Matmul Stage 1

Elliot (evc34), Ian (iyv2), Michael (mjw297)

October 1, 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 the development of a set of optimized matrix multiplication kernels, as well as lend insights into which optimizations are effective and why. In §2, we review our initial optimization experiments, as well as the motivation for these optimizations and their results. In §3, we discuss various software engineering principles we used to streamline kernel development. In §4 and §5, we present and evaluate our set of fully optimized kernels. Finally, in §6 and §7, we discuss future work and conclude.

2 Initial 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. Later in §4, we discuss how we combined and composed these optimizations.

2.1 Loop ordering

A blocked implementation of matrix multiplication contains a series of nested loops; one of the optimizations we attempted was the reordering of these loops. A blocked implementation has two different places where the loop ordering can be changed: the order in which the blocks are processed (i.e. outside loop ordering), and the 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 the 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 Figure 1. We see that the loop ordering of j, k, i was clearly fastest, and significantly faster than the slowest loop orderings.

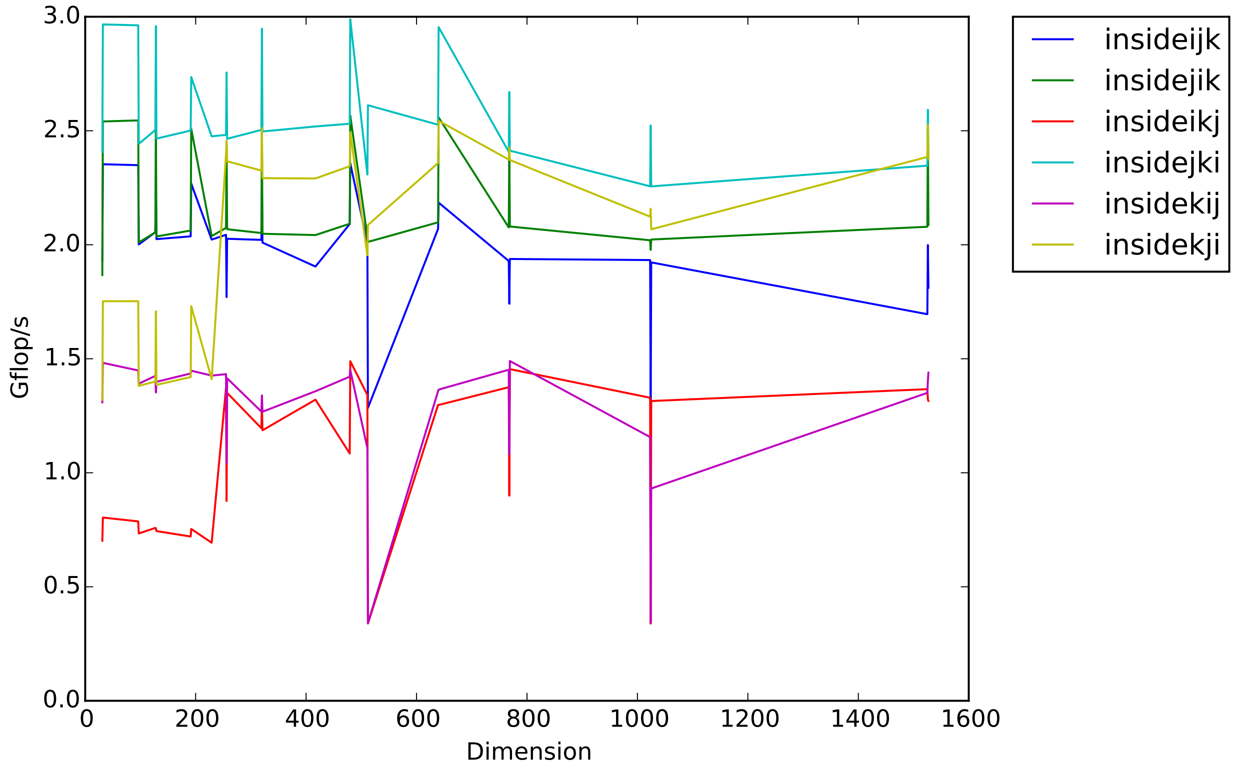

Figure 1: Timing results for the different inside loop orderings

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 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.

