Effortless Monitoring of Arithmetic Intensity with PAPI's Counter Analysis Toolkit

Daniel Barry, Anthony Danalis, Heike Jagode

Abstract With exascale computing forthcoming, performance metrics such as memory traffic and arithmetic intensity are increasingly important for codes that heavily utilize numerical kernels. 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 which native performance events are indexed by which event names, making it difficult for users to understand what specific events actually measure. This ambiguity seems particularly true for events related to hardware that resides beyond the compute core, such as events related to memory traffic. Still, traffic to memory is a necessary characteristic for determining arithmetic intensity. To alleviate this difficulty, PAPI's Counter Analysis Toolkit measures the occurrences of events through a series of benchmarks, allowing its users to discover the high-level meaning of native events. We (i) leverage the capabilities of the Counter Analysis Toolkit to identify the names of hardware events for reading and writing bandwidth utilization in addition to floating-point operations, (ii) measure the occurrences of the events they index during the execution of important numerical kernels, and (iii) verify their identities by comparing these occurrence patterns to the expected arithmetic intensity of the numerical kernels.

1 Introduction

Most of the major tools that high-performance computing (HPC) application developers use to conduct low-level performance analysis and tuning of their applica-

Daniel Barry, Anthony Danalis, Heike Jagode
Innovative Computing Laboratory (ICL) - University of Tennessee, Knoxville
Knoxville TN 37996
USA
email: dbarry@vols.utk.edu, {adanalis|jagode}@icl.utk.edu

tions typically rely on hardware performance counters to monitor hardware-related activities. The kind of available counters 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 performance counters available on all modern CPUs, as well as GPUs, networks, and I/O systems [14, 8, 9]. Additionally, PAPI supports transparent power monitoring capabilities for various platforms, including GPUs (AMD, NVIDIA) and Intel Xeon Phi [5], enabling PAPI users to monitor power in addition to traditional hardware performance counter 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 we discuss in this paper. 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 "inter-core" resources has already become a major drag on overall application performance.

Furthermore, we discuss one of PAPI's new features: the Counter Analysis Toolkit (CAT), which is designed to improve the understanding of these inter-core events. Specifically, the CAT integrates methods based on micro-benchmarking to gain a better handle on Nest/Offcore/Uncore/NorthBridge counter-related events—depending on the hardware vendor. For simplicity, hereafter we will refer to such counters as Uncore, regardless of the vendor.

We aim to define and verify accurate mappings between particular high-level concepts of performance metrics and 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 paper we outline the new PAPI Counter Analysis Toolkit, describe its objective, and then focus on the micro-kernels that are used to measure and correlate different native events to compute the arithmetic intensity on the Intel Broadwell, Intel Skylake, and IBM POWER9 architectures.

2 Counter Analysis Toolkit

Native performance events are often appealing to scientific application developers who are interested in understanding and improving the performance of their code. However, in modern architectures it is not uncommon to encounter 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-grained granularities of measurements when it comes to resources that are shared between the compute cores (e.g., off-chip caches and main memory).

In earlier work [2], we explored the use of benchmarks that employ techniques such as pointer chasing [1, 3, 4, 10, 11, 12, 13] to stress the memory hierarchy as well as micro-benchmarks with different branching behaviors to test different branch-related events. The CAT, which was released with PAPI version 6.0.0, has built upon these earlier findings by significantly expanding the kinds of tests performed by our micro-benchmarks, 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 (made after the PAPI 6.0.0 release) publicly available through the PAPI project's Git repository.

CAT currently contains benchmarks for testing four different aspects of CPUs: data caches, instruction caches, branches, and floating-point operations (FLOPs). The micro-benchmarks 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, which is currently included with CAT, uses specific combinations of parameters that we have determined appropriate for revealing important differences between different native events. More details on the actual tests are discussed in the following sections.

2.1 Data Cache tests

Figure 1 shows a plot of the data generated when the data cache read benchmark is executed. As shown in the figure, there are six regions that correspond to six different parameter sets. In the first four regions, the access pattern is random ("RND"), and it is sequential ("SEQ") in the last two. 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 between 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. The third parameter that varies between the six regions is the size of the contiguous block of memory 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 76 measure-

