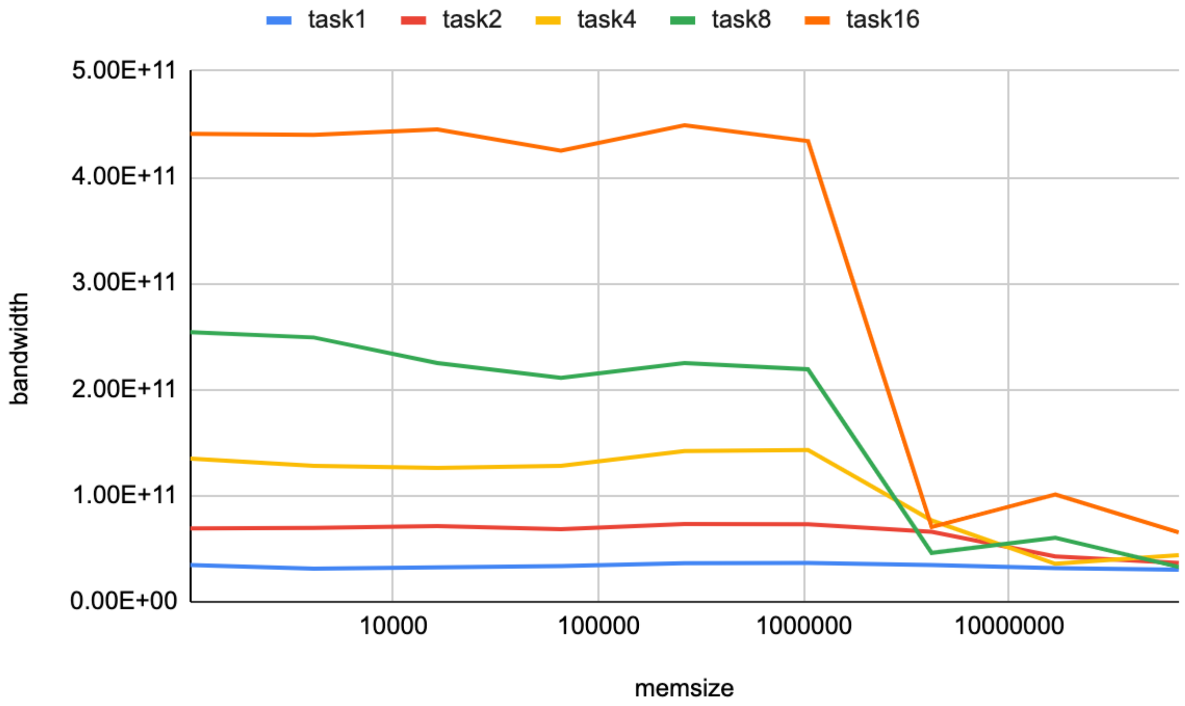
EC527
Lab1 report
Bin Xu
Seyed Reza Sajjadinasab

Part 1



1a. Memory size: 262144

1b. Memory size: 67108864,   ratio: 7 / 9

1c. Memory size: 4194304, 2 times, in task 4 and task 16

1d. 8 tasks will get close bandwidth as what we get from 1 task

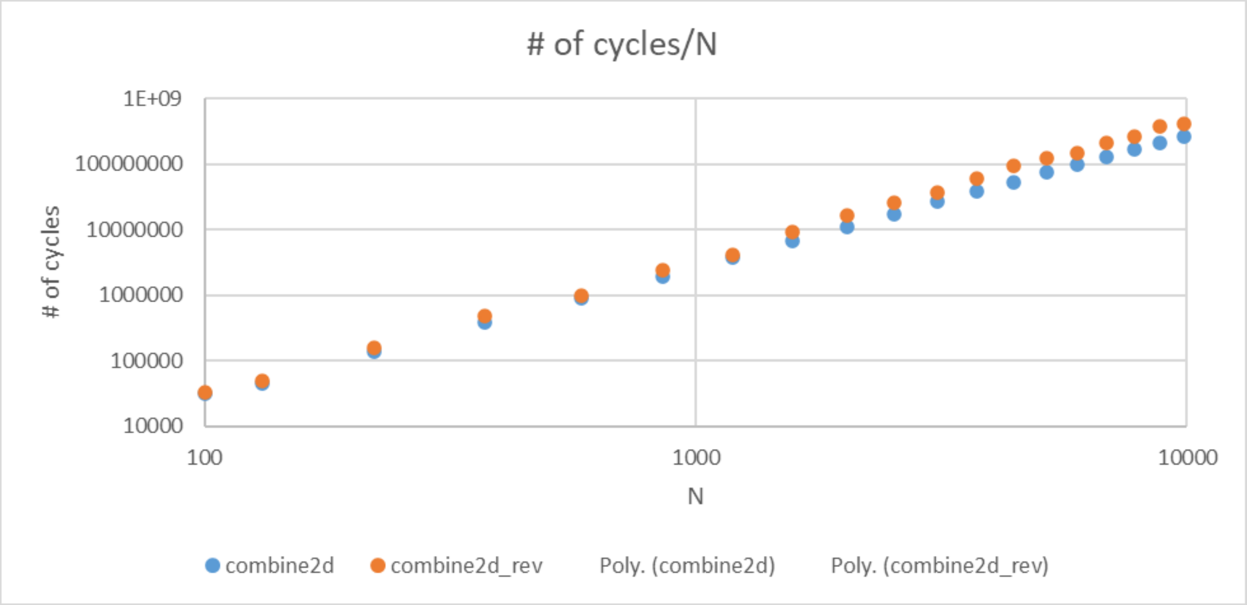
Part 2

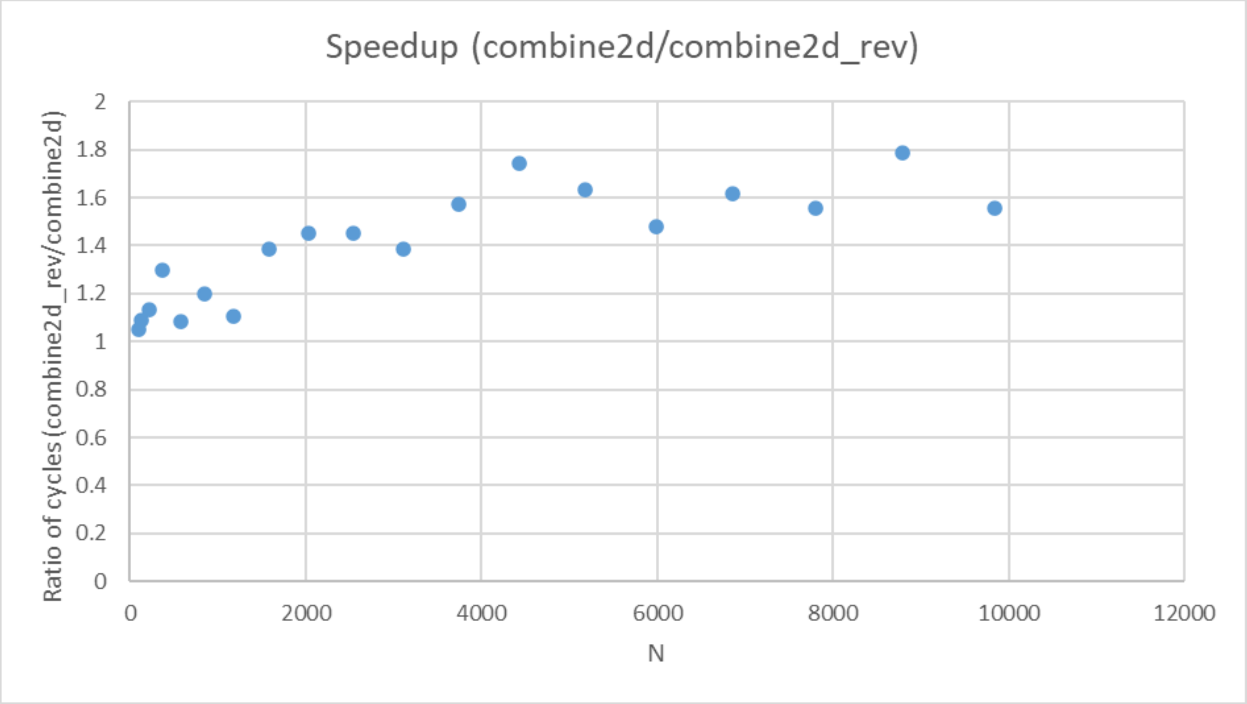
2a.

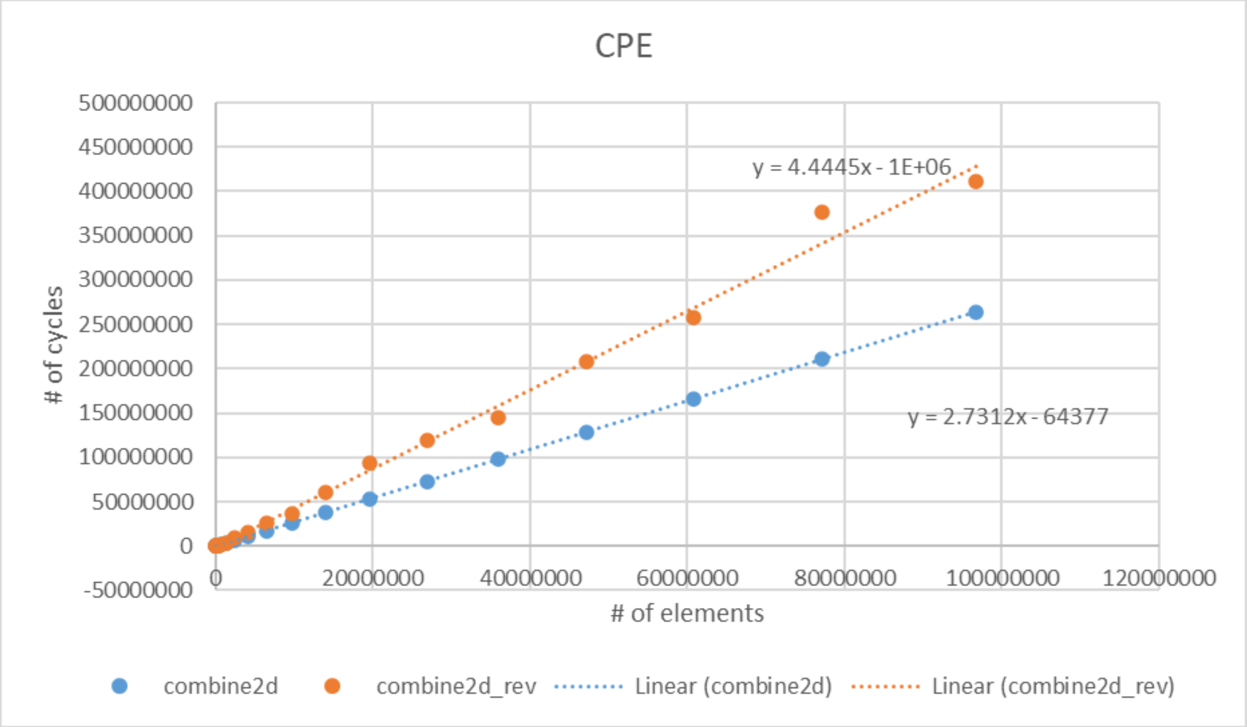
The last big n that we can obtain the result for was 43458. By doubling the `NUM_TESTS` again to 80, we received the following error:

COULDN'T ALLOCATE 2.809771e+11 BYTES STORAGE (row_len=187409)

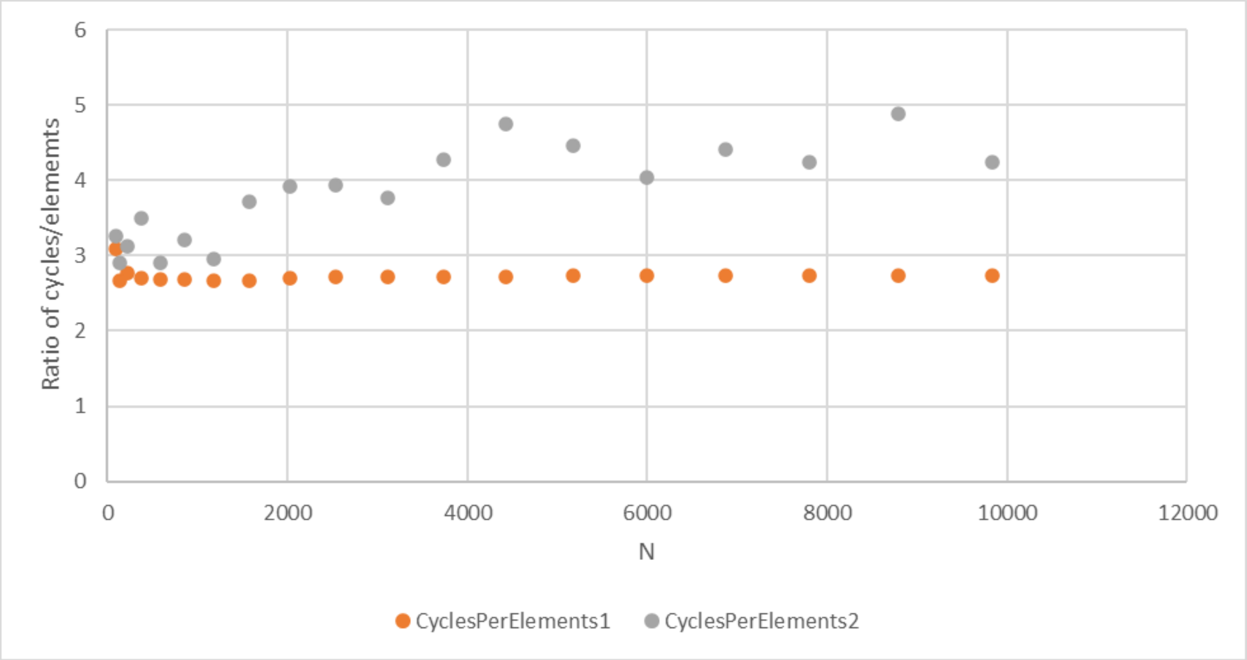
Looking into these Graphs we can see that the combine2d is faster than combine2d_rev.







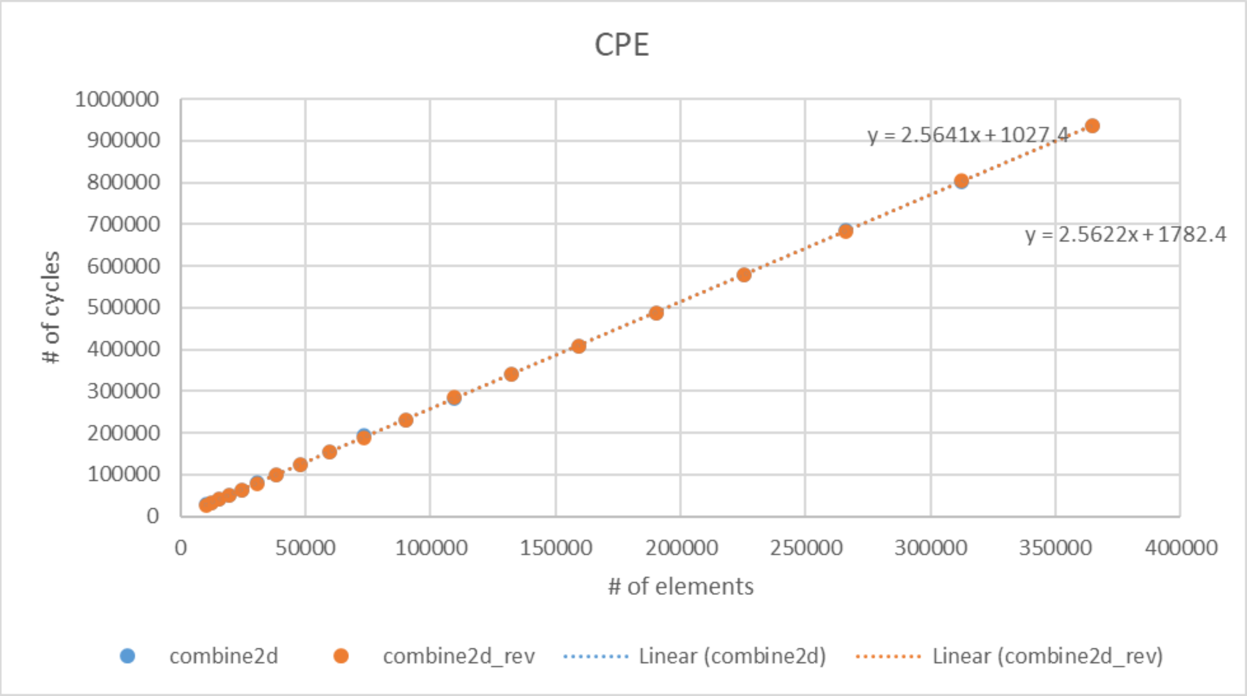
According to the CPE we can say that combine2d is roughly 2 times faster than combine2d_rev.



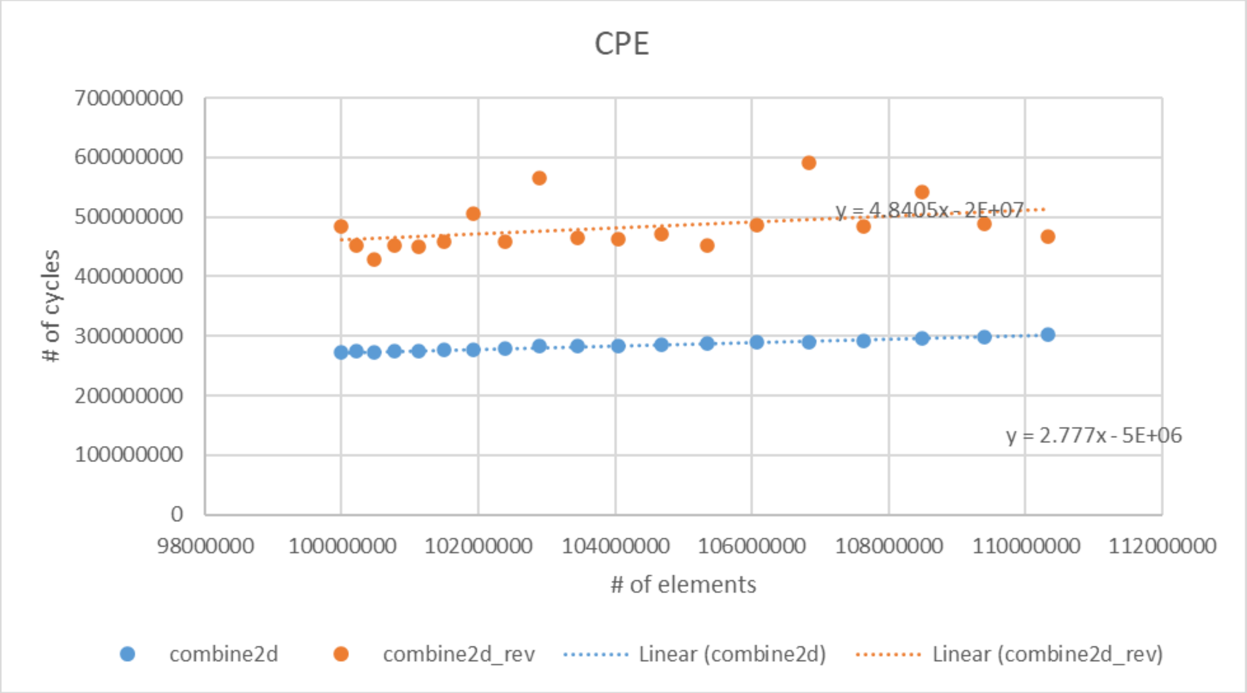
As we can see for N>1000 there is a strange behavior in the data, especially for combined2d_rev.

To explain the reason for this behavior, first we should know what exactly these two programs do. They are accumulating an array with a constant variable(accumulator). To this end, we should iterate over an array. One of them starts from the beginning with step size of 1, and the other from the beginning, but a step of row length. For the latter, every time we want to read an element from the array we have a miss unless it had already been stored in the cache. In this regard, the size of the cash and the number of elements can make a big difference. On the other hand, In the first one, we are using the maximum locality of blocking that we can, and the miss rate is negligible. So, we can see a more linear graph.

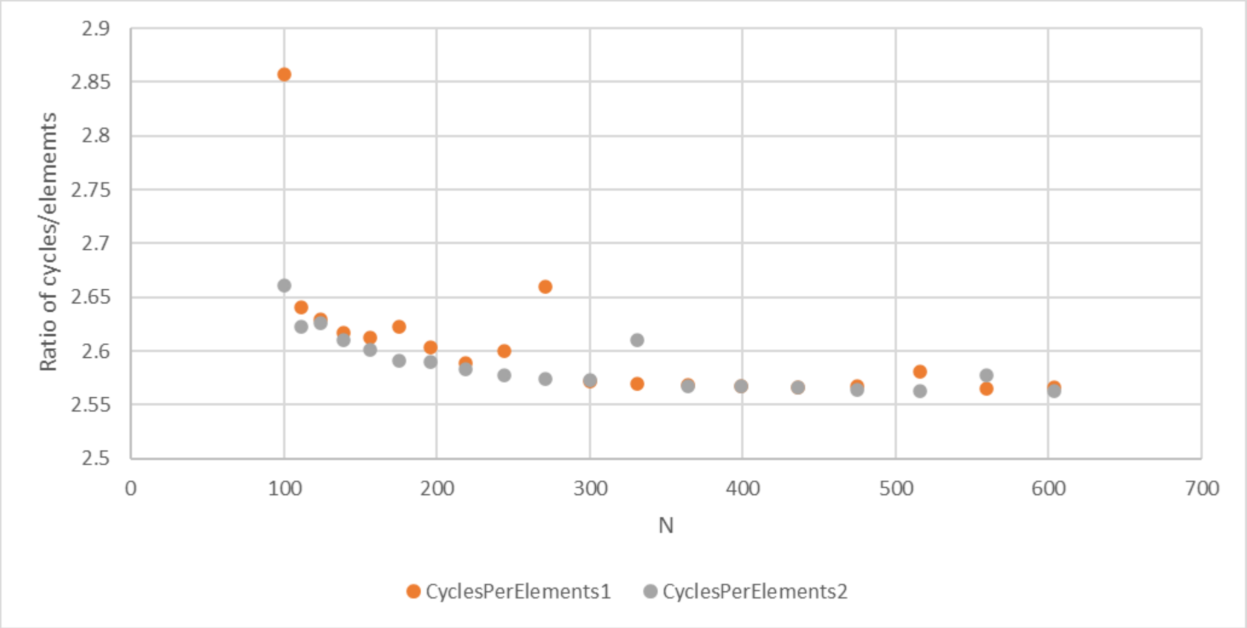
2b.
To calculate the CPE we need to use the square of n since we are using it for a matrix. For small range for small values of n we have:



