;
7      for ( k = 0; k < N; k++ )
8        sum += A[idx][i%P][k] * x[idx][k];
9      /* Store result in memory. */
10     y[idx][i] = sum;
11   }
```

Listing 5.2: Batched, Capped GEMV.

Listing 5.2 shows the code which executes a batch of *numThreads* independent capped GEMV operations (one per physical core). The purpose of this batched, capped GEMV is to

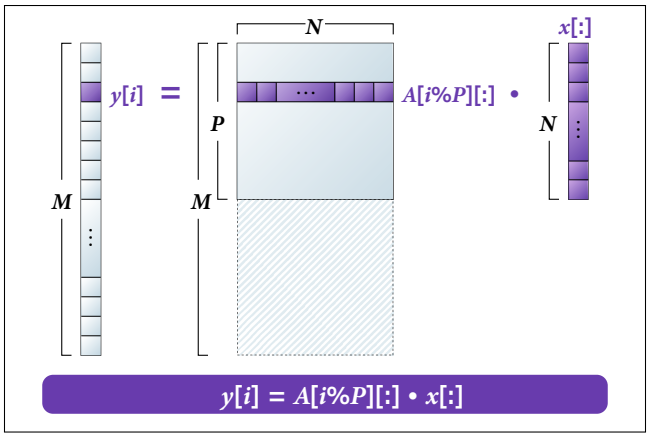


Figure 5.1: Capped GEMV Memory Usage Schematic.

occupy all physical cores with work, without introducing communication between OpenMP threads. This version of the kernel creates *numThreads* times more reading and writing volume than the single-threaded, capped GEMV.

The memory access pattern is the same as the unmodified GEMV, when the size of the capped matrix `A` exceeds the size of the caches. Namely, each thread must read a total of `M`×`N`+`M`+`N` elements and write `M` elements.

5.3.2 GEMM

Suppose we have three matrices $A, B, C \in \mathbb{R}^{N \times N}$, with each element being a double-precision, floating-point number. The GEMM operation is defined as $C = AB$, where the element in row i, column j of C is the dot product of the ith row of A and the jth column of B:

$$c_{ij} = \sum_{k=1}^{N} a_{i,k} b_{k,j} \qquad (1 \leq i, j \leq N) \tag{5.2}$$

```
1  #pragma omp parallel for schedule(static)
2  for ( i = 0; i < N; i++ )
3    for ( j = 0; j < N; j++ ){
4      /* Dot product of row of A and column of B. */
5      sum = 0.0;
6      for ( k = 0; k < N; k++ ) {
7        sum += A[i][k] * B[k][j];
8      }
9      /* Store result in memory. */
10     C[i][j] = sum;
11   }
```

Listing 5.3: Reference GEMM.

Line 7 of Listing 5.3 shows that the whole matrix `B` is accessed by the two inner most loop, since it's only indexed by `k` and `j`. Therefore, if matrix `B` is small enough (as it is in most of our experiments), it will be cached and will not incur traffic from main memory in any but the first iteration of the outer most loop ($i = 0$). Matrix `A` will be accessed one row at a time, and each row will be reused multiple times back-to-back, while it still resides in

cache. Therefore each line of `A` will incur traffic from main memory only once. Similarly to the case of GEMV, the output matrix `C` is not read, but only written by the kernel, but most modern hardware architectures will impose a read operation for each element written into `C`. Therefore, the whole kernel will cause a total of $3 \times N^2$ elements to be read from main memory when the matrices fit in the cache, and higher when they do not.

```
#pragma omp parallel for schedule(static)
for ( idx = 0; idx < numThreads; idx++ )
  for ( i = 0; i < N; i++ )
    for ( j = 0; j < N; j++ ){
      /* Dot product of row of A and column of B. */
      sum = 0.0;
      for ( k = 0; k < N; k++ ) {
        sum += A[idx][i][k] * B[idx][k][j];
      }
      /* Store result in memory. */
      C[idx][i][j] = sum;
    }
```

Listing 5.4: Batched GEMM.

Listing 5.4 shows the code which executes a batch of *numThreads* independent GEMM operations (one per physical core). As in the case with the batched, capped GEMV, the purpose of this code is to load each physical core with work such that there is no communication among the OpenMP threads. So there is *numThreads* times as much reading and writing as there is for the single-threaded, reference GEMM.

5.4 Benchmark Results

In Figure 5.2a, we see the measurements for reading and writing memory traffic taken during the execution of the GEMM benchmark using a single thread running on one core. The dashed lines reflect the expected memory traffic multiplied by 8 (8 bytes per double-precision element) and divided by 64 (each memory read or write occurs in 64-byte chunks, or half cache-line size). The IBM POWER9 architecture has the "capability to fetch only 64 bytes

of data (half cache lines), instead of the normal full cache-line size of 128 bytes of data from the memory" [58].

The shaded, banded region in the figure represents the range of problem sizes for which we expect the measurements to diverge from the expectations. This derives from the fact that the expectations are formulated under the assumption of caching. But when the problem sizes fall within or surpass this shaded region, meaningful caching ceases to occur. The lower bound of this region assumes that all three matrices (A, B, and C) are cached during the GEMM operation. This bound is computed by setting the size of the L3 cache (5MB) equal to the memory consumed by A, B, and C and solving for N:

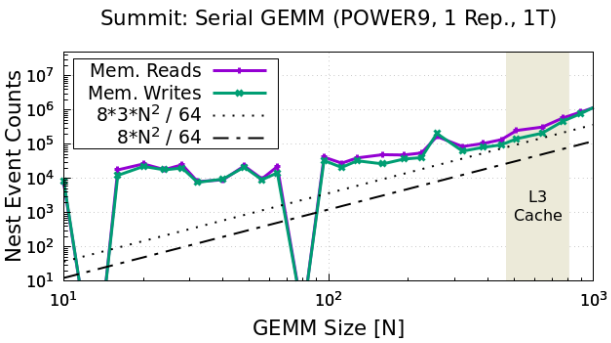
$$8 \times (3 \times N^2) = 5 \times 1024^2 \implies N \approx 467 \tag{5.3}$$

The upper bound of this region assumes that the entirety of only one of A, B, or C is cached during the GEMM operation. This bound is computed by setting the size of the L3 cache equal to the memory consumed by one of the matrices:

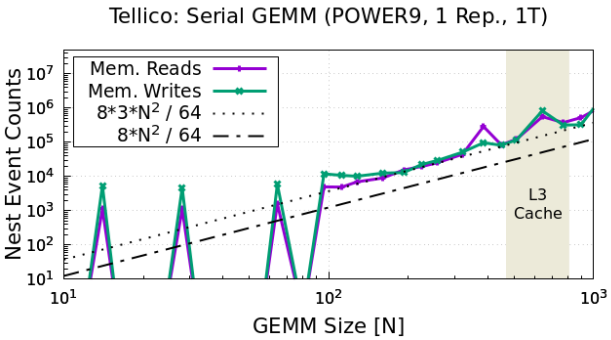
$$8 \times N^2 = 5 \times 1024^2 \implies N \approx 809 \tag{5.4}$$

On Summit, there are 21 usable cores in a socket organized in 11 core pairs, and there is a total of 110 MB of L3 cache. Each core pair is delegated a 10MB cache slice, therefore each core can use up to 5MB of L3 cache without creating contention. When the other cores on the same socket are idle, *their* local L3 cache slices can be re-appropriated by the active core, giving the active core 110 MB worth of cache.

As shown by Figure 5.2a, when measuring the behavior of a single-threaded (serial) GEMM kernel, neither the reading nor writing memory traffic adhere to the expectations. We also observe this to be the case when we repeat the experiment on our testbed, Tellico, a system in which we have privileged access to the Nest counters (see Figure 5.2b). The memory traffic deviates from the expectation whether or not we measure it *via* PCP, and the measurements for small matrices are dominated by noise.

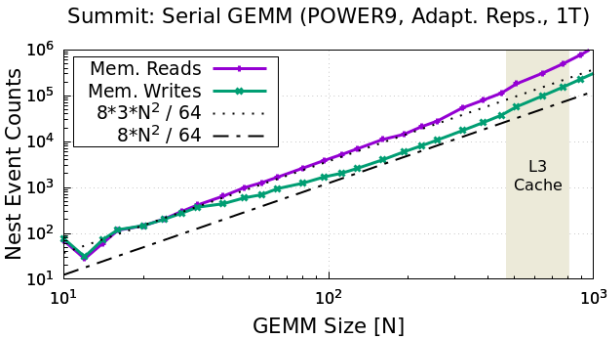


(a) PCP Hardware Events.

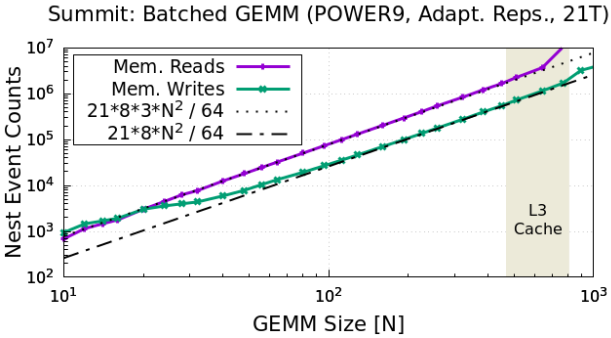


(b) Perf_uncore Hardware Events.

Figure 5.2: Memory Traffic of Single-Threaded GEMM Operation on IBM POWER9 Measured with (a) PCP Events and (b) Perf_uncore Events Using Only 1 Repetition.



(a) Single-Threaded GEMM.



(b) Batched GEMM.

