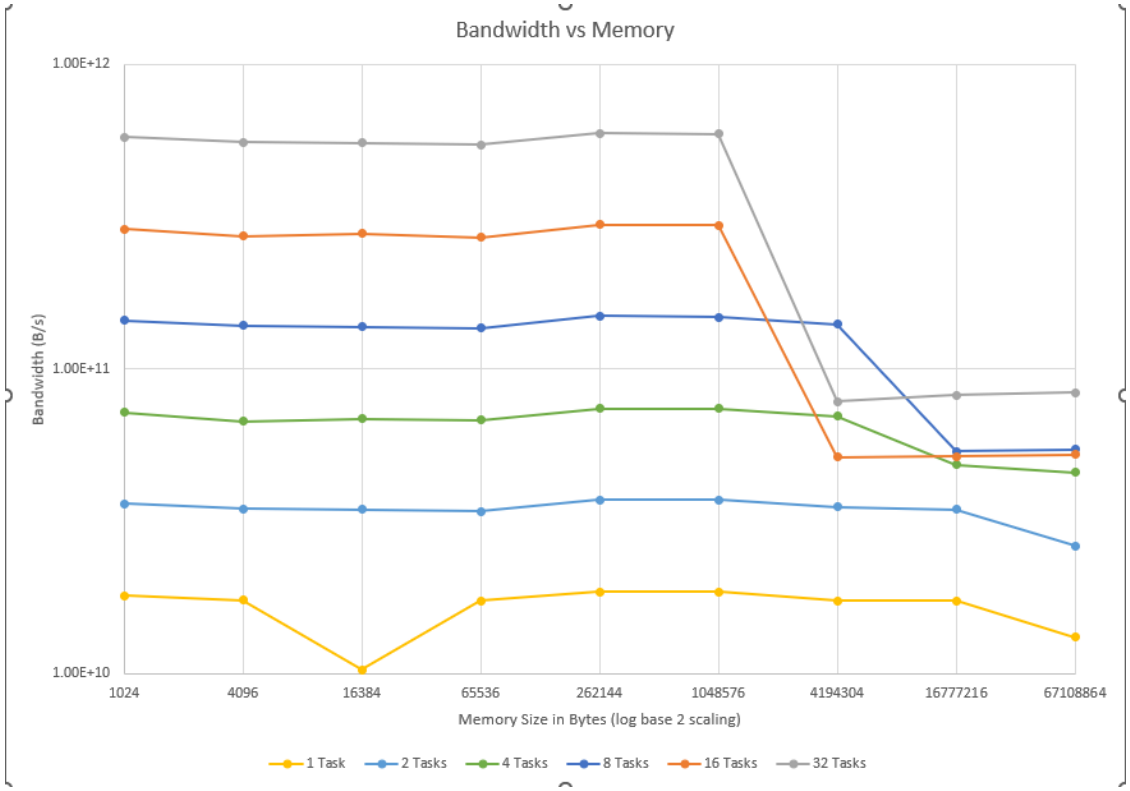
Evan Lang
Yang Lu
EC 527
Lab 1: Memory Optimization

1.



a.

tasks	memsize	bandwidt	BW/task
1	1024	1.81E+10	1.81E+10
1	4096	1.74E+10	1.74E+10
1	16384	1.03E+10	1.03E+10
1	65536	1.73E+10	1.73E+10
1	262144	1.86E+10	1.86E+10
1	1048576	1.86E+10	1.86E+10
1	4194304	1.74E+10	1.74E+10
1	16777216	1.74E+10	1.74E+10
1	67108864	1.32E+10	1.32E+10

b. The lowest observed bandwidth is 1.03E+10 B/s. This occurred at memory size 16384 B. The highest bandwidth occurred at 1048576 B, with a measured bandwidth of 1.86E+10 B/s. The ration between the lowest and highest is 1 : 1.809.

c. The highest bandwidths observed all occurred at 32 tasks, between 1024B memory size and 1048576 B memory size. The absolute highest measured of

these occurred at 262144 memory size, with a bandwidth of 5.95E+11 B/s.

