CS 5220 Matrix Multiplication

Stage 1 Report

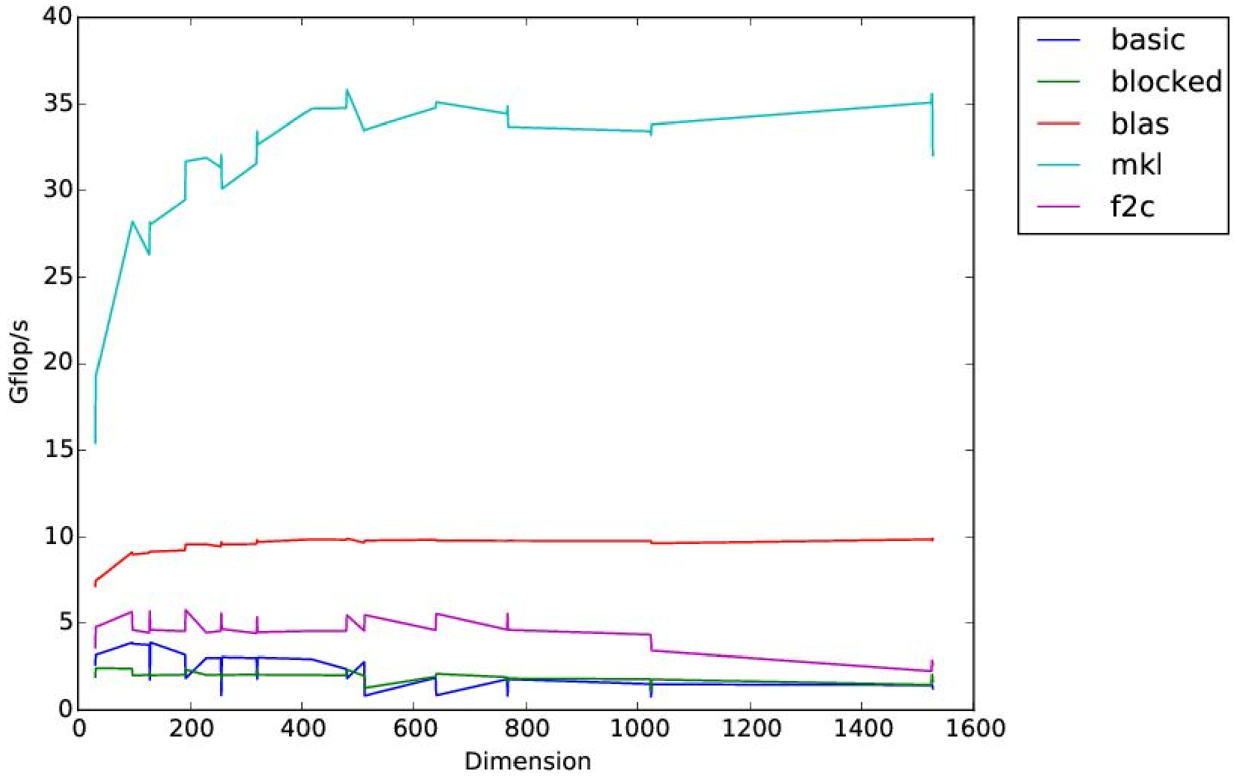
Guo Yu (gy63), Kaiyan Xiao (kx58), Scott Wu (ssw74)

September 17th, 2015

Introduction

In this assignment, we improve upon the performance of matrix multiplication. We focus on serial performance tuning on square matrices, focusing on data alignment in cache to reduce cache misses and increase operational intensity. In the prototyping stage, we look at the techniques listed on the matmul slide which include changing loop nesting, testing different compiler options, block size and copy optimization. We mainly test each of these tactics individually (not mixed with other changes), so that we can analyze and identify what works and what does not.

First we ran matmul on the cluster with default options to get a sense of how the implementations compared and what our goal might be.



