CS 5220
Project 1 - Matrix Multiplication

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

September 17, 2015

1 Overview

In this project, we tried several methods to fine-tune square matrix multiplication. Based on the `dgemm_blocked.c`, we tried unrolling inner loop multiplication, modifying loop sequence to take advantage of SSE, experimenting with different optimization flags, multiple blocks sizes and copy optimizations.

The next section reflects our various optimization attempts and their results.

2 Optimization Attempts

2.1 Block Multiplication with Multiple Block Sizes

2.1.1 Approach

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

2.1.2 Results

Figure 1 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.

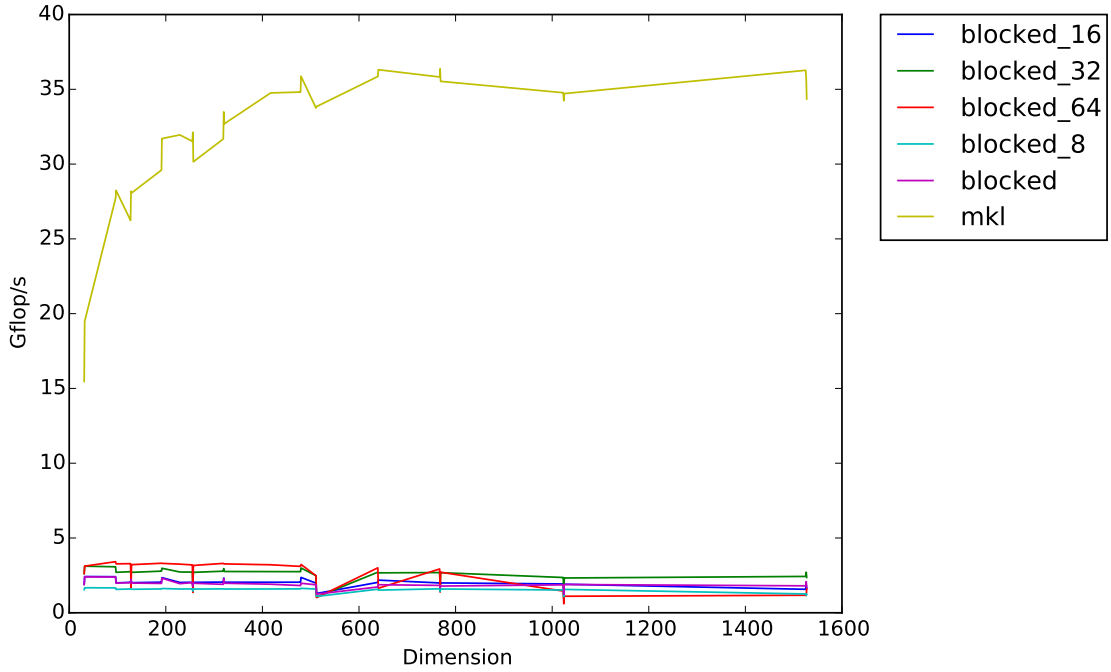

Figure 1: Block size variation

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

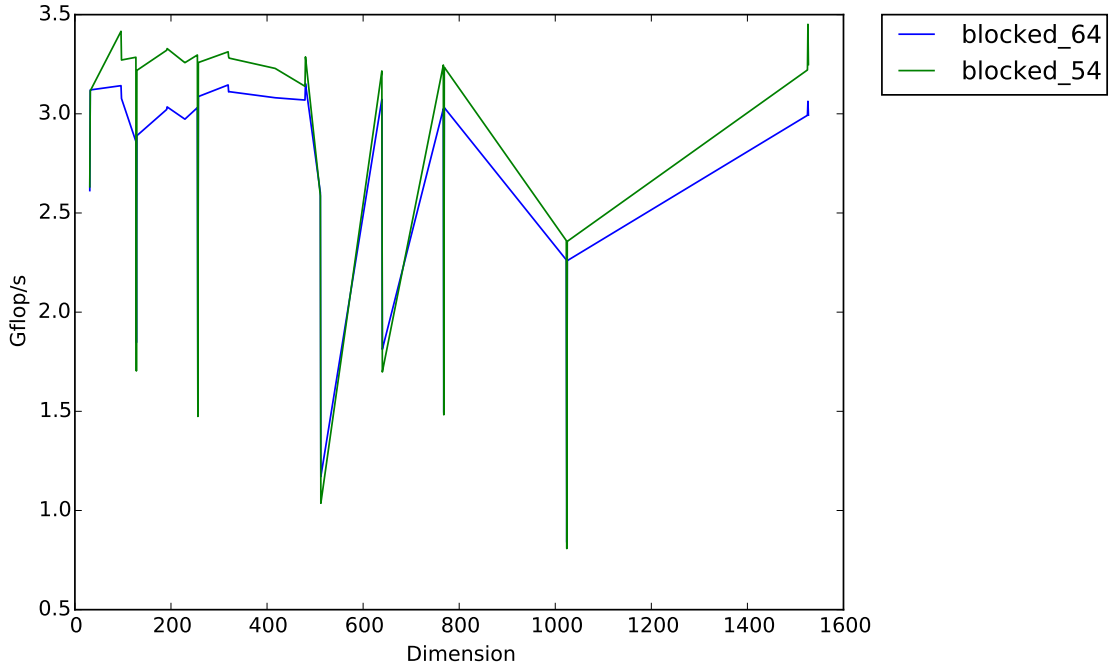

Figure 2: Block size variation

2.2 Block Multiplication with Manual Loop Unrolling

2.2.1 Approach

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

2.2.2 Results

Figure 3 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.

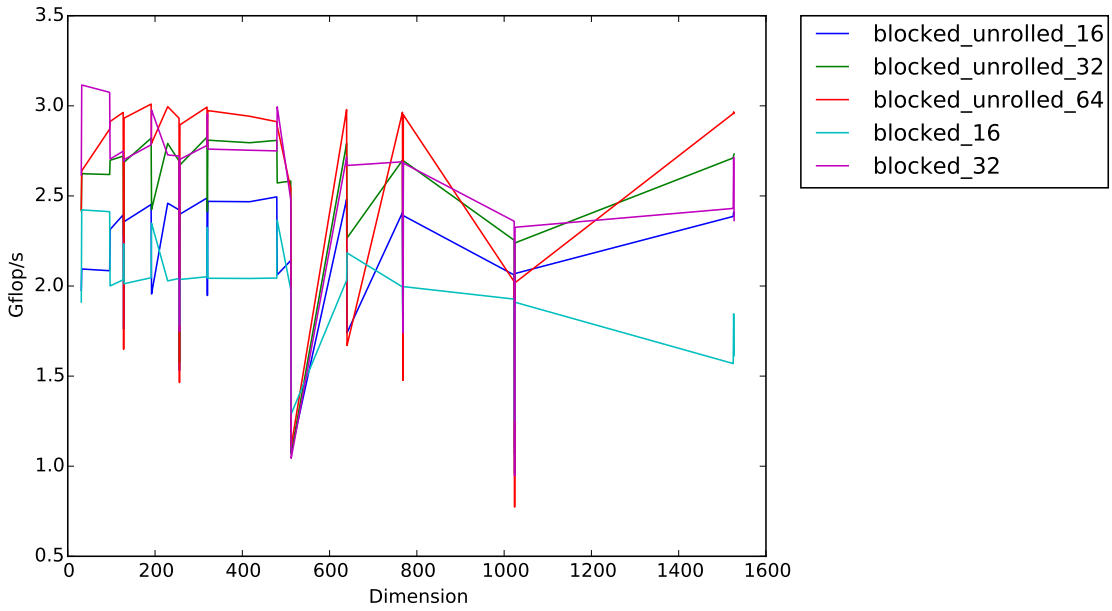

Figure 3: Unrolled blocks vs regular blocks

2.3 Compiler Optimization Flags

2.3.1 Approach

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

2.3.2 Results

Figure 4 shows the performance of blocked multiplication with the flags, `-O3 -march=native -funroll-loops`.

