Matmul Stage 1

Elliot (evc34), Ian (ivy2), Michael (mjw297)

September 17, 2015

1 Overview

Matrix multiplication is a ubiquitous linear algebra building block that can be used to solve a wide variety of problems, and its popularity is an impetus for its optimization. In this paper, we present our progress on writing an optimized matrix multiplication kernel. In §2, we review the optimizations we used and attempted, as well as the motivation for these optimizations and their results. In §3, we discuss the process of composing our optimizations into a single optimized matrix multiplication kernel. In §4, we discuss future work and proposed optimizations.

2 Optimizations

The space of all combinations of optimizations is very large; in order to explore the space efficiently, we analyzed optimizations in a modular fashion where each optimization is deployed without any other optimizations present. In this section, we discuss the motivation and results for these optimizations.

2.1 Loop ordering

A blocked implementation of matrix multiplication contains a series of nested loops, and one of the optimizations we attempted was to change the ordering of these loops. A blocked implementation has two different places where the loop ordering can be changed: changing the order in which the blocks are processed (i.e. outside loop ordering), and changing the loop ordering inside of each block multiplication (i.e. inside loop ordering).

2.1.1 Inside Loop Ordering

When multiplying blocks, we see that there are three index variables: i, j, and k associated with three loops. Changing the ordering of these loops affects the stride and regularity at which we access memory, which can have a significant effect upon performance. For example, striding down the column of a row-major matrix is less efficient that striding across row because the longer stride has less spatial locality and fits poorly in cache.

We note that there are $3! = 6$ possible orderings of these loops. In order to determine which was fastest, we tested and compared all 6 on the totient node. Note that we kept the outside loop ordering constant for all of these, as we assumed that the inside and outside loop orderings were orthogonal, or at least close enough that the difference was not significant. The results can be found in the Figure 1. We see that the loop ordering of j, k, i was clearly fastest, and significantly faster than the slowest loop orderings.

2.1.2 Outside Loop Ordering

Similarly, the order in which we choose which blocks to multiply can also be changed. Again, we compared the six different possible orderings, while keeping the inside loop ordering fixed