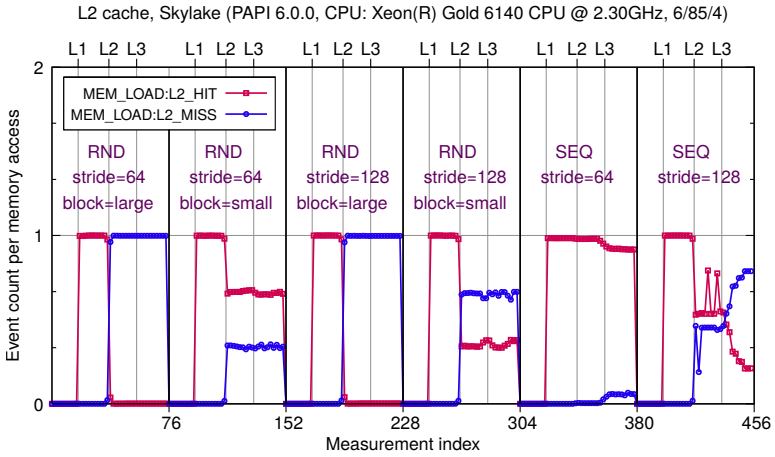


Fig. 1: L2 Data Cache Events.

ments, and within each set of 76 measurements the X-axis corresponds to the size of the buffer that the benchmark uses. To improve the readability of Figure 1, at the top of the graph, we have marked the measurement indices within each region that correspond to the sizes of the three caches (L1, L2, L3) of the testbed we used (Skylake 6140). 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, or a streamed traversal of each cache line in the buffer). 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, as can be seen in the figure.

The red curve with square points depicts the number of hits in the L2 cache per memory access (hit rate). In each of the six 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 six regions, where the red curve stays at one hit count per memory access between the markers for the L1 and L2 cache sizes (shown at the top 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.

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

block=small: For small working sets ("block=small"), 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.

block=small, stride=64: 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.

block=small, stride=128: 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 last two regions of the graph show the results when the buffer is accessed sequentially ("SEQ"). In these regions, the notion of "block" does not apply (since the whole buffer is accessed as one contiguous block) and the access pattern is so simple that prefetching is most efficient. The only limiting factor is the bandwidth of the memory subsystem beyond the L2 cache, which is stressed twice as much when the stride is 128 bytes, leading to a lower hit rate than the case of 64 byte stride.

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

2.2 Instruction Cache tests

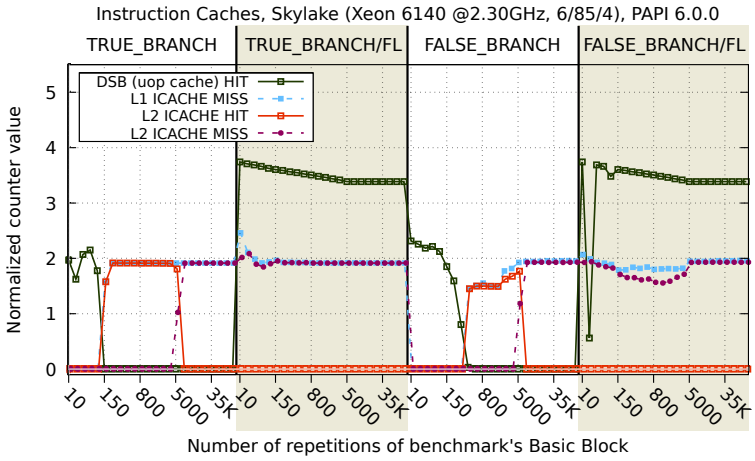


Fig. 2: Instruction Cache Events.

Unlike the case we discussed in the previous section—where the same micro-benchmark code was used while key parameters were varied to achieve different results—the instruction benchmark consists of a series of automatically generated micro-benchmark functions that have a variable number of instructions. In figure 2 we plot the data generated when the instruction cache benchmark is executed. The data in the figure are in four regions. Within each region, the micro-benchmark functions have the same design, but varying numbers of repetitions of their basic block, which are displayed in the X-axis. The difference between 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** `TRUE_BRANCH/FL`**:** The code is the same as in the first region; however, a large array is accessed at the end of each iteration, so that unified caches are flushed ("`FL`").
3. **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.
4. **region** `FALSE_BRANCH/FL`**:** The code is the same as in the third region, but it also performs the large data traversal to flush the caches ("`FL`").

Normalization of data for the purpose of readability: [1] In each of the four regions, we normalize the raw counter 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:

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

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 2, the green line with the hollow 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 the first and third regions of the graph, the green line reveals, for small benchmark codes (fewer than 150 repetitions of the basic block) most instructions are

[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 they cannot be visualized in a readable way, not even in a logarithmic graph.

delivered to the back-end from the DSB. In regions two and four, however, we see a normalized value above 3.5, which corresponds to millions of events. This is due to the loop that accesses the large array in order to flush the unified caches (L2 and L3). The code of the loop is tiny (a simple read from an array and accumulation into a scalar), and thus, it easily fits in the DSB but executes tens of millions of times in order to flush the L3.

