CS 5220
Project 1 - Matrix Multiplication

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

September 30, 2015

1 Introduction

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

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

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

2 Implementation Overview

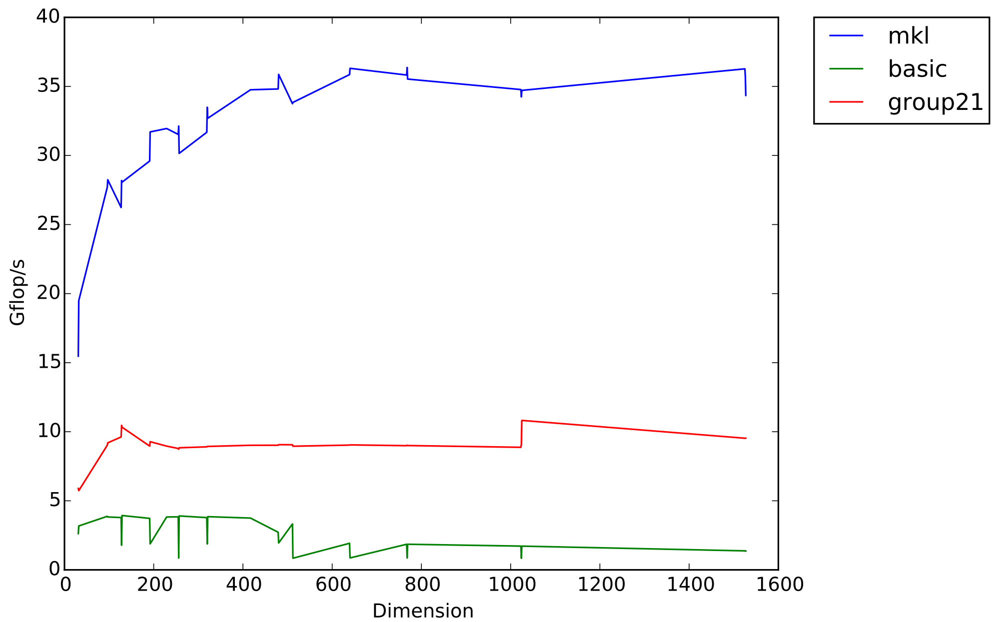

Figure 1: Performance comparison: Intel MKL, our implementation and the naive implementation

Our DGEMM implementation has 2 distinct modes of operation. On square matrices with leading dimension < `BLOCK_THRESHOLD`, our DGEMM uses tight loops with copy optimization. (Section **??**). `BLOCK_THRESHOLD` was picked to be 1000 elements. The L2 cache of the Intel Xeon E5-2620 v3 processor is 15MB and we are able to fit both A and B in the cache.

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 and 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 create an aligned stored area for C. The newly allocation dynamic memory uses zero padding so that all block multiplication is similar. (Section **??**)

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. [**?**] Furthermore, since A is transposed due to copy optimization, the innermost loop of the block multiplication is performed with stride 1.

3 Design Decisions

3.1 Compiler Flags

3.2 Copy Optimization

3.3 Padded Blocking

3.4 Memory Alignment

4 Steps that did not work

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 **??** 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.