Figure 5.3: Memory Traffic of (a) Single-Threaded GEMM versus (b) Batched GEMM on IBM POWER9 Measured with PCP Events.

To amortize the noisy component of the memory traffic measurements, we can execute multiple GEMM operations and take the average of their aggregate memory traffic [17]. But how many repetitions are necessary?

From inspection of Figures 5.2a and 5.2b, the memory traffic deviates less from the expectation for larger GEMM operations (while they still fit in the cache). This makes intuitive sense because there is more memory traffic for larger problem sizes. Hence, it takes more time for the operation to execute, giving the counters sufficient time to update; whereas, smaller operations execute too quickly for the counters to accurately reflect the hardware activity which took place. It follows that *fewer* repetitions are needed for larger problem sizes.

In Figure 5.3a, we adapt the number of repetitions for a given problem size (N) per Equation 5.5.

$$Repetitions(N) = \begin{cases} \lfloor 514 - 0.246 \times N \rfloor, & N < 2048 \\ 10, & N \geq 2048 \end{cases} \tag{5.5}$$

This is a simple formula for ensuring that we will execute around 500 repetitions for small matrix sizes and linearly drop to 10 repetitions for the largest sizes. These numbers are design decisions based on empirical observations in Figures 5.2a and 5.2b and are not the unique solution of some underlying formula. Other qualitatively similar numbers of repetitions would also work. In Section 5.5, we show that unlike the linear algebra kernels, large 3D-FFT problems do not suffer from the same level of noise and do not require repetitions.

In Figure 5.3a, we observe that when we use this adaptive repetition scheme the average memory traffic measurements exhibit much lower noise and are closer to the expectation. However, the amount of traffic diverges from the expectation as the problem size increases (while the matrices still fit in the cache). We repeat the experiment using a batched GEMM so that each physical core is assigned its own GEMM operation (see Figure 5.3b). In this trial, the work done by each core is independent of that done by every other core, so there is no OpenMP communication. We can observed that in this case the measured traffic matches the expectation very well, as soon as the matrices exceed trivially small sizes and only diverges when the matrices exceed the size of the cache, as expected (see section 5.3.2).

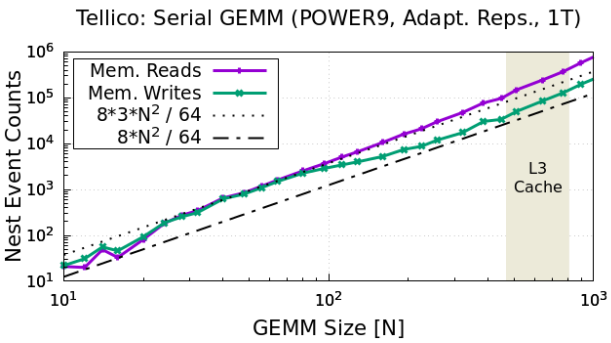
This raises the question of what is causing this extraneous memory traffic when executing a single-threaded kernel. Could this be an artifact of PCP? To ascertain this, we repeated the experiment by measuring Nest hardware events directly on the Tellico testbed, where we have elevated privileges and thus access to perf_uncore events; the results are shown in Figures 5.4a and 5.4b.

As before, there is more memory traffic than expected for larger problem sizes and a gradual deviation from the expectation in the single-threaded execution, despite not using PCP, and this deviation disappears when keeping all cores busy.

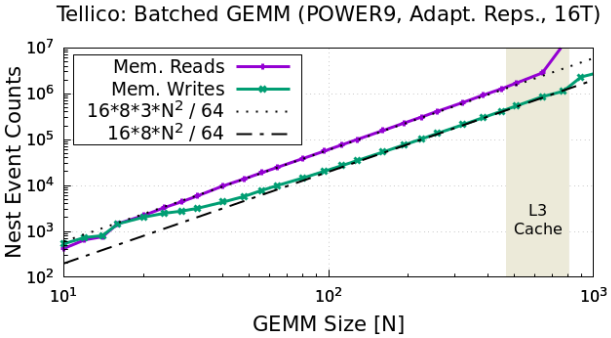
Looking at these results we see that in the single-threaded execution (both on Summit and Tellico) the memory traffic does not jump when the matrix size `N` exceeds the threshold 807 which corresponds to the 5MB cache slice per core. However, when all 21 cores are active there are no other available cache slices that a given core can use, because all of the L3 slices are already in use, and the memory traffic jumps drastically when each of the 21 batched GEMM operations exceed 5MB of data.

In Figure 5.5a, we show the results of running the capped GEMV kernel. For small sizes we use a square matrix `A` (*i.e.*, `M=N=P`) and therefore the kernel performs a regular GEMV operation. When the size `M` reaches the point where the matrix exceeds the size of the L3 cache, then we fix the width (`N`) of the matrix, and proceed with the capped GEMV, where `P=N` and only the size `M` of the vector `y` grows. Since each thread has access to 5MB of L3 cache, this transition happens when `M=N=P=1280`.

Looking at the figure one can see that the amount of reading from memory perfectly matches our expectations. Specifically, when `M`<1280 the reading traffic matches what is expected for a square GEMV ($\texttt{M}^2 + 2\times\texttt{M}$), and for larger sizes the reading traffic matches what is expected for a capped GEMV (`M`×`N`+`M`+`N`). However, there is more writing traffic than expected, and we do not observe steady memory writing until N exceeds 10^4. In an effort to investigate whether this behavior is due to PCP, we repeated the experiment using our Tellico tested (and perf_uncore events). The results of this experiment, shown in Figure 5.5b, show that there is extraneous memory writing traffic in this environment as well. We also reproduced this behavior on an Intel Skylake architecture using perf_uncore, although we do not include this graph due to space limitations. Thus, this is neither a

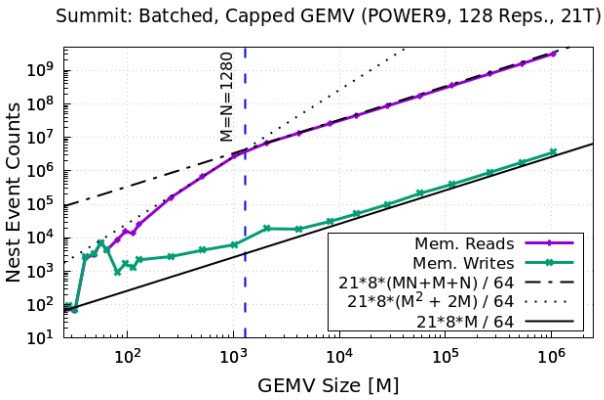


(a) Single-Threaded GEMM.

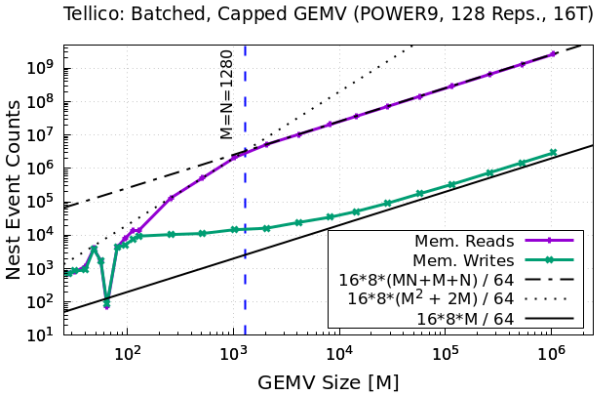


(b) Batched GEMM.

Figure 5.4: Memory Traffic of (a) Single-Threaded GEMM versus (b) Batched GEMM on IBM POWER9 Measured with Perf_uncore Events.



(a) PCP Hardware Events.



(b) Perf_uncore Hardware Events.

Figure 5.5: Memory Traffic of Batched, Capped GEMV on POWER9 Measured with (a) PCP Events versus (b) Perf_uncore Events.

PCP-related nor POWER9-specific phenomenon. Figures 5.5a and 5.5b reinforce the need to have large-enough problem sizes which consume sufficient time to attain stable, low-noise measurements; that such a phenomenon is not PCP-specific; and measurements taken *via* PCP are as accurate as those taken directly from the Nest counters.

These experiments demonstrate that in order to make meaningful measurements of memory traffic, there needs to be enough of it taking place, regardless of the measuring infrastructure, or the target architecture. Quantifying this effect, as we have done here, is one of the contributions of this chapter and has implications for application developers who wish to understand the behavior of their memory-bound codes.

5.5 Application: 3D-FFT

Utilizing what we have learned about data movement between processing cores and main memory as well as the Performance Co-Pilot, we now evaluate the fidelity of the PCP component's capabilities to monitor memory traffic of production applications (and without using multiple kernel repetitions) on Summit. We perform a case study on select data re-sorting routines in a distributed 3D-FFT [60, 76] executed. The 3D-FFT is a workhorse kernel utilized by various applications, such as HACC [50], GESTS [91], and QMCPACK [63], among others. This implementation utilizes both CPU and GPU cores in addition to network communication. The 3D-FFT is defined as

$$\tilde{A}_{uvw} = \sum_{x=0}^{N-1}\sum_{y=0}^{N-1}\sum_{z=0}^{N-1} A_{xyz} exp(\text{-}2\pi i\frac{wz}{N}) exp(\text{-}2\pi i\frac{vy}{N}) exp(\text{-}2\pi i\frac{ux}{N})$$

$$(0 \leq u, v, w \leq N-1) \tag{5.6}$$

where $A, \tilde{A} \in \mathbb{C}^{N\times N\times N}$ are three-dimensional arrays containing complex double-precision, floating-point numbers.