The dashed light-blue line with solid square points depicts the (normalized) miss rate of the L1 instruction cache. In regions one and three of the graph, we see that the L1 only experiences misses when the code becomes large. Interestingly, in regions two and four, we can see that the L1 instruction cache experiences misses even at very small code sizes. This is most likely due to the flushing of the L3 cache, which is inclusive and therefore invalidates the L1 instruction cache.

Likewise, the (normalized) L2 miss rate, displayed by the purple curve with the solid points, follows a similar pattern as the L1 miss rate.

The (normalized) L2 hit rate, depicted by the red curve with the hollow square points, shows a peak for moderately sized codes, and zero for smaller and larger codes. In addition, we can observe that the L2 hit rate in the first region—where all the code in the cache is used—is higher than the hit rate in the third region—where the false branch causes part of the code to be fetched but not executed.

In summary, the goal of this work is to generate benchmarks that make these curves different from one another, so we can distinguish between performance events that have semantic differences. While Figure 2 holds a significant amount of data, the curves shown are notably distinct from each other, which substantiates the validity of this effort.

2.3 Branch Tests

Figure 3 shows a plot of the data generated when the branch benchmark is executed. This test consists of a series of different hand-crafted micro-benchmarks (currently eleven), each of which exhibits different behavior from the others with respect to one or more branch instructions. Consequently, when all micro-benchmarks are used, each type of branch event produces a unique signature, as can be seen in the figure.

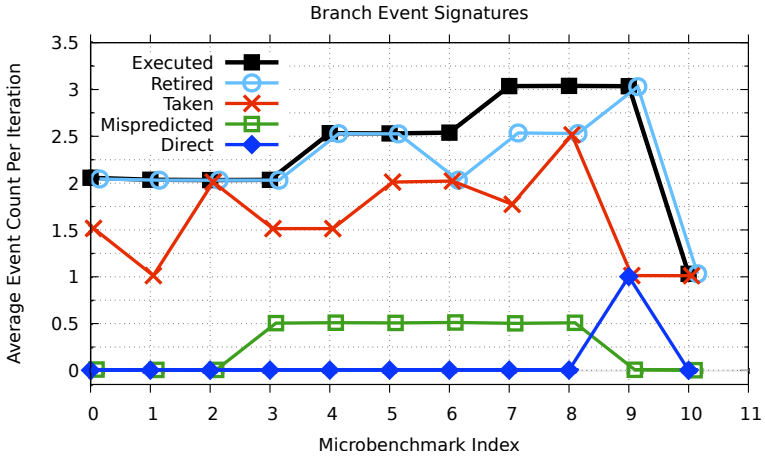


Fig. 3: Branch Events.

Listing 1: Branch benchmark #5.

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

Listing 2: Branch benchmark #9.

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

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

To illustrate the workings of these micro-benchmarks we show the key loop of two of them in the code Listings 1 and 2. These two codes correspond to the measurements shown in the graph at index 5 and 9, respectively.

Looking at the 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_EXEC:ALL_DIRECT_JMP`). On the other hand, at index 9 the 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 hollow square points indicates that benchmark #5 will experience branch mispredictions 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 `X` points indicates that in benchmark #5 two conditional branches are taken at each iteration (`BR_INST_EXEC:TAKEN_CONDITIONAL`), while in the case of benchmark #9 only one conditional branch is taken at each iteration. Although not shown in this graph, benchmark #9 also triggers a direct jump to be taken (`BR_INST_EXEC:TAKEN_DIRECT_JUMP`). At first glance, it might be puzzling that benchmark #9 only records 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 hollow round points and the black curve with solid square points indicate that benchmark #5 executes two and a half branches per iteration and all of them are retired (i.e., they are not discarded due to 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 [2].

2.4 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 [7, 6]. For the sake of consistency across architectures, we closely examine the double-precision FLOPs.

Figure 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

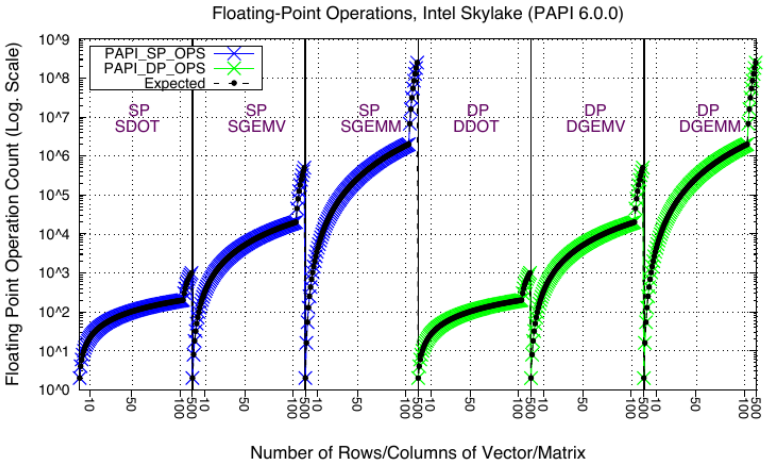


Fig. 4: Single-Precision and Double-Precision Floating-Point Events.

