Project 1 - Matrix Multiplication

Team 27 - David Eckman, Bryce Evans and Batu Inal

Motivation

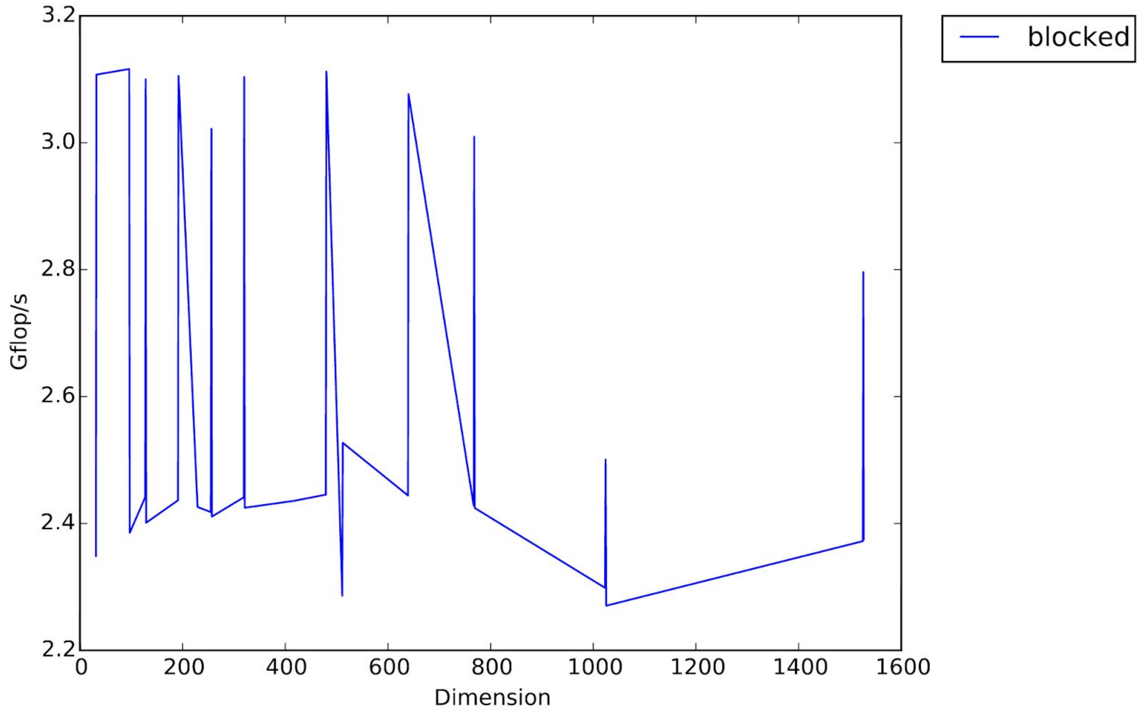
Our assignment was to enhance the performance (GFlops/s) of matrix multiplication (C = AB for M x M square matrices) by improving arithmetic intensity, memory access, and cache efficiency.

