CS 5220 Matrix Multiplication

Final Report

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

October 1st, 2015

1 Introduction

In this project, we improve upon the performance of matrix multiplication. We focus on serial performance tuning on square matrices, looking at data alignment in cache to improve locality and reduce cache misses, and increase operational intensity. In the prototyping stage, we looked at the techniques listed on the matmul slide which include changing loop nesting, testing different compiler options, block size and copy optimization. In the final stage, we looked into combining techniques we found from other groups and those we planned to experiment with. In summary, we achieved 20 Gflops/s using blocked copy optimization with aggressive compiler options.

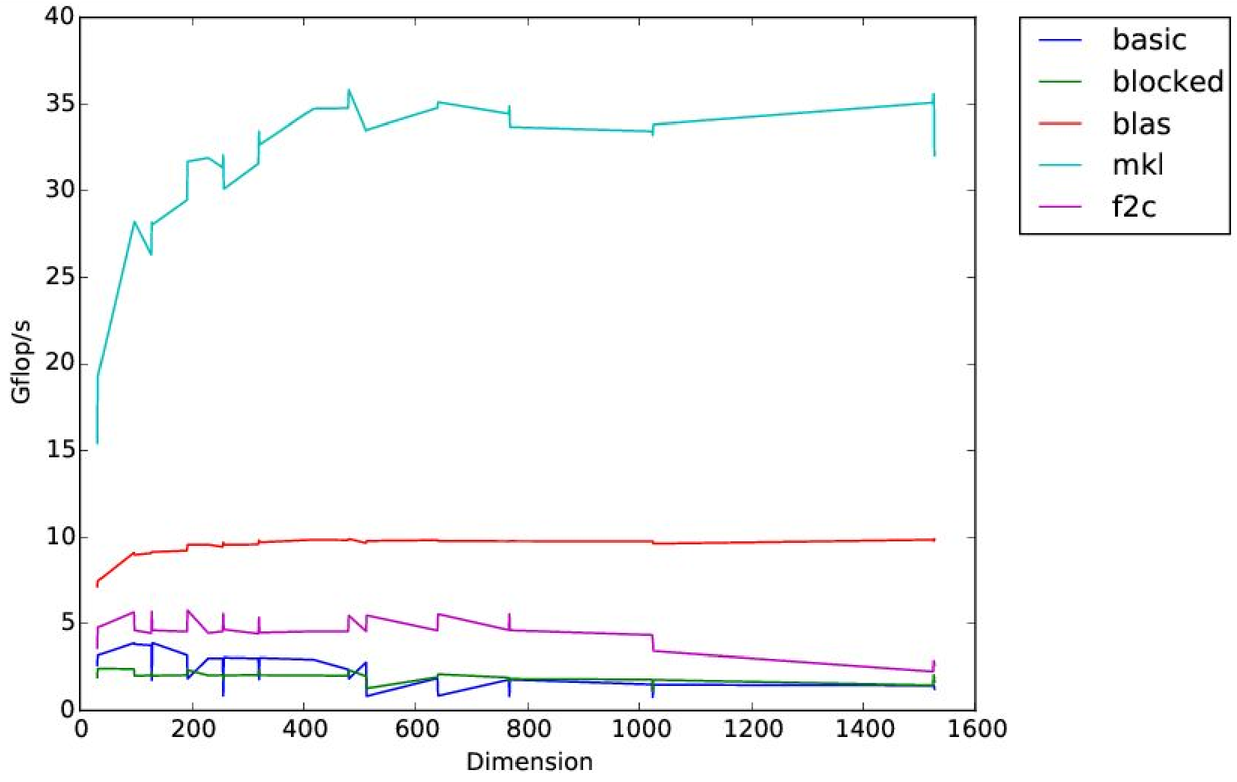
2 Contents

3 Stage 1 Results

In stage 1, we experimented with many different optimization methods to see which techniques may be viable for the final project.