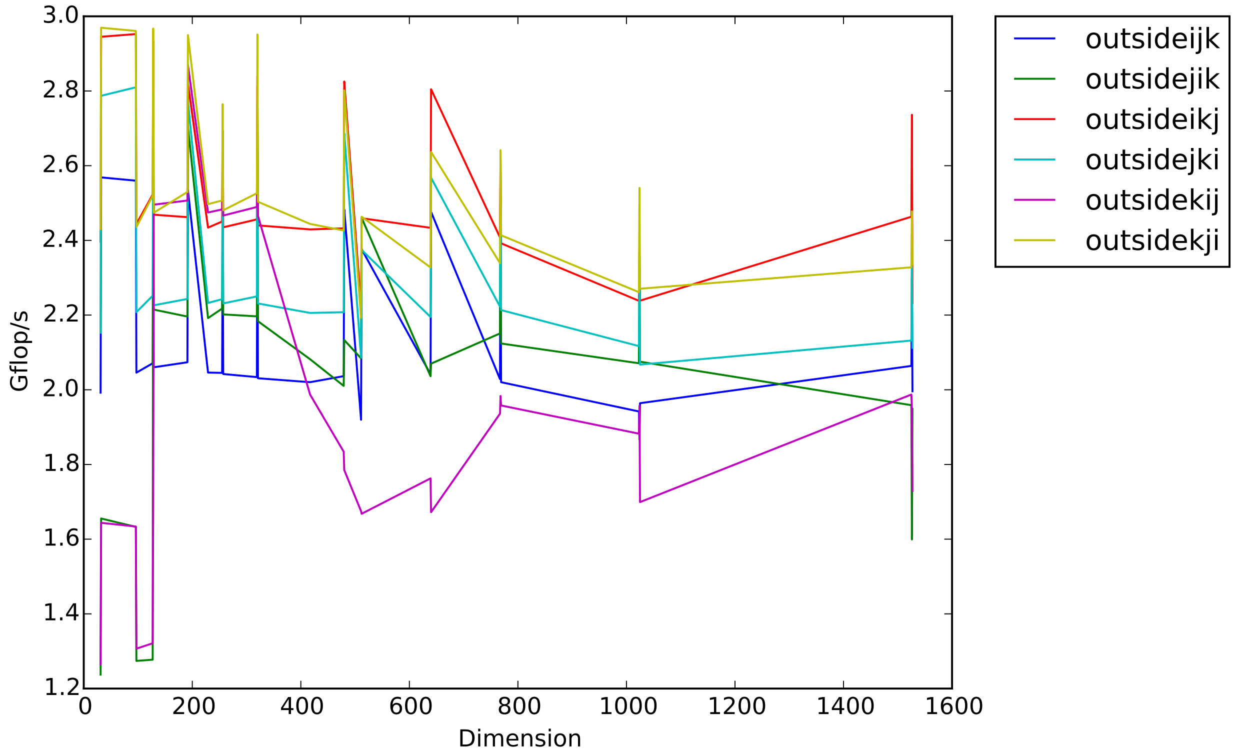

Figure 2: Timing results for the different outside loop orderings

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.

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.