The default implementations for matmul

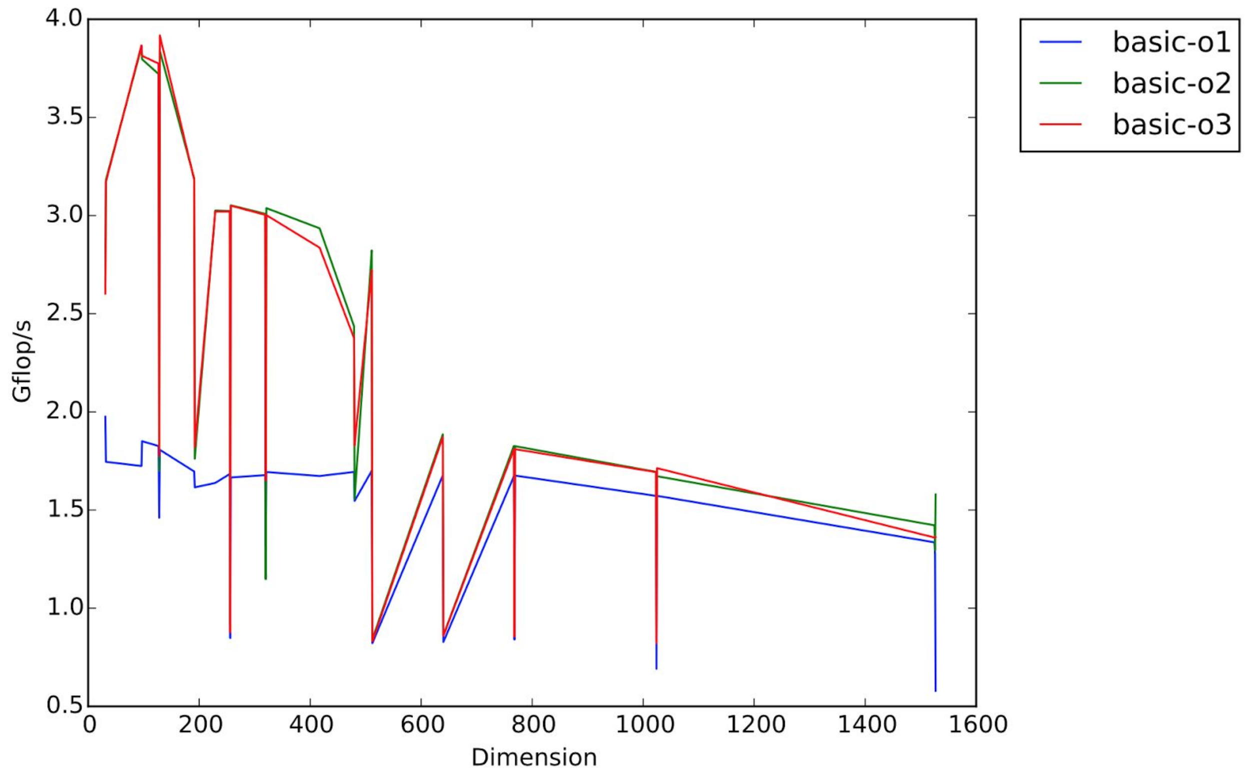
It was quite obvious that the Intel MKL could be considered the "golden standard" in terms of performance. Relatively, OpenBLAS seemed to be something reasonable to compare to.

Finally, we can see that the basic implementation simply performs poorly compared to everything else.

Different Compiler Optimization Options

We also ran mutmal under different compiler optimization options, and the compiler we were using was Intel ICC compiler. We modified the OPTFLAGS option in Makefile.in.icc, and ran the basic method on the compute node under different optimization levels. The following plot shows the effect on performance of different compiler optimization. It is clear that optimization level one(basic-o1) has the worst performance, while level two and level three have similar performances.

