COEN 193 Report Spring

Anthony Bryson

June 2024

1 Introduction

My focus for this quarter of COEN 193 was in further exploring the hardware event counts of target applications. My work from previous quarters revealed a link between L2 cache related events and the efficiency loss of applications as the number of threads grew. As such, I looked to learn more about this correlation and experiment with methods to reduce the counts of these events to hopefully reduce the efficiency loss.

As with previous quarters, all events were measured on the WAVE HPC [1] cpu nodes. I utilized PAPI [2] to gather the hardware event counts. I again made use of Dashing [3, 4] as a starting point for my analysis, then generated custom graphs to focus in on a few select events.

2 Progress

As discussed earlier, my work for this quarter was guided by the results I gathered in the winter. These results showed how L2 cache events scaled in a similar manner to the efficiency loss for the *dgeqrf* application. This likely means interactions with the L2 cache were related to the efficiency loss of this test application. However, I looked to further test this by rerunning *dgeqrf* with various thread placement and binding policies. Following these test, I also experimented with different block sizes to see what effect that would have.

2.1 Thread Mapping

To experiment with the thread placement and binding, I used the `OMP_PLACES` and `OMP_PROC_BIND` environment variables. These allowed me to set these policies and execute the application with all the same problem sizes. I ran *dgeqrf* with many combinations of values for these variables, namely `OMP_PLACES` $= \{cores, sockets, threads, \text{[unspecified]}\}$ and `OMP_PROC_BIND` $= \{close, spread\}$. The goal behind this was to see how these different policies affected the number of L2 misses and what impact his had on the overall runtime and scaling of the application. Since these can result in threads being placed in such a way that they have minimal shared L2 cache, I hope to see the number of L2 misses decrease, or at the least become less correlated to the efficiency loss as a whole.

2.2 Block Size

I also experimented with the block size of the underlying algorithm. For this specific application, I previously let the LAPACK [5] library calculate the ideal value for nb (block size), which ended up being 32. I decided to run the application with a different value for nb, being 64, to see how this affected the cache interactions. To do this, I went into the definition of *dgeqrf* and altered the computation of nb.

3 Results

Just as I had done with previous results, I first put the data into Dashing to get an idea of which resources were most correlated to the efficiency loss in these tests. From there I picked a few key events and plotted these alongside the runtime of the application to get an idea of how they each grow.

3.1 Thread Mapping

Figures 1 and 2 are the sunburst graphs from previous test runs of *dgeqrf* where the thread placement and binding policies were not specified. Figure 1 on OpenBLAS [6] shows the highest correlation is between L2 and efficiency loss. We are hoping to see this have a more even distribution when we map the thread placement to space out the threads. Figure 2 on Intel MKL shows a more even distribution in which events are most correlated, meaning out of the box MKL [7] is already well optimized to not have too many L2 misses. However, we will still look at how thread placement and binding affects this as well.

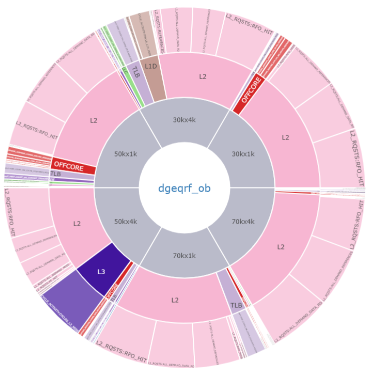


Figure 1: Sunburst graph of relative correlation of events to efficiency loss for OpenBLAS implementation of DGEQRF

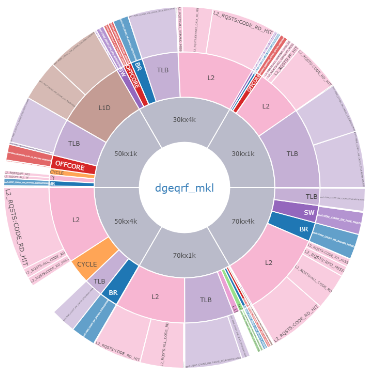


Figure 2: Sunburst graph of relative correlation of events to efficiency loss for Intel MKL implementation of DGEQRF

3.1.1 OpenBLAS

We will first look at the changes in resource correlations for the OpenBLAS backed applications. Running the application with OMP_PLACES $=$ *threads* produces a sunburst graph that is identical to that of Figure 1. This implies that when OMP_PLACES is not specified, it defaults to thread level granularity in the distribution of thread placements.