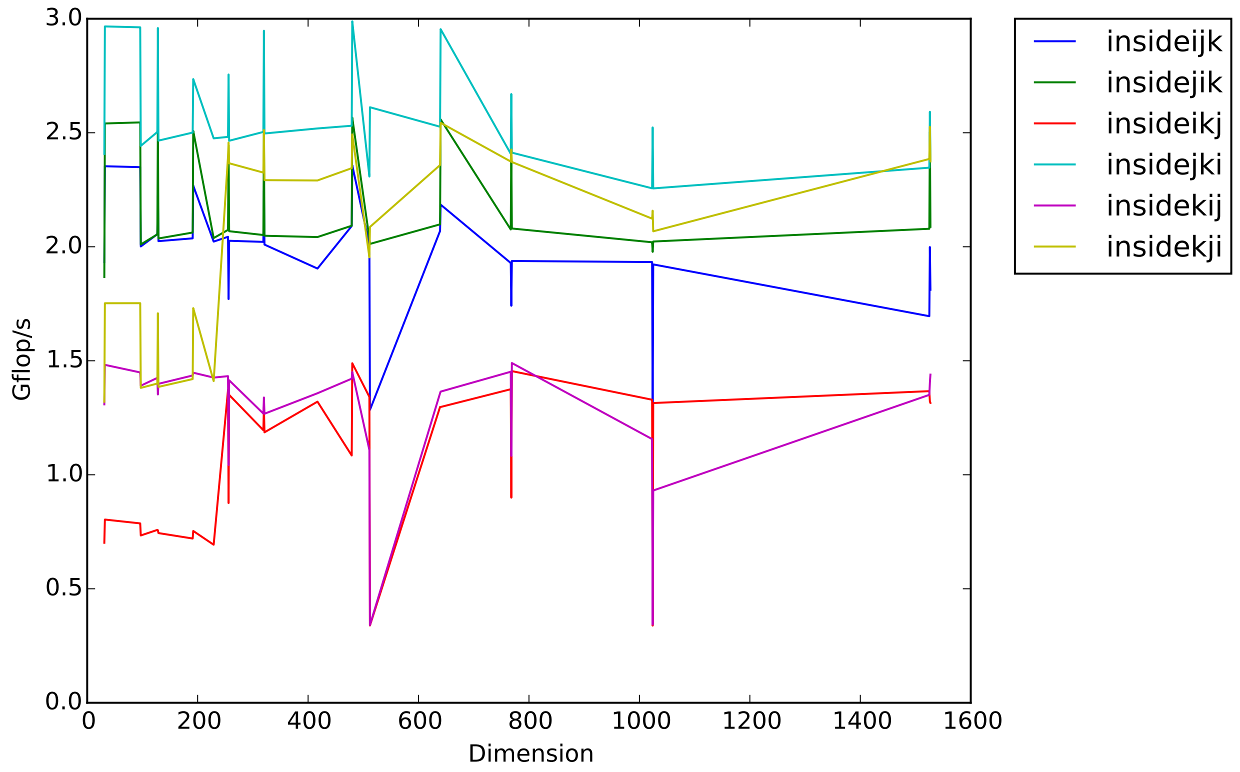

Figure 1: Timing results for the different inside loop orderings

as j, k, i, which was found to be the fastest in the previous subsection. The timing results are found in Figure 2.

We see that some loop orderings are definitely faster than others, but unlike with the inside loop orderings, there is no clear best ordering as both k, j, i and i, k, j are fastest on different size matrices. We chose to go with the outside loop ordering i, k, j

2.2 Copy Optimization

Copy optimizations involve copying and rearranging some or all of a piece of data into a chunk of memory such that operating on the copied data is more efficient than operating on the original data.

In the context of a naive matrix multiplication $A \times B$, the innermost loop of the multiplication accesses matrix B with unit stride but accesses matrix A with a stride of M where M is the number of rows and columns of A and B.This non-unit stride vastly reduced how effectively we could operate on matrix A. First, the large stride has poor spatial locality which can lead to poor caching. Similarly, operating on non-contiguous elements of A makes vectorization hard or impossible.

To solve this problem, we used a copy optimization in which we simply transposed A to a new array in row-major order. This allowed us to access both A and B in unit stride in the innermost loop.

