COMP 429/529: Project 2

Muhammed Esad Simitcioglu

December 24, 2023

In Part 1, I parallelized the sequential Marching Cubes algorithm using CUDA. I executed it with a single thread to compare the performance difference between 1 CPU thread and 1 GPU thread.
In Part 2, I divided the computation of each cube into threads using CUDA. I conducted experiments with various thread and block combinations and documented my observations.
In Part 3, I calculated all frames in a single kernel launch rather than creating a kernel for each frame. This approach aimed to eliminate kernel overhead.
In Part 4, I designed two separate pipelines (streams) to calculate one frame and copy the previous frame's solution to a file. By doing this, I managed to overlap two independent operations using streams and events.

Important note 1: I parallelize the algorithms with respect to a 1D thread block. In the experiments, it is specified that I should use "threadNum," but it is unclear whether it refers to the count in one dimension (1D threadNum) or the sum across all dimensions. I have opted to use a 1D thread block to mitigate any uncertainty and potential risks.

Important Note 2: In all experiments, I divided the value of 'n' by two due to hardware limitations and guidance from the TA. Additionally, I could not conduct experiments involving the calculation of 32768 frames due to hardware limitations.

Important Note 3: I have omitted the additional experiment details, such as those about the kernel, memcpy, and other aspects, to keep the report concise. The results of the experiments can be found in the folder that I have submitted.

1 Part 1: Single thread with CUDA

In the first part of this assignment, I developed a parallelized version of the sequential marching cube algorithm. This involved integrating CUDA API calls and altering certain variable declarations, such as omitting the 'new' keyword and allocating memory in the main using cudaMalloc().

```
    __global__ void MarchCubeCUDA(
    int blockNum,
    int threadNum,
    Rect3 *domainP,
    float3 *cubeSizeP,
    float twist,
    float isoLevel,
    float3 *meshVertices,
    float3 *meshNormals)
{

    int NumX = static_cast<int>(ceil(domainP->size.x / cubeSizeP->x
    ));
    int NumY = static_cast<int>(ceil(domainP->size.y / cubeSizeP->y
    ));
    int NumZ = static_cast<int>(ceil(domainP->size.z / cubeSizeP->z
    ));

    int index = blockIdx.x * blockDim.x + threadIdx.x;

    for (int ix = index * (NumX / (threadNum / blockNum)); ix <=
    index * (NumX / (threadNum / blockNum)) + (NumX / (threadNum /
    blockNum)); ix++)
    {
        for (int iy = 0; iy < NumY; ++iy)
        {
            for (int iz = 0; iz < NumZ; ++iz)
            {
                float3 intersect[12];

                float x = domainP->min.x + ix * cubeSizeP->x;
                float y = domainP->min.y + iy * cubeSizeP->y;
                float z = domainP->min.z + iz * cubeSizeP->z;

                float3 min = make_float3(x, y, z);

                // create a cube made of 8 vertices
                float3 pos[8];
                float sdf[8];

                Rect3 space = {min, *cubeSizeP};

                float mx = space.min.x;
                float my = space.min.y;
                float mz = space.min.z;

                float sx = space.size.x;
                float sy = space.size.y;
                float sz = space.size.z;

                pos[0] = space.min;
                pos[1] = make_float3(mx + sx, my, mz);
                pos[2] = make_float3(mx + sx, my, mz + sz);
                pos[3] = make_float3(mx, my, mz + sz);
                pos[4] = make_float3(mx, my + sy, mz);
                pos[5] = make_float3(mx + sx, my + sy, mz);
                pos[6] = make_float3(mx + sx, my + sy, mz + sz);
                pos[7] = make_float3(mx, my + sy, mz + sz);

                // fill in the vertices of the cube
                for (int i = 0; i < 8; ++i)
                {
                    float sd = opTwist(pos[i], twist);
                    if (sd == 0)
                        sd += 1e-6;
                    sdf[i] = sd;
                }

                // map the vertices under the isosurface to intersecting edges
                int signConfig = Intersect(pos, sdf, intersect, isoLevel);

                // now create and store the triangle data
                int offset = (NumZ * NumY * ix + NumZ * iy + iz) * 16;

                Triangulate(twist, meshVertices + offset, meshNormals + offset, signConfig, intersect);
            }
        }
    }
}
```

I utilized identical code for both Part 1 and Part 2. The code executes on a single-thread GPU by setting the thread count and block count to 1, allowing for straightforward comparisons between CPU and GPU thread performance.

