CS 5220 - Homework 1 - Matrix Multiplication

Group Number: 5

Group Members:

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

Optimizations used or attempted

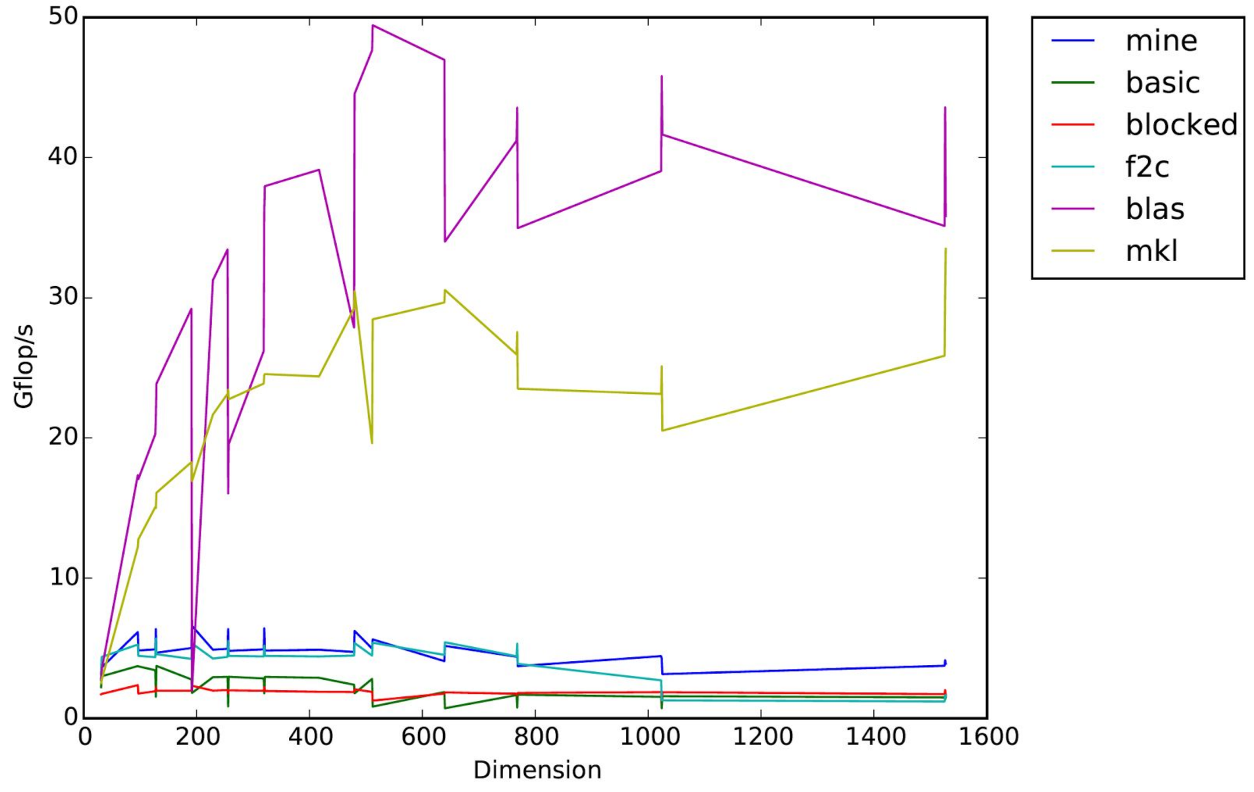
1. Use blocks to divide the matrices into smaller submatrices that fit into cache. Experiment with different block sizes.
2. Copy matrix A and save it in row-major form before passing it to a kernel function to compute the innermost loop of A*B.
3. Implement a faster kernel function.
4. Use different optimization flags.
5. Using multilevel blocking for optimizing access from L1, L2 and L3 cache