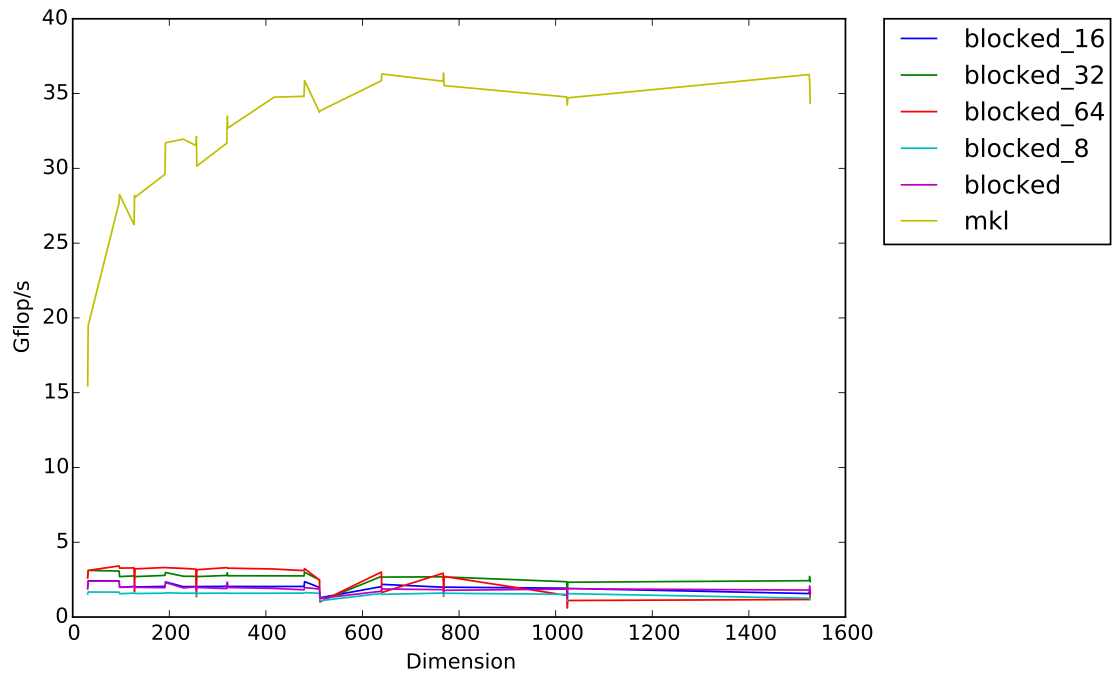

Figure 2: Block size variation

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

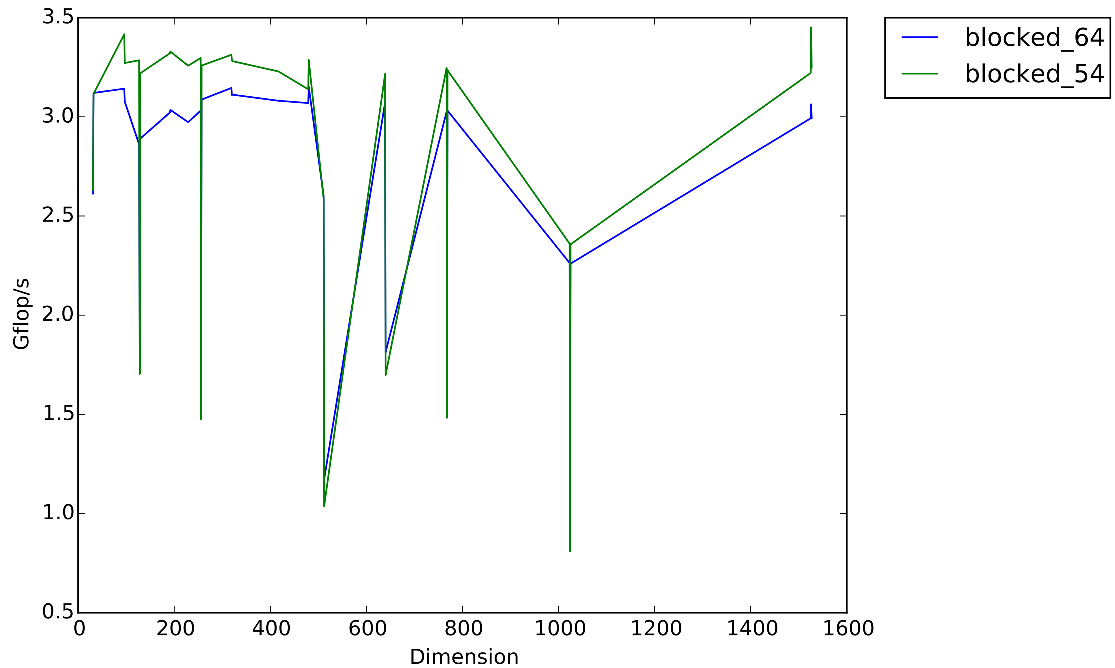

Figure 3: Block size variation

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 **??** 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.