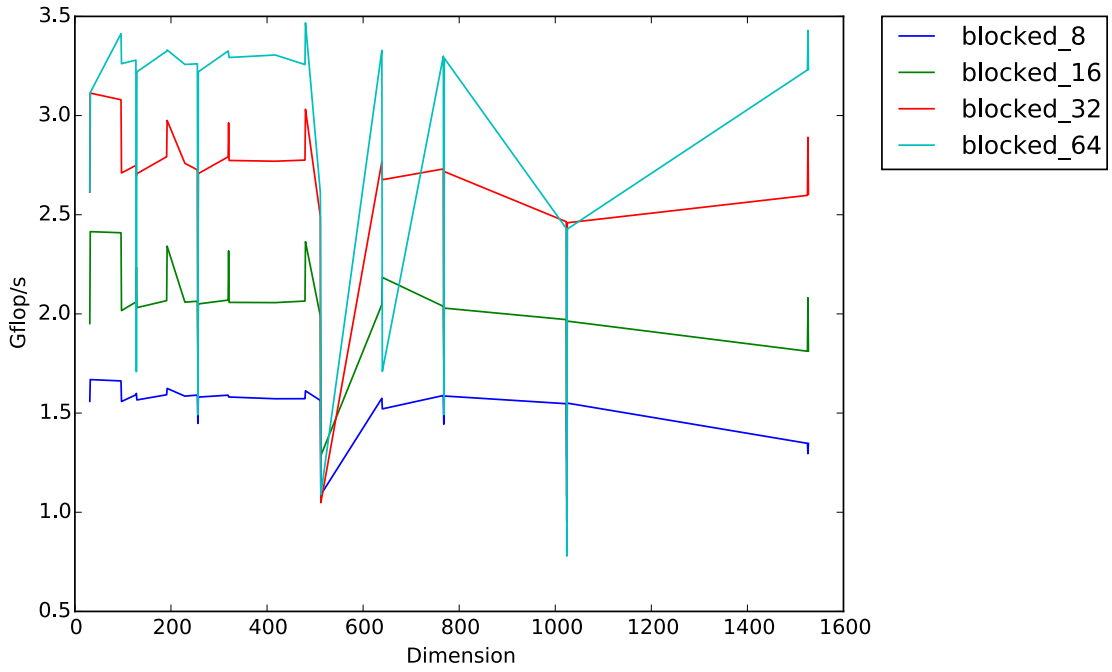

Figure 4: Compiler flags `-O3 -march=native -funroll-loops`

Figure 5 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.

