HW 1: Tuning on a Single Core

Group 13: Taejoon Song, Marc Aurele Gilles, Gregory Granito
{ts693, mtg79, gdg38}@cornell.edu

1. Introduction

In this assignment, we were attempting to optimize the single core matrix multiplication algorithm. We were given a variety of implementations that ranged from very naive to professionally tuned. We started with a blocked implementation of matmul and attempted to make changes to achieve a higher performance.

We attempted four different optimizations in order to improve performance.

1. Optimizing the ordering of the loops
2. Optimizing the size of the blocks
3. Optimizing the compilation settings
4. Using Block indexing to achieve better cache utility

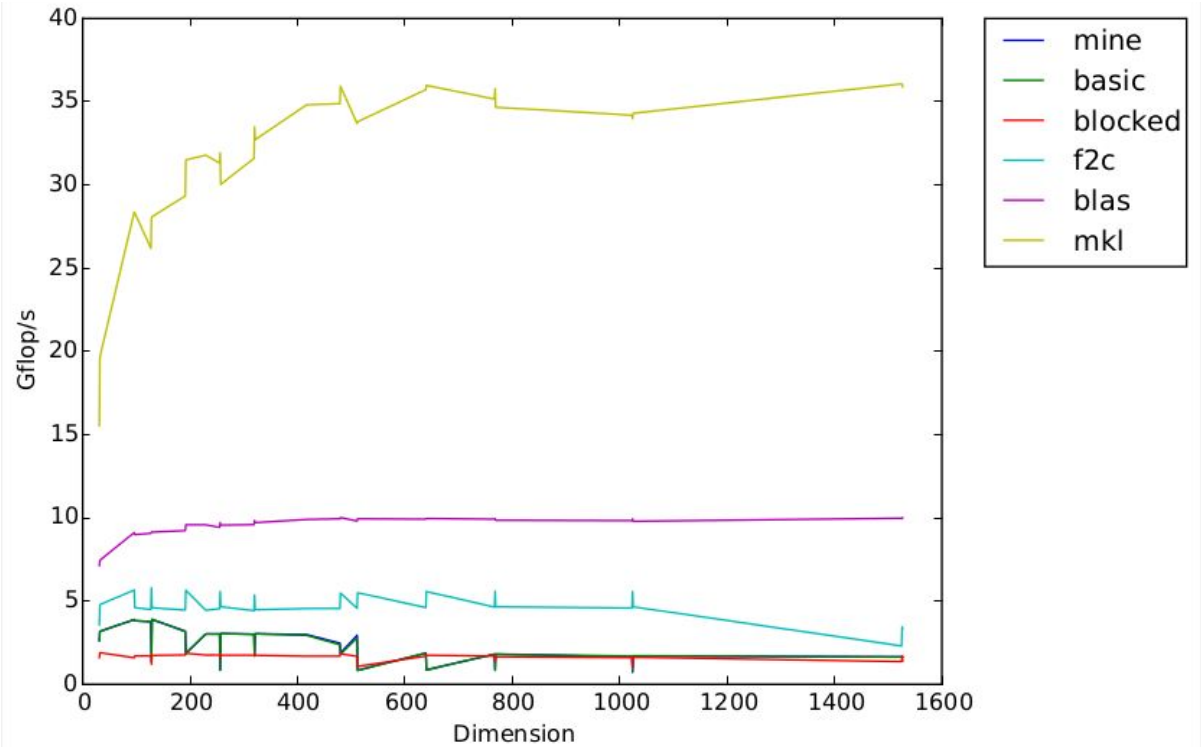
We focussed on these optimizations because the matrix multiplication task seems to be memory bound. So improvements that make better use of the cache are likely to have a larger impact on the overall performance.

By doing these optimizations, we were able to get our performance up to around 5 GFLOP/s. This is about one third the rate that the BLAS based algorithms run at.

2. Optimizations

First we ran the suite of matrix multiplication algorithms as is to establish the baseline performances.

