CS 5220
Project 1 - Matrix Multiplication

Sheroze Sheriffdeen(mss385)
Weici Hu(wh343)
Qinyu Wang(qw78)

October 1, 2015

1 Introduction

Double Precision **GE**neral **M**atrix **M**ultiplication (DGEMM) is an important operation in problems that arise in scientific and engineering computing applications. Our DGEMM function takes 2 dense square double precision matrices A and B stored in column major format and returns,

$$A \times B = C \tag{1}$$

where '$\times$' represents matrix multiplication. This document describes an optimized DGEMM implementation and details the design decisions used to improve performance.

2 Implementation Overview

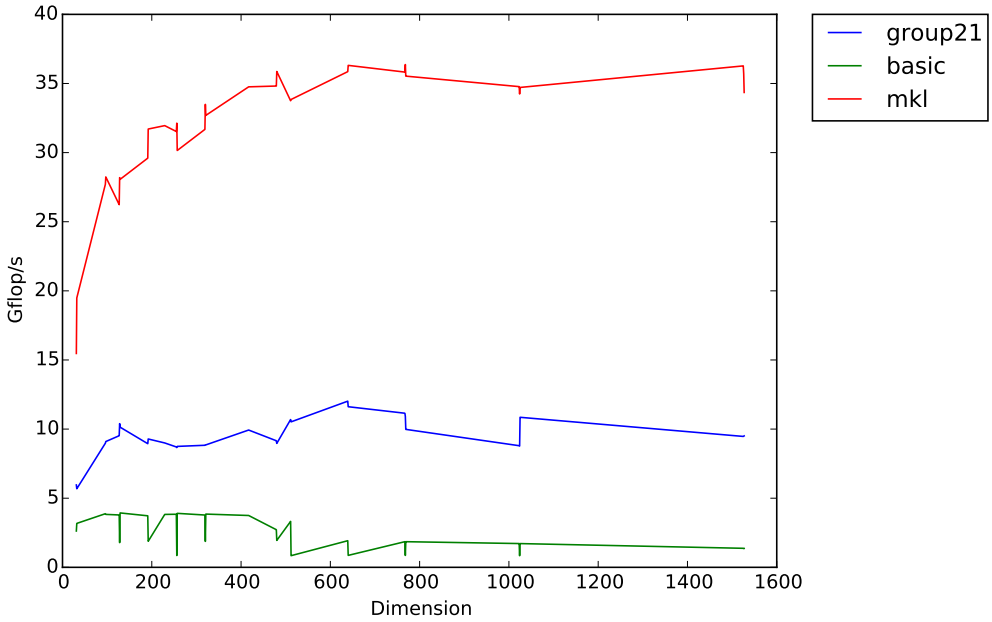

Figure 1: Performance comparison: Intel MKL vs our implementation vs the naive implementation

Our DGEMM implementation has 2 distinct modes of operation: hybrid between tight loops and blocking. On square matrices with leading dimension < `BLOCK_THRESHOLD`, our DGEMM uses tight loops with copy optimization. (Section 3.2). `BLOCK_THRESHOLD` was picked to be 400 elements. After this threshold, the blocked approach performs better.

On matrices with the leading dimension > `BLOCK_THRESHOLD`, we switch to a blocked matrix multiplication. Each block has 32×32 elements. Prior to blocking the matrices, we perform copy optimization on A (transpose and copy over data to a 64 byte aligned block obtained by Intel's `mm_malloc`). Furthermore, we also copy over data from B to an aligned block and also create an aligned storage area for C. The newly allocation dynamic memory uses zero padding so that all block multiplication is similar. (Section 3.3)

On each block, we perform the naive matrix multiplication using tight loops. To make use of the aligned data, we provide hints to the compiler by adding the `__assume_aligned` clause prior to the loop using the matrix data. [1] Furthermore, since A is transposed due to copy optimization, the innermost loop of the block multiplication is performed with stride 1.

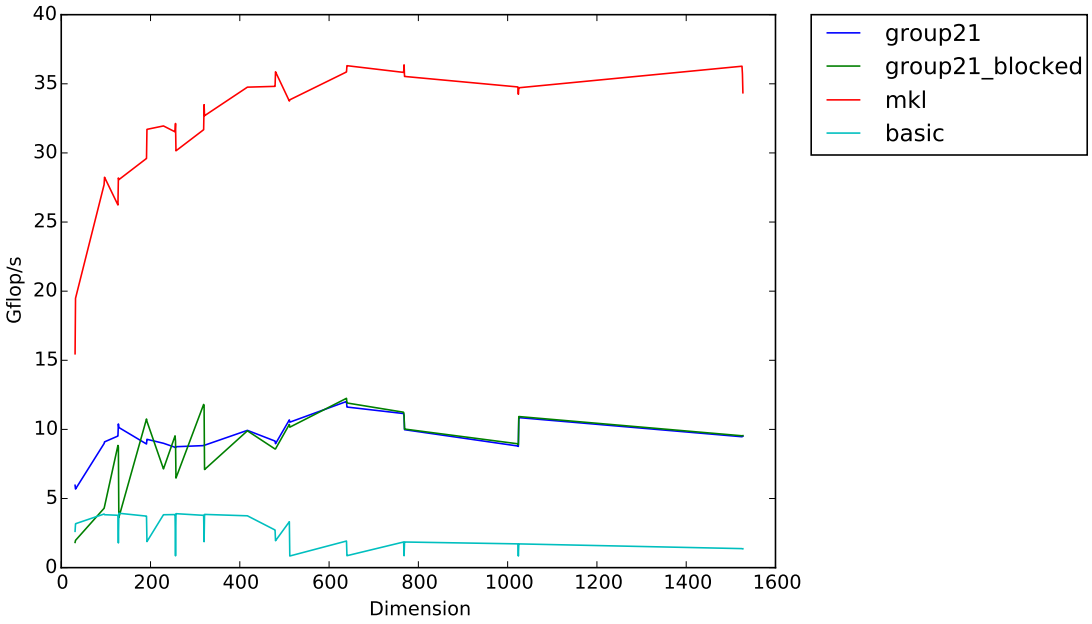

Figure 2: Performance comparison: final implementation (hybrid) vs only blocking

The compiler flags play a major part in improving the DGEMM routine performance. The flags used in the final implementation are `-O3 -fast -funroll-loops -ipo -xCORE-AVX2 -no-prec-div -ansi-alias -restrict`. The makefile used is `Makefile.in.icc`. The design decisions of each compiler flag is described in Section 3.1.