1.1 Experiment

1.1.1 Test with Resoulition 32

Frame Count 1 - 1 CPU Thread Execution Time: 0.030580

Frame Count 1 - 1 GPU Thread Execution Time: 0.19608

Frame Count 5 - 1 CPU Thread Execution Time: 0.185170

Frame Count 5 - 1 GPU Thread Execution Time: 0.881040

Frame Count 10 - 1 CPU Thread Execution Time: 0.368987

Frame Count 10 - 1 GPU Thread Execution Time: 1.750602

1.1.2 Test with Resoulition 128

Frame Count 1 - 1 CPU Thread Execution Time: 1.532970

Frame Count 1 - 1 GPU Thread Execution Time: 8.968515

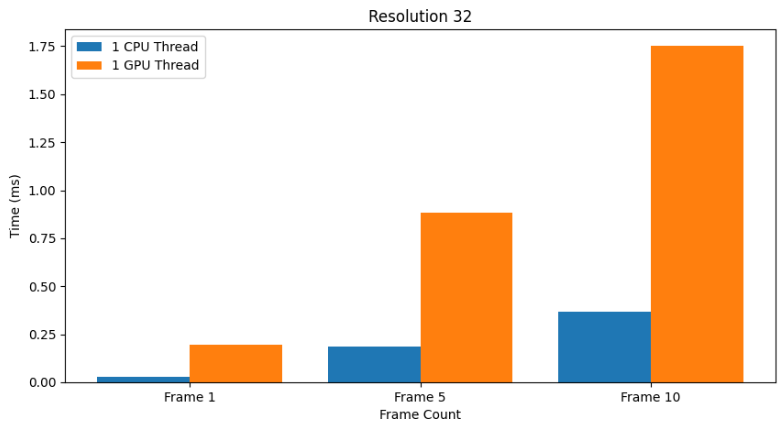
Frame Count 5 - 1 CPU Thread Execution Time: 9.072800

Frame Count 5 - 1 GPU Thread Execution Time: 45.438270

Frame Count 10 - 1 CPU Thread Execution Time: 18.520537

Frame Count 10 - 1 GPU Thread Execution Time: 90.980972

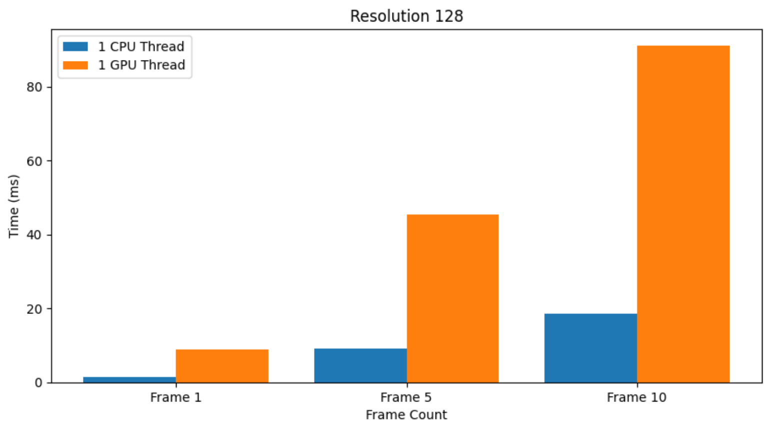
Results



(a) Resolution 32 Execution Time Result

Explanation of Speedup Curve

There's a notable difference in execution time between using 1 CPU thread and 1 GPU thread, with the CPU thread often being faster. This discrepancy is due to several factors. For instance, launching a kernel to a GPU thread incurs various overheads, such as synchronization, communication, and data transfer (since data needs to be moved from device to host). Additionally, GPU threads are optimized for distributing work across many threads rather than performing large operations per thread, making them individually less powerful than CPU threads. The significant performance difference becomes more apparent as I increase the resolution.



(a) Resolution 128 Execution Time Result

2 Part 2: Parallel CUDA execution

In the second part of this assignment, I utilized the same code from the first part. The primary difference lies in experimenting with various thread and block numbers to truly understand the advantages of parallelization over serial execution on the CPU.

