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 conclusions.

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 the 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.

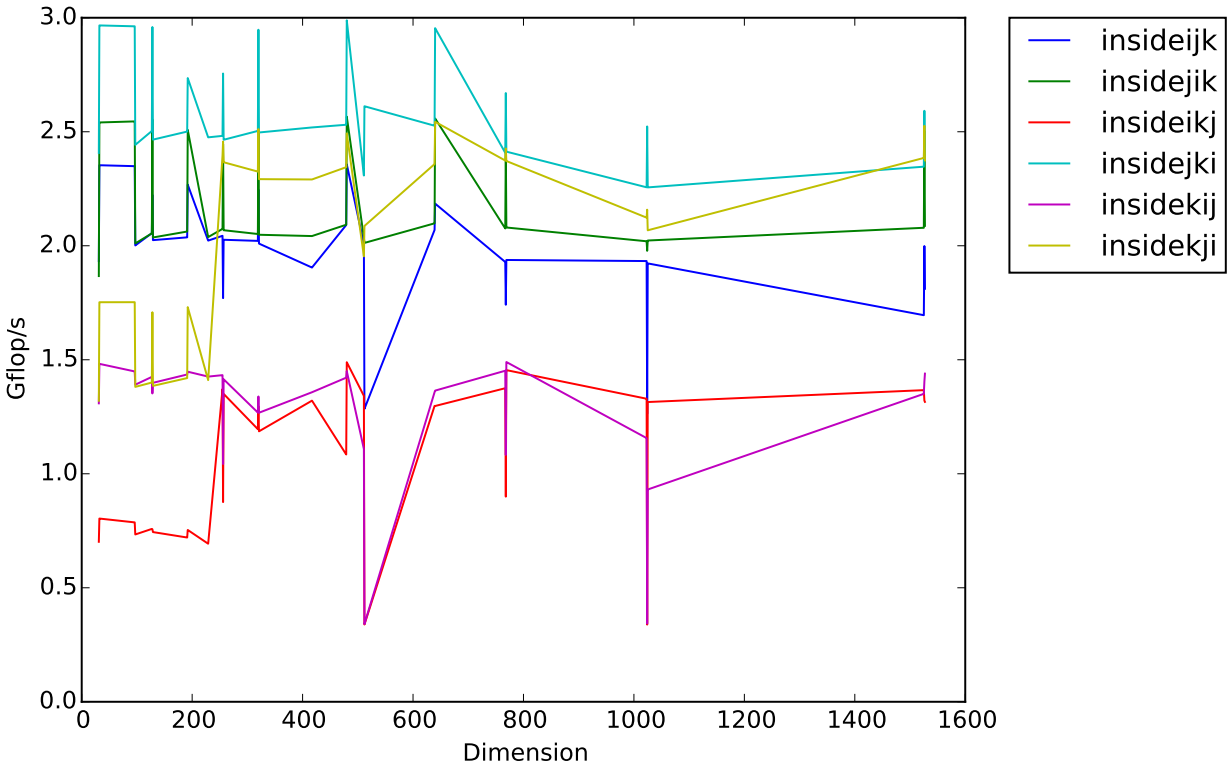

Figure 1: Timing results for the different inside loop orderings

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 even impossible.

