Optimizing the Lattice-Boltzmann method using OpenCL

Carlo Alessi
ca16296

1. Introduction

The aim of this project was to parallelize the Lattice-Boltzmann method to run on high performance heterogeneous hardware consisting of central processing units (CPUs) and graphics processing units (GPUs), using OpenCL (Open Computing Language). The OpenCL API was used to launch compute kernels on the Nvidia Tesla K20 GPUs.

In section 2 I discuss the OpenCL design of the application. In section 3 I report a small range of experiments showing the difference between the performance gained using different types of reductions, work-group sizes and kernel sizes. Section 4 concludes with a short discussion of the main results.

2. OpenCL Design

The project started from the given code, with two out of five of the main functions already ported in kernel format. The cells array were copied from the host to the device each time step, so that accelerate_flow() and propagate() were computed on the GPU. Once the computation completed, the tmp_cells were copied back to the host to compute collision(), rebound() and av_velocity().

The data movements, performed at each time step, made up most of the program runtime. In order to transfer the data from the host to the device, and back from the device to the host, just once, all the functions had to be ported in kernel format.

Once all the functions were ported in kernel format, all the computations were performed in the device. It was avoided the data transfer at each time step, and moved clEnqueueWriteBuffer() and clEnqueueReadBuffer() respectively before and after the time steps. It was achieved an overall speedup of 4.8x.

The next optimization carried out, merged as much kernels as possible. In order to achieve that, all kernels were transformed to the same dimension. The nature of the problem, a two-dimensional flattened array, suggested to launch the kernels as a one-dimensional work-items. It was achieved a speedup of 1.06x.
Merging all the kernels but accelerate_flow() reduced memory accesses and gave a speedup of 1.4x. The research did not put effort in merging accelerate_flow() with the other kernels because it was found, manually profiling the code[1], that the former made up only 12% of the computation and that, if merged perfectly, it would have saved around a second for the 256x256 grid. Moreover, branches handling on the GPU is more difficult than on the CPU, due a primitive structure of GPU cores, which instead rely on massive

1 The Vtune profiler only reported the CPU time, but the time of interest was the wall time spent on the GPU.

parallelism. Therefore, accelerate_flow() was retained as a separate kernel.

The speedup gained from using single-precision floating point was 2.05x. It was erased the branch in accelerate_flow(), achieving a speedup of 1.02x. The enqueue of the latter was non-blocking so that other kernels were enqueued without waiting for the latter to finish, giving a speedup of 1.04x. It preserved the correctness of the program as the command queue was in-order.

3. Experiments

The following sections report a small range of experiments regarding the work-group size, the reduction and the kernel size.

3.1 Performance tuning

The experiment carried out suggested to plot the elapsed time rather than the speedup. In this case, the latter made less sense as there was not baseline for the ideal scaling.

The study experimented different work-group sizes to find performance sweet spots. The chart in Figure 1 shows how the Elapsed time changed when different work-group sizes were set.

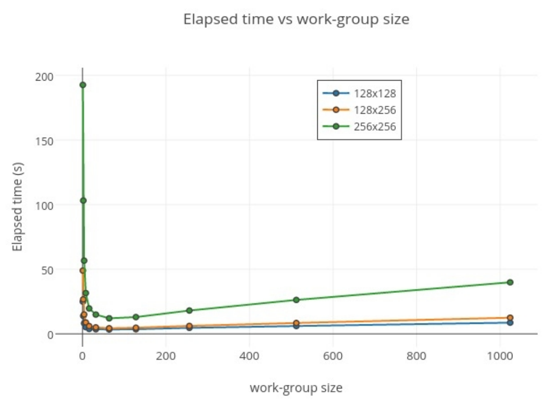

Figure 1: Elapsed time for work-group size from 1 to 1024. Different grid sizes.

The research uncovered that setting a work-group size between 1 and 8 did not allow to exploit parallelism. In particular, using a work-group size of 1 transformed the application in SISD (Single Instruction Single Data), incrementing the number of pulls from the work-group queue. To keep the processing elements busy, the work-group should contain as many work-items as possible. For instance, the hardware used performed well with multiples of 8. Indeed, the elapsed time decreased significantly for a work-group size of 16, 32, 64, 128.

In order to fill the SIMD lanes, the work-group size was incremented to find the lowest execution time, achieved when the work-group size reached 64 or 128. Further increments of the work-group size, marginally slowed the program, suggesting that bigger work-groups were expensive to maintain.

It was important to set a proper work-group size for average_velocity() as it was launched separately and computed a single row. This fact was key to understand that the work-group size is upper bounded by both hardware and software.

3.2 Reduction

While CPUs don't have special hardware for local memory, GPUs have effective on-chip caches which can benefit from local memory. Therefore, the program reduced the values within work-groups on the device, and the values across work-groups on the Host.

The first reduction used was completely serial. By taking advantage of the addition operator, which is associative and commutative, two more reductions were experimented. The associative reduction parallelized the operation and gave a speedup of 1.03x. The commutative reduction grouped the active work-items into contiguous blocks, leading to better SIMD efficiency. However, this approach did not boost the performance.

3.3 Concatenating work-items

The last experiment carried out chose the kernel size that worked best.
Using smaller kernels can be efficient because each one uses minimal resources. However, concatenating many small work-items can harm the performance, due to a primitive structure of the GPU cores. On the other hand, as the results must be stored in global memory, and since reading and writing to it is expensive, concatenating many small kernels into one may save some extra work.
The chart in figure 2 shows how the performance dropped when the work-items were merged to do extra iterations on the grids. It was uncovered that the program performed better when the GPU was oversubscribed with many small work-items, that exploited the large number of compute units that compose the GPU.

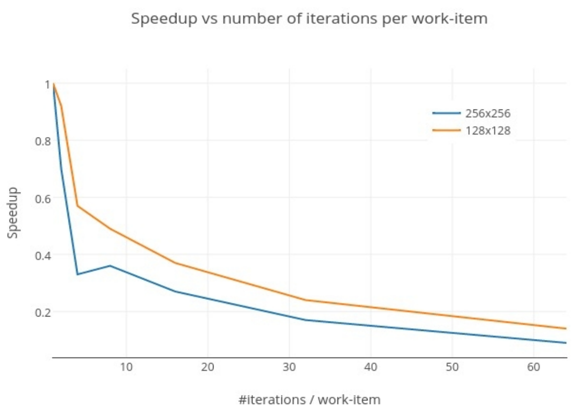

Figure 2: Performance drop. The iterations assigned to each work-item were 1, 2, 4, 8, 16, 32 and 64.

4. Concluding remarks

Table 1 summaries the results achieved for the Lattice-Boltzmann method optimization using different standard libraries such as OpenMP, OpenMPI and OpenCL.

256x256	Elapsed time (s)	Speedup	Ideal scaling
Serial	256.6	/	/
OpenMP	18.3	**14x**	16x
Open MPI	6	**43x**	64x
OpenCL	9.7	**26x**	32x

Table 1: Elapsed time (s) and speedup for different approaches.

It was not achieved a great speedup because the memory layout did not suit to the GPU memory access pattern. A substantial increment would have been achieved if it was used a SoA (Structure of Array).