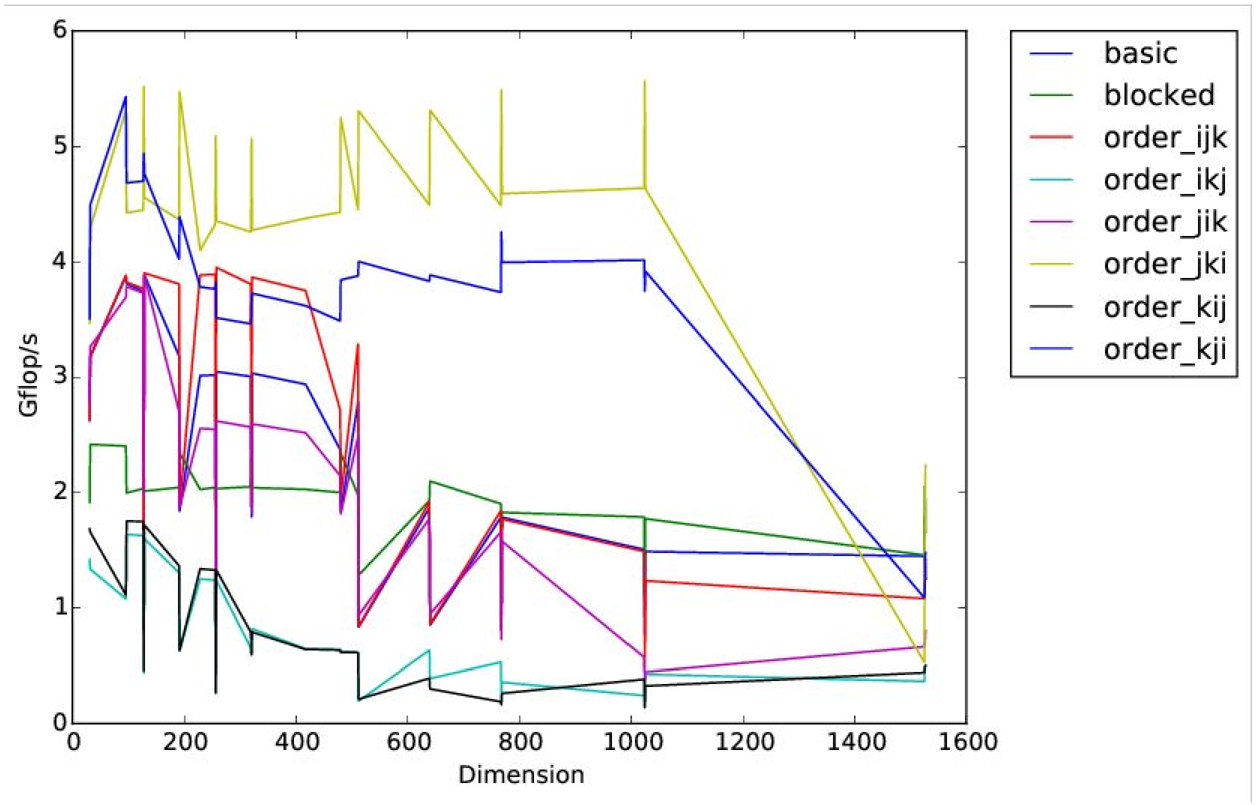
We have also tried to modify some other optimization options, but it seemed most optimization options had already been enabled in level 3 optimization and showed little difference.



Different Compiler Optimization Options

Looping Order

For basic_dgemm, the naive implementation loops through i first, j second and k last. There is good reason to believe that the order i,j,k is not good since going through a row is "non-unit-stride" access. The following plot shows the result of different looping orders of naive implementation of matrix multiplication. The order j,k,i performs the best.