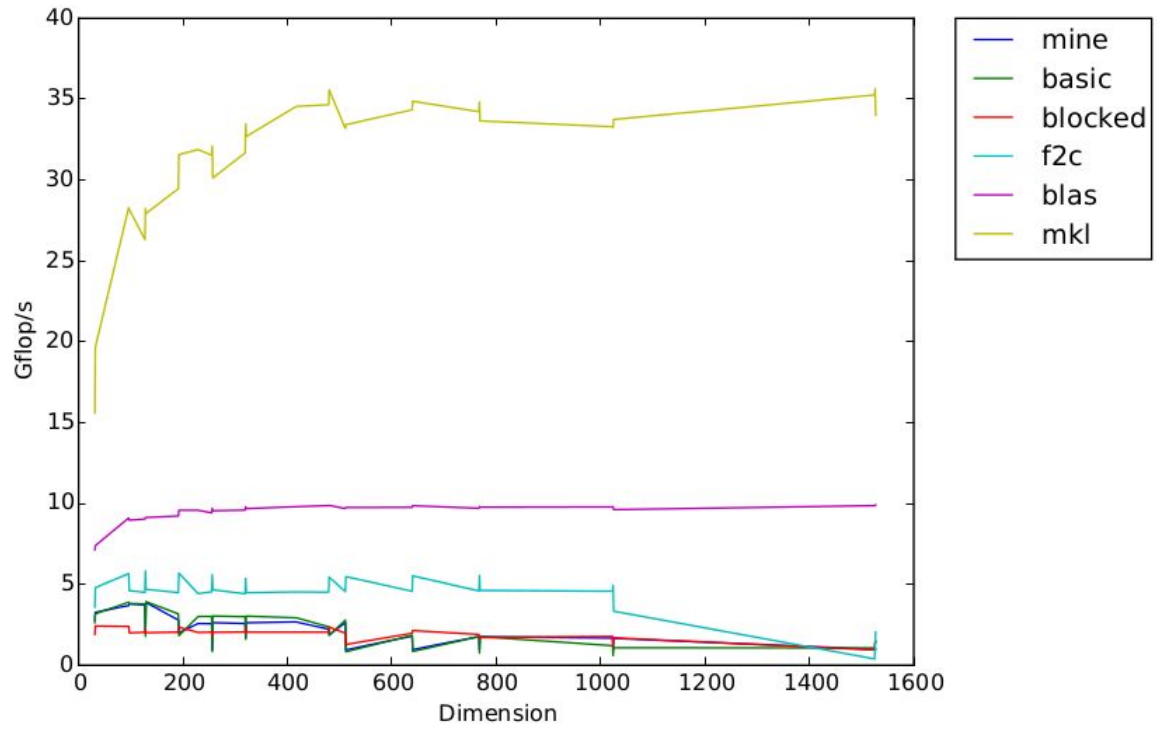
Figure 1. Baseline



2.1. Loop Order

As an initial pass at optimization, we attempted to improve performance of the basic matmul algorithm by varying the ordering of the loops. That is, we swapped the ordering of the loops and observed whether this yielded improved performance. When we swapped the middle and outer loops we found that performance did not change substantially, and if it did the results were worse than the original implementation.

