COMP 429 - PARALLEL PROGRAMMING

FALL 23 - DIDEM UNAT

PROJECT 2 REPORT

BATU ORHUN GUNDUZ 79886 & EMRE AKCA 80567

Part 1 Discussion

Single threaded serial CPU implementation is way faster than its GPU counterpart. For n=128 and f=1, CPU code finished in 2 seconds while GPU code finished in 32 seconds. Similarly, for n=32 and f=64, CPU core finished in 3.4 seconds while GPU core finished in 48.9 seconds. This is because a single CPU core has a higher performance that GPU. CPU has less but powerful ("probably fat") cores, that's why it executes faster than GPU in single thread comparison.

Part 2 Discussion

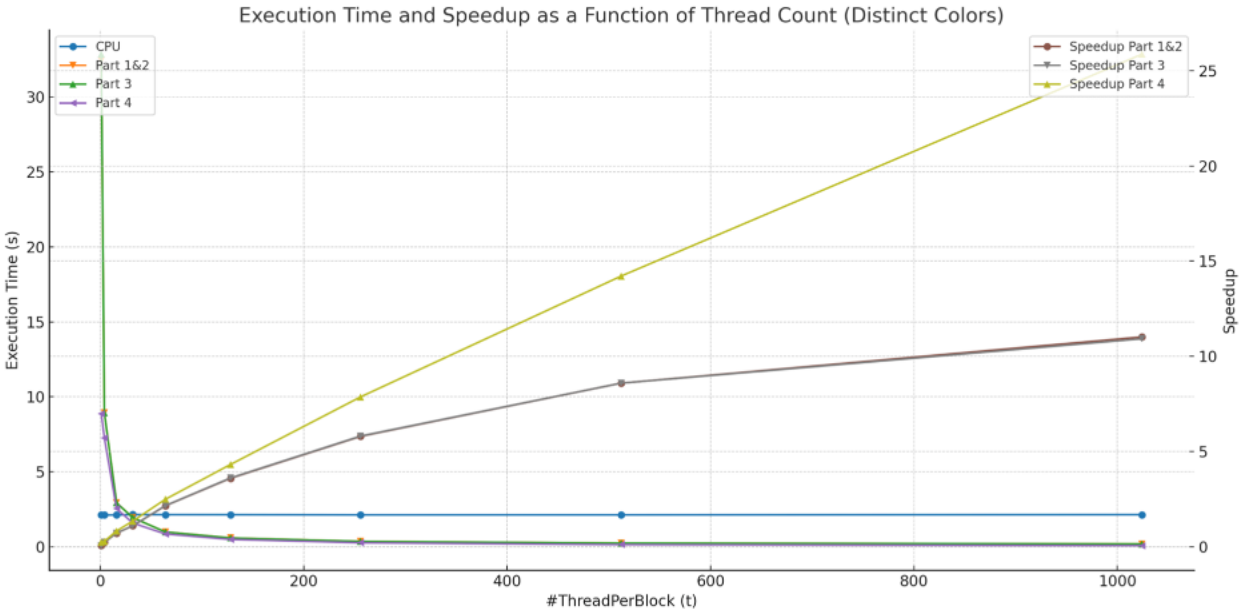
In this part, after 128 threads there started to occur underutilization problems, which especially can be seen in Graph-2. This is because memory is starting to be the main performance bottleneck and probably the warping system of NVIDIA GPU's which fails to fulfill the ideal performance in strong scaling. High thread counts increase memory access and bandwidth usage. Therefore, for n=128 & f=1 case, kernel time elapsed was the predominant bottleneck up to 256 threads but from then on, memcpy and kernel almost be equal in 512 threads and memory is now the effective bottleneck for higher number of threads. However, even with small margins, 1024 thread count with default 1 block was the scenario that achieves the best performance. In below screenshots, one can observe the kernel vs memcpy times.

```
resolution (n) = 128, number of frames (f) = 1
blocks (b) = 1, threads (t) = 128
CPU
Time taken: 2.144627 | Performance: 0.977863 mln_cubes/sec
no error
no error

Part1&2
Time taken:
Kernels: 0.475689 sec | Performance: 4.408663 mln_cubes/sec
Memcpy:  0.120431 sec
Extra:   0.000000 sec
Total:   0.596120 sec
```

