Assignment #3: Porting the lattice Boltzmann code to OpenCL

Name: *Christos Mourouzi,* ***User Id:*** *cm16663,* ***Canditate No.:*** *33747*

Introduction

The aim of this project is to parallelize the lattice Boltzmann, D2Q9 (BGK method) algorithm, by taking advantage of the OpenCL framework.

The report begins with a description of some first experiments made for optimizing the given OpenCL code. Then, the improvement of the code is explained until reaching the final optimized version. Afterwards, a comparison between different local work-group sizes is made. In the end, the runtime measurements of the best serial, OpenMP, MPI and OpenCL code, are compared.

First approach

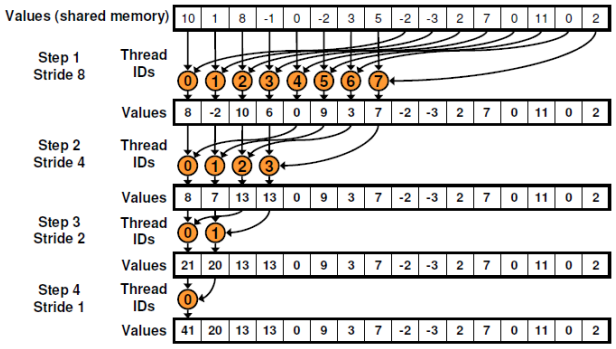
The given code for this assignment had already the *accelerate_flow()* function and the *propagate()* function implemented as OpenCL kernels. Firstly, a merge of *rebound()* and *collision()* was made. Then, the new *rebound_collision()* function was implemented as a kernel function and got parallelized. This resulted to no more writes and reads to and from the device memory in each time step. Also, a simple reduction method[1] for the addition of the *tot_u* value of each work-item was used. With these initial improvements there was a speed-up of up to x8 of the initial execution time.

OpenCL Optimization

After the improvements of the first approach, more exploration on other optimizations of the parallel code took place as they are described below.

Efficient reduction method

A more efficient method of reduction was examined. The *tot_u* values of each work-item is computed and stored in a local float array in the local memory of each work-group. If the worker handles a cell that is an obstacle (rebound step), then the *tot_u* value is equal to zero. After the rebound and the collision step take place, the work-items are synchronized and add their *tot_u* values with a sequential addressing method[2]. The sum of all values of each work-group is stored finally in the first place of the local array and the first work-item of each group is then responsible to store it in the proper place in a global array. When all time steps are finished, the host computes the correct average velocity for each time step outside of the main loop using the *tot_cells* value already calculated in the beginning of the host code. In this way, there is no need of reading the partial sum of each group from the device in each time step. This reduces the runtime further as the table below suggests.



128x128	128x256	256x256
5.1 s	8.3 s	17.8 s

Table 1. Execution time in seconds with the new reduction method

[1] https://github.com/HandsOnOpenCL/Exercises-Solutions/tree/master/Solutions/Exercise09

[2] http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf

Structure of arrays vs Array of Structures

The given code uses an array of structures (AoS) approach for storing each cell's speeds. OpenCL kernels can not get structures with pointers as arguments. In order to take advantage of memory coalescence's high bandwidth on a GPU, the speeds of each cell are stored in a float array as the Figure 1 shows. Execution time was decreased even more by using structures of arrays approach and the results are shown in Table 2.

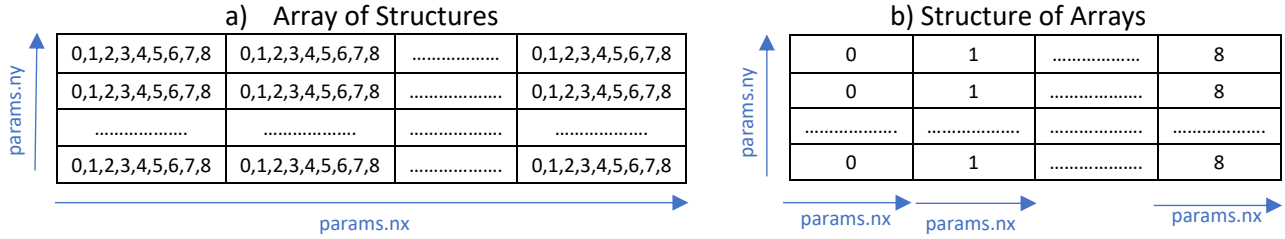


Figure 1. a) Speeds 0-8 of each cell are stored in a float array[NSPEEDS] in an array of structures, b) Speeds 0-8 of each cell are stored in groups corresponding to the index of their speed in a float array.

128x128	128x256	256x256
3.3 s	4.5 s	12.6 s

Table 2. Execution time in seconds with the use of structure of arrays

Using local memory

Local memory is shared within a work-group. That means, that only work-items within a work-group can use this kind of memory and threads from other work-groups can not access it. Copy from global memory to local memory and backwards of obstacles, cells and tmp_cells, was tried. Surprisingly, the results showed that using local memory was less efficient than using global memory. The cost of reading from global memory to local memory and backwards was higher from just reading directly the appropriate values of the global memory array's values.

