Hakan Çapuk - 76707
Ege Doğukan Seven - 76215

COMP 429 - Parallel Programming - Project 2 Report

Part 1:

In this part, when we run the experiments with thread and block counts of 1, we run the kernel function with 1 thread in the gpu. The results in 3 different problem sizes can be seen below. We are running the experiments with block and thread count of 1.

Experiment	CPU Run time (s)	GPU run time (s)
n: 256, f:1	12.08	85.11
n: 64, f:64	16.07	101.88
n: 8 f:32768	30.34	176.85

GPU's are created to be better at parallel processing, as their data movement and kernel overheads are high. In order to use the advantage of GPU's, we need to execute our operations dividing them to a high number of threads. CPU on the other hand, is better at handling serial executions than GPU, and it does not need data movements between device and host memory while doing this. Thus, CPU is usually much faster than GPU when doing operations with single threads.

Part 2: MarchingCubesCUDA

In this part, we divided the cube calculations to individual threads. In order to do this, we first calculate cubesPerThread by dividing total cube count to total thread count (determined by the program's arguments), and give this number to the kernel function. Each thread calculates this number of cubes in order to maintain similar work load and maximize parallelism. If total cube count is lower than the total number of threads, meaning that each thread would get 0 cubes as the division is integer, they are instead each given 1 cube to calculate. And the excessive threads do not do calculations as they are out of bounds from the cubes. We allocated global memory that would be enough for 1 frame for each of meshVertices and meshNormals, and also allocated memory for domainP and cube_sizeP, though it may not be necessary. We decided to just allocate them on device memory and give them as parameters to the kernel function that way, as the 2 structs are much smaller compared to normal and vertice arrays. We also reset meshVertices_d and meshNormals_d (memset them to 0) before each kernel launch, and copy results of the kernel to host memory in each iteration.

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

- In some cases, making thread counts 1024 was causing the kernel to not launch, and the program to exit as if it did its job with no errors. This was a bit problematic at first, but we were able to identify it using print statements and cuda-gdb. Later we were informed that gcc and

cuda module versions in the cluster might be the cause of this, and when we changed their versions the problem was solved. One other problem we encountered was that we were creating the intersect array as a dynamic array inside of the kernel function using “new” keyword, thus causing the program to run out of memory with high thread counts. We solved this by creating the array in a static way, and creating it if it is guaranteed to be used, i.e. the thread will do calculations using it.

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

- Usually, higher thread and block counts give better performance, but it is actually dependent on the problem size too. As we are dividing total cube count to total number of threads, after some point, if number of threads get higher than the cube count, the performance gets to a stall, and might actually reduce. The reason for this is that after reaching a certain threshold, each thread calculates 1 cube anyways, and creating more threads would just cause the excessive threads to do nothing, and cause more overhead. The main idea is that we want to raise thread and block counts until the total number of threads reaches the total number of cubes, to maximize parallelism. You may find more detailed results in the experiment section.

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

- In our runs, with first two problem sizes (n 256, f1 and n 64, f 64), memcopy overheads were similar, each around 0.49 seconds, in the third size however (n 8, f 32768), memcopy overhead rose to 1.05 seconds. As frame number increases, it can be seen that memcopy overhead becomes larger as the number of loops and number of memcpy’s increase. Kernel overheads are not only dependent on memory size, but dependent on block/thread counts as well. In our runs, kernel overheads get closer to memcopy overheads when we increase the number of threads per block, and when the thread count is 1024, they get the closest (kernel overhead: 0.58 s, memcopy overhead: 0.49 s for n 256, f1).

Part 3: MarchCubeCudaMultiframe

In this part, we simply moved the host frame loop to the kernel. Each thread would still calculate its number of cubes, but this time after calculating the cubes on one frame, would move to other frames until all its cubes in all frames are calculated. For example, if cubesPerThread is 2, thread <0,0> calculates the first 2 cubes of the first frame, then calculates the first 2 cubes of the second frame and so on.

• Have the kernel execution overheads changed? How about memory copies?

- Kernel overheads get smaller, while memory copy overheads stay similar. Memory copy overheads are expected to stay similar, as in both part 2 and part 3, they copy frame size chunks to the host in a loop anyways. Kernel overheads are smaller as the kernel function is faster than part 2 in especially large problems. The reason for this is that the threads do more

work before getting killed, and as killing them and creating them for each frame would introduce kernel overheads in GPU that are not caused by calculations, part 3 is faster.

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

- With high number of frames and small problem size, kernel overheads get smaller in Part 3. And with lower number of frames and larger problems, as the thread count increases, kernel overheads get smaller in part 3 than in part 2 as well. In problems with high number of frames, in part 2, we are executing smaller kernels many times. This causes some launch and synchronization overheads, as in each iteration of the loop, kernel launches and the host code waits for it to complete its execution before moving to next iterations. This problem is solved in part 3, as the loop is inside of the kernel this time, and there are much less launches of kernels, actually just 1 in part 3. This gets rid of the loop and synchronization overheads that were happening in host code in each iteration. While the number of frames are small however, this difference become less and less, as the loop and synchronization overheads get lower and lower with smaller number of loop iterations in part 2. The performances tend to get closer than they are in experiments with higher number of frames.