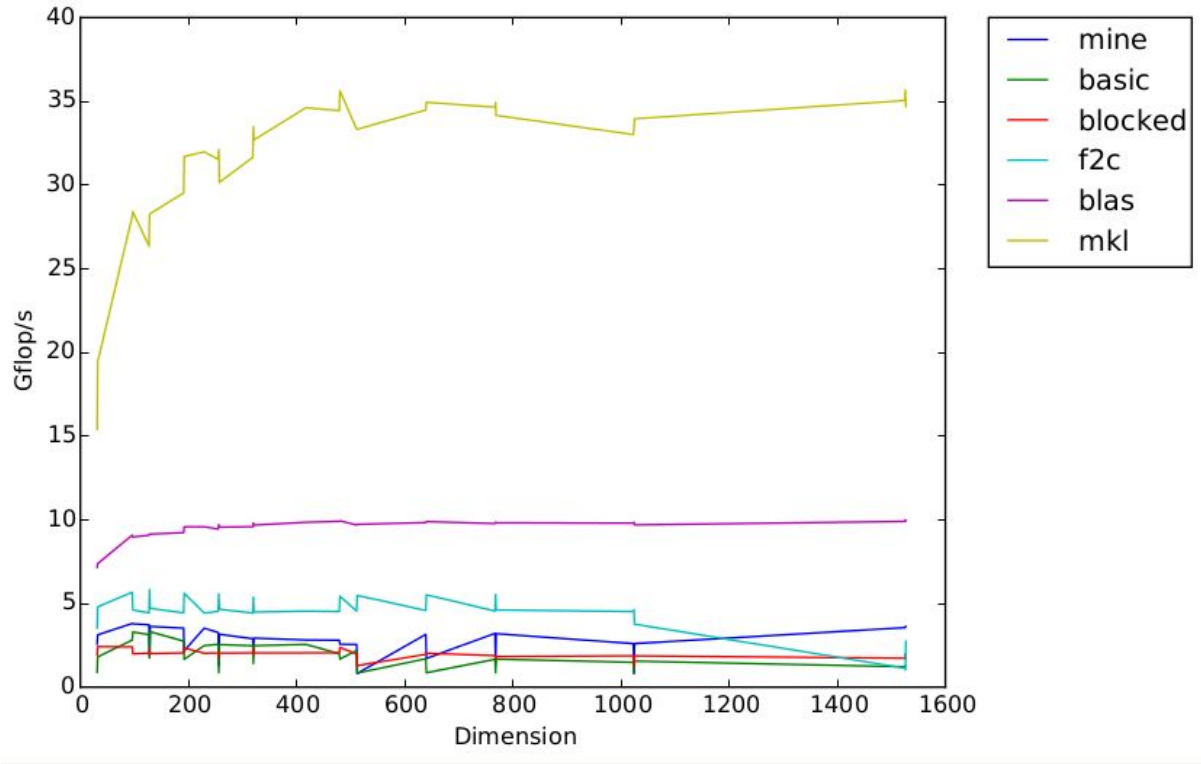
Figure 2. Results of Swapping Middle and Outer Loops



2.2. Blocking

After deciding that the modifying loop order does not drastically impact the results of the algorithm, we moved on to tuning the block size used by the blocking algorithm. Tuning the block size leads to utilizing the L1 cache more efficiently. We manually tried different block sizes.

