COEN 193 Report Winter

Anthony Bryson

March 2024

1 Introduction

This quarter for COEN 193 I continued research on hardware events resulting from the execution of target parallel applications. My goal was to determine the significance of events in relation to the overall efficiency loss of target applications. I picked up where I left off last quarter with utilizing PAPI [1] to measure the hardware counters. I also made use of Dashing [2, 3] to help analyze the data gathered.

2 Progress

This quarter I began by polishing up my initial integration of the PAPI library in the existing test applications. Once that was working as intended, I was able to look into the events supported by the WAVE HPC [4] and compile a list of the events I would gather counts for. After Dr. Cho and I had discussed the event selection, I ran the test applications and gathered data which was then put into the Dashing framework for analysis.

2.1 Polishing and Corrections

Polishing involved adjusting the test applications to correctly measure the per thread hardware counts. My implementation from the previous quarter only measured the counts of the master thread. With this change, I decided upon averaging the counts gathered by each thread, however this was later changed to a sum to better reflect how work was divided between the threads.

I also made changes to how the execution time was measured. Since each problem size was executed multiple times due to the limited number of events that could be measured simultaneously, I averaged the execution time of each iteration. This value was then used as the time for the specific problem size.

2.2 Selecting Events

PAPI provides documentation on the events it is capable of measuring hardware counters for. It also comes with binaries to list which of them are supported by the hardware. For this research, all events selected and measured were from the WAVE cpu nodes.

Using the PAPI binaries provided me with a list of events. I ran tests with these events and found some did not work on the cpu nodes. Additionally, many events were aliases of other events. So I spent time going through each event, familiarizing myself with what it was counting and whether or not it was a duplicate, before filtering out a large number of them. I provided this list along with descriptions of what each event was to Dr. Cho, who helped me refine it further to 50 events of interest.

2.3 Gathering Data

With PAPI set up correctly and the events selected, I was able to run the test applications to gather hardware counter data. The applications I gathered data for are *dgemm*, *dsymm*, and *dgeqrf*, each with a version using the OpenBLAS library [5] and the Intel MKL library [6] as a backbone to the LAPACK [7] calls. Additionally I implemented my own naive *dgemm* algorithm as a baseline. This version made use of parallelism through OMP, but did not include any further optimizations such as blocking, tiling, or advanced algorithms.

For each version I varied the problem size as well as the number of threads. I used different combinations of 10k, 20k, and 30k for the values of m, n, and k for *dgemm* and m and n for *dsymm*. For *dgeqrf* I first ran the application on large square matrices with similar values for m and n. Later I ran it on tall rectangular matrices with values of 30k, 50k, and 70k for m by 1k and 4k for n.

For each of the applications, I ran them using 1, 4, 12, 20, 36, and 48 threads. The cpu nodes in the HPC have 48 cores which can hyperthread to have 96 threads. I decided to limit the number of threads in my tests to 48 to avoid the cores hyperthreading.

2.4 Using Dashing

Dashing is an analysis framework and visualization tool for performance counters. It calculates belief values for each count, which are then aggregated based on which resource they correspond to, so that linear regression can be performed. It then generates a number graphs that show the correlation that specific resources have to the efficiency loss of the application as it scales.

Dashing supports reading performance counter data from a csv file, however it requires a very specific format as described in the documentation. I found it easiest to run all my tests first for the various problem sizes, then reformat the output data with a script.

There were a few additions that had to be made before I could run the framework to analyze this data. The first is a result of Dashing grouping events based on which resource they utilized. I needed to provide a mapping of the events I measured to their respective resource. The second was a config file to specify which graphs to generate, which data to analyze, and how many threads I used.