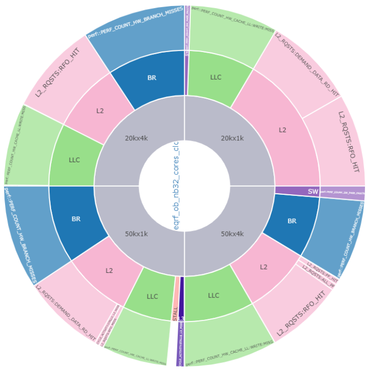


Figure 3: OpenBLAS with CORES and CLOSE parameters

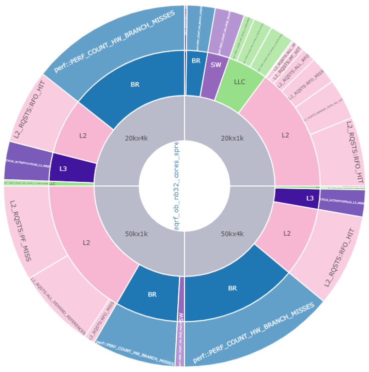


Figure 4: OpenBLAS with CORES and SPREAD parameters

In Figures 3 and 4 we can see these correlations when the application is run with OMP_PLACES $=$ *cores*. Both of these graphs show a much lower relative correlation with L2, and instead includes branch events in blue and LLC events in green. The important thing to look at here is the shift from fully L2 correlation to a balance of L2 and LLC correlation. This likely occurs since the thread placement policy prevents multiple threads from being run on the same core, meaning there are fewer threads sharing a specific L2 cache. With L2 events such as cache misses decreasing, the relative correlation of LLC with efficiency loss goes up.

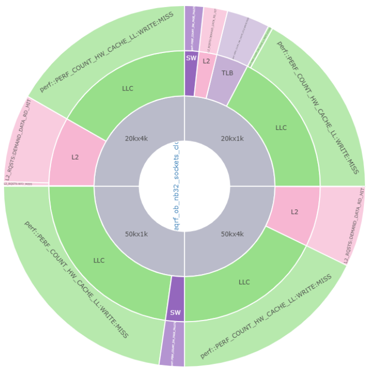


Figure 5: OpenBLAS with SOCKETS and CLOSE parameters

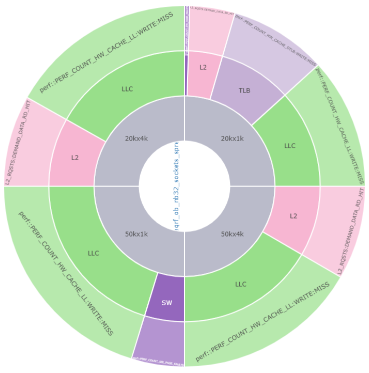


Figure 6: OpenBLAS with SOCKETS and SPREAD parameters

Figures 5 and 6 show the correlations when OMP_PLACES $=$ *sockets*. This causes threads to be the most spread out across the available cpu threads. These graphs reveal LLC to be the most correlated resource to the efficiency loss. I believe this is the result of the threads being very spaced out and allowing better utilization of the L2 cache. As a result, the LLC, which is shared by all cpu threads and therefore unaffected by thread placements, becomes much more correlated with the application's efficiency loss.

We can then look at graphs of the runtimes of the application as well as the hit rates for L2 and LLC. These are shown in Figure 7 and show an interesting pattern. As the placement goes from *threads* to *cores* to *sockets*, the overall runtime decreases slightly. Additionally, the LLC hit rate, shown in red, increases slightly when `OMP_PROC_BIND` = *spread*. While I don't understand the cause of this change in the hit rate, the decrease in runtime suggests L2 was a bottleneck in the application and changing the placement policy to *sockets* improved the efficiency the most.