3.1 Default Tests

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

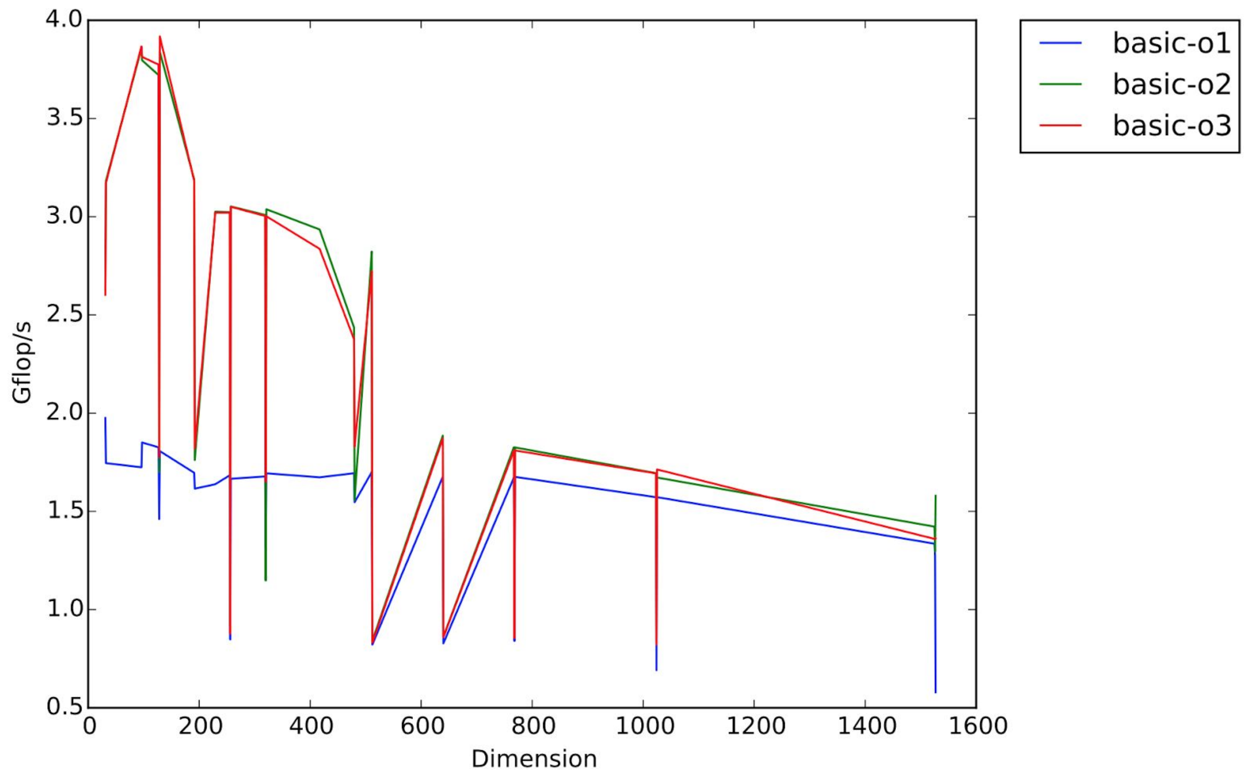


The default implementations for matmul

It was quite obvious that the Intel MKL could be considered the "golden standard" in terms of performance. OpenBLAS (run without loading the module) still performs well at 10 Gflops. Finally, we can see that the basic implementation simply performs poorly compared to everything else.

3.2 Compiler Optimization

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 has the worst performance, while level two and level three have similar performances. As we learn later, this is because without more aggressive optimization flags, there is nothing much to optimize.



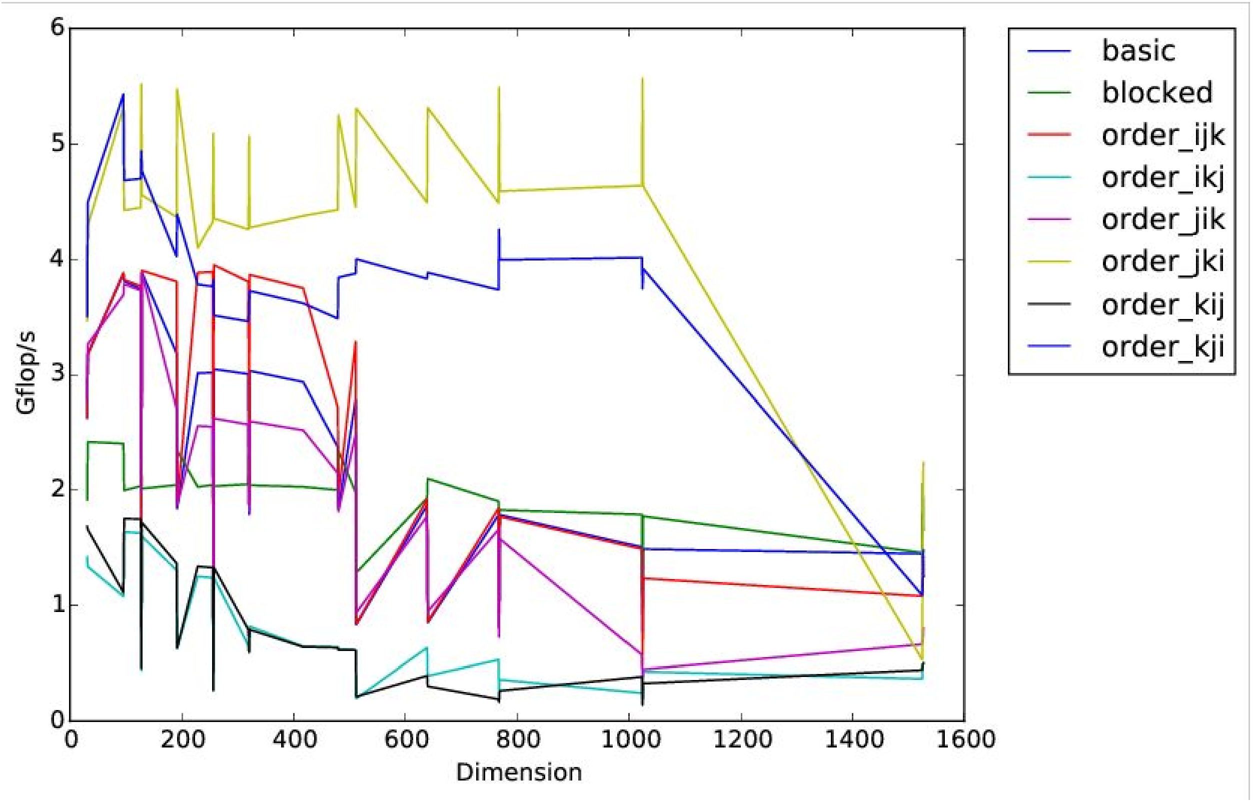
Different Compiler Optimization Options

3.3 Loop Order

The naive implementation of matrix multiplication 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. If we take a look at the inner loop read and write elements:

```
C[oj+i] += A[ok+i] * B[oj+k];
```

We see that two elements increment rows by i, and one increments by k. Thus, to improve locality, i should be the inner loop, k in the middle loop, and j in the outer loop. Furthermore, since the access to B does not depend on i, we can store it in a variable before the loop.