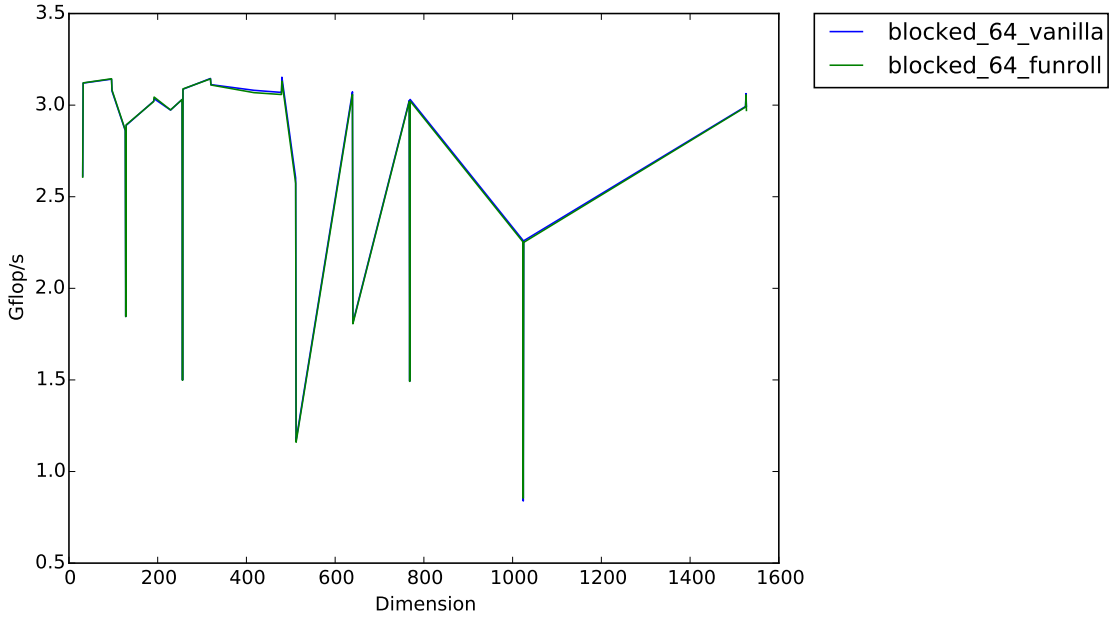

Figure 5: Compiler flags `-O3 -funroll-loops`

2.4 Loop reordering

2.4.1 Approach

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

2.4.2 Results

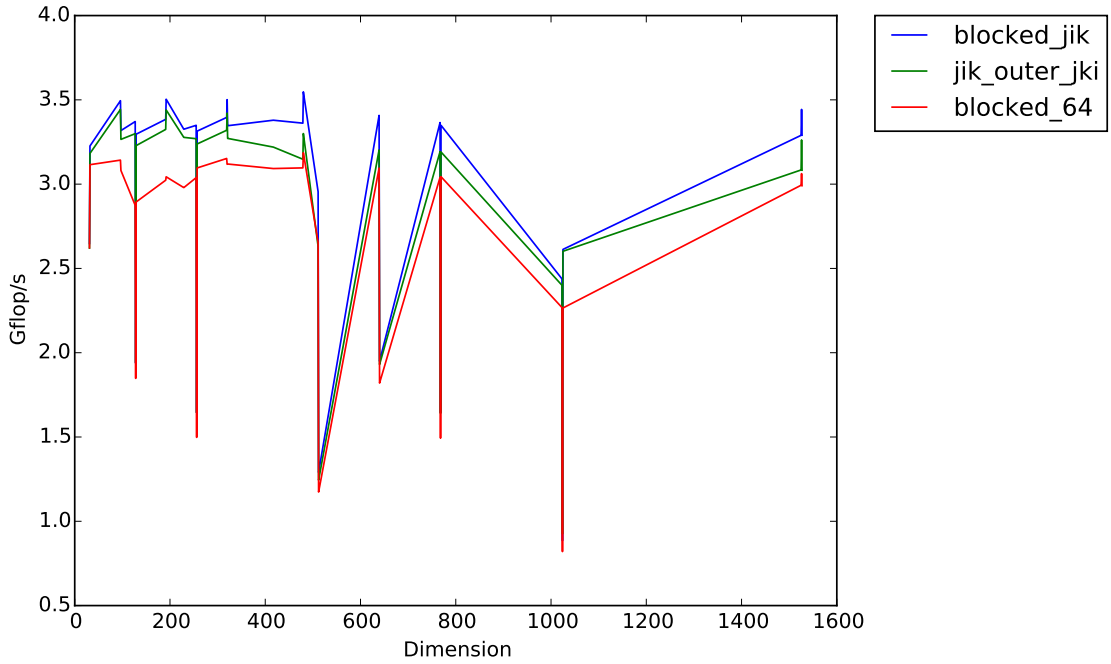

Figure 6: Loop reordering `ijk` vs `jik` vs `jik` and outer loop `jki`

2.5 Copy Optimization

2.5.1 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.

