Low-Level Parallel Programming

Assignment 4
Deadline: March 8 23:59

Uppsala University, Spring 2021

1 Deadlines, Demonstrations, Submissions, and Bonus Points

Each lab report and source codes zipped together has to be submitted by its corresponding deadline. During the lab session on the same day, demonstrate your working solution to a lab assistant. Be prepared to answer questions.

2 Instructions

For this assignment you need to first activate code that creates a visual effect from the *desiredPositions*, i.e. the positions each agent wants to move to. The image shows visually how contended each location has been recently. Activate this by using the actual heatmap as an image in the *paint* function, in *MainWindow.cpp*:

```
// Comment this:
// Qimage image;

// Uncomment this:
const int heatmapSize = model.getHeatmapSize();
QImage image((uchar*)*model.getHeatmap(), heatmapSize, \\
  heatmapSize, heatmapSize * sizeof(int), QImage::Format_ARGB32);
```

Then, add a call to the function *updateHeatmapSeq()* in your tick function, after calculating desiredPositions:

```
for(agent: agents){
   agent->computeNextDesiredPosition();
   ..
}
//insert this
updateHeatmapSeq();
```

Updating the heatmap is **independent** of moving the agents. The algorithm itself is divided into several steps, as following:

1. Fade out the *heatmap* from the previous tick.
2. For all agents, increment the heat of the locations they want to go to.
3. Scale the heatmap, so that each location has the same size as the current visual representation in MainWindow.
4. Apply a gaussian blur filter[1] on the scaled heatmap. Convert each pixel into an ARGB value, where the color is red, and the opacity represents the heat.

2.1 Heterogeneous computation

Your assignment is to parallelize the function *updateHeatmapSeq* using CUDA. You may want to divide the algorithm into more than one kernel, but this is not a requirement.

In assignment 2 you had the option of implementing your solution for the GPU. This was however not *heterogeneous computation* in its strictest sense, as the CPU was idling while the result was being computed. In heterogeneous computing, different types of processors (e.g., CPU and GPU) are working in parallel to achieve the goal. Instead of having the CPU waste time idling, design your solution so that the collision handling from assignment 3 is run at the same time as the new heatmap calculation.

For the heatmap calculation you need to do the following:

Parallelize the heatmap creation: It is necessary to consider data races in this step. Atomic operations can help you with this task.
Parallelize the heatmap scaling: You can let each thread scale one pixel.
Parallelize the blur filter: An easy way to solve this is to let each thread calculate the output for one pixel. Take great care when figuring out how your memory access patterns will be across the threads. Make sure that your memory reads are *coalesced* as much as possible, as this will massively effect your performance. Looking at the algorithm further, you see that each pixel is read several times, by different threads. The *shared memory* on the GPU has over 10x the bandwidth and at least 10x lower latency, compared to the global memory. Use this to your advatage when designing your solution.

2.1.1 Requirements

1. The code should run on the GPU at the same time as the collision handling from assignment 3 is being run on the CPU. Confirm this.
2. There must be no data races.
3. For the blur filter, you must use more than one CUDA block.
4. You must somewhere in your solution take advantage of the shared memory.

2.2 Evaluation

Time each of the three heatmap steps that you have parallelized as well as the full sequential version. Plot the times for the "scenario.xml". If you have a single kernel, you can use clock64() within the kernel code to get elapsed time in GPU clock cycles. If you divide the algorithm into several kernels, you can use CUDA events for timing[2].

2.3 Questions

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

[1] A gaussian blur filter sets each pixel to a weighted average of its surrounding pixels, creating a blurred effect.
[2] `http://devblogs.nvidia.com/parallelforall/how-implement-performance-metrics-cuda-cc/`

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

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?

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

2.4 Bonus Assignment

You have now made an algorithm that operates on GPU and CPU simultaneously (heatmap and safe movement). Depending on how well you implemented assignment 3 and your kernels, and the hardware in the machine you run on, the workload balance between the CPU and the GPU may be quite uneven. Your assignment is to first timestamp the start and end of the heatmap and safe movement tasks. Use this information to find out how long before the GPU the CPU finishes its work (i.e. the imbalance).

$$imbalance = \frac{(max(end_{gpu}, end_{cpu}) - min(end_{gpu}, end_{cpu}))}{(max(end_{gpu}, end_{cpu}) - start)}$$

Workload imbalance means we are wasting precious compute resources. Generally, imbalance should be avoided if possible.

Address the load imbalance by adjusting your heatmap computation such that parts of the work is done on the CPU. This will decrease the GPU runtime. Experimentally calibrate the amount of work to be done on CPU so that the heatmap and safe movement tasks finish at roughly the same time. Use your timings to find out a good CPU/GPU workload ratio (i.e., increase the amount of work done on the CPU until the right balance is achieved). If you wish you may create a solution that balances itself, but this is not a requirement.

If it is the case that the GPU is already the faster task of the two, then you may pick a section to move to the CPU anyway. Measure then how the imbalance changes. Finally, choose the best solution, which in this case is to execute the task completely on the GPU.

2.5 Bonus Assignment Questions

A Describe and motivate what parts of the work you offloaded to the CPU.

B Using your timestamp measurements, make a graph showing the workload imbalance before and after your adjustments. Additionally, measure the total runtime of the program. What speedup did you obtain? Explain why.

3 Submission Instructions

Submit the code and a **short** report including all answers to above questions in **PDF** format to studium.

1. Your code **has** to compile.
2. **Clean** your project (run **make submission** which will copy code files from demo and libpedsim and make a tarball called submission.tar.gz.
3. Document how to choose different versions (serial, C++ Threads, and OpenMP implementation).
4. Include a specification of the machine you were running your experiments on.
5. State the question and task number.
6. State on the assignment whether all group members have contributed equally towards the solution - it will be important for your final grade.
7. Include all generated plots.
8. Put everything into a zip file and name it team-n-lastname$_1$-lastname$_2$-lastname$_3$.zip, where n is your team number.
9. Upload the zip file on studium by the corresponding deadline.

4 Acknowledgment

This project includes software from the PEDSIM[3] simulator, a microscopic pedestrian crowd simulation system, that was initially adapted for the Low-Level Parallel Programming course 2015 at Uppsala University.

[3] http://pedsim.silmaril.org/