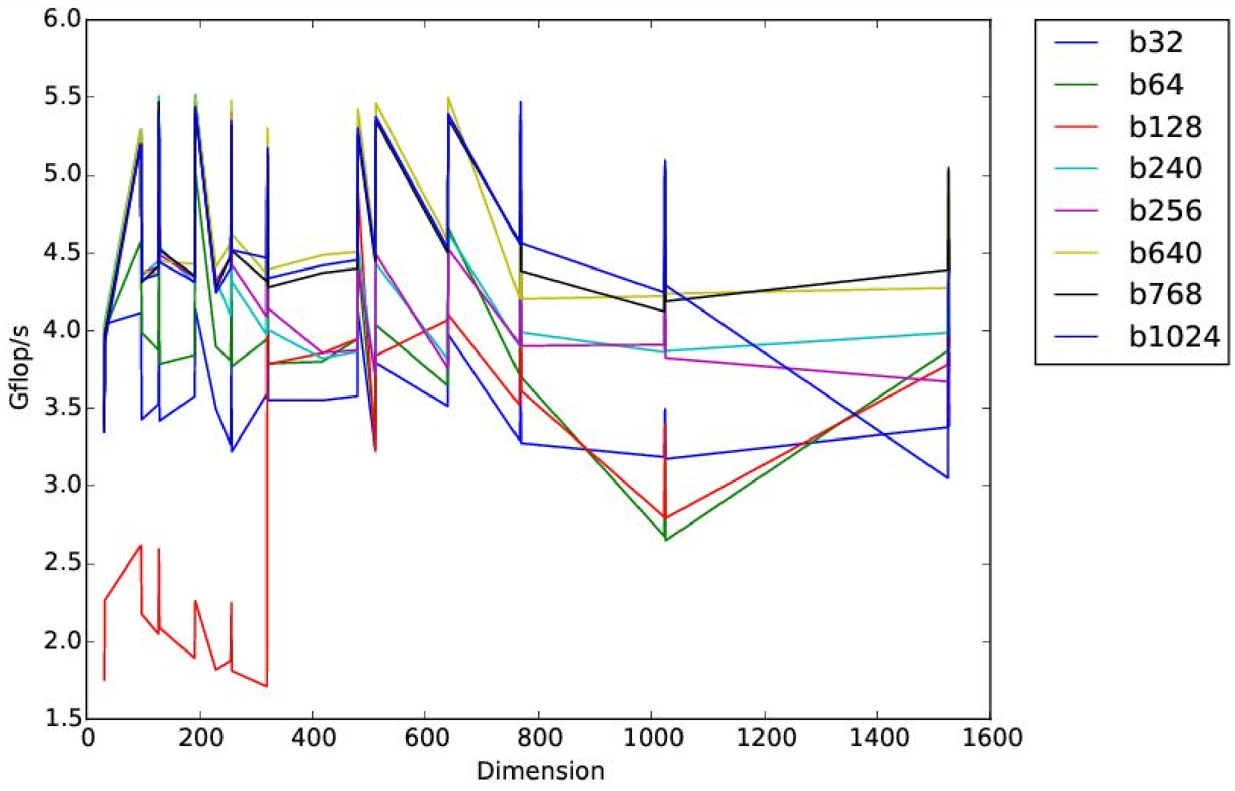


Different loop orders vs basic and blocked implementations

3.4 Blocking and Block Size

According to the "Tuning on a Single Core" note by Professor 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.

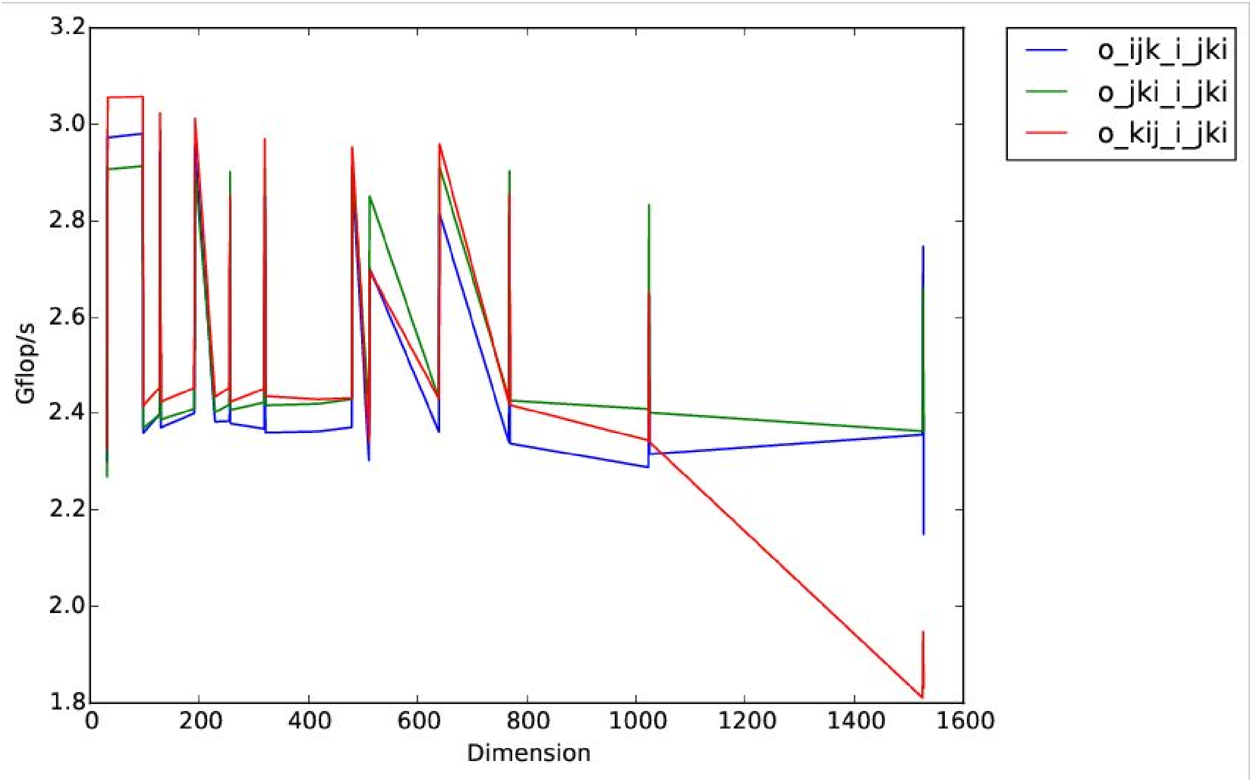
For now, we only studied partitioning the whole matrix once, and consider the j,k,i inner looping order we found earlier. The following plot shows the results of performances for different blocks. Interestingly, we find that large block sizes, such as 640 and 768, performed the best. We could not find an explanation for this behavior, but the trend did not follow in later techniques.



Different block sizes

3.5 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 as block size grows bigger, the effect of outer looping orders will make less impact.