2.5.2 Results

There is a clear improvement in performance and the conflict misses are converted to gains in performance as seen in Figure 7. Copy optimization is clearly a step in the right direction.

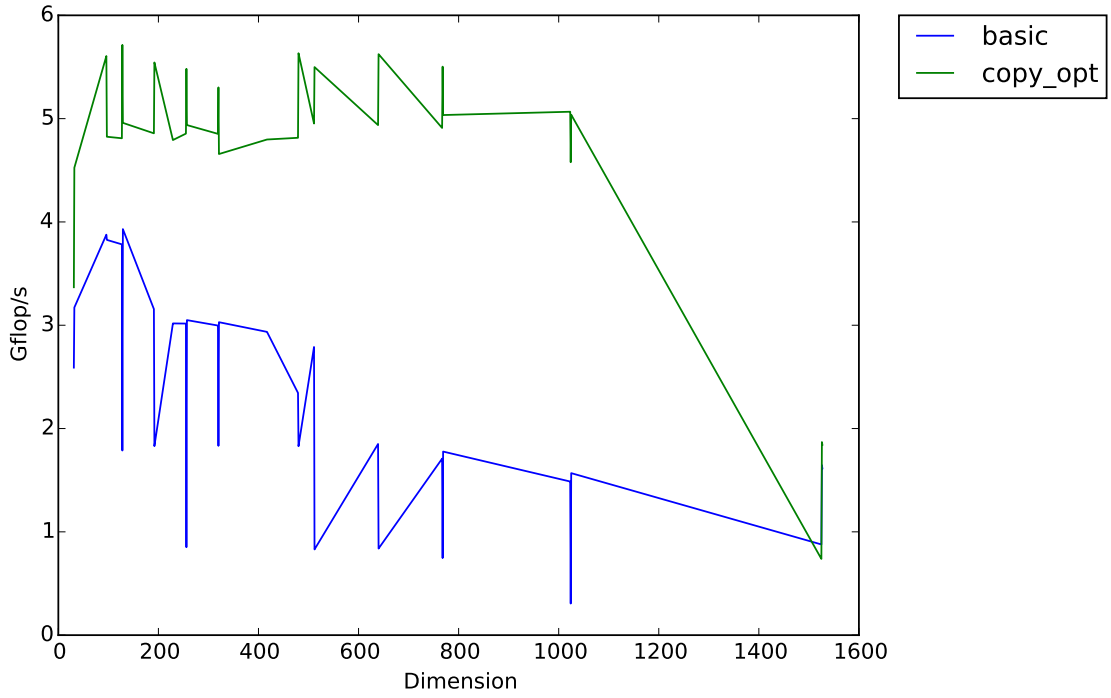

Figure 7: Basic DGEMM vs DGEMM with Copy Optimization

2.6 Compiler flags on Copy Optimization

2.6.1 Approach

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

2.6.2 result

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

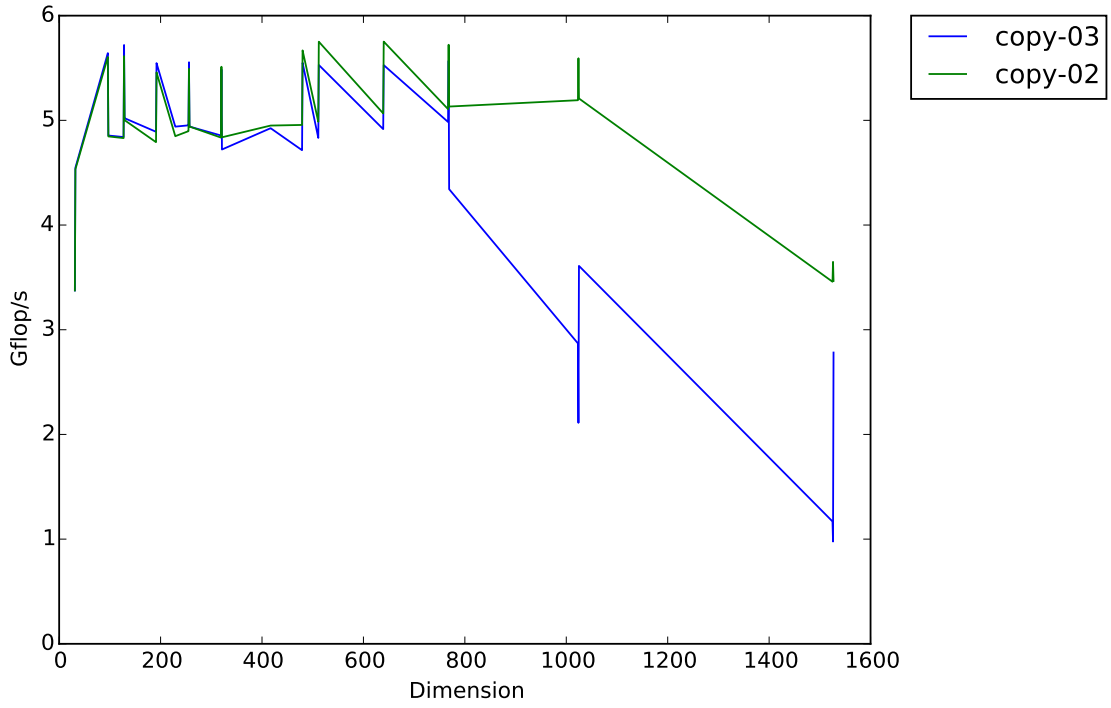

Figure 8: -03 vx -02 on Copy Optimization

2.7 Restrict Keyword

2.7.1 Approach

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

2.7.2 Results

Figure 9 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.

