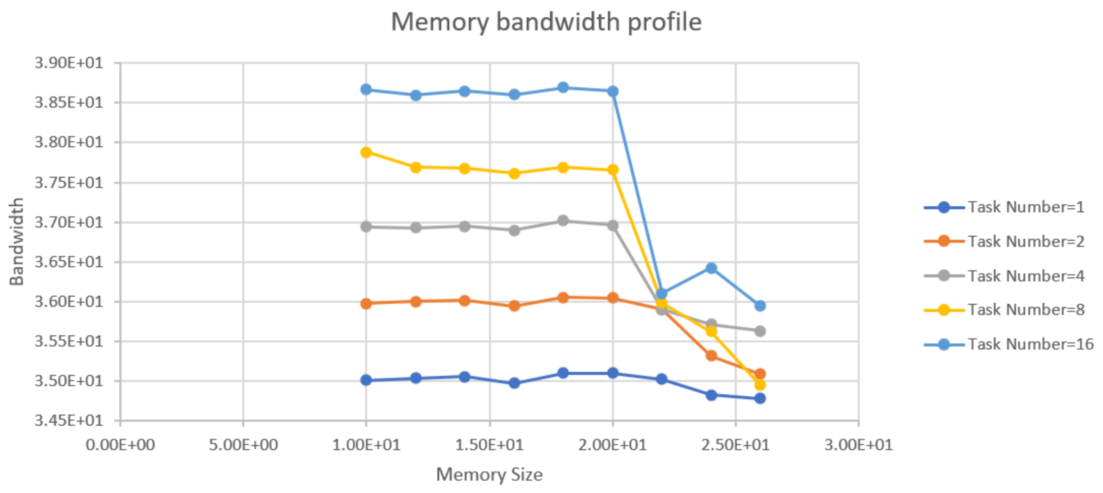
Lab 1 — Memory Optimization

Part 1. Memory bandwidth profile of your system



1a:

The memory size of 1048576 gives the highest bandwidth.

1b:

The memory size of 67108864 gives the lowest bandwidth.

Across all the 1-task tests, the ratio between the highest and lowest bandwidths is 1.249.

1c:

16 tasks and 262144 memory size gives the highest total bandwidth.

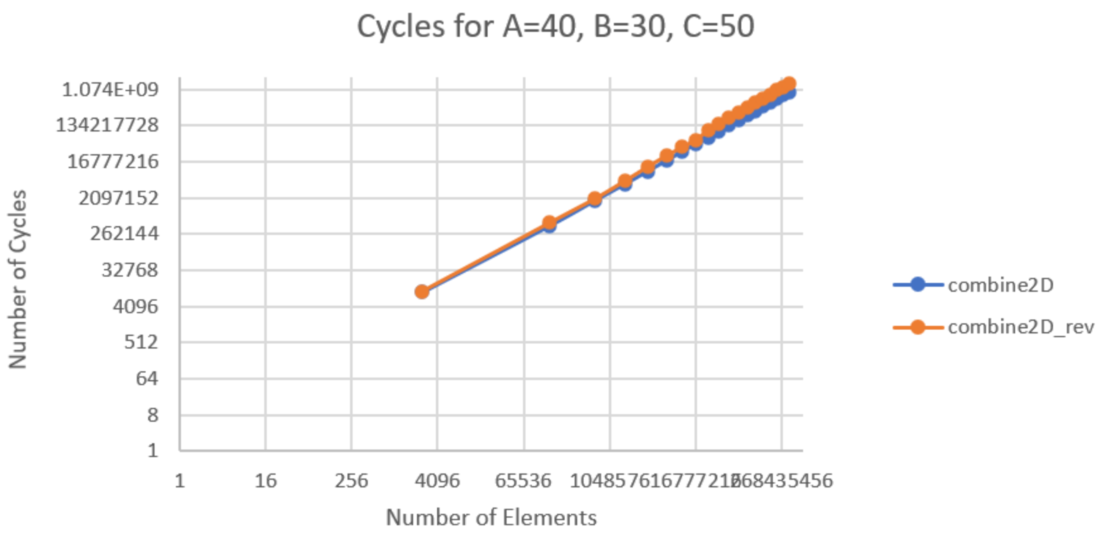
1d:

When there is 1 task, and using the largest memory size, the bandwidth is 2.95e10. As the data shown in the output, even though there are 2 tasks, each task cannot get the bandwidth as large as 1 task.

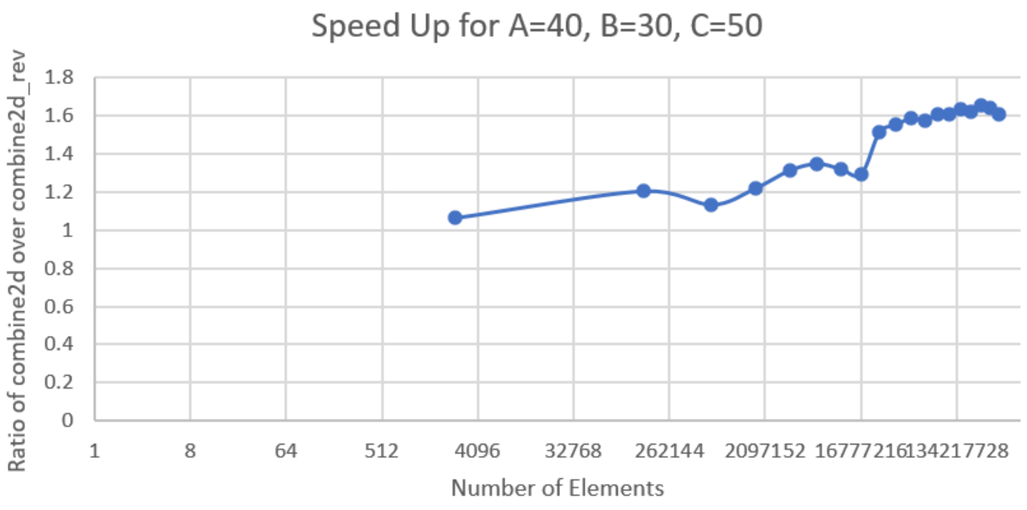
For example, when there are 2 tasks, each task can get the bandwidth of 1.82e10, and when there are 4 tasks, each task can only get 1.33e10.

