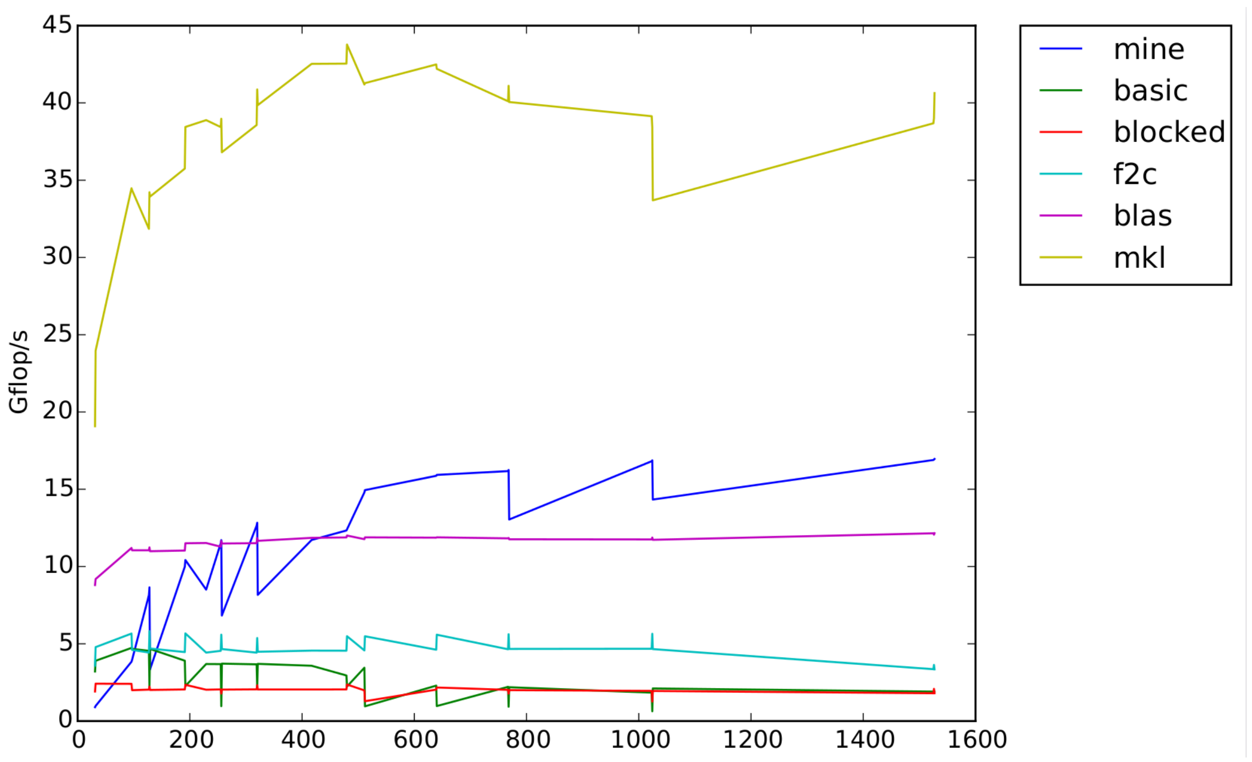
CS 5220 – Project 1

Eric Gao – emg222
Junteng Jia – jj585



We wrote this code from scratch and we beat blas :)

Contents

Our approach

We made six major improvements:

1. Block multiplication.
2. Copy optimization.
3. Vectorization.
4. Data alignment.
5. Further blocking
6. Compiler optimization

Block multiplication

We split the matrices into several blocks. We chose the blocks to be size 32, which means that each block takes up 8KB ($32^2 * 8$). This means we can fit all three blocks of the three matrices we care about into the L1 cache of the Phi processors since the L1 cache is 32KB. As a result, we can write a very fast kernel that allows us to take advantage of the fact that we know the size of the matrix and the fact that everything is in the L1 cache. This way, we can unroll our loops so we don't have to make unnecessary branching operations. We can then use this fast kernel to multiply blocks together so that we get the final result.