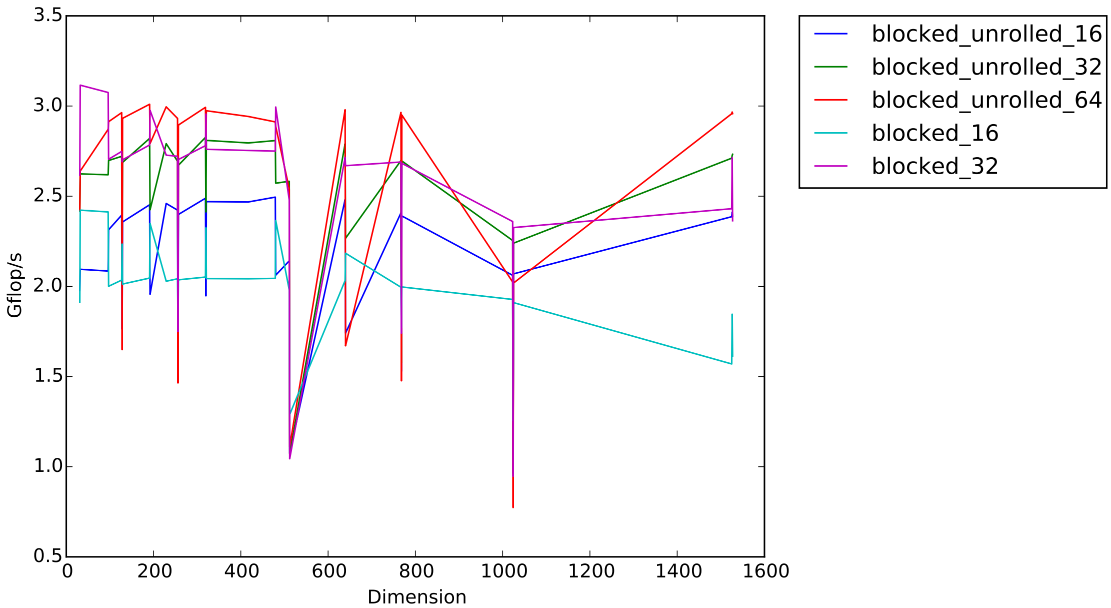

Figure 4: Unrolled blocks vs regular blocks

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

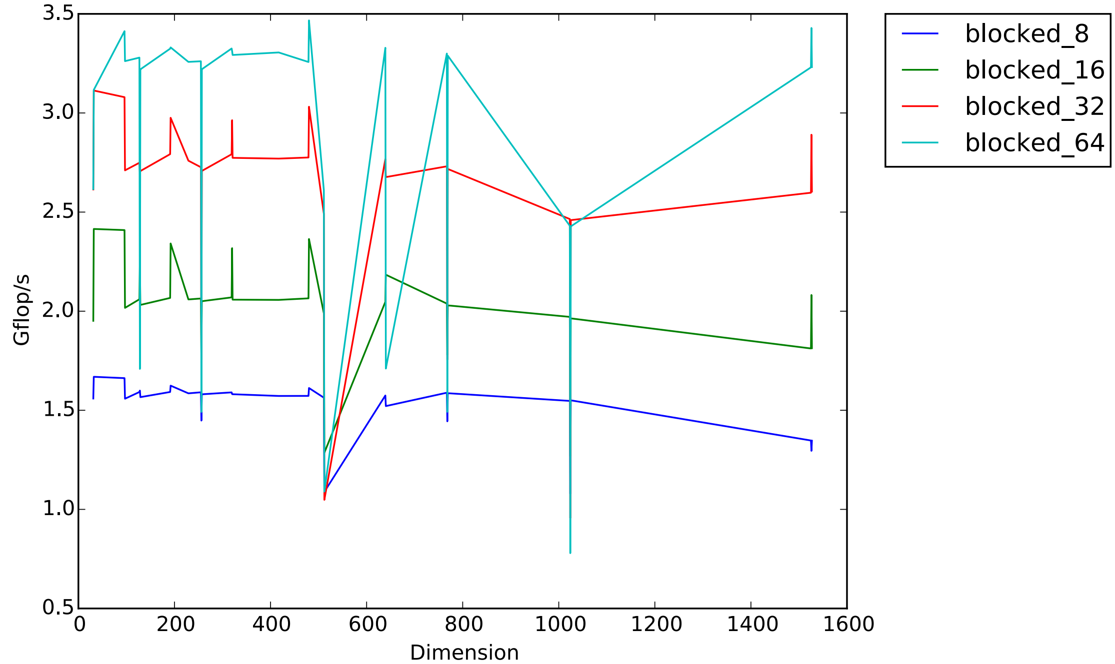

Figure 5: Compiler flags `-O3 -march=native -funroll-loops`

Figure **??** 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.