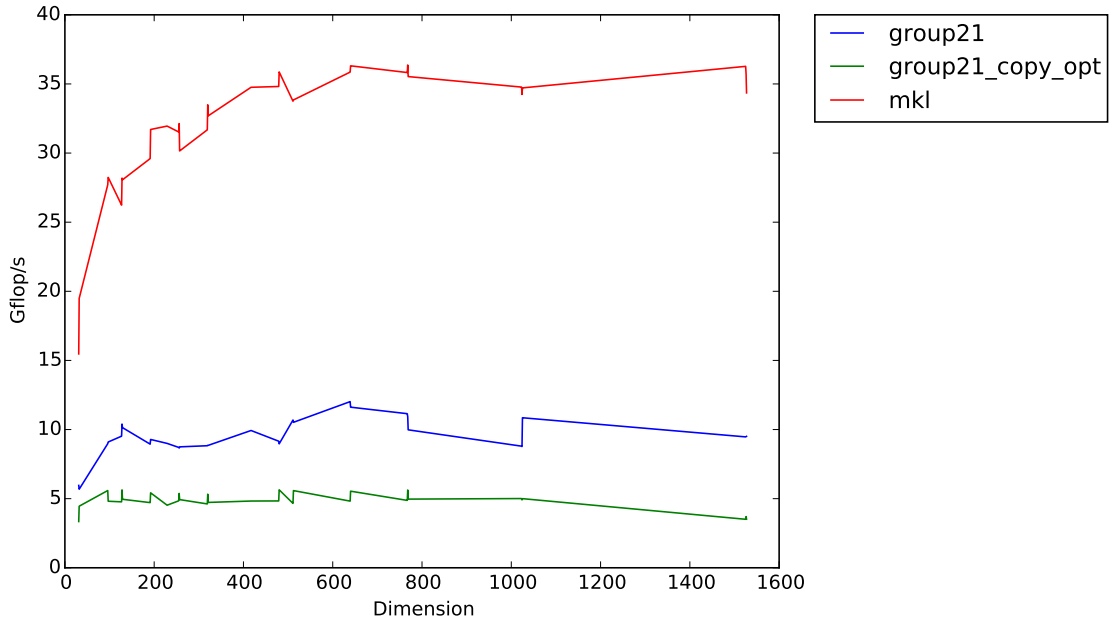

Figure 3: Performance comparison: final implementation vs stage 1 implementation

3 Design Decisions

3.1 Compiler Flags

The final implementation uses the flags, `-O3 -fast -funroll-loops -ipo -xCORE-AVX2 -axCORE-AVX2 -no-prec-div -ansi-alias -restrict -opt-prefetch`.

3.1.1 Optimization Level 3

O3 optimization performs aggressive loop transformations such as Fusion, Block-Unroll-and-Jam, and collapsing IF statements. According to the Intel Developer zone, [7] O3 is recommended for for applications that have loops that heavily use floating-point calculations and process large data sets.

3.1.2 Unrolling loops

The core matrix multiplication kernel performs a dot product with a stride of 1. This behavior is ideal for loop unrolling and is turned on by adding `-funroll-loops`. Results from manually unrolling loops are described in Section 5.2.

3.1.3 Interprocedural Optimization

The compiler flag `-ipo` turns on interprocedural optimization to improve performance of the program by letting the compiler analyze the entire program. [6]

3.1.4 Processor Specification

The processor dispatch option `-axCORE-AVX2` enables the program to choose the most suitable code path for the Intel Xeon E5-2620 v3 processor. The `-xCORE-AVX2` flag enables processor specific code

generation. [9]

3.1.5 Accuracy and Precise Division

Using the `-no-prec-div` flag gives slightly less accurate results than full IEEE division. Compiler converts floating point division operations to multiplication by a reciprocal, improving multiplication performance. We also use the `-fast` flag which may result in slight reduction of accuracy but improvement in performance.

3.1.6 Aliasing

Copy optimization on A in the small matrix case and aligned dynamic memory allocation in the blocked case guarantees that no two pointers in a function point to the same memory location. By using the `-restrict` flag, we can tell the compiler to prevent alias checking during runtime. [8] We also use the `-ansi-alias` flag to tell the compiler that the program adheres to ISO C Standard aliasability rules. Experimentation results are described in 5.7

3.1.7 Prefetching

3.2 Copy Optimization

Consider the equation of an element of C,

$$C_{ij} = \sum_k A_{ik} B_{kj} \tag{2}$$

In the naive implementation, the innermost loop walks over one of the matrices with stride not equal to 1. In a matrix with size 2^N, and using column-major storage, the addresses of the elements in a row are separated by powers of 2. Since the L1 cache is direct-mapped, we run into large number of cache misses.

To prevent this behavior, we can switch the storage of A from column-major to row-major. This also improves multiplication for all matrix sizes since the innermost loop has stride 1. The experimentation results for copy optimization are described in Section 5.5.

3.3 Padded Blocking

In `dgemm_blocked.c` and our initial matrix multiplication attempt, (Section 5.1), we have extra logic to handle cases where the block size may not be square. Removing this additional logic can improve performance of multiplication since there is less branching logic for blocking. In the setup phase, we allocate matrices padded to the block sizes. Since the padding is done with zeros, the result of the final multiplication is unchanged. Refer to the `dgemm_group21.c:padded_transpose` function for the implementation of padding.