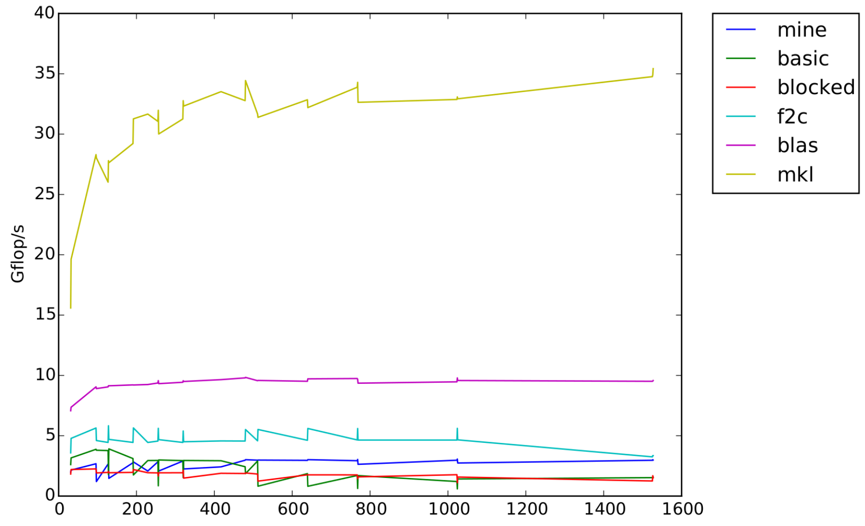


Figure 1: After block multiplication

Copy Optimization

We arranged the data so that within each block of A, the entries are row major and the blocks themselves are arranged in row major order. For B, the entries in the block are row major and the blocks themselve are arranged in column major order. We arrange them in this order so that we can have a stride of 1 for our kernel. This is to maximize spatial locality.

Also, by doing this, we can handle fringe cases well. We make sure that matrices that we are multiplying have a leading dimension that is a multiple of our block size. We just add zeros on the edges to ensure this.

