

# CS 5220

## Project 1 - Matrix Multiplication

Sheroze Sherifdeen(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 in scientific and engineering computing applications. DGEMM takes 2 dense square double precision matrices  $A$  and  $B$  stored in column major format and returns,

$$A \times B = C \quad (1)$$

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

### 2 Implementation Overview

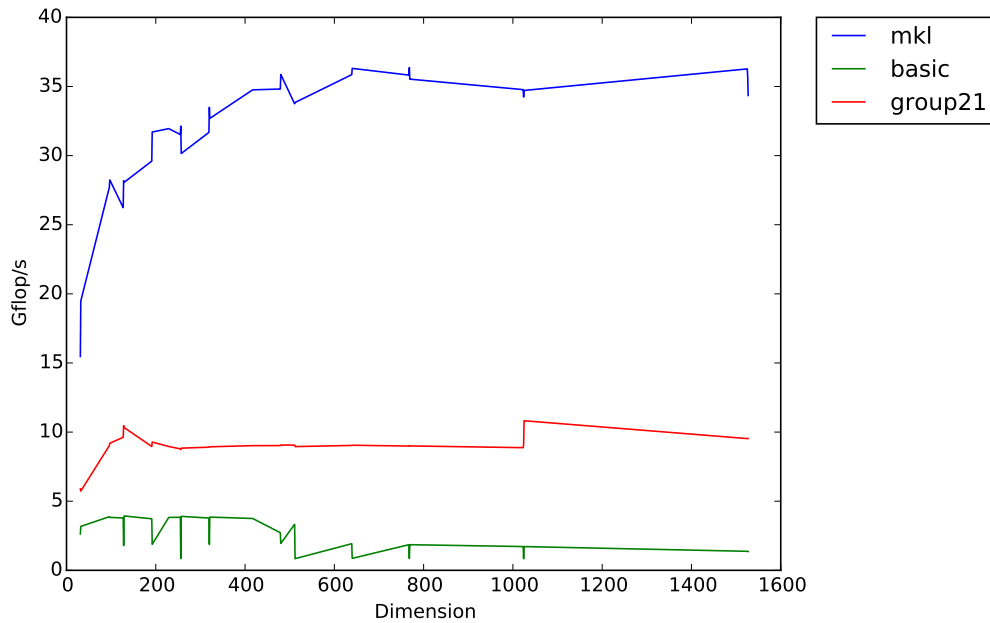


Figure 1: Performance comparison: Intel MKL, our implementation and the naive implementation

Our DGEMM implementation has 2 distinct modes of operation. On square matrices with leading dimension  $< \text{BLOCK\_THRESHOLD}$ , our DGEMM uses tight loops with copy optimization. (Section 3.2). `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  $> \text{BLOCK\_THRESHOLD}$ , we switch to a blocked matrix multiplication. Each block has  $32 \times 32$  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 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.

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.

## 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 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`.

#### 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 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  $C$ ,

$$C_{ij} = \sum_k A_{ik} B_{kj} \quad (2)$$

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 the initial blocking attempt 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. During copy optimization and copying to initially setup multiplication, 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.

## 3.4 Memory Alignment

To increase the efficient of data loads (in the case of reading  $A$  and  $B$  elements) and stores (writing back to  $C$ ), it is useful to force the compiler make the dynamically allocated memory blocks aligned on specific byte boundaries. [1]