And for large number of n:

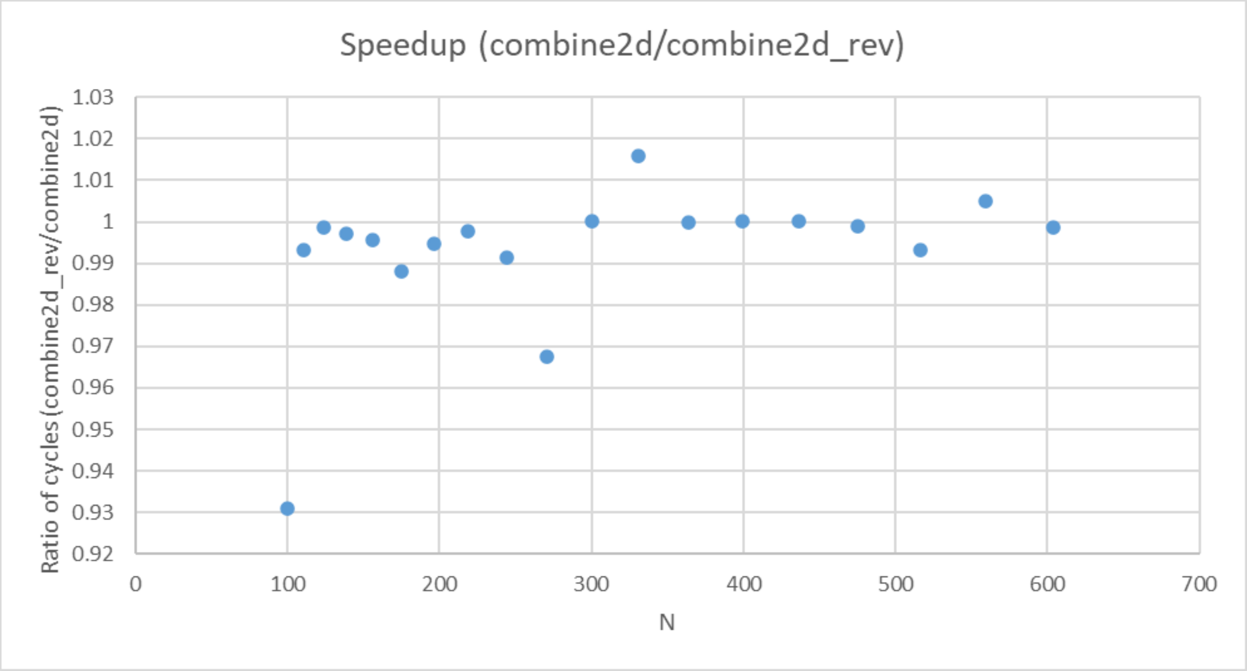


2c.
For the graph (d) we have



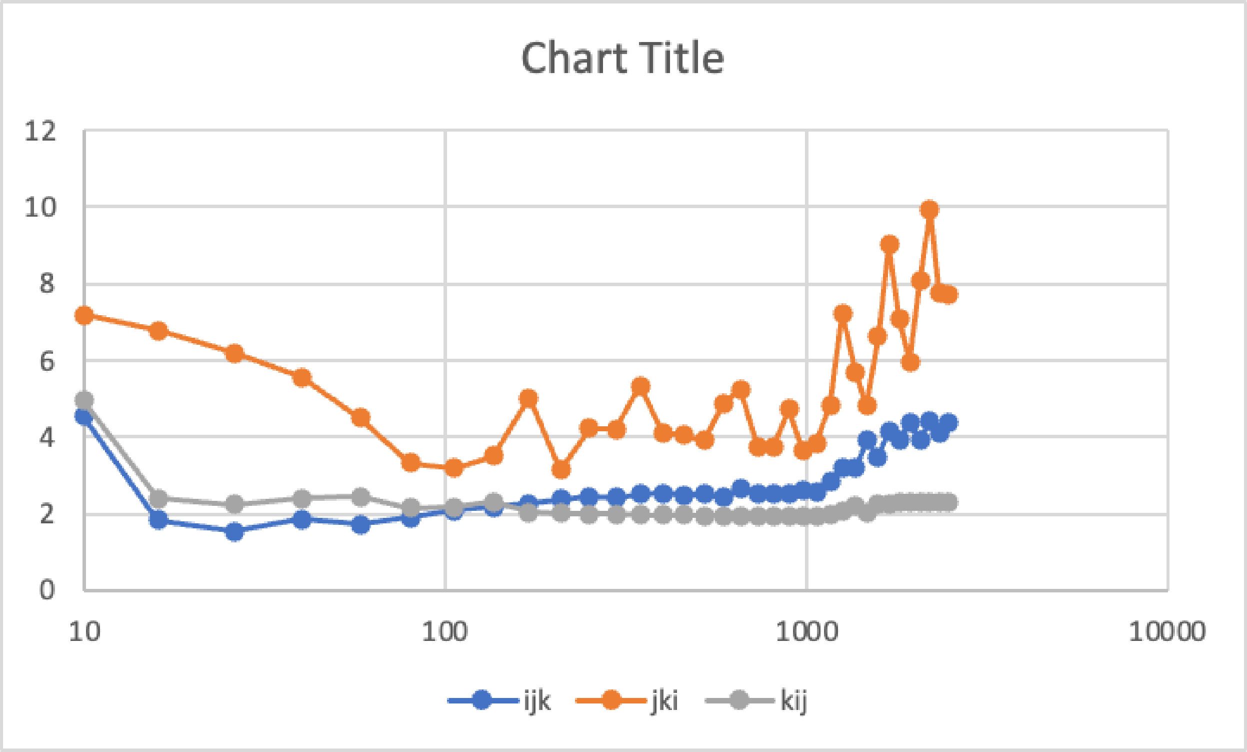
As we can see in small numbers the performance is increasing! This is because of the compulsory miss that we have in small ranges.

Also, the graph (e) was interesting. Below is a version of it for small numbers.


Speedup (combine2d/combine2d_rev)
Ratio of cycles (combine2d_rev/combine2d)
1.03
1.02
1.01
1
0.99
0.98
0.97
0.96
0.95
0.94
0.93
0.92
0
100
200
300
400
500
600
700
N

Part 3

```
#define A   2  /* coefficient of x^2 */
#define B   4  /* coefficient of x */
#define C   10  /* constant term */
#define NUM_TESTS 35   /* Number of different sizes to test */
```

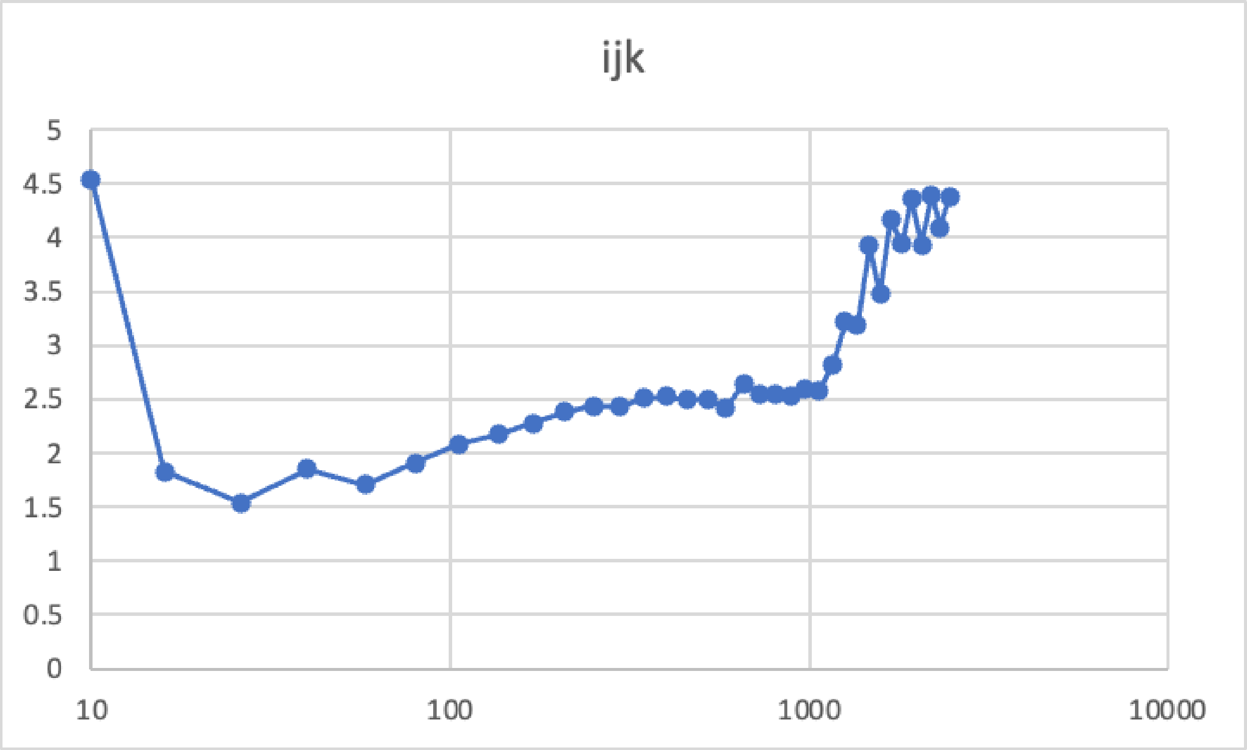


3a.
We set horizontal line as log row length, and vertical line as total cycle / (row length ^ 3)
We test above 2000 array sizes. We set 35 iterations.
As we can get info from below graph, we get plateaus is between row length 500 and 1066, 10 to 100.

2 plateaus.

