Memory Traffic and Complete Application Profiling with PAPI Multi-Component Measurements

Daniel Barry, Heike Jagode, Anthony Danalis, Jack Dongarra
Innovative Computing Laboratory, University of Tennessee, Knoxville, TN USA
dbarry@vols.utk.edu
{*jagode, adanalis, dongarra*}*@icl.utk.edu*

***Abstract*—Some of the most important categories of performance events count the data traffic between the processing cores and the main memory. However, since these counters 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 paper 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 counters 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.**

I. Introduction

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 counters 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 processes, applications must run with elevated privileges in order to access these hardware counters. 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 (PCP) [1] to export nest-related information to ordinary users.

The middleware library PAPI [2] offers a component that interfaces with PCP. Through this component, third-party performance tools that depend on PAPI to access hardware counters, such as TAU [3], Score-P [4], Vampir [5], Caliper [6], 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 hardware 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 counters 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 counters, especially when there are multiple sources of 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 paper, enables the user to assess the efficiency of the whole execution.

The following contributions are achieved in this work.

- We showcase monitoring of performance 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 hardware 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.

[1] In the rest of this document we will use the term "nest" since our work is focused on IBM systems.

979-8-3503-1199-0/23/$31.00 ©2023 IEEE
DOI 10.1109/IPDPSW59300.2023.00070

- 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 [7]. This serves as a basis for comparison of the fidelity of measurements from Summit using PCP.

The memory traffic performance events we measure on these systems are listed in Table I.

TABLE I: Architectures and Performance Events

System	Arch.	Performance 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

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" [8].

II. 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 [9], 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.

A. 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 *i*th element of vector y is the dot product of the vector x and the *i*th 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 1: Reference GEMV Kernel.

Per line 6 of Listing 1, the reference GEMV incurs one read from memory to retrieve the *k*th element of a row of matrix A and one read to retrieve the *k*th 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 $i_p = i\%P$.

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

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.

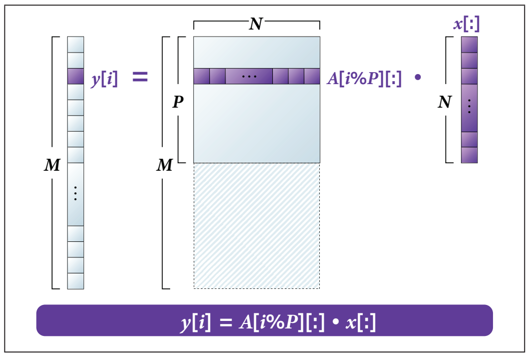


Fig. 1: Capped GEMV memory usage schematic.

This capping of memory allocation is shown visually in Figure 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.

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

Listing 2: Batched, Capped GEMV.

Listing 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 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.

B. 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:

$$c_{ij} = \sum_{k=1}^{N} a_{i,k} b_{k,j} \qquad (1 \leq i,j \leq N) \tag{2}$$

```
#pragma omp parallel for schedule(static)
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[i][k] * B[k][j];
    }
    /* Store result in memory. */
    C[i][j] = sum;
  }
```

Listing 3: Reference GEMM.

Line 7 of Listing 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 4: Batched GEMM.

Listing 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.

III. Benchmark Results

In Figure 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" [10].

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*:

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

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:

$$8 \times N^2 = 5 \times 1024^2 \implies N \approx 809 \tag{4}$$

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.

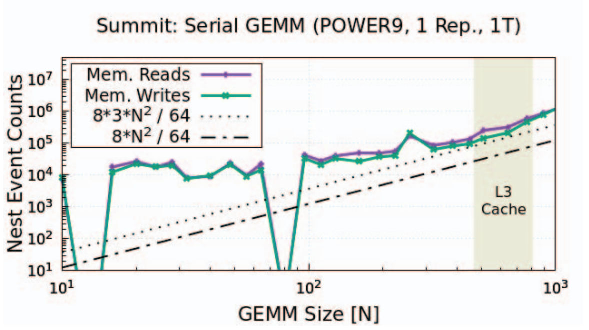
As shown by Figure 2a, when measuring the behavior of a single-threaded (serial) GEMM kernel, neither the reading nor writing memory traffic adhere to the expectations. We also observe this to be the case when we repeat the experiment on our testbed, Tellico, a system in which we have privileged access to the nest counters (see Figure 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.