32	1024	5.77E+11	1.80E+10
32	4096	5.55E+11	1.73E+10
32	16384	5.54E+11	1.73E+10
32	65536	5.45E+11	1.70E+10
32	262144	5.95E+11	1.86E+10
32	1048576	5.93E+11	1.85E+10
32	4194304	7.85E+10	2.45E+09
32	16777216	8.23E+10	2.57E+09
32	67108864	8.37E+10	2.61E+09

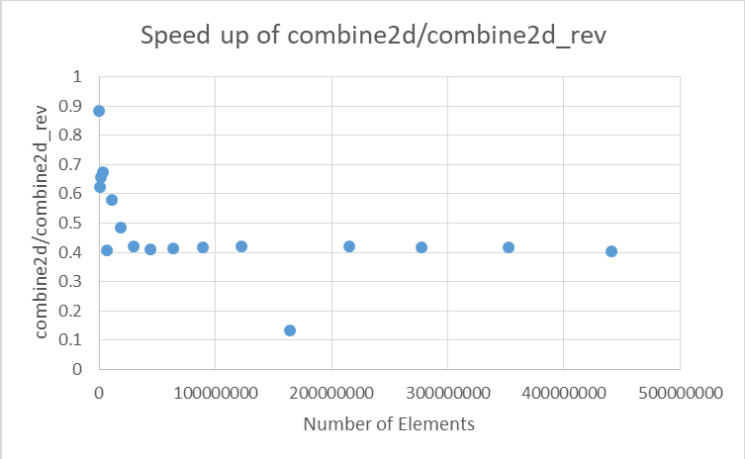
d. For the highest memory size at one task a bandwidth of 1.32E+10 B/s was observed. The same bandwidth per task was observed at 2 tasks. By 4 tasks the Bandwidth per task degraded to 1.15E+10 (B/s)/task. This is the most amount of tasks that maintained a E+10 (B/s)/tasks, making it the largest amount of tasks you can run that maintain a close value to the bandwidth per task of running one task.

2.

a. Note all data can be found in Lab1Q2.xlsx

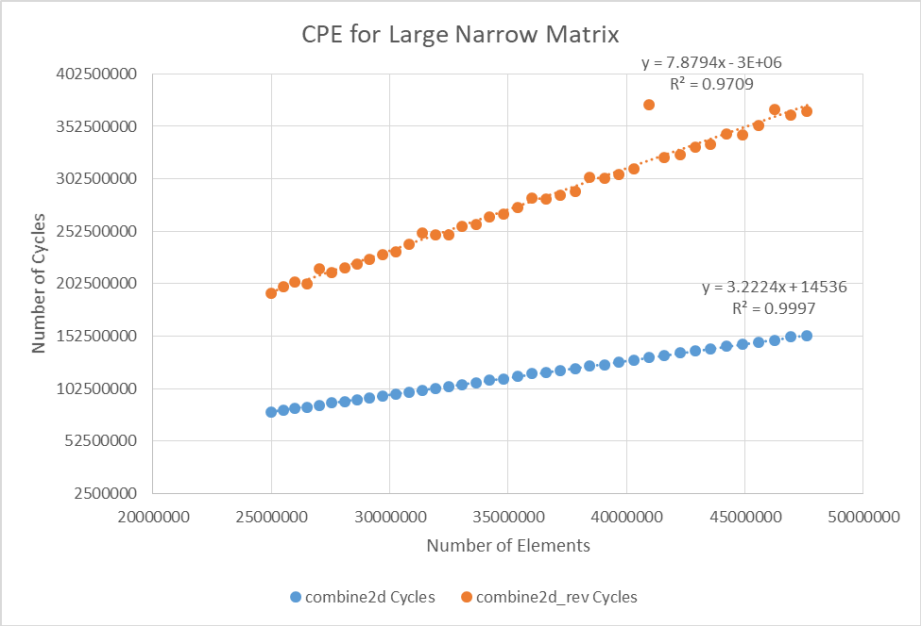
i. With a matrix containing approximately 9.22E9 elements, the compiler will start running into execution issues and take a large amount of time to run and/or crash.