2.1 Experiment 1

2.1.1 Test with Resoulition 32 - Frame Count 64

Serial version Execution Time: 0.030580

Thread 1 Execution Time : 10.176974
Speedup: 0.003007

Thread 4 Execution Time: 4.564233
Speedup: 0.006694

Thread 16 Execution Time: 2.399325
Speedup: 0.012748

Thread 32 Execution Time: 1.806730
Speedup: 0.016903

Thread 64 Execution Time: 0.307173
Speedup: 0.099529

Thread 128 Execution Time: 0.306576
Speedup: 0.099651

Thread 256 Execution Time: 0.320545
Speedup: 0.095337

Thread 512 Execution Time: 0.354043
Speedup: 0.086305

Thread 1024 Execution Time: 0.073733
Speedup: 0.414479

2.1.2 Test with Resoulition 128 - Frame Count 1

Serial version Execution Time: 1.532970

Thread 1 Execution Time : 8.991048
Speedup: 0.170466

Thread 4 Execution Time: 2.714463
Speedup: 0.564852

Thread 16 Execution Time: 1.120137
Speedup: 1.368676

Thread 32 Execution Time: 0.864296
Speedup: 1.773806

Thread 64 Execution Time: 0.565373
Speedup: 2.710664

Thread 128 Execution Time: 0.474325
Speedup: 3.235706

Thread 256 Execution Time: 0.139285
Speedup: 11.026192

Thread 512 Execution Time: 0.147881
Speedup: 10.363774

Thread 1024 Execution Time: 0.073257
Speedup: 20.943461

2.2 Experiment 2

2.2.1 Test with Resoulition 128 - Frame Count 1

Serial version Execution Time: 1.532970

Block 1 Execution Time : 0.139332
Speedup: 11.00

Block 10 Execution Time: 1.020594
Speedup: 1.50

Block 20 Execution Time: 1.499038

Speedup: 1.02

Block 40 Execution Time: 2.477216

Speedup: 0.62

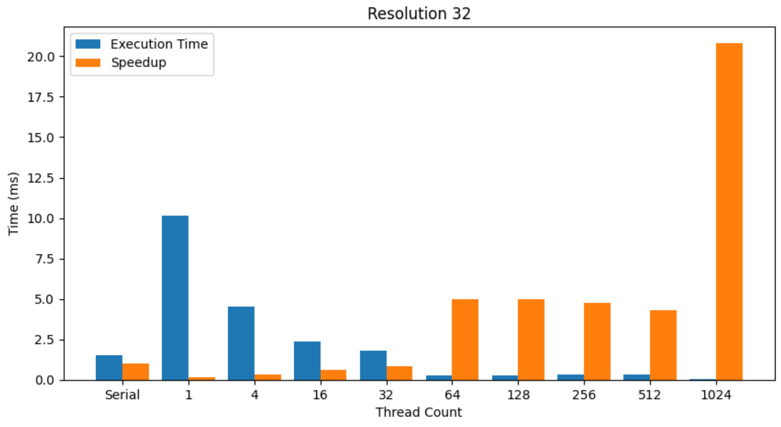
Block 80 Execution Time: 4.365658

Speedup: 0.35

Block 160 Execution Time: 8.149004

Speedup: 0.19

Results

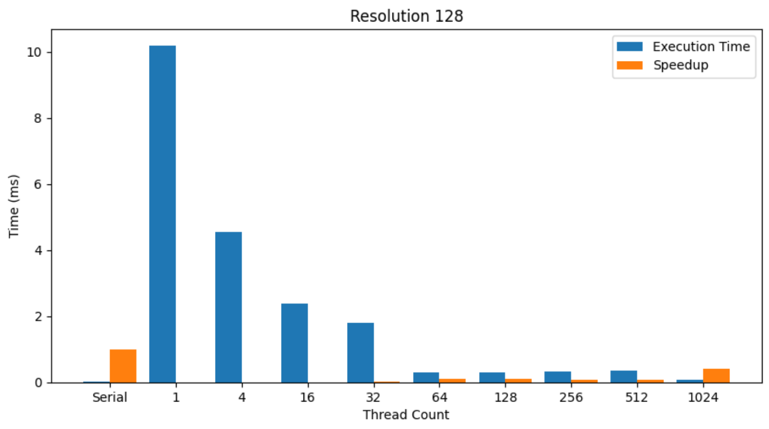


(a) Resolution 32 Execution Time Result

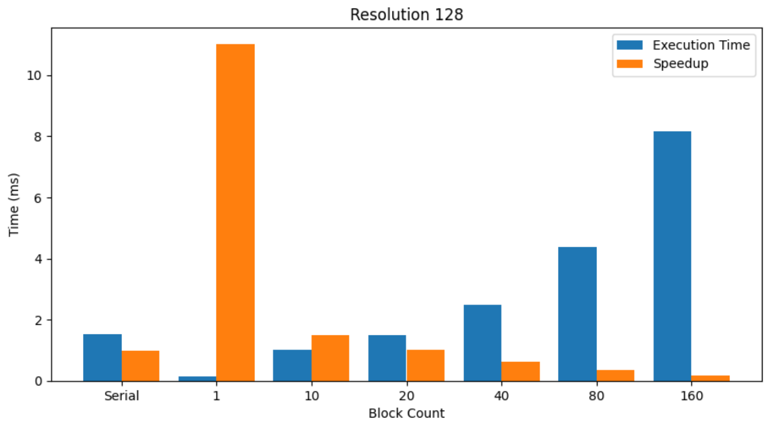
Have you observed any issues with high thread counts while working on this part? If so, why do you think it happened?