Part 2: Testing code transformations — Task: Optimize combining data from 2D arrays

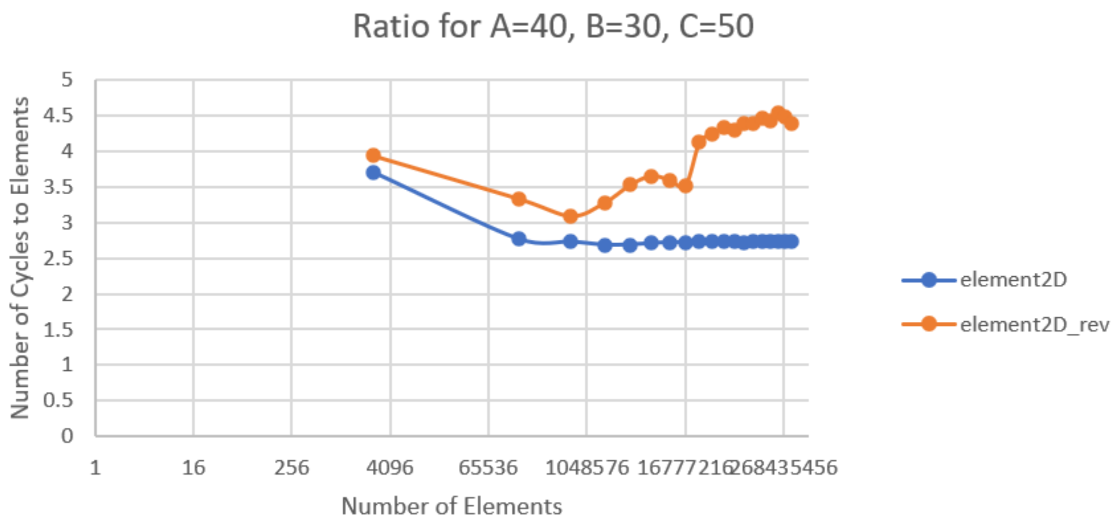
```
#define A     40   /* coefficient of x^2 */
#define B   30   /* coefficient of x */
#define C 50   /* constant term */
```



Cycles for A=40, B=30, C=50
Number of Cycles
1.074E+09
134217728
16777216
2097152
262144
32768
4096
512
64
8
1
1
16
256
4096
65536
1048576
16777216
268435456
Number of Elements
combine2D
combine2D_rev

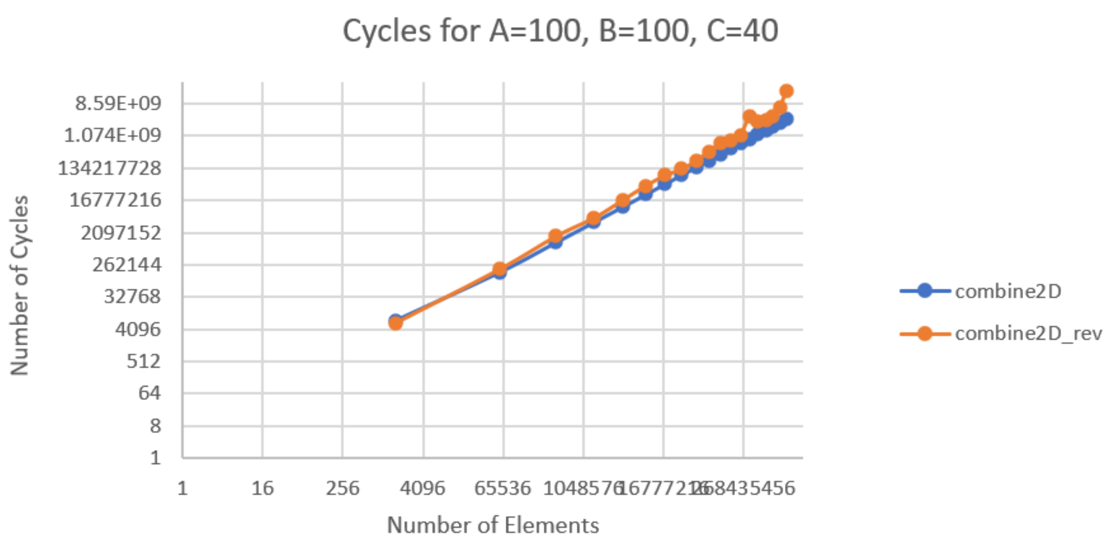


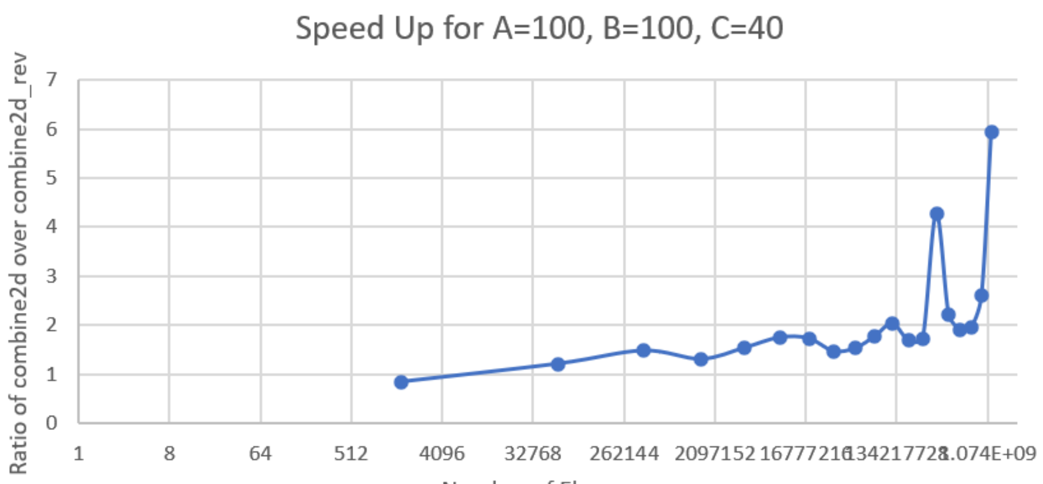
Speed Up for A=40, B=30, C=50
Ratio of combine2d over combine2d_rev
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
1
8
64
512
4096
32768
262144
2097152
16777216
134217728
Number of Elements