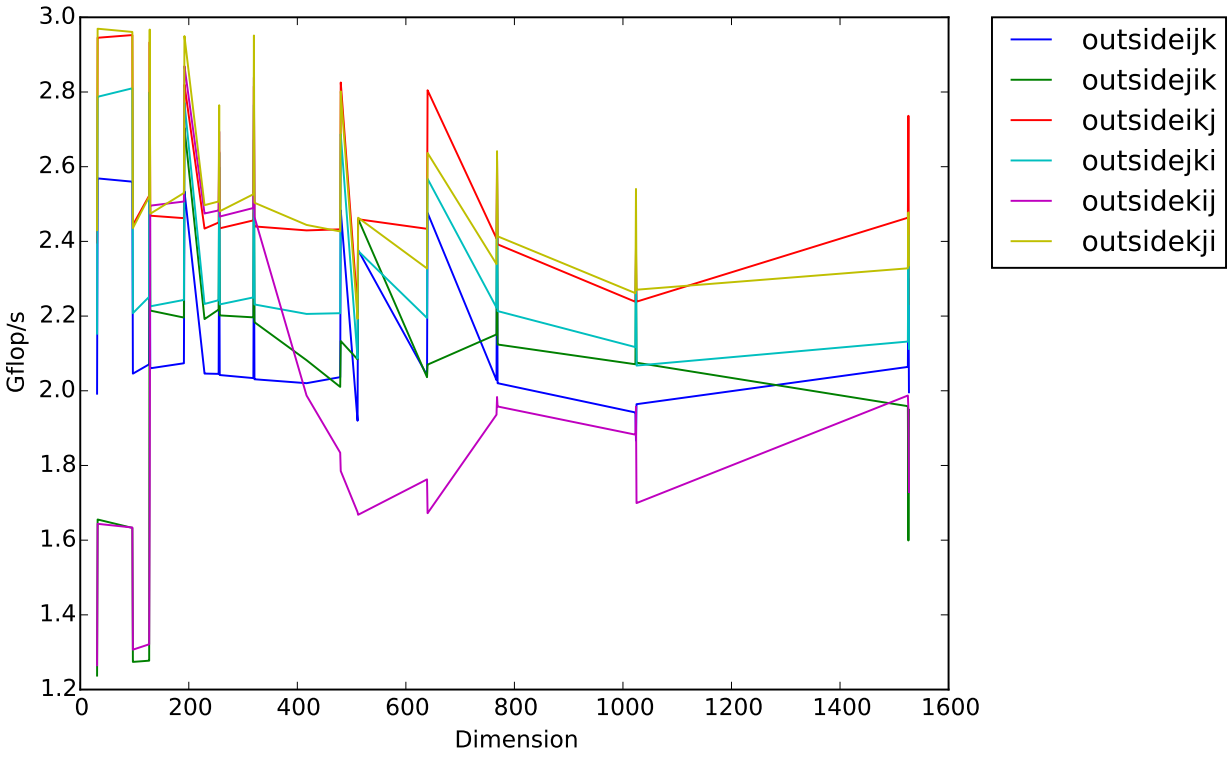

Figure 2: Timing results for the different outside loop orderings

To solve this problem, we used a copy optimization in which we simply transposed A into 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 one of the most effective approaches 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.

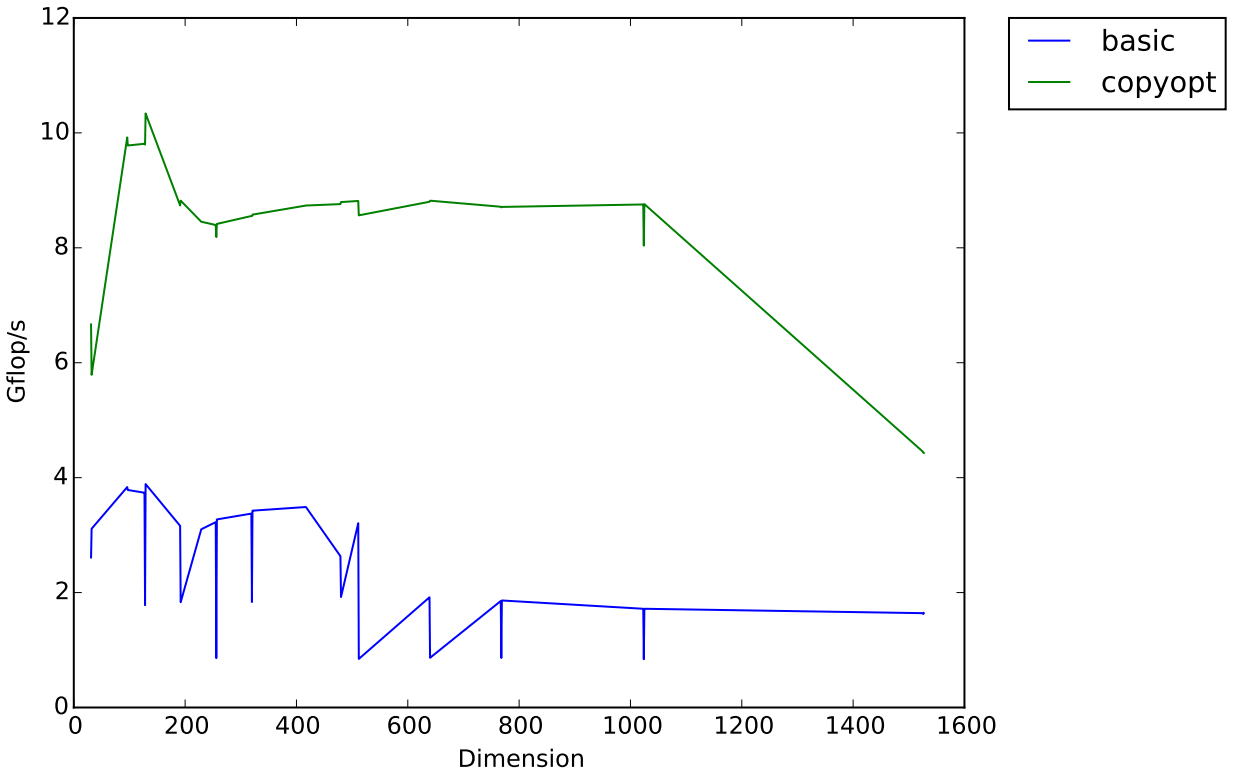

Figure 3: Performance of matrix multiplication with copy optimization.