Furthermore, the block size was picked to ensure that the blocks of A and B both fit in the L1 cache of the processor. The L1 cache of the Intel Xeon E5-2620 v3 has the ability to hold 6 x 32KB data. Then, to fit both A and B blocks, we have to satisfy, 2 * `BLOCK_SIZE` * `BLOCK_SIZE` * 8B = 192KB. Then block sizes < 100 would fit in the L1 cache. Our experiments in Section 5.1 guided us to pick 32 as the `BLOCK_SIZE`.

3.4 Memory Alignment

To increase the efficient of data loads (in the case of reading elements of A and B) and stores (writing back to C), it is useful to force the compiler to make the dynamically allocated memory blocks aligned on specific byte boundaries. [1] In the blocked approach of the final implementation, we reallocate memory for A, B, and C aligned to 64 byte boundaries using `_mm_malloc`. To tell the compiler about the alignment of the matrices, in the innermost loop of the multiplication of a block, we use the clause `__assume_aligned`. [1]

4 Unsuccessful Optimization Attempts

4.1 Fitting to L2 cache

The L2 cache of the Xeon processor can hold 6 x 256KB data. Then, we should be able to increase the block size by $\sqrt{(8)}$ and notice an improvement in performance. Our experiments showed that the performance decreases when we increase the `BLOCK_SIZE` to 128.

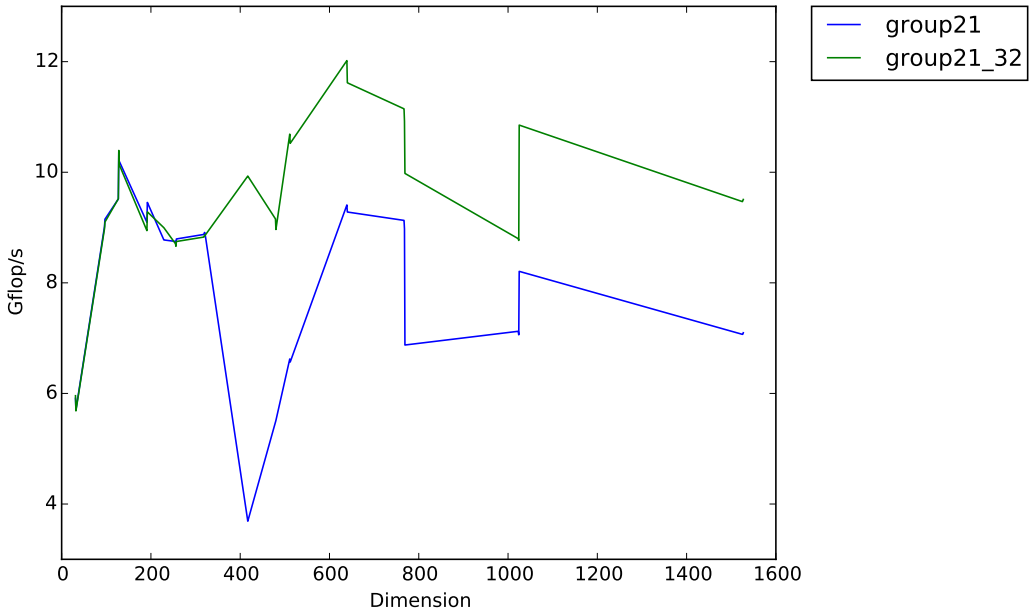

Figure 4: Performance comparison: `BLOCK_SIZE` 128 vs 32

We believe that this performance decrease occurs due to blocks not fitting in the L1 cache.

4.2 Blocking approach in smaller matrices

We adopted a hybrid multiplication approach since on smaller matrices, the blocked approach performs worse than tight loops with copy optimization.

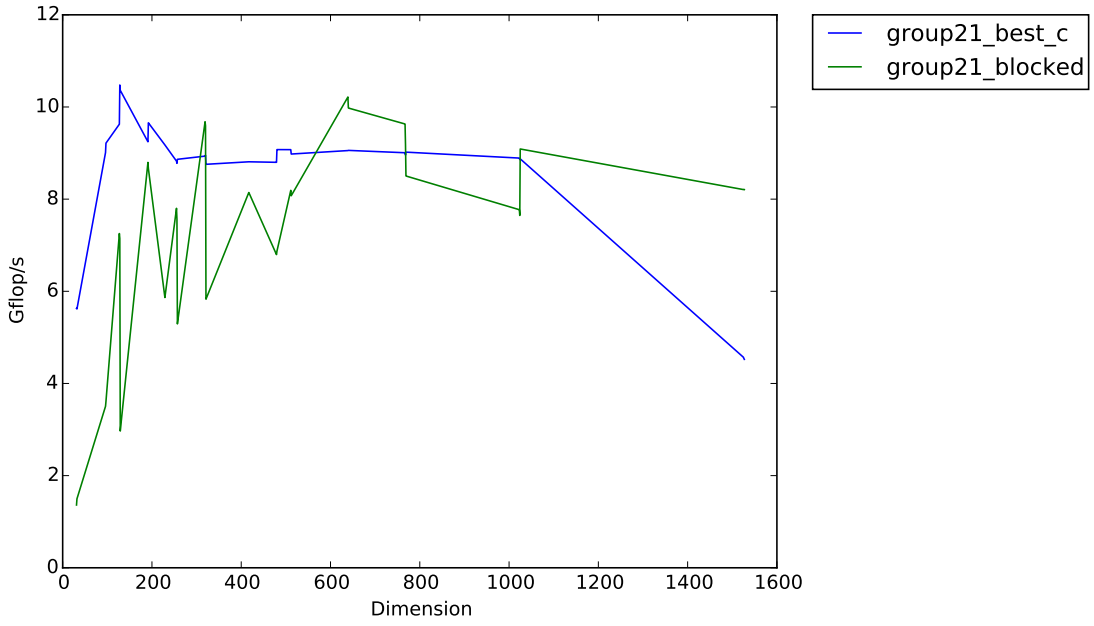

Figure 5: Performance comparison: Tight loops with copy optimization vs blocked approach

We believe that on smaller matrix sizes, the overhead of aligned memory reallocation and blocking reduces performance compared to tight loops with copy optimization. Also, in figure 4.2, we see significant drops in performances in matrix sizes 129, 229, 257 and 321. The matrices with leading dimension < 300 would fit in the L2 cache and the blocking approach should perform as well as the tight loops. But the zig zag pattern we see show, on average, lower performance than the tight loops approach. We were unable to diagnose the cause of the zig-zag pattern in the performance.