Methods

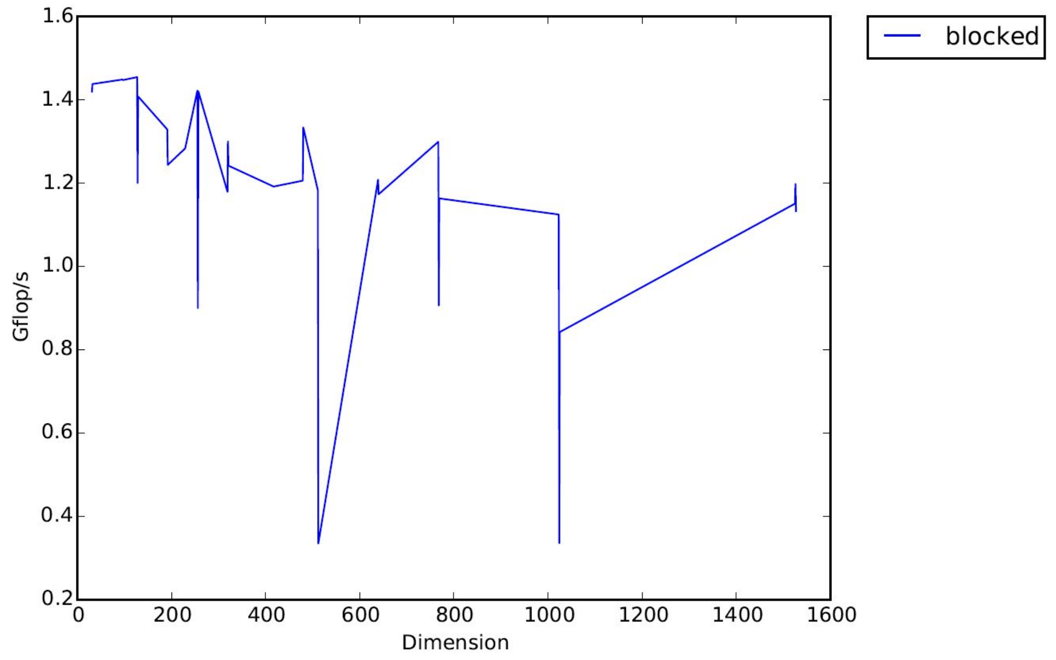
In our preliminary analysis, we considered three main modifications: (1) changing the order of loops for matrix multiplication both within and among blocks; (2) changing the block size; and (3) copying and restructuring matrices in memory, and combinations thereof. We also considered changes to the optimization level used by the compiler.

Method 1 - Changing the Order of Loops

The matrix multiplication algorithm that involves blocking using three nested loops to determine the order of updates both within and among blocks. We tested all orderings of the loops for complete verification. For all combinations, we also tried various optimization flags and block sizes, but found these variables to be independent and have no noticeable significance our selection of the best ordering. Graph 1.1 shows the performance of the ordering which we found to be the most effective at increasing flop rate: k-j-i for both the outer and inner sets of loops. This was an increase of over 100% over the worst case ordering of k-j-i for both loops, as shown in Graph 1.2.



Graph 1.1: Gflops/s vs Dimension for k-j-i Outer, k-j-i Inner (Block Size 16)



Graph 1.2: Gflops/s vs Dimension for k-i-j Outer, k-i-j Inner (Block Size 16)

Given the matrices are in column major order and we are optimizing on reducing cache thrashing, it is important that we keep the columns in memory for as long as possible. This includes using a column entirely while in memory and not needing to reload it later. The update formula `C[j*lda+i] += A[k*lda+i] * B[j*lda+k] (c_ij += a_ik * b_kj)` illustrates how the order of the indices affect the access patterns. By putting i in the innermost loop, we fix the kj element of B and pass through column j of C and column k of A. This access pattern is the most favorable because of the column major alignment in memory. When j is in the innermost loop, we would fix the ik element of A and pass through row i of C and row k of B. This would be the least desirable access pattern.

Note: While we found very little performance difference between k, j, i, and j, k, i, the former was slightly faster and we have not yet determined why. While theoretically index j should be on the outer most loop because it controls column changes the most, our hypothesis is that the cache is not large enough to keep two entire columns and so these columns are occasionally lost from cache and require a reload.

Method 2 - Block Size Relative to Cache Size

For the second method, our approach was to partition the matrices into smaller square matrices (blocks) and multiply them. By using this approach we aimed to improve our arithmetic intensity by performing many multiplication operations with the same data. We also aimed to improve our memory access patterns by choosing the block size so that a block each from matrices A, B, and C could fit into our L1 or L2 cache.