Part 4: Double Buffer

- In part 4, we used the same logic of the kernel function of part 2. This time however, we allocated global memory that would be enough for 2 frames instead of one, and launched the kernel with different pointers according to the current frame iteration. We have two streams, stream1 and stream2, where stream1 executes the kernels and stream2 does the memcopy operations. If current frame number is an even number, we would launch the kernel with the usual meshVertices_d and meshNormals_d pointers, and the kernel would write frameSize number of float3 elements to the arrays, thus using it from start and to the half of the total global arrays. If the current frame number is odd however, we would launch the kernel with meshVertices_d + frameSize, and meshNormals_d + frameSize pointers, meaning the kernel would write starting from the half point of the global arrays and until their end. This way, consecutive frames would not write to each others space, and allow the memcopies to copy the elements as wanted. We made the memcopy operations start not from the first frame, but the second. Meaning that the first frame would launch kernel_frame:0, second frame would launch kernel_frame:1 and memcopy_frame:0 and so on. And the last frame would memcopy the last frame's writes as well, as otherwise the last object does not get written. We were able to synchronize execution of memcopy to be after execution of its corresponding kernel, in order to not copy the vertices and normals before their calculation is done, using cudaStreamSynchronize. We are also setting the copied portion of global arrays to 0 after each copy, as otherwise non-consecutive frames get written a top of each other, resulting in interleaved shapes.

• What is the obtained speedup compared to Part 2?

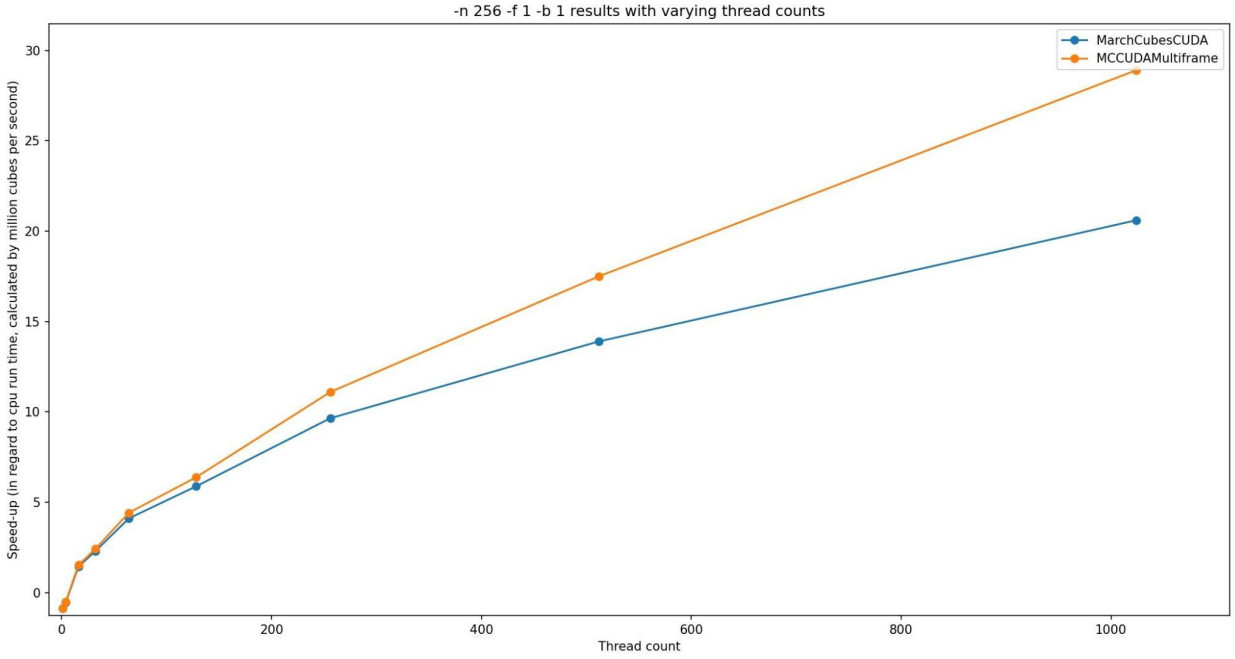
- In most cases, our part 4 function was running faster than part 2, however we did not get enormous amounts of speedups. You can see detailed comparisons in the experiments section.