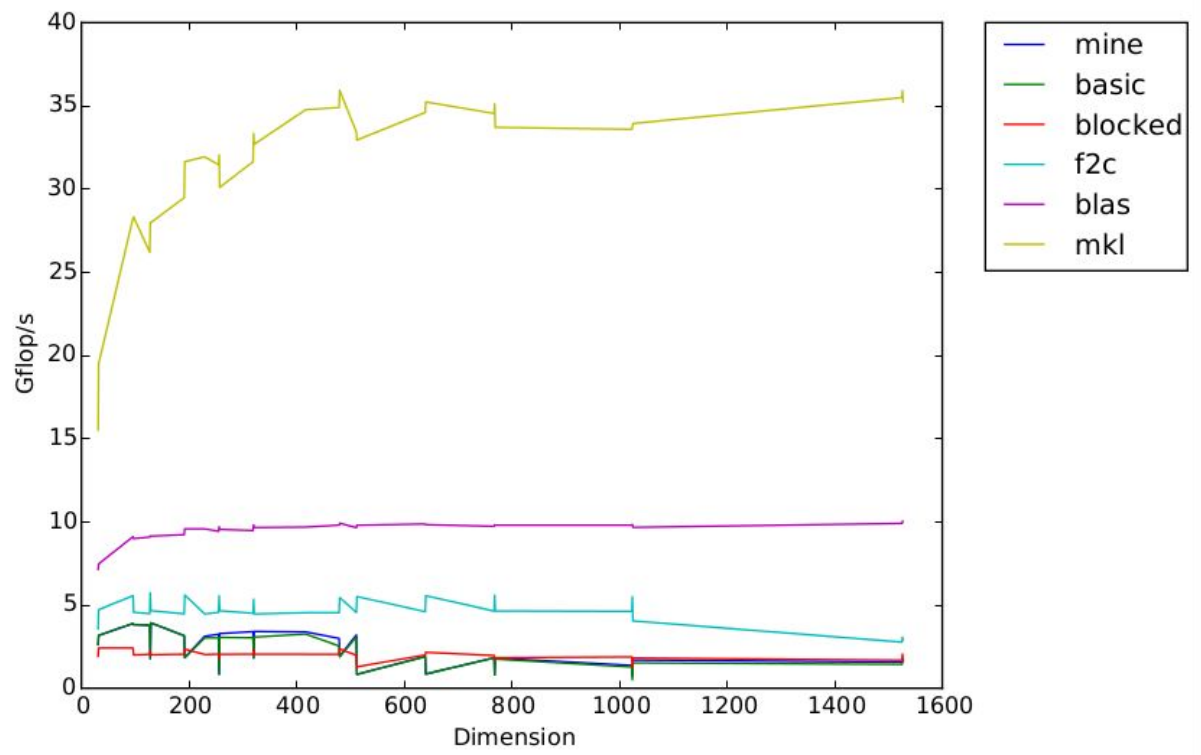
Figure 3. Block Size of 128



2.3. Compiler Optimization

To see if we could achieve better results, we attempted to determine the effect that the compiler settings has on the performance. For a sense of the effect that the optimizations a compiler makes, we started by removing all compiler flags.

Figure 4. Without the -O3 flag



Then we turned on all of the compiler flags to see how much of an improvement we were able to achieve.