This 3D-FFT implementation decomposes the input data array A into a two-dimensional $r \times c$ virtual processor grid with each element in the grid corresponding to a distinct MPI rank. This means the data array local to a single MPI rank is of size $\frac{N}{r} \times \frac{N}{c} \times N$. Each MPI

rank is assigned to a socket (two per compute node) on Summit. Since each socket has its own Nest, we measure PCP events per MPI rank. The re-sorting routines are as follows:

- store_1st_colwise_forward (S1CF)
- store_2nd_colwise_forward (S2CF)
- store_1st_planewise_forward (S1PF)
- store_2nd_planewise_forward (S2PF)

The structure and performance of S1PF and S2PF are similar to those of S1CF and S2CF, respectively, so we only show the results of S1CF and S2CF.

Figures 5.6-5.9 illustrate the range between the minimum and maximum measurements (of 50 runs) using a 2-by-4 virtual processor grid. Pursuant to organically measuring a production application job, we do not use the average of multiple repetitions as we did for the BLAS benchmarks. Since there is relatively little measurement variation between runs for large problems, only one run would be necessary.

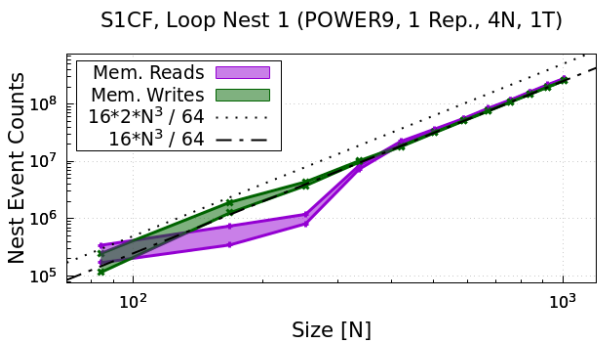
5.5.1 S1CF

```
1 #pragma omp parallel for schedule(static)
2 for ( plane = 0; plane < PLANES; plane++ )
3   for ( row = 0; row < ROWS; row++ )
4     for ( col = 0; col < COLS; col++ )
5       tmp[plane][row][col]
6       = in[plane*ROWS*COLS + row*COLS + col];
```

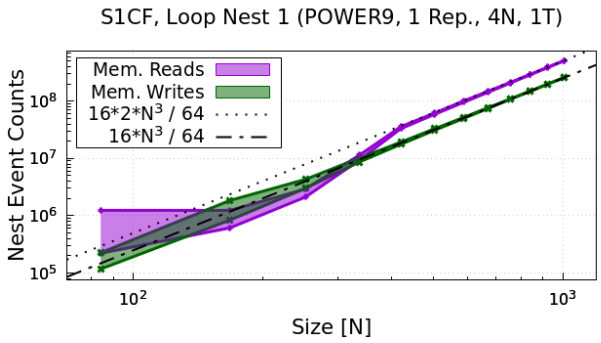
Listing 5.5: S1CF, Loop Nest 1

This first loop nest in S1CF copies the contents of the 1D array `in` into the 3D array `tmp`. Since this routine is executed once per MPI rank, `in` and `tmp` each contain $\frac{N}{r} \times \frac{N}{c} \times N =$ `PLANES`$\times$`ROWS`$\times$`COLS` *double complex* elements, each of which are 16 bytes. Aggregating across all MPI ranks results in N^3 elements copied from the `in` arrays to the `tmp` arrays.

As we discussed in Section 5.4, the event measurements from the DGEMM shown in Figure 5.3b corroborate that a write to memory automatically incurs a read from memory, as we observe a read for not only matrices `A` and `B`, but also `C`, even though `C` seemingly only needs to be written. Therefore, we expect to observe two reads (one for each of `in` and `tmp`) but only one write (for `tmp`). In Figure 5.6a, we indeed observe one write; however, we only observe one read instead of two. In the IBM POWER9 architecture, "hardware may detect Stride-N streams in intervals when they access elements that map to sequential

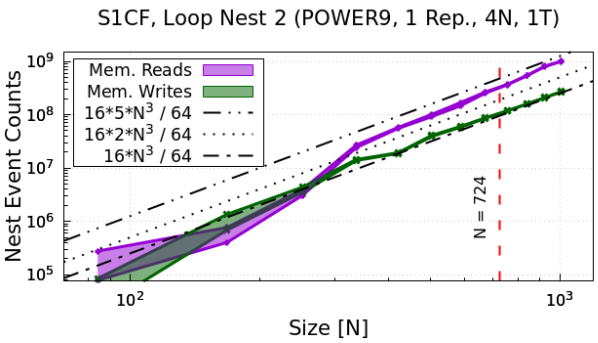


(a) S1CF, Loop Nest 1 with No Additional Compiler Optimizations.

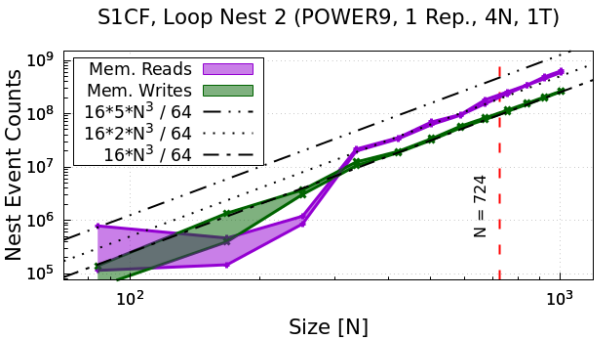


(b) S1CF, Loop Nest 1 with Compiler Optimization *-fprefetch-loop-arrays*.

Figure 5.6: Memory Traffic of Loop Nest 1 in S1CF.



(a) S1CF, Loop Nest 2 with No Additional Compiler Optimizations.



(b) S1CF, Loop Nest 2 with Compiler Optimization *-fprefetch-loop-arrays*.

Figure 5.7: Memory Traffic of Loop Nest 2 in S1CF.

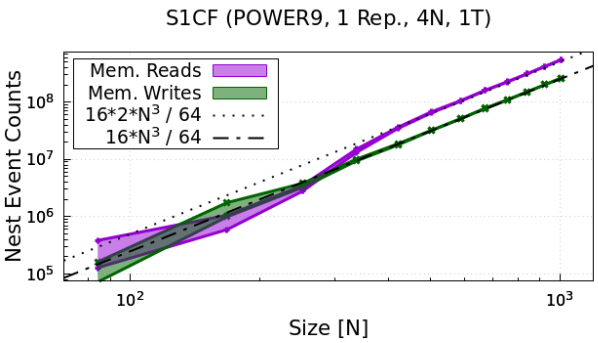
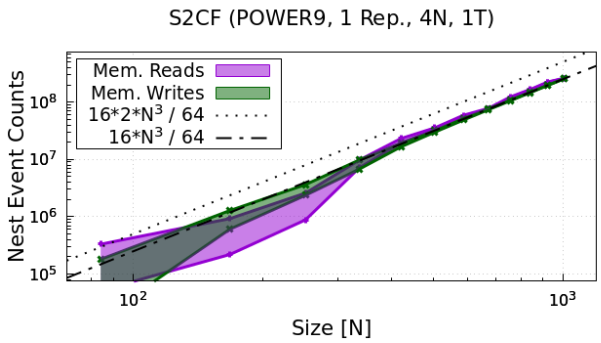
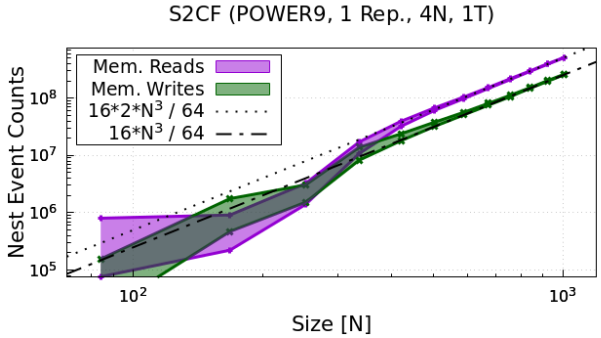


Figure 5.8: S1CF with No Additional Compiler Optimizations.



(a) S2CF with No Additional Compiler Optimizations.



(b) S2CF with Compiler Optimization *-fprefetch-loop-arrays*.

Figure 5.9: Memory Traffic of S2CF.

cache blocks" [IBM Corporation]. In the case of the DGEMM, there was indeed a strided access—that of the `B` matrix—for which the hardware has detected a strided data stream. However, in S1CF, the hardware does not detect a strided data stream because `in` and `tmp` are accessed sequentially (*i.e.*, there is no stride). In the presence of a strided data stream, the writes to variables will not bypass the cache, so they will be read by the cache. In the absence of such a stream, the writes indeed bypass the cache.

We can prevent cache-avoidant writes to memory by compiling the application using the *-fprefetch-loop-arrays* flag with GCC. This flag provides the two assembly instructions shown in Lines 2 and 3 in Listing 5.6.

```
1 tmp[plane][row][col]
2  404:    2c 4a 00 7c      dcbt    0,r9
3  408:    ec 41 00 7c      dcbtst  0,r8
4 = in[plane*ROWS*COLS + row*COLS + col];
```

Listing 5.6: S1CF, Loop Nest 1 Assembly

The **dcbtst** instruction enables software prefetching by "[causing] a single-line prefetch into the L3 cache" [58]. This forces `tmp` to be read into the cache from memory. In Figure 5.6b, we observe the effect of this assembly instruction, as there are now two reads: one for `in` and also for `tmp`.