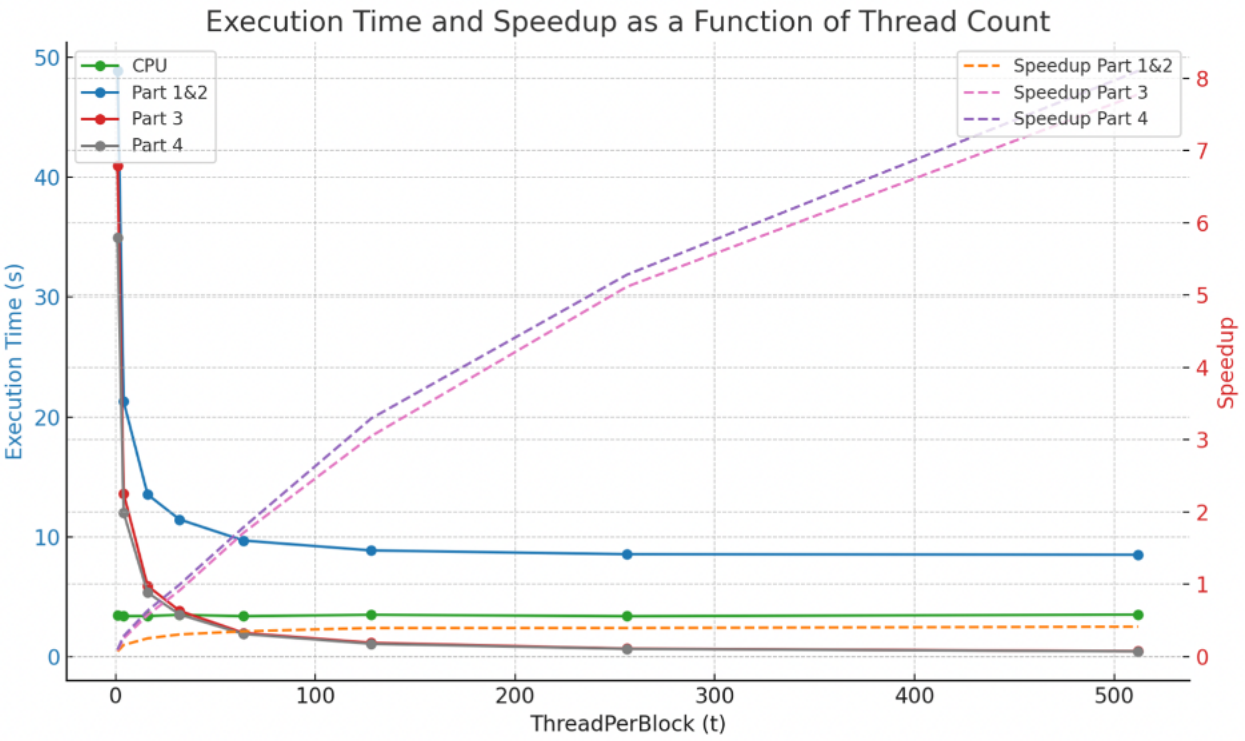
```
resolution (n) = 128, number of frames (f) = 1
blocks (b) = 1, threads (t) = 256
CPU
Time taken: 2.131573 | Performance: 0.983852 mln_cubes/sec
no error
no error

Part1&2
Time taken:
Kernels: 0.246488 sec | Performance: 8.508130 mln_cubes/sec
Memcpy:  0.121028 sec
Extra:   0.000000 sec
Total:   0.367516 sec
```

```
resolution (n) = 128, number of frames (f) = 1
blocks (b) = 1, threads (t) = 512
CPU
Time taken: 2.132487 | Performance: 0.983430 mln_cubes/sec
no error
no error

Part1&2
Time taken:
Kernels: 0.127628 sec | Performance: 16.431713 mln_cubes/sec
Memcpy:  0.120526 sec
Extra:   0.000000 sec
Total:   0.248154 sec
```

