CS 5220 - Homework 1 - Matrix Multiplication

Final Report - Part 2

Please note that this is in addition to our previous report (submitted by Bob Chen)

Group Number: 05

Group Members:

Nimit Sohoni (nss66), Bob Chen (kc853), Amiraj Dhawan (ad867)

Optimizations used or attempted

1. Use blocks to divide the matrices into smaller submatrices that fit into cache. Experiment with different block sizes.
2. Copy A and B matrices into smaller block matrices and multiply these.
3. Transpose A into row-major form before passing it to a kernel function to compute the innermost loop of A*B.
4. Use different optimization flags (we end up using the icc compiler with -O3).
5. Using multilevel blocking for optimizing access from L1, L2 and L3 caches.
6. Using AVX vectorized commands such as fmadd, mul, etc.

Reasons for using the aforementioned Optimizations

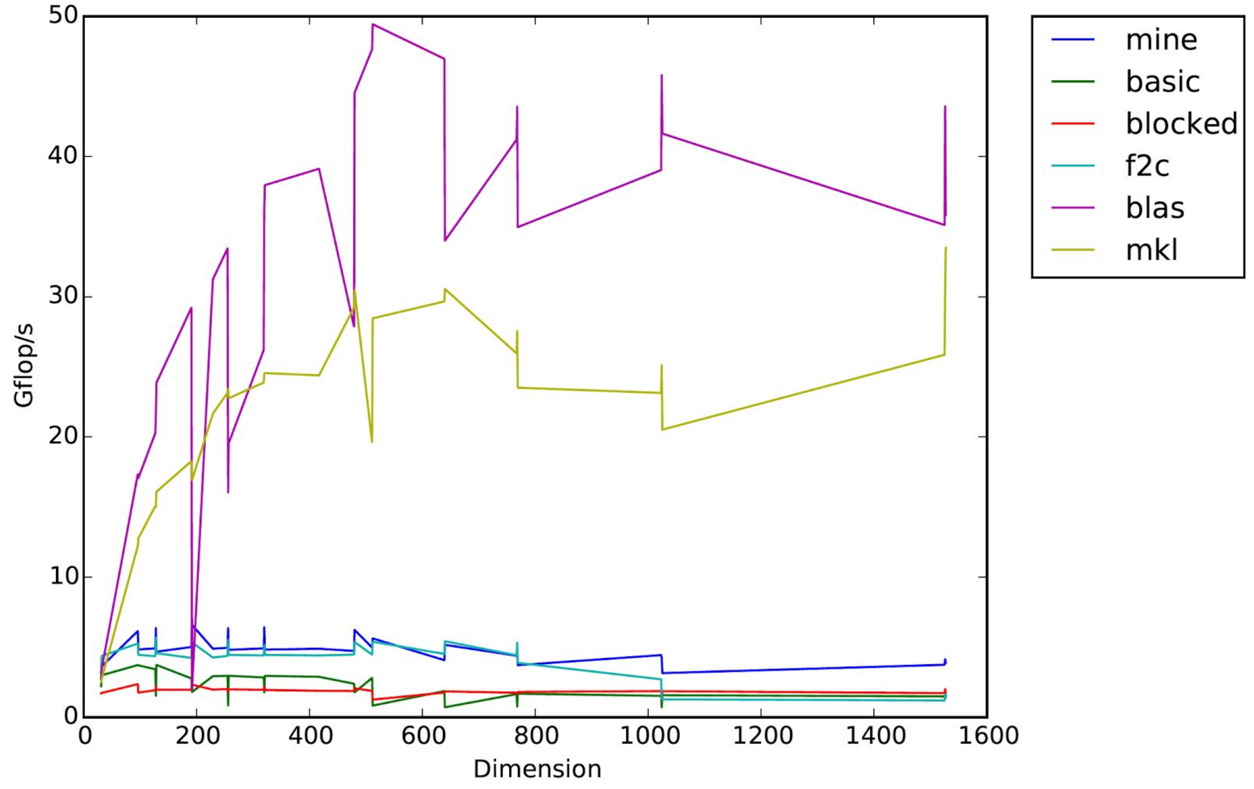
1. If we divide the big matrices into small enough submatrices, the program can fit the entire submatrices into the cache and therefore reduce cache misses.
2. Copying into smaller submatrices, instead of just treating different sections of each matrix as a block while it was still stored in the original array, would ensure that we could store multiple rows (or the entire submatrix) in the cache at a time. Thus, multiplying two submatrices should be fast since multiple rows and columns of each would be in the cache at a time, and with a small enough block size we should never have any cache misses at all during that operation after the initial loading.
3. If we use matrix A in its column-major form, its innermost loop will not be efficient, because if A is column major, we will fetch consecutive data in one column each time we compute A[i,k]*B[k,j] and that is wasteful - we will have a lot of cache misses. However, if we have converted A into A', we can take advantage of the spatial locality (since only one row of A will be needed at a time when computing one row of C), and therefore make the computation more efficient.
4. Different compiler options, such as icc with -O3 versus "plain" gcc, result in speeds differing by orders of magnitudes.
5. We also tried multiple (3) levels of blocking, where the outermost blocks would fit in the L3 Cache, the second level of blocks would fit in the L2 Cache, and the smallest level of