In the process of addressing this part of the assignment, I encountered issues when using a high thread count. The main problem stems from the fact that having an excessive number of threads can be counterproductive. This is because, in our design, I need to distribute individual cubes among the threads. If the number of threads greatly exceeds the dimensions of the cubes (specifically, NumX, NumY, and NumZ), many threads end up remaining idle. This idleness introduces unnecessary overhead and, consequently, diminishes performance. Furthermore, allocating threads beyond the GPU's capacity can lead to practical issues. In some cases, when I attempted to allocate an exceptionally high thread count, such as 1024 threads, it resulted in kernel launch failures due to hardware limitations. This occurrence explains the anomalous values observed with such high thread counts.

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



(a) Resolution 128 Execution Time Result



(a) Resolution 128 Execution Time Result

The optimal performance is achieved when using 256 threads but it changes when I change the problem size and frame count. This choice aligns with the algorithm's requirements, as it necessitates a fixed number of threads to traverse the cube dimensions: NumX, NumY, and NumZ. Deploying more threads than necessary doesn't enhance computational efficiency but introduces additional overhead to the process. Therefore, avoiding using significantly more threads than required is advisable to achieve the best performance.

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

As I increase the problem size, both the memcpy and kernel overheads tend to increase. The exact relationship between these two overheads varies depending on the problem size and frame number parameters. In certain scenarios, there are settings where the memcpy and kernel overheads approach parity, resulting in roughly equal values. For instance, in a specific configuration with n=128 (problem size), f=1 (frame count), and t=128 (thread count), I observe that the memcpy and kernel overheads become approximately equal. However, it's important to note that achieving this equilibrium is influenced by various factors, and the specific settings where they match can differ in other contexts.

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

In the third part of this assignment, I undertook a significant code refactoring to implement a persistent kernel approach, where the frame loop was integrated within the kernel itself. This architectural change addressed the challenges observed in Part 2 while optimizing kernel execution overhead and memory copy operations. The foremost objective of this refactoring was to alleviate the kernel execution overhead that was evident in Part 2. In Part 2, each frame required a separate kernel launch, resulting in substantial setup and management overhead. In contrast, in Part 3, I adopted a more efficient approach by leveraging a single kernel launch to process all frames. This strategy significantly reduced the kernel execution time, particularly when dealing with a large number of frames, thus making it more efficient and scalable in such scenarios. However, this optimization came with an associated trade-off in the form of increased memory copy overhead. In Part 3, I now copy the outputs of all frames from the device to the host memory simultaneously. This can result in higher memory copy times, especially when the problem size is relatively small and the frame count is substantial.

```
__global__ void MarchCubeCUDAMultiframe(
    int blockNum,
    int threadNum,
    Rect3 *domainP,
    float3 *cubeSizeP,
    int frameNum,
    float maxTwist,
    float isoLevel,
    float3 *meshVertices,
    float3 *meshNormals,
    float3 *meshVertices_buffer,
    float3 *meshNormals_buffer)
{
                    // Rest of the code...

                    Triangulate(twist, meshVertices + offset, meshNormals + offset, signConfig, intersect);

                    int totalNumVertices = NumX * NumY * NumZ * 16;

                    // Calculate the global index for the current thread's cube
                    int globalIdx = (NumZ * NumY * ix + NumZ * iy + iz) * 16;

                    // Copy calculated data to buffers for each vertex of the cube
                    for (int i = 0; i < 16; ++i)
                    {
                        int idx = globalIdx + i;

                        // Ensure we're within the bounds of the mesh arrays
                        if (idx < totalNumVertices)
                        {
                            // Copy data to buffers
                            meshVertices_buffer[idx + totalNumVertices * frame] = meshVertices[idx];
                            meshNormals_buffer[idx + totalNumVertices * frame] = meshNormals[idx];

                            // Reset the original mesh data for the next iteration
                            meshVertices[idx] = make_float3(0, 0, 0);
                            meshNormals[idx] = make_float3(0, 0, 0);
                        }
                    }
}
```

