Optimizing Matrix Multiplication

Group 12 - Alan Cheng (ayc48), Jason Setter (jls548), Saul Toscano (st684)

Introduction

In most programming, we use highly productive programming languages, such as Python and Java, to simply describe problems in a language the computer can understand after numerous translations. However, when we are concerned about execution performance, such as in high performance computing, we use special techniques to speed up the execution. We attempt to speed up very common, computationally intensive tasks by exploiting knowledge of microarchitecture and parallelizable work that may ignored in typical programming. This takes additional time and effort, but due to the high use of the code and difficulty of the computation, this additional effort is well worth our efforts.

As an example, we use problem of matrix multiplication. With a naive implementation, there are many computations used and performance is slow. However, with close manipulation of the order computations are performed and careful consideration to the processor cache, it is possible for over an order of magnitude improvement in performance.

Optimization

Overview

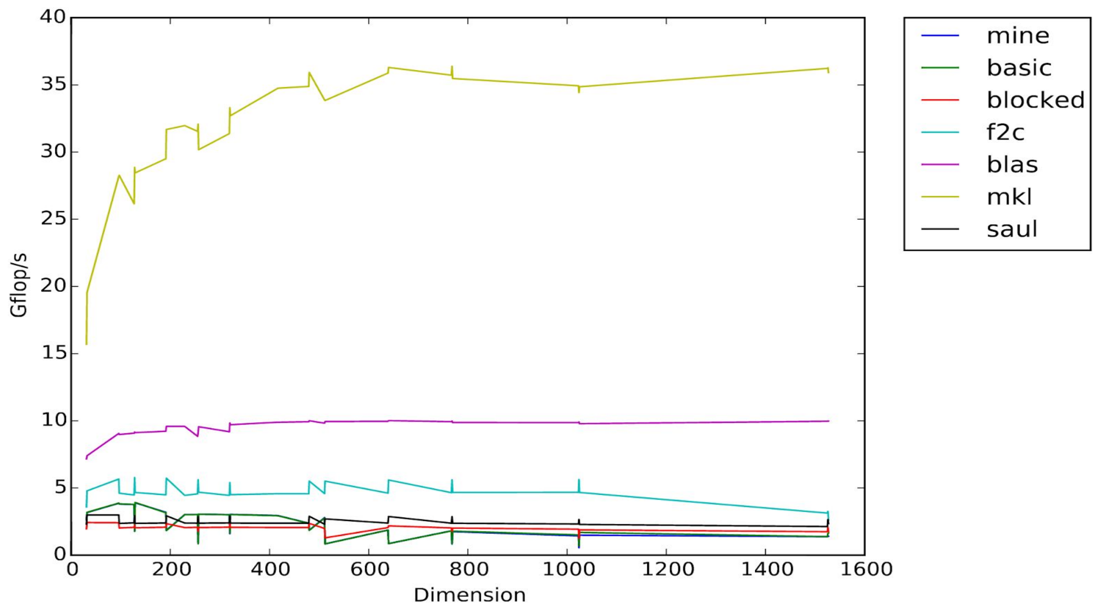
Before we can finely tune our matrix multiplication kernel, we must first try a variety of optimization strategies to see what works well and what does not. The methods we tried can be roughly organized into 4 categories: loop reordering, copy optimization (of which we tried two), compiler flags, and hierarchical blocking. We discuss each type of optimization technique in the remainder of this section.

We try these techniques separately at first, and then later integrate the ideas together, as well as add some additional optimizations inspired by the work of our peers.

Loop Reordering

First, we tried the provided blocking approach, but with j, k, i ordering instead of i, j, k ordering. We built on top of the blocked code because of its performance advantage over the basic three-loop version, and the reference Fortran DGEMM does better with j, k, i ordering[1], so we decided to try it for ourselves.