- `-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.
- `-unroll-aggresive`: The `-unroll-aggresive` flag informs the compiler to perform aggressive loop unrolling which can reduce loop overheads at the cost of increased binary sizes.

- `-fno-alias`: The `-fno-alias` flag informs the compiler to assume that pointers are never aliased. We did not use this flag to compile our code because in general, we cannot assume that matrices are not aliased.
- `-align`: The `-align` flag informs the compiler to naturally align variables and arrays whenever possible which helps vector instruction performance.
- `-restrict`: The `-restrict` flag enables the `restrict` annotation.

We compiled the naive matrix multiplication with all these `icc` flags enabled, yet the performance is only marginally improved, as shown by the line titled `compiler` in Figure 4. Notably, compiler flags do have a greater impact on more complex kernels.

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 considerably increased the performance of the naive matrix multiplication, as shown by the line labelled `annotated` in Figure 4.

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.

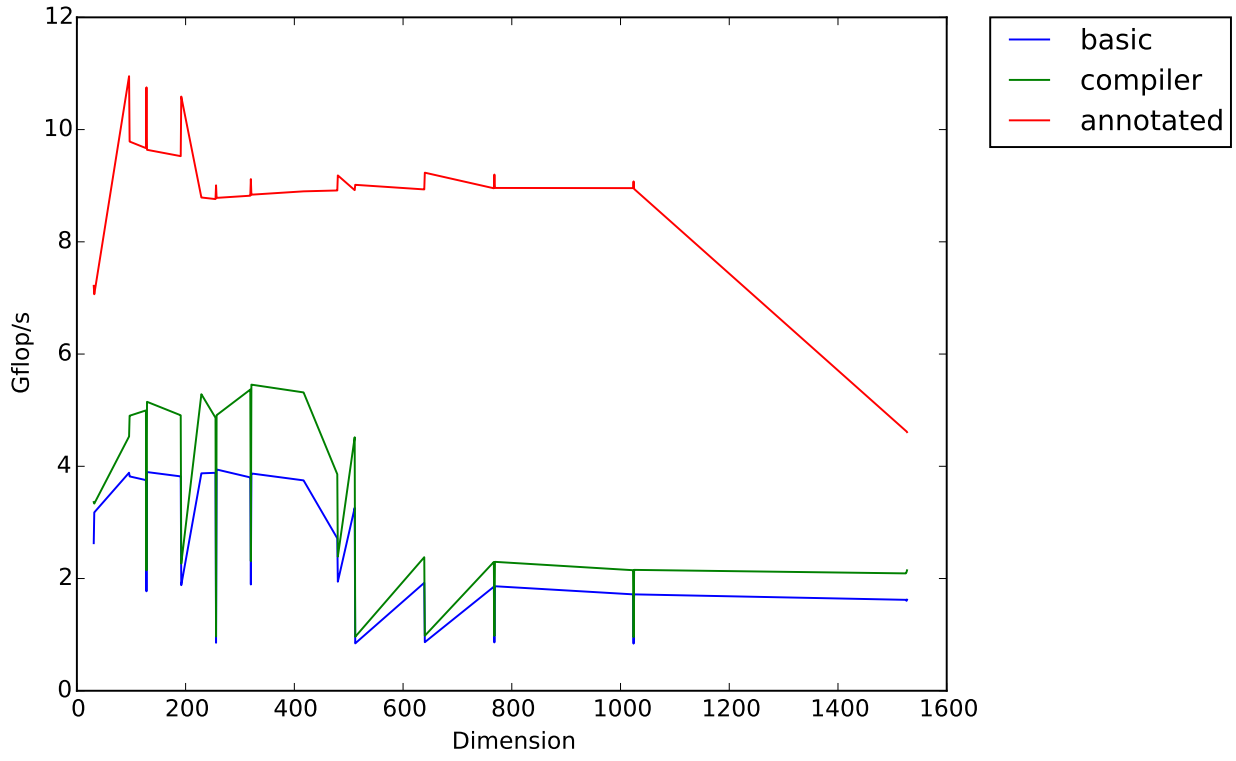

Figure 4: Performance of compiler flags and annotations.