blocks would fit in the L1 Cache. The intuition behind this was that the L1 cache miss, and L2 cache miss are faster than an L3 cache miss. Once the L1 cache 3rd level block is done, the next 3rd-level block is loaded from the from L2 cache and results in L1 cache miss, instead of propagating it to the L3 cache miss or memory. Similarly, once the L2 cache second-level block is done, we load the next from the L1 cache.

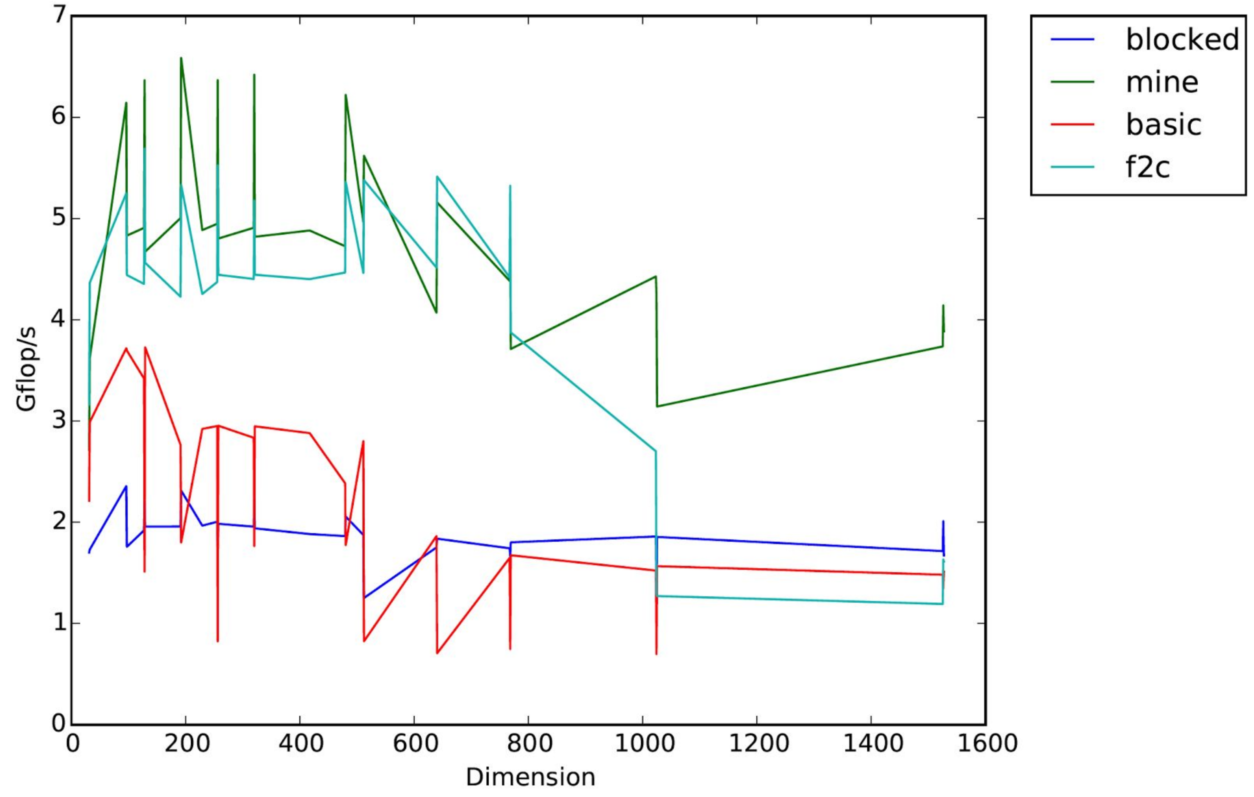
6. See Bob's report for a more detailed explanation of these.

