TDT4200 - Task 4

Daniel Romanich

November 2018

1 Basic Profiling

a) Percentage of time executing instructions

By looking at the graph that shows instruction execution counts I can see that about 85% of the time is spent being inactive. Therefore the warps are executing instructions about 15% of the total time.

b) Memory dependency

By looking at the cake diagram for stall reasons the memory dependency accounts for 27.85% of the stalling.

c) Fewest warps

I found two lines that have the least number of threads executing them;
1. The second for loop in the bounding box iterator.
2. The first calculation in the isPointInTriangle function.

d) Worst memory access pattern

The two worst memory access patterns is the workItem access and the frameBuffer moditications (274-278). Here the workItem access is the worst with 4x as many accesses as necessary.

e) Main limiting factor

The current limiting factor is the number of registers used for each thread (50). Since the GPU (GTX 1070) can provide 65536 registers, and a block is using 51200 registers, only one block can be executed at a time (32 warps).

f) Occupancy of the kernel

The achieved occupancy of the kernel was 26.8%.

g) Average execution time

From running the kernel 5 times the average execution time I got was: 2170ms. The parameters I used was depth=4, width=7680 and height=4320.

2 Low-hanging fruits

a) Occupancy

Occupancy is the ratio of active warps on an SM to the maximum number of active warps supported by the SM.

b) Number of warps

There are several limiting factors to how many warps that can be active in an SM. The first limit is obviously the number of blocks per SM and warps per block. However, this is usually pretty tweekable. Another limiting factor is the number of registers per SM. If threads use too many registers, there will not be enough registers for all warps, and thus it will limit the amount that can run at the same time. Shared memory is also a factor. In the same ways as the registers, there is a finite amount of shard memory, and the amount of shared memory used by threads can therefore limit the number of active warps.

c) Latency hiding

Latency hiding is the notion of swapping warps when they are waiting for data from memory. Since the GPU is very fast at swapping warps it can hide the "memory" latency by running swapping warps that is waiting for memory with warps that is ready. Thus, the latency hiding also increases the throughput as less time is spent waiting for memory.

d) Improved occupancy by optimizing memory access

I used the technique from the lecture to optimize the GPUItem memory access. Since accessing an array of structs can be very taxing due to cache lines, it is suggested to use a struct of arrays. I ended up dividing the scale and distanceOffset into four different arrays. After doing this nsight stopped complaining about inefficient memory access.

The occupancy was increased by 0.1%, from 26.8% up to 26.9%, so clearly not a big increase.

e) Improved occupancy by optimizing launch parameters

In this I decided to change the threadsPerWorkQueueBlock (workqueue in the table) and threadsPerVertexBlock (vertex in the table) to see the result. The initial values are 32 for both. The initial occupancy was as stated above 26.8%. Below is a diagram of parameters and their achieved occupancy.

Workqueue	Vertex	Occupancy
32	32	26.9%
16	32	29.8%
64	16	19.7%
32	16	29.2%
16	16	36.9%
8	8	42%
10	10	42.8%

Here we can see that smaller (down to a certain point) dimensions actually increase the occupancy by quite a bit. However, with the current code some of these parameters can be dangerous (as I later found out) because they will not fill up a warp with 32 threads. Therefore 8x8 would be the best solution here, as it yields a block size that is a multiple of 32.

f) Execution time

The execution time after the optimizations above is on avg: 1470ms. This yields a speedup of 1.47x (46%).

3 Engaging the warp drive

a) Individual threads

Ideally we would like all threads to execute for approximately as long. This is because every thread in a warp has to execute the same instruction at the same time. If one thread is executing while the others are done (or we have divergence) we are doing very little work because 31 other threads are idle.

b) Rendering large triangles on the whole warp

I solved this by simply checking if the square it is iterating over covers an area larger than the threshold. If that is the case, vote true in the ballot. After the vote I iterate over every thread and check if it needs to be rendered by everyone. I then send the required information to the other threads so that they can only render a small portion of the pixels. Once a box is done, the selected thread has its bit set to 0 (so that we can get the next thread with the __ffs() function). After all large triangles has been rendered, the ones who still has a small triangle to render does so.

i) Occupancy after optimizing rendering

The overall occupancy actually went down when comparing it to the previous solution. However, by comparing it to the theoretical limit it is almost exactly the same (84% of the theoretical occupancy was achieved).

j) Speedup after optimizing

The time spent rendering after the optimization was a lot lower. The time I got by optimizing the parameters was 1470ms, while with this optimization I achieved an average of 790ms. This is a speedup of 2.75x compared to the initial speed, and 1.86x compared to the parameter optimization. The reason to why I think I got a pretty large speedup from this optimization is due to the image resolution. The parameters resembles 8K resolution, meaning that the large triangles will be a lot larger than the small triangles, and therefore the load balancing will be a lot more effective. I tested this theory by running d=5 instead of 4 in full HD resolution, and the execution time was actually slower than in the handout solution (However, this is also due to the kernel parameters, as they were optimized for the resolution).