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, optimized 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.

Improved Build System In order to create, build, and run a kernel using the original `matmul-` build system, one has to write the kernel's code, modify a Makefile to include the new kernel, and create a portable patch script to run the kernel on the cluster. Given a small number of kernels, this system is adequate. However, kernel development involves a lot of rapid prototyping in which kernels are quickly copied, modified, and compared. Using the original build system, this becomes onerous. Modifying the Makefile is easy to forget, and creating a script for each kernel not only creates a tremendous amount of clutter but also takes an annoying amount of time. The large overhead discourages prototyping leading to less informed optimization decisions and unexplored possibilities.

To eliminate this overhead, we first enhanced the Makefiles. We added a bit of Makefile logic to eliminate the need to update a Makefile with the name of every kernel to build. Instead, the Makefile automatically detects all the files of the form `dgemm_*.[cf]` and targets them for building and running. Moreover, we created a single portable batch script that runs the kernel specified as a command line argument. We then modified our Makefile to invoke this batch script with the appropriate arguments. This eliminates the need to have a script for every kernel, and instead, we rely on a single script.

4 Optimized Kernels

After experimenting with various simple optimizations, we developed a comprehensive set of kernels that combine and compose various optimizations as well as introduce more complex optimizations. Concretely, we developed four fully optimized kernels. In this section, we discuss the motivation and design of each kernel. In §5, we evaluate the performance of each kernel.

4.1 Copy Optimized Blocks

The `copyopt` kernel presented in §2 performs well for small matrices, but performs poorly for large matrices. More specifically, when the matrix does not fit in L3 cache, the naive matrix multiplication algorithm exhibits poor spatial locality which significantly limits performance.

To adopt the benefits of copy optimization without the disadvantage of poor cache locality, we created the `big_blocked` kernel: a kernel that uses copy optimization and a blocked data access pattern. Concretely, the `big_blocked` kernel performs a blocked matrix multiplication in which a block A_{ik} of A is multiplied by a block B_{kj} of B to produce a block C_{ij} of C. Before each block multiplication, A_{ik} is copied into a buffer in row-major order.

We performed a parametric sweep on the block size of the `big_blocked` kernel to decide on an optimal block size. The results of the sweep can be found in Figure 5. A block size of 128 was selected.

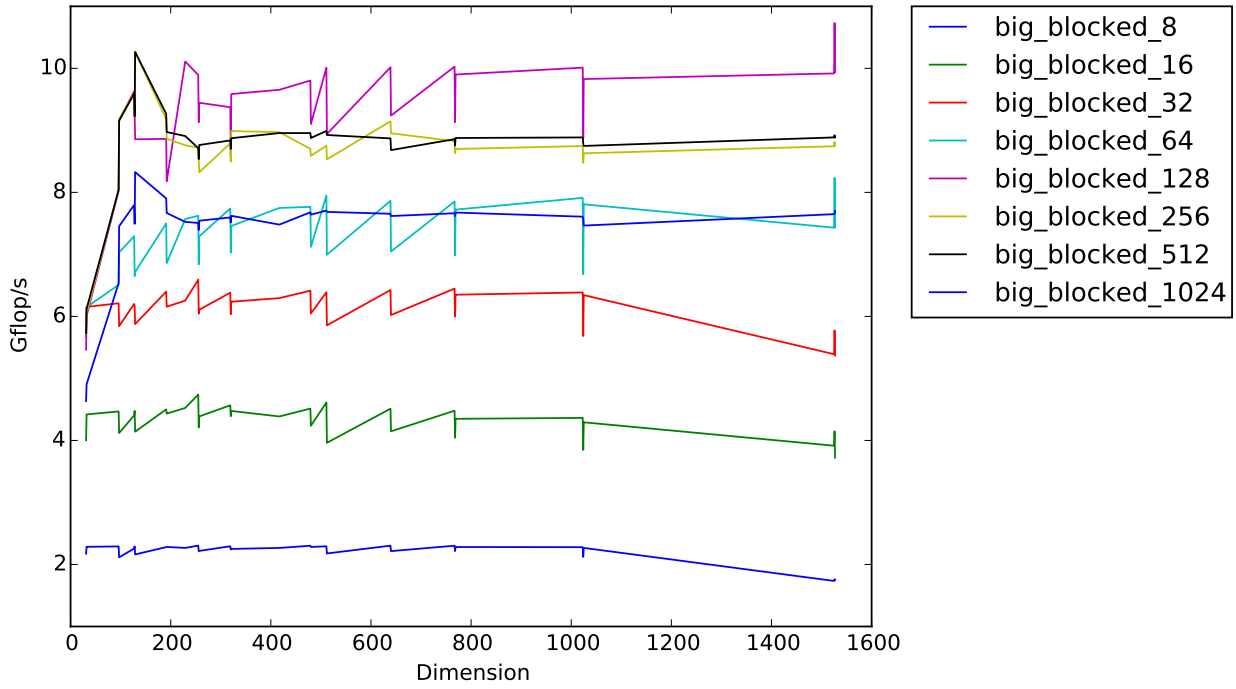

Figure 5: Parametric sweep of block size for the `big_blocked` kernel.