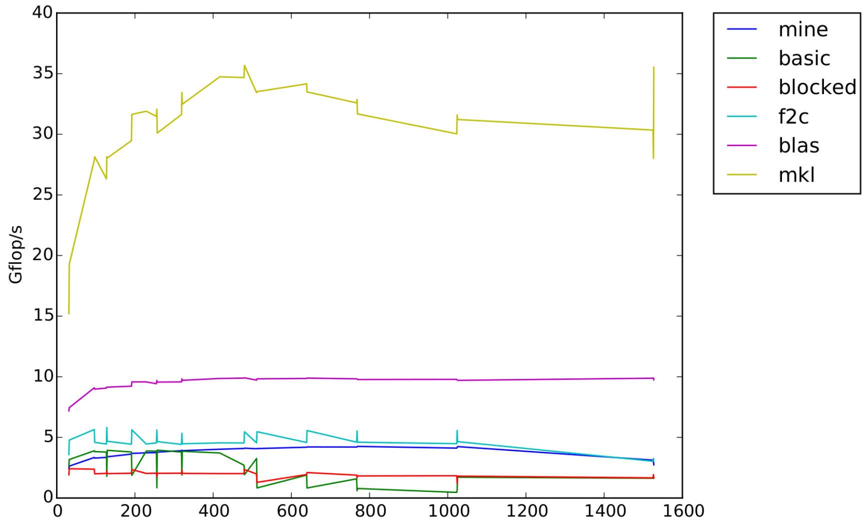


Figure 2: After copy optimization

Vectorization

We aligned our data to 64 bytes in memory and we multiply the blocks in our matrices in such an order so that we can take advantage of Vector FMAs. Specifically, we start with the (0, 0) element of a block in A. We multiply that by the first row of a block in B. We add that to the first row of a block in C. We then iterate across the 0th row of A and iterate over the rows of the block in B. Then our outer loop iterates over the rows of A. This way, on our inner loop, we can take advantage of vector operations. When we compiled our code with the flags to show us whether we have vectorization, we obtained the following output:

```
LOOP BEGIN at dgemm_mine.c(121,13) inlined into dgemm_mine.c(199,17)
remark #15388: vectorization support: reference C_row.286 has aligned access
remark #15388: vectorization support: reference C_row.286 has aligned access
remark #15388: vectorization support: reference B_row.288 has aligned access
remark #15399: vectorization support: unroll factor set to 8
remark #15300: LOOP WAS VECTORIZED
remark #15448: unmasked aligned unit stride loads: 2
remark #15449: unmasked aligned unit stride stores: 1
remark #15475: --- begin vector loop cost summary ---
remark #15476: scalar loop cost: 13
remark #15477: vector loop cost: 4.000
remark #15478: estimated potential speedup: 3.220
remark #15479: lightweight vector operations: 6
remark #15480: medium-overhead vector operations: 1
remark #15488: --- end vector loop cost summary ---
LOOP END
```

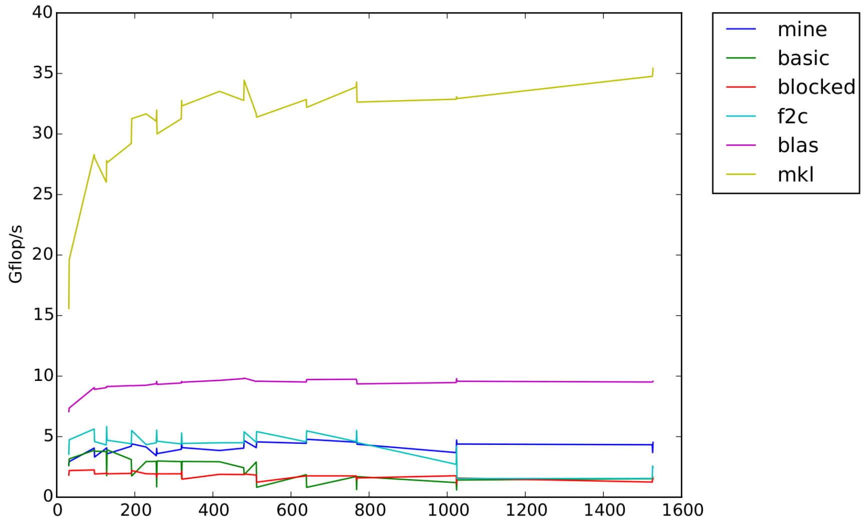


Figure 3: After vectoriztion

Data alignment

We used the functions _mm_malloc() and _mm_free() so that we can guarantee that our matrices are aligned properly in memory. Specifically, we ensure that they are aligned to 64 bytes. We do this to ensure that we can take advantage of vector operations.