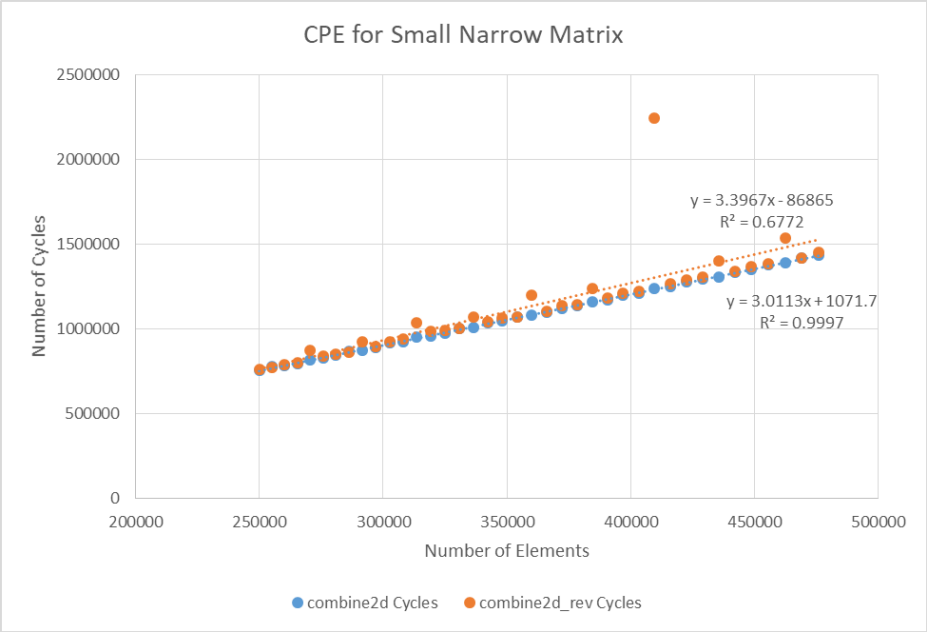
ii. combine2D is faster by roughly 40% when compared to combine2D_rev.



iii. The graph starts displaying interesting behavior after roughly 4.5E6 elements. Specifically, the CPE of combine2d_rev after this point starts increasing and zig-zagging after having previously leveled out. I believe that this behavior is due to the cache needing to access the same block repeatedly, hence, causing the cache to flush the potential spatially cached blocks.

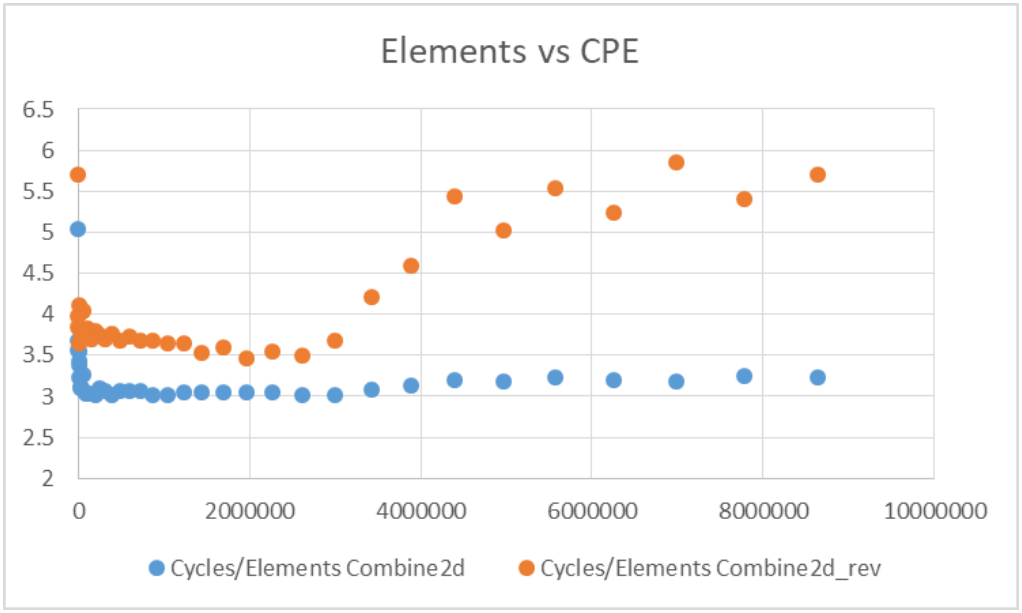
b.





