# HW 1: Tuning on a Single Core

**Group 13**: Taejoon Song, Marc Aurele Gilles, Gregory Granito
{ts693, mtg79, gdg38}@cornell.edu

## 1. Introduction

In this assignment, we were attempting to optimize the single core matrix multiplication algorithm. We were given a variety of implementations that ranged from very naive to professionally tuned. We started with a blocked implementation of matmul and attempted to make changes to achieve a higher performance.

We attempted four different optimizations in order to improve performance.

1. Optimizing the ordering of the loops
2. Optimizing the size of the blocks
3. Optimizing the compilation settings
4. Using Block indexing to achieve better cache utility
5. Tuning the algorithm to have a higher vectorizability
6. Minimizing the number of branches that exist in our algorithm

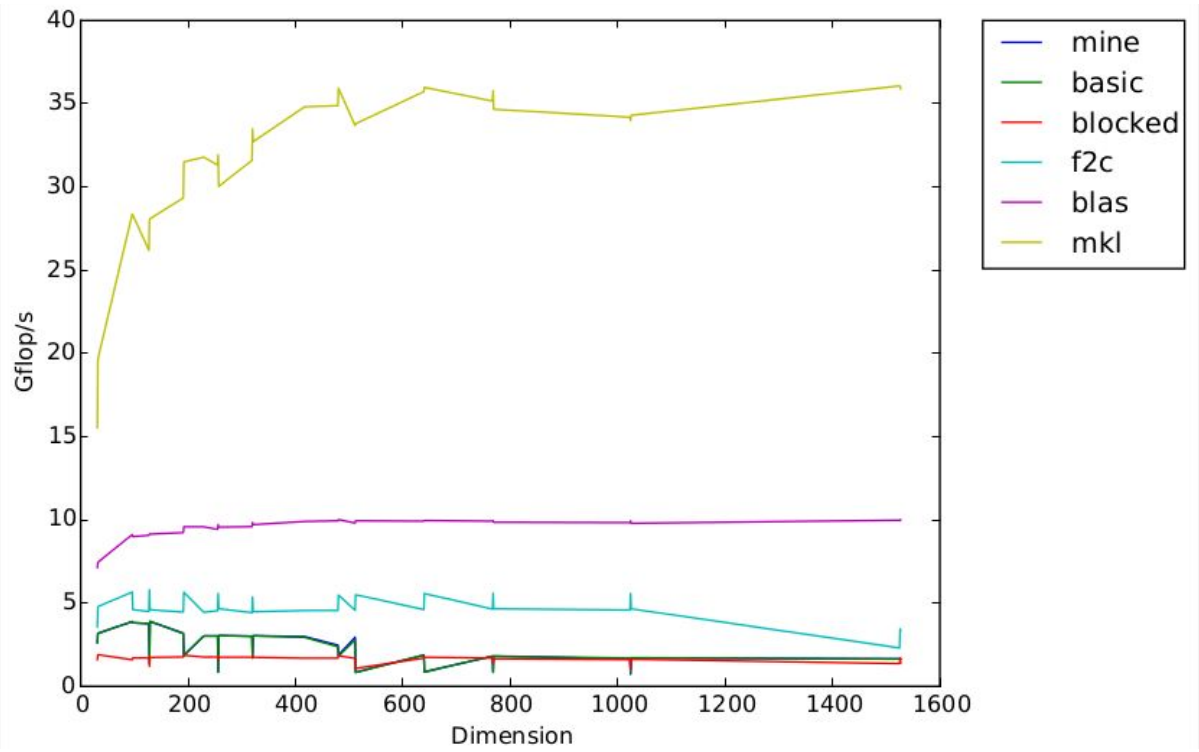
We focussed on these optimizations because the matrix multiplication task seems to be memory bound. So improvements that make better use of the cache are likely to have a larger impact on the overall performance.

By doing these optimizations, we were able to get our performance up to around 8.3 GFLOP/s.

## 2. Optimizations

First we ran the suite of matrix multiplication algorithms as is to establish the baseline performances.