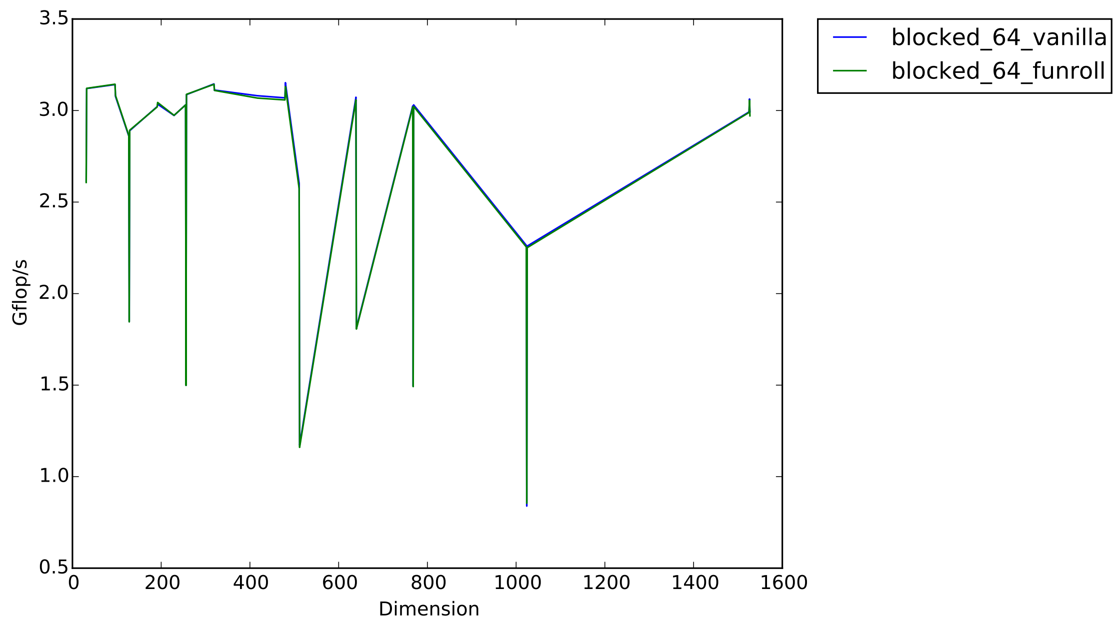

Figure 6: Compiler flags `-O3 -funroll-loops`

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

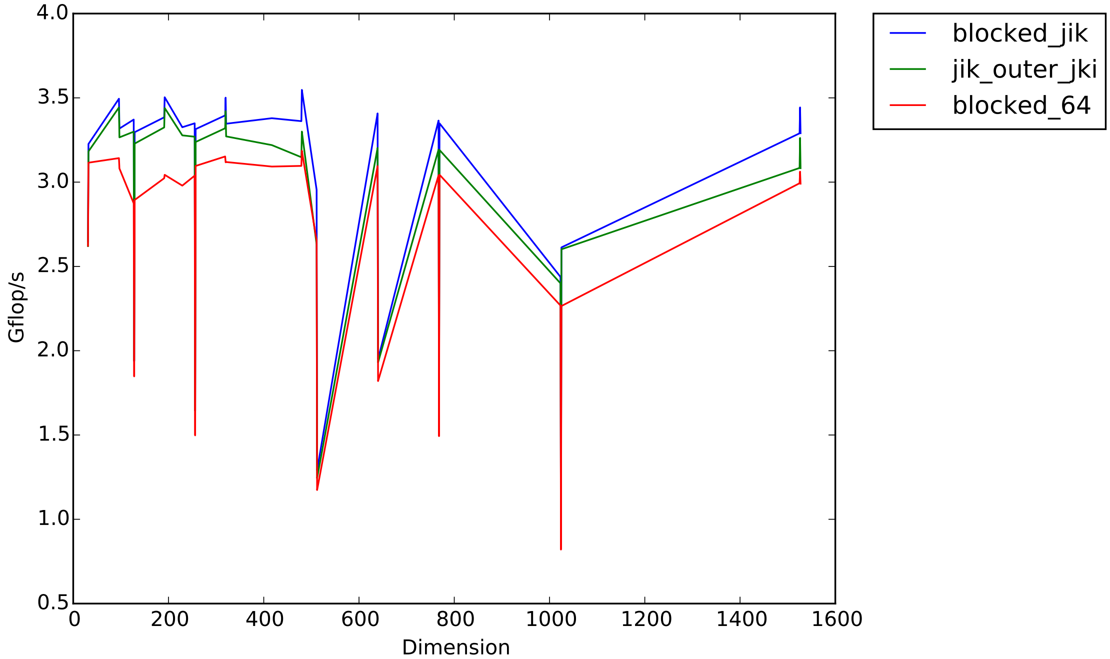

Figure 7: Loop reordering `ijk` vs `jik` vs `jik` and outer loop `jki`

