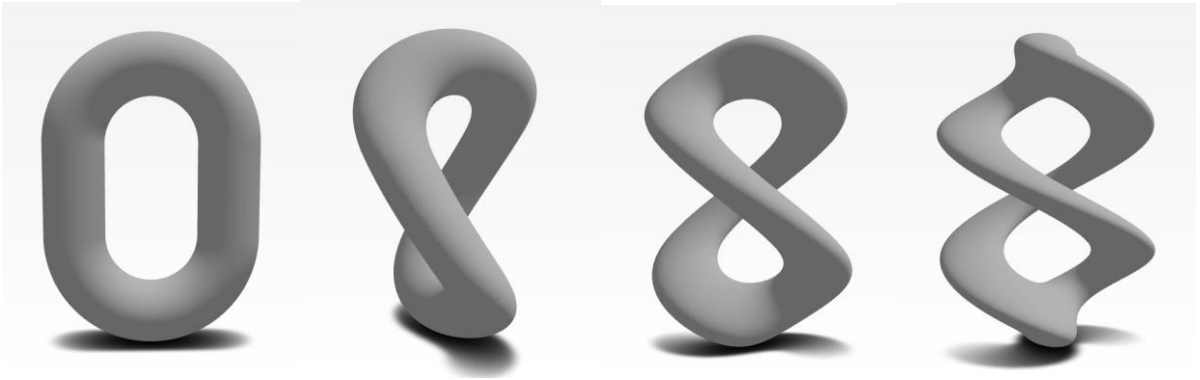
COMP429 Parallel Programming Project-2 Report

Emir Şahin 72414

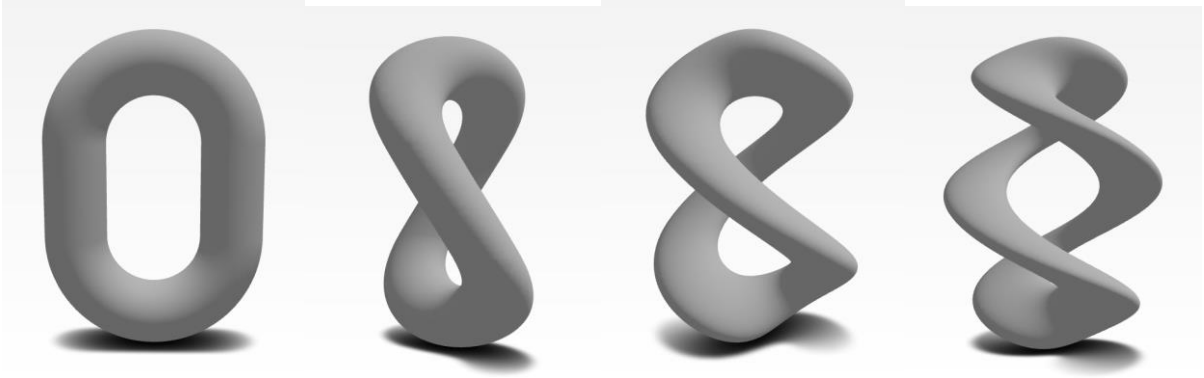
Introduction

I'd like to start by providing screenshots of the .obj files viewed through the online viewer. The .obj files were generated with the following parameters: n = 128 (Halved as it was advised in the mail), f = 4, b = 1, t = 1024. I found these values to be comfortable for generating the images after experiencing many values through testing.

