CS 5220 Assignment 1 - Phase 1 Report

Group 24: Xiang Long (XL483), Yu Su (YS576), Joshua Cohen (JBC264)

1 Index ordering

The first optimization we tried was to permute the ordering that the matrix indices were being looped over. This could possibly improve performance through better cache locality if the fastest inner loops were making smaller strides between memory locations that are accessed. We take the convention that loop indices i, j and k denote the positions in the formula below that represents our matrix multiplication:

$$\mathbf{C}_{ij}^{new} = \mathbf{C}_{ij} + \sum_k \mathbf{A}_{ik}\mathbf{B}_{kj}$$

This is the same convention taken by the existing code. Starting with the given blocked implementation, we permuted the index order in basic_dgemm, thereby changing the order that elements of the arrays were accessed. For the best-performing permutations, we made a further improvement by modifying square_dgemm into the same index order so that the blocks are accessed in the same sequence (denoted by −block in the graph below).

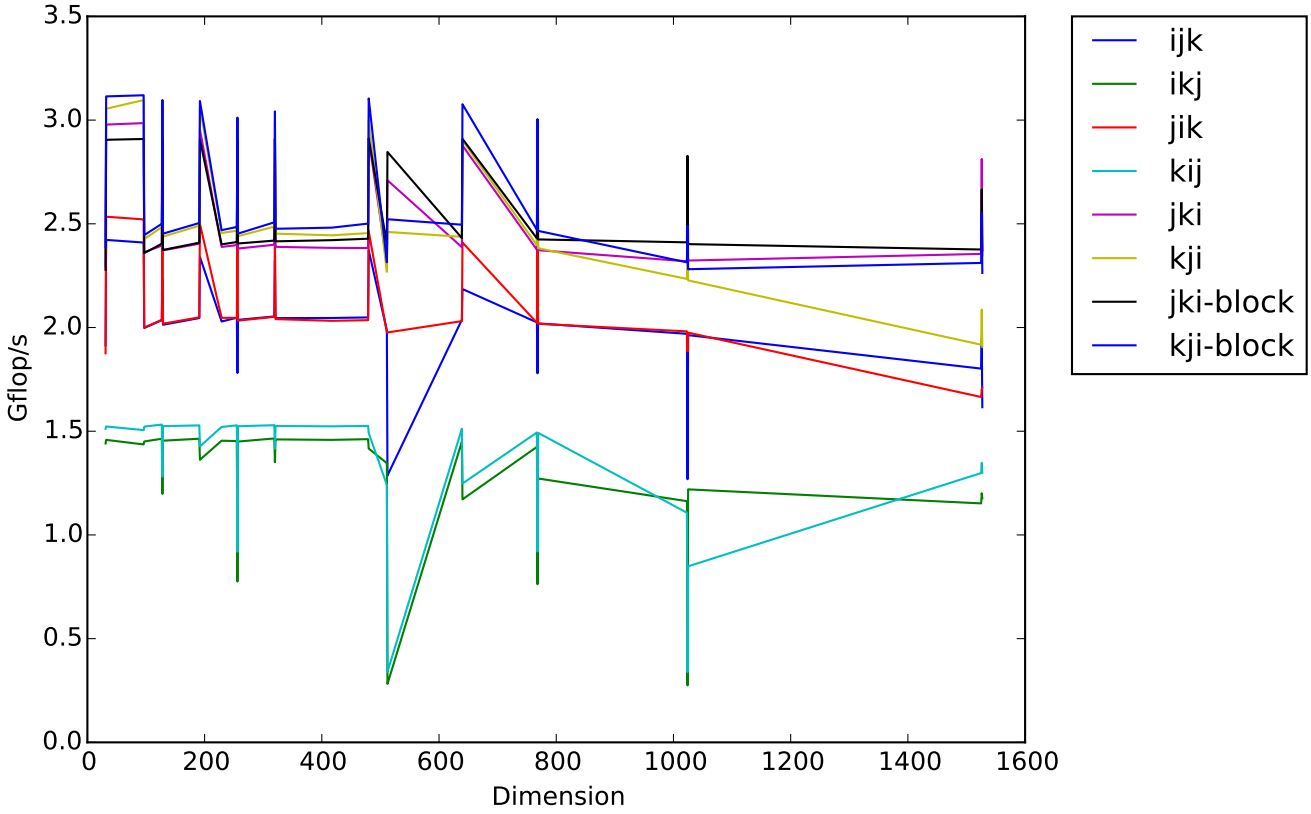

Figure 1: Comparison of loop index orderings. ijk denotes index i is the outermost loop and k is innermost.