4.2 Three Tiered Blocks

We attempted to extend our previous blocking implementations with multiple levels of blocking. The original idea behind this method was to have 3 layers of blocking that each corresponded to a level of caching. Based on the results of the `membench` program, we estimated the L3 cache to be 15MB, the L2 cache to be 256kB, and the L1 cache to be 64kB. Each of these caches could thus store square matrices with a dimension of 1402, 181, and 45, respectively. Since these weren't very nice numbers we decided to round down to the nearest powers of two, setting the block sizes at 1024, 128, and 32, respectively. While this certainly

yielded a performance boost over the basic version, this implementation still fell well short of other kernels.

A second attempt to optimize this method was by adding padding to the lowest level of blocking to help create more uniform memory access patterns. In doing this we also experimented with making the lowest level of blocking smaller to prevent too many unnecessary zeros being calculated. This yielded some performance boost, but nothing significant. We also played around with the size of the two other blocking levels as well as only doing two levels of blocking, but most of these attempts only hindered performance.

Our final attempt to increase performance was to incorporate the results from previous blocking attempts. These other attempts always found that big block sizes, larger than 128, yielded the best results. For this reason we increased the all the block sizes and only went down one level of blocking if the matrix at the current blocking level was not a square. This yielded the best and most consistent results of all the attempts.

4.3 Padded Blocks

Consider a loop of the following form:

```
for (int i = 0; i < I; ++i) {
  for (int j = 0; j < J; ++j) {
    do_stuff(i, j);
  }
}
```

When `I` and `J` are values that are unknown at compile time, the compiler is limited in the optimizations it can perform. For example, if it unrolls the loop, it must be careful that the unroll factor divides `I` or `J`. If it does not, then it has to perform some of the computation in the unrolled loop and insert remainder loops to perform the remaining computation.

When `I` and `J` are known at compile time, the compiler can perform additional optimizations. For example, it can perform more education loop unrolling and can more efficiently use vector instructions.

The `padded_blocked` kernel uses copy optimization and padding in such a way that our inner most loops have bounds known at compile time. Concretely, the `padded_blocked` defines a compile time constant `BLOCK_SIZE`. The kernel performs a blocked matrix multiplication where a block A_{ik} of A is multiplied by a block B_{kj} of B to produce a block C_{ij} of C. Before each block multiplication, A_{ik} is copied into a buffer of size `BLOCK_SIZE` $\times$ `BLOCK_SIZE` in row-major order. Similarly B_{kj} is copied into a buffer of the same size in column-major order. If ether A_{ik} or B_{kj} is smaller than the buffer, then the remaining elements are zeroed.

We performed a parametric sweep on the block size of the `padded_blocked` kernel to decide on an optimal block size. The results of the sweep can be found in Figure 6. A block size of 128 was selected.

4.4 Dynamic Padded Blocks

Figure 6 shows that the performance of the `padded_blocked` kernel performs very well when the matrix size is a multiple of the block size. On the other hand, the kernel performs