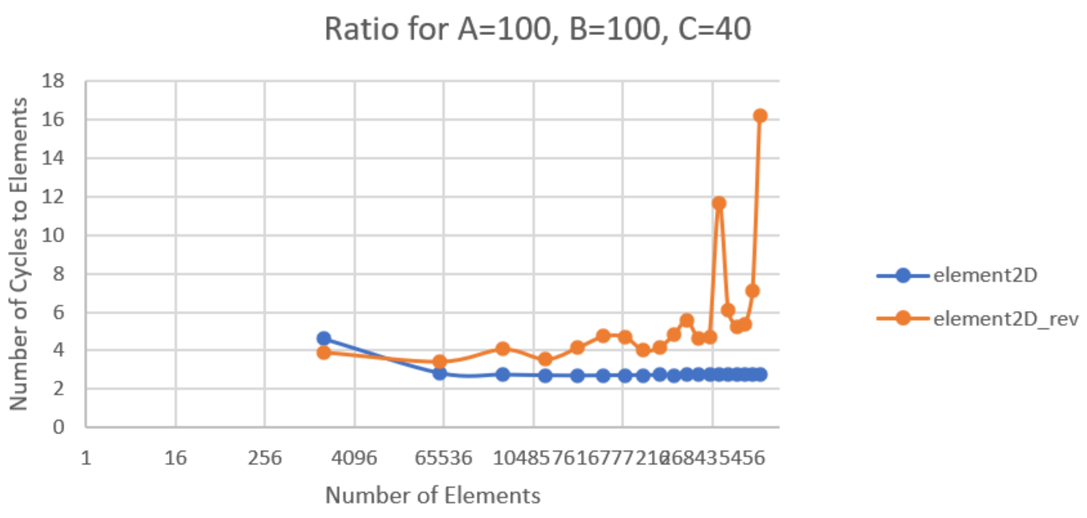


Ratio for A=40, B=30, C=50
Number of Cycles to Elements
5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
1
16
256
4096
65536
1048576
16777216
268435456
Number of Elements
element2D
element2D_rev

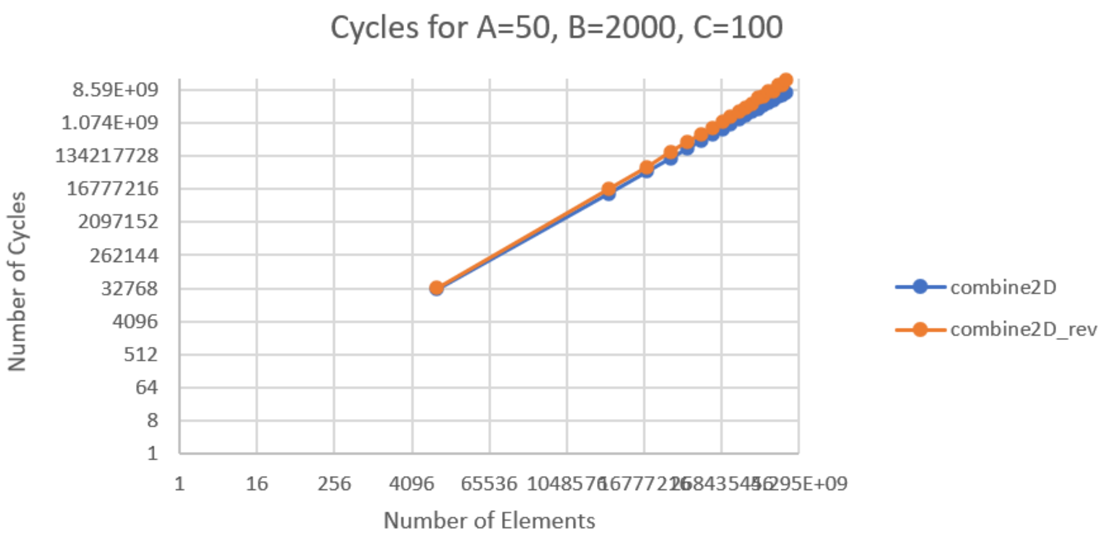
```
#define A     100   /* coefficient of x^2 */
#define B   100   /* coefficient of x */
#define C 40   /* constant term */
```

