Low-Level Parallel Programming Uppsala University – Spring 2015 Report for Lab 1 by Team 21

Jonathan Sharyari, Peng Sun

7th March 2015

Solution Description

The goal of this assignment was to parallelize the rendering of a heatmap for the pedestrian simulator. Doing this was straighforward, considering the nature of the problem. We were handed a sequential solution of the problem, with the function "heatmapUpdateSeq", which performed four distinctly different tasks:

1. fading the heatmap
2. updating the heatmap with agent information
3. scaling the heatmap
4. adding a gaussian blur filter on the heatmap

Each of these tasks was divided into its own CUDA kernel, and by nature, no race conditions occur, neither between the tasks nor the collision detection mechanism introduced earlier. In order to parallelize these tasks, such that the tasks may run at the same time as other computations are performed on the CPU, we used pthreads and scheduled the tasks (using fade, update, scale and blur as short-hand notation) in the following way:

CPU	GPU
WhereToGo	Fade
Synchronize with GPU	
Collision detection	Update
	Scale
	Blur
Synchronize with GPU	

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

Note: we use four steps, rather than three.

Two of the functions, fade and scale, will have full spatial access patterns, meaning that adjacent memory locations will always be used in the next iteration. This is something the CPU handles well, and the GPU likewise, considering the fact that each location can be manipulated independently.

The update function on the other hand has bad spatial and temporal locality. This is technically a difficult situation, and more so for the GPU than for the CPU, as the likelihood that memory will be close to the GPU block that needs it is smaller, than with the CPU (as it has several layers of cache, with a better reach). The data being accessed is (potentially) random. This function is fast, as the amount of work being performed is low, but the gain of parallelizing it is either low, or takes longer time to execute than it would sequentially.

The blur function uses a two-dimensional block of data, of size 5*5. This means that there will be between 5 and 10 cache lines in use for each position in the scaled heatmap. This is a fairly good situation for the CPU, and as the information is only read, it would be highly parallelizable on the CPU as well, with high spatial and temporal locality. CUDA likely does this even better, as the computation is performed in two-dimensional blocks, yielding good temporal locality in both dimensions, as opposed to only one.

B. Compare the runtimes of each of the heatmap steps. Discuss the runtimes.
The program is rather slow, thus we changed the total number of ticks to be merely 100, in order to easier compare the statistics. We obtain the following results (sequential collision detection, and agent updates):
Sequential: **53.3136s**
Parallel : 15.4278s
of which the four cuda functions take:
fade: 21.00ms
update: 3,22ms
scale: 710.55ms
block: 2009.69ms

Note that these steps were timed without the time for memory transfers back and forth between the GPU. This is what takes most of the time for the heatmap in the parallel version. Nevertheless, the CUDA-version is notably faster than the sequential version.

Our experiments were run under Ubuntu Linux ~14.04 (64~bit) on an Intel Core~i3~550 of 3.2~GHz with an 4~MB L2 cache and a 4~GB RAM.

C. How much speedup do you get compared to the sequential CPU version?
53.3136/15.4278 ~= 3.46.
So around 346% faster with the cuda version. The speedup is likely to increase, were the size of the heatmap to increase (being more or less stable when changing the number of agents).

D. How much data does your implementation copy to shared memory?
If we interpret this correctly, as how much data needs to be moved between the CPU and GPU during these runs, the bottleneck of the CUDA version, it could be calculated as such:
fade: 2 * SIZE * INT
update: 2 * SIZE + 2 * NUM_AGENTS*DOUBLE
scale: SIZE + SCALED_SIZE * INT
blur: 2 * SCALE_SIZE * INT
Using SIZE 1024 and SCALED_SIZE 5120 and 2000 agents (ignoring the fact that some are removed, as they are duplicates), we're moving 113920 bytes, or 111.25 kilobytes of data each tick.

Run instructions
Without an argument specifying the type of parallelism, the serial version is used, whereas --pthreads, --openmp, --vector and --cuda activate pthreads, openmp, vector and cuda respectively (in case both are set, the last occurance will be dominating). The number of threads can in both cases be

set by --np X. The value of X is ignored in the serial case. Timing mode is enabled by adding --timing-mode to the commandline. An xml-formatted scenario may be specified by writing the name of the xml file in the command-line, if none is mentioned, "scenario.xml" will be used.

If no collision detection method is specified, pthreads is assumed. For sequential, use --cseq and --comp for openMP. Note that openMP requires that the number of threads is specified.

The flag --chmap turns on the CUDA version of the heatmap management.

```
Serial: ./demo
OpenMP: ./demo --openmp
Pthreads: ./demo --pthreads --np 4
Vector ./demo --timing-mode –-vector
CUDA ./demo --cuda hugeScenario.xml
```

Other

We certify that this report is solely produced by us, except where explicitly stated otherwise and clearly referenced, and that we can individually explain any part of it at the moment of submitting this report.

This assignment was contributed to equally by Jonathan and Peng, whereas the third member of the group, Markus, chose not to participate for the bonus assignment.