Project 2 Report

Ahmad Jareer ajreer20 0074982
Omar Al Asaad oasaad20 0075155

Graphs
- bash$ ./mcubes -n 256 -f 1 -b 1 -t:
- bash$ ./mcubes -n 64 -f 64 -b 1 -t
- bash$ ./mcubes -n 8 -f 32768 -b 1 -t
- bash$ ./mcubes -n 8 -f 32768 -b -t
- bash$ ./mcubes -n 256 -f 1 -b -t

Part 1
- Questions:

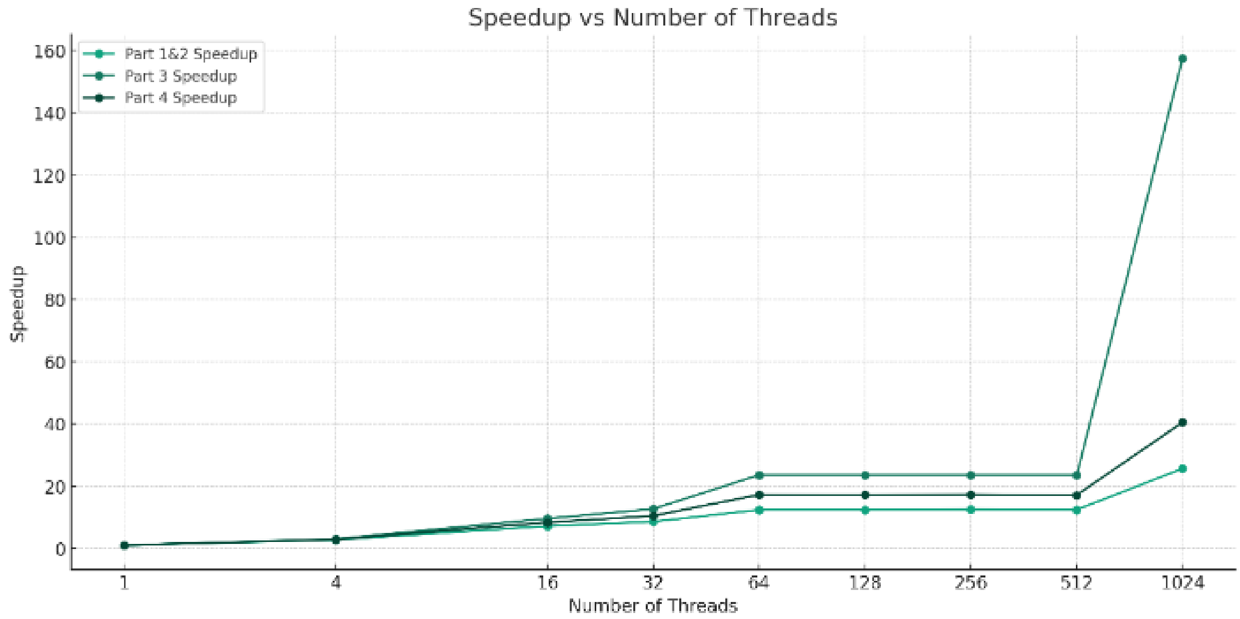
Part 2
- Questions:

