CS 5220: Homework 1

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

September 30, 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 $\boldsymbol{B}$

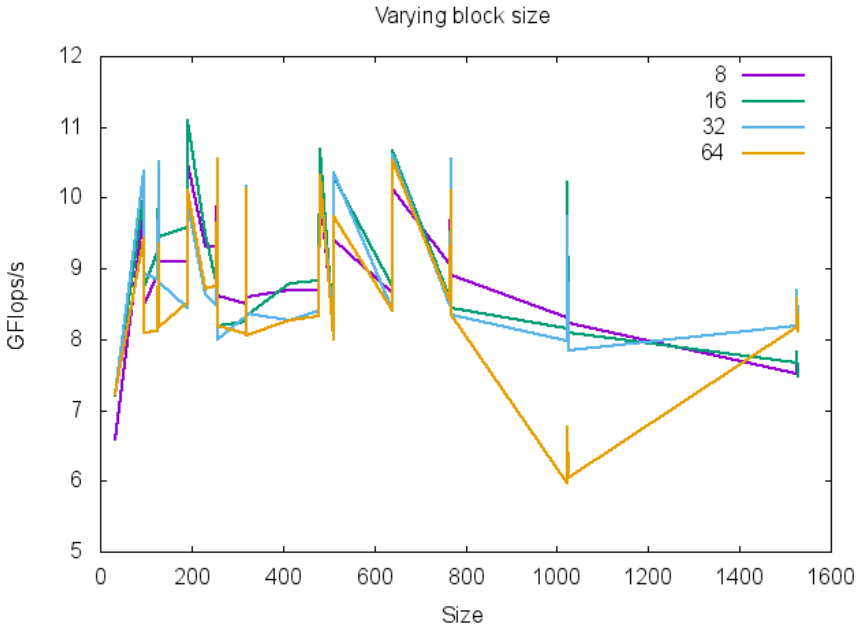

Figure 1: Performance of DGEMM for varying values of `block_size` with constant `rect_length` of 256.

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`

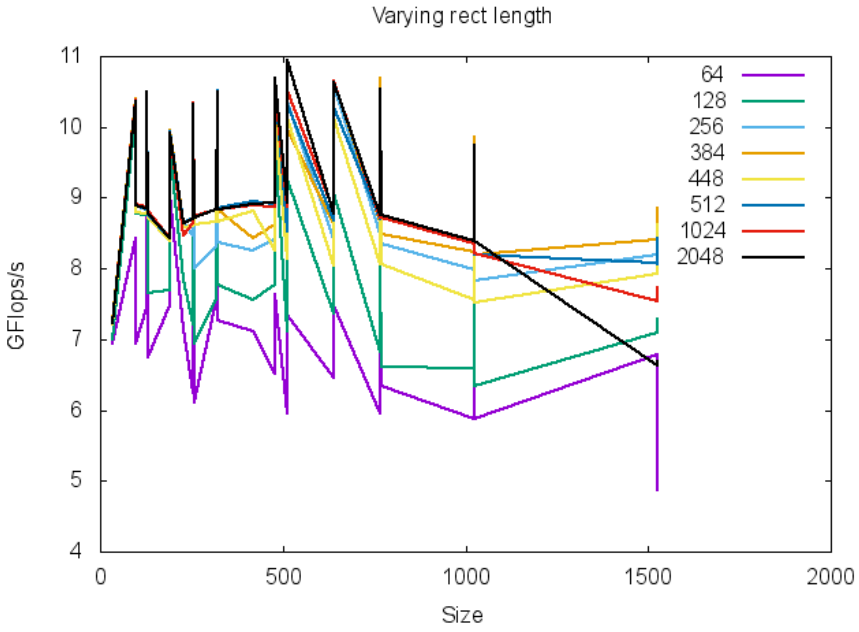

Figure 2: Performance of DGEMM for varying values of `rect_length` with constant `block_size` of 32.

`-no-prec-div -ansi-alias, -axCORE-AVX2 -restrict`. These compiler flags 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.

Preliminary 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.