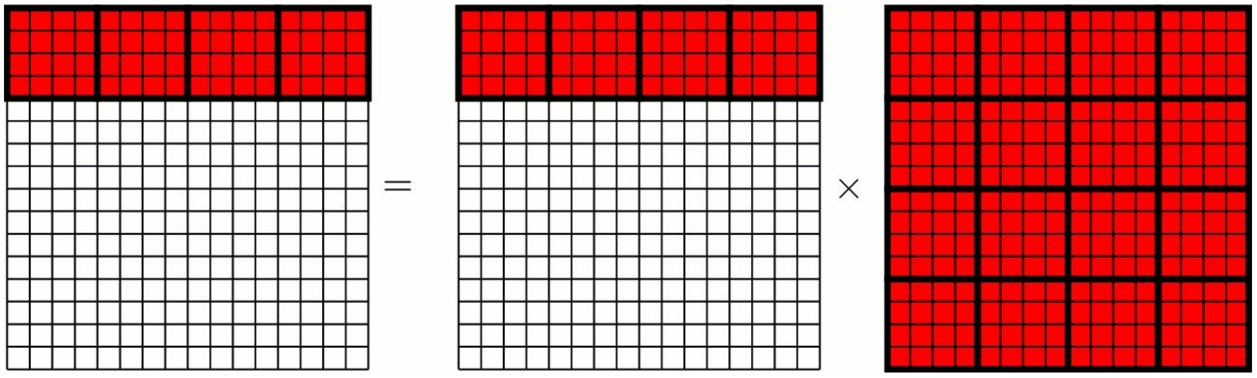
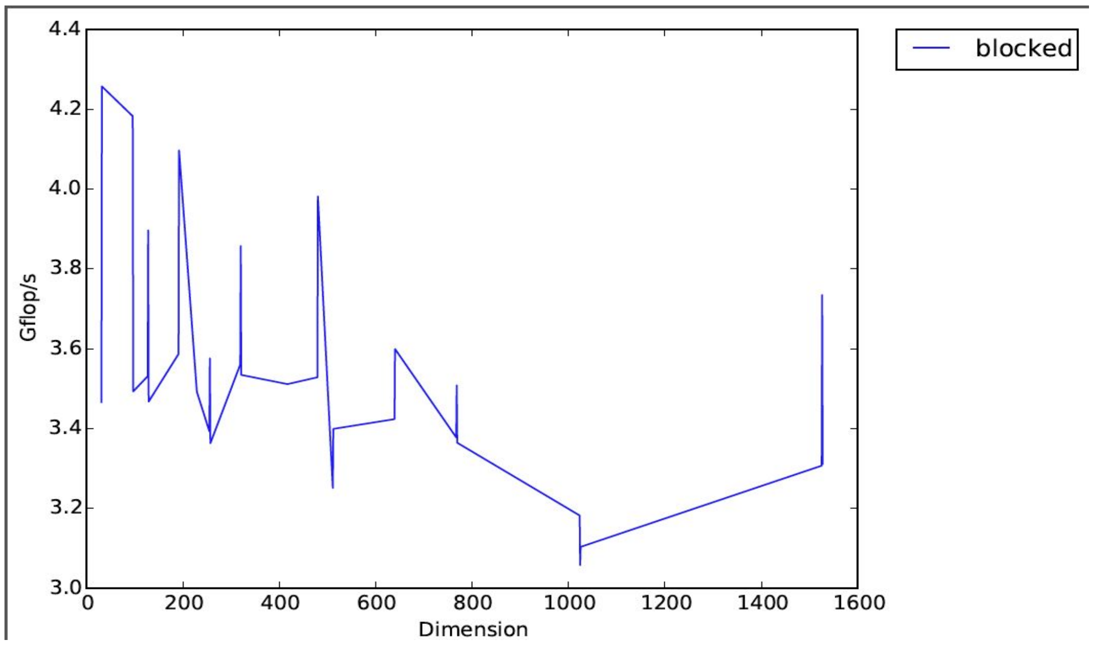


Diagram 2.1: Matrix multiplication instantiated from smaller blocks.

On the compute nodes of the totient cluster, L1 cache was 32KB 8-way set-associative, L2 cache was 256 KB 8-way set-associative and L3 was 15MB (shared) 20-way set-associative. Our initial approach was to utilize the L1 cache without regarding the L2 or the L3 cache. Our calculations for the block size required in order to fit three blocks into L1 cache was as follows:

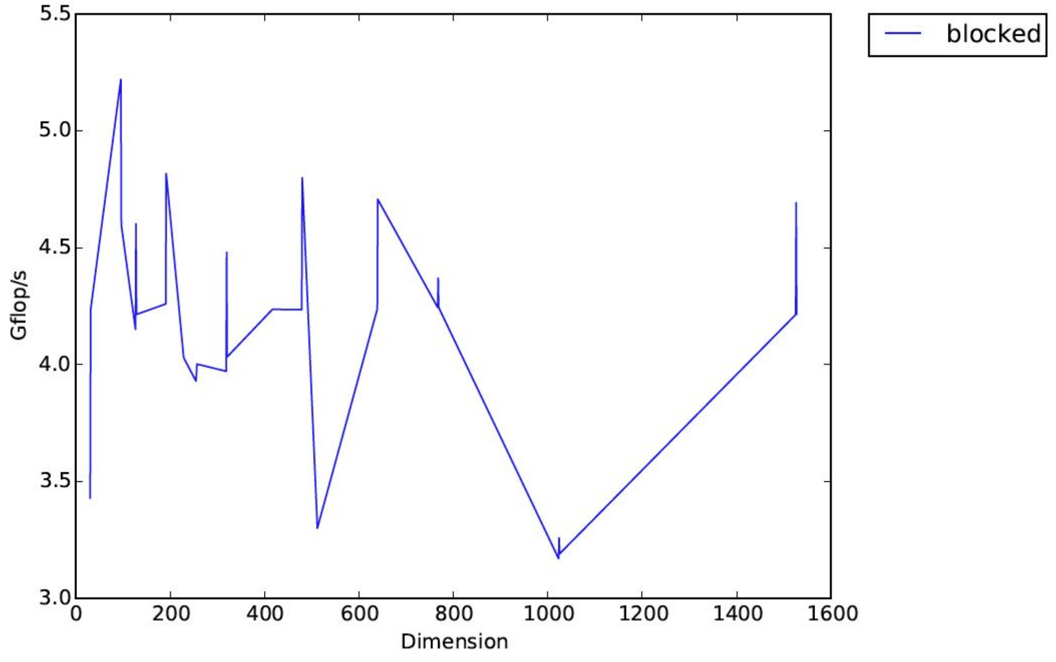
- 32KB * 1024B/KB * 1double/8B = 4096 doubles
- 4096 doubles / 3 matrices ~= 1365 doubles/matrix
- Since there are N^2 values in a matrix, sqrt(1365) ~= 36 (Block Size).

Graph 2.1 shows the performance of the algorithm for a block size of 36.



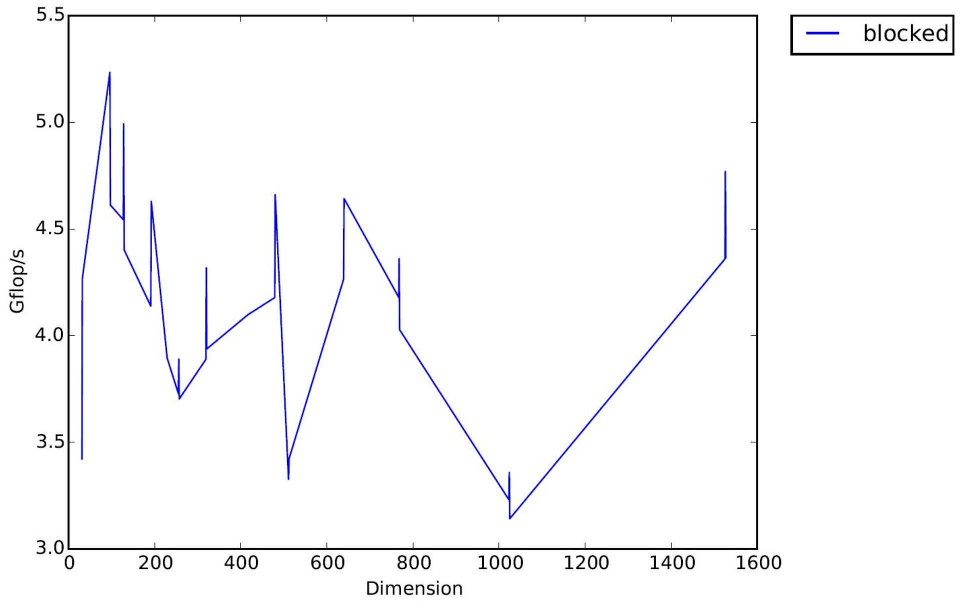
Graph 2.1: Gflops/s vs Dimension for Block Size of 36

We predicted that the overhead of going to the L2 cache if there was a miss in L1 cache would be very low. Therefore we made the exact same calculations for fitting three blocks in the L2 cache. The calculated block size was 104 and the resulting performance is shown in Graph 2.2.