3 Results

For each of the *dgemm*, *dsymm*, and *dgeqrf* applications I had Dashing generate a number of graphs. This included line graphs that show the normalized efficiency loss and hardware counts as the number of threads increased, as well as a sunburst graph that showed the normalized similarity between resource counts and efficiency loss. It should be mentioned for the sunburst graphs that each resource is scaled based on its correlation score as a percentage of the total correlation score for that specific problem size. So a single resource making up a significant portion of the section for its problem size means it has a high correlation compared to the other resources for that problem, but not necessarily a high correlation in general.

3.1 DGEMM

Figures 1 to 3 are the sunburst graphs for the various *dgemm* applications. From these we can see events regarding L2 cache are most correlated to the efficiency loss across all problem sizes, however L1 data cache events are also relatively significant for the MKL application and for smaller matrices on the other two applications.

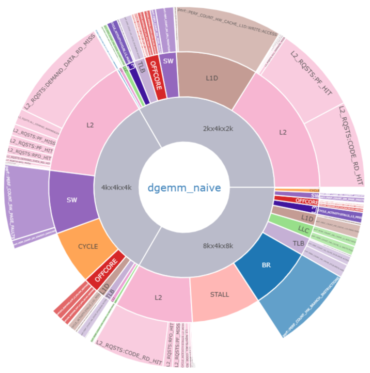

Figure 1: Sunburst graph of relative correlation of events to efficiency loss for naive implementation of DGEMM

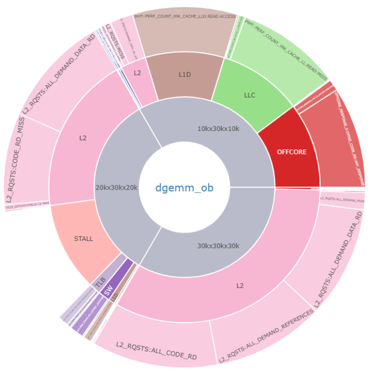

Figure 2: Sunburst graph of relative correlation of events to efficiency loss for OpenBLAS implementation of DGEMM

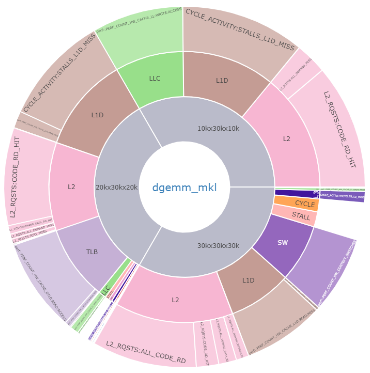

Figure 3: Sunburst graph of relative correlation of events to efficiency loss for Intel MKL implementation of DGEMM

Looking then to the line graphs in Figures 4 to 6 we can visualize just how correlated L2 cache and L1 data cache are to the efficiency loss. Each graph displays a number of lines representing each hardware counter. I've disabled all the lines except those for events on L2 cache to make it more understandable.

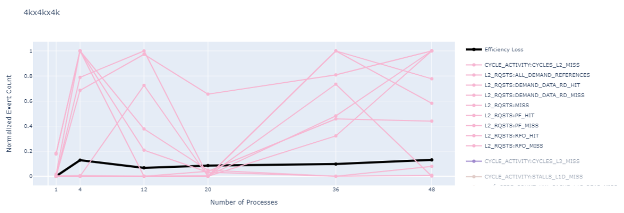

Figure 4: Normalized L2 cache hardware counts and efficiency loss for naive DGEMM implementation

Comparing these hardware counter lines to the efficiency loss line, we can see the counter values do not grow in a similar manner to the efficiency loss. This means there is a low correlation between the two, despite L2 being the most correlated of any of the resources I measured. This could be due to *dgemm* being a highly optimized routine such that single events or resources don't significantly contribute to efficiency loss.