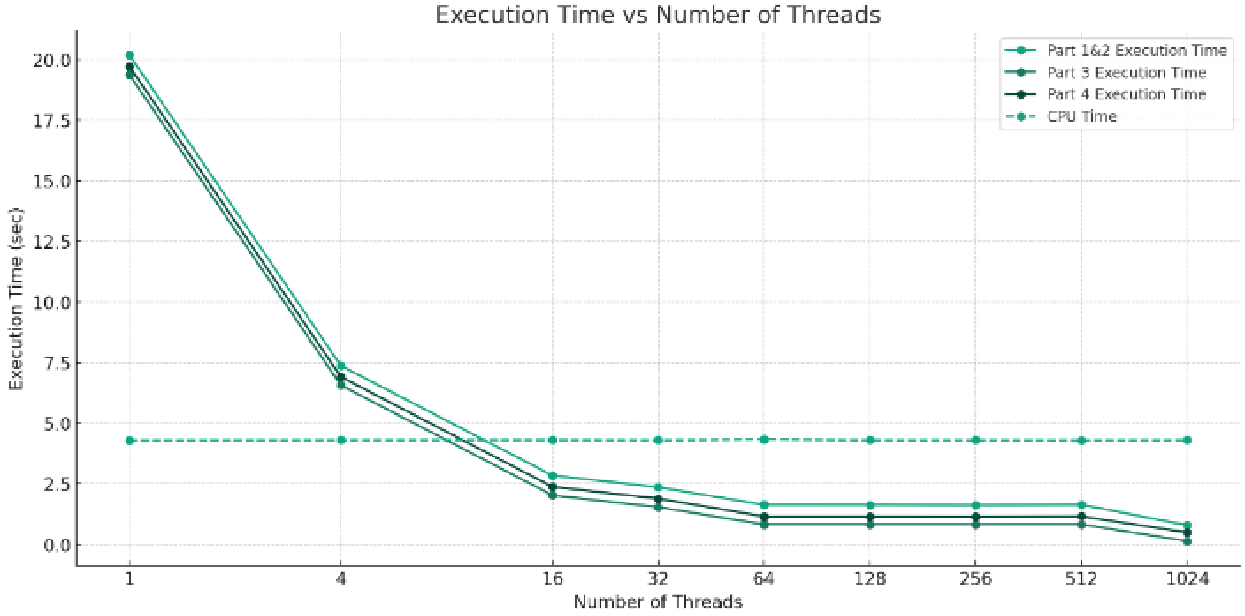
Part 3
- Questions:

Part 4
- Questions:

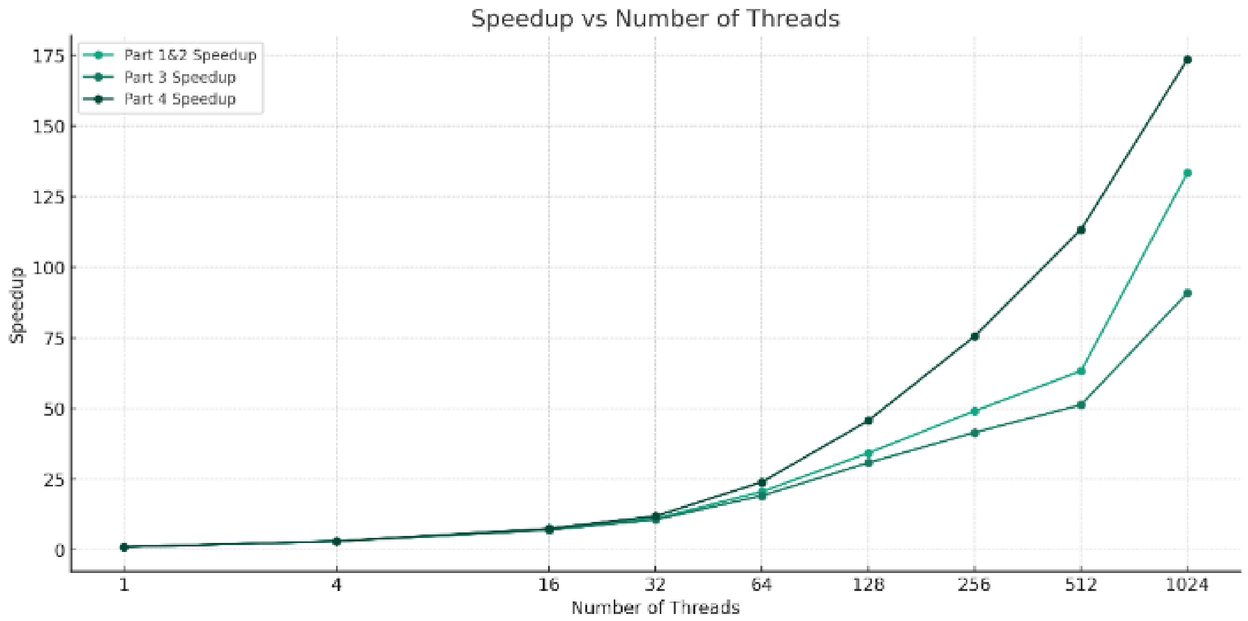
Graphs

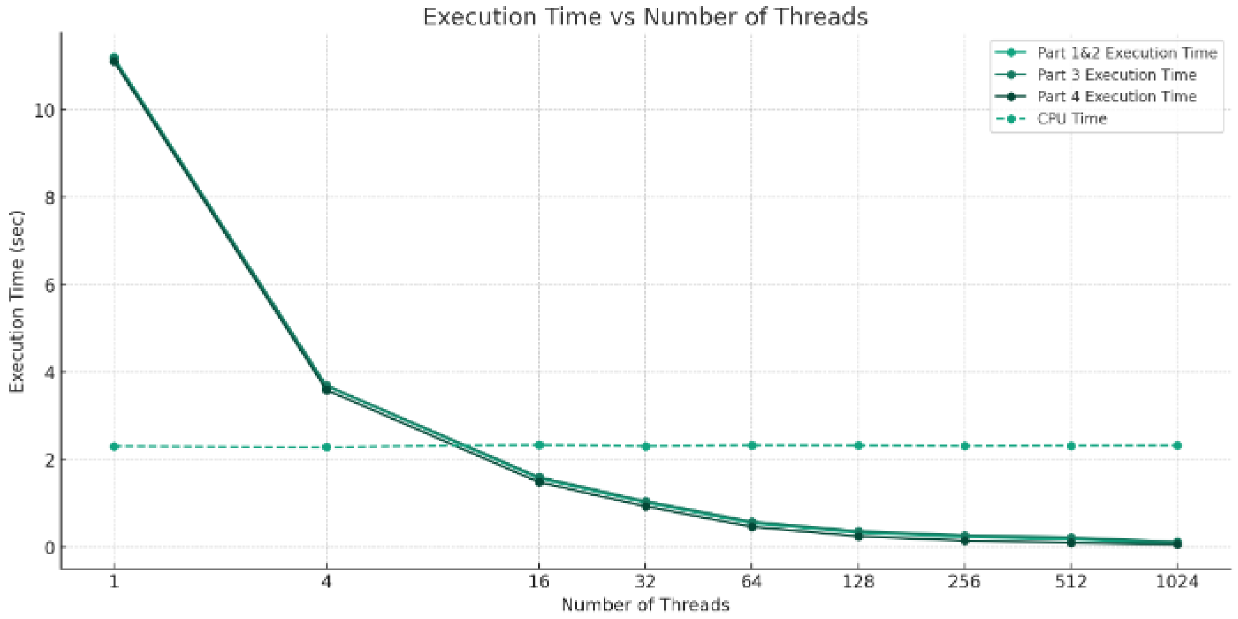
bash$ ./mcubes -n 256 -f 1 -b 1 -t:



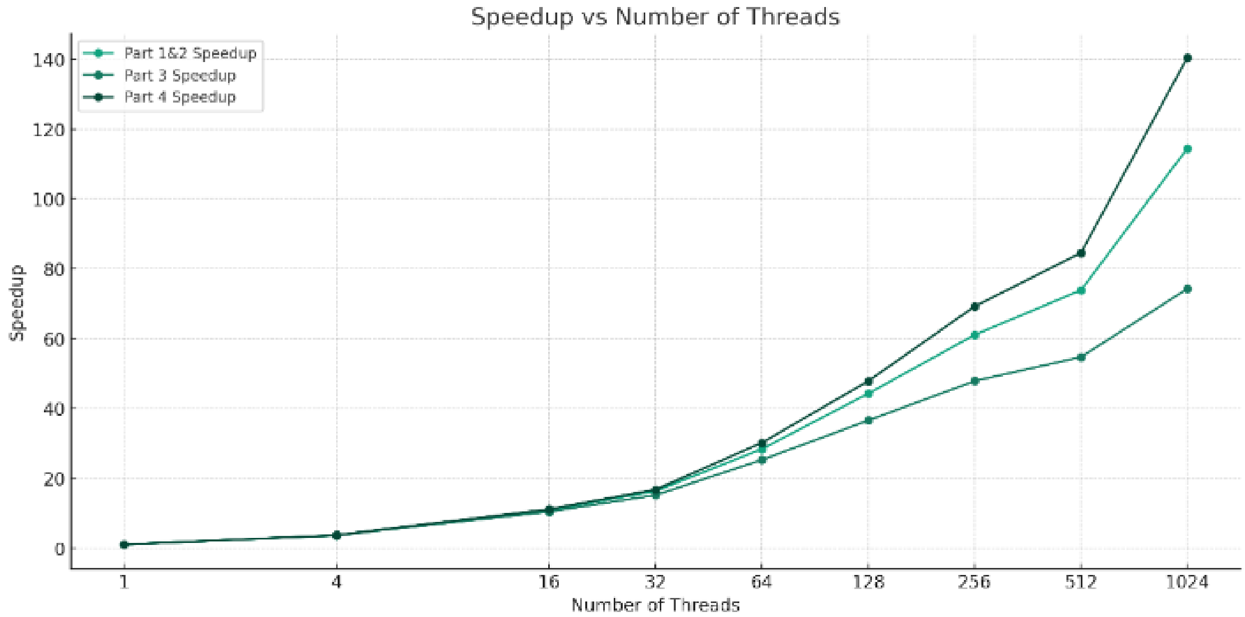


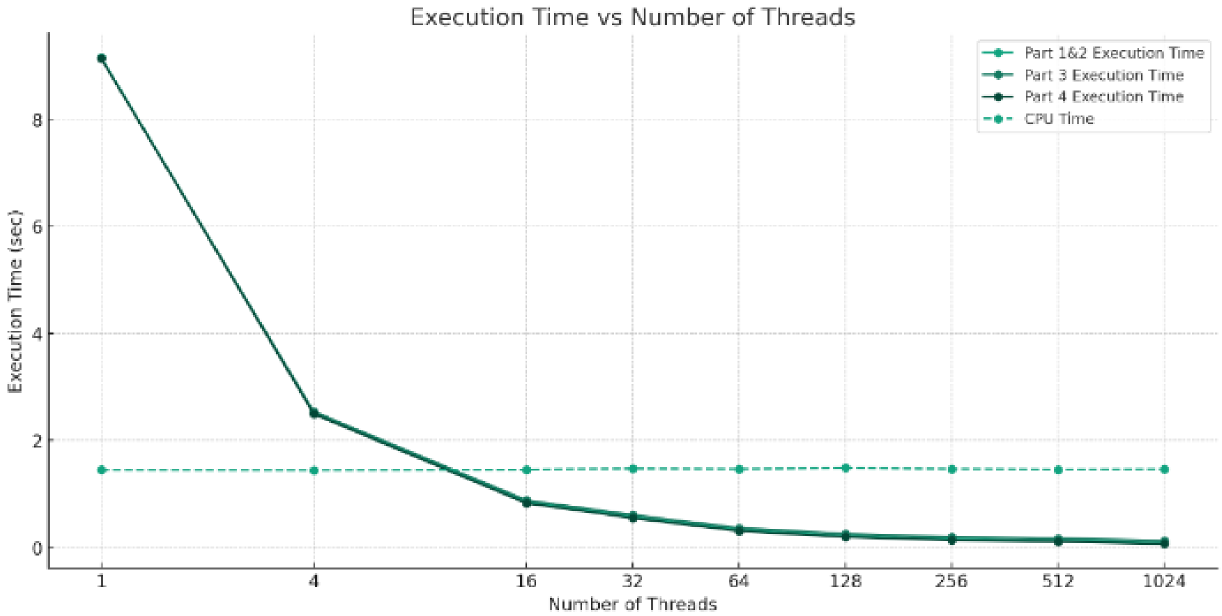
```
bash$ ./mcubes -n 64 -f 64 -b 1 -t
```



