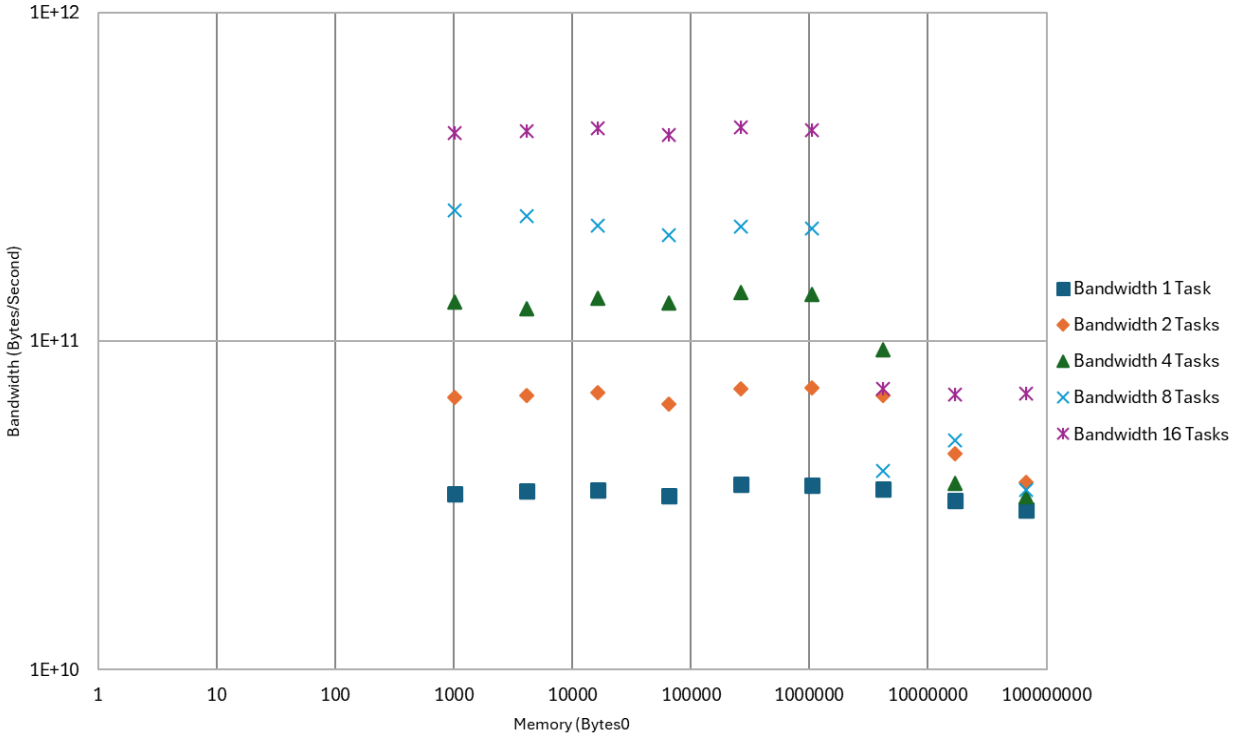
Chih Han “Josh” Yeh

Lab 1

ENGEC 527

Part 1.



a. The highest bandwidth for running 1 task is 3.681×10^{10} bytes per second at 262144 bytes of memory. The closest data bandwidth to this data point would be 3.647×10^{10} bytes per second at 1048576 bytes of memory.

b. When running 1 task, 67108864 bytes of memory generated the lowest bandwidth at 3.064×10^{10} bytes per second. The ratio between the highest and lowest bandwidth is around 1.20.

c. At 16 tasks and a memory size of 262144 bytes, it generated the highest total bandwidth of 4.494×10^{11} bytes per second. Including this data point, there are a total of 6 memory bandwidths on 16 tasks that are very close to each other (4.314×10^{11} bytes/sec at 1024 bytes, 4.369×10^{11} bytes/sec at 4096 bytes, 4.447×10^{11} bytes/sec at 16384, 4.243×10^{11} bytes/sec at 65536 bytes, 4.390×10^{11} bytes/sec at 1048576 bytes).

d. The bandwidth achieved at memory size of 67108864 Bytes with running 1 task is 3.060×10^{10} bytes per second. With the same memory size, this machine can run 4 tasks with similar memory bandwidth at around 3.36×10^{10} bytes/second.

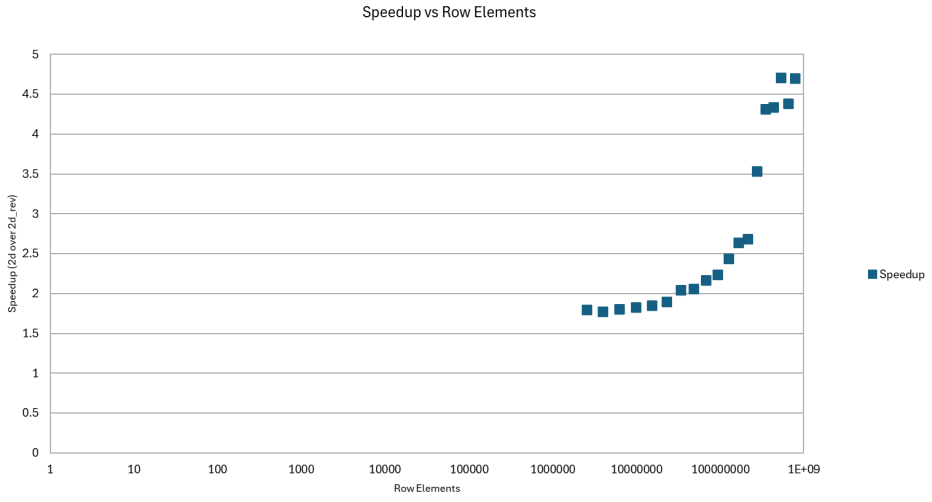
Part 2.

a.