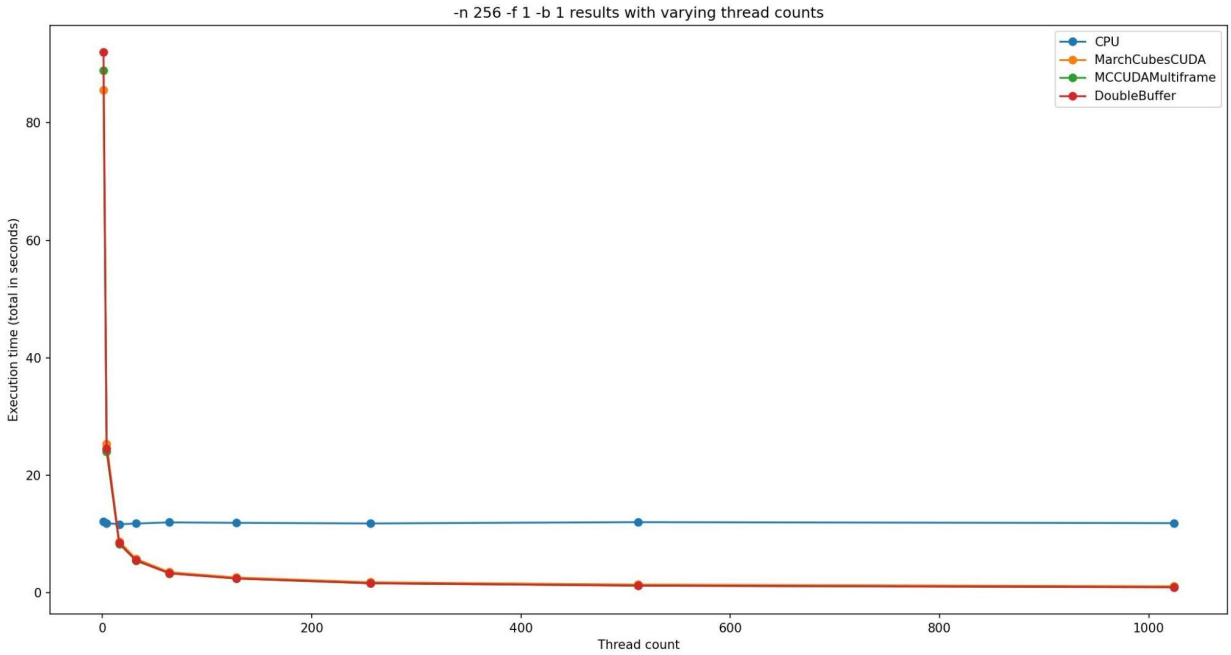
• What is the observed memory usage compared to previous approaches?

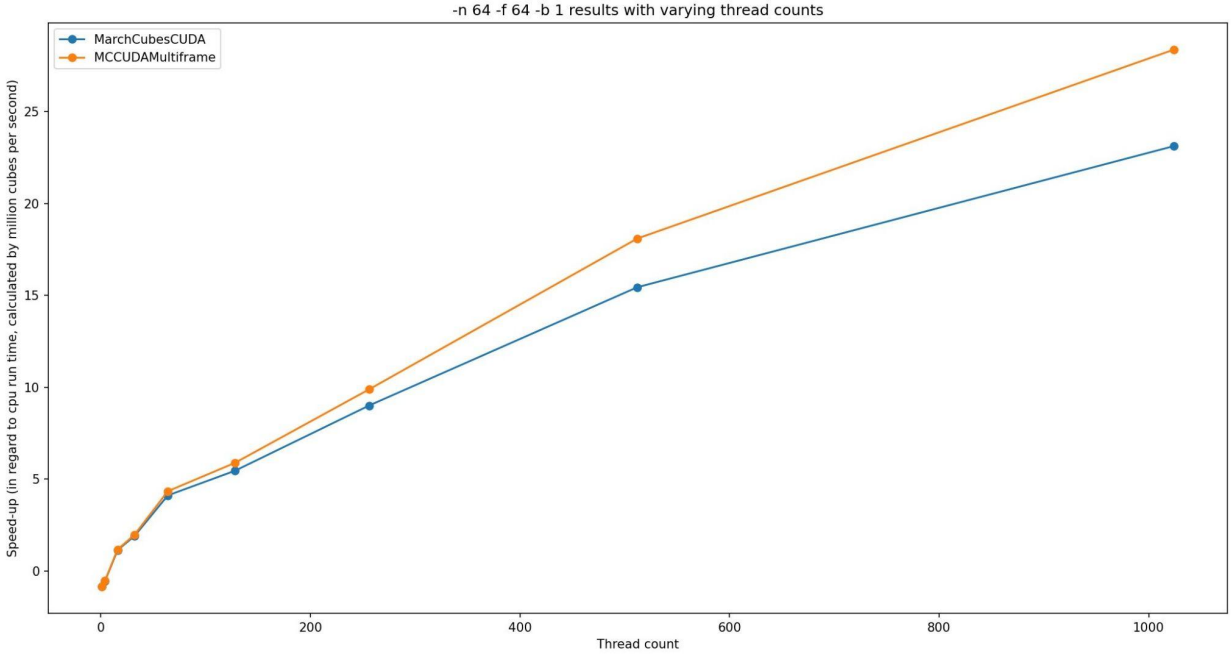
- In part2, we were allocating enough memory for just 1 frame, while in part 3 we were allocating memory that would be enough for all frames. In part 4 however, we allocate memory for 2 frames, thus making its memory usage higher than part 2, but much lower than part 3.