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 [9]. But how many repetitions are necessary?

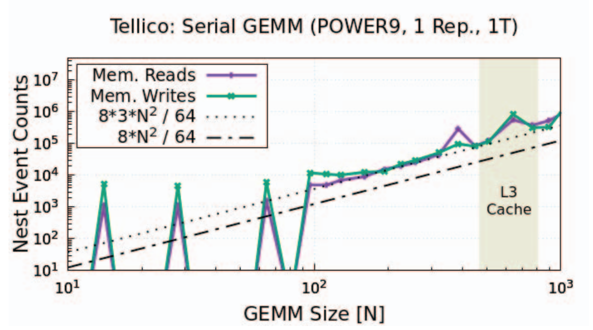
From inspection of Figures 2a and 2b, the memory traffic deviates less from the expectation for larger GEMM operations (while they still fit in the cache). This makes intuitive sense because there is more memory traffic for larger problem sizes. Hence, it takes more time for the operation to execute, 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 3a, we adapt the number of repetitions for a given problem size (*N*) per Equation 5.

$$Repetitions(N) = \begin{cases} \lfloor 514 - 0.246 \times N \rfloor, & N < 2048 \\ 10, & N \geq 2048 \end{cases} \tag{5}$$



(a) PCP Events.



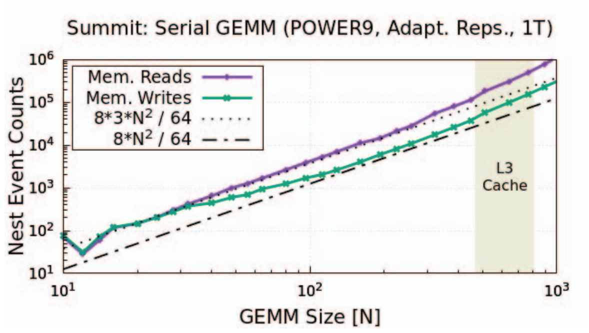
(b) Perf_uncore events.

Fig. 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.

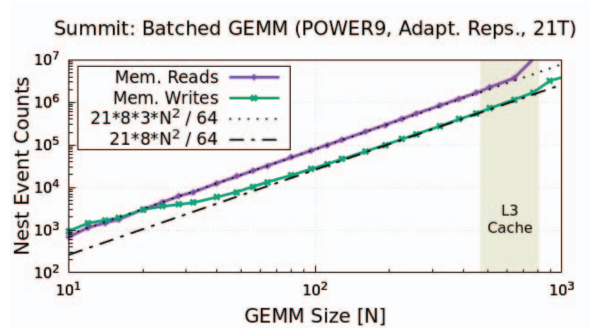
This is a simple formula for ensuring that we will execute around 500 repetitions for small matrix sizes and linearly drop to 10 repetitions for the largest sizes. These numbers are design decisions based on empirical observations in Figures 2a and 2b and are not the unique solution of some underlying formula. Other qualitatively similar numbers of repetitions would also work. In Section IV, 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 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 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 II-B).

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 events directly on the Tellico

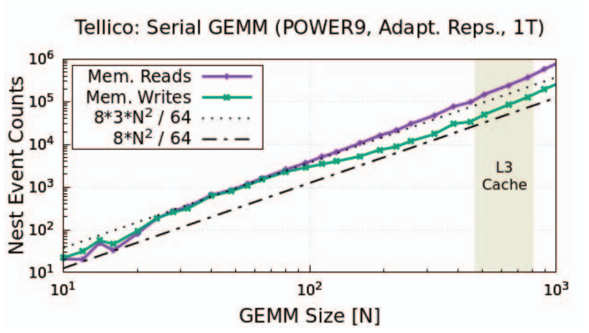


(a) Single-threaded GEMM.

