TDT4200 - Task 4

Daniel Romanich

November 2018

1 More OpenMP

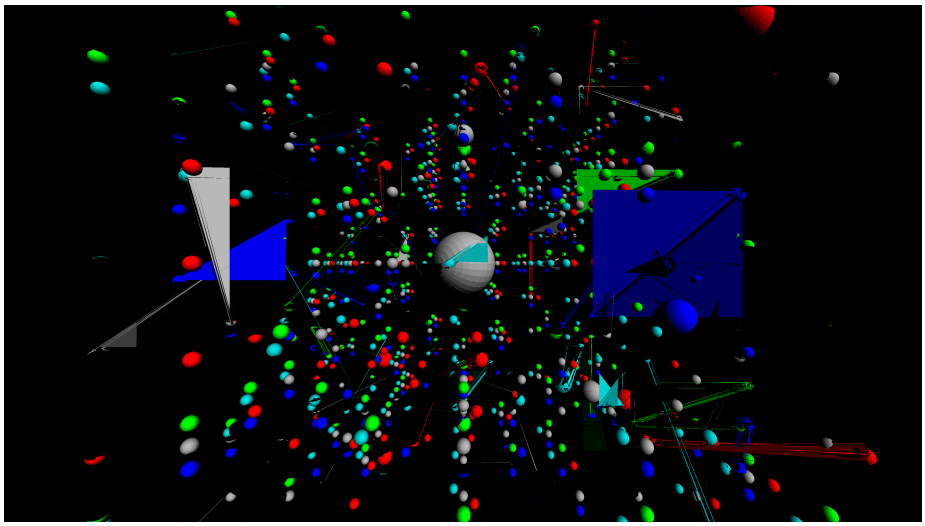
1.1 CPU-Rendering

In order to get a run-time around 10 seconds I used a resolution of 7680x4320 with a depth level of 3. The table under shows the time spent running the program 5 times:

Run-times in milliseconds
10653ms
10767ms
10714ms
10698ms
10796ms

1.2 Race-conditions

After some digging I found that the only race-condition was the transformedMesh. To solve this I simply removed the reference pointer (&) to the transformedMesh element (would be the same as cloning it). With the race condition the picture in 8K resolution with d=3 would end up looking something like this (different every time):



1.3 Static, Dynamic & Guided

Before doing this task I measured how long each iteration of the loop took to compute. The result was very spread; some iterations were very fast, while others were a lot slower. With this knowledge my assumption is that dynamic with low chunk size is the most efficient. This is because the time spent rendering the different items can vary A LOT, meaning that larger chunks can be unlucky and take a long time. Therefore, I would assume Dynamic with low chunk size is fastest, guided and dynamic with larger chunk size depends on the size of the chunks; while static should be the slowest.

Secheduling method	Runtime	Speedup
Static	2450ms avg	4.4x
Dynamic (Chunks of 1)	2170ms avg	4.9x
Dynamic (Chunks of 2)	2175ms avg	4.9x
Dynamic (Chunks of 5)	2330ms avg	4.6x
Guided	2340ms avg	4.6x

Looking at the results above, my assumptions were relatively correct when compared to reality. As stated, I expected dynamic with lower chunks to be faster, and this is because some of the items take a lot longer to render than others.

2 Getting started

2.1 Print device name

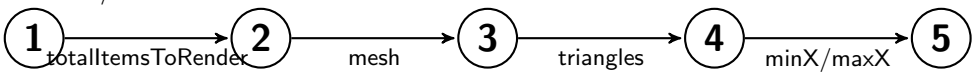
To print the device name I used the code seen below:

```
cudaDeviceProp props;
checkCudaErrors(cudaGetDeviceProperties(&props, 0));
std::cout << "Name: " << props.name << std::endl;
```

The result printed was: "Name: GeForce GTX 1070".

3 Some planning

I found this exercise a bit weird, as the tree would only be a "callgraph" where each of the for loops would be the child of the previous. Anyways, the graph (tree) below illustrates the loop nesting. The different nodes are:
1: TotalItemsToRender
2: Mesh
3: Triangles
4: minX/maxX
5: minY/maxY



The expense of the loops below is based on the execution time, as well as how the iterations scale with different variables. I have also counted in how unstable their run-time is, because this might have a big impact on the parallelization done in later steps.

TotalItemsToRender: *long*. This loop scales exponentially with the depth (26^d). The problem with this loop is that its execution time per iteration is really unstable.

Mesh: *short*. This loop has a constant size (pretty small, 5 for the spheres), and it is just as unstable as the itemRender loop.

Triangles: *medium*. This loop has a constant size (960 (2880 / 3) for the spheres), and it is not perfectly stable, but is more stable than any of the previous loops.

maxX and maxY: *medium*. This loop scales with the size of the image and the size of the current object to be rendered.

3.1 Dividing strategy

By looking at the execution time of the different for loops, it is clear that it would be unwise to divide the item for loop into different threads; simply because the execution time of the different threads can vary widely. Using the information I gathered in the previous task; it seems like parallelizing the triangle for loop might be a very good idea, as its execution time does not vary nearly as much as the mesh and the item for loop. However, the easiest solution would still be to parallelize it by dividing the items to each thread. My plan will be to try both of the techniques and see how they work out.

4 Setting things up

I did everything explained in the task, except memset for the depthbuffer. Since the memset would set values on each byte instead of every 4 (integer) I couldn't figure out how to write 2^{24} to the buffer (2^{32} did not seem to work), so I just did a memCpy from the cpu depthBuffer to the gpuDepthbuffer.

5 You may fire when ready!

Initially I tried to follow my game-plan of parallelizing the triangle loop, however, it ended up being super-slow, probably because of memory issues. Therefore I ended up running kernels equal to the number of items that needed to be rendered (had to split them up into more blocks for depths greater than 3). This gave a significant speedup from the single threaded ones, but I pretty sure the parallelizing of triangles would be faster than my solution.

6 Solving issues and evaluating the result

6.1 Race condition

The race-condition I found was tied to the depthBuffer. I solved it by suing the atomicMin to evaluate if the depth was lower than in the buffer, then check if the new buffer value matched the depth. By doing this, there is no way we have a race condition between two threads that cause the wrong thread to write the pixel.

6.2 Evaluating the result

As mentioned above, my solution is not the most efficient. This can be seen when comparing the result with the single and multi-threaded CPU solutions. Running the same settings as in previous exercises gave me an average run-time of 3100ms. This means we get a speedup of about 3.5x versus the single-threaded solution, and a speedup (slowdown) of about 0.7x versus the multi-threaded solution.

If we also count the allocation-time we get an average run-time of 3400ms. This is a speedup of about 3.1x versus single-threaded and a speedup (slowdown) of about 0.6x versus the multi-threaded approach.