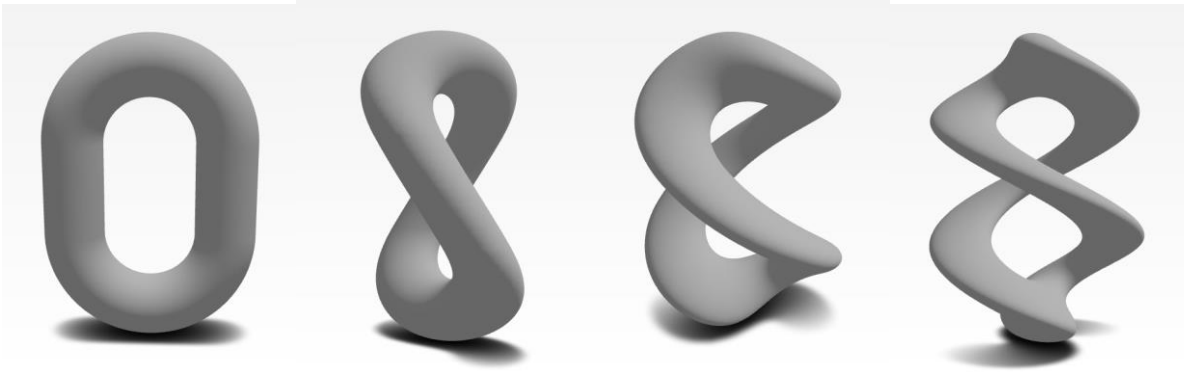
CPU:



Discrete Kernel:



Persistent Kernel:



Part – I

I ran the code on the CPU through the serial baseline, and on the GPU with a single thread. The GPU was slower than the CPU with a factor of ≈ 5. The data transfer was not significant at all, which made me attribute this observation to various advantages of a CPU, a larger cache. I believe I'd need to know more about the cluster to make further inferences. I ran the program with n = 64, f = 1, b = 1 and t = 1.

Part – II

I haven't really encountered any problems working with high thread counts. I conducted the first experiment with n = 128, during the first experiment, I observed the fastest result with 512 threads at "46.803 mln_cubes/sec" (with a single block), there is a slight drop-off with 1024 threads at "46.760 mln_cubes/sec". I decided to stick with n = 256 for the second experiment, with 512 threads and a single block the result was "52.362 mln_cubes/sec", with 10 blocks it was "511.943 mln_cubes/sec" reaching the peak at 80 blocks with "1194.455 mln_cubes/sec". I observed that 80 blocks and 512 threads resulted in the best performance. I found one setting where the kernel overhead and the data transfer speed were similar at "0.0325" and "0.0306" respectively with n = 64, f = 4, b = 40, t = 256. I'm sure there are many other configurations to obtain results where the kernel overhead and the data transfer speed are similar, this configuration is one of the first ones I've discovered experimenting. When I decreased the frame number from 4 to 2, the data transfer speed almost doubled from "0.0307" to "0.0154". I then kept the frame number constant at 2 and decreased n from 128 to 64 to 32. with each decrease, the data transfer speed was multiplied by around 8, the time going from "0.12242" to "0.01535" to "0.00198". There is a direct inverse correlation between the problem size and the memcopy overhead.

Although they're impossible to read I'll be attaching my notes where I decided on the threadIdx to (xi, yi, zi) conversion method. After deciding on the (xi, yi, zi) values, the rest was almost the same as the CPU implementation. I removed the offsets for the meshVertices_h and meshNormals_h values, every result copied back to the host is –tested and converted into an .obj file- overwritten.

Part – III

Going in the reverse direction of my last experiment in part - II, I started with f = 2 and n = 32 and started moving upwards. At all the values n = 32, 64, 128 both the kernel overhead and the memcopy overhead were very similar to the values in part – II. I then kept the n value stable at 64 and slowly increased the f value from 2 to 4 to 8. Each time, the memcopy overhead almost exactly doubled and remained comparable to the results when the same parameters were used in part – II. I then increased the n value to 128 and ran the program with f = 2 and f = 4, the memory overhead again almost exactly doubled. I set f = 8 and n = 8, when I ran the same configuration on part – II and part – III, the memcopy overhead for part – III was almost half of part – II. I set n = 64 and f = 2 and ran both parts, the memcopy overhead results were comparable; I then set n = 128 and f = 2 and ran both parts again, the memcopy overhead results were again comparable.

The implementation of this part is similar to Part – II but I had to add an extra dimension to account for the "current frame" of the thread, I'll be attaching my notes for this part as well.