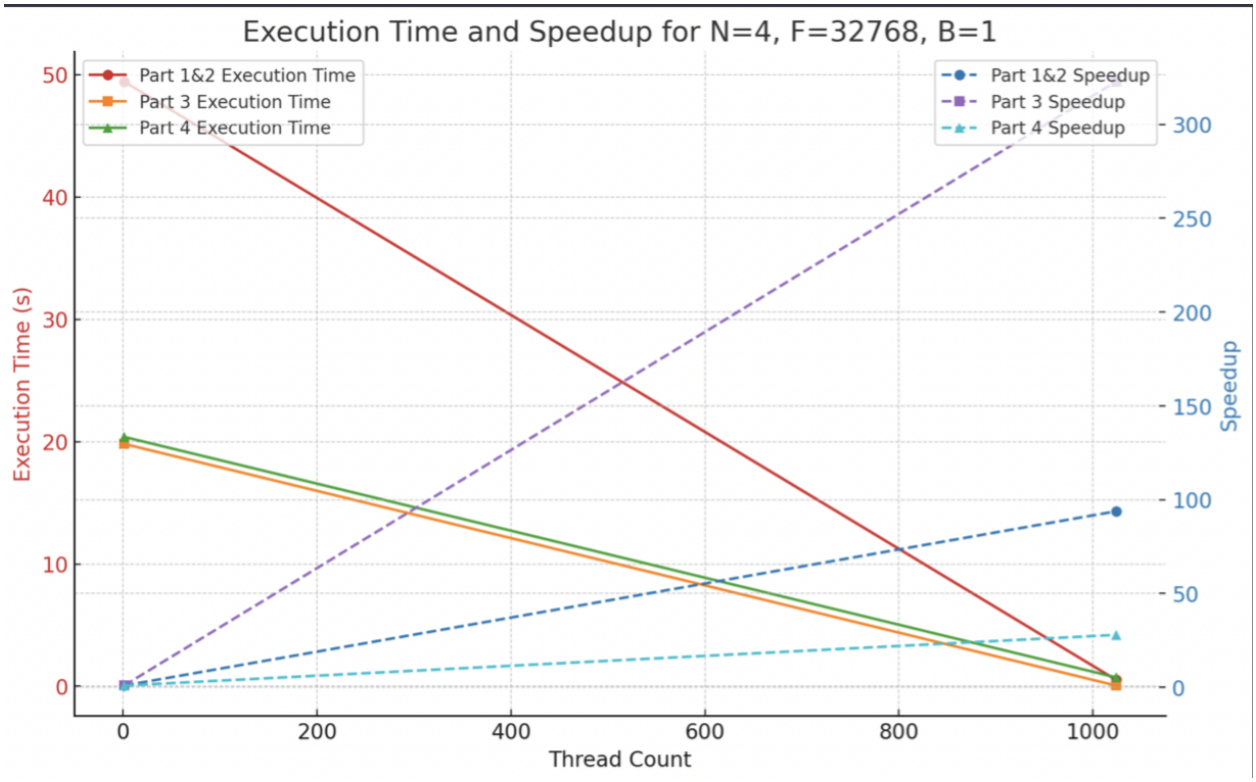
```
resolution (n) = 128, number of frames (f) = 1
blocks (b) = 1, threads (t) = 1024
CPU
Time taken: 2.147212 | Performance: 0.976686 mln_cubes/sec
no error
no error

Part1&2
Time taken:
Kernels: 0.074049 sec | Performance: 28.321190 mln_cubes/sec
Memcpy:  0.120805 sec
Extra:   0.000000 sec
Total:   0.194854 sec
```



Graph 1 (please also check Table-1)



Graph 2 (also please check Graph 7)



Graph 3

Part 3 Discussion

In this part, we sacrificed memory usage efficiency to data locality principle by dramatically reducing the memory copy operations' frequency, which enables us to achieve higher performances. Therefore, we observed an improvement. Kernel launching overhead is thus decreased since now one kernel does the job instead of multiple kernels, but execution of that kernel itself may be suffered. Please check the graphs for the speedup.