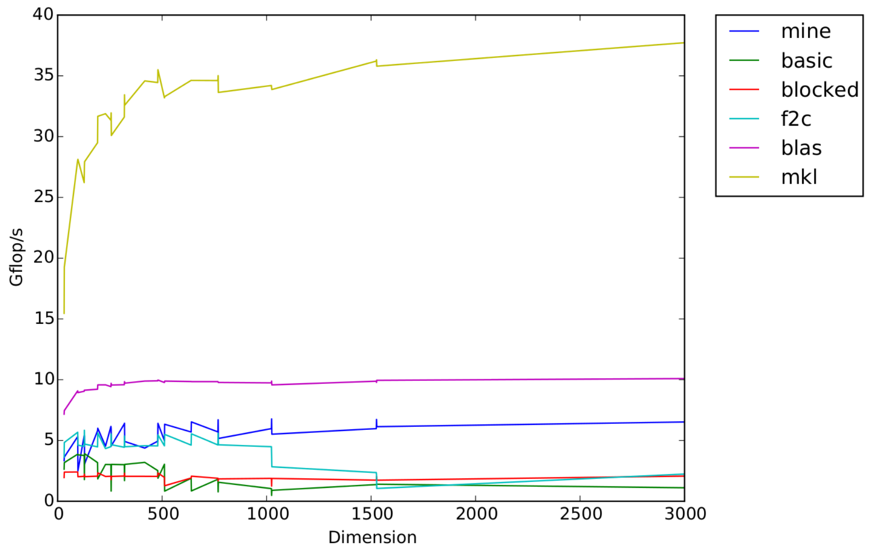


Figure 4: After data alignment

Further blocking

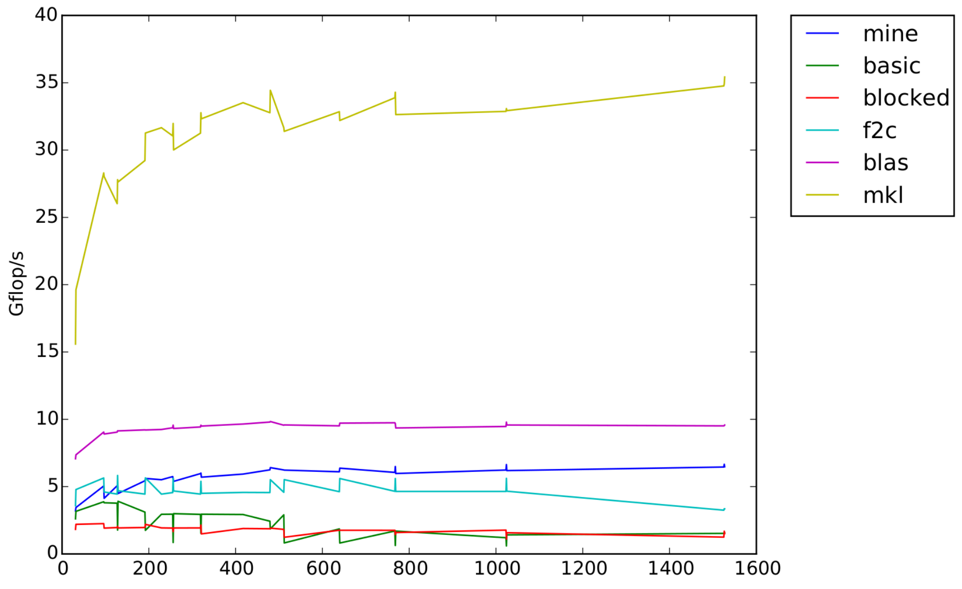


Figure 5: After further blocking

Compiler optimization

To fully utilize the power of intel compiler, we used the following compiler flag:

```
-O3 -fast -axCORE-AVX2 -unroll-aggressive -mtune=core-avx2 -ipo -xHost
```

- The 03 flag tells the optimizer to perform aggressive optimizations.
- The unroll-aggressive is a intel compiler option that tells the compiler to unroll the loops that the compiler believes would lead to a speed up.
- The ipo flags tells the compiler to do interprocedure optimizations, so this means that the compiler is going to attempt to reduce the amount of time that exists when functions are in different files.
- The xHost flag applies a high level vectorization to the code.
- The axCORE-AVX2 and mtune=core-avx2 flags tell the compiler to take advantage of the fact that our CPU has access to the AVX2 vectorization operations, and therefore we want our code to be tuned to this AVX2 instruction set.

Using compiler optimization is what we learned from group 20 and group 24 during peer review.