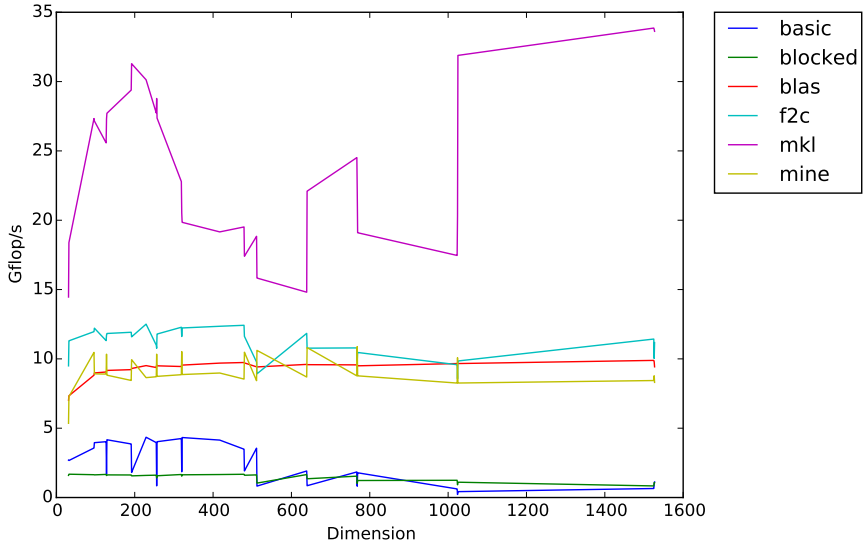

Figure 3: Results of our DGEMM compared to other implementations.

Post Review Modifications

Alignment and Blocking

After receiving reviews from two other groups, several ideas were tested, however the preliminary code remained primarily unchanged. It was pointed out that the allocations of $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$ should not be altered to force alignment in `matmul.c`, so it was attempted to perform copy optimization in `dgemm_mine.c` to use aligned local arrays the size of the blocks. This was implemented and can be found in the `blocked_sys` branch of the git repository. Alignments along 32 byte boundaries were used due to the 256-bit vector operators for the processors on Totient (with 32 bytes being 256-bits).

The reason for alignment is to allow better optimization and use of the vector operators on the processor. Printing out compiler information on loop optimization proved all local aligned variables were identified by the compiler and optimized. However, the operations could not be made effective enough to overcome the cost of the local recopying of the matrices into local, aligned memory. Figure 4 shows the comparison of our blocked code with copy optimization compared to the benchmark matrix multiply routines. The `rect_length` and `block_size` were chosen primarily through experimentation, however it is logical that the values giving the best performance are multiples of the variable alignments. As can be seen, the results are worse than those shown in our preliminary results, Figure 3. Due to the results, it was decided to not use copy optimization and alignments in memory.

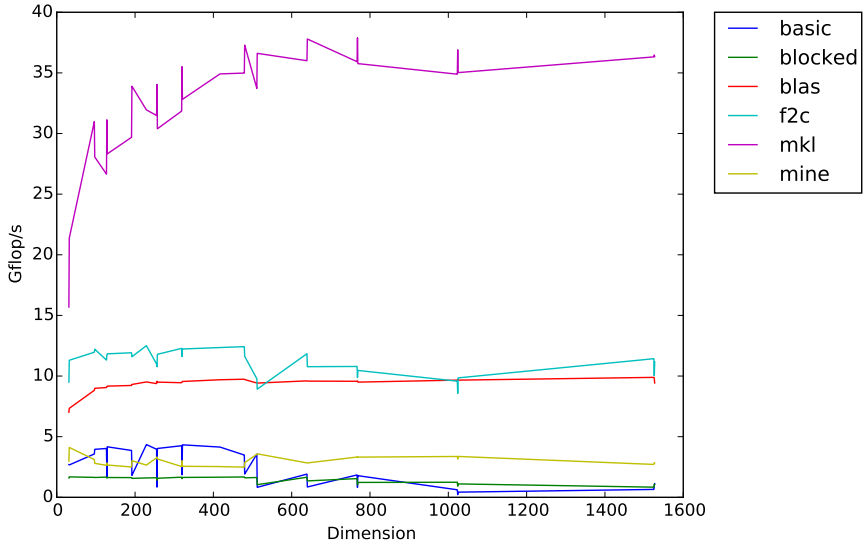

Figure 4: Results of our blocked DGEMM with copy optimization compared to other implementations. `rect_length`=64, `block_size`=32.

Reduction of Arithmetic Operations

The structure of the original code, without copy optimization, was then scrutinized. Multiple operation calculations of array indexes were removed and instead reduced to several simple calculations over several loop structures. This increased the memory required slightly, however greatly reduced the number of arithmetic operations in the entire matrix multiply function. As an example, the value of $\boldsymbol{B}$ multiplying various elements in $\boldsymbol{A}$ in the inner most loop can be stored as a real (in our case `BRHS`) , eliminating the need to access the array for $\boldsymbol{B}$. This provided enough of a gain in performance to consistently out perform matrix multiply using BLAS.

Change of `rect_length`

After reducing the number of arithmetic operations inside loops concerning elements in the blocks, values of `block_size` and `rect_length` were varied again in the same fashion described previously. This was considered due to the suggestion in the review from Group 18, which pointed out that since this is a serial code, it should be safe, and almost as fast memory access, to fit our blocks into the L3 cache. While we agree with this in theory, in our testing a `rect_length` of 512 provided the best results even though it is larger than the L2 cache of 256KB and much smaller than the L3 cache of 15MB. This brings the final sizes for `rect_length` and `block_size` to 512 and 32, respectively.