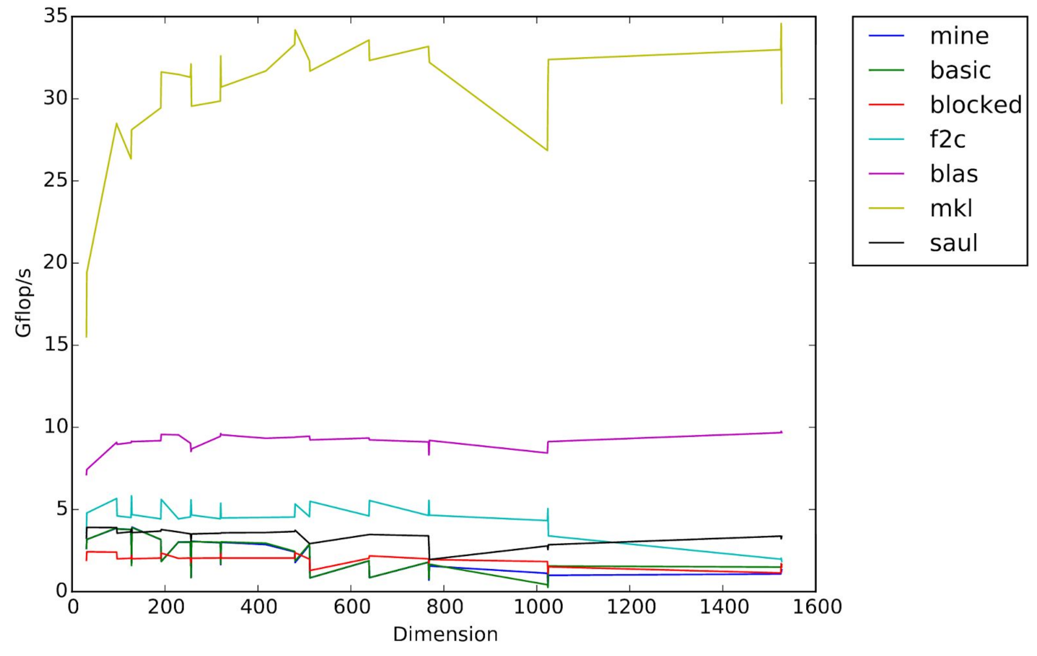
[1] http://www.cs.cornell.edu/~bindel/class/cs5220-f11/notes/serial-tuning.pdf



We can see in the plot that our method (black line-saul) does better than the naive blocked, basic and “mine” methods. However, the method has several conflict misses. Since our method is better than the naive blocked approach, we can conclude that the new order of the loops improved the method.