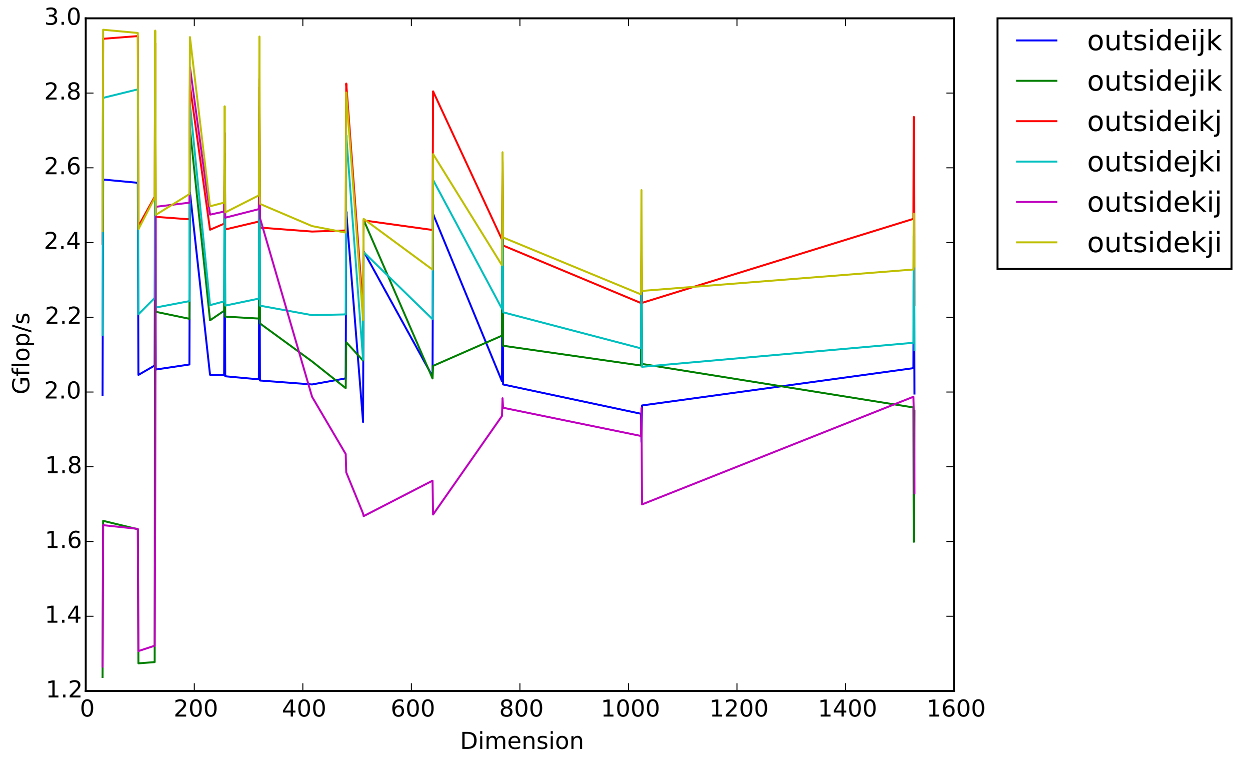

Figure 2: Timing results for the different outside loop orderings

Our use of copy optimizations was the most successful approach that we found for increasing performance. This alone boosted our performance above all other implementations except for MKL and OpenBLAS, as shown in Figure 3.

2.3 Compiler Flags and Annotations

Compilers such as `icc` and `gcc` are equipped with a wide variety of command line flags that can be used to increase the performance of the compiled code. Similarly, the compilers understand a set of source code annotations that can inform the compiler of certain assumptions it can make to optimize code. Empirically, we found that that `icc` produced the fastest code; in this section, we present the `icc` flags and annotations we explored.

2.3.1 Compiler Flags

We experimented with the following `icc` flags.

- `-xCORE-AVX2`: The `-x` flag informs the compiler which platform-specific optimizations it can use. Since we have a Xeon E5 v3 processor, we use the `CORE-AVX2` flag.
- `-fast`: The `-fast` flag enables entire program optimization. Its a meta-flag enables a set of other flags that increase performance including `-O3`, `-ipo`, and `no-prec-div`.

