CUDA MARCHING CUBES - BEYZA ÇAVUŞOĞLU

CODE EXPLANATION:

Part 1&2: Parallel CUDA execution:

1D thread indexing is used but actually the iterations are done by thinking for 3D structure. Here, the important part is being able to calculate how 1D index can be used and 3D iterations in x,y,z can be performed. For that, I have searched online mathematical correspondence, and it can be done as below:

If you have the 1D array A[index] and you want to see what that corresponds to in 3D,

```
width_index=index/(height*depth);  //Note the integer division . This is x
height_index=(index-width_index*height*depth)/depth; //This is y
depth_index=index-width_index*height*depth- height_index*depth;//This is z
```

```
int thread_idx = (blockDim.x * blockIdx.x + threadIdx.x) * NumCubes ;

int x_axis = thread_idx / (NumY*NumZ);
int y_axis = (thread_idx - x_axis * NumY * NumZ) / NumZ;
int z_axis = thread_idx - x_axis * NumZ * NumY - y_axis * NumZ  ;
```

Instead of 3 nested loops as in the serial version, we can utilize parallel implementation, threads will iterate as if they are having 3d indexes in GPU.

Part 3: Inter-Frame + Intra-Frame Parallelization:

```
    /////////////////////////////////////////////////////////////////////
    //                        PART 1&2 RUN:                          //
    /////////////////////////////////////////////////////////////////////
    if (part == -1 || part == 1 || part == 2)
    {
        for (frame = 0; frame < frameNum; frame++)
        {
```

Normally in part 1&2, we were doing kernel launch per frame as shown above figure and this was called the discrete kernel approach. Instead, in this part, we will only launch the kernel one time, but the loop iterating each frame will be inside the kernel which is called a persistent kernel.

```
    for (int i = 0; i < NumFrames; i++)
{

        int x_axis = thread_idx / (NumY*NumZ);
        int y_axis = (thread_idx - x_axis * NumY * NumZ) / NumZ;
        int z_axis = thread_idx - x_axis * NumZ * NumY - y_axis * NumZ  ;

        int FrameOffsetInd = i * FrameOffset ;

    for (int j = 0; j < NumCubes; j++)
    {
        // check the limits
        if(y_axis == NumY) { // if out of boundaries for y
            y_axis = 0;
            x_axis += 1;
        }

        if(z_axis == NumZ) { // if out of boundaries for z
            z_axis = 0;
            y_axis += 1;
        }

        float x = domainP->min.x + x_axis * cubeSizeP->x;
        float y = domainP->min.y + y_axis * cubeSizeP->y;
        float z = domainP->min.z + z_axis * cubeSizeP->z;
```

The frame loop in the main() function is actually moved inside the kernel, launching only happens ones but frame iteration is inside the kernel as it can be seen above.

Part 4 - Double Buffer:

The host is automatically asynchronous with kernel launches and we can use streams to control asynchronous behavior. Enabling concurrency with streams is possible (Reference: https://on-demand.gputechconf.com/gtc/2014/presentations/S4158-cuda-streams-best-practices-common-pitfalls.pdf). A stream is a queue of device work, the host places work in the queue and continues on immediately, device schedules work from streams when resources are free. Operations within the same stream are ordered and cannot overlap, but operations in different streams are unordered and can overlap. So, in this part, we need to use different streams to overlap execution and data movement, we can compute frame i in 1 stream and frame (i-1) data can be sent in another stream.

```
  cudaStream_t stream1;
  cudaStream_t stream2;
  cudaStream_t stream3;
  cudaStream_t stream4;

  cudaStreamCreate(&stream1);
  cudaStreamCreate(&stream2);
  cudaStreamCreate(&stream3);
  cudaStreamCreate(&stream4);
```

First, the streams should be created. Stream 1 and stream 4 is used for kernel launch, stream 2 and stream 3 is used for moving the meshVertices and meshNormals data separately to be able to overlap the work more.