A notable difference is that the copied memory is copied as a whole from the device to the host.

Part –IV

For this part I read “Professional CUDA C Programming; John Cheng, Max Grossman, Ty McKercher” to get help. Under chapter 6 “Overlapping Kernel Execution and Data Transfer”. Both the speedup and the memory usage seem comparable between this part and part II. I am assuming this is because I am not picking the right job-size, thread, block values although I’ve tried quite a few of them. I initialized the device vertices and normals variables as arrays of size 2. Iterating through the frames the kernel and the data transfer switch between using the 0^{th} index of the device vertices and normals variables and the 1^{st} index of the device vertices and normals variables. They are grouped into 2 streams, as was appropriate for this setting.

Experiments

I’ve run the experiments; I’ll be submitting my results along with the report and the source code. Here are some of the experiments’ results plotted:

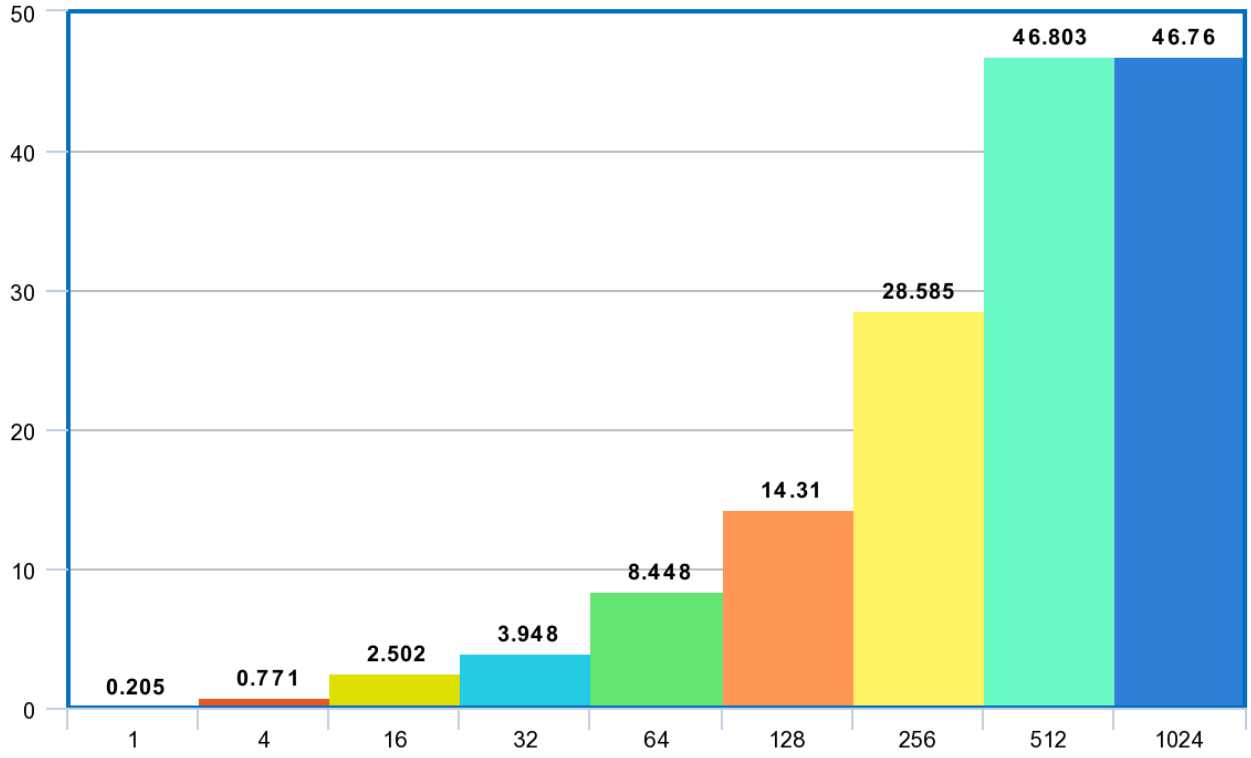


Figure – 1: mln_cubes/sec for varying thread counts.