Copy Optimization

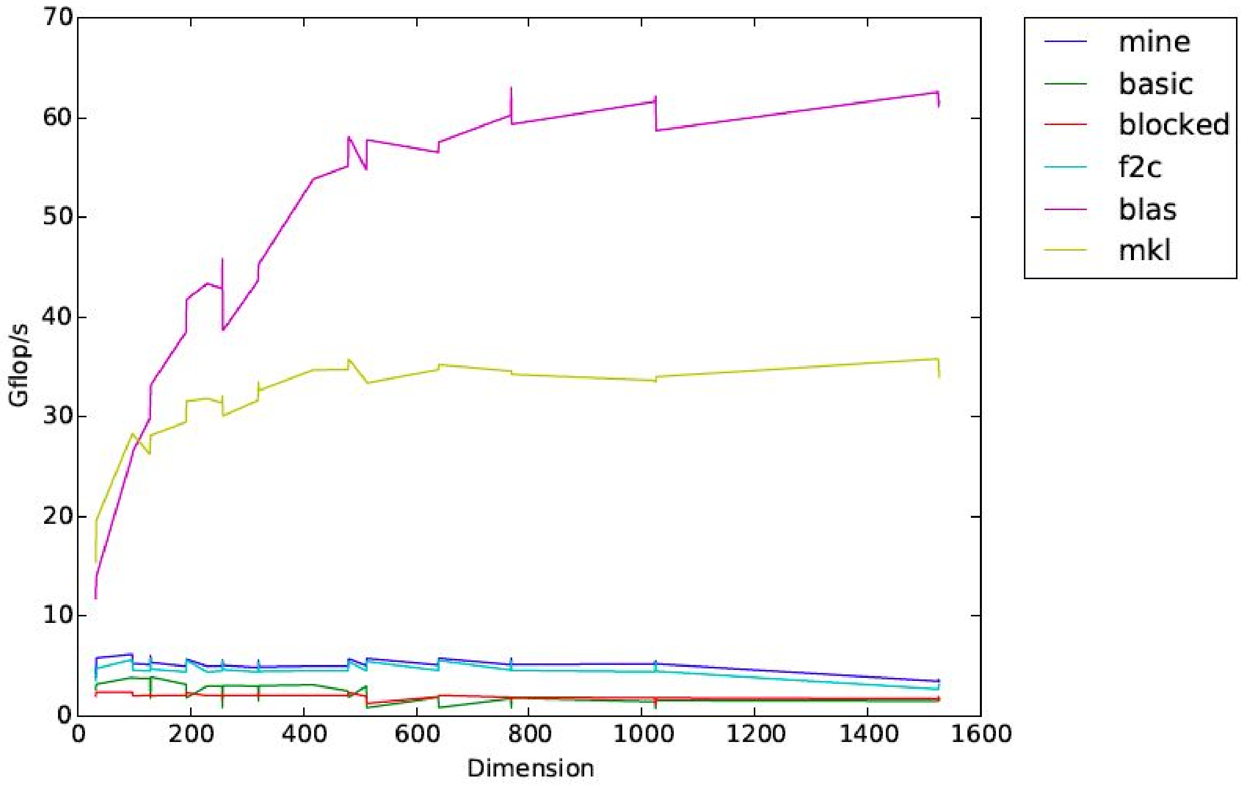
Next, we improved upon the loop-reordered code mentioned above by introducing copy optimization. More specifically, we copy the matrices C and B in this approach. We hoped that copying the data into contiguous blocks of memory would reduce the number of conflict misses.



We can see in the plot that the method (black line) does better than the naive blocked, basic, “mine” methods, our previous method. When the dimension is larger than 1200, our method outperforms the f2c method. Although this method did reduce the number of conflict misses, it still encountered several conflict misses.

Copy Optimization with Transpose

We also tried an alternative copy optimization technique where we transpose matrix A before performing the matrix multiplication. This time, for simplicity, we built off of the basic three-loop approach. By transposing matrix A (recall that our matrices are column-major), we can calculate each element of C by iterating over columns of A instead of rows of A, netting us unit stride rather than stride equal to the row length. By using this transformation, we hope that the structure of how we access the elements is beneficial to how the elements are stored in the cache, by bringing in the cache line of data we plan to use.