Results

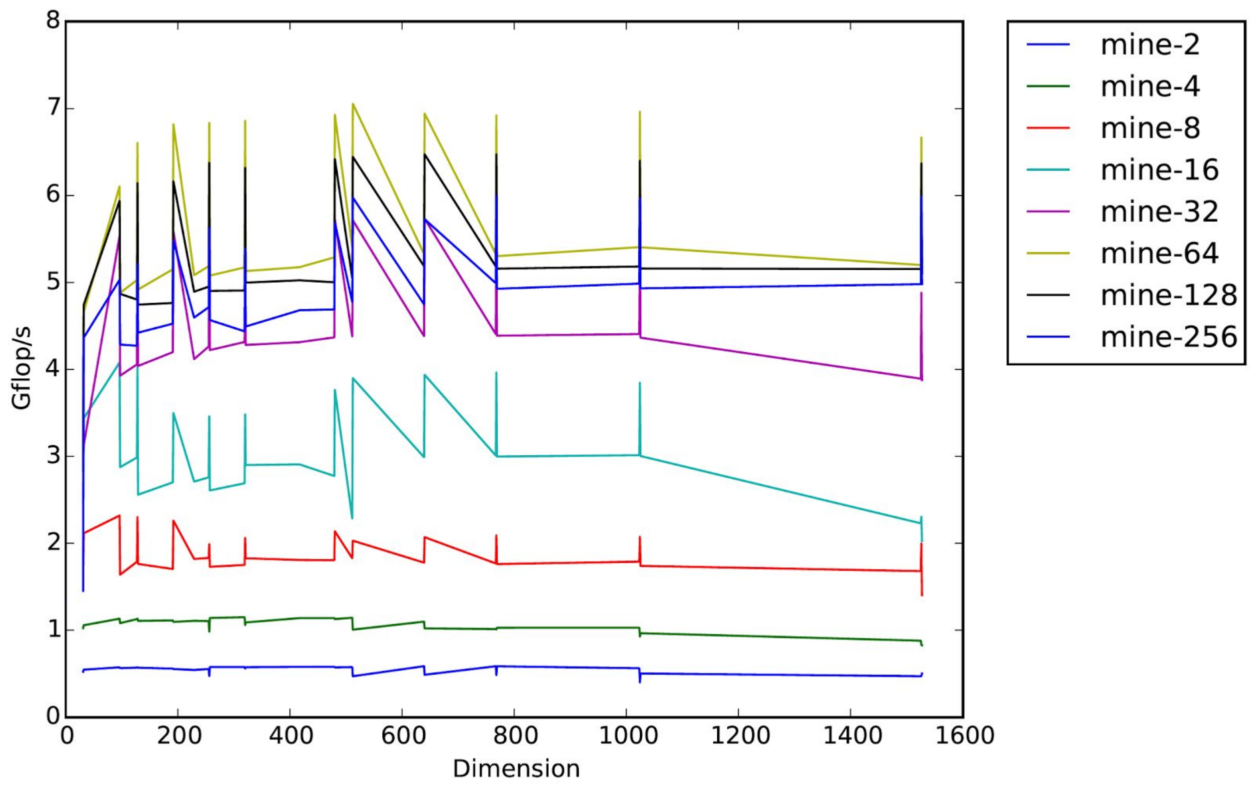
The plots from our old results follow. Unfortunately, the fastest working version we had was essentially the same code as what we submitted last time (see dgemm_mine.c in this repo). Most of the more advanced optimizations that we tried simply did not work very well.



This is the overall performance; note that the line by our code is above the f2c line.



The following is a plot of performance by block size:



Once again, the best performance was found with a block size of 64, followed closely by 128.

What didn’t and did work

1. A block size of 64 was confirmed to be the best for our algorithm after multiple different sizes were run with several repeated rounds of testing.
2. However, copying and multilevel blocking didn’t seem to help matters much. The additional overhead of copying matrices and the additional logic required may have overridden any speed gains resulting by the more efficient use of memory.
3. Transposing A did succeed in a much faster algorithm, because of the reduced cache misses. However, we are still nowhere near BLAS or MKL.
4. Usage of AVX commands did not speed up our code. In fact, it became significantly slower. One reason for this is that certain optimization flags (for instance, -O3), when used in conjunction with AVX, introduced numerical errors into the code that were above the required thresholds. We are still not sure exactly why this is, other than that the compiler was over-aggressive in its optimization. However, even when error bounds checking was turned off, our code with AVX was much slower. The fact that AVX command only work on 256-bit “vectors” made it hard to figure out how to best use those commands for the much larger vector multiplications that we have to do.
5. See Bob’s report for additional explanations.