Average number of cycles per innermost loop iteration10 to 100: 1.870521452
Average number of cycles per innermost loop iteration 500 to 1066: 2.545950711

First plateaus is from 16 to 80
second plateaus is from 656 to 1066



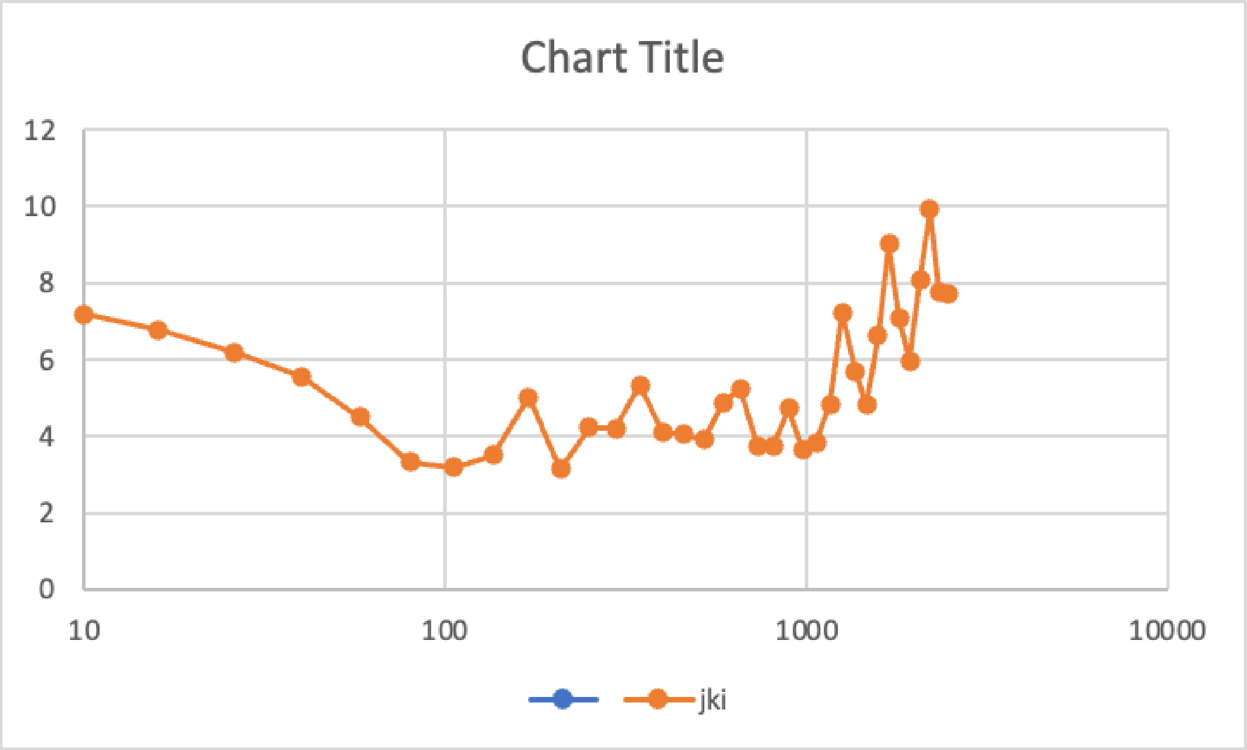
3b.

As we can get info from below graph, we get plateaus is between matrix size 100 and 1000;

1 plateaus.

Average number of cycles per innermost loop iteration: 4.187595323

Plateaus is from 250 to 1066



3c.
As we can get info from below graph, we get plateaus is between row length 10 and 100, 100 and 1000, 1000 to 2000;

3 plateaus

Average number of cycles per innermost loop iteration 10 to 100: 2.301051555
Average number of cycles per innermost loop iteration 100 to 1000: 2.004975796
Average number of cycles per innermost loop iteration 1000 to 2000: 2.250764096

First plateaus is from 16 to 106
Second plateaus is from 170 to 1066
Third plateaus is from 1576 to 2458

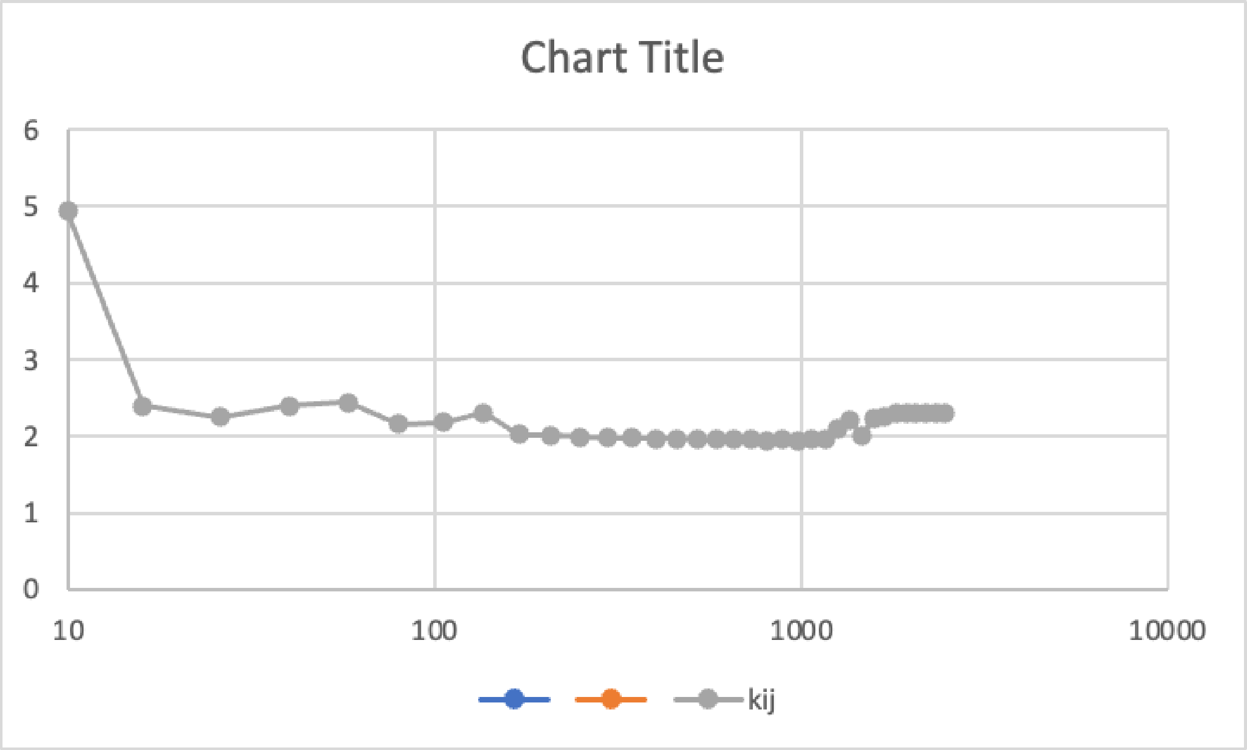


Chart Title
6
5
4
3
2
1
0
10
100
1000
10000
kij

Part 4

4a.

Note: There is another version of blocking with 6 nested loops as presented in the course slides. The only difference is that in the five nested loops version two of the loops have been merged. So, intrinsically they are both the same. As a result we just use the version that is provided on the slides with a slight difference in the stride of the loops. Note that the result of 5 and 6 nested loops are the same and it behaves as we expected.

In this code we are taking care of those block sizes that are not proponent to the length of the matrix by adding an extra checking for corner values.

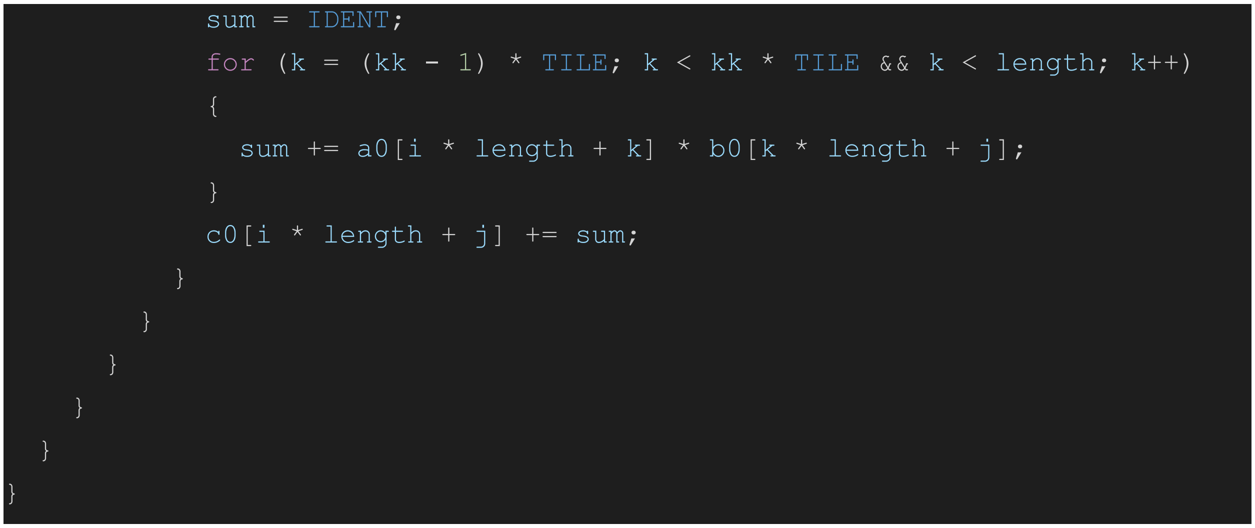
Blocking improves the performance by reducing cache miss and increasing the amount of data that can be stored in the cache. It depends on the matrix size.

As the matrix size increases, the potential performance improvement from blocking also increases.

There is a trade-off between the size of the blocks and the overhead of managing the blocks, which can decrease the overall performance improvement if the blocks are too small or too large.

Add below code:

```
void mmm_bijk(matrix_ptr a, matrix_ptr b, matrix_ptr c)
{
  long int i, j, k, ii, kk, jj;
  long int length = get_matrix_row_length(a);
  data_t *a0 = get_matrix_start(a);
  data_t *b0 = get_matrix_start(b);
  data_t *c0 = get_matrix_start(c);
  data_t r;
  double sum;
  long int NumOfTile = floor(length / TILE) + 2;

  for (ii = 1; ii < NumOfTile; ii++)
  {
    for (jj = 1; jj < NumOfTile; jj++)
    {
      for (kk = 1; kk < NumOfTile; kk++)
      {
        for (i = (ii - 1) * TILE; i < ii * TILE && i < length; i++)
        {
          for (j = (jj - 1) * TILE; j < jj * TILE && j < length; j++)
          {
```

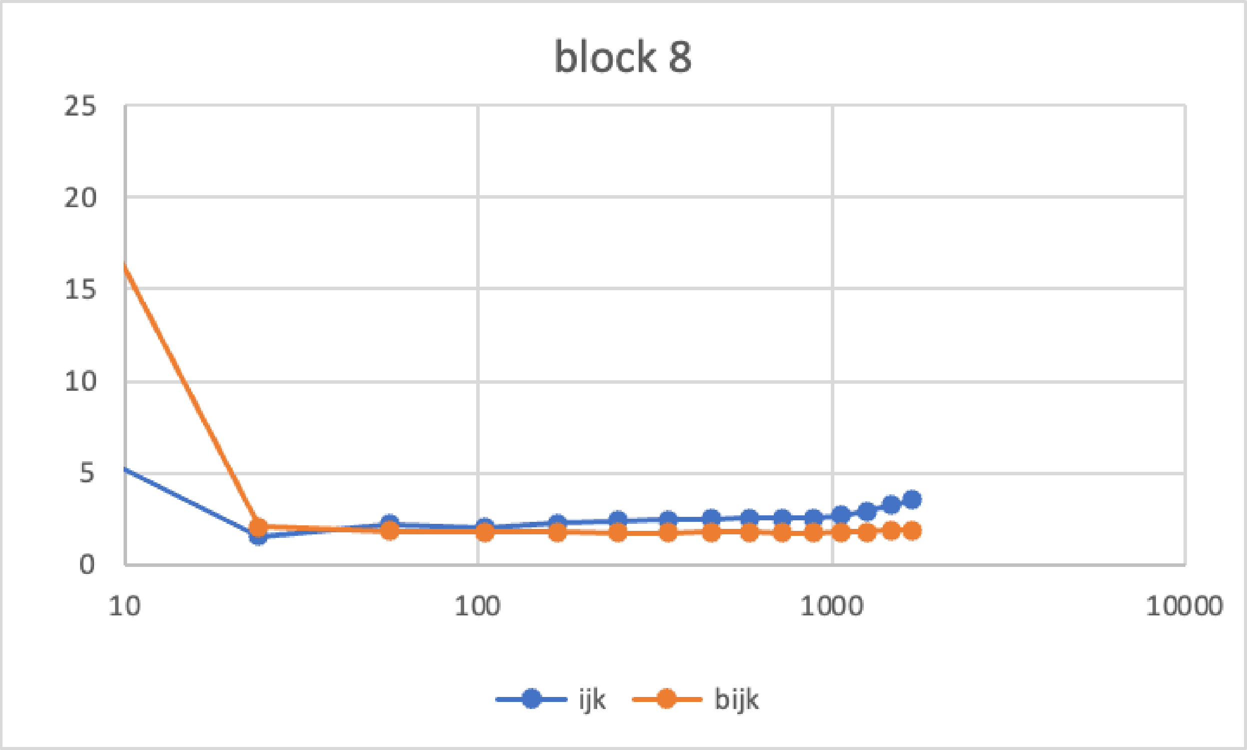


```
            sum = IDENT;
            for (k = (kk - 1) * TILE; k < kk * TILE && k < length; k++)
            {
              sum += a0[i * length + k] * b0[k * length + j];
            }
            c0[i * length + j] += sum;
          }
        }
      }
    }
  }
}
```

