Group 12 - Alan Cheng (ayc48), Jason Setter (jls548), Saul Toscano (st684)

Matrix Multiplication

In most programming, we use highly productive programming languages, such as Python and Java, to simply describe problems in a language the computer can understand after numerous translations. However, when we are concerned about execution performance, such as in high performance computing, we use special techniques to speed up the execution. We attempt to speed up very common, computationally intensive tasks by exploiting knowledge of microarchitecture and parallelizable work that may ignored in typical programming. This takes additional time and effort, but due to the high use of the code and difficulty of the computation, this additional effort is well worth it.

As an example, we use problem of matrix multiplication. With a naive implementation, there are many computations used and slow performance. However, with close manipulation of the order computations are performed, and careful consideration to the processor cache, it is possible for orders of magnitude of improvement in performance.

Optimization methods used

1. We implemented the blocking approach in the j,k,i ordering.
2. We implemented the blocking approach in the j,k,i ordering doing copy optimization. Specifically, we copy the matrices C and B in our approach.
3. We copied and transposed matrix A before using the basic approach.
4. Same as (3), but with some more optimization flags on: -march=corei7-avx -ftree-vectorize
5. We hierarchically expanded blocked matmul for each cache size

Reasons for using the previous methods (correspond to numbers above)

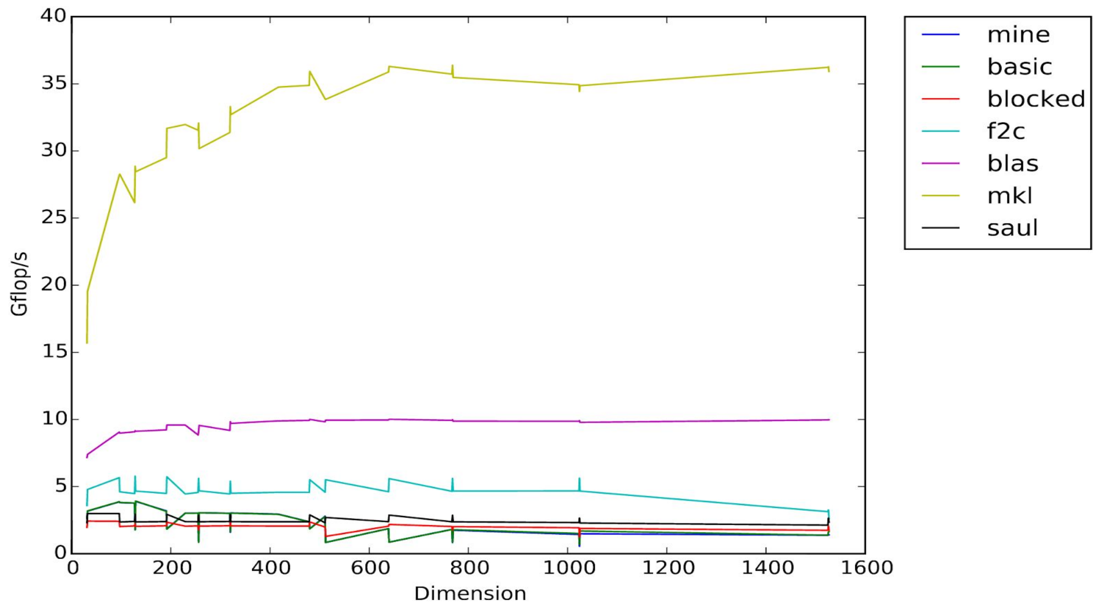
1. We implemented the blocking approach because the little blocks will fit into cache. Furthermore, according to http://www.cs.cornell.edu/~bindel/class/cs5220-f11/notes/serial-tuning.pdf, the reference Fortran DGEMM does better with the j, k, i ordering.
2. We do copy optimization to reduce the number of conflict misses.
3. By transposing matrix A, we can calculate each element of C by iterating over columns of A instead of rows of A, which nets us unit stride.
4. The -ftree-vectorize flag is suggested in the serial tuning notes, and -march=corei7-avx seems to be the appropriate setting for our Xeon E5-2620 according to the venerable

experts on StackOverflow:
http://stackoverflow.com/questions/943755/gcc-optimization-flags-for-xeon