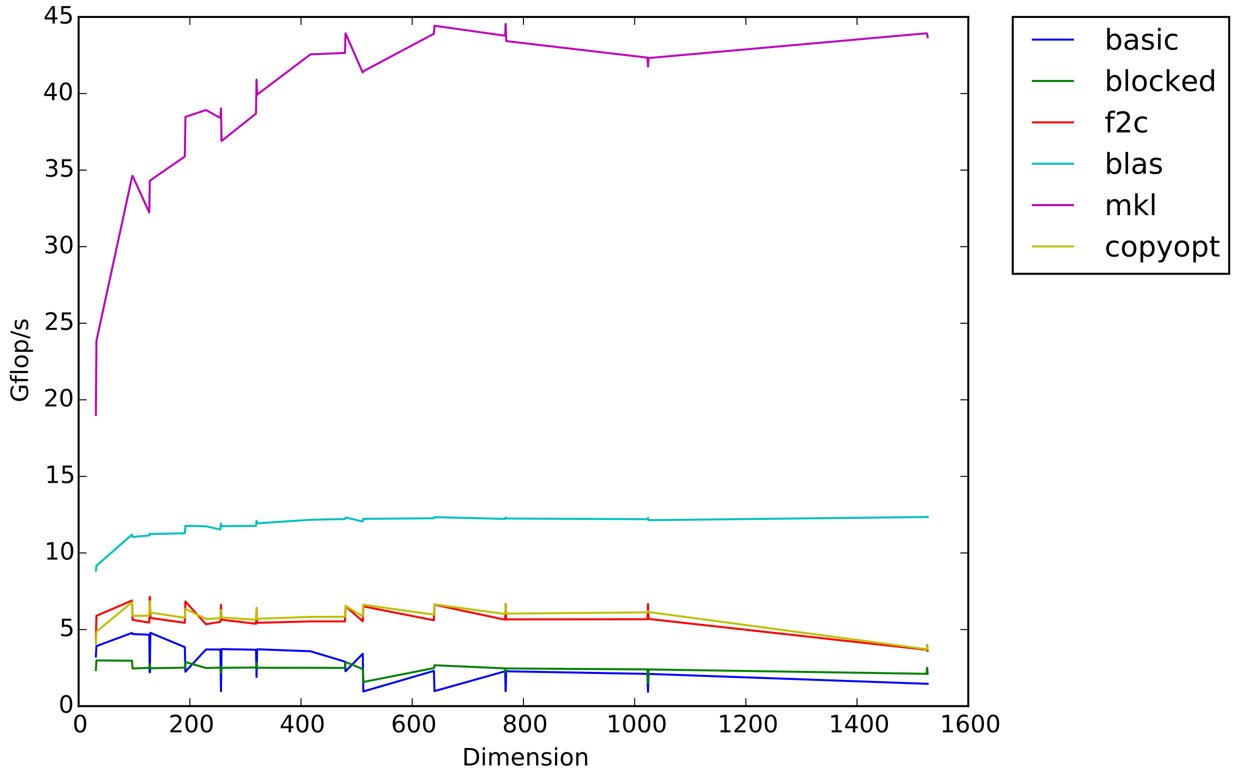

Figure 3: Performance of matrix multiplication with copy optimization.

- `-fast`: The `-fast` flag enables entire program optimization. It's a meta-flag that enables a set of other flags that increase performance including `-O3`, `-ipo`, and `no-prec-div`.
- `-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 with the benefit of increased performance. This is also enabled by `-fast`.
- `-ipo`: The `-ipo` flag informs the compiler to perform inter-procedural 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` flags 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.

  TODO: fill out when you have cluster access
- `-unroll-aggresive`

- `-opt-prefetch=4`
- `-fno-alias`
- `-parallel`
- `-align`
- `-restrict`

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 not yet experiment with aligning data structures.

TODO: update this

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.