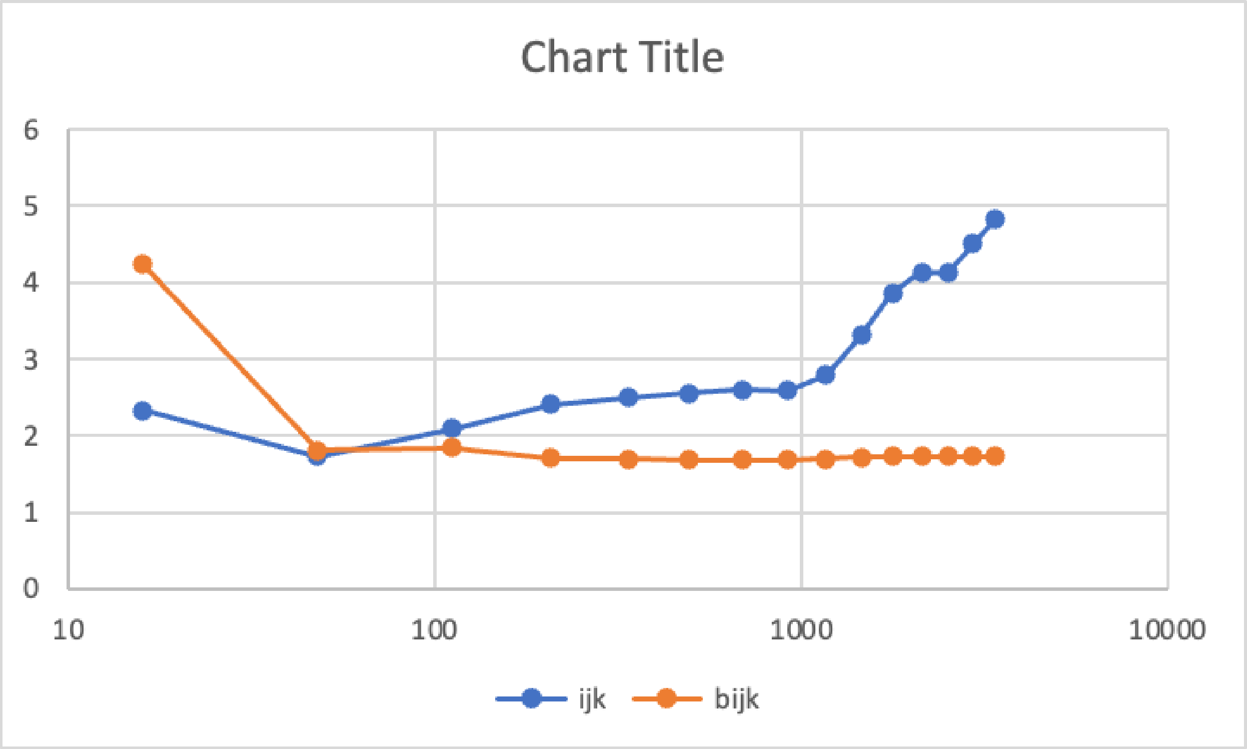
4b.

When the block size is 16, matrix multiplication with blocking gives better performance. After block reaches a certain point, the performance is reduced. The performance of 64 block is lower than the performance of block 32.

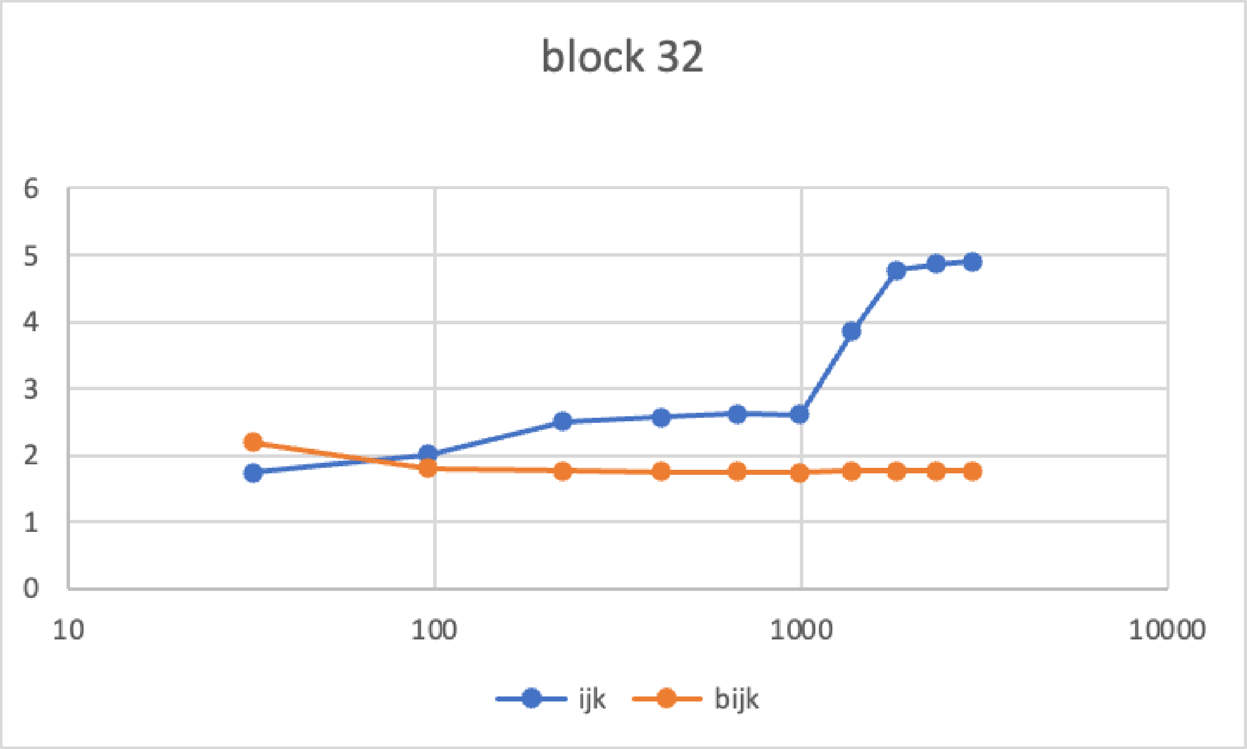
Block 8:



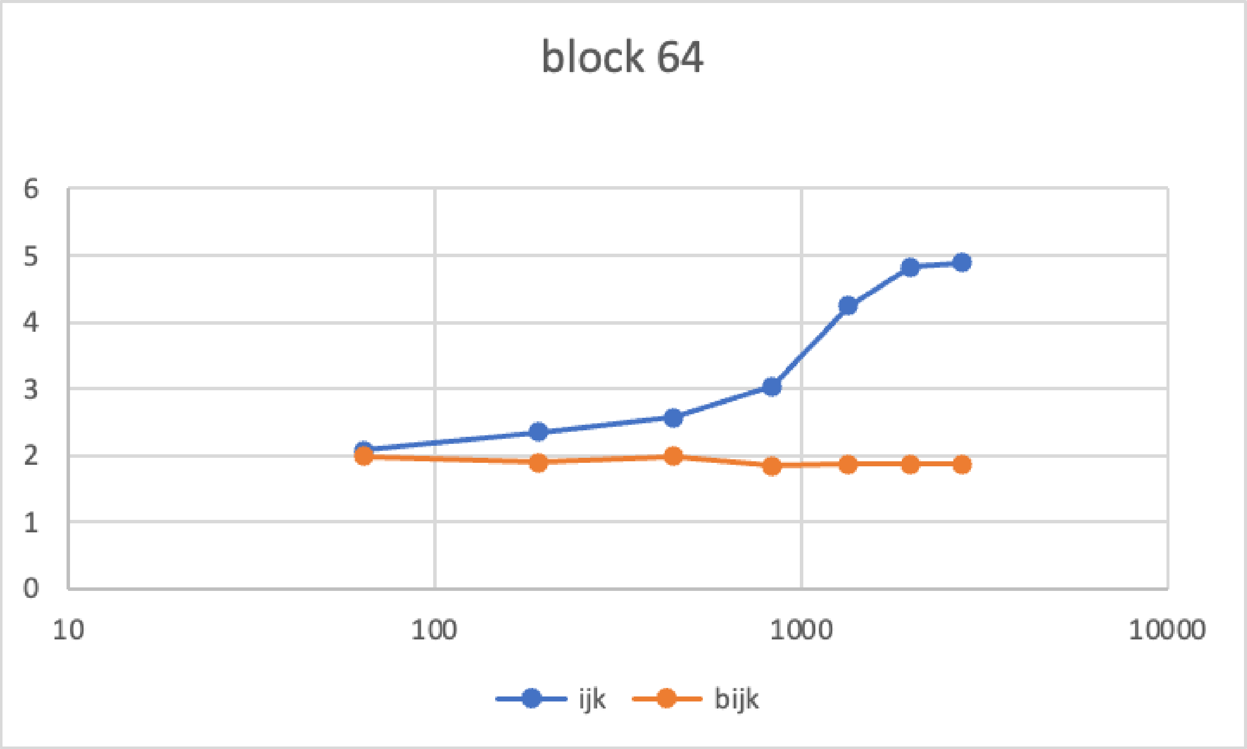
Block 16:



Block 32;



Block 64



4c.
Fp utilization = (number of flops performed by the code) / (maximum number of flops that the system's hardware is capable of performing)

The Gflops of intel core i7 9700 is 412.5e9 according to the Geekbench 4 SGEMM. The number of flops performed by the code is
Number of floating-point operations/execution time.
The number of floating point operations is 3 * N^3. (N is the length of matrix)
The execution time is:
Cycles per innermost loop iteration * Total number of innermost loop iterations / Clock frequency
So we have: (For row length 1688 the Cycles per innermost loop iteration was 1.89 for block size of 8.)
Execution time = 1.89 * N^3/3.0e9.
So we have number of flops performed by the code = 4.76e9