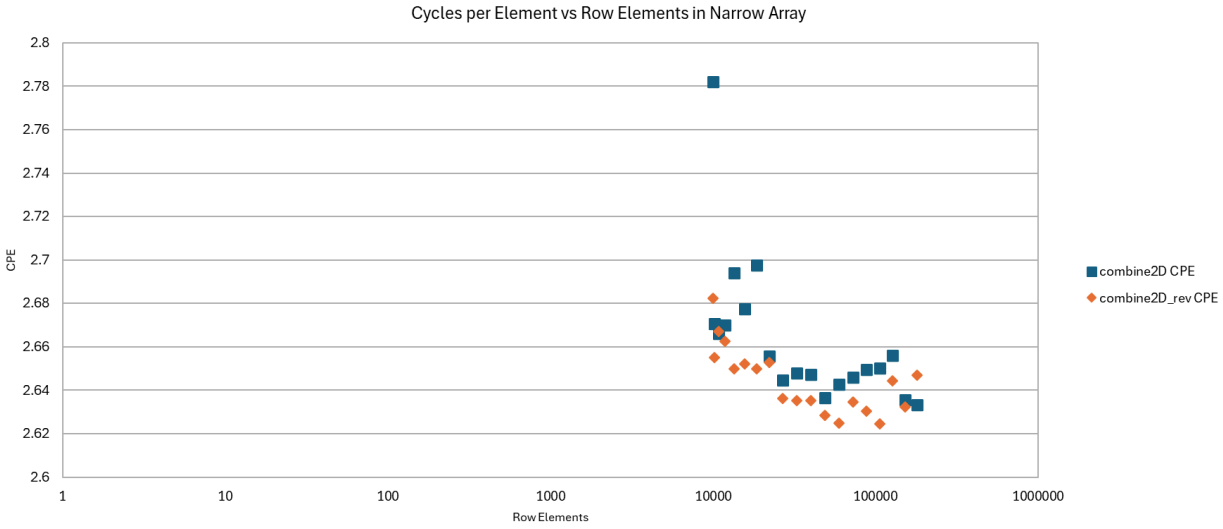
i. The coefficients of the array size that was still able to execute at a reasonable time were 64 for A, 320 for B, and 1600 for C ($64x^2 + 320x + 1600$), even though this array size seemed to already significantly slow down the machine from executing the program. An attempt to successively double the current iteration resulted in the machine not generating output for several minutes, therefore I chose 64, 320, and 1600 as my coefficients.

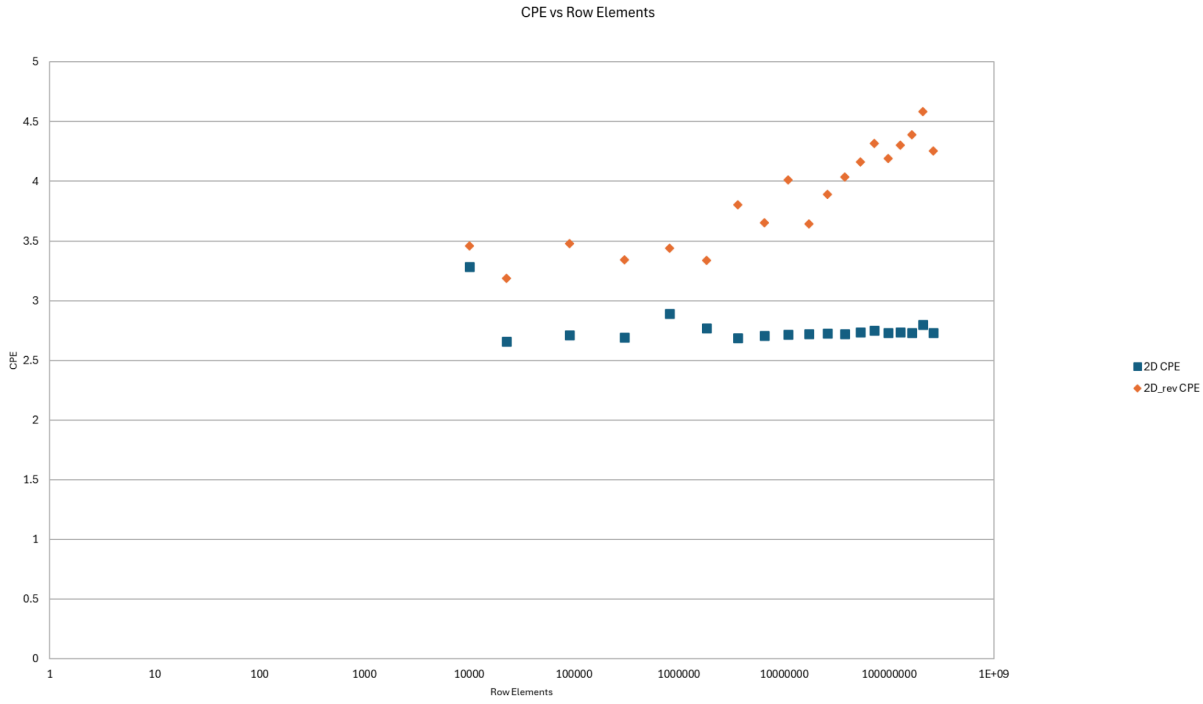
ii. The function combine2D is significantly faster than the function combine2D_rev. At larger scale computation, combine2D runs at nearly half of combine2D_rev's cycle count from row lengths ranging from 1600 to 9500. At Row lengths ranging from 10000 and beyond, the cycle count ratio between combine2D_rev and combine2D drastically increases on a logarithmic scale, from doubling to almost 5 times the cycle count.

iii. The speedup of combine2D over combine2D_rev appears to be linear (first 10 data points) with row lengths ranging from 1600 to 9664 (row elements from 2560000 to 93392896, respectively). Then, the speedup drastically increases on a logarithmic scale, with the ratio increasing from around 2.0 to 4.7.