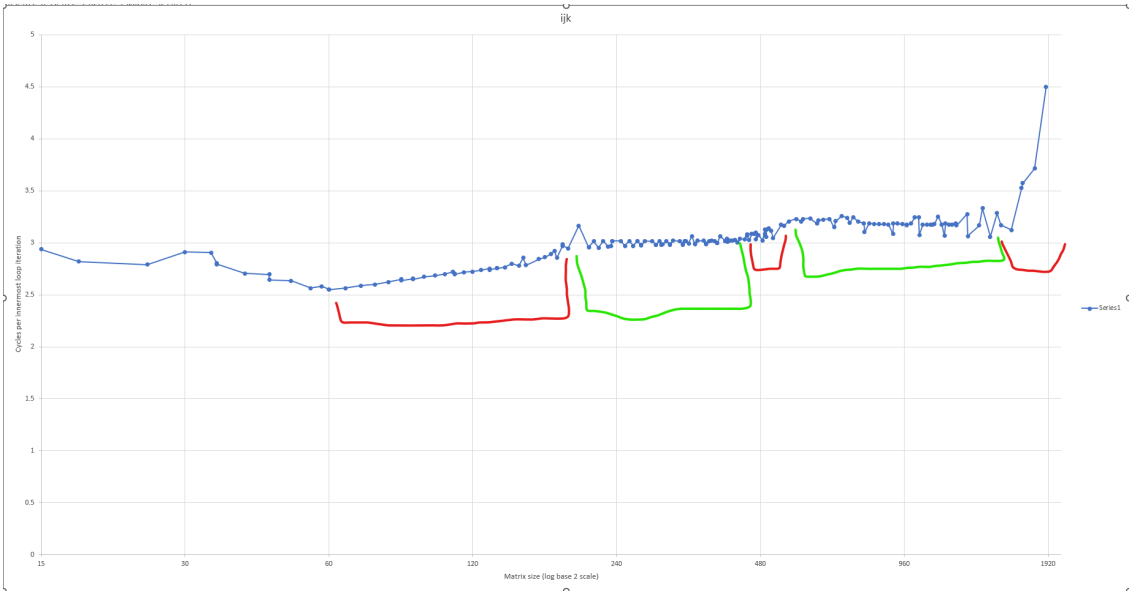
The CPE for combine2d and combine2d_rev for small narrow matrices are relatively similar. However, for a large number of elements, the CPE for combine2d_rev balloons to nearly double it's small matrix equivalent. The CPE for combine2d on the otherhand only increases minimally even for larger matrix sizes. This behavior can be explained by the fact that since combine2d is column oriented, the number of misses in cache caused by using a column oriented loop will start increasing as element size increases. This behavior in turn also drives up the number of cycles needed per instruction, which ultimately causes CPE to increase.

c.



For the elements vs CPE graph, it starts off with a high CPE for both combine2d and combine2d_rev. The reason for this is because at small matrix sizes, the cache will encounter more misses simply because the matrix is too small for the processor to leverage a cache's spatial locality. However, as the matrix size gets bigger, the advantages of spatial locality start applying. This is especially the case for combine2d as it's CPE even at above 8,000,000 elements barely changes. The reason combine2d_rev is different after around 2,000,000 elements is because the cache starts encountering cache misses due to the inefficiencies of column oriented looping. As a result, CPE will start increasing as the number of elements increases.

3. *Important note, data was gathered using row length as the x-axis rather than the matrix size. In order to convert between these values please cube the row length to find its corresponding matrix size.*

a. 