3.1 Experiment 1

3.1.1 Test with Resoulition 32 - Frame Count 64

Serial version Execution Time: 0.030580

Thread 1 Execution Time : 22.781873
Speedup: 0.001342

Thread 4 Execution Time: 7.873548
Speedup: 0.003878

Thread 16 Execution Time: 2.834909
Speedup: 0.010778

Thread 32 Execution Time: 1.937560
Speedup: 0.015777

Thread 64 Execution Time: 0.167604
Speedup: 0.182056

Thread 128 Execution Time: 0.169846
Speedup: 0.179968

Thread 256 Execution Time: 0.168440
Speedup: 0.181398

Thread 512 Execution Time: 0.168160
Speedup: 0.181600

Thread 1024 Execution Time: 0.167807
Speedup: 0.181843

3.1.2 Test with Resoulition 128 - Frame Count 1

Serial version Execution Time: 1.532970

Thread 1 Execution Time : 22.671929
Speedup: 0.067691

Thread 4 Execution Time: 6.931954
Speedup: 0.221203

Thread 16 Execution Time: 2.564995
Speedup: 0.598677

Thread 32 Execution Time: 2.440341
Speedup: 0.627869

Thread 64 Execution Time: 2.076332
Speedup: 0.738767

Thread 128 Execution Time: 2.372405
Speedup: 0.646748

Thread 256 Execution Time: 0.376049
Speedup: 4.083871

Thread 512 Execution Time: 0.178567
Speedup: 8.597913

Thread 1024 Execution Time: 0.177730
Speedup: 8.629882

3.2 Experiment 2

3.2.1 Test with Resoulition 128 - Frame Count 1

Serial version Execution Time: 1.532970

Block 1 Execution Time : 0.170309
Speedup: 8.998

Block 10 Execution Time: 1.624410
Speedup: 0.944

Block 20 Execution Time: 2.164919
Speedup: 0.708

Block 40 Execution Time: 3.769440
Speedup: 0.406

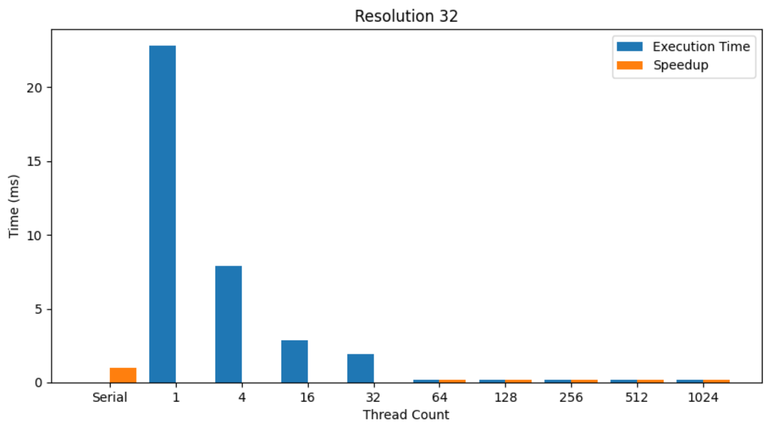
Block 80 Execution Time: 6.155917
Speedup: 0.249

Block 160 Execution Time: 10.925071
Speedup: 0.140

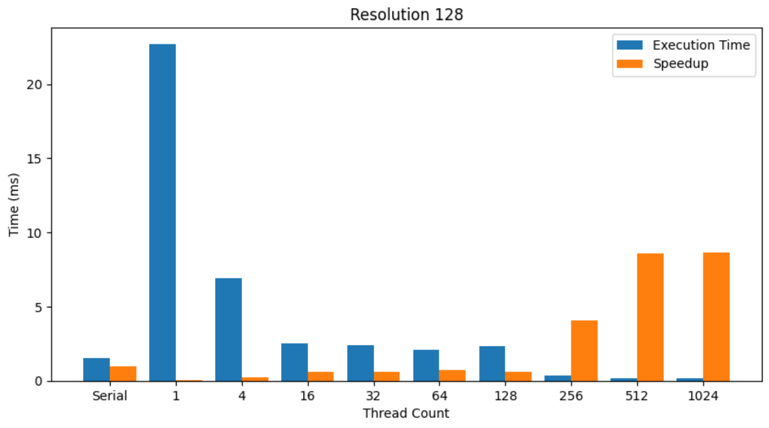
Results
Have the kernel execution overheads changed? How about memory copies?

Kernel Execution Overheads:
In Part 2, employing a separate kernel launch for each frame introduced overhead. Each kernel launch incurs setup and management time, which adds to the overall execution time. In Part 3, with the persistent kernel approach, I significantly reduce kernel execution overhead by using all available threads efficiently. This is because I now have just one kernel launch for all frames.