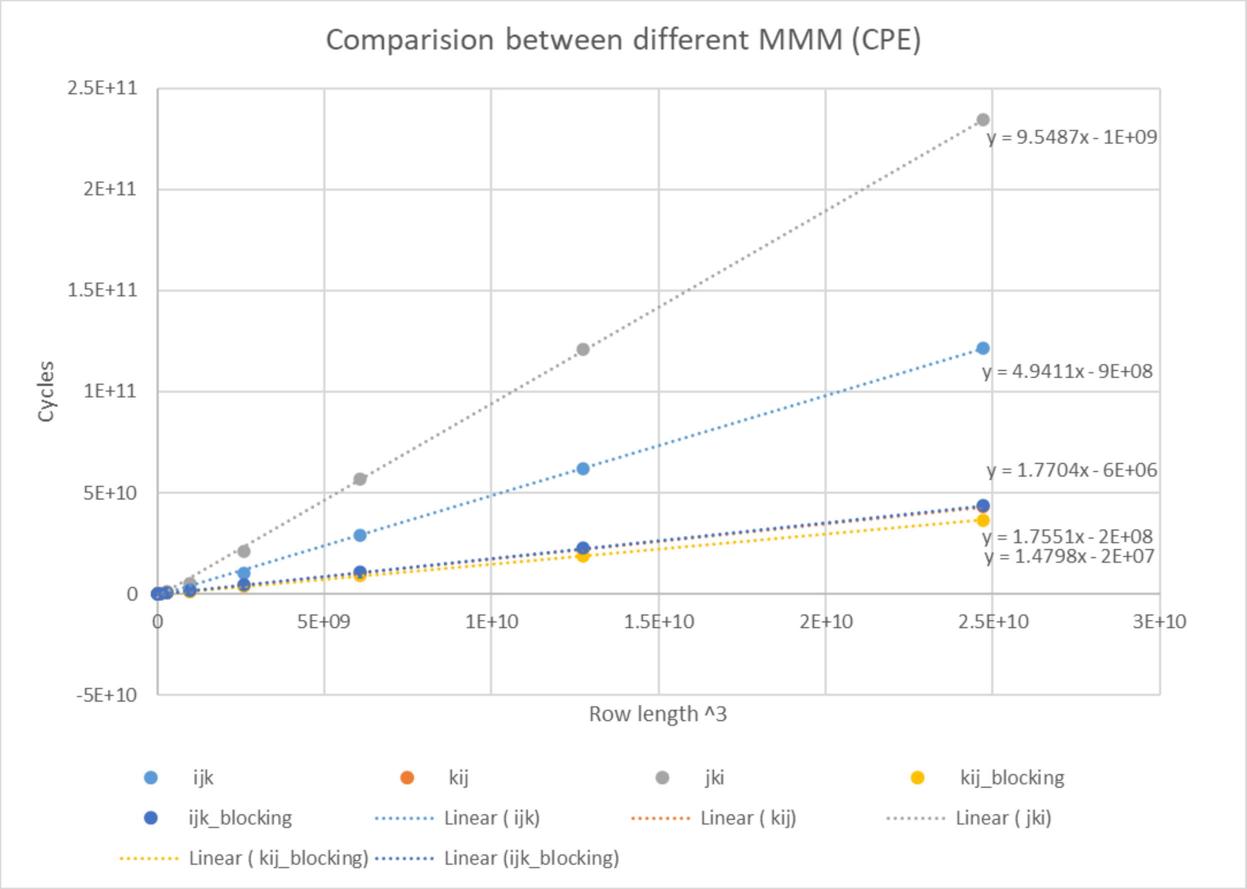
The we have
FP utilization = 4.76e9/412.5e9 = 1.15%

If we use the CPE for Cycles per innermost loop iteration. According to what we can see on the 4e (The best CPE is 1.48) we will have an fp utilization of 1.47%.

So,

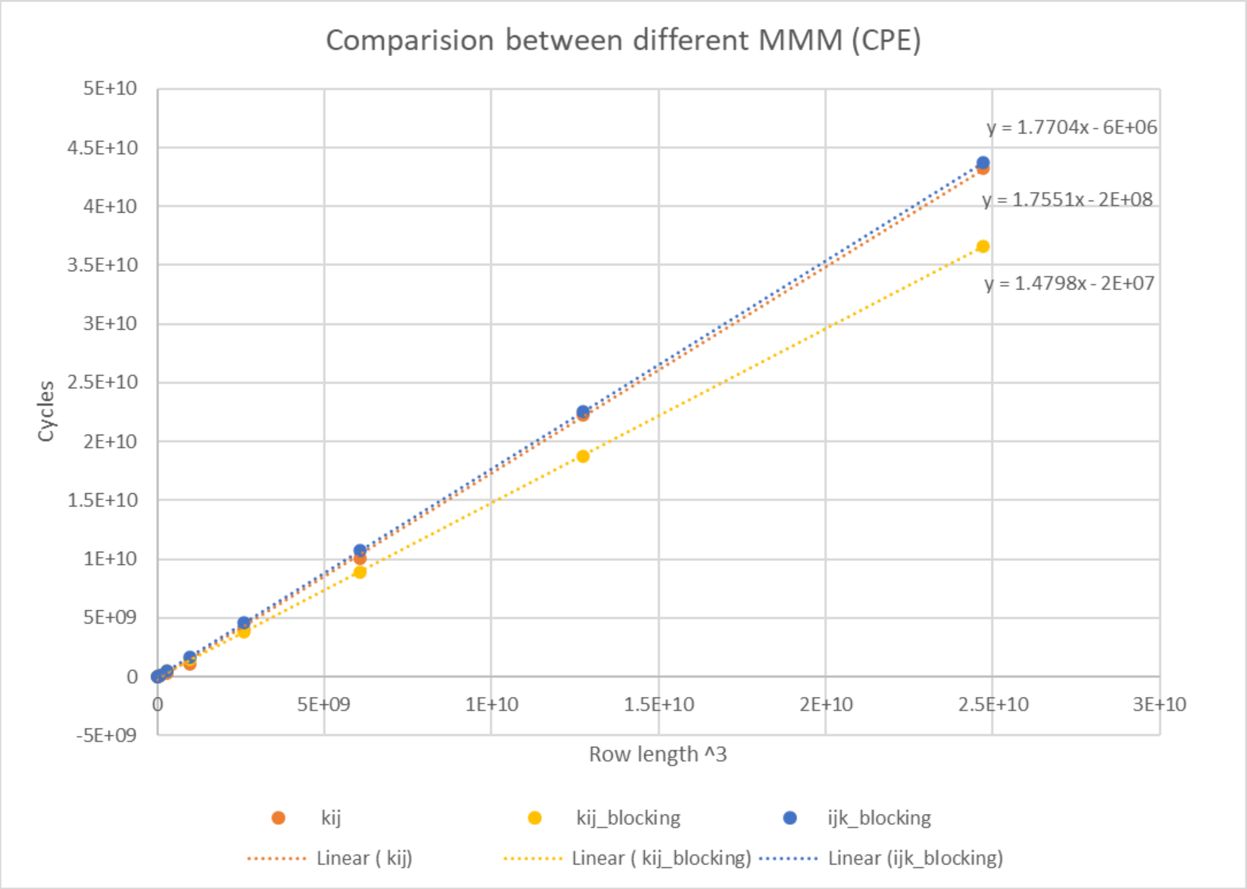
4d. The thing is that we are probably no longer bound by memory bandwidth. Another thing is that the ijk blocking has a problem intrinsically. Basically, kij is the best algorithm among others because of how it is indexing the values of matrices. It uses a stride of 1 for all reading & writing inside of the innermost loop, but ikj is not. So we can use blocking for kij to have a better performance.

4e. As described above the best thing to do is to change it in kij order. The result is as below:



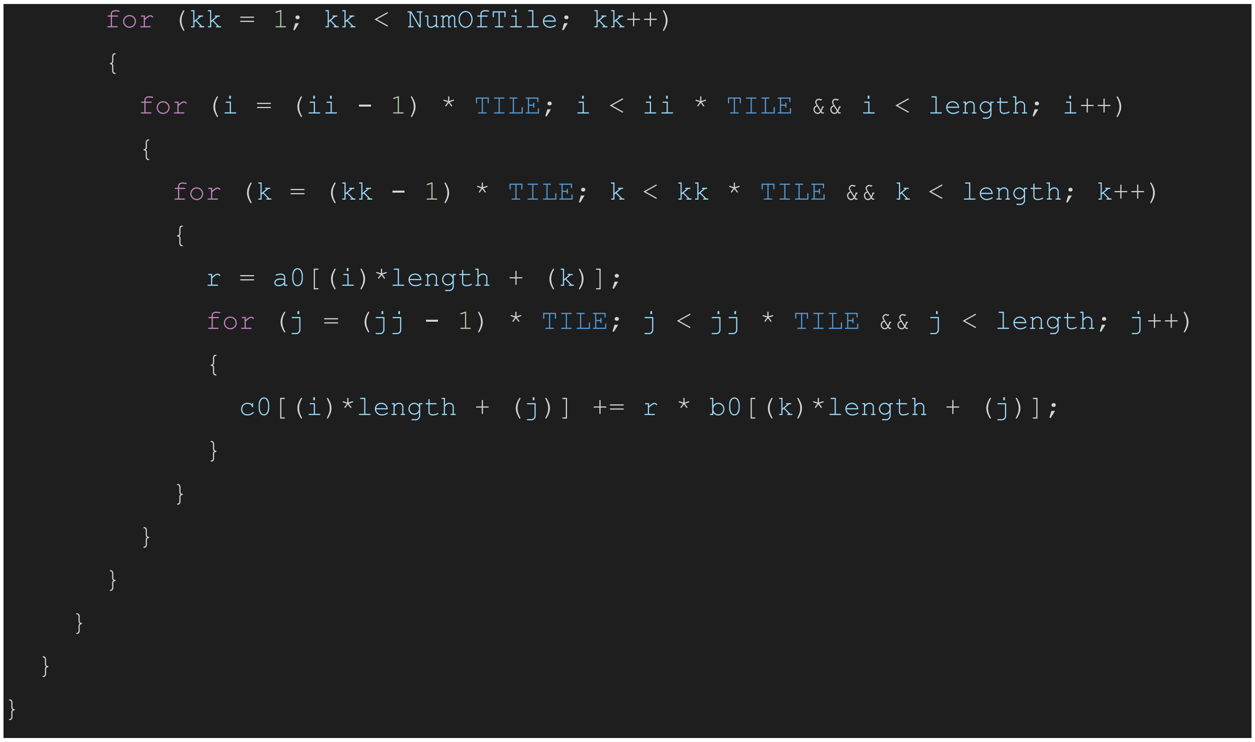
We can see that the blocking version of ijk is 3 times faster than the non-blocking one. Also, the blocking version of kij which has the best performance, is almost 15 percent better than the non blocking version.

To make the difference clear we omit the results of ijk and jki because they are too bad compared to others. The new version is as follows.