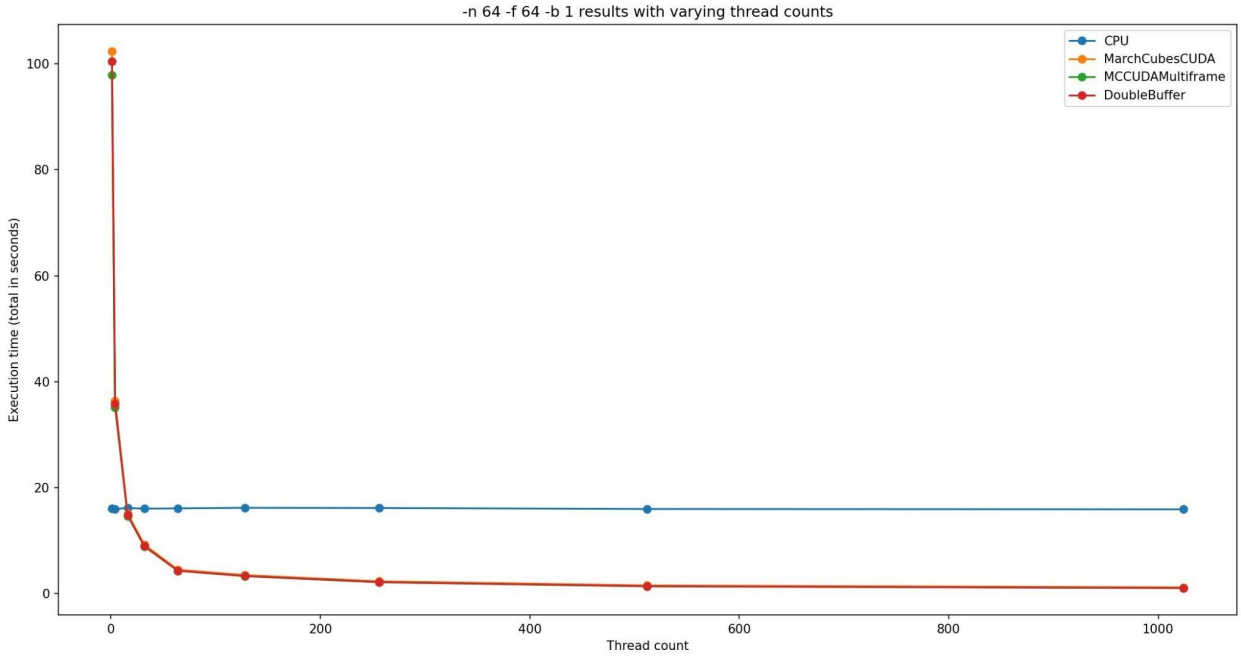
Experiments

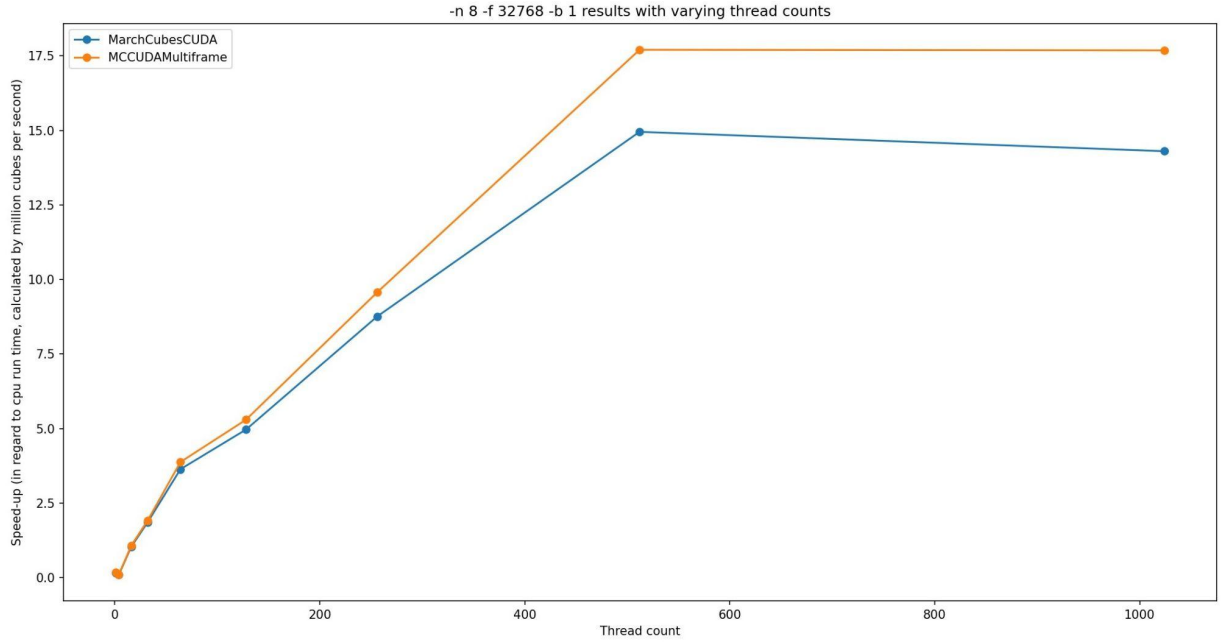
Part1: (Varying thread counts per block)
In this part, we experimented with varying thread per block counts, while keeping block count as 1. We have 3 figures which correspond to each experiment specified in the project document, regarding thread counts, and 3 figures that compare the total run times of the experiments. The speed-up is calculated by dividing the million cubes per second result of the corresponding cuda kernel functions to the million cubes per second results of the serialized cpu run. From the results, it can be seen that the kernels run the fastest when they have 1024 as thread count, except the last experiment. The reason for this is that as explained above, with small problems