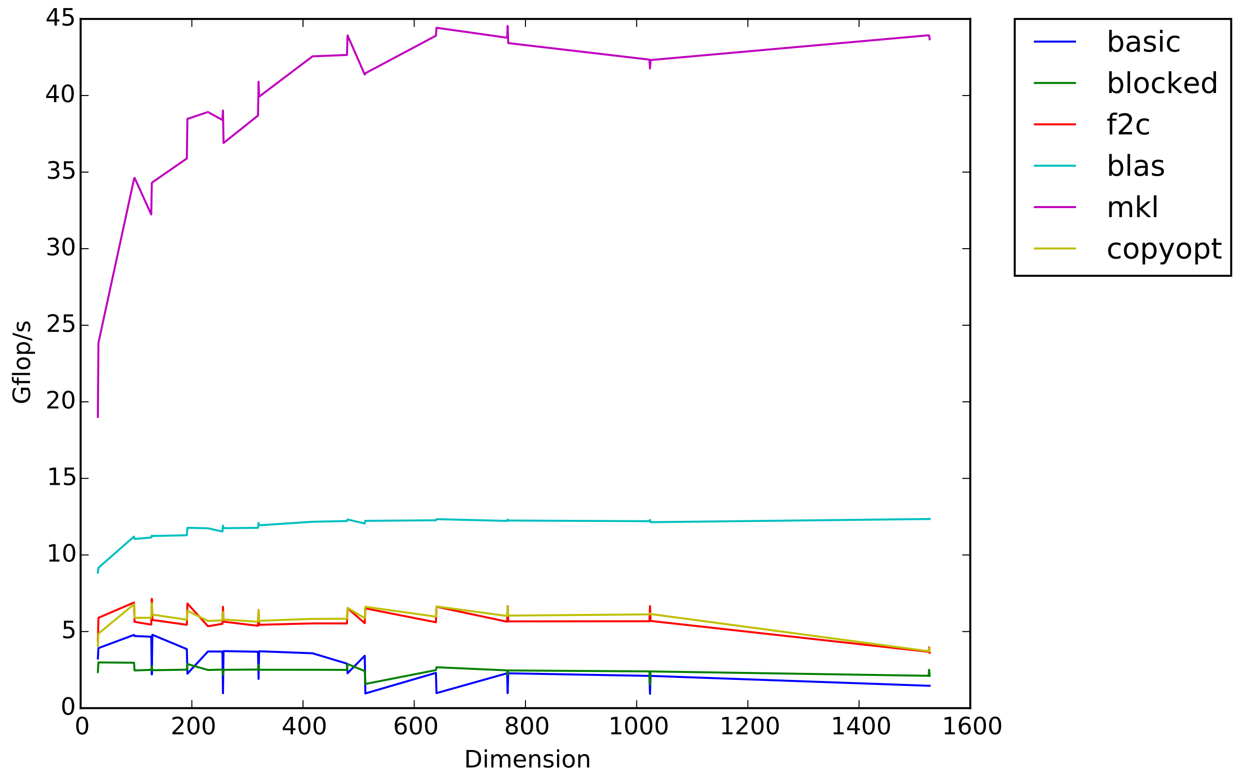

Figure 3: Performance of matrix multiplication with copy optimization.

- `-ansi-alias`: The `-ansi-alias` flag tells the compiler that our code adheres to the ANSI alias guidelines and allows the compiler to make aggressive optimizations.
- `-no-prec-div`: The `-no-prec-div` flag decreases the accuracy of floating point division at the benefit of increased performance. This is also enabled by `-fast`.
- `-ipo`: The `-ipo` flag informs the compiler to perform inter-process optimization. When `-ipo` is enabled, the compiler will optimize code from multiple files when they are linked together.
- `-prof-gen` and `-prof-use`: The `-prof-gen` and `-prof-use` allow for profile-guided optimization. A binary compiled with `-prof-gen` is instrumented such that whenever it runs, it generates a profile documenting the most frequently executed code paths. A binary compiled with `-prof-use` is built to optimize the frequently executed code paths documented in the profiles.

We compiled the naive matrix multiplication with all these `icc` flags enabled, yet the performance is negligible at best, as shown by the line titled `compiler` in Figure 4. We hypothesize that the compiler flags may be more beneficial when used to compile less naive implementations.

2.3.2 Compiler Annotations

We explored two `icc` annotations.

- `restrict`: By default, when a function receives multiple pointers as arguments, the compiler must assume that the two pointers may point to overlapping regions of memory. This assumption can prevent the compiler from automatically vectorizing the code which can negatively affect performance. By annotating pointer arguments with `restrict`, the compiler instead assumes the pointers are not aliased and performs the appropriate optimizations.
- `aligned`: The alignment of data structures can affect the performance of vectorized instructions. Ideally, each vector of data accessed by a vector instruction is 16-byte aligned. The `aligned` annotation can manipulate the alignment of data structures to enforce alignment.

We compiled the naive matrix multiplication with all pointer arguments annotated with `restrict`. This increased the performance of the naive matrix multiplication to that of the Fortran implementation, as shown by the line labelled `annotated` in Figure 4. We did experiment yet with aligning data structures.