5 Optimization Experiments

5.1 Block Multiplication with Multiple Block Sizes

5.1.1 Approach

Working off of `dgemm_blocked.c`, we tried different block sizes to examine the performance changes.

5.1.2 Results

Figure 6 shows the performance of different approaches with various block sizes. (block sizes are a multiple of 2). The performance gain for varying block sizes is not immense but a block size of 64 performs better than other block sizes.

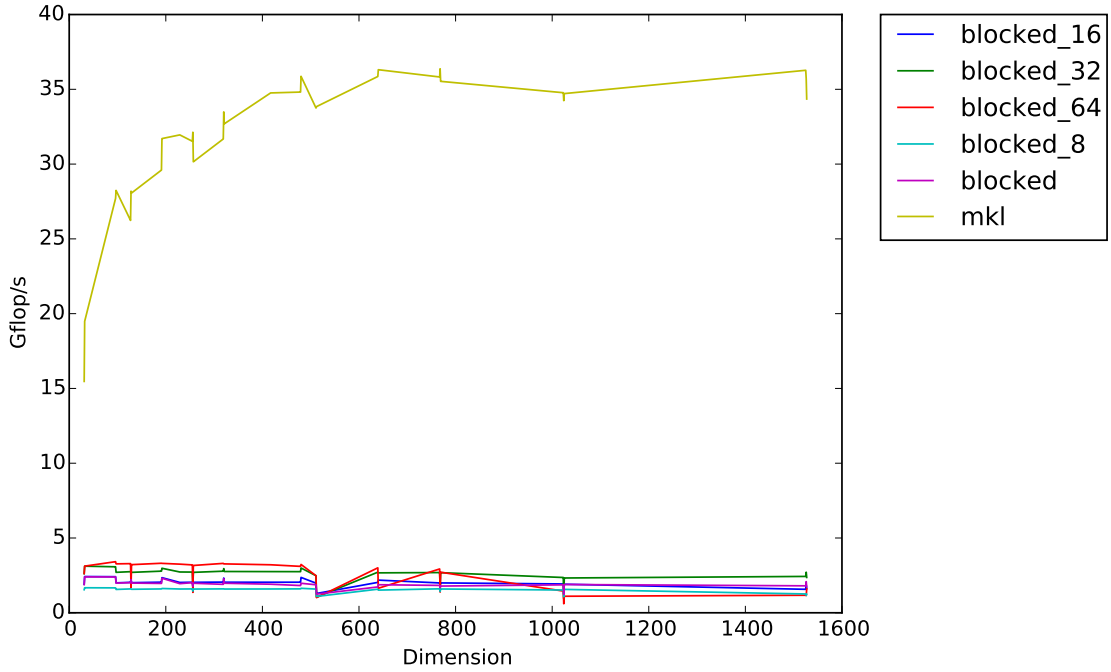

Figure 6: Block size variation

In addition, we attempted block sizes that are not a multiple of 2. Figure 7 is the comparison of the performance against a block size of 64.

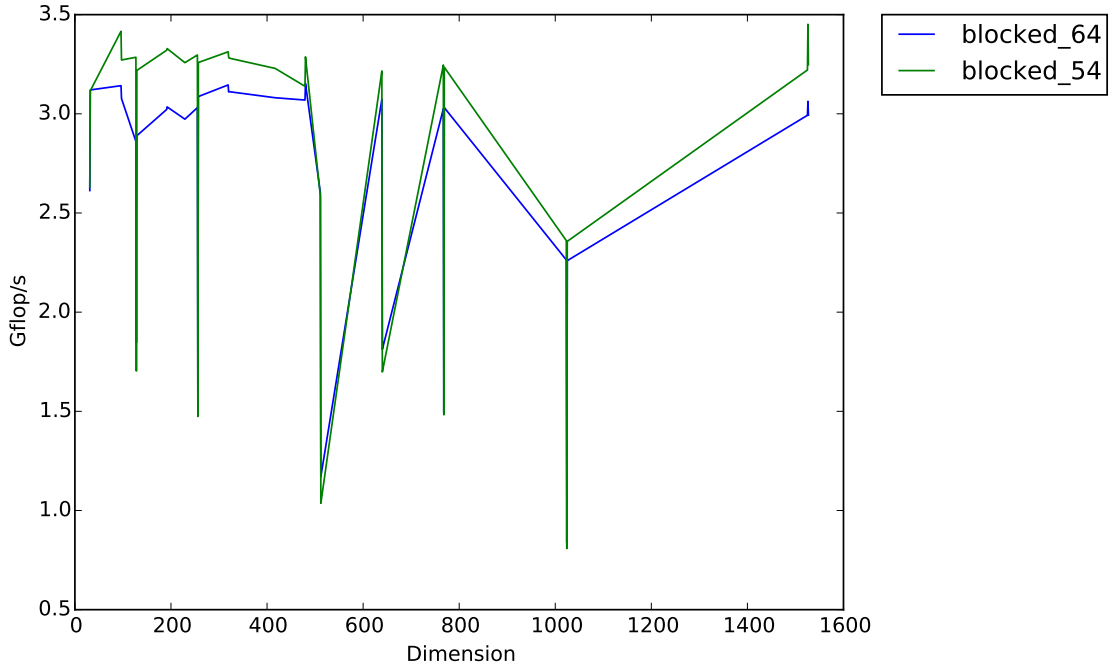

Figure 7: Block size variation

5.2 Block Multiplication with Manual Loop Unrolling

5.2.1 Approach

In this approach, we manually unrolled 4 computations in the inner most loop of the matrix multiplication of a block.