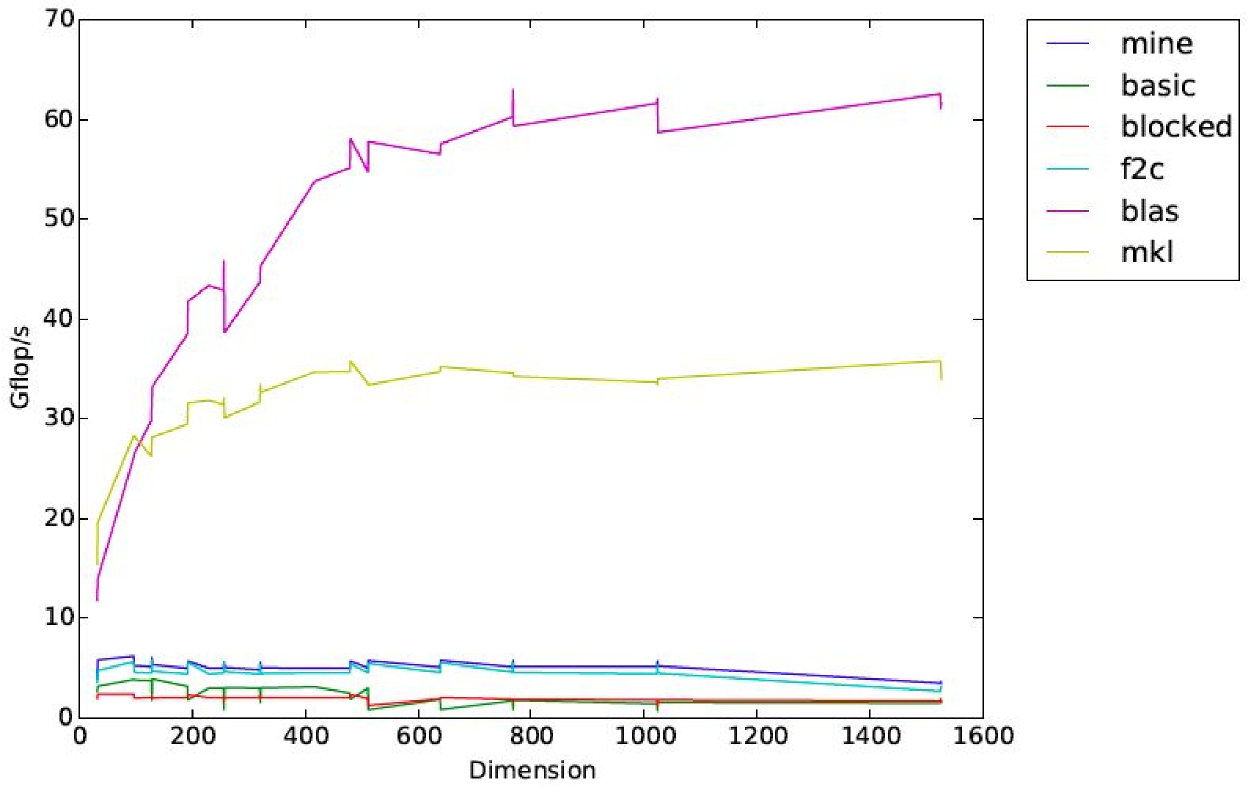


We observe that the “transpose A” approach (blue line, “mine”) improves the performance of the basic algorithm by approximately 2-3 times, despite the extra copying of the matrix. This suggests that the high stride of the basic method is a very large bottleneck and definitely needs to be addressed.

Compiler Flags

Another optimization approach we attempted was fiddling with the compiler flags. Building off of the copy optimization with transpose version of our code, we tried turning on the following flags: -march=corei7-avx -ftree-vectorize. The -ftree-vectorize flag is suggested in the serial tuning notes, and

-march=corei7-avx seems to be the appropriate setting for our Xeon E5-2620 according to the venerable experts on StackOverflow[2].



Unfortunately, this optimization attempt (blue line, “mine”) was unsuccessful--there seems to be no noticeable improvement over the copy optimization with transpose code upon which this was based. We surmise that the code isn’t complex enough to do any further meaningful compiler optimizations. The optimization flags would be worth looking into again after our code gets more complex.

Hierarchical Blocking

Finally, we hierarchically expanded blocked matrix multiplication for each cache size. Initially, the idea of creating blocked code for the matrix multiplication code was to take advantage of the fact we understand that if we minimize the number of evictions from the cache, we will be able to reduce the performance loss of retrieving data from memory. However, we only have one size, which theoretically should be the size of the L3 cache. We extend this idea of hierarchically decomposing the algorithm at multiple levels such that a subset of the data will fit in the L2 cache, L1 cache, and the register file.