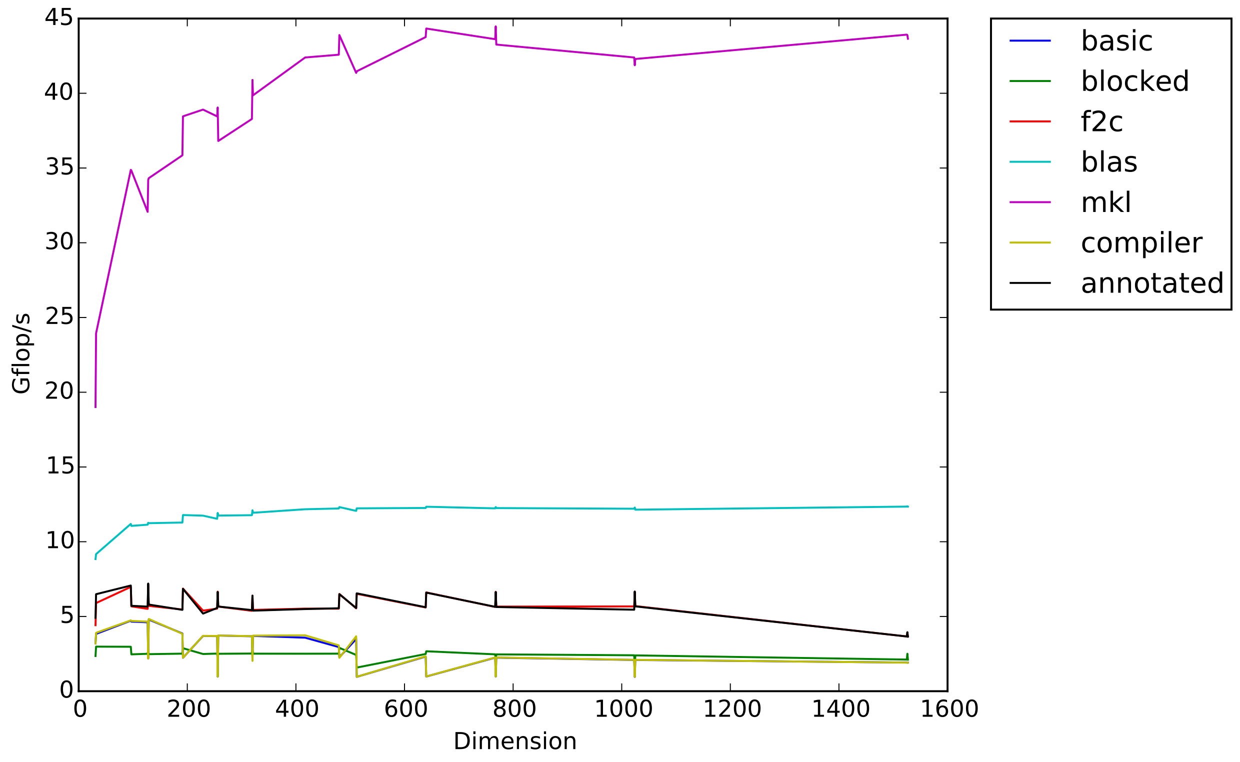

Figure 4: Performance of compiler flags and annotations.

2.4 Vector Instructions

In addition to scalar instructions, most modern instruction set architectures also include vector instructions: instructions which operate on multiple data in a single cycle. These instructions provide data-level parallelism that can greatly increase the performance of code that exclusively uses scalar instructions.

We tried to include explicit SSE instructions to operate on multiple doubles at once. This, however, was unsuccessful. We suspect the Intel compiler already did a good job of leveraging vector instructions since the performance only decreased when we tried to use methods from `xmmintrin.h`. Although we did end up getting this implementation to work, the added overhead and complications of using data types and operations from `xmmintrin.h` only resulted in slowing down our implementation.

3 Composing Optimizations

After we evaluated optimizations individually, we composed the optimizations together in hopes of aggregating the benefits of each optimization. Unfortunately, the optimizations were not orthogonal, and composing them proved non-trivial.

For example, loop reordering conflicted with copy optimization since the order of the loops dictated the access patterns. Similarly, the `aligned` annotation greatly improved the performance of the naive matrix multiplication but had negligible effects on the optimized matrix multiplication.

For now we've prioritized the copy optimizations because they were far more successful at increasing performance. We also had to modify where we did our copying when combined with blocking. Since we changed one matrix to be row-major order, we had to modify how parameters were being passed between the functions for blocking.

The final performance of the fully optimized kernel is shown in Figure 5 as the line labelled `mine`.

4 Future work

There are some further possible attempts at optimization that we would like to try, but have not yet managed to implement in this first stage of the assignment.

- We could more rigorously experiment with the block size to determine which is fastest. We used ad-hoc manual tuning to arrive at our current block size. Further auto-tuning and principled reasoning would likely yield the optimal block size.