5.2.2 Results

Figure 8 compares the performance of the unrolled blocked version against the vanilla blocked approach. The unrolled versions clearly perform better than the original blocked approach but not by much.

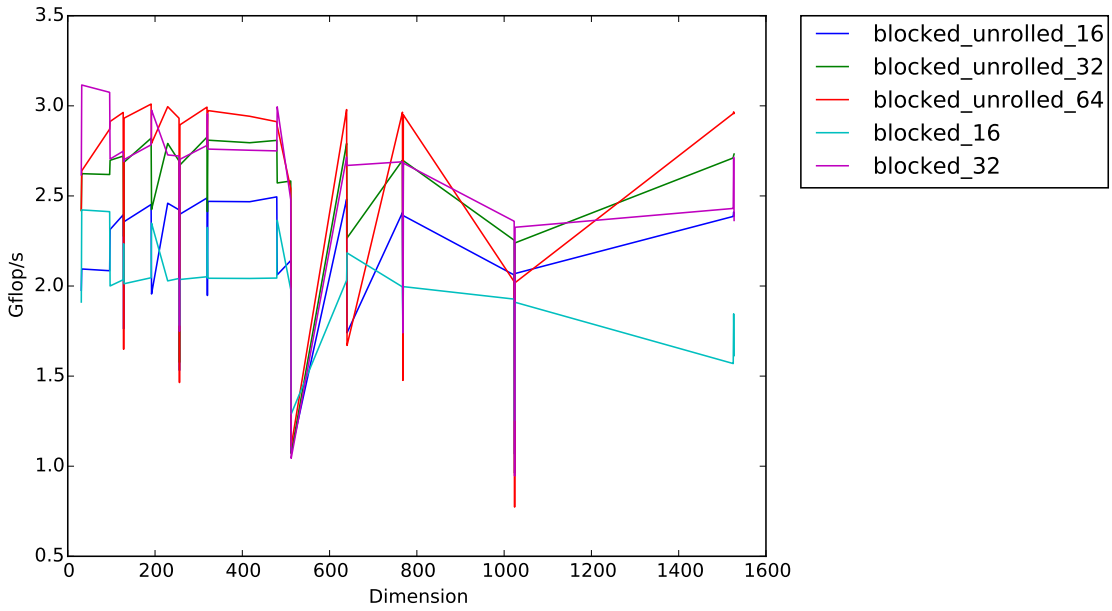

Figure 8: Unrolled blocks vs regular blocks

5.3 Compiler Optimization Flags

5.3.1 Approach

Using the blocked approach as a baseline, we examine the effect of various compiler flags on the multiplication.

5.3.2 Results

Figure 9 shows the performance of blocked multiplication with the flags, `-O3 -march=native -funroll-loops`.

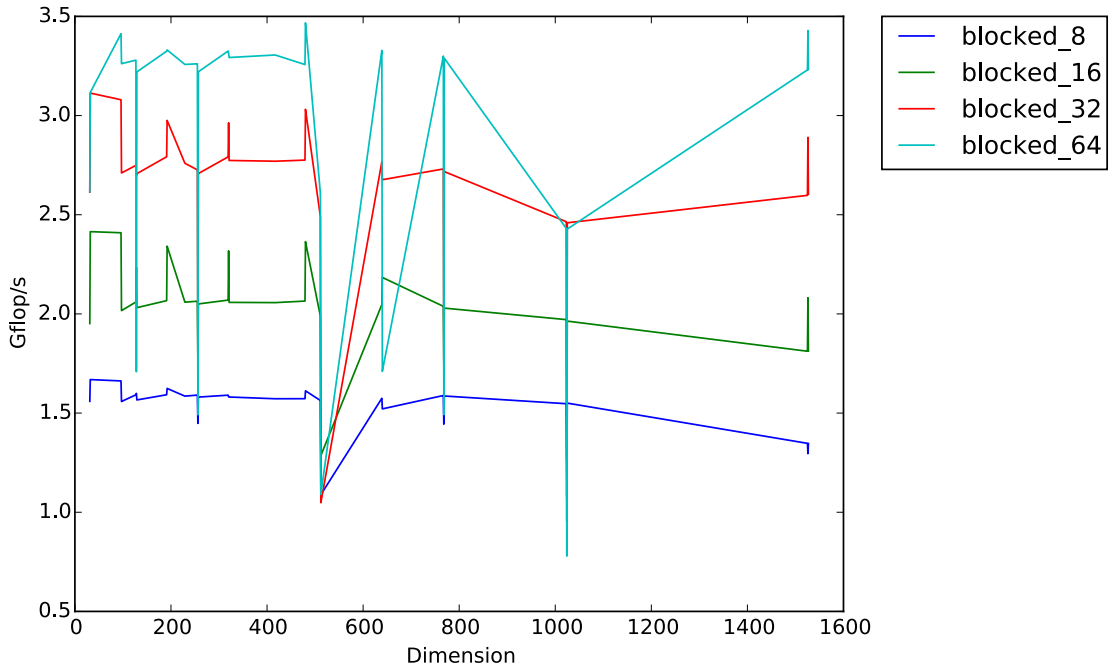

Figure 9: Compiler flags `-O3 -march=native -funroll-loops`

Figure 10 shows the performance of blocked multiplication with the flags, `-O3 -funroll-loops` vs just `-O3`. It appears that merely having the `-O3` flag performs as well as having loops unrolled.