**Figure 1. Baseline**

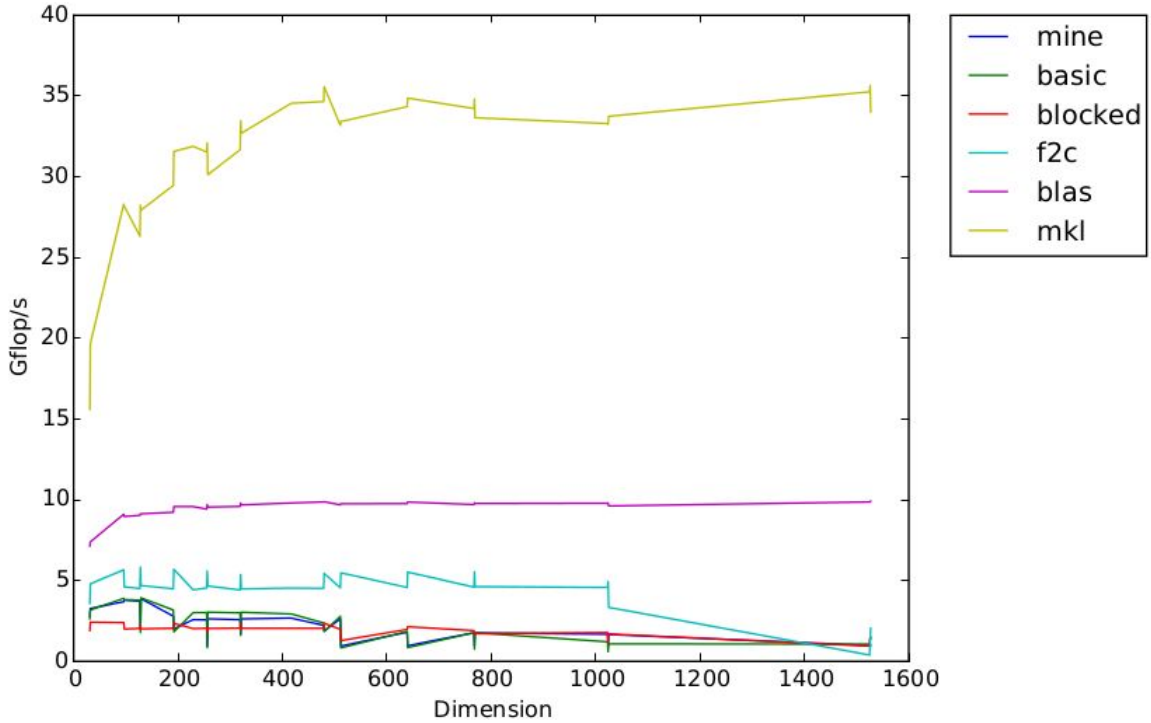



### 2.1. Loop Order

As an initial pass at optimization, we attempted to improve performance of the basic matmul algorithm by varying the ordering of the loops. That is, we swapped the ordering of the loops and observed whether this yielded improved performance. When we

swapped the middle and outer loops we found that performance did not change substantially, and if it did the results were worse than the original implementation.

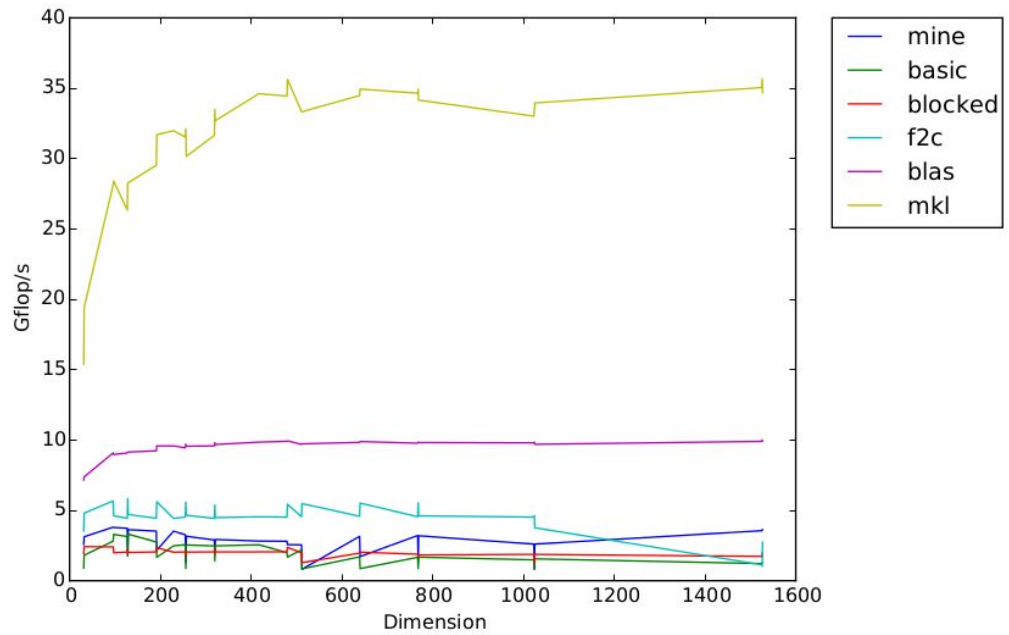
**Figure 2. Results of Swapping Middle and Outer Loops**