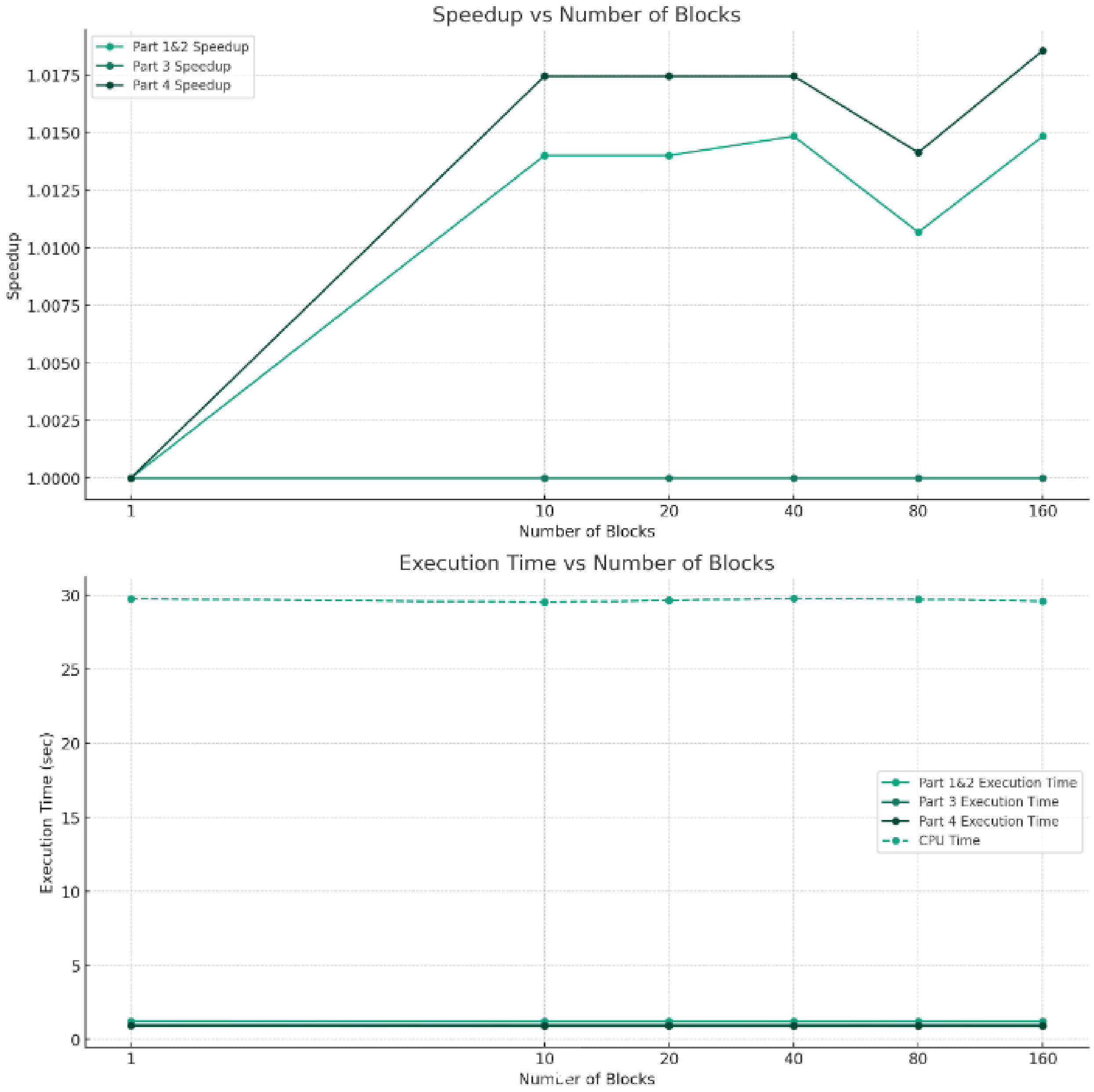


```
bash$ ./mcubes -n 8 -f 32768 -b 1 -t
```