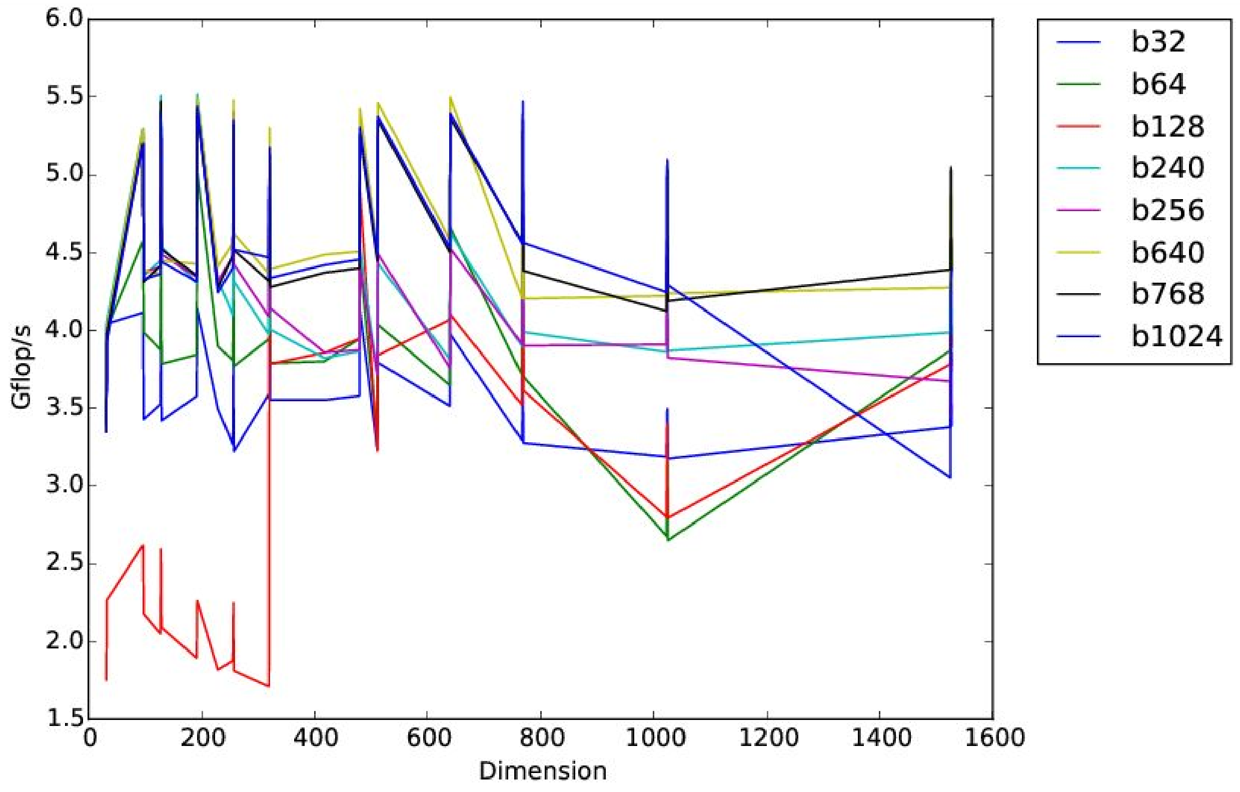


Different loop orders vs basic and blocked implementations

Block Size

According to the "Tuning on a single core" note by Prof. Bindel, we second tried the idea of blocking. By dividing original matrices into small blocks, each of certain block size, we hope that small block of matrix can fit into cache. In specific, since L3 cache is shared, we want to find a block size so that these blocks can fit into L2 cache. And if we do it recursively, as suggested by the note, the smaller blocks of these blocks should fit in L1 cache, and so on.

For now, we only studied partitioning the whole matrix once, and consider the j,k,i inner looping order we found earlier. The following plot shows the results of performances for different blocks. There is a potential that the higher the block size is, the higher Gflops/s we get. We choose block_size = 768.