For the scenario that has too many frames, discrete kernel approach may be inefficient compared to persistent kernel approach. But with problem size increasing and frame size decreasing, utilizing separate kernels might not be that of a problem. But here, we should exclusively account for the case where memory bandwidth comes into the such that its burden makes the kernel overhead meaningless, which makes persistent kernel approach the ultimate effective solution since it requires less memory transfers.

Part 4 Discussion

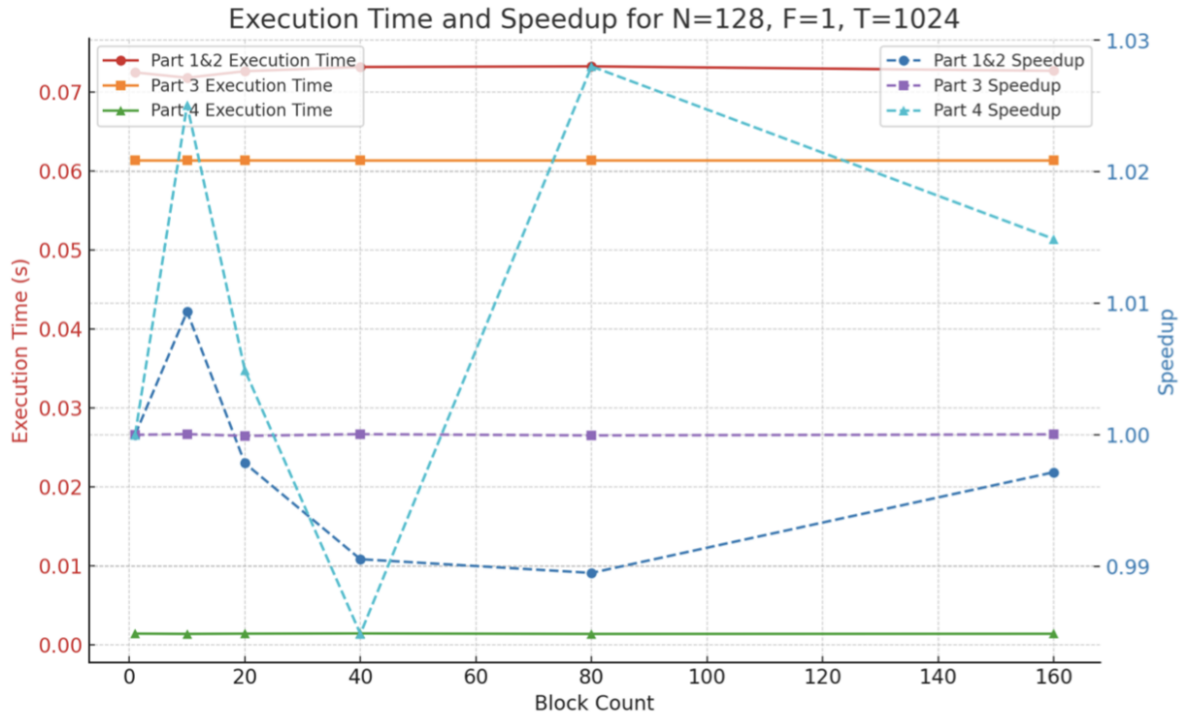
In this part, we observed a consistent speed-up which increases as thread count increases which can be seen in graphs above. Part 4 total time elapsed is strictly better than every other parts as we utilize concurrent CUDA events. In this way, we both effectively use the memory bandwidth with respect to time passed and compute along the way. For higher number of threads, part 4 converges to doubling part 2 performance, especially after 256 threads, performance difference further becomes clear in proportions (0.367516 sec to 0.270689 sec).