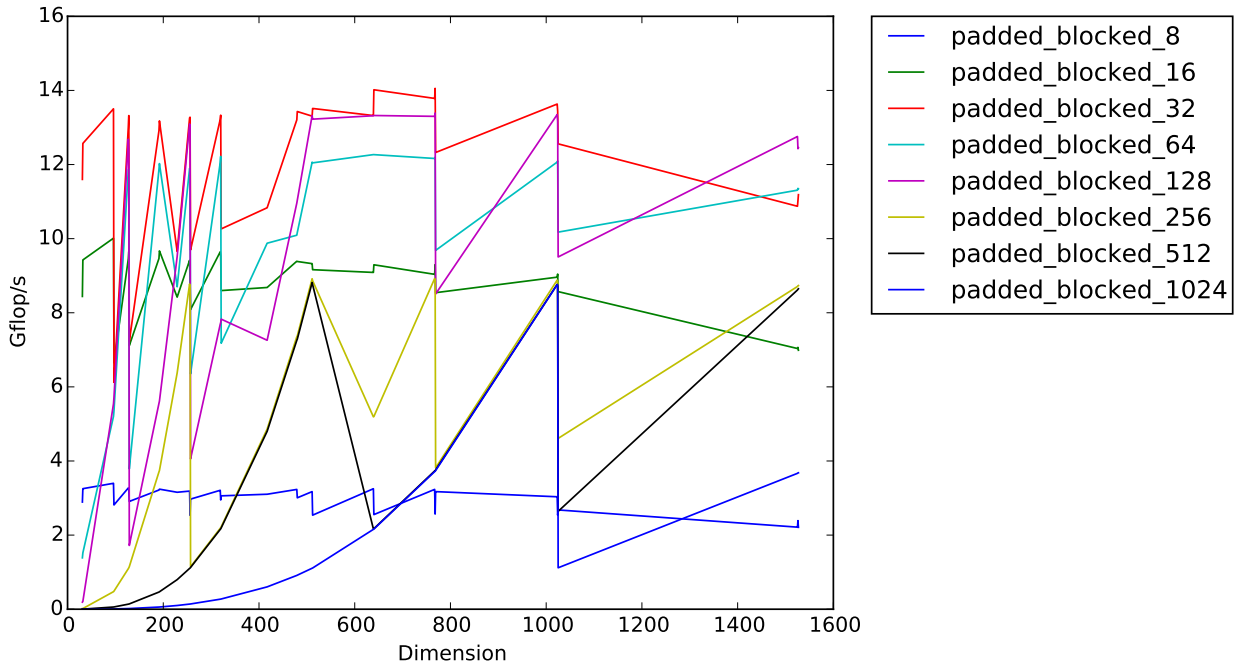

Figure 6: Parametric sweep of block size for the `padded_blocked` kernel.

poorly when matrix size is just over a multiple of the block size. This is consistent with our intuition. When the matrix size is just over a multiple of the block size, the final set of blocks in the multiplication are mostly empty, so the algorithm performs superfluous computation.

The `padded_if` kernel is a hybrid between the `padded_blocked` and `big_blocked` kernel. When multiplying two blocks of size `BLOCK_SIZE` $\times$ `BLOCK_SIZE`, the kernel uses the optimized multiplication from the `padded_blocked` kernel. However, when the size of the blocks being multiplied are smaller than `BLOCK_SIZE` $\times$ `BLOCK_SIZE`, the kernel uses the unoptimized multiplication from the `big_blocked` kernel. Thus, the `padded_if` kernel does not perform superfluous computation, but it does have some overhead in dispatching between two different loops.

5 Evaluation

The performance of our fully optimized kernels is shown in Figure 7. The performance of the `big_blocked` and `3_level_blocking` kernel are consistent for all sizes of matrices but generally perform worse than the `padded_blocked` kernel. This indicates that the compiler's ability to efficiently compile loops with a fixed number of iterations has a large impact on the efficiency of the code.

Moreover, the `padded_if` kernel expectedly performs more consistently than the `padded_blocked` kernel because it does not perform any superfluous computations. Similarly the `padded_blocked` kernel outperforms the `padded_if` kernel when the matrix size is a multiple of the block size. This is because the `padded_blocked` kernel does not pay the overhead of dynamically select-

ing between two different loops.