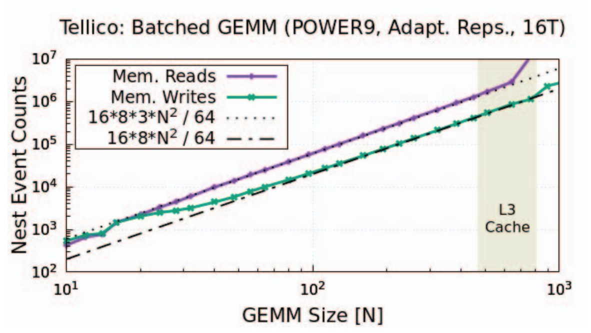


(b) Batched GEMM.

Fig. 3: Memory traffic of (a) single-threaded GEMM versus (b) Batched GEMM on IBM POWER9 measured with PCP events.



(a) Single-threaded GEMM.



(b) Batched GEMM.

Fig. 4: Memory traffic of (a) single-threaded GEMM versus (b) Batched GEMM on IBM POWER9 measured with perf_uncore events.

testbed, where we have elevated privileges and thus access to perf_uncore events; the results are shown in Figures 4a and 4b.

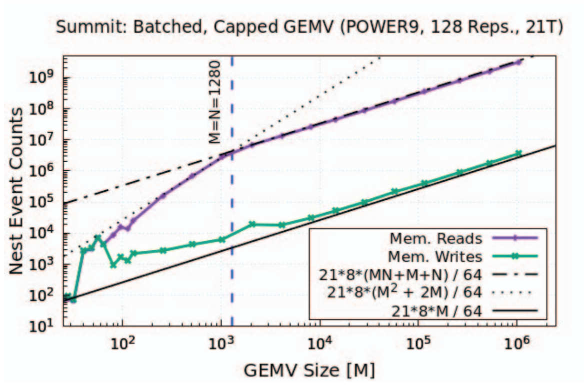
As before, there is more memory traffic than expected for larger problem sizes and a gradual deviation from the expectation in the single-threaded execution, despite not using PCP, and this deviation disappears when keeping all cores busy.

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.

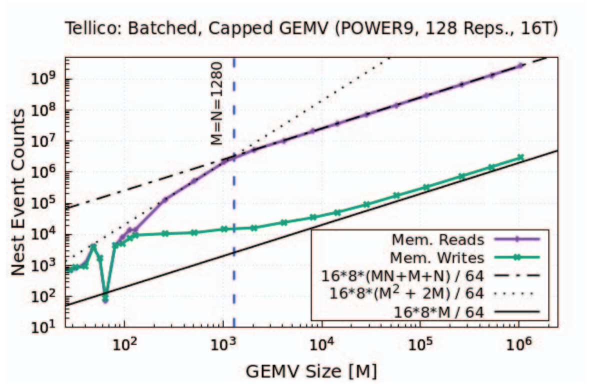
In Figure 5a, we show the results of running the capped GEMV kernel. For small sizes we use a square matrix A (*i.e.*, M=N=P) and therefore the kernel performs a regular GEMV operation. When the size M reaches the point where the matrix exceeds the size of the L3 cache, then we fix the width (N) of the 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 ($M^2 + 2\times M$), and for larger sizes the reading traffic matches what is expected for a capped GEMV ($M\times N+M+N$). 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 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 PCP-related nor POWER9-specific phenomenon. Figures 5a and 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 performance hardware 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 paper and has implications for application developers who wish to understand the behavior of their memory-bound codes.



(a) PCP Events.



(b) Perf_uncore Events.

Fig. 5: Memory traffic of batched, capped GEMV on POWER9 measured with (a) PCP events, versus (b) perf_uncore events.

IV. 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 [11], [12] executed. The 3D-FFT is a workhorse kernel utilized by various applications, such as HACC [13], GESTS [14], and QMCPACK [15], among others. This implementation utilizes both CPU and GPU cores in addition to network communication. The 3D-FFT is defined as

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

$$(0 \le u,v,w \le N-1) \tag{6}$$

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_1st_planewise_forward (S1PF)
- store_2nd_colwise_forward (S2CF)
- 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 6-9 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.