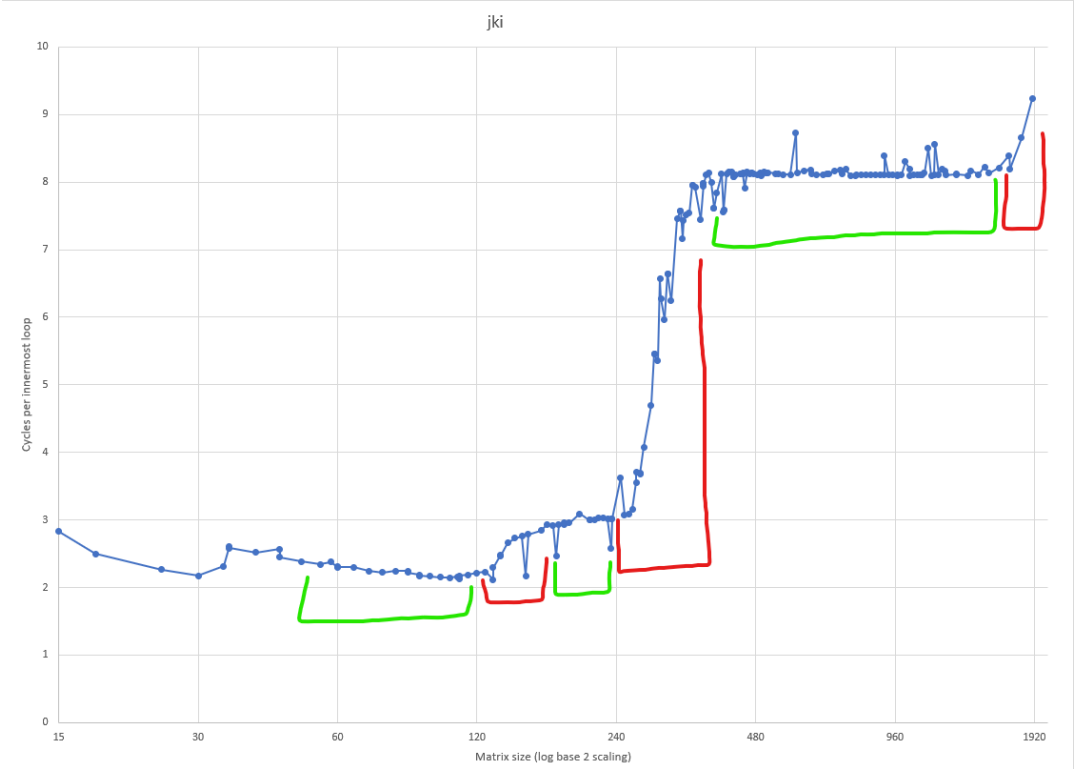
Transitions are marked in red, plateaus are marked in green. All collected data and graphs can be found in the included excel document Lab1Q3.xlsx

i. There are 2 noticeable plateaus. The first occurs between the row lengths of 185 and 435. The second takes place between the row lengths 550 and 1230.

ii. For the first plateau between the matrix sizes of 185 and 435, the cycles per innermost loop iterations is approximately 3. For the second plateau between the row lengths 550 and 1230, the cycles per innermost loop iterations is approximately 3.2.

iii. There are 3 transitions, the first between the matrix sizes 60 and 180, the second between the row lengths 440 and 500. Near the end of bounds at a row length of 1235 the cycles per innermost loop iterations begins to increase sharply as the matrix size is reaching the limits of the cache.

b.



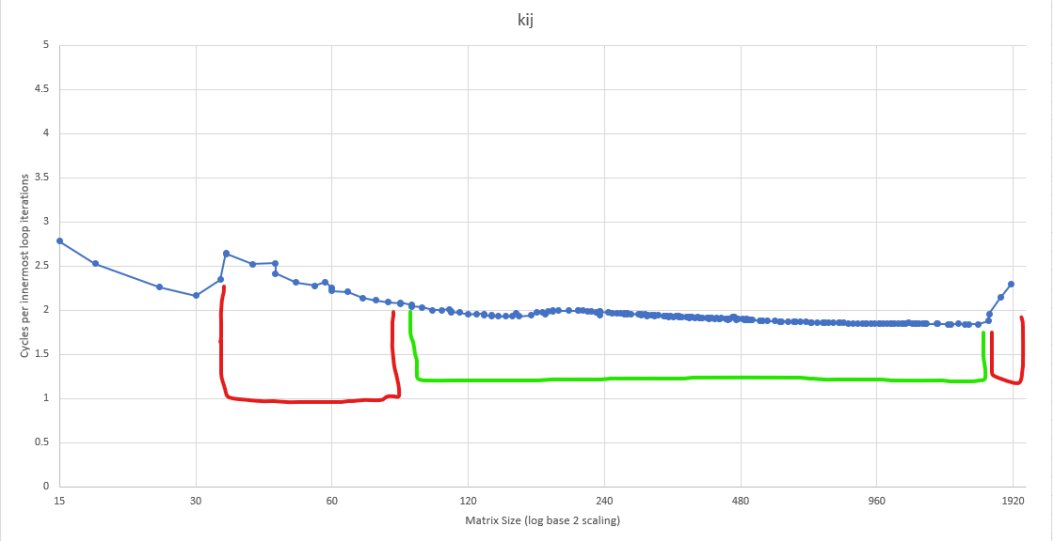
Transitions are marked in red, plateaus are marked in green. All collected data and graphs can be found in the included excel document Lab1Q3.xlsx

