1DL550 Low-level parallel programming

Assignment 4

CUDA

Aljaz Kovac, Simon Jaklovsky, Jakob Nordgren (Group 14)

Statement of participation

All group members participated equally on the assignment.

Questions

Describe the memory access patterns for each of the three heatmap creation steps. How well does the GPU handle these access patterns?

Counting Agents

We access the i'th element of desiredX or desiredY, where i = blockIdx.x * blockDim.x + threadIdx.x .This is coalesced access. Access of the heatmap elements is dependent on the values of the desiredX and desiredY arrays, so there is really no pattern there.

Scaling

Although the data is in a flat array, it is indexed as a 2d array, for example:

int value = hm[y*SIZE + x];

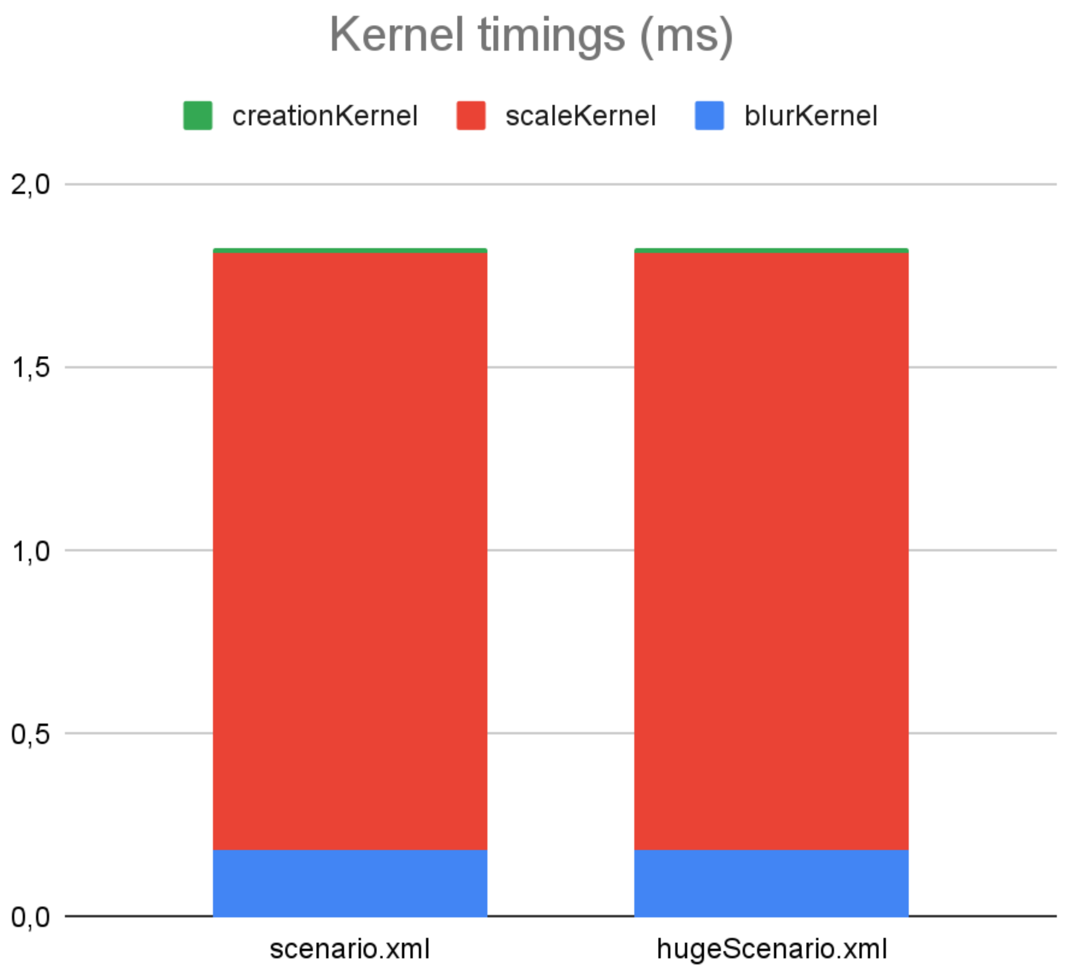
Coalescing is achieved by setting x = threadIdx.x . Then each warp will access nearby locations in memory. However, when filling the scaled heatmap array, memory access is non-coalesced since we have to access non-adjacent elements.