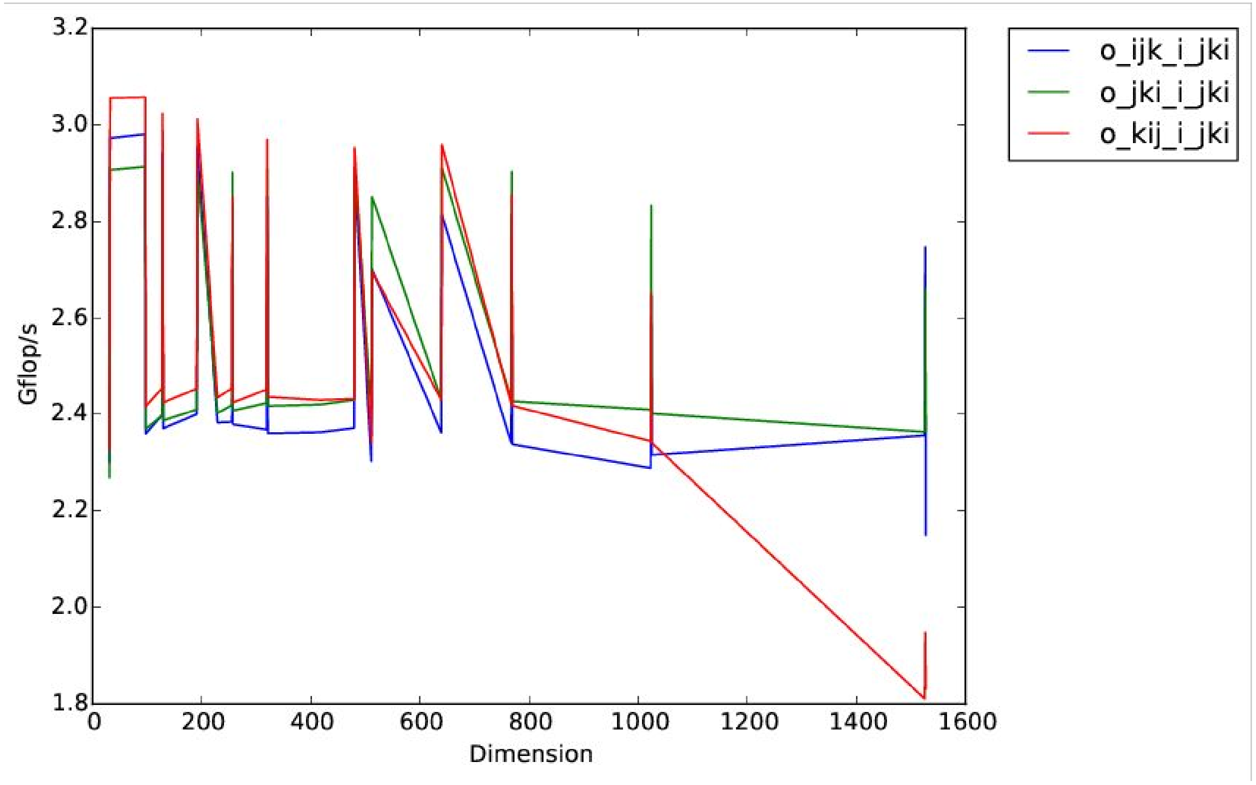


Different block sizes

Blocking Outer Loop Order

We used the same idea as tuning inner looping order to tune the order of looping blocks. To do this we fix the looping order of each block matrix multiplication as j,k,i as we have found earlier. Also we used the block size 16 as in naive implementation. The following plot shows the difference of performance for different (not all possible) outer looping order. The difference is so small that we decided to stick with the same order as inner loop, i.e., j,k,i.