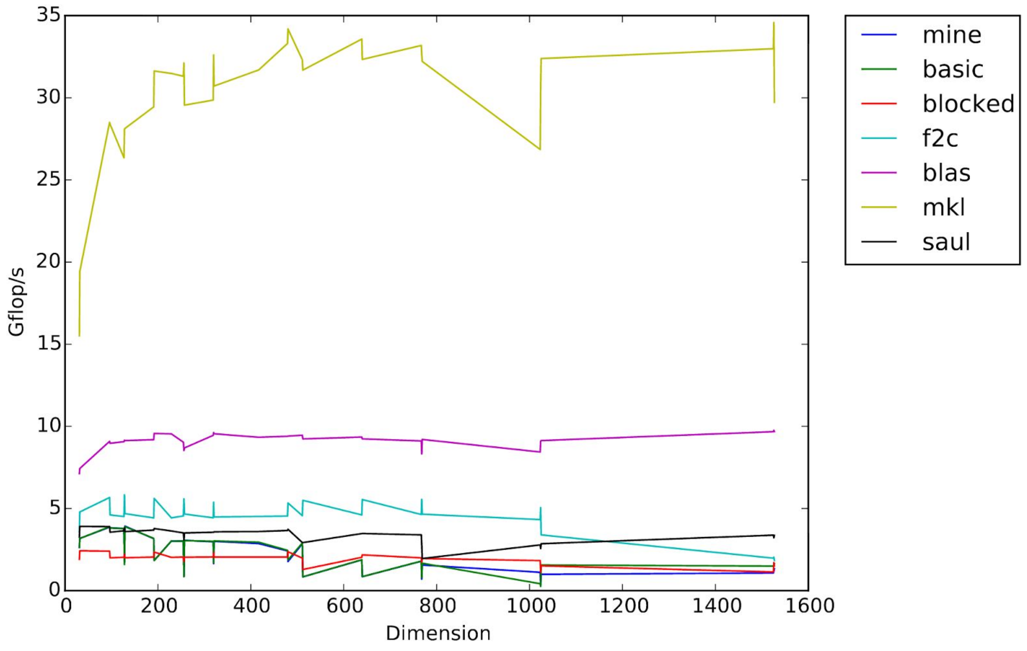
5. Initially, the idea of creating blocked code for the matrix multiplication code was to take advantage of the fact we understand that if we minimize the number of evictions from the cache, we will be able to reduce the performance loss of retrieving data from memory. However, we only have one size, which theoretically should be the size of the L3 cache. We extend this idea of hierarchically decomposing the algorithm at multiple levels such that a subset of the data will fit in the L2 cache, L1 cache, and the register file.