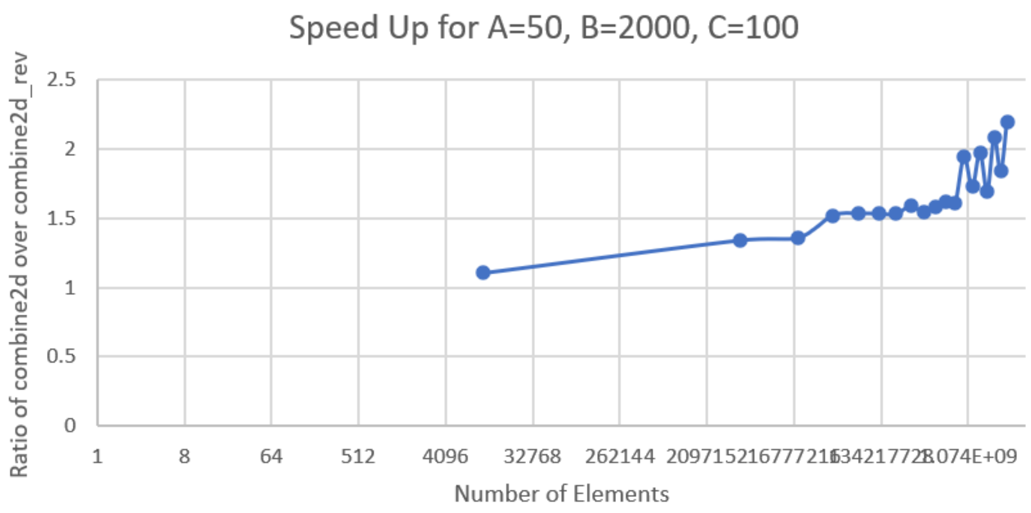


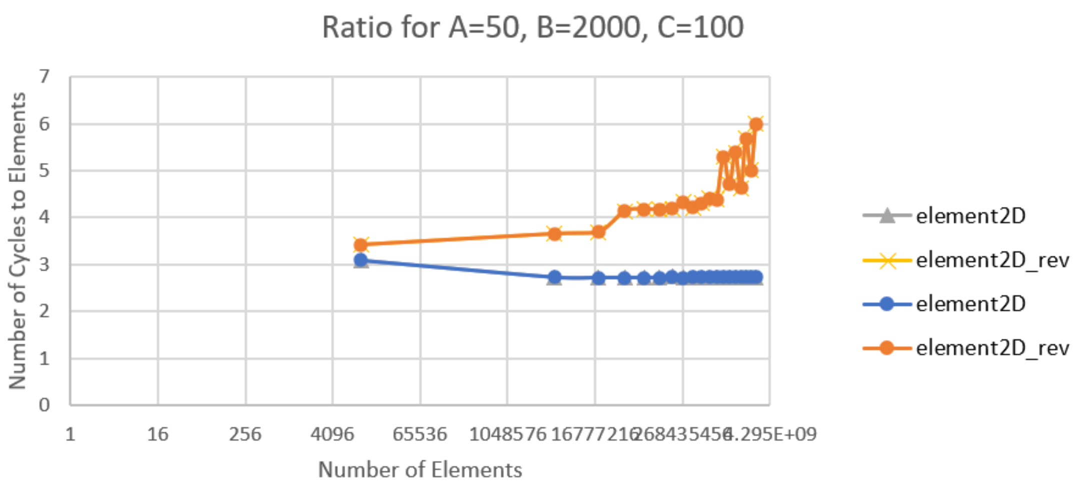




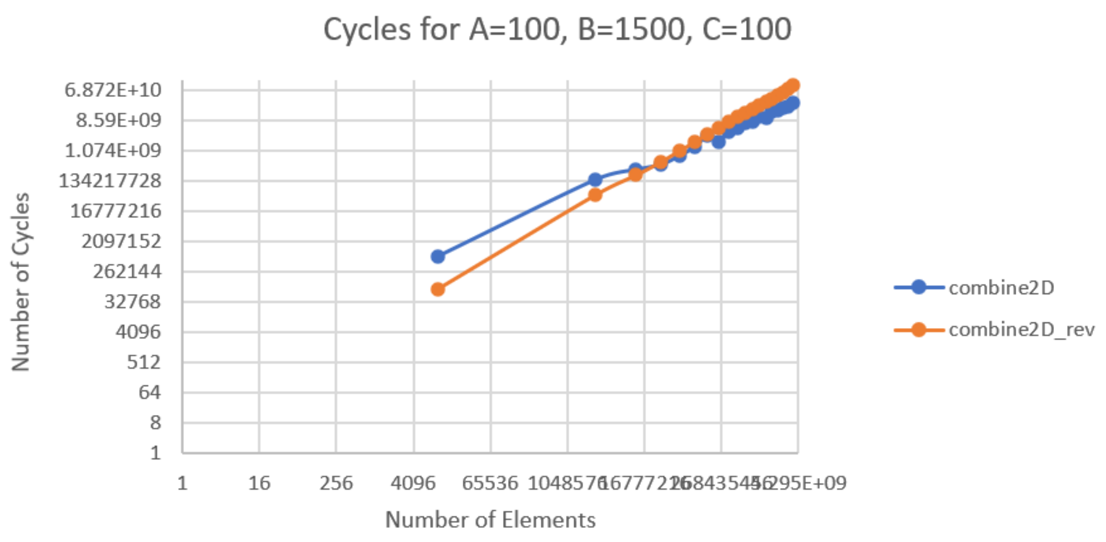
```
#define A    50   /* coefficient of x^2 */
#define B  2000   /* coefficient of x */
#define C 100   /* constant term */
```

