# CS 5220: Homework 1

Group 19: Robert Chiodi (rmc298), Zhiyun Ren (zr54), Sania Nagpal (sn579)

September 16, 2015

## Optimizations Used

### Inner Loop Order

The optimization of the double precision general matrix multiply function began with testing loop orders in the naive implementation. This was the first step since even a blocked algorithm still requires the inner loops in which multiplication is performed. In order to have the most cache hits possible inside the inner most loop, the loop order of j, k, i was decided, where the matrix multiplication is written as:

$$\boldsymbol{C}_{i,j} = \boldsymbol{A}_{i,k}\boldsymbol{B}_{k,j} \tag{1}$$

Since the matrices are organized in column major format, moving over to a new column requires a large stride equal to the length of the side of the matrix, $M$. By having the j index be the slowest loop, large strides in memory are minimized, increasing cache hits. The i index is chosen as the fastest index since it will only require unit stride in memory for both $\boldsymbol{C}_{i,j}$ and $\boldsymbol{A}_{i,k}$. This was proven to be the fastest loop order while testing loop orders in the naive implementation.

### Blocking

In order to reduce the working set so that even large matrices will fit in the L2 cache, blocking can be used. In our implementation, two different dimensions control the blocking. The variable `rect_length` controls the number of rows in the $\boldsymbol{A}$ block and $\boldsymbol{C}$ block. The remaining square side length in the $\boldsymbol{B}$ blocks and the number of columns in $\boldsymbol{A}$ and $\boldsymbol{C}$ blocks are controlled by the integer `block_size`. The block dimensions were split in this way in order to allow blocks to have more rows than columns, where unit stride occurs, encouraging better vectorization. The blocking loop was performed in the same order as the inner loop (j, k, i) for the same reasons mentioned in the last section.

As mentioned before, the importance of blocking is so that the working set can fit into the L2 cache (since the L3 cache is shared among cores). For this reason, the number of elements in a block should require less memory than the L2 cache's capacity. From [1], the processors on the compute nodes has 256 KB 8-way associative caches. With $\boldsymbol{C}$, $\boldsymbol{A}$, and $\boldsymbol{B}$

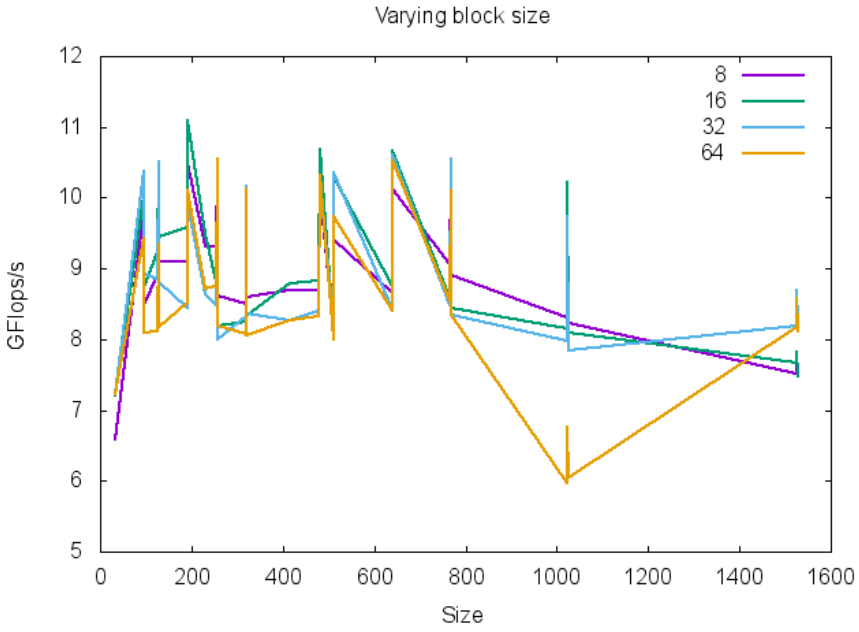


Figure 1: Performance of DGEMM for varying values of `block_size`.

being accessed in the innermost loop, the number of bytes required per block can be found by

$$(\texttt{rect\_length} \times \texttt{block\_size} \times 2 + \texttt{block\_size}^2) \times 8\,\text{Bytes} \tag{2}$$

To fit in the cache, the size of the block, in Bytes, must be less than 256,000 Bytes. Different block sizes were tried. The effect of changing `block_size` and `rect_length` can be seen in Figure 1 and Figure 2, respectively. From these two plots, it was decided to use a value of 384 for `rect_length` and a value of 32 for `block_size`. Using Eq. 2, it is found that these values for `rect_length` and `block_size` utilizes 80% of the L2 cache. There is also some overhead associated with the looping and other constants which must also fit in the cache. The arrays have also been aligned with 16 Byte boundaries, and both 384 and 32 are even multiples of 16. This allows more effective striding in memory, which also contributed to the choices for `rect_length` and `block_size`.