Graph 2.2: Gflops/s vs Dimension for Block Size of 104

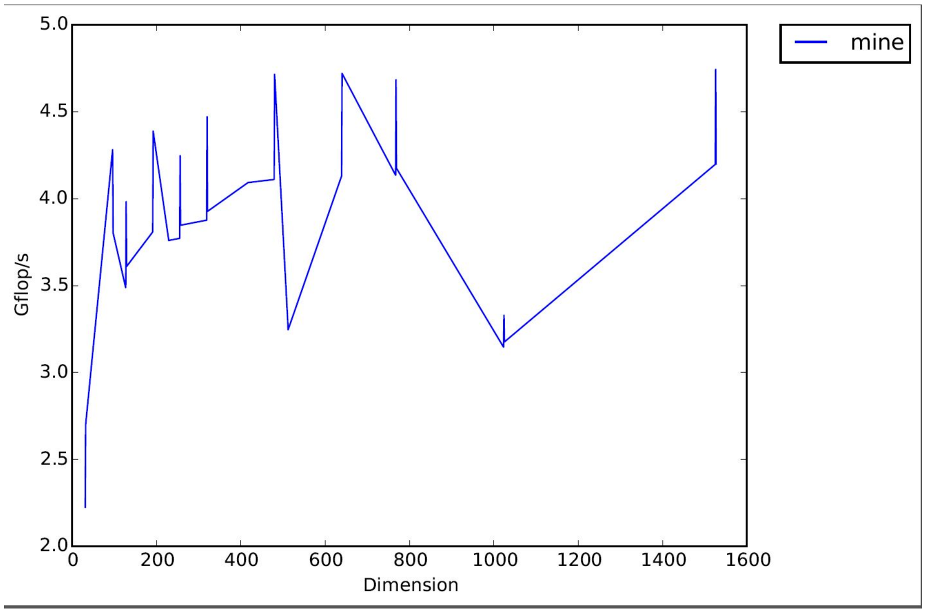
As it can be observed from Graph 2.2 and Graph 2.1, we have significant improvement in performance from a median of roughly 3.7 GFlops/s to around 4.2 GFlops/s. We knew that there would be a huge overhead in going from L2 cache to L3 cache but still wanted to test out a bigger block size; to see whether theory matched practice. For the sake of testing we tested a block size of 128. As it is also depicted from Graph 2.3, there was loss in performance in the dimension range of 200-400 as the size of the block was increased, which could be accounted by an increase in cache misses at L1 and L2.



Graph 2.3: Gflops/s vs Dimension for Block Size of 128

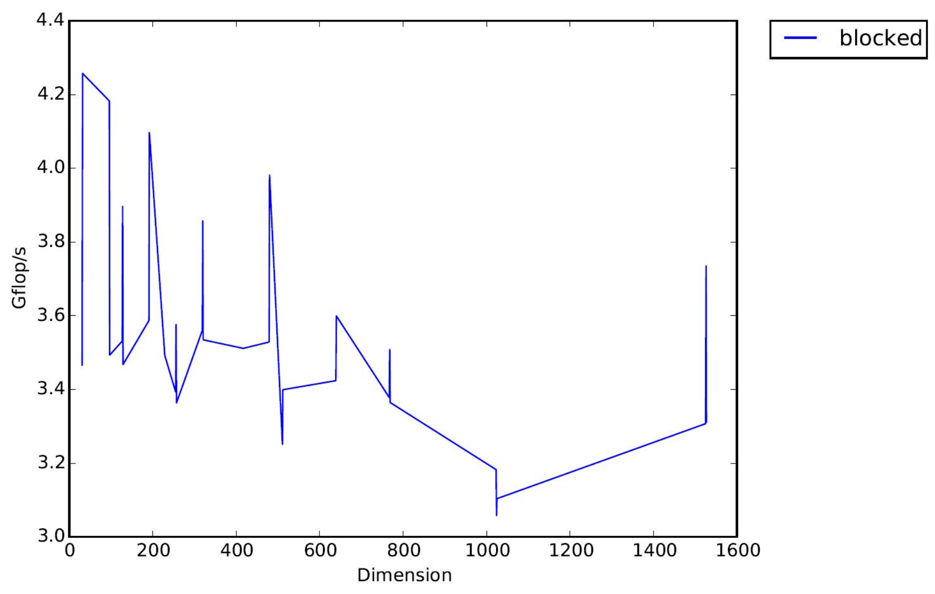
Method 3 - Copy and Restructure the Matrices in Memory