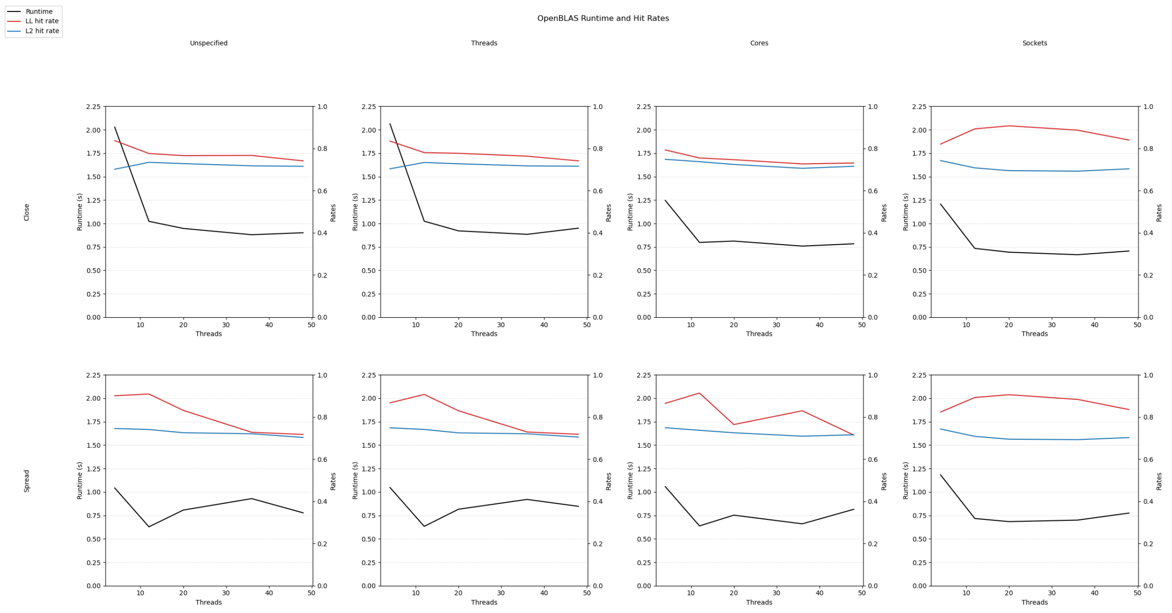


Figure 7: Runtimes and Hit Rates of DGEQRF with OpenBLAS and various thread policies

3.1.2 Intel MKL

Next we will look at how the correlations changed for the MKL backed applications. As discussed earlier, there was no dominating resource in terms of relative correlation to efficiency loss. The main ones seen when no thread placement policies were specified were L2 and TLB, with a number of lesser resources showing up seemingly randomly.