While Stream 1 is doing the kernel computation on “meshVertices_d” for even frames, in the meantime Stream 4 will do kernel launch on the “meshVertices_d + frameSize” (which is the data’s next frame portion) for the odd frames. So, the data movement for even frames is done for “meshVertices_d + frameSize” which is the reverse logic (that part of vertices are only manipulated for odd vertices by Stream 4. The code for the related logic is shown below:

```
if (frame % 2 == 0){
    MarchCubeCUDATwoPointers<<<numBlocks, numThreads, 0, stream1>>>(domain_d, cubeSize_d, twist, 0, meshVertices_d, meshNormals_d, NumCubes_Thread);
} else {
    // while for even frames meshvertices are done, in that time stream4 can do the next frame
    MarchCubeCUDATwoPointers<<<numBlocks, numThreads, 0, stream4>>>(domain_d, cubeSize_d, twist, 0, meshVertices_d + frameSize, meshNormals_d + frameSize, NumCubes_Threa
}
if (frame % 2 == 0){
     if (frame > 0){ // not the first stream bcz if its first, nothing to carry yet
        // wait kernel launch to finish first by stream1
        cudaStreamSynchronize(stream1);
        cudaStreamSynchronize(stream4);

        // Use 2 different streams to move vertices and normals , overlapping is better :)
        cudaMemcpyAsync(meshVertices_h + offset - frameSize, meshVertices_d + frameSize, frameSize * sizeof(float3), cudaMemcpyDeviceToHost, stream2);
        cudaMemcpyAsync(meshNormals_h + offset - frameSize, meshNormals_d + frameSize, frameSize * sizeof(float3), cudaMemcpyDeviceToHost, stream3);
        cudaMemset(meshNormals_d + frameSize, 0, frameSize * sizeof(float3));
        cudaMemset(meshVertices_d + frameSize, 0, frameSize * sizeof(float3));
     }
} else {
    // wait kernel launch to finish first by stream1
    cudaStreamSynchronize(stream1);
    cudaStreamSynchronize(stream4);

    // Use 2 different streams to move vertices and normals , overlapping is better :)
    cudaMemcpyAsync(meshVertices_h + offset - frameSize, meshVertices_d, frameSize * sizeof(float3), cudaMemcpyDeviceToHost, stream2);
    cudaMemcpyAsync(meshNormals_h + offset - frameSize, meshNormals_d, frameSize * sizeof(float3), cudaMemcpyDeviceToHost, stream3);
    cudaMemset(meshNormals_d, 0, frameSize * sizeof(float3));
    cudaMemset(meshVertices_d, 0, frameSize * sizeof(float3));

}
```

Note: MarchCubeCUDATwoPointers kernel will actually be same as MarchCubeCUDA in Part 2. The difference is in the main() when doing kernel launch and data copying.

EXPERIMENT RESULTS:

For running the different configurations, a shell script (autom.sh) is prepared. Due to -shorter, I had to shrink the n size to smaller.

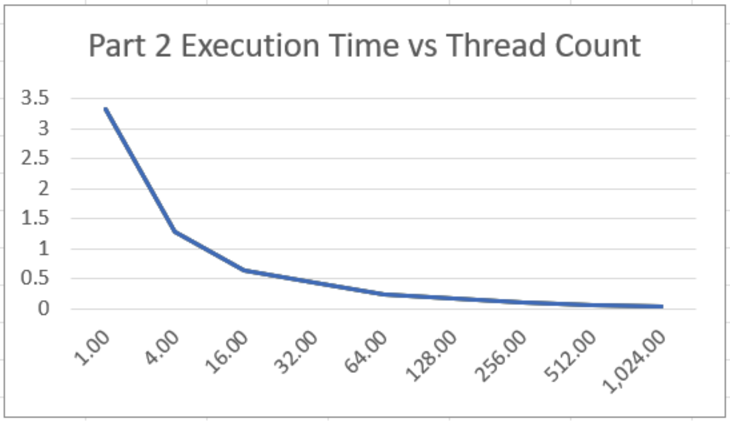
PART 1) SINGLE CPU VS SINGLE GPU THREAD:

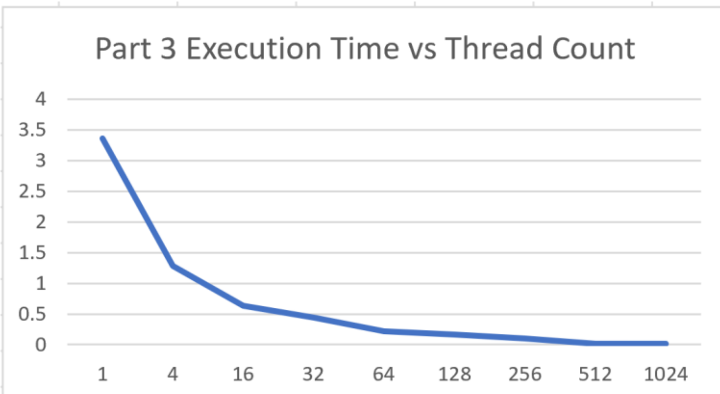
Experiments:	**Single CPU Execution Time (sec)**	**Single GPU Execution Time (sc)**
n = 64, f = 1	1.610471	3.320509
n = 8, f = 64	1.643373	1.341422
n = 4, f = 32768	1.610588	76.407076

Single CPU is performing better than Single GPU in different configurations. The reason can be the data movement and kernel launch overheads can be too much for GPUs in 1 thread compared to CPU. But when the thread count is increased, GPU execution time will be much more lower than CPU, because parallelization benefits will be more than overheads.

EXPERIMENT 1) Strong Scaling with different thread counts (1, 4, 16, 32, 64, 128, 256, 512, 1024):