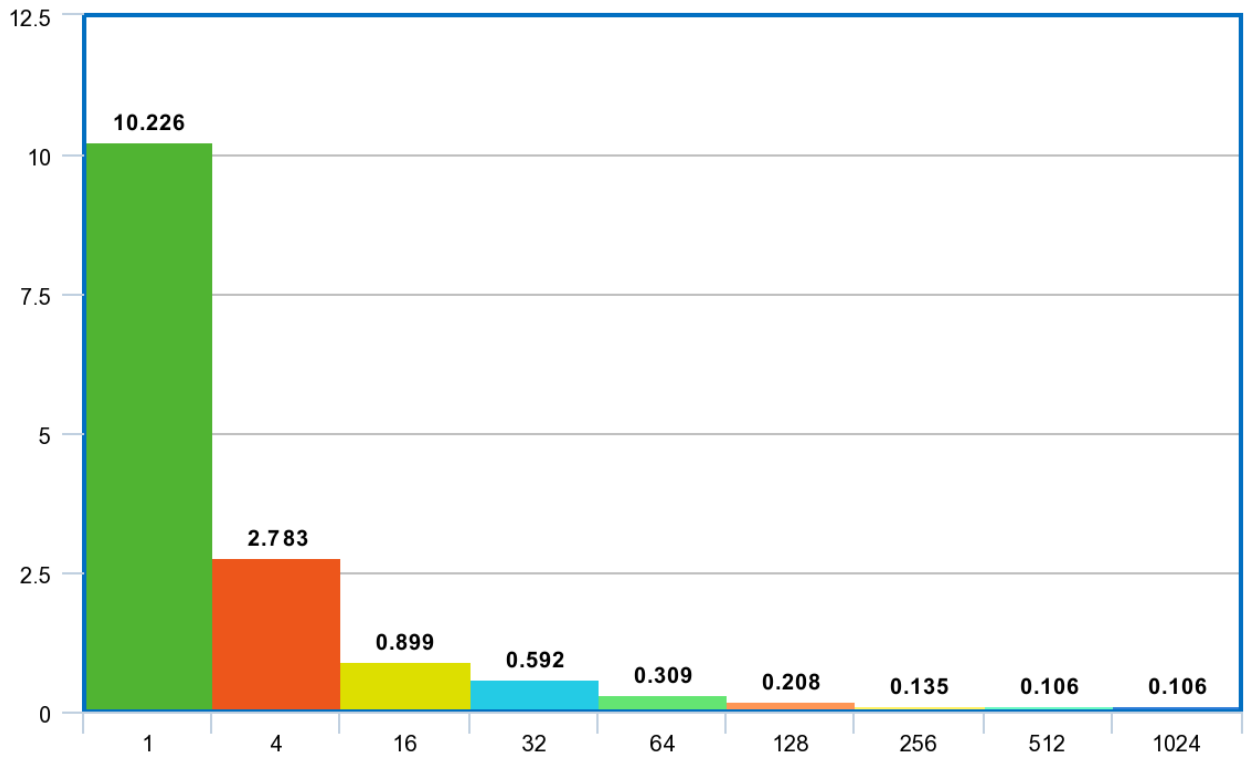


Figure – 2: Execution times for varying thread counts.

