Low-level Programming: Assignment 4

Group 13 - 12.03.2025

Songtao Hu, Sivakorn Lerttripinyo, Alexander Verdieck

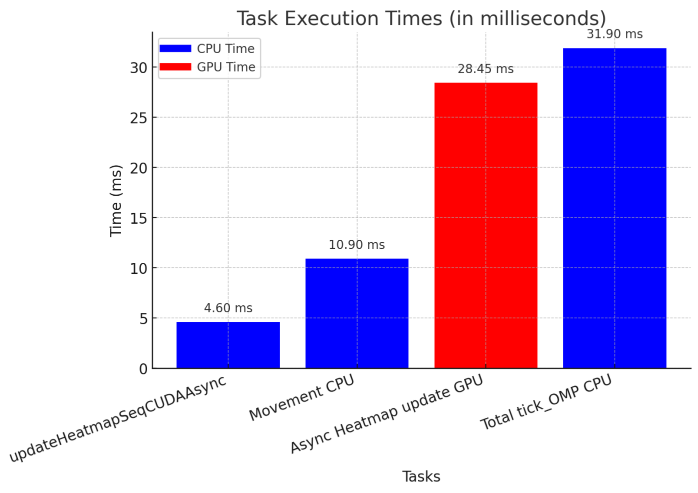
Proving that heatmap processing on GPU runs asynchronously to collision detection on CPU

At the starting point of a tick, CPU and GPU timers for timing overall CPU and GPU time are started.

Then, the host calls a function to send heatmap processing work to the GPU. Another CPU timer to capture only the time for running this function is created, and it captures this timing result as “updateHeatmapSeqCUDAAsync”.

Then, another CPU timer is created and used to capture only the time for doing collision detection. The result is captured as “Movement CPU time”. After this, the program needs to wait so that both GPU and CPU finish their work because the results are required for the next tick. Please note that they just need to wait for each other, but they still work asynchronously. It is possible that one of them finishes its work before.

After waiting, the GPU and CPU timers for measuring overall time are then stopped and captured as “Async Heatmap update GPU” and “Total tick_OMP CPU”, respectively. The result can be plotted as the following.



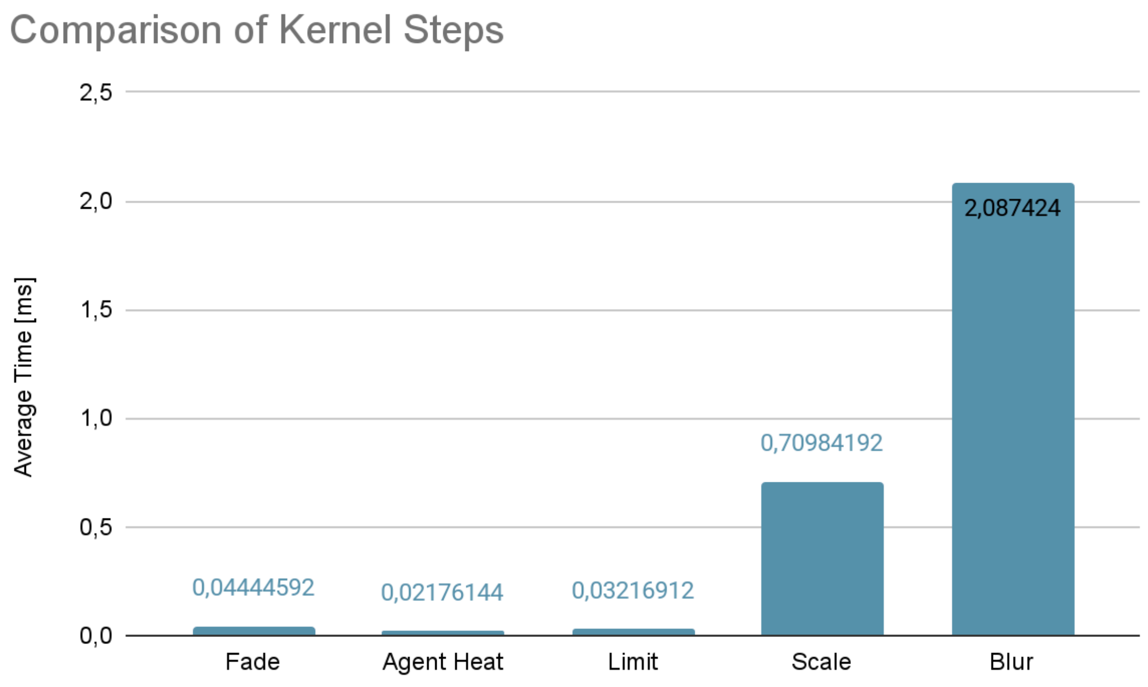
In the host, the function to send the heatmap processing to the GPU is called before performing the collision detection on the CPU. If the CPU waits for the GPU to finish its heatmap processing work, the summation of updateHeatmapSeqCUDAAsync and Movement CPU time will equal Total tick_OMP CPU. However, it is not the case shown in the plot. The summation is significantly less than total CPU time. This result means after the function to send the heatmap work to the GPU is executed, the host does not wait for the GPU to finish its work and continue to do the collision detection. There is only one synchronization in the end to make sure that both work complete before starting the next tick. You can see from the graph that Async Heatmap update GPU is a little less than Total tick_OMP CPU. In our program, the collision detection is faster than the heatmap processing. After the latter finishes, there is only a little additional time before the current tick finishes. This also shows that the collision detection has already been done while the heatmap processing is not finished yet.

A. Describe the memory access patterns for each of the three heatmap creation steps. Is the memory access pattern within a warp coalesced or random? (Think about the access pattern during fading and blur, does blur step update all or just some pixels?)