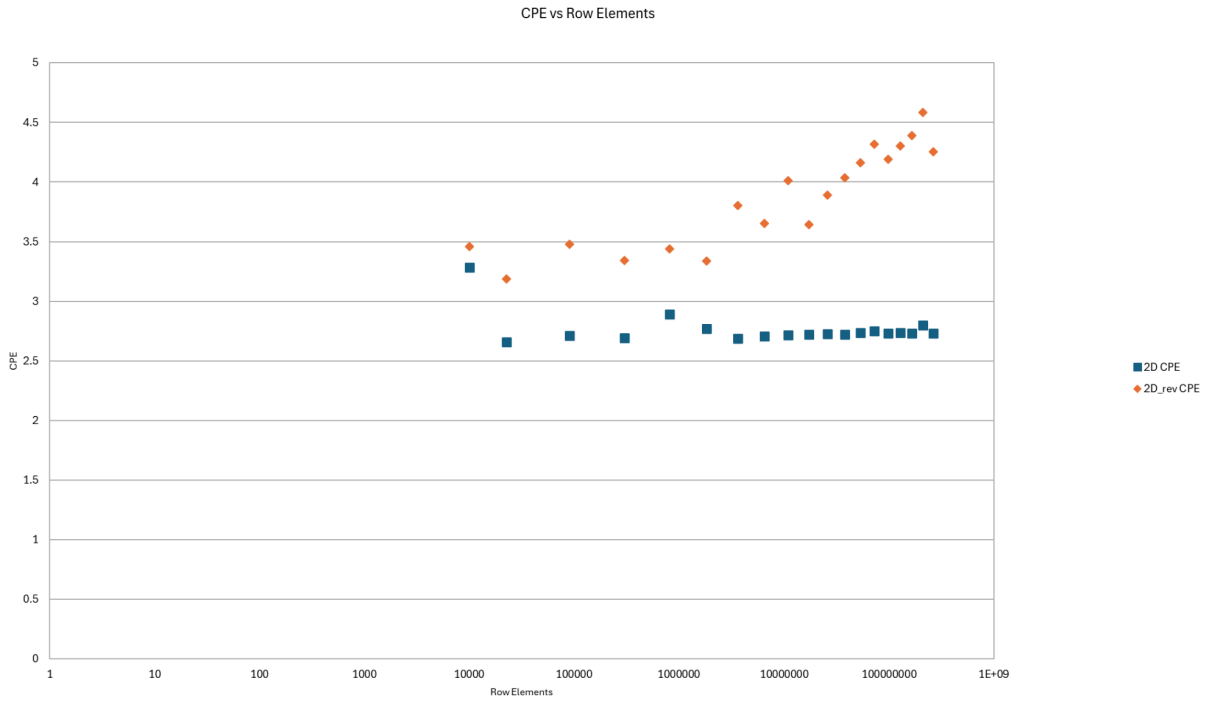


b. Under a narrow array, both methods seem to generate the same number of cycles, as the CPE between them is fundamentally the same, appearing close to each other on data. The wider array displays a large discrepancy, in which the CPE of combine2D_rev drastically increases on a logarithmic scale while combine2D seems to be the same in CPE even at larger arrays with more row elements.
Coefficients (for narrow array: A = 10, B = 0, C = 100, for wider array: A = 100, B = 0, C = 100)





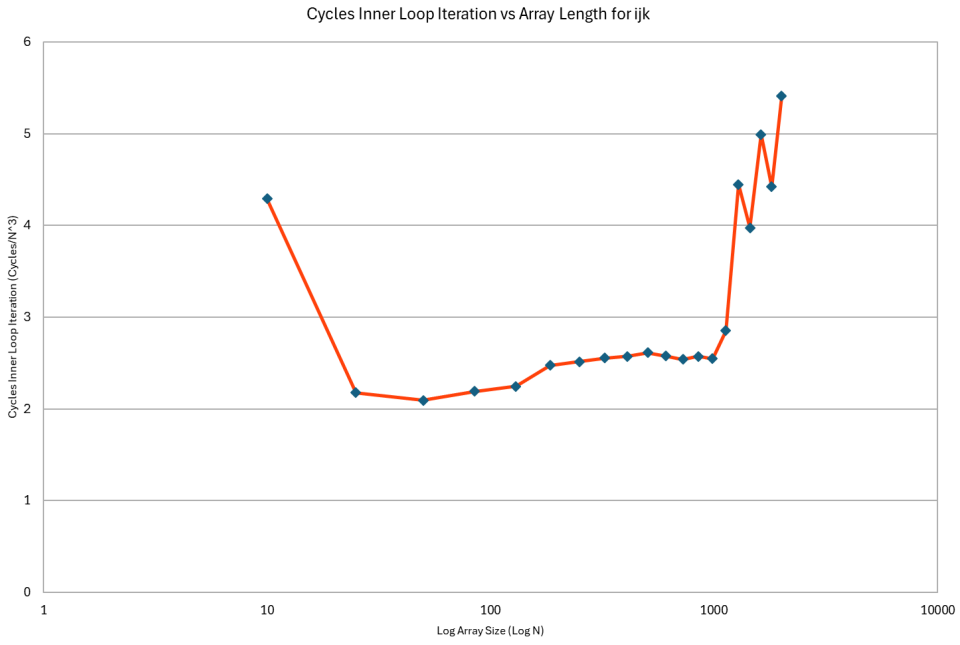
c.



Part 3.

Quadratic Equation for matrix generation: $5x^2 + 10x + 10$ (A = 5, B = 10, C = 10)

a.