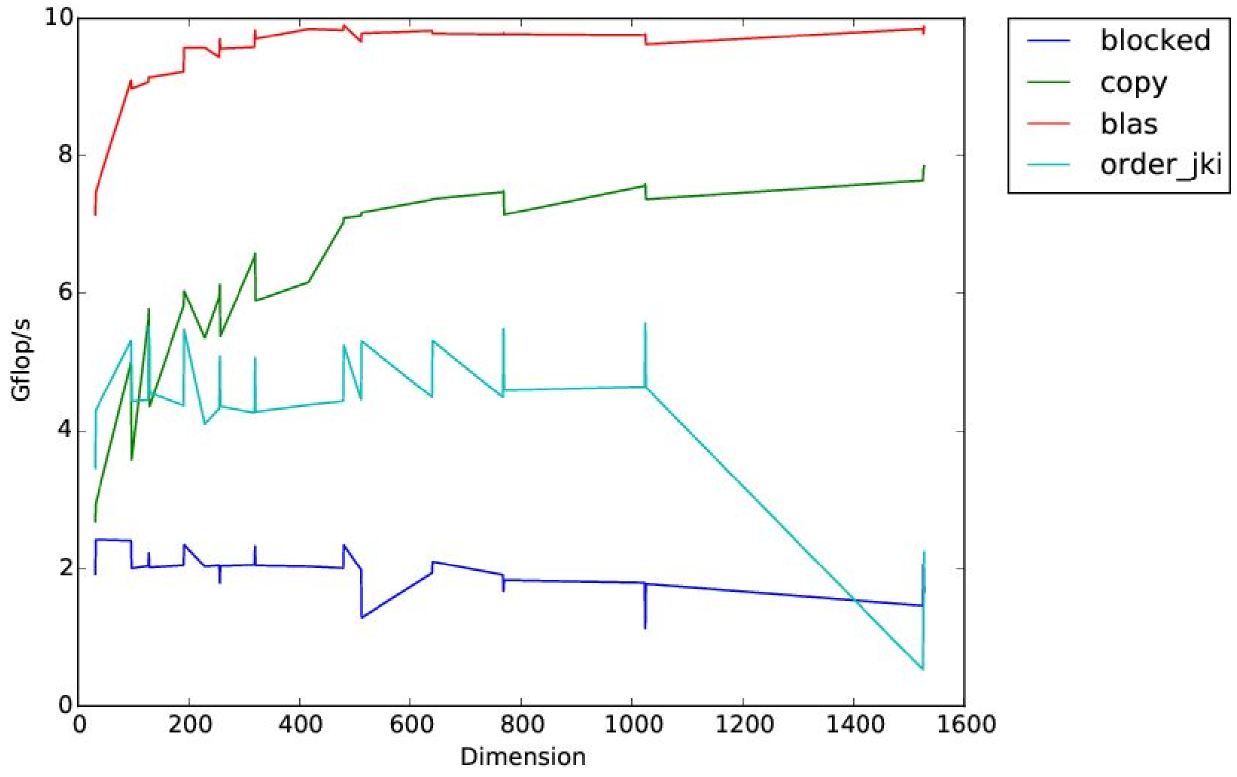


Outer loop order with jki inner loop order

3.6 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.

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 false BLAS.

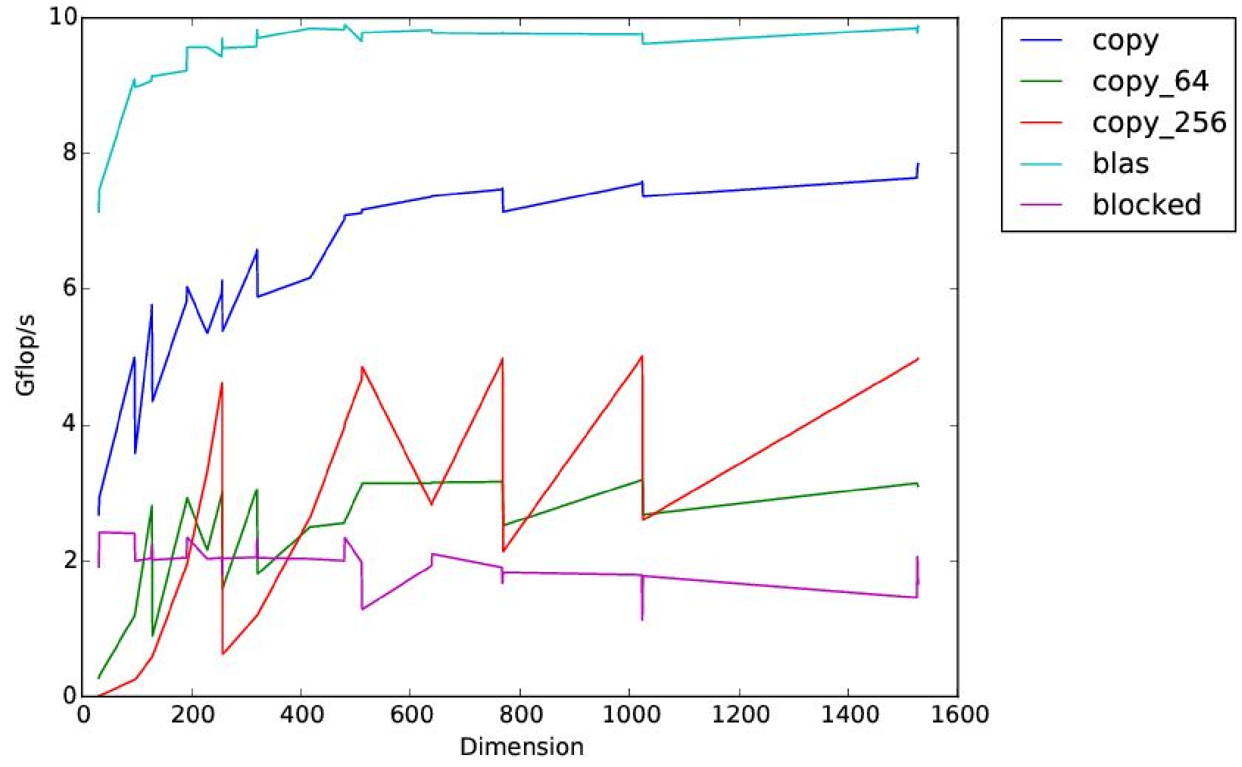


Copy-optimized blocking vs BLAS, loop order and naive blocking

3.7 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.

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.