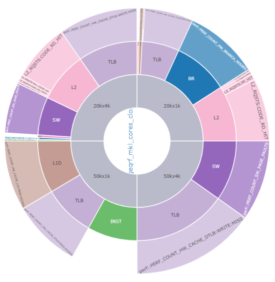


Figure 8: MKL with CORES and CLOSE parameters

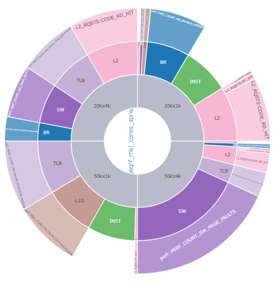


Figure 9: MKL with CORES and SPREAD parameters

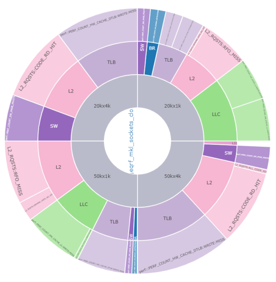


Figure 10: MKL with SOCKETS and CLOSE parameters

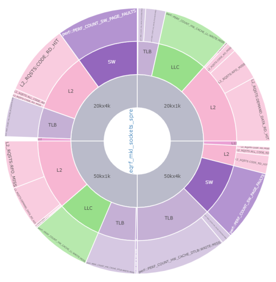


Figure 11: MKL with SOCKETS and SPREAD parameters

If we then look to Figures 8 to 11 where various policies have been applied, we can see a similar pattern across all of them. Specifically, L2 and TLB are sizeable slices for all problem sizes, with branch events, software events, and LLC appearing in only certain setups. So it is difficult to find any patterns in resource correlation if they do exist. We can also take a look at the runtimes and hit rate graphs in Figure 12. These graphs reveal the best runtimes came from the *cores* and *sockets* placements, regardless of the `OMP_PROC_BIND` policy.

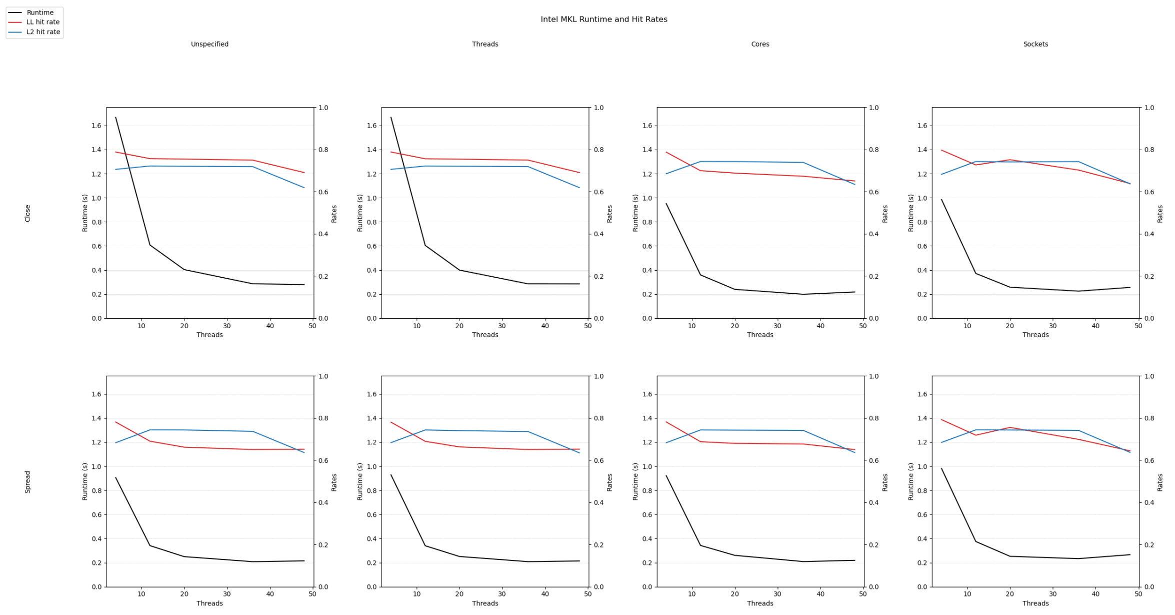


Figure 12: Runtimes and Hit Rates of DGEQRF with MKL and various thread policies

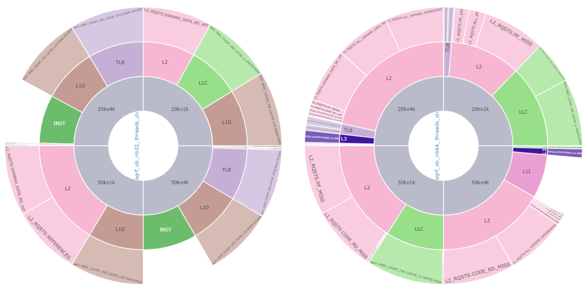
While there doesn't seem to be much to gather from these graphs, we can see that all placement and binding policies have very similar runtimes, with one exception. When `OMP_PLACES` = *threads* and `OMP_PROC_BIND` = *close*, we see a much worse runtime across all thread counts. This is likely due to all the threads being grouped together on the processor, whereas every other policy combination spaces them out in some capacity.