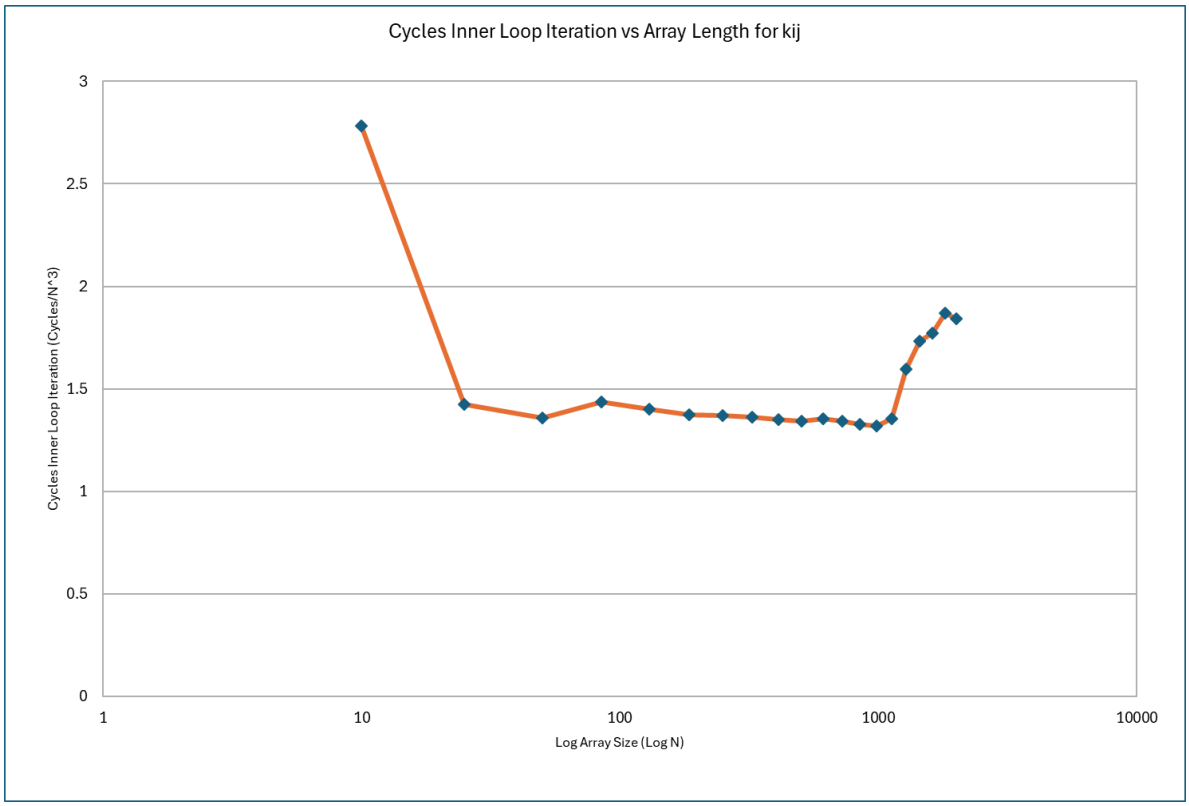


i. For the method of ijk, there only appears to be one plateau. The plateau starts approximately when the length of the array reaches above 300 (325 on the plot) and continues until the array length reaches 980 (985 on the plot), then the cycles per inner loop iteration drastically increase past that point, producing a jagged pattern whilst increasing (a /_/\ pattern on an incline).

ii. The inner loop iteration of the plateau centers around 2.50, with numbers ranging from 2.51 to 2.57. Averaging the cycles in the plateau, the number results in 2.544 cycles.

iii. As explained in part i, the transition occurs when the array length exceeds 300, or as the data points out in plot, 325 (325 x 325). The second transition occurs when the array length reaches and exceeds 985 (985 x 985).

b.

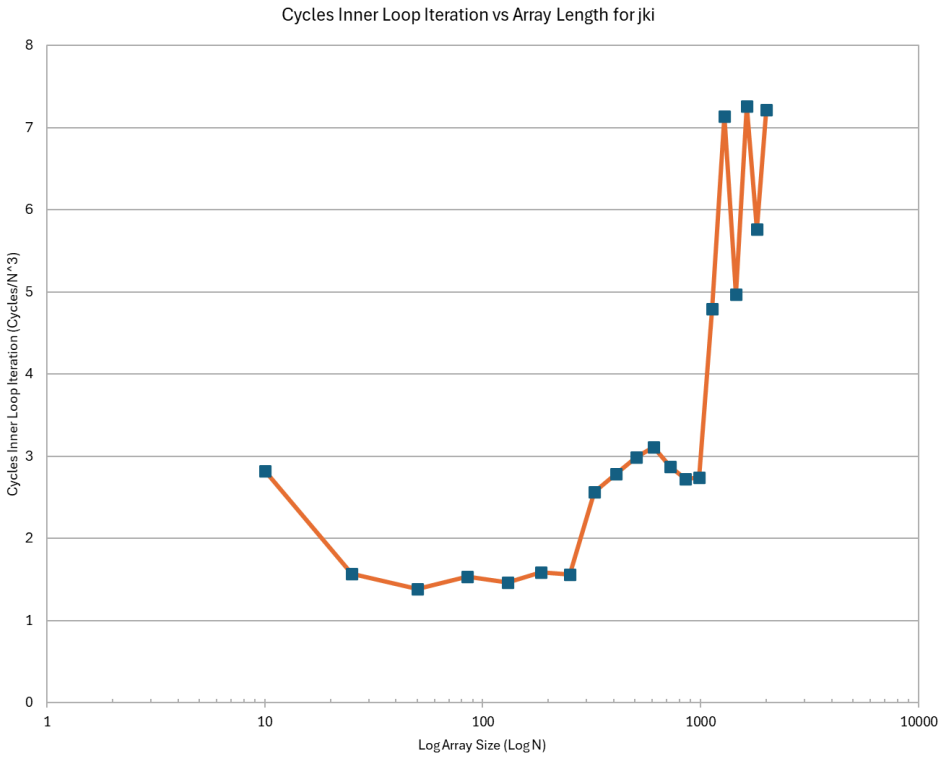


i. For the kij method, the plateau is singular but longer in range. Aside from the starting point (10x10), the plateau would start from 25x25 to 985x985 before the cycles per inner loop iteration would increase from ranges of 1.3-1.4 to 1.7-1.8.

ii. The inner loop iteration of the plateau centers around 1.40, with numbers ranging from 1.37 to 1.48. Averaging the cycles in the plateau, the number results in 1.387 cycles per inner loop iteration.

iii. As explained in part i, the singular transition occurs when the array length exceeds 985, or as the data points out in the plot.

c.