Copy-optimized blocking at different block sizes

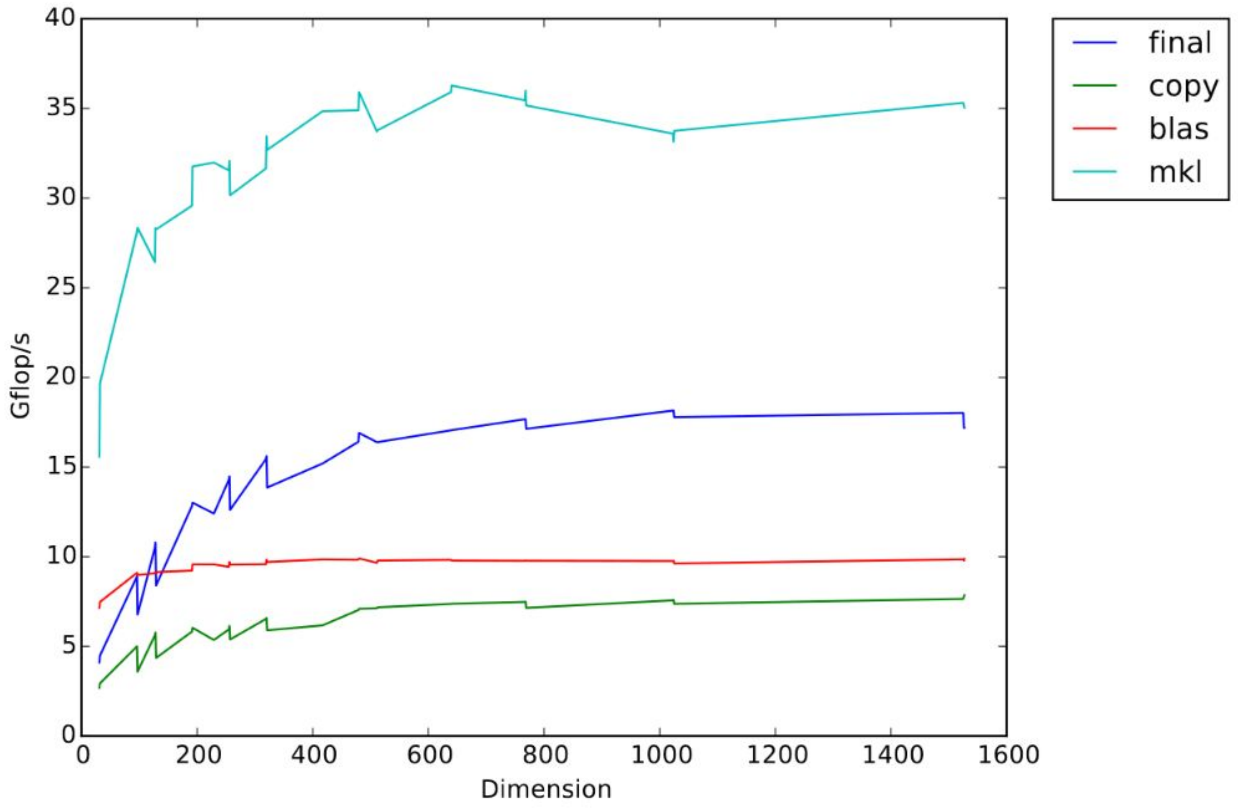
4 Final Results

After reviewing feedback on our Stage 1 report and investigating algorithms used by other groups, we tested them with our best attempt so far. Additionally, we looked into writing AVX instructions manually.

4.1 Compiler Flags Revisited

It turns out there are many more compiler flags available to us. After taking a look at Group 16's report (https://github.com/kenlimmj/matmul-), we found many compiler flags for aggressive optimization. Aside from -parallel (since we doing serial tuning), adding the compiler flags to icc resulted in almost double the performance.

```
OPTFLAGS = -O3 -fast -ftree-vectorize -opt-prefetch
-unroll-aggressive -ansi-alias -restrict -xCORE-AVX2
```



Compiler options added to copy optimizations

If we use objdump and check the instructions used by the compiler, we see that it actually utilizes the AVX vector operations for us. We can see this happen when we add the vectorization report by adding the flags

```
-qopt-report-phase=vec -qopt-report=5 -qopt-report-file=stdout
```

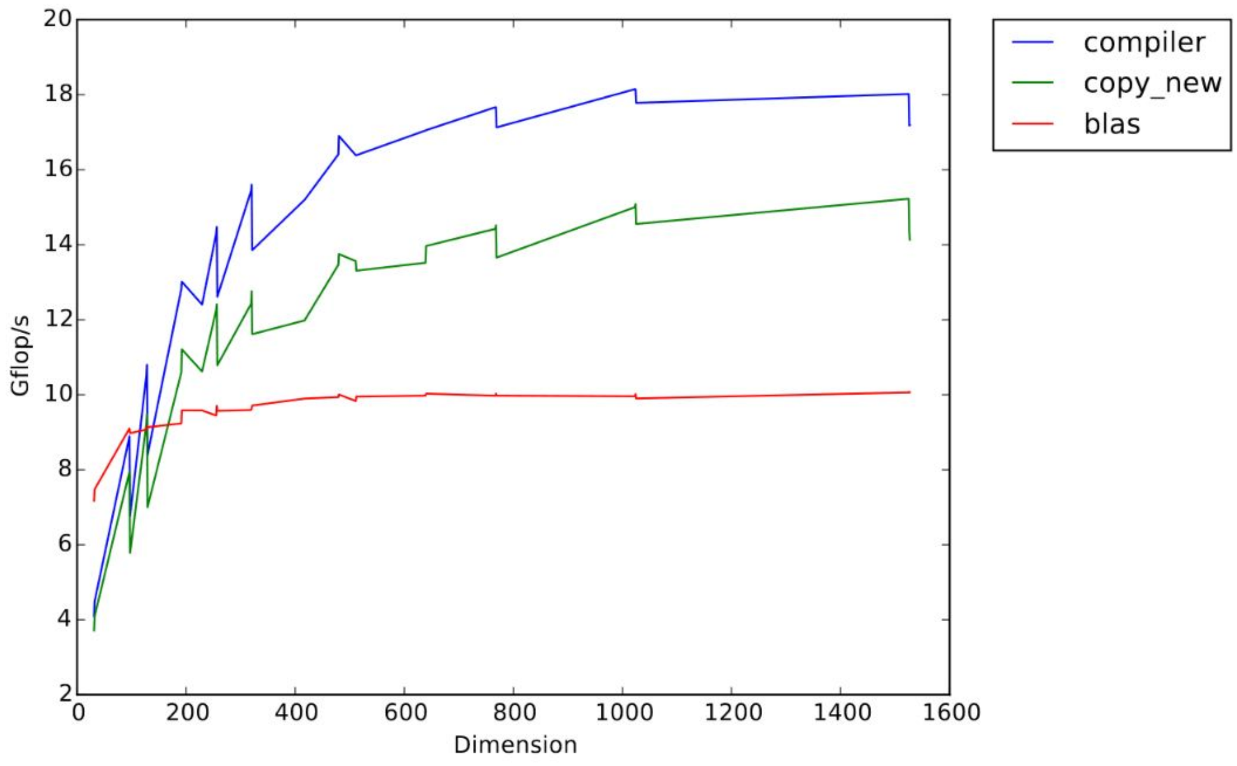
The report shows that the kernel loop is successfully vectorized.