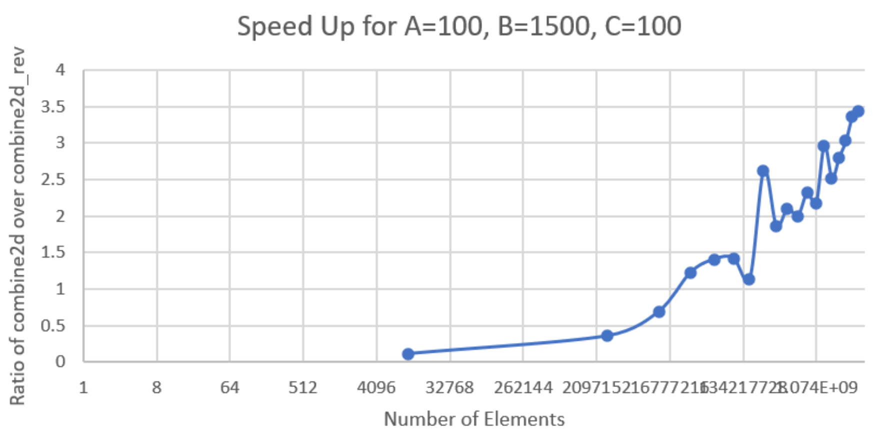


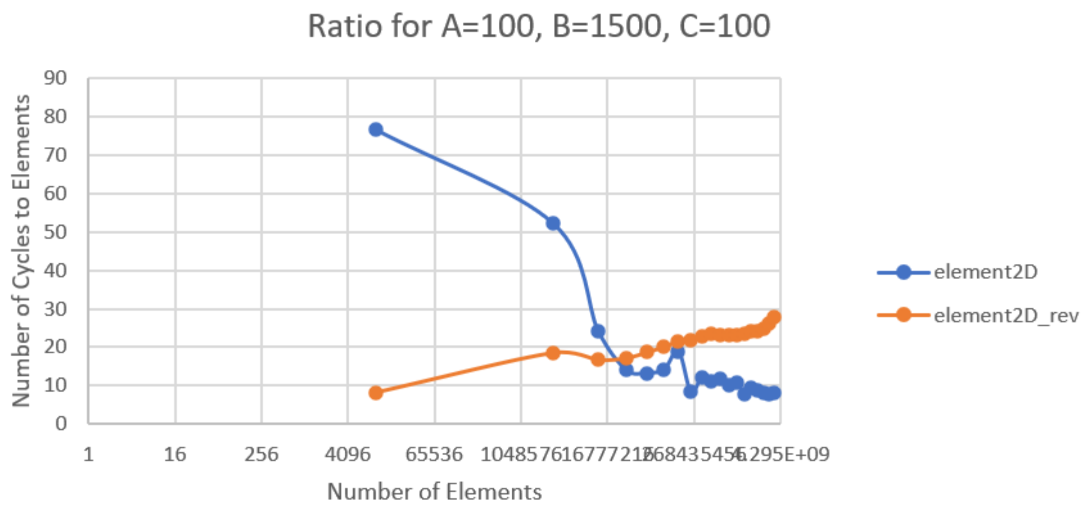




```
#define A     100   /* coefficient of x^2 */
#define B   1500   /* coefficient of x */
#define C 100   /* constant term */
```







2a:

When size >80000. The codes report error saying "COULDN'T ALLOCATE Xe+10 BYTES STORAGE" when I set A=100, B=100 and C=40000, where the overall size is 82000.

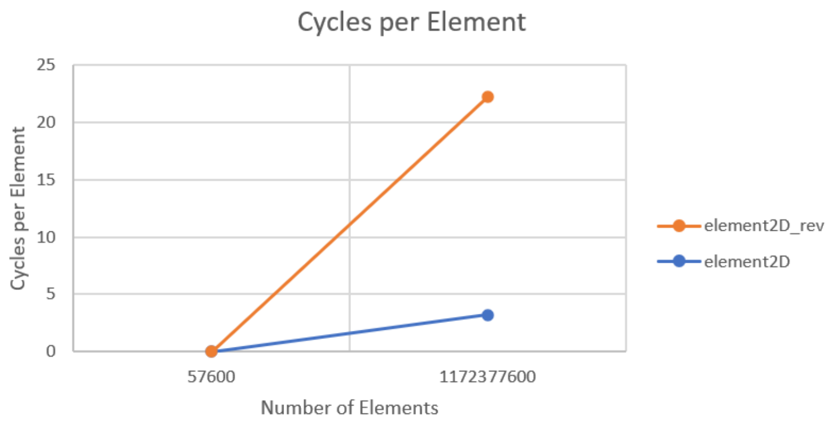
Also, when the size is closed to 60000 to 70000, it runs very slowly. In order to get to the boundary, I kept doubling the A and B to make the codes run into execution problems.

The element2D function is quicker. It is faster by 1.6 ratio.

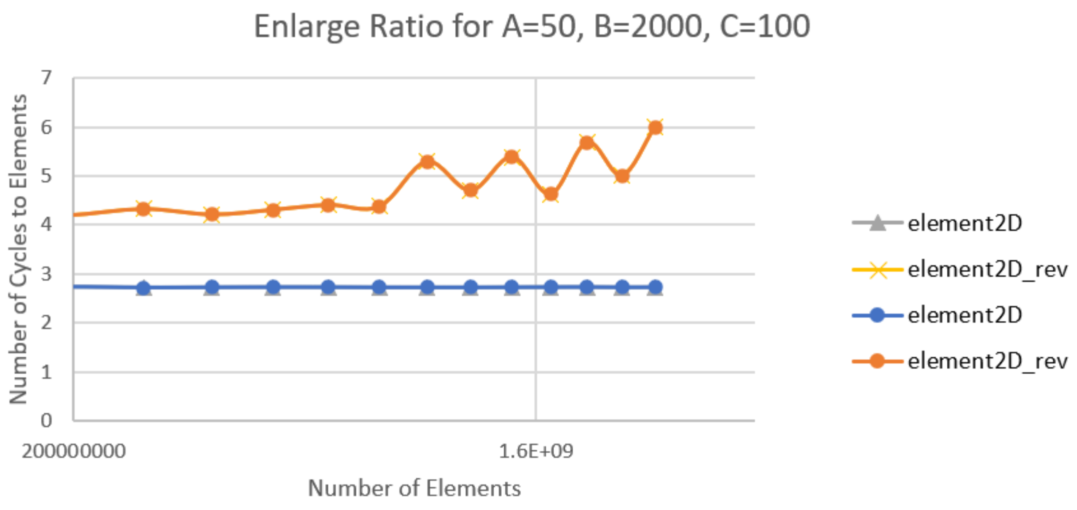
When the array size is larger than 3000, there becoming something unusual. That is to say, in the ratio image, there comes some sudden transition.