- **For n = 64, f = 1:**

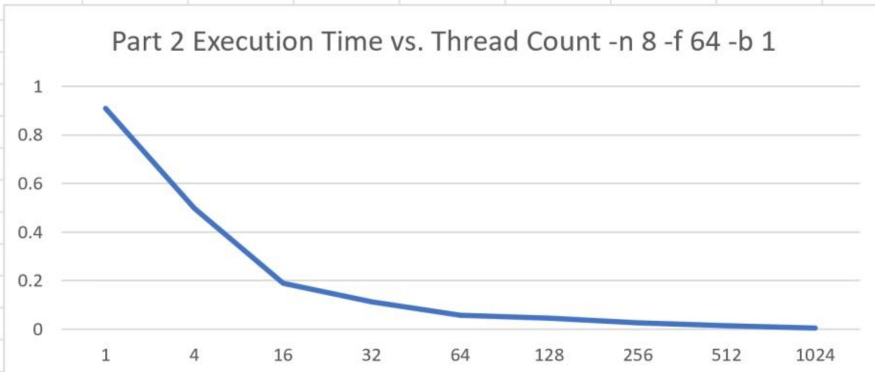


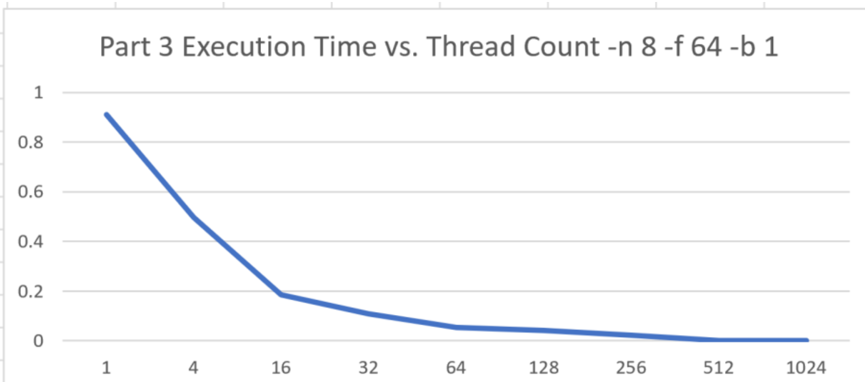


-n 64 -f 1 -b 1							
Thread Count	Serial Execution Time	Part 2 Execution Time	Part 3 Exe. Time	Part 4 Exe. Time	Speed Up - Part 2	Speed Up - Part 3	Speed Up - Part 4
1	1.610471	3.320509	3.362006	3.301432	0.485007268	0.479020858	0.487809835
4	1.610471	1.291057	1.285602	1.254783	1.247405033	1.252697958	1.283465747
16	1.610471	0.647843	0.632286	0.613726	2.485897046	2.547060982	2.624087948
32	1.610471	0.450599	0.436334	0.418098	3.574066964	3.690913383	3.851898359
64	1.610471	0.237769	0.219646	0.204203	6.773258919	7.332120776	7.886617728
128	1.610471	0.179014	0.161148	0.14552	8.996341068	9.993738675	11.06700797
256	1.610471	0.11786	0.10036	0.084943	13.66427117	16.04694101	18.95943162
512	1.610471	0.071872	0.015084	0.039656	22.40748831	106.766839	40.61102986
1024	1.610471	0.031809	0.015073	0.000009	50.62941306	106.8447555	50.62941306

For this configuration, the best speedup results are for 1024 threads. For all, one thing can be noticed when the thread count is increased, the execution time decreases. When the parts are compared, Part 4 is the fastest one in general.