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 **??**. Copy optimization is clearly a step in the right direction.

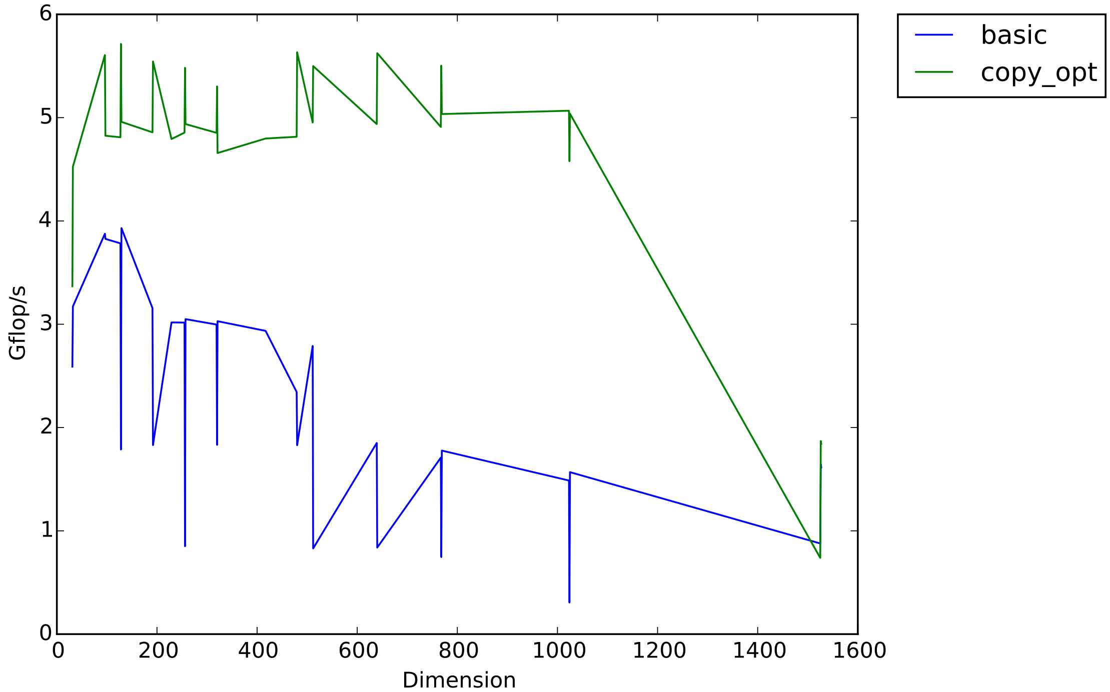

Figure 8: Basic DGEMM vs DGEMM with Copy Optimization

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.