4.2 Annotations

As Professor Bindel mentioned in class, and as Group 6 (https://github.com/mwhittaker/matmul-) reported in Stage 1, the `aligned` and `restrict` annotations can improve memory access times. According to Intel's article on data alignment,

> " For the Intel® Many Integrated Core Architecture (Intel® MIC Architecture) such as the **Intel® Xeon Phi™ Coprocessor**, memory movement is optimal when the data starting address lies on 64 byte boundaries. Thus, it is desired to force the compiler to create data objects with starting addresses that are modulo 64 bytes. "
>
> https://software.intel.com/en-us/articles/data-alignment-to-assist-vectorization

Adding the `restrict` keyword to the kernel function arguments, `__assume_aligned(A, 64)` to the beginning of the kernel and `__attribute__((aligned(64)))` to malloc improved performance by a few Gflops/s.

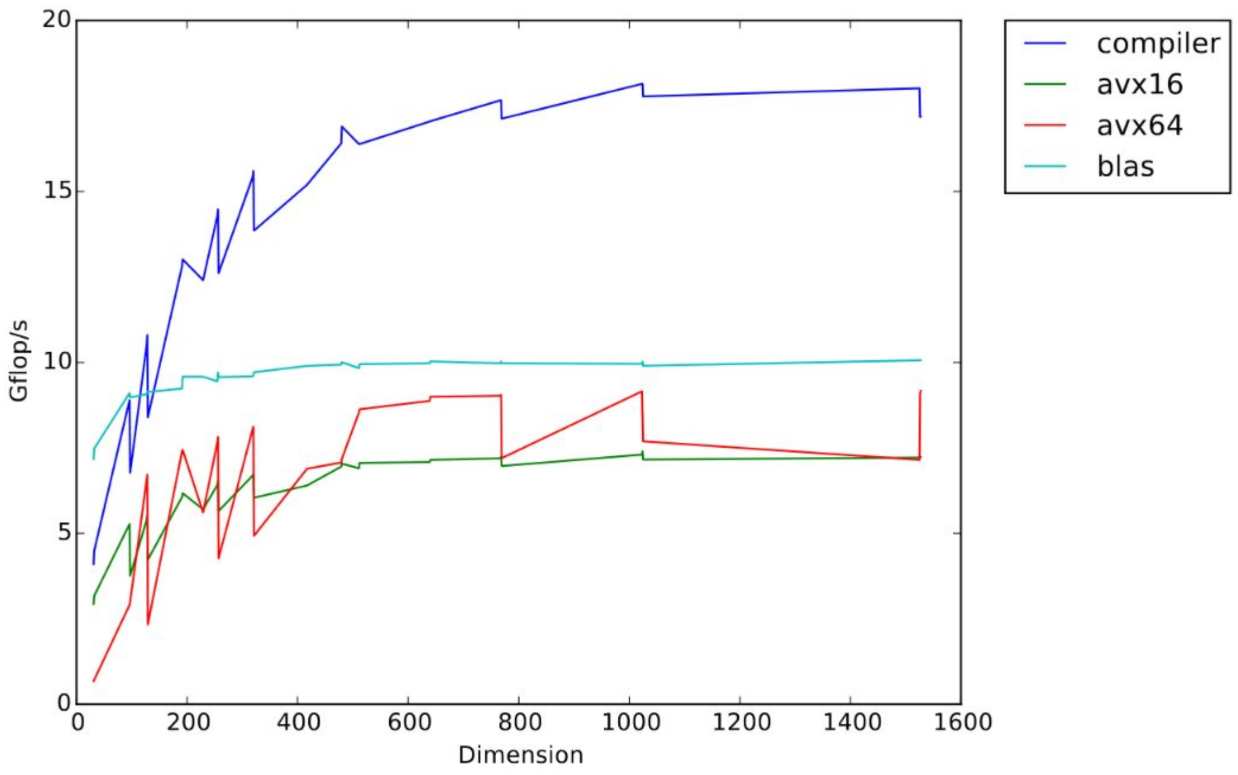


Compiler and copy_new both have compiler flags, but copy_new does not have annotations

4.3 AVX Instructions

Instead of having the compiler generate AVX instructions for us, we also attempted at manually specifying instructions. The first attempt with utilizing AVX was to substitute our inner loop with unrolled AVX instructions (block size 16), effectively performing a dot product 4 elements at a time. However, this actually degraded our performance.

After reading a couple slides from CERN, we see that the pipeline for AVX instructions is not efficient for only a few AVX operations. We increase the block size, but still do not see much improvement in performance.



Copy optimization with compiler and annotations vs manual AVX instructions

4.4 Vectorization Report

Vectorization report provides very important information about what optimizations compilers have been doing for us. More specifically, the vectorization report informs us which loops are being vectorized and which loops are not. Moreover, the report provides detailed information about each vectorized loop. For example, scalar loop cost, vector loop cost, estimated potential speedup and number of lightweight/heavy-overhead vector operations.

To get a vectorization report, we add the following set of compiler flag

```
-qopt-report-phase=vec -qopt-report=5 -qopt-report-file=stdout
```

where parameter 5 stands for asking non-vectorized loops and adding dependency information.