## 4 Steps that did not work

### 4.1 Fitting to L2 cache

## 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 2 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.

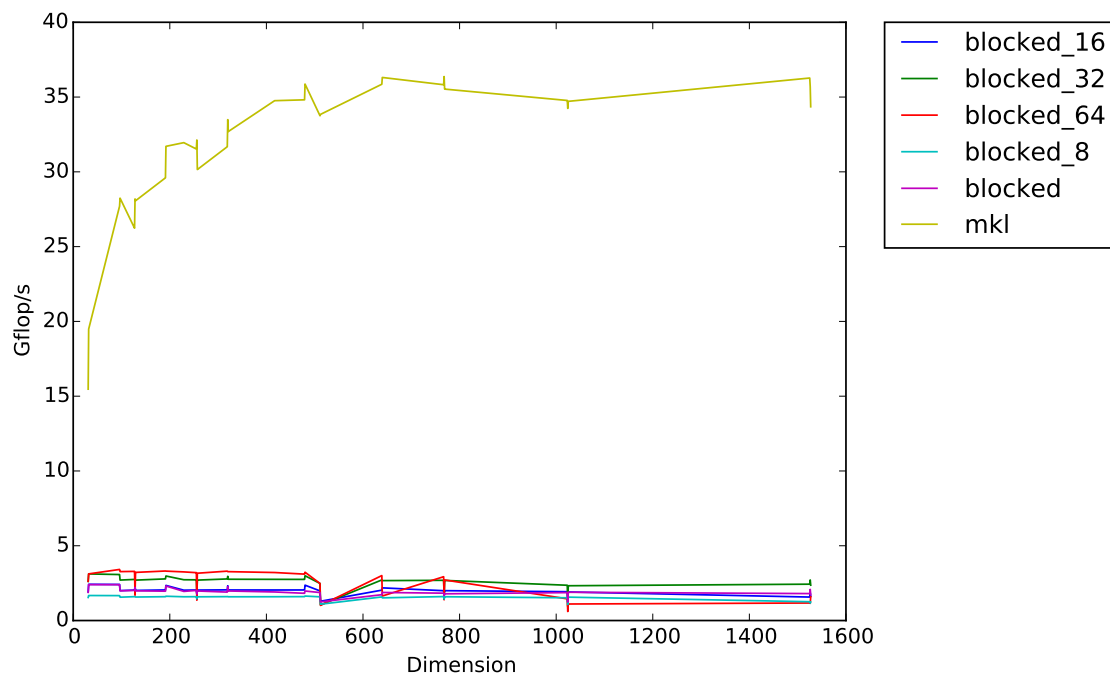


Figure 2: Block size variation

In addition, we attempted block sizes that are not a multiple of 2. Figure 3 is the comparison of the performance against a block size of 64.