Results and conclusions about our methods

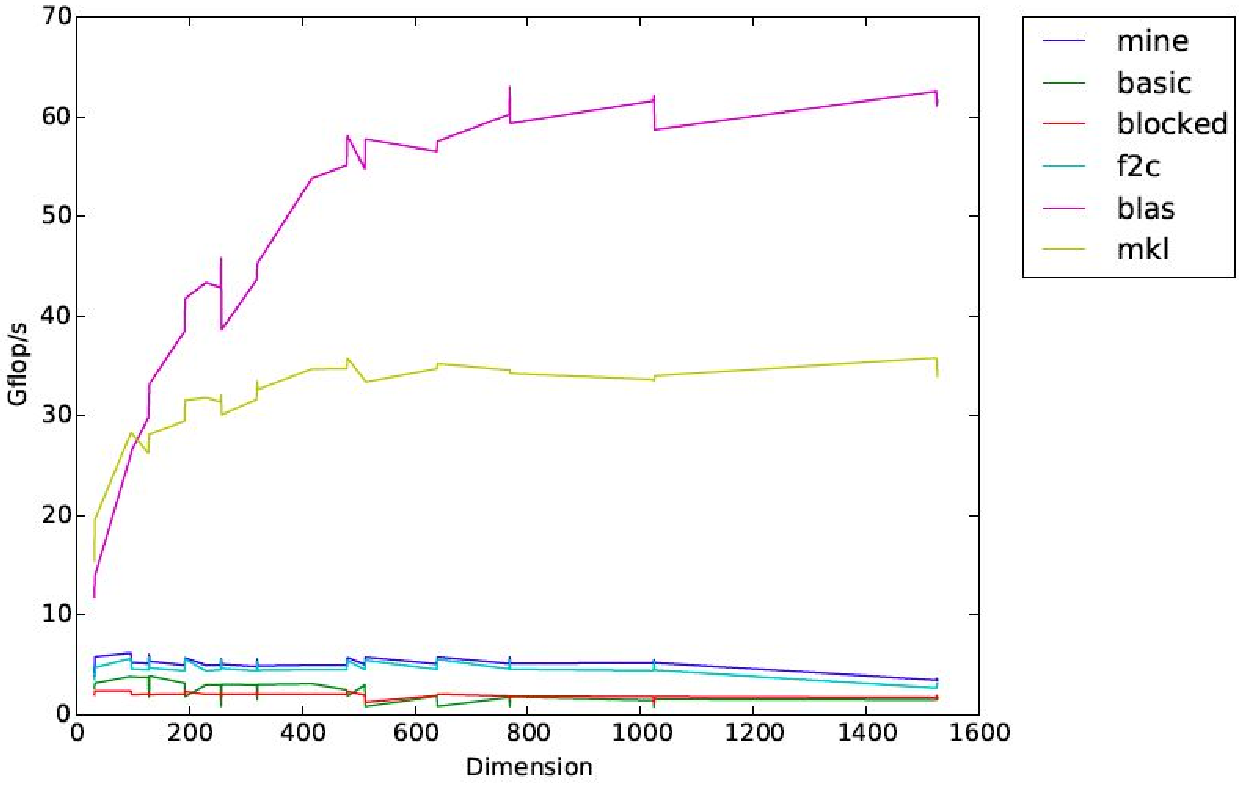
1. We can see in the plot that our method (black line-saul) does better than the naive blocked, basic and "mine" methods. However, the method has several conflict misses, and it is much slower than the mkl and blas methods. Since our method is better than the naive blocked approach, we can conclude that the new order of the loops improved the method.