("SP") implementations of the 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.

Within each region in Figure 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 kernels. For each N, the benchmark executes the BLAS kernel of the floating-point precision corresponding to the region. In Figure 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 $2 * N$ 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 $2 * N^3$ 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 re-

gions because the single- precision BLAS kernels do not invoke double-precision FLOPs.

3 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 real 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 new monitoring features for bandwidth utilization and arithmetic intensity help to make meaningful predictions for a real application? And,
3. How reliable are FLOP and memory bandwidth utilization performance counters 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, 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 $(2n * 8$ 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.

3.1 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 counters 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 1 shows how the two PAPI FLOPs presets are derived from the native counters as they are available on our three chosen architectures. In this paper, however, we exclusively focus on double-

Table 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*256B_PACKED_DOUBLE +`
	`8*512B_PACKED_DOUBLE`
		`FP_ARITH:SCALAR_SINGLE +`
		`4*FP_ARITH:128B_PACKED_SINGLE +`
		`8*256B_PACKED_SINGLE +`
		`16*512B_PACKED_SINGLE`

Broadwell	`FP_ARITH:SCALAR_DOUBLE +`
	`2*FP_ARITH:128B_PACKED_DOUBLE +`
	`4*256B_PACKED_DOUBLE`
		`FP_ARITH:SCALAR_SINGLE +`
		`4*FP_ARITH:128B_PACKED_SINGLE +`
		`8*256B_PACKED_SINGLE`

Power9	`PM_DP_QP_FLOP_CMPL`	`PM_SP_FLOP_CMPL`

precision arithmetic, and thus we will not include `PAPI_SP_OPS` measurements in our analyses.

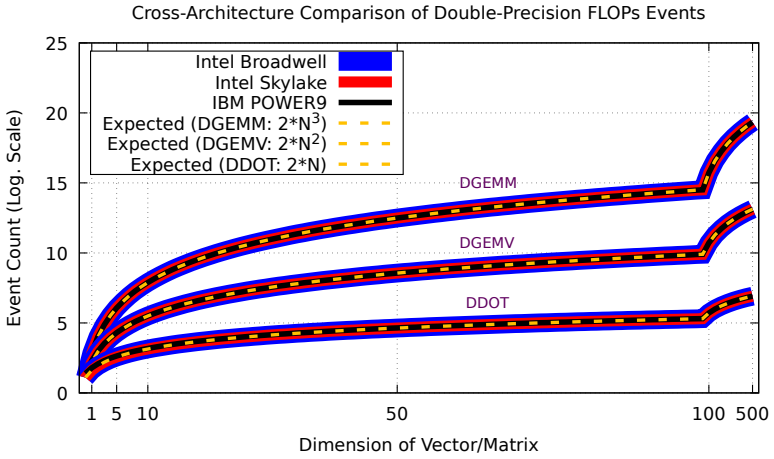


Fig. 5: BLAS FLOPs on the Broadwell, Skylake, and POWER9 architectures.

Figure 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—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 1) reliably measure double-precision floating-point operations for various kernels with different computational characteristics.

In Figures 6 and 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 of memory,

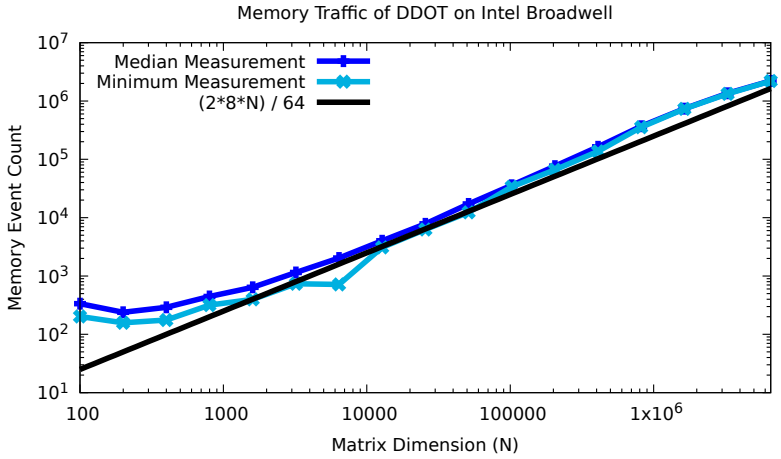


Fig. 6: DDOT Memory Accesses on the Intel Broadwell architecture.