2b:

The element2D_rev has a higher CPE since in the for loop in combine2d_rev the function first get into the loop of j then i, thus according to the textbook it will miss much more times than the element2D. So, the cycles per element for the element2D_rev are much more than for the element2D.



2c:

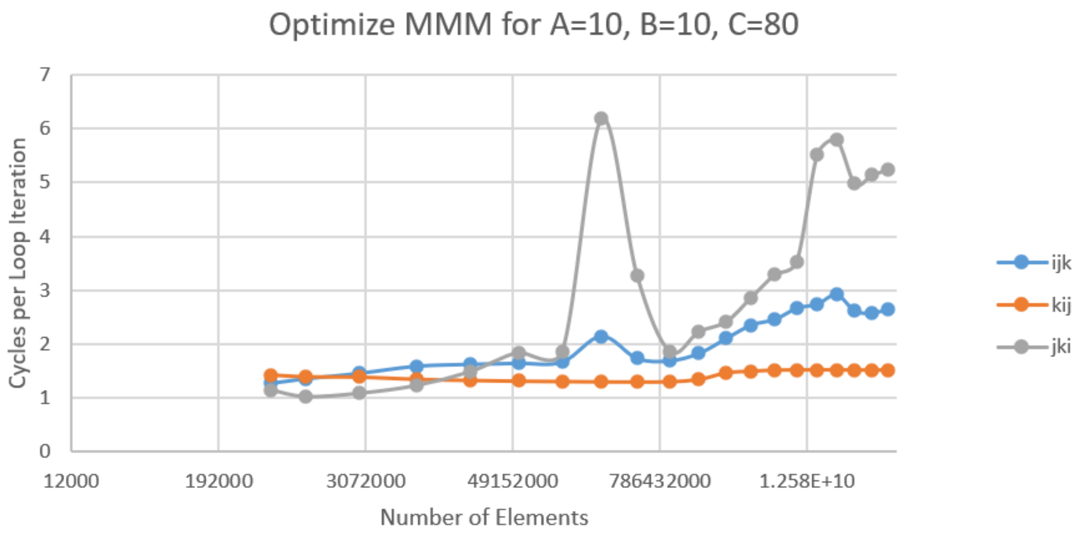


For making it more clearly to see, I enlarge the area from the above result to see the

transition areas, ranging from 2e08 to 1.6e09.
The cycles to element ratio (part d) may be improving for a short while because there is spatial or temporal locality taking place in the function. It may be spike up after the dip, as a new cache block is brought in from memory.
The ratio for combined2D_rev (part e) might be more jumpy owing to accessing memory more often than the regular combined2D function. And the very jumpy behavior is probably because of the cache utilization of the program's access pattern.

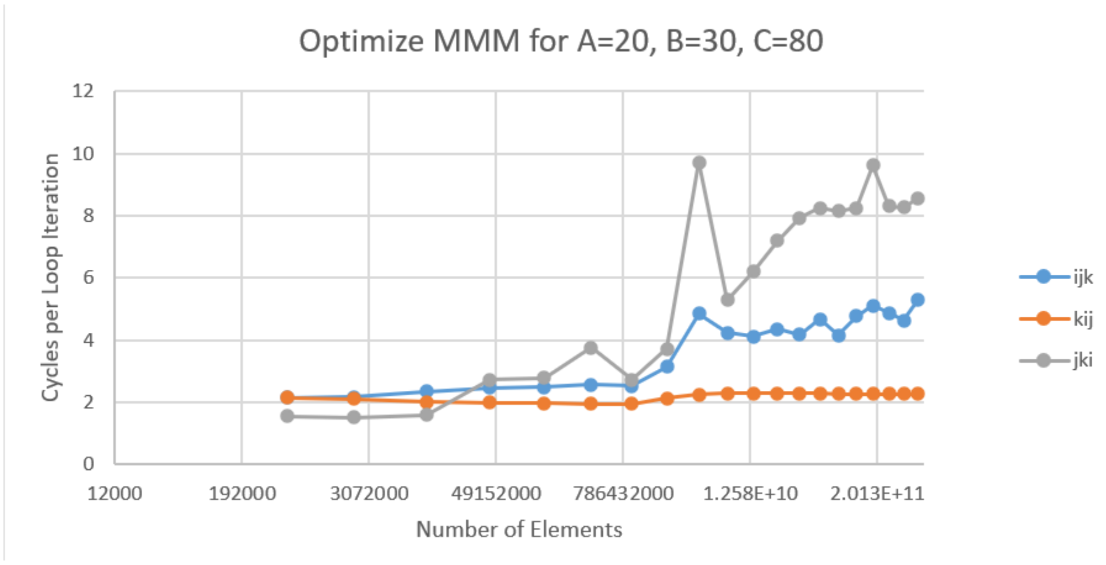
Part 3: Use these ideas on a more complex code — Task: Optimize MMM using loop interchange.
3a:
If the matrix size is larger than 2K, the total data size is calculated as 2000*2000*8=31MB, which is larger than the L3 cache. While the L3 cache is 12288K, which is 12MB, so that it is larger than the cache.

```
#define A    10   /* coefficient of x^2 */
#define B    10   /* coefficient of x */
#define C    80   /* constant term */
```



For "ijk" loop permutation, it has 2 plateaus. As the point for each plateau is (27.96,2.15) and (34.35,2.91). The number of cycles per innermost loop iteration is 2.15 and 2.91.
For "kij" loop permutation, there are no plateaus since the whole function increases gradually.
For "jki" loop permutation, there are 2 plateaus, At the point for each plateau is (27.96, 6.18) and (34.35, 5.79). The number of cycles per innermost loop iteration is 6.18 and 5.79.

```
#define A    20   /* coefficient of x^2 */
#define B    30   /* coefficient of x */
#define C    80   /* constant term */
```