i. There are 3 noticeable plateaus. The first occurs between the row lengths of 55 and 120. The second takes place between the row lengths 170 and 235. The third takes place between 400 and 1530.

ii. For the first plateau between the matrix sizes of 55 and 120, the cycles per innermost loop iterations is approximately 2.3. For the second plateau between the row lengths 170 and 235, the cycles per innermost loop iterations is approximately 3. For the third plateau between the row lengths of 400 and 1350, the cycles per innermost loop iterations is approximately 8.11.

iii. There are 3 transitions, the first between the matrix sizes 125 and 170, the second between the matrix sizes 240 and 380. Near the end of bounds at a row length of 1400 the cycles per innermost loop iterations begins to increase sharply as the matrix size is reaching the limits of the cache.

c.



Transitions are marked in red, plateaus are marked in green. All collected data and graphs can be found in the included excel document Lab1Q3.xlsx

i. There is one noticeable plateau, found between the row lengths (cubed root of the matrix size) of 110 and 1600

ii. The approximate cycles per innermost loop iteration within the plateau is 1.925, with an initial average of 1.95, a middle average of 1.91, and an ending average of 1.86.

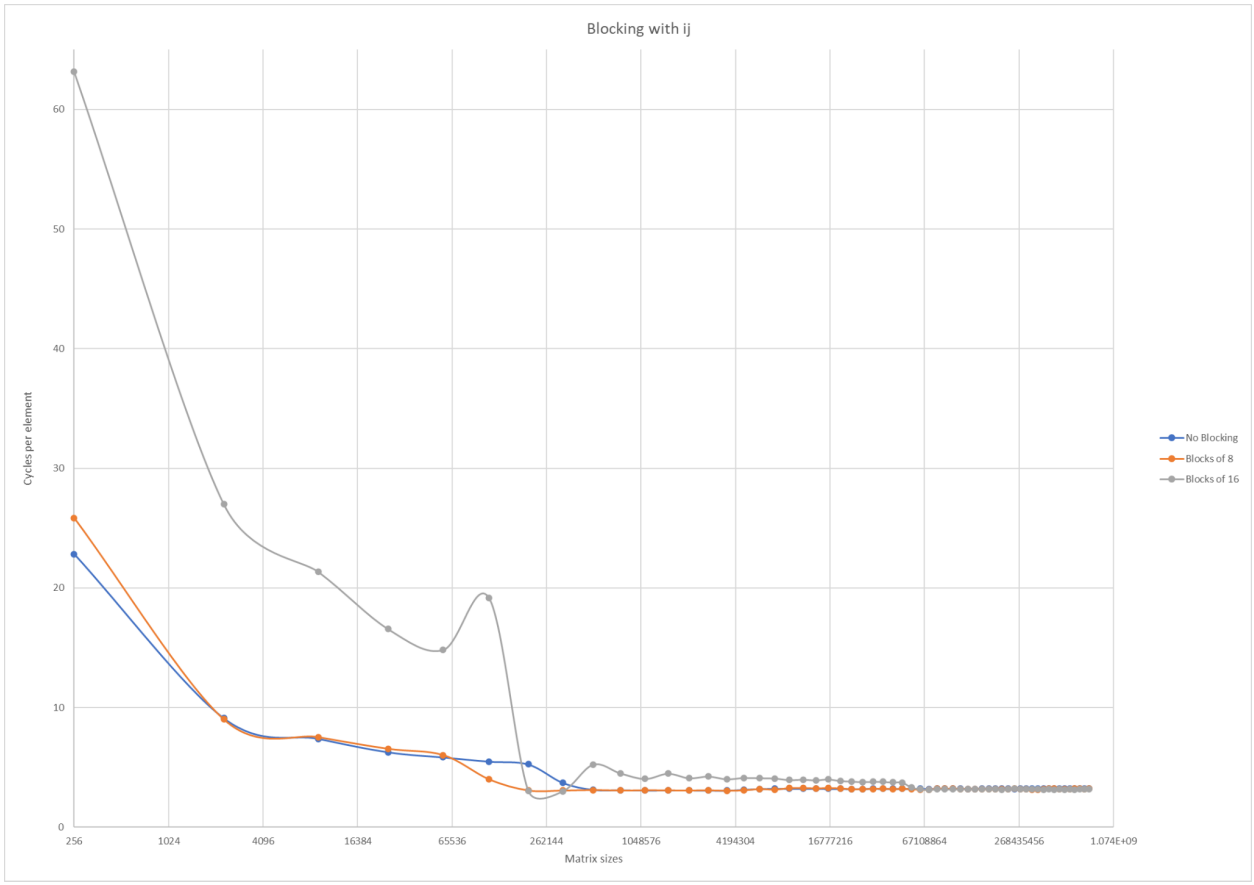
iii. There are 2 noticeable transitions in kij. The initial transition between matrix sizes of 15 (lowest measured without data ruined by initial memory misses) and 100. Near the end of bounds at a row length of 1400 the cycles per innermost loop iterations begins to increase sharply as the matrix size is reaching the limits of the cache.