and high thread counts, some of the threads do not do anything, as they are out of bounds of the problem. Thus in smaller problems, the performance rises before a certain threshold of total thread count is reached, and remains similar or goes lower because of kernel overheads if we go beyond that point. In the below graphs, you can see the speed-ups of kernel functions for part 2 and part 3, in regard to cpu function. Below the speed-up graphs, you can also see execution time graphs, that show the total execution time of cpu function and cuda functions accordingly.

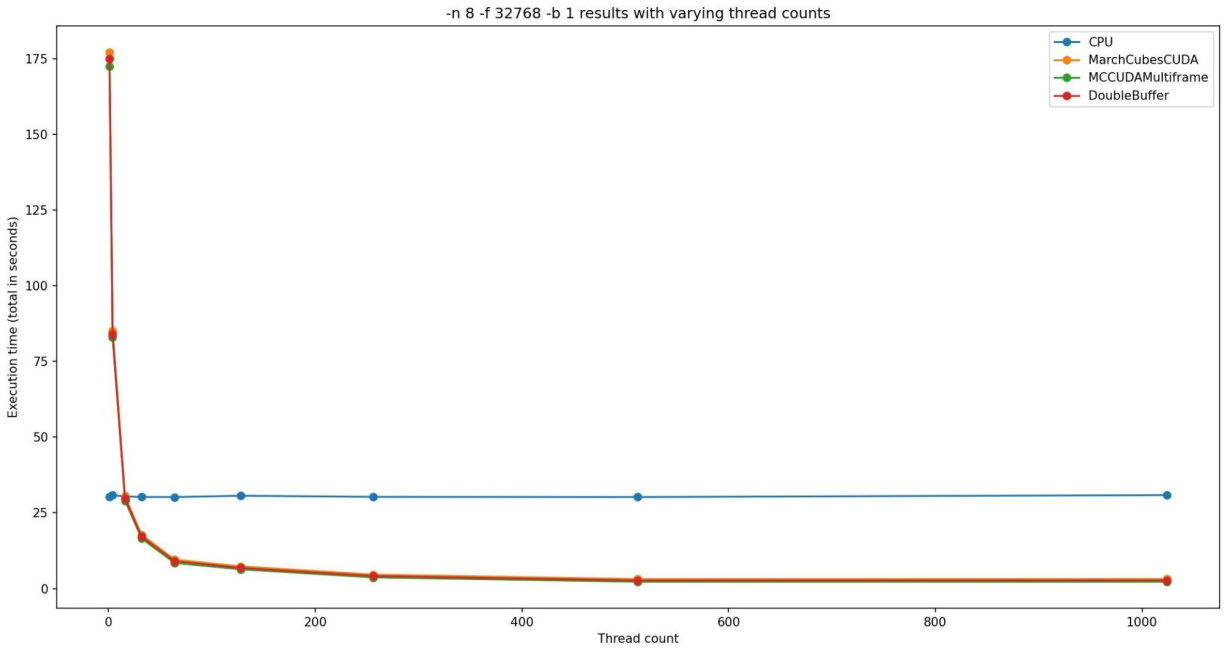



-n 256 -f 1 -b 1 results with varying thread counts
CPU
MarchCubesCUDA
MCCUDAMultiframe
DoubleBuffer
Execution time (total in seconds)
Thread count
0
20
40
60
80
0
200
400
600
800
1000