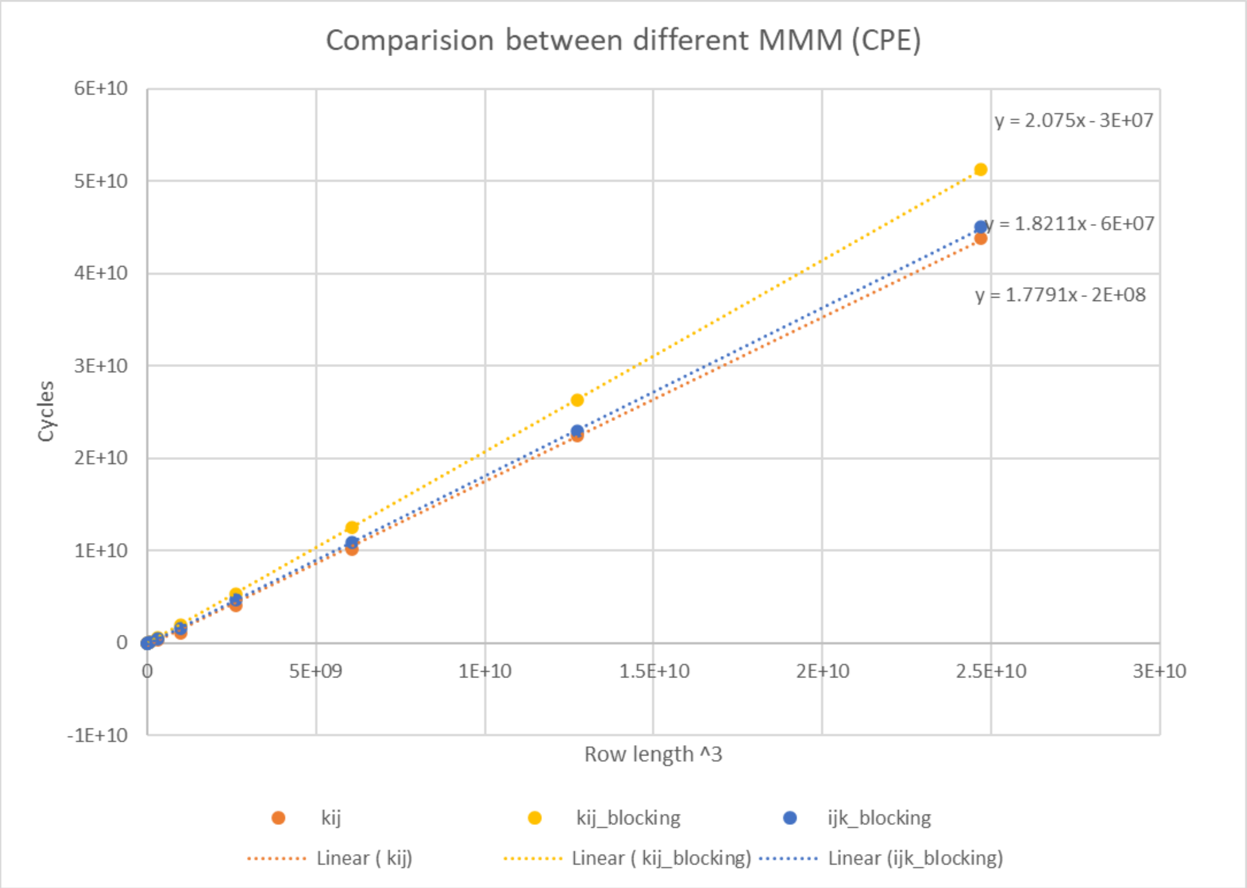
Here is the code that we use for kij_blocking: (Here TILE is the block size and we set to be 32)

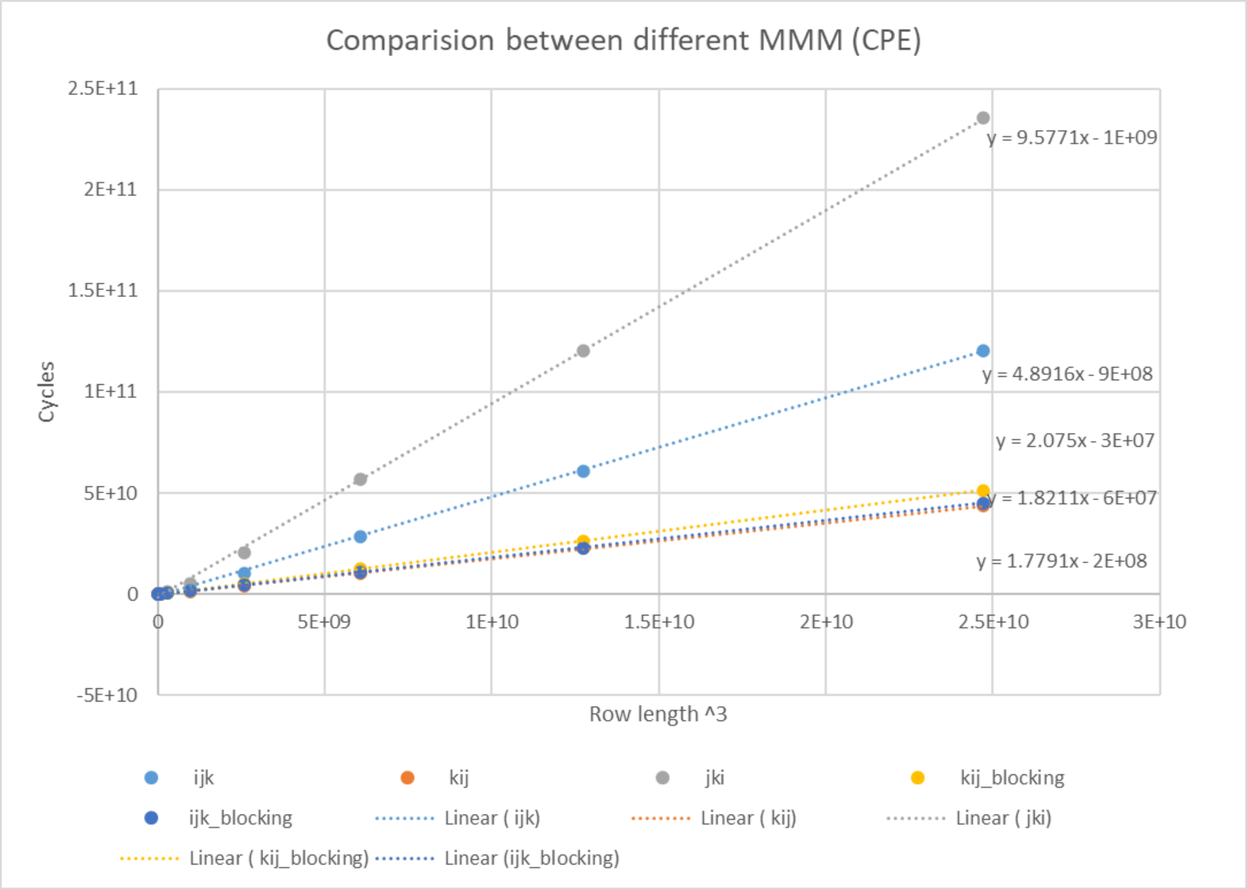
```
void mmm_bkij(matrix_ptr a, matrix_ptr b, matrix_ptr c)
{
  long int i, j, k, ii, jj, kk;
  long int length = get_matrix_row_length(a);
  long int length2 = length * length;
  data_t *a0 = get_matrix_start(a);
  data_t *b0 = get_matrix_start(b);
  data_t *c0 = get_matrix_start(c);
  data_t r;

  long int NumOfTile = floor(length / TILE) + 2;

  for (ii = 1; ii < NumOfTile; ii++)
  {
    for (jj = 1; jj < NumOfTile; jj++)
    {
```



```
      for (kk = 1; kk < NumOfTile; kk++)
      {
        for (i = (ii - 1) * TILE; i < ii * TILE && i < length; i++)
        {
          for (k = (kk - 1) * TILE; k < kk * TILE && k < length; k++)
          {
            r = a0[(i)*length + (k)];
            for (j = (jj - 1) * TILE; j < jj * TILE && j < length; j++)
            {
              c0[(i)*length + (j)] += r * b0[(k)*length + (j)];
            }
          }
        }
      }
    }
  }
}
```

We also calculate the result for block size of 8.





Here we can see that kij is the best overall, and blocking doesn't increase the performance, but it did for ijk.

Part 5

5a.
We have two versions of transpose and transpose_rev methods, blocking and non-blocking.
New code with blocking in the test_transpose.c

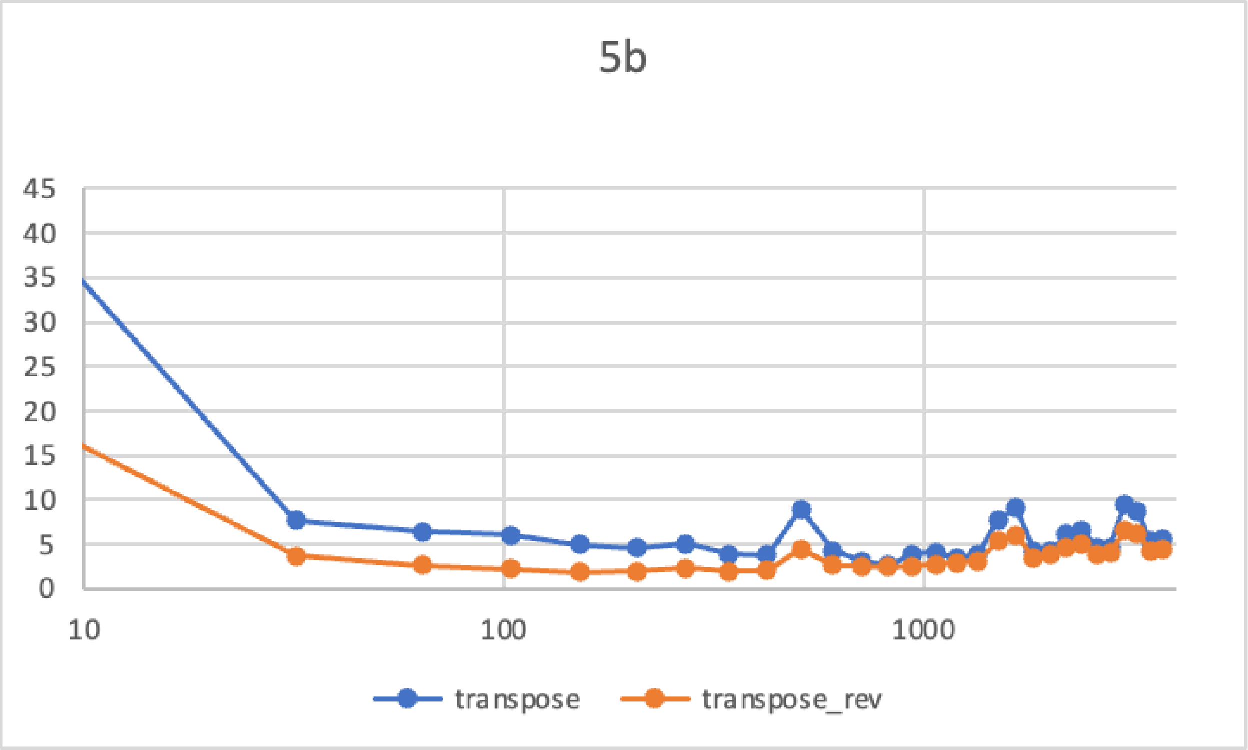
```
void transpose(array_ptr v, data_t *dest)
{
  long int i, j, k, m;
  long int length = get_row_length(v);
  data_t *data = get_array_start(v);
  for(i = 0; i < length; i += blocksize) {
     for (j = 0; j < length; j += blocksize) {
        //transpose block between [i,j]
        for (k = i; k < i + blocksize; k++) {
           for (m = j; m < j + blocksize; m++) {
              dest[k + m * length] = data[k + m * length];
           }
        }
     }
  }
}

void transpose_rev(array_ptr v, data_t *dest)
{
  long int i, j, k, m;
  long int length = get_row_length(v);
  data_t *data = get_array_start(v);

  for(i = 0; i < length; i += blocksize) {
     for (j = 0; j < length; j += blocksize) {
        //transpose block between [i,j]
        for (k = i; k < i + blocksize; k++) {
           for (m = j; m < j + blocksize; m++) {
              dest[m + k * length] = data[m + k * length];
           }
        }
     }
  }
}
```

5b.