```
1 #pragma omp parallel for schedule(static)
2 for ( col = 0; col < COLS; col++ )
3   for ( plane = 0; plane < PLANES; plane++ )
4     for ( row = 0; row < ROWS; row++ )
5       out[col*PLANES*ROWS + plane*ROWS + row]
6       = tmp[plane][row][col];
```

Listing 5.7: S1CF, Loop Nest 2

This second loop nest copies the contents from the 3D array `tmp` into the 1D array `out`. Accounting for all MPI ranks, N^3 elements are copied from the `tmp` arrays to the `out` arrays.

Figure 5.7a shows that we measure the expected one write per iteration of the innermost loop shown in Listing 5.7, but we measure more than the expected reads. Listing 5.7 shows that `tmp` is accessed in strides. The dimensions of `tmp` are ordered as `PLANES` by `ROWS` by

`COLS`, but the loop nest traverses `tmp` in order of `COLS` by `PLANES` by `ROWS`. Due to this asymmetry in how `tmp` is arranged in memory versus how it is traversed, a stride is present. As N increases, this stride quickly exceeds the size of a cache line. This means only a single element per cache line's worth of contiguous elements in `tmp` is usable—without an additional read from memory—unless it can be cached.

When reading an element from `tmp`, an entire cache line is read. Since `tmp` is accessed in strides of `PLANES`×`ROWS`, the next element in the cache line is not used until another `PLANES`×`ROWS` elements (and their corresponding cache lines' worth of data) have been read. Thus, $4 \times \frac{16N^2}{8}$ bytes are read before the next contiguous element in `tmp` is used (division by 8 due to there being 8 processes).

Due to the strided access pattern, before the next contiguous element from `tmp` is read, an additional `PLANES`×`ROWS` elements are read from `out` (since each write to `out` incurs a read in the presence of a stride). These elements' corresponding cache lines' worth of data are read, but since `out` is accessed sequentially, each element in a cache line is immediately used. This means `out` does not consume more of the cache space beyond its `PLANES`×`ROWS` elements. This means that an additional $\frac{16N^2}{8}$ bytes of data occupy the cache before the two sequential reads of `tmp` occur.

The size of the L3 cache is exceeded under the condition:

$$4 \times \frac{16 \times N^2}{8} + \frac{16 \times N^2}{8} = 5 \times 1024^2 \implies N \approx 724 \tag{5.7}$$

For $N > 724$, an entire cache line must be read per iteration of the innermost loop in Listing 5.7. Since a cache line is 64 bytes and a *double complex* element is 16 bytes, then $\frac{64}{16} = 4$ times as many reads must occur in order to supply all of `tmp`. And since there is one read per write to `out`, then we expect to observe up to 5 reads per iteration of the innermost loop in Listing 5.7 when $N > 724$.

When we re-compile the second loop nest using the *-fprefetch-loop-arrays* compiler flag, there is a significant improvement in performance due to more effective prefetching, as shown in Figure 5.7b.

```
#pragma omp parallel for schedule(static)
for ( plane = 0; plane < PLANES; plane++ )
  for ( row = 0; row < ROWS; row++ )
    for ( col = 0; col < COLS; col++ )
      out[col*PLANES*ROWS + plane*ROWS + row]
      = in[plane*ROWS*COLS + row*COLS + col];
```

Listing 5.8: S1CF, Combined Loop Nest

Listing 5.8 shows S1CF written as a single loop nest. This new loop nest accesses `out` in strides but `in` sequentially. We expect there to be one write per innermost loop iteration and two reads: one from `in` and—due to a strided access pattern—one from `out`, which is significantly less reading than we observed in the original S1CF. This is precisely what we observe in Figure 5.8.

5.5.2 S2CF

```
/* c = cols in virtual proc. grid. */
X = COLS/c;
Y = c;
#pragma omp parallel for schedule(static)
for( plane = 0; plane < PLANES; plane++ )
  for( x = 0; x < X; x++ )
    for( y = 0; y < Y; y++ )
      for( row = 0; row < ROWS; row++ )
  out[plane*X*Y*ROWS + x*Y*ROWS + y*ROWS + row]
  = in[y*PLANES*X*ROWS + plane*X*ROWS + x*ROWS + row];
```

Listing 5.9: S2CF

The loop nest in Listing 5.9 copies the contents from the 1D array `in` into the 1D array `out`. Accounting for all MPI ranks, N^3 elements are copied from the `in` arrays to the `out` arrays.

In Figure 5.9a, there are as many reads and writes as we expect to observe. Since there is not a stride present, the stores bypass the cache, as observed in the case of the S1CF routine. Strictly speaking, S2CF is not completely stride-free because the dimensions of `in` are ordered `Y` by `PLANES` by `X` by `ROWS`, but `in` is traversed `PLANES` by `X` by `Y` by `ROWS`. But since the innermost dimension of the traversal matches the innermost dimension of the ordering of `in`, the effect of the stride is amortized.

For a larger-scale job, of which the results are shown in Figure 5.10, we use 16 compute nodes on a 4-by-8 virtual processor grid to perform computations on the problem sizes $N = \{1344,\ 2016\}$. We do not use the *-fprefetch-loop-arrays* compiler flag for this job. We expect two reads per write in S1CF and one read per write in S2CF.

5.5.3 Acquiring a Broad View of Application Behavior

In the study presented below, we use PAPI to simultaneously monitor three disparate performance metrics—GPU power, network traffic, and memory traffic—of a GPU-enabled application running on a distributed memory machine. The application is a modified version of the 3D-FFT code we used before, adapted to utilize the GPUs for the 1D-FFT operations. For our experiment we use 32 compute nodes and a 8-by-8 virtual processor grid. GPU power and network traffic are measured using the events in Table 5.2. The performance profile for this experiment is shown in Figure 5.11.

This experiment illustrates the inner workings of the different phases of the 3D-FFT, which utilize different categories of hardware. The 1D-FFT phases entail host memory getting copied to the GPU–a large amount of host memory being read; the batch of 1D-FFT's executed–a spike in GPU power; and the results getting copied back to the host–a large amount of host memory being written to. By cross-referencing the memory traffic from the PAPI PCP component with the GPU power available *via* the PAPI NVML component, we observe this progression: a GPU power spike occurs precisely between the transition from a high volume of host memory reading to a relatively high volume of memory writing. Furthermore, we observe approximately twice as much memory reading as we do memory writing during the first and third data re-sorting phases. This agrees with the previous observations of strided memory accesses incurring a read for every write.

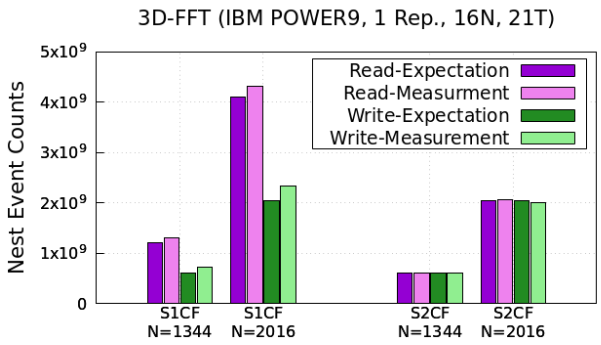


Figure 5.10: Performance of **S1CF** and **S2CF**.

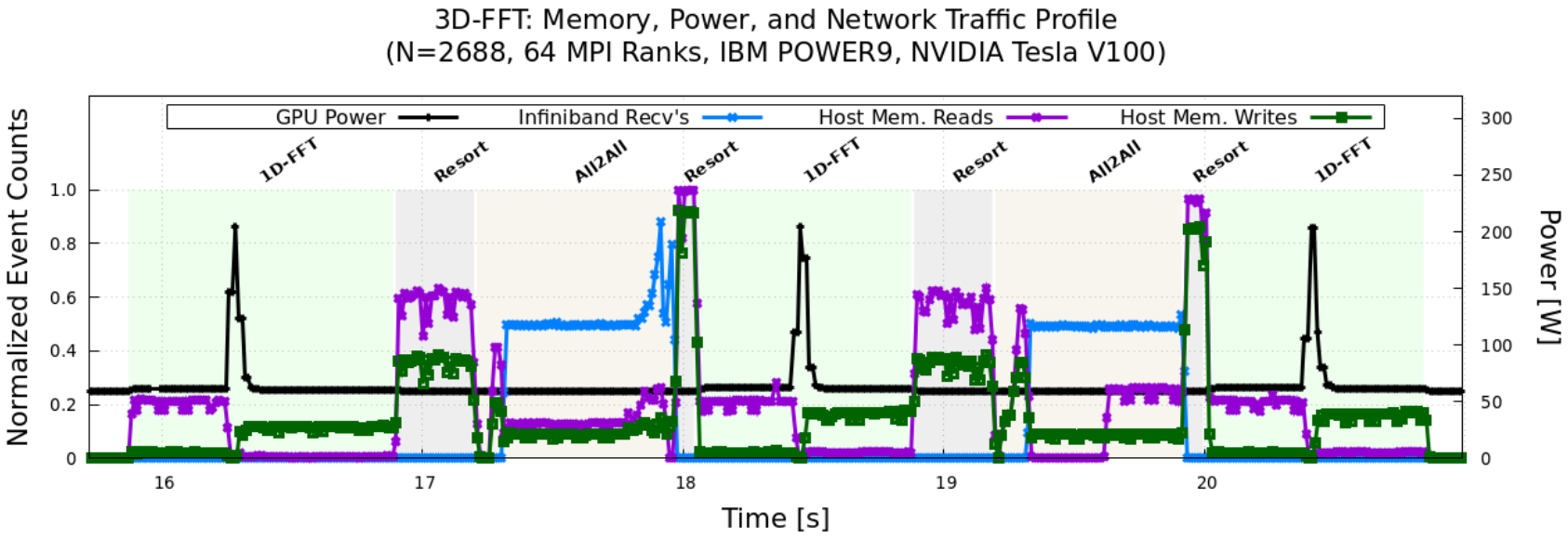


Figure 5.11: Performance Profile of a Single 3D-FFT Rank.