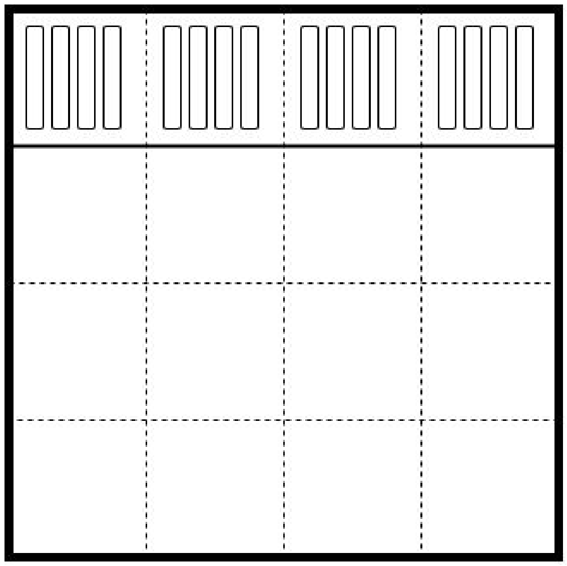
One thing we found with vectorization report is that copy optimization is vectorized while dot products in AVX instructions are not. And this partly explains why copy optimization is doing better than AVX in our case.

We also found 1 high overhead vectorized loop in the copy optimization implementation. But since it is in the part where we copy the result back, whose computational complexity is of order O(n^2), it would not affect the overall performance too much for large n.

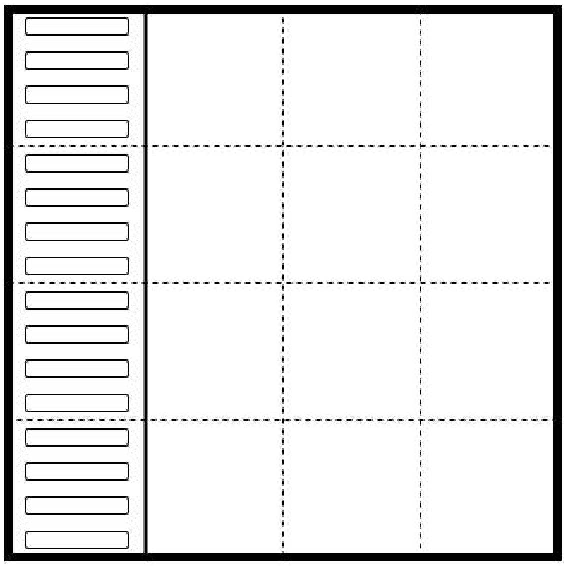
4.5 AVX and Diagonal Sums

In order to have a vectorizable loop, it seemed like we could not perform any sort of horizontal sum. Instead, we "comb" matrices using the load function to get adjacent vectors and multiply corresponding vectors to obtain matrix diagonals.

As an example, consider two matrix blocks A and B, where A and B are of size 16 x 16.

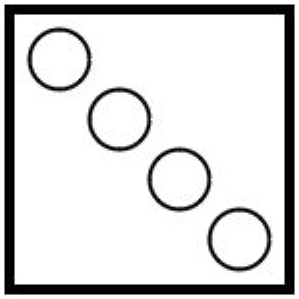


A



B

From an overall perspective, if we take a 4 element row of A and multiply it by a 4 element column of B, we will get a 4 x 4 matrix in C. Looking closer, if we take each of the shown vectors in A (call them A_0, A_1, … , A_{15}) and those in B (B_0, … , B_{15}) and sum the element-wise products, we end up with the 4 elements making up the diagonal in C.



C

Furthermore, if we perform a 1 element right cyclic shift on $B_0, \ldots, B_{15}$ and perform the multiplications again, we get the next diagonal over in C. Continue doing shifts and arithmetic and we get the entire 4 x 4 matrix in C. This is equivalent to Professor Bindel's kernel except ending up with a 4 x 4 matrix instead of a 2 x 2 matrix.

Note that for efficiently loading of memory in A and B, we require a small change to the copy algorithm.

The algorithm in code/pseudocode is as follows:

```
__m256d ma[BLOCK_SIZE]; // Matrix A comb
__m256d mb[BLOCK_SIZE]; // Matrix B comb
__m256d mc[4]; // Matrix C 4 x 4 result
__m256d md[4]; // Matrix C diagonals

for (j = 0; j < BLOCK_SIZE; j+=4) {
        // Matrix B
        mb[t = 0…BLOCK_SIZE] = _mm256_load_pd(B + j*BLOCK_SIZE + 4*t);

        for (i = 0; i < BLOCK_SIZE; i+=4) {
                // Matrix A
                ma[t = 0…BLOCK_SIZE] = _mm256_load_pd(A + i*BLOCK_SIZE + 4*t);

                // Previous values of C
                mc[t = 0…4] = _mm256_load_pd(C + j*BLOCK_SIZE + 4*i + 4*t);

                // Initialize diagonals
                md[t = 0…4] = _mm256_setzero_pd();

                // Diagonal 0
               md[0] = _mm256_fmadd_pd(ma[0…BLOCK_SIZE], mb[0…BLOCK_SIZE], md[0]);

                // Shift matrix B
               mb[t] = _mm256_permute4x64_pd(mb[0…BLOCK_SIZE], 0x39);

                // Repeat fmadd and permute for all 3 diagonals

                // Order diagonals to be stored
                mc[0] = _mm256_add_pd(
                _mm256_shuffle_pd(
                        _mm256_shuffle_pd(md0, md2, 0x0),
                        _mm256_shuffle_pd(md3, md1, 0x9),
                        0xc
                ), mc[0]);

                // Repeat above monstrosity for other vectors of matrix C
                // Store back into result
                for (t = 0; t < 4; ++t)
                   _mm256_store_pd(C+ j*BLOCK_SIZE + 4*i + 4*t, mc[t]);
        }
}
```