Figure 5. With flags: -O3 -ftree-vectorize -funroll-loops -ffast-math

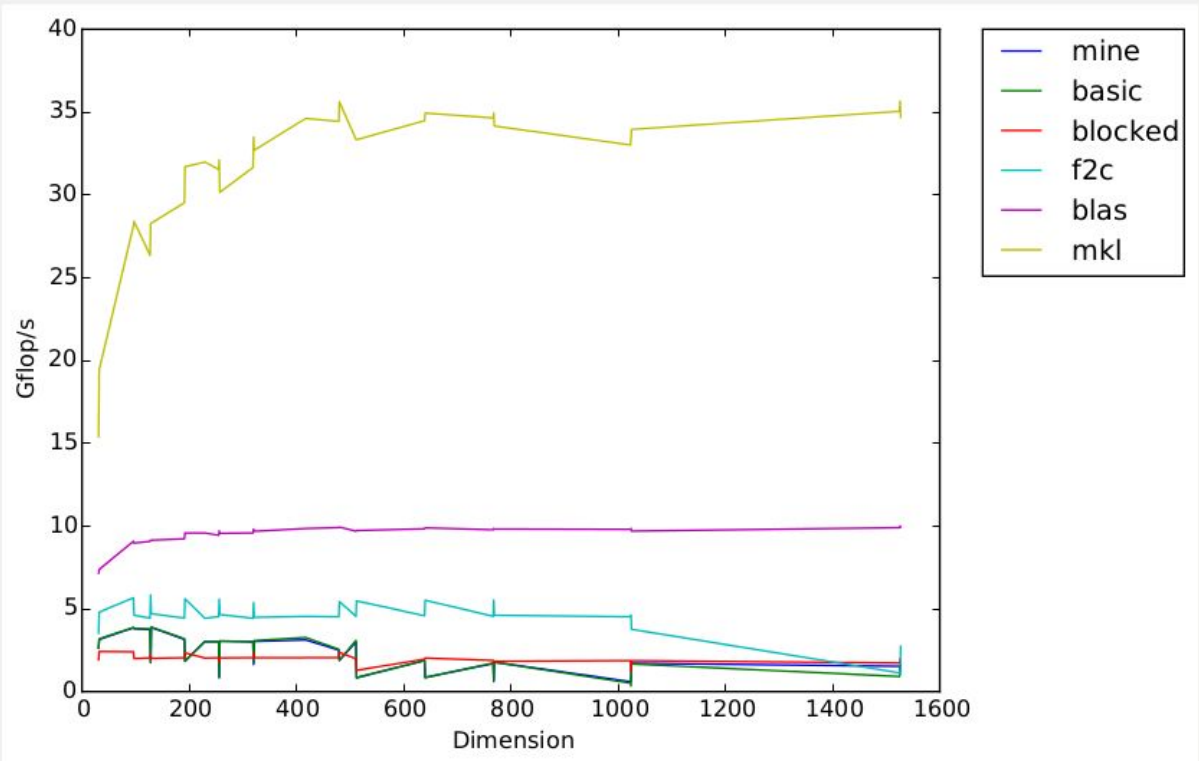
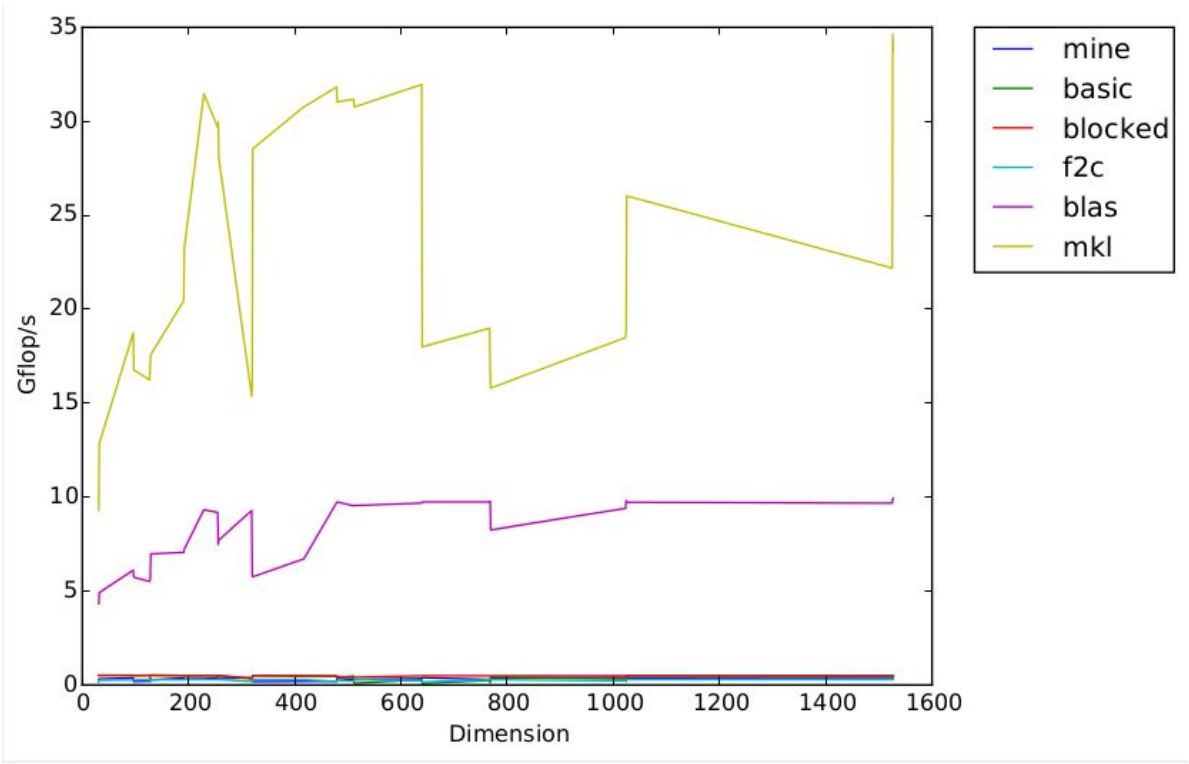


Figure 6. With flags: -O0 (Turn off O flag)



When we turn off the optimization option to -O0, we can see there are huge degradation on performance.