-n 64 -f 64 -b 1 results with varying thread counts
MarchCubesCUDA
MCCUDAMultiframe
Speed-up (in regard to cpu run time, calculated by million cubes per second)
Thread count
0
5
10
15
20
25
0
200
400
600
800
1000

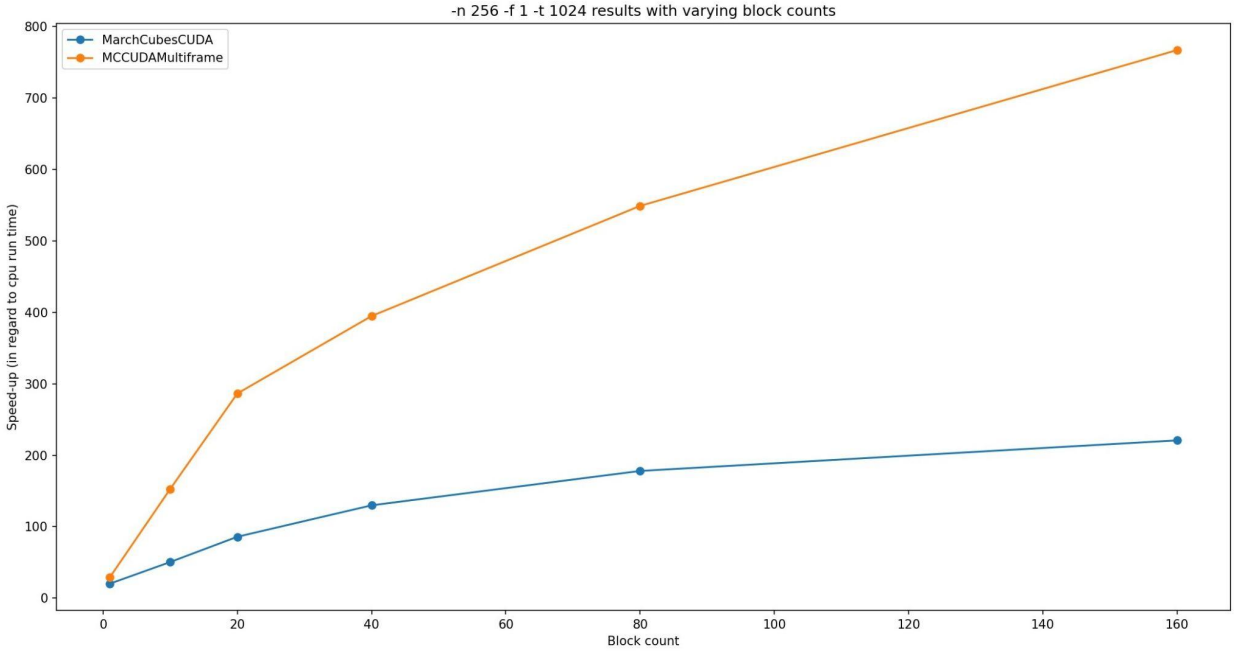

-n 64 -f 64 -b 1 results with varying thread counts
CPU
MarchCubesCUDA
MCCUDAMultiframe
DoubleBuffer
Execution time (total in seconds)
Thread count
0
20
40
60
80
100
0
200
400
600
800
1000