Memory Copies:
In Part 2, due to separate kernel launches, I copied the results of each frame's computation from the device to the host memory after each kernel execution. This resulted in multiple memory copies. In Part 3, I compute the results for all



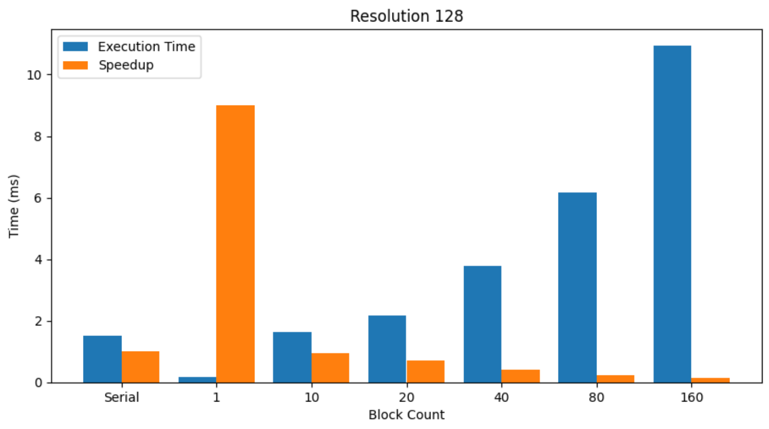
(a) Resolution 32 Execution Time Result



(a) Resolution 128 Execution Time Result

frames in a single kernel launch using the persistent kernel approach. While this reduces kernel execution overhead, it increases memory copy overhead because I need to copy all frame outputs from the device to host memory at once. This can lead to higher memory copy times compared to Part 2. As mentioned, the impact of these changes may be less noticeable with lower thread counts, as fewer threads result in reduced overhead. However, as the thread count increases, the difference between Part 2 and Part 3 becomes more pronounced. Part 3 generally exhibits shorter kernel execution times but potentially higher memory copy times due to copying all frames' outputs simultaneously.

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



(a) Resolution 128 Execution Time Result

and the number of frames is small? Explain

High Number of Frames and Small Problem Size: In Part 2, when dealing with a large number of frames and a small problem size, the kernel execution time is notably higher compared to Part 3's approach, especially as the frame count increases. This is because in Part 2, I create a new kernel with many threads for each frame, resulting in numerous kernel launches. In Part 3, I aim to mitigate this by moving the frame iteration loop inside the kernel. However, this optimization comes at the cost of increased memory copy time. This is because I now copy all frames from the device to the host in a single operation.

Large Problem Size and Small Number of Frames: In the case of a large problem size and a small number of frames, Part 3's approach may not offer a substantial performance advantage. This is because the primary benefit of moving the frame iteration loop inside the kernel is to handle a large number of frames efficiently. When the frame count is small, the optimization provided by Part 3 may not significantly impact performance, and the overhead of copying all frames from the device to the host becomes more noticeable.