3.2 Block Size

The change in resource correlation when changing the block size to 64 has a seemingly different effect depending on the placement policy. The main takeaway from this is there doesn't seem to be any pattern to how increasing the block size changes the application's execution. There may not be much to learn from the resource correlation graphs, so we will just briefly look at them.

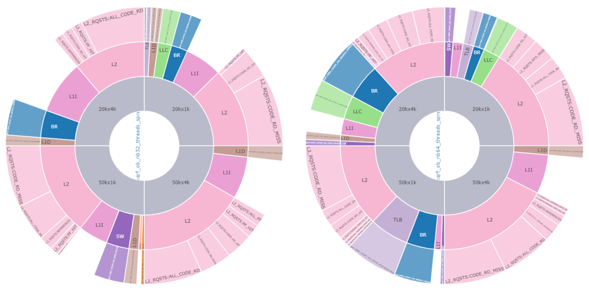
3.2.1 Threads

Figure 13 shows the change in correlation between $nb = 32$ and $nb = 64$ for the `OMP_PLACES` $=$ *threads* and `OMP_PROC_BIND` $=$ *close* policies. With the increase in block size, we can see the correlation of L1D events goes away and is largely replaced by L2 events. Figure 14 shows the same change in block size, but with `OMP_PROC_BIND` $=$ *spread*. In this setup, there doesn't appear to be any noteworthy changes in resource correlation.



(a) block size 32 **(b)** block size 64

Figure 13: OpenBLAS with THREADS and CLOSE parameters

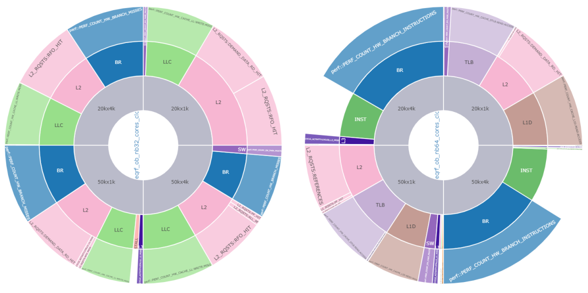


(a) block size 32 **(b)** block size 64

Figure 14: OpenBLAS with THREADS and SPREAD parameters

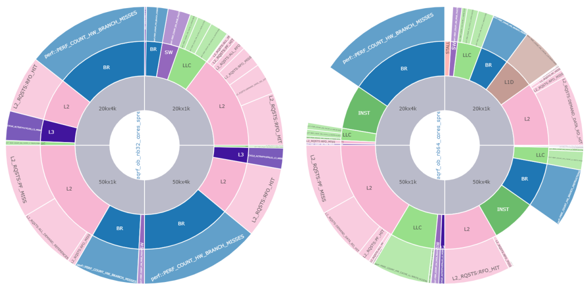
3.2.2 Cores

Next looking at the results for the policy `OMP_PLACES` $=$ *cores*, we can see more inconsistent results. Figure 15 shows a slight decrease in L2 event correlation, and Figure 16 shows a drop in the branch events and a minor correlation with LLC events.



(a) block size 32 **(b)** block size 64

Figure 15: OpenBLAS with CORES and CLOSE parameters



(a) block size 32 **(b)** block size 64

Figure 16: OpenBLAS with CORES and SPREAD parameters

Overall it doesn't seem there is anything to gather from analyzing the resource and efficiency loss correlation. However, despite no consistent pattern for these, the runtimes of the application were lower across the board for the larger block size.

3.2.3 Cache Hit Rates and Runtimes

If we look to Figure 17 for the hit rates of the L2 and LLC, they scale almost identically to the rates for the 32 block size shown in Figure 7. However, the LLC hit rates for the 64 block size application are actually slightly higher for every thread placement policy. The trade off appears to be an even smaller decrease in the L2 cache hit rate. Visually, the hit rates appear to increase by roughly 0.08 for LLC and decrease by about 0.02 for L2.