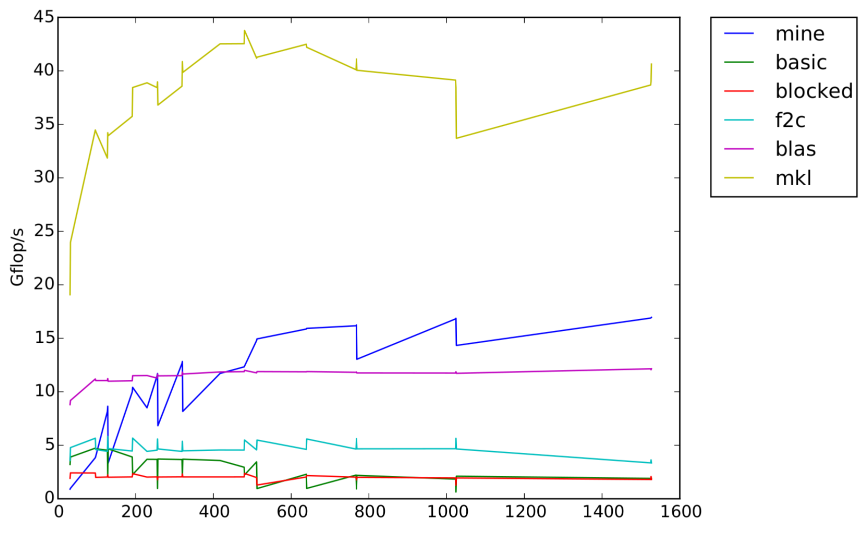


Figure 6: After compiler optimization

L2 blocking

We then implemented a multi-level blocking strategy. We first divided the matrices into blocks so that each block could fit into the L1 cache. We then created larger blocks that contained several smaller blocks and performed a block matrix multiply using those larger blocks. We created the larger blocks that contained larger submatrices that could fit into the L2 cache. Specifically, we had 4 smaller blocks in every larger block, resulting in a matrix of 64 x 64 doubles if our smaller blocks were 32 x 32 doubles. The result of this had a minor improvement on performance for larger matrices. The overhead of the copy optimization resulted in lower FLOP rates for smaller matrices. The results are seen below:

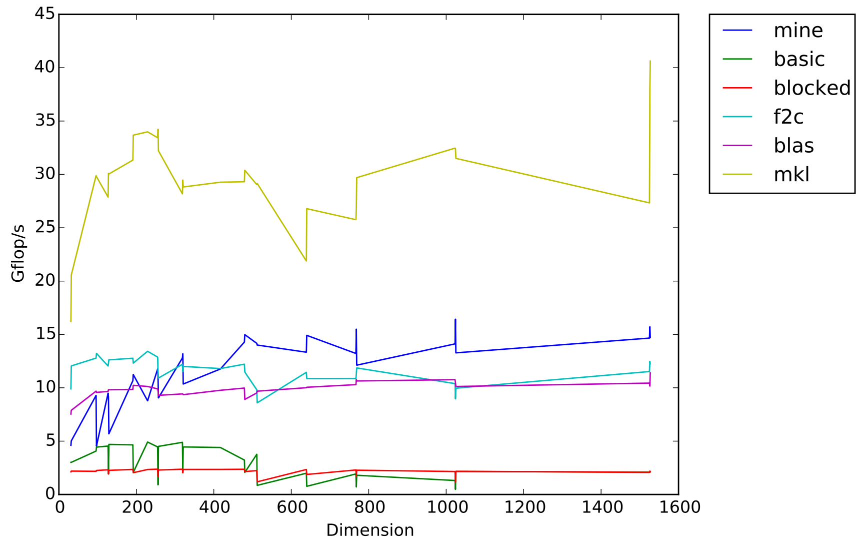


Figure 7: Without L2 blocking.