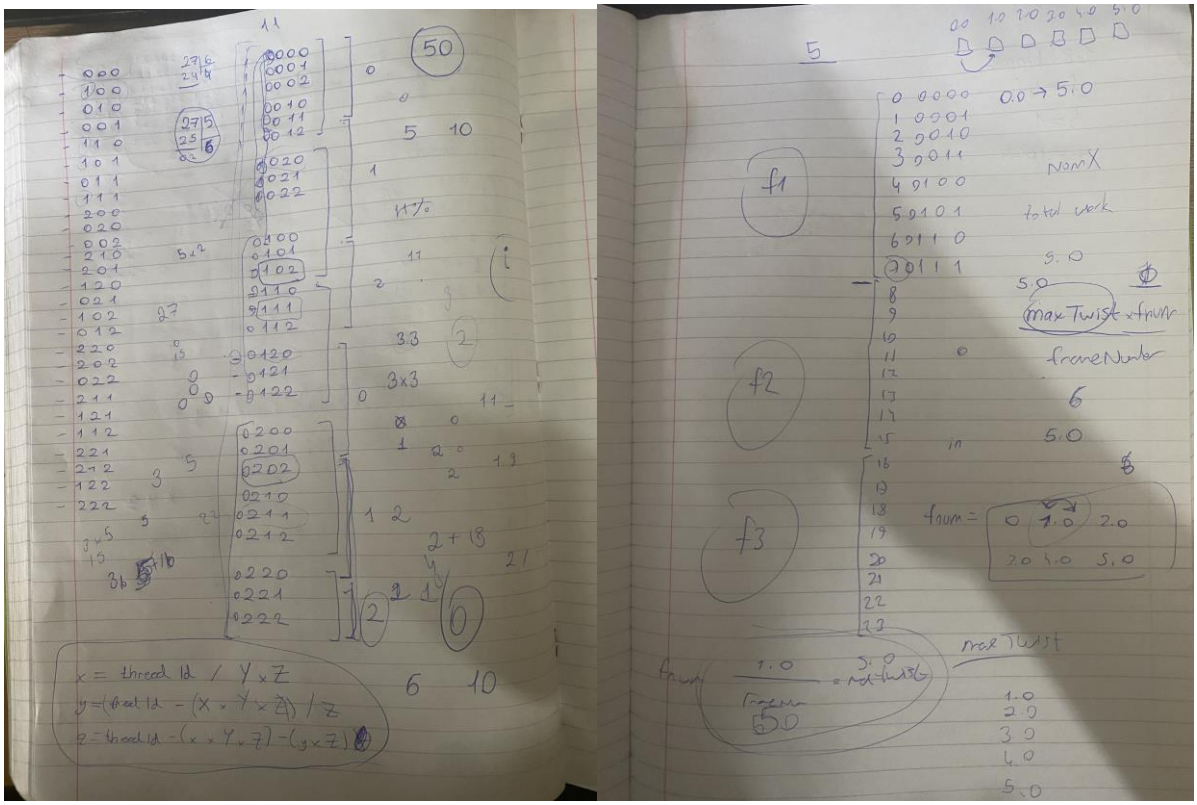


Figure – 3/4: My notes on the 3D index calculation through threadID (left) and adding another dimension to account for the frame number (right).