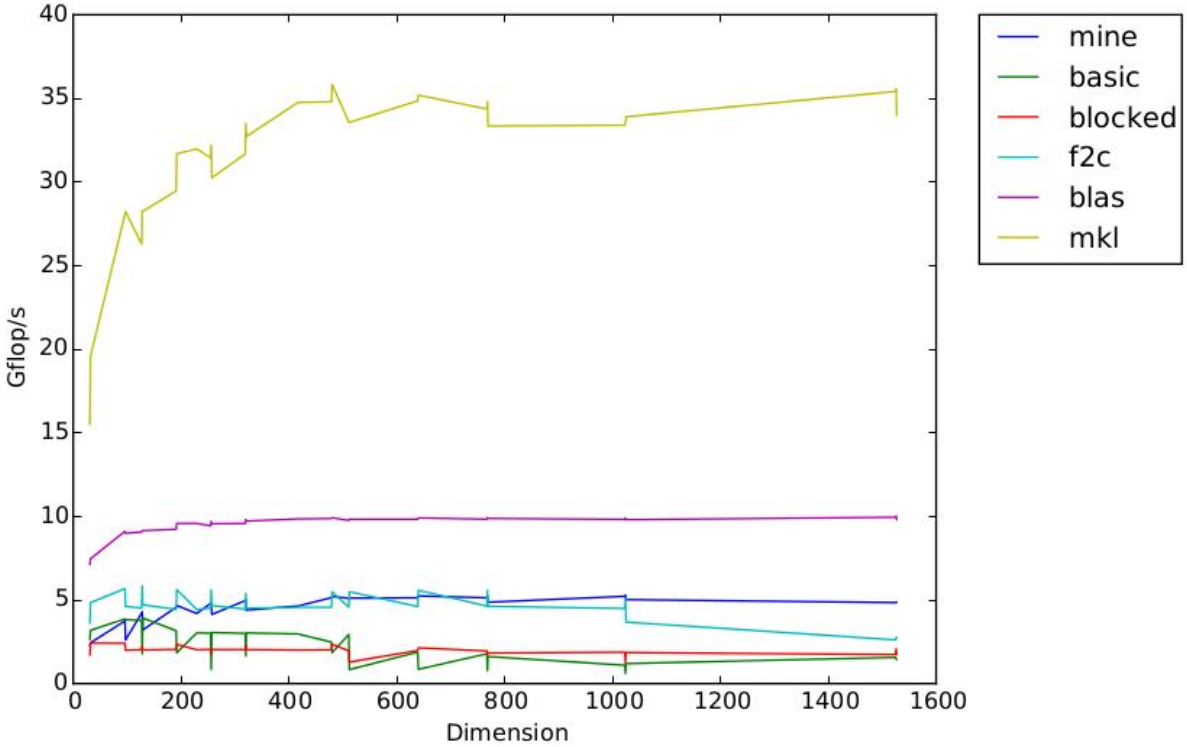
Overall, changing the compiler settings did not seem to have a big impact on the performance of our matrix multiplication algorithm, as long as we stick with O3 level of optimization options.

2.4. Block Indexing

In order to achieve better performance using the block based matrix multiplication algorithm, we modified the indexing of the matrices from column indexing to block indexing .The block indexing is an example of copy optimization. Indeed, before we start computing the matrix multiplication, we allocate a new space in memory to store a copy of each of the matrix with a block indexing.

The new copy of the matrix is padded in a way that makes the size of the copy of the matrix divisible by the chosen size for the blocks . This is done in order to avoid having to handle special cases in the kernel. The result matrix C is reindexed back to the original indexing after the computation is over. Ideally, with the new block indexing scheme there would be better cache utilization and this would result in a higher overall performance. After implementing the new scheme, we modified the block size until we discovered an optimal size of 16.

Figure 7. Block Based Indexing with Block Size of 16



Overall doing the block indexing scheme and tuning the block size seemed to substantially improve the performance.

3. Evaluation

3.1. What Worked

Overall, the best methods of optimization seemed to be:

1. Tuning the Block Size
2. Using a Block indexing scheme

Simply by tuning the block size, we were able to double the computation speed. This indicates that, on large matrices, the problem is heavily memory access bound. Utilizing the cache optimally is likely to lead to substantially improved performance.

This is also present when examining the results of using a block indexing scheme. This scheme allows for a greater cache hit rate. Since the problem is memory access bound, any changes that result in better cache usage will likely have large effects on performance.

3.2. What Didn't

In general, modifying loop order and compiler settings did not have a substantial impact on the performance. Neither of these experiments affected cache utilization substantially. Since this problem appears to be memory bounded, and these changes didn't noticeably change the cache utilization, it makes sense that they didn't make substantial changes to the performance.

4. Next Steps

There are still many modifications we can make to our current experiments that will likely further improve our matrix multiplication performance.

4.1. Improving Block Indexing

The algorithm that we used to implement the block indexing scheme still has room for optimizations. Currently there is an if statement in the reindexing algorithm. Removing this will likely improve performance substantially.

4.2. Utilizing L2/L3 Cache

Currently, the block size was tuned to achieve performance using the L1 cache. However, since the task is memory access bound, it is likely that better performance can be achieved by making better use of the L2 and L3 caches.