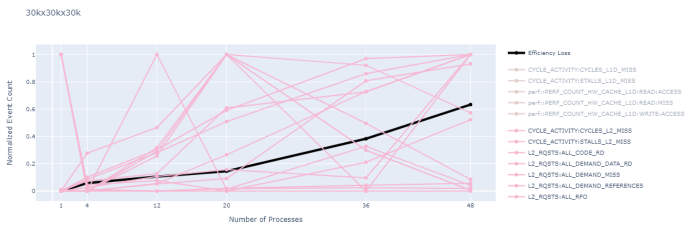

Figure 5: Normalized L2 cache hardware counts and efficiency loss for OpenBLAS DGEMM implementation

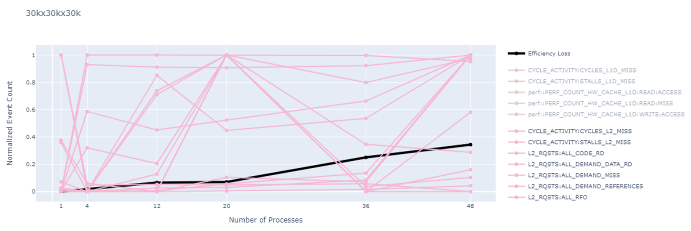

Figure 6: Normalized L2 cache hardware counts and efficiency loss for Intel MKL DGEMM implementation

3.2 DSYMM

The sunburst figures for the *dsymm* data reveal a much simpler relative correlation across all the problem sizes I tested. As seen in Figures 7 and 8, L2 cache events are by far the most correlated to efficiency loss of all events. The one noticeable exception to this is for the $m = 30k$, $n = 30k$ matrices with the Intel MKL implementation, which shows L1 data cache to be more correlated.

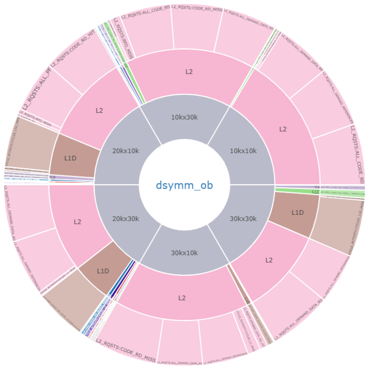

Figure 7: Sunburst graph of relative correlation of events to efficiency loss for OpenBLAS implementation of DSYMM

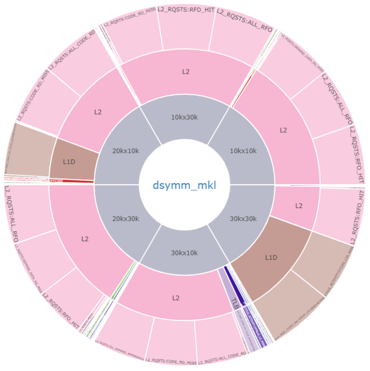

Figure 8: Sunburst graph of relative correlation of events to efficiency loss for Intel MKL implementation of DSYMM

So examining the line graph for this $m = 30k$, $n = 30k$ size in Figure 10, we can see a few lines grow similarly to efficiency loss, those being `CYCLE_ACTIVITY:CYCLES_L1D_MISS`[1], `perf::PERF_COUNT_HW_CACHE_L1D:READ:ACCESS`[2], and `L2_RQSTS:RFO_HIT`[3]. These three events are the same events listed on the outer ring of the corresponding section of Figure 8. So we can tell that these events are highly correlated to the efficiency loss for this specific problem size.

[1] `CYCLE_ACTIVITY:CYCLES_L1D_MISS` Stalled cycles with pending L1D load cache misses
[2] `perf::PERF_COUNT_HW_CACHE_L1D:READ:ACCESS` L1 data cache read hits
[3] `L2_RQSTS:RFO_HIT` RFO requests that hit L2 cache

It is worth noting that these events are not as highly correlated for the other problem sizes I tested, as they are not listed on the sunburst graph for most of these. However, performing a similar analysis on the OpenBLAS implementation using Figures 7 and 9 reveals a single event with a highly similar growth pattern, that being the line for `L2_RQSTS:DEMAND_DATA_RD_MISS`[4].

My main takeaway from this is while the event counts overall appear to be inconsistent when comparing to the efficiency loss, L2 cache events as a whole, and to a lesser degree L1 data cache events, have a possibly significant correlation to the efficiency loss of the *dsymm* application.