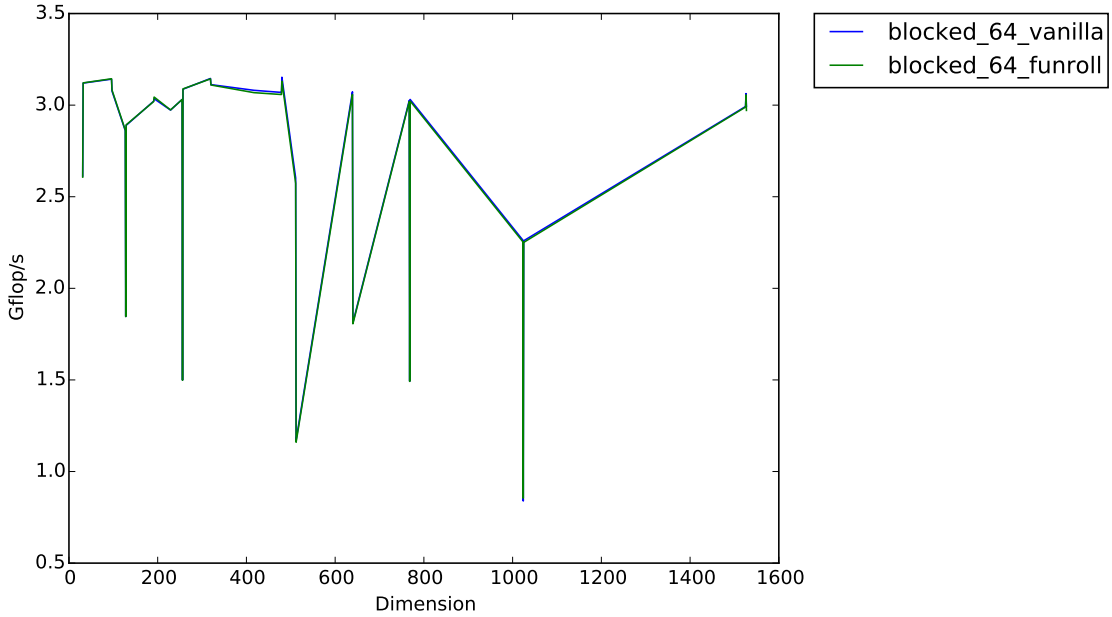

Figure 10: Compiler flags `-O3 -funroll-loops`

5.4 Loop reordering

5.4.1 Approach

The current block multiplication in the innermost loop does not have unit stride with i, j, k loop ordering.

5.4.2 Results

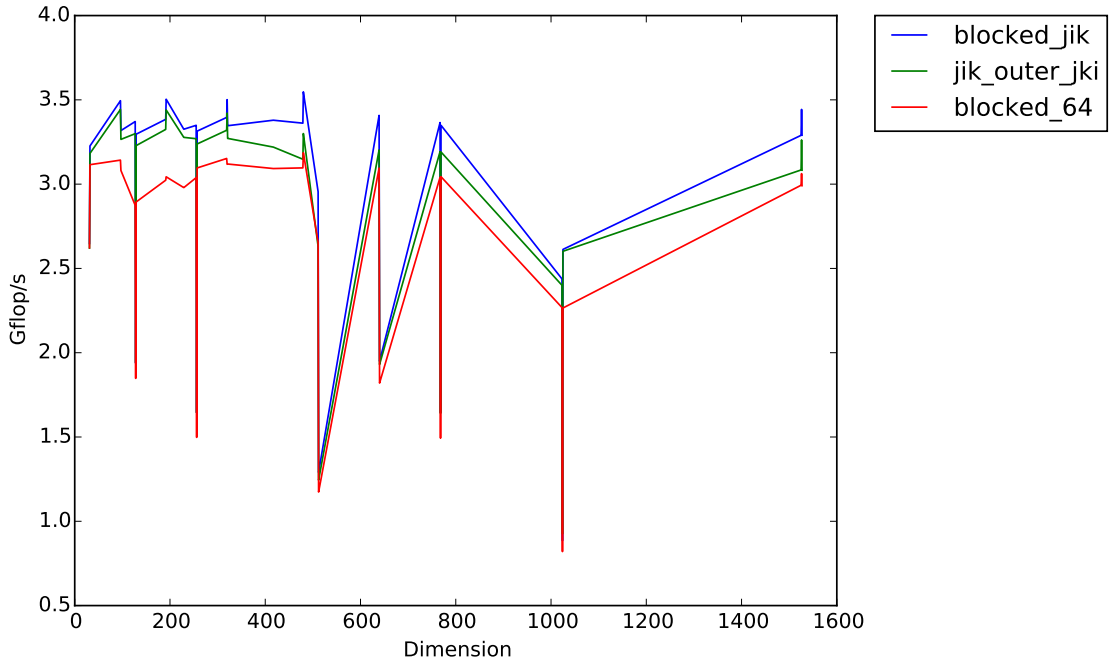

Figure 11: Loop reordering `ijk` vs `jik` vs `jik` and outer loop `jki`

5.5 Copy Optimization

5.5.1 Approach

In the basic version of dgemm, we see drops near matrix sizes that are a multiple of 2. This is caused by conflict misses due to associative caches. To prevent this, we attempted a copy optimization over the basic dgemm implementation.

5.5.2 Results

There is a clear improvement in performance and the conflict misses are converted to gains in performance as seen in Figure 12. Copy optimization is clearly a step in the right direction.