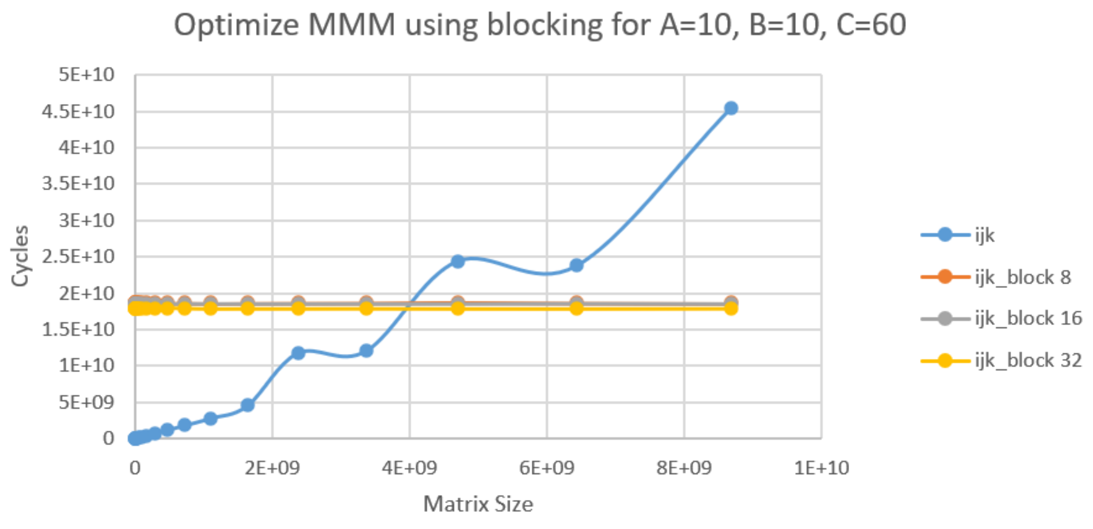


For “ijk” loop permutation, it has 2 plateaus. As the point for each plateau is (28.53, 2.56), (31.93, 4.83) and (37.41, 5.11). And the number of cycles per innermost loop iteration is 2.56, 4.83 and 5.11.
For “kij” loop permutation, there are no plateaus since the whole function increases gradually.
For “jki” loop permutation, there are 2 plateaus, At the point for each plateau is (28.53, 3.76), (31.93, 9.71) and (37.41, 9.61). And the number of cycles per innermost loop iteration is 3.76, 9.71 and 9.61.

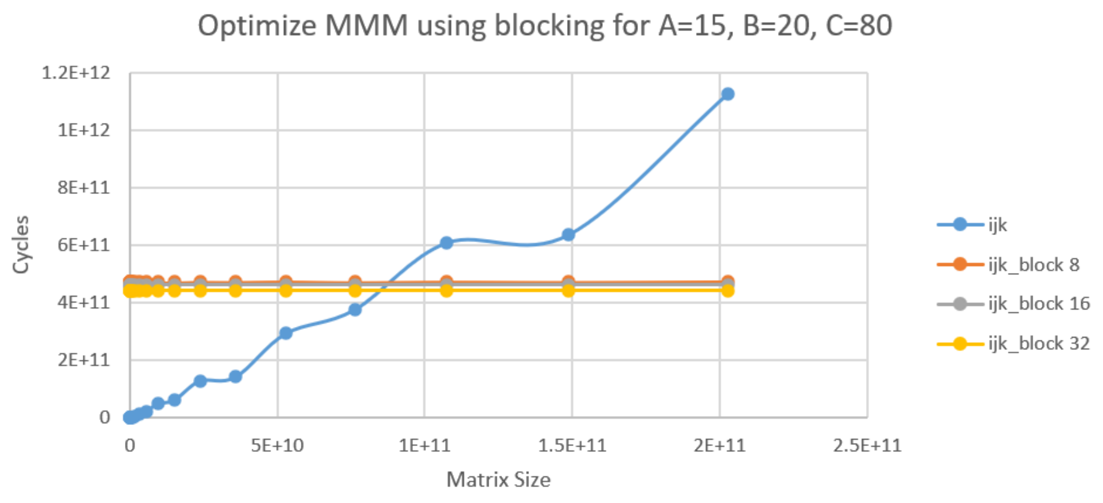
As seen from above we can conclude there are at least one plateau at each looping pattern, for the pattern of "jki", there are most spikes and it may due to the misses per iteration.
Also, as we can see from the “kij” pattern, it utilized the cache most efficiently, therefore it provides the lowest missing rate and appears very smooth compared to others.
The transitions occur when the size of the array is much larger than the cache, and the looping pattern is producing much more misses.

Part 4: Use these ideas on more complex code — Task: Optimize MMM using blocking

```
#define A     5   /* coefficient of x^2 */
#define B     10   /* coefficient of x */
#define C     60   /* constant term */
```



```
#define A     15   /* coefficient of x^2 */
#define B     20   /* coefficient of x */
#define C     80   /* constant term */
```



4a:

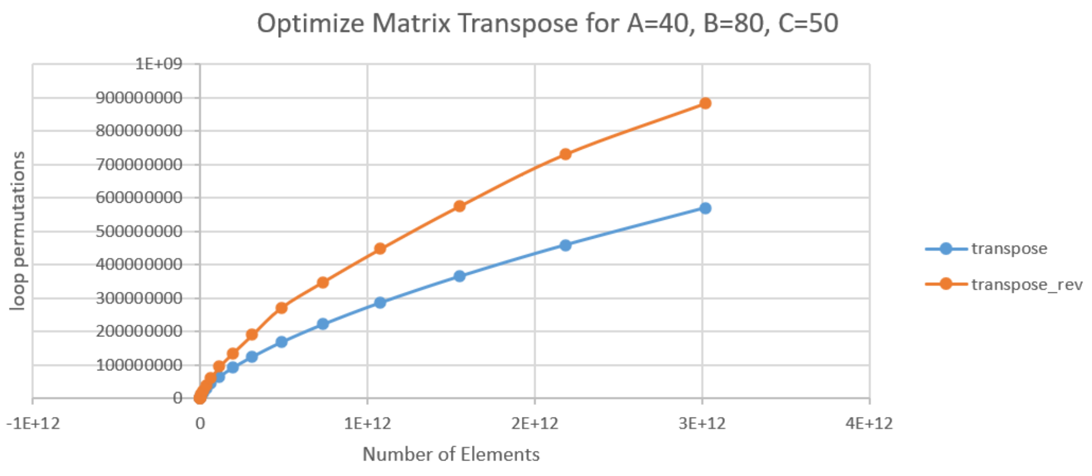
According to the graph shows, when the matrix size increases, the effect of the blocking is still significantly better than the non-blocking matrix.