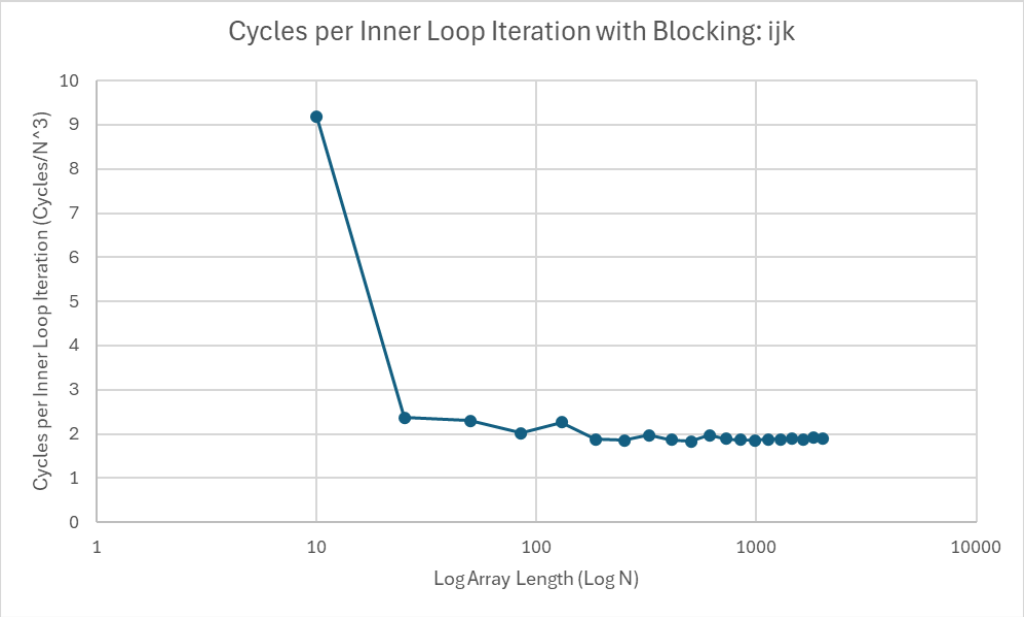
i. For the jki, method there are two plateaus but the latter one is relatively short. Rather it looks like two transition points but only one plateau. The array length which the plateau ends is around 325 which the cycles would increase from a range of 1.5 to over 2.5. The second plateau is only two data points from array lengths 850 to 985 where the cycles center around 2.75 per inner loop iteration/

ii. The inner loop iteration of the first plateau centers around 1.50, with numbers ranging from 1.46 to 1.56. Averaging the cycles in the plateau, the number results in 1.517 cycles per inner loop iteration. The second and shorter plateau centers around 2.7 cycles, averaging to 2.73 cycles per inner loop iteration.

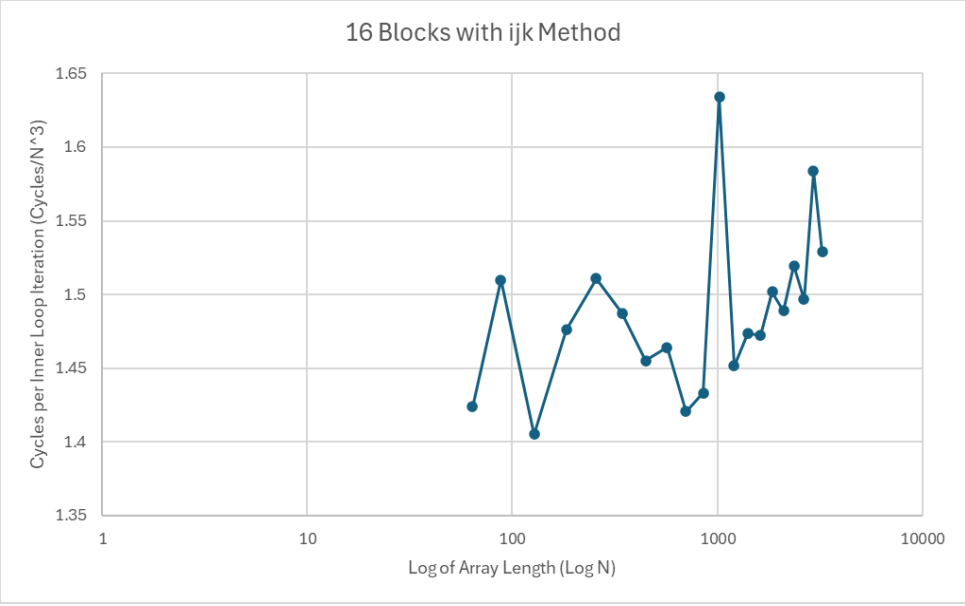
iii. As explained in part i, the transition point occurs when the array length exceeds 250, and the second transition occurs when the array length exceeds 985.

Part 4.