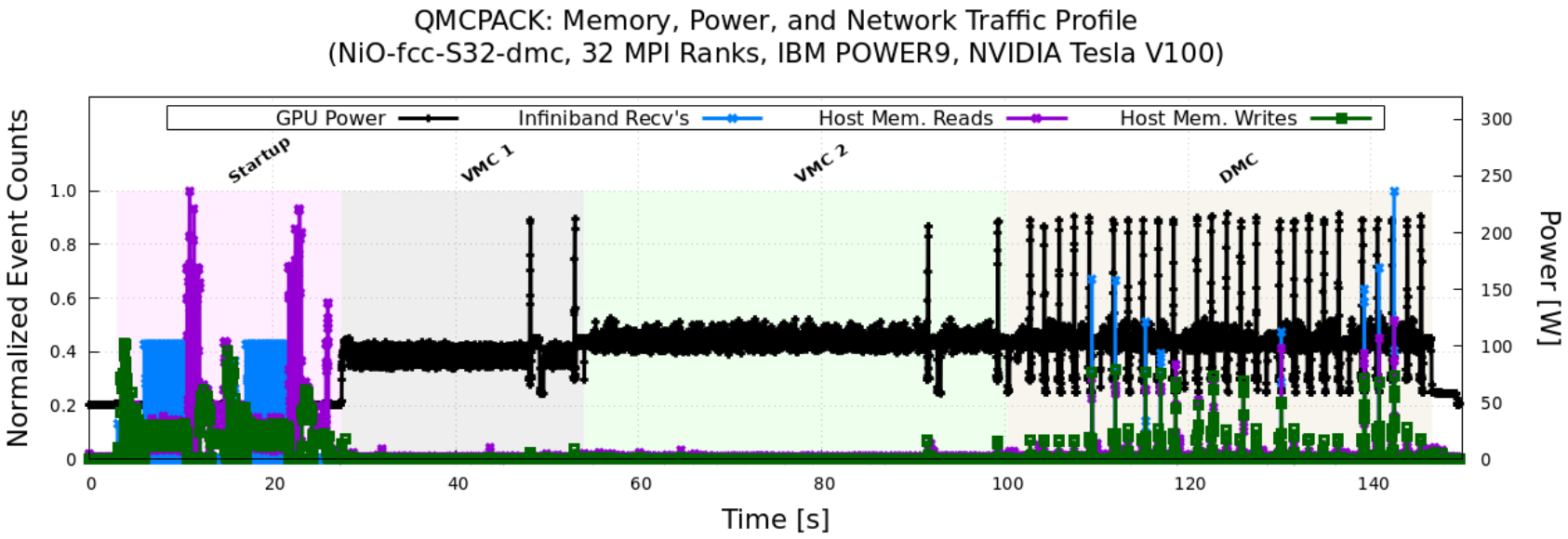


Figure 5.12: Performance Profile of a Single QMCPACK Rank.

Table 5.2: Supplemental Hardware Events

Hardware	**PAPI Component**	**Hardware Event**
NVIDIA Tesla V100 GPU	nvml	nvml:::Tesla_V100−SXM2−16GB: device_0:power
Mellanox ConnectX-5 Ex	infiniband	infiniband ::: mlx5_[0|1]_1_ext: port_recv_data

During the second and fourth re-sorting phases, we observe approximately equal amounts of memory reading and writing. This is once again due to the innermost dimension of the data traversal matching the innermost dimension of the ordering of data array, which effectively nullifies the effect the stride that is occasionally present. These two re-sorting phases also realize higher bandwidth due to better locality in their access patterns. We observe a jump in network activity as measured *via* the PAPI Infiniband component during the two "All2All" phases. Using only hardware events exposed *via* the various components of PAPI, we are able to uniquely identify each region of the 3D-FFT's execution profile. QMCPACK is a hybrid application which implements various Quantum Monte Carlo (QMC) algorithms to solve the Schrödinger equation. For more information, please refer to J. Kim *et al.* [63]. The example problem [NVIDIA Corporation] used in our QMCPACK experiment (on Summit) executes the Variational Monte Carlo (VMC) method with no drift, then the VMC method with drift, and finally, a Diffusion Monte Carlo (DMC) method. Figure 5.12 demonstrates that the different stages in the execution of QMCPACK are distinguishable by monitoring separate hardware components simultaneously. Figure 5.11 is qualitatively equivalent to Figure 5.12, reinforcing the objective of PAPI's multi-component monitoring capabilities for more fine-grained insights into applications' performance running on heterogeneous HPC systems.

5.6 Conclusions and Future Work

In this chapter, we have shown several examples of measuring memory traffic using the Performance Co-Pilot. This technology is particularly useful on systems in which users cannot securely be granted elevated privileges. In such an environment, measuring application performance remains crucial.

The first major takeaway from the experimental results outlined in this chapter is that adapting the number of successive executions of performance-critical kernels serves a technique to accurately measure memory traffic. By measuring repeated executions of a computational kernel, the noisy memory traffic from other processes gets amortized. Doing adaptively fewer repetitions for larger problem sizes saves both memory and execution time.

Unfortunately, our results also highlight the fact that measuring the memory traffic of small kernels that do not naturally repeat leads to measurements fraught with noise, regardless of the measuring infrastructure or architecture. Additionally, we observed that memory traffic measurements from the PAPI PCP component are as accurate as those measured directly from the perf_uncore counters.

Memory traffic measurements taken using PCP are sensitive to micro-architectural details, such as localized L3 cache slices. It is useful for a performance-conscious programmer to account for such peculiarities by executing a batch of kernels. In the case of IBM POWER9, this was done by executing an operation on each available CPU core.

Lastly, we generated profiles of the GPU power, memory traffic, and network traffic for two distributed applications: the 3D-FFT and QMCPACK. The 3D-FFT case study also revealed nuanced architectural behavior, such as writing to memory while bypassing the cache, as well as the GCC compiler's capability to toggle this feature off. This does show that the programmer must be conscious of hardware details in order to thoroughly interpret counter results.

PAPI provides a homogeneous interface to access different counters, allowing programmers to simultaneously monitor multiple, orthogonal performance metrics of interest *via* a single API. Without this tool, the programmer would be tasked with instrumenting application code with each performance back-end individually. PAPI is particularly useful for such hybrid applications as the 3D-FFT and QMCPACK, which utilize multiple types of processors and other hardware components.

The main focus for future work is to extend these techniques to accurately measure memory traffic for other BLAS operations in upcoming IBM systems (*e.g.*, POWER10), as well as other forthcoming architectures. It also will be important to develop programming techniques to accurately measure other categories of inter-core events than those solely related to memory traffic.

Chapter 6

Conclusions and Future Work

6.1 Disclosure

This chapter uses content from three of my published papers [17], [18], and [16] (© 2021 Springer, 2023-2024 IEEE as appropriate):

- Barry, D., Danalis, A., and Jagode, H. (2021). **"Effortless Monitoring of Arithmetic Intensity with PAPI's Counter Analysis Toolkit."** In *Tools for High Performance Computing 2018 / 2019*, pp. 195-218, Cham. Springer International Publishing, https://doi.org/10.1007/978-3-030-66057-4_11.

- Barry, D., Jagode, H., Danalis, A., and Dongarra, J. (2023). **"Memory Traffic and Complete Application Profiling with PAPI Multi-Component Measurements."** In *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 393-402, https://doi.org/10.1109/IPDPSW59300.2023.00070.

- Barry, D., Danalis, A., and Dongarra, J. (2024). **"Automated Data Analysis for Defining Performance Metrics from Raw Hardware Events."** In *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 716-725, https://doi.org/10.1109/IPDPSW63119.2024.00134.

I am the primary author of these papers. Coauthors for the published papers include Heike Jagode, Anthony Danalis, and Jack Dongarra. Reused coauthor contributions are limited to aid in the development of the branching and instruction cache benchmarks, the ideas of problem repetitions (shown in Section 3.5.1), rounding (shown in Section 4.6), and the capped GEMV (shown in Section 5.3.1), textual improvements, and high-level guidance.

6.2 Conclusions

Performance analysis serves as a crucial tool for understanding and improving the execution of applications on HPC systems. Monitoring hardware events, which track low-level hardware operations, and aggregating their occurrences provides us with some of the most useful metrics for performance analysis.

However, there are challenges with defining performance metrics using hardware events. One challenge is determining which hardware events (among those present on the target architecture) comprise a given performance metric. This issue is compounded by the vast numbers of hardware events in modern architectures. Even if the specific hardware events needed to define a performance metric are known, then there persists the secondary challenge of deducing how to properly scale and combine them. Furthermore, from one architecture to another, the specific hardware-event combinations tend to vary greatly. The research efforts in this dissertation mitigate these issues.

Chapter 3 outlined the Counter Analysis Toolkit and showed how it can be used to identify which hardware events belong to which hardware attributes (*e.g.*, data caches, instruction caches, branching units, and floating-point units), as well as which hardware events are best suited for measuring traffic to main memory. Chapter 3 also demonstrates that the arithmetic intensity of three important BLAS operations (DOT, GEMV, GEMM) can be computed on three different architectures (Intel Broadwell, Intel Skylake, IBM POWER9) *via* measuring hardware events and explained how PAPI's Performance Co-Pilot (PCP) component allows users to monitor memory traffic in the absence of elevated privileges. Finally, this chapter provides a study on the reliability of the PCP measurements

and explained how the noise and overhead in the measurements can be mitigated, even for small kernels that do not perform enough operations to amortize the noise on their own.