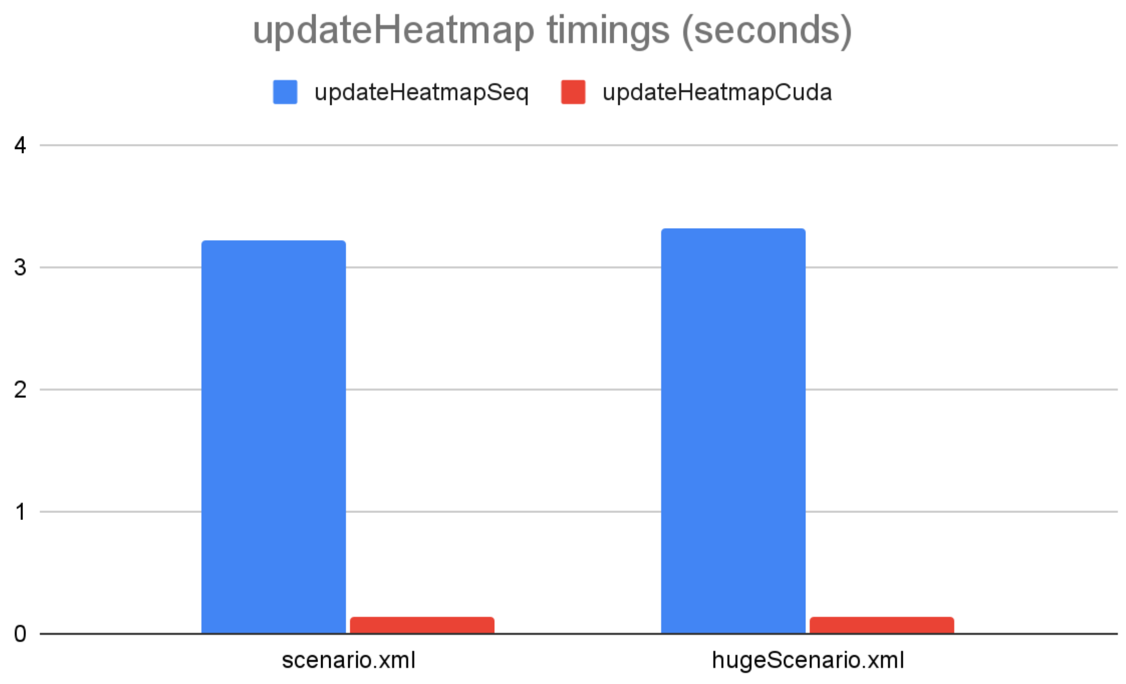
Blur

The same principle as for the scaling step, where we have to access non-adjacent elements when calculating the sum. As is evident in the graph below, this leads to poor performance.

Plot and compare the performance of each of the heatmap steps. Discuss and explain the execution times.

The vast majority of execution time is spent in the scaleKernel. We believe this is due to the awkward memory access patterns described in question 1. As expected, the creationKernel, which has coalesced memory access, is the fastest of the three by an order of magnitude. Execution times were measured by recording each individual kernel invocation in CUDA.


Kernel timings (ms)
creationKernel
scaleKernel
blurKernel
2,0
1,5
1,0
0,5
0,0
scenario.xml
hugeScenario.xml


updateHeatmap timings (seconds)
updateHeatmapSeq
updateHeatmapCuda
4
3
2
1
0
scenario.xml
hugeScenario.xml

What speedup do you obtain compared to the sequential CPU version? Given that you use N threads for the kernel, explain why you do not get N times speedup.

Roughly a 22 times speedup is achieved. Our implementation unfortunately needs to allocate and free memory on the GPU at each tick, which is not good for performance. In assignment 2 we had the same issue. Using the Nvidia profiler we saw that most of the time was spent allocating and freeing memory. Optimally one would allocate the memory at the start of the program and free it at the end.

How much data does your implementation copy to shared memory?

We decided to use a static shared array of the following size: 16 * 16 * CELLSIZE * CELLSIZE, which equals 256 (number of threads) * 25 (size of a cell) = 6400 elements.

Machine specification

```
interactive -A uppmax2021-2-28 -M snowy -p core -n 1 -c 8 -t 1:00:01 --gres=gpu:1
```

Run the program

To observe both collision detection and CUDA for the heatmap calculations, run the demo with the OpenMP implementation:

```
$ demo/demo --implementation OMP scenario.xml
```

```
$ demo/demo --implementation OMP hugeScenario.xml
```