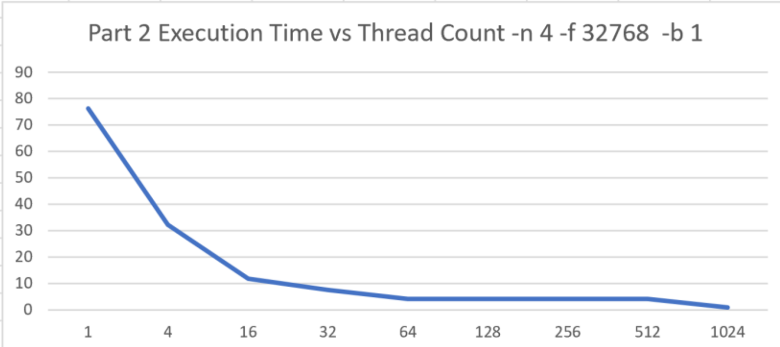
- **For n = 8, f = 64:**

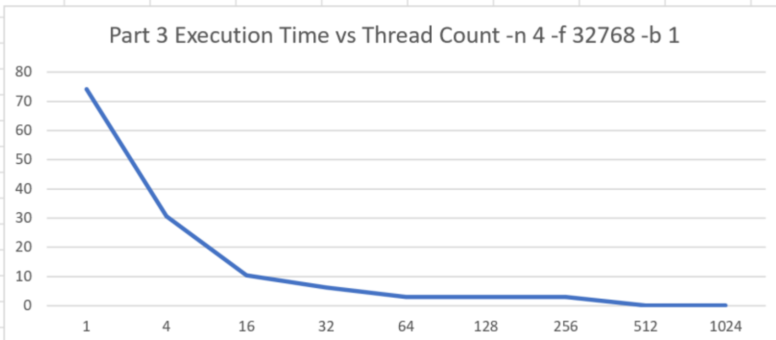


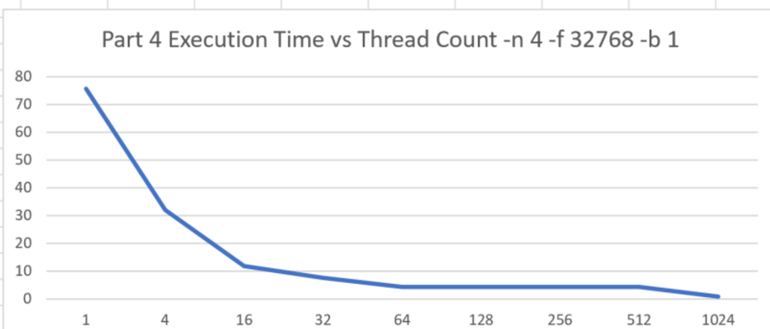


			-n 8 -f 64 -b 1				
Thread Count	Serial Execution Time	Part 2 Execution Time	Part 3 Exe. Time	Part 4 Exe. Time	Speed Up - Part 2	Speed Up - Part 3	Speed Up - Part 4
1	1.643373	0.909367	0.910844	0.895566	1.807161465	1.804231021	1.835010485
4	1.643373	0.497726	0.495259	0.487451	3.301762415	3.31820926	3.371360403
16	1.643373	0.189898	0.186229	0.186192	8.653977398	8.824474169	8.826227765
32	1.643373	0.112362	0.109343	0.110024	14.62570086	15.02952178	14.93649567
64	1.643373	0.057367	0.053941	0.055746	28.64666097	30.46612039	29.47965773
128	1.643373	0.044321	0.041327	0.043119	37.07887909	39.76511724	38.11250261
256	1.643373	0.025391	0.022673	0.024735	64.72265763	72.48149782	66.43917526
512	1.643373	0.014842	0.001945	0.014521	110.724498	844.9218509	113.1721645
1024	1.643373	0.004087	0.001914	0.003385	402.0976266	858.6065831	485.4868538