Another method we attempted was to copy the A and B matrices in a way that would improve memory access patterns during the block multiplication operations. More specifically, we restructured the two matrices so that the elements of each block are stored in contiguous memory in column major form and the blocks themselves are stored in column major form. We believed that the improved access patterns in the block-block multiplications would outweigh the computational overhead of copying the matrices. We chose column-major for within and among the blocks of the new matrices because of the ordering of the loops; with the index i in the inner loop, matrices C and A are accessed vertically.

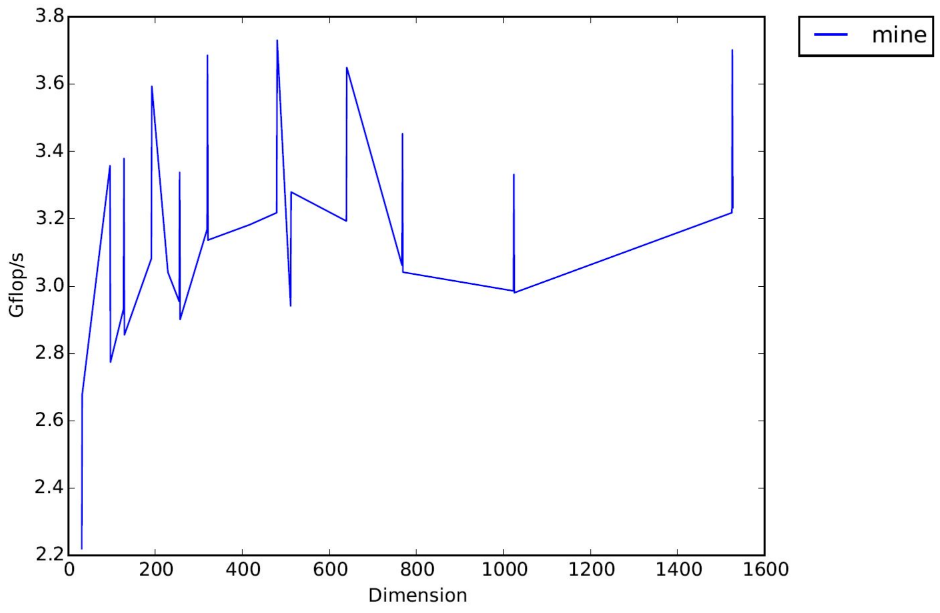


Graph 3.1: Gflops/s vs Dimension for Copying Matrices

Graph 3.1 shows how this new method tested under the loop order k-j-i and block size of 104. Overall, the performance is below that of the method which does not copy matrices at the same settings. The difference between the performances of the two methods is about 0.5 GFlops/s or less. In particular, the matrix-copying method underperformed for low dimensions and performed competitively for high dimensions. This pattern is shown again when we compared the matrix-copying method and the standard blocking method at a block-size setting of 32 (Graphs 3.2 and 3.3, respectively).



Graph 3.2: Gflops/s vs Dimension for Standard Blocking at Size 32



Graph 3.3: Gflops/s vs Dimension for Matrix Copying at Size 32

While the matrix copying method underperforms the standard blocking method, we will continue to experiment with method because the improved memory access patterns have the potential for performance improvement if they can be exploited. It may be that another factor, such as the block size, greatly affects the copying matrices method. We may also want to revisit the ordering of the loops to see if these too can boost the performance of this new method. Lastly, we will experiment with storing matrix C in this manner.

Compiler Flags

We tried an array of compiler flags for optimizing code. Given we were strictly interested in performance (as opposed to code size, compilation time, or memory usage) we focused on optimization levels, namely -O1, -O2, -O3. While O3 was the fastest and gave decent results, we ultimately found -Ofast to be the fastest. While it is documented to not be guaranteed for correctness, in our tests, we found no errors for our code execution.

Summary

From these methods, our current best implementation is to use a k-j-i loop ordering when multiplying blocks and elements within blocks, coupled with a block size of 104. We will continue to experiment with different block-sizes and copying/aligning matrices as well as any ideas discovered during the peer review process.