After limiting ourselves to 2 frames given in each turn, our memory usage efficiency also improved significantly. Using 30% less GPU memory compared to part 3 (and previous parts), we could achieve a better performance.

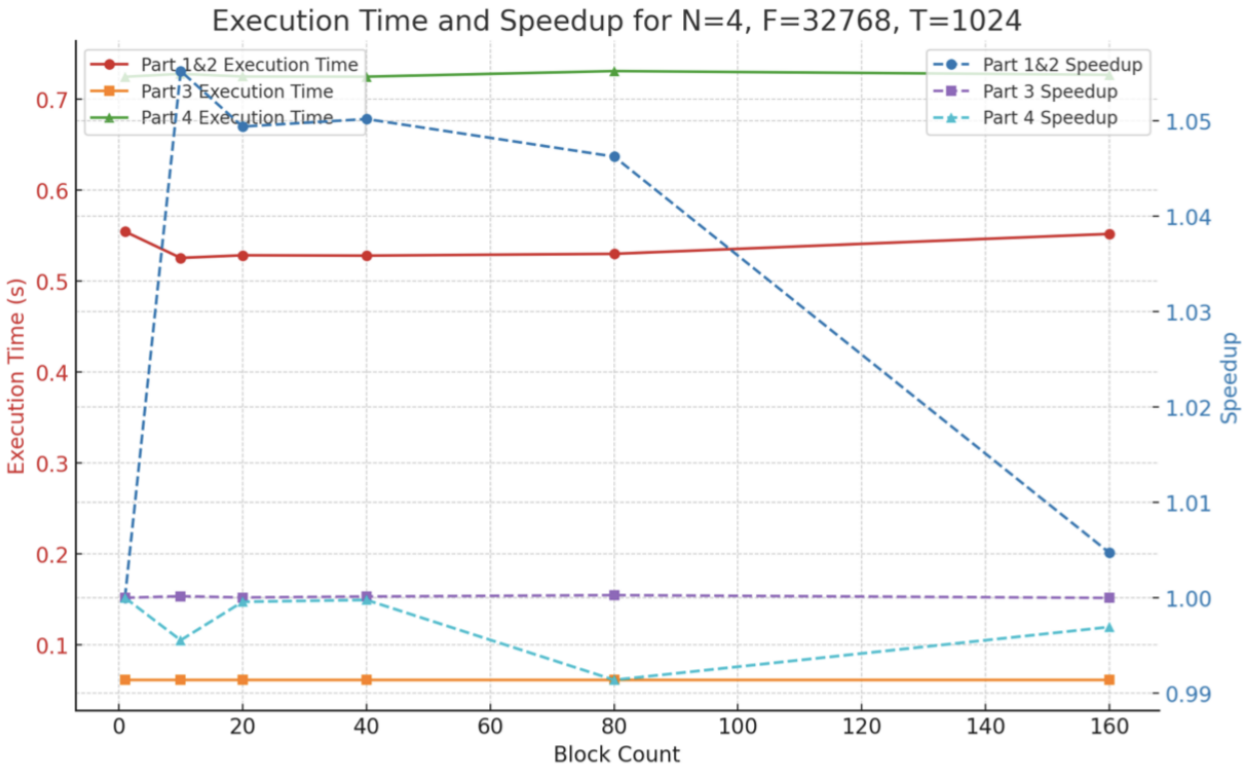
```
Part1&2
Time taken:
Kernels: 1.990623 sec | Performance: 0.627944 mln_cubes/sec
Memcpy:  0.364727 sec
Extra:   1.110604 sec
Total:   3.465955 sec
GPU memory usage: used = 874.000000, free = 31627.125000 MB, total = 32501.125000 MB
no error
no error

Part3
Time taken:
Kernels: 0.013354 sec | Performance: 93.603588 mln_cubes/sec
Memcpy:  0.036463 sec
Extra:   1.438830 sec
Total:   1.488647 sec
GPU memory usage: used = 874.000000, free = 31627.125000 MB, total = 32501.125000 MB

Part4
Time taken:
Total: 1.154032 sec
GPU memory usage: used = 510.000000, free = 31991.125000 MB, total = 32501.125000 MB
[bgunduz21@it01 project-2-comp429]$
```