- **For n = 4, f = 32768:**



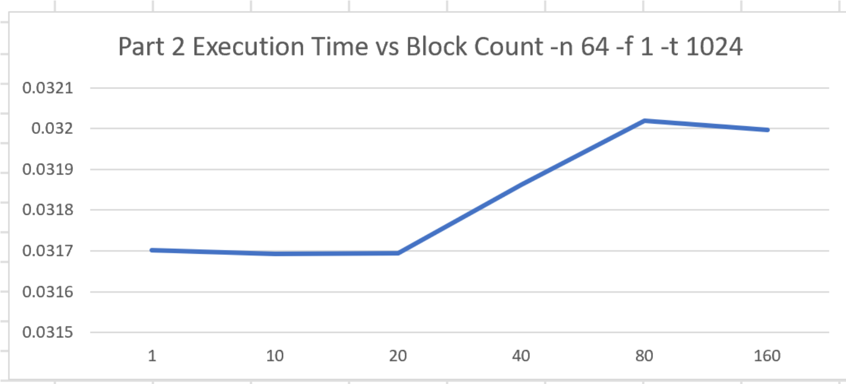


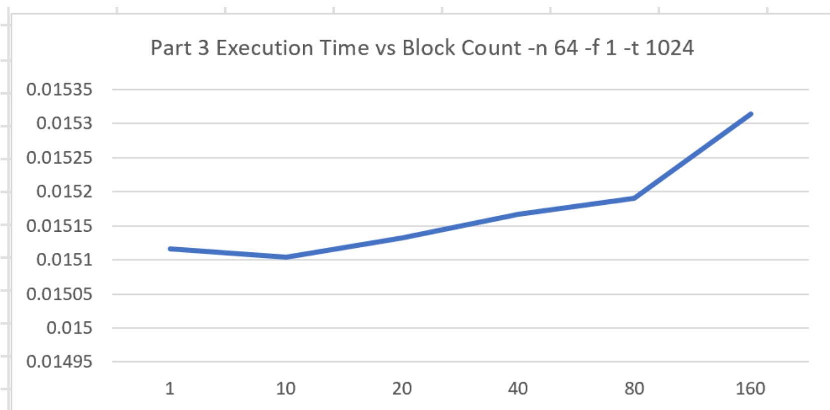


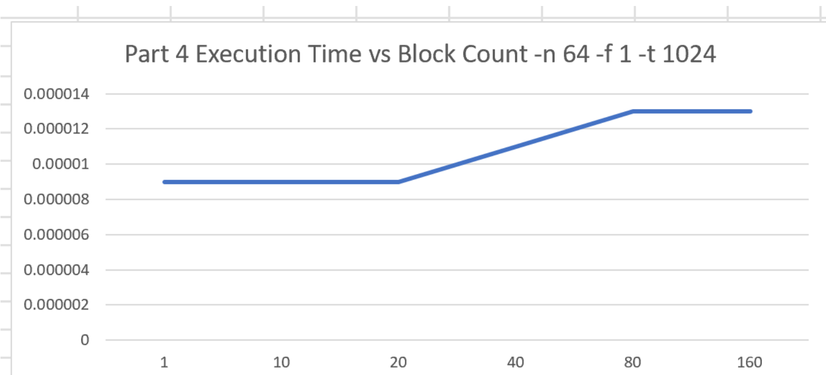
-n 4 -f 32768 -b 1							
Thread Count	Serial Execution Time	Part 2 Execution Time	Part 3 Exe. Time	Part 4 Exe. Time	Speed Up - Part 2	Speed Up - Part 3	Speed Up - Part 4
1	1.610588	76.407076	74.252094	75.617326	0.021079042	0.021690809	0.021299193
4	1.610588	32.196536	30.649677	31.952582	0.050023642	0.052548286	0.050405567
16	1.610588	11.911204	10.50802	11.802872	0.135216222	0.153272263	0.136457296
32	1.610588	7.566233	6.265176	7.494319	0.21286524	0.257069873	0.214907852
64	1.610588	4.238077	3.006309	4.212116	0.380028017	0.535736014	0.382370286
128	1.610588	4.229209	3.000419	4.204436	0.380824878	0.536787695	0.38306874
256	1.610588	4.228284	2.998677	4.193567	0.380908189	0.537099528	0.384061588
512	1.610588	4.242701	0.120418	4.215714	0.379613836	13.37497716	0.382043943
1024	1.610588	0.985105	0.12054	0.728299	1.634940438	13.36144019	2.211437885

EXPERIMENT 2) Fixed Thread Count but Different Block Counts (1, 10, 20, 40, 80, 160):