A. *S1CF*

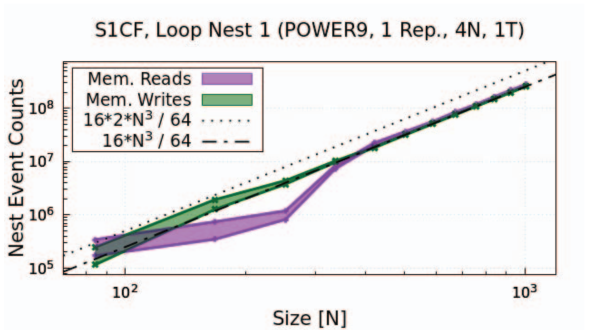
```
#pragma omp parallel for schedule(static)
for ( plane = 0; plane < PLANES; plane++ )
  for ( row = 0; row < ROWS; row++ )
    for ( col = 0; col < COLS; col++ )
      tmp[plane][row][col]
      = in[plane*ROWS*COLS + row*COLS + col];
```

Listing 5: S1CF, Loop Nest 1

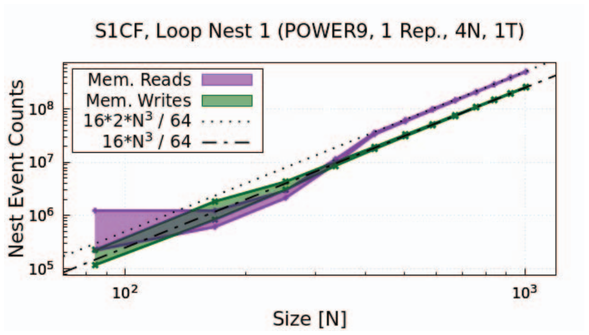
This first loop nest in S1CF copies the contents of the 1D array `in` into the 3D array `tmp`. Since this routine is executed once per MPI rank, `in` and `tmp` each contain $\frac{N}{r} \times \frac{N}{c} \times N =$ `PLANES`×`ROWS`×`COLS` *double complex* elements, each of which are 16 bytes. Aggregating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

As we discussed in Section III, the event measurements from the DGEMM shown in Figure 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 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 cache blocks" [16]. 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 6.

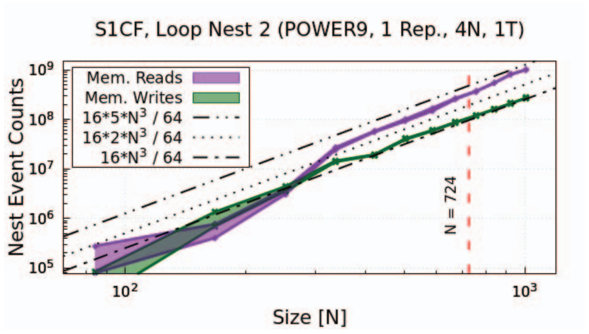


(a) **S1CF, Loop Nest 1** with no additional compiler optimizations.

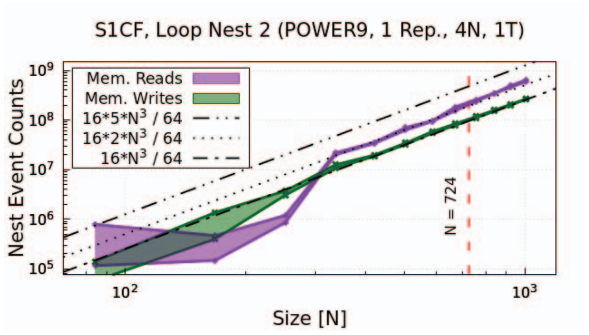


(b) **S1CF, Loop Nest 1** with compiler optimization *-fprefetch-loop-arrays*.

Fig. 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*.

Fig. 7: Memory Traffic of Loop Nest 2 in S1CF.