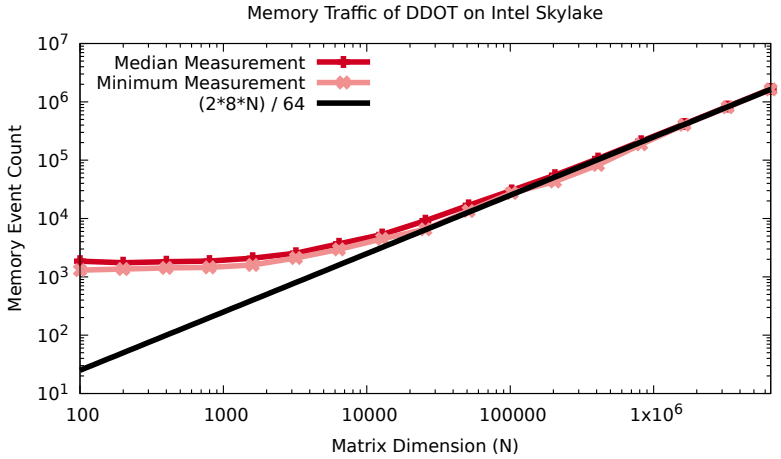


Fig. 7: DDOT Memory Accesses on the Intel Skylake architecture.

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 6 and 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 6 and 7 show that the PAPI counters on both the Intel Broadwell and Skylake architectures measure the correct memory traffic for the DDOT operation. In Section 4, we elaborate further on the actual PAPI events that we used for measuring memory traffic.

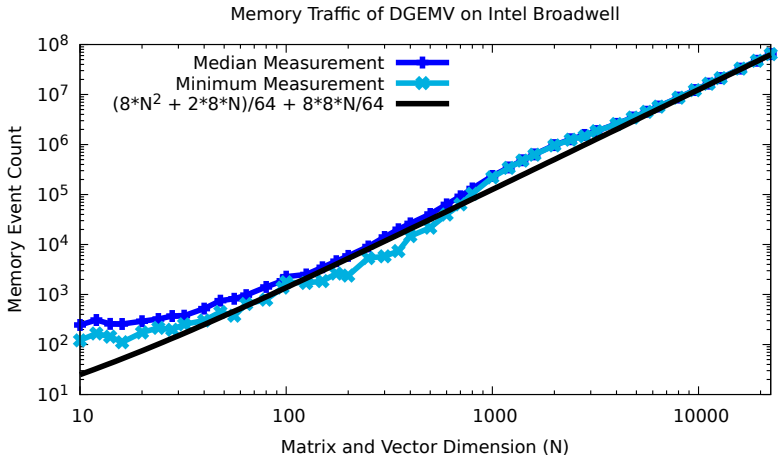


Fig. 8: DGEMV Memory Accesses on the Intel Broadwell architecture.

Figure 8 and Figure 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, totalling $8*(N^2+2*N)$ reads. It incurs a write for each of the elements in the result vector, which would total $8*N$ writes. But other micro-benchmarks indicate that the cache writes back to memory in whole counts of a cache line. To account for this, we instead include the term $8*8*N$ ($8*8bytes = 64bytes$, which is the size of a cache line) in the expectation formula shown in Figures 8 and 9. This term would theoretically have more influence on the total expectation for memory traffic than $8*N$, but since the reads 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 $8*N$ writes, are visually indistinguishable, we include $8*8*N$. Thus, the total expectation for the memory traffic of the DGEMV operation is the number of reads plus the number of writes di-

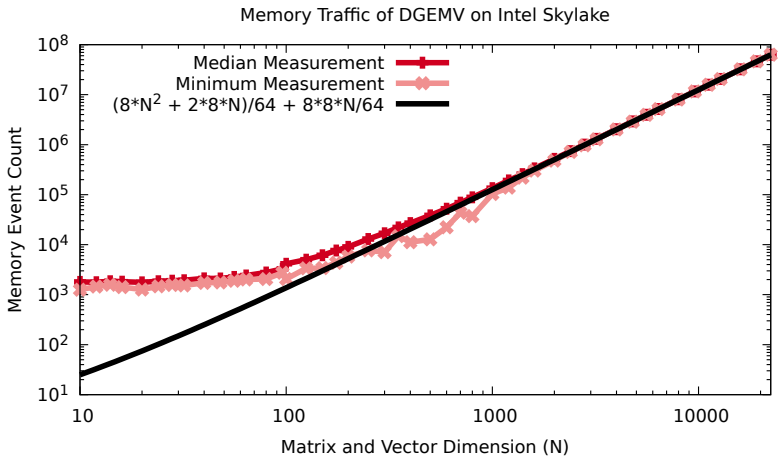


Fig. 9: DGEMV Memory Accesses on the Intel Skylake architecture.

vided by 64, $\frac{(8*(N^2+2*N)+8*8*N)}{64}$, by virtue of the memory traffic events we 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 8 and 9 show that the PAPI counters on both the Intel Broadwell and Skylake architectures measure the correct memory traffic for the DGEMV operation.