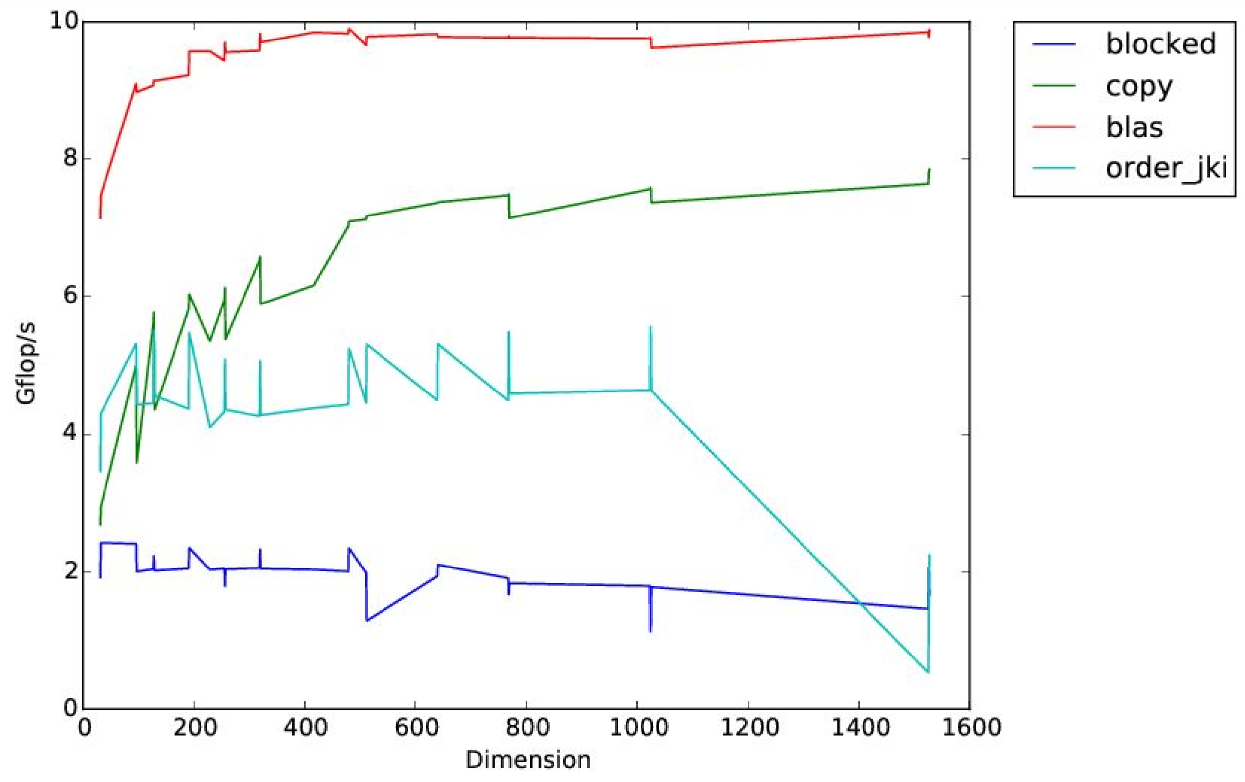
Also note that when block size grows bigger, the effect of outer looping orders will make less impact.



Outer loop order with jki inner loop order

Copy Optimization

Blocking by itself does not perform well enough. We still have to take large strides across columns. In Copy Optimization, we first copy the blocks to contiguous segments in memory. That way, multiplying each block improves the locality of the reads, and the possibility that the entire blocks fit in L2 or L3 cache. Although the initial and final copying is an overhead, we expect it to be minimal as the matrices grow large because copying is an n^2 operation while multiplying is n^3. In this implementation, we use 0-padded blocks of size 16.