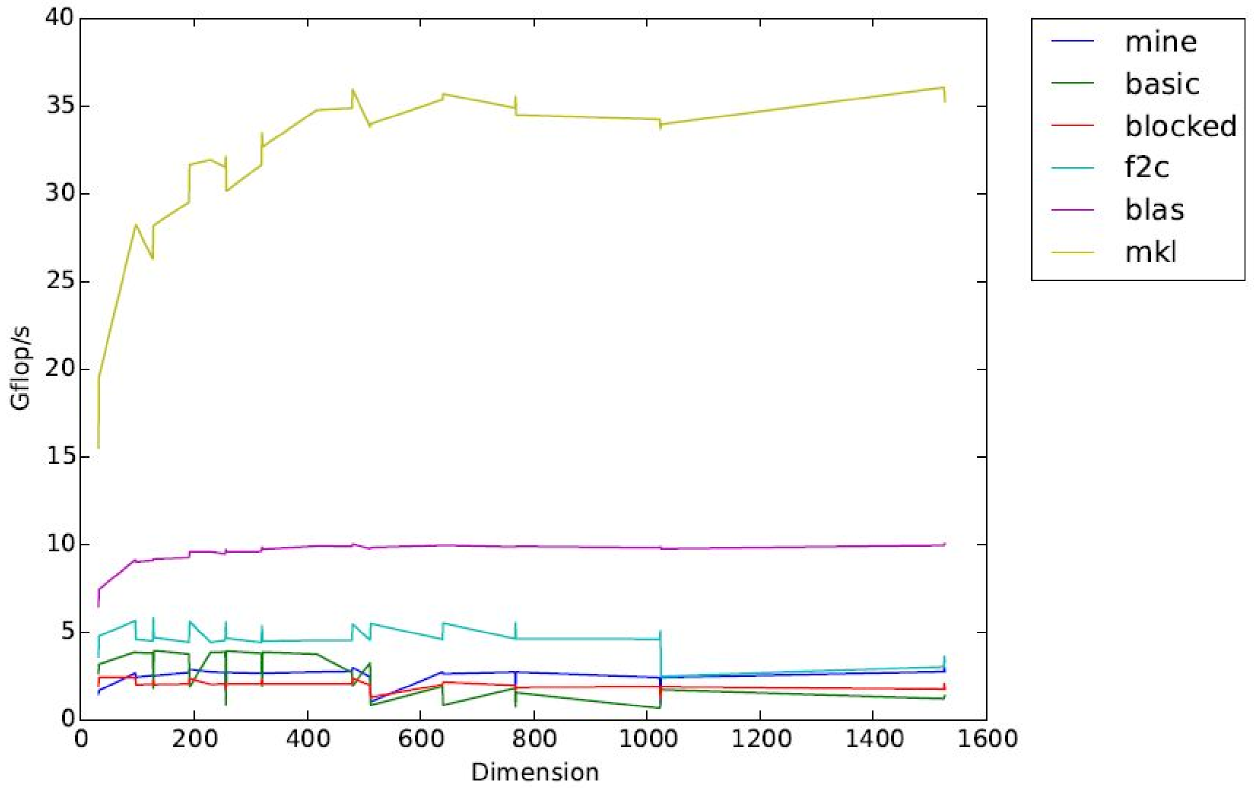
The implementation of a hierarchical block structure is not too hard to do by simply passing the previous memory level’s ijk indices and the size of the cache to the next level. Next we must determine the block sizes at each cache. We look up the microarchitecture’s cache sizes for our cluster[3] and determine that the cache sizes for each core are 2.5MB for the L3 cache (shared for a total of 15MB), 256KB for the L2 cache, and 32 KB for the L1 cache. We approximate that the register is probably small at maybe 1.5KB.