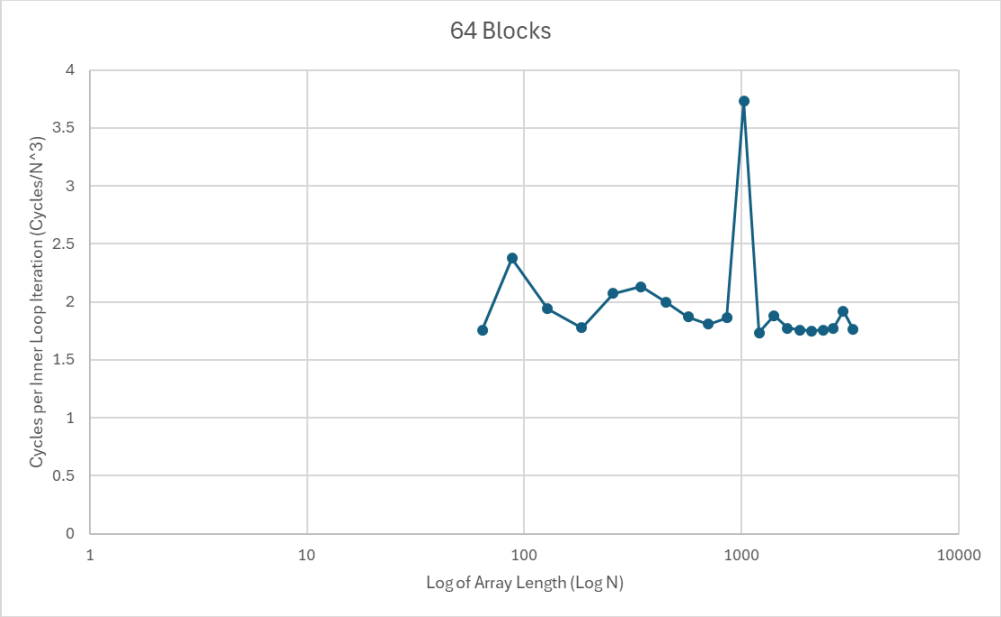
a. The blocking function significantly reduces the cycles and computation time for all three methods as the array size/length/elements progressively increase to higher numbers. Blocking divides the matrices into smaller blocks that fit into the CPU's cache (from L1 to L2 down the hierarchy). This method reduces cache misses whilst reducing the CPU run-time, improving performance compared to the non-blocking method. In a graphical/plot view, instead of having transitional points, the graph appears just to have a flat plateau. Below is the plot with the same quadratic equation while using blocking with 32-sized blocks. The average cycles per inner loop iteration is 2.322 for the entire data set, contrasted with 3.10 of the cycles without blocking.



b. Below are the plots I have achieved using the quadratic equation: $8x^2 + 16x + 64$ (A = 8, B = 16, C = 64) and blocking with block sizes of 16 and 64.

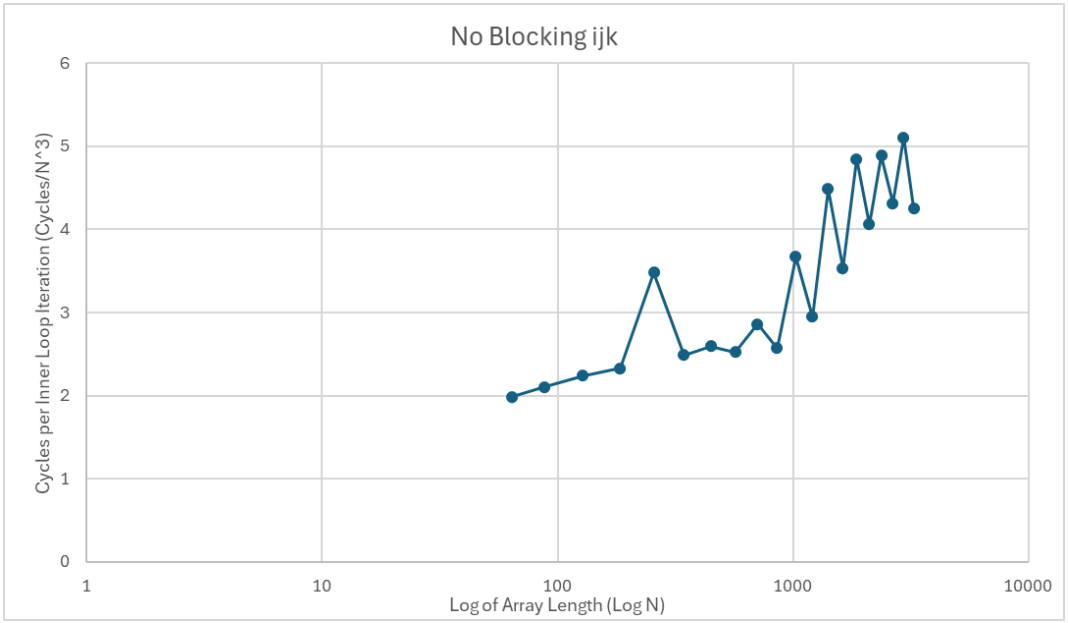


Both blocking method displays a significant increase in cycles per inner loop iteration $\frac{Cycles}{N^3}$ at the 1024 x 1024 array mark (1.63 and 3.73 respectively).



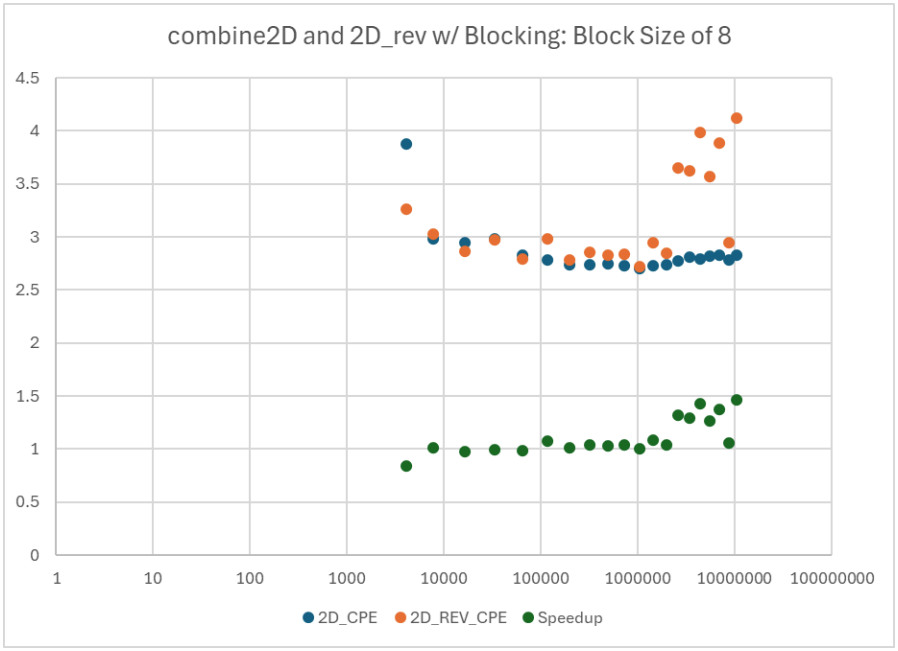
c. The difference in analysis could be due to a few factors. Even though the system utilizes a much more powerful processor compared to the one from the textbook that produced the theoretical analysis, the cache size of the modern machine is still relatively smaller, compared to the Pentium III server processors. However , when comparing the values of the cycles, they are still significantly lower in the scales and magnitudes of 10 times. Also, the system is not perfect as it could also be running background tasks that may enforce resource utilization of those said tasks creating bottlenecks when emulating this simulation.