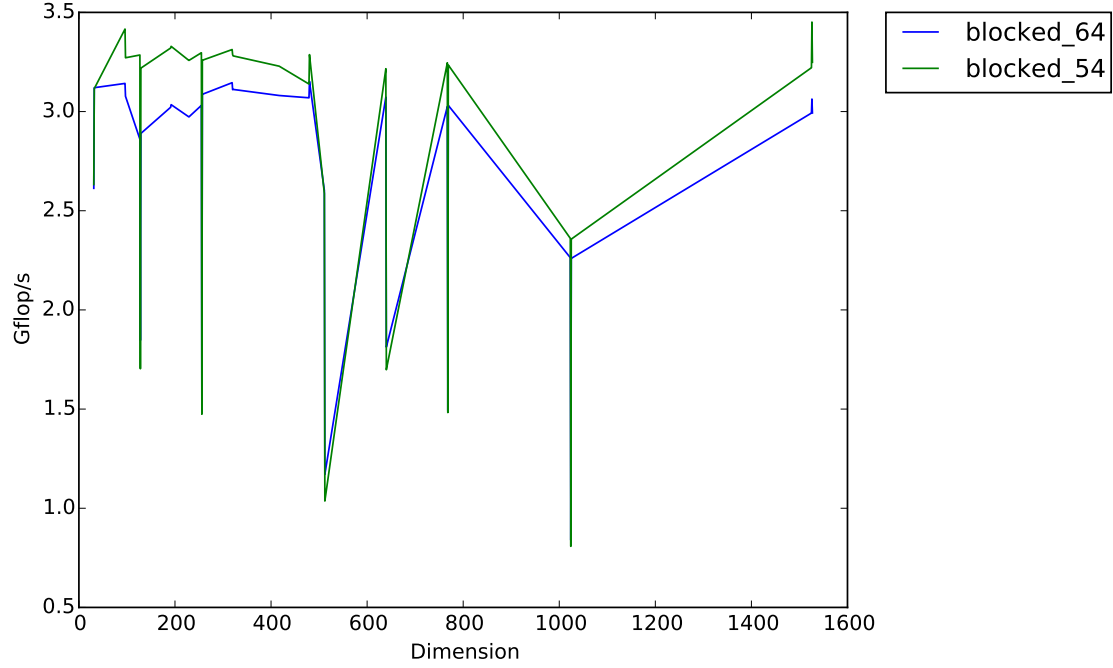


Figure 3: Block size variation

## 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 4 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.

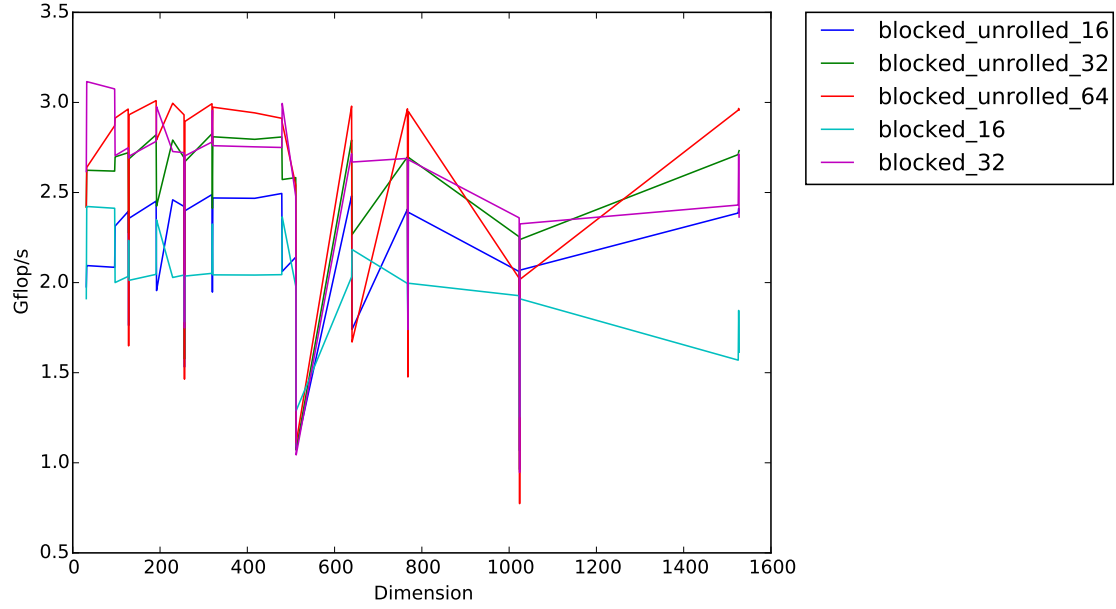