[2] http://stackoverflow.com/questions/943755/gcc-optimization-flags-for-xeon
[3] http://www.cpu-world.com/CPUs/Xeon/Intel-Xeon%20E5-2620%20v3.html

Next, we determine the largest blocks of the matrix multiply can be fit at each level of memory. We assume a write-allocate microarchitecture, which seems reasonable in most memory designs, so we must fit both input block matrices as well as the output (written) matrix which is 3 blocks of data. Furthermore, since we are interested 1 dimensional length of the square block, we take the square root of the size. We also scale down the block dimensions to be safe. So the maximum dimension to fit in each level would be:

dimension = sqrt (<size of cache> / (<8B per double> * 3))

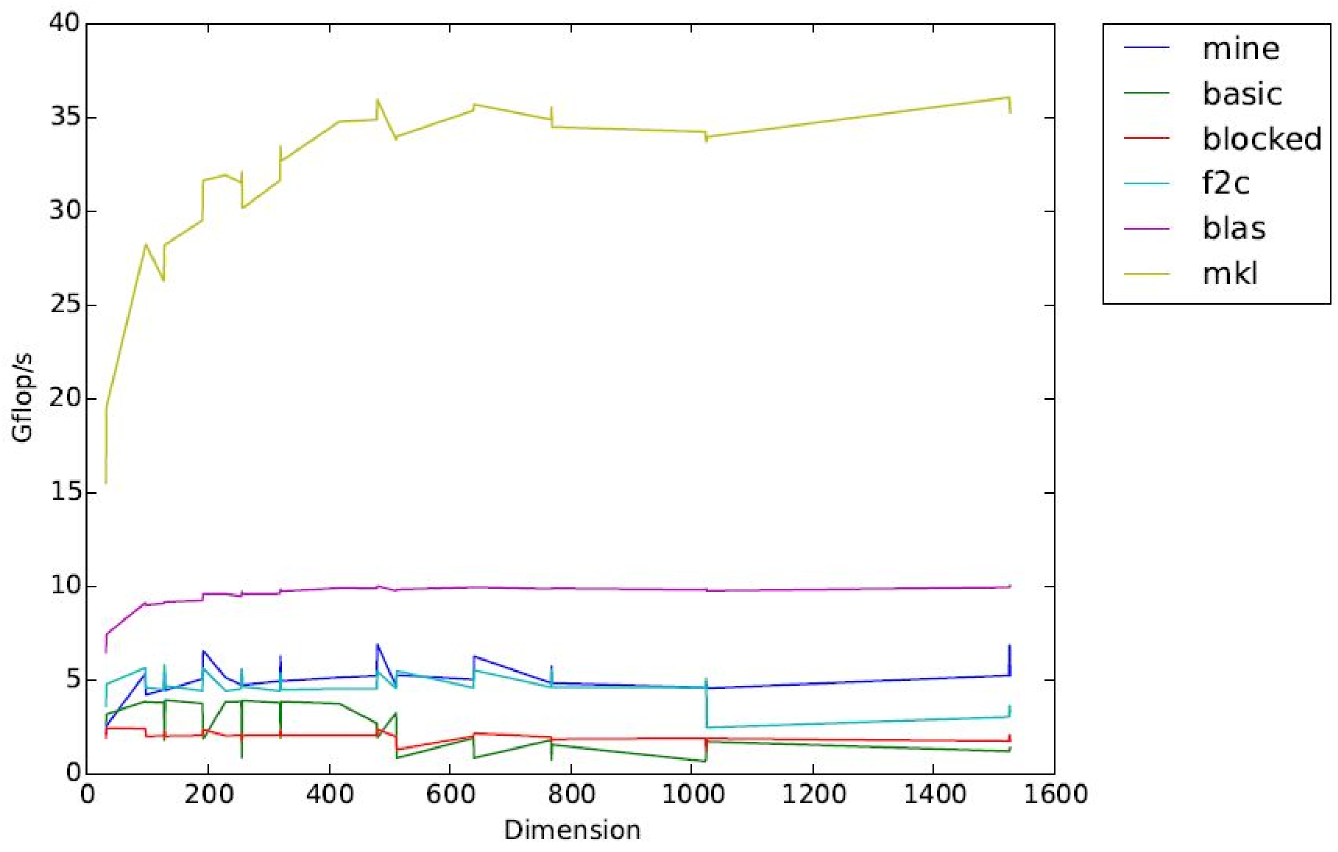
Memory Level	**Size**	**Maximum Block dimension**	**Used block dim**
L3	15 MB	310	288
L2	256 KB	100	96
L1	32 KB	36	32
reg_file	1.5KB	8	8