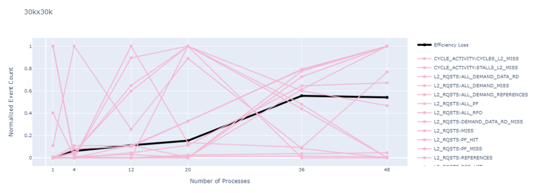

Figure 9: Normalized L2 cache hardware counts and efficiency loss for OpenBLAS DSYMM implementation

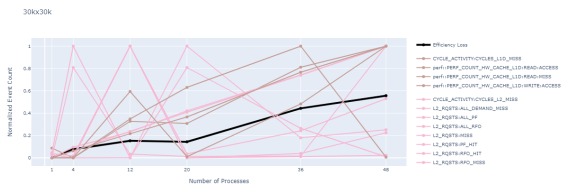

Figure 10: Normalized L1D and L2 cache hardware counts and efficiency loss for Intel MKL DSYMM implementation

3.3 DGEQRF

Looking at the graphs for *dgeqrf* on the tall rectangular matrices revealed much more interesting results. First for the OpenBLAS version in Figure 11, we can see another instance of L2 cache being the main correlated resource. However, upon analyzing the line graph in Figure 12, not only can we see a significant efficiency loss curve, but also three lines of the hardware counts very closely following this growth pattern. The lines of `L2_RQSTS:ALL_DEMAND_DATA_RD`[5], `L2_RQSTS:ALL_DEMAND_REFERENCES`[6], and `L2_RQSTS:RFO_HIT` are all very similar to the efficiency loss. Moreover, these three events are the three main events listed in the outer ring of the sunburst graph, regardless of the problem size. So it is possible that these three events are large contributors of the OpenBLAS implementation's efficiency loss.

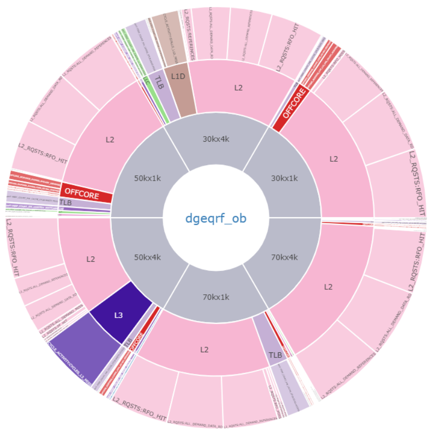

Figure 11: Sunburst graph of relative correlation of events to efficiency loss for OpenBLAS implementation of DGEQRF

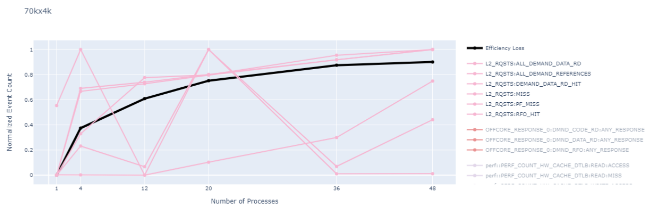

Figure 12: Normalized L2 cache hardware counts and efficiency loss for OpenBLAS DGEQRF implementation

[4]`L2_RQSTS:DEMAND_DATA_RD_MISS` Demand Data Read requests that miss L2 cache
[5]`L2_RQSTS:ALL_DEMAND_DATA_RD` Any data read request to L2 cache
[6]`L2_RQSTS:ALL_DEMAND_REFERENCES` All demand requests to L2 cache

Analyzing the Intel MKL version of *dgeqrf* in Figure 13 we can see a much more even split of relative correlation among resources. On top of L2 cache events, there are also TLB and Branch events that are present in most problem sizes. Looking then to the line graph in Figure 14 we can see a much slower growing efficiency loss curve than with the OpenBLAS version. However none of the event counts appear to be growing very similarly to the efficiency loss. It is likely that the Intel MKL implementation of *dgeqrf* improved whatever was causing the high efficiency drop off which subsequently reduced counts of the three highly correlated events in the OpenBLAS version.