Figure 4: Unrolled blocks vs regular blocks

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

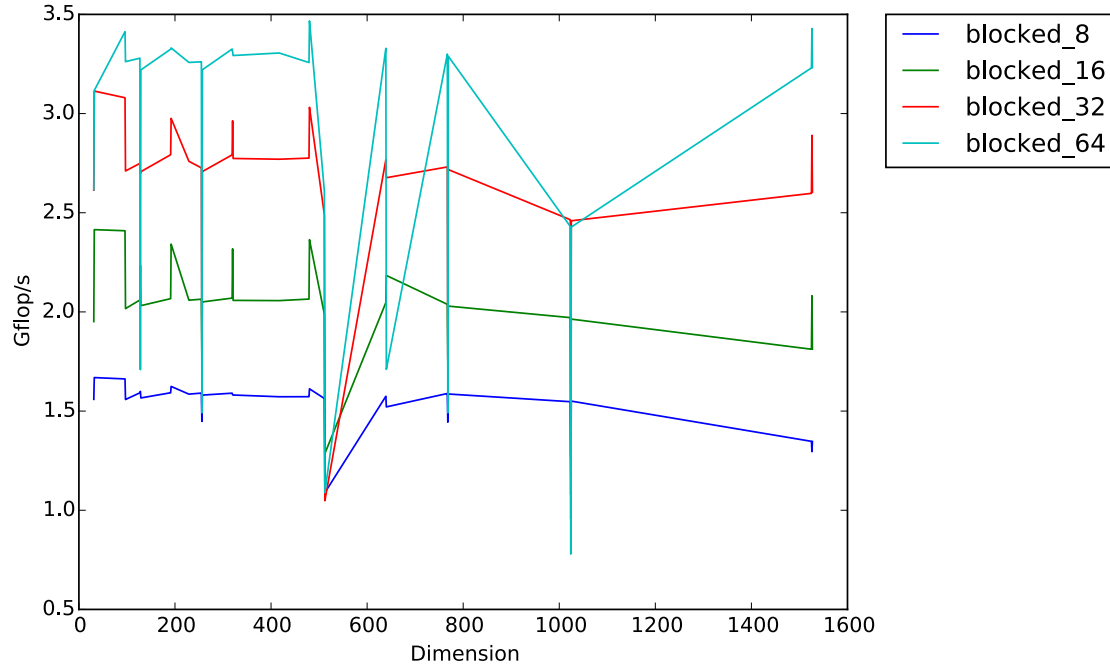


Figure 5: Compiler flags `-O3 -march=native -funroll-loops`

Figure 6 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.