4.

All collected data and graphs can be found in the included excel document Lab1Q3.xlsx. Test_mmm_block was run with varying block sizes, ranging from 8 to 64. The run measured cycles from matrix sizes of zero (which is ignored) to the closest value to 800, in intervals matching the block size used in each run. The blocked code can be found at lines 275-298 within test_mmm_block.c and the new parameter bsize at line 18.

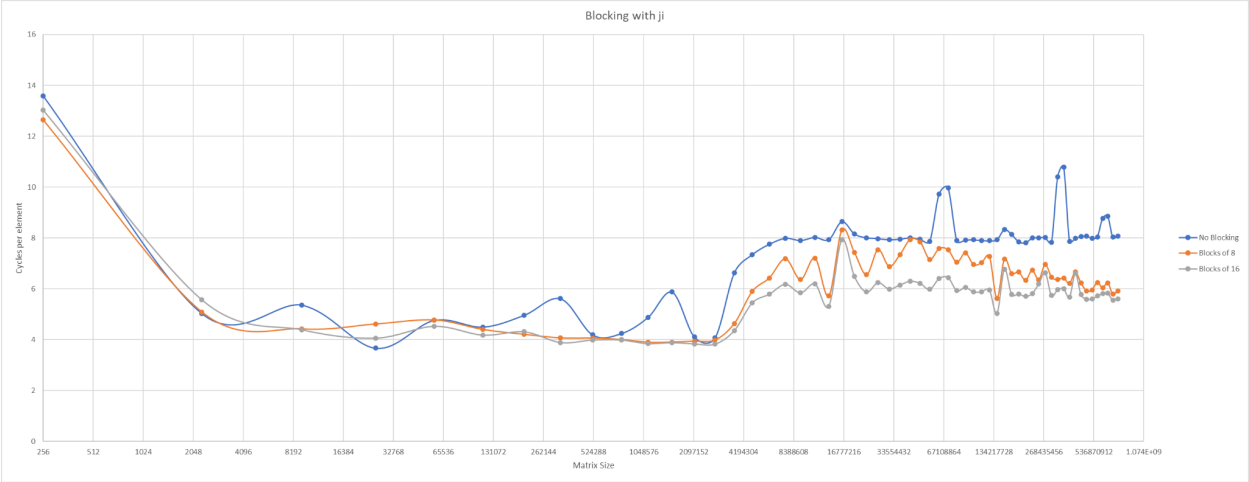
a. Ignoring the spikes at specific matrix values (256,384,512,768) blocking prevented the increase in cycles per innermost loop iteration as matrix size increased. This can be seen when comparing the 'no blocking' data to the rest. Specifically, when comparing the blocks of 8 data to no blocking, the lack of a 'transition' period where the cycles per innermost loop iterations increases in no blocking is most evident. Ultimately this means that block prevents cycle increases as matrix size increases.

b. As shown in the data, using blocks of 16 is by far the optimal block size. Its cycle performance ranged from 10% to 35% better than the rest of the block sizes. Its massive performance advantage was maintained even as matrix size increased, with the cycle performance only experiencing slight degradation when compared to performance at smaller matrix sizes.