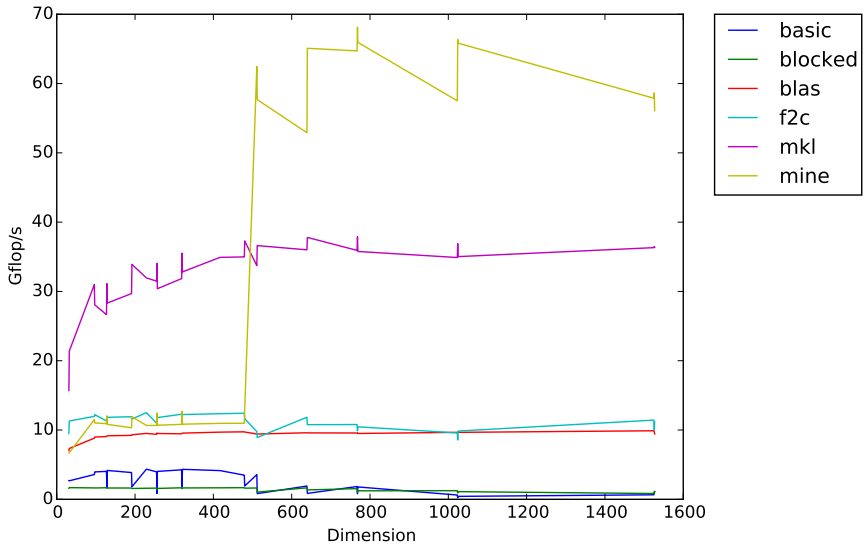

Figure 5: Results for our DGEMM when using the `-parallel` compiler flag.

Use of a faster Totient Node

While running tests on Totient, it was noticed that we would occasionally get significantly better results. After investigating, the better results appeared to consistently happen when the jobs were submitted to Totient nodes other than Totient-02 due to the presence of a job already running on it. This was true even when requesting the entire node through the PBS script. For an additional performance gain, the job submission script for `matmul-mine` was altered to run on Totient-06.

Compiler Parallelization

With the inclusion of one line in our `dgemm_mine.c` (`#pragma parallel`) and the `-parallel` compiler directive in the Makefile, the performance of the code can be very quickly boosted for moderately large matrix sizes. This parallelization is out of the scope of this assignment, however, should be noted due to the minimal work required for massive performance gains. Figure 5 shows our routines performance compared to other benchmarks when using the `-parallel` compiler flag. Large gains in performance can be seen at moderate to high matrix sizes.

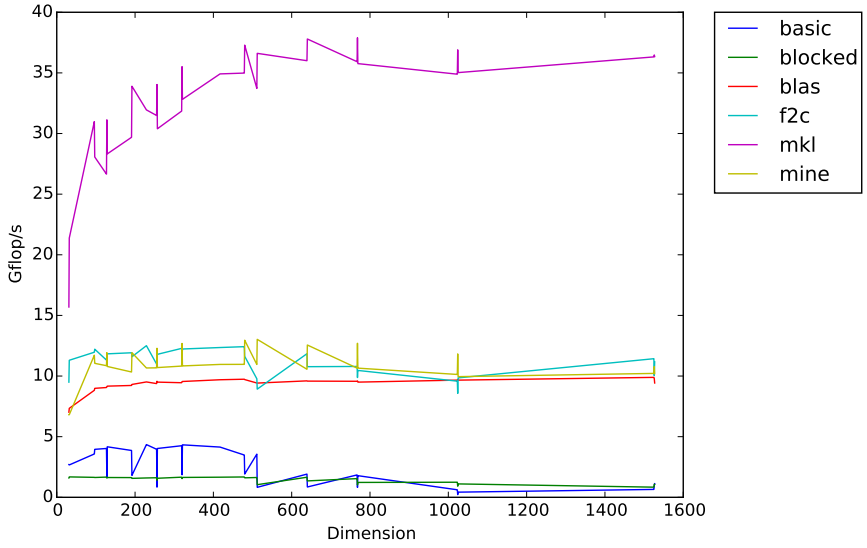

Figure 6: Final results for our DGEMM compared to the other benchmark functions.

Final Results

Our final DGEMM routine without the `-parallel` flag is able to remain consistently above 9 GFlop/s and out perform BLAS, as shown in Figure 6. All routines have been compiled with the same Intel compiler given the same flags, which is the reason for the improved performance of f2c compared to that presented by other groups. Since we removed memory alignment defined in `matmul.c`, the performance of MKL has returned to its previous glory, as expected.

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.