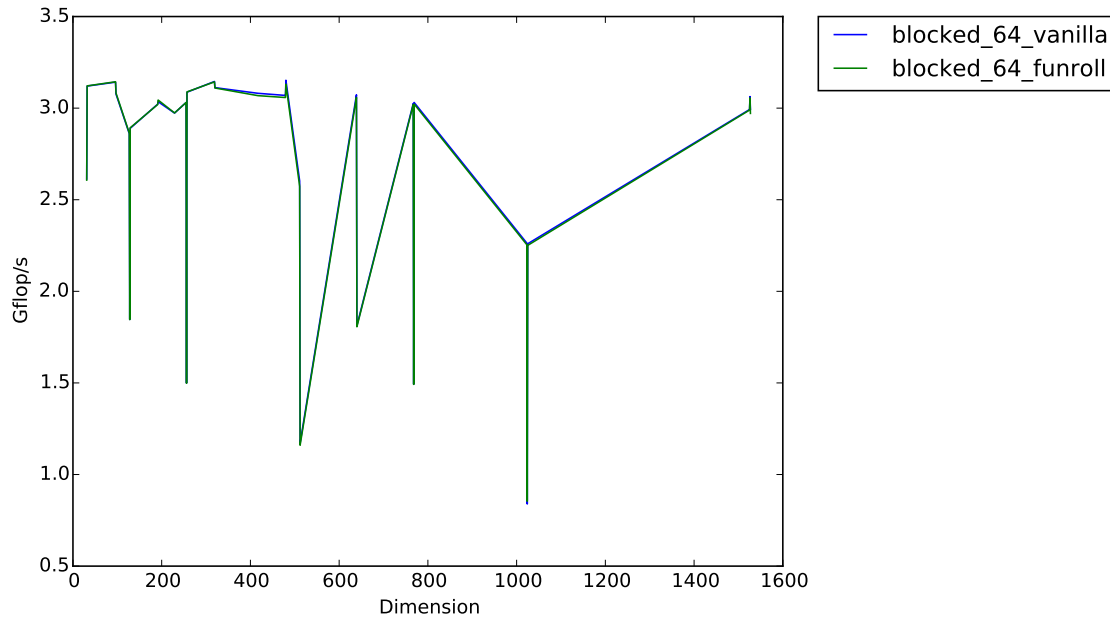


Figure 6: Compiler flags `-O3 -funroll-loops`



## 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

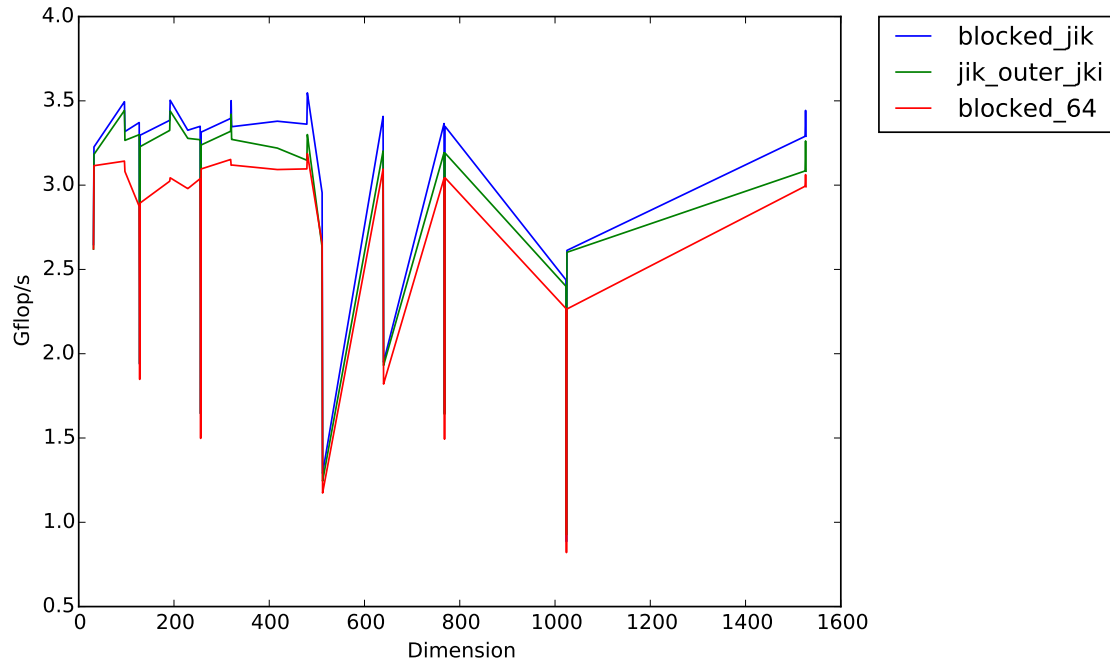


Figure 7: Loop reordering ijk vs jik vs jik and outer loop jki

## 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 8. Copy optimization is clearly a step in the right direction.

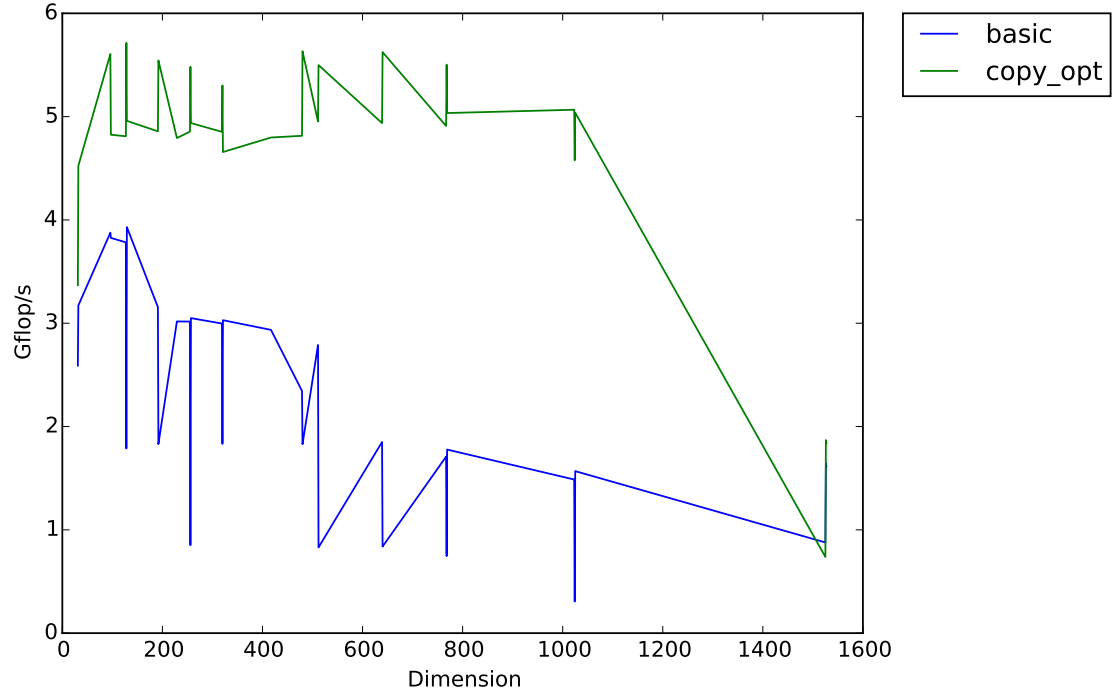