Chapter 4 introduced a mathematical analysis to automatically parse through thousands of hardware events and identify those most pertinent to specific hardware attributes. This analysis operates on the data output from the CAT benchmarks run on the Intel Sapphire Rapids CPU and the AMD MI250X GPU architectures. This chapter demonstrates that this analysis is applicable to CPU branching units, memory subsystem, and both CPU and GPU floating-point units.

The high noise levels in hardware events measurements are addressed by collecting multiple measurements for the same event and quantifying the run-to-run variability among the set of measurements using the maximum RNMSE. Hardware events with a high variability are disregarded for further analysis. Additionally, the specialized QR factorization scheme allows us to account for low levels of measurement noise, which is often present in practical hardware event monitoring.

Chapter 4 also established a scheme for pivoting in the QR factorization that chooses the hardware events that most strongly correlate to the hardware attributes exercised by the CAT benchmarks. After selecting the best linearly independent hardware events, least squares provides *correct and meaningful* linear combinations of hardware events for the desired performance metrics. The least squares error indicates how well the linear combination approximates the performance metric. This is useful because it indicates when performance metric cannot be defined using hardware events on a given architecture.

Chapter 5 showed several examples of measuring memory traffic accurately using the Performance Co-Pilot. *This is useful because it quantifies the amount of memory traffic needed in order for the measurements to be accurate.* The insights gained from these studies are critical for systems in which users do not have the elevated privileges needed to monitor memory traffic *via* conventional methods.

The experiments in Chapter 4 show how performance analysts can amortize noise to more accurately measure memory traffic by adapting the number of successive executions of performance-critical kernels. Furthermore, using adaptively *fewer* repetitions for larger problem sizes saves both memory and execution time. Regardless of the measuring

infrastructure, these results reiterate that memory traffic measurements of small kernels that execute quickly are susceptible to high levels of measurement noise. Significantly, the memory traffic measurements from PCP are as accurate as those measured using the perf_uncore counters directly.

Chapter 5 shows generated profiles of the GPU power, memory traffic, and network traffic for two distributed applications: the 3D-FFT and QMCPACK. The 3D-FFT case study also revealed nuanced architectural behavior, such as writing to memory while bypassing the cache, as well as the GCC compiler's capability to toggle this feature off.

The studies in Chapter 5 also serve as a framework for validating the functionality and accuracy of the various components provided by PAPI, which provides a homogeneous interface to access the counters across the heterogeneous hardware components utilized by HPC applications.

6.3 Future Work

6.3.1 Accelerating Benchmarking

For the purposes of more time-efficient benchmarking, future work will entail exploiting the ability of the hardware counters to monitor multiple events simultaneously. This contrasts with the current strategy (Chapter 3), wherein the benchmarks monitor one event per kernel execution. This poses a challenge because different hardware events occupy varying numbers of registers. However, minimizing the number of kernel executions would lower the execution time. Another strategy to accelerate the benchmarking process would be to monitor hardware events in the manner of one-per-core, using all available compute cores simultaneously. This would require knowing *a priori* which events are strictly related to compute core resources (as opposed to shared resources, such as caches), which are not influenced by activity on other cores. Future work also will focus on benchmarking for the hardware subsystems in GPUs that are shared among the compute cores. This extends the work in this dissertation that focuses on benchmarking the shared resources within CPUs, in addition to both the CPU and GPU compute cores.

6.3.2 Refining Automated Analysis

Future work pertaining to the automated analysis of hardware event data (from Chapter 4) will entail methods to develop different measures to quantify hardware event noise and more rigorously select noise suppression thresholds. Exploring alternative pivoting criteria for the specialized QR factorization scheme could also prove beneficial. Another useful direction would be to develop methods for automatically defining performance metrics that are non-linear combinations of hardware events (*e.g.*, arithmetic intensity).

6.3.3 Expanding Inter-core Measurement Techniques

The main focus for the future work in benchmarking methods for the inter-core events is to extend the techniques from Chapter 5 to accurately measure memory traffic for other BLAS operations in upcoming systems, which employ event-monitoring infrastructure similar to the Performance Co-Pilot, as well as other forthcoming infrastructures. It also will be important to develop programming techniques to accurately measure other categories of inter-core events than those solely related to memory traffic.

Bibliography

[1] Abdulah, S., Cao, Q., Pei, Y., Bosilca, G., Dongarra, J., Genton, M. G., Keyes, D. E., Ltaief, H., and Sun, Y. (2021). Accelerating geostatistical modeling and prediction with mixed-precision computations: A high-productivity approach with parsec. *IEEE Transactions on Parallel and Distributed Systems*, 33(4):964–976. 12

[2] Abel, A. and Reineke, J. (2020). nanobench: A low-overhead tool for running microbenchmarks on x86 systems. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 34–46, Los Alamitos, CA, USA. IEEE Computer Society. 15

[3] Advanced Micro Devices, Inc. MI200 Performance Counters and Metrics. https://rocm.docs.amd.com/en/docs-5.7.1/understand/gpu_arch/mi200_performance_counters.html. 5, 10, 56

[4] Advanced Micro Devices, Inc. Preliminary Processor Programming Reference (PPR) for AMD Family 17h Model 31h, Revision B0 Processors. https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/55803-ppr-family-17h-model-31h-b0-processors.pdf. 5

[5] Advanced Micro Devices, Inc. Preliminary Processor Programming Reference (PPR) for AMD Family 19h Model 11h, Revision B1 Processors Volume 1 of 6. https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/55901_B1_pub_053.zip. 5

[6] Advanced Micro Devices, Inc. Preliminary Processor Programming Reference (PPR) for AMD Family 19h Model 11h, Revision B1 Processors Volume 3 of 6. https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/55901_B1_pub_053.zip. 5

[7] Advanced Micro Devices, Inc (2023). System Management Interface API Guide v5.3. https://docs.amd.com/bundle/ROCm-SMI-v5.3/page/index.html. 9

[8] Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Ltaief, H., Luszczek, P., and Tomov, S. (2009). Numerical linear algebra on emerging architectures:

The plasma and magma projects. In *Journal of Physics: Conference Series*, volume 180, page 012037. IOP Publishing. 5

[9] Alappat, C. L., Hofmann, J., Hager, G., Fehske, H., Bishop, A. R., and Wellein, G. (2020). Understanding hpc benchmark performance on intel broadwell and cascade lake processors. In *High Performance Computing: 35th International Conference, ISC High Performance 2020, Frankfurt/Main, Germany, June 22–25, 2020, Proceedings 35*, pages 412–433. Springer. 12

[10] ALCF (2023a). Aurora fact sheet. https://www.alcf.anl.gov/sites/default/files/2022-06/ALCF-Aurora_0.pdf. 5, 56

[11] ALCF (2023b). Getting Started on Aurora. https://docs.alcf.anl.gov/aurora/hardware-overview/machine-overview/. 58

[12] Anzt, H., Cojean, T., Flegar, G., Göbel, F., Grützmacher, T., Nayak, P., Ribizel, T., Tsai, Y. M., and Quintana-Ortí, E. S. (2022). Ginkgo: A Modern Linear Operator Algebra Framework for High Performance Computing. *ACM Transactions on Mathematical Software*, 48(1):2:1–2:33. 5

[13] Atchley, S., Zimmer, C., Lange, J., Bernholdt, D., Melesse Vergara, V., Beck, T., Brim, M., Budiardja, R., Chandrasekaran, S., Eisenbach, M., Evans, T., Ezell, M., Frontiere, N., Georgiadou, A., Glenski, J., Grete, P., Hamilton, S., Holmen, J., Huebl, A., Jacobson, D., Joubert, W., Mcmahon, K., Merzari, E., Moore, S., Myers, A., Nichols, S., Oral, S., Papatheodore, T., Perez, D., Rogers, D. M., Schneider, E., Vay, J.-L., and Yeung, P. K. (2023). Frontier: Exploring exascale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '23, New York, NY, USA. Association for Computing Machinery. 1, 5, 56, 58

[14] Austin, B. and Wright, N. J. (2014). Measurement and interpretation of micro-benchmark and application energy use on the cray xc30. In *2014 Energy Efficient Supercomputing Workshop*, pages 51–59. IEEE. 12

[15] Bailey, D., Harris, T., Saphir, W., Van Der Wijngaart, R., Woo, A., and Yarrow, M. (1995). The nas parallel benchmarks 2.0. Technical report, Technical Report NAS-95-020, NASA Ames Research Center. 11

[16] Barry, D., Danalis, A., and Dongarra, J. (2024). Automated data analysis for defining performance metrics from raw hardware events. In *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1, 8, 55, 122