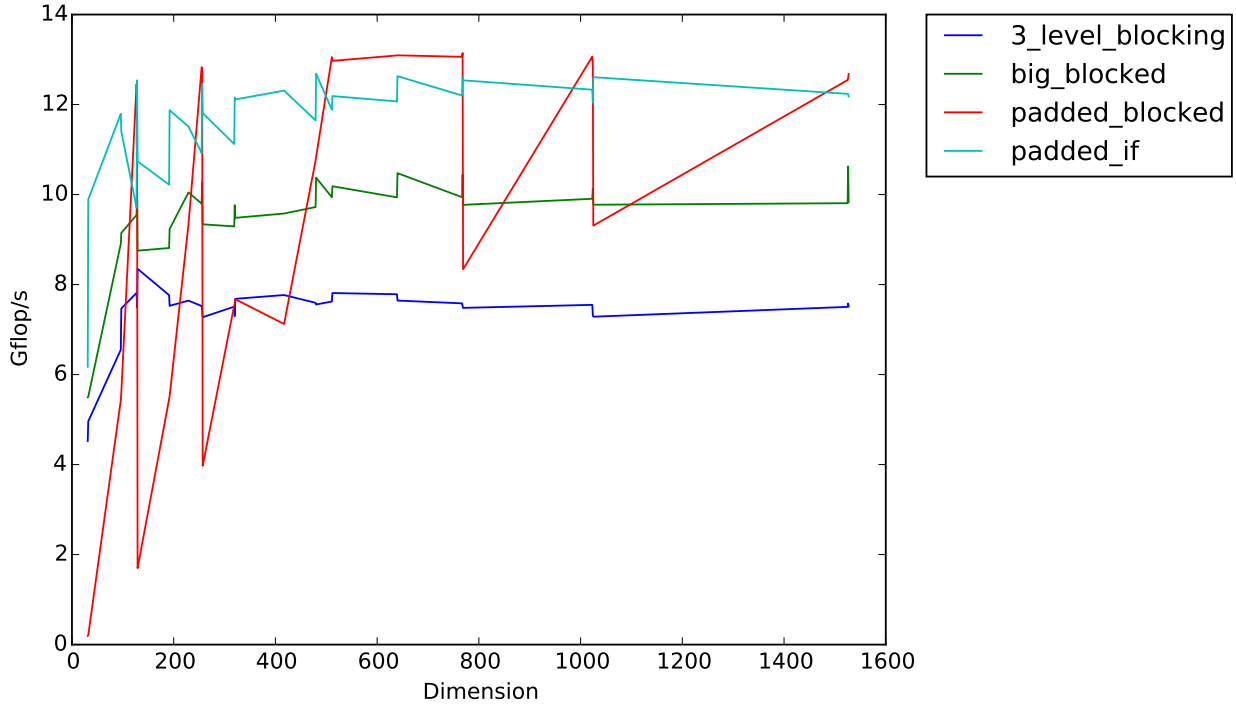

Figure 7: Evaluation of optimized kernels.

The performance of our optimized kernels compared to the performance of all the kernels is shown in Figure 8. The same plot without the `mkl` or `blas` kernels is shown in Figure 9. The peak Gflop/s of our kernels is roughly 4 times lower than that of `mkl` and `blas`. On the other hand, our kernels achieve almost 6 times the number of Gflop/s of the naive matrix multiplication kernel.

Moreover, the running time of all kernels when run on a variety of matrices is shown in Table 1. Notably, the `padded_blocked` and `padded_if` kernels are only a couple of seconds slower than the `mkl` and `blas` kernels.

6 Future work

There are various other optimizations that we did not try which may further increase the performance of a matrix multiplication kernel.

- We did not use any explicit vector instructions in our kernels because the overhead of correctly using explicit vector instructions was very high, and we were uncertain if doing so would significantly increase the performance of the kennel. In the future, we could introduce explicit vector instructions.
- To multiply two $N \times N$ matrices, all our kernels perform at least N^3 multiplications and additions. We could implement more sophisticated algorithms which reduce the

blas	0:45
mkl	0:45
padded_blocked	0:47
padded_if	0:48
big_blocked	0:54
3_level_blocking	0:56
annotated	1:06
copyopt	1:07
f2c	1:11
blocked	1:39
compiler	1:58
basic	2:04

Table 1: Running time of all kernels.

needed number of multiplications. A more sophisticated algorithm would likely be more difficult to vectorize making it efficient on very large matrices and very inefficient on small and modest sized matrices.