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

Listing 6: S1CF, Loop Nest 1 Assembly

The **dcbtst** instruction enables software prefetching by "[causing] a single-line prefetch into the L3 cache" [10]. This forces `tmp` to be read into the cache from memory. In Figure 6b, we observe the effect of this assembly instruction, as there are now two reads: one for `in` and also for `tmp`.

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

Listing 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 7a shows that we measure the expected one write per iteration of the innermost loop shown in Listing 7, but we measure more than the expected reads. Listing 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:

$$4 \times \frac{16 \times N^2}{8} + \frac{16 \times N^2}{8} = 5 \times 1024^2 \implies N \approx 724 \qquad (7)$$

For $N > 724$, an entire cache line must be read per iteration of the innermost loop in Listing 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 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 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 8: S1CF, Combined Loop Nest

Listing 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 8.

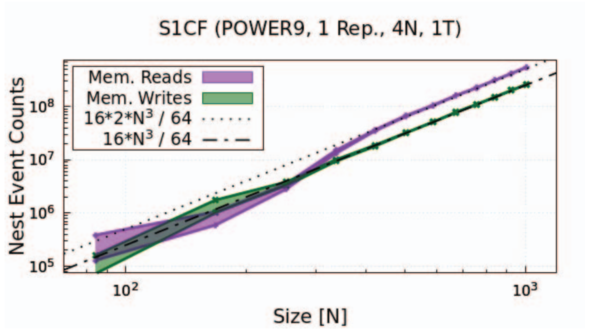


Fig. 8: **S1CF** with no additional compiler optimizations.

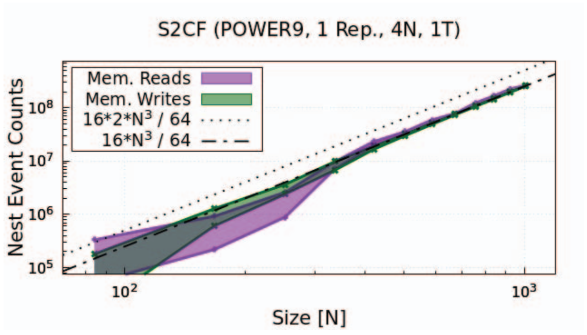
B. S2CF

```
/* 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];
```

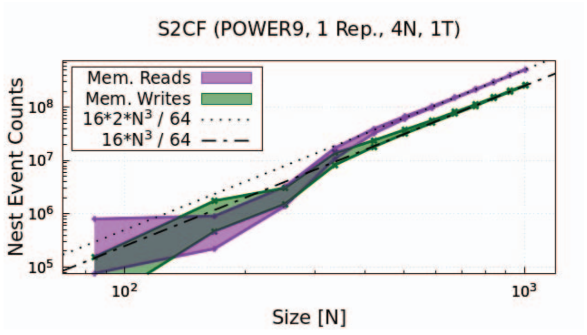
Listing 9: S2CF

The loop nest in Listing 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 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.



(a) **S2CF** with no additional compiler optimizations.



(b) **S2CF** with compiler optimization *-fprefetch-loop-arrays*.

Fig. 9: Memory Traffic of S2CF.

For a larger-scale job, of which the results are shown in Figure 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.

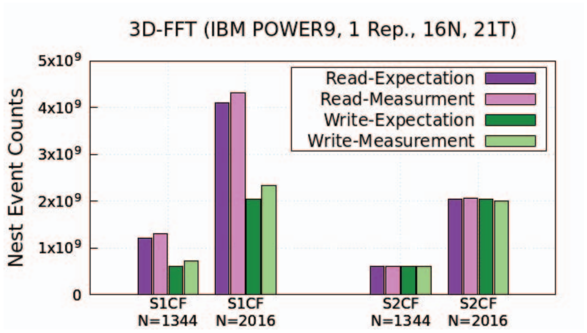


Fig. 10: Performance of **S1CF** and **S2CF**.