## 2.2. Blocking

Using a blocking scheme instead of the naive vector dot product implementation increases cache hits as for large matrices. Indeed if the dimension of the matrix is large, a single column or row of the matrix will not fit into cache, therefore each column/row has to be fetched from main memory every single time it is used. We figured out the optimal matrix size that would fit into each level of cache, and adjusted cache size accordingly. Since we want a copy of a block of A,B and C to fit into each level of cache and each double is 8 bit, the theoretical maximum dimension is $\sqrt{S/(3\times 8)}$. , we then choose the biggest multiply number which has a factor of 8 as the “theoretical best”. We used this “theoretical best” as a guideline, and tried a few values around it until we find an optimal value. We found that setting the dimension of the smaller block size (which fit into L1) to 16 doubles , and the intermediate block size (which fit into L2) to 126 provided good results.

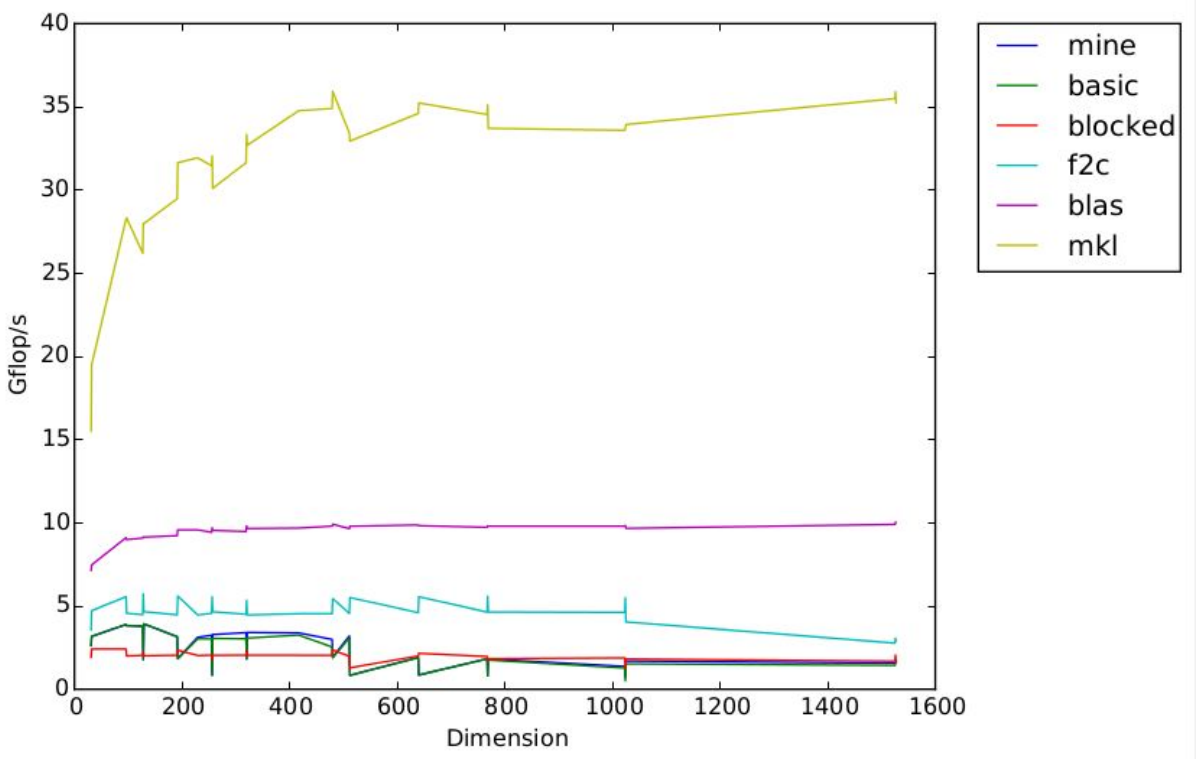
**Figure 3. Block Size of 128**



## 2.3. Compiler Optimization

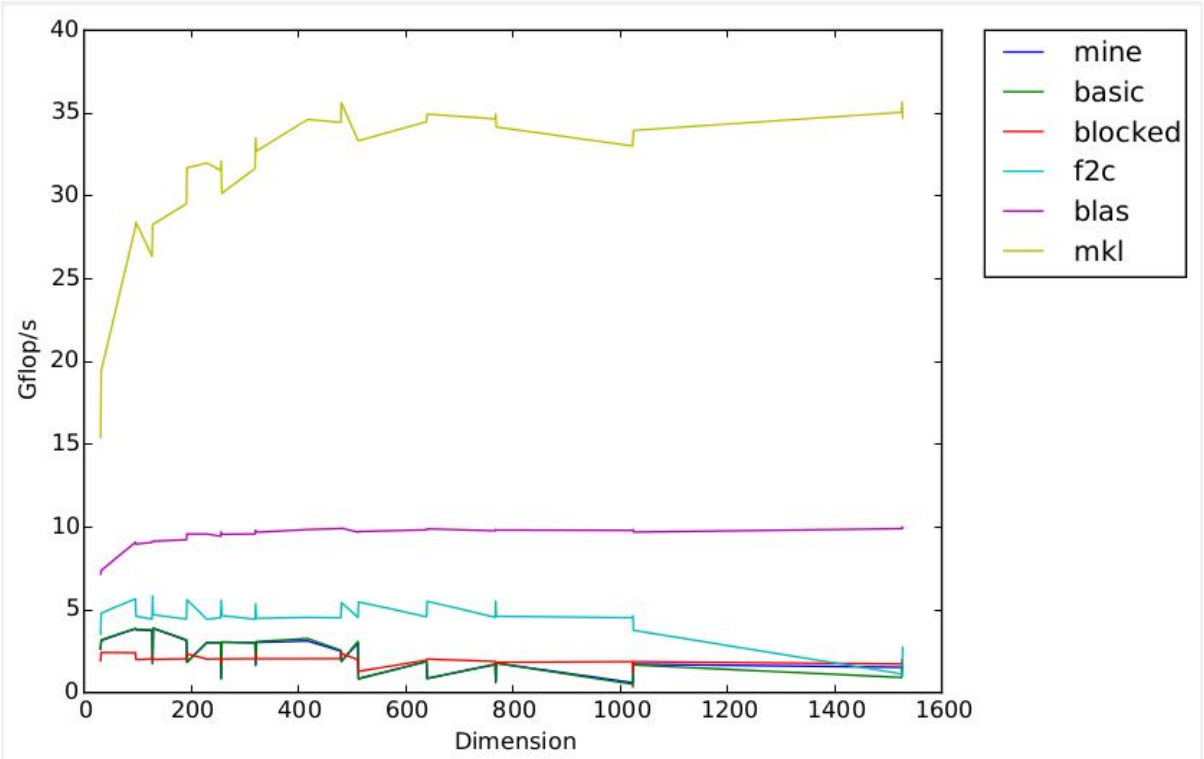
To see if we could achieve better results, we attempted to determine the effect that the compiler settings has on the performance. For a sense of the effect that the optimizations a compiler makes, we started by removing all compiler flags.