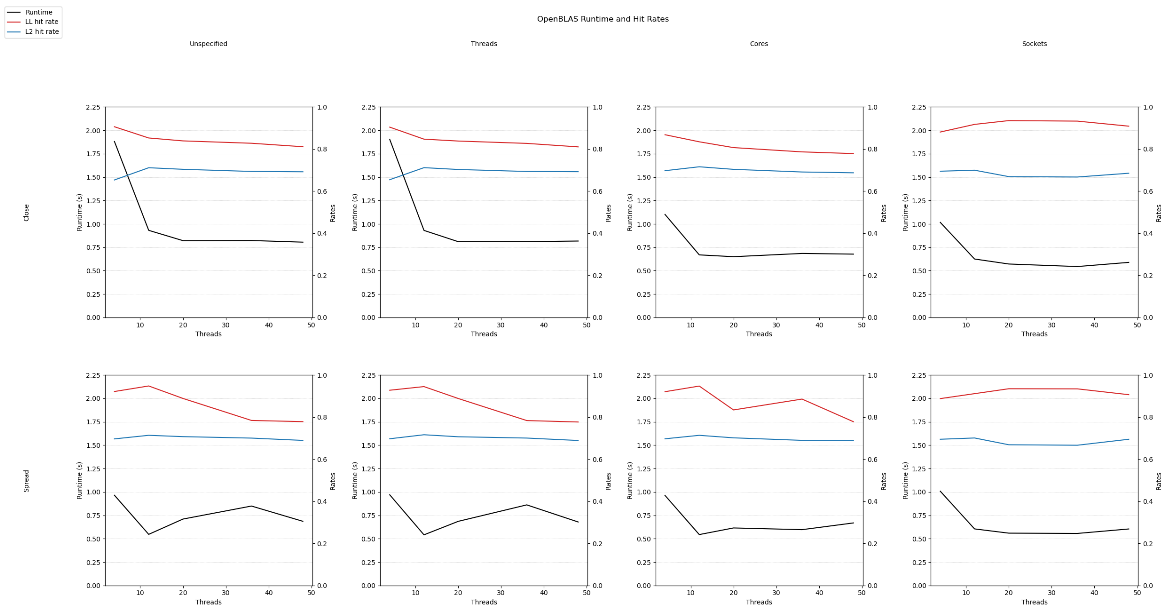


Figure 17: Runtimes and Hit Rates of DGEQRF with OpenBLAS, various thread policies, and block size of 64

Figures 7 and 17 show the runtimes and hit rates of the application with the problem size of $20k\ x\ 1k$. With the larger block size, we can see that alongside the changes in hit rates, there is also a decrease in the runtimes by about 10%. Just like with the hit rates, this is consistent across all policies. However, the same graphs for larger problems sizes, for example $50k\ x\ 1k$ and $50k\ x\ 4k$, show an even larger decrease in execution time, occasionally surpassing 20%. Therefore, we can conclude that the LAPACK calculated block size is not optimal for this specific application.

4 Conclusion

In my tests this quarter I experimented with various thread placement and binding policies for the *dgeqrf* application. While there was some impact on several resources, the most significant change was in the decrease in L2 cache events and increase in LLC events in terms of correlation to the efficiency loss. I found that the most important factor in the runtime of the application was ensuring that the policies spaced out the threads, which could be done using $\texttt{OMP\_PROC\_BIND} = \textit{spread}$ and/or $\texttt{OMP\_PLACES} = \textit{cores}/\textit{sockets}$.

I also experimented with using different block sizes for the OpenBLAS application. While the change in correlation between resources and efficiency loss was seemingly random, there was a decrease in runtime possibly due to the higher LLC hit rate. There is still room for experimenting with larger block sizes to find the optimal value for this application. As such, future steps should be taken in this direction.

References

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

[2] 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/.

[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] 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.

[5] 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..

[6] 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.

[7] 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.