[17] Barry, D., Danalis, A., and Jagode, H. (2021). Effortless monitoring of arithmetic intensity with papi's counter analysis toolkit. In *Tools for High Performance Computing 2018 / 2019*, pages 195–218, Cham. Springer International Publishing. 17, 57, 93, 102, 122

[18] Barry, D., Jagode, H., Danalis, A., and Dongarra, J. (2023). Memory traffic and complete application profiling with papi multi-component measurements. In *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 393–402. 88, 122

[19] Bischof, C. H. and Quintana-Ortí, G. (1998). Computing rank-revealing qr factorizations of dense matrices. *ACM Trans. Math. Softw.*, 24(2):226–253. 14

[20] Björck, Å. (1996). *Numerical Methods for Least Squares Problems*. Society for Industrial and Applied Mathematics. 13

[21] Bodor, A., Csabai, I., Mahoney, M. W., and Solymosi, N. (2012). rcur: an r package for cur matrix decomposition. *BMC bioinformatics*, 13:1–6. 14

[22] Boehme, D., Gamblin, T., Beckingsale, D., Bremer, P.-T., Gimenez, A., LeGendre, M., Pearce, O., and Schulz, M. (2016). Caliper: Performance introspection for hpc software stacks. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 550–560. 9, 57, 90

[23] Brown, L., Bennett, P. M., Cowan, M., Leach, C., and Oppe, T. C. (2010). Using application benchmark results from ti-10 to find appropriate hpcmp architectures. In

2010 DoD High Performance Computing Modernization Program Users Group Conference, pages 463–471. IEEE. 11

[24] Brunst, H. and Knöpfer, A. (2011). Vampir. In Padua, D., editor, *Encyclopedia of Parallel Computing*, pages 2125–2129. Springer US. 9, 57, 90

[25] Cao, Q., Pei, Y., Akbudak, K., Bosilca, G., Ltaief, H., Keyes, D., and Dongarra, J. (2021). Leveraging parsec runtime support to tackle challenging 3d data-sparse matrix problems. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 79–89. IEEE. 12

[26] Chan, T. F. (1987). Rank revealing qr factorizations. *Linear Algebra and its Applications*, 88-89:67–82. 14

[27] Chandrasekaran, S. and Ipsen, I. C. (1994). On rank-revealing factorisations. *SIAM Journal on Matrix Analysis and Applications*, 15(2):592–622. 14

[28] Cooper, K. and Xu, X. (2018). Efficient characterization of hidden processor memory hierarchies. In *Computational Science – ICCS 2018*, pages 335–349, Cham. Springer International Publishing. 12, 19

[29] da Cunha, R. D., Becker, D., and Patterson, J. C. (2002). New parallel (rank-revealing) qr factorization algorithms. In Monien, B. and Feldmann, R., editors, *Euro-Par 2002 Parallel Processing*, pages 677–686, Berlin, Heidelberg. Springer Berlin Heidelberg. 14

[30] Danalis, A., Jagode, H., Hanumantharayappa, Ragate, S., and Dongarra, J. (2019). Counter inspection toolkit: Making sense out of hardware performance events. In *Tools for High Performance Computing 2017*, pages 17–37, Cham. Springer International Publishing. 15, 19, 28, 65

[31] Danalis, A., Luszczek, P., Dongarra, J., Marin, G., and Vetter, J. S. (2011). Blackjackbench: Portable hardware characterization. In *Proceedings of the Second International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems*, PMBS '11, page 7–8, New York, NY, USA. Association for Computing Machinery. 15, 19

[32] Das, S., Kanungo, B., Subramanian, V., Panigrahi, G., Motamarri, P., Rogers, D., Zimmerman, P., and Gavini, V. (2023). Large-scale materials modeling at quantum accuracy: Ab initio simulations of quasicrystals and interacting extended defects in metallic alloys. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. 2

[33] Deakin, T., Price, J., Martineau, M., and McIntosh-Smith, S. (2016). Gpu-stream v2. 0: Benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming models. In *High Performance Computing: ISC High Performance 2016 International Workshops, ExaComm, E-MuCoCoS, HPC-IODC, IXPUG, IWOPH, P^ 3MA, VHPC, WOPSSS, Frankfurt, Germany, June 19–23, 2016, Revised Selected Papers 31*, pages 489–507. Springer. 12

[34] Dessole, M. and Marcuzzi, F. (2022). Deviation maximization for rank-revealing qr factorizations. *Numerical Algorithms*, 91(3):1047–1079. 13

[35] Dongarra, J., Heroux, M. A., and Luszczek, P. (2016). High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems. *The International Journal of High Performance Computing Applications*, 30(1):3–10. 11

[36] Dongarra, J., Luszczek, P., and Tsai, Y. M. (2019). Hpl-mxp mixed-precision benchmark. https://hpl-mxp.org/. 11

[37] Dongarra, J., Moore, S., Mucci, P., Seymour, K., and You, H. (2004). Accurate cache and tlb characterization using hardware counters. In *Computational Science - ICCS 2004*, pages 432–439, Berlin, Heidelberg. Springer Berlin Heidelberg. 15

[38] Dongarra, J. J. (1987). The linpack benchmark: An explanation. In *International Conference on Supercomputing*, pages 456–474. Springer. 11

[39] Dongarra, J. J. and Tomov, S. (2014). Matrix algebra for gpu and multicore architectures (magma) for large petascale systems. Technical report, Univ. of Tennessee, Knoxville, TN (United States). 14

[40] Drineas, P., Mahoney, M. W., and Muthukrishnan, S. (2008). Relative-error cur matrix decompositions. *SIAM Journal on Matrix Analysis and Applications*, 30(2):844–881. 14

[41] Eranian, S. (2006). Perfmon2: A Flexible Performance Monitoring Interface for Linux. In *Proc. of the 2006 Ottawa Linux Symposium*, pages 269–288. Citeseer. 9

[42] Futral, W. S. et al. (2018). Level-2 Milestone 6459: Sierra Applications Development Environment. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States). 12

[43] Gates, M., Kurzak, J., Charara, A., YarKhan, A., and Dongarra, J. (2019). Slate: Design of a modern distributed and accelerated linear algebra library. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–18. 5

[44] Geimer, M., Wolf, F., Wylie, B. J. N., Ábrahám, E., Becker, D., and Mohr, B. (2010). The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719. 9

[45] Golub, G. and Kahan, W. (1965). Calculating the singular values and pseudo-inverse of a matrix. *Journal of the Society for Industrial and Applied Mathematics Series B Numerical Analysis*, 2(2):205–224. 14

[46] Golub, G., Klema, V., and Stewart, G. W. (1976). Rank degeneracy and least squares problems. Technical report, STANFORD UNIV CA DEPT OF COMPUTER SCIENCE. 14

[47] González-Domínguez, J., Taboada, G. L., Fragüela, B. B., Martín, M. J., and Touriño, J. (2010). Servet: A benchmark suite for autotuning on multicore clusters. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–9. 12, 19

[48] Gu, M. and Eisenstat, S. C. (1996). Efficient algorithms for computing a strong rank-revealing qr factorization. *SIAM J. Sci. Comput.*, 17:848–869. 14

[49] H. McCraw and J. Ralph and A. Danalis and J. Dongarra (2014). Power Monitoring with PAPI for Extreme Scale Architectures and Dataflow-based Programming Models. pages 385–391. 18

[50] Habib, S., Morozov, V., Frontiere, N., Finkel, H., Pope, A., and Heitmann, K. (2013). Hacc: Extreme scaling and performance across diverse architectures. In *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–10. 106

[51] Haidar, A., Jagode, H., Vaccaro, P., YarKhan, A., Tomov, S., and Dongarra, J. (2019). Investigating power capping toward energy-efficient scientific applications. *Concurrency and Computation: Practice and Experience*, 31(6):e4485. 12

[52] Haidar, A., Jagode, H., YarKhan, A., Vaccaro, P., Tomov, S., and Dongarra, J. (2017). Power-aware computing: Measurement, control, and performance analysis for intel xeon phi. In *2017 IEEE high performance extreme computing conference (HPEC)*, pages 1–7. IEEE. 12

[53] Henning, J. L. (2006). Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17. 11

[54] Hong, Y. P. and Pan, C.-T. (1992). Rank-revealing qr factorizations and the singular value decomposition. *Mathematics of Computation*, 58(197):213–232. 14

[55] Huck, K. A., Porterfield, A., Chaimov, N., Kaiser, H., Malony, A. D., Sterling, T., and Fowler, R. (2015). An autonomic performance environment for exascale. *Supercomputing frontiers and innovations*, 2(3):49–66. 9

[IBM Corporation] IBM Corporation. Power isa version 3.0 b. https://ibm.box.com/s/1hzcwkwf8rbju5h9iyf44wm94amnlcrv. 112

[57] IBM Corporation (2018a). Power9 performance monitor unit user's guide, 2018. https://wiki.raptorcs.com/w/images/6/6b/POWER9_PMU_UG_v12_28NOV2018_pub.pdf. 5, 28, 45, 91

[58] IBM Corporation (2018b). Power9 processor user's manual. https://www.ibm.com/developerworks/community/files/basic/anonymous/api/library/35a0c17a-cd5e-4750-8f73-d98b6880d77b/document/828804a0-e5d7-480c-bad1-cf21342c3889/media/POWER9%20Processor.pdf. 28, 45, 50, 99, 112

[59] Intel Corporation (2023). Using the oneAPI Level Zero Interface. https://www.intel.com/content/www/us/en/developer/articles/technical/using-oneapi-level-zero-interface.html. 9

[60] Jagode, H. (2006). Fourier transforms for the bluegene / l communication network. Master's thesis, EPCC, The University of Edinburgh. 106

[61] Jagode, H., YarKhan, A., Danalis, A., and Dongarra, J. (2016). Power management and event verification in papi. In *Tools for High Performance Computing 2015: Proceedings of the 9th International Workshop on Parallel Tools for High Performance Computing, September 2015, Dresden, Germany*, pages 41–51. Springer. 12

[62] Jolliffe, I. T. (2002). *Principal component analysis*. Springer. 14