Additional Remarks

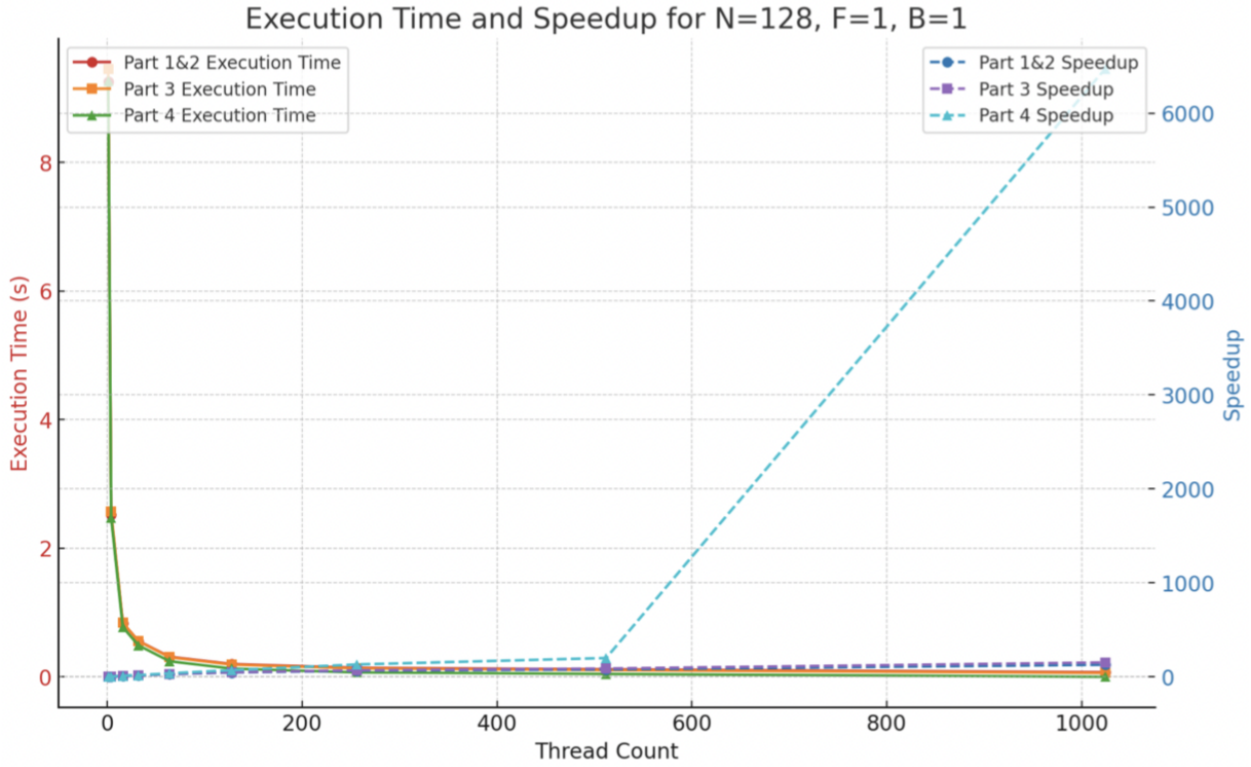
Note that execution time is stable and speedup values are not very meaningful in this context, as our thread count is fixed. But from this a posteriori information, one can claim that GPU resources are well utilized across arbitrary block configurations.