Reasons for using the aforementioned Optimizations

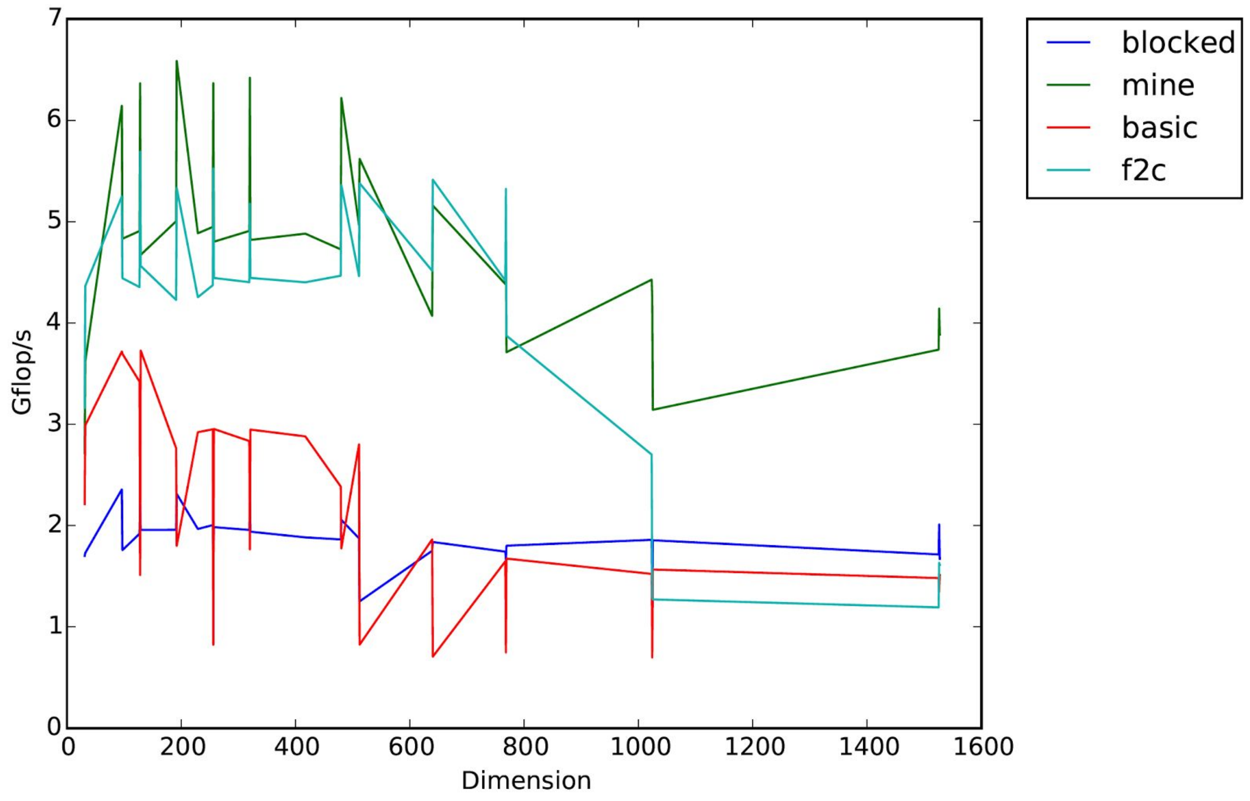
1. By dividing the big matrix into smaller submatrices, the program can fit them into cache and reduce communication between CPU and memory. Once the matrices are loaded into the cache, the computation is essentially matrix-matrix multiplication and it has better operational intensity than matrix-vector multiplication or simply vector-vector multiplication. By changing the block size, we can find a sweet point where all the submatrices can fit into the cache.
2. 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.
3. Because the innermost loop is the most time-consuming part of the program, it is reasonable to deal with this first to get a sufficiently fast program.
4. At this stage, different compilers and flags give us a rough idea what can be achieved with an optimized version of the code. It will be more useful later when we have got a good kernel function and arrangement for memory usage.
5. We decided to do multi-level blocking of the matrices to faster the memory access patterns depending on the L1, L2 and L3 level cache sizes. The plan was to do 3 levels of blocking, where in the first level of blocks were to be of size which can fit in L3 Cache, the second level of blocks (made within the first level blocks) which could fit L2 Cache, and the final level of blocks (made within the second level blocks) which could fit L1 Cache. The intuition behind this implementation plan 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 block is picked up from L2 cache and results in L1 cache miss in place of propagating it to the L3 cache miss. We are yet to complete the implementation of this as of now and cannot comment about the performance boost/drop.