Figures 10 and 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 $8*3*N^2$ reads. It incurs a write for each of the elements of the result matrix, totalling either $8*8*N^2$ or $8*N^2$ writes, depending on whether the writebacks to memory occur per cache lines written or per elements written, respectively. Since the writes 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 10 and 11. Unlike the DDOT and DGEMV operations, the DGEMM operation is sensitive to the size of the cache of the CPU

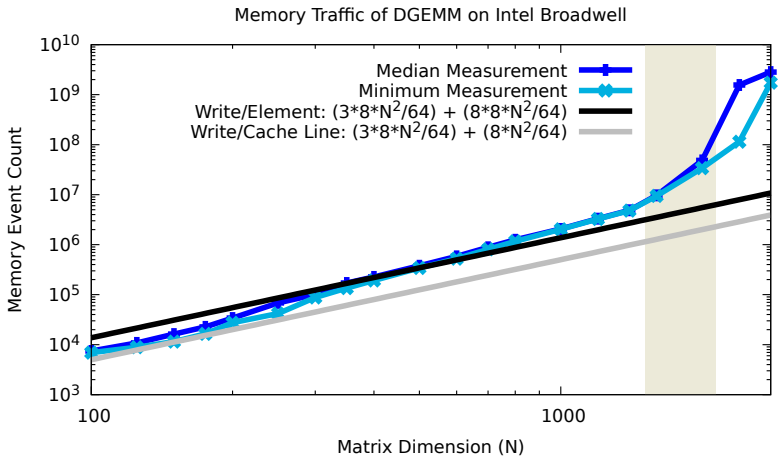


Fig. 10: DGEMM Memory Accesses on the Intel Broadwell architecture.

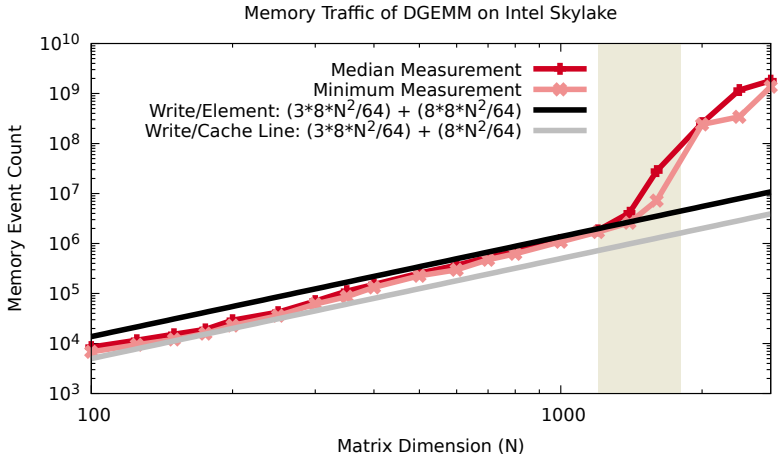


Fig. 11: DGEMM Memory Accesses on the Intel Skylake architecture.

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 hardware caches the entire second operand matrix but only a row of the first operand matrix, then we

establish a lower bound on the maximum dimension of the 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 DGEMM operation caches the entire first and second operand matrices, 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 10 and 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 10 and 11 show that we obtain the correct measurements for memory traffic for the DGEMM operation utilizing the PAPI counters on the Intel Broadwell and Skylake architectures.

4 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 events which we utilize for the PAPI counters 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 Uncore 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 12 and 13, respectively. The events measured in Figure 14

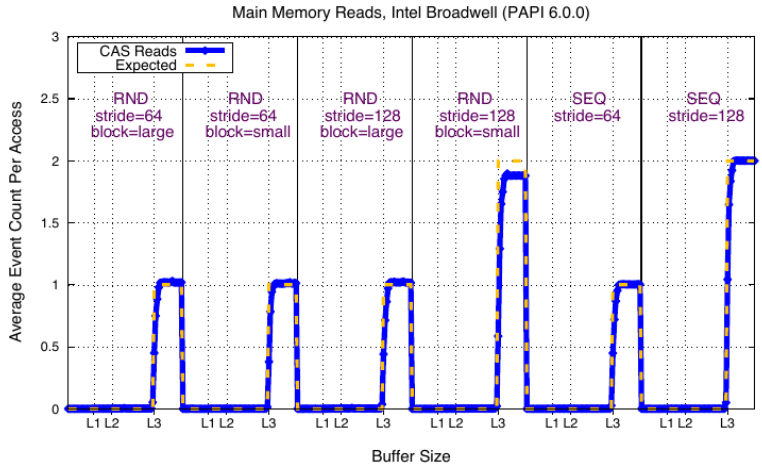


Fig. 12: Memory reading traffic on the Broadwell architecture.