- **For n = 64, f = 1:**

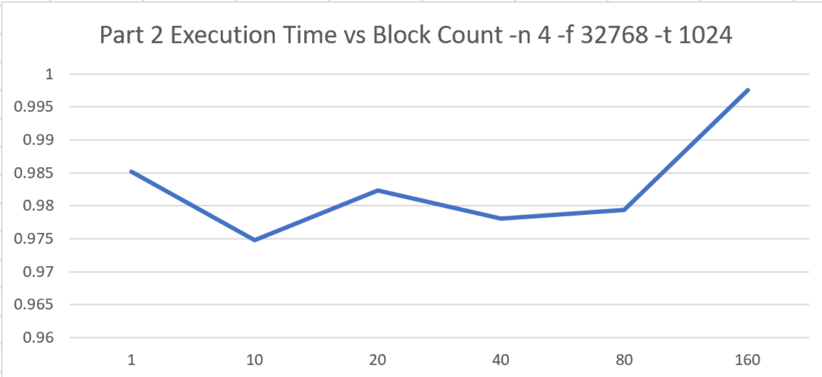


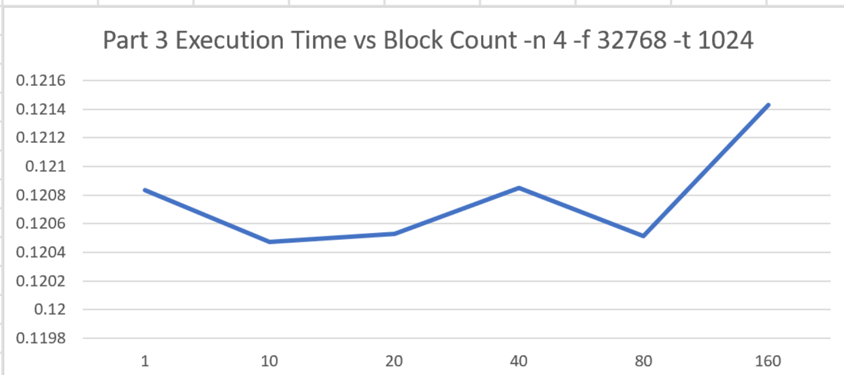


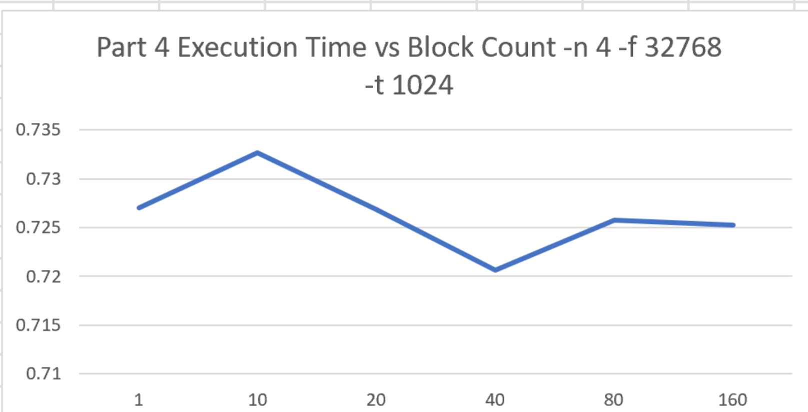


-n 64 -f 1 -t 1024							
Block Count	Serial Execution Time	Part 2 Execution Time	Part 3 Exe. Time	Part 4 Exe. Time	Speed Up - Part 2	Speed Up - Part 3	Speed Up - Part 4
1	1.610471	0.031701	0.015116	0.000009	50.80189899	106.5408177	178941.2222
10	1.610471	0.031692	0.015104	0.000009	50.81632589	106.6254635	178941.2222
20	1.610471	0.031694	0.015132	0.000009	50.8131192	106.4281655	178941.2222
40	1.610471	0.031863	0.015167	0.000011	50.54360857	106.1825674	146406.4545
80	1.610471	0.03202	0.01519	0.000013	50.29578389	106.0217907	123882.3846
160	1.610471	0.031997	0.015314	0.000013	50.33193737	105.1633146	123882.3846

- **For n = 4, f = 32768:**







			-n 4 -f 32768 -t 1024				
Block Count	Serial Execution Time	Part 2 Execution Time	Part 3 Exe. Time	Part 4 Exe. Time	Speed Up - Part 2	Speed Up - Part 3	Speed Up - Part 4
1	1.610588	0.985189	0.120836	0.727037	1.634801038	13.32870999	2.215276527
10	1.610588	0.974739	0.12047	0.73267	1.652327444	13.36920395	2.198244776
20	1.610588	0.982285	0.12053	0.726821	1.639634118	13.36254874	2.215934873
40	1.610588	0.978084	0.12085	0.72068	1.646676564	13.32716591	2.234817117
80	1.610588	0.979413	0.120512	0.725758	1.64444213	13.36454461	2.219180498
160	1.610588	0.997588	0.121433	0.725245	1.614482131	13.26318217	2.220750229

Memcpy overhead increases when the problem size increase, when its small the overhead is not effecting the results that much. The best thread count is 1024 and the best block count is 40 for my results. And I also observed the help of dividing the work into streams to overlap the execution and data movement, it increased speedup.