**Figure 4. Without the -O3 flag**



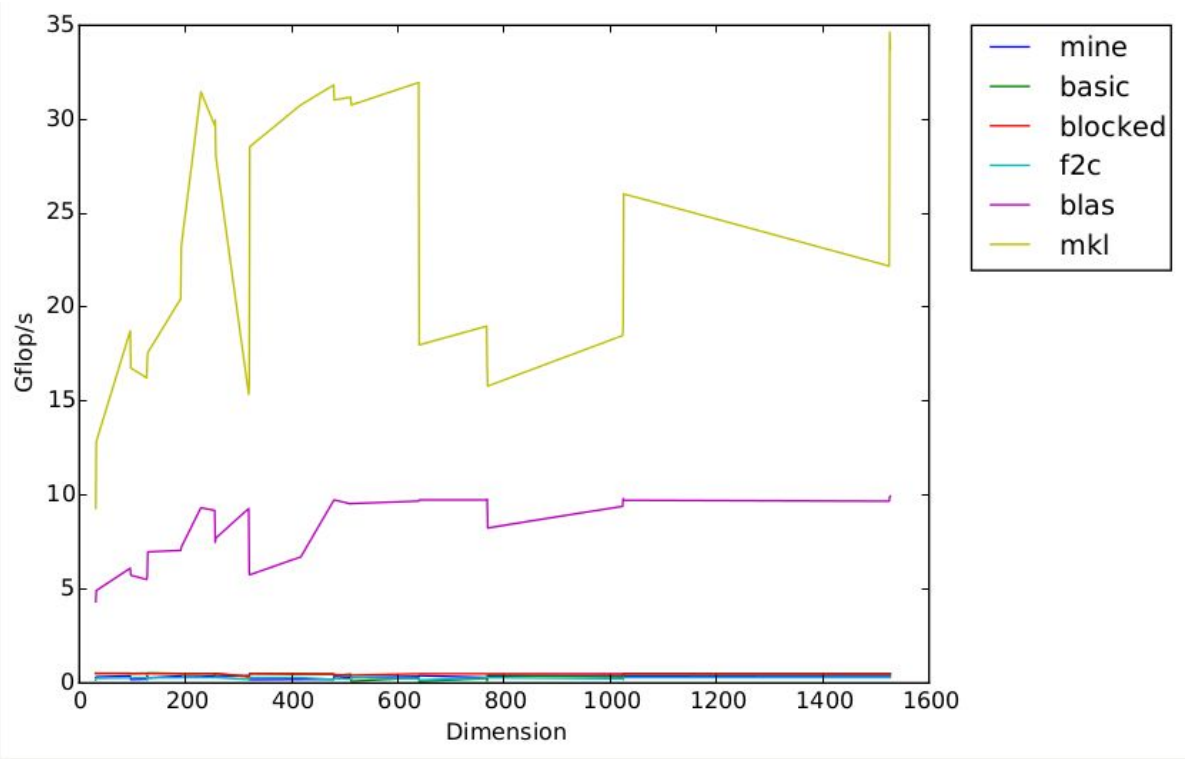

Then we turned on all of the compiler flags to see how much of an improvement we were able to achieve.

**Figure 5. With flags: -O3 -ftree-vectorize -funroll-loops -ffast-math**



When we turn off the optimization option to -O0, we can see there are huge degradation on performance.

**Figure 6. With flags: -O0 (Turn off O flag)**




Overall, changing the compiler settings did not seem to have a big impact on the performance of our matrix multiplication algorithm, as long as we stick with O3 level of optimization options.

## 2.4. Block Indexing

In order to achieve better performance using the block based matrix multiplication algorithm, we modified the indexing of the matrices from column indexing to block indexing .The block indexing is an example of copy optimization. Indeed, before we start computing the matrix multiplication, we allocate a new space in memory to store a copy of each of the matrix with a block indexing. The new copy of the matrix is padded in a way that makes the size of the copy of the matrix divisible by the chosen size for the L1 blocks. This is done in order to avoid having to handle special cases in the kernel, discussed in the next section. The result matrix C is reindexed back to the original indexing after the computation is over. Ideally, with the new block indexing scheme there would be better cache utilization and this would result in a higher overall performance. Below is a figure illustrating how we reindexed matrices (note that we did not reindex B in the same way as A and C):

**Illustration of block indexing on a 4x4 matrix**

Default indexing

| | | | |
|---|---|---|---|
| 0 | 4 | 8 | 12 |
| 1 | 5 | 9 | 13 |
| 2 | 6 | 10 | 14 |
| 3 | 7 | 11 | 15 |