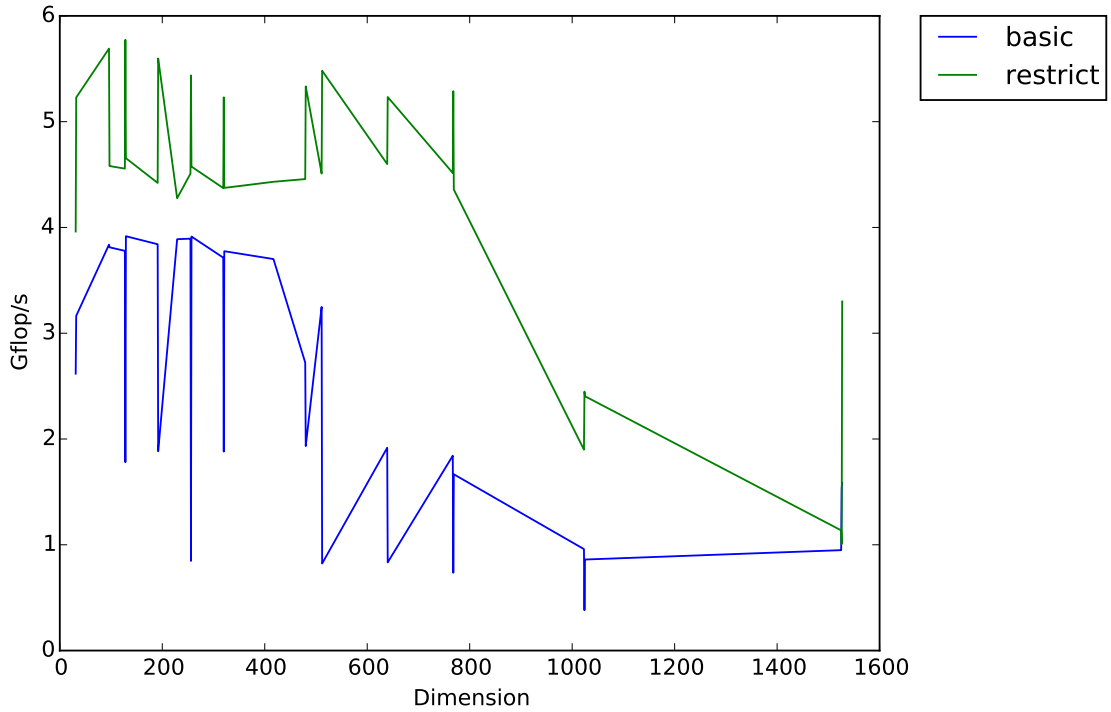

Figure 9: Basic DGEMM vs DGEMM with `restrict` keyword

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

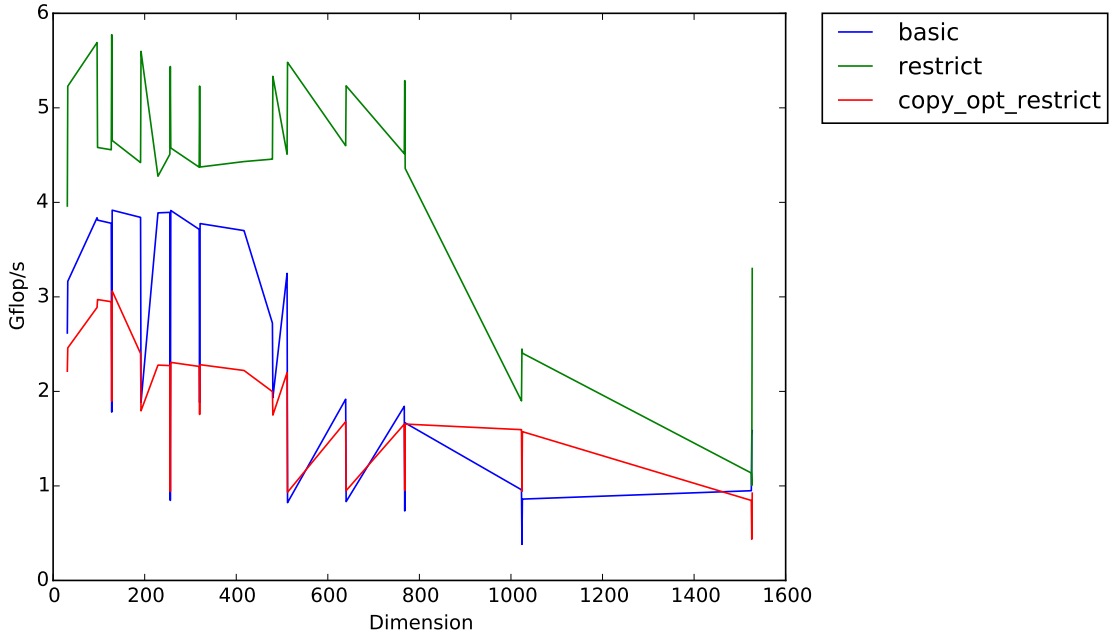

Figure 10: Basic DGEMM vs DGEMM with `restrict` keyword

3 Next Steps

3.1 AVX Instructions

We want to try AVX instructions to take advantage of using multiple registers at once. However we find that there are still bugs in our code and it is difficult to debug on the cluster machines. So we are currently configuring the Intel icc and Parallel Studio on our own laptop.

3.2 Copy Optimization with Blocks

We are also interested in combine copy optimization with block implementation. As shown above, we have implemented the copy optimization on the basic implementation, and when the matrix becomes too big the copy optimization fails. We believe by using the block implementation, we can avoid the performance drop.

3.3 Autotuning

Furthermore, we are interested in automating much of the block size search process once the memory layout and copy optimization ideas are fleshed out.