For Intel(R) Core(TM) i7-9700 , L1 cache size is 256 KB. L3 cache size is 12 MB(12288KB). We test different sizes of matrix. For L1 cache, it is below 40 row_length. From the test, Transpose_rev provides better performance than Transpose_rev.

We add below code:

```
void transpose(array_ptr v, data_t *dest)
{
  long int i, j, k, m;
  //matrix row length
  long int length = get_row_length(v);
  data_t *data = get_array_start(v);
  for(i = 0; i < length; i++) {
     for (j = 0; j < length; j++) {
        dest[i + j * length] = data[j + i * length];
     }
  }
}

void transpose_rev(array_ptr v, data_t *dest)
{
  long int i, j, k, m;
  long int length = get_row_length(v);
  data_t *data = get_array_start(v);

  for(i = 0; i < length; i ++) {
     for (j = 0; j < length; j ++) {
        dest[j + i * length] = data[i + j * length];
     }
  }
}
```



Vertical line is cycles per innermost loop iteration

5c.

In the matrix transpose, we need to take advantage of locality and cache blocking. We change the original matrix transpose operation in the loop. We split the matrix into different blocks, and check which block size is more efficient.

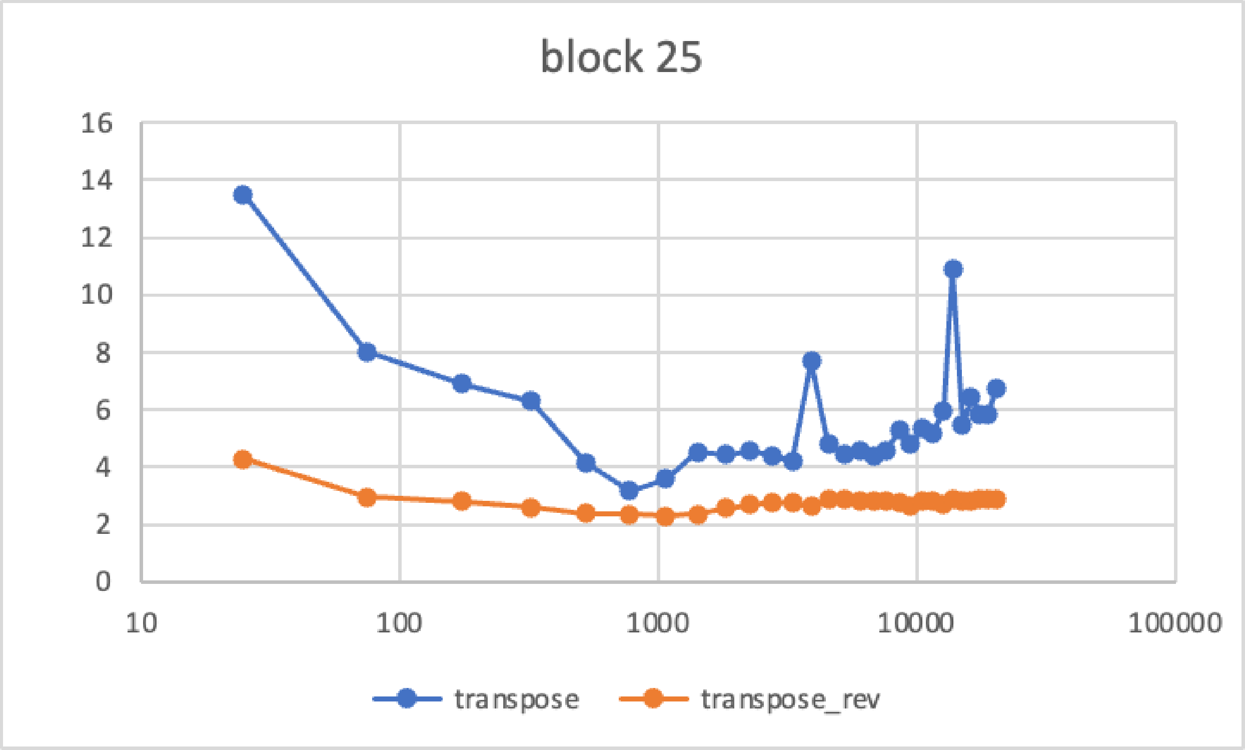
Blocking is done in the four loops. Two loops are used to traverse the array. The other two are used to traverse blocks. Block is loaded into the cache.

Generally, performance of blocking code is close to the performance of non-blocking code.

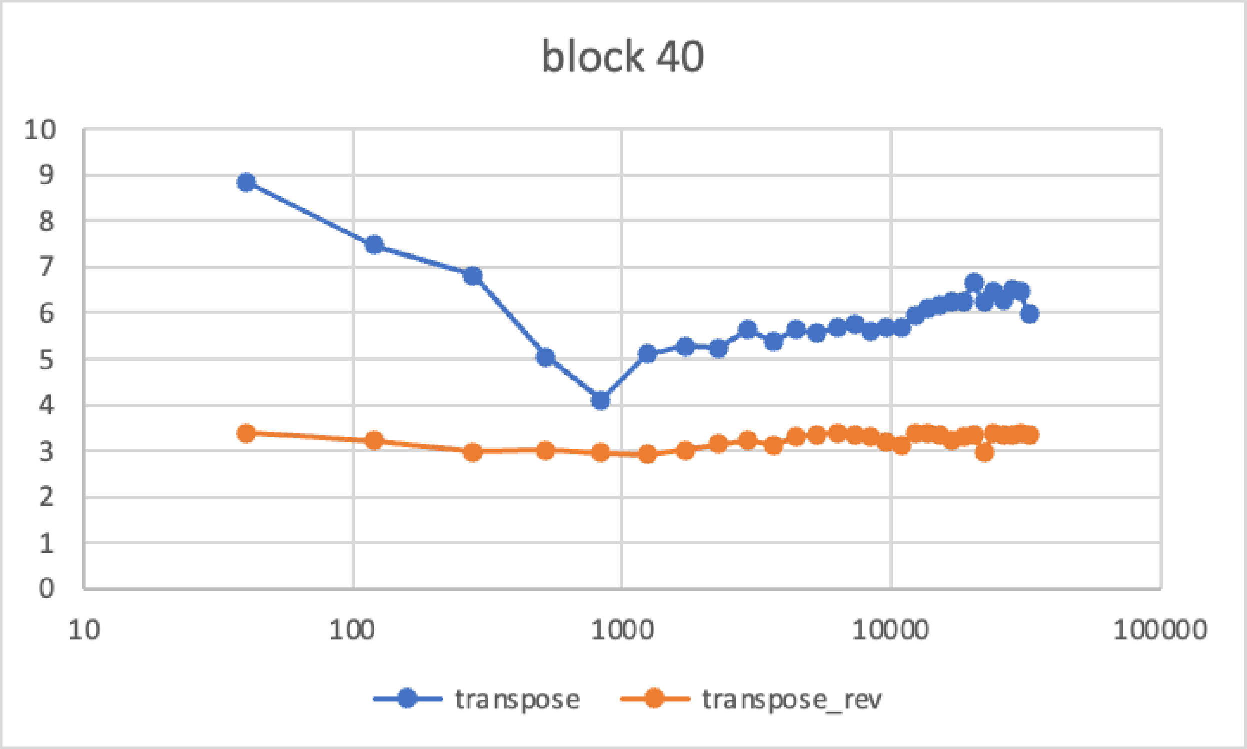
From the below charts, Blocking transpose_rev provides better performance than blocking transpose.

25 blocks and 55 blocks give better performance.

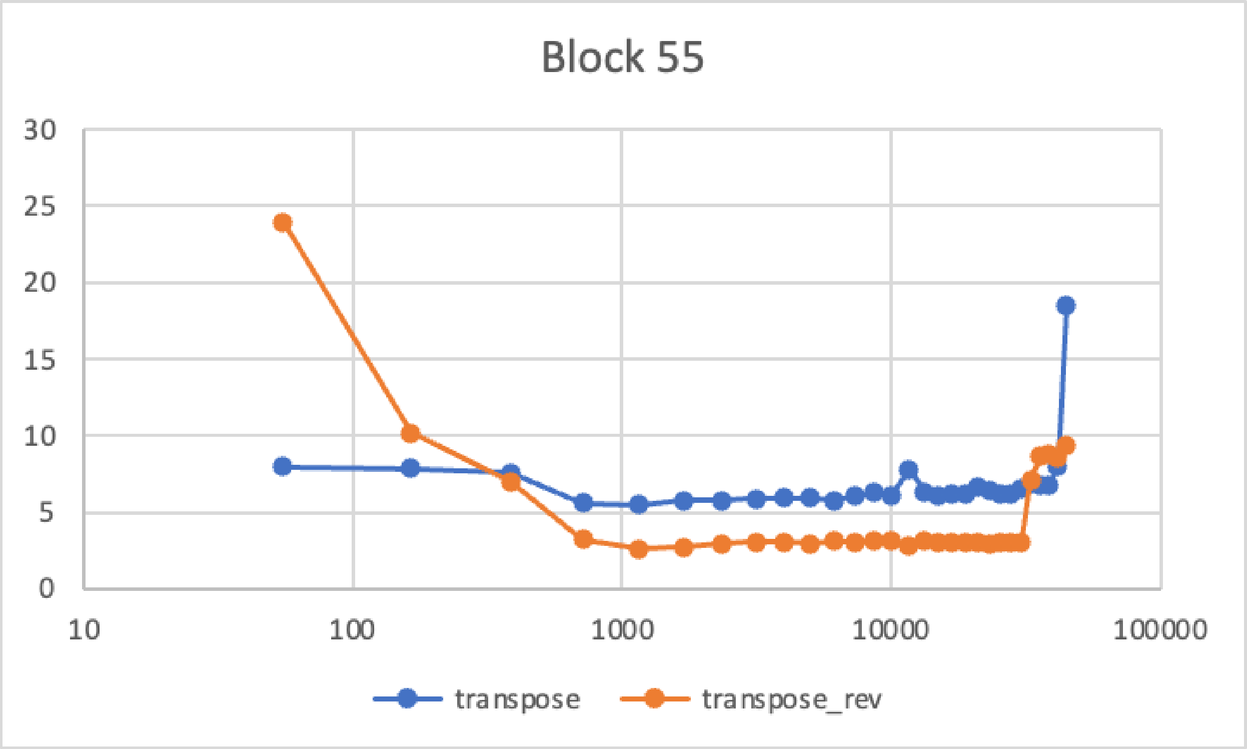
25 blocks:



40 blocks:



Block 50:


Block 55
30
25
20
15
10
5
0
10
100
1000
10000
100000
transpose
transpose_rev

Part 6

6a. 20 hours
6b. No
6c. No
6d. Yes