Block indexing of A and C

| | | | |
|---|---|---|---|
| 0 | 1 | 8 | 9 |
| 2 | 3 | 10 | 11 |
| 4 | 5 | 12 | 13 |
| 6 | 7 | 14 | 15 |

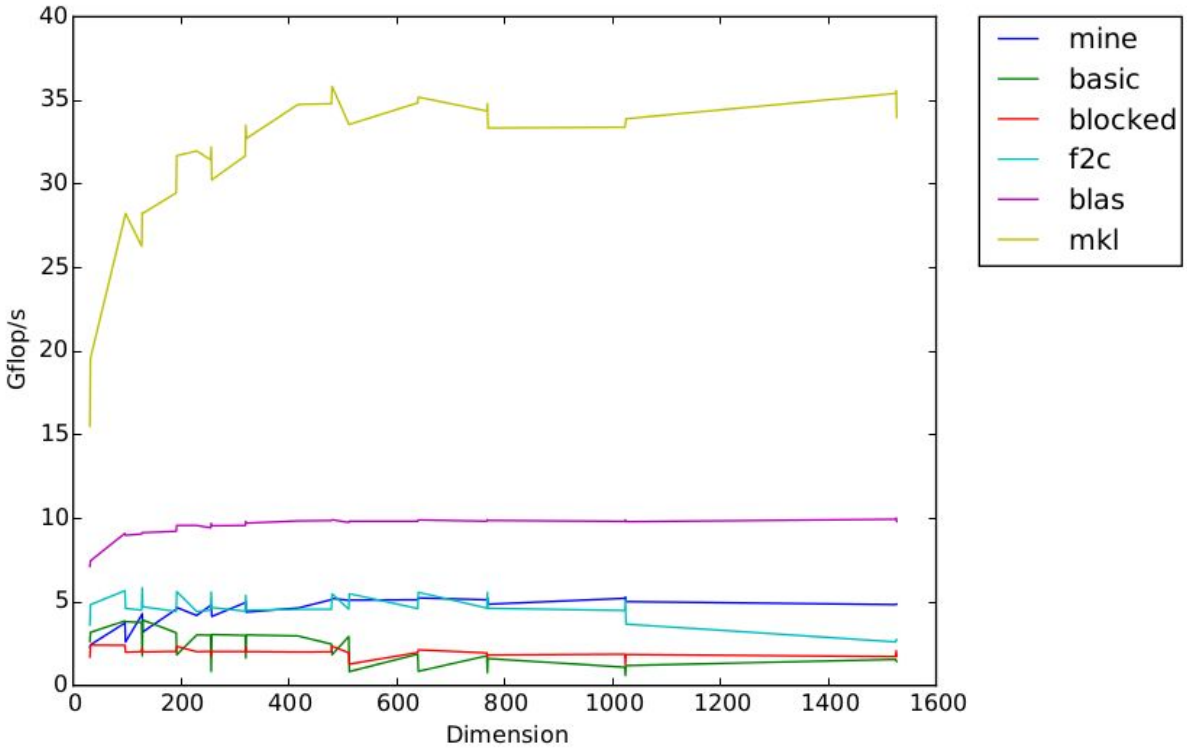
Block indexing of B

| | | | |
|---|---|---|---|
| 0 | 2 | 8 | 10 |
| 1 | 3 | 9 | 11 |
| 4 | 6 | 12 | 14 |
| 5 | 7 | 13 | 15 |

Colors represent the different blocks of the block-indexed matrices

With this modification we were able to increase performance. We found that 16 was the optimal block size under this scheme by manually tuning the value. After optimization, we found that our overall results were substantially improved.

**Figure 7. Block Based Indexing with Block Size of 16**




Overall doing the block indexing scheme and tuning the block size seemed to substantially improve the performance.

## 2.5. Tuning for Vectorization

In order to take advantage of the vector register of the compute nodes, we attempted to ease the vectorization of our routine.

### 2.5.1. Aligning memory

Following the directives of a paper by Intel [1], we attempted to align the data to assist vectorization. [1] provides a brief description of data alignment and its goals is: “Data alignment is a method to force the compiler to create data objects in memory on specific byte boundaries. This is done to increase efficiency of data loads and stores to and from the processor. Without going into great detail, processors are designed to efficiently move data when that data can be moved to and from memory addresses that are on specific byte boundaries.”