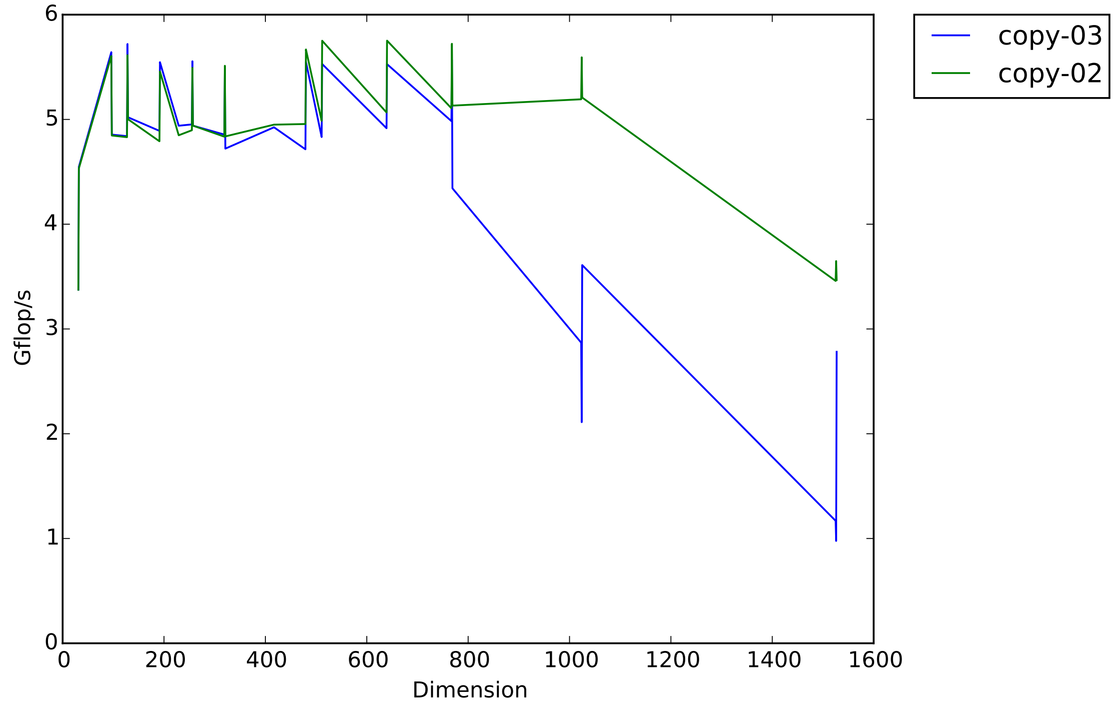

Figure 9: -03 vx -02 on Copy Optimization

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 **??** 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.