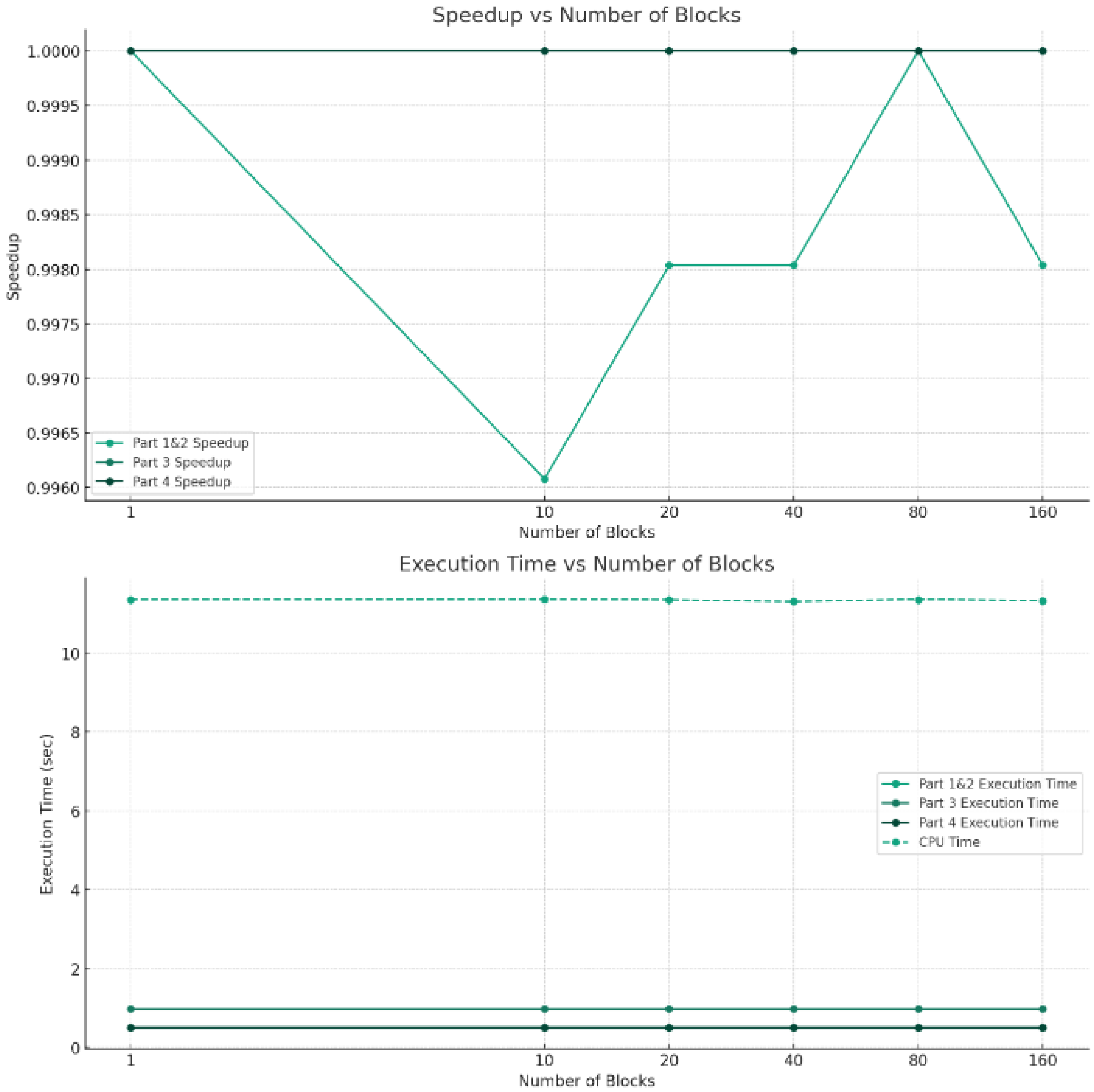

Speedup vs Number of Threads


Execution Time vs Number of Threads

```
bash$ ./mcubes -n 8 -f 32768 -b -t
```



```
bash$ ./mcubes -n 256 -f 1 -b -t
```



Part 1

Questions:

For this part, you need to provide the performance comparison of running a single GPU thread and a single CPU thread. Explain why you are seeing these results in your report.