This tuning consisted of two steps:

1. When we declare a new place in memory to allocate to the “block indexed” array, we allocate an “aligned” array, this is done using the statement:

   double *restrict bA= (double*) _mm_malloc(pad_size*pad_size*sizeof(double),64);

   (note we also “restrict” the pointer to the new place in memory to let the compiler know that this is the only pointer pointing to the array)
2. Tell the compiler that the memory accesses are aligned, this is done using statements __assume(index_offset%8==0);inside of our kernel. Note we use (index_offset%8==0) as the elements of our array are double precision (8 byte each) and memory is aligned on 64 byte boundaries, therefore boundaries are at every 64/8=8 elements of the array. We have one such statement for each matrix A,B and C.

After the adjustments, the innermost loop of our kernel looks similar to:

```
__assume(index_offset_A%8==0);
__assume(index_offset_B%8==0);
__assume(index_offset_C%8==0);

        for (k = 0; j < L1_BlockSize; ++k) {
          cij += A[sub_BA_A+k] * B[sub_BA_B+k];
        }
```

Aligning memory proved to be very effective, and increased our performance by 1 to 2 Gflop/s.

### 2.5.2. Keeping scalar operations to a minimum

As we want the compiler to vectorize as much operations as possible, we tried to make as few scalar operations as possible, as these are not vectorizable. A simple way to reduce the number of scalar operations is to precompute the index offset in the matrix, and update it only when needed. For example we changed the line inside the inner loop:

```
cij += A[((bk*nblock+bi))*L1_BS*L1_BS+L1_BS*i+k] * B[((bj*nblock)+bk)*L1_BS*L1_BS+L1_BS*k+j];
```

into

```
Cij+= A[index_offset_A+j] * B[index_offset_B+j];
```

where index_offset_A and index_offset_B are updated only in the outer loops.

### 2.5.3. Results

After these changes, using the command :

```
icc -O3 -qopt-report-phase=vec -qopt-report=5 -qopt-report-file=stdout -restrict -c dgemm_mine.c
```

we obtain the following vectorization report of the inner loop:

**Vectorization report**

```
LOOP BEGIN at dgemm_mine.c(124,13) inlined into dgemm_mine.c(339,5)
   remark #15388: vectorization support: reference C_161 has aligned access   [ dgemm_mine.c(126,17) ]
   remark #15388: vectorization support: reference C_161 has aligned access   [ dgemm_mine.c(126,17) ]
   remark #15388: vectorization support: reference A_161 has aligned access   [ dgemm_mine.c(126,17) ]
   remark #15388: vectorization support: reference B_161 has aligned access   [ dgemm_mine.c(126,17) ]
   remark #15427: loop was completely unrolled
   remark #15399: vectorization support: unroll factor set to 8
   remark #15300: LOOP WAS VECTORIZED
   remark #15448: unmasked aligned unit stride loads: 3
   remark #15449: unmasked aligned unit stride stores: 1
   remark #15475: --- begin vector loop cost summary ---
   remark #15476: scalar loop cost: 19
   remark #15477: vector loop cost: 4.000
   remark #15478: estimated potential speedup: 4.750
```

```
   remark #15479: lightweight vector operations: 6
   remark #15480: medium-overhead vector operations: 1
   remark #15488: --- end vector loop cost summary ---
LOOP END
```

Through optimizing vectorization, we were able to get an estimated potential speed-up of the vectorization is 4.750, up from 2.5 before implementing these changes.