From the results above, where the result of this treatment is shown as the blue “mine” line, we can see that there is some promise with some improvement over the basic and baseline-blocked code. Hopefully in conjunction with the previous techniques, we can see some multiplicative improvement.

Combining Techniques

After these optimizations were performed separately, we began merging our ideas together. First, we merged the hierarchical blocking with the transpose code. Additionally, we added the *restrict* keyword to the pointers of the matrices in the multiplication A, B, and C(as well as adding the -restrict option in the ICC Makefile). This informs the compiler that accesses using the pointer will not have any aliasing, providing some room for optimizations. With these combined optimizations, we get the results below.



The optimizations together do very well (with the addition of the restrict keyword too). The peak flop rate is now at 6800 GFlops/second[4].

We also try to add the copy optimization, but it seems to get worse performance (with a peak of 4k). We assume that the reason for the decrease in performance is due to the other manipulations of how the data operations are performed. With the hierarchical caching and transposition of the matrices, we already have the data we want in the cache, so we don't need to also use copy optimization.

[4] Unfortunately, we lost the timing-mine.csv file for this graph, and when we ran the the code a week later, we were only able to hit a peak flop rate of 5600 GFlop/s. We suspect that this is due to non-exclusive node access as the deadline drew near (the suggestion to use "`qsub -l nodes=1:ppn=24 job-foo.pbs`" for exclusive node did not end up starting any jobs successfully for some reason).

Additional Optimizations

We also took the time to look through some of our peers' reports to see what kinds of optimizations they tried and find some more ideas. Inspired by Group 25 (Eric Lee, Ben Schulman, Wensi Wu), we decided to try incorporating a memory alignment optimization in our code. By aligning memory for the matrix A that we copied and transposed (since we can't modify the input matrices), we hoped to improve performance by increasing the efficiency of memory accesses. Unfortunately, unlike Group 25, we saw no noticeable increase in performance upon implementing the data alignment constructs suggested by Intel[5]. This may be due to the fact that accessing the columns of the transposed matrix was already fast to begin with, so there was little room for optimization in this area.

We tried a number of new compiler optimization flags that we hadn't tried last time as well:

- -restrict: mentioned above, allows use of the restrict keyword to tell the compiler that there will be no aliasing
- -xCORE-AVX2: allows the compiler to perform vectorization
- -fast: turns on various aggressive optimization options, including -no-prec-div (allows division optimizations) and -ipo (interprocedural optimization)

However, as before, we observed no significant improvement from enabling these flags. As before, we believe that, even after our merges, our code is not yet sufficiently complex enough to take advantage of these compiler optimizations.

Conclusion

In conclusion, by making informed choices of how we perform the matrix multiplication by transposing the matrix, using the restrict keyword for the matrix pointers, and implementing hierarchical caching, we are able to outperform the naive implementation by up to nearly an order of magnitude over the naive implementation.

In addition, we experienced some of the tradeoffs of writing high-performance code. As more and more tweaking is done, the harder the code is to read and reason about. For instance, though we did not try it ourselves, some groups used explicit vector instructions in their matrix multiply, which resulted in code that would be quite intimidating to an uninformed reader.

We are not able to reach the performance of mkl, which also informs us that while it is an enriching learning exercise to try to optimize matrix multiplication by ourselves, for real-world applications, using libraries for these kinds of basic operations is the way to go.

[5] https://software.intel.com/en-us/articles/data-alignment-to-assist-vectorization