However, in the cases of small matrix sizes, as the missing for the original "ijk" is not large, the performance of both two does not vary very differently. This is probably because of the effects of the block. The block helps the CPU performs better reading by putting them from the cache, but when the data size becoming large enough, the block itself will not have much usage.

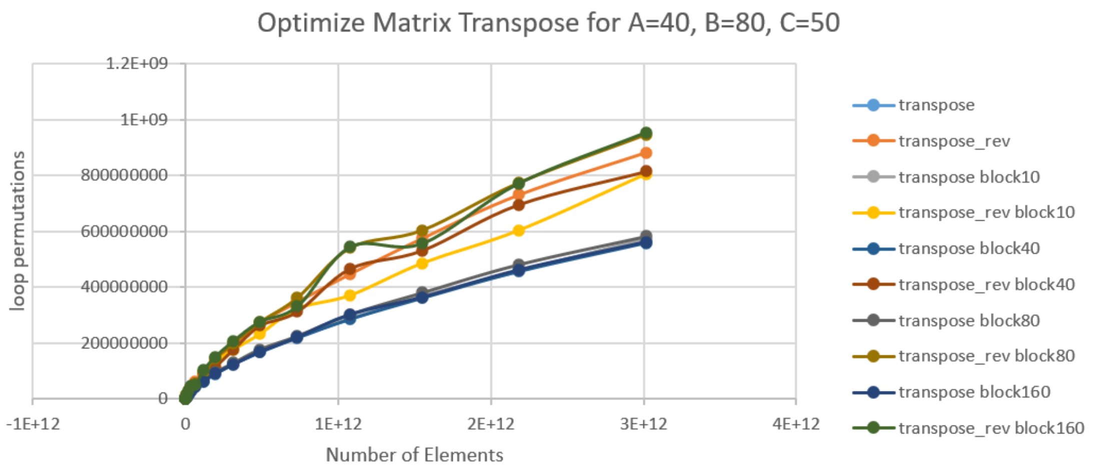
4b:
4c:
4d:
4e:
For now, the great difference between block and non-blocked is because that the matrix size is the multiplies of the block size. If we change the block size into a prime number, then it will greatly increase the misses and enlarge the distinction between the block and non-blocked method.

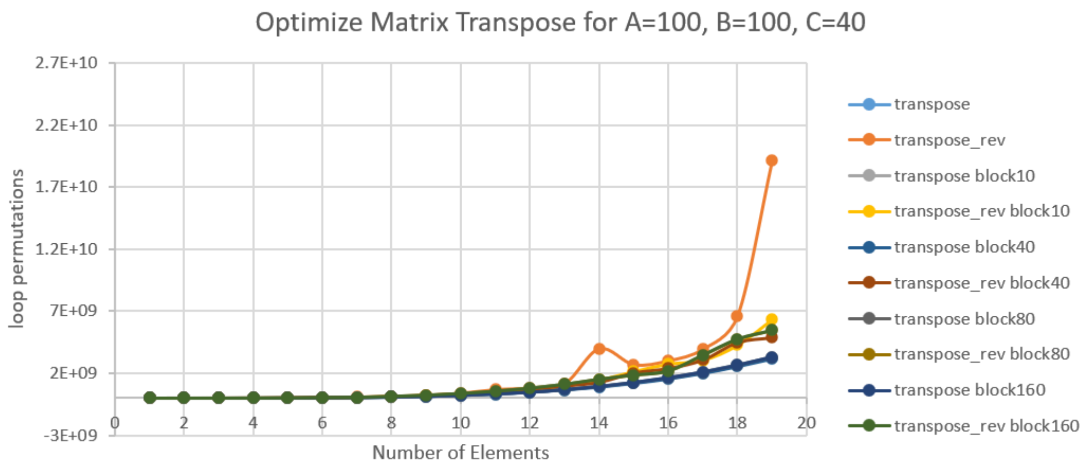
Part 5: Use these ideas on an entirely new application. Task: Optimize Matrix Transpose

```
#define A    40   /* coefficient of x^2 */
#define B    30   /* coefficient of x */
#define C    50   /* constant term */
```



As shown in the graph and output, the “ij” pattern performs better than the “ji” since I used “accumulator = accumulator OP data[i*length+j];” in the codes.





I tried several block sizes, such as 10, 40, 80 and 160 since the overall size can be divided by these numbers and thus does not affect the block miss. Also, I tried to set the block size to be 71, which cannot be divided and the result does not change much.
And as it shows in the images, when using blocks, if the block size gets large enough, it will not make the calculation very efficiently.
In conclusion, if there is not much noise, the blocking version is more efficient than the non-blocking version, since in the blocking version, once a particular block is used, it is not referenced again, whereas in the non-blocking version, the same elements may be brought into the cache multiple times.

Part 6: Quality Control
6a:
About 10 hours.
6b:
Really spend some time figuring out what Part4 wants me to do. Also, maybe be more clearly about the horizonal and vertical sizes, such as log10 or log2? Also, I really hope that TA can give more specific ideas rather than telling me the principles again.
6c:
Not for this one.
6d:
The lab is great. I really appreciate that it is open 24 hours so that I can go there whenever I want to.