4 Part 4: Double Buffer

In the fourth part of this assignment, I made a key modification to the main function to process two frames concurrently. This change allows me to optimize execution time and reduce memory usage. When the frame count (frameNum) is greater than 2, the program now calculates the result for one frame while simultaneously saving the output for the previous frame. This approach capitalizes on the overlapping of computations and memory operations, leading to improved execution time in specific scenarios. Additionally, this approach showcases enhanced memory usage efficiency. It's important to note that the core functionality of the code remains the same as in Part 2, with the primary alteration being the redesign of the main function to enable this concurrent processing of frames.

```
int offset = 0;
        start = high_resolution_clock::now();
        for (frame = 0; frame < frameNum; frame++)
        {
            //
    ////////////////////////////////////////////////////////
            //                    Launch the kernel
      //
            //
    ////////////////////////////////////////////////////////
            //          Copy the result back to host (async)
      //
            //
    ////////////////////////////////////////////////////////

            int idx = frame % 2;

            if (frame > 0) {
                cudaStreamWaitEvent(computeStream, transferDone, 0)
    ;
            }

            MarchCubeCUDA<<<numBlocks, numThreads / numBlocks, 0,
    computeStream>>>(numBlocks, numThreads, domain_d, cubeSize_d,
    twist, 0, meshVertices_dd[idx], meshNormals_dd[idx]);
            cudaEventRecord(computeDone, computeStream);

            if (frame > 0) {
                cudaStreamWaitEvent(transferStream, computeDone, 0)
    ;

                checkCudaErrors(cudaMemcpyAsync(meshVertices_h,
    meshVertices_dd[1 - idx], frameSize * sizeof(float3),
    cudaMemcpyDeviceToHost, transferStream));
                checkCudaErrors(cudaMemcpyAsync(meshNormals_h,
    meshNormals_dd[1 - idx], frameSize * sizeof(float3),
    cudaMemcpyDeviceToHost, transferStream));
                cudaEventRecord(transferDone, transferStream);
            }

            if (saveObj || correctTest) {
                cudaStreamWaitEvent(transferStream, computeDone, 0)
    ;

                checkCudaErrors(cudaMemcpy(meshVertices_h + offset,
     meshVertices_dd[idx], frameSize * sizeof(float3),
    cudaMemcpyDeviceToHost));
                checkCudaErrors(cudaMemcpy(meshNormals_h + offset,
    meshNormals_dd[idx], frameSize * sizeof(float3),
    cudaMemcpyDeviceToHost));

                if (saveObj) {
                    string filename = "part_4_link_f" + to_string(
    frame) + "_n" + to_string(cubesRes) + ".obj";
                    WriteObjFile(frameSize, meshVertices_h + offset
    , meshNormals_h + offset, filename);
                }
                if (correctTest) {
                    TestCorrectness(frameSize, meshVertices_h + offset, meshNormals_h + offset, frame);
                }
                checkCudaErrors(cudaMemset(meshVertices_dd[idx], 0, frameSize * sizeof(float3)));
                checkCudaErrors(cudaMemset(meshNormals_dd[idx], 0, frameSize * sizeof(float3)));
            }

            offset += frameSize;
            twist += 1.0 / float(frameNum) * maxTwist;
        }
        cudaDeviceSynchronize();

        if (frameNum > 0) {
            int idx = (frameNum - 1) % 2;
            checkCudaErrors(cudaMemcpy(meshVertices_h + (frameNum - 1) * frameSize, meshVertices_dd[idx], frameSize * sizeof(float3), cudaMemcpyDeviceToHost));
            checkCudaErrors(cudaMemcpy(meshNormals_h + (frameNum - 1) * frameSize, meshNormals_dd[idx], frameSize * sizeof(float3), cudaMemcpyDeviceToHost));
        }
    }
```