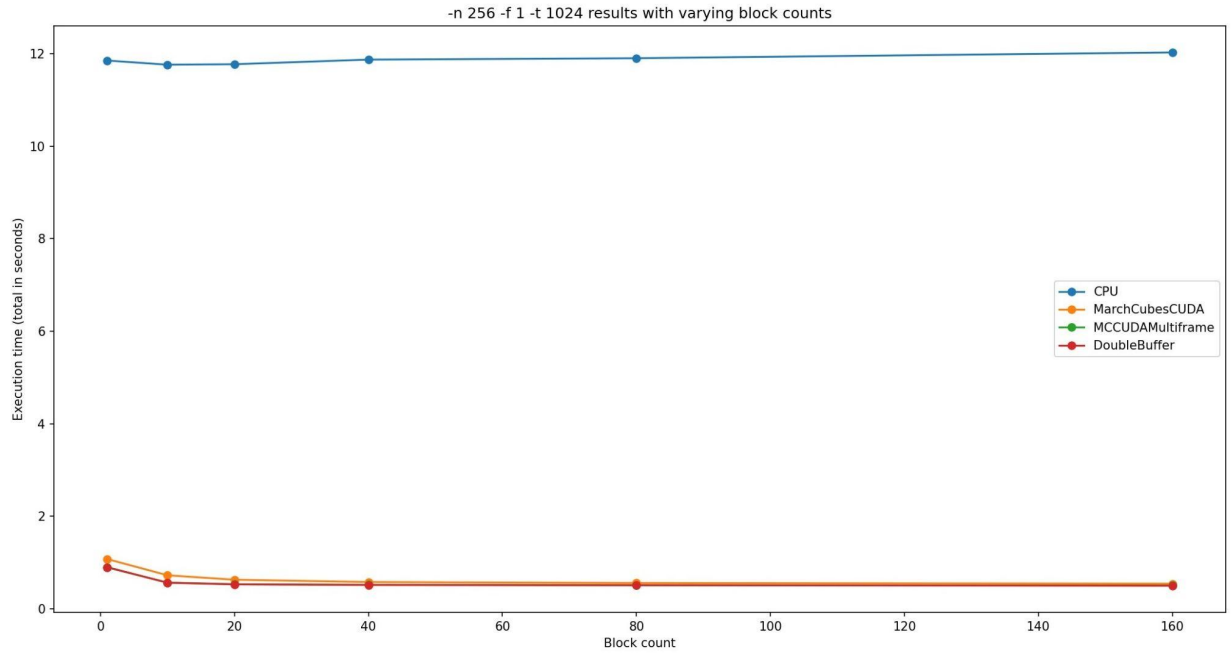

-n 8 -f 32768 -b 1 results with varying thread counts
MarchCubesCUDA
MCCUDAMultiframe
Speed-up (in regard to cpu run time, calculated by million cubes per second)
Thread count
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
0
200
400
600
800
1000