C. 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.

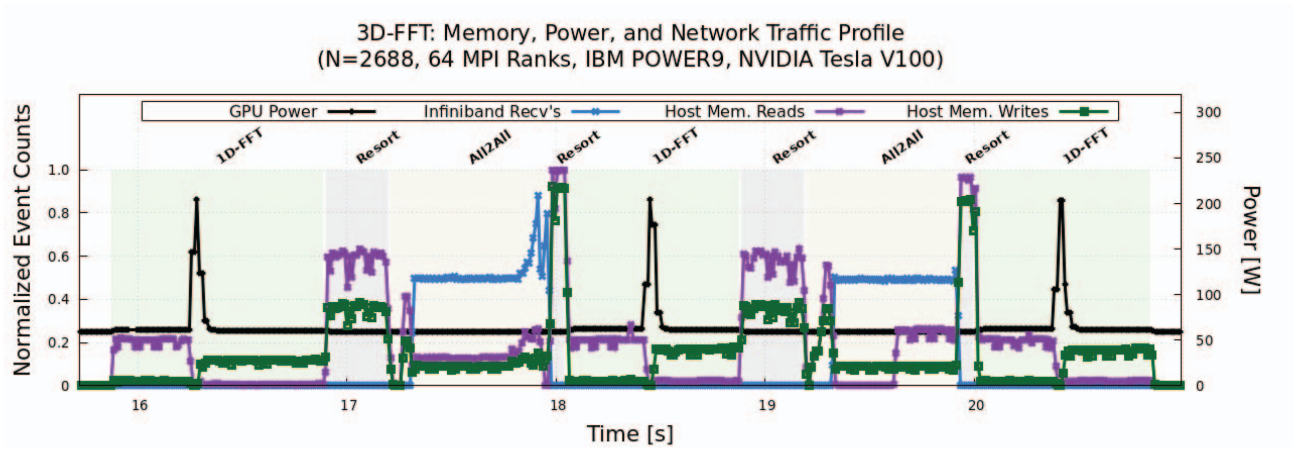


Fig. 11: Performance profile of a single 3D-FFT rank.

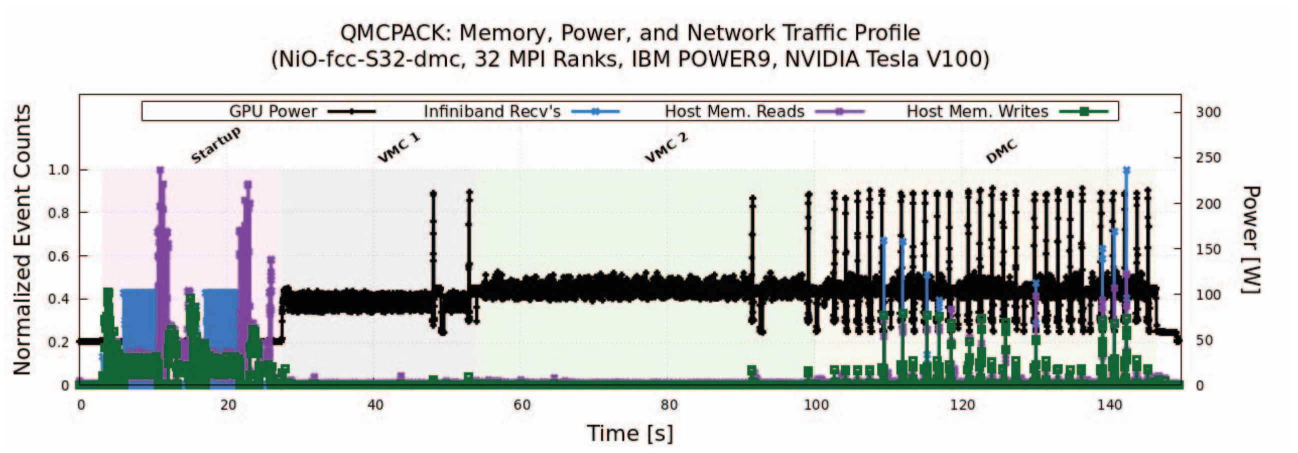


Fig. 12: Performance profile of a single QMCPACK rank.