[63] Kim, J. et al. (2018). Qmcpack: An open source ab initio quantum monte carlo package for the electronic structure of atoms, molecules and solids. *Journal of Physics Condensed Matter*, 30(19). 106, 120

[64] Lawson, C. L. and Hanson, R. J. (1995). *Solving least squares problems*. SIAM. 13

[65] Li, B., León, E. A., and Cameron, K. W. (2017). Cos: A parallel performance model for dynamic variations in processor speed, memory speed, and thread concurrency. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '17, pages 155–166, New York, NY, USA. Association for Computing Machinery. 16

[66] Liberty, E., Woolfe, F., Martinsson, P.-G., Rokhlin, V., and Tygert, M. (2007). Randomized algorithms for the low-rank approximation of matrices. *Proceedings of the National Academy of Sciences*, 104(51):20167–20172. 14

[67] Limited, F. (2022). A64fx pmu events. https://github.com/fujitsu/A64FX/blob/master/doc/A64FX_PMU_Events_v1.3.pdf. 5

[68] LLNL (2018). Coral-2 benchmarks. https://asc.llnl.gov/coral-2-benchmarks/. 11

[69] LLNL (2023). Benchpark. https://github.com/LLNL/benchpark. 15

[70] Mahoney, M. W. and Drineas, P. (2009). Cur matrix decompositions for improved data analysis. *Proceedings of the National Academy of Sciences*, 106(3):697–702. 14

[71] Malik, H., Hemmati, H., and Hassan, A. E. (2013). Automatic detection of performance deviations in the load testing of large scale systems. In *2013 35th international conference on software engineering (ICSE)*, pages 1012–1021. IEEE. 16

[72] Malik, H., Jiang, Z. M., Adams, B., Hassan, A. E., Flora, P., and Hamann, G. (2010). Automatic comparison of load tests to support the performance analysis of large enterprise systems. In *2010 14th European conference on software maintenance and reengineering*, pages 222–231. IEEE. 16

[73] Malony, A. D., Biersdorff, S., Shende, S., Jagode, H., Tomov, S., Juckeland, G., Dietrich, R., Poole, D., and Lamb, C. (2011). Parallel performance measurement of heterogeneous parallel systems with gpus. In *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP '11, pages 176–185, Washington, DC, USA. IEEE Computer Society. 18

[74] Martinsson, P.-G., Rokhlin, V., Shkolnisky, Y., and Tygert, M. (2008). Id: A software package for low-rank approximation of matrices via interpolative decompositions, version 0.2. 14

[75] McCalpin, J. D. et al. (1995). Memory bandwidth and machine balance in current high performance computers. *IEEE computer society technical committee on computer architecture (TCCA) newsletter*, 2(19-25). 5, 11

[76] McCraw, H., Terpstra, D., Dongarra, J., Davis, K., and R., M. (2013). Beyond the CPU: Hardware Performance Counter Monitoring on Blue Gene/Q. In *Proceedings*

of the International Supercomputing Conference 2013, ISC'13, pages 213–225. Springer, Heidelberg. 18, 106

[77] McIntosh-Smith, S., Price, J., Deakin, T., and Poenaru, A. (2019). A performance analysis of the first generation of hpc-optimized arm processors. *Concurrency and Computation: Practice and Experience*, 31(16):e5110. 12

[78] McVoy, L. and Staelin, C. (1996). `lmbench`: portable tools for performance analysis. In *ATEC'96: Proceedings of the Annual Technical Conference on USENIX 1996 Annual Technical Conference*, pages 23–23, Berkeley, CA, USA, USENIX Association. USENIX Association. 11, 12, 19

[79] Mei, X. and Chu, X. (2016). Dissecting gpu memory hierarchy through microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):72–86. 12

[80] Molnar, I. (2009). perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/. 9

[81] Mucci, P. J., London, K., and Thurman, J. (1998). The cachebench report. *University of Tennessee, Knoxville, TN*, 19. 11, 12, 19

[82] Mullen, J. S., Wolf, M. M., and Klein, A. (2013). Pakck: Performance and power analysis of key computational kernels on cpus and gpus. In *2013 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE. 12

[NVIDIA Corporation] NVIDIA Corporation. Qmcpack. https://catalog.ngc.nvidia.com/orgs/hpc/containers/qmcpack. 120

[84] NVIDIA Corporation (2023a). NVIDIA CUDA Profiling Tools Interface (CUPTI) - CUDA Toolkit. https://developer.nvidia.com/cupti. 9

[85] NVIDIA Corporation (2023b). NVIDIA Management Library (NVML). https://developer.nvidia.com/nvidia-management-library-nvml. 9

[86] OLCF (2019). Summit user guide. https://docs.olcf.ornl.gov/systems/summit_user_guide.html. 5, 13, 91

[87] OLCF (2023). Frontier user guide. https://docs.olcf.ornl.gov/systems/frontier_user_guide.html. 58

[88] Pearce, O., Scott, A., Becker, G., Haque, R., Hanford, N., Brink, S., Jacobsen, D., Poxon, H., Domke, J., and Gamblin, T. (2023). Towards collaborative continuous benchmarking for hpc. In *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, pages 627–635. 15

[89] Phansalkar, A., Joshi, A., and John, L. K. (2007). Analysis of redundancy and application balance in the spec cpu2006 benchmark suite. In *Proceedings of the 34th annual International Symposium on Computer architecture*, pages 412–423. 16

[90] Pouchet, L.-N. (2012). Polybench/c: The polyhedral benchmark suite. https://web.cs.ucla.edu/~pouchet/software/polybench/. 11

[91] Ravikumar, K., Appelhans, D., and Yeung, P. K. (2019). Gpu acceleration of extreme scale pseudo-spectral simulations of turbulence using asynchronism. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA. Association for Computing Machinery. 106

[92] Red Hat (2021). Performance Co-Pilot. https://pcp.io/index.html. 9, 90

[93] RIKEN-CCS (2021). About fugaku. https://www.r-ccs.riken.jp/en/fugaku/about/. 5

[94] Ritter, M., Tarraf, A., Geiß, A., Daoud, N., Mohr, B., and Wolf, F. (2022). Conquering noise with hardware counters on hpc systems. In *2022 IEEE/ACM Workshop on Programming and Performance Visualization Tools (ProTools)*, pages 1–10. 10, 16, 68

[95] Sandoval, J. (2011). Foundations for automatic, adaptable compilation. Doctoral dissertation, Rice University. 12, 19

[96] Schlütter, M., Philippen, P., Morin, L., Geimer, M., and Mohr, B. (2014). Profiling Hybrid HMPP Applications with Score-P on Heterogeneous Hardware. In *Parallel Computing: Accelerating Computational Science and Engineering (CSE)*, volume 25 of *Advances in Parallel Computing*, pages 773 – 782. IOS Press. 9, 57, 90

[97] Shende, S. S. and Malony, A. D. (2006). The Tau Parallel Performance System. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311. 9, 57, 90

[98] Sid-Lakhdar, W. et al. (2023). PAQR: Pivoting Avoiding QR Factorization. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 322–332. IEEE. 13, 71, 80

[99] Spradling, C. D. (2007). Spec cpu2006 benchmark tools. *ACM SIGARCH Computer Architecture News*, 35(1):130–134. 11

[100] Sussman, A., Lo, N., and Anderson, T. (2011). Automatic computer system characterization for a parallelizing compiler. In *2011 IEEE International Conference on Cluster Computing*, pages 216–224. 12, 19

[101] Terpstra, D., Jagode, H., You, H., and Dongarra, J. (2009). Collecting Performance Data with PAPI-C. *Tools for High Performance Computing 2009*, pages 157–173. 9, 18, 57, 90

[102] Top500 (2023). November 2023. https://www.top500.org/lists/top500/2023/11/. 2, 5

[103] Treibig, J., Hager, G., and Wellein, G. (2010). LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proc. of the First International Workshop on Parallel Software Tools and Tool Infrastructures*. 9

[104] Vazhkudai, S. S., De Supinski, B. R., Bland, A. S., Geist, A., Sexton, J., Kahle, J., Zimmer, C. J., Atchley, S., Oral, S., Maxwell, D. E., et al. (2018). The design, deployment, and evaluation of the coral pre-exascale systems. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 661–672. IEEE. 11

[105] Vergara Larrea, V. G., Joubert, W., Brim, M. J., Budiardja, R. D., Maxwell, D., Ezell, M., Zimmer, C., Boehm, S., Elwasif, W., Oral, S., et al. (2019). Scaling the summit: deploying the world's fastest supercomputer. In *High Performance Computing: ISC High Performance 2019 International Workshops, Frankfurt, Germany, June 16-20, 2019, Revised Selected Papers 34*, pages 330–351. Springer. 5, 12, 56

[106] Voronin, S. and Martinsson, P.-G. (2017). Efficient algorithms for cur and interpolative matrix decompositions. *Advances in Computational Mathematics*, 43:495–516. 14

[107] Weaver, V. M., Terpstra, D., and Moore, S. (2013). Non-determinism and overcount on modern hardware performance counter implementations. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 215–224. IEEE. 10

[108] Wolf, J. (1999). *Programming Methods for the Pentium™ III Processor's Streaming SIMD Extensions Using the VTune™Performance Enhancement Environment.* Intel Corporation. 9

[109] Wong, H., Papadopoulou, M.-M., Sadooghi-Alvandi, M., and Moshovos, A. (2010). Demystifying gpu microarchitecture through microbenchmarking. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pages 235–246. IEEE. 12

Vita

Daniel Barry was born in Knoxville, Tennessee in August 1995. He earned his Bachelor's of Science degree in Computer Engineering in 2018 at the University of Tennessee, Knoxville (UTK), where he majored in both Computer Engineering and Mathematics. Later that year, Daniel began working as a Graduate Research Assistant at UTK in the Performance Tools Group of the Innovative Computing Laboratory (ICL) under the guidance of Dr. Heike Jagode, Dr. Anthony Danalis, and Dr. Jack Dongarra. While in ICL, Daniel pursued his Doctor of Philosophy degree in Data Science and Engineering. He expects to complete his PhD in December of 2025.