Figure 8: Basic DGEMM vs DGEMM with Copy Optimization

## 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.

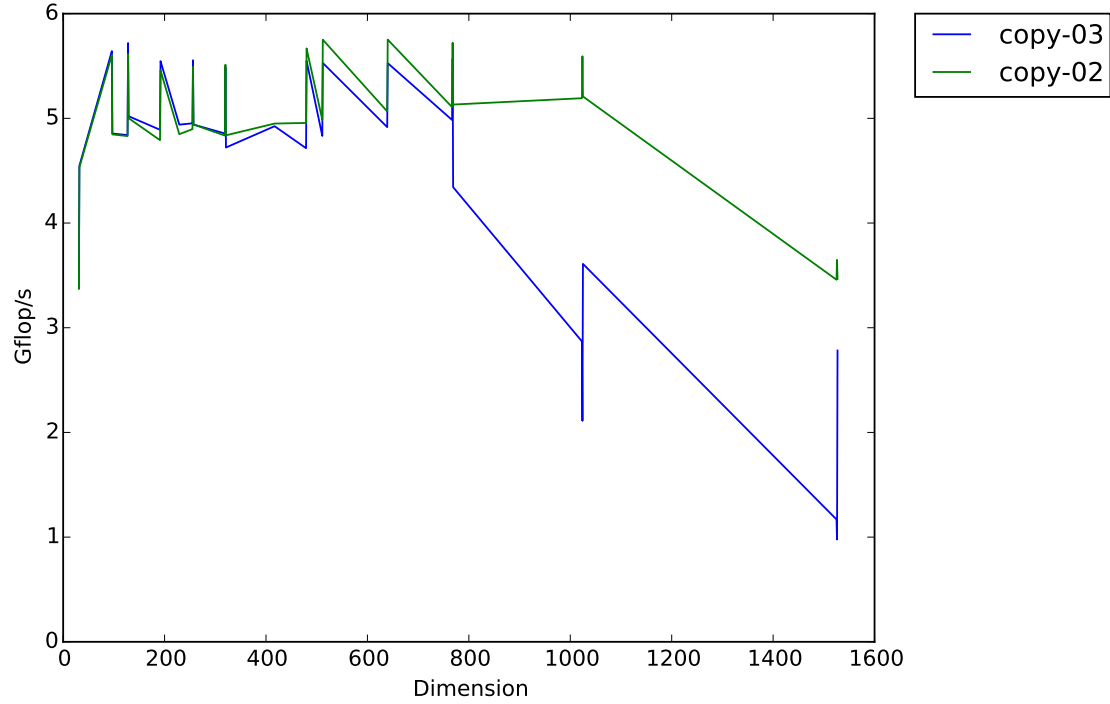


Figure 9: -03 vx -02 on Copy Optimization

## 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 10 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.

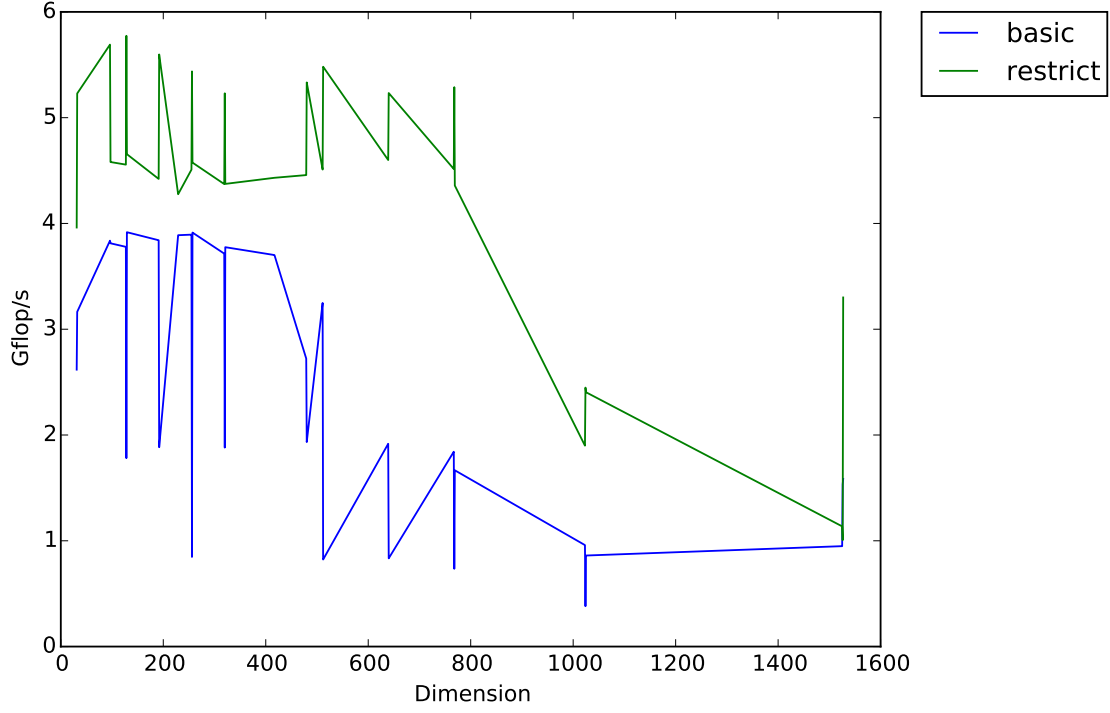