5. 

*Blocking with **ij** above, the blue is no blocking, orange is block sizes of 8, gray is block sizes of 15.*

   a. (This is actually 5b) For all 3 methods the smaller array sizes are in a period of transition finishing around a matrix size 518400. After this a period of plateau occurred where all methods' CPE experienced little change. It should be noted that using blocks of 16 resulted in the worst performance of all of the values, especially at smaller matrix sizes.



*Blocking with **ji** above, the blue is no blocking, orange is block sizes of 8, gray is block sizes of 16.*

b. Both methods of blocking experienced a plateaus at lower matrix sizes, with only small changes in their cpe. Non blocking experienced the most volatility in this region. All 3 methods then experienced a transition starting at matrix size 2822400 and ending at 7485696. Both blocking methods had greater performance than the non blocking run, with blocks of 16 reporting the lowest CPE.

C.

```
data_t combine2D(array_ptr v)
{
  long int i, j, jj, ii;
  long int length = get_row_length(v);
  data_t *data = get_array_start(v);
  data_t accumulator;

  /* Start with 0 or 1 (for adding or multiplying respectively) */
  accumulator = IDENT;
  for(ii=0; ii < length; ii += bsize){
    for(jj=0; jj < length; jj += bsize){
      for (i = ii; i < ii+bsize; i++) {
        for (j = jj; j < jj+bsize; j++) {
          accumulator = accumulator OP data[i*length+j];
        }
      }
    }
  }

  /* We are done, return the answer */
  return accumulator;
}

/* Combine2D_rev:  Like combine2D but the loops are interchanged. */
data_t combine2D_rev(array_ptr v)
{
  long int i, j, jj, ii;
  long int length = get_row_length(v);
  data_t *data = get_array_start(v);
  data_t accumulator;

  /* Start with 0 or 1 (for adding or multiplying respectively) */
  accumulator = IDENT;
  for(jj=0; jj < length; jj += bsize){
    for(ii = 0; ii < length; ii += bsize){
      for (j = jj; j < jj+bsize; j++) {
        for (i = ii; i < ii+bsize; i++) {
          accumulator = accumulator OP data[i*length+j];
        }
      }
    }
  }

  /* Return the answer */
  return accumulator;
}
```

*The blocked **ij** and **ji** functions in test_transpoce.c, visible at lines 243 - 288*

Blocking was accomplished by nesting the for loops in another set of nested for loops which increment by the block size. The 2 original for loops operate between each interval of the block size.This should break the runs down into more consumable memory sizes for the cache to handle and prevent cache misses. As shown in our data this did not help in the case of the **ij** loops, as they already enjoy high spatial locality based on how they operate. In fact, blocking hindered performance at low matrix sizes. However **ji**, which does not enjoy the same spatial locality as **ij**, benefited noticeably especially at the larger block size of 16. Throughout the runs the blocked CPE was lower than non-blocked CPE. *Not shown in the graphs above is blocking*

using block sizes of 32. In both test cases the results with a block size of 32 fell between the values for 8 and 16. It was removed from the graphs within the report to maintain visual clarity, however the data and graphs containing performance with a block size of 32 can be found in Lab1Q5.xlsx. Ultimately for **ij** the best performance for blocking was at a size of 8. Its performance is extremely similar to the non blocking performance. For **ji** the best performance by far was obtained using a block size of 16.

6.

a. This lab took us about 10 hours to finish.
b. Each part of the lab felt fair. There wasn't any part that took an unreasonable amount of time to finish.
c. All the necessary skills needed to complete this lab were mentioned in class.
d. The only issue encountered was the wording on some of the problems. For example, in part 2, it asks to "Get data for two narrow ranges of array sizes, one small and one large." This was confusing as the values and size of a "narrow", "small", and "large" array is subjective. Generally speaking, we found ourselves constantly needing to spend a few extra minutes trying to understand the wording and word choices.