7 Conclusion

TODO: say something grandiose

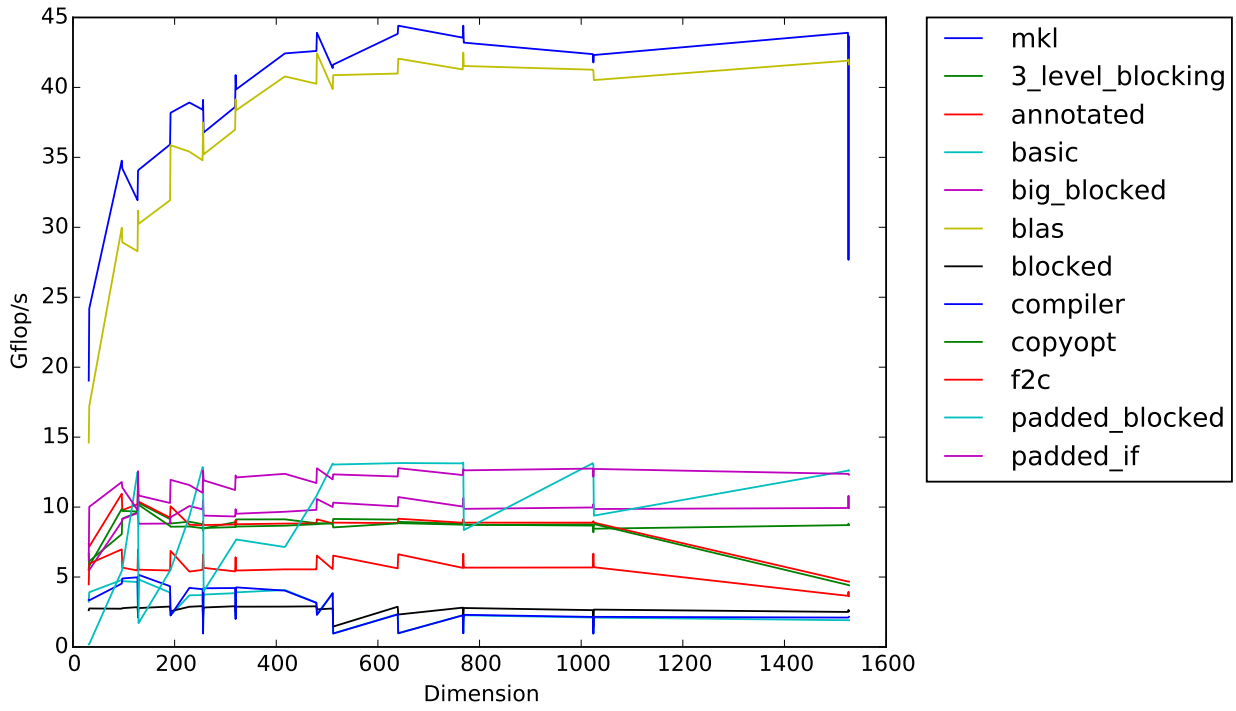

Figure 8: Evaluation of all kernels.

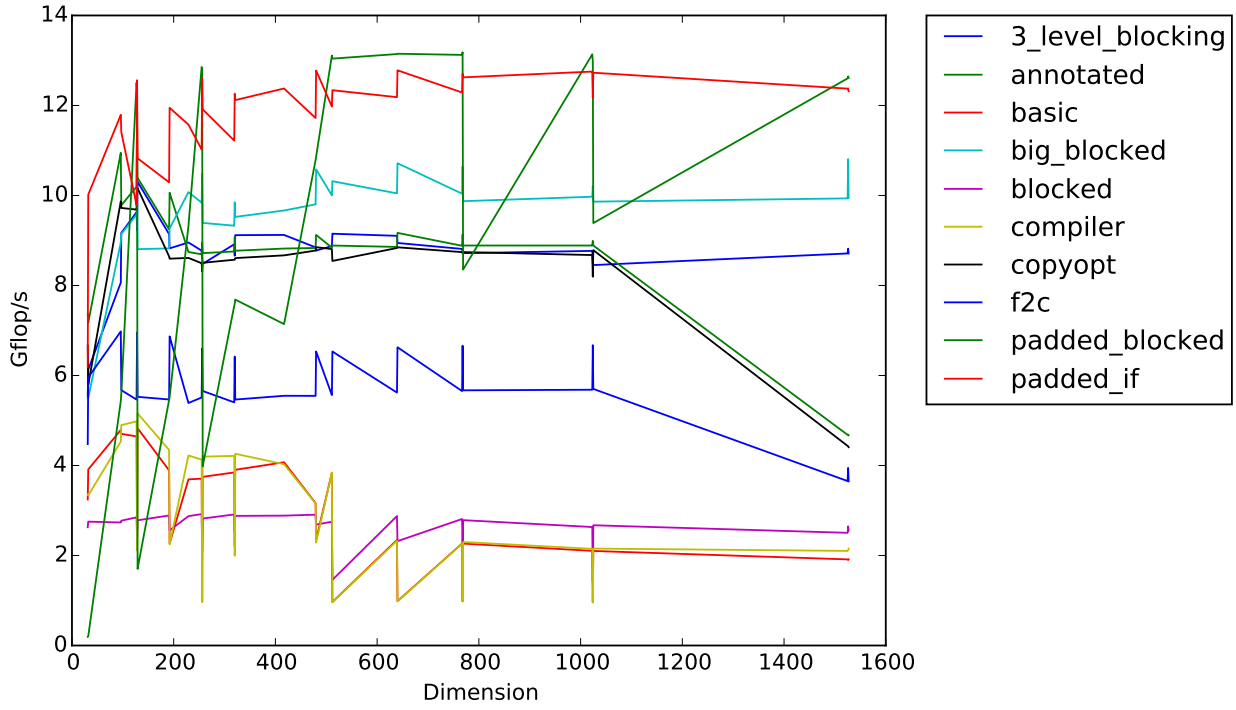

Figure 9: Evaluation of all kernels except `blas` or `mkl`.