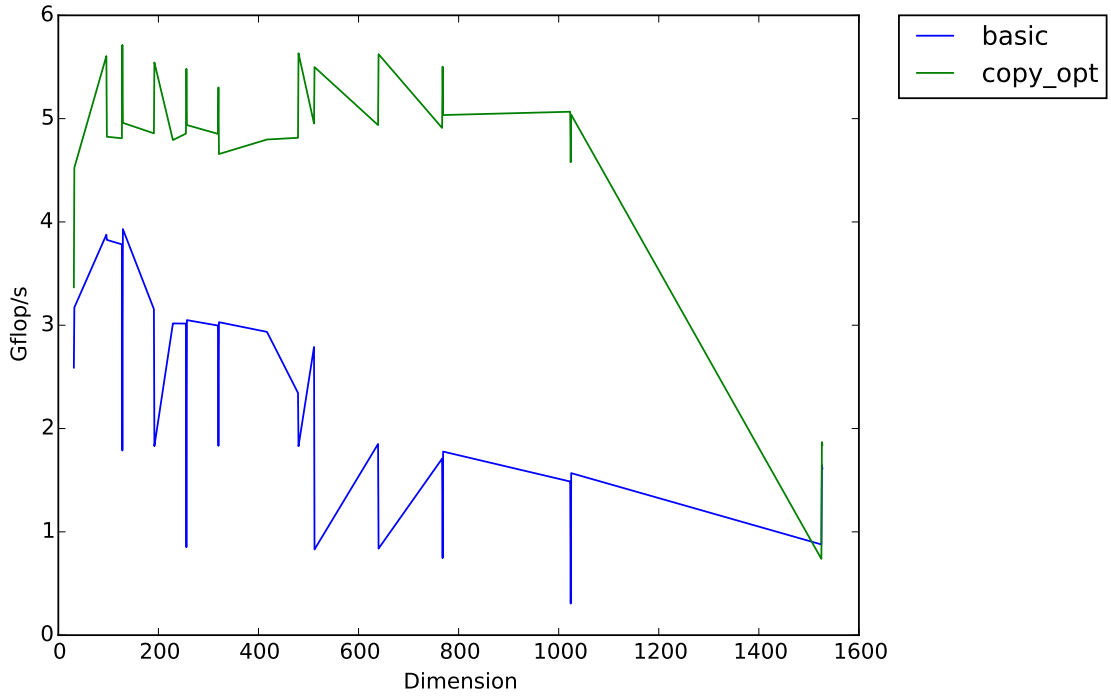

Figure 12: Basic DGEMM vs DGEMM with Copy Optimization

5.6 Compiler flags on Copy Optimization

5.6.1 Approach

We use `-O3` and `-O2` optimization flags when we compile the copy optimization code.

5.6.2 result

`-O2` optimizer is performing better than `-O3` especially when the size of the matrix grows larger.

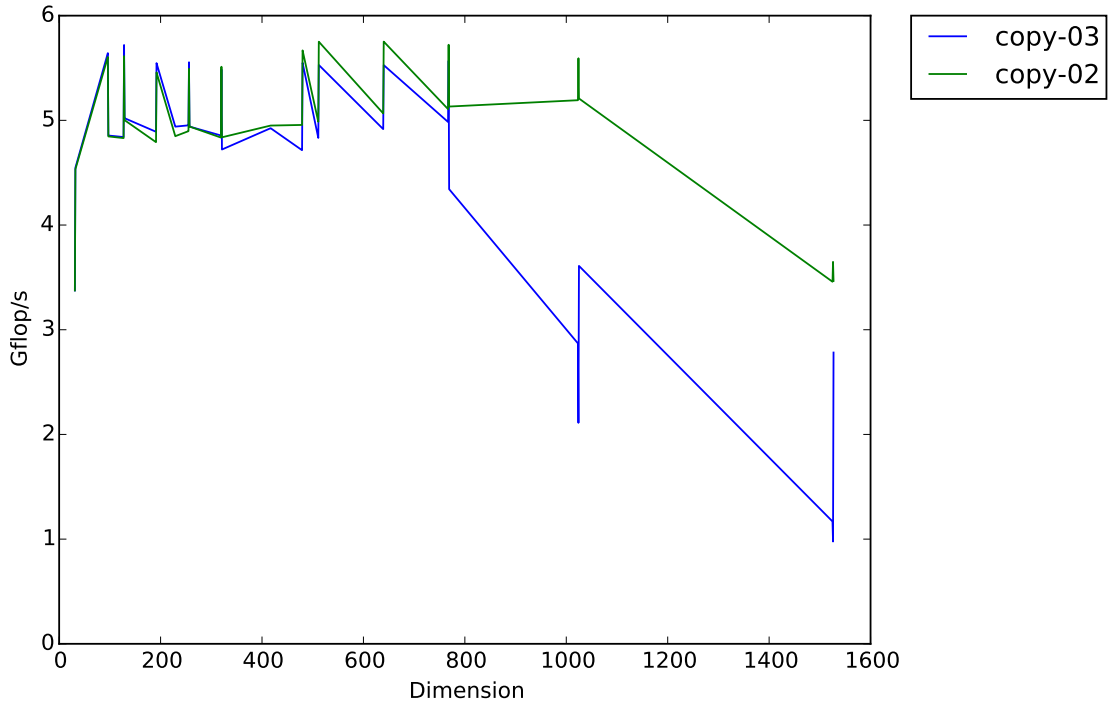

Figure 13: -03 vx -02 on Copy Optimization

5.7 Restrict Keyword

5.7.1 Approach

Telling the compiler that our matrix pointers will not be aliasing is another approach suggested on the writeup.

5.7.2 Results

Figure 14 shows the performance difference between the basic DGEMM implementation and the DGEMM implementation with the `restrict keyword`. `restrict` keyword provides good performance benefits. There is a caveat here since we assumed that the pointer `A` and `B` passed to `square_dgemm` will not alias.