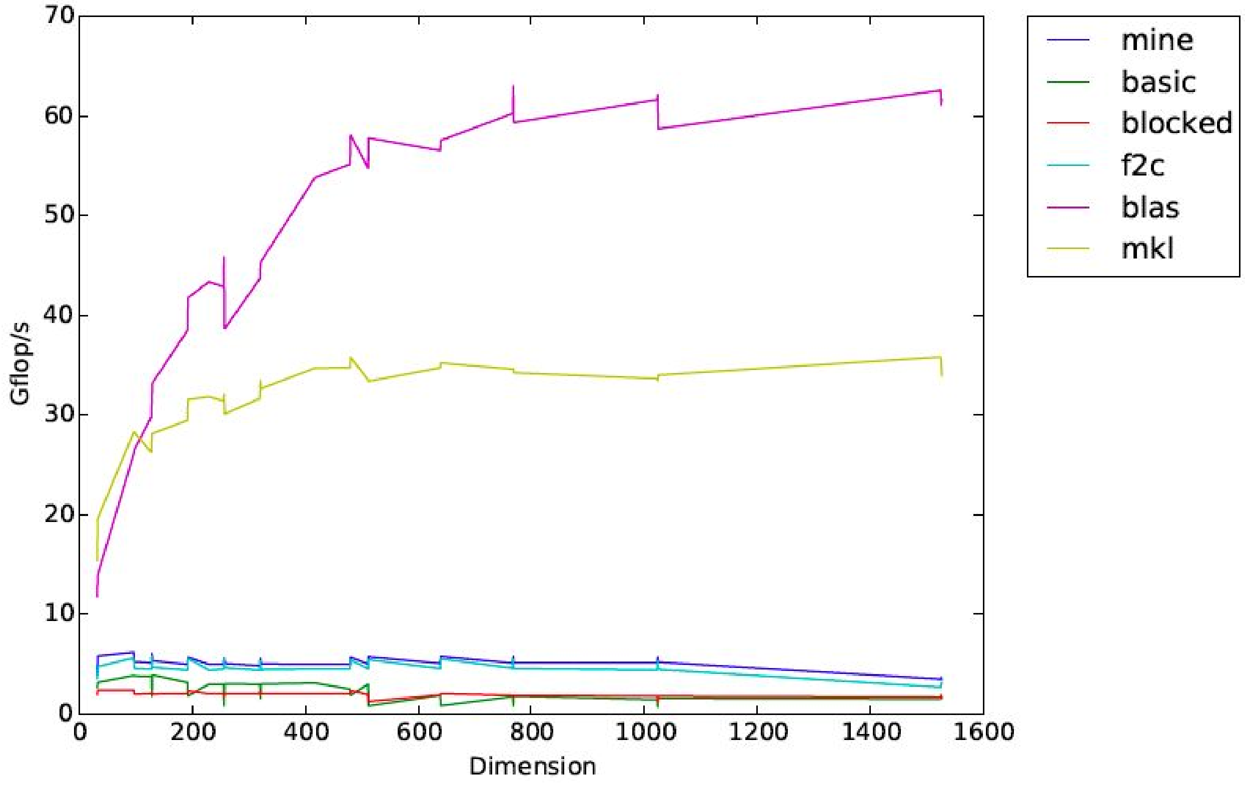
2. We can see in the plot that the method (black line) does better than the naive blocked, basic, “mine” methods, our previous method. When the dimension is larger than 1200, our method outperforms the f2c method. Although this method reduced the number of conflict misses, it still has several conflict misses, and it is much slower than the mkl and blas methods.



3. We observe that the “transpose A” approach (blue line, “mine”) improves the performance of the basic algorithm by approximately 2-3 times, despite the extra copying of the matrix. This suggests that the high stride of the basic method is a very large bottleneck and definitely needs to be addressed.



4. There seems to be no noticeable improvement over (3). My wager is that the code isn't complex enough to do any further meaningful compiler optimizations. The optimization flags would be worth looking into again after our code gets more complex.