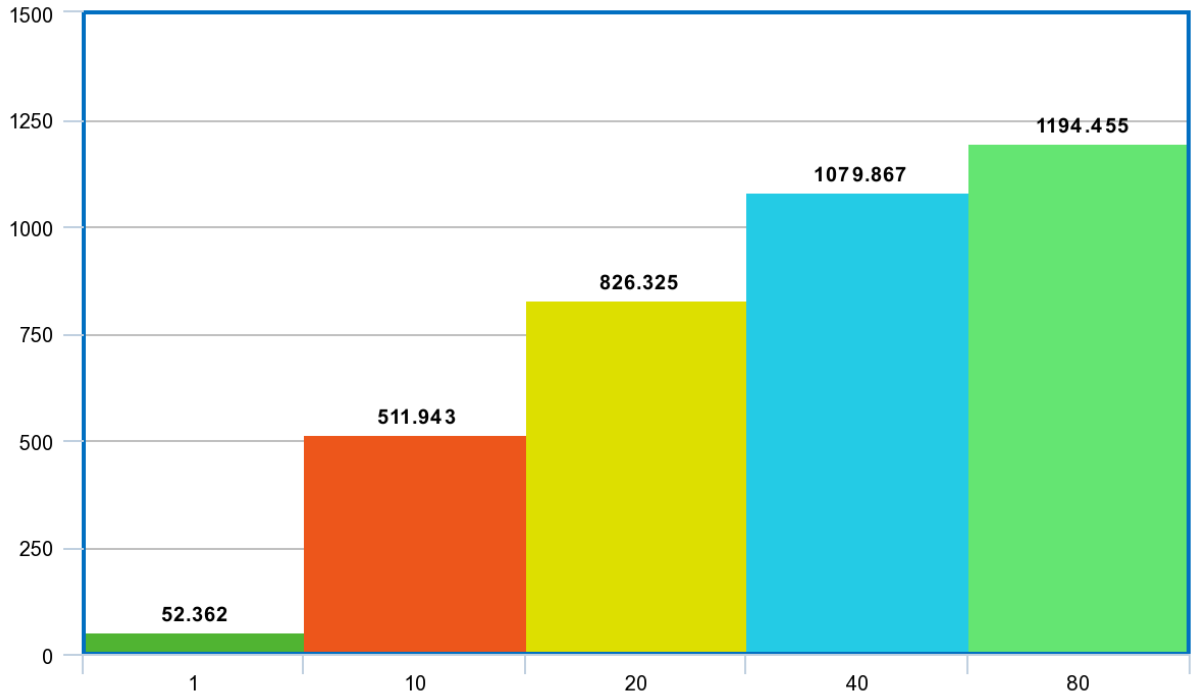


Figure – 5: mln_cubes/sec for varying block sizes.

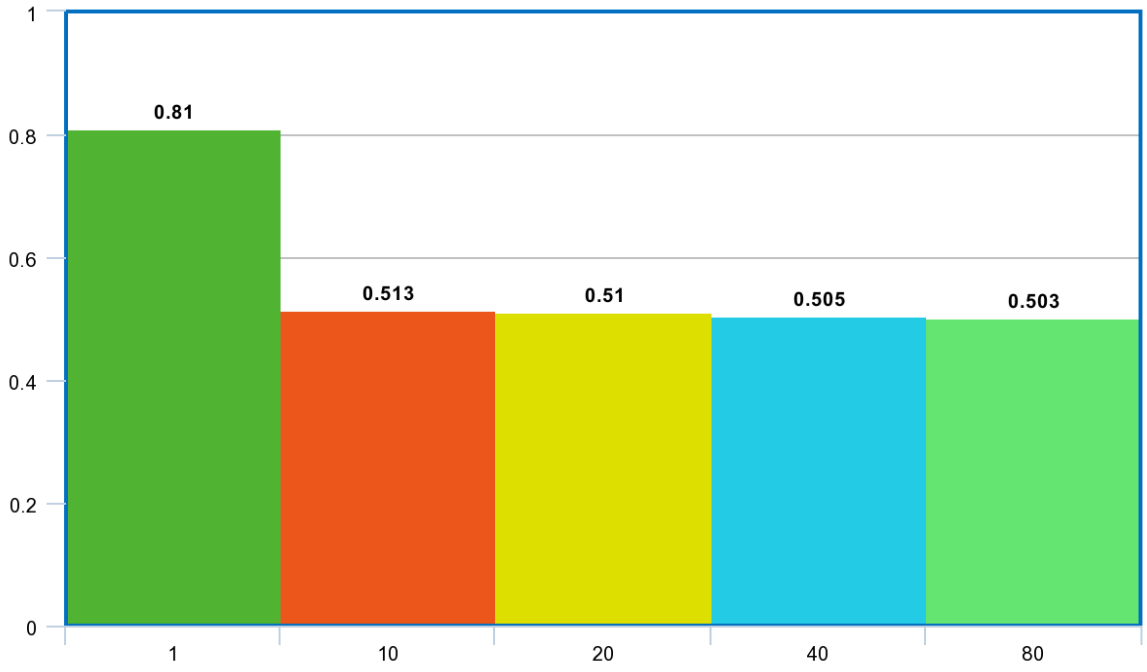


Figure – 6: Execution times for varying block sizes.