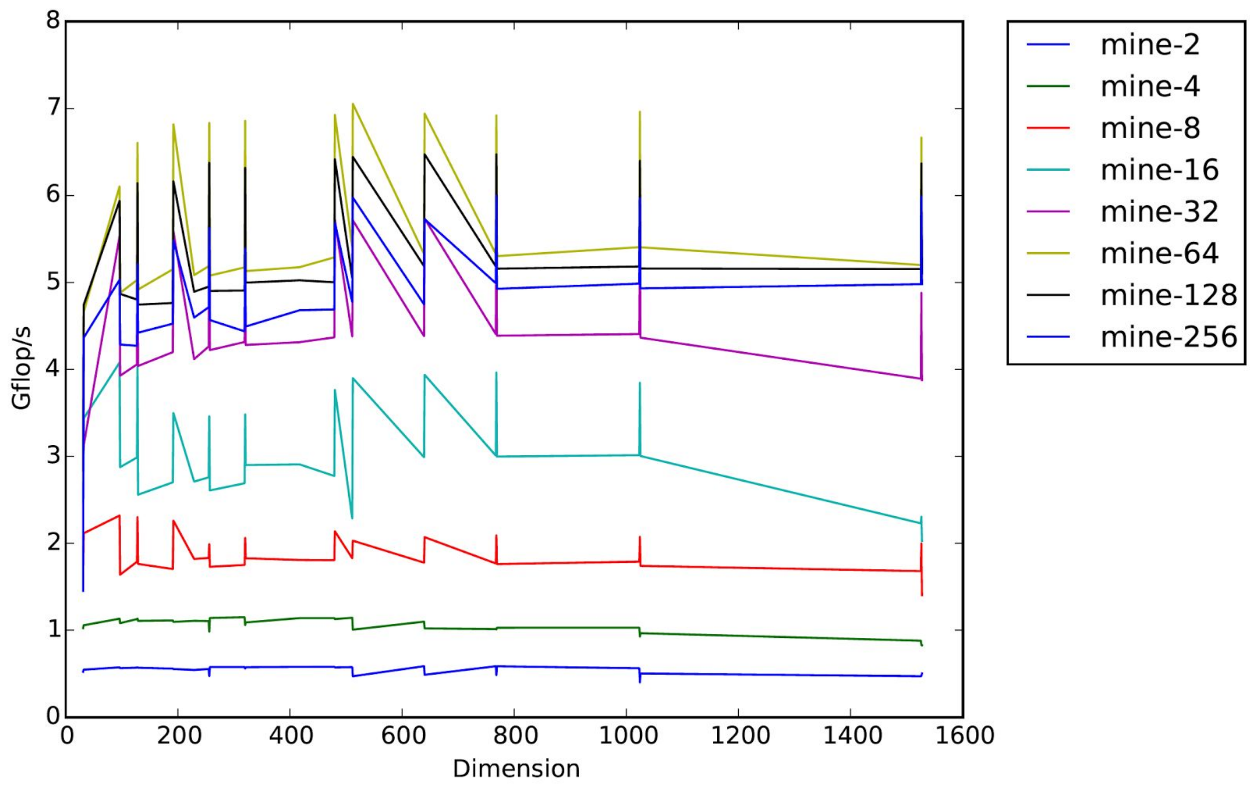
Results



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



We started from the dgemm_blocked.c code and implemented the transpose of A and a faster kernel.



This result shows that different block sizes will affect the performance greatly. As the block goes up, the speed goes up and then goes down after passing a “sweet spot”, which is 64 is our case. In fact, we tried block sizes up to 1365, but 64 seemed to do the best. The reason we at first expected 1536 to do the best is that the L1 cache size is 32 kilobytes - i.e. 32,768 bytes - which means that it can fit 4,096 doubles, which are 8-bytes. 4096 / 3 ≈ 1365.3 > 1365, so 1365 is the maximum block size such that we can simultaneously fit a full row block of A, column block of B, and row block of C in the L1 cache.

The results for different flags are not very meaningful because it is done before the kernel was fully implemented. We will continue working on that.

What didn’t and did work

1. It is clear that block size has a significant influence on the performance. Now we only have results with a few block sizes, but we can clearly see the difference it makes. What we can do is probably use another layer of blocking to further divide the matrices so that we can make use of L1 and L2 cache as well as the register.
2. Before we make changes to the storage of A, the performance will stay below the basic implementation. It is clear that the cost to transpose A is well paid off in saving the memory traffic in the kernel function.
3. After we have a transposed A, the kernel loop has a good improvement in its speed. But we haven’t fully explored its full potential. We will continue this work by further optimizing

this function. Eventually we want a function that deals with small submatrices (maybe 8*8) and does the job with a speed comparable to optimized library.

4. We attempted usage of different flags too early and saw little difference. However, we can achieve more after implementing the above mentioned changes. We will also look into AVX and other methods to improve vectorization of the code by the compiler.