In a heatmap creation step, there are three substeps including fading the value by 20 percent, adding an intensity to the particular pixel, and limiting the value to 255. The first and third substep access the memory with a warp coalesced because pixels are accessed in a row-major order. However, in the second substep (adding the intensity), the heatmap's pixel is accessed randomly depending on the value of the desired position's arrays.

In a heatmap scaling and applying the blur filter, pixels' access is coalesced because all accesses depend on grid and block ID, which is managed by a warp and configured to run in a row-major order manner.

B. Plot and compare the performance of each of the heatmap steps. Discuss and explain the execution times. (Which type of plot is suitable?)



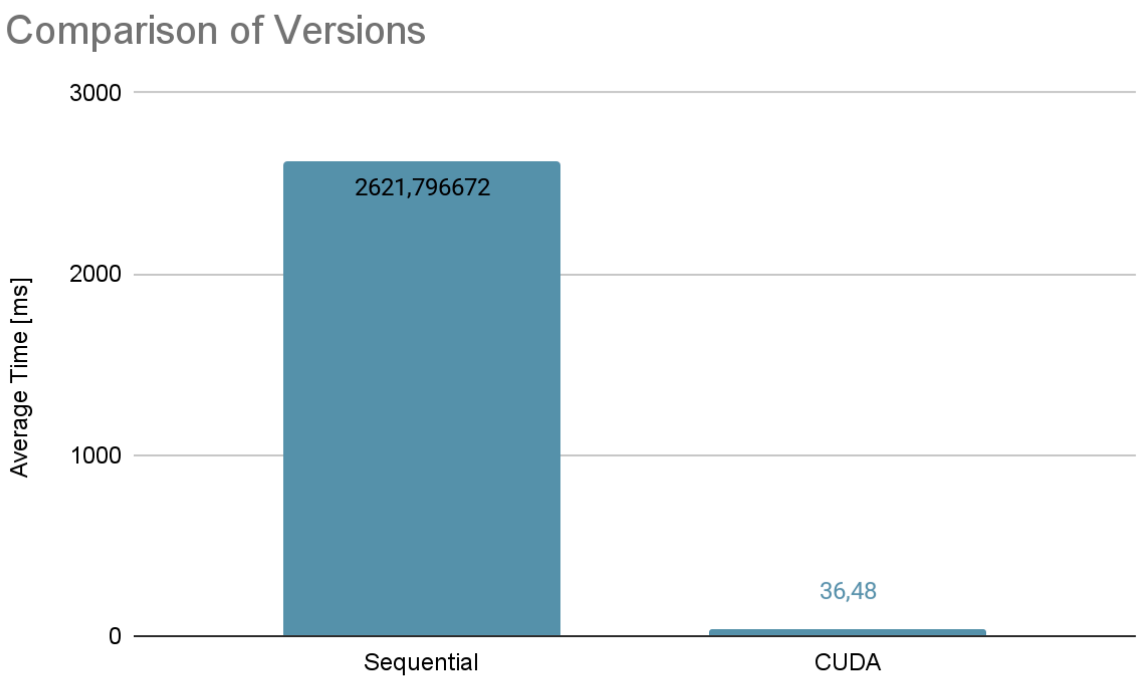
We have 5 different kernels performing each one step. The first only applies the fade on each pixel, the second applies the intensity addition for a desired position of each agent on the heatmap, the third limits each pixel again to the max value of 255. Before applying the blur filter, the heatmap is scaled by a factor of 5 in the fourth kernel. Finally, the blur is applied in the fifth kernel.

The result from the plot is reasonable. The first three kernels are fast because they only apply a few instructions, such as one multiplication and a single if-statement to limit the value. On the contrary, the most complex step, applying the blur filter, takes the longest time. It not only has the complex logic, but it also has to copy data from and to the shared memory.

Each kernel uses a significant number of threads since each thread is assigned to be responsible for either one pixel or one agent depending on the kernel functionality.

C. What speed up 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 speed up?

The comparison is done between the timing when running a heatmap processing and a collision detection asynchronously and that when running synchronously only on CPU.



We get a speedup around 70. This is by far not as much as we used threads; however, we also have to take the overhead from CUDA into account. Before running the code, all of the data has to be transferred to the GPU. Once the CUDA calculations are done, the data has to be copied back to the host. All of the kernels can also only be run in order, so that we have an implied barrier after each kernel, similar to the implied barrier in OMP at the end of each parallel region. The blur kernel also uses thread synchronisation in the kernel itself.

D. How much data does your implementation copy to shared memory?

A shared memory is used in the blur filter kernel. The amount of data copied to a shared memory of each block equals the value of the block's width plus 4 multiplied by the block's length plus 4. The reason for adding 4 to each side's length is because the blur filter in the sequential version starts from (2,2). It means that there should be the padding size of 2 for each side of up, down, left, and right. Adding additional padding to the left and right increases the total length by 4, and vice versa. When copying data, all threads within each block will finish transferring data to its shared memory before performing a filtering on their responsible pixel and copying that pixel into a blurred heatmap.

Running the code and group work

The code was run on our own machine with an Amd Ryzen 7 and an Nvidia RTX 3060.
There might be problems while compiling on crocodile, because the cuda_runtime.h can only be found in .cu files.
To run the code you have to use the command:

```
./demo/demo --export-trace --omp --max-steps=200 (also includes cuda)
./demo/demo --export-trace --seq --max-steps=200 (sequential)
```

In the beginning we tried to come up with our own implementations each, to later compare. Because of some issues, we later used only one working version and tried to debug/fix the CUDA implementation to work asynchronously, which was harder than anticipated.