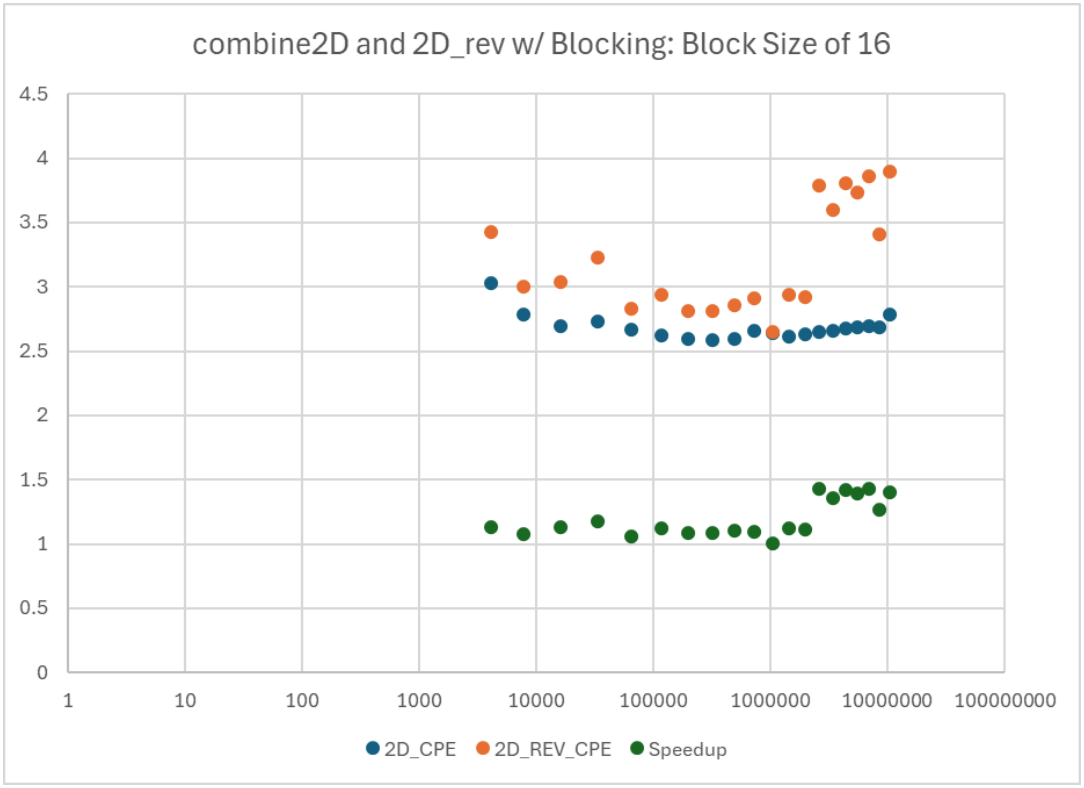
d. Below is the plot of the running matrix computation using the same array size without blocking ($8x^2 + 16x + 64$). The greatest distinction between blocking and non-blocking is the slope of data sets when plotted against the log of array length. Without blocking the entire data set when the array length increases exponentially, the cycles per inner loop iteration also do the same. There appear to be no plateaus and at the steepest of the slope, the cycles start to generate a jagged-like pattern (/_/\). Blocking practically flattens the graph out when caching with 64-sized blocks. Unlike non-blocking, when the array length increases, the number $\frac{Cycles}{N^3}$ seems to plateau and asymptotes towards 1.7.



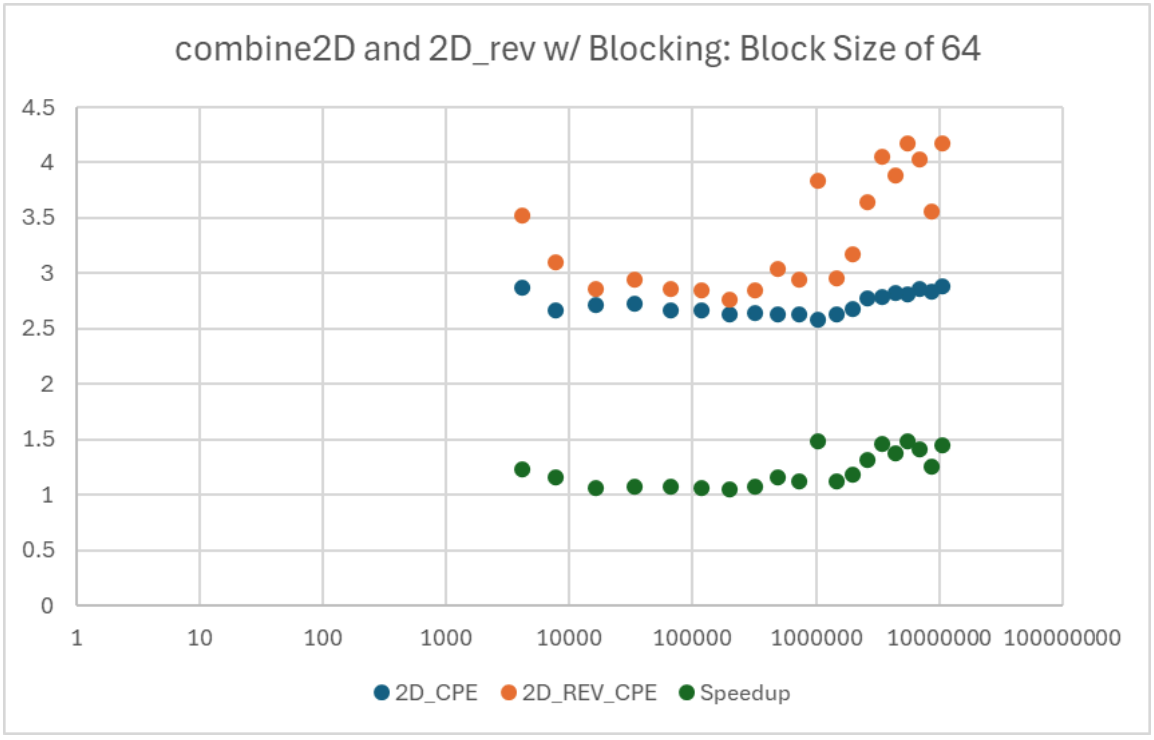
Part 5.