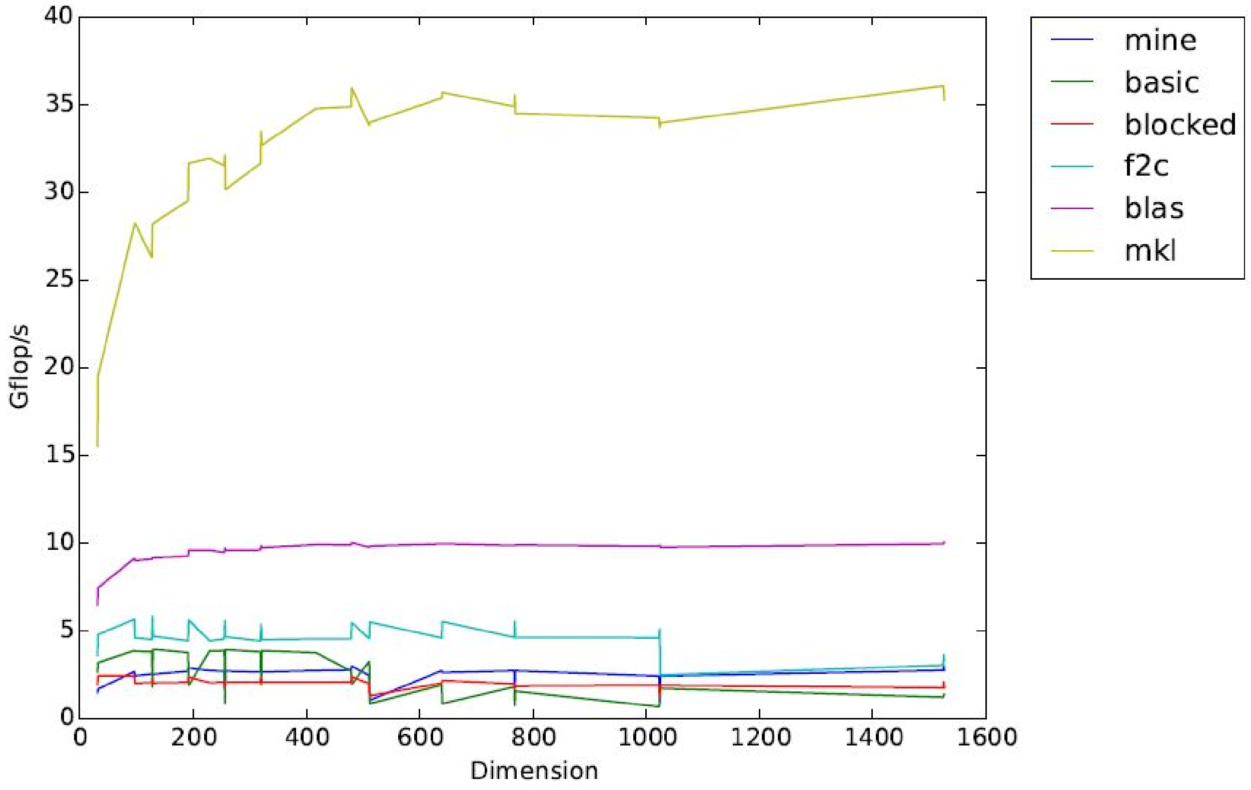
1. The implementation of a hierarchical block structure is not too hard to do by simply passing the previous memory level's ijk indices and the size of the cache to the next level. Next we must determine the block sizes at each cache. We look up the microarchitecture's cache sizes for our cluster (at http://www.cpu-world.com/CPUs/Xeon/Intel-Xeon%20E5-2620%20v3.html) and determine that the cache sizes for each core are 2.5MB for the L3 cache (shared for a total of 15MB), 256KB for the L2 cache, and 32 KB for the L1 cache. We approximate that the register is probably small at maybe 1.5KB.

   Next, we determine the largest blocks of the matrix multiply can be fit at each level of memory. We assume a write-allocate microarchitecture, which seems reasonable in most memory designs, so we must fit both input block matrices as well as the output (written) matrix which is 3 blocks of data. Furthermore, since we are interested 1 dimensional length of the square block, we take the square root of the size. We also scale down the block dimensions to be safe. So the maximum dimension to fit in each level would be:

   dimension = sqrt (<size of cache> / (<8B per double> * 3))

Memory Level	**Size**	**Maximum Block dimension**	**Used block dim**
L3	15 MB	310	288
L2	256 KB	100	96
L1	32 KB	36	32
reg_file	1.5KB	8	8

   The results are in the figure below, with the added code shown as the blue "mine" line. As we can see, there is some promise with some improvement over the basic and baseline-blocked code. Hopefully in conjunction with the previous techniques, we can see some multiplicative improvement.


Gflop/s
Dimension
mine
basic
blocked
f2c
blas
mkl
0
5
10
15
20
25
30
35
40
0
200
400
600
800
1000
1200
1400
1600