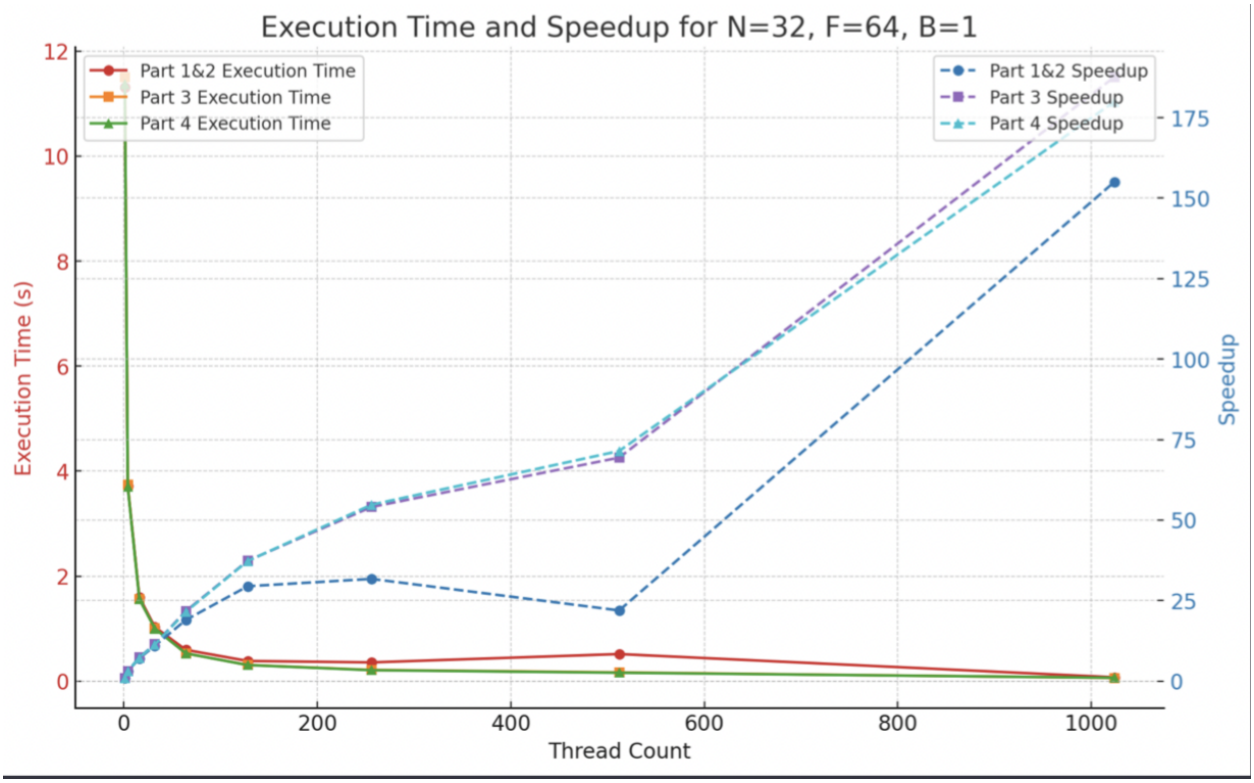
Graph 4



Graph 5



Graph 6 - Outlier to Graph 1



Graph 7 - Alternative Run for Graph 2

Thread Count	CPU	Part 1&2	Part 3	Part 4	Which takes longer
1	2.13042	32.4127	32.8288	28.8778	kernel
4	2.12254	8.94018	8.92706	7.25599	kernel
16	2.12383	2.94014	2.93106	2.54548	kernel
32	2.15251	1.94411	1.94425	1.58817	kernel
64	2.14634	0.995363	0.99197	0.859211	kernel
128	2.14463	0.59612	0.592848	0.496058	kernel
256	2.13157	0.367516	0.366317	0.270689	kernel
512	2.13249	0.248154	0.248151	0.150039	equal
1024	2.14721	0.194854	0.196585	0.083041	memcpy

Table 1 (Sample Data from the First Run of Graph 1 Configuration)