4.1 Experiment 1

4.1.1 Test with Resoulition 32 - Frame Count 64

Serial version Execution Time: 0.030580

Thread 1 Execution Time : 10.164158
Speedup: 0.00301

Thread 4 Execution Time: 4.586808
Speedup: 0.00666

Thread 16 Execution Time: 2.396109
Speedup: 0.01276

Thread 32 Execution Time: 1.806424
Speedup: 0.01692

Thread 64 Execution Time: 0.308512
Speedup: 0.09922

Thread 128 Execution Time: 0.307505
Speedup: 0.09945

Thread 256 Execution Time: 0.317915
Speedup: 0.09604

Thread 512 Execution Time: 0.351737
Speedup: 0.08689

Thread 1024 Execution Time: 0.072344
Speedup: 0.42291

4.1.2 Test with Resoulition 128 - Frame Count 1

Serial version Execution Time: 1.532970

Thread 1 Execution Time : 8.136598
Speedup: 0.18829

Thread 4 Execution Time: 2.504090
Speedup: 0.61251

Thread 16 Execution Time: 1.016717
Speedup: 1.50678

Thread 32 Execution Time: 0.789567
Speedup: 1.94232

Thread 64 Execution Time: 0.522793
Speedup: 2.93268

Thread 128 Execution Time: 0.437275
Speedup: 3.51194

Thread 256 Execution Time: 0.132618
Speedup: 11.56

Thread 512 Execution Time: 0.140985
Speedup: 10.87

Thread 1024 Execution Time: 0.077735
Speedup: 19.71

4.2 Experiment 2

4.2.1 Test with Resoulition 128 - Frame Count 1

Serial version Execution Time: 1.532970

Block 1 Execution Time : 0.166419
Speedup: 9.21

Block 10 Execution Time: 0.863692
Speedup: 1.77

Block 20 Execution Time: 1.117829
Speedup: 1.37

Block 40 Execution Time: 1.662955

Speedup: 0.92

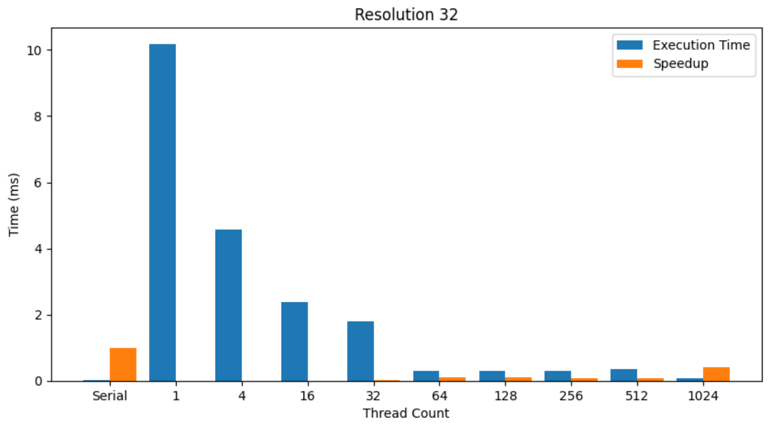
Block 80 Execution Time: 2.733266

Speedup: 0.56

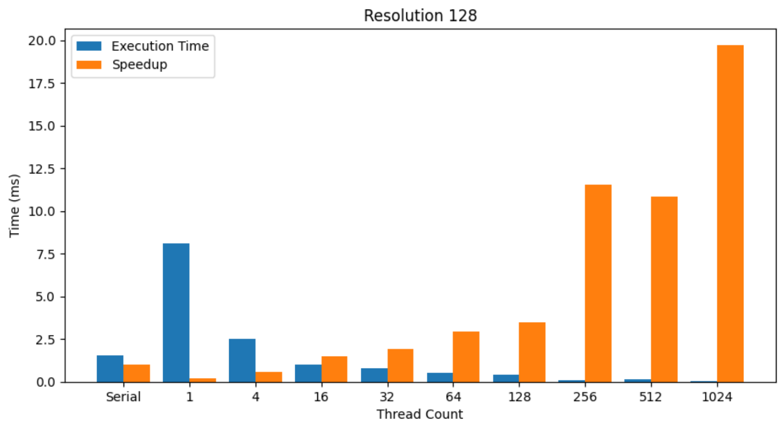
Block 160 Execution Time: 4.822452

Speedup: 0.32

Results



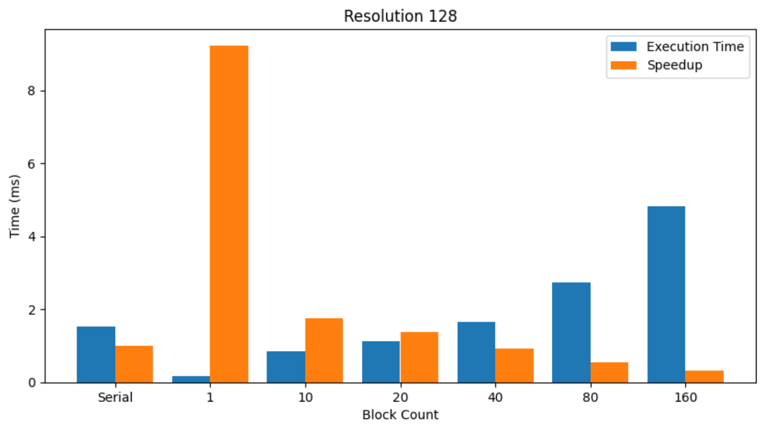
(a) Resolution 32 Execution Time Result



(a) Resolution 128 Execution Time Result

What is the obtained speedup compared to Part 2?

The obtained results were promising, particularly in scenarios where the



(a) Resolution 128 Execution Time Result

resolution was 128 or higher and the thread count exceeded 128. In these cases, I observed that the memcpy (memory copy) and kernel execution times in Part 2 became comparable. This phenomenon highlighted the true potential of this approach, as I achieved significantly reduced execution times compared to Part 2.
The speedup obtained compared to Part 2 was substantial in such scenarios. While the exact speedup may vary depending on the specific parameters, it's evident that this approach allows for considerable performance improvement. The ability to overlap memory copies and computations plays a crucial role in achieving this speedup, demonstrating the effectiveness of this optimization technique.
What is the observed memory usage compared to previous approaches?

The observed memory usage in each task varies based on the specific requirements and optimizations applied. In Task 2, where I calculated one frame at a time, memory was allocated for a single frame's length, as it aligned with the task's approach. In Task 3, where I aimed to process the entire frame, memory was allocated for the full frame length to facilitate this approach effectively. Finally, in Task 4, I introduced asynchronous two-frame operations, requiring memory allocation for two frames to enable concurrent processing. This allocation was optimized to match the specific needs of this task. Therefore, the observed memory usage in each task was tailored to the respective task's goals and strategies, resulting in varying memory allocation approaches.

5 Formulas Used

a. *Speedup*

$$Speedup = \frac{\mathrm{T1}}{\mathrm{Tp}}$$