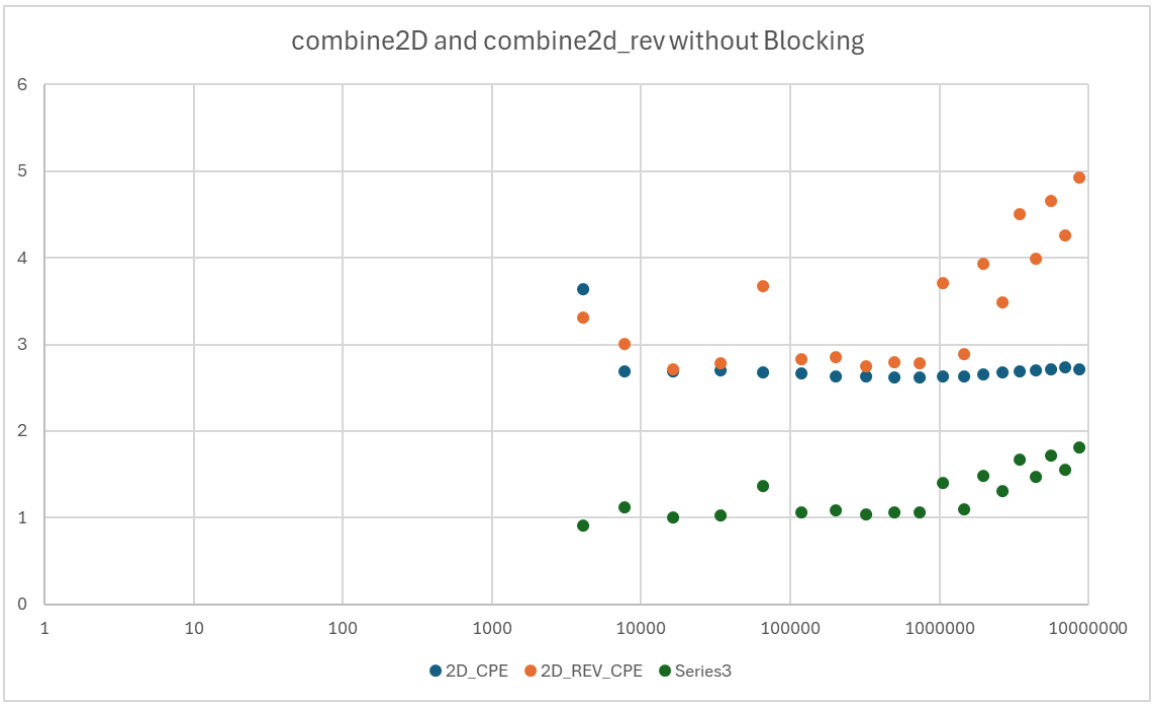
B. This would generate matrix sizes ranging from 64 x 64 to 3256 x 3256. When observing the plots side by side, the patterns of each plot look very similar, combine2D would remain as a plateau after the initial 64 by 64 matrix generation, whilst juxtaposed with combine2D_rev which would start as a plateau and then transition to very large cycles when the size of the matrix reach above 1000 x 1000. Below are the plots I have generated comparing the cycles per element of both methods over 4 different blocking methods and a control, I used a quadratic equation for matrix generation of: $8x^2 + 16x + 64$ (A = 8, B = 16, C = 64). The X-Axis is the array elements (Array Length x Arary Length) on a log scale, and the Y-Axis is the cycles (output from the program)




combine2D and 2D_rev w/ Blocking: Block Size of 16
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
1
10
100
1000
10000
100000
1000000
10000000
100000000
2D_CPE
2D_REV_CPE
Speedup


combine2D and 2D_rev w/ Blocking: Block Size of 32
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
1
10
100
1000
10000
100000
1000000
10000000
100000000
2D_CPE
2D_REV_CPE
Speedup


combine2D and 2D_rev w/ Blocking: Block Size of 64
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
1
10
100
1000
10000
100000
1000000
10000000
100000000
2D_CPE
2D_REV_CPE
Speedup


combine2D and combine2d_rev without Blocking
6
5
4
3
2
1
0
1
10
100
1000
10000
100000
1000000
10000000
2D_CPE
2D_REV_CPE
Series3

Part 6.

a. The entirety of the assignment took over 20 hours to complete.

b. Parts 3 to 5 took over 10 hours to complete, this included changing the code, attempting and testing data collection, and also understanding what the assignment required us to do.

c. A lot of skills regarding programming in the C language which I have never done before this class.

d. I want to assume the issues pertain more to hardware because most of the time was waiting for the system to execute and generate outputs based on the program and not knowing if iterations of the code even worked.