## Compiler Optimizations and Alignments

As just mentioned, the memory allocation was changed so that the `A`, `B`, and `C` arrays were aligned along 16 Byte boundaries. This only had a significant impact on performance at relatively small matrix sizes. Compiler optimizations, on the other hand, had a large impact on performance at all scales, especially at sizes that were even powers of 2. It is believed this is due to better optimization of memory to avoid cache misses caused by the 8-way set associativity. The intel compiler was used with flags of: `-O3, -funroll-loops, -ftree-vectorize -vec -ipo -xHost -no-prec-div -ansi-alias, -axCORE-AVX2 -restrict`. These compiler flags

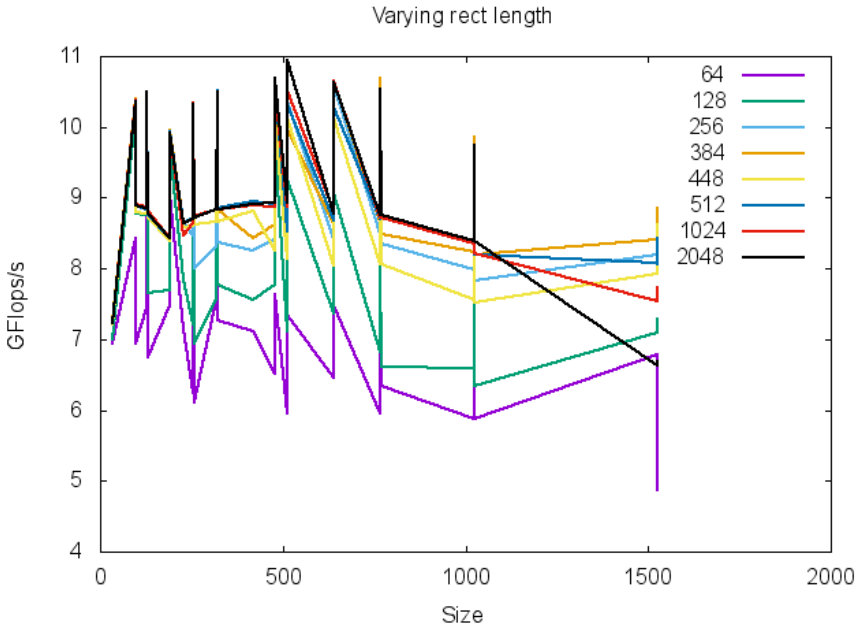


Figure 2: Performance of DGEMM for varying values of `rect_length`.

were found from [2]. Instructions on enforcing memory alignment in arrays were found from [3].

### Optimization Attempts That Did Not Work

One optimization that was attempted and did not work was first copying the inner loops that involved unit strides (only moving down a single column) into a separate array the size of the processors vector operator. The result of $\boldsymbol{A}_{i,k}\boldsymbol{B}_{k,j}$ was then stored in a separate array, and later copied back to $\boldsymbol{C}_{i,j}$. While this did allow the compiler to better optimize the for loops, the added complexity, loops, and memory access far outweighed the benefit.

## Results

Using the above optimization steps, the final performance of our written DGEMM subroutine can be seen in Figure 3 compared to a basic nested loop implementation, naive blocking, BLAS, FORTRAN DGEMM, and Intel's MKL. Our implementation is comparable in performance to the BLAS routine. An interesting observation seen during testing is the degraded performance of Intel's MKL routine when array allocation in `matmul.c` was changed to force 16 Byte alignment.

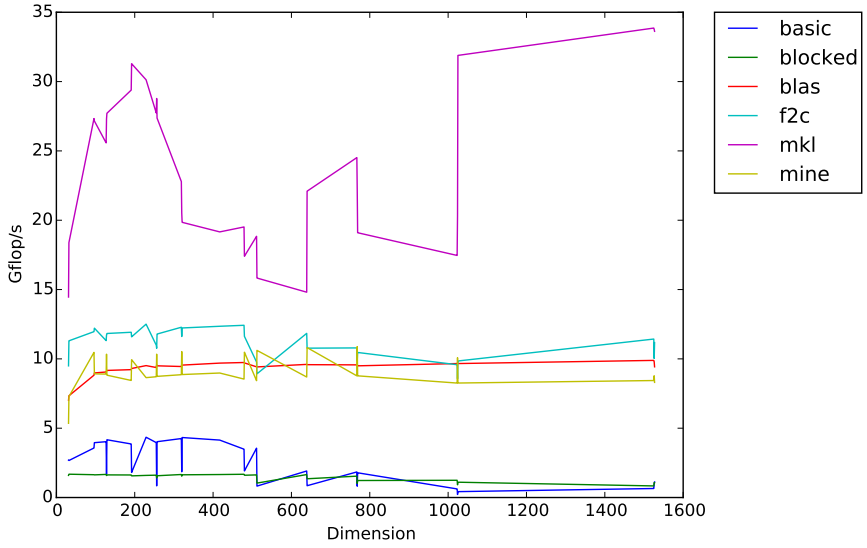


Figure 3: Results of our DGEMM compared to other implementations.

# References

[1] Gennadly Shvets. Intel xeon e5-2620 v3. [Online; accessed 16-Sept-2015] http://www.cpu-world.com/CPUs/Xeon/Intel-Xeon%20E5-2620%20v3.html.

[2] Yang Wang. Step by step performance optimization with intel. [Online; accessed 16-Sept-2015] https://software.intel.com/en-us/articles/step-by-step-optimizing-with-intel-c-compiler.

[3] Rakesh Krishnaiyer. Data alignment to assist vectorization. [Online; accessed 16-Sept-2015] https://software.intel.com/en-us/articles/data-alignment-to-assist-vectorization.