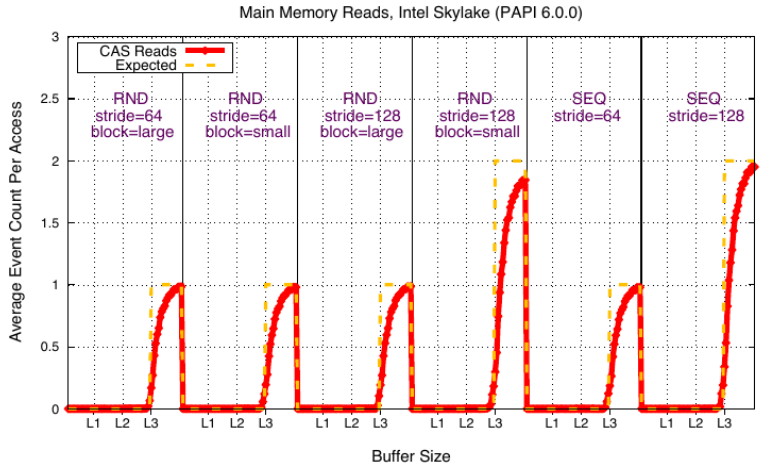


Fig. 13: Memory reading traffic on the Skylake architecture.

exhibit similar behavior to those of memory reading events measured in Figures 12 and 13. Note that the expectation in the third and fourth regions in Figure 14 varies from those in Figures 12 and 13 since the size of a cache line on the IBM POWER9 architecture is 128 Bytes [7]. Subsequent cross-referencing of [6] 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

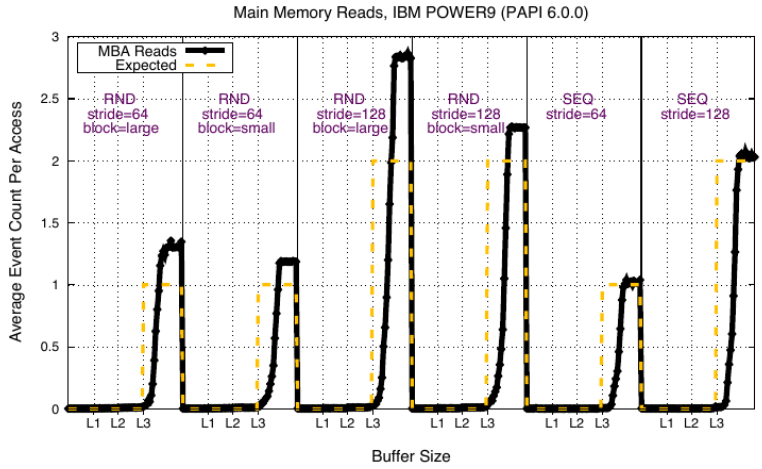


Fig. 14: Memory reading traffic on the POWER9 architecture.

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]
```

4.1 IBM POWER9 Measurements via PCP

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

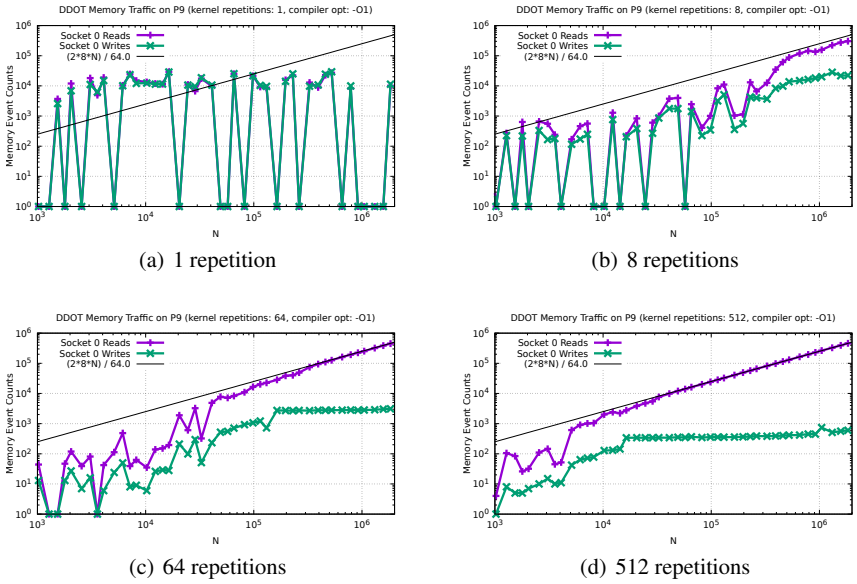
If a measurement infrastructure—e.g., PCP—is susceptible to noise, it is usually beneficial to take measurements of operations that take longer to complete and result in larger measurements in order to amortize the noise. This approach, however,

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.

Listing 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 }
```