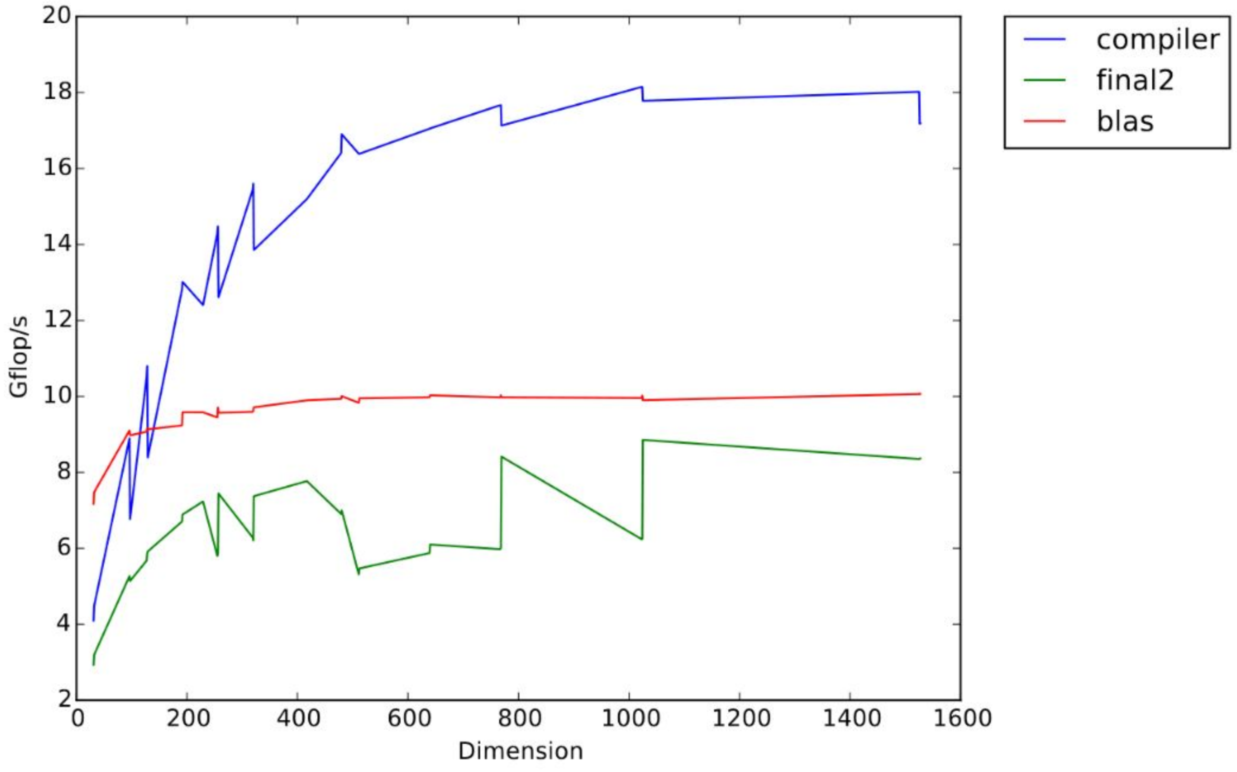
The set of shuffle operations near the end arranges the elements from the diagonals so that the final store operation directly copies the results to matrix C. If we store the diagonals directly, we would end up with a matrix that looks like

```
0 1 2 3
3 0 1 2
2 3 0 1
1 2 3 0
```

Whereas we would like to end up with a matrix that looks like

```
0 1 2 3
0 1 2 3
0 1 2 3
0 1 2 3
```

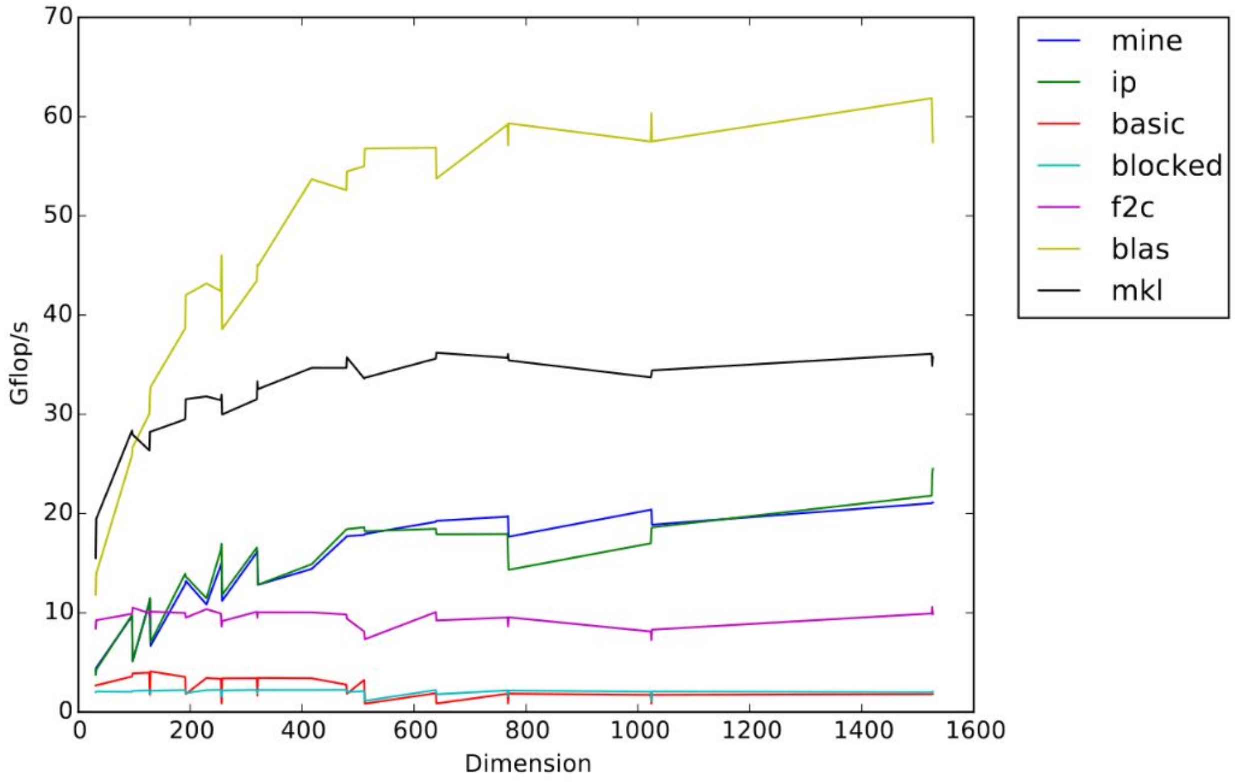
However, despite the completely vectorized AVX instructions, we did not see any performance increase.



Final2 is our attempt at manual AVX instructions, but still does not compare to copy optimization

5 Conclusion

We finally choose copy optimized code (with block size 32 and aligned size 64) with compiler flags and annotations introduced in previous sections. The performance goes up to around 20 Gflops/s. Later we also include another compiler flag -ip, which enables interprocedural optimization in a single source file. The difference is negligible, but note that for larger n the performance with -ip goes to about 25 Gflops/s. We would expect that using compiler flag -ip would be a good idea for larger n.



Final set of results, with our best method as just over 25 Gflops/s