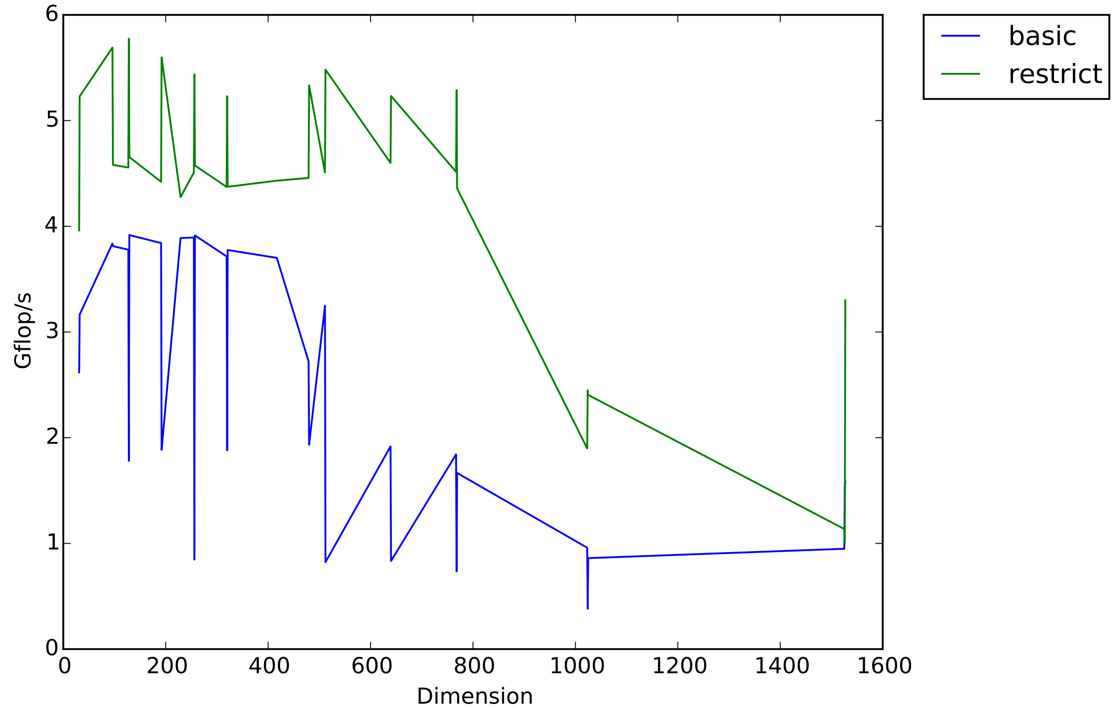

Figure 10: Basic DGEMM vs DGEMM with `restrict` keyword

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 **??** shows that the performance decreases. This may be due to incorrect implementation of the restrict keyword in our code.

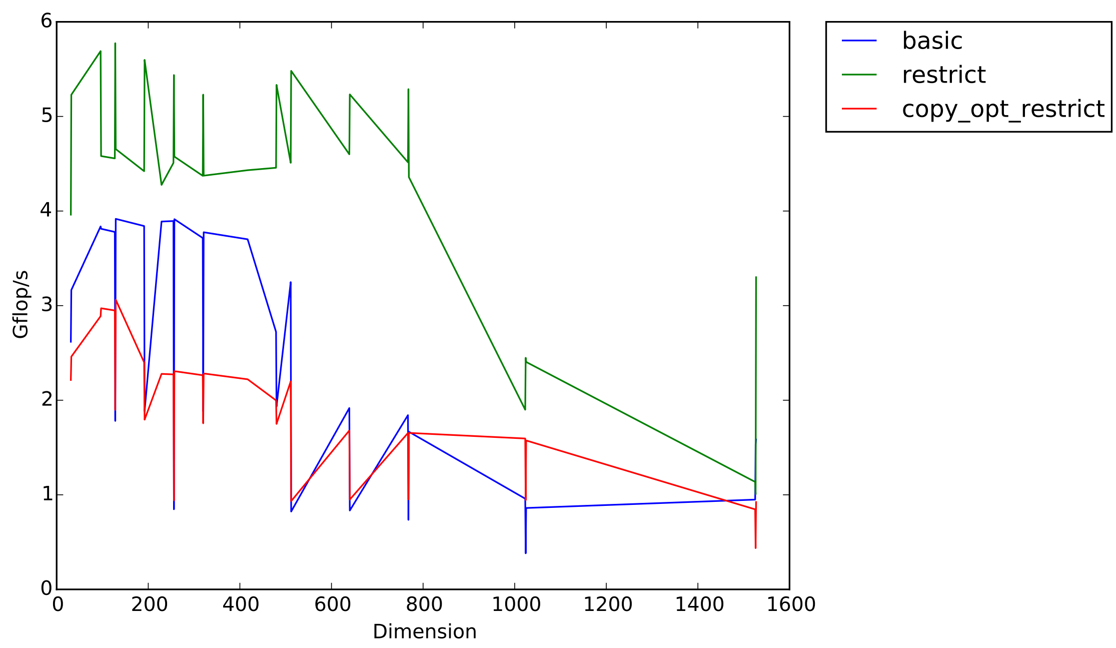

Figure 11: Basic DGEMM vs DGEMM with `restrict` keyword

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