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 1,2), and every time we execute the operation we provide it 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 12,13,14). 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.



(a) 1 repetition (b) 8 repetitions

(c) 64 repetitions (d) 512 repetitions

Fig. 15: POWER9 measurements of memory traffic events via PCP for DDOT benchmark.

The results of this study can be seen in Figure 15. In these graphs, for any given vector size N the expected number of reads is given by the equation:

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

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" [7] (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 15 correlate to the measured writes for small N, these reads are regarded as noise.

The graph shown in 15(a) 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 15(b) 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 15(c) we used 64 repetitions of the operation and the

measurements start converging to the expected values much earlier. Finally, in 15(d) 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.

5 Conclusion

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 counters that may require elevated privileges, or have cryptic names.

In this paper we discussed our effort to simplify the effort of measuring these counters and quantifying their reliability through PAPI. In particular, we outlined CAT, a new tool that was released with PAPI 6.0.0, and showed how it can be used to identify which native 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 successfully computed on three modern 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.

To summarize, this paper 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 the PAPI counters. 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 new monitoring features for bandwidth utilization 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 Uncore 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 bandwidth utilization performance counters on the different architectures? Per our experiments, the PAPI counters report the expected FLOPs for the three BLAS operations on the Intel Broadwell, Intel Skylake, and IBM POWER9 architectures. The PAPI counters 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 PAPI counters provide reliable FLOP and memory traffic event counts across the three architectures we have examined.

Acknowledgment

This material is based upon work supported in part by the National Science Foundation under award No. 1450429 "PAPI-EX."

References

1. K. Cooper and X. Xu. Efficient characterization of hidden processor memory hierarchies. In *Computational Science – ICCS 2018*, pages 335–349, Cham, 2018. Springer International Publishing.
2. A. Danalis, H. Jagode, Hanumantharayappa, S. Ragate, and J. Dongarra. Counter inspection toolkit: Making sense out of hardware performance events. In *Tools for High Performance Computing 2017*, pages 17–37, Cham, 2019. Springer International Publishing.
3. A. Danalis, P. Luszczek, G. Marin, J. S. Vetter, and J. Dongarra. BlackjackBench: Portable hardware characterization with automated results' analysis. *The Computer Journal*, June 2013.
4. J. Gonzlez-Domnguez, G. L. Taboada, B. B. Fragela, M. J. Martn, and J. Tourio. Servet: A benchmark suite for autotuning on multicore clusters. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–9, 2010.
5. H. McCraw and J. Ralph and A. Danalis and J. Dongarra. Power Monitoring with PAPI for Extreme Scale Architectures and Dataflow-based Programming Models. pages 385–391, Sep 2014.
6. IBM Corporation. Power9 performance monitor unit users guide. `https://wiki.raptorcs.com/w/images/6/6b/POWER9_PMU_UG_v12_28NOV2018_pub.pdf`, 2018.
7. IBM Corporation. Power9 processor users 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`, 2018.
8. A. D. Malony, S. Biersdorff, S. Shende, H. Jagode, S. Tomov, G. Juckeland, R. Dietrich, D. Poole, and C. Lamb. 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, 2011. IEEE Computer Society.
9. H. McCraw, D. Terpstra, J. Dongarra, K. Davis, and M. R. 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, June 2013.
10. L. McVoy and C. Staelin. `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., January 24-26 1996. USENIX Association.
11. P. J. Mucci and K. London. The CacheBench Report. Technical report, Computer Science Department, University of Tennessee, Knoxville, TN, 1998.
12. J. Sandoval. Foundations for automatic, adaptable compilation. Doctoral dissertation, Rice University, 2011.
13. A. Sussman, N. Lo, and T. Anderson. Automatic computer system characterization for a parallelizing compiler. In *2011 IEEE International Conference on Cluster Computing*, pages 216–224, 2011.
14. D. Terpstra, H. Jagode, H. You, and J. Dongarra. Collecting Performance Data with PAPI-C. *Tools for High Performance Computing 2009*, pages pp. 157–173, 2009.