## 2.6. Minimizing Branching

According to the book “Introduction to High Performance Computing for Scientists and Engineers”, branching is very expensive, therefore we tried to get rid of the branching in the main function. The branching came from the fact that we use a L2-blocking on top of an L1-blocking strategy, therefore we had to check if the L1 block inside the L2 block that we are trying to actually exists, as to not get a segmentation fault (an alternative would have been to pad the new matrix so that it is divisible by the size of an L2 block, but we figured that this would increase the size of the matrix too much). Note that this problem did not arise when we dealt only with the L1 block size because we padded the matrix with zeros so that the new matrix was always divisible by the L1 block size. Therefore the 6-nested loops in our main function (which itself called the kernel function which has 3 nested loops) looked like:

**Main Loop with branching**

```
for (L2bk=0; L2bk < L2nblock; ++L2bk){
for (L2bj=0; L2bj < L2nblock; ++L2bj){
for (L2bi=0; L2bi < L2nblock; ++L2bi){
for (bk = 0; bk < L2_BS; ++bk) {
        for (bj = 0; bj < L2_BS; ++bj) {
                for (bi = 0; bi < L2_BS; ++bi) {
                        if((L2bi*L2_BS+bi<nblock)&&(L2bj*L2_BS+bj<nblock) && (L2bk*L2_BS +bk<nblock)){
                        do_block(M, nblock, bA, bB, bC, L2bi*L2_BS+bi, L2bj*L2_BS+bj, L2bk*L2_BS+bk);}
                }
        }
}
}
}
}
```

To get rid of this “if statement”, we had to deal with 8 cases separately, depending on if the indexes i,k,j were in an L2_block which contains the right or lower boundary of the matrix. The table below describes the 8 cases (NOT means index is not at boundary, and boundary means the index is in the L2 block which contains the right or lower boundary of the matrix)

| | index i | index j | index k |
|---|---|---|---|
| Case 1 | NOT | NOT | NOT |
| Case 2 | boundary | NOT | NOT |
| Case 3 | NOT | boundary | NOT |
| Case 4 | NOT | NOT | boundary |
| Case 5 | boundary | boundary | NOT |
| Case 6 | boundary | NOT | boundary |
| Case 7 | NOT | boundary | boundary |
| Case 8 | boundary | boundary | boundary |

For example, here is case 7:

**Case 7: j at boundary, k at boundary, i not at boundary**

---

```
L2bj=L2nblock-1;
L2bk=L2nblock-1;
for (L2bi=0; L2bi < L2nblock-1; ++L2bi){
for (bk = 0; bk < rem; ++bk) { % up to remainder! not blocksize
    int Ak=L2bk*L2_BS +bk;
    for (bj = 0; bj < rem; ++bj) { % up to remainder! not blocksize
        int Aj=L2bj*L2_BS+bj;
        for (bi = 0; bi < L2_BS; ++bi) {
            int Ai=L2bi*L2_BS+bi;
            do_block(M, nblock, bA, bB, bC, Ai, Aj, Ak);
        }
    }
}
}
```

---

Unfortunately this made the code significantly longer and harder to read, for no significant change in the performance.

## 3. Evaluation

### 3.1. What Worked

Overall, the best methods of optimization seemed to be:

1. Tuning the Block Size
2. Using a Block indexing scheme
3. Improving vectorization

Simply by tuning the block size, we were able to double the computation speed. This indicates that, on large matrices, the problem is heavily memory access bound. Utilizing the cache optimally is likely to lead to substantially improved performance.

This is also present when examining the results of using a block indexing scheme. This scheme allows for a greater cache hit rate. Since the problem is memory access bound, any changes that result in better cache usage will likely have large effects on performance.
Additionally, by increasing the compiler's ability to vectorize our code, we were able to get a greater instructional efficiency that lead to a substantial improvement in performance.

### 3.2. What Didn't

In general, modifying loop order and compiler settings did not have a substantial impact on the performance. Neither of these experiments affected cache utilization substantially. Since this problem appears to be memory bounded, and these changes didn't noticeably change the cache utilization, it makes sense that they didn't make substantial changes to the performance.
Additionally, we had hoped that minimizing the number of branches in our code would improve performance. However, this did not have a substantial effect. This is likely because our algorithm is memory bound. The penalty for clearing a pipeline is primarily computational. This caused our changes to result in minimal performance gains.

## 4. Conclusion

In this homework, we optimized matrix multiplication algorithm. We attempted various optimization techniques, and found that some technique affects the performance significantly, while some did not. Especially, we could get the best result with using Block indexing by achieving better cache utility. Quantitatively, we were able to get our performance up to around 8.3 GFLOP/s. We were able to beat f2c. Our algorithm still has a lot of improvements that can be made; however, it is substantially better than the untuned algorithm that we started with.
Final version: peak (8.3 Gflop/s)