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 II. The performance profile for this experiment is shown in Figure 11.

TABLE II: Supplemental Performance Events

Hardware	PAPI Component	Performance 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

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. 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 native 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.* [15]. The example problem [17] 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 12 demonstrates that the different stages in the execution of QMCPACK are distinguishable by monitoring separate hardware components simultaneously. Figure 11 is qualitatively equivalent to Figure 12, reinforcing the objective of PAPI's multi-component monitoring capabilities for more fine-grained insights into applications' performance running on heterogeneous HPC systems.

V. Conclusion and Future Work

In this paper 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 paper 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 backend 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 nest and uncore hardware events than those solely related to memory traffic.

Acknowledgment

We thank the anonymous reviewers for their improvement suggestions. 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 award No. 1900888 "ANACIN-X."

References

[1] "Performance Co-Pilot (PCP)," https://pcp.io/index.html.
[2] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting Performance Data with PAPI-C," *Tools for High Performance Computing 2009*, pp. 157–173, 2009.
[3] S. S. Shende and A. D. Malony, "The Tau Parallel Performance System," *Int. J. High Perform. Comput. Appl.*, vol. 20, no. 2, pp. 287–311, May 2006.
[4] M. Schlütter, P. Philippen, L. Morin, M. Geimer, and B. Mohr, "Profiling Hybrid HMPP Applications with Score-P on Heterogeneous Hardware," in *Parallel Computing: Accelerating Computational Science and Engineering (CSE)*, ser. Advances in Parallel Computing, vol. 25. IOS Press, 2014, pp. 773 – 782.
[5] H. Brunst and A. Knöpfer, "Vampir," in *Encyclopedia of Parallel Computing*, D. Padua, Ed. Springer US, 2011, pp. 2125–2129.
[6] D. Boehme, T. Gamblin, D. Beckingsale, P.-T. Bremer, A. Gimenez, M. LeGendre, O. Pearce, and M. Schulz, "Caliper: Performance introspection for hpc software stacks," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 550–560.
[7] IBM Corporation, "Power9 performance monitor unit user's guide, 2018," https://wiki.raptorcs.com/w/images/6/6b/POWER9_PMU_UG_v12_28NOV2018_pub.pdf.
[8] OLCF, "Summit user guide," https://docs.olcf.ornl.gov/systems/summit_user_guide.html.
[9] D. Barry, A. Danalis, and H. Jagode, "Effortless monitoring of arithmetic intensity with papi's counter analysis toolkit," in *Tools for High Performance Computing 2018 / 2019*. Cham: Springer International Publishing, 2021, pp. 195–218.
[10] IBM Corporation, "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, 2018.
[11] H. Jagode, "Fourier transforms for the bluegene / l communication network," Master's thesis, EPCC, The University of Edinburgh, 2006.
[12] H. McCraw, D. Terpstra, J. Dongarra, K. Davis, and R. Musselman, "Beyond the CPU: Hardware Performance Counter Monitoring on Blue Gene/Q," in *Proceedings of the International Supercomputing Conference 2013*, ser. ISC'13. Springer, Heidelberg, June 2013, pp. 213–225.
[13] S. Habib, V. Morozov, N. Frontiere, H. Finkel, A. Pope, and K. Heitmann, "Hacc: Extreme scaling and performance across diverse architectures," in *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–10.
[14] K. Ravikumar, D. Appelhans, and P. K. Yeung, "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*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356209
[15] J. Kim *et al.*, "Qmcpack: An open source ab initio quantum monte carlo package for the electronic structure of atoms, molecules and solids," *Journal of Physics Condensed Matter*, vol. 30, no. 19, Apr. 2018.
[16] IBM Corporation, "Power isa version 3.0 b," https://ibm.box.com/s/1hzcwkwf8rbju5h9iyf44wm94amnlcrv.
[17] NVIDIA Corporation, "Qmcpack," https://catalog.ngc.nvidia.com/orgs/hpc/containers/qmcpack.