Figure 10: Basic DGEMM vs DGEMM with `restrict` keyword

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

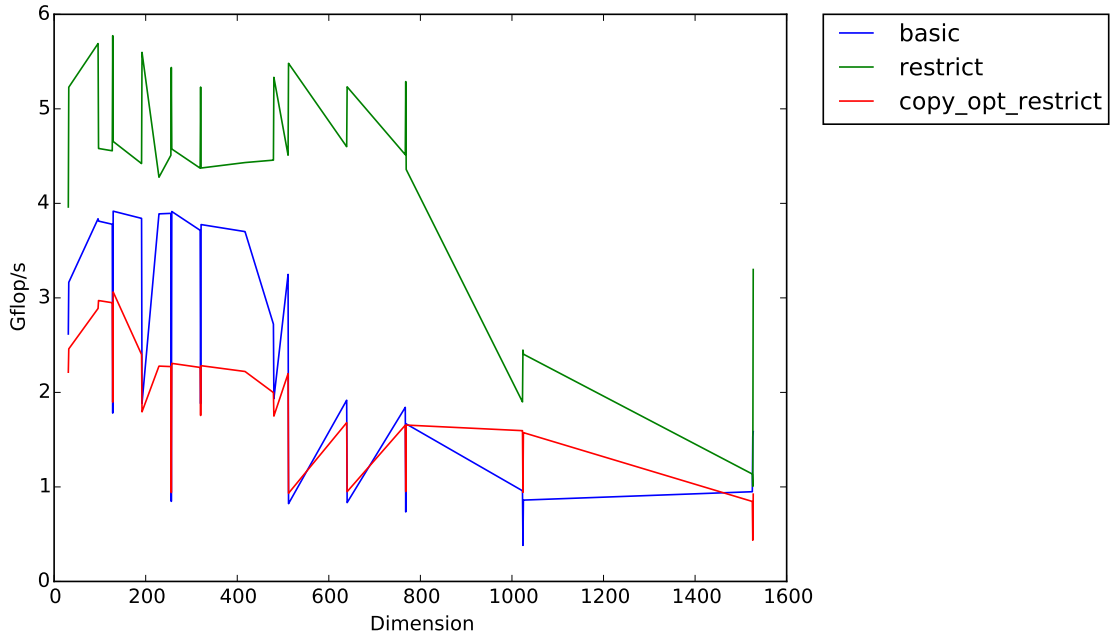


Figure 11: Basic DGEMM vs DGEMM with `restrict` keyword

## 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](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-generation-and-processor-specific-optimizations>