Both jki and kji ordering performed better than the others, and this makes sense since the innermost loop is over i, which is a row index and so incrementing it causes the smallest stride when the matrices are in column-major order. We took forward kji with block iterations in the same ordering, since in later experiments we found it actually outperforms jki ordering with the same optimizations.

2 Block sizes

Next we experimented with varying the block size in the blocked implementation by modifying the BLOCK_SIZE constant. The aim is to have blocks that fit within the caches in the processors in order to have better cache locality. Each Xeon E5-2620 v3 core has 64KB and 256KB of L1 and L2 caches respectively. The aim is to fit blocks into these sizes, and we did not consider the L3 cache since it is shared between cores and conflicts there is more out of our control. Since each matrix element requires 8 bytes of storage, and all three matrices are read from in a single add-multiply operation, we can perform the following calculation to obtain an "ideal" block size that will fit three blocks in the L2 cache:

$$\text{BLOCK\_SIZE} = \sqrt{\frac{2^{18}\text{B in L2 cache}}{8 \times 3}} \approx 104.51$$

We can perform a similar calculation for the 64KB L1 cache. Although this tells us that 104×104 sized blocks are likely to be optimal, we prefer powers of 2 as they have the potential of facilitating future optimizations based on algorithms for matrix multiplication that have a better asymptotic complexity. In any case the block size can always be easily changed in the future if we decide to retune this part of the kernel. Therefore we predict that the closest power of 2 to 104, i.e. 128, will have the best performance. The graph below shows the results of our experiments for different block sizes:

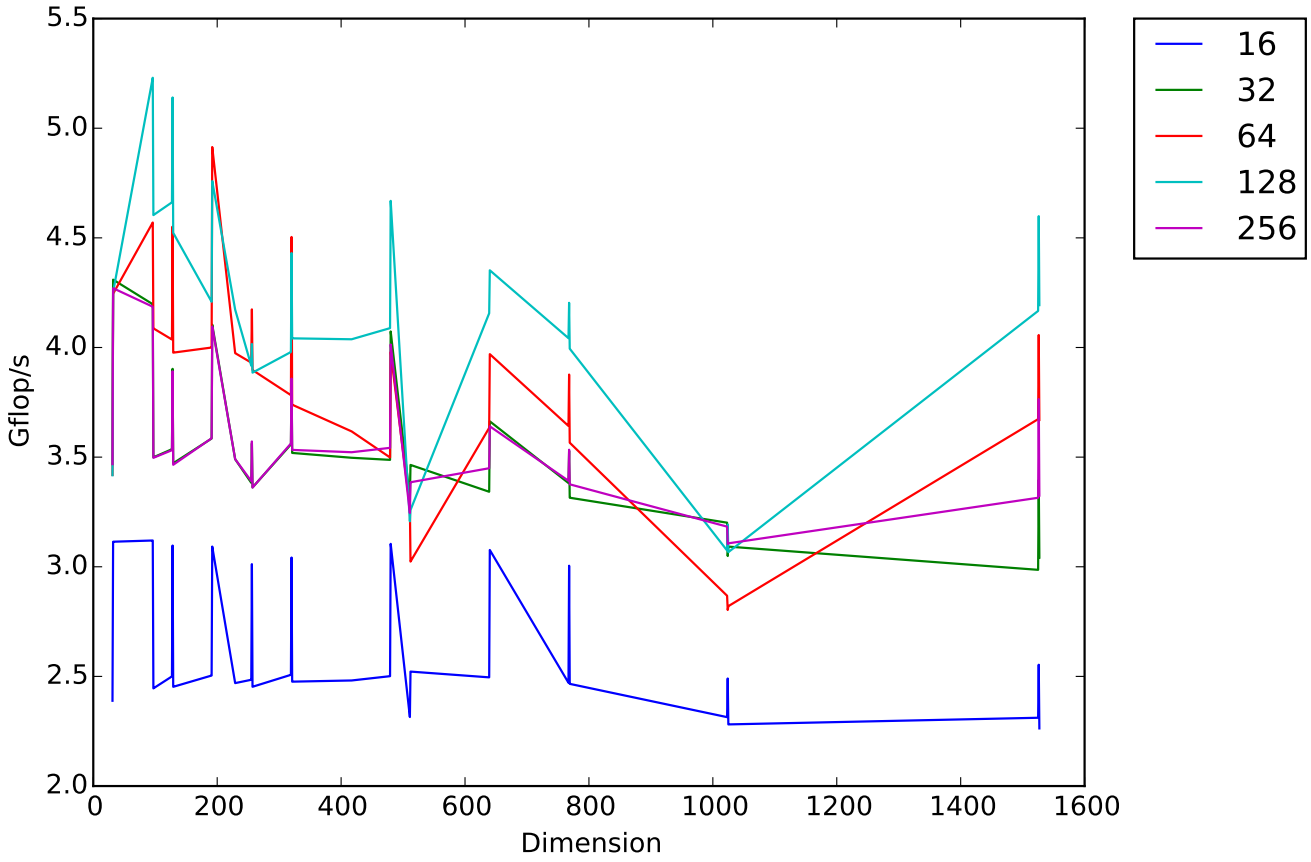

Figure 2: Comparison of block sizes with different values of the BLOCK_SIZE constant.

The results confirm our prediction.

3 Copy optimization

Even though we may have obtained a good block size, we could still have bad cache locality since each column in a block is in reality one length of the full matrix away from each other. To maximize the possibility of cache hits, we should place all the data in a block into a single contiguous region of memory. This is done by copying the relevant

data from the current blocks in each of the matrices to their own small matrices, compute the product and then write back the result. We implemented this technique and the following graph shows the performance results, again with a variation in block sizes:

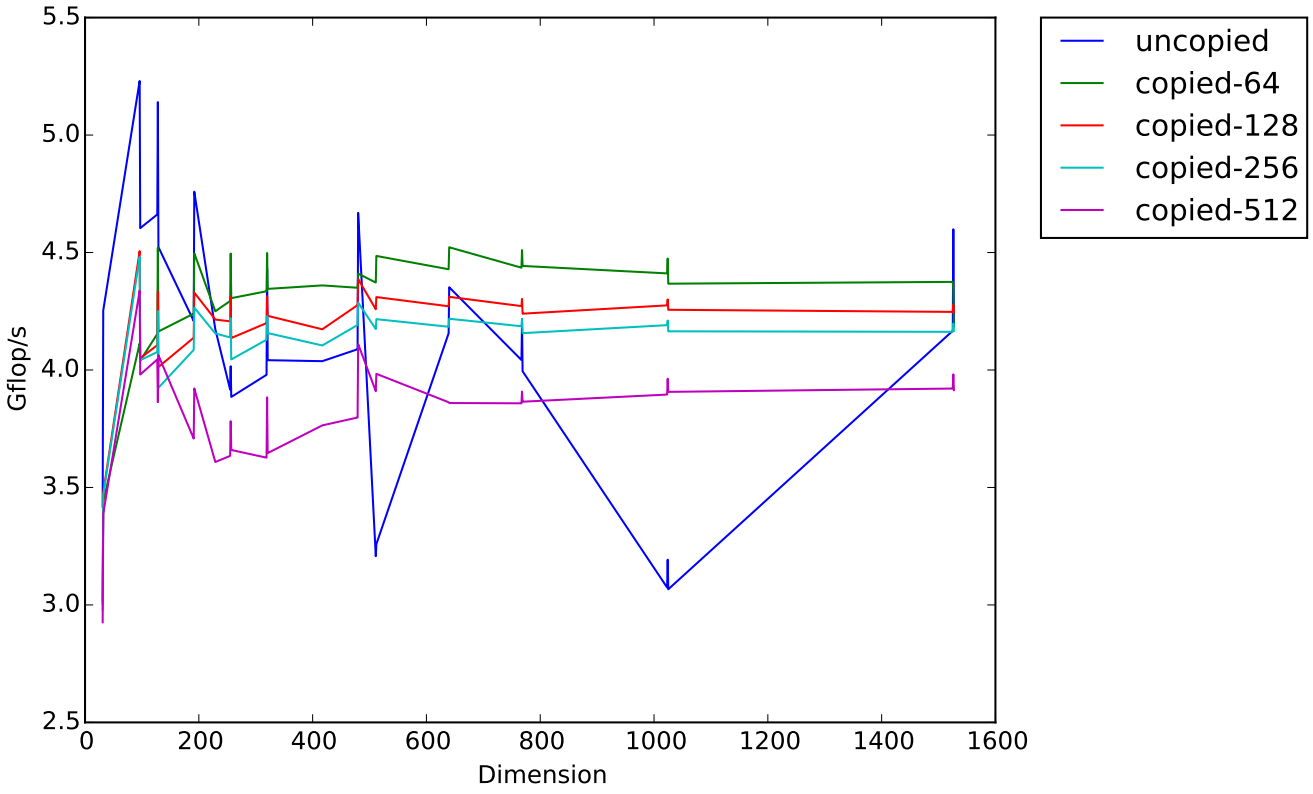

Figure 3: Performance with or without copy optimization, with different block sizes.

We found that this time a block size of 64 actually performs better than 128, which is against our theoretical prediction from before. Careful examination of our code revealed that we were allocation and freeing new blocks of memory every time the do_block function is called. Since the malloc function could have higher overheads when it is asked to allocate larger blocks of memory, this could be the dominant factor in why we obtained a counter-intuitive result. We modified our code so that it now pre-allocates fixed-sized empty blocks that could be reused for multiple do_block function calls in the vast majority of cases. It will only allocate more blocks of memory for some specific do_block calls if we reached a boundary condition at the edge of one of the matrices such that the default size for a block must change. The results for the updated code is shown below:

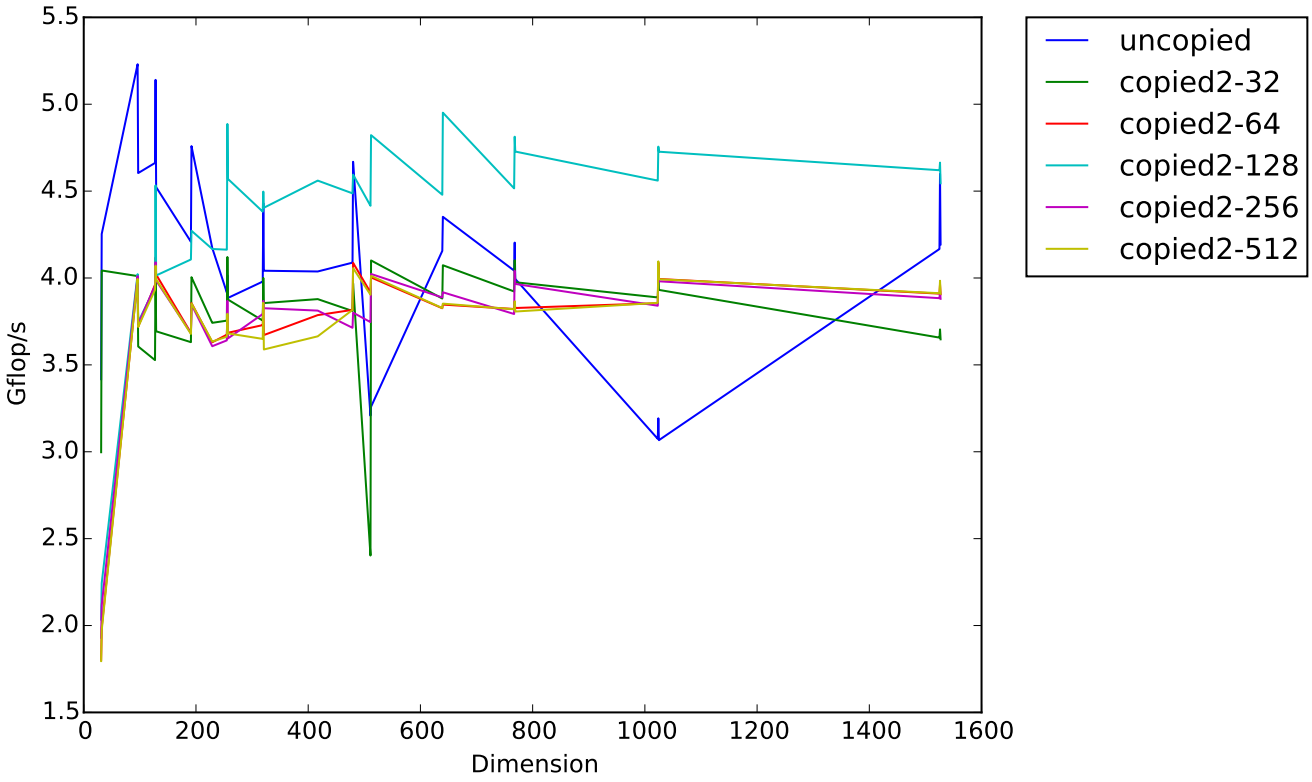

Figure 4: Performance with or without copy optimization, with different block sizes, take 2.