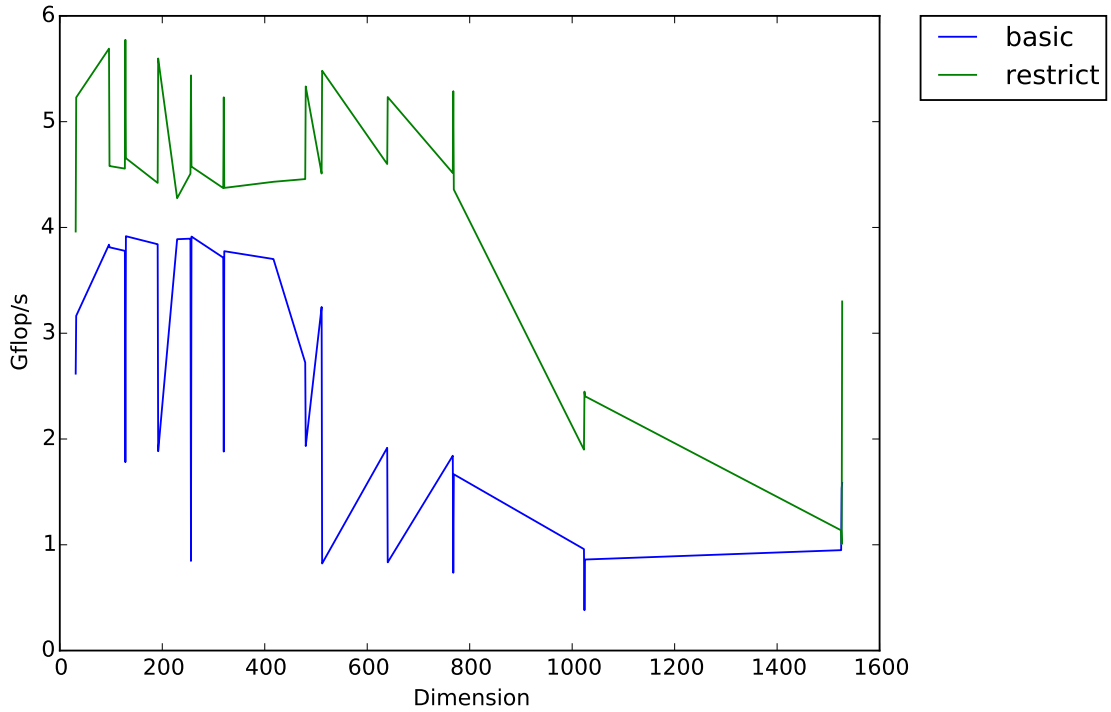

Figure 14: Basic DGEMM vs DGEMM with `restrict` keyword

To eliminate the aliasing assumption, we intended to couple the `restrict` keyword and copy optimization to provide a stronger guarantee of not aliasing. The result as shown in figure 15 shows that the performance decreases. This may be due to incorrect implementation of the restrict keyword in our code.

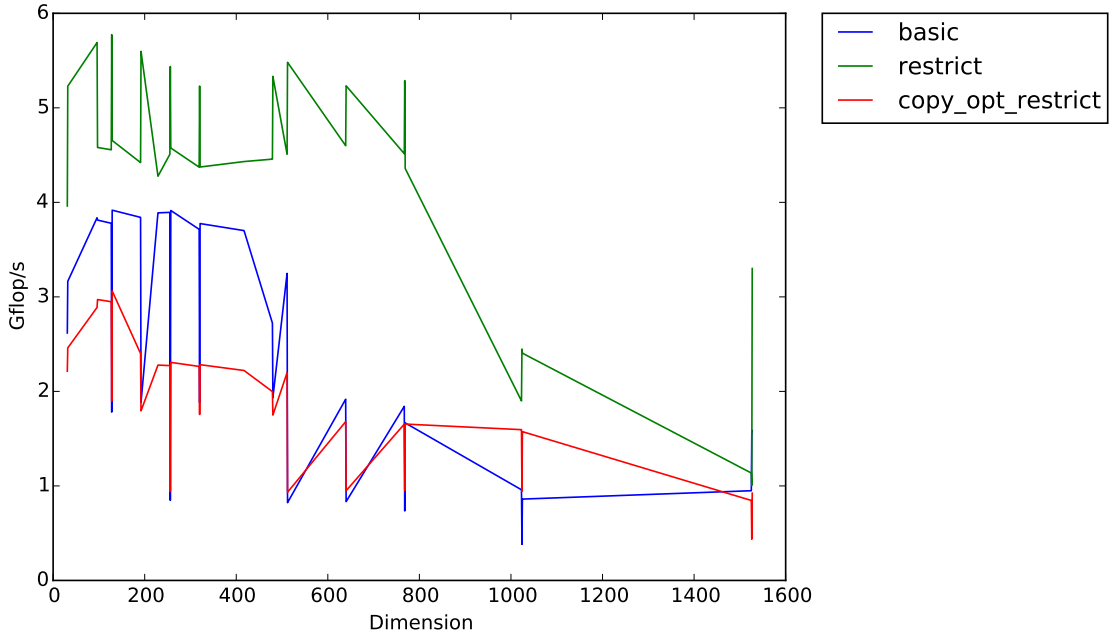

Figure 15: Basic DGEMM vs DGEMM with `restrict` keyword

References

[1] Data Alignment to Assist Vectorization. (n.d.). Retrieved September 30, 2015, from https://software.intel.com/en-us/articles/data-alignment-to-assist-vectorization

[2] Memory Management for Optimal Performance on Intel® Xeon Phi™ Coprocessor: Alignment and Prefetching. (n.d.). Retrieved September 30, 2015, from https://software.intel.com/en-us/articles/memory-management-for-optimal-performance-on-intel-xeon-phi-coprocessor-alignment-and

[3] Manpage of ICC. (n.d.). Retrieved September 30, 2015, from http://scv.bu.edu/computation/bladecenter/manpages/icc.html

[4] Quick-Reference Guide to Optimization with Intel® Compilers. (n.d.). Retrieved September 30, 2015, from https://software.intel.com/sites/default/files/compiler_qrg12.pdf

[5] Getting the Most out of your Intel® Compiler with the New Optimization Reports. (n.d.). Retrieved September 30, 2015, from https://software.intel.com/en-us/articles/getting-the-most-out-of-your-intel-compiler-with-the-new-optimization-reports

[6] Improving Performance with Interprocedural Optimization. (n.d.). Retrieved September 30, 2015, from https://software.intel.com/en-us/node/590470

[7] Step by Step Performance Optimization with Intel® C Compiler. (n.d.). Retrieved September 30, 2015, from https://software.intel.com/en-us/articles/step-by-step-optimizing-with-intel-c-compiler

[8] A Guide to Vectorization with Intel® C Compilers. (n.d.). Retrieved October 1, 2015, from https://d3f8ykwhia686p.cloudfront.net/1live/intel/CompilerAutovectorizationGuide.pdf

[9] Intel® Compiler Options for Intel® SSE and Intel® AVX generation and processor-specific optimizations. (n.d.). Retrieved October 1, 2015, from https://software.intel.com/en-us/articles/performance-tools-for-software-developers-intel-compiler-options-for-sse-gene\\ration-and-processor-specific-optimizations