128x128 (s)	128x256	256x256
5.5s	10.1 s	29 s

Table 3. Execution time in seconds using local memory

Using private memory and merging propagate() with rebound_collision()

Furthermore, private memory and merging the propagate step with the rebound and the collision step got into test. Instead of reading and writing from and to the *cells[]* and *tmp_cells[]* arrays, a private float array was created for each work-item. This array reads and stores the proper values of the propagate step from the *cells[]* array. Then these values are used for the calculation of the rebound and the collision step, while the *tmp_cells[]* array stores these computed values. At the end of each time step the pointers of the *cells[]* and *tmp_cells[]* are swapped and then are reset as arguments in *accelerate_flow()* and *propagate_rebound_collision()*. This got the fastest execution time, and was chosen as the final version of the code.

Different work-group sizes for reduction

As the local memory is only used for the reduction of the *tot_u* value, different work-groups sizes were examined. As the Figure 2 shows, the most efficient work-group size was with 256 work-items per group.

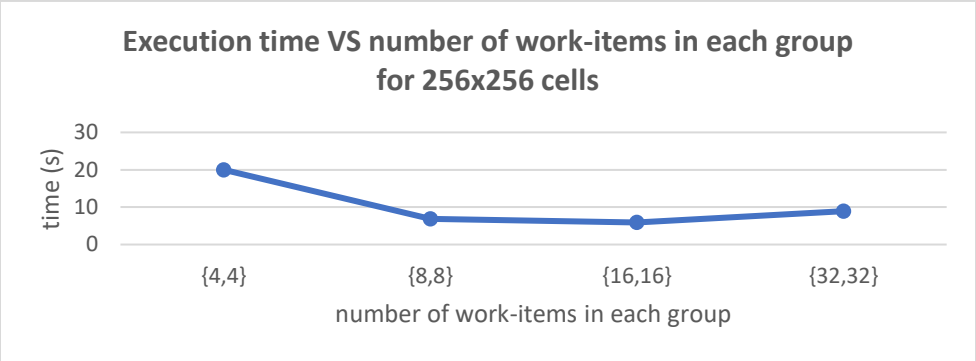


Figure 2. Execution time vs number of work-items in each groups for 256x256 problem

Fastest execution times for the four problems

After finding the most efficient parallel optimizations, the fastest runtimes for all four problems were measured. The final execution times for all four problems are shown in the table below.

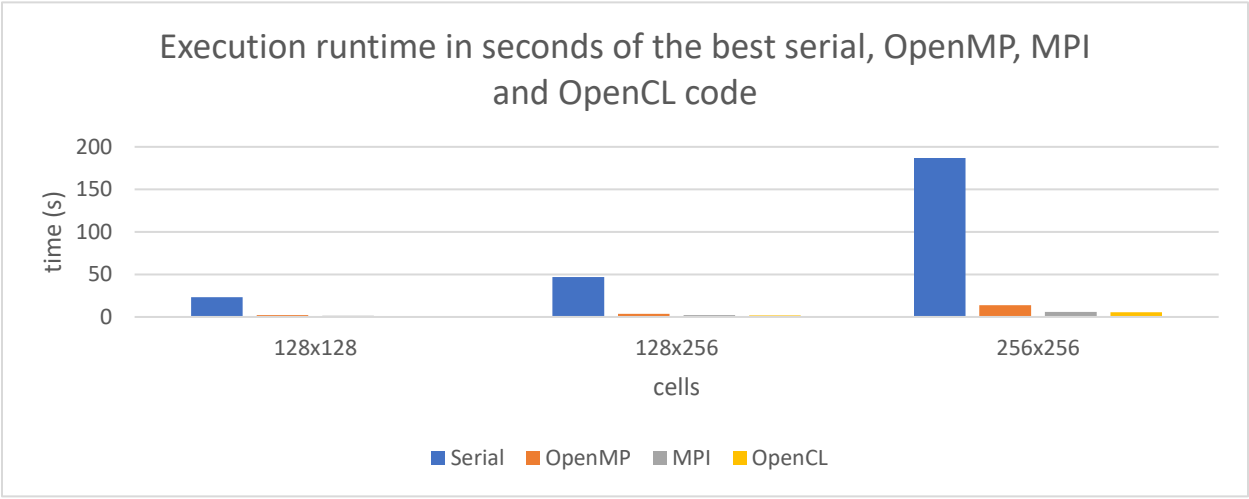
128x128	128x256	256x256	1024x1024
1.18 s	1.81 s	5.8 s	16.3 s

Table 4. Fastest execution times running the code with OpenCL framework on a GPU.

The results, show the enormous computation power that OpenCL can offer. Writing simple device and host codes and taking advantage of the new GPUs' and CPUs' technologies can lead to impressive results and decrease the amount of time needed for the computation of complex and difficult problems.

Comparison of different parallel techniques and conclusion

The chart below shows a comparison between the best execution times of all assignments for the 128x128, 128x256 and 256x256 problems. The best optimized serial code, the OpenMP, the MPI and the OpenCL code, are compared.



The final findings show that OpenCL parallelism with a GPU device is the most efficient of all, due to its high memory bandwidth. MPI comes second, as it uses a distributed memory model with the OpenMP come as third, as it uses a share memory model. More could be explored through all of the assignments. The sure thing we learnt from this unit is that computers nowadays, can take charge of all kind of complex algorithmic and computational problems by taking advantage of advanced techniques, working parallel with each other and porting their whole computation power as one to solve almost anything in the fastest and most efficient way.