- We could change how we handle the parts of matrices that do not fit cleanly into blocks. Currently those are handled in a naive way, but one alternative would be to pad the matrix with zeros so that all (not almost all) multiplication occurs in consistently sized blocks. If the multiplication of blocks is sufficiently well optimized, then this could be noticeably faster.

- We could add an additional layer of constant-sized blocking. There is a possibility that the compiler can introduce more optimization if it knows the exact size of a loop at

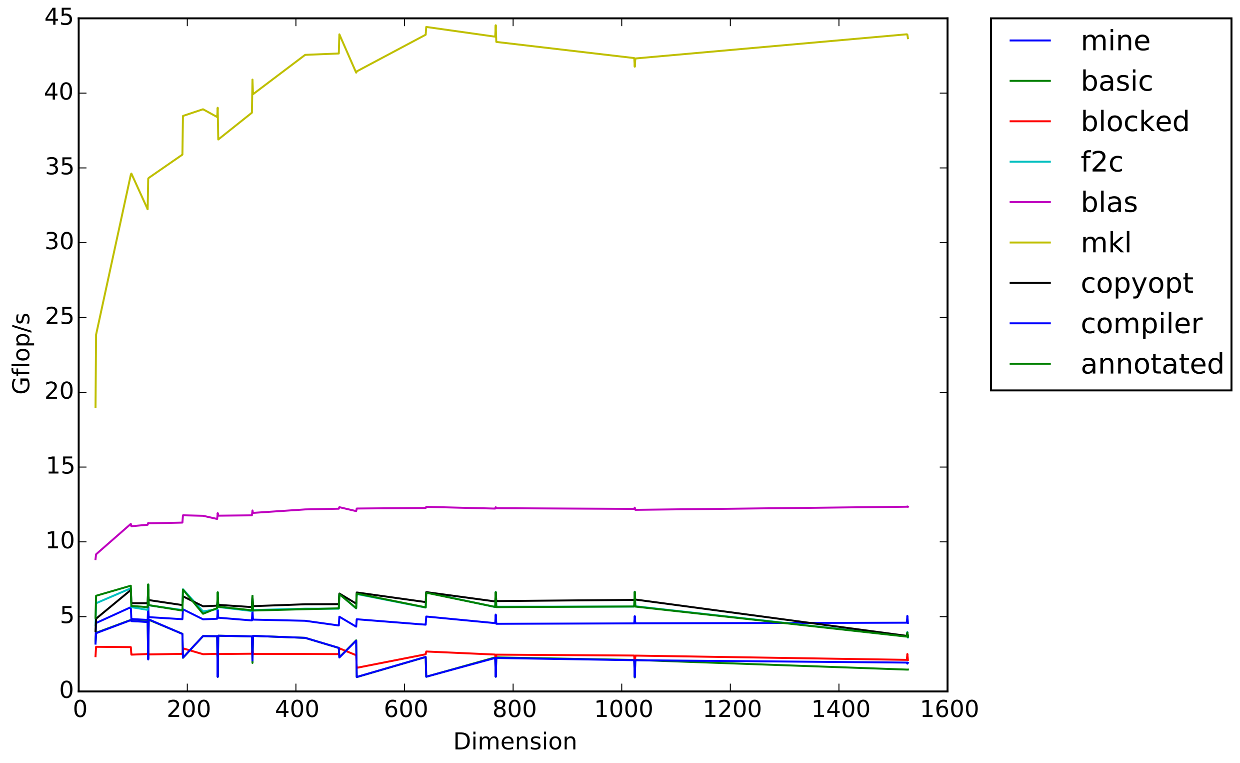

Figure 5: Fully optimized matrix multiplication.

compile time. We could divide our smaller blocked multiplication into even smaller blocks of constant size.

- We could perform finer-grained and additional copy optimization. Currently, when we multiple two column-major matrices A and B, we transpose A. Preferably, we would transpose blocks of A rather than transposing it in its entirety. Furthermore, there could also be benefits to copying blocks of B into contiguous memory regions.
- We could experiment with the `aligned` annotation.