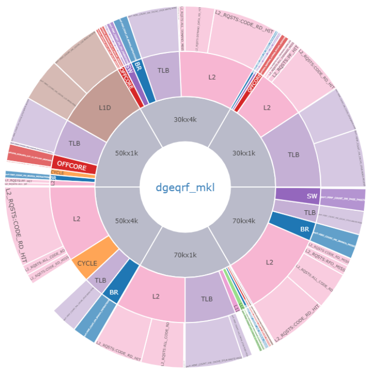

Figure 13: Sunburst graph of relative correlation of events to efficiency loss for Intel MKL implementation of DGEQRF

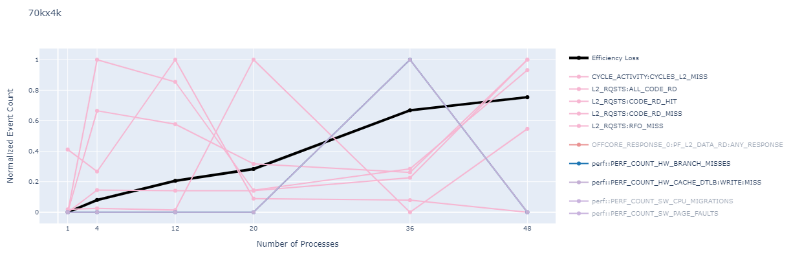

Figure 14: Normalized L1D and L2 cache hardware counts and efficiency loss for Intel MKL DSYMM implementation

4 Future Work

For all of my experiments this quarter I focused on varying the problem size and the number of threads the applications used. Other variables, such as where threads were mapped to and whether they are bound, were held constant across all the tests. In the next quarter I plan to experiment with changing these, along with other variables, to see how they affect the hardware counts. This could include testing how the applications perform when hyperthreading as well.

Aside from expanding these constraints, I plan to experiment more with the L2 cache events because they were prevalent in all the applications I tested. If possible, I would like to better understand their significance in the overall efficiency loss.

References

[1] H. Jagode, A. Danalis, J. Dongarra, D. Barry, G. Congiu, and A. Pal, *PAPI: Performance Application Programming Interface*. The University of Tennessee, Knoxville, 2023. http://icl.utk.edu/papi/.

[2] T. Z. Islam, A. Ayala, Q. Jensen, and K. Z. Ibrahim, "Toward a programmable analysis and visualization framework for interactive performance analytics," in *2019 IEEE/ACM International Workshop on Programming and Performance Visualization Tools (ProTools)*, pp. 70–77, 2019.

[3] T. Z. Islam, J. J. Thiagarajan, A. Bhatele, M. Schulz, and T. Gamblin, "A machine learning framework for performance coverage analysis of proxy applications," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 538–549, 2016.

[4] Santa Clara University, *WAVE High Performance Computing (HPC)*, 2023. https://wiki.wave.scu.edu/doku.php?id=wave_hpc.

[5] Z. Xianyi, M. Kroeker, W. Saar, W. Qian, Z. Chothia, C. Shaohu, and L. Wen, *OpenBlas: An optimized BLAS library*, 2023. OpenBLAS 0.3.24.

[6] Intel Corporation, *Intel®-Optimized Math Library for Numerical Computing on CPUs & GPUs*, 2023. https://software.intel.com/content/www/us/en/develop/tools/math-kernel-library.html.

[7] W. da Silva Pereira, I. Kozachenko, J. Langou, J. Demmel, J. Dongarra, and J. Langou, *LAPACK—Linear Algebra PACKage*, 2022. LAPACK 3.11.0 is a software package provided by Univ. of Tennessee, Univ. of California, Berkeley, Univ. of Colorado Denver and NAG Ltd..