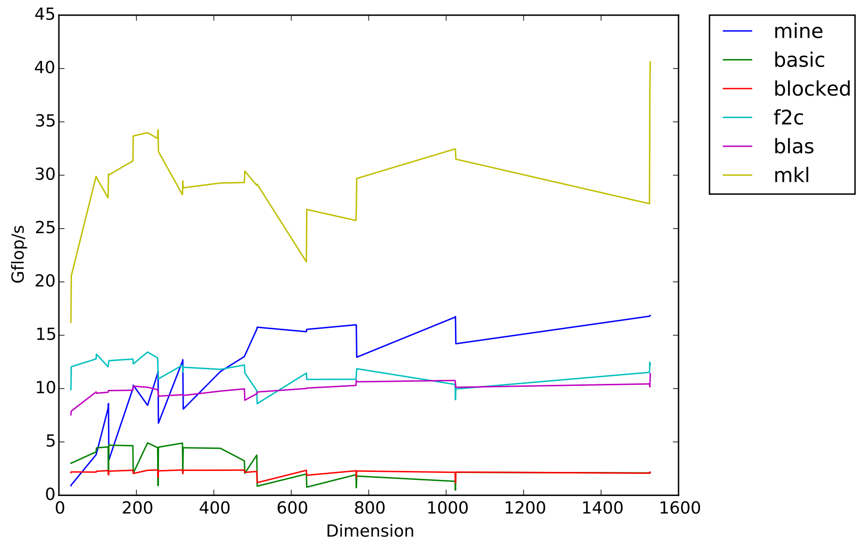


Figure 8: With L2 blocking. Notice that the FLOP rate is marginally higher for larger matrices but lower for smaller matrices.

Looking at this, we can try just branching based on the size of the input matrix, which isn't too expensive since it is being done outside of all loops. When we do this, we get the following result. However, the compiler cannot optimize a lot of the code out if we are doing this conditional branching. This causes the overall result to be worse.

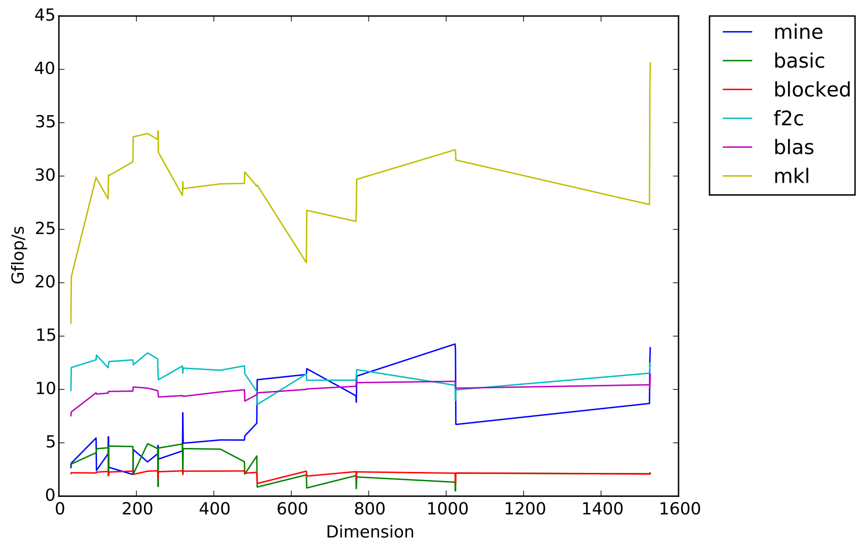


Figure 9: With L2 blocking but branching to use L2 blocking only if the size of the leading dimension of the matrix is greater than 384.

Summary

Through this project, we learned:

- Library functions such as mkl is very well optimized, we should use those libraries as building blocks when we are coding.
- Whether we can fully utilize vector ALU will decide the performance of our serial code.
- Thinking about the data transfer in cache and register while we are coding can gain us a lot performance.
- Intel compiler is amazingly good at optimization, we should take advantage of that.
- Peer assessment is interesting and useful. Its a combination of completing and collaborating.

Our results compared to other matrix multiplication implementation. Our code is comparable to the blas implementation (with out optimization) for smaller matrices and runs faster for larger matrices.