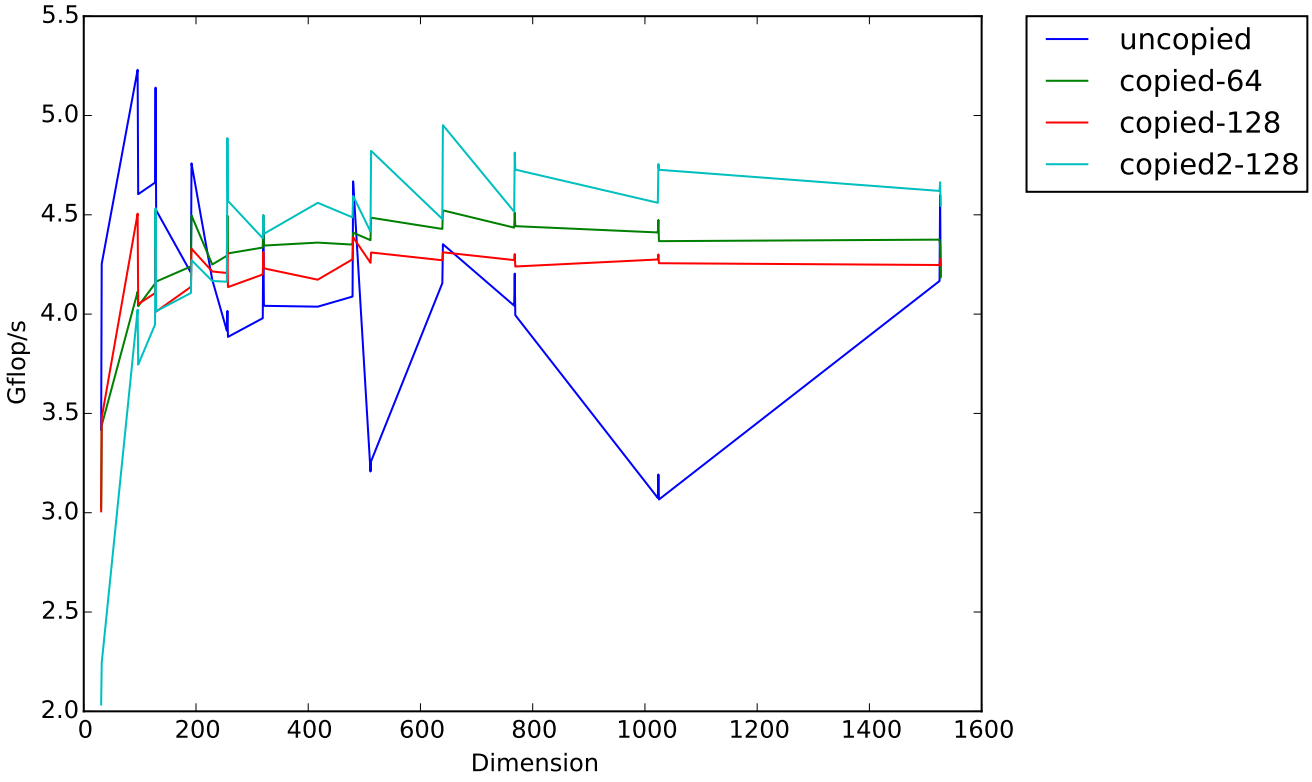

Figure 5: Comparison of the best-performing copy optimization implementations against not employing copy optimization.

This once again confirmed our prediction that a block of 128 is the best out of powers of 2, and the new implementation also significantly outperforms the old one.

4 Compiler optimizations

In addition to the standard optimizations performed with the "`-O3`" flag, compilers can also optimize the code more specific to the target platform. For example it may generate instructions specific to a processor architecture, thereby performing some types of tasks more efficiently, or it could unroll loops and execute the code in multiple iterations at once using vector registers. Using the icc compiler, we found the following flags improved the performance greatly:

```
-O3 -fast -xCORE-AVX2 -unroll-agressive -fp-model fast
```

These flags tell the compiler to generate AVX2 instructions, which the Xeon Phi processors are equipped to handle[1], unroll loops aggressively, and use floating-point computations that are less precise but faster. For gcc, we decided upon these flags:

```
-O3 -march=native -mtune-native -ftree-vectorize -funroll-loops -ffast-math -msse4
```

Unfortunately the GCC toolchain on the Totient cluster does not seem to support AVX2 instructions. We therefore set it to use the best available instructions and tuning for the platform the code is being compiled on, which if done on the Totient head node it should be the same architecture as the compute nodes. The compiler is likely to generate SSE instructions in this case. We also directed GCC to perform the same optimizations of more advanced unrolling of loops and using faster but imprecise floating-point methods.

A further compiler optimization we performed was to specify all of the pointers representing matrices as `restrict`. This guarantees to the compiler that different pointers will not alias and so it is able to perform further optimizations based on this assumption. The following graph shows the performance of our code compiled with icc and gcc, with or without our optimization flags, and with or without marking pointers as `restrict`:

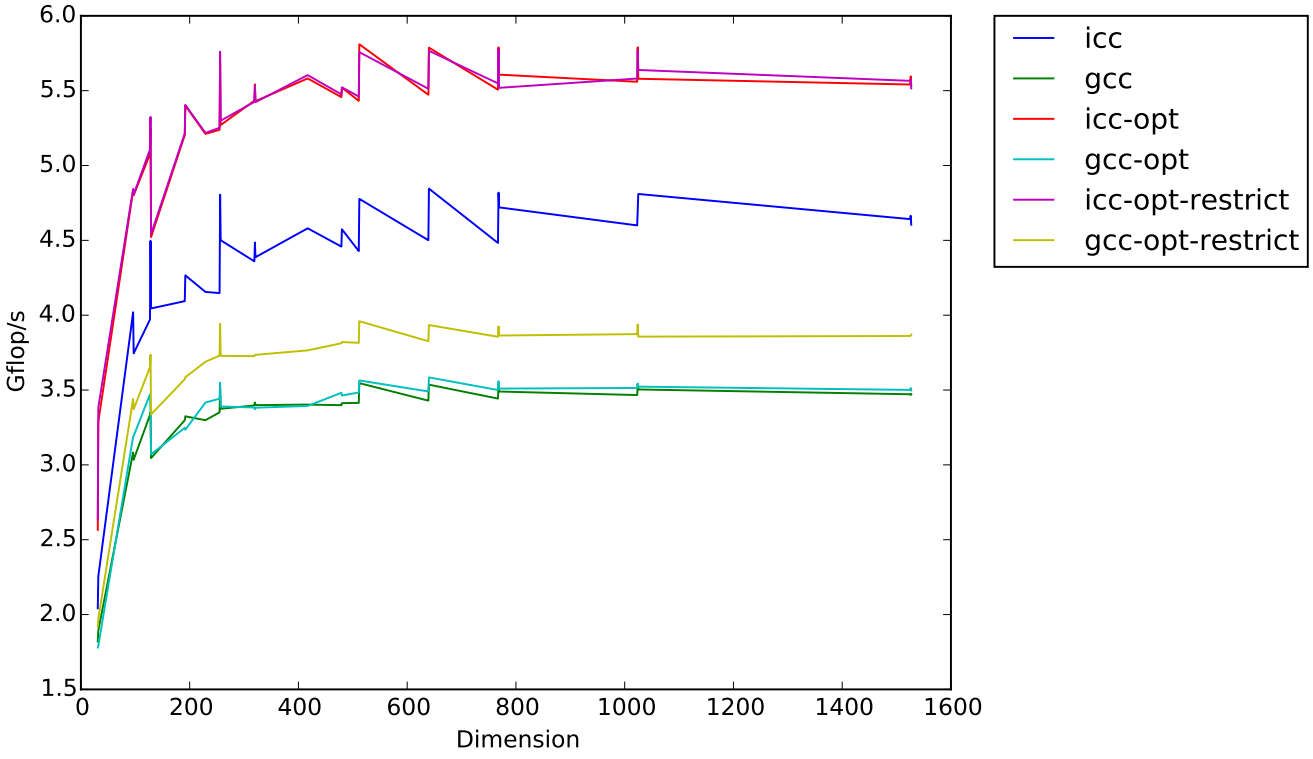

Figure 6: Comparison of compilation modes.

[1] http://ark.intel.com/products/83352/Intel-Xeon-Processor-E5-2620-v3-15M-Cache-2_40-GHz

The results show that icc-compiled code greatly outperforms gcc, the optimization flags makes a significant improvement, and marking pointers as restrict also makes a big improvement if gcc is used. Although restrict does not seem to make a big difference for optimized icc, we decided to keep it in the code in case it improves future optimizations.

5 Conclusion

We have tuned the existing blocked implementation through modifying the loop index orders and block sizes, and we have implemented copy optimization as well as enabling flags for compiler optimizations. Our best-tuned performance is compared against the prescribed implementations as below:

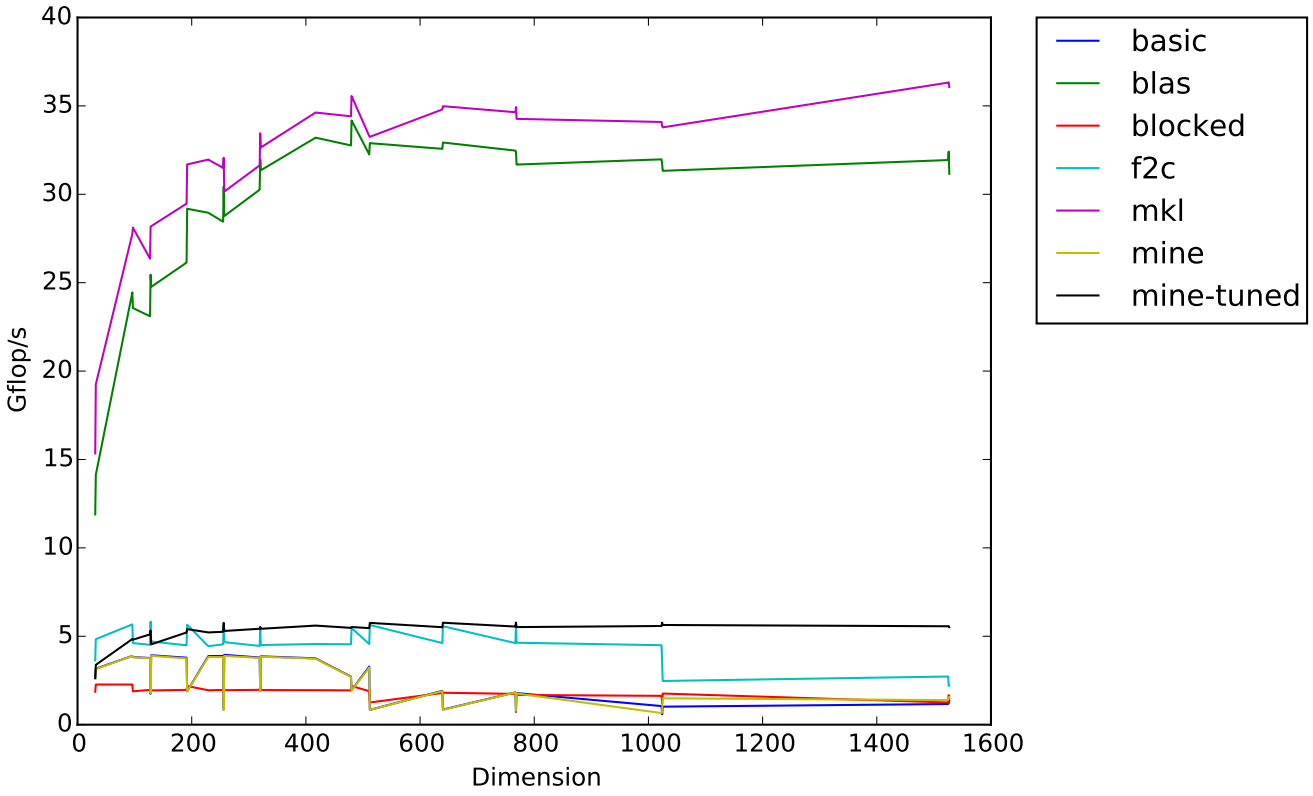

Figure 7: Comparison of our tuning against existing implementations.

A detailed table with the actual figures is given in the next page. Our tuned implementation already significantly outperforms the original ones except BLAS and MKL, and is approaching 5.8GFLOPS. We plan to perhaps further improve blocking performance by adding block layers. We will also explore auto-tuning and more advanced matrix multiplication algorithms, which could yield more opportunities to optimize in addition to lower asymptotic complexity.

Matrix Size	mine	blocked	tuned
31	2613.04	2388.94	2636.11
32	3142.65	3114.24	3383.66
96	3886.49	3119.8	4844.08
97	3814.8	2445.39	4805.93
127	3752.21	2500.01	5107.39
128	1773.67	3096.84	5324.48
129	3891.35	2452.51	4541.76
191	3746.38	2503.97	5214.54
192	1882.45	3092.19	5404.6
229	3829.71	2469.46	5217.86
255	3821.91	2484.61	5251.93
256	848.628	3012.05	5760.26
257	3889.34	2452.69	5299.44
319	3774.51	2507.23	5424.96
320	1888.54	3042.11	5531.25
321	3845.1	2476.07	5423.15
417	3729.15	2481.46	5603.88
479	2702.33	2501.06	5475.52
480	1932.99	3104.5	5522.28
511	3232.74	2314.59	5460.85
512	832.796	2521.99	5757.42
639	1895.43	2495.68	5512.81
640	841.93	3076.58	5766.44
767	1818.98	2470.95	5548.65
768	775.443	3004.79	5775.5
769	1763.34	2466.04	5518.94
1023	656.536	2314.25	5581.03
1024	709.53	2490.54	5771.06
1025	1491.04	2281.1	5638.26
1525	1388.22	2311.41	5565.64
1526	1507.07	2553.24	5575.08
1527	1561.24	2262.93	5515.06

Table 1: Tuned vs. untuned performance in MFLOPS.