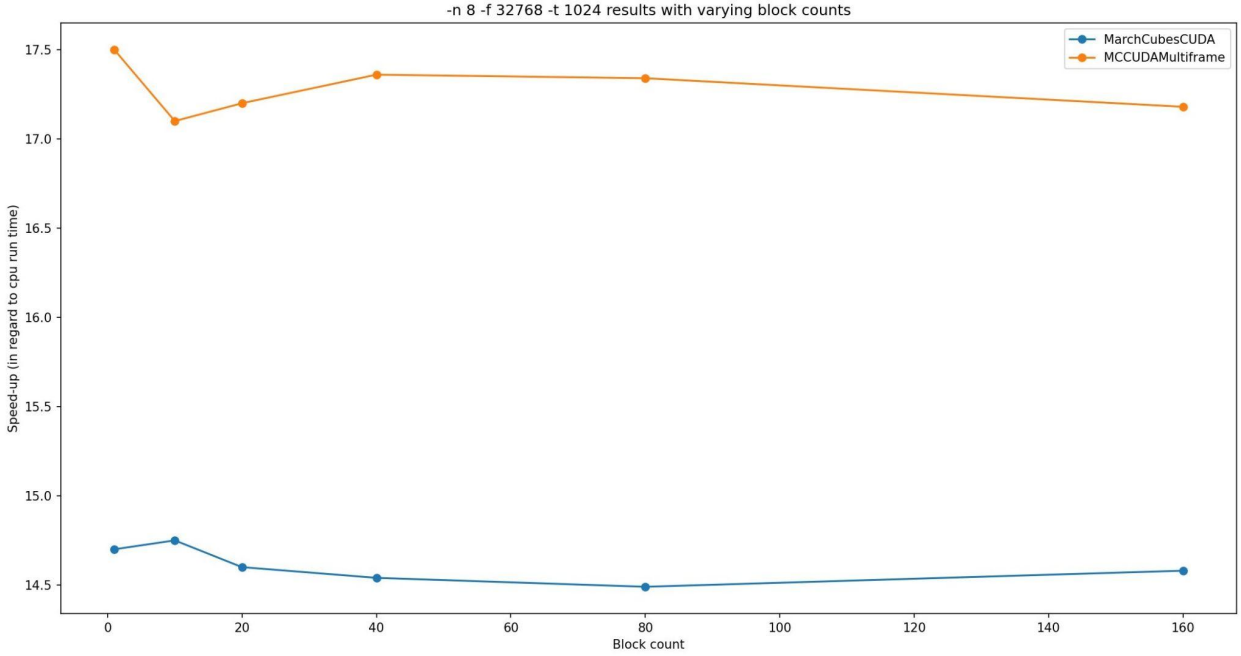


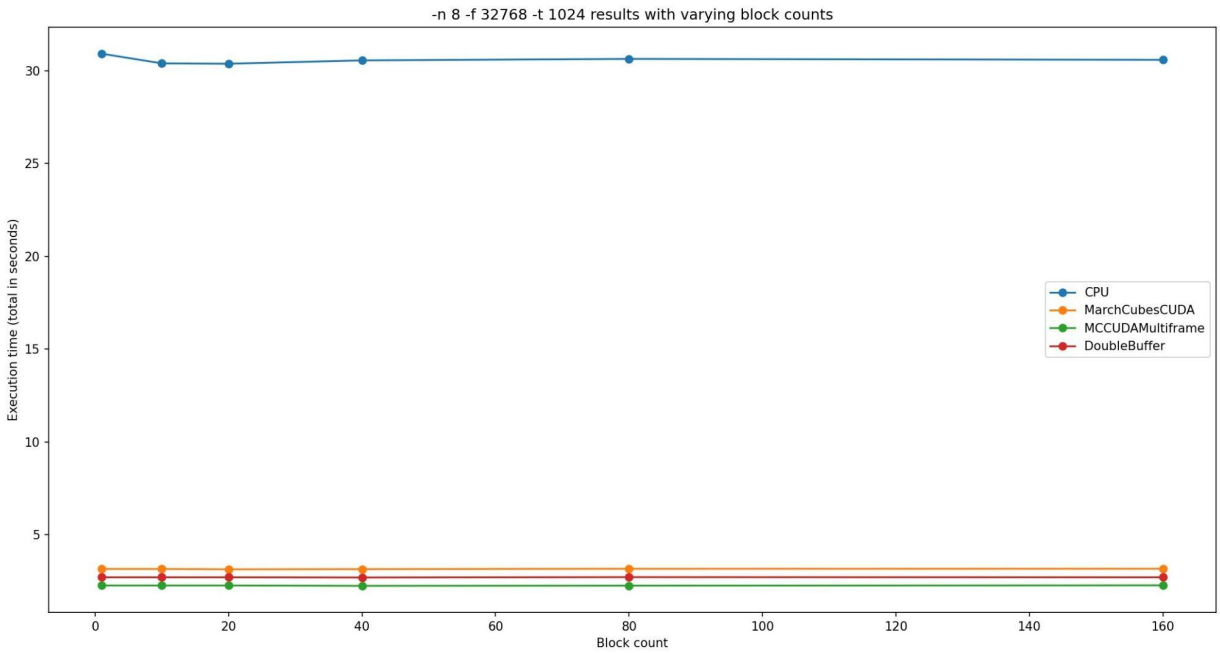
Part 2: (Varying block counts with thread per block count of 1024)

In this part, we experimented with varying block counts while keeping the thread count as 1024, as it was the fastest observed in experiment part 1. In the below figures, it can be seen that as the block count rises, so does the performance, if the problem size is large enough. In the second figure, it is seen that performance does not change much with the higher block counts, as the problem size is small enough that in all runs, individual threads calculate 1 cube anyways, and the excessive threads do not contribute to the calculations.




-n 256 -f 1 -t 1024 results with varying block counts
Execution time (total in seconds)
Block count
CPU
MarchCubesCUDA
MCCUDAMultiframe
DoubleBuffer
0
2
4
6
8
10
12
0
20
40
60
80
100
120
140
160


-n 8 -f 32768 -t 1024 results with varying block counts
Speed-up (in regard to cpu run time)
Block count
MarchCubesCUDA
MCCUDAMultiframe
14.5
15.0
15.5
16.0
16.5
17.0
17.5
0
20
40
60
80
100
120
140
160



MarchCubeCUDA and Double Buffer function comparisons (total time in seconds)

(-n 256 -f 1 -b 1)

Thread count	MarchCubeCUDA run time (s)	Double Buffer run time (s)
1	85.6	92.07
4	25.31	24.5
16	8.701	8.42
32	5.75	5.51
64	3.47	3.29
128	2.56	2.39
256	1.75	1.58
512	1.36	1.177
1024	1.076	0.896

(-n 64 -f 64 -b 1)

Thread count	MarchCubeCUDA run time (s)	Double Buffer run time (s)
1	102.37	100.49
4	36.39	35.87
16	15.13	14.88
32	9.254	9.026
64	4.547	4.379
128	3.528	3.374
256	2.363	2.217
512	1.582	1.447
1024	1.214	1.10

(-n 8 -f 32768 -b 1)

Thread count	MarchCubeCUDA run time (s)	Double Buffer run time (s)
1	177.15	174.9
4	85.12	84.03
16	30.45	29.7
32	17.8	17.17
64	9.58	9.07
128	7.31	6.82
256	4.62	4.151
512	3.128	2.69
1024	3.149	2.701

(-n 256 -f 1 -t 1024)

Block count	MarchCubeCUDA run time (s)	Double Buffer run time (s)
1	1.07	0.901
10	0.726	0.565
20	0.630	0.529
40	0.581	0.517
80	0.559	0.51
160	0.544	0.503

(-n 8 -f 32768 -t 1024)

Block count	MarchCubeCUDA run time (s)	Double Buffer run time (s)
1	3.15	2.702
10	3.155	2.69
20	3.131	2.70
40	3.143	2.699
80	3.168	2.71
160	3.162	2.703

You can find the 2 sbatch files we have written to post jobs to clusters. It includes requested gpu, memory and other required details. It also has commands for each of the experiments we were requested to perform.

You can also find the output files we got from completion of sbatch runs, that show the experiment results in a detailed way, though we believe it is not needed.