We noticed that a single CPU thread is much faster than a single GPU thread. CPU: Average Time: 4.2892786 sec and the GPU Average Time: 20.188837 sec after averaging the runs. While CPUs are designed for general-purpose tasks, they excel in sequential processing with fewer but more complex cores, and GPUs are optimized for parallel processing. For this specific reason, we noticed these results which in turn assure us that our results are correct.

Part 2

Questions:

q1) Have you observed any issues with high thread counts while working on this part? If so, why do you think it happened?

There are insignificant improvements in average execution time at higher thread counts. As the thread count increases, the overhead also increases. Additionally, GPU cores have a finite number of resources; when too many threads are competing for these resources, it can lead to decreased performance.

q2) What is the thread and block count that provides the best performance?

For the best block count it can be anything since we split the tasks evenly between the threads and blocks; that's also evident from the results we achieved since they're all averaging around the same execution time and speedup. For thread count the best one is 1024 it achieves the highest speedup.

q3) How does the memcopy overhead scale with increasing problem size? How about kernel overheads? Compare the two, and try to find the setting where they are equal.

Memcpy shows slight variations across different thread counts but generally remains within a narrow range. This indicates that memcopy overhead is relatively consistent regardless of the number of threads. However, as the problem size increases (indicated by the -n and -f parameters), the memcopy times do not show a significant increase. This suggests that memcopy overhead is not heavily influenced by the size of the problem, possibly because the amount of data transferred between CPU and GPU remains relatively constant for these particular computations.

Kernel execution times decrease notably as the number of threads increases. This is a clear indication of the GPU's efficiency in parallel processing: more threads lead to better utilization of the GPU's parallel computation capabilities, resulting in faster kernel execution. The decrease in kernel execution times is more visible with larger problem sizes, suggesting that the GPU's parallelism is more effectively used in larger computations.

Comparing the two overheads, memcopy times are relatively stable, whereas kernel times decrease with increasing parallelism. They are equal when they are using 1024 threads. That is when they’re the most close

Part 3

Questions:

q1) Have the kernel execution overheads changed? How about memory copies?

In Part 3, there isn't much of a change in kernel execution overheads, but we see a significant change in memory copy time, and that is because we only copy back once instead of doing that every frame.

q2) How does the kernel overhead compare to Part 2 with a high number of frames and small problem size? What if the problem size is large and the number of frames is small? Explain

In Part 2, with a high number of frames and a smaller problem size, the kernel overhead is generally lower. This is because smaller problems require less computation per kernel launch, and high frame rates benefit from the GPU's ability to efficiently process multiple, smaller tasks in parallel. Therefore, the GPU is correctly utilized in handling a greater number of simpler tasks rapidly. On the other hand, in Part 3, with a larger problem size but fewer frames, the kernel overhead tends to increase. This increase is due to each kernel having to perform more computations per run. Larger problems demand more of the GPU's resources per task, potentially leading to longer processing times per kernel. The parallel processing advantage of the GPU is lessened when individual tasks become more complex and computationally intensive.

Part 4

Questions:

q1) What is the obtained speedup compared to Part 2?

The speed up in part 4 is much higher than the speed up in part 2; since in part 2 we wait for each kernel call to start copying but in part 4 we do the copying while the kernel call is ongoing.

q2) What is the observed memory usage compared to previous approaches?

In part 1&2 we allocated one frame and we kept writing in it and sending it back while in part 3 we used all frames and sent them back after computing the values all at once. In part 4 we allocated 2 frames we used one for copying and the other for computation.