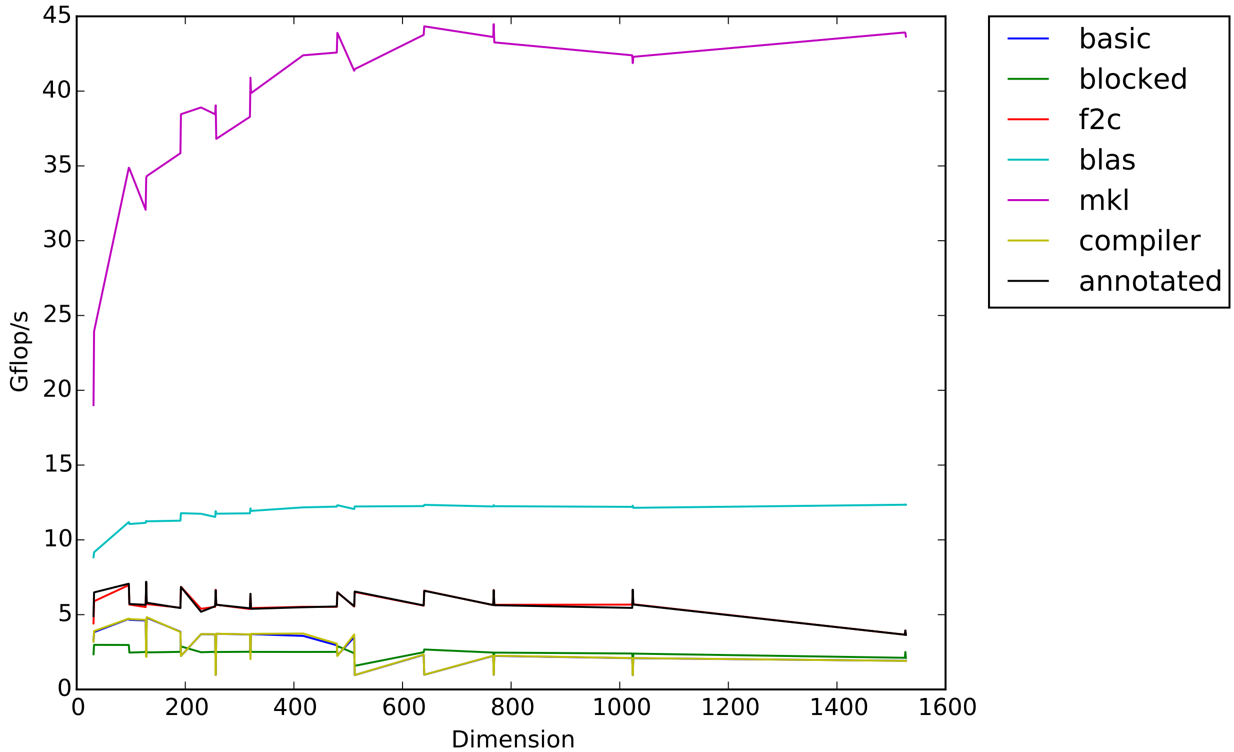

Figure 4: Performance of compiler flags and annotations.

3 Software Engineering

The development of an efficient matrix multiplication kernel is a complicated task that involves a lot of rapid prototyping, and the efficiency with which we can develop kernels is limited by how easy it is to write correct code and by the overhead of creating, modifying, and running kernels. In this section, we describe a couple of the software engineering principles we adhered to that increased the overall productivity of kernel development. Concretely, we discuss our use of helper functions and unit tests as well as our improvements to the build system.

While not directly related to kernel optimization, we feel that our organization and principled software development is a significant achievement that indirectly has a large impact on the quality of kernels written.

3.1 Helper Functions and Unit Tests

Matrix multiplications involve a lot of low-level pointer arithmetic, and in a programming language like C, erroneous pointer arithmetic or invalid memory access can be *very difficult* to debug. Moreover, matrix multiplication involves operations on a combination of row-major and column-major matrices using various complex access patterns involving many nested layers of blocking transposing and copying which greatly increase the likelihood of incorrect code. To combat the likelihood of programming error and reduce the time spent debugging, we developed a modular set of universal helper functions accompanied with a set of unit tests.

For example, one unavoidable part of matrix multiplication is array indexing. We store all two-dimensional matrices as one-dimensional C arrays which means that indexing a matrix involves computing an offset. This offset computation depends on whether the array is stored in row-major or column-major order. Things get even more complicated when indexing into a matrix that is embedded in a larger matrix or when transposing matrices converting them from column-major to row-major.

Rather than manually inlining the index calculations, we instead developed two helper functions that compute array offsets for column-major and row-major matrices. Thus, if we want to access the i, j^{th} element of a column-major matrix A, we don't have to struggle to choose between `A[j + N*i]`, `A[i + N*j]`, `A[j + M*i]`, or `A[i + M*j]`. Instead, we write `A[cm(N, M, i, j)]`. Similarly, we access row-major arrays using `A[rm(N, M, i, j)]`.

We have written similar helper functions for transposing, copying, and clearing arrays and subarrays. Each helper function is verified with a set of unit tests, and each function is inlined to avoid any cost of function calls. Overall, this modularity and verification has greatly reduced a large number of common errors that take a significant amount of time to debug allowing us to spend more time focused on development and optimization.

Enhanced Makefiles TODO: Discuss the use of helper functions and test cases, the modification of the Makefile, and the use of a single runner pbs file. Argue that this was significant work and had positive impact.

4 Optimized Kernels

TODO: Overview that we have a set of optimized kernels and that the performance of each will be deferred till later.

4.1 Padded Blocks

4.2 Dynamic Padded Blocks

4.3 Three Tiered Blocks

4.4 Copy Optimized Blocks

5 Evaluation

TODO: Show final plots and evaluate our solution

6 Future work

TODO: Update 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 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 multiply 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.

7 Conclusion

TODO: say something grandiose