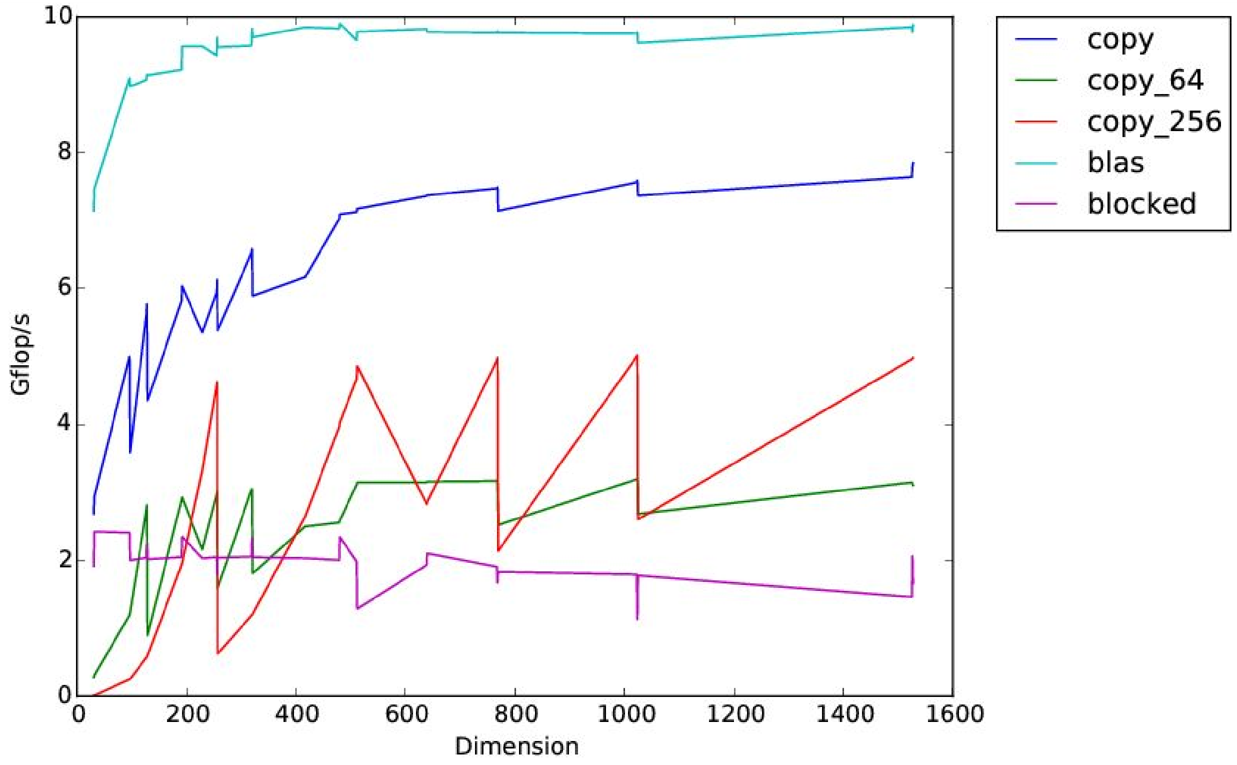


Copy-optimized blocking vs BLAS, loop order and naive blocking

The Copy-Optimized blocking starts slow (possibly because of the copying overhead), but achieves over 7 Gflops/s at higher dimensions. Also the performance pattern looks more smooth and is similar to that of BLAS.

Copy-Optimization and Block Size

Earlier, we saw some increased performance in certain block sizes, so we tested a few larger block sizes with Copy-Optimization as well. Changing block sizes makes more sense for Copy-Optimized blocks because we know that a contiguous block will probably be entirely in cache and we can control that size to fill it.



Copy-optimized blocking at different block sizes

However, it seems like we do not get the expected increase. We are using 0-padded matrices, which explains the extremely poor performance at small dimensions, but even at dimensions 256, 512 etc. the other block sizes did not perform better than a block size of 16.

Future Work

We now have a sense of the techniques that worked and those that didn't. It is clear Copy-Optimization improves performance a lot. It is also clear that using jki as the inner looping order outperforms other inner looping orders. For compiler options, we will stick with icc and O3.

For the final part of the assignment, we will first combine all these analyses and do some additional minor optimizations (such as caching a variable in the inner loop). Furthermore, we can use icc's vectorize report option to get feedback on which loops are vectorized and which are not. We can also try using the __aligned__ directive discussed in lecture. Finally, we will be looking at other math libraries for insight on optimization and perhaps SIMD instructions.